Node.js – популярная программная платформа, позволяющая легко и просто создавать масштабируемые серверные приложения на JavaScript. Она дает возможность писать эффективный и надежный код на единственном языке, с непревзойденным уровнем пригодности к повторному использованию, применяя при этом полный стек технологий.

В книге описаны асинхронная, однопоточная архитектура платформы, а также шаблоны асинхронного управления потоком выполнения и потоками данных. Рассмотрен подробный список реализаций распространенных, а также некоторых уникальных шаблонов проектирования в Node.js. В конце книги предложено детальное обсуждение более продвинутых идей, таких как «универсальный JavaScript» и масштабируемость. А в заключение перечислены основные идеи Node.js, которые пригодятся для создания приложений уровня предприятия.

Прочитав эту книгу, вы:

- познакомитесь с множеством шаблонов проектирования серверных приложений на языке JavaScript и узнаете, как и в каких ситуациях они применяются;
- освоите разработку асинхронного кода, используя такие конструкции, как обратные вызовы, объекты Promise, генераторы и синтаксис async/await;
- научитесь выявлять наиболее важные проблемы и применять уникальные приемы для достижения высочайшей масштабируемости и модульности в своих приложениях;
- узнаете, как организовать свои модули, чтобы обеспечить их слаженную работу;
- освоите стандартные приемы решения типичных проблем проектирования и программирования;
- исследуете последние тенденции JavaScript, научитесь писать код, способный выполняться в Node.js и в браузере
- познакомитесь с библиотекой React и ее экосистемой, помогающей писать универсальные приложения.

Кому адресована эта книга

Книга адресована разработчикам и архитекторам программного обеспечения, обладающим основными навыками JavaScript, желающим получить глубокое понимание, как проектируются и разрабатываются приложения уровня предприятия на основе Node.js. Элементарное знакомство с Node.js поможет извлечь максимальную выгоду из книги.

Шаблоны проектирования Node.js

Шаблоны проектирования Node.JS

Марио Каскиаро
Лучано Маммино

ISBN 978-5-97060-485-4

ДМК издательство
www.дмк.рф

Интернет-магазин www.dmkpress.com
Книга-почтой: orders@alians-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)782-3889, books@alians-kniga.ru

PACKT PUBLISHING

ДМК издательство

Марио Каскиаро, Лучано Маммино

Шаблоны проектирования Node.js

Ian Goodfellow, Yoshua Bengio, Aaron Courville

Node.js Design Patterns

Second Edition

Get the best out of Node.js by mastering its most powerful components and patterns to create modular and scalable applications with ease

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM – MUMBAI

Марио Каскиаро, Лучано Маммино

Шаблоны проектирования Node.js

Воспользуйтесь самыми мощными компонентами
и шаблонами платформы Node.js для создания масштабируемых
модульных приложений

ДМК Пресс
издательство
Москва, 2017

УДК 004.738.5:004.4Node.js
ББК 32.973.202-018.2
К28

Каскиаро М., Маммино Л.
К28 Шаблоны проектирования Node.js / пер. с анг. А. Н. Киселева. – М.: ДМК Пресс, 2017. – 396 с.: ил.

ISBN 978-5-97060-485-4

Node.js – программная платформа, позволяющая легко и просто создавать масштабируемые серверные приложения на языке JavaScript.

В книге описаны асинхронная, однопоточная архитектура платформы, а также шаблоны асинхронного управления потоком выполнения и потоками данных. Рассмотрен подробный список реализаций распространенных, а также некоторых уникальных шаблонов проектирования в Node.js.

Издание адресовано разрабтчикам и архитекторам программного обеспечения, обладающим основными навыками владения JavaScript и желающим получить глубокое понимание, как проектируются и разрабатываются приложения уровня предприятия на основе Node.js.

УДК 004.738.5:004.4Node.js
ББК 32.973.202-018.2

ISBN 978-1-78588-558-7 (анг.)
ISBN 978-5-97060-485-4 (рус.)

Содержание

Об авторах

Марио Каскиаро (Mario Casciaro) – программный инженер и предприниматель, увлекающийся технологиями, наукой и разработкой программного обеспечения с открытым исходным кодом. Получив степень магистра в области разработки программного обеспечения, Марио начал карьеру в компании IBM, где несколько лет занимался созданием различных корпоративных продуктов, таких как Tivoli Endpoint Manager, Cognos Insight и SalesConnect. Затем перешел в компанию D4H Technologies, специализирующуюся на SaaS, чтобы руководить разработкой нового важного проекта по управлению операциями в режиме реального времени. В настоящее время Марио является соучредителем и генеральным директором компании Sponsorama.com, разрабатывающей обеспечивающей корпоративное спонсорство развития интернет-проектов.

Марио является автором первого издания книги *Node.js Design Patterns*.

Благодарности

Работая над первым изданием этой книги, я и не предполагал, что она будет иметь такой успех. Я очень благодарен всем, кто прочел первое издание книги, приобрел его, оставил отзыв и рекомендовал его своим друзьям в Twitter и других интернет-форумах. И конечно же, я выражаю благодарность читателям второго издания, всем вам, кто читает его сейчас. Это значит, что наши усилия не пропали даром. Я также прошу вас присоединиться к моему дружескому поздравлению Лучано, соавтору второго издания, проделавшему огромную работу по обновлению и добавлению новых бесценных сведений в эту книгу. Все лавры по созданию этого издания принадлежат ему, потому что я выступал только в роли советчика. Работа над книгой – нелегкая задача, но Лучано удивил меня и всех сотрудников издательства Packt своей преданностью делу, профессионализмом и техническими навыками, демонстрируя способность достичь любой задуманной им цели. Работать вместе с Лучано было приятно и почетно, и я с нетерпением жду продолжения сотрудничества. Также хочу поблагодарить всех, кто работал над книгой, сотрудников издательства Packt, технических рецензентов – Тане (Tane) и Джоэл (Joel) – и всех друзей, внесших ценные предложения и идеи: Энтони Уолли (Anton Whalley, @dhigit9), Алессандро Синелли (Alessandro Cinelli, @cirpo), Андреа Джулиано (Andrea Giuliano, @bit_shark) и Андреа Мангано (Andrea Mangano, @ManganoAndrea). Спасибо всем дарящим мне свою любовь друзьям, моей семье и, самое главное, моей подруге Мириам, партнеру во всех моих приключениях, наполняющей любовью и радостью каждый день моей жизни. Нас еще ждут сотни тысяч новых приключений.

Лучано Маммино (Luciano Mammino) – инженер, родившийся в том же 1987 году, в котором компания Nintendo выпустила в Европе игру Super Mario Bros, совершенно случайно ставшую его любимой видеоигрой. Программировать он начал в возрасте 12 лет, на старом отцовском компьютере с процессором Intel 386, операционной системой DOS и интерпретатором qBasic.

После получения степени магистра в области компьютерных наук развивал навыки программирования в основном как веб-разработчик, работающий в качестве фрилансера с компаниями и стартапами, разбросанными по всей Италии. Будучи в течение трех лет техническим директором и соучредителем сайта Sbaam.com в Италии и Ирландии, он решил переехать в Дублин, где поступил на работу в компанию Smartbox старшим PHP-инженером.

Ему нравится разрабатывать библиотеки с открытым исходным кодом и работать на таких платформах, как Symfony и Express. Он убежден, что JavaScript все еще находится в самом начале своего славного пути, и в будущем его влияние на веб- и мобильные технологии будет только расти. По этой причине он проводит большую часть своего свободного времени, совершенствуя знания JavaScript и экспериментируя с Node.js.

Благодарности

Прежде всего хочу сказать огромное спасибо Марио за предоставленную мне возможность и оказанное доверие поработать вместе с ним над новым изданием этой книги. Это был интересный эксперимент, и я надеюсь, что он станет началом дальнейшего нашего сотрудничества.

Появление этой книги стало возможным только благодаря невероятно эффективной работе команды издательства Packt, в частности неустанным усилиям и терпению Онкара (Onkar), Решма (Reshma) и Прякта (Prajakta). А также благодаря помощи технических рецензентов Тана Пайпера (Tane Piper) и Джоэла Пурра (Joel Purra), их опыт работы с Node.js имел решающее значение для повышения качества этой книги.

Моя огромная благодарность (и море пива) моим друзьям Энтони Уолли (Anton Whalley, @dhigit9), Алессандро Синелли (Andrea Giuliano, @cirpo), Андреа Джулиано (Andrea Giuliano , @bit_shark) и Андреа Мангано (Andrea Mangano, @ManganoAndrea), поддерживавшим меня всю дорогу, делившимся со мной своим опытом и значительно повлиявшим на эту книгу.

Также большое спасибо хочу сказать Рикардо (Ricardo), Хосе (Jose), Альберто (Alberto), Марцин (Marcin), Начо (Nacho), Давиду (David), Артуру (Arthur) и всем моим коллегам из Smartbox за радость работы с ними и мою мотивацию к совершенствованию как программного инженера. Я не могу представить себе лучшую команду.

Выражаю глубочайшую благодарность моей семье, поддерживавшей меня все это время. Спасибо, мама, ты для меня постоянный источник вдохновения и силы. Спасибо, папа, за все уроки, похвалы и советы, я скучаю по нашим разговорам, я действительно скучаю по тебе. Спасибо моему брату Давиду и моей сестре Алессии, державшим меня в курсе грустных и радостных моментов, что позволяло мне чувствовать себя частью большой семьи.

Благодарю Франко и его семью за поддержку многих моих инициатив и за то, что они делятся со мной мудростью и жизненным опытом.

Спасибо моим «умным» друзьям Джанлуке (Gianluca), Флавио (Flavio), Антонио (Antonio), Валерио (Valerio) и Луке (Luca) за отлично проведенное вместе время и их призывы продолжать работу над этой книгой.

А также спасибо моим «менее умным» друзьям Дамиано (Damiano), Пьетро (Pietro) и Себастьяно (Sebastiano) за их дружбу, наш смех и радость при совместных прогулках по Дублину.

И напоследок, но не в последнюю очередь, спасибо моей подруге Франческе (Francesca). Спасибо за любовь и поддержку моих идей, даже сумасшедших. Я с нетерпением жду написания следующих страниц книги нашей совместной жизни.

О технических рецензентах

Тане Пайпер (Tane Piper) – опытнейший разработчик из Лондона (Великобритания). На протяжении более 10 лет работал в нескольких агентствах и компаниях, занимаясь разработкой программного обеспечения на разных языках, таких как Python, PHP и JavaScript. С платформой Node.js работает с 2010 года и был одним из тех, кто первым в Великобритании и Ирландии заговорил о применении JavaScript на стороне сервера в 2011–2012 годах. Также внес свой вклад в становление проекта jQuery.

В настоящее время работает консультантом по инновационным решениям в Лондоне, пишет главным образом React- и Node-приложения. В свободное время страстно увлекается дайвингом и фотографией.

Я хотел бы поблагодарить мою подругу Элину, изменившую последние два года моей жизни и вдохновившую меня на рецензирование этой книги.

Джоэл Пурра (Joel Purra) познакомился с компьютерами в раннем детстве, рассматривая их как еще один вид устройств для видеоигр. Некоторое время использовал компьютеры (иногда ломавшиеся и затем ремонтировавшиеся) только для запуска самых новых игр. Попытка изменить игру Lunar Lander заставила его в раннем подростковом возрасте начать программировать, вызвала интерес к созданию цифровых инструментов. Вскоре, после появления дома подключения к Интернету, разработал свой первый веб-сайт для электронной коммерции, начав таким образом собственный бизнес. Его карьера началась в раннем возрасте. В 17 лет Джоэл начал изучать компьютерное программирование в школе при атомной электростанции. После окончания школы поступил в военное училище в Швеции, где получил образование в области телекоммуникаций, а затем закончил университет Линчёпинга (Linköping), получив степень магистра информационных технологий и программной инженерии. Начиная с 1998 года участвовал в нескольких проектах разных компаний, как успешных, так и не очень, с 2007 года работает консультантом. Джоэл родился, вырос и получил образование в Швеции, но как внештатный разработчик в течение нескольких лет путешествовал по пяти континентам с рюкзаком и жил за границей. Постоянно учится решать новые задачи, одной из его целей является создание программного обеспечения для широкого общественного использования. Имеет свой веб-сайт http://joelpurra.com/.

Я хотел бы поблагодарить сообщество разработчиков открытого программного обеспечения за предоставление строительных блоков, так необходимых внештатным консультантам для создания малых и больших программных систем. Nanos gigantum humeris insidentes (Карлики на плечах гигантов). Помните, чтобы суметь, нужно пытаться!

Предисловие

Появление платформы Node.js многими рассматривается как смена правил игры – крупнейший за десятилетия сдвиг в области веб-разработки. Это определяется не только техническими возможностями платформы, но и парадигмой, вносимой ею в веб-разработку.

Во-первых, приложения на платформе Node.js написаны на языке Интернета – JavaScript, единственном языке программирования, изначально поддерживаемом большинством веб-браузеров. Эта его черта позволяет писать все компоненты приложения на одном языке и совместно использовать код клиентом и сервером. Платформа Node.js способствует развитию языка JavaScript. Разработчики, использующие JavaScript на сервере и в браузере, вскоре оценят его прагматизм и гибридный характер, определяемый промежуточным положением между объектно-ориентированным и функциональным программированием.

Вторым важным фактором является однопоточная асинхронная архитектура платформы. Помимо очевидных преимуществ с точки зрения производительности и масштабируемости, платформа меняет сам подход к реализации приемов параллельной обработки. Мьютексы заменяются очередями, потоки – обратными вызовами и событиями, а синхронизация – причинно-следственной связью.

И последний, но самый важный аспект платформы Node.js, заключается в ее экосистеме: диспетчер пакетов npm с постоянно растущей базой данных модулей, активным и дружелюбным сообществом, а самое главное, с собственной культурой, основанной на простоте, прагматизме и чрезвычайной модульности.

Но эти особенности разработки на платформе Node.js вызывают смешанные чувства, по сравнению с работой на других серверных платформах, и практически все, кто впервые сталкивается с этой парадигмой, чувствуют себя неуверенно при решении даже самых простых задач проектирования и написания кода. Постоянно возникают вопросы: «Как организовать код?», «Существует ли лучший способ решения?», «Как улучшить модульность приложения?», «Как эффективно обрабатывать набор асинхронных вызовов?», «Как можно удостовериться, что рост приложения не приведет к его краху?» или просто «Есть ли правильный способ сделать это?»

К счастью, Node.js – уже достаточно зрелая платформа, и на большинство этих вопросов можно легко ответить с помощью шаблонов проектирования, проверенных приемов программирования и рекомендуемых методик. Цель этой книги – познакомить вас с существующими шаблонами, методами и рекомендациями, предоставляющими проверенные временем решения часто возникающих проблем, и научить вас их использованию для решения конкретных задач.

Прочтя эту книгу, вы узнаете следующее:

- «подход Node»: рекомендуемая точка зрения на задачи проектирования при использовании платформы Node.js. Например, как выглядят на платформе Node.js различные традиционные шаблоны проектирования или как создавать модули, решающие только одну задачу;
- набор шаблонов для решения общих проблем проектирования при использовании платформы Node.js: вам будет предоставлено что-то вроде «швейцарского

армейского ножа» из готовых к использованию шаблонов, для эффективного решения повседневных задач разработки и проектирования;

- порядок написания эффективных модульных приложений для Node.js: понимание основных строительных блоков и принципов создания больших и хорошо организованных приложений для Node.js, обеспечивающее применение этих принципов к новым задачам, к которым не применимы существующее шаблоны.

В книге будет представлено несколько библиотек и технологий, таких как LevelDb, Redis, RabbitMQ, ZMQ, Express и многие другие. Они используются для демонстрации шаблонов или технологий, делая примеры более полезными, а также знакомят с экосистемой платформы Node.js и ее набором решений.

Если вы уже используете или планируете использовать платформу Node.js в работе над сторонним проектом или проектом с открытым исходным кодом, применение широко распространенных шаблонов и методов позволит вам быстро найти общий язык при совместном использовании кода и проектных решений, а также поможет понять будущее платформы Node.js и узнать, как внести свой вклад в ее развитие.

Какие темы охватывает книга

Глава 1 «Добро пожаловать в платформу Node.js» служит введением в проектирование приложений на платформе Node.js, знакомя с шаблонами, лежащими в основе самой платформы. Она охватывает экосистему и философию платформы Node.js, содержит краткое введение в версию Node.js 6, спецификацию ES2015 и шаблон «Реактор» (Reactor).

Глава 2 «Основные шаблоны Node.js» знакомит с основами асинхронного программирования и шаблонами проектирования Node.js, описывая и сравнивая обратные вызовы и генерацию событий – шаблон «Наблюдатель» (Observer). В этой главе также рассматриваются система модулей Node.js и соответствующий шаблон «Модуль» (Module).

Глава 3 «Шаблоны асинхронного выполнения с обратными вызовами» содержит набор шаблонов и методов эффективного выполнения асинхронных операций в Node.js. Эта глава знакомит со способами смягчения проблемы «ада обратных вызовов», основанными на обычном JavaScript и библиотеке async.

Глава 4 «Шаблоны асинхронного выполнения с использованием спецификации ES2015, и не только», продолжает исследование способов выполнения асинхронных операций с применением объектов Promise, генераторов и библиотеки Async/Await.

Глава 5 «Программирование с применением потоков данных» подробно рассматривает один из самых важных шаблонов Node.js: «Поток данных» (Stream). Знакомит с обработкой данных путем преобразования и объединения их потоков.

Глава 6 «Шаблоны проектирования» посвящена вызывающей споры теме традиционных шаблонов платформы Node.js. Охватывает наиболее популярные традиционные шаблоны проектирования и демонстрирует, насколько нетрадиционно они могут выглядеть в Node.js. Здесь также приводятся некоторые новые шаблоны проектирования, применимые только к JavaScript и Node.js.

Глава 7 «Связывание модулей» содержит анализ различных способов связывания модулей приложения в единое целое. Здесь представлены такие шаблоны проектирования, как «Внедрение зависимостей» (Dependency Injection) и «Локатор служб» (Service Locator).

Глава 8 «Универсальный JavaScript для веб-приложений» рассматривает одну из наиболее интересных черт современных веб-приложений на JavaScript – возможность совместного использования кода приложения клиентом и сервером. В этой главе демонстрируются основные принципы универсальности JavaScript путем создания простого веб-приложения с помощью React, Webpack и Babel.

Глава 9 «Дополнительные рецепты асинхронной обработки» предлагает подход, помогающий исключить несколько общих проблем разработки и проектирования с помощью готовых к использованию решений.

Глава 10 «Шаблоны масштабирования и организации архитектуры» описывает основные методы и шаблоны масштабирования приложений на платформе Node.js.

Глава 11 «Шаблоны обмена сообщениями и интеграции» посвящена наиболее важным шаблонам организации обмена сообщениями, создания и интегрирования сложных распределенных систем с использованием использующих ZMQ и AMQP.

Что потребуется для работы с книгой

Для работы с примерами, приведенными в книге, потребуется установить платформу Node.js версии 6 (или выше) и диспетчер пакетов npm версии 3 (или выше). Некоторые примеры требуют использования транскомпилятора, такого как Babel. Необходимы также навыки работы с командной строкой, умение установить npm и запустить приложение на Node.js. Также понадобятся текстовый редактор для работы с кодом и современный веб-браузер.

Кому адресована эта книга

Эта книга адресована разработчикам, уже знакомым с платформой Node.js и желающим получить максимальную отдачу от ее использования с точки зрения производительности, качества проектирования и масштабируемости. От вас требуется лишь понимание основ технологии, поскольку книга включает описание основных понятий. Разработчики со средним опытом работы в Node.js также найдут в книге полезные для себя методы.

Наличие знаний в области теории проектирования программного обеспечения станет преимуществом при изучении некоторых из представленных концепций.

Эта книга предполагает наличие у читателя знаний о разработке веб-приложений, языке JavaScript, веб-службах, базах и структурах данных.

Соглашения

В этой книге используется несколько разных стилей оформления текста для выделения разных видов информации. Ниже приведены примеры этих стилей с объяснением их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, фиктивные адреса URL, пользовательский ввод и ссылки в Twitter будут выглядеть так: «Спецификация ES2015 вводит ключевое слово `let` для объявления переменных, с областью видимости внутри блока».

Блоки программного кода оформляются так:

```
const zmq = require('zmq')
const sink = zmq.socket('pull');
sink.bindSync("tcp://*:5001");

sink.on('message', buffer => {
  console.log(`Message from worker: ${buffer.toString()}`);
});
```

Чтобы обратить ваше внимание на определенный фрагмент блока кода, соответствующие строки или элементы будут выделены жирным:

```
function produce() {
  //...
  variationsStream(alphabet, maxLength)
    .on('data', combination => {
      //...
      const msg = {searchHash: searchHash, variations: batch};
      channel.sendToQueue('jobs_queue', new Buffer(JSON.stringify(msg)));
      //...
    })
  //...
}
```

Ввод и вывод в командной строке будут выделены так:

```
node replier
node requestor
```

Новые термины и важные слова будут выделены жирным. Слова, которые выводятся на экран, например в меню или диалоговых окнах, будут оформляться так: «Для иллюстрации задачи создадим небольшое приложение **web spider**, запускаемое из командной строки и принимающее на входе URL-адрес, которое загружает страницу в локальный файл».

Предупреждения и важные сообщения будут выделены так.

Подсказки и советы будут выглядеть так.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут вам максимально полезны. Отзывы и пожелания можно посылать по адресу feedback@packtpub.com, указав название книги в теме письма. Если существует область, в которой вы хорошо разбираетесь, и у вас есть желание написать книгу, загляните в руководство для авторов: www.packtpub.com/authors.

Поддержка пользователей

Если вы – счастливый обладатель книги, изданной в Packt, мы готовы предоставить вам дополнительные услуги, чтобы ваша покупка принесла вам максимальную пользу.

Загрузка примеров исходного кода

С помощью своей учетной записи на http://www.packtpub.com вы сможете загрузить файлы с примерами кода для всех книг издательства Packt, купленных вами. Где бы вы ни купили эту книгу, вы сможете посетить страницу http://www.packtpub.com/support и зарегистрироваться для получения файлов по электронной почте.

Для скачивания файлов кода необходимо выполнить следующие действия:

1) войдите или зарегистрируйтесь на веб-сайте, указав свой адрес электронной почты и пароль;
2) наведите указатель мыши на вкладку **SUPPORT** (Поддержка) вверху;
3) щелкните на ссылке **Code Downloads & Errata** (Загрузка примеров кода и опечатки);
4) введите название книги в поле **Search** (Поиск);
5) выберите книгу, для которой хотите скачать файлы с исходным кодом;
6) выберите в раскрывающемся меню, где вы приобрели эту книгу;
7) щелкните на ссылке **Code Download** (Загрузить код), чтобы запустить загрузку.

После загрузки архива извлеките файлы с помощью последней версии одной из программ:

- WinRAR / 7-Zip для Windows;
- Zipeg / iZip / UnRarX для Mac;
- 7-zip / PeaZip для Linux.

Кроме того, пакет примеров кода к книге хранится в GitHub, и его можно найти на странице http://bit.ly/node_book_code. Пакеты с кодом примеров для других книг и видеоуроки можно найти на странице https://github.com/PacktPublishing/. Загляните туда!

Список опечаток

Мы приняли все возможные меры, чтобы гарантировать качество наших книг, но ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – в тексте или в коде, мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги. Если вы нашли опечатку, пожалуйста, сообщите о ней, для этого посетите страницу http://www.packtpub.com/submit-errata, выберите название книги, щелкните на ссылке **Errata Submission Form** (Форма отправки сообщения об ошибке) и опишите найденную ошибку. Как только ваше сообщение будет проверено, оно будет принято и добавлено в общий список.

Для просмотра общего списка замеченных опечаток перейдите на страницу https://www.packtpub.com/books/content/support и введите название книги в поле для поиска. Требуемая информация будет показана в разделе **Errata**.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательство Packt очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты copyright@packtpub.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Вопросы

Любые вопросы, касающиеся данной книги, вы можете присылать по адресу questions@packtpub.com. Мы постараемся решить возникшие проблемы.

Глава 1

Добро пожаловать в платформу Node.js

Некоторые принципы и шаблоны проектирования явным образом определяют действия разработчика при работе с платформой Node.js и ее экосистемой. Наиболее своеобразными из них являются: асинхронная обработка и стиль программирования, в значительной степени опирающийся на использование обратных вызовов. Начнем с подробного рассмотрения основополагающих принципов и закономерностей, предназначенных не только для создания корректного кода, но и для принятия эффективных решений, когда речь заходит о больших и сложных задачах.

Другим аспектом, характеризующим платформу Node.js, является ее философия. Освоение платформы Node.js на самом деле означает намного больше, чем просто изучение очередной новой технологии, оно включает ознакомление с ее культурой и сообществом. Здесь будет показано, как эти факторы влияют на способы разработки приложений и компонентов, а также порядок их взаимодействия с компонентами, созданными сообществом.

Помимо этих аспектов, следует учитывать, что в последних версиях платформы Node.js появилась поддержка многих функций, описанных в спецификации ES2015 (другое ее название ES6), делающих язык еще более выразительным и удобным в использовании. Важно освоить эти новые синтаксические и функциональные дополнения, чтобы писать более лаконичный и удобочитаемый код, а также разобраться в альтернативных подходах к реализации шаблонов проектирования, приведенных в этой книге.

В этой главе будут рассмотрены следующие вопросы:

- философия платформы Node.js, «подход Node»;
- версия 6 платформы Node.js и спецификация ES2015;
- шаблон «Реактор» (Reactor) – главный механизм асинхронной архитектуры Node.js.

Философия Node.js

Каждая платформа обладает собственной философией – набором принципов и рекомендаций, подготавливаемых сообществом, которые определяют идеологию развития платформы и порядок проектирования и разработки приложений. Некоторые из этих принципов определяются самой технологией, некоторые – экосистемой, другие берут начало в тенденциях сообщества или являются продуктом эволюции различ-

ных идеологий. Некоторые из принципов платформы Node.js заложены непосредственно ее создателем Райаном Далем (Ryan Dahl) и всеми теми, кто занимался ее ядром, другими яркими фигурами сообщества, а некоторые принципы унаследованы из культуры языка JavaScript или навеяны влиянием философии Unix.

Ни одно из этих правил не является догмой, их применение должно соотноситься со здравым смыслом, но знакомство с ними станет источником вдохновения при разработке собственных программ.

Обширный перечень различных философий разработки программного обеспечения можно найти в Википедии на странице http://en.wikipedia.org/wiki/List_of_software_development_philosophies.

Небольшое ядро

Ядро Node.js разработано в соответствии с несколькими принципами, один из которых заключается в предоставлении минимального набора функциональных возможностей, оставляя реализацию всех дополнительных возможностей за модулями экосистемы, не входящими в ядро. Этот принцип оказывает огромное влияние на платформу Node.js, поскольку дает сообществу полную свободу экспериментов в области применения пользовательских модулей, не навязывая какое-то одно, медленно развивающееся решение, встроенное в жестко контролируемое стабильное ядро. Сведение основного набора функций к минимуму удобно не только с точки зрения обслуживания, но и полезно с точки зрения позитивного воздействия на развитие всей экосистемы.

Небольшие модули

Платформа Node.js использует идею *модуля* как основное средство структурирования программного кода. Модули являются строительными блоками приложений и библиотек, часто называемых *пакетами* (термин «пакет» нередко используется взамен термина «модуль», потому что стало обычным делом, когда пакет состоит из единственного модуля). Один из самых активно продвигаемых принципов платформы Node.js заключается в разработке небольших модулей, не только с точки зрения размера, но и, что более важно, с точки зрения охватываемых возможностей.

Этот принцип исходит из философии Unix, в частности из двух следующих ее заповедей:

- чем меньше, тем лучше;
- любая программа должна делать что-то одно, но делать это хорошо.

Платформа Node.js возносит эти идеи на новый уровень. Официальный диспетчер пакетов платформы Node.js помогает решить проблему зависимостей, гарантируя каждому установленному пакету наличие собственного отдельного набора зависимостей, что позволяет программе использовать множество пакетов без появления конфликтов. Подход Node обеспечивает оптимальный уровень повторного использования кода, поскольку при его применении приложения состоят из большого числа малых, четко направленных зависимостей. Хотя это считается непрактичным или даже совершенно не осуществимым в других платформах, для разработки на платформе Node.js рекомендуется именно такая методика. Как следствие npm-пакеты нередко содержат менее 100 строк кода и предназначены для реализации только одной-единственной функции.

Кроме того, естественные преимущества многократно используемых небольших модулей заключаются в следующем:

- простота понимания и использования;
- легкость тестирования и поддержки;
- оптимальность при обмене с браузером.

Наличие небольших, узкоспециализированных модулей обеспечивает возможность совместного многократного использования фрагментов кода. Это позволяет поднять принцип «не повторяйся» (Don't Repeat Yourself, DRY) на совершенно новый уровень.

Небольшая общедоступная область

Помимо того что модули Node.js должны иметь небольшой размер и конкретную направленность, они обычно экспортируют минимальный набор функциональных возможностей. Основным преимуществом является повышение удобства и четкости программного интерфейса, что уменьшает вероятность неправильного использования. Как правило, пользователям компонента требуется весьма ограниченный, конкретный набор функций, а не расширение его функциональности или погружение в его особенности.

В платформе Node.js широко используется шаблон определения модулей, экспортирующих только одну функциональную возможность, например функцию или конструктор, при этом доступ к более сложным аспектам или дополнительным возможностям осуществляется через свойства, экспортируемые этой функцией или конструктором. Это помогает пользователю разделить важные и вторичные возможности. Часто модули экспортируют только одну функцию и ничего кроме нее, обеспечивая явную единственную точку входа.

Еще одной характерной чертой многих модулей платформы Node.js является их предназначение для использования, а не для расширения. Сокрытие внутреннего содержания модулей, запрещающее любое его расширение, может показаться лишенным гибкости подходом, но он обеспечивает сокращение вариантов использования, упрощает реализацию, облегчает обслуживание и повышает удобство использования.

Простота и прагматизм

Вы когда-нибудь слышали о принципе KISS (Keep It Simple, Stupid – делай это проще, дурачок), запрещающем использование средств более сложных, чем это необходимо, или знаменитую цитату:

«*Все гениальное просто*» (Леонардо да Винчи).

Ричард П. Гэбриел (Richard P. Gabriel), выдающийся специалист в области вычислительной техники, придумал термин «чем хуже, тем лучше» для описания модели, где предпочтение отдается программному обеспечению минимального размера и с простой функциональностью. В своем эссе *The Rise of «Worse is Better»* он пишет:

> «*Архитектура должна быть простой, это касается как реализации, так и интерфейса. Реализация должна быть проще интерфейса. Простота является наиболее важным фактором при разработке*».

Простота архитектуры, как противоположность богатого возможностями программного обеспечения, является хорошей методикой по следующим причинам: тре-

бует меньше усилий для реализации, ускоряет доставку при меньших затратах, облегчает адаптацию, поддержку и понимание. Эти факторы положительно влияют на активность сообщества, что позволяет наращивать и совершенствовать программное обеспечение.

Этот принцип платформы Node.js согласуется с применением JavaScript, который является весьма прагматичным языком. В нем часто используются простые функции, замыкания и литералы объектов вместо сложной иерархии классов. В чистом виде объектно-ориентированные конструкции обычно пытаются воспроизвести реальный мир с помощью математических терминов компьютерной системы, без учета несовершенства и сложности самого реального мира. В действительности программное обеспечение всегда лишь приблизительно отражает реальность и более успешными являются попытки добиться высокого быстродействия при разумной сложности, а не создание близкого к совершенству программного обеспечения посредством огромных усилий и непомерно больших объемов кода.

На протяжении данной книги этот принцип будет использован многократно. Например, значительное число традиционных шаблонов проектирования, таких как «Одиночка» (Singleton) или «Декоратор» (Decorator), могут иметь весьма тривиальную и не всегда самую надежную реализацию, но (как правило) практический, без излишних сложностей подход обеспечивает четкую и ясную архитектуру.

Введение в Node.js 6 и ES2015

На момент написания книги последние главные версии платформы Node.js (4, 5 и 6) включали расширенную поддержку новых особенностей из стандарта ECMAScript 2015 (ES2015, или ES6), направленную на то, чтобы сделать язык JavaScript еще более гибким и удобным.

Некоторые из этих новых особенностей широко будут применяться в примерах кода в данной книге. Их идеи достаточно новы для сообщества Node.js, поэтому стоит коротко остановиться на наиболее важных чертах стандарта ES2015, уже поддерживаемых платформой Node.js в настоящее время. Здесь подразумевается использование версии Node.js 6.

В зависимости от версии Node.js корректная работа некоторых из этих особенностей гарантирует включение **строгого режима**. Для этого достаточно добавить в начало сценария оператор `"use strict"`. Обратите внимание, что оператор `"use strict"` представляет собой обычную строку, и при его записи можно использовать одинарные или двойные кавычки. Для краткости в примерах кода эта строка будет опущена, но следует помнить, что ее необходимо добавлять для правильной работы примеров.

Приведенный ниже перечень не является полным, он служит просто введением в некоторые особенности ES2015, поддерживаемые платформой Node.js, и предназначен для облегчения понимания приведенных в книге примеров кода.

Ключевые слова let и const

Исторически сложилось, что язык JavaScript поддерживает только две области видимости – область видимости функции и глобальную область – для управления временем существования и видимостью переменных. Например, если объявить переменную в теле инструкции `if`, переменная станет доступна вне ее после выполнения тела. Для большей ясности рассмотрим пример:

```
if (false) {
    var x = "hello";
}
console.log(x);
```

Этот код работает не так, как можно было бы ожидать, и выведет в консоль `undefined`. Такое поведение становилось причиной многих ошибок и привносило массу разочарований, что послужило причиной введения в стандарт ES2015 нового ключевого слова `let` для объявления переменных с областью видимости, соответствующей блоку. Изменим предыдущий пример, как показано ниже:

```
if (false) {
   let x = "hello";
}
console.log(x);
```

Попытка выполнить этот код вызовет ошибку `ReferenceError: x is not defined`, поскольку в нем предпринята попытка вывести значение переменной, определенной внутри другого блока.

Чтобы привести более значимый пример, используем ключевое слово `let` для определения временной переменной, применяемой в качестве индекса цикла:

```
for (let i=0; i < 10; i++) {
  // что-то делаем здесь
}
console.log(i);
```

Как и предыдущий пример, этот код вызовет ошибку `ReferenceError: i is not defined`.

Такая защита, обеспечиваемая ключевым словом `let`, позволяет писать более безопасный код, поскольку, если случайно обратиться к переменной, принадлежащей другой области видимости, возникнет ошибка, указывающая на допущенную оплошность, что позволит избежать потенциально опасных побочных эффектов.

Стандарт ES2015 вводит еще одно ключевое слово `const`. Это ключевое слово позволяет объявлять постоянные переменные. Рассмотрим небольшой пример:

```
const x = 'This will never change';
x = '...';
```

Этот код вызовет ошибку `TypeError: Assignment to constant variable`, поскольку предпринята попытка изменить значение константы.

В любом случае важно отметить, что ключевое слово `const` действует не так, как константы во многих других языках, где это ключевое слово позволяет определять переменные, доступные только для чтения. В самом деле, в стандарте ES2015 сказано, что ключевое слово `const` не требует, чтобы присвоенное значение было постоянным, – постоянной должна быть связь со значением. Чтобы пояснить эту идею, покажем, что переменная, объявленная с ключевым словом `const`, допускает, например, следующее:

```
const x = {};
x.name = 'John';
```

При изменении свойства объекта фактически изменяется значение переменной (объект), но связь между переменной и объектом остается неизменной, поэтому этот

код не вызывает ошибок. И наоборот, если попытаться присвоить переменной другое значение, это приведет к изменению связи между переменной и ее значением, что вызовет ошибку:

```
x = null; // Вызовет ошибку
```

Константы очень полезны в ситуациях, когда требуется защитить скалярное значение от случайного изменения где-то в коде или, в более общем случае, когда необходимо защитить переменную с уже присвоенным ей значением от случайного повторного присваивания в другом фрагменте кода.

Ключевое слово `const` рекомендуется использовать при подключении модуля к сценарию, чтобы переменная, которой присвоен модуль, не могла быть случайно переназначена:

```
const path = require('path');
// .. работа с модулем path
let path = './some/path'; // вызовет ошибку
```

Для создания неизменяемого объекта ключевого слова `const` недостаточно, с этой целью следует использовать метод спецификации ES5 `Object.freeze()` (https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze) или модуль `deep-freeze` (https://www.npmjs.com/package/deep-freeze).

Стрелочные функции

Одной из самых ценных возможностей, появившихся в ES2015, является поддержка стрелочных функций. Стрелочные функции поддерживают более лаконичный синтаксис объявления функций, что особенно полезно при определении функций обратного вызова. Чтобы лучше осознать преимущества этого синтаксиса, рассмотрим сначала пример классической реализации фильтрации массива:

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter(function(x) {
  return x%2 === 0;
});
```

Предыдущий код можно переписать иначе, использовав синтаксис стрелочных функций:

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter(x => x%2 === 0);
```

Определение функции, вызываемой функцией `filter`, встраивается непосредственно в вызов, причем ключевое слово `function` опускается, остается только список параметров, за которыми следует `=>` (стрелка), а за ней, в свою очередь, тело функции. Если аргументов несколько, их необходимо заключить в круглые скобки и разделить запятыми. Кроме того, когда аргументы отсутствуют, перед стрелкой необходимо добавить пустые скобки: `() => {...}`. Если тело функции содержит лишь одну строку, отпадает необходимость в ключевом слове `return`, поскольку оно применяется неявно. Если потребуется поместить в тело функции несколько строк кода, можно заключить их в фигурные скобки, но имейте в виду, что в этом случае ключевое слово `return` автоматически не подразумевается и его следует указывать явно, как показано в следующем примере:

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter(x => {
  if (x%2 === 0) {
    console.log(x + ' is even!');
    return true;
  }
});
```

Стрелочные функции обладают еще одной важной особенностью, которую необходимо учитывать: они привязаны к лексической области видимости. Это означает, что внутри стрелочной функции доступны те же значения, что и в родительском блоке. Поясним это на примере:

```
function DelayedGreeter(name) {
  this.name = name;
}

DelayedGreeter.prototype.greet = function() {
  setTimeout( function cb() {
    console.log('Hello ' + this.name);
  }, 500);
};

const greeter = new DelayedGreeter('World');
greeter.greet(); // выведет "Hello undefined"
```

В этом коде определяется простой прототип `greeter`, принимающий аргумент с именем. Затем в прототип добавляется метод `greet`. Эта функция должна вывести `Hello` и имя, определенное в текущем экземпляре, через 500 миллисекунд после ее вызова. Но эта функция работает неправильно, поскольку область видимости внутри функции обратного вызова таймера (`cb`) отличается от области видимости метода `greet` и значение `this` в ней не определено (`undefined`).

До того как в платформе Node.js появилась поддержка стрелочных функций, для исправления этой ошибки необходимо было выполнять связывание функции `greet` с помощью `bind`:

```
DelayedGreeter.prototype.greet = function() {
  setTimeout( (function cb() {
    console.log('Hello' + this.name);
  }).bind(this), 500);
};
```

Но поскольку теперь имеются стрелочные функции, и они привязаны к своей лексической области, для решения этой проблемы можно просто использовать стрелочную функцию:

```
DelayedGreeter.prototype.greet = function() {
  setTimeout( () => console.log('Hello' + this.name), 500);
};
```

Это очень удобные функции, их применение часто позволяет сократить и упростить код.

Синтаксис классов

Стандарт ES2015 вводит новый синтаксис наследования прототипов, делающий его более похожим на наследование в классических объектно-ориентированных языках, таких как Java и C#. Важно подчеркнуть, что новый синтаксис не изменяет внутреннего подхода к управлению объектами среды выполнения JavaScript, они по-прежнему наследуют свойства и функции через прототипы, а не через классы. Хотя новый альтернативный синтаксис весьма удобен и читабелен, важно понимать, что это всего лишь синтаксический сахар.

Рассмотрим его применение на тривиальном примере. Начнем с функции `Person`, реализованной с использованием классического подхода, базирующегося на прототипах:

```
function Person(name, surname, age) {
  this.name = name;
  this.surname = surname;
  this.age = age;
}

Person.prototype.getFullName = function() {
  return this.name + '' + this.surname;
};

Person.older = function(person1, person2) {
  return (person1.age >= person2.age) ? person1 : person2;
};
```

Как видите, объект `Person` содержит свойства `name` (имя), `surname` (фамилия) и `age` (возраст). Прототип снабжается вспомогательной функцией, которая облегчает получение полного имени объекта `Person`, и универсальной вспомогательной функцией, доступной непосредственно из прототипа `Person`, возвращающей старший из двух экземпляров объекта `Person`, переданных в качестве входных данных.

Теперь посмотрим, как реализовать тот же пример, воспользовавшись удобным синтаксисом классов из ES2015:

```
class Person {
  constructor (name, surname, age) {
    this.name = name;
    this.surname = surname;
    this.age = age;
  }

  getFullName () {
    return this.name + ' ' + this.surname;
  }

  static older (person1, person2) {
    return (person1.age >= person2.age) ? person1 : person2;
  }
}
```

Этот синтаксис удобнее для чтения и проще для понимания. Он явно обозначает конструктор класса `constructor` и объявляет функцию `older` статическим методом.

Две приведенные выше реализации полностью взаимозаменяемы, но важнейшей особенностью нового синтаксиса является возможность расширения прототипа

Person с помощью ключевых слов extend и super. Предположим, что требуется создать класс PersonWithMiddlename:

```
class PersonWithMiddlename extends Person {
  constructor (name, middlename, surname, age) {
    super(name, surname, age);
    this.middlename = middlename;
  }

  getFullName () {
    return this.name + '' + this.middlename + '' + this.surname;
  }
}
```

Главное, что стоит отметить в этом третьем примере, – его синтаксис действительно напоминает синтаксис других объектно-ориентированных языков. В объявлении указывается наследуемый класс, определяется новый конструктор, вызывающий конструктор родительского класса с помощью ключевого слова super, а затем переопределяется метод getFullName для поддержки отчества.

Расширенные литералы объектов

Наряду с новым синтаксисом классов стандарт ES2015 добавляет синтаксис расширенных литералов объектов. Этот синтаксис обеспечивает лаконичное определение переменных и функций как членов объекта, позволяет определять имена вычисляемых элементов во время их создания, а также удобные методы чтения/записи значений свойств.

Поясним это на примерах:

```
const x = 22;
const y = 17;
const obj = { x, y };
```

Константа obj – это объект, содержащий ключи x и y со значениями 22 и 17 соответственно. То же самое можно проделать с функциями:

```
module.exports = {
  square (x) {
    return x * x;
  },
  cube (x) {
    return x * x * x;
  }
};
```

В данном случае будет создан модуль, экспортирующий функции square и cube, доступные как свойства с такими же именами. Обратите внимание, что не нужно указывать ключевое слово function.

Рассмотрим другой пример, демонстрирующий использование вычисляемых свойств:

```
const namespace = '-webkit-';
const style = {
  [namespace + 'box-sizing'] : 'border-box',
  [namespace + 'box-shadow'] : '10px10px5px #888888'
};
```

Здесь получится объект со свойствами `-webkit-box-sizing` и `-webkit-box-shadow`.

А теперь рассмотрим пример использования нового синтаксиса для определения методов чтения/записи:

```
const person = {
  name : 'George',
  surname : 'Boole',

  get fullname () {
    return this.name + '' + this.surname;
  },

  set fullname (fullname) {
    let parts = fullname.split('');
    this.name = parts[0];
    this.surname = parts[1];
  }
};

console.log(person.fullname); // "George Boole"
console.log(person.fullname = 'Alan Turing'); // "Alan Turing" console.log(person.name); // "Alan"
```

В этом примере с помощью синтаксиса методов чтения/записи определяются три свойства: два обычных, `name` и `surname`, и вычисляемое `fullname`. Как видно из результатов вызовов метода `console.log`, к вычисляемому свойству можно обращаться как к самому обычному свойству – читать его значение и записывать в него новые значения. Обратите внимание, что второй вызов метода `console.log` выводит `Alan Turing`. Это связано с тем, что по умолчанию любая функция `set` возвращает значение, предоставляемое функцией `get` этого же свойства, в данном случае это `get fullname`.

Коллекции Map и Set

Для создания хешированных коллекций пар ключ/значение разработчики на JavaScript используют обычные объекты. Стандарт ES2015 вводит новый прототип `Map`, специально предназначенный для использования хешированных коллекций пар ключ/значение, более безопасным, гибким и интуитивно понятным способом. Рассмотрим небольшой пример:

```
const profiles = new Map();
profiles.set('twitter', '@adalovelace');
profiles.set('facebook', 'adalovelace');
profiles.set('googleplus', 'ada');

profiles.size; // 3
profiles.has('twitter'); // true
profiles.get('twitter'); // "@adalovelace"
profiles.has('youtube'); // false
profiles.delete('facebook');
profiles.has('facebook'); // false
profiles.get('facebook'); // undefined
for (const entry of profiles) {
  console.log(entry);
}
```

Как видите, прототип `Map` имеет несколько удобных методов, например `set`, `get`, `has`, `delete`, и атрибут `size` (обратите внимание, что в массивах вместо атрибута `size` используется атрибут `length`). Кроме того, имеется возможность перебора всех записей с помощью синтаксиса `for...of`. Каждая извлекаемая в цикле запись будет массивом, содержащим ключ в первом элементе и значение – во втором. Это вполне интуитивный и не требующий пояснений интерфейс.

Но самое главное, что делает коллекции пар ключ/значение действительно интересными, – это возможность использования функций и объектов в качестве ключей, что невозможно при использовании простых объектов, так как в объектах все ключи автоматически преобразуются в строки. Это открывает новые возможности. Например, используя данную особенность, можно создать небольшую среду для тестирования:

```
const tests = new Map();
tests.set(() => 2+2, 4);
tests.set(() => 2*2, 4);
tests.set(() => 2/2, 1);
for (const entry of tests) {
  console.log((entry[0]() === entry[1]) ? 'PASS' : 'FAIL');
}
```

Как показано в предыдущем примере, мы сохраняем функции как ключи, а ожидаемые результаты – как значения. Затем выполняем обход коллекции пар ключ/значение, вызывая все функции. Следует также отметить, что при обходе коллекции пар ключ/значение пары извлекаются в том же порядке, в каком были сохранены, что не гарантируется обычными объектами.

Наряду с прототипом `Map` стандарт ES2015 также вводит прототип `Set`. Он позволяет создавать множества, или списки уникальных значений:

```
const s = new Set([0, 1, 2, 3]);
s.add(3); // не будет добавлено
s.size; // 4
s.delete(0); s.has(0); // false

for (const entry of s) {
  console.log(entry);
}
```

Как видите, интерфейс `Set` очень похож на интерфейс `Map`. Здесь имеются методы `add` (вместо `set`), `has`, `delete` и свойство `size`. Также существует возможность перебора элементов множества, которые в данном случае являются значениями, в нашем примере – числами. И наконец, множества могут хранить объекты и функции.

Коллекции WeakMap и WeakSet

Стандарт ES2015 определяет также и «слабые» версии прототипов `Map` и `Set`, с именами `WeakMap` и `WeakSet`.

Коллекция `WeakMap` очень похожа на `Map` с точки зрения интерфейса, но между ними имеются два существенных различия, которые следует учитывать: в `WeakMap` отсутствует возможность обхода элементов в цикле, и она позволяет использовать в качестве ключей только объекты. Невозможность перебора выглядит существенным недостатком, но тому есть веская причина. В самом деле, коллекция `WeakMap` имеет следующую отличительную особенность: она позволяет сборщику мусора утилизи-

ровать объекты, используемые как ключи, как только исчезают последние ссылки на них за пределами WeakMap. Это особенно полезно при хранении метаданных, связанных с объектом, которыми можно удалять в течение обычного жизненного цикла. Рассмотрим пример:

```
let obj = {};
const map = new WeakMap();
map.set(obj, {key: "some_value"});
console.log(map.get(obj)); // {key: "some_value"}
obj = undefined; // теперь obj и связанные с ним данные
                 // будут очищены при следующей сборке мусора
```

Здесь создается обычный объект obj. Затем во вновь созданной коллекции WeakMap с именем map сохраняются некие метаданные этого объекта. К этим метаданным можно получить доступ с помощью метода map.get. Затем, после удаления объекта путем присваивания ему значения undefined, объект будет корректно утилизирован сборщиком мусора вместе с его метаданными в коллекции.

Подобно WeakMap, коллекция WeakSet является слабой версией коллекции Set, которая поддерживает интерфейс, похожий на интерфейс коллекции Set, но позволяет хранить только объекты и не дает возможности выполнять обход элементов. Опять же, разница Set и WeakSet заключается в том, что объекты очищаются сборщиком мусора после освобождения ссылок на них за пределами WeakSet:

```
let obj1= {key: "val1"};
let obj2= {key: "val2"};
const set= new WeakSet([obj1, obj2]);
console.log(set.has(obj1)); // true
obj1= undefined; // теперь obj1 удален из set
console.log(set.has(obj1)); // false
```

Важно понимать, что WeakMap и WeakSet не лучше или хуже Map и Set, они просто лучше подходят для определенных случаев.

Литералы шаблонов

Стандарт ES2015 предлагает новый альтернативный, более мощный синтаксис определения строк: литералы шаблонов. Этот синтаксис использует обратные кавычки (`) в качестве ограничителей, что обеспечивает определенные преимущества, по сравнению со строками в обычных одиночных (') или двойных кавычках ("). Главные преимущества заключаются в возможности интерполировать переменные или выражения внутрь строк с помощью конструкций ${expression} (именно поэтому этот синтаксис носит название «шаблон») и разместить одну строку в нескольких строках в исходном коде. Рассмотрим небольшой пример:

```
const name = "Leonardo";
const interests = ["arts", "architecture", "science", "music",
                   "mathematics"];
const birth = { year : 1452, place : 'Florence' };
const text = `${name} was an Italian polymath
  interested in many topics such as
  ${interests.join(', ')}.He was born
  in ${birth.year} in ${birth.place}.`;
  console.log(text);
```

Этот код выведет следующее:

```
Leonardo was an Italian polymath interested in many topics such as arts, architecture, science, music,
mathematics.
He was born in 1452 in Florence.
```

Загрузка примеров кода
Подробное описание действий по загрузке примеров кода было приведено в предисловии в книге. Кроме того, пакет с примерами размещен в GitHub на странице http://bit.ly/node_book_code.
Примеры ко многим другим книгам и видеоуроки можно найти на странице https://github.com/PacktPublishing/.

Другие особенности ES2015

Еще одной чрезвычайно интересной особенностью, добавленной в стандарте ES2015 и доступной начиная с версии Node.js 4, является поддержка объектов `Promise`. Подробно эти объекты будут рассмотрены в *главе 4* «Шаблоны асинхронного выполнения с использованием спецификации ES2015, и не только».

Ниже приводится перечень других интересных возможностей, предоставляемых стандартом ES2015 и ставших доступными начиная с версии Node.js 6:

- значения по умолчанию параметров функций;
- дополнительные параметры;
- оператор расширения;
- деструктивное присваивание;
- `new.target` (рассматривается в главе 2 «Основные шаблоны Node.js»);
- прокси-объекты (рассматривается в главе 6 «Шаблоны проектирования»);
- объект `Reflect`;
- символы.

Более широкий и актуальный список возможностей, поддерживаемых стандартом ES2015, можно найти в официальной документации с описанием Node.js, на странице https://nodejs.org/en/docs/es6/.

Шаблон Reactor

Этот раздел посвящен шаблону Reactor (Реактор), лежащему в основе асинхронной природы платформы Node.js. Рассмотрев основные идеи данного шаблона, такие как однопоточная архитектура и неблокирующие операции ввода/вывода, мы убедимся, что он является фундаментом всей платформы Node.js.

Медленный ввод/вывод

Операции ввода/вывода, несомненно, являются самыми медленными из всех основных операций в компьютере. Доступ к ОЗУ занимает несколько наносекунд (10^{-9} секунды), а доступ к данным на диске или в сети занимает несколько миллисекунд (10^{-3} секунды). С пропускной способностью та же история: ОЗУ имеет скорость передачи порядка нескольких ГБ/с, в то время как для диска и сети эта скорость варьируется от нескольких МБ/с до гипотетического ГБ/с. Операции ввода/вывода обычно не требуют особых затрат ресурсов центрального процессора, но привносят задержку между отправкой запроса и завершением операции. Следует также учи-

тывать человеческий фактор, поскольку очень часто ввод в приложении выполняет реальный человек, например щелкает на кнопках или отсылает сообщения в чат, поэтому скорость и частота операций ввода/вывода зависят не только от технических аспектов, что может сделать их на много порядков медленнее, чем скорость доступа к диску или сети.

Блокирующий ввод/вывод

При использовании традиционного, блокирующего механизма ввода/вывода вызов функции, соответствующий запросу ввода/вывода, будет блокировать выполнение потока до завершения операции. Это может продолжаться от нескольких миллисекунд, в случае доступа к диску, до нескольких минут и дольше, если поступление данных зависит от действий пользователя, например ожидание нажатия клавиши. Следующий псевдокод демонстрирует типичную операцию с сокетом, блокирующую работу потока:

```
// поток блокируется, пока не будут доступны данные
data = socket.read();
//данные доступны
print(data);
```

Нетрудно понять, что веб-сервер, реализующий блокирующий ввод/вывод, не в состоянии обрабатывать несколько соединений в одном потоке. Любая операция ввода/вывода через сокет будет блокировать обработку других соединений. По этой причине для параллельной обработки в веб-серверах создается новый поток или процесс (или повторно используется один из имеющихся в пуле) для каждого из обрабатываемых подключений. Таким образом, блокировка потока операцией ввода/вывода не влияет на обработку других запросов, поскольку все они обрабатываются в отдельных потоках.

Приведенная на рис. 1.1 схема иллюстрирует такой сценарий.

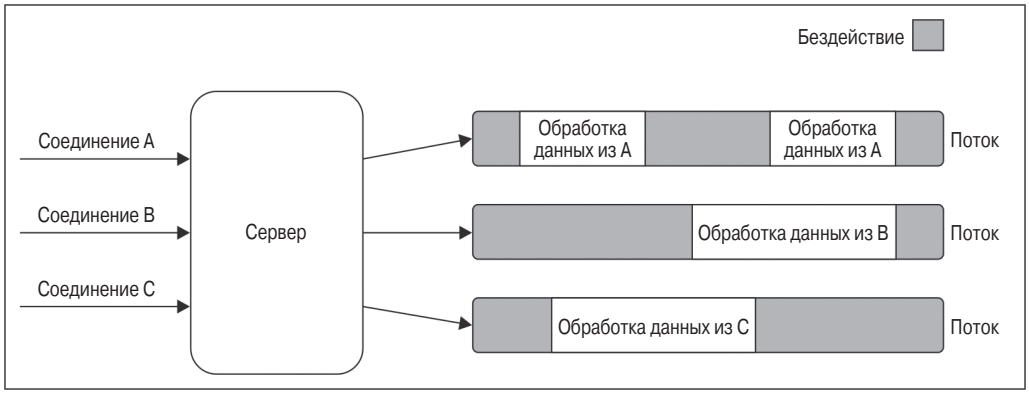

Рис. 1.1 ❖ Параллельная обработка нескольких подключений

Приведенная на рис. 1.1 схема наглядно показывает, сколько времени потоки находятся в состоянии простоя, ожидая новых данных, получаемых из связанных с ними соединений. Если вдобавок учесть, что ввод/вывод может блокировать запрос, на-

пример при взаимодействии с базами данных или файловой системой, становится понятным, сколько раз каждый из потоков будет заблокирован в ожидании результатов операций ввода/вывода. К сожалению, потоки потребляют значительные объемы системных ресурсов – расходуют память и вызывают переключение контекста, поэтому иметь достаточно долго выполняющийся поток для каждого подключения и не использовать его большую часть времени является далеко не лучшим решением с точки зрения эффективности.

Неблокирующий ввод/вывод

Помимо блокирующего ввода/вывода большинство современных операционных систем поддерживают и другой механизм доступа к ресурсам, называемый неблокирующим вводом/выводом. При его использовании системные вызовы немедленно возвращают управление, не ожидая выполнения чтения или записи данных. Если на момент вызова отсутствуют какие-либо результаты, функция возвращает предварительно определенную константу, указывающую на невозможность вернуть в этот момент данные.

Например, в операционных системах семейства Unix для включения неблокирующего режима работы существующего дескриптора файла (с флагом `O_NONBLOCK`) используется функция `fcntl()`. После перевода ресурса в неблокирующий режим любая операция чтения завершится кодом возврата `EAGAIN`, если ресурс не имеет готовых для чтения данных.

Основным шаблоном реализации неблокирующего ввода/вывода является активный опрос ресурса в цикле, пока он не сможет вернуть реальные данные. Этот шаблон носит название **цикл ожидания** (busy-waiting). Следующий псевдокод демонстрирует чтение из нескольких ресурсов с помощью неблокирующего ввода/вывода и опроса в цикле:

```
resources = [socketA, socketB, pipeA];
while(!resources.isEmpty()) {
  for(i = 0; i < resources.length; i++) {
    resource = resources[i];
    //попытка чтения
    let data = resource.read();
    if(data === NO_DATA_AVAILABLE)
      //на данный момент нет данных для чтения
      continue;
    if(data === RESOURCE_CLOSED)
      //ресурс закрыт, удаляем его из списка
      resources.remove(i);
    else
      //данные получены, обрабатываем их
      consumeData(data);
  }
}
```

Как видите, с помощью этой простой методики можно обрабатывать несколько ресурсов в одном потоке, но она не слишком эффективна. В самом деле, в предыдущем примере цикл тратит драгоценное время центрального процессора на обход ресурсов, недоступных большую часть времени. Алгоритмы опроса обычно отличаются огромным количеством времени центрального процессора, потерянного зря.

Демультиплексирование событий

Цикл ожидания определенно не является идеальным способом неблокирующей работы с ресурсами, но, к счастью, большинство современных операционных систем поддерживает собственный, более эффективный механизм параллельной, неблокирующей работы с ресурсами. Этот механизм называется **синхронным демультиплексированием событий**, или **интерфейсом уведомления о событиях**. При его использовании осуществляются сборка и постановка в очередь событий ввода/вывода, поступающих из набора наблюдаемых ресурсов, и блокировка появления новых, доступных для обработки событий. Следующий псевдокод демонстрирует алгоритм синхронного демультиплексирования событий для чтения из двух ресурсов:

```
socketA, pipeB;
watchedList.add(socketA, FOR_READ);                  //[1]
watchedList.add(pipeB, FOR_READ);
while(events = demultiplexer.watch(watchedList)) {   //[2]
  //цикл событий
  foreach(event in events) {                         //[3]
    //Чтение нового блока с обязательным возвратом данных
    data = event.resource.read();
    if(data === RESOURCE_CLOSED)
      //при закрытии ресурса он удаляется из списка наблюдаемых
      demultiplexer.unwatch(event.resource);
    else
      //при поступлении реальных данных обрабатываем их
      consumeData(data);
  }
}
```

Ниже описываются три самых важных шага в предыдущем псевдокоде.

1. Ресурсы добавляются в структуру данных вместе с конкретными операциями, в этом примере с операцией чтения `read`.
2. Механизм оповещения о событиях настраивается на слежение за группой ресурсов. Этот вызов является синхронным и блокирует выполнение до готовности любого из наблюдаемых ресурсов к чтению. Когда это происходит, демультиплексор возобновляет работу и становится доступным для обработки нового набора событий.
3. Обрабатывается любое событие, возвращаемое демультиплексором. В данной точке ресурс, связанный с событием, гарантированно готов к чтению и не блокирует выполнения. После обработки всех событий поток вновь блокируется демультиплексором событий до доступности для обработки новых событий. Это называется **циклом событий**.

Интересно посмотреть на работу этого шаблона при обработке нескольких операций ввода/вывода внутри одного потока, без использования цикла ожидания. На рис. 1.2 представлена схема обработки веб-сервером нескольких соединений при помощи синхронного демультиплексирования событий и одного потока.

Схема на рис. 1.2 помогает понять порядок параллельной обработки в однопоточном приложении с помощью синхронного демультиплексирования событий и неблокирующего ввода/вывода. Как видите, использование одного-единственного потока не исключает возможности одновременного выполнения нескольких задач, связан-

ных с вводом/выводом. Выполнение задач распределено по времени, а не разделено на несколько потоков. Явное преимущество заключается в сведении к минимуму общего времени простоя потока, как это видно на схеме. Но это не единственная причина выбора данной модели. На самом деле наличие единственного потока также оказывает благотворное влияние на весь подход к реализации параллельной обработки в целом. Позже будет показано, как отсутствие конкуренции и синхронизации нескольких потоков позволяет воспользоваться гораздо более простой стратегией параллельной обработки.

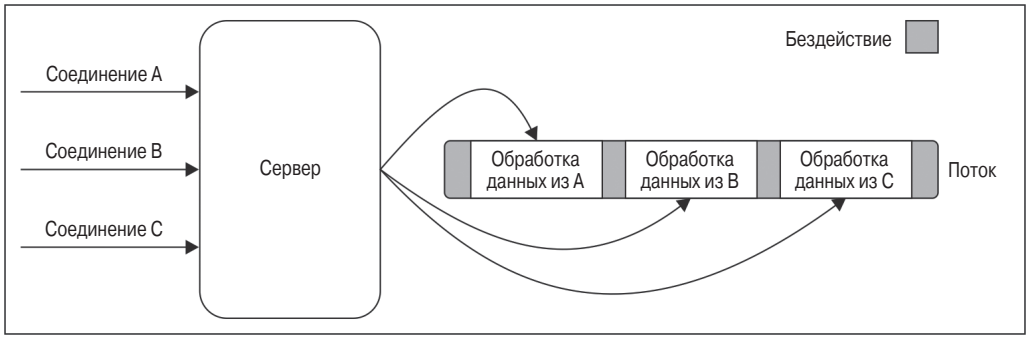

Рис. 1.2 ❖ Схема обработки нескольких соединений при помощи синхронного демультиплексирования

В следующей главе модель параллельной обработки, используемой в платформе Node.js, будет рассмотрена более подробно.

Введение в шаблон Reactor

Теперь можно переходить к рассмотрению шаблона Reactor (Реактор), представляющего конкретизацию алгоритмов, рассматривавшихся в предыдущем разделе. Основная его идея заключается в наличии обработчика (который в Node.js представлен функцией обратного вызова), связанного с любой операцией ввода/вывода, который будет вызван непосредственно при появлении события в цикле событий. Структура шаблона Reactor приводится на рис. 1.3.

Вот что происходит в приложении, использующем шаблон Reactor:

1) приложение создает новую операцию ввода/вывода, передав запрос **демультиплексору событий**. Также приложение определяет обработчика для вызова после завершения операции. Отправка нового запроса **демультиплексору событий** не приводит к блокировке, управление немедленно возвращается приложению;
2) после завершения обработки набора операций ввода/вывода **демультиплексор событий** добавляет новые события в **очередь событий**;
3) в этом месте **цикл событий** выполняет обход элементов в **очереди событий**;
4) для каждого события вызывается соответствующий обработчик;
5) обработчик, являющийся частью кода приложения, возвращает управление **циклу событий** (**5a**). Однако во время выполнения обработчика могут запрашиваться новые асинхронные операции (**5b**), что приводит к добавлению новых операций в **демультиплексор событий** (1) до возврата управления **циклу событий**;

6) после обработки всех элементов **очереди событий** цикл вновь заблокируется **демультиплексором событий** и повторится при появлении нового события.

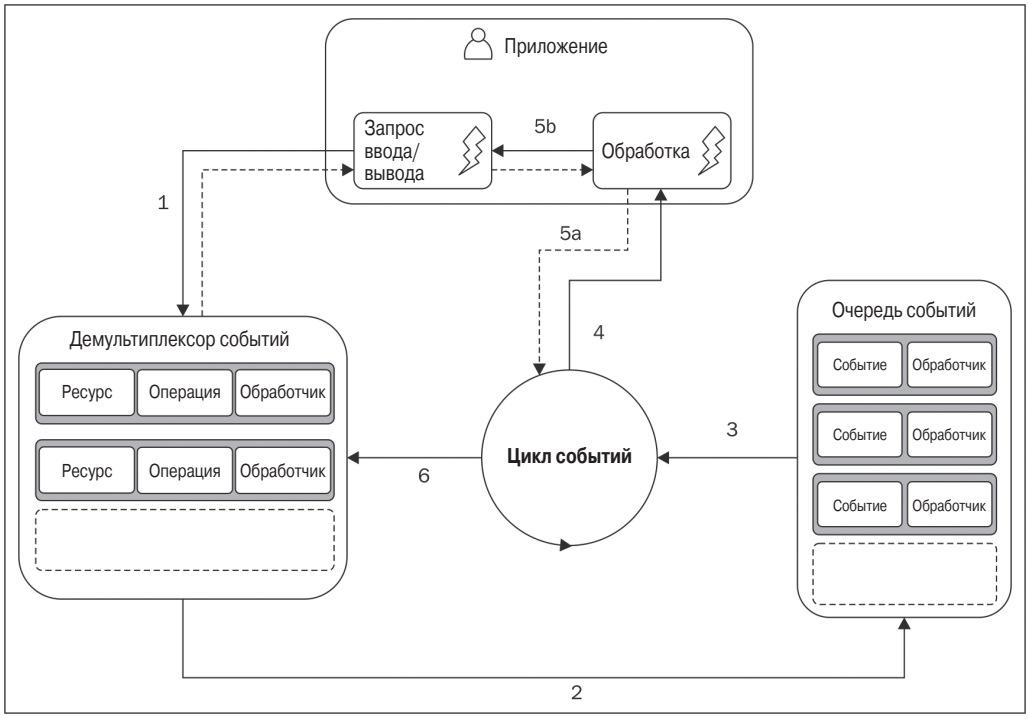

Рис. 1.3 ❖ Структура шаблона Reactor

Суть асинхронной обработки заключается в следующем: приложение в некоторый момент времени желает обратиться к ресурсу (без блокировки) и передает обработчика, который должен быть вызван некогда в будущем, после завершения операции.

Приложение на платформе Node.js завершится автоматически, когда в демультиплексоре событий не останется отложенных операций и событий в очереди.

Теперь определим шаблон в терминах Node.js.

Шаблон Reactor обеспечивает обработку операций ввода/вывода, блокируя выполнение до момента доступности новых событий из набора наблюдаемых ресурсов с последующей обработкой каждого события вызовом связанного с ним обработчика.

Неблокирующий движок libuv платформы Node.js

Каждая операционная система имеет собственный интерфейс **демультиплексора событий**: `epoll` в Linux, `kqueue` в Mac OS X и программный интерфейс **I/O Completion Port (IOCP)** в Windows. Кроме того, любая операция ввода/вывода может вести себя по-разному в зависимости от типа ресурса, даже в пределах одной операционной системы. Например, в Unix обычные файлы не поддерживают неблокирующих операций, поэтому для имитации неблокирующей модели поведения необходи-

мо использовать отдельный поток вне цикла событий. Подобные несоответствия внутри и между различными операционными системами требуют более высокого уровня абстракции для демультиплексора событий. Именно для этого команда разработчиков ядра платформы Node.js создала C-библиотеку **libuv**, обеспечивающую совместимость Node.js со всеми основными платформами и нормализующую неблокирующую модель поведения для различных типов ресурсов. На настоящий момент библиотека libuv является низкоуровневым движком ввода/вывода платформы Node.js.

Помимо абстрагирования основных системных вызовов, библиотека libuv реализует шаблон Reactor, обеспечивая программный интерфейс для создания циклов событий, управления очередью событий, выполнения асинхронных операций ввода/вывода и организации очереди заданий разных типов.

Более подробные сведения о библиотеке libuv можно найти в бесплатной электронной книге Никхил Маразе (Nikhil Marathe): http://nikhilm.github.io/uvbook/.

Рецепт платформы Node.js

Шаблон Reactor и библиотека libuv являются основными строительными блоками Node.js, но для построения полной платформы необходимы еще три компонента:

- набор привязок, ответственных за обертывание и использование библиотеки libuv, а также других низкоуровневых JavaScript-функций;
- JavaScript-движок **V8**, изначально разработанный компанией Google для браузера Chrome. Он является одной из причин быстроты и эффективности Node.js. ДвижокV8 отличается революционным дизайном, высокой скоростью и эффективным управлением памятью;
- ядро JavaScript-библиотеки (по-другому **node-core**), реализующее высокоуровневый программный интерфейс Node.js.

Вот и весь рецепт платформы Node.js, полная архитектура которой представлена на рис. 1.4.

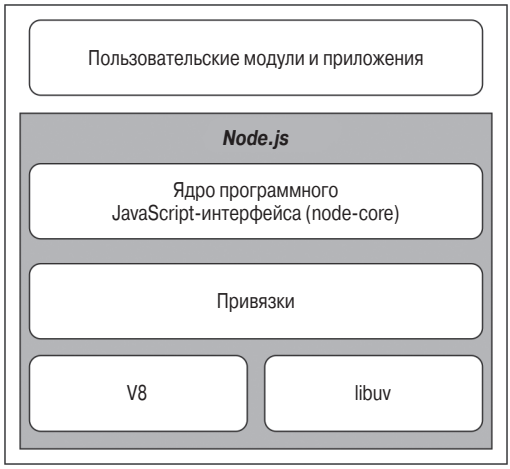

Рис. 1.4 ❖ Архитектура платформы Node.js

Итоги

В этой главе мы узнали, что платформа Node.js основана на нескольких важных принципах, которые обеспечивают основу для создания эффективного и многократного используемого кода. Философия и выбор подхода к проектированию самой платформы оказывают сильное влияние на структуру и поведение любого создаваемого на ней приложения и модуля. Обычно разработчикам, переходящим на нее с другой технологии, эти принципы незнакомы, и у них возникает обычная инстинктивная реакция, направленная на борьбу с этими изменениями, они пытаются найти знакомые модели в мире, требующем для своего понимания реальных сдвигов в мышлении.

С одной стороны, асинхронный характер шаблона Reactor требует другого стиля программирования, основанного на обратных вызовах и происходящего позднее действий, исключает заботы о потоках и состоянии конкуренции. С другой стороны, шаблон Module (Модуль), с его простотой и минимализмом, создает новые интересные сценарии повторного использования кода, обеспечивающими простоту обслуживания и удобство использования.

И наконец, помимо очевидных чисто технических преимуществ, заключающихся в быстроте, эффективности и использовании JavaScript в качестве основы, платформа Node.js вызывает большой интерес также из-за своих принципов, которые были здесь рассмотрены. Многие, осознав сущность этого мира, воспринимают его как возвращение к истокам, к более гуманному способу программирования, применительно к объемам кода и его сложности, вследствие чего разработчики в конечном итоге влюбляются в Node.js. Появление ES2015 делает платформу еще более интересной и обеспечивает новые сценарии, позволяющие воспользоваться всеми ее преимуществами посредством еще более выразительного синтаксиса.

В следующей главе будут рассмотрены два основных шаблона асинхронной обработки, используемые в Node.js: Callback (Обратный вызов) и Event Emitter (Генератор событий). Будет также пояснена разница между синхронным и асинхронным кодом и как можно избежать написания непредсказуемых функций.

Глава 2

Основные шаблоны Node.js

Освоение асинхронной природы платформы Node.js, вообще говоря, не тривиальная задача, особенно для тех, кто переходит на нее с таких языков, как PHP, где обычно не приходится иметь дело с асинхронным кодом.

В синхронном программировании используется привычная концепция, представляющая код как ряд последовательных вычислений, направленных на решение конкретной задачи. Любая операция является блокирующей, поскольку только после ее завершения можно переходить к выполнению следующей. Такой подход упрощает понимание и отладку кода.

Напротив, в асинхронном программировании некоторые операции, например чтение файла или сетевой запрос, могут выполняться в фоновом режиме. После вызова асинхронной операции следующая операция выполняется немедленно, даже если предыдущая операция еще не закончила выполняться. Операции в фоновом режиме могут закончить свое выполнение в любое время, и все приложение должно быть реализовано так, чтобы должным образом отреагировать на завершение асинхронного вызова.

Хотя этот неблокирующий подход практически всегда гарантирует более высокую производительность, по сравнению с всегда блокирующим сценарием, он привносит трудную для восприятия парадигму, которая может стать весьма громоздкой в больших приложениях, требующих сложного управления потоками.

Платформа Node.js предлагает ряд инструментов и шаблонов проектирования для разработки оптимального асинхронного кода. Важно освоить их, чтобы уверенно ими пользоваться для создания производительных приложений, простых для понимания и отладки.

В этой главе будут рассмотрены два наиболее важных шаблона асинхронной обработки: Callback (Обратный вызов) и Event Emitter (Генератор событий).

Шаблон Callback

Обратные вызовы являются воплощением обработчиков в шаблоне Reactor (Реактор), о котором шла речь в предыдущей главе. Они относятся к одной из особенностей, которые определяют стиль программирования на платформе Node.js. Обратные вызовы – это функции, вызываемые для передачи результата операции, и это именно то, что необходимо при работе с асинхронными операциями. Они заменяют использование инструкции `return`, которая всегда выполняется синхронно. Язык JavaScript

отлично подходит для реализации шаблона Callback, поскольку, как уже упоминалось, функции в нем являются самыми обычными объектами и легко могут присваиваться переменным, передаваться в аргументах, возвращаться из других функций и храниться в структурах данных.

Еще одной конструкцией, идеально подходящей для реализации шаблона Callback, является **замыкание**. Действительно, с помощью замыканий можно ссылаться на окружение, где была создана функция, и всегда иметь доступ к контексту асинхронной операции, независимо от того, где и когда будет осуществлен обратный вызов.

Чтобы освежить сведения о замыканиях, можно обратиться к статье на сайте Mozilla Developer Network: https://developer.mozilla.org/ru/docs/Web/JavaScript/Closures.

В этом разделе будет рассмотрен особый стиль программирования, основанный на обратных вызовах, а не инструкциях `return`.

Стиль передачи продолжений

В JavaScript обратный вызов представляет собой функцию, переданную в аргументе другой функции и вызываемую после завершения операции. В функциональном программировании этот способ передачи результата называется **стилем передачи продолжений** (Continuation-Passing Style, CPS). Это общая концепция, и она не всегда связана с асинхронными операциями. В самом деле, эта концепция просто указывает, что результат будет передан в другую функцию (обратного вызова) вместо возврата в непосредственно вызвавшую функцию.

Синхронный стиль передачи продолжений

Для пояснения этой идеи рассмотрим простую синхронную функцию:

```
function add(a, b) {
  return a + b;
}
```

Здесь нет ничего особенного, поскольку результат возвращается вызвавшей функции с помощью инструкции `return`. Это так называемый **прямой стиль**, который является наиболее распространенным способом возврата результата в синхронном программировании. Эквивалентная ему запись предыдущей функции в стиле передачи продолжения будет выглядеть следующим образом:

```
function add(a, b, callback) {
  callback(a + b);
}
```

Эта функция `add()` является синхронной CPS-функцией, то есть она вернет значение, только когда функция обратного вызова завершит выполнение. Следующий код иллюстрирует это утверждение:

```
console.log('before');
add(1, 2, result => console.log('Result: ' + result));
console.log('after');
```

Поскольку функция `add()` является синхронной, предыдущий код выведет следующее:

```
before
Result: 3
after
```

Асинхронный стиль передачи продолжений

Теперь рассмотрим случай, когда функция `add()` является асинхронной:

```
function additionAsync(a, b, callback) {
  setTimeout(() => callback(a + b), 100);
}
```

В этом примере для имитации асинхронного вызова используется функция `setTimeout()`, которой передается функция обратного вызова. Попробуем вызвать функцию `additionAsync` и убедимся, что порядок выполнения операторов изменился:

```
console.log('before');
additionAsync(1, 2, result => console.log('Result: ' + result));
console.log('after');
```

Этот код выведет следующее:

```
before
after
Result: 3
```

Поскольку `setTimeout()` запускает асинхронную операцию, она не будет ждать завершения функции обратного вызова и немедленно вернет управление функции `additionAsync()`, а та, в свою очередь, вызвавшей ее функции. Это свойство платформы Node.js имеет решающее значение, поскольку обеспечивает передачу управления обратно циклу событий, как только будет сделан асинхронный запрос, позволяя, таким образом, обработать новое событие из очереди.

Это иллюстрирует схема на рис. 2.1.

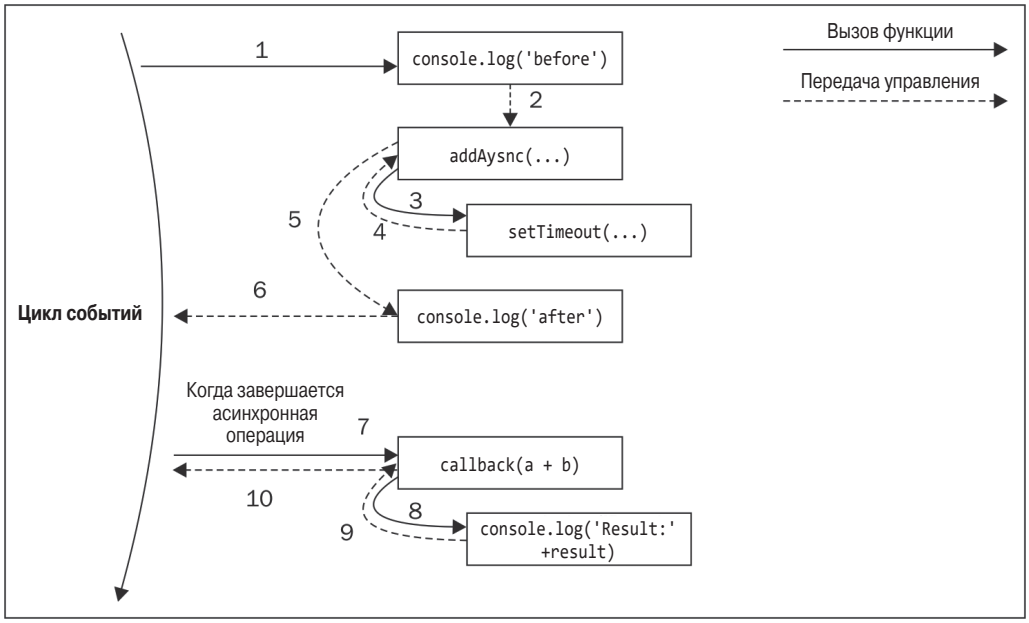

Рис. 2.1 ❖ Асинхронная обработка событий

После завершения асинхронной операции выполнение возобновляется, начиная с функции обратного вызова, переданной в асинхронную функцию, которая ее вызвала. Выполнение запускается **циклом событий**, поэтому изначально мы имеем пустой стек. В этой ситуации особенно ярко проявляются характерные черты JavaScript. Благодаря замыканию несложно сохранить контекст вызывающей асинхронной функции, даже если функция обратного вызова вызывается в другой момент времени и из другого места.

Синхронные функции блокируют выполнение до своего завершения. Асинхронная функция возвращает управление немедленно, а ее результат, позднее в цикле событий, передается в обработчик (в данном случае это функция обратного вызова).

Обратные вызовы, не связанные со стилем передачи продолжений

Имеется несколько обстоятельств, которые при наличии аргумента обратного вызова могут заставить решить, что функция является асинхронной или использует стиль передачи продолжений, но это не всегда верно. Возьмем, например, метод `map()` объекта `Array`:

```
const result = [1, 5, 7].map(element => element - 1);
console.log(result); // [0, 4, 6]
```

Очевидно, что обратный вызов здесь используется только для обхода элементов массива, а не для передачи результата операции. В самом деле, результат возвращается синхронно, с использованием прямого стиля. Назначение обратного вызова обычно четко указано в документации программного интерфейса.

Синхронный или асинхронный?

Как было продемонстрировано выше, порядок выполнения инструкций радикально меняется в зависимости от синхронного или асинхронного характера функции. Это сильно влияет на поток выполнения всего приложения как в отношении корректности, так и эффективности. Ниже приводятся анализ этих двух парадигм и их ловушки. В целом необходимо избегать создания непоследовательного и запутанного программного интерфейса, поскольку это может повлечь ряд проблем, которые очень сложно обнаружить и воспроизвести. Продолжая анализ, рассмотрим в качестве примера непредсказуемую асинхронную функцию.

Непредсказуемая функция

Одной из наиболее опасных ситуаций является наличие программного интерфейса, который в одних условиях работает синхронно, а в других – переходит в асинхронный режим. В качестве примера рассмотрим следующий код:

```
const fs = require('fs');
const cache = {};
function inconsistentRead(filename, callback) {
  if(cache[filename]) {
    //вызывается синхронно
    callback(cache[filename]);
  } else {
    //асинхронная функция
    fs.readFile(filename, 'utf8', (err, data) => {
      cache[filename] = data;
```

```
      callback(data);
    });
  }
}
```

Приведенная выше функция использует переменную `cache` для хранения результата операции чтения файла. Имейте в виду, что это просто пример, в нем отсутствует обработка ошибок, а сама логика кэширования не является оптимальной. Кроме того, эта функция таит в себе опасность, поскольку ведет себя асинхронно, если кэш не содержит данных, которые должны быть получены в результате выполнения функции `fs.readFile()`, но будет выполняться синхронно для всех последующих обращений к файлу, уже имеющемуся в кэше, немедленного вызывая функцию обратного вызова.

Высвобождение Залго

А теперь посмотрим, как использование непредсказуемых функций, таких как выше, легко сможет поломать приложение. Рассмотрим следующий код:

```
function createFileReader(filename) {
  const listeners = [];
  inconsistentRead(filename, value => {
    listeners.forEach(listener => listener(value));
  });

  return {
    onDataReady: listener => listeners.push(listener)
  };
}
```

При вызове приведенной выше функции создается новый объект, выступающий в качестве обработчика, что позволяет задать несколько обработчиков для операции чтения файла. Все обработчики будут вызваны сразу после завершения операции чтения, как только данные станут доступны. Эта функция использует функцию `inconsistentRead()` для реализации своей функциональности. А теперь попробуем использовать функцию `createFileReader()`:

```
const reader1 = createFileReader('data.txt');
reader1.onDataReady(data => {
  console.log('First call data: ' + data);

  //...чуть позже попытаемся вновь прочитать данные из
  //того же файла
  const reader2 = createFileReader('data.txt');
  reader2.onDataReady( data => {
    console.log('Second call data: ' + data);
  });
});
```

Предыдущий код выведет следующее:

```
First call data: some data
```

Как видите, обратный вызов во второй операции так и не был выполнен. Давайте выясним, почему так получилось.

- Во время создания `reader1` функция `inconsistentRead()` выполняется асинхронно, поскольку в кэше еще нет результата. Таким образом, имеется возможность

зарегистрировать обработчика, что и будет сделано в другой итерации цикла событий, до завершения операции чтения.

- Затем в итерации цикла событий создается `reader2`, когда файл уже находится в кэше. В этом случае внутренний вызов `inconsistentRead()` будет выполнен синхронно. То есть он выполнит функцию обратного вызова немедленно, и, соответственно, все обработчики события `reader2` также будут вызваны синхронно. Однако регистрация обработчиков выполняется после создания `reader2`, поэтому они никогда не будут вызваны.

Поведение обратного вызова в функции `inconsistentRead()` действительно непредсказуемо, поскольку зависит от многих факторов, таких как частота вызова, переданное в аргументе имя файла и время, необходимое для загрузки файла.

Продемонстрированную здесь ошибку чрезвычайно сложно выявить и воспроизвести в условиях реального приложения. Представьте себе использование аналогичной функции на веб-сервере, где одновременно может обрабатываться несколько запросов – некоторые из этих запросов могут зависнуть без видимых причин и регистрации каких-либо ошибок. Такое поведение явно относится к категории весьма неприятных дефектов.

Исаак З. Шлуетер (Isaac Z. Schlueter), создатель проекта npm и бывший руководитель проекта Node.js, в одной из статей в своем блоге сравнил применение подобных непредсказуемых функций с выпуском на волю Залго.

Залго (Zalgo) – это интернет-легенда о зловещей сущности, считающейся причиной безумия, смерти и разрушения мира. Если вы ничего не знаете о Залго, рекомендуем познакомиться с ним.

Оригинальное сообщение Исаака З. Шлуетера можно найти на странице http://blog.izs.me/post/591427421 43/designing-apis-for-asynchrony.

Использование синхронного программного интерфейса

Пример высвобождения Залго учит, что необходимо четко определять характер программного интерфейса: синхронный или асинхронный.

Один из способов исправления функции `inconsistentRead()` заключается в том, чтобы сделать ее полностью синхронной. Это возможно, поскольку платформа Node.js поддерживает ряд программных интерфейсов для основных операций ввода/вывода, соответствующих прямому синхронному стилю. Например, можно использовать функцию `fs.readFileSync()` вместо ее асинхронного эквивалента:

```
const fs = require('fs');
const cache = {};
function consistentReadSync(filename) {
  if(cache[filename]) {
    return cache[filename];
  } else {
    cache[filename] = fs.readFileSync(filename, 'utf8');
    return cache[filename];
  }
}
```

Как видите, здесь вся функция приведена в соответствие с прямым стилем. Для синхронной функции нет никакого смысла использовать стиль передачи продолжений. В самом деле, можно утверждать, что при реализации синхронного программного интерфейса всегда следует применять только прямой стиль. Это позволит избе-

жать путаницы, связанной со стилями, и достичь наилучшей эффективности с точки зрения производительности.

Шаблон
Используйте прямой стиль при создании синхронных функций.

Имейте в виду, что переход от стиля CPS к прямому стилю, от асинхронного режима выполнения к синхронному и наоборот может потребовать изменения стиля всего кода. Например, в данном случае придется полностью изменить интерфейс функции `createFileReader()` и адаптировать его для синхронной работы.

Кроме того, использование синхронного программного интерфейса вместо асинхронного связано со следующими проблемами:

- синхронный интерфейс можно использовать не всегда;
- синхронный интерфейс будет блокировать цикл событий и отложенные параллельные запросы. Он ломает модель параллельной обработки в JavaScript, приостанавливая работу всего приложения. Далее в книге будет продемонстрировано его влияние на приложения.

Риск блокирования цикла событий в функции `consistentReadSync()` невелик, поскольку синхронный интерфейс ввода/вывода вызывается только один раз для любого файла, а во всех последующих вызовах будет использоваться содержимое кэша. Если имеется лишь ограниченное число статических файлов, функция `consistentReadSync()` не будет оказывать особого влияния на работу цикла событий. Но все сразу изменится при одновременном чтении множества файлов. Использование синхронного ввода/вывода на платформе Node.js не рекомендовано во многих ситуациях. Однако в отдельных случаях оно является наиболее простым и эффективным решением. Всегда следует оценивать конкретные условия использования для выбора верного варианта реализации. Одним из наглядных примеров является разработка синхронного блокирующего интерфейса для чтения конфигурационных файлов при начальной загрузке приложения.

Блокирующий интерфейс следует использовать, только когда он не влияет на способность приложения обслуживать параллельные запросы.

Отложенное выполнение

Другой альтернативный способ исправления функции `inconsistentRead()` заключается в том, чтобы сделать ее чисто асинхронной. Хитрость в том, чтобы выполнить синхронный обратный вызов «в будущем», а не немедленно в том же цикле событий. На платформе Node.js это можно осуществить с помощью метода `process.nextTick()`, откладывающего выполнение функции до следующей итерации цикла событий. Механизм его работы очень прост, этот метод принимает обратный вызов в качестве аргумента и добавляет его в начало очереди событий, до всех обработчиков, ожидающих события ввода/вывода, и немедленно возвращает управление. Обратный вызов будет выполнен в следующей же итерации цикла событий.

Применим этот способ для исправления функции `inconsistentRead()`:

```
const fs = require('fs');
const cache = {};
function consistentReadAsync(filename, callback) {
  if(cache[filename]) {
    process.nextTick(() => callback(cache[filename]));
```

```
  } else {
    //асинхронная функция
    fs.readFile(filename, 'utf8', (err, data) => {
      cache[filename] = data;
      callback(data);
    });
  }
}
```

Теперь данная функция гарантированно выполняет обратный вызов асинхронно, при любых обстоятельствах.

Еще одним программным интерфейсом отложенного выполнения является `setImmediate()`. Хотя цели этих интерфейсов схожи, они обладают различной семантикой. Обратные вызовы, отложенные с помощью метода `process.nextTick()`, выполняются до обработки любых событий ввода/вывода, тогда как `setImmediate()` помещает их в очередь за уже находящимися в очереди событиями ввода/вывода. Поскольку `process.nextTick()` обеспечивает обработку перед любой запланированной операцией ввода/вывода, это может привести в определенных обстоятельствах к задержкам, например при рекурсивных вызовах, что невозможно при использовании `setImmediate()`. Более подробно разницу между этими двумя программными интерфейсами мы рассмотрим, когда будем анализировать использование отложенных вызовов синхронного выполнения вычислительных задач.

Шаблон
Асинхронное выполнение обратного вызова гарантируется за счет использования метода `process.nextTick()`.

Соглашения Node.js об обратных вызовах

Программные интерфейсы в Node.js, использующие стиль передачи продолжений и обратные вызовы, подчиняются ряду конкретных соглашений. Эти соглашения применяются в программном интерфейсе ядра Node.js, но им также следует подавляющее большинство пользовательских модулей и приложений. Поэтому очень важно разобраться в них, чтобы гарантировать соответствие им любого создаваемого вами асинхронного программного интерфейса.

Обратные вызовы передаются последними

Все методы ядра платформы Node.js подчиняются стандартному соглашению: параметр с функцией обратного вызова должен быть последним в списке параметров. В качестве примера приведем следующий программный интерфейс ядра Node.js:

```
fs.readFile(filename, [options], callback)
```

Как демонстрирует сигнатура этой функции, обратный вызов всегда находится в последней позиции, даже при наличии необязательных аргументов. Это соглашение направлено на улучшение удобочитаемости функции при определении обратного вызова непосредственно в ней.

Ошибки передаются первыми

При использовании стиля CPS в обратный вызов вместе с результатом всегда передается ошибка. На платформе Node.js любая ошибка CPS-функции всегда передается обратному вызову в первом аргументе, а фактический результат – в аргументах на-

чиная со второго. Если операция завершилась успешно, то есть без ошибок, в первом аргументе передается `null` или `undefined`. Следующий код демонстрирует определение функции обратного вызова, соответствующее данному соглашению:

```
fs.readFile('foo.txt', 'utf8', (err, data) => {
  if(err)
    handleError(err);
  else
    processData(data);
});
```

Рекомендуется всегда проверять наличие ошибки, поскольку иначе затрудняется отладка кода и повышается риск появления сбоев. Другое важное соглашение, которое также следует принять во внимание, заключается в том, что ошибка всегда должна иметь тип `Error`. То есть обычные строки или числа никогда не должны передаваться в качестве объектов ошибок.

Распространение ошибок

Распространение ошибок при применении в функции прямого синхронного стиля реализуется с помощью оператора `throw`, который заставляет ошибку продвигаться вверх по стеку вызовов, пока не найдется ее обработчик.

При применении асинхронного CPS-стиля распространение ошибок реализуется путем простой передачи ошибки в следующий обратный вызов в цепочке, например:

```
const fs = require('fs');
function readJSON(filename, callback) {
  fs.readFile(filename, 'utf8', (err, data) => {
    let parsed;
    if(err)
      //распространение ошибки и выход из текущей функции
      return callback(err);

    try {
      //анализ содержимого файла
      parsed = JSON.parse(data);
    } catch(err) {
      //перехват обрабатываемых ошибок
      return callback(err);
    }
    //ошибок нет, передаются только данные
    callback(null, parsed);
  });
};
```

Обратите внимание, как в примере выше вызывается функция `callback` для передачи допустимого результата и для распространения ошибки. Отметьте также, что для распространения ошибки применяется оператор `return`. Это обеспечивает выход сразу после вызова функции `callback`, чтобы избежать выполнения следующей строки в функции `readJSON`.

Неперехваченные исключения

Как продемонстрировано в функции `readJSON()`, чтобы избежать передачи любых исключений в обратный вызов `fs.readFile()`, вызов метода `JSON.parse()` заключен

в блок try...catch. Исключение, перехваченное внутри асинхронного обратного вызова, будет передано циклу событий и никогда не достигнет следующего обратного вызова.

В противном случае на платформе Node.js ошибка станет неустранимой, и приложение просто завершится с выводом сообщения об ошибке в stderr. Для демонстрации удалим блок try... catch в коде функции readJSON():

```
const fs = require('fs');
function readJSONThrows(filename, callback) {
  fs.readFile(filename, 'utf8', (err, data) => {
    if(err) {
      return callback(err);
    }
    //нет ошибок, передаются только данные
    callback(null, JSON.parse(data));
  });
};
```

Эта новая функции лишена возможности перехватывать исключения, возникающие в методе JSON.parse(). Теперь попытаемся выполнить парсинг недопустимого файла JSON:

```
readJSONThrows('nonJSON.txt', err => console.log(err));
```

Это приведет к внезапному прекращению работы приложения с выводом в консоль сообщения о возникшем исключении:

```
SyntaxError: Unexpected token d
    at Object.parse (native)
    at [...]
    at fs.js:266:14
    at Object.oncomplete (fs.js:107:15)
```

Приведенная выше трассировка стека начинается где-то в модуле fs.js, а именно в месте, где встроенный программный интерфейс завершил чтение и вернул результат через цикл событий в функцию fs.readFile(). Это явно указывает, что исключение попало в стек из функции обратного вызова, а затем прямо в цикл событий, где оно было, наконец, перехвачено, и сообщение о нем выведено в консоль.

Это также означает, что обертывание вызова readJSONThrows() блоком try... catch не будет работать, поскольку стек, где работает блок, отличается от того, где будет вызвана функция обратного вызова. Следующий код демонстрирует описанный только что антишаблон:

```
try {
  readJSONThrows('nonJSON.txt', function(err, result) {
    //...
  });
} catch(err) {
  console.log('This will not catch the JSON parsing exception');
}
```

Оператор catch в коде выше никогда не перехватит ошибку парсинга JSON, поскольку она будет передана обратно в стек, где возникло исключение. Как только что

было продемонстрировано, этот стек заканчивается циклом событий, а не функцией, запустившей асинхронную операцию.

Как уже было упомянуто ранее, приложение прерывает выполнение, когда исключение достигает цикла событий. Однако все еще имеется шанс выполнить некоторые операции и зафиксировать факт ошибки, прежде чем приложение завершит работу. В самом деле, когда это происходит, платформа Node.js генерирует специальное событие `uncaughtException` перед завершением процесса. Следующий код демонстрирует его использование:

```
process.on('uncaughtException', (err) => {
  console.error('This will catch at last the ' +
  'JSON parsing exception: ' + err.message);
  // Прерывание приложения с возвратом 1 (ошибка)
  // в качестве кода завершения:
  // если опустить эту строку, работа приложения
  // будет продолжена
  process.exit(1);
});
```

Важно понимать, что продолжение работы приложения, находящегося в состоянии неперехваченного исключения, не гарантирует нормального его выполнения и может привести к непредвиденным проблемам. Например, незавершенность запущенных запросов ввода/вывода и несогласованность состояния замыканий могут привести к невозможности продолжить выполнение. Поэтому, особенно в условиях производственной эксплуатации, рекомендуется завершать приложение после появления необработанного исключения.

Система модулей и ее шаблоны

Модули являются не только средством структурирования нетривиальных приложений, но и основным механизмом сокрытия информации путем закрытия доступа ко всем функциям и переменным, которые явно не отмечены, как предназначенные для экспорта. Этот раздел посвящен модульной системе Node.js и наиболее распространенным шаблонам ее использования.

Шаблон Revealing Module

Одной из основных проблем языка JavaScript является отсутствие пространств имен. Программы, выполняемые в глобальной области видимости, захламляют ее своими внутренними данными и данными своих зависимостей. Популярным методом решения этой проблемы является шаблон Revealing Module (Открытый модуль), например:

```
const module = (() => {
  const privateFoo = () => {...};
  const privateBar = [];

  const exported = {
    publicFoo: () => {...},
    publicBar: () => {...}
  };
```

```
  return exported;
})();
console.log(module);
```

Этот шаблон использует возможность автоматического вызова функции для создания ограниченной области видимости и экспортирования только тех элементов, которые должны быть общедоступными. В примере выше переменная `module` содержит только экспортируемый программный интерфейс, в то время как остальное содержимое модуля недоступно извне. Ниже будет показано, что этот шаблон используется в качестве основы модульной системы платформы Node.js.

Пояснения относительно модулей Node.js

Для стандартизации экосистемы JavaScript была создана группа CommonJS. Одно из ее самых популярных предложений носит название **CommonJS modules**. Модульная система платформы Node.js разработана на основе этой спецификации с добавлением нескольких нестандартных расширений. При описании ее работы используем аналогию с шаблоном открытого модуля, когда каждый модуль запускается в собственной области видимости – все переменные определяются локально и не загрязняют глобального пространства имен.

Самодельный загрузчик модулей

Для пояснения работы системы модулей реализуем аналогичную систему с нуля. Следующий код определяет функцию, имитирующую подмножество функциональных возможностей оригинальной функции `require()` платформы Node.js.

Начнем с создания функции, которая загрузит содержимое модуля, заключит в собственную область видимости и выполнит его:

```
function loadModule(filename, module, require) {
  const wrappedSrc=`(function(module, exports, require) {
      ${fs.readFileSync(filename, 'utf8')}
    })(module, module.exports, require);`;
  eval(wrappedSrc);
}
```

Исходный код модуля, по сути, заключен в функцию, подобно тому, как это делалось в шаблоне Revealing Module (Открытый модуль). Разница заключается в передаче модулю списка переменных, в частности переменных `module`, `exports` и `require`. Просто отметим, что аргумент `exports` обертывающей функции инициализируется содержимым `module.exports`, о чем пойдет речь ниже.

Имейте в виду, что это только пример, и в реальном приложении редко возникает необходимость в выполнении исходного кода. Такие функции, как `eval()`, или функции модуля `vm` (http://nodejs.org/api/vm.html) достаточно легко могут использоваться непредвиденным способом или с непредвиденными входными данными, что делает систему уязвимой для атак посредством внедрения кода. Их всегда следует использовать с крайней осторожностью или вообще отказаться от их применения.

А теперь посмотрим, что хранят эти переменные, реализовав собственную функцию `require()`:

```
const require = (moduleName) => {
  console.log(`Require invoked for module: ${moduleName}`);
  const id = require.resolve(moduleName);    //[1]
  if(require.cache[id]) {                    //[2]
    return require.cache[id].exports;
  }

  //метаданные модуля
  const module = {                           //[3]
    exports: {}, id: id
  };
  //Пополнить кэш
  require.cache[id] = module;                //[4]

  //загрузить модуль
  loadModule(id, module, require);           //[5]

  //вернуть экспортируемые переменные
  return module.exports;                     //[6]
};
require.cache = {};
require.resolve = (moduleName) => {
  /* извлечение полного идентификатора модуля по имени модуля */
};
```

Приведенная выше функция имитирует поведение оригинальной функции `require()` платформы Node.js, используемой для загрузки модулей. Конечно, она предназначена исключительно для учебных целей, поскольку не полностью и не точно отражает внутреннее поведение реальной функции `require()`, но она дает отличную возможность понять внутреннее устройство системы модулей Node.js, порядок определения и загрузки модулей. Ниже описывается, как работает наша самодельная система модулей.

1. На входе принимается имя модуля и первым действием определяется полный путь к модулю, обозначенный здесь как `id`. Эта задача возлагается на метод `require.resolve()`, реализующий конкретный алгоритм поиска (он будет рассмотрен ниже).
2. Если модуль был загружен ранее, он должен быть доступен через кэш и в этом случае возвращается немедленно.
3. Если модуль еще не был загружен, создается среда для первой загрузки. В частности, создается объект `module` со свойством `exports`, инициализированным пустым литералом объекта. Это свойство будет использовано кодом модуля для экспорта общедоступного интерфейса.
4. Объект `module` сохраняется в кэше.
5. Исходный код модуля читается из файла и выполняется, как уже было показано выше. Модуль связывается с вновь созданным объектом `module` и ссылкой на функцию `require()`. Модуль экспортирует свой общедоступный интерфейс, изменяя или подменяя объект `module.exports`.
6. И наконец, содержимое `module.exports` – общедоступный интерфейс модуля – возвращается вызывающей программе.

Как видите, в работе системы модулей платформы Node.js нет ничего загадочного. Вся хитрость заключается в обертке, создаваемой вокруг исходного кода модуля, и искусственной среде, в которой он запускается.

Определение модуля

Знакомство с работой пользовательской функции `require()` позволяет понять порядок определения модуля. Следующий код послужит примером:

```
//загрузка зависимости
const dependency = require('./anotherModule');

//закрытая функция
function log() {
  console.log(`Well done ${dependency.username}`);
}

//экспортируемый общедоступный интерфейс
module.exports.run = () => {
  log();
};
```

Основная идея заключается в том, что все компоненты модуля являются закрытыми, если не присвоены явно переменной `module.exports`. Содержимое этой переменной кэшируется и возвращается при загрузке модуля с помощью `require()`.

Глобальное определение

Даже если все переменные и функции, объявленные в модуле, определены с локальной областью видимости, все равно остается возможность определить глобальную переменную. Система модулей в действительности экспортирует специальную переменную `global`, которой можно воспользоваться для этой цели. Все, что присвоено этой переменной, будет в конечном итоге автоматически размещено в глобальной области видимости.

Размещение в глобальной области считается плохой практикой и сводит на нет все преимущества системы модулей. Поэтому данную возможность следует использовать, только когда это действительно необходимо.

module.exports и exports

Многие начинающие разработчики на платформе Node.js не понимают разницы между использованием `exports` и `module.exports` для экспортирования общедоступного интерфейса. Чтобы прояснить этот вопрос, воспользуемся кодом пользовательской функции `require`. Переменная `exports` является лишь ссылкой на начальное значение `module.exports`. Как упоминалось выше, это значение, по существу, является простым литералом объекта, созданным до загрузки модуля.

Это означает, что можно лишь добавлять новые свойства в объект, на который ссылается переменная `exports`, как показано ниже:

```
exports.hello = () => {
  console.log('Hello');
}
```

Присваивание другого значения переменной `exports` лишено смысла, поскольку не меняет содержимого `module.exports`, оно приведет только к изменению значения самой переменной. Поэтому следующий код неправилен:

```
exports = () => {
  console.log('Hello');
}
```

Если необходимо экспортировать что-то, отличное от объекта литерала, например функцию, экземпляр или даже строку, следует выполнить присваивание `module.exports`:

```
module.exports = () => {
  console.log('Hello');
}
```

Функция require является синхронной

Еще одна важная деталь, которую следует учитывать, – самодельная функция `require` является синхронной. В самом деле, она возвращает содержимое модуля, используя простой прямой стиль, и обратный вызов при этом не нужен. То же верно для оригинальной функции `require()` платформы Node.js. Следовательно, любое присваивание переменной `module.exports` также должно быть синхронным. Например, следующий код является неправильным:

```
setTimeout(() => {
  module.exports = function() {...};
}, 100);
```

Это свойство оказывает существенное влияние на определение модулей, поскольку заставляет ограничиваться только синхронным кодом. Это одна из наиболее важных причин, почему библиотеки ядра платформы Node.js предлагают синхронные альтернативы для большинства имеющихся в них асинхронных интерфейсов.

Если при инициализации модуля потребуется выполнить асинхронные операции, следует определить и экспортировать неинициализированный модуль, который будет асинхронно инициализирован позже. Проблема этого подхода – в том, что загрузка такого модуля с помощью функции `require` не гарантирует его готовности к использованию. В *главе 9 «Дополнительные рецепты асинхронной обработки»* эта проблема будет подробно рассмотрена и предложены шаблоны ее решения.

Для общего развития можно отметить, что изначально платформа Node.js применяла асинхронную версию `require()`, но вскоре она была исключена, поскольку усложняла функциональность, предназначенную для использования только при инициализации, где асинхронные операции ввода/вывода приносят больше сложностей, чем преимуществ.

Алгоритм разрешения

Термином *«ад зависимостей»* обозначают ситуацию, когда разные зависимости, в свою очередь, зависят от определенной общей зависимости, но требуют различных несовместимых ее версий. Платформа Node.js элегантно решает эту проблему путем загрузки нужной версии модуля в соответствии с тем, откуда загружается модуль. Все достоинства этой возможности обеспечиваются диспетчером npm и алгоритмом разрешения в функции `require`.

Рассмотрим краткое описание этого алгоритма. Как упоминалось выше, функция `resolve()` принимает имя модуля (здесь носит название `moduleName`) и возвращает полный путь к модулю. Этот путь используется для загрузки кода, а также для идентификации модуля. Алгоритм разрешения можно разделить на три основные части.

- **Модули в файлах**: если имя в `moduleName` начинается с `/`, оно считается абсолютным путем к модулю и возвращается без изменений. Если имя начинается с `./`, оно считается относительным путем, откладываемым от загружающего модуля.

- **Модули ядра**: если значение `moduleName` не начинается с `/` или `./`, алгоритм начнет с попытки найти модуль среди модулей ядра платформы Node.js.
- **Модули в пакетах**: если в ядре не будет найдено модуля, соответствующего значению `moduleName`, поиск будет продолжен в первом каталоге `node_modules`, находящемся выше загружающего модуля. Алгоритм продолжает поиск совпадения в каталогах `node_modules` все выше и выше в дереве каталогов, пока не достигнет корневого каталога файловой системы.

В случае с модулями в файлах и пакетах значению `moduleName` могут соответствовать отдельные файлы и каталоги. В частности, алгоритм попытается найти следующие соответствия:

- `<moduleName>.js`;
- `<moduleName>/index.js`;
- каталог/файл, указанный в свойстве `main` файла `<moduleName>/package.json`.

Полное официальное описание алгоритма можно найти на странице http://nodejs.org/api/modules.html#modules_all_together.

Каталог `node_modules` является тем местом, куда npm устанавливает зависимости любого пакета. Это значит, что при использовании описанного алгоритма каждый пакет может иметь собственные зависимости. Например, рассмотрим следующую структуру каталогов:

```
myApp
├── foo.js
└── node_modules
    ├── depA
    │   └── index.js
    ├── depB
    │   ├── bar.js
    │   └── node_modules
    │   └── depA
    │       └── index.js
    └── depC
        ├── foobar.js
        └── node_modules
            └── depA
                └── index.js
```

В предыдущем примере `myApp`, `depB` и `depC` зависят от `depA`, но все они имеют собственные версии зависимостей! Согласно правилам алгоритма разрешения, вызов `require('depA')` загрузит разные файлы в разных модулях, например:

- при вызове `require('depA')` из `/myApp/foo.js` будет загружен файл `/myApp/node_modules/depA/index.js`;
- при вызове `require('depA')` из `/myApp/node_modules/depB/bar.js` будет загружен файл `/myApp/node_modules/depB/node_modules/depA/index.js`;
- при вызове `require('depA')` из `/myApp/node_modules/depC/foobar.js` будет загружен файл `/myApp/node_modules/depC/node_modules/depA/index.js`.

Алгоритм разрешения в ядре платформы Node.js обеспечивает надежное управление зависимостями и позволяет иметь в приложении сотни и даже тысячи пакетов и избежать конфликтов и проблем, связанных с совместимостью версий.

Алгоритм разрешения применяется неявно при вызове функции `require()`, но при необходимости его можно явно использовать в любом модуле, вызывая `require.resolve()`.

Кэширование модулей

Каждый модуль необходимо загрузить и выполнить только один раз, поскольку любой последующий вызов `require()` возвращает его кэшированную версию. Это демонстрирует код самодельной функции `require`. Кэширование имеет решающее значение для производительности, но оно также определяет следующие важные следствия:

- позволяет иметь циклические зависимости модулей;
- гарантирует возврат одного и того же экземпляра при загрузке одного и того же модуля из заданного пакета.

Кэш модуля доступен через переменную `require.cache`, что позволяет обратиться к нему напрямую. Широко распространенный способ удаления модуля из кэша путем удаления соответствующего ключа из переменной `require.cache` удобен во время тестирования, но опасен в условиях реальной эксплуатации.

Циклические зависимости

Многие рассматривают циклические зависимости как внутренний недостаток проекта, но они могут присутствовать в любом реальном проекте, так что имеет смысл, по крайней мере, разобраться, как они обрабатываются на платформе Node.js. Анализируя самодельную функцию `require()`, можно получить представление, как они работают и какие опасности таят.

Предположим, имеются два модуля:

- модуль `a.js`:

```
exports.loaded = false;
const b = require('./b');
module.exports = {
  bWasLoaded: b.loaded,
  loaded: true
};
```

- модуль `b.js`:

```
exports.loaded = false;
const a = require('./a');
module.exports = {
  aWasLoaded: a.loaded,
  loaded: true
};
```

А теперь попробуем загрузить их в модуле `main.js`:

```
const a = require('./a');
const b = require('./b');
console.log(a);
console.log(b);
```

Приведенный выше код выведет следующее:

```
{ bWasLoaded: true, loaded: true }
{ aWasLoaded: false, loaded: true }
```

Такой результат демонстрирует опасность циклических зависимостей. Оба модуля полностью инициализированы на момент загрузки из основного модуля, модуль `a.js` будет неполон при его загрузке в `b.js`. В частности, его состояние будет таким, какое было достигнуто на момент его загрузки в `b.js`. То же самое, но наоборот, произойдет при изменении порядка загрузки модулей в `main.js`. Теперь уже модуль `a.js` получит

неполную версию модуля `b.js`. Ясно, что положение становится очень шатким при потере контроля над порядком загрузки модулей, что весьма вероятно в достаточно большом проекте.

Шаблоны определения модулей

Система модулей является не только механизмом загрузки зависимостей, но и инструментом определения программных интерфейсов. Во всех задачах проектирования интерфейсов основным фактором является баланс между закрытой и общедоступной функциональностью. Цель заключается в достижении наибольшего удобства использования программного интерфейса при максимальном сокрытии информации, с учетом других качеств программного обеспечения, таких как *расширяемость* и *повторное использование*.

В этом разделе рассматриваются несколько наиболее популярных шаблонов определения модулей в Node.js, каждый из которых характеризуется своим индивидуальным балансом сокрытия информации, расширяемости и повторного использования.

Именованный экспорт

Самым простым способом экспортирования общедоступного интерфейса является использование **именованного экспорта**, когда все значения, которые должны быть общедоступными, присваиваются свойствам объекта в переменной `exports` (или в переменной `module.exports`). В этом случае экспортируемый объект становится контейнером, или пространством имен для набора связанных функций.

Следующий модуль реализует этот шаблон:

```
//файл logger.js
exports.info = (message) => {
  console.log('info: ' + message);
};

exports.verbose = (message) => {
  console.log('verbose: ' + message);
};
```

Экспортируемые функции будут доступны как свойства загруженного модуля, как демонстрирует следующий код:

```
//файл main.js
const logger = require('./logger');
logger.info('This is an informational message');
logger.verbose('This is a verbose message');
```

Большинство модулей в ядре Node.js использует именно этот шаблон.

Спецификация CommonJS позволяет применять только переменную `exports` для экспортирования общедоступных элементов. Поэтому шаблон именованного экспорта является единственным, совместимым со спецификацией CommonJS. Использование `module.exports` представляет собой расширение этой спецификации платформой Node.js для поддержки более широкого диапазона шаблонов определения модулей, рассматриваемых ниже.

Экспорт функций

Один из самых популярных шаблонов определения модулей заключается в присваивании переменной `module.exports` единственной функции. Его главным преимуще-

ством является экспортирование единственной функциональной возможности с явной точкой входа в модуль, что упрощает его понимание и использование. Это также соответствует принципу *малой площади* общедоступной области. Этот подход к определению модулей также известен в сообществе как **шаблон Substack**, названный так в честь одного из самых активных его адептов Джеймса Халлидея (известного еще под псевдонимом substack). Рассмотрим данный шаблон на следующем примере:

```
//файл logger.js
module.exports = (message) => {
  console.log(`info: ${message}`);
};
```

Одно из возможных расширений этого шаблона – использование экспортируемой функции как пространства имен для других общедоступных интерфейсов. Это очень мощный механизм, поскольку он все еще обеспечивает явную и единственную точку входа в модуль (экспортируемую функцию). Кроме того, такой подход дает возможность экспортировать другие функциональные возможности, являющиеся дополнениями и расширениями. Следующий код показывает, как расширить этот модуль, экспортировав функцию как пространство имен:

```
module.exports.verbose = (message) => {
  console.log(`verbose: ${message}`);
};
```

А этот код демонстрирует использование модуля, определенного выше:

```
//файл main.js
const logger = require('./logger');
logger('This is an informational message');
logger.verbose('This is a verbose message');
```

Несмотря на то что простое экспортирование функции, на первый взгляд, может показаться ограничением, на самом деле это идеальный способ сосредоточиться на одной, наиболее важной особенности модуля, скрывая вторичные и внутренние особенности, доступные как свойства самой экспортируемой функции. Модульность платформы Node.js активно поощряет следование **принципу единственной ответственности** (Single Responsibility Principle, SRP): каждый модуль должен отвечать только за одну функциональную возможность, и эта ответственность должна быть полностью реализована этим модулем.

Шаблон (substack)
Экспортируйте основные функциональные возможности модуля в виде единственной функции. Экспортированная функция используется как пространство имен для прочих вспомогательных возможностей.

Экспорт конструктора

Модуль, экспортирующий конструктор, является частным случаем модуля, экспортирующего функцию. Разница лишь в том, что этот новый шаблон позволяет пользователю создавать новые экземпляры с помощью конструктора, а также расширять прототип модуля и создавать новые классы. Ниже приводится пример реализации этого шаблона:

```
//файл logger.js
function Logger(name) {
```

```
  this.name = name;
}

Logger.prototype.log = function(message) {
  console.log(`[${this.name}] ${message}`);
};

Logger.prototype.info = function(message) {
  this.log(`info: ${message}`);
};

Logger.prototype.verbose = function(message) {
  this.log(`verbose: ${message}`);
};

module.exports = Logger;
```

А вот как можно использовать этот модуль:

```
//файл main.js
const Logger = require('./logger');
const dbLogger = new Logger('DB');
dbLogger.info('This is an informational message');
const accessLogger = new Logger('ACCESS');
accessLogger.verbose('This is a verbose message');
```

Точно так же можно экспортировать класс ES2015:

```
class Logger {
  constructor(name) {
    this.name = name;
  }

  log(message) {
    console.log(`[${this.name}] ${message}`);
  }

  info(message) {
    this.log(`info: ${message}`);
  }

  verbose(message) {
    this.log(`verbose: ${message}`);
  }
}

module.exports = Logger;
```

Учитывая, что классы в ES2015 являются лишь синтаксическим сахаром представления прототипов, этот модуль используется точно так же, как альтернативная версия, основанная на прототипах.

Экспорт конструктора или класса тоже обеспечивает единственную точку входа в модуль, но, по сравнению с substack-структурой, экспортирует больше внутренних элементов модуля. С другой стороны, он обеспечивает более широкие возможности расширения.

Один из вариантов этого шаблона заключается в применении защиты от вызовов без использования ключевого слова `new`. Этот несложный прием обеспечивает использование модуля в качестве *фабрики*. Посмотрим, как это работает:

```
function Logger(name) {
  if(!(this instanceof Logger)) {
    return new Logger(name);
  }
  this.name = name;
};
```

Здесь все просто: мы проверяем существование ссылки `this` и ее соответствие экземпляру `Logger`. Если одно из этих условий не выполняется, значит, функция `Logger()` была вызвана без ключевого слова `new`, тогда мы создаем новый экземпляр с помощью ключевого слова `new` и возвращаем вызывающему коду. Эта технология также позволяет использовать модуль в качестве фабрики:

```
//файл logger.js
const Logger = require('./logger');
const dbLogger = Logger('DB');
accessLogger.verbose('This is a verbose message');
```

Намного более ясный подход к реализации защиты обеспечивает синтаксис `new.target`, введенный в стандарте ES2015 и доступный начиная с версии 6 платформы Node.js. Этот синтаксис экспортирует свойство `new.target` – «метасвойство», доступное внутри любой функции, которому во время выполнения присваивается значение `true`, если функция вызвана с помощью ключевого слова `new`.

Перепишем фабрику `Logger` с использованием этого синтаксиса:

```
function Logger(name) {
  if(!new.target) {
    return new LoggerConstructor(name);
  }
  this.name = name;
}
```

Этот код полностью эквивалентен предыдущему, так что можно сказать, что синтаксис `new.target` является полезным синтаксическим сахаром ES2015, делающим код более простым и естественным.

Экспорт экземпляра

Механизм кэширования функции `require()` можно использовать для простого определения экземпляров с готовым состоянием, создаваемых в конструкторе или фабрике, которые могут совместно использоваться различными модулями. Следующий код демонстрирует пример реализации этого шаблона:

```
//файл logger.js
function Logger(name) {
  this.count = 0;
  this.name = name;
}
Logger.prototype.log = function(message) {
  this.count++;
  console.log('[' + this.name + '] ' + message);
};
module.exports = new Logger('DEFAULT');
```

Этот новый модуль можно использовать так:

```
//файл main.js
const logger = require('./logger');
logger.log('This is an informational message');
```

Поскольку модуль кэшируется, любой другой модуль, импортирующий модуль `logger`, получит один и тот же экземпляр объекта с одним и тем же состоянием. Этот шаблон похож на создание **объекта-одиночки** (singleton), но он не гарантирует уникальности экземпляра во всем приложении, как это делает традиционный шаблон Singleton (Одиночка).

Анализируя алгоритм разрешения, мы видели, что модуль может встречаться несколько раз в дереве зависимостей приложения. Это приводит к существованию нескольких экземпляров одного и того же логического модуля, запущенных в контексте одного и того же приложения на платформе Node.js. В *главе 7 «Связывание модулей»* будут проанализированы последствия экспорта готовых экземпляров и некоторые альтернативные шаблоны.

Расширение только что описанного шаблона включает экспорт конструктора, используемого для создания экземпляра, вместе с самим экземпляром. Это дает пользователям возможность создавать новые экземпляры объекта или даже расширять его при необходимости. Для этого необходимо присвоить экземпляр новому свойству, как это сделано в следующей строке кода:

```
module.exports.Logger = Logger;
```

После этого экспортированный конструктор можно использовать для создания других экземпляров класса:

```
const customLogger = new logger.Logger('CUSTOM');
customLogger.log('This is an informational message');
```

С точки зрения удобства это похоже на использование экспортируемой функции в качестве пространства имен. Модуль экспортирует по умолчанию экземпляр объекта в качестве той части функциональности, которую требуется использовать большую часть времени, в то время как более продвинутые функции, позволяющие создавать новые экземпляры или расширять объекты, будут доступны как второстепенные экспортируемые свойства.

Изменение других модулей или глобального пространства

Модуль может ничего не экспортировать. Это кажется бессмысленным, но не стоит забывать, что модуль может изменять глобальное пространство и любые объекты, включая другие кэшированные модули. Обратите внимание, что в целом это считается плохой методикой, но поскольку при определенных обстоятельствах (например, при тестировании) этот шаблон может быть полезным и безопасным, он время от времени используется, поэтому с ним стоит познакомиться. Было заявлено, что модули способны изменять другие модули или объекты в глобальной области, но это всего лишь то, что называется **партизанским латанием** (monkey patching). Под этим термином понимается изменение существующих объектов или расширение их возможностей во время выполнения.

В следующем примере показано, как добавить новую функцию в другой модуль:

```
//файл patcher.js

// ./logger является другим модулем
require('./logger').customMessage = () => console.log('This is a new
  functionality');
```

Новый модуль `patcher` используется просто:

```
//файл main.js

require('./patcher');
const logger = require('./logger');
logger.customMessage();
```

В предыдущем примере модуль `patcher` должен быть импортирован перед первым использованием модуля `logger`, чтобы применить к нему изменения.

Описанная здесь технология опасна. Основная ее проблема заключается в том, что изменение глобального пространства имен или других модулей связано с побочными эффектами. Другими словами, оно влияет на состояние сущностей, что может привести к не всегда предсказуемым последствиям, особенно когда несколько модулей взаимодействует с одними и теми же сущностями. Предположим, что два разных модуля меняют значение одной и той же глобальной переменной или одно и то же свойство одного и того же модуля. Последствия этого непредсказуемы (какому из модулей это удалось?), но самое главное заключается в том, что это повлияет на работу всего приложения.

Шаблон Observer

Другим важным шаблоном, широко применяемым на платформе Node.js, является шаблон Observer (Наблюдатель). Вместе с шаблонами Reactor (Реактор), Callback (Обратный вызов) и Module (Модуль) шаблон Observer является одним из столпов платформы и абсолютно необходимым условием использования многих не входящих в ядро и пользовательских модулей.

Шаблон Observer является идеальным решением для моделирования реактивной природы платформы Node.js и прекрасным дополнением к обратным вызовам. Приведем его официальное определение:

> ***Шаблон (Observer)*** *определяет объект (называемый субъектом), способный уведомить ряд наблюдателей (или обработчиков) об изменении своего состояния.*

Основным его отличием от шаблона обратного вызова является способность субъекта уведомить нескольких наблюдателей, в то время как традиционный обратный вызов передачи продолжения возвращает результат только одному обработчику – функции обратного вызова.

Класс EventEmitter

В традиционном объектно-ориентированном программировании шаблон Observer требует создания интерфейсов, конкретных классов и иерархии. На платформе Node.js все это осуществляется гораздо проще. Шаблон Observer изначально встроен в ядро и доступен в виде класса `EventEmitter`. Класс `EventEmitter` позволяет зарегистрировать в качестве обработчиков одну или несколько функций, вызываемых при появлении события конкретного типа. Рисунок 2.2 визуально поясняет эту идею:

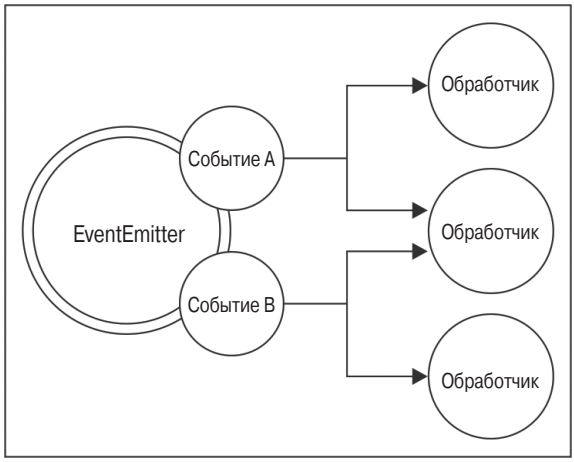

Рис. 2.2 ❖ Реализация шаблона Observer в виде класса EventEmitter

Класс `EventEmitter` является прототипом и экспортируется модулем `events`, входящим в ядро. Следующий код демонстрирует получение ссылки на этот класс:

```
const EventEmitter = require('events').EventEmitter;
const eeInstance = new EventEmitter();
```

Ниже перечислены основные методы класса `EventEmitter`:

- `on(event, listener)`: позволяет зарегистрировать новый обработчик (функцию) для заданного типа событий (строка);
- `once(event, listener)`: регистрирует новый обработчик, который будет удален после первого же события;
- `emit(event, [arg1], [...])`: создает новое событие и определяет дополнительные аргументы, передаваемые обработчикам;
- `removeListener(event, listener)`: удаляет обработчика событий определенного типа.

Все приведенные методы возвращают экземпляр класса `EventEmitter`, что позволяет составлять из них цепочки. Функция-обработчик имеет сигнатуру `function([arg1], [...])`, поэтому у нее не возникает проблем с получением аргументов, поставляемых вместе с событием. Внутри обработчика ссылка `this` указывает на экземпляр `EventEmitter`, пославший событие.

Как видите, на платформе Node.js имеется большая разница между реализацией обработчиков и традиционных функций обратного вызова, в частности первый аргумент не является ошибкой, это могут быть любые данные, передаваемые функцией `emit()` в момент ее вызова.

Создание и использование класса EventEmitter

Рассмотрим практическое использование класса `EventEmitter`. Самый простой путь: создать новый экземпляр и немедленно его использовать. Следующий код демонстрирует функцию, которая использует класс `EventEmitter` для уведомления своих або-

нентов в режиме реального времени о нахождении совпадения с определенным шаблоном в содержимом файлов из заданного списка:

```
const EventEmitter = require('events').EventEmitter;
const fs = require('fs');

function findPattern(files, regex) {
  const emitter = new EventEmitter();
  files.forEach(function(file) {
    fs.readFile(file, 'utf8', (err, content) => {
      if(err)
        return emitter.emit('error', err);

      emitter.emit('fileread', file);
      let match;
      if(match = content.match(regex))
        match.forEach(elem => emitter.emit('found', file, elem));
    });
  });
  return emitter;
}
```

Экземпляр класса `EventEmitter`, созданный в функции выше, генерирует три события:

- `fileread`: генерируется после чтения файла;
- `found`: генерируется при обнаружении совпадения;
- `error`: генерируется в случае ошибки чтения файла.

Рассмотрим пример использования функции `findPattern()`:

```
findPattern(
    ['fileA.txt', 'fileB.json'],
    /hello \w+/g
  )
  .on('fileread', file => console.log(file + ' was read'))
  .on('found', (file, match) => console.log('Matched "' + match +
    '" in file ' + file))
  .on('error', err => console.log('Error emitted: ' + err.message));
```

В примере выше осуществляется регистрация обработчиков для каждого из трех типов событий, генерируемых экземпляром `EventEmitter`, созданным функцией `findPattern()`.

Распространение ошибок

Класс `EventEmitter`, подобно обратным вызовам, не возбуждает исключений при появлении ошибок, поскольку они будут потеряны в цикле событий, так как события генерируются асинхронно. Вместо этого соглашение предписывает генерировать специальное событие `error` и передавать объект `Error` в качестве аргумента. Именно это было реализовано в функции `findPattern()`.

Рекомендуется всегда регистрировать обработчика события `error`, так как платформа Node.js обрабатывает это событие специальным образом и автоматически возбудит исключение с последующим завершением программы, если обработчик этого события не будет найден.

Создание произвольного наблюдаемого объекта

Иногда создание нового наблюдаемого объекта непосредственно из класса `EventEmitter` оказывается недостаточным, поскольку порой нецелесообразно экспортировать функциональные возможности, выходящие за рамки простой генерации новых событий. Фактически чаще бывает желательно создать общий наблюдаемый объект, что можно сделать, расширив класс `EventEmitter`.

Для демонстрации этого шаблона попробуем реализовать функцию `findPattern()` в объекте следующим образом:

```
const EventEmitter = require('events').EventEmitter;
const fs = require('fs');

class FindPattern extends EventEmitter {
  constructor (regex) {
    super();
    this.regex = regex;
    this.files = [];
  }

  addFile (file) {
    this.files.push(file);
    return this;
  }

  find () {
    this.files.forEach( file => {
      fs.readFile(file, 'utf8', (err, content) => {
        if (err) {
          return this.emit('error', err);
        }
        this.emit('fileread', file);
        let match = null;
        if (match = content.match(this.regex)) {
           match.forEach(elem => this.emit('found', file, elem));
        }
      });
    });
    return this;
  }
}
```

Здесь прототип `FindPattern` расширяет класс `EventEmitter` с помощью функции `inherits()` из модуля `util` ядра. В результате он становится полноценным классом наблюдаемого объекта. Ниже приводится пример его использования:

```
const findPatternObject = new FindPattern(/hello \w+/);
findPatternObject
  .addFile('fileA.txt')
  .addFile('fileB.json')
  .find()
  .on('found', (file, match) => console.log(`Matched "${match}"
    in file ${file}`))
  .on('error', err => console.log(`Error emitted ${err.message}`));
```

Как видите, объект `FindPattern` имеет тот же набор методов, являясь вместе с тем наблюдаемым объектом, что обеспечивается наследованием функциональности класса `EventEmitter`.

Этот шаблон достаточно широко применяется в экосистеме Node.js, например объект `Server` из модуля `HTTP` ядра определяет такие методы, как `listen()`, `close()`, `setTimeout()`, и внутренне наследует класс `EventEmitter`, что позволяет ему генерировать событие `request` при получении нового запроса, событие `connection` при создании нового соединения и событие `closed` при закрытии сервера.

Другими значимыми примерами расширения класса `EventEmitter` являются потоки данных Node.js. Мы подробно исследуем их в *главе 5* «*Программирование с применением потоков данных*».

Синхронные и асинхронные события

Подобно обратным вызовам, события могут генерироваться как синхронно, так и асинхронно. Крайне важно не смешивать этих двух подходов в одном и том же экземпляре класса `EventEmitter`, но еще важнее избегать проблем, описанных в разделе «*Высвобождение Залго*», при генерации событий одного типа.

Основное различие между генерацией синхронных и асинхронных событий заключается в способе регистрации обработчиков. Когда события создаются асинхронно, программа может зарегистрировать новые обработчики после инициализации `EventEmitter`, поскольку события гарантированно не будут возбуждены до следующего витка цикла событий. Именно это происходит в функции `findPattern()` – она иллюстрирует общепринятый подход, который используется в большинстве модулей платформы Node.js.

Напротив, синхронная генерация событий требует, чтобы все обработчики были зарегистрированы до того, как функция `EventEmitter` начнет посылать какие-либо события. Рассмотрим пример:

```
const EventEmitter = require('events').EventEmitter;
class SyncEmit extends EventEmitter {
  constructor() {
    super();
    this.emit('ready');
  }
}

const syncEmit = new SyncEmit();
syncEmit.on('ready', () => console.log('Object is ready to be used'));
```

Если бы событие `ready` было асинхронным, приведенный код работал бы просто отлично, но поскольку событие генерируется синхронно и обработчик регистрируется после отправки события, в результате обработчик никогда не будет вызван и ничего не выведет в консоль.

Существуют ситуации, когда по разным причинам имеет смысл использовать функцию `EventEmitter` синхронно. Поэтому очень важно четко отразить поведение функции `EventEmitter` в документации, чтобы избежать путаницы и неправильного ее использования.

Класс EventEmitter и обратные вызовы

Распространенной дилеммой при определении асинхронного программного интерфейса является необходимость выбора между классом `EventEmitter` и обратными вызовами. Часто выбор определяется семантикой: обратные вызовы следует использовать, когда результат должен возвращаться асинхронным способом, а события должны применяться, когда необходимо просто сообщить о том, что что-то произошло.

Но, помимо этого простого принципа, много путаницы вносит тот факт, что эти две парадигмы, по сути, эквиваленты и позволяют достичь одного и того же результата.

Рассмотрим в качестве примера следующий код:

```
function helloEvents() {
  const eventEmitter= new EventEmitter();
  setTimeout(() => eventEmitter.emit('hello', 'hello world'), 100);
  return eventEmitter;
}

function helloCallback(callback) {
  setTimeout(() => callback('hello world'), 100);
}
```

Две функции – `helloEvents()` и `helloCallback()` – можно считать эквивалентными с точки зрения результата. Первая сообщает об истечении заданного интервала времени с помощью события, а вторая использует для этой цели обратные вызовы, передавая тип события в качестве аргумента. Их отличие заключается только в разной удобочитаемости, семантике и объеме кода. Хотя не представляется возможным сформулировать детерминированный набор правил выбора одного из стилей, тем не менее здесь будет дано несколько подсказок, помогающих принять правильное решение.

Во-первых, можно сказать, что обратным вызовам присуще определенное ограничение, когда речь заходит о поддержке различных типов событий. Тем не менее имеется возможность отличать разные события, передавая их тип как аргумент функции обратного вызова или принимая в качестве аргументов несколько функций обратных вызовов, каждая из которых будет соответствовать одному типу событий. Однако это явно нельзя счесть элегантной реализацией программного интерфейса. В этой ситуации применение функции `EventEmitter` обеспечивает не только лучший интерфейс, но и более компактную реализацию.

Еще один случай, в котором класс `EventEmitter` выглядит предпочтительнее, – когда событие может возникнуть несколько раз или ни разу. Функции обратных вызовов подразумевают точно один вызов, независимо от успешности операции. Когда существует возможность повторения, с семантической точки зрения это больше похоже на событие, о котором нужно уведомить, а не на результат. Поэтому в данном случае применение `EventEmitter` предпочтительнее.

И наконец, интерфейс, использующий обратные вызовы, может осуществить уведомление только с помощью вызова конкретной функции, в то время как с помощью `EventEmitter` можно уведомить нескольких наблюдателей.

Комбинирование EventEmitter и обратных вызовов

Существуют ситуации, когда имеет смысл использовать `EventEmitter` вместе с обратными вызовами. Этот шаблон чрезвычайно полезен, когда требуется реализовать принцип *малой площади* с помощью экспорта традиционной асинхронной функции

как основной функциональной возможности, возлагая расширенные возможности на `EventEmitter`. Одним из примеров применения этого шаблона может служить модуль `node-glob` (https://npmjs.org/package/glob), реализующий поиск файлов с применением шаблонных символов. Главной точкой входа в модуль является экспортируемая функция со следующей сигнатурой:

```
glob(pattern, [options], callback)
```

Эта функция принимает в первом аргументе шаблон поиска, за ним следуют набор параметров и функция обратного вызова, которой передается список всех файлов, соответствующих шаблону. В то же время функция возвращает `EventEmitter`, позволяющий получить более подробный отчет о состоянии процесса. Например, можно получать уведомления в режиме реального времени о найденных совпадениях, обрабатывая события `match`, получить список всех найденных файлов через событие `end` и узнать о прерывании процесса вручную, подписавшись на событие `abort`. Следующий код демонстрирует это:

```
const glob = require('glob');
glob('data/*.txt', (error, files) => console.log(`All files found:
  ${JSON.stringify(files)}`))
  .on('match', match => console.log(`Match found: ${match}`));
```

Как видите, этот подход обеспечивает простоту, четкость, единственную точку входа и вместе с тем дополнительные инструменты с более мощными, но менее важными функциями, что является довольно распространенной практикой на платформе Node.js, причем сочетание `EventEmitter` с традиционными обратными вызовами является одним из способов достичь этого.

Шаблон
Создается функция, принимающая функцию обратного вызова и возвращающая `EventEmitter`, что обеспечивает простую и явную точку входа для выполнения основных функций и генерацию событий с помощью `EventEmitter`, предоставляющих более подробную информацию.

Итоги

В этой главе была определена разница между синхронным и асинхронным кодом. Затем мы рассмотрели шаблоны использования обратных вызов и генерации событий для нескольких базовых сценариев асинхронной обработки. Были описаны основные различия между двумя шаблонами и порядок выбора одного из них для решения конкретных задач. Мы только что сделали первый серьезный шаг и подготовились к знакомству с более сложными шаблонами асинхронной обработки.

Следующая глава будет посвящена более сложным сценариям. В ней мы посмотрим, как использовать шаблоны Callback (Обратный вызов) и Event Emitter (Генератор событий) для управления асинхронными потоками выполнения.

Глава 3

Шаблоны асинхронного выполнения с обратными вызовами

Переход от синхронного стиля программирования на такую платформу, как Node.js, где стиль передачи продолжений и асинхронные программные интерфейсы являются нормой, может быть болезненным. Разработка асинхронного кода требует другого подхода, особенно когда дело касается потока управления. Асинхронный код способен сделать порядок выполнения операторов в приложениях на платформе Node.js труднопредсказуемым, поэтому решение даже таких простых задач, как перебор файлов, выполнение заданий в нужной последовательности или ожидание завершения группы операций, требует от разработчика применения новых подходов и способов, чтобы избежать неэффективного и неудобочитаемого кода. Здесь легко можно угодить в ловушку проблемы «ада обратных вызовов» и столкнуться с разбуханием кода по горизонтали, а не по вертикали, с увеличением вложенности, делающей даже простые процедуры неудобочитаемыми и тяжело поддерживаемыми.

В этой главе мы увидим, как приручить обратные вызовы и писать чистый, управляемый асинхронный код, применяя определенные правила и шаблоны. Также мы увидим, как использование таких библиотек, как `async`, может значительно упростить задачу, сделав код удобочитаемым и простым в обслуживании.

Сложности асинхронного программирования

Потерять контроль над асинхронным кодом, написанным на JavaScript, несложно. Замыкания и встроенные определения анонимных функций обеспечивают последовательность программного кода, ограждающую разработчиков от необходимости перескакивать между фрагментами кода. Это отлично согласуется с принципом KISS, упрощает анализ код и сокращает время разработки. К сожалению, принесение в жертву таких характеристик кода, как модульность, возможность повторного использования и удобство поддержки, рано или поздно приведет к неконтролируемому увеличению уровней вложенности обратных вызовов, росту размера функций и плохой организации кода. Как правило, для достижения нужной функциональности не требуется создание замыканий, так что это скорее вопрос дисциплины, чем проблема асинхронного программирования. Умение вовремя обнаружить, что код становится

громоздким, или даже предвидеть, что он может стать громоздким, и принять наиболее адекватное решение является основным отличием дилетанта от профессионала.

Создание простого поискового робота

Для иллюстрации этой проблемы создадим небольшого поискового робота (веб-паука) – приложение, запускаемое из командной строки, которое принимает URL-адрес и загружает его содержимое в локальный файл. В представленном в этой главе коде будет использоваться несколько npm-зависимостей:

- `request`: библиотека для упорядочения HTTP-вызовов;
- `mkdirp`: небольшая утилита для рекурсивного создания каталогов.

Кроме того, мы часто будем ссылаться на локальный модуль `./utilities`, содержащий несколько вспомогательных функций. Содержимое этого файла не будет здесь приведено, но вы найдете его в загружаемом пакете примеров вместе с файлом `package.json`, содержащим полный список зависимостей, доступном по адресу: http://www.packtpub.com.

Основные функциональные возможности приложения размещены в модуле `spider.js`. Ознакомимся с его содержимым. Начнем с загрузки всех необходимых зависимостей:

```
const request = require('request');
const fs = require('fs');
const mkdirp = require('mkdirp');
const path = require('path');
const utilities = require('./utilities');
```

Затем создадим функцию `spider()`, принимающую URL-адрес, и функцию обратного вызова, которая будет вызываться по завершении процесса загрузки:

```
function spider(url, callback) {
  const filename = utilities.urlToFilename(url);
  fs.exists(filename, exists => {                    //[1]
    if(!exists) {
      console.log(`Downloading ${url}`);
      request(url, (err, response, body) => {       //[2]
        if(err) {
          callback(err);
        } else {
          mkdirp(path.dirname(filename), err => {    //[3]
            if(err) {
              callback(err);
            } else {
              fs.writeFile(filename, body, err => {  //[4]
                if(err) {
                  callback(err);
                } else {
                  callback(null, filename, true);
                }
              });
            }
          });
        }
```

```
      });
    } else {
      callback(null, filename, false);
    }
  });
}
```

Она выполняет следующие действия:

1) проверяет, выполнялась ли загрузка содержимого URL ранее, определяя наличие уже созданного файла:

   ```
   fs.exists(filename, exists => ...
   ```

2) если файл не найден, выполняется загрузка содержимого URL с помощью следующей строки кода:

   ```
   request(url, (err, response, body) => ...
   ```

3) затем создается каталог, куда будет помещен файл, если он не был создан ранее:

   ```
   mkdirp(path.dirname(filename), err => ...
   ```

4) и наконец, тело HTTP-ответа записывается в локальный файл:

   ```
   fs.writeFile(filename, body, err => ...
   ```

Наконец, необходимо вызвать функцию `spider()` и передать ей адрес URL (в данном случае он извлекается из аргументов командной строки):

```
spider(process.argv[2], (err, filename, downloaded) => {
  if(err) {
    console.log(err);
  } else if(downloaded){
    console.log(`Completed the download of "${filename}"`);
  } else {
    console.log(`"${filename}" was already downloaded`);
  }
});
```

Теперь можно попробовать запустить приложение, но перед этим нужно удостовериться, что в каталоге проекта имеются модуль `utilities.js` и файл `package.json`, содержащий полный список зависимостей, и установить все зависимости, выполнив следующую команду:

```
npm install
```

Далее можно запустить модуль `spider`, чтобы загрузить веб-страницу, с помощью следующей команды:

```
node spider http://www.example.com
```

Приложение веб-паука требует указывать протокол (например, `http://`) в URL-адресе. Кроме того, не ждите от него загрузки контента HTML-ссылок и ресурсов, таких как изображения, поскольку это лишь простой пример, демонстрирующий работу асинхронного кода.

Ад обратных вызовов

Просматривая код функции `spider()`, несложно заметить, что, несмотря на простоту алгоритма, получившийся в результате код содержит несколько уровней отступов

и тяжело читается. Реализация той же функции с применением прямого блокирующего стиля будет выглядеть проще, и у нее будет немного шансов выглядеть настолько неправильной. Но при использовании асинхронного стиля передачи продолжения дело обстоит совершенно иначе, и неумелое применение замыканий приводит к очень неудачному коду.

Изобилие замыканий и непосредственных определений обратных вызовов, превращающее код в нечитаемый и необслуживаемый набор операторов, называется **адом обратных вызовов**. Это один из наиболее известных и трудных в исправлении антишаблонов платформы Node.js и языка JavaScript в целом. Вот как выглядит типичная структура кода, вызывающая эту проблему:

```
asyncFoo( err => {
  asyncBar( err => {
    asyncFooBar( err => {
      //...
    });
  });
});
```

Как видите, код принимает форму пирамиды из-за глубокой вложенности, поэтому неофициально такой код называют «**обреченной пирамидой**».

Очевидной проблемой предыдущего фрагмента является явная неудобочитаемость. Из-за слишком большего количества уровней вложенности практически невозможно отследить, где заканчивается одна функция и начинается другая.

Еще одной его проблемой является использование одноименных переменных в разных областях видимости. Для описания назначения переменных мы часто используем похожие или даже идентичные имена. Типичным примером является аргумент `error` функций обратного вызова, используемый для передачи ошибок. Некоторые разработчики для различения переменных в разных областях видимости используют различные варианты одного и того же имени, например `err`, `error`, `err1`, `err2` и т. д.; другие предпочитают всегда использовать одно и то же имя, например `err`. Обе эти альтернативы далеки от совершенства, способны привести к путанице и увеличивают вероятность ошибок.

Кроме того, имейте в виду, что замыкания привносят определенные накладные расходы с точки зрения производительности и потребления памяти. Они также могут создавать утечки памяти, которые сложно выявить, поскольку любой контекст, ссылающийся на активное замыкание, при сборке мусора сохраняется.

Более подробные сведения о работе замыканий в V8 можно найти в блоге Вячеслава Егорова (Vyacheslav Egorov), инженера-программиста, работающего в Google над V8: http://mrale.ph/blog/2012/09/23/grokking-v8-closures-for-fun.html.

Как видите, код функции `spider()` соответствует ситуации, называемой адом обратных вызовов, и характеризуется всеми описанными выше недостатками. Их исправлению с помощью соответствующих шаблонов и методик будет посвящена вся эта глава.

Использование обычного JavaScript

Теперь, после знакомства с первым примером ада обратных вызовов, понятно, чего следует избегать, но это не единственная проблема, возникающая при разработке

асинхронного кода. На самом деле существует несколько ситуаций, когда управление выполнением набора асинхронных задач требует использования конкретных шаблонов и методик, особенно если используется только обычный JavaScript без каких-либо внешних библиотек. Например, последовательный обход элементов коллекции с помощью асинхронной операции реализуется сложнее, чем простой вызов `forEach()` для массива, поскольку для этого требуется прием, похожий на рекурсию.

В этом разделе мы узнаем, как избежать ада обратных вызовов и реализовать некоторые наиболее распространенные шаблоны управления, используя только обычный JavaScript.

Дисциплина обратных вызовов

Первое правило, которое следует иметь в виду, разрабатывая асинхронный код: не злоупотреблять замыканиями при определении обратных вызовов. Замыкания выглядят заманчивыми, поскольку не заставляют задумываться над таким вопросами, как модульность и повторное использование, но, как уже упоминалось выше, они привносят больше недостатков, чем преимуществ. Как правило, чтобы избежать ада обратных вызовов, не требуется каких-либо библиотек, магических методик и смены парадигмы, достаточно простого здравого смысла.

Ниже перечислены некоторые основные принципы, помогающие избежать роста уровня вложенности и улучшающие организацию кода в целом.

- Выходите при первой возможности. Используйте `return`, `continue` или `break` в зависимости от контекста, чтобы немедленно выйти из охватывающего оператора, – избегайте сложных (и вложенных) операторов `if...else`. Это поможет избавиться от глубокой вложенности кода.
- Создавайте именованные функции обратного вызова, чтобы они не образовывали замыканий, а промежуточные результаты передавайте в аргументах. Кроме того, именованные функции заметнее в трассировке стека.
- Стремитесь к повышению модульности кода. Разбивайте код на небольшие функции, пригодные к многократному использованию, где только возможно.

Применение дисциплины обратных вызовов

Для демонстрации возможностей перечисленных принципов применим их для решения проблемы ада обратных вызовов в приложении веб-паука.

Начнем с реорганизации шаблона проверки наличия ошибок, исключив из него оператор `else`. Это позволяет выйти из функции сразу после появления ошибки. То есть заменим следующий фрагмент:

```
if(err) {
  callback(err);
} else {
  //код, выполняемый в отсутствие ошибок
}
```

улучшенной версией:

```
if(err) {
  return callback(err);
}
// код, выполняемый в отсутствие ошибок
```

Этот простой прием сразу устраняет лишний уровень вложенности. Он прост и не требует сложной реорганизации.

Часто, выполняя описанную только что оптимизацию, программисты забывают о выходе после вызова функции обратного вызова. При реализации сценария обработки ошибок типичным источником сбоев является приведенный ниже код:

```
if(err) {
  callback(err);
} // код, выполняемый в отсутствие ошибок
```

Никогда не забывайте, что выполнение будет продолжено после возврата из функции обратного вызова. Поэтому важно вставить инструкцию `return`, чтобы избежать выполнения оставшейся части функции. Также обратите внимание, что возвращаемый функцией результат не имеет особого значения, поскольку фактический результат (или ошибка) передается асинхронно методу обратного вызова. Значение, возвращаемое асинхронной функции, обычно игнорируется. Эта особенность позволяет использовать краткую запись:

```
return callback(...)
```

В противном случае пришлось бы использовать более длинную запись:

```
callback(...)
return;
```

В качестве второй оптимизации функции `spider()` можно попытаться выявить повторно используемые фрагменты ее кода. Например, запись строки в файл можно вынести в отдельную функцию:

```
function saveFile(filename, contents, callback) {
  mkdirp(path.dirname(filename), err => {
    if(err) {
      return callback(err);
    }
    fs.writeFile(filename, contents, callback);
  });
}
```

Следуя тому же принципу, можно создать универсальную функцию `download()`, принимающую URL и имя файла. Внутри нее можно использовать созданную ранее функцию `saveFile()`.

```
function download(url, filename, callback) {
  console.log(`Downloading ${url}`);
  request(url, (err, response, body) => {
    if(err) {
      return callback(err);
    }
    saveFile(filename, body, err => {
      if(err) {
        return callback(err);
      }
      console.log(`Downloaded and saved: ${url}`);
      callback(null, body);
    });
  });
}
```

Наконец, изменим функцию `spider()`, которая благодаря проделанной реорганизации теперь будет выглядеть так:

```
function spider(url, callback) {
  const filename = utilities.urlToFilename(url);
  fs.exists(filename, exists => {
    if(exists) {
      return callback(null, filename, false);
    }
    download(url, filename, err => {
      if(err) {
        return callback(err);
      }
      callback(null, filename, true);
    })
  });
}
```

Принцип действия и интерфейс функции `spider()` остались неизменными, изменилась лишь организация кода. Применение описанных выше принципов позволило существенно уменьшить уровень вложенности кода, увеличить возможность его многократного использования и упростить тестирование. Теперь можно рассмотреть вопрос об экспорте функций `saveFile()` и `download()`, чтобы их можно было использовать в других модулях. Это также упростит тестирование этих модулей.

Проведенная реорганизация демонстрирует, что для получения желаемого результата достаточно не злоупотреблять замыканиями и анонимными функциями. Следование этой дисциплине требует минимальных усилий и применения обычного JavaScript.

Последовательное выполнение

Теперь рассмотрим шаблоны асинхронного управления выполнением. Начнем с анализа потока последовательного выполнения.

Последовательное выполнение набора заданий означает их запуск по одному, друг за другом. Порядок выполнения имеет значение и должен быть сохранен, поскольку результат предыдущего задания может повлиять на выполнение следующего. Приведенная на рис. 3.1 схема иллюстрирует эту идею:

Рис. 3.1 ❖ Последовательное выполнение заданий

Существуют различные варианты этого способа выполнения:

- последовательное выполнение набора известных заданий, без цепочек и передачи результатов;
- передача результатов одного задания на вход следующего (этот способ известен также как *цепочка*, *конвейер* или *каскад*);

- последовательный обход элементов коллекции и выполнение асинхронного задания для каждого из них.

Последовательное выполнение, несмотря на всю тривиальность при использовании прямого блокирующего стиля, обычно является главной причиной ада обратных вызовов при применении асинхронного стиля передачи продолжений.

Последовательное выполнение известного набора заданий

Последовательное выполнение уже рассматривалось при реализации функции `spider()` из предыдущего раздела. С помощью простых правил мы реализовали последовательное выполнение набора известных заданий. Опираясь на этот код, можно обобщить реализованное решение с помощью следующего шаблона:

```
function task1(callback) {
  asyncOperation(() => {
    task2(callback);
  });
}
function task2(callback) {
  asyncOperation(result () => {
    task3(callback);
  });
}
function task3(callback) {
  asyncOperation(() => {
    callback(); //обратный вызов в завершающем задании
  });
}

task1(() => {
  //выполнится после завершения task1, task2 и task3
  console.log('tasks 1, 2 and 3 executed');
});
```

Приведенный шаблон демонстрирует вызов каждого следующего задания после выполнения общей асинхронной операции. Шаблон ориентирован на модульность заданий и демонстрирует, что в асинхронном коде можно обойтись без замыканий.

Последовательные итерации

Описанный шаблон прекрасно справляется, если количество заданий известно заранее. Это позволяет точно определить момент вызова очередного задания в последовательности, а что, если требуется выполнить асинхронную операцию для каждого элемента коллекции? В этом случае не получится жестко определить последовательность заданий, ее придется строить динамически.

Веб-паук, версия 2 Для демонстрации примера последовательных итераций введем новую функцию в приложение. Теперь требуется рекурсивно загрузить все содержащиеся на веб-странице ссылки. Для этого нужно извлечь все ссылки со страницы, а затем рекурсивно и последовательно вызвать веб-паука для каждой из них.

Первым делом изменим функцию `spider()`, чтобы она запускала рекурсивную загрузку всех ссылок, используя функцию `spiderLinks()`, которая будет определена позже.

Кроме того, вместо проверки существования файла теперь нужно попытаться прочитать его и начать работу со ссылками в нем, это позволит возобновить прерванную загрузку. И наконец, добавим новый параметр `nesting`, ограничивающий глубину рекурсии. Окончательный код будет выглядеть так:

```
function spider(url, nesting, callback) {
  const filename = utilities.urlToFilename(url);
  fs.readFile(filename, 'utf8', (err, body) => {
    if(err) {
      if(err.code ! == 'ENOENT') {
        return callback(err);
      }

      return download(url, filename, (err, body) => {
        if(err) {
          return callback(err);
        }
        spiderLinks(url, body, nesting, callback);
      });
    }

    spiderLinks(url, body, nesting, callback);
  });
}
```

Последовательное сканирование ссылок Теперь можно создать ядро новой версии приложения веб-паука, функцию `spiderLinks()`, загружающую все ссылки с помощью алгоритма асинхронных последовательных итераций. Обратите внимание на подход, применяемый в следующем блоке кода:

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if(nesting === 0) {
    return process.nextTick(callback);
  }
  const links = utilities.getPageLinks(currentUrl, body);  //[1]
  function iterate(index) {                                //[2]
    if(index === links.length) {
      return callback();
    }

    spider(links[index], nesting - 1, err => {             //[3]
      if(err) {
        return callback(err);
      }
      iterate(index + 1);
    });
  }
  iterate(0);                                              //[4]
}
```

Наиболее важные элементы новой функции перечислены ниже.

1. Получение списка всех ссылок на странице с помощью функции `utilities.getPageLinks()`. Эта функция возвращает только внутренние ссылки (с тем же именем хоста).

2. Перебор ссылок с помощью локальной функции `iterate()`, которой передается индекс следующей анализируемой ссылки. Эта функция сначала проверяет равенство индекса и длины массива ссылок; если они равны, сразу же вызывается функция `callback()`, поскольку были обработаны все элементы.
3. Теперь все готово для обработки ссылки. Мы вызываем функцию `spider()`, уменьшая `nesting`, и выполняем следующую итерацию после завершения операции.
4. Последним действием в функции `spiderLinks()` является вызов `iterate(0)`, инициирующий итерации.

Представленный алгоритм позволяет обойти элементы массива путем последовательного выполнения асинхронной операции, которой в нашем случае является функция `spider()`.

Теперь можно попробовать запустить новую версию приложения и посмотреть, как она рекурсивно загрузит все ссылки на веб-странице, одну за другой. Для прерывания процесса, который может занять длительное время при наличии большого количества ссылок, всегда можно использовать сочетание клавиш **Ctrl**+**C**. Чтобы позднее возобновить загрузку, достаточно запустить приложение с тем же URL, что был указан при первом запуске.

Теперь, когда приложение веб-паука потенциально может загрузить весь веб-сайт, следует проявлять осторожность при его использовании. Например, не устанавливайте слишком высокий уровень вложенности и не заставляйте паука работать более нескольких секунд. Некрасиво перегружать сервер тысячами запросов. В некоторых случаях это может быть сочтено незаконным. Будьте ответственными!

Шаблон Код функции `spiderLinks()` является наглядным примером, как осуществить обход элементов коллекции с применением асинхронной операции. Следует отметить, что этот шаблон можно адаптировать для применения в любой другой ситуации, где требуется асинхронно выполнить последовательный обход элементов коллекции или вообще любого списка заданий. Этот шаблон можно обобщить, как показано ниже:

```
function iterate(index) {
  if(index === tasks.length)  {
    return finish();
  }
  const task = tasks[index];
  task(function() {
    iterate(index + 1);
  });
}

function finish() {
  //обход завершен
}

iterate(0);
```

Важно отметить, что алгоритмы такого типа становятся действительно рекурсивными, если `task()` является синхронной операцией. В этом случае стек не будет обновляться в каждом цикле и может возникнуть риск превысить максимальный размер стека вызовов.

Это очень мощный шаблон, и его можно адаптировать для целого ряда ситуаций. Например, основываясь на нем, можно организовать преобразование значений элементов массива, передавать результаты из одной операции в другую для реализации алгоритма свертки, преждевременно завершать цикл по определенному условию и даже осуществлять итерации по элементам бесконечной коллекции.

Это решение можно обобщить еще больше, заключив его в функцию со следующей сигнатурой:

```
iterateSeries(collection, iteratorCallback, finalCallback)
```

Мы предлагаем это вам в качестве самостоятельного упражнения.

Шаблон (последовательный итератор)
Последовательное выполнение заданий из списка путем создания функции `iterator`, которая вызывает следующее доступное задание в коллекции и гарантирует выполнение следующего шага итерации после завершения текущего задания.

Параллельное выполнение

Существуют определенные ситуации, когда порядок выполнения асинхронных заданий из набора не важен и достаточно простого уведомления о завершении всех заданий. Для таких случаев лучше подходит параллельное выполнение, схема которого представлена на рис. 3.2.

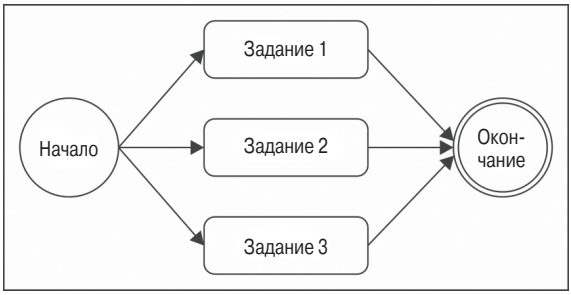

Рис. 3.2 ❖ Параллельное выполнение заданий

Подобное предложение выглядит странным, если учесть однопоточный характер платформы Node.js, но, как упоминалось в главе 1 «Добро пожаловать в платформу Node.js», неблокирующая природа Node.js может обеспечить параллельное выполнение. В действительности слово «параллельная» здесь не вполне подходит, поскольку в данном случае оно не означает, что задания выполняются одновременно, а скорее указывает, что их выполнение не блокирует программного интерфейса и чередуется с выполнением цикла событий.

Как упоминалось ранее, задание возвращает управление циклу событий, когда запрашивает новую асинхронную операцию, позволяя циклу событий выполнить другие задания. Правильным определением выполнения такого рода является слово «конкурентная», но здесь для простоты и далее будет использоваться слово «параллельная».

Схема на рис. 3.3 демонстрирует порядок параллельного выполнения двух асинхронных заданий в программе на платформе Node.js.

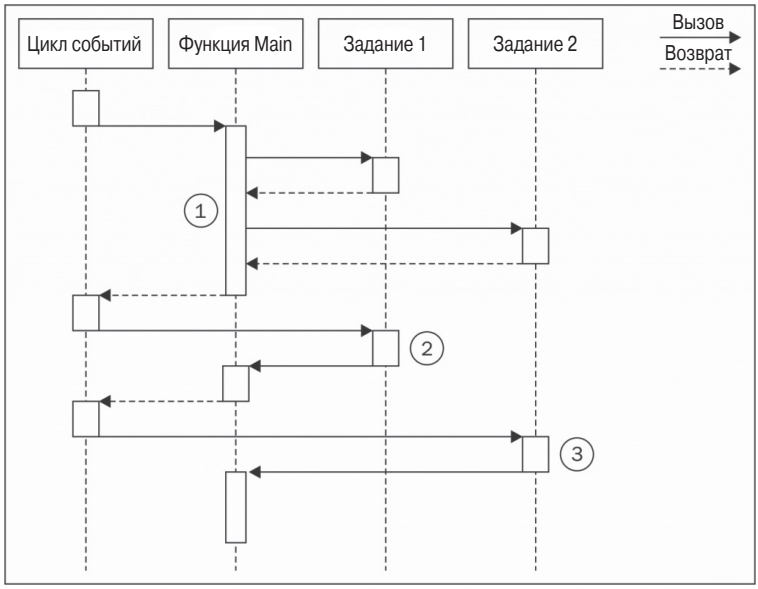

Рис. 3.3 ❖ Параллельное выполнение двух асинхронных заданий

В схеме на рис. 3.3 показана функция `Main`, выполняющая две асинхронные задачи:

1) функция `Main` запускает выполнение заданий 1 и 2. Поскольку здесь запуск является асинхронной операцией, управление немедленно возвращается функции `Main`, которая передает его циклу событий;
2) после завершения асинхронной операции в задании 1 управление передается циклу событий. Кроме того, когда задание 1 завершится, оно уведомит функцию `Main`;
3) после асинхронной операции в задании 2 цикл событий вызывает его функцию обратного вызова, возвращая управление заданию 2. Когда задание 2 завершится, оно уведомит функцию `Main`. Теперь функция `Main` знает, когда завершатся задания 1 и 2, поэтому она может продолжить выполнение или вернуть результаты операций через другую функцию обратного вызова.

Проще говоря, на платформе Node.js можно параллельно выполнять только асинхронные операции, поскольку их параллельная обработка осуществляется внутренним неблокирующим программным интерфейсом. На платформе Node.js синхронные (блокирующие) операции не могут выполняться параллельно, если их выполнение не чередуется с асинхронными операциями или не откладывается с помощью функции `setTimeout()` или `setImmediate()`. Более подробно об этом рассказывается в *главе 9 «Дополнительные рецепты асинхронной обработки»*.

Веб-паук, версия 3

Приложение веб-паука выглядит идеальным кандидатом для применения идеи параллельного выполнения. На данный момент приложение выполняет рекурсивную загрузку связанных страниц последовательно. Мы можем заметно улучшить производительность этого процесса, перейдя к параллельной загрузке.

Для этого потребуется изменить функцию `spiderLinks()`, чтобы одновременно запустить все задания `spider()` и вызвать заключительную функцию обратного вызова только после их завершения. Изменим функцию `spiderLinks()`, как показано ниже:

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if(nesting === 0) {
    return process.nextTick(callback);
  }
  const links = utilities.getPageLinks(currentUrl, body);
  if(links.length === 0) {
    return process.nextTick(callback);
  }
  let completed = 0, hasErrors = false;
  function done(err) {
    if(err) {
      hasErrors = true;
      return callback(err);
    }
    if(++completed === links.length && !hasErrors) {
      return callback();
    }
  }

  links.forEach(link => {
    spider(link, nesting - 1, done);
  });
}
```

Поясним произведенные изменения. Как уже упоминалось, теперь все задания `spider()` запускаются одновременно. Это достигается путем обхода массива ссылок с запуском заданий без ожидания завершения запущенных ранее заданий:

```
links.forEach(link => {
  spider(link, nesting - 1, done);
});
```

Далее применяется хитрый прием, обеспечивающий ожидание приложением завершения всех заданий, заключающийся в передаче функции `spider()` специальной функции обратного вызова `done()`. Функция `done()` увеличивает значение счетчика при завершении каждого задания `spider`. Когда количество завершенных загрузок станет равно длине массива ссылок, будет вызвана завершающая функция обратного вызова:

```
function done(err) {
  if(err) {
    hasErrors = true;
    return callback(err);
  }
  if(++completed === links.length && !hasErrors) {
    callback();
  }
}
```

Если теперь, после описанных изменений, попробовать запустить приложение для той же веб-страницы, можно заметить существенное ускорение всего процесса, поскольку все загрузки выполняются параллельно, не дожидаясь обработки предыдущих ссылок.

Шаблон

Организацию параллельного выполнения можно выразить в виде небольшого, элегантного шаблона и затем адаптировать и повторно использовать его в различных ситуациях. Универсальную версию шаблона можно представить с помощью следующего кода:

```
const tasks = [ /* ... */ ];
let completed = 0;
tasks.forEach(task => {
  task(() => {
    if(++completed === tasks.length) {
      finish();
    }
  });
});

function finish() {
  //все задания выполнены
}
```

Внеся незначительные изменения, в этом шаблоне можно организовать накопление результатов выполнения заданий в коллекции, фильтровать или преобразовывать элементы массива, вызывать функцию обратного вызова `finish()` при завершении одного или заданного количества заданий (последняя ситуация, в частности, называется **конкурентной гонкой**).

Шаблон (неограниченного параллельного выполнения)
Параллельный запуск набора асинхронных заданий с последующим ожиданием их завершения путем подсчета количества обратных вызовов, выполненных ими.

Устранение состояния гонки в параллельных заданиях

Параллельное выполнение набора заданий может вызывать проблемы при использовании блокирующего ввода/вывода в сочетании с несколькими потоками. Но, как только что было продемонстрировано, к платформе Node.js это не относится, поскольку здесь параллельный запуск нескольких асинхронных заданий является простой операцией с небольшой затратой ресурсов. Одно из наиболее важных преимуществ платформы Node.js заключается в том, что здесь параллельная обработка является обычной практикой, а не сложной технологией, используемой только в случае крайней необходимости.

Еще одной важной характеристикой модели параллельной обработки на платформе Node.js являются способ синхронизации заданий и устранение состояния гонки за ресурсами. Для решения этих проблем в многопоточном программировании используются такие конструкции, как блокировки, мьютексы, семафоры и мониторы, и этот аспект считается одним из самых сложных в реализации параллельной обработки, поскольку оказывает значительное влияние на производительность. На платформе Node.js отсутствует необходимость в каком-то особенном механизме синхронизации, поскольку все здесь выполняется в одном потоке! Но это не означает, что здесь не может возникнуть состояния гонки за ресурсами, напротив, с ним приходится сталкиваться довольно часто. Проблема заключается в задержке между вызовами асинхронных операций и уведомлением об их результатах. Чтобы привести конкретный

пример, вновь обратимся к приложению веб-паука, в частности к его последней версии, которая страдает проблемой состояния гонки (сможете сами определить, где она возникает?).

Проблема заключается в функции `spider()`, где выполняется проверка существования файла перед загрузкой содержимого URL:

```
function spider(url, nesting, callback) {
  const filename = utilities.urlToFilename(url);
  fs.readFile(filename, 'utf8', (err, body) => {
    if(err) {
      if(err.code !== 'ENOENT') {
        return callback(err);
      }
      return download(url, filename, function(err, body) {
//...
```

Проблема заключается в том, что два задания, работающих с одним URL, могут обращаться в методе `fs.readFile()` к одному и тому же файлу, прежде чем одно из них успеет завершить загрузку и создать этот файл, в результате оба задания начнут загрузку. Эту ситуацию иллюстрирует схема на рис. 3.4.

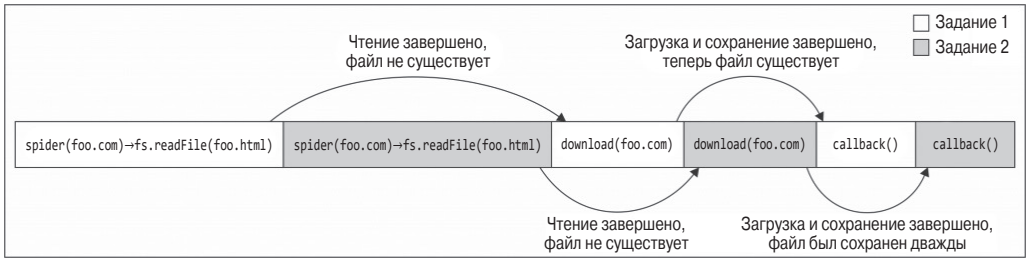

Рис. 3.4 ❖ Состояние гонки

На приведенной схеме показано, как осуществляется поочередное выполнение заданий 1 и 2 в одном потоке платформы Node.js и как асинхронные операции действительно могут попадать в состояние гонки. В данном случае два задания `spider` в конечном итоге выполнили загрузку одного и того же файла.

Как исправить это? Ответ на этот вопрос гораздо проще, чем кажется: достаточно завести переменную, позволяющую исключить обработку одного и того же URL несколькими заданиями `spider()`. Например:

```
const spidering = new Map();
function spider(url, nesting, callback) {
  if(spidering.has(url)) {
    return process.nextTick(callback);
  }
  spidering.set(url, true);

//...
```

Это исправление не требует особых комментариев. Здесь просто выполняется немедленный выход из функции, если данный `url` уже присутствует в словаре `spidering`,

иначе он помещается в словарь и выполняется его загрузка. В данном случае нет смысла снимать блокировку (удалять элемент из словаря), потому что не требуется дважды загружать один и тот же URL, даже если выполнение заданий `spider` разнесено по времени.

Состояние гонки приводит ко множеству проблем даже в однопоточной среде. В некоторых случаях оно может вызвать повреждение данных, и, как правило, такие ошибки очень трудно отлаживать из-за их эфемерной природы. Поэтому всегда проверяйте невозможность подобных ситуаций при параллельном выполнении заданий.

Ограниченное параллельное выполнение

Зачастую бесконтрольный запуск параллельных заданий приводит к чрезмерной нагрузке. Представьте себе запущенные одновременно тысячи операций чтения файлов, обращений к URL-адресам или запросов к базам данных. Обычной проблемой в таких ситуациях становится исчерпание ресурсов, например задействование всех доступных приложению дескрипторов файлов при попытке одновременно открыть слишком большое количество файлов. Кроме того, такое поведение может вызвать в веб-приложении уязвимость **отказ в обслуживании** (Denial of Service, DoS). Во всех подобных ситуациях имеет смысл ограничить число заданий, выполняющихся одновременно. То есть можно добиться определенной предсказуемой нагрузки на сервер и гарантировать, что приложение не исчерпает всех ресурсов. На рис. 3.5 представлена схема, иллюстрирующая параллельное выполнение пяти заданий с ограничением, не допускающим одновременного выполнения более двух заданий.

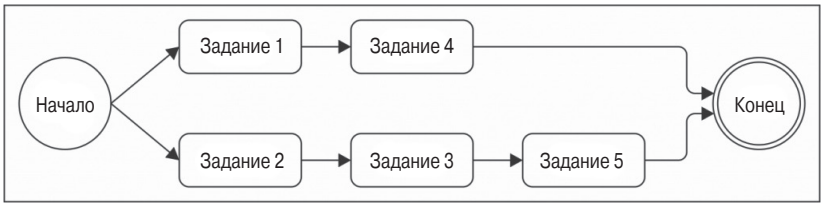

Рис. 3.5 ❖ Ограничение числа заданий, выполняющихся одновременно

Из схемы на рис. 3.2 должно быть ясно, как действует следующий алгоритм:

1) первоначально запускается столько заданий, сколько позволяет заданный предел;
2) при завершении очередного задания запускается одно или несколько заданий до достижения заданного предела.

Ограничение параллельной обработки

Ниже приводится шаблон параллельного выполнения набора задач с ограничением параллельной обработки:

```
const tasks = ...
let concurrency = 2, running = 0, completed = 0, index = 0;
function next() {                                          //[1]
  while(running < concurrency && index < tasks.length) {
    task = tasks[index++];
    task(() => {                                           //[2]
```

```
      if(completed === tasks.length) {
        return finish();
      }
      completed++, running--;
      next();
    });
    running++;
  }
}
next();

function finish() {
  //все задания выполнены
}
```

Этот алгоритм можно рассматривать как сочетание последовательного и параллельного выполнения. В самом деле, в нем можно заметить сходство с шаблонами, представленными ранее в этой главе:

1) здесь имеются функция-итератор `next()` и внутренний цикл, обеспечивающий параллельный запуск стольких заданий, сколько позволяет предел параллельной обработки;
2) другим важным аспектом является передаваемая в каждое задание функция обратного вызова, проверяющая, завершено ли выполнение всех заданий из списка. Если еще остались невыполненные задания, она вызывает функцию `next()` для запуска следующих заданий.

Довольно просто, не так ли?

Глобальное ограничение параллельной обработки

Приложение веб-паука идеально подходит для применения ограничения параллельной обработки. В самом деле, чтобы избежать одновременной обработки тысяч ссылок, нужно предусмотреть ограничение, обеспечив таким образом предсказуемое количество одновременных загрузок.

Версии платформы Node.js до 0.11 не допускали более пяти одновременных HTTP-подключений к одному хосту. Но это значение можно было изменить при необходимости. Более подробные сведения можно найти в официальной документации на странице http://nodejs.org/docs/v0.10.0/api/http.html#http_agent_maxsockets. Начиная с версии Node.js 0.11 это ограничение было убрано.

Мы могли бы применить к функции `spiderLinks()` только что рассмотренный шаблон, но в результате получилась бы только ограниченная параллельная обработка ссылок, найденных на одной странице. Установив порог равным, например, 2, мы разрешили бы одновременную загрузку не более двух ссылок для каждой страницы. Но поскольку одновременно может загружаться сразу несколько ссылок, каждая страница запустит еще две загрузки, вызывая таким образом рост общего числа параллельных загрузок в геометрической прогрессии.

Очереди спасут положение На самом деле требуется ограничить общее количество операций загрузки, выполняемых параллельно. Для этого можно было бы несколько изменить предыдущий шаблон, но мы предпочли оставить это вам в качестве самостоятельного упражнения и хотим воспользоваться представившейся возмож-

ностью, чтобы рассмотреть еще один механизм, использующий очереди для ограничения параллельного выполнения нескольких заданий. Рассмотрим, как это будет работать.

Реализуем простой класс `TaskQueue`, сочетающий в себе очередь с рассмотренным выше алгоритмом. Создадим для этого новый модуль `taskQueue.js`:

```
class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency;
    this.running = 0;
    this.queue = [];
  }

  pushTask(task) {
    this.queue.push(task);
    this.next();
  }
  next() {
    while(this.running < this.concurrency && this.queue.length) {
      const task = this.queue.shift();
      task(() => {
        this.running--;
        this.next();
      });
      this.running++;
    }
  }
};
```

Конструктор класса принимает на входе только предельное количество заданий, выполняющихся параллельно, но вместе с тем инициализирует переменные `running` и `queue`. Ранее для подсчета запущенных заданий использовалась переменная, теперь ее место займет массив `queue`, играющий роль очереди заданий, ожидающих выполнения.

Метод `pushTask()` просто добавляет новое задание в очередь и запускает обработку вызовом метода `this.next()`.

Метод `next()` запускает несколько заданий из очереди, учитывая установленное ограничение.

Как видите, этот метод похож на рассмотренный ранее шаблон ограничения параллельной обработки. По сути, он запускает максимально возможное количество заданий из очереди, не превышая установленного ограничения. После завершения задания он корректирует счетчик выполняющихся заданий и запускает следующую порцию заданий, повторно вызывая метод `next()`. Интересной особенностью класса `TaskQueue` является поддержка динамического добавления новых элементов в очередь. Другое преимущество заключается в наличии центрального органа, отвечающего за ограничение количества выполняемых заданий, который может совместно использоваться всеми экземплярами выполняющихся функций. В данном случае это функция `spider()`, которая будет представлена чуть позже.

Веб-паук, версия 4 Теперь используем обобщенную очередь для ограничения количества одновременно выполняющихся заданий в приложении веб-паука. Нач-

нем с загрузки новой зависимости и создания экземпляра класса TaskQueue с ограничением, равным 2:

```
const TaskQueue = require('./taskQueue');
const downloadQueue = new TaskQueue(2);
```

Затем нужно изменить функцию spiderLinks(), чтобы она могла использовать вновь созданный экземпляр downloadQueue:

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if(nesting === 0) {
    return process.nextTick(callback);
  }

  const links = utilities.getPageLinks(currentUrl, body);
  if(links.length === 0) {
    return process.nextTick(callback);
  }

  let completed = 0, hasErrors = false;
  links.forEach(link => {
    downloadQueue.pushTask(done => {
      spider(link, nesting - 1, err => {
        if(err) {
          hasErrors= true;
          return callback(err);
        }
        if(++completed === links.length && !hasErrors) {
          callback();
        }
        done();
      });
    });
  });
}
```

Новая реализация функции очень проста и очень похожа на реализацию алгоритма параллельного выполнения без ограничений, которая приводилась выше в этой главе. Это связано с тем, что управление параллельным выполнением делегируется объекту TaskQueue, и остается только проверить завершение всех заданий. Единственный интересный аспект этой версии – порядок определения заданий:

- вызывая функцию spider(), мы передаем ей свою функцию обратного вызова;
- функция обратного вызова проверяет завершение всех заданий, выполняемых этой функцией spiderLinks(). Если это так, вызывается функция, переданная в вызов spiderLinks();
- по завершении задания вызывается функция обратного вызова done(), которая гарантирует запуск следующих заданий из очереди.

После внесения этих небольших изменений можно вновь запустить модуль spider. На этот раз одновременно будет выполняться не более двух загрузок.

Библиотека async

Если вернуться к рассмотренным ранее шаблонам управления выполнением, можно заметить, что все они могут использоваться как основа для разработки более общих

решений. Например, алгоритм неограниченного параллельного выполнения можно заключить в функцию, которая получает список заданий, выполняет их параллельно и вызывает заданную функцию обратного вызова, после того как все они будут завершены. Обертывание алгоритмов управления выполнением функциями обеспечивает более простой декларативный способ определения асинхронного управления выполнением, что и осуществляет библиотека (https://npmjs.org/package/async). Библиотека `async` – очень популярный инструмент управления асинхронным кодом на платформе Node.js и в JavaScript в целом. Она включает ряд функций, которые значительно упрощают выполнение набора заданий в различных конфигурациях, и несколько вспомогательных элементов для асинхронной работы с коллекциями. Несмотря на существование других аналогичных библиотек, библиотека `async` благодаря своей популярности является стандартом де-факто платформы Node.js.

Перейдем сразу же к демонстрации ее возможностей.

Последовательное выполнение

Библиотека `async` наиболее полезна при реализации сложных асинхронных потоков выполнения, но одной из проблем ее применения является правильный выбор вспомогательной функции для конкретной задачи. Например, для организации последовательного выполнения предлагает набор из 20 различных функций, включая: `eachSeries()`, `mapSeries()`, `filterSeries()`, `rejectSeries()`, `reduce()`, `reduceRight()`, `detectSeries()`, `concatSeries()`, `series()`, `whilst()`, `doWhilst()`, `until()`, `doUntil()`, `forever()`, `waterfall()`, `compose()`, `seq()`, `applyEachSeries()`, `iterator()` и `timesSeries()`.

Правильный выбор функции позволяет получить компактный и удобочитаемый код, но требует определенного опыта и навыков. В примерах ниже будут охвачены лишь некоторые из возможных случаев, но они должны стать прочной основой для понимания и эффективного использования остальной части библиотеки.

Для демонстрации применения библиотеки `async` на практике займемся адаптацией под нее приложения веб-паука. Начнем прямо с версии 2, рекурсивно и последовательно загружающей все ссылки.

Но прежде необходимо установить библиотеку `async` для конкретного проекта:

```
npm install async
```

Затем загрузить новую зависимость в модуле `spider.js`:

```
const async = require('async');
```

Последовательное выполнение известного набора заданий

Начнем с реорганизации функции `download()`. Как упоминалось ранее, она должна последовательно выполнять три задания:

1) загрузка содержимого URL;
2) создание нового каталога, если он еще не существует;
3) сохранение содержимого URL в файл.

Для такой организации выполнения заданий лучше всего подойдет функция `async.series()`, имеющая следующую сигнатуру:

```
async.series(tasks, [callback])
```

Она получает список заданий и функцию обратного вызова, вызываемую после завершения всех заданий. Каждое задание представлено функцией, принимающей функцию обратного вызова, которая вызывается по завершении задания:

```
function task(callback) {}
```

Удобной особенностью библиотеки async являются следование соглашению в Node.js об обратных вызовах и автоматическая передача ошибок вверх по стеку. То есть если любое из заданий передаст ошибку функции обратного вызова, библиотека async пропустит задания, оставшиеся в списке, и сразу выполнит завершающую функцию обратного вызова.

Имея это в виду, рассмотрим функцию download(), адаптированную для взаимодействия с библиотекой async:

```
function download(url, filename, callback) {
  console.log(`Downloading ${url}`);
  let body;

  async.series([
    callback => {                                        //[1]
      request(url, (err, response, resBody) => {
        if(err) {
          return callback(err);
        }
        body = resBody;
        callback();
      });
    },
    mkdirp.bind(null, path.dirname(filename)),           //[2]
    callback => {                                        //[3]
      fs.writeFile(filename, body, callback);
    }
  ], err => {                                            //[4]
    if(err) {
      return callback(err);
    }
    console.log(`Downloaded and saved: ${url}`);
    callback(null, body);
  });
}
```

Если вспомнить версию этого кода, характеризующуюся адом обратных вызовов, можно оценить организацию заданий с помощью библиотеки async. Здесь пропадает необходимость во вложенных обратных вызовах, поскольку достаточно простого линейного списка заданий, по одному для каждой асинхронной операции, которые будут последовательно выполнены библиотекой async. Вот как определяется каждое из заданий:

1) первое задание реализует загрузку содержимого URL. Кроме того, мы сохранили тело ответа в переменной замыкания (body) для использования в других заданиях;
2) второе задание создает каталог для загруженных страниц. Оно определяется как частично примененная функция mkdirp(), связанная с путем к каталогу. В результате код делается короче на несколько строк и улучшается его удобочитаемость;
3) и наконец, запись содержимого URL в файл. В этом случае мы не использовали приема частичного применения (как во втором задании), поскольку переменная body будет доступна только после завершения первого здания в серии. Но

и здесь можно сократить код на несколько строк, использовав механизм автоматического управления ошибками в библиотеке `async`, передавая функцию обратного вызова задания непосредственно в вызов `fs.writeFile()`;

4) после завершения всех заданий вызывается заключительная функция обратного вызова, переданная в `async.series()`. В данном случае выполняется обработка ошибок с последующей передачей переменной `body` в функцию `callback` функции `download()`.

В этой конкретной ситуации возможной альтернативой `async.series()` могла бы стать функция `async.waterfall()`, которая также последовательно выполняет задания, но, кроме этого, позволяет передавать результаты текущего задания на вход следующего. В данном случае это можно было бы использовать для передачи переменной `body` во все задания последовательности. Попробуйте самостоятельно реализовать ту же функцию в обычном каскадном стиле, а затем оцените различия.

Последовательный перебор

В предыдущем разделе было продемонстрировано последовательное выполнение известного заранее набора заданий с помощью функции `async.series()`. Этот же способ можно было бы использовать для реализации функции `spiderLinks()` второй версии веб-паука, но библиотека `async` предлагает вспомогательную функцию `async.eachSeries()` для обхода коллекции, которая лучше подходит в этом конкретном случае. Воспользуемся ею для повторной реализации функции `spiderLinks()` (версия 2, последовательная загрузка):

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if(nesting === 0) {
    return process.nextTick(callback);
  }

  const links = utilities.getPageLinks(currentUrl, body);
  if(links.length === 0) {
    return process.nextTick(callback);
  }

  async.eachSeries(links, (link, callback) => {
    spider(link, nesting - 1, callback);
  }, callback);
}
```

Если сравнить этот код, использующий библиотеку `async`, с реализацией на обычном JavaScript, преимущество библиотеки `async` станет заметным, с точки зрения организации и удобочитаемости кода.

Параллельное выполнение

Библиотека `async` содержит множество функций для работы с параллельными потоками выполнения, таких как `each()`, `map()`, `filter()`, `reject()`, `detect()`, `some()`, `every()`, `concat()`, `parallel()`, `applyEach()` и `times()`. Они следуют той же логике, что и рассмотренные выше функции для последовательного выполнения, с той разницей, что выполняют задания параллельно.

Для демонстрации реализуем версию 3 веб-паука с применением одной из этих функций, выполняющую загрузку с помощью неограниченного количества параллельных потоков.

Взяв за основу последовательную версию функции `spiderLinks()`, использованную ранее, мы легко можем адаптировать ее для параллельной обработки:

```
function spiderLinks(currentUrl, body, nesting, callback) {
  // ...
  async.each(links, (link, callback) => {
    spider(link, nesting - 1, callback);
  }, callback);
}
```

Эта функция ничем не отличается от функции последовательной загрузки, за исключением использования `async.each()` вместо `async.eachSeries()`. Это наглядно демонстрирует возможности абстрагирования асинхронного потока с помощью таких библиотек, как `async`. Код больше не привязан к конкретному потоку выполнения, поскольку в нем отсутствуют строки, предназначенные для этого. Он содержит только логику приложения.

Ограниченное параллельное выполнение

Если задаться вопросом, может ли библиотека `async` ограничивать параллельную обработку заданий, ответ на этот вопрос будет утвердительным: да, может! Для этого предназначено несколько функций, а именно `eachLimit()`, `mapLimit()`, `parallelLimit()`, `queue()` и `cargo()`.

Попробуем использовать одну из них для реализации четвертой версии веб-паука, выполняющей загрузку ссылок параллельно, с ограничением одновременно действующих потоков. К счастью, библиотека `async` содержит функцию `async.queue()`, аналогичную классу `TaskQueue`, разработанному ранее в этой же главе. Функция `async.queue()` создает новую очередь, использующую функцию `worker()` для выполнения набора заданий с заданным ограничением `concurrency`:

```
const q = async.queue(worker, concurrency);
```

Функция `worker()` принимает запускаемое задание (`task`) и функцию (`callback`) для вызова после завершения задания:

```
function worker(task, callback)
```

Следует заметить, что заданием в данном случае может быть все, что угодно, не только функция. Функция `worker` автоматически выбирает наиболее подходящий способ выполнения задания. Добавлять новые задания в очередь можно вызовом метода `q.push(task, callback)`. Функция обратного вызова, связанная с заданием, будет вызвана функцией `worker` после завершения выполнения задания.

Теперь еще раз изменим код реализации параллельного выполнения с глобальным ограничением, применив функцию `async.queue()`. Прежде всего создадим очередь:

```
const downloadQueue = async.queue((taskData, callback) => {
  spider(taskData.link, taskData.nesting - 1, callback);
}, 2);
```

Код получился очень простым. Здесь создается новая очередь с ограничением параллельной обработки, равным 2. Роль функции `worker` играет стрелочная функция, которая просто вызывает функцию `spider()` и передает ей данные, соответствующие заданию. Затем реализуем функцию `spiderLinks()`:

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if(nesting === 0) {
    return process.nextTick(callback);
  }
  const links = utilities.getPageLinks(currentUrl, body);
  if(links.length === 0) {
    return process.nextTick(callback);
  }
  const completed = 0, hasErrors = false;
  links.forEach(function(link) {
    const taskData = {link: link, nesting: nesting};
    downloadQueue.push(taskData, err => {
      if(err) {
        hasErrors = true;
        return callback(err);
      }
      if(++completed === links.length&& !hasErrors) {
        callback();
      }
    });
  });
}
```

Этот код должен выглядеть знакомым, поскольку он практически повторяет реализацию с использованием объекта `TaskQueue`. В данном случае также важно отметить, как производится добавление нового задания в очередь. Вместе с заданием мы передаем функцию обратного вызова, проверяющую завершение всех заданий для текущей страницы и вызывающую заключительную функцию обратного вызова.

Благодаря функции `async.queue()` мы с легкостью смогли повторить функционал объекта `TaskQueue`, что еще раз демонстрирует возможность с помощью библиотеки `async` избежать реализации шаблонов асинхронного управления выполнением с нуля, сократив усилия на разработку и объем требуемого кода.

Итоги

В начале этой главы мы сказали, что программирование на платформе Node.js может быть затруднено из-за асинхронной природы Node.js, особенно это касается тех, кто раньше разрабатывал программы на других платформах. Однако в этой главе мы показали, что асинхронный программный интерфейс можно подчинить своим желаниям различными способами, начав с примера использования простого JavaScript, ставшего основой для анализа более сложных методов. Затем мы рассмотрели имеющиеся разнообразные средства, обеспечивающие решение большинства возникающих проблем, вместе с выбором стиля программирования на любой вкус. В качестве примера была выбрана библиотека `async`, упрощающая реализацию наиболее широко распространенных способов выполнения.

Эта глава служит введением в более продвинутые технологии, такие как объекты `Promise` и генераторы, использованию которых будет в основном посвящена следующая глава. Зная все эти методы, вы сможете выбирать лучшие решения для конкретных задач или же совместно использовать некоторые из них в одном и том же проекте.

Глава 4

Шаблоны асинхронного выполнения с использованием спецификации ES2015, и не только

В предыдущей главе мы узнали, как писать асинхронный код, используя обратные вызовы, и их отрицательное влияние на код, выражающееся в создании таких проблем, как **ад обратных вызовов**. Обратные вызовы являются основой асинхронного программирования на JavaScript и в Node.js, но за прошедшие годы появились альтернативные подходы, позволяющие упростить работу с асинхронным кодом.

В этой главе рассматриваются самые популярные из таких альтернатив, объекты `Promise` и генераторы, а также новейший синтаксис **async await**, который будет введен в JavaScript как часть спецификации ECMAScript 2017.

Мы увидим, как эти альтернативы могут упростить управление асинхронными потоками. И наконец, сравним все эти подходы, выявив плюсы и минусы каждого из них, чтобы иметь возможность разумно подойти к выбору подхода, наилучшим образом соответствующего требованиям проекта на платформе Node.js.

Promise

В предыдущих главах уже упоминалось, что **стиль передачи продолжения** (Continuation Passing Style, CPS) не является единственным способом реализации асинхронного кода. В действительности экосистема JavaScript предлагает интересные альтернативы традиционному шаблону обратных вызовов. Одной из самых распространенных таких альтернатив является объект `Promise`, которому уделяется все больше внимания, особенно сейчас, когда он стал частью спецификации ECMAScript 2015 и обрел встроенную поддержку, начиная с версии 4, платформы Node.js.

Что представляет собой объект Promise?

Выражаясь простым языком, объект `Promise`[1] является абстракцией, позволяющей функциям возвращать объект `Promise`, представляющий конечный результат асинхронной операции. Мы говорим, что объект `Promise` **ожидает**, если асинхронная операция еще не завершилась, **выполнен** – если операция завершилась успешно, и **отклонен** – если возникла ошибка. После того как объект `Promise` будет выполнен или отклонен, он считается **установившимся**.

Чтобы получить результат выполнения или ошибку (*причину*), вызвавшую отклонение, можно использовать метод `then()` объекта `Promise`:

```
promise.then([onFulfilled], [onRejected])
```

Здесь `onFulfilled()` – это функция, которой передается результат выполнения асинхронной операции, а `onRejected()` – функция, которой передается причина отклонения. Обе функции являются необязательными.

Чтобы получить представление, как применение объектов `Promise` может изменить код, рассмотрим следующий фрагмент кода:

```
asyncOperation(arg, (err, result) => {
  if(err) {
    //обработка ошибки
  }
  //работа с результатом
});
```

Объекты `Promise` позволяют преобразовать этот типичный CPS-код в более структурированный и элегантный код, например:

```
asyncOperation(arg)
  .then(result => {
    //работа с результатом
  }, err => {
    //обработка ошибки
  });
```

Одним из важнейших свойств метода `then()` является синхронный возврат другого объекта `Promise`. Если любая из функций – `onFulfilled()` или `onRejected()` – вернет значение *x*, метод `then()` вернет один из следующих объектов `Promise`:

- выполненный со значением *x*, если *x* является значением;
- выполненный с объектом *x*, где *x* является объектом `Promise` или thenable-объектом[2];
- отклоненный с причиной отклонения *x*, где *x* является объектом `Promise` или thenable-объектом.

thenable-объект – это `Promise`-подобный объект, имеющий метод `then()`. Этот термин используется для обозначения объекта, фактическая реализация которого *отличается* от реализации `Promise`.

---

[1] Имя объекта Promise переводится на русский язык как «обещание», то есть функция, возвращая объект `Promise`, «обещает» выполнить асинхронное задание некогда в будущем. – *Прим. ред.*

[2] Дословно: «объект с методом `then()`». Этот термин используется в спецификации ECMAScript 6. – *Прим. ред.*

Эта особенность позволяет создавать цепочки из объектов `Promise`, облегчая объединение и компоновку асинхронных операций в различных конфигурациях. Кроме того, если не указывается обработчик `onFulfilled()` или `onRejected()`, результат или причина отклонения автоматически направляется следующему объекту `Promise` в цепочке. Это дает возможность, например, автоматически передавать ошибку вдоль всей цепочки, пока она не будет перехвачена обработчиком `onRejected()`. Составление цепочек объектов `Promise` делает последовательное выполнение заданий тривиальной операцией:

```
asyncOperation(arg)
  .then(result1 => {
    //возвращает другой объект Promise
    return asyncOperation(arg2);
  })
  .then(result2 => {
    //возвращает значение
    return 'done';
  })
  .then(undefined, err => {
    // здесь обрабатываются все возникшие в цепочке ошибки
  });
```

Схема на рис. 4.1 иллюстрирует другую точку зрения на работу цепочки объектов `Promise`:

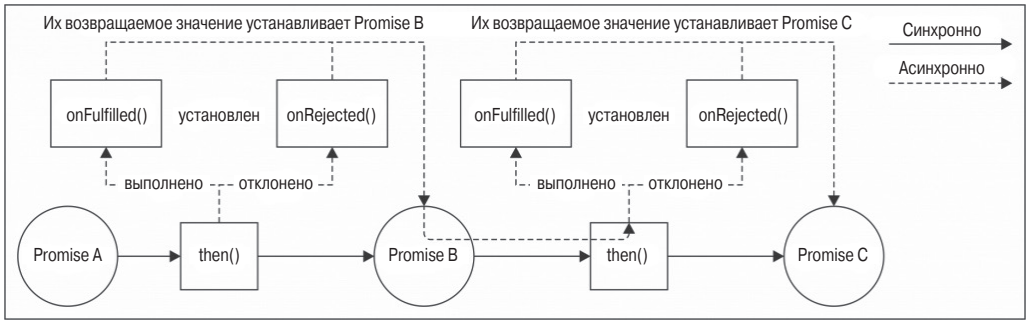

Рис. 4.1 ❖ Цепочка объектов Promise

Другим важным свойством объектов `Promise` является гарантированный асинхронный вызов функций `onFulfilled()` и `onRejected()`, даже при синхронном выполнении, как в предыдущем примере, где последняя функция `then()` в цепочке возвращает строку `'done'`. Такая модель поведения защищает код от непреднамеренного высвобождения Залго (*глава 2 «Основные шаблоны Node.js»*), что без дополнительных усилий делает асинхронный код более последовательным и надежным.

А теперь самое интересное: если в обработчике `onFulfilled()` или `onRejected()` возбудить исключение (оператором `throw`), возвращаемый методом `then()` объект `Promise` автоматически будет отклонен с исключением в качестве причины отказа. Это огромное преимущество перед CPS, потому что исключение автоматически будет передаваться вдоль по цепочке, а это означает, что можно использовать оператор `throw`.

Исторически сложилось, что существует множество библиотек, реализующих объекты `Promise`, большинство которых не совместимо друг с другом, что препятствует созданию `then`-цепочек из объектов `Promise`, созданных разными библиотеками.

Сообщество JavaScript провело сложную работу по преодолению этого ограничения, в результате была создана спецификация **Promises/A+**. Эта спецификация детально описывает поведение метода `then` и служит основой, обеспечивающей возможность взаимодействий между объектами `Promise` из различных библиотек.

Более подробные сведения о спецификации Promises/A+ можно найти на ее официальном сайте https://promisesaplus.com.

Реализации Promises/A+

Как в JavaScript, так и в Node.js есть несколько библиотек, реализующих спецификацию Promises/A+. Ниже перечислены наиболее популярные из них:

- Bluebird (https://npmjs.org/package/bluebird);
- Q (https://npmjs.org/package/q);
- RSVP (https://npmjs.org/package/rsvp);
- Vow (https://npmjs.org/package/vow);
- When.js (https://npmjs.org/package/when);
- объекты `Promise` из ES2015.

По существу, они отличаются только наборами дополнительных возможностей, не предусмотренных стандартом Promises/A+. Как упоминалось выше, этот стандарт определяет модель поведения метода `then()` и процедуру разрешения объекта `Promise`, но не регламентирует других функций, например порядка создания объекта `Promise` на основе асинхронной функции с обратным вызовом.

В примерах ниже мы будем использовать методы, поддерживаемые объектами `Promise` стандарта ES2015, поскольку они доступны в Node.js, начиная с версии 4, и не требуют подключения внешних библиотек.

Для справки ниже перечислены методы объектов `Promise`, определяемые стандартом ES2015.

Конструктор (new Promise(function(resolve, reject) {})): создает новый объект `Promise`, который разрешается или отклоняется в зависимости от функции, переданной в аргументе. Конструктору можно передать следующие аргументы:

- `resolve(obj)`: позволяет разрешить объект `Promise` и вернуть результат `obj`, если `obj` является значением. Если `obj` является другим объектом `Promise` или thenable-объектом, результатом станет результат выполнения `obj`;
- `reject(err)`: отклоняет объект `Promise` с указанной причиной `err`. В соответствии с соглашением `err` должен быть экземпляр `Error`.

Статические методы объекта `Promise`:

- `Promise.resolve(obj)`: возвращает новый объект `Promise`, созданный из thenable-объекта, если `obj` – thenable-объект, или значение, если `obj` – значение;
- `Promise.all(iterable)`: создает объект `Promise`, который разрешается результатами выполнения, если все элементы итерируемого объекта `iterable` выполнились, и отклоняется при первом же отклонении любого из элементов. Любой элемент итерируемого объекта может быть объектом `Promise`, универсальным thenable-объектом или значением;
- `Promise.race(iterable)`: возвращает объект `Promise`, разрешаемый или отклоняемый, как только разрешится или будет отклонен хотя бы один из объектов

Promise в итерируемом объекте iterable, со значением или причиной этого объекта Promise.

Методы экземпляра Promise:

- promise.then(onFulfilled, onRejected): основной метод объекта Promise. Его модель поведения совместима со стандартом Promises/A+, упомянутым выше;
- promise.catch(onRejected): удобная синтаксическая конструкция, заменяющая promise.then(undefined, onRejected).

Стоит отметить, что некоторые реализации предлагают другой асинхронный механизм – механизм **отложенных вычислений**. Мы не будем рассматривать его, поскольку он не является частью стандарта ES2015, но желающие узнать о нем больше могут обратиться к документации для Q (https://github.com/kriskowal/q#using-deferreds) или When.js (https://github.com/cujojs/when/wiki/Deferred).

Перевод функций в стиле Node.js на использование объектов Promise

В JavaScript не все асинхронные функции и библиотеки поддерживают объекты Promise изначально. Обычно типичные функции, основанные на обратных вызовах, требуется преобразовать так, чтобы они возвращали объекты Promise. Этот процесс называется **переводом на использование объектов Promise**.

К счастью, соглашения об обратных вызовах, используемые на платформе Node.js, позволяют создавать функции, способные переводить любые функции в стиле Node.js на использование объектов Promise. Это несложно осуществить с помощью конструктора объекта Promise. Создадим новую функцию promisify() и добавим ее в модуль utilities.js (чтобы ее можно было использовать в приложении веб-паука):

```
module.exports.promisify = function(callbackBasedApi) {
  return function promisified() {
    const args = [].slice.call(arguments);
    return new Promise((resolve, reject) => {        //[1]
      args.push((err, result) => {                   //[2]
        if(err) {
          return reject(err);                        //[3]
        }
        if(arguments.length <= 2) {                  //[4]
          resolve(result);
        } else {
          resolve([].slice.call(arguments, 1));
        }
      });
      callbackBasedApi.apply(null, args);            //[5]
    });
  }
};
```

Приведенная выше функция возвращает другую функцию – promisified(), которая является версией callbackBasedApi, возвращающей объект Promise. Вот как она работает:

1) функция promisified() создает новый объект с помощью конструктора Promise и немедленно возвращает его;

2) в функции, что передается конструктору `Promise`, мы передаем специальную функцию обратного вызова для вызова из `callbackBasedApi`. Поскольку функция обратного вызова всегда передается в последнем аргументе, мы просто добавляем ее в список аргументов (`args`) функции `promisified()`;
3) если специальная функция обратного вызова получит ошибку, объект `Promise` немедленно отклоняется;
4) в случае отсутствия ошибки осуществляется разрешение объекта `Promise` со значением или массивом значений, в зависимости от количества результатов, переданных функции обратного вызова;
5) в заключение вызывается `callbackBasedApi` с созданным списком аргументов.

Большинство реализаций поддерживает вспомогательный метод преобразования типичных функций в стиле Node.js в функции, возвращающие объекты `Promise`. Например, библиотека Q содержит функции `Q.denodeify()` и `Q.nbind()`, библиотека Bluebird имеет `Promise.promisify()`, а When.js содержит `node.lift()`.

Последовательное выполнение

Теперь, после знакомства с теорией, можно вернуться к приложению веб-паука и переписать его с использованием объектов `Promise`. Начнем с версии 2, загружающей ссылки последовательно.

Первым делом нужно в модуле `spider.js` загрузить реализацию объектов `Promise` (будет использована позже), а затем перевести функции на использование объектов `Promise`:

```
const utilities = require('./utilities');

const request = utilities.promisify(require('request'));
const mkdirp = utilities.promisify(require('mkdirp'));
const fs = require('fs');
const readFile = utilities.promisify(fs.readFile);
const writeFile = utilities.promisify(fs.writeFile);
```

Теперь преобразуем функцию `download()`:

```
function download(url, filename) {
  console.log(`Downloading ${url}`);
  let body;
  return request(url)
    .then(response => {
      body = response.body;
      return mkdirp(path.dirname(filename));
    })
    .then(() => writeFile(filename, body))
    .then(() => {
      console.log(`Downloaded and saved: ${url}`);
      return body;
    });
}
```

Важно отметить, что мы также зарегистрировали обработчик `onRejected()` для объекта `Promise`, возвращаемого функцией `readFile()`, чтобы обработать ситуацию, когда страница не была загружена (файла не существует). Обратите также внимание, как используется `throw` для возбуждения ошибки в обработчике.

После преобразования функции `spider()` можно изменить ее основной вызов:

```
spider(process.argv[2], 1)
  .then(() => console.log('Download complete'))
  .catch(err => console.log(err));
```

Обратите внимание: здесь впервые использован синтаксический сахар `catch` для обработки ошибок в функции `spider()`. Взглянув на приведенный выше код, можно с удовлетворением заметить, что в нем полностью отсутствует логика передачи ошибок, которая была необходима при использовании обратных вызовов. Это огромное преимущество, поскольку позволяет значительно сократить объем стереотипного кода и исключает возможность пропуска асинхронных ошибок.

Теперь, чтобы получить законченную версию 2 приложения веб-паука, осталось только реализовать функцию `spiderLinks()`, чем мы и займемся далее.

Последовательные итерации

На данный момент, на примере приложения веб-паука, мы рассмотрели объекты `Promise` и приемы их использования для создания простой элегантной реализации последовательного потока выполнения. Но этот код обеспечивает выполнение лишь известного заранее набора асинхронных операций. Поэтому, чтобы восполнить пробелы в исследовании последовательного выполнения, нам нужно разработать фрагмент, реализующий итерации с помощью объектов `Promise`. И снова прекрасным примером для демонстрации станет функция `spiderLinks()` из версии 2 веб-паука.

Реализуем этот подход:

```
function spiderLinks(currentUrl, body, nesting) {
  let promise = Promise.resolve();
  if(nesting === 0) {
    return promise;
  }
  const links = utilities.getPageLinks(currentUrl, body);
  links.forEach(link => {
    promise = promise.then(() => spider(link, nesting - 1));
  });
  return promise;
}
```

Для асинхронного обхода всех ссылок на веб-странице нужно динамически создать цепочку объектов `Promise`.

1. Начнем с определения «пустого» объекта `Promise`, разрешаемого как `undefined`. Он будет служить началом цепочки.
2. Затем в цикле присвоим переменной `promise` новый объект `Promise`, полученный вызовом метода `then()` предыдущего объекта `Promise` в цепочке. Это и есть шаблон асинхронных итераций с использованием объектов `Promise`.

В конце цикла переменная `promise` будет содержать объект `Promise`, который вернул последний вызов `then()` в цикле, поэтому он будет разрешен после разрешения всех объектов `Promise` в цепочке.

Теперь реализация версии 2 веб-паука с объектами `Promise` закончена, и ее можно проверить.

Последовательные итерации – шаблон

Чтобы завершить раздел, описывающий реализацию последовательного выполнения, сформулируем шаблон последовательных итераций по набору объектов `Promise`:

```
let tasks = [ /* ... */ ]
let promise = Promise.resolve();
tasks.forEach(task => {
  promise = promise.then(() => {
    return task();
  });
});
promise.then(() => {
  //Все задания выполнены
});
```

Замена цикла `forEach()` функцией `reduce()` позволяет еще больше сократить код:

```
let tasks = [ /* ... */ ]
let promise = tasks.reduce((prev, task) => {
  return prev.then(() => {
    return task();
  });
}, Promise.resolve());
promise.then(() => {
  // Все задания выволнены
});
```

Как всегда, с помощью простой адаптации этого шаблона можно поместить результаты выполнения всех заданий в массив, реализовать алгоритм отображения, создать фильтр и т. д.

Шаблон (последовательных итераций с помощью объектов `Promise`): этот шаблон динамически строит цепочку объектов `Promise` в цикле.

Параллельное выполнение

Параллельное выполнение реализуется с применением объектов `Promise` так же тривиально просто. Для этого достаточно лишь воспользоваться встроенной функцией `Promise.all()`. Эта вспомогательная функция создает еще один объект `Promise`, который выполнится, только когда выполнятся все объекты `Promise`, указанные в аргументе. Поскольку это выполнение происходит параллельно, порядок разрешения объектов `Promise` не важен.

Продемонстрируем такой подход на версии 3 веб-паука, загружающей все ссылки параллельно. Внесем изменения в функцию `spiderLinks()` для реализации параллельного выполнения с помощью объектов `Promise`:

```
function spiderLinks(currentUrl, body, nesting) {
  if(nesting === 0) {
    return Promise.resolve();
  }

  const links = utilities.getPageLinks(currentUrl, body);
  const promises = links.map(link => spider(link, nesting - 1));
```

```
  return Promise.all(promises);
}
```

Суть шаблона заключается в одновременном запуске всех заданий `spider()` в цикле `elements.map()`, помещающем в коллекцию все объекты `Promise`. На этот раз в цикле не предусмотрено ожидание завершения загрузки предыдущей ссылки перед началом следующей, все задания запускаются в цикле одновременно, одно за другим. После этого вызывается метод `Promise.all()`, возвращающий новый объект `Promise`, который перейдет в установившееся состояние после выполнения всех объектов `Promise` из массива, то есть когда завершатся все задания загрузки, что нам и требуется.

Ограниченное параллельное выполнение

К сожалению, интерфейс объектов `Promise` в стандарте ES2015 не предусматривает способа ограничить число параллельно выполняемых заданий, но мы можем воспользоваться приемом для ограничения параллельной обработки с помощью обычного JavaScript. Фактически шаблон, реализованный на основе класса `TaskQueue`, легко можно адаптировать для поддержки заданий, возвращающих объекты `Promise`. Для этого достаточно изменить метод `next()`:

```
next() {
  while(this.running < this.concurrency && this.queue.length) {
    const task = this.queue.shift();
    task().then(() => {
      this.running--;
      this.next();
    });
    this.running++;
  }
}
```

Вместо обработки задания с обратным вызовом здесь вызывается метод `then()` возвращаемого объекта `Promise`. В остальном код идентичен старой версии класса `TaskQueue`.

Вернемся к модулю `spider.js` и изменим его для поддержки новой версии класса `TaskQueue`. Во-первых, обязательно нужно создать новый экземпляр `TaskQueue`:

```
const TaskQueue = require('./taskQueue');
const downloadQueue = new TaskQueue(2);
```

Затем снова изменим функцию `spiderLinks()`. Изменения довольно просты:

```
function spiderLinks(currentUrl, body, nesting) {
  if(nesting === 0) {
    return Promise.resolve();
  }

  const links = utilities.getPageLinks(currentUrl, body);
  //следующая проверка необходима, поскольку создаваемый ниже
  //объект Promise никогда не установится в отсутствие заданий
  if(links.length === 0) {
    return Promise.resolve();
  }
```

```
  return new Promise((resolve, reject) => {
    let completed = 0;
    let errored = false;
    links.forEach(link => {
      let task = () => {
        return spider(link, nesting - 1)
        .then(() => {
          if(++completed === links.length) {
            resolve();
          }
        })
        .catch(() => {
          if (!errored) {
            errored = true;
            reject();
          }
        });
      };
      downloadQueue.pushTask(task);
    });
  });
}
```

В предыдущем коде имеется пара моментов, требующих пояснения:

- во-первых, необходимо вернуть новый объект `Promise`, созданный с помощью конструктора `Promise`. Как будет показано ниже, это позволит разрешить объект `Promise` вручную, после завершения всех заданий в очереди;
- во-вторых, обратите внимание на порядок определения задания. Здесь к объекту `Promise`, возвращаемому функцией `spider()`, подключается обработчик `onFulfilled()`, подсчитывающий количество завершенных заданий. Если это число равно числу ссылок на текущей странице, значит, обработка завершена и можно вызвать функцию `resolve()` внешнего объекта `Promise`.

Когда определяется спецификация Promises/A+, функции обратного вызова `onFulfilled()` и `onRejected()` для метода `then()` должны вызываться только один раз и не обе сразу (либо одна, либо другая). Совместимая реализация объектов `Promise` должна гарантировать, что даже при многократном разрешении или отклонении объект `Promise` изменит свое состояние только один раз.

Теперь можно опробовать версию 4 веб-паука с объектами `Promise`. Во время ее работы задания будут выполняться параллельно, но не более двух сразу.

Обратные вызовы и объекты Promise в общедоступных программных интерфейсах

Как мы узнали в предыдущих разделах, объекты `Promise` являются достойной заменой обратных вызовов. Их применение делает код более простым и наглядным. Объекты `Promise` дают заметные преимущества, но их применение требует от разработчика понимания множества нетривиальных идей, чтобы использовать объекты `Promise` правильно и умело. По этой и ряду других причин в некоторых случаях практичнее отдать предпочтение обратным вызовам.

Предположим, требуется создать библиотеку, выполняющую асинхронные операции. Какой интерфейс выбрать? Ориентированный на обратные вызовы или на объекты `Promise`? Нужно ли выбрать один из вариантов или же существует способ для совмещения их преимуществ?

Эта проблема встает перед многими популярными библиотеками, и имеются, по крайней мере, два способа ее решения, которые стоит рассмотреть, поскольку они помогают реализовать универсальный программный интерфейс.

Первый способ используется такими библиотеками, как `request`, `redis` и `mysql`. Он заключается в реализации простого программного интерфейса, основанного только на обратных вызовах и оставляющего разработчику возможность переводить библиотечные функции на использование объектов `Promise`. С этой целью некоторые библиотеки включают вспомогательные функции, осуществляющие такой перевод, но это не отменяет необходимости для разработчика подумать заранее о выборе интерфейса и выполнить необходимые преобразования.

Второй способ более прозрачный. Он также предлагает программный интерфейс, ориентированный на обратные вызовы, но делает аргумент с функцией обратного вызова необязательным. Всякий раз, когда в аргументе передается функция обратного вызова, библиотечная функция работает в обычном режиме, выполняя обратный вызов при успешном завершении или ошибке. Если функция обратного вызова не передается, библиотечная функция немедленно возвращает объект `Promise`. Такой подход эффективно сочетает обратные вызовы и объекты `Promise`, позволяя разработчику выбирать нужный интерфейс в момент вызова, без необходимости заранее переводить библиотечные функции на использование объектов `Promise`. Этот подход используют многие библиотеки, в частности библиотеки `mongoose` и `sequelize`.

Рассмотрим простую реализацию второго подхода на примере. Предположим, что требуется реализовать модуль заглушки, выполняющий деление асинхронно:

```
module.exports = function asyncDivision (dividend, divisor, cb) {
  return new Promise((resolve, reject) => {                // [1]

    process.nextTick(() => {
      const result = dividend / divisor;
      if (isNaN(result) || !Number.isFinite(result)) {
        const error = new Error('Invalid operands');
        if (cb) { cb(error); }                             // [2]
          return reject(error);
        }
        if (cb) { cb(null, result); }                      // [3]
          resolve(result);
    });

  });
};
```

Код модуля очень прост, но некоторые его детали нуждаются в пояснении.

- Во-первых, он возвращает новый объект `Promise`, созданный с помощью конструктора `Promise`. Вся логика сосредоточена в функции, передаваемой конструктору в аргументе.
- В случае ошибки объект `Promise` отклоняется, но если была передана функция обратного вызова, она вызывается для передачи ошибки дальше.

- После получения результата объект `Promise` разрешается. И снова, если была передана функция обратного вызова, с ее помощью возвращается результат.

А теперь продемонстрируем использование этого модуля с применением обратного вызова и объекта `Promise`:

```
// использование с обратным вызовом
asyncDivision(10, 2, (error, result) => {
  if (error) {
    return console.error(error);
  }
  console.log(result);
});

// использование с объектом Promise
asyncDivision(22, 11)
    .then(result => console.log(result))
    .catch(error => console.error(error));
```

Как видите, стоит приложить лишь немного усилий – и разработчик, использующий данный модуль, сможет выбирать стиль, наилучшим образом соответствующий его потребностям, без необходимости выполнять преобразование функций, когда ему понадобится использовать объекты `Promise`.

Генераторы

Стандарт ES2015 вводит еще один механизм, который, помимо прочего, можно использовать для упрощения асинхронным выполнением приложений на платформе Node.js. Речь идет о **генераторах**, также называемых **полусопрограммами** (semi-coroutines). Это обобщенная разновидность подпрограмм, которые могут иметь различные точки входа. Обычная функция (подпрограмма) имеет только одну точку входа, которая соответствует ее началу. Генераторы подобны функциям, но их выполнение может быть приостановлено (с помощью оператора `yield`) и возобновлено позже. Генераторы особенно полезны при реализации итераторов, и это должно показаться знакомым, поскольку ранее уже упоминалось, что итераторы можно использовать для реализации важных шаблонов асинхронного выполнения, таких как последовательное и ограниченное параллельное выполнение.

Введение в генераторы

Перед тем как перейти к практике управления асинхронным выполнением с использованием генераторов, познакомимся с некоторыми основными понятиями. Начнем с синтаксиса. Функция-генератор может быть объявлена путем добавления символа * (звездочка) после ключевого слова `function`:

```
function* makeGenerator() {
  //тело
}
```

Выполнение функции `makeGenerator()` можно приостановить с помощью ключевого слова `yield` и вернуть вызвавшему объекту некоторое значение:

```
function* makeGenerator() {
  yield 'Hello World';
  console.log('Re-entered');
}
```

В этом примере оператор `yield` приостанавливает функцию и возвращает строку `Hello World`. Когда выполнение генератора возобновится, работа продолжится с оператора `console.log('Re-entered')`.

По сути, функция `makeGenerator()` является фабрикой, которая при вызове возвращает новый объект генератора:

```
const gen = makeGenerator();
```

Наиболее важным методом объекта генератора является `next()`. Он запускает/возобновляет генератор и возвращает объект следующего вида:

```
{
  value: <полученное значение>
  done: <true, если выполнение завершено>
}
```

Этот объект содержит возвращаемое генератором значение (`value`) и флаг (`done`), указывающий, что генератор завершил свое выполнение.

Простой пример

Для демонстрации применения генераторов создадим новый модуль `fruitGenerator.js`:

```
function* fruitGenerator() {
  yield 'apple';
  yield 'orange';
  return 'watermelon';
}

const newFruitGenerator = fruitGenerator();
console.log(newFruitGenerator.next());    //[1]
console.log(newFruitGenerator.next());    //[2]
console.log(newFruitGenerator.next());    //[3]
```

Предыдущий код выведет следующее:

```
{ value: 'apple', done: false }
{ value: 'orange', done: false }
{ value: 'watermelon', done: true }
```

Ниже кратко поясняется происшедшее:

- первый вызов `newFruitGenerator.next()` запускает генератор, продолжающий выполняться до первой команды `yield`, которая приостанавливает генератор и возвращает вызвавшему объекту значение `apple`;
- второй вызов `newFruitGenerator.next()` возобновляет выполнение генератора со второй команды `yield`, которая снова приостанавливает генератор и возвращает значение `orange`;
- последний вызов `newFruitGenerator.next()` заставил генератор выполнить последнюю команду, оператор `return`, которая вернула значение `watermelon` и присвоила свойству `done` объекта `result` значение `true`.

Генераторы в роли итераторов

Чтобы понять, как генераторы могут пригодиться при реализации итераторов, создадим один из них. Для этого поместим в новый модуль `iteratorGenerator.js` следующий код:

```
function* iteratorGenerator(arr) {
  for(let i = 0; i <arr.length; i++) {
    yield arr[i];
  }
}

const iterator = iteratorGenerator(['apple', 'orange', 'watermelon']);
let currentItem = iterator.next();
while(!currentItem.done) {
  console.log(currentItem.value);
  currentItem = iterator.next();
}
```

Он должен вывести список элементов массива:

```
apple
orange
watermelon
```

В этом примере каждый вызов метода `iterator.next()` возобновляет цикл `for` в генераторе, что приводит к выполнению очередного шага с возвратом следующего элемента массива. Этот пример демонстрирует, что генераторы сохраняют состояние между вызовами. При возобновлении выполнения генератор восстанавливает состояние цикла и всех переменных на момент, когда он был приостановлен.

Передача значений обратно в генератор

В заключение исследования основных возможностей генераторов рассмотрим порядок передачи генераторам значений. Это осуществляется очень просто. Достаточно просто передать аргумент в метод `next()`, значение которого станет возвращаемым значением оператора `yield`.

Для демонстрации создадим новый простой модуль:

```
function* twoWayGenerator() {
  const what = yield null;
  console.log('Hello ' + what);
}

const twoWay = twoWayGenerator();
twoWay.next();
twoWay.next('world');
```

Предыдущий код выведет `Hello world`. Это означает, что произошло следующее:

- после первого вызова метода `next()` генератор дойдет до первого оператора `yield` и приостановится;
- после вызова `next('world')` генератор возобновит работу с места, где он был приостановлен, то есть с инструкции `yield`, но сейчас уже имеется значение, переданное генератору. Это значение присваивается переменной `what`. Затем генератор выполнит инструкцию `console.log()` и завершит работу.

Аналогично можно заставить генератор возбудить исключение. Это реализуется с помощью метода `throw` генератора, как показано в следующем примере:

```
const twoWay = twoWayGenerator();
twoWay.next();
twoWay.throw(new Error());
```

Этот фрагмент возбудит исключение в функции `twoWayGenerator()` в момент, когда оператор `yield` вернет значение внутри генератора. Это исключение ничем не будет отличаться от исключения, возбужденного внутри генератора, то есть его можно перехватить и обработать как любое другое исключение, с помощью блока `try...catch`.

Асинхронное выполнение с генераторами

На первый взгляд, утверждение, что генераторы могут помочь с выполнением асинхронных операций, может показаться странным. Но мы можем подтвердить его, создав специальную функцию, принимающую генератор в аргументе и позволяющую использовать асинхронный код внутри генератора. Эта функция позаботится о возобновлении генератора после завершения асинхронной операции. Назовем эту функцию `asyncFlow()`:

```
function asyncFlow(generatorFunction) {
  function callback(err) {
    if(err) {
      return generator.throw(err);
    }
    const results = [].slice.call(arguments, 1);
    generator.next(results.length> 1 ? results : results[0]);
  }
  const generator = generatorFunction(callback);
  generator.next();
}
```

Функция выше принимает генератор, создает его экземпляр и сразу же запускает:

```
const generator = generatorFunction(callback);
generator.next();
```

Также `generatorFunction()` принимает специальную функцию обратного вызова, которая вызывает `generator.throw()` при появлении ошибок или возобновляет выполнение генератора, передавая обратно результаты, полученные от функции обратного вызова:

```
if(err) {
  return generator.throw(err);
}
const results = [].slice.call(arguments, 1);
generator.next(results.length> 1 ? results : results[0]);
```

Для демонстрации возможностей этой простой функции создадим новый модуль `clone.js`, который без особых на то причин будет клонировать себя. Вставим в него вновь созданную функцию `asyncFlow()` и добавим основной код:

```
const fs = require('fs');
const path = require('path');
```

```
asyncFlow(function* (callback) {
  const fileName = path.basename(__filename);
  const myself = yield fs.readFile(fileName, 'utf8', callback);
  yield fs.writeFile(`clone_of_${filename}`, myself, callback);
  console.log('Clone created');
});
```

Самое примечательное в этом коде, что с помощью функции `asyncFlow()` удалось написать асинхронный код, применив линейный подход, который обычно используется для блокирующих функций! Чуть ниже мы поясним, в чем заключается необычность полученного кода. Функция обратного вызова, переданная в каждую асинхронную функцию, будет возобновлять работу генератора сразу после завершения асинхронной операции. Ничего сложного, но результат впечатляет.

Эта технология имеет две другие разновидности: одна – с использованием объектов `Promise` и другая – с применением преобразователей.

Преобразователь, используемый в управлении потоком выполнения на основе генератора, – это просто функция, которая получает все аргументы исходной функции, за исключением функции обратного вызова, и возвращает другую функцию, которая принимает только функцию обратного вызова. Например, версия `fs.readFile()` с преобразователем могла бы иметь следующий вид:

```
function readFileThunk(filename, options) {
  return function(callback){
    fs.readFile(filename, options, callback);
  }
}
```

Преобразователи и объекты `Promise` позволяют создавать генераторы, не нуждающиеся в функциях обратного вызова. Например, версия `asyncFlow()` с преобразователем могла бы иметь следующий вид:

```
function asyncFlowWithThunks(generatorFunction) {
  function callback(err) {
    if(err) {
      return generator.throw(err);
    }
    const results = [].slice.call(arguments, 1);
    const thunk = generator.next(results.length> 1 ? results :
                  results[0]).value;
    thunk && thunk(callback);
  }
  const generator = generatorFunction();
  const thunk = generator.next().value;
  thunk && thunk(callback);
}
```

Хитрость заключается в том, чтобы прочитать возвращаемое значение `generator.next()`, содержащее преобразователь. Затем вызвать сам преобразователь, внедрив специальную функцию обратного вызова. Просто! Это позволяет написать следующий код:

```
asyncFlowWithThunks(function* () {
  const fileName = path.basename( filename);
```

```
  const myself = yield readFileThunk( filename, 'utf8');
  yield writeFileThunk(`clone_of_${fileName}`, myself);
  console.log("Clone created");
});
```

Точно так же можно реализовать версию функции `asyncFlow()`, принимающую объект `Promise` в `yield`. Мы оставляем это вам в качестве самостоятельного упражнения, поскольку такая реализация требует минимальных изменений в функции `asyncFlowWithThunks()`. Можно также реализовать функцию `asyncFlow()`, которая принимает в `yield` и объекты Promise, и преобразователи, используя тот же принцип.

Управление выполнением на основе генераторов с использованием решения co

Наверное, вы уже догадались, что экосистема Node.js поддерживает несколько решений для обработки асинхронных потоков с помощью генераторов, например одно из старейших решений называется `suspend` (https://npmjs.org/package/suspend) – оно поддерживает объекты `Promise`, преобразователи, обратные вызовы в стиле Node.js, а также простые обратные вызовы. Кроме того, большинство библиотек поддержки `Promise`, упоминавшихся выше в этой главе, предоставляет вспомогательные функции для использования объектов `Promise` с генераторами.

Все эти решения основаны на тех же принципах, что были продемонстрированы на примере функции `asyncFlow()`, то есть мы могли бы воспользоваться одним из них, а не писать свое собственное.

В примерах в этом разделе будет использоваться решение `co` (https://npmjs.org/package/co). Оно поддерживает несколько типов сущностей, передаваемых через `yield`:

- преобразователи;
- объекты `Promise`;
- массивы (параллельное выполнение);
- объекты (параллельное выполнение);
- генераторы (делегирование);
- функции генераторов (делегирование).

Решение `co` также имеет собственную экосистему пакетов:

- веб-фреймворки, наиболее популярным из них является `koa` (https://npmjs.org/package/koa);
- библиотеки реализации конкретных шаблонов выполнения;
- библиотеки для популярных программных интерфейсов, придающие им поддержку `co`.

Мы используем решение `co` для повторной реализации приложения веб-паука с использованием генераторов.

Чтобы привести функции в стиле Node.js к виду, позволяющему использовать преобразователи, мы используем небольшую библиотеку `thunkify` (https://npmjs.org/package/thunkify).

Последовательное выполнение

Начнем практическое исследование генераторов и решения `co` с изменения версии 2 приложения веб-паука. Прежде всего необходимо загрузить зависимости и сгенерировать версии функций, пригодные для работы с преобразователями. Все это мы поместим в начало модуля `spider.js`:

```
const thunkify = require('thunkify');
const co = require('co');

const request = thunkify(require('request'));
const fs = require('fs');
const mkdirp = thunkify(require('mkdirp'));
const readFile = thunkify(fs.readFile);
const writeFile = thunkify(fs.writeFile);
const nextTick = thunkify(process.nextTick);
```

Здесь явно видно сходство с кодом, приводившимся в этой главе выше, где добавляли в программный интерфейс поддержку объектов `Promise`. В этой связи интересно отметить, что при использовании версии функций с поддержкой объектов `Promise` вместо версий с поддержкой преобразователей код останется точно таким же благодаря тому, что решение `co` поддерживает и преобразователи, и объекты `Promise`. Фактически при желании можно было бы использовать преобразователи и объекты `Promise` в одном и том же приложении и даже в одном и том же генераторе. Это огромное преимущество с точки зрения гибкости, поскольку позволяет управление потоком выполнения на основе генераторов с любым решением, уже имеющимся в нашем распоряжении.

А теперь преобразуем функцию `download()` в генератор:

```
function* download(url, filename) {
  console.log(`Downloading ${url}`);
  const response = yield request(url);
  const body = response[1];
  yield mkdirp(path.dirname(filename));
  yield writeFile(filename, body);
  console.log(`Downloaded and saved ${url}`);
  return body;
}
```

При использовании генераторов и решения `co` реализация функции `download()` становится тривиально простой. Для этого нужно лишь преобразовать ее в функцию генератора и использовать `yield` вместо вызовов асинхронных функций (как преобразователей).

Далее обратим внимание на функцию `spider()`:

```
function* spider(url, nesting) {
  const filename = utilities.urlToFilename(url);
  let body;
  try {
    body = yield readFile(filename, 'utf8');
  } catch(err) {
    if(err.code !== 'ENOENT') {
      throw err;
    }
    body = yield download(url, filename);
  }
  yield spiderLinks(url, body, nesting);
}
```

Интересная деталь, которую следует отметить, – возможность использования блока `try...catch` для обработки исключений. Кроме того, теперь можно использовать

throw для передачи ошибок! Обратите также внимание на строку, где yield возвращает функцию download(), которая не является ни преобразователем, ни функцией с Promise, а просто еще одним генератором. Это возможно благодаря применению решения co, которое добавляет поддержку генераторов в инструкцию yield.

И наконец, можем преобразовать и функцию spiderLinks(), в которой реализуются итерации для последовательной загрузки ссылок веб-страницы. Благодаря генераторам она также становится тривиально простой:

```
function* spiderLinks(currentUrl, body, nesting) {
  if(nesting === 0) {
    return nextTick();
  }

  const links = utilities.getPageLinks(currentUrl, body);
  for(let i = 0; i <links.length; i++) {
    yield spider(links[i], nesting - 1);
  }
}
```

Предыдущий код не требует особых пояснений. Не существует шаблона организации последовательных итераций, поскольку генераторы и co самостоятельно проделывают всю работу, позволяя писать код асинхронных итераций, как если бы использовался блокирующий прямой стиль программного интерфейса.

Теперь перейдем к самой важной части – точке входа в программу:

```
co(function* () {
  try {
    yield spider(process.argv[2], 1);
    console.log('Download complete');
  } catch(err) {
    console.log(err);
  }
});
```

Это единственное место, где необходимо вызвать co(...), чтобы обернуть генератор. После этого co автоматически и рекурсивно будет обертывать любой генератор, передаваемый оператору yield, так что остальная часть программы полностью независима от использования или неиспользования co, несмотря на то что co постоянно присутствует за кулисами.

Теперь можно запустить версию веб-паука на основе генераторов.

Параллельное выполнение

Генераторы отлично подходят для реализации последовательных алгоритмов, но не могут использоваться для организации параллельного выполнения набора задач, по крайней мере, только с помощью операторов yield и генераторов. Фактически шаблон использования генераторов в этих случаях просто вырождается в шаблон обратных вызовов или применения объектов Promise, которые передаются оператору yield и используются с генераторами.

К счастью для конкретного случая неограниченного параллельного выполнения, co поддерживает эту возможность изначально, позволяя просто передавать оператору yield массива объектов Promise, преобразователей, генераторов или функций генераторов.

С учетом этого версия 3 приложения веб-паука может быть реализована простой реорганизацией функции `spiderLinks()`:

```
function* spiderLinks(currentUrl, body, nesting) {
  if(nesting === 0) {
    return nextTick();
  }

  const links = utilities.getPageLinks(currentUrl, body);
  const tasks = links.map(link => spider(link, nesting - 1));
  yield tasks;
}
```

Здесь мы просто собрали вместе все задания загрузки, которые, по сути, являются генераторами, и с помощью оператора `yield` вернули получившийся массив. Все эти задания будут выполняться параллельно, а затем, после завершения всех заданий, будет возобновлено выполнение генератора (`spiderLinks`).

Функция `co` позволяет вернуть с помощью `yield` массив, однако того же эффекта параллельного выполнения можно достичь с применением функций обратного вызова, подобно тому, как было показано ранее в этой главе. Давайте воспользуемся этой возможностью и перепишем `spiderLinks()` еще раз:

```
function spiderLinks(currentUrl, body, nesting) {
  if(nesting === 0) {
    return nextTick();
  }

  //возвращает преобразователь
  return callback => {
    let completed = 0, hasErrors = false;
    const links = utilities.getPageLinks(currentUrl, body);
    if(links.length === 0) {
      return process.nextTick(callback);
    }

  function done(err, result) {
    if(err && !hasErrors) {
        hasErrors = true;
        return callback(err);
      }
      if(++completed === links.length && !hasErrors) {
        callback();
      }
    }

    for(let i = 0; i < links.length; i++) {
      co(spider(links[i], nesting - 1)).then(done);
    }
  }
}
```

Для параллельного запуска функции `spider()` используется решение `co`, выполняющее генератор и возвращающее объект `Promise`. То есть мы можем дождаться разрешения объекта `Promise` и вызвать функцию `done()`. Как правило, все библиотеки для

реализации управления выполнением на основе генераторов имеют сходные функциональные возможности, поэтому решение на основе генераторов всегда можно преобразовать в решение на основе объектов `Promise`, если потребуется.

Для параллельного запуска нескольких заданий загрузки достаточно просто повторно использовать шаблон обратных вызовов, описанный ранее в этой главе. Обратите также внимание, что функция `spiderLinks()` теперь реализована как преобразователь (она перестала быть генератором). Это позволило получить функцию обратного вызова, которая вызывается после завершения всех параллельных заданий.

Шаблон (преобразование генератора в преобразователь)
Преобразование генератора в преобразователь с целью получить возможность запускать его параллельно или использовать его преимущества в алгоритмах, основанных на обратных вызовах или объектах `Promise`.

Ограниченное параллельное выполнение

Теперь, зная, как поступать с непоследовательными потоками выполнения, мы легко сможем реализовать версию 4 приложения веб-паука, поскольку остается только добавить ограничение числа одновременно выполняемых заданий загрузки. Для этого можно применить несколько подходов; вот некоторые из них:

- использовать версию с обратными вызовами на основе реализованного ранее класса `TaskQueue`. Для этого потребуется превратить функции в преобразователи и добавить генератор, используемый в качестве задания;
- использовать версию класса `TaskQueue`, основанную на объектах `Promise`, и преобразовать каждый генератор, используемый как задание, в функцию, возвращающую объект `Promise`;
- использовать библиотеку `async` и превратить в преобразователи все вспомогательные функции, которые планируется задействовать, а также превратить все генераторы в функции, осуществляющие обратные вызовы, которые сможет использовать эта библиотека;
- использовать одну из библиотек экосистемы `co`, специально разработанных для этого вида управления выполнением, например **co-limiter** (https://npmjs.org/package/co-limiter);
- реализовать собственный алгоритм, основанный на шаблоне производитель/потребитель (producer/consumer), который внутренне использует библиотека co-limiter.

Здесь мы выбрали последний вариант, поскольку он позволит подробно рассмотреть шаблон, тесно связанный с сопрограммами (а также с потоками и процессами).

Шаблон производитель/потребитель

Наша цель – использовать очереди для поддержания постоянного количества рабочих процессов, соответствующего заданному ограничению. Основой для реализации этого алгоритма станет класс `TaskQueue`, определенный ранее в этой же главе:

```
class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency;
    this.running = 0;
    this.taskQueue = [];
    this.consumerQueue = [];
```

```
    this.spawnWorkers(concurrency);
  }

  pushTask(task) {
    if (this.consumerQueue.length !== 0) {
      this.consumerQueue.shift()(null, task);
    } else {
      this.taskQueue.push(task);
    }
  }

  spawnWorkers(concurrency) {
    const self = this;
    for(let i = 0; i < concurrency; i++) {
      co(function* () {
        while(true) {
          const task = yield self.nextTask();
          yield task;
        }
      });
    }
  }

  nextTask() {
    return callback => {
      if(this.taskQueue.length !== 0) {
        return callback(null, this.taskQueue.shift());
      }

      this.consumerQueue.push(callback);
    }
  }
}
```

Проанализируем новую реализацию класса `TaskQueue`. Начнем с конструктора. Обратите внимание на вызов метода `this.spawnWorkers()`, отвечающего за запуск рабочих процессов.

Рабочие процессы устроены просто; это генераторы, завернутые в вызов `co()` и запускающиеся немедленно, так что все они могут выполняться параллельно. Внутренне каждый выполняет бесконечный цикл, который блокируется (`yield`) в ожидании поступления нового задания в очередь (`yield self.nextTask()`). Когда это происходит, он передает управление заданию (любой объект, допустимый для оператора `yield`) и ожидает его завершения. Вам может быть интересно, как происходит ожидание появления следующего задания в очереди. Все дело в методе `nextTask()`. Рассмотрим более подробно, что происходит в нем:

```
nextTask() {
  return callback => {
    if(this.taskQueue.length !== 0) {
      return callback(null, this.taskQueue.shift());
    }
    this.consumerQueue.push(callback);
  }
}
```

Поясним, что происходит в этом методе, который является основой шаблона.

1. Метод возвращает преобразователь, допустимый в операторе `yield`, когда используется решение `co`.
2. Функция обратного вызова, возвращаемая преобразователем, вызывается при передаче следующего задания в функцию `taskQueue` (если имеется). Это немедленно разблокирует рабочий процесс и передает ему следующее задание через оператор `yield`.
3. Если в очереди больше нет заданий, в `consumerQueue` передается функция обратного вызова. Это переводит рабочий процесс в режим *ожидания*. Обратные вызовы в функции `consumerQueue` вызываются при появлении нового задания для обработки, что возобновит выполнение соответствующего рабочего процесса.

Теперь, чтобы понять, как функция `consumerQueue` выводит рабочие процессы из состояния бездействия, необходимо проанализировать метод `pushTask()`. Метод `pushTask()` вызывает первую функцию обратного вызова в `consumerQueue`, если имеется, что, в свою очередь, разблокирует рабочий процесс. Если обратные вызовы отсутствуют, это означает, что все рабочие процессы заняты, тогда просто добавляется новый элемент в функцию `taskQueue`.

В классе `TaskQueue` рабочие процессы выступают в роли потребителей, а код, использующий `pushTask()`, можно считать производителем. Этот шаблон демонстрирует, насколько генератор может быть похож на поток (или процесс). Фактически взаимодействие производителя и потребителя является самой распространенной задачей при изучении методов взаимодействий, кроме того, как упоминалось выше, этот вид взаимодействий также часто применяется в сопрограммах.

Ограничение количества параллельных заданий загрузки

Теперь, имея опыт реализации алгоритма ограничения параллельной обработки с помощью генераторов и шаблона производитель/потребитель, можно применить его для ограничения одновременно выполняющихся заданий в приложении веб-паука (версия 4). Сначала загрузим и инициализируем объект `TaskQueue`:

```
const TaskQueue = require('./taskQueue');
const downloadQueue = new TaskQueue(2);
```

Затем изменим функцию `spiderLinks()`. Она практически полностью идентична версии в реализации выполнения неограниченного количества заданий, поэтому здесь приводятся только изменившиеся фрагменты кода:

```
function spiderLinks(currentUrl, body, nesting) {
  //...
  return (callback) => {
    //...
    function done(err, result) {
      //...
    }
    links.forEach(function(link) {
      downloadQueue.pushTask(function *() {
        yield spider(link, nesting - 1);
        done();
      });
    });
  }
}
```

В каждом из заданий, сразу после загрузки, вызывается функция `done()`, что позволяет подсчитать количество загруженных ссылок, а также уведомить преобразователя через функцию обратного вызова о загрузке всех ссылок.

В качестве упражнения попробуйте самостоятельно реализовать версию 4 приложения веб-паука с помощью других четырех методов, приведенных в начале этого раздела.

Async/await с использованием Babel

Обратные вызовы, объекты `Promise` и генераторы являются средствами разработки асинхронного кода на JavaScript и на платформе Node.js. Как мы видели, генераторы интересны тем, что дают возможность приостанавливать выполнение функции и возобновлять ее позднее. Теперь мы можем адаптировать эту возможность, чтобы разработчики могли писать асинхронный код, который «выглядит» заблокированным в каждой асинхронной операции, ожидающей результатов перед выполнением следующего оператора.

Проблема в том, что функции-генераторы предназначены в основном для создания итераторов, и асинхронный код, написанный с их помощью, выглядит несколько громоздким. В нем сложно разобраться, его трудно читать и поддерживать.

Однако есть надежда, что в ближайшем будущем этот синтаксис станет более чистым. Фактически интересное предложение содержит спецификация ECMAScript 2017, определяющая синтаксис функции `async`.

Более подробные сведения о текущем состоянии предложения async/await можно найти на странице https://tc39.github.io/ecmascript-asyncawait/.

Согласно описанию, функция `async` призвана значительно улучшить модель реализации асинхронного кода на уровне языка путем добавления двух новых ключевых слов: `async` и `await`.

Чтобы выяснить порядок использования этих ключевых слов и в чем их полезность, рассмотрим короткий пример:

```
const request = require('request');

function getPageHtml(url) {
  return new Promise(function(resolve, reject) {
    request(url, function(error, response, body) {
      resolve(body);
    });
  });
}

async function main() {
  const html = await getPageHtml('http://google.com');
  console.log(html);
}

main();
console.log('Loading...');
```

Здесь определяются две функции: `getPageHtml` и `main`. Первая просто получает HTML-код удаленной веб-страницы по заданному URL. Стоит отметить, что эта функция возвращает объект `Promise`.

Функция `main` более интересная, поскольку в ней используются новые ключевые слова `async` и `await`. Прежде всего обратите внимание на дополнительный префикс `async`. Он означает, что функция выполняет асинхронный код и позволяет использовать в ее теле ключевое слово `await`. Ключевое слово `await` перед вызовом функции `getPageHtml` требует от интерпретатора JavaScript «дождаться» разрешения объекта `Promise`, возвращаемого функцией `getPageHtml`, прежде чем переходить к выполнению следующей инструкции. То есть выполнение функции `main` приостанавливается в ожидании завершения асинхронного кода без блокировки нормального выполнения остальной части программы. Фактически мы увидим в консоли строку `Loading...`, и через мгновение в ней появится HTML-код домашней страницы Google.

Разве такой код не является более простым и понятным?

К сожалению, это предложение еще не является окончательным, и даже если оно будет одобрено, придется ждать выхода следующей версии стандарта ECMAScript, чтобы оно было интегрировано в платформу Node.js и появилась возможность использовать этот новый синтаксис как встроенный.

Так что же делать сегодня? Просто ждать? Нет, конечно же, нет! Уже сейчас имеется возможность использовать ключевые слова `async` и `await` благодаря транскомпиляторам, таким как Babel.

Установка и запуск Babel

Babel – это компилятор JavaScript (точнее, транскомпилятор), преобразующий код на JavaScript в иной код на JavaScript с помощью преобразователей синтаксиса. Преобразователи синтаксиса позволяют использовать новый синтаксис, такой как ES2015, ES2016, JSX и т. д., путем получения обратно совместимого эквивалентного кода, который может выполняться в современных средах JavaScript, таких как браузеры или Node.js.

Установить Babel для проекта можно с помощью npm, выполнив следующую команду:

```
npm install --save-dev babel-cli
```

Кроме того, необходимо установить расширения для поддержки синтаксического анализа и преобразования async/await:

```
npm install --save-dev babel-plugin-syntax-async-functions
babel-plugin-transform-async-to-generator
```

Теперь предположим, что требуется запустить приведенный пример (`index.js`). Это можно сделать с помощью следующей команды:

```
node_modules/.bin/babel-node --plugins
"syntax-async-functions,transform-async-to-generator" index.js
```

Эта команда преобразует исходный код модуля `index.js` будет изменен на лету и применит преобразователи для поддержки async/await. Этот новый обратно совместимый код будет сохранен в памяти и сразу же выполнен средой Node.js.

Кроме того, транскомпилятор Babel можно настроить на работу в качестве процессора сборки, сохраняющего сгенерированный код в файлы, чтобы затем можно было развернуть и запустить созданный им код.

Более подробные сведения об установке и настройке транскомпилятора Babel можно найти на его официальном сайте https://babeljs.io.

Сравнение

В этой главе было рассмотрено несколько вариантов реализации асинхронных операций на языке JavaScript. Каждое из этих решений имеет свои плюсы и минусы, перечисленные в табл. 4.1.

Таблица 4.1. Плюсы и минусы разных способов реализации асинхронных операций в JavaScript

Решения	Плюсы	Минусы
Обычный JavaScript	• Не требует применения дополнительных библиотек и технологий. • Часто обладает лучшей производительностью. • Позволяет реализовать специальные и продвинутые алгоритмы	Может потребоваться дополнительный код и относительно сложные алгоритмы
`async` (библиотека)	• Упрощает разработку наиболее распространенных шаблонов управления выполнением. • Поддерживает решения на основе обратных вызовов. • Хорошая производительность	• Вводит внешнюю зависимость. • Подходит не для всех вариантов управления выполнением
Объекты `Promise`	• Наиболее простой и самый распространенный шаблон управления выполнением. • Упрощает обработку ошибок. • Часть спецификации ES2015. • Гарантированный отложенный вызов `onFulfilled` и `onRejected`	• Требует преобразования программных интерфейсов, основанных на обратных вызовах. • Несколько снижает производительность
Генераторы	• Делает неблокирующий программный интерфейс похожим на блокирующий. • Упрощает обработку ошибок. • Часть спецификации ES2015	• Требует наличия библиотеки управления выполнением. • Все еще требует обратных вызовов или объектов `Promise` для реализации непоследовательного выполнения. • Требует приведения программных интерфейсов к преобразователям или объектам `Promise`, не основанным на генераторах
Ключевые слова `async` и `await`	• Делает неблокирующий программный интерфейс похожим на блокирующий. • Наглядный и интуитивно понятный синтаксис	• Пока не имеет встроенной поддержки в JavaScript и Node.js. • В настоящее время требует использования Babel или другого транскомпилятора и определенной настройки

Стоит отметить, что в этой главе представлены лишь самые популярные решения реализации асинхронных операций, считающиеся самыми перспективными, но имеются и другие решения, такие, например, как `Fibers` (https://npmjs.org/package/fibers) и `Streamline` (https://npmjs.org/package/streamline).

Итоги

В этой главе рассматривалось несколько альтернативных подходов для работы с асинхронными потоками, включая объекты `Promise`, генераторы и недавно появившийся синтаксис async/await.

Было описан порядок использования этих подходов для реализации асинхронного кода, который в результате становится более коротким, наглядным и понятным. Мы

обсудили наиболее важные преимущества и недостатки этих подходов, выяснили, что при всей полезности требуется определенное время на их освоение. Поэтому ими не следует полностью заменять обратные вызовы, которые по-прежнему остаются очень полезными во многих ситуациях. Разработчик должен иметь возможность выбрать наилучшее решение для конкретной проблемы, с которой он столкнулся. При создании общедоступных библиотек, выполняющих асинхронные операции, необходимо реализовать такой интерфейс, чтобы его могли без особых усилий использовать разработчики, ориентирующиеся на обратные вызовы.

Следующая глава будет посвящена еще одной интересной теме, тоже имеющей отношение к асинхронным операциям, которая также является одной из основ всей экосистемы Node.js, а именно потокам данных.

Глава 5

Программирование с применением потоков данных

Потоки данных (streams) являются одним из наиболее важных компонентов и шаблонов Node.js. В сообществе распространен девиз: «Потоки данных – это все!» – и одного его достаточно, чтобы охарактеризовать роль потоков данных в Node.js. Доминик Tapp (Dominic Tarr), внесший большой вклад в развитие платформы Node.js, определяет потоки данных как лучшую и самую недооцененную идею. Существует несколько причин, делающих потоки данных на платформе Node.js столь привлекательными. Это связано не только с их чисто техническими характеристиками, такими как производительность и эффективность, но и с их элегантностью и тем, как они вписываются в философию Node.js.

В этой главе рассматриваются следующие темы:

- почему потоки данных так важны на платформе Node.js;
- использование и создание потоков данных;
- потоки данных как парадигма программирования: использование их возможностей в различных контекстах, а не только в области ввода-вывода;
- шаблоны конвейеризации и подключение потоков в различных конфигурациях.

Исследование важности потоков данных

В платформах, основанных на событиях, таких как Node.js, наиболее эффективным способом обработки ввода/вывода являются работа в режиме реального времени, получение входных данных по мере их доступности и отправка выходных данных по мере их создания приложением.

В этом разделе мы получим первое представление о потоках данных в Node.js и познакомимся с их сильными сторонами. Но имейте в виду, что это будет только общий обзор, поскольку за ним, в этой же главе, последует более подробный анализ использования и создания потоков данных.

Буферизация и потоковая передача данных

Почти все асинхронные программные интерфейсы, рассматривавшиеся выше в этой книге, использовали *режим буферизации*. Для операций ввода режим буферизации

требует помещения в буфер всех поступающих данных, которые затем передаются функции обратного вызова, сразу после завершения их чтения из ресурса.

Схема на рис. 5.1 иллюстрирует эту парадигму.

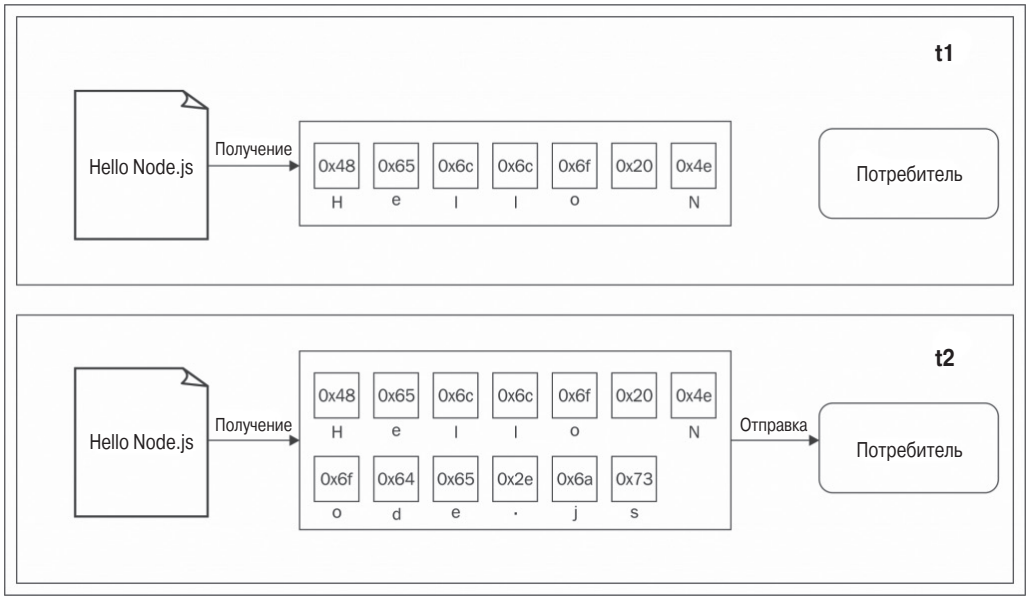

Рис. 5.1 ❖ Буферизация

На рис. 5.1 можно видеть, что в момент времени **t1** от ресурса получены и сохранены в буфер некоторые данные. В момент времени **t2** получен еще один фрагмент данных, последний, завершающий операцию чтения и вызывающий отправку потребителю всего буфера.

Потоки данных, напротив, позволяют обрабатывать данные по мере их поступления из ресурса. Это демонстрирует схема на рис. 5.2.

На этот раз, как показывает схема, при получении каждого нового блока данных он тут же отправляется потребителю, который имеет возможность обработать его сразу же, не дожидаясь сохранения всех данных в буфере.

Но чем различаются эти два подхода? Различия можно разделить на две основные категории:

- эффективность с точки зрения памяти;
- эффективность с точки зрения времени.

Однако потоки данных в Node.js имеют еще одно важное преимущество – возможность объединения. А теперь посмотрим, как эти свойства влияют на разработку приложений.

Эффективность с точки зрения памяти

Прежде всего потоки данных позволяют сделать то, что не представляется возможным при использовании режима буферизации с последующей обработкой всех данных разом. Например, рассмотрим случай, когда требуется прочитать очень большой

файл, скажем, порядка сотни мегабайтов или даже нескольких гигабайтов. Очевидно, что использование для этого программного интерфейса, который создает большой буфер для размещения файла целиком, – не самая лучшая идея. А теперь представьте, что требуется параллельно прочитать несколько таких файлов. В этом случае приложение легко заполнит всю доступную ему память. Кроме того, буферы V8 не могут быть больше *0x3FFFFFFF* байт (чуть меньше 1 ГБ). Как результат приложение упрется в непреодолимую стену до исчерпания физической памяти.

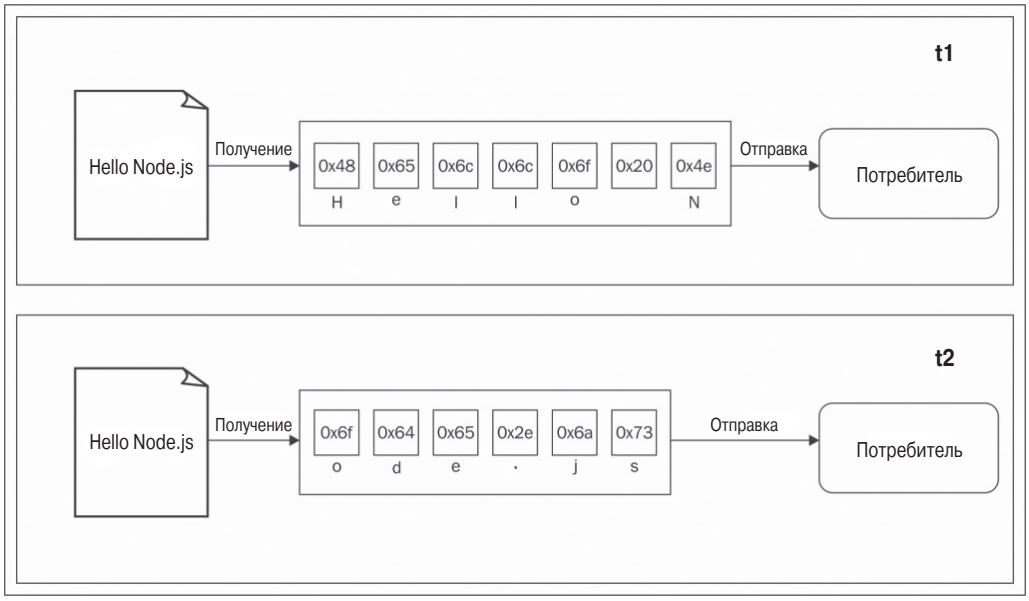

Рис. 5.2 ❖ Поток данных

Сжатие при использовании буферизующего API

Чтобы привести конкретный пример, рассмотрим простое приложение с интерфейсом командной строки (CLI), сжимающее файлы в формат Gzip. При применении буферизующего программного интерфейса такое приложение для платформы Node.js будет выглядеть следующим образом (обработка ошибок опущена для краткости):

```
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];
fs.readFile(file, (err, buffer) => {
  zlib.gzip(buffer, (err, buffer) => {
    fs.writeFile(file + '.gz', buffer, err => {
      console.log('File successfully compressed');
    });
  });
});
```

Теперь поместим приведенный код в файл `gzip.js` и запустим его командой:

```
node gzip <путь к файлу>
```

Если выбрать достаточно большой файл, скажем, чуть больше 1 ГБ, мы получим сообщение об ошибке, извещающее, что сжимаемый файл превысил максимально допустимый размер буфера:

```
RangeError: File size is greater than possible Buffer: 0x3FFFFFFF bytes
```

Это именно то, что должно было произойти, и свидетельствует о выборе неправильного подхода.

Сжатие с использованием потоков данных

Простейший способ исправить приложение и заставить его работать с большими файлами – использовать потоковый программный интерфейс. Посмотрим, как это можно сделать. Изменим содержимое только что созданного модуля, как показано ниже:

```
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];

fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream(file + '.gz'))
  .on('finish', () => console.log('File successfully compressed'));
```

«И это все?» – можете вы спросить. Да, как уже упоминалось, потоки данных привлекают своим интерфейсом и возможностью объединения, обеспечивая таким образом наглядность и элегантность кода. Все это будет продемонстрировано ниже, а сейчас важно лишь понять, что эта программа без проблем справится с файлами любого размера, в идеале расходуя фиксированный объем памяти. Попробуйте сами (но учтите, что сжатие больших файлов может занять определенное время).

Эффективность с точки зрения времени

А теперь рассмотрим пример приложения, которое сжимает файл и выгружает его на удаленный сервер HTTP, который, в свою очередь, распаковывает его и сохраняет в своей файловой системе. Если клиент реализован с использованием буферизующего программного интерфейса, выгрузка начнется только после чтения и сжатия всего файла целиком. Кроме того, распаковка файла на сервере начнется лишь после получения всех данных. Лучшее решение для достижения того же результата – использование потоков данных. На клиентской машине потоки данных позволят сжимать и отправлять фрагменты данных по мере их чтения из файловой системы, а на сервере – распаковывать фрагменты по мере их получения от удаленного узла. Продемонстрируем это, реализовав приложение. Начнем со стороны сервера.

Создадим модуль `gzipReceive.js`:

```
const http = require('http');
const fs = require('fs');
const zlib = require('zlib');

const server = http.createServer((req, res) => {
  const filename = req.headers.filename;
  console.log('File request received: ' + filename);
  req
```

```
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream(filename))
    .on('finish', () => {
      res.writeHead(201, {'Content-Type': 'text/plain'});
      res.end('That's it\n');
      console.log(`File saved: ${filename}`);
  });
});

server.listen(3000, () => console.log('Listening'));
```

Благодаря потокам данных сервер читает фрагменты из сети, распаковывает их и сохраняет по мере их получения.

Клиентская сторона приложения находится в модуле `gzipSend.js`:

```
const fs = require('fs');
const zlib = require('zlib');
const http = require('http');
const path = require('path');
const file = process.argv[2];
const server = process.argv[3];

const options = {
  hostname: server,
  port: 3000,
  path: '/',
  method: 'PUT',
  headers: {
    filename: path.basename(file),
    'Content-Type': 'application/octet-stream',
    'Content-Encoding': 'gzip'
  }
};

const req = http.request(options, res => {
  console.log('Server response: ' + res.statusCode);
});

fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(req)
  .on('finish', () => {
  console.log('File successfully sent');
});
```

Здесь снова используются потоки данных для чтения блоков из файла, их сжатия и отправки.

Чтобы опробовать приложение, запустите сервер следующей командой:

```
node gzipReceive
```

Затем запустите клиента, указав файл для отправки и адрес сервера (например, `localhost`):

```
node gzipSend <путь к файлу> localhost
```

Если выбрать достаточно большой файл, станет заметна легкость, с какой потоки переносят данные между клиентом и сервером. Но почему эта парадигма позволяет передавать данные намного эффективнее, в сравнении с буферизующим программным интерфейсом? Схема на рис. 5.3 дает ответ на этот вопрос.

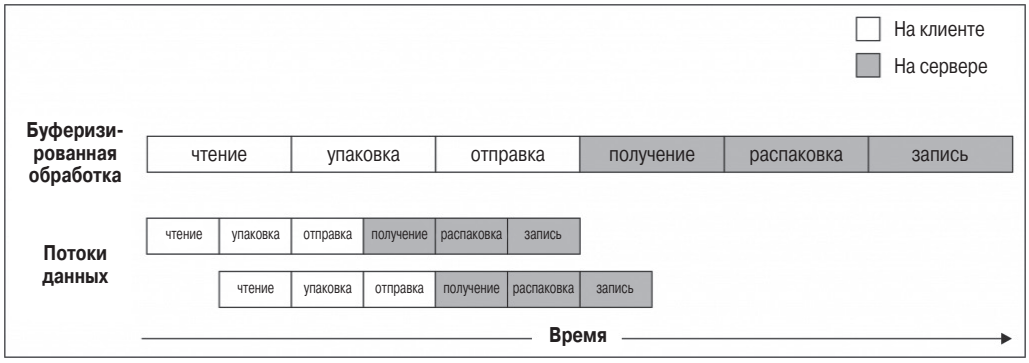

Рис. 5.3 ❖ Секрет эффективности потоков данных

Обработка файла включает ряд этапов, выполняемых последовательно:

1. [Клиент] Чтение из файловой системы.
2. [Клиент] Сжатие данных.
3. [Клиент] Отправка на сервер.
4. [Сервер] Получение от клиента.
5. [Сервер] Распаковка данных.
6. [Сервер] Запись данных на диск.

Для завершения обработки необходимо последовательно пройти все этапы от начала до конца. Схема на рис. 5.3 показывает, что при использовании буферизующего программного интерфейса этот процесс является строго последовательным. Для сжатия данных нужно дождаться чтения всего файла целиком, для отправки нужно дождаться окончания чтения и сжатия всего файла и т. д. При использовании потоков данных конвейер начинает работу сразу после получения первого блока, не дожидаясь чтения всего файла. Но самое интересное, что перед чтением следующего фрагмента данных нет необходимости ждать завершения предыдущего набора заданий, вместо этого параллельно начинает выполняться другой конвейер. Это возможно, потому что все задания являются асинхронными и их можно выполнять на платформе Node.js параллельно, единственным ограничением является сохранение порядка, в котором фрагменты поступают на этапы обработки (и потоки данных в платформе Node.js сами позаботятся об этом).

Как следует из рис. 5.3, благодаря использованию потоков данных весь процесс занимает меньше времени, поскольку не тратится время на ожидание чтения и обработки сразу всех данных .

Способность к объединению

Пример выше дает представление о том, как можно объединять потоки данных с помощью метода `pipe()`, соединяющего различные этапы обработки, каждая из которых отвечает только за одну операцию, что соответствует безупречному стилю платфор-

мы Node.js. Это возможно благодаря единому интерфейсу потоков данных, который позволяет им взаимодействовать друг с другом. Единственное условие: следующий поток в конвейере должен поддерживать тип данных, посылаемых предыдущим потоком, которые могут быть двоичным, текстовым или даже объектом, как будет показано далее в этой главе.

Для демонстрации широты возможностей добавим уровень шифрования в созданное приложение `gzipReceive/gzipSend`.

Для этого нужно изменить код клиента, добавив в конвейер еще один поток данных, а именно возвращаемый функцией `crypto.createChipher()`:

```
const crypto = require('crypto');
// ...
fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(crypto.createCipher('aes192', 'a_shared_secret'))
  .pipe(req)
  .on('finish', () => console.log('File succesfully sent'));
```

Аналогично нужно изменить код сервера, чтобы расшифровать данные перед распаковкой:

```
const crypto = require('crypto');
// ...
const server = http.createServer((req, res) => {
  // ...
  req
    .pipe(crypto.createDecipher('aes192', 'a_shared_secret'))
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream(filename))
    .on('finish', () => { /* ... */ });
});
```

Приложив немного усилий (чтобы написать несколько строк кода), мы добавили в приложение уровень шифрования. Для этого потребовалось просто включить существующий поток преобразования в имеющийся конвейер. Аналогично можно добавлять и комбинировать и другие потоки, подобно деталям конструктора Lego.

Очевидно, что главным преимуществом этого подхода является повторное использование, но, как следует из примеров выше, потоки данных также позволяют писать более наглядный и модульный код. По этим причинам потоки часто используются не только для реализации операций ввода/вывода, но и как средство для упрощения и повышения модульности кода.

Начало работы с потоками данных

В предыдущем разделе мы узнали, почему потоки являются настолько мощным средством и что на платформе Node.js они присутствуют везде, начиная с модулей ядра. Например, модуль `fs` содержит поток `createReadStream()` для чтения из файла и поток `createWriteStream()` для записи в файл, HTTP-объекты `request` и `response`, по своей сути, являются потоками, а модуль `zlib` позволяет сжимать и распаковывать данные с помощью потокового интерфейса.

Теперь, когда понятно, почему так важны потоки данных, вернемся назад и приступим к их более подробному изучению.

Анатомия потоков данных

Каждый поток данных в платформе Node.js является реализацией одного из четырех базовых абстрактных классов, доступных через модуль `stream` ядра:

- `stream.Readable`;
- `stream.Writable`;
- `stream.Duplex`;
- `stream.Transform`.

Все классы в модуле `stream` также являются экземплярами класса `EventEmitter`. Потоки данных способны генерировать несколько типов событий, таких как `end`, когда поток `Readable` завершает чтение, или `error`, когда что-то пошло не так.

Обратите внимание, что в примерах, представленных в этой главе, мы часто будем опускать обработку ошибок ради экономии места. Однако в реальных приложениях рекомендуется всегда регистрировать обработчики событий `error` для всех потоков.

Одна из причин гибкости потоков данных заключается в их способности справляться не только с двоичными данными, но и практически с любыми значениями JavaScript. На самом деле они поддерживают два режима работы:

- **двоичный режим**: этот режим используется для потоковой передачи данных в виде фрагментов, например буферов или строк;
- **объектный режим**: этот режим используется для передачи через поток данных последовательности дискретных объектов (что позволяет использовать практически любое значение JavaScript).

Эти два режима позволяют использовать потоки не только для ввода/вывода, но и как инструмент элегантного комбинирования этапов обработки в функциональном стиле, как будет продемонстрировано далее в этой главе.

В этой главе мы будем использовать в основном интерфейс потоков Node.js, известный также как **Version 3**, реализованный в версии 0.11 платформы Node.js. Более подробную информацию об отличиях от старых интерфейсов можно найти на странице https://strongloop.com/strongblog/whats-new-io-js-beta-streams3/.

Потоки данных для чтения

Поток для чтения – это источник данных. На платформе Node.js он реализуется абстрактным классом `Readable` в модуле `stream`.

Чтение из потока

Существуют два способа получения данных из потока для чтения: **дискретный** и **непрерывный**. Рассмотрим эти режимы более подробно.

Дискретный режим По умолчанию чтение из потока заключается в подключении обработчика события `readable`, которое уведомляет о наличии новых данных, готовых для чтения. Затем в цикле выполняется чтение данных, пока не опустошится внутренний буфер. Это можно сделать с помощью метода `read()`, выполняющего синхронное чтение из буфера и возвращающего объект `Buffer` или `String`, представляющий фрагмент данных. Метод `read()` имеет следующую сигнатуру:

```
readable.read([size])
```

При использовании этого подхода данные извлекаются из потока по требованию.

Для демонстрации создадим новый модуль `readStdin.js`, реализующий простую программу, которая читает стандартный ввод (поток для чтения) и записывает прочитанное в стандартный вывод:

```
process.stdin
  .on('readable', () => {
    let chunk;
    console.log('New data available');
    while((chunk = process.stdin.read()) !== null) {
      console.log(
        `Chunk read: (${chunk.length}) "${chunk.toString()}"`
      );
    }
  })
  .on('end', () => process.stdout.write('End of stream'));
```

Метод `read()` является синхронной операцией, извлекающей фрагмент данных из внутренних буферов потока для чтения. По умолчанию он возвращает объект `Buffer`, если поток работает в двоичном режиме.

Используя поток для чтения, работающий в двоичном режиме, можно читать строки вместо буферов, вызывая метод `setEncoding(encoding)` потока и передавая ему допустимый формат кодировки (например, `utf8`).

Данные читаются только обработчиком `readable`, который вызывается, как только новые данные становятся доступными. Метод `read()` возвращает значение `null`, если во внутренних буферах нет данных; в таком случае нужно ждать поступления следующего события `readable`, уведомляющего о возможности продолжить чтение, или события `end`, оповещающего об окончании потока. При работе в двоичном режиме имеется возможность определить требуемый объем данных путем передачи значения `size` методу `read()` в качестве параметра. Это особенно полезно при реализации сетевых протоколов или при парсинге данных в специфичных форматах.

Теперь можно запустить модуль `readStdin` на выполнение и поэкспериментировать с ним. Введите несколько символов в консоли и нажмите клавишу **Enter**, чтобы вывести в стандартный вывод только что введенные данные. Чтобы завершить работу потока со стандартным событием `end`, введите символ `EOF` (конец файла) (**Ctrl**+**Z** в Windows или **Ctrl**+**D** в Linux).

Можно также попытаться соединить программу с другими процессами с помощью оператора конвейера (`|`), который перенаправляет стандартный вывод одной программы на стандартный ввод другой. Например:

```
cat <путь к файлу> | node readStdin
```

Этот пример наглядно демонстрирует, что потоковая парадигма является универсальным интерфейсом, позволяющим программам общаться между собой, на каком бы языке они не были написаны.

Непрерывный режим Другой способ чтения из потока – подключение обработчика события. В результате поток переключается в непрерывный режим, когда дан-

ные не извлекаются вызовом метода `read()`, а передаются обработчику события сразу после их поступления. Например, реализация непрерывного режима в приложении `readStdin` будет выглядеть следующим образом:

```
process.stdin
  .on('data', chunk => {
  console.log('New data available');
  console.log(
    `Chunk read: (${chunk.length}) "${chunk.toString()}"`
  );
})
.on('end', () => process.stdout.write('End of stream'));
```

Непрерывный режим унаследован от старой версии интерфейса потоков данных (которую также называют **Streams1**), имевшей меньшую гибкость в управлении потоками данных. С введением интерфейса **Streams2** непрерывный режим перестал быть режимом по умолчанию, для переключения на него необходимо подключить обработчик события или явно вызвать метод `resume()`. Чтобы временно приостановить генерацию событий, можно вызвать метод `pause()` – это заставит поток кэшировать все поступающие данные во внутреннем буфере.

Вызов `pause()` не переключает поток обратно в дискретный режим.

Реализация потоков для чтения

Теперь, узнав, как осуществляется чтение данных из потока, мы можем перейти к знакомству с приемами реализации нового потока для чтения. Для этого необходимо создать новый класс, унаследовав прототип `stream.Readable`. Конкретный поток должен включать реализацию метода `_read()` со следующей сигнатурой:

```
readable._read(size)
```

Внутренне класс `Readable` будет вызывать метод `_read()`, который, в свою очередь, начнет заполнять внутренний буфер вызовом метода `push()`:

```
readable.push(chunk)
```

Обратите внимание, что метод `read()` вызывается потребителями потока, в то время как метод `_read()` реализуется подклассом и никогда не должен вызываться напрямую. Подчеркивание в начале имени метода обычно указывает, что метод не является общедоступным и не должен вызываться непосредственно.

Чтобы показать, как создаются новые потоки для чтения, реализуем поток, генерирующий случайные строки. Создадим новый модуль `randomStream.js`, содержащий код генератора строк:

```
const stream = require('stream');
const Chance = require('chance');
const chance = new Chance();

class RandomStream extends stream.Readable {
  constructor(options) {
    super(options);
  }
```

```
  _read(size) {
    const chunk = chance.string();                          //[1]
    console.log(`Pushing chunk of size: ${chunk.length}`);
    this.push(chunk, 'utf8');                               //[2]
    if(chance.bool({likelihood: 5})) {                      //[3]
      this.push(null);
    }
  }
}

module.exports = RandomStream;
```

В начале файла выполняется загрузка зависимостей. Здесь нет ничего интересного, за исключением загрузки npm-модуля `chance` (`https://npmjs.org/package/chance`) – библиотеки, позволяющей генерировать все виды случайных значений, от чисел до строк и целых предложений.

Затем создается новый класс `RandomStream`, родителем которого является `stream.Readable`. Для инициализации внутреннего состояния и обработки переданных необязательных аргументов в приведенном примере вызывается конструктор родительского класса.

Имеется возможность передать объекту следующие необязательные параметры:

- `encoding` – используется для преобразования значений с типом данных `Buffer` в значения с типом данных `String` (значение по умолчанию `null`);
- `objectMode` – флаг включения объектного режима (по умолчанию получает значение `false`);
- `highWaterMark` – верхнее ограничение объема данных, сохраняемых в буфере, по достижении которого прекращается чтение из источника (по умолчанию получает значение, равное 16 КБ).

А теперь перейдем к методу `_read()`.

- Этот метод создает случайную строку с помощью `chance`.
- Помещает ее во внутренний буфер для чтения. Обратите внимание: мы сохраняем данные типа `String`, поэтому указываем ее кодировку `utf8` (в этом нет необходимости при сохранении двоичных значений с типом данных `Buffer`).
- Завершает поток случайным образом с вероятностью 5%, передавая значение `null` во внутренний буфер, который играет роль признака `EOF`, то есть конца потока.

Следует отметить, что эта реализация метода `_read()` игнорирует аргумент `size`, поскольку он носит рекомендательный характер. Мы можем без опаски сохранить в буфер все данные, но если метод `push()` вызывается несколько раз, следует проверить, не вернул ли он значение `false`, которое означает, что было достигнуто ограничение `highWaterMark` и нужно отказаться от добавления дополнительных данных.

Теперь класс `RandomStream` готов и его можно использовать. Добавим новый модуль `generateRandom.js`, в котором создадим экземпляр класса `RandomStream` и прочитаем из него данные:

```
const RandomStream = require('./randomStream');
const randomStream = new RandomStream();

randomStream.on('readable', () => {
  let chunk;
```

```
  while((chunk = randomStream.read()) !== null) {
    console.log(`Chunk received: ${chunk.toString()}`);
  }
});
```

Теперь есть все, что нужно для проверки пользовательского потока. Запустите модуль `generateRandom`, как обычно, и на экране появится ряд случайных строк.

Потоки данных для записи

Потоки для записи – это приемники данных. На платформе Node.js они реализуются с помощью абстрактного класса `Writable`, доступного в модуле `stream`.

Запись в поток

Передача данных в поток для записи не требует особых усилий, достаточно вызвать метод `write()`, который имеет следующую сигнатуру:

```
writable.write(chunk, [encoding], [callback])
```

Аргумент `encoding` является необязательным и может быть указан, если `chunk` является значением типа `String` (по умолчанию получает значение `utf8` и игнорируется, если `chunk` является значением типа `Buffer`). Функция `callback` вызывается сразу после передачи данных `chunk` в целевой ресурс и тоже является необязательным аргументом.

Чтобы известить поток об окончании данных, следует вызвать метод `end()`:

```
writable.end([chunk], [encoding], [callback])
```

В вызов этого метода можно передать последний фрагмент данных (`chunk`). В этом случае функция `callback` вызывается подобно обработчику события `finish`, которое генерируется после передачи всех записанных данных в целевой ресурс.

Теперь рассмотрим практический пример, создав небольшой HTTP-сервер, который выводит случайную последовательность строк:

```
const Chance = require('chance');
const chance = new Chance();

require('http').createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});        //[1]
  while(chance.bool({likelihood: 95})) {                     //[2]
    res.write(chance.string() + '\n');                       //[3]
  }
  res.end('\nThe end...\n');                                 //[4]
  res.on('finish', () => console.log('All data was sent'));  //[5]
}).listen(8080, () => console.log('Listening on http://localhost:8080'));
```

Этот HTTP-сервер записывает строки в объект `res`, который является экземпляром класса `http.ServerResponse` и одновременно потоком для записи. Происходящее описывается ниже.

1. Сначала записывается HTTP-заголовок ответа. Обратите внимание, что метод `writeHead()` не является частью интерфейса потока данных – это вспомогательный метод, предоставляемый классом `http.ServerResponse`.
2. Затем выполняется цикл, который завершается с вероятностью 5% (метод `chance.bool()` возвращает `true` в 95% случаев).

3. В теле цикла в поток записывается случайная строка.
4. После выхода из цикла вызывается метод `end()` потока, оповещающий его об окончании записи данных. Кроме того, в этот метод передается заключительная строка.
5. И наконец, регистрируется обработчик события `finish`, который будет вызван после передачи всех данных в сокет.

Теперь можно запустить этот небольшой модуль `entropyServer.js`. Для проверки сервера откройте страницу `http://localhost:8080` в браузере или введите в терминале следующую команду:

```
curl localhost:8080
```

В этот момент сервер должен начать отправлять случайные строки HTTP-клиенту (имейте в виду, что некоторые браузеры могут буферизировать данные, что сделает работу потоков не слишком наглядной).

Обратное давление

Подобно трубопроводам, в которых текут жидкости, потоки данных в Node.js также имеют узкие места, которые проявляются, когда данные записываются быстрее, чем поток может передавать их. Эту проблему можно ослабить с помощью буферизации входных данных, но если поток не обеспечивает обратной связи с пишущим кодом, могут возникать ситуации, когда объем данных, накапливающихся во внутреннем буфере, постоянно растет, что приводит к нежелательным затратам памяти.

Для предотвращения такой ситуации метод `writable.write()` возвращает значение `false`, если размер внутреннего буфера превышает ограничение `highWaterMark`. Потоки для записи имеют свойство `highWaterMark`, ограничивающее размер внутреннего буфера, после достижения которого метод `write()` начинает возвращать значение `false`, указывающее, что приложение должно приостановить запись. Когда буфер пустеет, генерируется событие `drain`, оповещающее о возможности продолжить запись. Этот механизм называется **обратным давлением**.

Понятие обратного давления в равной степени относится и к потокам для чтения. В них также наблюдается эффект обратного давления, который возникает, когда метод `push()`, вызываемый в методе `_read()`, возвращает значение `false`. Но эта проблема относится к реализации потоков, поэтому с ней реже приходится сталкиваться.

Давайте посмотрим, как учесть обратное давление в потоках для записи, изменив созданный ранее модуль `entropyServer`:

```
const Chance = require('chance');
const chance = new Chance();

require('http').createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});

  function generateMore() {                          //[1]
  while(chance.bool({likelihood: 95})) {
    let shouldContinue = res.write(
    chance.string({length: (16 * 1024) - 1})         //[2]
  );
    if(!shouldContinue) {                            //[3]
```

```
      console.log('Backpressure');
      return res.once('drain', generateMore);
    }
  }
  res.end('\nThe end...\n',() => console.log('All data was sent'));
}
generateMore();
}).listen(8080, () => console.log('Listening on http://localhost:8080'));
```

Наиболее важные аспекты поясняются ниже.

1. Основная логика заключена в функции `generateMore()`.
2. Чтобы повысить вероятность появления эффекта обратного давления, размер фрагмента данных был увеличен до 16 кБ – 1 Б, что максимально приблизило его к ограничению `highWaterMark` по умолчанию.
3. После записи фрагмента выполняется проверка значения, возвращаемого вызовом `res.write()`. Значение `false` означает, что внутренний буфер полон и следует прекратить отправку данных. В этом случае выполняются выход из функции и регистрация еще одного цикла записи, ожидающего события `drain`.

Если теперь снова запустить сервер, а затем послать ему запрос с помощью `curl`, высока вероятность появления эффекта обратного давления, поскольку сервер генерирует данные значительно быстрее, чем сокет может их обрабатывать.

Реализация потоков для записи

Теперь можно реализовать поток для записи, унаследовав прототип `stream.Writable` и реализовав метод `_write()`. Осуществим это прямо сейчас, обсуждая детали по пути.

Создадим поток для записи, получающий объекты следующего формата:

```
{
  path: <путь к файлу>
  content: <строка или буфер>
}
```

Для каждого из этих объектов поток должен сохранять содержимое `content` в файл, созданный по заданному пути `path`. Очевидно, что входными данными этого потока являются объекты, а не строки или буферы, то есть поток должен работать в объектном режиме.

Создадим модуль `toFileStream.js`:

```
const stream = require('stream');
const fs = require('fs');
const path = require('path');
const mkdirp = require('mkdirp');

class ToFileStream extends stream.Writable {
  constructor() {
    super({objectMode: true});
  }

  _write (chunk, encoding, callback) {
    mkdirp(path.dirname(chunk.path), err => {
      if (err) {
        return callback(err);
      }
```

```
      fs.writeFile(chunk.path, chunk.content, callback);
    });
  }
}
module.exports = ToFileStream;
```

Первым делом загружаются все используемые зависимости. Обратите внимание на модуль `mkdirp`, который, как упоминалось в предыдущих главах, необходимо установить с помощью NPM.

Затем создается класс нового потока, наследующий `stream.Writable`.

Мы должны вызвать родительский конструктор, чтобы инициализировать его внутреннее состояние. Кроме того, объекту передается параметр, указывающий, что поток будет работать в объектном режиме (`objectMode: true`). Ниже перечислены другие необязательные параметры, принимаемые `stream.Writable`:

- `highWaterMark` (значение по умолчанию 16 КБ): ограничение, управляющее обратным давлением;
- `decodeStrings` (значение по умолчанию `true`): включает автоматическое декодирование строк в двоичные буферы перед передачей в метод `_write()`. Этот параметр игнорируется в объектном режиме.

Далее следует реализация метода `_write()`. Как видите, этот метод принимает фрагмент данных, кодирует его (это имеет смысл только в двоичном режиме и когда параметр `decodeStrings` имеет значение `false`). Кроме того, этот метод принимает функцию `callback`, которая должна вызываться по завершении операции. Здесь нет необходимости передавать результат операции, но, если потребуется, можно передать ошибку, что вызовет возбуждение потоком события `error`.

Теперь создадим новый модуль `writeToFile.js` для проверки вновь созданного потока и выполним несколько операций записи в поток:

```
const ToFileStream = require('./toFileStream.js');
const tfs = new ToFileStream();

tfs.write({path: "file1.txt", content: "Hello"});
tfs.write({path: "file2.txt", content: "Node.js"});
tfs.write({path: "file3.txt", content: "Streams"});
tfs.end(() => console.log("All files created"));
```

Итак, мы создали и использовали первый пользовательский поток для записи. Запустите новый модуль, чтобы проверить его вывод. В результате будут созданы три новых файла.

Дуплексные потоки данных

Дуплексный поток – это поток, поддерживающий чтение и запись. Такие потоки могут пригодиться для создания сущностей, которые могут играть роль источника и приемника данных, таких как сетевые сокеты. Дуплексные потоки наследуют методы классов `stream.Readable` и `stream.Writable`, поэтому в них нет ничего нового. Это означает, что можно использовать методы `read()` и `write()`, а также обрабатывать события `readable` и `drain`.

Чтобы создать свой дуплексный поток, необходимо реализовать методы `_read()` и `_write()`. Параметры конструктора `Duplex()` внутренне направляются непосредственно в конструкторы потоков для чтения и записи. Это те же параметры, что были

рассмотрены в предыдущих разделах, кроме `allowHalfOpen` (значение по умолчанию `true`), значение `false`, в котором вызовет завершение работы обеих частей (для чтения и для записи) потока, если одна из них будет завершена.

Чтобы получить дуплексный поток, работающий в объектном режиме в одну сторону и в двоичном режиме в другую, необходимо вручную установить следующие свойства в конструкторе потока:

```
this._writableState.objectMode
this._readableState.objectMode
```

Преобразующие потоки данных

Преобразующие потоки – это специальный вид дуплексных потоков, предназначенный для преобразования данных.

В обычном дуплексном потоке нет никакой непосредственной связи между данными, которые читаются из потока, и данными, которые записываются в него (по крайней мере, сам поток не имеет к этому никакого отношения). Примером может послужить TCP-сокет, который просто отправляет данные в удаленный узел и получает данные из удаленного узла, и ему ничего не известно о взаимоотношениях между входными и выходными данными. Схема на рис. 5.4 иллюстрирует перемещение данных в дуплексном потоке.

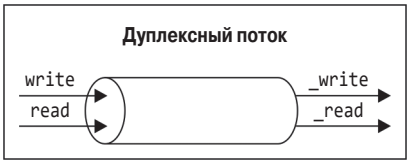

Рис. 5.4 ❖ Схема работы простого дуплексного потока

Преобразующие потоки, напротив, выполняют некоторое преобразование каждого фрагмента данных, получаемого со стороны записи, и передают преобразованные данные на сторону для чтения. Схема на рис. 5.5 иллюстрирует передачу данных преобразующим потоком.

Рис. 5.5 ❖ Схема работы преобразующего потока

Внешне интерфейс преобразующих потоков ничем не отличается от интерфейса дуплексных потоков. Но если при создании нового дуплексного потока необходимо реализовать только методы `_read()` и `_write()`, то при создании нового преобразующего потока необходимо также реализовать методы `_transform()` и `_flush()`.

Рассмотрим создание нового преобразующего потока на примере.

Реализация преобразующих потоков

Реализуем преобразующий поток, заменяющий все вхождения заданной строки. Для этого создадим новый модуль `replaceStream.js`. Перейдем непосредственно к его реализации:

```
const stream = require('stream');
const util = require('util');

class ReplaceStream extends stream.Transform {
  constructor(searchString, replaceString) {
    super();
    this.searchString = searchString;
    this.replaceString = replaceString;
    this.tailPiece = '';
  }

  _transform(chunk, encoding, callback) {
    const pieces = (this.tailPiece + chunk)                    //[1]
      .split(this.searchString);
    const lastPiece = pieces[pieces.length - 1];
    const tailPieceLen = this.searchString.length - 1;

    this.tailPiece = lastPiece.slice(-tailPieceLen);           //[2]
    pieces[pieces.length - 1] = lastPiece.slice(0,-tailPieceLen);

    this.push(pieces.join(this.replaceString));                //[3]
    callback();
  }

  _flush(callback) {
    this.push(this.tailPiece);
    callback();
  }
}

module.exports = ReplaceStream;
```

Как всегда, модуль начинается с загрузки зависимостей. В этот раз сторонние модули не используются.

Затем создается новый класс, наследующий базовый класс `stream.Transform`. Конструктор класса принимает два аргумента: `searchString` и `replaceString`. Нетрудно догадаться, что они определяют искомый текст и строку замены. Кроме того, в нем инициализируется внутренняя переменная `tailPiece`, которая будет использоваться методом `_transform()`.

А теперь рассмотрим метод `_transform()`, являющийся основой класса. Метод `_transform()` имеет практически ту же сигнатуру, что и метод `_write()` потока для записи, но вместо записи данных в целевой ресурс он помещает их во внутренний буфер вызовом метода `this.push()`, как это делалось в методе `_read()` потока для чтения. Это подтверждает связь двух сторон потока `Transform`.

Метод `_transform()` класса `ReplaceStream` реализует основной алгоритм. Поиск и замена строки в буфере являются простой задачей, но в данном случае осуществляется потоковая передача данных и возможна ситуация, когда искомая строка может оказаться разделена на несколько фрагментов. Ниже приводятся пояснения к коду.

1. Алгоритм разбивает фрагменты, используя `searchString` в качестве разделителя.
2. Затем берет последний элемент сформированного массива и извлекает последние `searchString.length - 1` символов. Результат присваивается переменной `tailPiece` и добавляется перед следующим фрагментом данных.
3. Наконец, все фрагменты, полученные от метода `split()`, объединяются с использованием разделителя `replaceString` и помещаются во внутренний буфер.

Когда запись в поток завершается, в переменной `tailPiece` могут оставаться данные, не переданные во внутренний буфер. Эту проблему решает метод `_flush()` – он вызывается непосредственно перед завершением потока, когда остается последняя возможность выполнить заключительные операции или вставить в поток какие-либо данные.

Метод `_flush()` принимает единственный параметр, функцию обратного вызова, которая должна быть вызвана после выполнения всех заключительных операций. Этот метод завершает работу класса `ReplaceStream`.

Теперь можно проверить работу нового потока. Для этого создадим еще один модуль `replaceStreamTest.js`, который записывает данные, а затем читает преобразованный результат:

```
const ReplaceStream = require('./replaceStream');

const rs = new ReplaceStream('World', 'Node.js');
rs.on('data', chunk => console.log(chunk.toString()));

rs.write('Hello W');
rs.write('orld!');
rs.end();
```

Чтобы несколько усложнить задачу, разобьем искомую строку (в данном случае `World`) на два фрагмента, а затем в непрерывном режиме прочитаем данные из потока и выведем преобразованные фрагменты. В результате должен появиться следующий вывод:

```
Hel
lo Node.js
!
```

Стоит упомянуть еще один, пятый вид потоков – `stream.PassThrough`. В отличие от других классов потоков, `PassThrough` не является абстрактным, и его экземпляр можно создать непосредственно, без необходимости реализовать какие-либо методы. Фактически это обычный преобразующий поток, который выводит фрагменты данных без применения каких-либо преобразований.

Соединение потоков с помощью конвейеров

Идея конвейера была введена в Unix Дугласом Маклороем (Douglas Mcllroy). Конвейер позволяет связать вывод одной программы с вводом другой. Взгляните на следующую команду:

```
echo Hello World! | sed s/World/Node.js/g
```

Здесь команда `echo` запишет строку `Hello World!` в стандартный вывод, который будет перенаправлен в стандартный ввод команды `sed` (благодаря оператору конвейера `|`), а затем команда `sed` заменит все слова `World` на `Node.js` и выведет результат в свой стандартный вывод (на этот раз в консоль).

Аналогично можно соединять потоки данных в Node.js, используя метод `pipe()` потока для чтения, который имеет следующий интерфейс:

```
readable.pipe(writable, [options])
```

Метод `pipe()` извлекает данные из потока `readable` и записывает их в поток для `writeable`. Кроме того, поток `writeable` автоматически завершается, когда поток `readable` сгенерирует событие `end` (если в аргументе `options` не передать параметра `{end: false}`). Метод `pipe()` возвращает переданный ему поток `writeable`, что позволяет создавать цепочки вызовов, если этот поток является и доступным для чтения (например, дуплексный или преобразующий поток).

Соединение двух потоков создает эффект *всасывания*, который обеспечивает автоматическое перетекание данных в поток `writeable`, благодаря чему отпадает необходимость вызывать методы `read()` или `write()`. Но самое главное, что не нужно контролировать обратное давление, поскольку это тоже делается автоматически.

Для демонстрации создадим новый модуль `replace.js`, который принимает текстовый поток из стандартного ввода, применяет к нему преобразование *замены* и передает результат в стандартный вывод:

```
const ReplaceStream = require('./replaceStream');
process.stdin
  .pipe(new ReplaceStream(process.argv[2], process.argv[3]))
  .pipe(process.stdout);
```

В примере данные передаются из стандартного ввода в поток `ReplaceStream`, а затем в стандартный вывод. Для проверки этого небольшого приложения воспользуемся конвейером Unix, чтобы отправить данные в стандартный ввод, как показано ниже:

```
echo Hello World! | node replace World Node.js
```

В результате должен появиться следующий вывод:

```
Hello Node.js
```

Этот простой пример демонстрирует, что потоки (и, в частности, текстовые потоки) являются универсальным интерфейсом, а конвейеры позволяют соединять их, как по волшебству.

События `error` не распространяются через конвейер автоматически. Например, следующий конвейер:

```
stream1
  .pipe(stream2)
  .on('error', function() {});
```

будет перехватывать только ошибки в потоке `stream2`, поскольку обработчик подключается именно к этому потоку. То есть для обработки ошибок в потоке `stream1` необходимо подключить еще один обработчик. Позже вы увидите шаблон, помогающий обойти это неудобство. Также следует отметить, что если в принимающем потоке возникает ошибка, он автоматически отсоединяется от передающего потока, что приведет к разрушению конвейера.

Библиотеки through и from для работы с потоками

Описанный выше способ создания пользовательских потоков данных не совсем соответствует принципам *Node*. Действительно, наследование потоков от базового

класса потоков нарушает принцип малой общедоступной области и требует определенного объема стереотипного кода. Это не означает, что потоки плохо спроектированы. В действительности, поскольку потоки данных являются частью ядра Node.js, они должны быть максимально гибкими и поддерживать возможность расширения в пользовательских модулях для достижения широкого диапазона конкретных целей.

Но в большинстве случаев нам не нужна вся широта возможностей, которую дает наследование прототипов, – достаточно простого и наглядного способа определения новых потоков. Поэтому сообществом Node.js были созданы свои решения этой проблемы. Отличным примером является небольшая библиотека through2 (https://npmjs.org/package/through2), упрощающая создание потоков преобразования. Библиотека through2 позволяет создать новый преобразующий поток вызовом простой функции:

```
const transform = through2([options], [_transform], [_flush])
```

Аналогично библиотека from2 (`https://npmjs.org/package/from2`) позволяет легко и просто создавать потоки для чтения, например:

```
const readable = from2([options], _read)
```

Преимущества использования этих небольших библиотек станут очевидны, как только мы начнем их использовать в остальной части этой главы.

Пакеты `through` (https://npmjs.org/package/through) и `from` (https://npmjs.org/package/from) являются оригинальными библиотеками, основанными на версии интерфейса потоков Streams1.

Управление асинхронным выполнением с помощью потоков данных

Приведенные примеры со всей очевидностью продемонстрировали полезность потоков не только для ввода/вывода, но и для обработки любых данных. Однако их достоинства не ограничиваются простым изменением внешнего вида кода – потоки данных можно с успехом использовать для управления выполнением, как будет показано в этом разделе.

Последовательное выполнение

По умолчанию потоки данных обрабатывают свои данные последовательно. Например, функция `_transform()` преобразующего потока никогда не будет вызвана для обработки следующего фрагмента данных, пока предыдущий ее вызов не обратится к функции `callback()`. Это важное свойство потоков данных имеет решающее значение для обработки всех фрагментов в правильном порядке, но его также можно использовать для превращения потоков в элегантную альтернативу традиционной модели управления выполнением.

Часто наглядный пример предпочтительнее пространных объяснений, поэтому перейдем непосредственно к демонстрации использования потоков данных для организации асинхронного и последовательного выполнения заданий. Реализуем функцию, объединяющую содержимое группы файлов, полученных на входе, с соблюдением порядка их следования. Создадим для этого новый модуль `concatFiles.js`, начав с зависимостей:

```
const fromArray = require('from2-array');
const through = require('through2');
const fs = require('fs');
```

Чтобы упростить создание преобразующих потоков, используем библиотеку through2, а поток для чтения из массива объектов создадим с помощью библиотеки from2-array.

Далее определим функцию concatFiles():

```
function concatFiles(destination, files, callback) {
  const destStream = fs.createWriteStream(destination);
  fromArray.obj(files)                          //[1]
    .pipe(through.obj((file, enc, done) => {    //[2]
      const src = fs.createReadStream(file);
      src.pipe(destStream, {end: false});
      src.on('end', done)                       //[3]
    }))
    .on('finish', () => {                       //[4]
      destStream.end();
      callback();
    });
}
module.exports = concatFiles;
```

Она последовательно перебирает элементы массива files, предварительно преобразовав его в поток данных. Ключевые аспекты функции поясняются далее.

1. Во-первых, с помощью from2-array из массива files создается поток для чтения.
2. Затем создается поток through (Transform) для последовательной обработки файлов. Для каждого файла создается свой поток для чтения, который подключается к потоку destStream, представляющему выходной файл. Параметр {end: false} в вызове метода pipe() гарантирует, что поток destStream не будет закрыт после завершения чтения исходного файла.
3. Когда все содержимое исходного файла будет записано в destStream, вызывается функция done, переданная в through.obj как обработчик завершения записи, которая в данном случае выполняет переход к следующему файлу.
4. После завершения обработки всех файлов генерируется событие finish; в ответ на него мы закрываем поток destStream и вызываем функцию callback(), переданную функции concatFiles(), которая оповещает о завершении всей операции.

А теперь проверим только что созданный модуль. Для этого создадим новый файл concat.js:

```
const concatFiles = require('./concatFiles');
concatFiles(process.argv[2], process.argv.slice(3), () => {
  console.log('Files concatenated successfully');
});
```

и запустим приведенную выше программу, передав ей в первом аргументе имя выходного файла и список объединяемых файлов, например:

```
node concat allTogether.txt file1.txt file2.txt
```

В результате должен быть создан новый файл с именем allTogether.txt, включающий содержимое файлов file1.txt и file2.txt в указанном порядке.

В функции `concatFiles()` мы смогли реализовать последовательное выполнение асинхронных операций с помощью одних только потоков данных. Как было показано в *главе 3 «Шаблоны асинхронного выполнения с обратными вызовами»*, чтобы реализовать то же самое на чистом JavaScript или с использованием библиотек, таких как `async`, потребовалось бы использовать итератор. Теперь у нас появился еще один способ достижения того же результата, который, как видите, также является весьма компактным и элегантным.

Шаблон
Использование потока данных или их комбинации упрощает последовательное выполнение набора асинхронных заданий.

Неупорядоченное параллельное выполнение

Мы только что убедились, что потоки данных обрабатывают фрагменты строго в определенном порядке, но иногда это свойство потоков может замедлить обработку, потому что мешает максимально использовать асинхронные возможности платформы Node.js. Если асинхронная операция обработки каждого фрагмента требует значительного времени, иногда общую производительность можно увеличить, запуская такие операции параллельно. Конечно, этот шаблон можно применить, только когда фрагменты данных никак не связаны между собой, с чем часто приходится сталкиваться при работе с объектными потоками данных, но очень редко при использовании двоичных потоков.

Предупреждение
Потоки данных нельзя использовать для параллельной обработки, когда важен порядок следования фрагментов.

Для параллельной обработки данных потоком `Transform` можно воспользоваться шаблонами из *главы 3 «Шаблоны асинхронного выполнения с обратными вызовами»*, внеся определенные коррективы и приспособив их к работе с потоками данных. Посмотрим, как это сделать.

Реализация неупорядоченной параллельной обработки

Перейдем сразу к примеру. Для этого создадим модуль `parallelStream.js` и определим универсальный поток `Transform`, выполняющий заданную функцию преобразования параллельно:

```
const stream = require('stream');

class ParallelStream extends stream.Transform {
  constructor(userTransform) {
    super({objectMode: true});
    this.userTransform = userTransform;
    this.running = 0;
    this.terminateCallback = null;
  }

  _transform(chunk, enc, done) {
    this.running++;
    this.userTransform(chunk, enc, this.push.bind(this),
                       this._onComplete.bind(this));
    done();
```

```
  }

  _flush(done) {
    if(this.running > 0) {
      this.terminateCallback = done;
    } else {
      done();
    }
  }

  _onComplete(err) {
    this.running--;
    if(err) {
      return this.emit('error', err);
    }
    if(this.running === 0) {
      this.terminateCallback && this.terminateCallback();
    }
  }

}

module.exports = ParallelStream;
```

Проанализируем этот новый класс. Как видите, его конструктор принимает функцию `userTransform()` и сохраняет ее в переменной экземпляра. Кроме того, он вызывает родительский конструктор и включает объектный режим работы.

Теперь перейдем к методу `_transform()`. В этом методе мы увеличиваем счетчик выполняемых заданий, вызываем функцию `userTransform()` и, наконец, сообщаем о завершении текущего этапа преобразования, вызывая функцию `done()`. Вся хитрость в том, что мы не ждем завершения функции `userTransform()` перед вызовом функции `done()`, – это происходит немедленно. С другой стороны, функция `userTransform()` получает специальную функцию обратного вызова – метод `this._onComplete()`, оповещающий о завершении функции `userTransform()`.

Метод `_flush()` вызывается непосредственно перед завершением потока, в котором мы откладываем генерацию события `finish`, отказываясь от вызова функции `done()` и присваивая ее переменной `this.terminateCallback`. Чтобы разобраться в механизме завершения потока, рассмотрим метод `_onComplete()`. Этот последний метод вызывается после завершения каждого асинхронного задания. Он проверяет счетчик выполняющихся заданий и, если он равен нулю, вызывает функцию `this.terminateCallback()`, которая завершит поток, сгенерировав событие `finish`, отложенное в методе `_flush()`.

Только что созданный класс `ParallelStream` упрощает создание потока `Transform`, выполняющего задания параллельно, но при его использовании следует учитывать, что порядок обработки элементов не сохраняется. Фактически асинхронные операции могут закончить обработку и вернуть данные в любой момент, независимо от того, когда они были запущены. Очевидно, что эта особенность делает его непригодным для работы с двоичными потоками, где порядок данных обычно имеет значение, но он наверняка пригодится для многих объектных потоков.

Реализация приложения мониторинга состояния URL-адреса

А теперь применим класс `ParallelStream` для реализации конкретной задачи. Предположим, что требуется разработать простую службу, осуществляющую мониторинг

состояния большого количества URL-адресов. Пусть все эти URL-адреса находятся в одном файле и разделены символами перевода строки.

Потоки данных предлагают очень эффективное и элегантное решение этой задачи, особенно если использовать класс `ParallelStream` для параллельной проверки URL-адресов.

Создадим для этого простого приложения новый модуль `checkUrls.js`:

```
const fs = require('fs');
const split = require('split');
const request = require('request');
const ParallelStream = require('./parallelStream');

fs.createReadStream(process.argv[2])                    //[1]
  .pipe(split())                                        //[2]
  .pipe(new ParallelStream((url, enc, push, done) => {  //[3]
    if(!url) return done();
    request.head(url, (err, response) => {
      push(url + ' is ' + (err ? 'down' : 'up') + '\n');
      done();
    });
  }))
  .pipe(fs.createWriteStream('results.txt'))            //[4]
  .on('finish', () => console.log('All urls were checked'));
```

Как видите, применение потоков делает программный код проще и элегантнее. Давайте посмотрим, как он работает.

1. Во-первых, из входного файла создается поток для чтения.
2. В конвейере содержимое этого файла обрабатывается функцией `split()` (https://npmjs.org/package/split) – поток `Transform`, который выделяет каждую строку в отдельный фрагмент.
3. Затем, для проверки URL-адреса, используется `ParallelStream`. Для этого вызывается метод `head`, который отправляет запрос и ожидает ответа. Функция обратного вызова передает результат операции в поток.
4. И наконец, все результаты поступают по конвейеру в файл `results.txt`.

Теперь можно запустить модуль `checkUrls` следующей командой:

```
node checkUrlsurlList.txt
```

Вот как выглядит содержимое файла `urlList.txt` со списком URL-адресов, использованных нами:

- http://www.mariocasciaro.me;
- http://loige.co;
- http://thiswillbedownforsure.com.

По завершении команды мы обнаружили файл `results.txt`, содержащий такие результаты:

```
http://thiswillbedownforsure.com is down
http://loige.co is up
http://www.mariocasciaro.me is up
```

Велика вероятность, что порядок следования результатов будет отличаться от порядка следования URL-адресов во входном файле. Это явно свидетельствует о том,

что поток выполняет задания параллельно, без учета порядка поступления фрагментов данных в поток.

Из любопытства можно заменить `ParallelStream` обычным потоком `through2` и сравнить их поведение и производительность (попробуйте сделать это самостоятельно). Мы обнаружили, что код с использованием `through2` выполняется дольше, поскольку URL-адреса проверяются последовательно, но при этом будет сохраняться порядок результатов в файл `results.txt`.

Неупорядоченное ограниченное параллельное выполнение

Если запустить приложение `checkUrls`, передав ему файл с тысячами или миллионами URL-адресов, это неминуемо приведет к сбою. Приложение сразу же создаст неконтролируемое количество соединений, отправив параллельно значительное количество данных, что способно вызвать нестабильность приложения и даже всей системы. Как уже упоминалось, чтобы не потерять контроль над ростом нагрузки и использованием ресурсов, необходимо ограничить число заданий, обрабатываемых параллельно.

Рассмотрим, как этого можно достичь с помощью потоков данных на примере модуля `limitedParallelStream.js` – адаптированной версии модуля `parallelStream.js` из предыдущего раздела.

Начнем его анализ с конструктора (изменившиеся части кода выделены жирным):

```
class LimitedParallelStream extends stream.Transform {
  constructor(concurrency, userTransform) {
    super({objectMode: true});
    this.concurrency = concurrency;
    this.userTransform = userTransform;
    this.running = 0;
    this.terminateCallback = null;
    this.continueCallback = null;
  }
//...
```

Мы добавили параметр `concurrency`, ограничивающий количество параллельных заданий, и на этот раз будут использоваться две функции обратного вызова: одна для метода `_transform` (`continueCallback`) и другая для метода `_flush` (`terminateCallback`).

Далее следует код метода `_transform()`:

```
_transform(chunk, enc, done) {
  this.running++;
  this.userTransform(chunk, enc, this._onComplete.bind(this));
  if(this.running < this.concurrency) {
    done();
  } else {
    this.continueCallback = done;
  }
}
```

Эта версия метода `_transform()` проверяет количество выполняющихся заданий, прежде чем вызвать `done()` и перейти к обработке следующего элемента. Если уже достигнуто максимальное число обрабатываемых потоков данных, мы просто сохраня-

ем функцию обратного вызова done() в переменной continueCallback, чтобы вызвать ее сразу после завершения текущего задания.

Метод _flush() остался точно таким же, как в классе ParallelStream, поэтому перейдем непосредственно к реализации метода _onComplete():

```
_onComplete(err) {
  this.running--;
  if(err) {
    return this.emit('error', err);
  }
  const tmpCallback = this.continueCallback;
  this.continueCallback = null;
  tmpCallback && tmpCallback();
  if(this.running === 0) {
    this.terminateCallback && this.terminateCallback();
  }
}
```

При завершении любого задания вызывается сохраненная прежде функция обратного вызова continueCallback(), которая разблокирует поток данных и запускает обработку следующего элемента.

Вот и весь модуль limitedParallelStream. Теперь его можно использовать в модуле checkUrls вместо parallelStream, и параллельная обработка заданий будет ограничена заданным значением.

Упорядоченное параллельное выполнение

Созданные ранее параллельные потоки данных возвращают результаты не по порядку, но иногда это не приемлемо. Действительно, в некоторых ситуациях необходимо, чтобы все фрагменты выводились в порядке следования входных данных. Но не надо терять надежду, поскольку все же имеется возможность и в этих случаях параллельно выполнять функцию преобразования. Для этого требуется лишь отсортировать возвращаемые результаты в порядке следования входных данных.

Этот подход предполагает использование буфера для переупорядочения фрагментов по мере их поступления от выполнившихся заданий. Мы не будем рассматривать здесь реализацию такого потока, поскольку она слишком объемная для данной книги. Вместо этого мы воспользуемся одним из пакетов NPM, предназначенных именно для этой цели, а именно through2-parallel (https://npmjs.org/package/through2-parallel).

Чтобы быстро проверить упорядоченное параллельное выполнение, внесем изменения в существующий модуль checkUrls. Предположим, что мы должны вернуть результаты параллельной обработки в том же порядке, в каком следуют URL-адреса во входном файле. Для этого можно воспользоваться пакетом through2-parallel:

```
//...
const throughParallel = require('through2-parallel');

fs.createReadStream(process.argv[2])
  .pipe(split())
  .pipe(throughParallel.obj({concurrency: 2},(url, enc, done) => {
      //...
    })
```

```
)
.pipe(fs.createWriteStream('results.txt'))
.on('finish', () => console.log('All urls were checked'));
```

Как видите, интерфейс `through2-parallel` очень похож на интерфейс `through2`. Единственная разница заключается в возможности определения ограничения параллельной обработки для функции преобразования. Если выполнить новую версию `checkUrls`, можно убедиться, что результаты в файле `results.txt` следуют в том же порядке, что и URL-адреса во входном файле.

Обратите внимание: даже притом, что порядок результатов совпадает с порядком входных данных, асинхронные задания выполняются параллельно и могут завершаться в любом порядке.

На этом мы завершаем анализ управления асинхронным выполнением с применением потоков данных. Следующий раздел будет посвящен шаблонам конвейерной обработки.

Шаблоны конвейерной обработки

Подобно настоящим трубопроводам, в которых текут жидкости, потоки данных в Node.js также могут иметь разнообразные соединения. Действительно, два различных потока можно объединить в один поток, разделить один поток на несколько или перенаправить поток, исходя из некоторого условия. В этом разделе мы рассмотрим наиболее важные приемы комбинирования, которые можно применять к потокам данных в Node.js.

Объединение потоков данных

В этой главе неоднократно подчеркивалось, что потоки образуют простую инфраструктуру для модульной организации и повторного использования кода, но остался неосвещенным еще один, последний фрагмент мозаики: как заключить в модуль и повторно использовать весь конвейер? Как объединить несколько потоков, чтобы внешне они выглядели как один? Схема на рис. 5.6 иллюстрирует, что это означает.

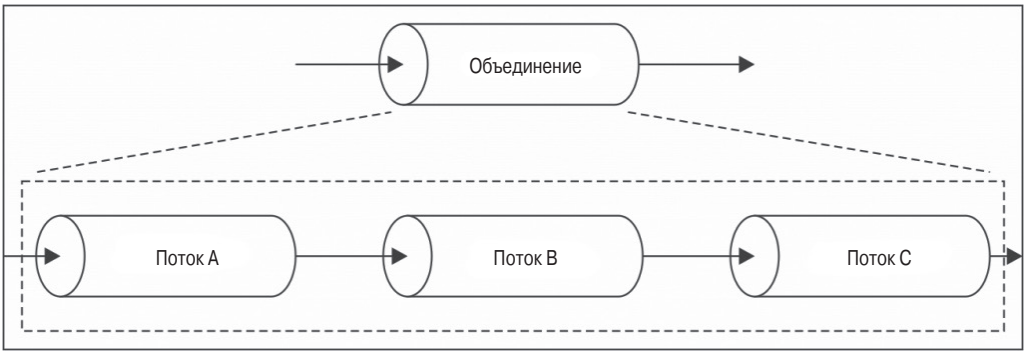

Рис. 5.6 ❖ Объединение потоков данных

Схема на рис. 5.6 помогает понять, как это работает:

- при записи в объединенный поток запись выполняется в первый поток в конвейере;
- при чтении из объединенного потока чтение выполняется из последнего потока в конвейере.

Объединенный поток обычно является дуплексным потоком, который составляется путем подключения первого потока к концу для записи и последнего – к концу для чтения.

Для создания дуплексного потока из двух различных потоков, одного для записи и одного для чтения, можно использовать модуль npm, например `duplexer2` (https://npmjs.org/package/duplexer2).

Но этого недостаточно, поскольку еще одним важным требованием к объединенному потоку является перехват им всех ошибок, возникающих внутри конвейера. Как упоминалось ранее, событие `error` не распространяется автоматически по конвейеру, поэтому для надлежащей обработки ошибок (а она необходима) придется явно подключить обработчик события `error` к каждому из потоков. Однако если объединенный поток действительно должен выглядеть как черный ящик, это исключает доступ к любому из потоков в середине конвейера, поэтому важно, чтобы объединенный поток выступал в качестве концентратора всех ошибок, возникающих в любом из составляющих его потоков.

Напомним, что объединенный поток обладает двумя основными преимуществами:

- его можно распространять как черный ящик, скрыв внутреннюю реализацию конвейера;
- упрощается обработка ошибок, поскольку отпадает необходимость подключать обработчики событий `error` к каждому потоку в конвейере – достаточно подключить его к объединенному потоку.

Объединение потоков является достаточно универсальной и общепринятой практикой, поэтому, в отсутствие каких-либо необычных требований, можно использовать для этого одно из уже существующих решений, например `multipipe` (https://www.npmjs.org/package/multipipe) или `combine-stream` (https://www.npmjs.org/package/combine-stream), и это только два из множества имеющихся.

Реализация объединенного потока данных

Рассмотрим простой пример объединения двух потоков преобразования:

- один сжимает и шифрует данные;
- второй расшифровывает и распаковывает их.

Эти потоки легко можно создать с помощью библиотек, таких как `multipipe`, объединив несколько потоков, доступных в ядре (файл `combinedStreams.js`):

```
const zlib = require('zlib');
const crypto = require('crypto');
const combine = require('multipipe');

module.exports.compressAndEncrypt = password => {
  return combine(
    zlib.createGzip(), crypto.createCipher('aes192', password)
  );
};
```

```
module.exports.decryptAndDecompress = password => {
  return combine(
    crypto.createDecipher('aes192', password), zlib.createGunzip()
  );
};
```

Эти объединенные потоки можно использовать как черные ящики, например для создания небольшого приложения, которое архивирует файлы, сжимая и шифруя их. Создадим для этого новый модуль `archive.js`:

```
const fs = require('fs');
const compressAndEncryptStream =
  require('./combinedStreams').compressAndEncrypt;

fs.createReadStream(process.argv[3])
  .pipe(compressAndEncryptStream(process.argv[2]))
  .pipe(fs.createWriteStream(process.argv[3] + ".gz.enc"));
```

Предыдущий код можно улучшить, преобразовав конвейер в объединенный поток, на этот раз не для того, чтобы получить еще один черный ящик, а только чтобы воспользоваться преимуществами агрегированной обработки ошибок. Фактически, как уже многократно упоминалось, следующий код перехватывает только ошибки последнего потока:

```
fs.createReadStream(process.argv[3])
  .pipe(compressAndEncryptStream(process.argv[2]))
  .pipe(fs.createWriteStream(process.argv[3] + ".gz.enc"))
  .on('error', err => {
    //Только ошибки последнего потока
    console.log(err);
  });
```

Однако объединение потоков позволяет элегантно решить эту проблему. Изменим код в файле `archive.js`, как показано ниже:

```
const combine = require('multipipe');
const fs = require('fs');
const compressAndEncryptStream =
  require('./combinedStreams').compressAndEncrypt;

combine(
  fs.createReadStream(process.argv[3])
  .pipe(compressAndEncryptStream(process.argv[2]))
  .pipe(fs.createWriteStream(process.argv[3] + ".gz.enc"))
).on('error', err => {
  //эта ошибка может возникнуть в любом из потоков конвейера
  console.log(err);
});
```

Как видите, теперь можно подключить обработчик события `error` непосредственно к объединенному потоку, и он будет вызываться для обработки любых событий `error`, сгенерированных в любом из внутренних потоков.

Чтобы выполнить модуль `archive`, достаточно указать пароль и файл в командной строке:

```
node archive mypassword /path/to/a/file.txt
```

Этот пример демонстрирует важность объединения потоков, поскольку, с одной стороны, данный прием позволяет создавать повторно используемые множества потоков, а с другой – упрощает обработку возникающих в конвейере ошибок.

Ветвление потоков данных

Имеется возможность разделить один поток для чтения на несколько потоков для записи. Это может пригодиться, чтобы отправить одинаковые данные по различным направлениям, например в два разных сокета или в два разных файла. Такая возможность позволяет применить разные преобразования к одним и тем же данным или разделить данные на основе определенных критериев. Схема на рис. 5.7 графически иллюстрирует этот шаблон.

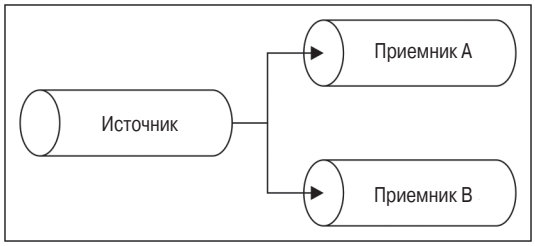

Рис. 5.7 ❖ Шаблон ветвления потока данных

Ветвление потоков Node.js является довольно тривиальной задачей, рассмотрим ее решение на конкретном примере.

Реализация нескольких генераторов контрольных сумм

Создадим небольшую утилиту для вывода хешей `sha1` и `md5` для заданного файла. Создадим новый модуль `generateHashes.js` и начнем с инициализации потоков для получения контрольных сумм:

```
const fs = require('fs');
const crypto = require('crypto');

const sha1Stream = crypto.createHash('sha1'); sha1Stream.setEncoding('base64');

const md5Stream = crypto.createHash('md5');
md5Stream.setEncoding('base64');
```

Пока ничего особенного. Следующий фрагмент модуля создает поток для чтения из файла и делит его на два потока для получения двух других файлов, один из которых будет содержать хеш `sha1`, а второй – контрольную сумму `md5`:

```
const inputFile = process.argv[2];
const inputStream = fs.createReadStream(inputFile);
inputStream
  .pipe(sha1Stream)
  .pipe(fs.createWriteStream(inputFile + '.sha1'));

inputStream
  .pipe(md5Stream)
  .pipe(fs.createWriteStream(inputFile + '.md5'));
```

Все просто, не правда ли? Переменная `InputStream` передается по конвейеру с одной стороны в `sha1Stream`, а с другой стороны в `md5Stream`. Здесь следует пояснить несколько неочевидных моментов:

- потоки `md5Stream` и `sha1Stream` автоматически закроются при завершении потока `inputStream`, если не передать в вызов `pipe()` параметра `{end: false}`;
- две ветви потока будут получать одни и те же фрагменты данных, поэтому следует проявлять повышенную осторожность, чтобы избежать побочных эффектов при выполнении операций с данными, поскольку они воздействуют на обе ветви;
- обратное давление здесь обрабатывается встроенными средствами, и скорость потока `inputStream` определяется скоростью самой медленной ветви!

Слияние потоков данных

Слияние является операцией, обратной ветвлению, и заключается в соединении нескольких потоков для чтения в один поток для записи, как показано на рис. 5.8.

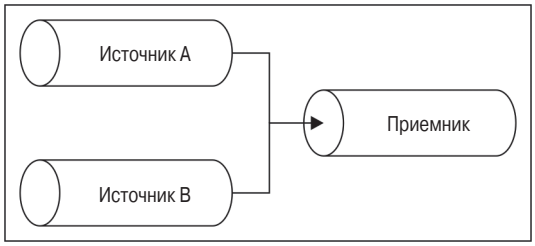

Рис. 5.8 ❖ Шаблон слияния потоков данных

Слияние нескольких потоков в один – в целом довольно простая операция; особого внимания требует только обработка события `end`, так как значение `auto` в параметре `end` вызовет завершение приемника при закрытии любого из источников. Это часто приводит к ошибкам, поскольку другие активные источники продолжают писать в уже завершенный поток. Решение данной проблемы заключается в использовании параметра `{end: false}` для всех источников с последующим вызовом `end()` в принимающем потоке, только когда чтение из всех источников закончено.

Архивирование нескольких каталогов

Для демонстрации напишем небольшую программу, создающую архив с содержимым двух разных каталогов. Для этого понадобятся два NPM-пакета:

- `tar` (https://npmjs.org/package/tar): библиотека для создания тарболлов (tar-архивов);
- `fstream` (https://npmjs.org/package/fstream): библиотека для создания объектных потоков из файлов.

Создадим новый модуль `mergeTar.js` и начнем его определение с инициализации:

```
const tar = require('tar');
const fstream = require('fstream');
const path = require('path');

const destination = path.resolve(process.argv[2]);
```

```
const sourceA = path.resolve(process.argv[3]);
const sourceB = path.resolve(process.argv[4]);
```

Предыдущий код загружает все зависимости и инициализирует переменные, содержащие имена файла архива и двух каталогов (`sourceA` и `sourceB`).

Далее создадим поток `tar` и подсоединим его к приемнику:

```
const pack = tar.Pack();
pack.pipe(fstream.Writer(destination));
```

Теперь инициализируем входящие потоки:

```
let endCount = 0;
function onEnd() {
  if(++endCount === 2) {
    pack.end();
  }
}

const sourceStreamA = fstream.Reader({type: "Directory", path: sourceA})
  .on('end', onEnd);

const sourceStreamB = fstream.Reader({type: "Directory", path: sourceB})
  .on('end', onEnd);
```

Приведенный выше код создает потоки для чтения содержимого двух каталогов (`sourceStreamA` и `sourceStreamB`). Далее к каждому из потоков-источников подключается обработчик события `end`, который закрывает поток `pack` после завершения чтения содержимого обоих каталогов.

И наконец, производится само слияние:

```
sourceStreamA.pipe(pack, {end: false});
sourceStreamB.pipe(pack, {end: false});
```

Здесь оба источника соединяются с потоком `pack` и отключается автоматическое завершение принимающего потока передачей параметра `{end: false}` в обоих вызовах метода `pipe()`.

Теперь простая утилита архивирования готова. Ее работу можно проверить, передав в первом аргументе целевой файл, за которым следуют два архивируемых каталога:

```
node mergeTar dest.tar /path/to/sourceA /path/to/sourceB
```

Завершая этот раздел, следует заметить, что существует еще несколько npm-модулей, упрощающих слияние потоков, например:

- `merge-stream` (https://npmjs.org/package/merge-stream);
- `multistream-merge` (https://npmjs.org/package/multistream-merge).

И в качестве последнего комментария к шаблону слияния потоков напомним, что данные, поступающие по конвейеру в принимающий поток, будут перемешаны случайным образом. Это приемлемо для некоторых видов объектных потоков (что демонстрирует последний пример), но часто нежелательно при работе с двоичными потоками.

Однако имеется еще один вариант этого шаблона, позволяющий выполнять слияние потоков с сохранением упорядоченности данных. Он заключается в последовательном извлечении данных из потоков-источников, когда каждый следующий на-

чинает генерировать фрагменты только после завершения предыдущего (это похоже на *цепочку* из источников). Как всегда, имеется несколько npm-пакетов для реализации такого решения. Одним из них является `multistream` (https://npmjs.org/package/multistream).

Мультиплексирование и демультиплексирование

Существует особый вариант шаблона слияния потоков, предназначенный не столько для обычного объединения нескольких потоков, сколько для организации общего канала вывода данных из нескольких потоков. Эта операция основана на другой идее, так как потоки-источники в общем канале остаются логически отделенными, что позволяет разделить потоки после достижения данными выходного конца общего канала. Схема на рис. 5.9 иллюстрирует этот случай.

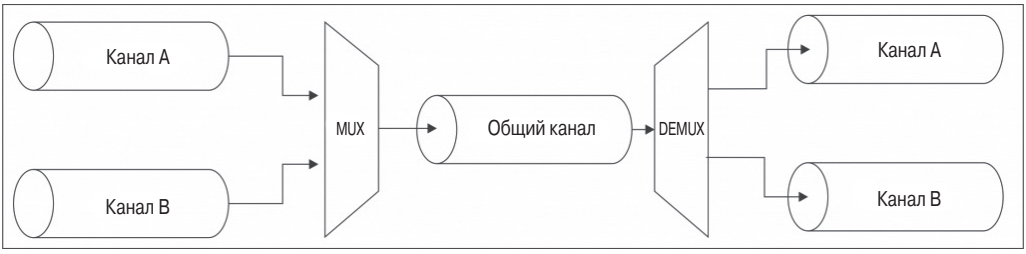

Рис. 5.9 ❖ Мультиплексирование и демультиплексирование

Операция объединения нескольких потоков (в данном случае их еще называют каналами), обеспечивающая возможность передачи через один поток, называется **мультиплексированием** (MUX, на рис. 5.9), а обратная операция – восстановление оригинальных потоков из общего потока – называется **демультиплексированием** (DEMUX, на рис. 5.9). *Устройства*, выполняющие эти операции, называются **мультиплексором** (или **mux**) и **демультиплексором** (или **demux**) соответственно. Эти операции широко применяются в области информатики и связи, поскольку являются одной из основ коммуникаций практически любого вида, таких как телефония, радио, телевидение и, конечно же, Интернет. Мы не будем подробно останавливаться на них в этой книге, поскольку они сами по себе являются обширной темой.

Вместо этого в данном разделе будет продемонстрировано использование общих потоков на платформе Node.js для передачи нескольких логически разных потоков с последующим их разделением на другом конце общего потока.

Инструмент удаленного журналирования

Рассмотрим следующий пример. Предположим, что требуется создать небольшую программу, запускающую дочерний процесс и перенаправляющую стандартный вывод и вывод сообщений об ошибках на удаленный сервер, который, в свою очередь, сохраняет эти два потока в два отдельных файла. В этом случае роль общей среды играет TCP-соединение, а двумя мультиплексируемыми каналами являются `stdout` и `stderr` дочернего процесса. Здесь будет применяться технология, называемая **коммутацией пакетов**, которую используют такие протоколы, как IP, TCP или UDP, и которая обеспечивает помещение данных в *пакеты*, позволяя при этом определять различные метаданные, необходимые для мультиплексирования, маршрутизации,

управления потоком, проверки данных на предмет их повреждения и т. д. Реализуемый для нужд этого примера протокол отличается минимализмом, поскольку данные помещаются в пакеты со структурой, показанной на рис. 5.10.

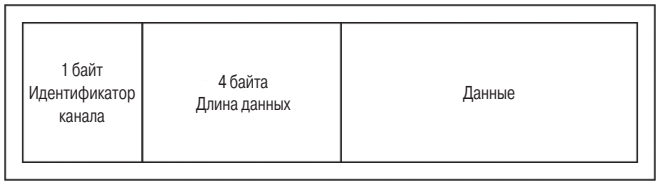

Рис. 5.10 ❖ Структура пакета

Как показано на рис. 5.10, пакет содержит не только сами данные, но и заголовок (*идентификатор канала* и *длина данных*), что позволяет различать данные из разных потоков и направлять пакеты в нужные каналы на выходе.

Мультиплексирование на стороне клиента Начнем разработку приложения со стороны клиента. Проявив недюжинную изобретательность, назовем соответствующий модуль `client.js`. Он будет той частью приложения, которая отвечает за запуск дочернего процесса и мультиплексирование его потоков.

Итак, начнем с определения модуля. Во-первых, нам потребуются некоторые зависимости:

```
const child_process = require('child_process');
const net = require('net');
```

Затем реализуем функцию, осуществляющую мультиплексирование списка источников:

```
function multiplexChannels(sources, destination) {
  let totalChannels = sources.length;
  for(let i = 0; i <sources.length; i++) {
    sources[i]
      .on('readable', function() {                          //[1]
        let chunk;
        while((chunk = this.read()) !== null) {
          const outBuff = new Buffer(1 + 4 + chunk.length);  //[2]
          outBuff.writeUInt8(i, 0);
          outBuff.writeUInt32BE(chunk.length, 1);
          chunk.copy(outBuff, 5);
          console.log('Sending packet to channel: ' + i);
          destination.write(outBuff);                       //[3]
        }
      }
      .on('end', () => {                                    //[4]
        if(--totalChannels === 0) {
          destination.end();
        }
      });
  }
}
```

Функция `multiplexChannels()` принимает потоки-источники для мультиплексирования и целевой канал, а затем выполняет следующие действия:

1) для каждого потока-источника регистрируется обработчик события `readable`, извлекающий данные из потока в дискретном режиме;
2) он заключает прочитанный фрагмент в пакет, который содержит 1 Б (UInt8) идентификатора канала, 4 Б (UInt32BE) размера пакета и фактические данные;
3) подготовленный пакет записывается в целевой поток;
4) и наконец, регистрируется обработчик события `end`, чтобы можно было завершить целевой поток после закрытия всех потоков-источников.

Данный протокол позволяет мультиплексировать до 256 различных потоков-источников, потому что для идентификации канала выделен только 1 байт.

А теперь заключительная часть кода клиента:

```
const socket = net.connect(3000, () => {                    //[1]
  const child = child_process.fork(                          //[2]
  process.argv[2],
  process.argv.slice(3),
    {silent: true}
  );
  multiplexChannels([child.stdout, child.stderr], socket); //[3]
});
```

В этом последнем фрагменте выполняются следующие операции:

1) создается новое TCP-подключение клиента к адресу `localhost:3000`;
2) выполняется запуск дочернего процесса, путь к которому указан в первом аргументе командной строки. Остальная часть массива `process.argv` передается дочернему процессу в виде аргументов. Параметр `{silent: true}` позволяет избежать наследования дочерним процессом `stdout` и `stderr` родительского процесса;
3) и наконец, `stdout` и `stderr` дочернего процесса мультиплексируются и передаются в сокет с помощью функции `mutiplexChannels()`.

Демультиплексирование на стороне сервера Теперь можно позаботиться о создании серверной части приложения (`server.js`), где выполняются демультиплексирование потоков из удаленного соединения и передача их по конвейеру в два разных файлах. Начнем с создания функции `demultiplexChannel()`:

```
const net = require('net');
const fs = require('fs');

function demultiplexChannel(source, destinations) {
  let currentChannel = null;
  let currentLength = null;

  source
    .on('readable', () => {                                  //[1]
    let chunk;
    if(currentChannel === null) {                            //[2]
      chunk = source.read(1);
      currentChannel = chunk && chunk.readUInt8(0);
    }
```

```
    if(currentLength === null) {                              //[3]
      chunk = source.read(4);
      currentLength = chunk && chunk.readUInt32BE(0);

      if(currentLength === null) {
        return;
      }
    }

    chunk = source.read(currentLength);                       //[4]
    if(chunk === null) {
      return;
    }
    console.log('Received packet from: ' + currentChannel);
    destinations[currentChannel].write(chunk);                //[5]
    currentChannel = null;
    currentLength = null;
  })
  .on('end', ()=> {                                           //[6]
    destinations.forEach(destination => destination.end());
    console.log('Source channel closed');
  });
}
```

Предыдущий код может показаться сложным, но на самом деле это не так. Благодаря особенностям потоков для чтения мы легко можем реализовать демультиплексирование нашего небольшого протокола, как описано ниже.

1. Выполняется чтение из потока с использованием дискретного режима.
2. Во-первых, если идентификатор канала еще не прочитан, читается и преобразуется в число первый байт из потока.
3. Затем извлекается размер данных. Для этого требуется прочитать 4 Б, поэтому возможно (хотя и маловероятно), что во внутреннем буфере окажется недостаточно данных, что заставит метод `this.read()` вернуть значение `null`. В таком случае парсинг прерывается, и попытка повторяется при появлении следующего события `readable`.
4. После чтения размера данных становится ясно, сколько данных нужно получить из внутреннего буфера, поэтому делается попытка прочитать их все.
5. После чтения всех данных их можно записать в нужный целевой канал, не забыв сбросить значения переменных `currentChannel` и `currentLength` (поскольку они будут использоваться при анализе следующего пакета).
6. И наконец, после завершения чтения из канала-источника необходимо закрыть все целевые каналы.

Теперь, имея код демультиплексирования потока-источника, определим новую функцию:

```
net.createServer(socket => {
  const stdoutStream = fs.createWriteStream('stdout.log');
  const stderrStream = fs.createWriteStream('stderr.log');
  demultiplexChannel(socket, [stdoutStream, stderrStream]);
}).listen(3000, () => console.log('Server started'));
```

Она сначала запускает TCP-сервер, прослушивающий порт `3000`, затем для каждого соединения создаются два потока для записи в разные файлы: один для стандартного вывода и другой для стандартной ошибки – это наши целевые каналы. И наконец, вызывается функция `demultiplexChannel()`, осуществляющая демультиплексирование потока `socket` в потоки `stdoutStream` и `stderrStream`.

Запуск приложения мультиплексирования/демультиплексирования Теперь можно проверить работу приложения мультиплексирования/демультиплексирования, но для этого нужно создать небольшую программу, генерирующую какой-либо вывод. Назовем ее `generateData.js`:

```
console.log("out1");
console.log("out2");
console.error("err1");
console.log("out3");
console.error("err2");
```

Отлично, теперь все готово для проверки приложения удаленного журналирования. Сначала запустим сервер:

```
node server
```

Затем клиента, передав ему путь к файлу программы, которая должна стать дочерним процессом:

```
node client generateData.js
```

Клиент должен запуститься практически сразу же, а в конце процесса стандартный вывод и вывод сообщений об ошибках приложения `generateData` будут переданы через общее TCP-соединение и демультиплексированы на сервере в два отдельных файла.

Имейте в виду: поскольку была использована функция `child_process.fork()` (http://nodejs.org/api/child_process.html#child_process_child_process_fork_modulepath_args_options), наш клиент сможет запускать только другие модули Node.js.

Мультиплексирование и демультиплексирование объектных потоков

Приведенный пример показал, как мультиплексировать и демультиплексировать двоичные или текстовые потоки, но этот подход также применим к объектным потокам. Самое большое отличие лишь в том, что у нас уже есть способ передачи данных в виде атомарных сообщений (объектов), что упрощает мультиплексирование и демультиплексирование за счет придания свойства `channelID` каждому из объектов.

Другой шаблон, охватывающий только демультиплексирование, заключается в маршрутизации поступающих данных на основании заданного условия. С помощью этого шаблона можно реализовать сложное разделение потоков, как показано на рис. 5.11.

Демультиплексор на рис. 5.11 принимает поток описывающих объектов, представляющих разных *животных*, и распределяет их по соответствующим целевым потокам на основе класса животного: *рептилии*, *амфибии* и *млекопитающие*.

На основе этого принципа можно также реализовать оператор `if...else` для потоков. Пример такой реализации вы найдете в пакете `ternary-stream` (https://npmjs.org/package/ternary-stream), который позволяет осуществить именно это.

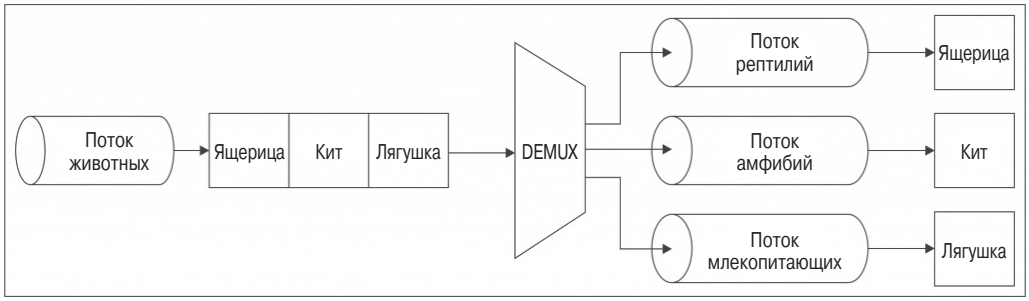

Рис. 5.11 ❖ Разделение потоков на основании заданного условия

Итоги

В этой главе мы пролили свет на потоки данных в Node.js и варианты их использования и одновременно приоткрыли дверь к неограниченным возможностям этой парадигмы программирования. Мы узнали причины популярности потоков в сообществе Node.js и овладели основными функциональными возможностями, что позволит вам лучше ориентироваться в этом новом мире. Были проанализированы некоторые современные шаблоны и порядок соединения потоков в различных конфигурациях, подчеркнута важность их совместимости, которая делает потоки настолько универсальными и мощными.

Если что-то нельзя осуществить с помощью одного потока, это наверняка можно сделать путем соединения его с другими потоками, и это соответствует философии *единственной ответственности*. Очевидно, что потоки не просто являются популярной особенностью платформы Node.js, – они также являются неотъемлемой ее частью, важным шаблоном обработки двоичных данных, строк и объектов. Не случайно им посвящена целая глава.

В следующей главе будут рассмотрены традиционные объектно-ориентированные шаблоны проектирования. Но не будем обманываться: несмотря на то что язык JavaScript является в некоторой степени объектно-ориентированным, в самой платформе Node.js используются два основных подхода: функциональный и смешанный. Перед тем как прочесть следующую главу, желательно избавиться от всех предрассудков.

Глава 6

Шаблоны проектирования

Шаблон проектирования – это типовое решение группы подобных задач. У этого термина весьма широкое толкование, и используется он в разных областях. Но, как правило, его связывают с широко известным набором объектно-ориентированных шаблонов, ставших популярными в 90-х благодаря книге «*Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education*»[1] практически легендарной **банды четырех** (Gang of Four, GoF), в которую входили Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides). Часто этот конкретный набор шаблонов называют *традиционными* шаблонами, или шаблонами проектирования GoF.

Применение этого набора объектно-ориентированных шаблонов проектирования в JavaScript не является таким же прямолинейным и формальным, как в классическом объектно-ориентированном языке. Как уже упоминалось, двойственность языка JavaScript, поддерживающего объектно-ориентированное программирование, но основанное на прототипах и динамической типизации, позволяет работать с функциями, как с обычными сущностями, и использовать функциональный стиль программирования. Эти особенности делают JavaScript очень универсальным языком с огромными возможностями, но в то же время вызывают разобщенность стилей программирования, соглашений, технологий и в конечном итоге шаблонов экосистемы. В JavaScript существует так много способов достижения одного и того же результата, что каждый разработчик имеет собственное мнение о лучшем подходе к решению проблемы. Ярким свидетельством явления является обилие фреймворков и библиотек в экосистеме JavaScript. Вероятно, их никогда не было так много ни в одном другом языке, особенно сейчас, когда платформа Node.js привнесла новые удивительные возможности в язык JavaScript и обеспечила создание множества новых сценариев.

В этом контексте применение традиционных шаблонов в JavaScript определяется его особенностями. Существует так много способов их реализации, что их традиционная строгая объектно-ориентированная реализация исключает их применение в качестве шаблонов. В некоторых случаях их реализация просто невозможна, поскольку язык JavaScript не поддерживает настоящих классов или абстрактных интерфейсов. Что остается неизменным, так это оригинальные идеи, лежащие в основе каждого из шаблонов, решаемая ими проблема и идея такого решения.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2013. ISBN: 978-5-496-00389-6. – *Прим. ред.*

В этой главе мы рассмотрим применение некоторых из наиболее важных шаблонов проектирования GoF на платформе Node.js с учетом ее философии, что позволит заново осмыслить их значение, взглянув на них с другой точки зрения. Вместе с традиционными будут представлены некоторые «менее традиционные» шаблоны проектирования, созданные внутри экосистемы JavaScript.

Ниже приводится перечень шаблонов, рассматриваемых в этой главе:

- Фабрика (Factory);
- Открытый конструктор (Revealing constructor);
- Прокси (Proxy);
- Декоратор (Decorator);
- Адаптер (Adapter);
- Стратегия (Strategy);
- Состояние (State);
- Макет (Template);
- Промежуточное программное обеспечение (Middleware);
- Команда (Command).

В этой главе предполагается наличие у читателя определенных знаний о механизме наследования в языке JavaScript. Обратите также внимание, что для описания шаблонов здесь часто будут использоваться наглядные универсальные схемы вместо стандартного UML, поскольку многие шаблоны реализуются на основе не только классов, но и объектов и даже функций.

Фабрика

Начнем с самого простого и широко распространенного в Node.js шаблона проектирования: **Фабрика (Factory)**.

Универсальный интерфейс для создания объектов

Мы уже подчеркивали, что в языке JavaScript часто отдается предпочтение функциональной парадигме, а не чисто объектно-ориентированному проектированию, из-за ее простоты, удобства использования и *малой общедоступной области*. Это особенно актуально при создании новых экземпляров объектов. Фактически обращение к фабрике часто оказывается гибче и удобнее, чем непосредственное создание нового объекта из прототипа с помощью оператора `new` или `Object.create()`.

Во-первых, фабрика позволяет отделить создание объекта от его реализации. Фактически обертывание фабрикой создания нового экземпляра дает большую гибкость и управляемость этому действию. С помощью фабрики можно создать новый экземпляр, используя замыкания, прототип и оператор `new`, метод `Object.create()`, или даже вернуть экземпляр с определенным состоянием. Код, использующий фабрику, может ничего не знать о порядке создания экземпляра. Суть в том, что использование оператора `new` привязывает код к одному конкретному способу создания объектов, в то время как язык JavaScript практически без затрат предоставляет гораздо большую гибкость. В качестве поясняющего примера рассмотрим простую фабрику объектов `Image`:

```
function createImage(name) {
  return new Image(name);
}
const image = createImage('photo.jpeg');
```

На первый взгляд, фабрика `CreateImage()` кажется совершенно ненужной. Почему бы просто не создать экземпляр класса `Image` с помощью оператора `new`? Для этого потребуется написать, например, следующую строку кода:

```
const image = new Image(name);
```

Как упоминалось чуть выше, использование оператора `new` вызывает привязку кода к конкретному типу объектов, в данном случае к типу `Image`. Фабрика обеспечивает большую гибкость. Предположим, позднее потребуется выполнить рефакторинг класса `Image` и разделить его на несколько более специализированных классов, по одному для каждого из поддерживаемых форматов изображений. Если все новые изображения создаются только с помощью фабрики, ее код можно изменить, как показано ниже, и это никак не повлияет на существующий код:

```
function createImage(name) {
  if(name.match(/\.jpeg$/)) {
    return new JpegImage(name);
  } else if(name.match(/\.gif$/)) {
    return new GifImage(name);
  } else if(name.match(/\.png$/)) {
    return new PngImage(name);
  } else {
    throw new Exception('Unsupported format');
  }
}
```

Кроме того, применение фабрик позволяет не экспортировать конструкторы создаваемых объектов, что исключает возможность их расширения и изменения (помните принцип малой общедоступной области?). В Node.js это можно осуществить путем экспорта только фабрики, оставив конструкторы закрытыми.

Механизм принудительной инкапсуляции

Благодаря замыканиям фабрика может использоваться как механизм инкапсуляции.

Инкапсуляция – это способ управления доступом к деталям реализации объекта путем предотвращения непосредственного внесения изменений в объект извне. Взаимодействие с объектом осуществляется только через общий интерфейс, отделяющий внутреннюю реализацию объекта от внешнего кода. Такая технология называется также **сокрытием информации**. Кроме того, инкапсуляция является одним из основополагающих принципов объектно-ориентированного программирования, основой наследования, полиморфизма и абстракции.

В языке JavaScript отсутствуют модификаторы доступа (например, нет возможности объявить закрытую переменную), поэтому единственными средствами инкапсуляции остаются области видимости функций и замыкания. Применение фабрики упрощает поддержку закрытых переменных. В качестве примера рассмотрим следующий код:

```
function createPerson(name) {
  const privateProperties = {};

  const person = {
    setName: name => {
      if(!name) throw new Error('A person must have a name');
```

```
      privateProperties.name = name;
    },
    getName: () => {
      return privateProperties.name;
    }
  };

  person.setName(name);
  return person;
}
```

В этом примере с помощью замыканий создаются два объекта: объект `person` – открытый интерфейс, возвращаемый фабрикой, и группа свойств `privateProperties`, доступных извне только через интерфейс объекта `person`. Например, предыдущий код гарантирует, что свойство `name` для любого объекта `person` никогда не будет пустым, что невозможно обеспечить, если бы `name` было просто свойством объекта `person`.

Фабрики – лишь один из способов создания закрытых свойств. Возможны и другие подходы:

- определение закрытых переменных в конструкторе (как рекомендует Дуглас Крокфорд (Douglas Crockford): http://javascript.crockford.com/private.html);
- использование соглашений, например добавление префикса с символом подчеркивания «_» или доллара «$» к имени свойства (хотя технически это не исключает доступа извне);
- использование коллекций `WeakMap`, появившихся в стандарте ES2015: (http://fitzgeraldnick.com/weblog/53/).

Более полную информацию по этому вопросу можно найти в статье, опубликованной на сайте Mozilla: https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Contributor_s_Guide/Private_Properties.

Создание простого профилировщика кода

А теперь рассмотрим полноценный пример использования фабрик. Создадим простой *профилировщик кода* как объект со следующими свойствами:

- метод `start()`, запускающий сеанс профилирования;
- метод `end()`, завершающий сеанс профилирования и выводящий время выполнения в консоль.

Начнем с создания файла `profiler.js` со следующим содержимым:

```
class Profiler {
  constructor(label) {
    this.label = label;
    this.lastTime = null;
  }

  start() {
    this.lastTime = process.hrtime();
  }

  end() {
    const diff = process.hrtime(this.lastTime);
    console.log(
      `Timer "${this.label}" took ${diff[0]} seconds and ${diff[1]} nanoseconds.`
    );
  }
}
```

В этом классе нет ничего особенного. Он использует высокоточный таймер по умолчанию для получения текущего времени в вызове метода `start()`, а в последующем вызове метода `end()` вычисляет прошедшее время и выводит результат в консоль.

Если использовать этот профилировщик в реальном приложении для получения времени выполнения различных процедур, он выведет в стандартный вывод огромный объем информации. Возможно, было бы предпочтительнее перенаправить данные профилирования в другое место, например в базу данных, или просто отключить профилировщик в рабочем режиме. Очевидно, что при непосредственном создании экземпляра профилировщика с помощью оператора `new` потребуется определенная дополнительная логика в коде клиента или в самом объекте профилировщика для переключения между различными режимами. Применение фабрики позволит абстрагировать создание объекта профилировщика в зависимости от режима работы приложения: во время разработки можно возвращать полноценный объект профилировщика, а в режиме эксплуатации – макет объекта с тем же интерфейсом, но с пустыми методами. Реализуем это в модуле `profiler.js`, заменив экспорт конструктора профилировщика на экспорт фабричной функции:

```
module.exports = function(label) {
  if(process.env.NODE_ENV === 'development') {
    return new Profiler(label);                          //[1]
  } else if(process.env.NODE_ENV === 'production') {
    return {                                             //[2]
      start: function() {},
      end: function() {}
    }
  } else {
    throw new Error('Must set NODE_ENV');
  }
};
```

Вновь созданная фабрика абстрагирует создание объекта профилировщика от его реализации:

- в режиме разработки возвращается новый, полноценный объект профилировщика;
- в режиме эксплуатации возвращается фиктивный объект с пустыми методами `start()` и `stop()`.

Здесь следует подчеркнуть, что благодаря динамической типизации JavaScript в одних случаях можно возвращать объекты, созданные с помощью оператора `new`, а в других – литералы объектов (это также называют **утиной типизацией**, https://ru.wikipedia.org/wiki/Утиная_типизация). Фабрика отлично справляется со своей работой, внутри фабричной функции мы можем создавать объекты как угодно, выполнять дополнительные действия или возвращать объекты другого типа, в зависимости от определенных условий, изолируя потребителя от всех этих подробностей. Потенциал этого простого шаблона не вызывает сомнений.

Теперь можно поэкспериментировать с профилировщиком. Вот один из примеров использования созданной фабрики:

```
const profiler = require('./profiler');

function getRandomArray(len) {
  const p = profiler('Generating a ' + len + ' items long array');
```

```
  p.start();
  const arr = [];
  for(let i = 0; i < len; i++) {
    arr.push(Math.random());
  }
  p.end();
}

getRandomArray(1e6);
console.log('Done');
```

Переменная `p` содержит экземпляр объекта `Profiler`, но при этом неизвестно, как он был создан и какова его реализация на момент использования в предыдущем коде.

Если поместить предыдущий код в файл `profilerTest.js`, можно проверить эти допущения. Для тестирования программы с включенным профилированием выполните следующую команду:

```
export NODE_ENV=development; node profilerTest
```

Эта команда обеспечит создание настоящего профилировщика и вывод полученных им сведений в консоль. Для проверки макета профилировщика можно выполнить следующую команду:

```
export NODE_ENV=production; node profilerTest
```

Представленный здесь пример является лишь элементарным применением шаблона фабрики, но он наглядно демонстрирует преимущества отделения создания объекта от его реализации.

Составные фабричные функции

Теперь, получив представление об особенностях реализации фабричных функций в Node.js, рассмотрим дополнительный шаблон, который лишь недавно привлек внимание сообщества JavaScript. Здесь речь пойдет о **составных фабричных функциях**, которые можно «объединять» для получения новых, расширенных фабричных функций. Их полезность определяется возможностью создавать объекты, способные «наследовать» поведение и свойства из различных источников, без создания сложных иерархий классов.

Поясним эту идею на простом наглядном примере. Предположим, что требуется разработать видеоигру, персонажи которой характеризуются разными моделями поведения: они способны *перемещаться* по экрану, *рубить* и *стрелять*. И, конечно же, персонажи должны обладать некоторыми базовыми свойствами, такими как количество жизней, координаты на экране и имя.

Требуется определить несколько видов персонажей, каждый из которых поддерживает одну из моделей поведения:

- **персонаж**: базовый персонаж; обладает свойствами для хранения количества жизней, координат и имени;
- **ходок**: персонаж, способный перемещаться;
- **рубака**: персонаж, умеющий рубить;
- **стрелок**: персонаж, способный стрелять (при наличии патронов!).

В идеале желательно иметь возможность определять новые виды персонажей, комбинируя модели поведения существующих персонажей. Причем требуется абсолют-

ная свобода выбора, например можно было бы определить следующие виды персонажей на основе существующих:

- **бегун**: персонаж, способный перемещаться быстрее ходока;
- **самурай**: персонаж, способный перемещаться и рубить;
- **снайпер**: персонаж, способный стрелять (но не двигаться);
- **вооруженный бегун**: бегун, способный стрелять;
- **самурай из вестерна**: персонаж, способный перемещаться, рубить и стрелять.

Как видите, желательна полная свобода при комбинировании функциональных возможностей базовых видов. Теперь должно быть очевидно, что такое моделирование с помощью классов и наследования будет непростой задачей.

Поэтому для решения поставленной задачи воспользуемся составными фабричными функциями и, в частности, модулем `stampit`: (https://www.npmjs.com/package/stampit).

Этот модуль предлагает интуитивно понятный интерфейс для определения фабричных функций, поддерживающих возможность комбинирования. По существу, он позволяет определять фабричные функции, генерирующие объекты с определенным набором свойств и методов, с помощью удобного гибкого интерфейса для их описания.

Посмотрим, насколько легко определить базовые типы для нашей игры. Начнем с базовых персонажей:

```
const stampit = require('stampit');

const character = stampit().
  props({
    name: 'anonymous',
    lifePoints: 100,
    x: 0,
    y: 0
  });
```

Здесь определена фабричная функция `character` для создания новых экземпляров базовых персонажей. Каждый персонаж имеет свойства: `name`, `lifePoints`, `x` и `y` со значениями по умолчанию `anonymous`, `100`, `0` и `0` соответственно. Метод `props` из модуля `stampit` позволяет задать эти свойства. Для использования данной фабричной функции нужно написать примерно такой код:

```
const c = character();
c.name = 'John';
c.lifePoints = 10;
console.log(c); // { name: 'John', lifePoints: 10, x:0, y:0 }
```

Теперь определим фабричную функцию `mover` для создания персонажа-ходока:

```
const mover = stampit()
  .methods({
    move(xIncr, yIncr) {
      this.x += xIncr;
      this.y += yIncr;
      console.log(`${this.name} moved to [${this.x}, ${this.y}]`);
    }
  });
```

Здесь для объявления всех доступных методов объекта, создаваемого данной фабричной функцией, был использован метод `methods` из модуля `stampit`. В этом определении `Mover` присутствует функция `move`, которая может увеличивать свойства `x` и `y`, определяющие позицию экземпляра. Обратите внимание, что доступ к свойствам экземпляра внутри метода можно получить с помощью ключевого слова `this`.

Теперь, основываясь на этих базовых идеях, можно легко добавить определение фабричных функций для типов `slasher` (рубака) и `shooter` (стрелок):

```
const slasher = stampit()
  .methods({
    slash(direction) {
      console.log(`${this.name} slashed to the ${direction}`);
    }
  });

  const shooter = stampit()
  .props({
    bullets: 6
  })
  .methods({
    shoot(direction) {
      if (this.bullets > 0) {
        --this.bullets;
        console.log(`${this.name} shoot to the ${direction}`);
      }
    }
  });
```

Обратите внимание на порядок использования методов `props` и `methods` при определении фабричной функции `shooter`.

Теперь, когда имеются определения всех базовых типов, можно переходить к созданию новых, более мощных фабричных функций:

```
const runner = stampit.compose(character, mover);
const samurai = stampit.compose(character, mover, slasher);
const sniper = stampit.compose(character, shooter);
const gunslinger = stampit.compose(character, mover, shooter);
const westernSamurai = stampit.compose(gunslinger, samurai);
```

Метод `stampit.compose` определяет новую составную фабричную функцию для создания объекта, обладающего методами и свойствами, которые определяются используемыми фабричными функциями. Как видите, этот мощный механизм обеспечивает полную свободу действий и позволяет оперировать моделями поведения, а не классами.

Для придания примеру завершенности создадим экземпляр объекта `westernSamurai` и проверим его в действии.

```
const gojiro = westernSamurai();
gojiro.name = 'Gojiro Kiryu';
gojiro.move(1,0);
gojiro.slash('left');
gojiro.shoot('right');
```

Этот код выведет следующее:

```
Yojimbo moved to [1, 0]
Yojimbo slashed to the left
Yojimbo shoot to the right
```

Более подробные сведения об объединении фабричных функций можно найти в статье Эрика Эллиота (Eric Elliot), автора этой идеи: https://medium.com/javascript-scene/introducing-the-stamp-specification-77f8911c2fee.

Реальное применение

Как уже упоминалось ранее, фабрики очень популярны в мире Node.js. Многие пакеты позволяют создавать новые экземпляры только с помощью фабрик, например:

- `Dnode` (https://npmjs.org/package/dnode): система **вызова удаленных процедур** (Remote Procedure Call, RPC) для Node.js. Заглянув в исходный код этого пакета, можно убедиться, что вся логика реализована в классе с именем `D`, но он не экспортируется вовне, единственный общедоступный интерфейс – фабричная функция, которая создает новые экземпляры этого класса. Исходный код пакета можно найти на странице https://github.com/substack/dnode/blob/34d1c9aa9696f13bdf8fb99d 9d039367ad873f90/index.js#L7-9;
- `Restify` (https://npmjs.org/package/restify): фреймворк для реализации REST API, позволяющий создавать новые экземпляры сервера с помощью фабрики `restify.createServer()`, которая внутренне создает новый экземпляр класса `Server` (который не экспортируется). Исходный код пакета можно найти на странице https://github.com/mcavage/node-restify/blob/5f31e2334b38361ac7ac1a5e5d852b7206ef7d94/lib/index.js#L91-116.

Другие модули экспортируют и классы, и фабрики, но указывают, что фабрики являются основным средством создания новых экземпляров. Вот примеры таких модулей:

- `http-proxy` (https://npmjs.org/package/http-proxy): библиотека для создания программируемых прокси-серверов, позволяющая создавать новые экземпляры с помощью функции `httpProxy.createProxyServer(options)`;
- простейший HTTP-сервер для платформы Node.js: новые экземпляры главным образом создаются с использованием функции `http.createServer()`, несмотря на то что это, по сути, более краткая форма записи `new http.Server()`;
- `bunyan` (https://npmjs.org/package/bunyan): популярная библиотека для журналирования. В описании авторы предлагают использовать фабрику `bunyan.createLogger()` как основной метод создания новых экземпляров, несмотря на то что она эквивалентна выполнению оператора `new bunyan()`.

Некоторые модули предоставляют фабрики для создания дополнительных компонентов. Популярными примерами являются пакеты `through2` и `from2` (рассматривались в *главе 5* «*Программирование с применением потоков данных*»), позволяющие упростить создание новых потоков данных с помощью фабрик, освобождая разработчиков от явного использования наследования и оператора `new`.

И наконец, в качестве примера пакетов, использующих составные фабрики, можно привести `react-stampit` (https://www.npmjs.com/package/react-stampit), использующий составные фабричные функции для облегчения создания виджетов, и `remitter` (https://www.npmjs.com/package/remitter), модуль обмена сообщениями на основе Redis.

Открытый конструктор

Шаблон открытого конструктора является относительно новым, рост популярности которого в сообществах Node.js и JavaScript вызван, в частности, его применением в нескольких библиотеках ядра, таких как `Promise`.

Этот шаблон косвенно рассматривался в *главе 4* «Шаблоны асинхронного выполнения с использованием спецификации ES2015, и не только», где речь шла об объектах `Promise`. Но давайте вернемся немного назад и исследуем конструктор `Promise` более подробно:

```
const promise = new Promise(function (resolve, reject) {
  // ...
});
```

Как видите, конструктор `Promise` принимает функцию, которая называется **исполняющей функцией**. Эта функция вызывается внутри конструктора `Promise` и используется для работы с ограниченной частью внутреннего состояния объекта `Promise`. Другими словами, она служит для экспортирования функций `resolve` и `reject`, чтобы их можно было вызвать при изменении внутреннего состояния объекта.

Преимущество этого подхода – в том, что только конструктор получает доступ к функциям `resolve` и `reject`, и после создания объекта `Promise` его можно безопасно передавать из одной части программы в другую, поскольку никакой другой код не сможет вызвать функции `resolve` и `reject` и изменить внутреннее состояние объекта `Promise`.

Именно поэтому Доменик Деникола (Domenic Denicola) назвал данный шаблон «открытым конструктором» в одной из статей в своем блоге.

Статья Доменика чрезвычайно интересная, она содержит исторический анализ происхождения шаблона и сравнивает некоторые аспекты этого шаблона с шаблоном, который используется потоками данных, и с другими шаблонами конструирования, применявшимися в ранних реализациях библиотек `Promise`. Полный текст статьи можно найти на странице https://blog.domenic.me/the-revealing-constructor-pattern/.

Генератор событий, доступный только для чтения

В этом разделе мы используем шаблон открытого конструктора для создания генератора событий, доступного только для чтения, – особого вида генераторов событий, не позволяющего вызывать свой метод `emit` (кроме как внутри функции, передаваемой конструктору в качестве аргумента).

Создадим файл `roee.js` и поместим в него код класса `Roee` (read-only event emitter):

```
const EventEmitter = require('events');

module.exports = class Roee extends EventEmitter {
  constructor (executor) {
    super();
    const emit = this.emit.bind(this);
    this.emit = undefined;
    executor(emit);
  }
};
```

Этот простой класс наследует класс `EventEmitter`, и его конструктор принимает единственный аргумент `executor` с функцией.

Конструктор вызывает функцию `super`, чтобы правильно инициализировать генератор событий вызовом родительского конструктора, затем создается резервная копия функции `emit` с последующим ее удалением, путем присваивания ей значения `undefined`.

В заключение вызывается функция `executor`, которой передается аргумент с резервной копией метода `emit`.

Здесь важно понимать то, что после присваивания методу `emit` значения `undefined` его не получится вызвать из другого места в коде. Резервная копия метода `emit` определена как локальная переменная, которая доступна только в функции `executor`. Этот механизм позволяет использовать метод `emit` только внутри исполняющей функции.

А теперь используем этот новый класс для создания простого класса, который генерирует событие *tick* каждую секунду и хранит количество всех сгенерированных событий. Поместим этот класс в новый модуль `ticker.js`:

```
const Roee = require('./roee');

const ticker = new Roee((emit) => {
  let tickCount = 0;
  setInterval(() => emit('tick', tickCount++), 1000);
});

module.exports = ticker;
```

Как видите, код получился довольно простым. Он создает новый экземпляр `Roee` и передает ему логику создания событий, завернутую в исполняющую функцию. Исполняющая функция получает аргумент `emit` и использует его для создания нового события `tick` раз в секунду.

А теперь рассмотрим небольшой пример использования этого модуля:

```
const ticker = require('./ticker');

ticker.on('tick', (tickCount) => console.log(tickCount, 'TICK'));
// ticker.emit('something', {}); <-- Это вызовет ошибку
```

Объект `ticker` можно использовать как любой другой объект, основанный на генераторе событий, это позволяет подключить любое количество обработчиков вызовом его метода `on`. Но при попытке вызвать метод `emit` наш код завершится с сообщением об ошибке `TypeError: ticker.emit is not a function`.

Этот пример неплохо демонстрирует использование шаблона открытого конструктора, тем не менее стоит отметить, что его защита генератора событий не вполне надежна, и ее можно обойти несколькими способами. Например, событие `ticker` можно генерировать с помощью метода `emit` прототипа, как показано ниже:

```
require('events').prototype.emit.call(ticker, 'someEvent', {});
```

Реальное применение

Несмотря на интересное решение, довольно трудно найти универсальный случай использования этого шаблона, кроме как в конструкторе `Promise`.

Стоит отметить, что в спецификации потоков данных, находящейся в стадии разработки, была сделана попытка применить этот шаблон как лучшую альтернативу ис-

пользуемому в настоящее время шаблону «Макет», позволяющую описывать модели поведения различных объектов потоков данных: https://streams.spec.whatwg.org.

Следует также отметить, что этот шаблон уже использовался в *главе 5 «Программирование с применением потоков данных»*, при реализации класса `ParallelStream`. Конструктор этого класса принимает функцию `userTransform` (исполняющую функцию).

Даже притом, что исполняющая функция вызывается не в конструкторе, а во внутреннем методе `_transform` потока, общая идея шаблона все равно остается в силе. Фактически такой подход позволяет передавать внутренние компоненты потока (например, функцию `push`) только конкретной логике преобразований, которая определяется на этапе конструирования при создании нового экземпляра `ParallelStream`.

Прокси

Прокси – это объект, управляющий доступом к другому объекту, который называется **субъектом**. Прокси и субъект имеют идентичные интерфейсы, и это позволяет прозрачно подменять одного другим: фактически этот шаблон имеет другое название: «**суррогат**». Прокси перехватывает все или часть операций, которые должны выполняться субъектом, расширяя или дополняя его модель поведения. На рис. 6.1 изображена схема работы этого шаблона:

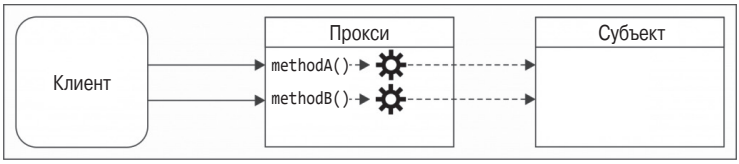

Рис. 6.1 ❖ Схема работы шаблона «Прокси»

Как показано на этой схеме, **прокси** и **субъект** имеют одинаковые интерфейсы, что обеспечивает их полную взаимозаменяемость с точки зрения клиента. **Прокси** направляет все операции субъекту, расширяя его модель поведения, выполняя подготовительные или заключительные операции.

Важно отметить, что здесь не идет речи о делегировании операций между классами; суть шаблона «Прокси» заключается в обертывании экземпляров субъекта, благодаря чему гарантируется сохранность их состояния.

Прокси могут пригодиться в нескольких ситуациях, например:

- **проверка данных**: прокси проверяет допустимость входных данных перед отправкой субъекту;
- **безопасность**: прокси проверяет наличие у клиента достаточных прав для выполнения операции и передает запрос субъекту только при положительном результате такой проверки;
- **кэширование**: прокси хранит внутренний кэш, вызывая субъекта, только если затребованные данные еще отсутствуют в кэше;
- **отложенная инициализация**: если создание субъекта сопряжено с большими накладными расходами, прокси может отложить эту операцию до момента, когда это станет действительно необходимо;

- **журналирование**: прокси перехватывает вызовы методов и их параметры, регистрируя их в журнале;
- **обращение к удаленным объектам**: прокси может обеспечить работу с удаленным объектом как с локальным.

Конечно, область применения шаблона «Прокси» намного шире, тем не менее примеры, перечисленные выше, дают достаточное представление о его возможностях.

Приемы реализации прокси

При проксировании объектов можно перехватывать вызовы всех методов субъекта или только часть из них, передавая остальные непосредственно субъекту. Существует несколько способов реализации такого поведения. Рассмотрим некоторые из них.

Композиция объектов

Прием композиции позволяет объединить один объект с другим с целью расширения или использования его функциональных возможностей. При использовании шаблона «Прокси» создается новый объект с тем же интерфейсом, как у субъекта, и ссылка на субъект сохраняется в прокси в виде переменной экземпляра или замыкания. Субъект может быть внедрен в момент создания прокси или создан самим прокси.

Вот один из примеров применения этого способа с помощью псевдокласса и фабрики:

```
function createProxy(subject) {
  const proto = Object.getPrototypeOf(subject);

  function Proxy(subject) {
    this.subject = subject;
  }

  Proxy.prototype = Object.create(proto);

  //подменяемый метод
  Proxy.prototype.hello = function(){
    return this.subject.hello() + ' world!';
  };

  //делегируемый метод
  Proxy.prototype.goodbye = function(){
    return this.subject.goodbye
        .apply(this.subject, arguments);
  };

  return new Proxy(subject);
}
module.exports = createProxy;
```

Для реализации прокси методом композиции необходимо перехватить вызовы методов, которые планируется обрабатывать (например, `hello()`), и просто делегировать остальные вызовы субъекту (как это сделано в случае с методом `goodbye()`).

Пример выше демонстрирует особый случай, когда субъект имеет прототип и требуется сохранить цепочку прототипов, чтобы инструкция `proxy instanceof Subject` возвращала `true`. Для этого было использовано **псевдоклассическое наследование**.

Этот дополнительный шаг нужен только для сохранения цепочки прототипов, что может пригодиться для совместимости прокси с кодом, изначально предназначенным для работы с субъектом.

Однако благодаря динамической типизации в языке JavaScript обычно удается избежать необходимости наследования и воспользоваться более непосредственными способами. Например, альтернативой предыдущей реализации прокси могут стать простой литерал объекта и фабрика:

```
function createProxy(subject) {
  return {
    //подменяемый метод
    hello: () => (subject.hello() + ' world!'),

    //делегируемый метод
    goodbye: () => (subject.goodbye.apply(subject, arguments))
  };
}
```

Чтобы создать прокси, который делегирует большую часть методов, можно использовать такую библиотеку, как `delegates` (https://npmjs.org/package/delegates).

Расширение объекта

Расширение объекта – наиболее прагматичный способ проксирования отдельных методов объекта, он заключается в изменении субъекта непосредственно, путем подмены реализации метода. Рассмотрим следующий пример:

```
function createProxy(subject) {
  const helloOrig = subject.hello;
  subject.hello = () => (helloOrig.call(this) + ' world!');

  return subject;
}
```

Этот метод определенно удобнее, когда требуется подменить только один или несколько методов, но его недостаток в том, что изменяется непосредственно объект `subject`.

Сравнение различных методов

Композицию можно считать *самым безопасным* способом проксирования, поскольку субъект в этом случае остается нетронутым – его исходная модель поведения не меняется. Единственный недостаток этого способа – в необходимости вручную делегировать все методы, даже если нужно подменить только один из них. При необходимости это может также потребовать делегировать доступ к свойствам субъекта.

Свойства объекта можно делегировать с помощью метода `Object.defineProperty()`. Более подробную информацию можно найти на странице https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty.

Расширение объекта, напротив, изменяет субъект, что не всегда допустимо, но избавляет от неудобств, связанных с делегированием. Поэтому прием расширения объекта считается наиболее прагматичным способом реализации прокси в JavaScript, и этому способу следует отдавать предпочтение, когда изменение субъекта не станет проблемой.

Однако существует одна ситуация, когда прием композиции становится практически обязательным. Это тот случай, когда требуется управлять инициализацией субъекта, чтобы, например, создавать его только при необходимости (*отложенная инициализация*).

Стоит отметить, что с помощью фабричной функции (в примерах выше это функция `createProxy()`) можно защитить свой код от применения приемов создания прокси.

Журналирование обращений к потоку для записи

Рассмотрим практический пример реализации шаблона «Прокси» – создадим объект, перехватывающий и журналирующий все обращения к методу `write()` потока для записи. Используем для этого прием композиции. Ниже приводится код из файла `loggingWritable.js`:

```
function createLoggingWritable(writableOrig) {
  const proto = Object.getPrototypeOf(writableOrig);
  function LoggingWritable(writableOrig) {
    this.writableOrig = writableOrig;
  }

  LoggingWritable.prototype = Object.create(proto);

  LoggingWritable.prototype.write = function(chunk, encoding, callback) {
    if(!callback && typeof encoding === 'function') {
      callback = encoding;
      encoding = undefined;
    }
    console.log('Writing ', chunk);
    return this.writableOrig.write(chunk, encoding, function() {
      console.log('Finished writing ', chunk);
      callback && callback();
    });
  };

  LoggingWritable.prototype.on = function() {
    return this.writableOrig.on
      .apply(this.writableOrig, arguments);
  };

  LoggingWritable.prototype.end = function() {
    return this.writableOrig.end
      .apply(this.writableOrig, arguments);
  };

  return new LoggingWritable(writableOrig);
}
```

Здесь создается фабрика, возвращающая проксированную версию переданного в аргументе объекта потока для записи. Мы переопределили метод `write()`, добавив вывод сообщения в консоль при каждом вызове этого метода и при завершении асинхронной операции. Кроме того, это хороший пример создания прокси для асинхронных функций, когда требуется подменять функции обратного вызова, что важно для таких платформ, как Node.js. Прочие методы, `on()` и `end()`, просто делегируются исходному потоку (другие методы потока опущены для краткости).

Теперь добавим несколько строк в модуль `loggingWritable.js`, чтобы проверить работу только что созданного прокси:

```
const fs = require('fs');

const writable = fs.createWriteStream('test.txt');
const writableProxy = createLoggingWritable(writable);
writableProxy.write('First chunk');
writableProxy.write('Second chunk');
writable.write('This is not logged');
writableProxy.end();
```

Прокси не изменяет оригинального интерфейса потока и модель его поведения, но, запустив предыдущий код, вы увидите, что все фрагменты, записываемые в поток, одновременно выводятся в консоль.

Место прокси в экосистеме – ловушки для функций и АОП

Во всех своих многочисленных формах шаблон «Прокси» является довольно популярным в Node.js и экосистеме. В самом деле, существует несколько библиотек, упрощающих создание прокси, причем большинство из них использует прием расширения объектов. В сообществе этот шаблон также называют **ловушками для функций**, или, реже, **аспектно-ориентированным программированием (АОП)**, что в действительности является общепринятой областью применения прокси. В АОП эти библиотеки обычно позволяют разработчику установить ловушки для перехвата управления *до* или *после* вызова конкретного метода (или набора методов) и выполнить свой код до и после выполнения метода соответственно.

Иногда прокси называют также **промежуточным программным обеспечением**, поскольку, подобно шаблону промежуточного программного обеспечения (который рассматривается далее в этой главе), они позволяют выполнять предварительную и заключительную обработку ввода/вывода функции. Иногда они дают возможность зарегистрировать несколько ловушек для одного и того же метода в виде конвейера промежуточного программного обеспечения.

В npm существует несколько библиотек, упрощающих реализацию ловушек для функций, например `hooks` (https://npmjs.org/package/hooks), `hooker` (https://npmjs.org/package/hooker) и `meld` (https://npmjs.org/package/meld).

Прокси в стандарте ES2015

Стандарт ES2015 определяет глобальный объект `Proxy`, доступный в Node.js, начиная с версии 6.

Программный интерфейс объекта `Proxy` включает конструктор `Proxy`, принимающий аргументы `target` (целевой объект) и `handler` (обработчик):

```
const proxy = new Proxy(target, handler);
```

Здесь `target` представляет объект, к которому применяется прокси (**субъект**, согласно каноническому определению), а `handler` – специальный объект, определяющий модель поведения прокси.

Объект обработчика `handler` содержит ряд необязательных методов с предопределенными именами, называемых **методами-ловушками** (например, `apply`, `get`, `set` и `has`), которые вызываются автоматически при выполнении соответствующих операций над экземпляром, к которому применяется прокси.

Чтобы лучше понять порядок работы этого программного интерфейса, рассмотрим пример:

```
const scientist = {
  name: 'nikola',
  surname: 'tesla'
};

const uppercaseScientist = new Proxy(scientist, {
  get: (target, property) => target[property].toUpperCase()
});

console.log(uppercaseScientist.name, uppercaseScientist.surname);
  // выведет NIKOLA TESLA
```

В этом примере программный интерфейс `Proxy` используется для перехвата обращений к свойствам объекта `target`, в данном случае `scientist`, и перевода исходных значений этих свойств в верхний регистр.

При внимательном рассмотрении примера можно заметить в программном интерфейсе нечто особенное, а именно возможность перехвата доступа к любому атрибуту объекта `target`. Это возможно, потому что программный интерфейс `Proxy` – не просто обертка, упрощающая создание прокси-объектов, как те, что мы определяли в предыдущих разделах этой главы; он тесно интегрирован в язык JavaScript и позволяет разработчикам перехватывать и настраивать множество операций, выполняемых над объектами. Эта особенность открывает новые интересные возможности, которые было сложно реализовать до появления *метапрограммирования*, *перегрузки операторов* и *виртуализации объектов*.

Для пояснения этой идеи рассмотрим еще один пример:

```
const evenNumbers = new Proxy([], {
  get: (target, index) => index * 2,
  has: (target, number) => number % 2 === 0
});

console.log(2 in evenNumbers); // true
console.log(5 in evenNumbers); // false
console.log(evenNumbers[7]);   // 14
```

В этом примере создается виртуальный массив, содержащий все четные числа. Его можно использовать как обычный массив, то есть обращаться к элементам массива, используя обычный синтаксис массивов (например, `evenNumbers[7]`), или проверить наличие элемента в массиве с помощью оператора `in` (например, `2 in evenNumbers`). Массив называется *виртуальным*, потому что он не хранит никаких данных.

Как видите, в качестве `target` этот прокси использует пустой массив и определяет ловушки `get` и `has` в обработчике `handler`:

- ловушка `get` перехватывает обращения к элементам массива, возвращая четное число для заданного индекса;
- ловушка `has` перехватывает вызов оператора `in` и проверяет, является заданное число четным или нет.

Программный интерфейс `Proxy` поддерживает также ряд других интересных ловушек, таких как `set`, `delete` и `construct`, и позволяет создавать прокси, которые можно отзывать по требованию, отключать все ловушки и восстанавливать исходную модель поведения объекта `target`.

Рассмотрение всех этих функций выходит за рамки данной главы; сейчас важно лишь понять, что программный интерфейс `Proxy` обеспечивает мощную основу для реализации шаблона проектирования «Прокси», когда в этом возникнет необходимость.

Более подробную информацию о программном интерфейсе `Proxy`, включая описание всех его возможностей и всех методов-ловушек, можно найти в статье на сайте проекта Mozilla: https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Proxy. Другие полезные сведения можно найти в подробной статье на сайте Google: https://developers.google.com/web/updates/2016/02/es2015-proxies.

Реальное применение

`Mongoose` (http://mongoosejs.com) – популярная библиотека **объектно-документного отображения** (Object-Document Mapping, ODM) для MongoDB. Она внутренне использует пакет `hooks` (https://npmjs.org/package/hooks) для создания ловушек, выполняющих предварительные и заключительные операции при вызове методов `init`, `validate`, `save` и `remove` своих объектов `Document`. Более подробную информацию можно найти в официальной документации на странице: http://mongoosejs.com/docs/middleware.html.

Декоратор

Декоратор – структурный шаблон, суть которого заключается в динамическом расширении модели поведения существующего объекта. Он отличается от классического наследования, добавляя новые функциональные возможности не ко всем объектам одного и того же класса, а только к явно декорированным экземплярам.

Своей реализацией этот шаблон очень похож на шаблон «Прокси», но вместо расширения или изменения модели поведения существующего объекта он дополняет его новыми возможностями, как показано на рис. 6.2.

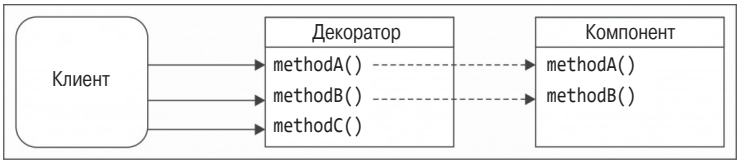

Рис. 6.2 ❖ Схема работы шаблона «Декоратор»

На рис. 6.2 объект Декоратор расширяет объект Компонент, добавляя метод `methodC()`. Существующие методы, как правило, делегируются декорируемому объекту без какой-либо обработки. Конечно, в случае необходимости этот шаблон без особых проблем можно совместить с шаблоном «Прокси» и перехватывать вызовы существующих методов.

Приемы реализации декораторов

Хотя шаблоны «Прокси» и «Декоратор» считаются концептуально разными и предназначенными для разных целей, тем не менее приемы их реализации практически полностью совпадают. Рассмотрим их.

Композиция

При использовании приема композиции декорируемый компонент обертывается новым объектом, который, как правило, наследует его. В этом случае в декораторе просто определяются новые методы, а вызовы существующих делегируются оригинальному компоненту:

```
function decorate(component) {
  const proto = Object.getPrototypeOf(component);

  function Decorator(component) {
    this.component = component;
  }
  Decorator.prototype = Object.create(proto);

  //новый метод
  Decorator.prototype.greetings = function() {
    return 'Hi!';
  };

  //делегируемый метод
  Decorator.prototype.hello = function() {
    return this.component.hello.apply(this.component, arguments);
  };

  return new Decorator(component);
}
```

Расширение объекта

Декорирование объекта можно также осуществить простым присоединением новых методов непосредственно к декорируемому объекту, например:

```
function decorate(component) {
  //новый метод
  component.greetings = () => {
    //...
  };
  return component;
}
```

К шаблону «Декоратор» применимы те же оговорки, что упоминались при рассмотрении шаблона «Прокси». А теперь рассмотрим этот шаблон на реальном примере!

Декорирование базы данных LevelUP

Прежде чем перейти непосредственно к следующему примеру, скажем несколько слов о модуле **LevelUP**, с которым мы будем работать.

Введение в LevelUP и LevelDB

Модуль **LevelUP** (https://npmjs.org/package/levelup) – это обертка вокруг хранилища ключ/значение **LevelDB** от Google для Node.js, первоначально предназначавшегося для реализации IndexedDB в браузере Chrome, но впоследствии получившего более широкое применение. База данных LevelDB определяется Домиником Тарром (Dominic Tarr) как «платформа Node.js в мире баз данных» из-за ее минимализма и расширяемости. Подобно Node.js, база данных LevelDB обеспечивает невероятно

высокую производительность и только базовый набор функций, позволяя разработчикам строить на ее основе практически любые базы данных.

Сообщество Node.js, а конкретно Род Вэгг (Rod Vagg), не упустило шанс использовать эту базу данных в Node.js, создав модуль LevelUP. Возникнув как обертка вокруг базы данных LevelDB, впоследствии этот модуль превратился в средство поддержки нескольких видов хранилищ, от хранилищ в памяти до баз данных NoSQL, таких как Riak и Redis, а также движков веб-хранилищ, таких как IndexedDB и localStorage, позволяя разработчикам использовать один и тот же программный интерфейс как на сервере, так и на клиенте, что открыло перед ними весьма интересные перспективы.

К настоящему времени вокруг модуля LevelUP сформировалась полноценная экосистема из дополнительных плагинов и модулей, расширяющих его крошечное ядро реализациями таких возможностей, как репликация, вторичные индексы, автоматические обновления, движки запросов и многих других. Кроме того, на основе модуля LevelUP разработаны полноценные базы данных, включая клоны CouchDB, такие как PouchDB (https://npmjs.org/package/pouchdb) и CouchUP (https://npmjs.org/package/couchup), и даже графовая база данных levelgraph (https://npmjs.org/package/levelgraph), способные работать как в Node.js, так и в браузере!

Более подробные сведения об экосистеме LevelUP можно найти на странице https://github.com/rvagg/node-levelup/wiki/Modules.

Реализация плагина для LevelUP

Следующий пример демонстрирует создание простого плагина для LevelUP с помощью шаблона «Декоратор», точнее с применением приема расширения объекта, который является самым простым и вместе с тем наиболее прагматичным и эффективным способом придания объектам дополнительных возможностей.

Для удобства будет использован пакет `level` (http://npmjs.org/package/level), связывающий `levelup` и адаптер по умолчанию `leveldown`, использующий LevelDB в качестве основы.

Нам требуется создать плагин для модуля LevelUP, позволяющий получать уведомления при сохранении в базу данных объекта, соответствующего образцу. Например, для образца `{a: 1}` уведомления должны присылаться при попытке сохранить в базу данных такие объекты, как `{a: 1, b: 3}` или `{a: 1, c: 'x'}`.

Начнем разработку плагина с создания нового модуля `levelSubscribe.js`. Поместим в него следующий код:

```
module.exports = function levelSubscribe(db) {

  db.subscribe = (pattern, listener) => {         //[1]
    db.on('put', (key, val) => {                  //[2]
      const match = Object.keys(pattern).every(
        k => (pattern[k] === val[k])              //[3]
      );
      if(match) {
        listener(key, val);                       //[4]
      }
    });
  };

  return db;
};
```

Это – основа плагина, как видите, это было не слишком сложно. Остановимся на некоторых аспектах предыдущего кода.

1. Объект db декорируется новым методом subscribe(). Этот метод просто присоединяется непосредственно к указанному экземпляру db (расширение объекта).
2. Осуществляется обработка всех операций put с базой данных.
3. Выполняется очень простой алгоритм проверки соответствия образцу, который сопоставляет все свойства в образце со свойствами в добавляемых данных.
4. При обнаружении соответствия вызывается обработчик listener.

Для проверки плагина поместим следующий код в файл levelSubscribeTest.js:

```
const level = require('level');                              //[1]
const levelSubscribe = require('./levelSubscribe');          //[2]

let db = level( dirname + '/db', {valueEncoding: 'json'});
db = levelSubscribe(db);

db.subscribe(
  {doctype: 'tweet', language: 'en'},                        //[3]
  (k, val) => console.log(val)
);
db.put('1', {doctype: 'tweet', text: 'Hi', language: 'en'}); //[4]
db.put('2', {doctype: 'company', name: 'ACME Co.'});
```

Вот что он делает:

1) сначала выполняется инициализация базы данных LevelUP, выбираются каталог для хранения файлов и кодировка по умолчанию для значений;
2) затем плагин присоединятся к декорируемому объекту db;
3) теперь можно задействовать новые возможности плагина, вызвав его метод subscribe() и передав ему искомый образец, соответствующий всем объектам со значениями свойств doctype: 'tweet' и language: 'en';
4) и наконец, выполняется сохранение нескольких значений в базе данных с помощью метода put. В ответ на первую попытку произойдет вызов обработчика, и сохраняемый объект будет выведен в консоль, потому что сохраняемый объект соответствует образцу. Однако вторая попытка ничего не выведет в консоль, поскольку сохраняемый объект не соответствует заявленным критериям.

Этот пример демонстрирует реальное применение шаблона «Декоратор» в его простейшей реализации методом расширения объекта. Он может выглядеть тривиальным, но, несомненно, обладает достаточной мощью при надлежащем использовании.

Для простоты этот плагин реализует только операцию put, но его несложно расширить даже для поддержки пакетных операций (https://github.com/rvagg/node-levelup#batch).

Реальное применение

Реальные примеры использования шаблона «Декоратор» можно найти в коде нескольких плагинов для LevelUP:

- level-inverted-index (https://github.com/dominictarr/level-inverted-index): плагин, добавляющий поддержку инвертированных индексов в базу данных LevelUP, которая позволяет выполнять простой текстовый поиск значений, хранящихся в базе данных;

- `level-plus` (https://github.com/eugeneware/levelplus): плагин, добавляющий возможность атомарных обновлений в базе данных LevelUP.

Адаптер

Шаблон «Адаптер» позволяет получить доступ к функциональности объекта с помощью альтернативного интерфейса. Как это следует из названия, реализация шаблона адаптирует объект так, что он может использоваться компонентами, требующими другого интерфейса. Это иллюстрирует схема на рис. 6.3.

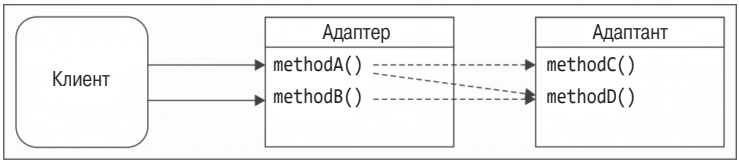

Рис. 6.3 ❖ Схема работы шаблона «Адаптер»

Схема на рис. 6.3 иллюстрирует тот факт, что Адаптер, по сути, является оберткой вокруг Адаптанта (адаптируемого объекта), предоставляя для него другой интерфейс. Схема также подчеркивает, что одна операция Адаптера может объединять вызовы нескольких методов Адаптанта.

Для реализации шаблона чаще применяется прием композиции, когда методы Адаптера служат мостами к методам Адаптанта. Схема шаблона проста и понятна, поэтому сразу перейдем к примеру.

Использование LevelUP через интерфейс файловой системы

Создадим адаптер для программного интерфейса LevelUP, преобразовав его в интерфейс, совместимый с интерфейсом модуля `fs`. В частности, все вызовы `readFile()` и `writeFile()` должны транслироваться в вызовы `db.get()` и `db.put()`, что позволит использовать базу данных LevelUP в качестве хранилища для файловой системы.

Начнем с создания нового модуля `fsAdapter.js`. Сначала загрузим зависимости и экспортируем фабрику `createFsAdapter()`, которая будет использоваться для создания адаптера:

```
const path = require('path');

module.exports = function createFsAdapter(db) {
  const fs = {};
  //...дальше идут следующие фрагменты кода
```

Внутри фабрики реализуем функцию `readFile()`, интерфейс которой совместим с одноименной функцией в модуле `fs`:

```
fs.readFile = (filename, options, callback) => {
  if(typeof options === 'function') {
    callback = options;
    options = {};
  } else if(typeof options === 'string') {
    options = {encoding: options};
```

```
  }
  db.get(path.resolve(filename), {                          //[1]
      valueEncoding: options.encoding
    },
    (err, value) => {
      if(err) {
        if(err.type === 'NotFoundError') {                  //[2]
          err = new Error(`ENOENT, open "${filename}"`);
          err.code = 'ENOENT';
          err.errno = 34;
          err.path = filename;
        }
        return callback && callback(err);
      }
      callback && callback(null, value);                    //[3]
    }
  );
};
```

В предыдущем коде были предприняты дополнительные усилия, чтобы приблизить модель поведения новой функции к оригинальной `fs.readFile()`. Ниже описываются действия, выполняемые новой функцией.

1. Для извлечения файла из класса `db` вызывается метод `db.get()`, которому в качестве ключа передается имя файла, причем всегда используется полный путь к файлу (получаемый с помощью `path.resolve()`). Значение `valueEncoding`, используемое базой данных, устанавливается равным значению параметра `encoding`.
2. Если ключ отсутствует в базе данных, генерируется ошибка с кодом `ENOENT`, который также используется оригинальным модулем `fs`, чтобы сообщить об отсутствии файла. Любые другие ошибки передаются функции обратного вызова `callback` (в рамках этого примера реализована адаптация только наиболее распространенных ошибок).
3. При успешном извлечении пары ключ/значение из базы данных значение передается вызвавшему объекту через функцию обратного вызова `callback`.

Как видите, созданная функция является довольно грубой копией. Она не является идеальной заменой функции `fs.readFile()`, но успешно справляется со своими обязанностями в наиболее распространенных случаях.

Для завершения этого небольшого адаптера рассмотрим реализацию функции `writeFile()`:

```
fs.writeFile = (filename, contents, options, callback) => {
  if(typeof options === 'function') {
    callback = options; options = {};
  } else if(typeof options === 'string') {
    options = {encoding: options};
  }
  db.put(path.resolve(filename), contents, {
    valueEncoding: options.encoding
  }, callback);
}
```

Поскольку мы не ставили цель написать идеальную обертку, мы игнорировали некоторые параметры, например разрешения (`options.mode`), и все получаемые из базы ошибки передаются дальше без какой-либо обработки.

Наконец, нам осталось только вернуть объект `fs` и завершить фабричную функцию следующей строкой кода:

```
  return fs;
}
```

Новый адаптер готов. Написав небольшой тестовый модуль, мы сможем проверить его работу:

```
const fs = require('fs');

fs.writeFile('file.txt', 'Hello!', () => {
  fs.readFile('file.txt', {encoding: 'utf8'}, (err, res) => {
    console.log(res);
  });
});

//попытка прочитать несуществующий файл
fs.readFile('missing.txt', {encoding: 'utf8'}, (err, res) => {
  console.log(err);
});
```

Предыдущий код использует оригинальный программный интерфейс `fs` для выполнения нескольких операций чтения и записи в файловой системе и должен вывести в консоль примерно следующее:

```
{ [Error: ENOENT, open 'missing.txt'] errno: 34, code: 'ENOENT', path: 'missing.txt' }
Hello!
```

А теперь заменим модуль `fs` адаптером:

```
const levelup = require('level');
const fsAdapter = require('./fsAdapter');
const db = levelup('./fsDB', {valueEncoding: 'binary'});
const fs = fsAdapter(db);
```

Эта программа должна вывести то же самое, но при записи и чтении указанных файлов не будет использоваться файловая система, поскольку все операции, выполняемые с помощью адаптера, будут преобразованы в операции с базой данных LevelUP.

Создание такого адаптера можно счесть бесполезным занятием. Действительно, какой смысл использовать базу данных вместо реальной файловой системы? Но не следует забывать, что сам модуль LevelUP включает адаптеры, позволяющие работать с базой данных даже в браузере. Одним из таких адаптеров является `level.js` (https://npmjs.org/package/level-js). Теперь польза от вновь созданного адаптера становится более очевидной – с его помощью можно использовать модуль `fs` в браузере! Например, веб-паук, созданный в *главе 3 «Шаблоны асинхронного выполнения с обратными вызовами»*, использует программный интерфейс `fs` для хранения загруженных веб-страниц. Этот адаптер позволит ему работать в браузере после очень незначительных изменений в коде! Важность адаптеров становится очевидной, когда дело доходит до использования кода в браузере, о чем подробно рассказывается в *главе 8 «Универсальный JavaScript для веб-приложений»*.

Реальное применение

Существует масса реальных примеров применения шаблона «Адаптер»: перечислим лишь наиболее значимые из них, с которыми стоит познакомиться и проанализировать.

- Мы уже знаем, что модуль LevelUP может работать в браузере с различными реализациями хранилищ, от LevelDB до IndexedDB. Это обеспечивают всевозможные адаптеры, открывающие доступ к внутреннему (закрытому) программному интерфейсу модуля LevelUP. Познакомьтесь с реализациями некоторых из них: https://github.com/rvagg/node-levelup/wiki/Modules#storage-back-ends.
- `jugglingdb` – многоцелевая ORM-надстройка для работы с разными базами данных и, конечно же, использует несколько адаптеров для совместимости с ними. Сведения о них можно найти на странице: `https://github.com/1602/jugglingdb/tree/master/lib/adapters`.
- Хорошим дополнением к примеру выше может служить `level-filesystem` (https://www.npmjs.org/package/level-filesystem) – полноценная реализация программного интерфейса `fs`, основанная на модуле LevelUP.

Стратегия

Шаблон «Стратегия» позволяет объекту, который называется *Контекстом*, поддерживать несколько вариантов логики работы путем выделения *переменных* частей в отдельные взаимозаменяемые объекты, называемые *Стратегиями*. Контекст реализует общую логику семейства алгоритмов, тогда как стратегия – переменную часть логики, что позволяет контексту адаптировать свое поведение в зависимости от различных факторов, таких как входные значения, конфигурация системы или предпочтения пользователя. Стратегии обычно являются частью семейства решений и реализуют один и тот же интерфейс, совпадающий с интерфейсом контекста. Схема на рис. 6.4 иллюстрирует только что описанный шаблон.

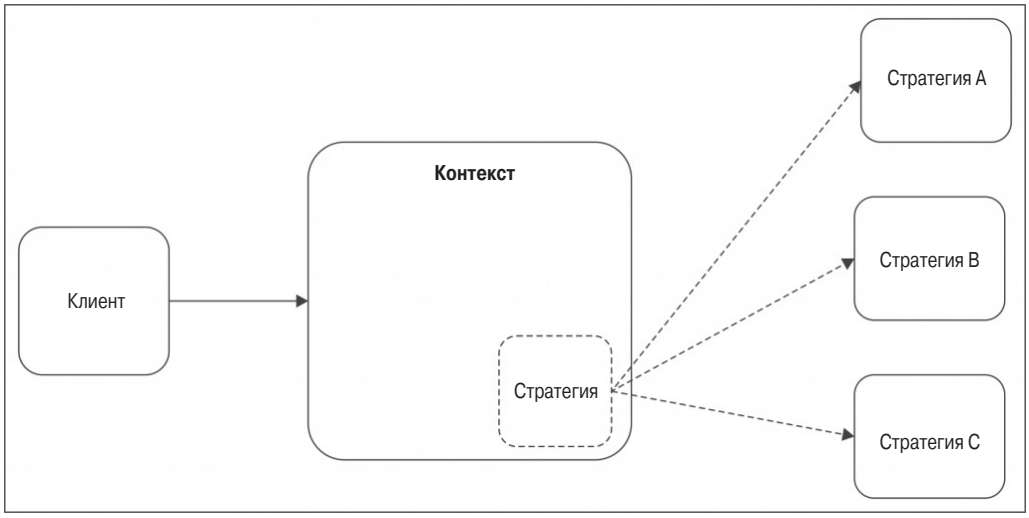

Рис. 6.4 ❖ Схема работы шаблона «Стратегия»

Схема на рис. 6.4 показывает, что объект контекста может включать в свою структуру различные стратегии, как если бы они были сменными частями механизма. Если взять, например, автомобиль, его шины можно считать стратегией адаптации к различным дорожным условиям. На заснеженных дорогах лучше подойдут зимние шины с шипами, в то время как для длительной поездки по автомагистралям предпочтительнее высокоскоростные шины. Но при этом не требуется изменять всю конструкцию автомобиля, то есть, например, снабжать автомобиль восемью колесами, чтобы сделать его вездеходом.

Этот шаблон не только помогает разделять задачи в рамках одного алгоритма, но и обеспечивает большую гибкость и возможность адаптации к различным вариантам одной и той же проблемы.

Шаблон «Стратегия» особенно полезен в ситуациях, когда поддержка вариантов алгоритма требует сложной условной логики (множества операторов `if…else` или `switch`) или смешивания различных алгоритмов из одного семейства. Предположим, что имеется объект `Order`, представляющий онлайн-заказ на сайте интернет-магазина. Этот объект имеет метод `pay()`, завершающий оформление заказа и выполняющий перевод средств со счета пользователя на счет интернет-магазина.

Имеются следующие варианты реализации поддержки различных платежных систем:

- в методе `pay()` можно использовать несколько операторов `if...else` для осуществления оплаты с использованием выбранной платежной системы;
- логику оплаты можно делегировать объекту стратегии для конкретного платежного шлюза, выбранного пользователем.

При выборе первого варианта объект `Order` не сможет обеспечить поддержку нового способа оплаты без внесения изменений в его код. Кроме того, рост числа вариантов оплаты приведет к усложнению кода объекта `Order`. Использование шаблона «Стратегия» позволит объекту `Order` поддерживать практически не ограниченное количество методов оплаты и ограничит его область ответственности сведениями о пользователе, приобретаемых товарах и их ценах, при этом осуществление платежа будет делегировано другому объекту.

А теперь продемонстрируем применение этого шаблона на простом примере.

Объекты для хранения конфигураций в нескольких форматах

Рассмотрим объект `Config`, хранящий параметры настройки приложения, например URL-адрес базы данных, обслуживаемый порт и т. д. Объект `Config` должен иметь не только простой интерфейс для доступа к этим параметрам, но и способ их импорта и экспорта для взаимодействия со средствами постоянного хранения, например с конфигурационным файлом. Кроме того, требуется поддерживать различные форматы хранения конфигурационных данных, например JSON, INI и YAML.

С учетом полученных сведений о шаблоне «Стратегия» можно сразу же выделить переменную часть объекта `Config` – функции сериализации и десериализации конфигурационных данных. Это и станет стратегией.

Создадим новый модуль `config.js` и определим в нем общую часть инструмента управления конфигурацией:

```
const fs = require('fs');
const objectPath = require('object-path');

class Config {
```

```
  constructor(strategy) {
    this.data = {};
    this.strategy = strategy;
  }

  get(path) {
    return objectPath.get(this.data, path);
  }
  //... остальная часть класса
```

Мы заключили конфигурационные данные в переменную экземпляра (`this.data`) и добавили методы `set()` и `get()` для доступа к свойствам конфигурации с помощью нотации путей через точку (например, `property.subProperty`), использовав npm-библиотеку `object-path` (https://npmjs.org/package/object-path). Кроме того, конструктор имеет параметр `strategy`, определяющий алгоритм парсинга и сериализации данных.

Теперь посмотрим, как пользоваться стратегиями, дописав оставшуюся часть класса `Config`:

```
  set(path, value) {
    return objectPath.set(this.data, path, value);
  }

  read(file) {
    console.log(`Deserializing from ${file}`);
    this.data = this.strategy.deserialize(fs.readFileSync(file, 'utf-8'));
  }

  save(file) {
    console.log(`Serializing to ${file}`);
    fs.writeFileSync(file, this.strategy.serialize(this.data));
  }
}
module.exports = Config;
```

Читая конфигурацию из файла, этот код делегирует десериализацию объекту `strategy`. Аналогично, с помощью объекта `strategy`, производится сериализация при сохранении конфигурации в файл. Эта простая конструкция позволяет объекту `Config` поддерживать различные форматы файлов при загрузке и сохранении данных.

Для демонстрации создадим несколько стратегий в файле `strategies.js`. Начнем со стратегии сериализации/десериализации данных в формате JSON:

```
module.exports.json = {
  deserialize: data => JSON.parse(data),
  serialize: data => JSON.stringify(data, null, '  ')
}
```

Действительно, ничего сложного! Стратегия просто реализует согласованный интерфейс, ожидаемый объектом `Config`.

Аналогично следующая стратегия реализует поддержку файлов в формате INI:

```
const ini = require('ini'); //-> https://npmjs.org/package/ini module.exports.ini = {
  deserialize: data => ini.parse(data),
  serialize: data => ini.stringify(data)
}
```

Теперь для проверки работы всего комплекса создадим файл `configTest.js` и попробуем загрузить и сохранить пример конфигурации с использованием различных форматов:

```
const Config = require('./config');
const strategies = require('./strategies');

const jsonConfig = new Config(strategies.json); jsonConfig.read('samples/conf.json');
jsonConfig.set('book.nodejs', 'design patterns'); jsonConfig.save('samples/conf_mod.json');

const iniConfig = new Config(strategies.ini); iniConfig.read('samples/conf.ini');
iniConfig.set('book.nodejs', 'design patterns'); iniConfig.save('samples/conf_mod.ini');
```

Тестовый модуль демонстрирует свойства шаблона «Стратегия». В нем определен единственный класс `Config`, реализующий общую часть управления конфигурацией, в то время как сменяемые стратегии, используемые для сериализации и десериализации, позволяют создавать различные экземпляры `Config`, поддерживающие файлы разных форматов.

В предыдущем примере была представлена только одна из возможных альтернатив выбора стратегии. Ниже перечислены другие возможные подходы.

- Создание двух разных семейств стратегий, одного для десериализации и второго для сериализации. Это позволит читать данные в одном формате и сохранять в другом.
- Динамический выбор стратегии в зависимости от расширения файла. Объект `Config` может хранить отображение `extension->strategy` и использовать его для выбора алгоритма по заданному расширению.

Как видите, имеется несколько вариантов использования стратегий, и правильный выбор зависит только от конкретных требований и нужного баланса между функциональными возможностями и простотой реализации.

Кроме того, реализация самого шаблона может быть разной, например в простейшей форме контекст и стратегия могут быть обычными функциями:

```
function context(strategy) {...}
```

Несмотря на то что предыдущая ситуация может показаться малозначительной, не следует недооценивать особенности такого языка программирования, как JavaScript, где функциям придается большое значение и они используется как полноценные объекты.

Но все эти варианты связывает общая идея шаблона. Как всегда, реализации могут несколько отличаться, но основные идеи остаются в них неизменными.

Реальное применение

`Passport.js` (http://passportjs.org) – фреймворк аутентификации для платформы Node.js, поддерживающий различные схемы аутентификации на веб-сервере. С помощью Passport можно без особых усилий реализовать вход в веб-приложение с помощью учетной записи Facebook или Twitter. Фреймворк `Passport` использует шаблон «Стратегия» для отделения общей логики процесса аутентификации от частей, которые могут меняться, а именно фактического этапа аутентификации. Например, можно использовать OAuth для получения маркера доступа к учетной записи Facebook или Twitter, или задействовать для проверки пары имени пользователя и пароля локальную базу данных. В фреймворке Passport это обеспечивают различные страте-

гии завершения процесса аутентификации, что позволяет ему поддерживать практически не ограниченное количество служб. Взгляните на перечень поддерживаемых провайдеров аутентификации на странице http://passportjs.org/guide/providers, чтобы получить представление о возможностях применения шаблона «Стратегия».

Состояние

Состояние – один из вариантов шаблона «Стратегия», в котором стратегия меняется в зависимости от состояния контекста. В предыдущем разделе был продемонстрирован порядок выбора стратегии на основании различных значений переменных, определяемых предпочтениями пользователя, параметрами конфигурации или вводом, и, после того как выбор был сделан, стратегия оставалась неизменной на все время существования контекста.

В шаблоне «Состояние», напротив, стратегии (в этом случае их называют *состояниями*) являются динамическими и могут изменяться в течение времени существования контекста, что позволяет адаптировать его поведение в зависимости от внутреннего состояния, как показано на рис. 6.5.

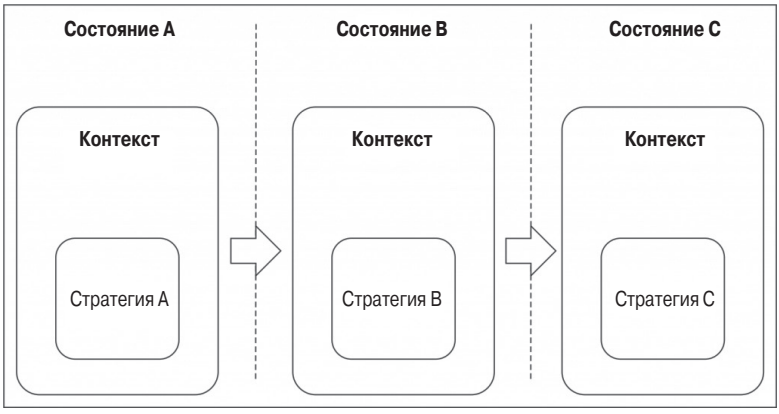

Рис. 6.5 ❖ Схема работы шаблона «Состояние»

Предположим, что имеются система бронирования отелей и объект `Reservation`, моделирующий бронирование номера в гостинице. Это классическая ситуация, когда требуется адаптировать поведение объекта в зависимости от его состояния. Рассмотрим следующую последовательность событий.

1. В момент бронирования пользователь может подтвердить бронь (с помощью `confirm()`), но не может ее отменить (с помощью `cancel()`), поскольку она еще не подтверждена. Однако он может удалить ее (с помощью `delete()`), если передумает.
2. После подтверждения брони вызов функции `confirm()` теряет смысл, но теперь становится возможным отменить бронь, но не удалить ее, поскольку запись о ней должна сохраниться.
3. За день до даты бронирования уже невозможно отменить бронь, поскольку это слишком поздно.

А теперь предположим, что нужно реализовать систему бронирования в виде одного монолитного объекта. Можно представить себе все операторы `if... else` или `switch` для включения и отключения доступа к действиям в зависимости от состояния брони.

В данном случае идеально подойдет шаблон «Состояние», содержащий три стратегии, реализующие три описанных метода (`confirm()`, `cancel()` и `delete()`) и только одну модель поведения, соответствующую моделируемому состоянию. Этот шаблон облегчает переключение объекта `Reservation` с одной модели поведения на другую, поскольку для этого достаточно *активировать* другую стратегию при каждом изменении состояния.

Переход между состояниями может инициироваться и управляться объектом контекста, кодом клиента или самими объектами состояния. Последний вариант предпочтительнее с точки зрения гибкости и независимости, поскольку контексту нет нужды заботиться обо всех возможных состояниях и переключениях между ними.

Реализация простого сокета, защищенного от сбоев

А теперь применим полученные знания на конкретном примере. Создадим клиентский TCP-сокет, обрабатывающий потерю соединения с сервером, – все данные сохраняются в очереди на время, пока сервер вновь не станет доступным, а после восстановления подключения предпринимается попытка отправить их. Этот сокет будет использоваться в контексте простой системы мониторинга, где ряд машин регулярно отправляет некоторые данные об использовании ресурсов. Если принимающий сервер окажется недоступным, сокет будет хранить данные в локальной очереди, пока подключение не восстановится.

Начнем с создания нового модуля `failsafeSocket.js`, представляющего объект контекста:

```
const OfflineState = require('./offlineState');
const OnlineState = require('./onlineState');

class FailsafeSocket{
  constructor (options) {                          //[1]
    this.options = options;
    this.queue = [];
    this.currentState = null;
    this.socket = null;
    this.states = {
      offline: new OfflineState(this),
      online: new OnlineState(this)
    };
    this.changeState('offline');
  }

  changeState (state) {                            //[2]
    console.log('Activating state: ' + state);
    this.currentState = this.states[state];
    this.currentState.activate();
  }

  send(data) {                                     //[3]
    this.currentState.send(data);
```

```
  }
}

module.exports = options => {
  return new FailsafeSocket(options);
};
```

Класс `FailsafeSocket` состоит из трех основных элементов:

1) конструктор инициализирует различные структуры данных, в том числе очередь, куда будут помещаться данные, пока сокет находится в автономном режиме. Кроме того, в нем создается набор из двух состояний: одно – для реализации поведения сокета в автономном режиме и другое – для работы в режиме соединения;
2) метод `changeState()` отвечает за переход из одного состояния в другое. Он просто обновляет значение переменной экземпляра `currentState` и вызывает метод `activate()` целевого состояния;
3) метод `send()` представляет функциональность сокета, которая должна изменяться в зависимости от его состояния. Как видите, это осуществляется путем делегирования операции активному в настоящее время состоянию.

А теперь рассмотрим, что представляют собой два состояния, начав с модуля `offlineState.js`:

```
const jot = require('json-over-tcp');              //[1]
module.exports = class OfflineState {
  constructor (failsafeSocket) {
    this.failsafeSocket = failsafeSocket;
  }

  send(data) {                                     //[2]
    this.failsafeSocket.queue.push(data);
  }

  activate() {                                     //[3]
    const retry = () => {
      setTimeout(() => this.activate(), 500);
    }

    this.failsafeSocket.socket = jot.connect(
      this.failsafeSocket.options,
      () => {
        this.failsafeSocket.socket.removeListener('error', retry);
        this.failsafeSocket.changeState('online');
      }
    );
    this.failsafeSocket.socket.once('error', retry);
  }
};
```

Модуль, созданный нами, отвечает за управление поведением сокета в автономном режиме. Он работает следующим образом:

1) вместо TCP-сокета используется небольшая библиотека `json-over-tcp` (https://npmjs.org/package/json-over-tcp), упрощающая отправку объектов в формате JSON через TCP-подключение;

2) метод `send()` просто помещает переданные ему данные в очередь, поскольку предполагается, что в автономном режиме больше ничего и не требуется;
3) метод `Activate()` пытается установить соединение с сервером с помощью библиотеки `json-over-tcp`. Если эта операция завершается неудачей, попытка повторяется через 500 миллисекунд. Такие попытки продолжаются до тех пор, пока соединение не будет установлено, и тогда объект `failsafeSocket` перейдет в состояние `online`.

Теперь реализуем модуль `onlineState.js` и стратегию `onlineState`:

```
module.exports = class OnlineState {
  constructor(failsafeSocket) {
    this.failsafeSocket = failsafeSocket;
  }

  send(data) {                                          //[1]
    this.failsafeSocket.socket.write(data);
  };

  activate() {                                          //[2]
    this.failsafeSocket.queue.forEach(data => {
      this.failsafeSocket.socket.write(data);
    });
    this.failsafeSocket.queue = [];

    this.failsafeSocket.socket.once('error', () => {
      this.failsafeSocket.changeState('offline');
    });
  }
};
```

Стратегия `OnlineState` довольна проста.

1. Метод `send()` записывает данные непосредственно в сокет, поскольку предполагается, что соединение установлено.
2. Метод `activate()` отсылает все данные из очереди, которые были помещены туда в автономном режиме, а также отслеживает все события `error`, которые воспринимаются как симптом перехода сокета в автономный режим (для простоты). Когда это происходит, выполняется переход в состояние `offline`.

Класс `failsafeSocket` готов. Чтобы проверить его, создадим примеры клиента и сервера. Начнем с серверного кода, поместив его в модуль `server.js`:

```
const jot = require('json-over-tcp');
const server = jot.createServer(5000);
server.on('connection', socket => {
  socket.on('data', data => {
    console.log('Client data', data);
  });
});
server.listen(5000, () => console.log('Started'));
```

Затем поместим код клиента, который представляет наибольший интерес, в модуль `client.js`:

```
const createFailsafeSocket = require('./failsafeSocket');
const failsafeSocket = createFailsafeSocket({port: 5000});
```

```
setInterval(() => {
  //send current memory usage
  failsafeSocket.send(process.memoryUsage());
}, 1000);
```

Сервер просто выводит полученные сообщения в консоль, а клиенты с помощью объекта `FailsafeSocket` отправляют сведения об использованной ими памяти каждую секунду.

Для проверки этой небольшой системы следует запустить клиента и сервер, а затем проверить функциональные возможности класса `failsafeSocket`, остановив и вновь запустив сервер. Состояние клиента должно смениться с `online` на `offline`, а сведения об использовании памяти, собранные в период недоступности сервера и помещенные в очередь, должны быть переданы серверу, как только он вновь включится.

Этот пример служит явным свидетельством, как шаблон «Состояние» может помочь увеличить модульность и удобочитаемость компонента, адаптирующего свое поведение в соответствии со своим состоянием.

Разработанный в этом разделе класс `FailsafeSocket` предназначен только для демонстрации шаблона «Состояние» и не является полноценным решением со 100%-ной надежностью для обработки проблем TCP-сокетов, связанных с потерей соединения. Например, представленное решение не проверяет получения сервером всех данных, записанных в поток сокета, что требует дополнительного кода, который, строго говоря, не относится к рассматриваемому шаблону.

Макет

Следующий шаблон, который мы рассмотрим, носит название «**Макет**». Он имеет много общего с шаблоном «Стратегия». Шаблон «Макет» состоит из определения абстрактного псевдокласса, представляющего костяк алгоритма, в котором некоторые действия остаются неопределенными. Подклассы могут затем *заполнить* пробелы в алгоритме, реализуя отсутствующие действия, называемых **методами макета**. Цель этого шаблона – дать возможность определить семейство классов, охватывающее все варианты алгоритма. UML-схема на рис. 6.6 демонстрирует только что описанную структуру.

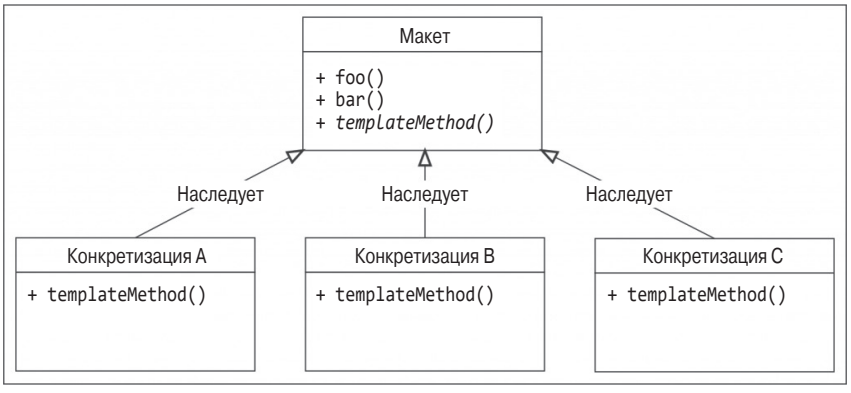

Рис. 6.6 ❖ Структура шаблона «Макет»

Три конкретных класса на рис. 6.6 наследуют класс макета, реализуя метод `templateMethod()`, который является *абстрактным* или *чисто виртуальным*, если пользоваться терминологией языка C++. Применительно к языку JavaScript это означает, что метод остается неопределенным или ему назначена функция, всегда возбуждающая исключение, которое указывает, что метод должен быть реализован. Шаблон «Макет» является более объектно-ориентированным (в классическом понимании), чем все рассмотренные выше, поскольку наследование является основной частью его реализации.

Шаблоны «Макет» и «Стратегия» имеют схожие цели, но отличаются структурой и реализацией. Оба позволяют изменять некоторые части алгоритма при непременном использовании общей части, но если шаблон «Стратегия» позволяет проделывать это динамически, во время выполнения, то шаблон «Макет» заставляет задать весь алгоритм в момент определения конкретного класса. С учетом этого шаблон «Макет» больше подходит в случаях, когда требуется создать заранее подготовленные варианты алгоритма. Поскольку выбор шаблона всегда зависит от разработчика, ему следует учесть все плюсы и минусы каждого из вариантов.

Макет диспетчера конфигурации

Чтобы получить более полное представление об отличиях шаблонов «Стратегия» и «Макет», реализуем повторно объект `Config`, который был определен в разделе с описанием шаблона «Стратегия», но на этот раз с применением шаблона «Макет». Как и предыдущая версия объекта `Config`, данная реализация должна иметь возможность загружать и сохранять конфигурационные свойства в файлах разных форматов.

Начнем с определения класса шаблона `ConfigTemplate`:

```
const fs = require('fs');
const objectPath = require('object-path');
class ConfigTemplate {
  read(file) {
    console.log(`Deserializing from ${file}`);
    this.data = this._deserialize(fs.readFileSync(file, 'utf-8'));
  }

  save(file) {
    console.log(`Serializing to ${file}`);
    fs.writeFileSync(file, this._serialize(this.data));
  }

  get(path) {
    return objectPath.get(this.data, path);
  }

  set(path, value) {
    return objectPath.set(this.data, path, value);
  }

  _serialize() {
    throw new Error('_serialize() must be implemented');
  }

  _deserialize() {
    throw new Error('_deserialize() must be implemented');
```

```
  }
}
module.exports = ConfigTemplate;
```

Новый класс `ConfigTemplate` определяет два метода шаблона – `_deserialize()` и `_serialize()` – для загрузки и сохранения конфигурации. Символ подчеркивания в начале их имен указывает, что они предназначены только для внутреннего использования. Это простейший способ выделения защищенных методов. Поскольку в языке JavaScript нет возможности объявить методы абстрактными, они определяются как заглушки, генерирующие исключения (если не переопределены конкретным подклассом).

Теперь создадим конкретный класс, например загружающий и сохраняющий конфигурацию в формате JSON:

```
const util = require('util');
const ConfigTemplate = require('./configTemplate');
class JsonConfig extends ConfigTemplate {
  _deserialize(data) {
    return JSON.parse(data);
  };

  _serialize(data) {
    return JSON.stringify(data, null, '   ');
  }
}
module.exports = JsonConfig;
```

Класс `JsonConfig` наследует макет, то есть класс `ConfigTemplate`, и включает конкретную реализацию методов `_deserialize()` и `_serialize()`.

Теперь класс `JsonConfig` может использоваться как автономный объект для хранения конфигурации, без необходимости указывать стратегию сериализации и десериализации, поскольку она *внедрена* в сам класс:

```
const JsonConfig = require('./jsonConfig');

constjsonConfig = new JsonConfig();
jsonConfig.read('samples/conf.json');
jsonConfig.set('nodejs', 'design patterns');
jsonConfig.save('samples/conf_mod.json');
```

Шаблон «Макет» позволил с минимальными усилиями получить новый, полностью рабочий диспетчер конфигурации путем повторного использования логики и интерфейса, унаследованных от родительского класса макета, и предоставив только реализацию абстрактных методов.

Реальное применение

Этот шаблон должен выглядеть знакомым. Он уже применялся в *главе 5 «Программирование с применением потоков данных»* для реализации пользовательских потоков данных. В этом контексте роль методов макета играли методы `_write()`, `_read()`, `_transform()` и `_flush()`. Для создания нового пользовательского потока необходимо унаследовать определенный абстрактный класс потока и реализовать методы макета.

Промежуточное программное обеспечение

Одним из наиболее выделяющихся шаблонов в Node.js, безусловно, является шаблон «Промежуточное программное обеспечение». К сожалению, это также один из наиболее запутанных шаблонов для разработчиков, не имеющих достаточного опыта, особенно для тех, кто ранее занимался разработкой корпоративного программного обеспечения. Причина такой дезориентации, вероятно, заключается в понимании самого термина «промежуточное программное обеспечение», который в жаргоне корпоративной архитектуры соответствует различным программным комплексам, предназначенным для абстрагирования от таких низкоуровневых механизмов, как программные интерфейсы ОС, сетевые коммуникации, управление памятью и т. д., чтобы разработчик мог сосредоточиться только на собственной логике приложения. В этом контексте термин «промежуточное программное обеспечение» часто ассоциируется с такими средствами, как CORBA, Enterprise Service Bus, Spring, JBoss, но в его более общем смысле он может означать любой программный слой, соединяющий средства низкоуровневого обслуживания и приложение (буквально: программное обеспечение, располагающееся в середине).

Промежуточное программное обеспечение в Express

Фреймворк Express (http://expressjs.com) сделал термин «промежуточное программное обеспечение» популярным в мире Node.js, привязав его к весьма специфичному шаблону проектирования. В Express промежуточное программное обеспечение представляет набор служб, которые обычно являются функциями, собранными в конвейер и отвечающими за обработку входящих HTTP-запросов и создание надлежащих ответов.

Фреймворк Express представляет собой ненавязчивый минималистский веб-фреймворк. Использование шаблона «Промежуточное программное обеспечение» является эффективной стратегией, упрощающей создание и распространение новых функциональных возможностей, добавляемых к приложению без необходимости наращивать минималистическое ядро фреймворка.

Промежуточное программное обеспечение Express имеет следующую сигнатуру:

```
function(req, res, next) { ... }
```

Здесь `req` представляет входящий HTTP-запрос, `res` – ответ, а `next` – функцию обратного вызова, вызываемую промежуточным программным обеспечением после выполнения им своих заданий, что, в свою очередь, приводит к вызову следующего промежуточного программного обеспечения в конвейере.

Ниже перечислены примеры заданий в конвейере Express:

- парсинг тела запроса;
- сжатие и распаковка запросов и ответов;
- доступ к журналам;
- управление сеансами;
- управление шифрованием данных в cookies;
- защита от **подделки межсайтовых запросов** (Cross-site Request Forgery, CSRF).

Как видите, все эти задания не связаны напрямую с основными функциями приложения или веб-сервера. Они скорее являются аксессуарами, компонентами поддержки остальной части приложения, позволяющими обработчикам запросов сосредото-

читься только на основной логике приложения. По своей сути эти задания являются промежуточным программным обеспечением.

Промежуточное программное обеспечение как шаблон

Технология, используемая для реализации промежуточного программного обеспечения в Express, не нова. Действительно, ее можно рассматривать как воплощение шаблонов «Перехватывающий фильтр» (Intercepting Filter) и «Цепочка обязанностей» (Chain of Responsibility). В более общем смысле под этим термином понимается **конвейер обработки**, напоминающий потоки. В настоящее время использование термина «промежуточное программное обеспечение» на платформе Node.js вышло далеко за границы фреймворка Express и обозначает определенный шаблон, согласно которому набор модулей, фильтров и обработчиков (в форме функций) связывается в асинхронную последовательность для выполнения предварительной и заключительной обработок любого типа данных. Главным преимуществом этого шаблона является его *гибкость*. В самом деле, этот шаблон позволяет создать инфраструктуру дополнительных встраиваемых модулей с невероятно малыми усилиями, обеспечивая ненавязчивый способ расширения системы новыми фильтрами и обработчиками.

Для знакомства с шаблоном «Перехватывающий фильтр» хорошей отправной точкой станет статья: http://www.oracle.com/technetwork/java/interceptingfilter-142169.html. Отличный обзор шаблона «Цепочка обязанностей» можно найти по адресу: http://java.dzone.com/articles/design-patterns-uncovered-chain-of-responsibility.

Схема на рис. 6.7 иллюстрирует компоненты шаблона «Промежуточное программное обеспечение»:

Рис. 6.7 ❖ Схема работы шаблона «Промежуточное программное обеспечение»

Важным компонентом этого шаблона является **диспетчер промежуточного программного обеспечения**, отвечающий за организацию и выполнение заданий. Ниже перечислены наиболее важные детали реализации шаблона.

- Новое промежуточное программное обеспечение можно зарегистрировать вызовом функции `use()` (имя этой функции соответствует общему соглашению, применяемому во многих реализациях шаблона, но, в принципе, можно использовать любое имя). Как правило, новое промежуточное программное обеспечение можно добавлять только в конец конвейера, но и это не является обязательным правилом.

- При получении новых данных, для их обработки запускается асинхронный поток последовательного выполнения зарегистрированных заданий. Каждый блок конвейера получает на входе результат выполнения предыдущего блока.
- Любое задание может принять решение об остановке дальнейшей обработки, просто не вызывая указанную функцию или передав ей признак ошибки. Обычно ошибки вызывают выполнение другой последовательности заданий, специально предназначенной для обработки ошибок.

Не существует единого строгого правила обработки и передачи данных по конвейеру. Возможные следующие варианты:

- расширение данных дополнительными свойствами и функциями;
- замена данных результатами обработки определенного вида;
- поддержание неизменности данных и возврат последних их копий как результатов обработки.

Выбор правильного подхода определяется реализацией диспетчера промежуточного программного обеспечения и самого вида обработки, выполняемой промежуточным программным обеспечением.

Создание фреймворка промежуточного программного обеспечения для ØMQ

А теперь продемонстрируем использование шаблона, создав фреймворк промежуточного программного обеспечения для библиотеки сообщений ØMQ (http://zeromq.org). Библиотека ØMQ (другое ее название – ZMQ или ZeroMQ) предоставляет простой интерфейс для обмена атомарными сообщениями по сети с использованием различных протоколов, отличается высокой производительностью и своим базовым набором абстракций, разработанных специально для упрощения реализации пользовательских инфраструктур обмена сообщениями. По этой причине библиотека ØMQ часто применяется для создания сложных распределенных систем.

В *главе 11* «*Шаблоны обмена сообщениями и интеграции*» библиотека ØMQ будет рассматриваться более подробно.

Библиотека ØMQ имеет чересчур низкоуровневый интерфейс, позволяющий использовать в сообщениях только строки и двоичные буферы, поэтому кодирование или нестандартное форматирование данных должно осуществляться пользователями библиотеки.

В следующем примере мы создадим инфраструктуру промежуточного программного обеспечения для абстракции предварительной и заключительной обработки данных, передаваемых через ØMQ-сокет, которая позволит не только работать с объектами в формате JSON, но и сжимать пересылаемые сообщения.

Перед работой над примером установите библиотеки ØMQ, руководствуясь инструкциями по адресу: http://zeromq.org/intro:get software. Для нужд примера подойдет любая версия из ветви 4.0.

Диспетчер промежуточного программного обеспечения

Первым шагом к созданию инфраструктуры промежуточного программного обеспечения для библиотеки ØMQ станет разработка компонента, отвечающего за выполнение конвейера промежуточного программного обеспечения при получении и отправке но-

вого сообщения. Создадим для этого новый модуль `zmqMiddlewareManager.js` и поместим в него следующий код:

```
module.exports = class ZmqMiddlewareManager {
  constructor(socket) {
    this.socket = socket;
    this.inboundMiddleware = [];                    //[1]
    this.outboundMiddleware = [];
    socket.on('message', message => {               //[2]
      this.executeMiddleware(this.inboundMiddleware, {
        data: message
      });
    });
  }

  send(data) {
    constmessage = {
      data: data
    };

    this.executeMiddleware(this.outboundMiddleware, message,
      () => {
        this.socket.send(message.data);
      }
    );
  }

  use(middleware) {
    if (middleware.inbound) {
      this.inboundMiddleware.push(middleware.inbound);
    }
    if (middleware.outbound) {
      this.outboundMiddleware.unshift(middleware.outbound);
    }
  }

  executeMiddleware(middleware, arg, finish) {
    function iterator(index) {
      if (index === middleware.length) {
        return finish && finish();
      }
      middleware[index].call(this, arg, err => {
        if (err) {
          return console.log('There was an error: ' + err.message);
        }
        iterator.call(this, ++index);
      });
    }

    iterator.call(this, 0);
  }
};
```

В начале класса определяется конструктор нового компонента. Он принимает аргумент с сокетом ØMQ и:

1) создает два пустых списка, которые будут содержать функции промежуточного программного обеспечения, один для обработки входящих сообщений и второй – исходящих;
2) сразу же начинает прием поступающих из сокета новых сообщений, подключая нового обработчика события `'message'`. Обработчик передает входящее сообщение в конвейер `inboundMiddleware`.

Метод `send` класса `ZmqMiddlewareManager` отвечает за выполнение промежуточного программного обеспечения при отправке нового сообщения через сокет.

На этот раз сообщение обрабатывается с помощью фильтров в списке `outboundMiddleware`, а затем передается в метод `socket.send()` для фактической отправки в сеть.

А теперь поговорим о методе `use`. Этот метод предназначен для добавления новых функций в конвейеры. Каждое промежуточное программное обеспечение включает пару функций. В данной реализации объект имеет два свойства, `inbound` и `outbound`, ссылающихся на функции, которые добавляются в соответствующие списки.

Обратите внимание, что функция `inbound` *добавляется* в конец списка `inboundMiddleware`, а функция `outbound` *вставляется* (с помощью `unshift`) в начало списка `outboundMiddleware`. Это объясняется тем, что добавленные функции для обработки входящих и исходящих сообщений должны выполняться в обратном порядке. Например, если входящие сообщения сначала распаковываются, а затем десериализуются с помощью библиотеки JSON, то исходящие сообщения, наоборот, сначала сериализуются, а затем сжимаются.

Важно понимать, что это соглашение об организации промежуточного программного обеспечения в пары не является обязательной частью общего шаблона, а представляет собой особенность реализации данного конкретного примера.

Последняя функция – `executeMiddleware` – ядро нашего компонента. Она отвечает за выполнение функций промежуточного программного обеспечения. Код этой функции должен выглядеть для вас уже знакомым, поскольку является реализацией шаблона асинхронных последовательных итераций, с которым мы познакомились в *главе 3 «Шаблоны асинхронного выполнения с обратными вызовами»*. Она поочередно извлекает функции из массива `middleware` и выполняет их одну за другой, передавая каждой в качестве входного параметра один и тот же объект `arg`. Этот хитрый прием позволяет передавать данные из одной функции в другую. В конце итерации вызывается функция обратного вызова `finish()`.

Для краткости мы опустили обработку ошибок в конвейере промежуточного программного обеспечения. Обычно, когда в одной из функций промежуточного программного обеспечения возникает ошибка, выполняется другой набор функций промежуточного программного обеспечения, специально предназначенный для обработки ошибок. Это можно без особых усилий реализовать с помощью продемонстрированной здесь технологии.

Промежуточное программное обеспечение для поддержки сообщений в формате JSON

Теперь, закончив реализацию диспетчера, можно создать пару функций промежуточного программного обеспечения для демонстрации обработки входящих и исходящих сообщений. Как уже упоминалось ранее, одной из целей нашей инфраструктуры является создание фильтра, сериализующего и десериализующего сообщения в фор-

мате JSON. Создадим для этого новое промежуточное программное обеспечение. В новый модуль `jsonMiddleware.js` поместим следующий код:

```
module.exports.json = () => {
  return {
    inbound: function (message, next) {
      message.data = JSON.parse(message.data.toString());
      next();
    },
    outbound: function (message, next) {
      message.data = new Buffer(JSON.stringify(message.data));
      next();
    }
  }
};
```

Код вновь созданного промежуточного программного обеспечения `json` достаточно прост:

- функция `inbound` десериализует полученное входящее сообщение и присваивает результат обратно свойству `data` объекта `message`, что позволит выполнить дополнительную обработку в контейнере;
- функция `outbound` сериализует данные из `message.data`.

Обратите внимание, насколько отличается промежуточное программное обеспечение, поддерживаемое этим фреймворком, от того, что используется в Express. Это совершенно нормально и наглядно демонстрирует, как можно адаптировать этот шаблон под конкретные требования.

Использование фреймворка промежуточного программного обеспечения ØMQ

Теперь мы готовы задействовать вновь созданную инфраструктуру промежуточного программного обеспечения. Для этого создадим очень простое приложение, состоящее из клиента, посылающего *ping-запрос* через регулярные промежутки времени, и сервера, возвращающего полученное сообщение.

Его реализация будет основана на шаблоне обмена сообщениями запрос/ответ, использующего пару сокетов req/rep, предоставляемую ØMQ (http://zguide.zeromq.org/page:all#Ask-and-Ye-Shall-Receive). Эти сокеты мы обернем диспетчером `zmqMiddlewareManager` для получения всех преимуществ созданной ранее инфраструктуры промежуточного программного обеспечения, включая сериализацию/десериализацию сообщений в формате JSON.

Сервер Начнем с создания серверной части (`server.js`). Модуль начинается с инициализации компонентов:

```
const zmq = require('zmq');
const ZmqMiddlewareManager = require('./zmqMiddlewareManager');
const jsonMiddleware = require('./jsonMiddleware');
const reply = zmq.socket('rep');
reply.bind('tcp://127.0.0.1:5000');
```

Предыдущий код загружает требуемые зависимости и подключает сокет `'rep'` (ответ) ØMQ к локальному порту. Далее инициализируем промежуточное программное обеспечение:

```
const zmqm = new ZmqMiddlewareManager(reply);
zmqm.use(jsonMiddleware.json());
```

Мы создали новый объект `ZmqMiddlewareManager` и добавили два элемента промежуточного программного обеспечения, один для сжатия/распаковки сообщений и другой для десериализации/сериализации сообщений в формате JSON.

Для краткости мы опустили реализацию промежуточного программного обеспечения `zlib`, но ее можно найти в загружаемых примерах кода.

Теперь можно переходить к обработке запросов, поступающих от клиента. Для этого достаточно добавить новое промежуточное программное обеспечение, на этот раз используемое в качестве обработчика запросов:

```
zmqm.use({
  inbound: function (message, next) {
    console.log('Received: ', message.data);
    if (message.data.action === 'ping') {
      this.send({action: 'pong', echo: message.data.echo});
    }
    next();
  }
});
```

Этот последний элемент промежуточного программного обеспечения, определяемый после `zlib` и `json`, будет использоваться для распаковки и десериализации сообщения, помещенного в переменную `message.data`. С другой стороны, все передаваемые методу `send()` данные будут обработаны промежуточным программным обеспечением `outbound`, которое в данном случае сериализует, а затем сжимает данные.

Клиент На стороне клиента, `client.js`, этого небольшого приложения сначала следует инициализировать новый сокет `'req'` (запрос) ØMQ, подключенный к порту 5000:

```
const zmq = require('zmq');
const ZmqMiddlewareManager = require('./zmqMiddlewareManager');
const jsonMiddleware = require('./jsonMiddleware');

const request = zmq.socket('req');
request.connect('tcp://127.0.0.1:5000');
```

Затем настроить фреймворк промежуточного программного обеспечения, как это было сделано для сервера:

```
const zmqm = new ZmqMiddlewareManager(request);
zmqm.use(jsonMiddleware.json());
```

Создать элемент промежуточного программного обеспечения для обработки входящих сообщений, поступающих от сервера:

```
zmqm.use({
  inbound: function (message, next) {
    console.log('Echoed back: ', message.data);
    next();
  }
});
```

Предыдущий код просто выводит все поступающие ответы в консоль.

И наконец, создадим таймер для регулярной отправки ping-запросов, используя при этом `mqMiddlewareManager`, чтобы воспользоваться преимуществами промежуточного программного обеспечения:

```
setInterval( () => {
  zmqm.send({action: 'ping', echo: Date.now()});
}, 1000);
```

Обратите внимание, что все элементы `inbound` и `outbound` явно определены с помощью ключевого слова `function`, без использования синтаксиса стрелочных функций. Это сделано намеренно, поскольку, как уже упоминалось в *главе 1 «Добро пожаловать в платформу Node.js»*, объявление стрелочной функции ограничивает область видимости функции ее лексической областью. Метод `call` , используемый для вызова функций, не влияет на внутреннюю область видимости стрелочных функций. Другими словами, если использовать стрелочную функцию, промежуточное программное обеспечение не распознает экземпляр `zmqMiddlewareManager` и возникнет ошибка `"TypeError: this.send is not a function"`.

Теперь можно проверить работу приложения. Первым запустим сервер:

```
node server
```

Затем клиента:

```
node client
```

В этот момент клиент должен начать отправлять сообщения и получать их обратно от сервера.

Фреймворк промежуточного программного обеспечения справляется со своей работой, он обеспечивает распаковку/сжатие и десериализацию/сериализацию сообщений, оставляя обработчикам выполнение их собственных задач!

Промежуточное программное обеспечение, использующее генераторы Koa

В предыдущих параграфах была рассмотрена реализация шаблона «Промежуточное программное обеспечение» с помощью обратных вызовов, на примере системы обмена сообщениями.

Как можно было заметить, шаблон «Промежуточное программное обеспечение» особенно хорошо подходит для реализации веб-фреймворков, как удобный механизм создания «слоев» логики, работающих с потоками данных, протекающими через ядро приложения.

Помимо Express, существуют другие веб-фреймворки, которые используют шаблон промежуточного программного обеспечения, например Koa (http://koajs.com/). Фреймворк Koa особенно интересен своим радикальным выбором реализации шаблона промежуточного программного обеспечения, заключающимся в применении только функций-генераторов, соответствующих спецификации ES2015, и полным отказом от обратных вызовов. Как будет показано ниже, этот выбор значительно упрощает процесс реализации промежуточного программного обеспечения. Но, прежде чем перейти к практическим примерам, рассмотрим схему на рис. 6.8, демонстрирующую еще один способ реализации шаблона, характерный для этого веб-фреймворка:

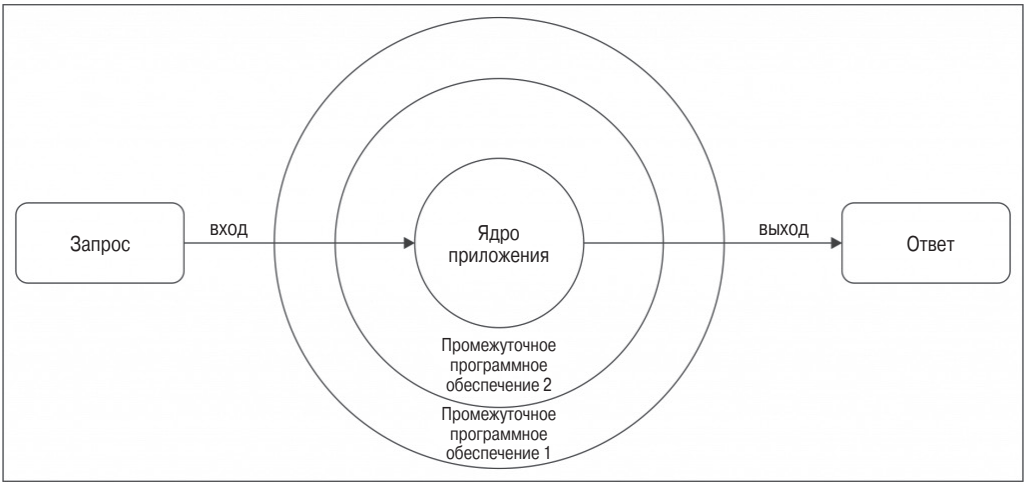

Рис. 6.8 ❖ Еще один подход к реализации шаблона «Промежуточное программное обеспечение»

Как показано на рис. 6.8, прежде чем попасть в ядро приложения, входящий запрос проходит через несколько промежуточных уровней программного обеспечения. Эта часть потока называется **входящей**, или **втекающей**. После достижения ядра приложения поток проходит все промежуточные уровни в обратном порядке. Это позволяет промежуточному программному обеспечению выполнить другие действия после основной логики приложения, перед отправкой ответа пользователю. Эта часть потока называется **исходящей**, или **вытекающей**.

Организацию, представленную на рис. 6.8, иногда называют «луковой», потому что промежуточное программное обертывает ядро приложения, подобно луковой шелухе.

А теперь создадим новое веб-приложение с использованием фреймворка Koa, чтобы продемонстрировать простоту реализации пользовательского промежуточного программного обеспечения с помощью функций-генераторов.

Это приложение реализует простой программный интерфейс, возвращающий текущее время на сервере в формате JSON.

Начнем с установки фреймворка Koa:

```
npm install koa
```

Затем создадим новый модуль `app.js`:

```
const app = require('koa')();

app.use(function *(){
  this.body = {"now": new Date()};
});

app.listen(3000);
```

Обратите внимание, что в данном случае ядро приложения определяется как функция-генератор в методе `app.use`. Ниже будет показано, что промежуточное программное обеспечение добавляется в приложение точно так же, и по существу ядро

приложения является не чем иным, как последним, или самым внутренним, слоем промежуточного программного обеспечения (и который не требуется возвращать другим уровням промежуточного программного обеспечения).

Первый проект приложения готов. Запустите его командой

```
node app.js
```

Затем, перейдя в браузере по адресу `http://localhost:3000`, можно увидеть результат его работы.

Обратите внимание, что фреймворк Koa сам позаботился о преобразовании ответа в JSON-строку и добавил правильный заголовок `content-type`, когда мы передали в теле ответа объект JavaScript.

Программный интерфейс работает. Теперь желательно защитить его от злоупотреблений и гарантировать обработку не более одного запроса от одного и того же пользователя в секунду.

Эту логику можно считать внешней по отношению к основной логике программного интерфейса, поэтому реализуем ее в виде нового, отдельного слоя промежуточного программного обеспечения. Создадим для этого еще один модуль `rateLimit.js`:

```
const lastCall = new Map();
module.exports = function *(next) {
  // вход
  const now = new Date();
  if (lastCall.has(this.ip) && now.getTime() -
      lastCall.get(this.ip).getTime() < 1000) {
    return this.status = 429; // Слишком много запросов
  }
  yield next;

  // выход
  lastCall.set(this.ip, now);
  this.set('X-RateLimit-Reset', now.getTime() + 1000);
};
```

Этот модуль экспортирует функцию-генератор, реализующую логику промежуточного программного обеспечения.

Первое, что нужно отметить, – это использование объекта `Map` для хранения времени получения последнего вызова от пользователя с данным IP-адресом. Этот объект `Map` будет применяться как своеобразная база данных в памяти для проверки частоты поступления запросов от конкретного пользователя. Конечно, такая реализация представляет собой всего лишь пример и не является идеальной при реальном применении, когда лучше всего использовать внешние накопители, такие как *Redis* или *Memcache*, и более совершенную логику для обнаружения избыточной нагрузки.

Как видите, тело промежуточного программного обеспечения состоит из двух логических частей, *входящей* и *исходящей*, разделенных оператором `yield next`. Входящая часть выполняется до ядра приложения, поэтому здесь проверяется превышение ограничения частоты вызовов. Если ограничение превышено, пользователю возвращается код состояния HTTP *429* (слишком много запросов), и выполняется возврат, чтобы прервать поток выполнения.

Переход к следующему слою промежуточного программного обеспечения осуществляется оператором `yield next`. В нем заключено все волшебство: оператор `yield`

в функции-генераторе приостанавливает ее выполнение и передает управление следующему слою промежуточного программного обеспечения в списке, и только после выполнения последнего слоя (ядра приложения) начинает формироваться исходящий поток, и управление передается промежуточному программному обеспечению в обратном порядке, пока вновь не будет вызван первый слой.

Когда промежуточное программное обеспечение снова получит управление и функция-генератор возобновится, она сохранит время успешного вызова и добавит в запрос заголовок `X-RateLimit-Reset`, оповещающий пользователя о том, когда можно будет сделать новый запрос.

Полноценную надежную реализацию ограничения частоты запросов можно найти в модуле `koajs/ratelimit`: https://github.com/koajs/ratelimit.

Для подключения этого промежуточного программного обеспечения нужно добавить следующую строку в файл `app.js` перед уже имеющимся вывозом метода `app.use`, содержащим логику ядра приложения:

```
app.use(require('./rateLimit'));
```

Для проверки нового приложения перезагрузите сервер и снова откройте браузер. Если быстро обновить страницу несколько раз, вам наверняка удастся превысить ограничение частоты вызовов, и перед вами появится сообщение об ошибке «*Too Many Requests*». Это сообщение автоматически добавляется фреймворком Koa при установке кода состояния 429 и пустого тела ответа.

Действующую реализацию шаблона промежуточного программного обеспечения на базе генераторов можно найти в репозитории `koajs/compose` (https://github.com/koajs/compose), где хранится основной модуль для преобразования массива генераторов в новый генератор, выполняющий оригинальные генераторы в конвейере.

Команда

Другим важным шаблоном проектирования в Node.js является шаблон **«Команда»**. В общем случае командой может считаться любой объект, инкапсулирующий всю информацию, необходимую для выполнения действия. То есть вместо непосредственного вызова метода или функции создается объект, представляющий намерение выполнить такой вызов. Ответственность за материализацию намерения возлагается на другой компонент, преобразующий это намерение в фактические действия. Традиционно этот шаблон базируется на четырех основных компонентах, как это показано на рис. 6.9.

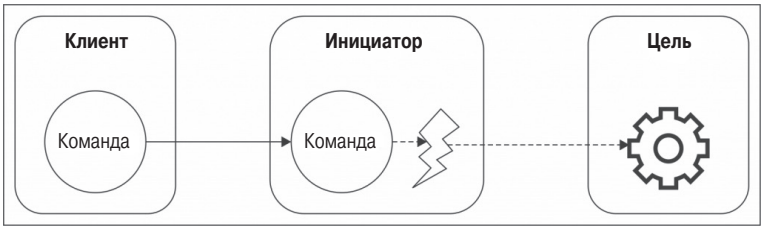

Рис. 6.9 ❖ Схема работы шаблона «Команда»

Типичную организацию шаблона можно описать так.

- **Команда**: объект, инкапсулирующий сведения, необходимые для вызова метода или функции.
- **Клиент**: создает команду и передает ее инициатору.
- **Инициатор**: отвечает за выполнение команды целью.
- **Цель** (или **Получатель**): субъект инициатора. Может быть отдельной функцией или методом объекта.

Как будет показано ниже, эти четыре компонента могут существенно изменяться в зависимости от реализации шаблона, как мы уже неоднократно наблюдали в других шаблонах.

Применение шаблона «Команда» вместо непосредственного выполнения операции имеет ряд преимуществ и полезных применений:

- выполнение команды можно запланировать на более позднее время;
- команды могут сериализоваться и пересылаться по сети. Это позволяет раздавать задания удаленным компьютерам, передавать команды с браузера на сервер, создавать RPC-системы и т. д.;
- команды упрощают хранение истории операций, выполненных системой;
- команды являются важной частью некоторых алгоритмов синхронизации данных и разрешения конфликтов;
- запланированную команду можно отменить, если необходимость в ней отпала. Кроме того, имеется возможность вернуть (откатить) приложение к состоянию, предшествовавшему выполнению команды;
- команды можно группировать. Это позволяет создавать атомарные транзакции или реализовать механизмы, выполняющие все операции в группе одновременно;
- к наборам команд можно применять различные преобразования, например удалять дубликаты, соединять и разделять их, или применять более сложные алгоритмы, такие как **оперативные преобразования** (Operational Transformation, OT), являющиеся основой для большей части современного программного обеспечения для совместной работы, работающего в режиме реального времени, например для совместного редактирования текста.

Более подробные сведения об OT можно найти на странице: http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation.

Приведенный перечень демонстрирует важность этого шаблона, особенно на таких платформах, как Node.js, где сетям и асинхронному выполнению уделяется большое внимание.

Гибкость шаблона

Как уже упоминалось, шаблон «Команда» в языке JavaScript можно реализовать разными способами. Мы рассмотрим лишь некоторые из них, только чтобы продемонстрировать основную идею.

Шаблон «Задание»

Начнем с самой простой реализации: **шаблона «Задание»**. Простейшим способом создания объектного представления вызова функции на языке JavaScript, конечно же, является замыкание:

```
function createTask(target, args) {
  return () => {
    target.apply(null, args);
  }
}
```

Этот шаблон уже рассматривался и неоднократно использовался на протяжении всей книги, в частности в *главе 3 «Шаблоны асинхронного выполнения с обратными вызовами»*. Этот прием позволял использовать отдельный компонент для управления и планирования выполнением заданий, по сути являющийся эквивалентом инициатора в шаблоне «Команда». Например, вспомните порядок определения заданий для передачи библиотеке `async`. Или, еще лучше, вспомните совместное использование преобразователей вместе с генераторами. Сам шаблон обратных вызовов можно рассматривать как упрощенную версию шаблона «Команда».

Более сложные команды

А теперь рассмотрим пример создания более сложных команд, с поддержкой *отката* и *сериализации*. Начнем с определения *цели* команд, небольшого объекта, отвечающего за отправку сведений об изменении состояния в службу, похожую на Twitter. Для простоты используем фиктивную реализацию такой службы:

```
const statusUpdateService = {
  statusUpdates: {},
  sendUpdate: function(status) {
    console.log('Status sent: ' + status);
    let id = Math.floor(Math.random() * 1000000);
    statusUpdateService.statusUpdates[id] = status;
    return id;
  },

  destroyUpdate: id => {
    console.log('Status removed: ' + id);
    delete statusUpdateService.statusUpdates[id];
  }
};
```

Теперь создадим команду, посылающую сведения о новом состоянии:

```
function createSendStatusCmd(service, status) {
  let postId = null;

  const command = () => {
    postId = service.sendUpdate(status);
  };

  command.undo = () => {
    if(postId) {
      service.destroyUpdate(postId);
      postId = null;
    }
  };

  command.serialize = () => {
    return {type: 'status', action: 'post', status: status};
```

```
  };
  return command;
}
```

Функция выше является фабрикой, создающей новые команды *sendStatus*. Каждая команда реализует следующие три возможности:

1) сама команда является функцией, которая при ее вызове запускает действие, другими словами, она реализует шаблон «Задание», рассмотренный выше. При выполнении команда отправляет новое сообщение об изменении состояния, используя методы целевой службы;
2) функция `undo()`, присоединенная к основному заданию, отменяет результат операции. В данном случае просто вызывается метод `destroyUpdate()` целевой службы;
3) функция `Serialize()` создает объект JSON, содержащий всю информацию, необходимую для воссоздания объекта команды.

Теперь реализуем инициатора, начав с конструктора и метода `run()`:

```
class Invoker {
  constructor() {
    this.history = [];
  }
  run (cmd) {
    this.history.push(cmd);
    cmd();
    console.log('Command executed', cmd.serialize());
  }
}
```

Метод `run()` является простейшей реализацией инициатора. Этот метод отвечает за сохранение команды в переменной экземпляра `history` с последующим ее выполнением. Далее добавим новый метод, задерживающий выполнение команды:

```
delay (cmd, delay) {
  setTimeout(() => {
    this.run(cmd);
  }, delay)
}
```

Затем реализуем метод `undo()`, отменяющий последнюю команду:

```
undo () {
  const cmd = this.history.pop();
  cmd.undo();
  console.log('Command undone', cmd.serialize());
}
```

И наконец, чтобы иметь возможность запускать команды на удаленном сервере, необходимо реализовать их сериализацию и передачу по сети с помощью веб-службы:

```
  runRemotely (cmd) {
    request.post('http://localhost:3000/cmd',
      {json: cmd.serialize()},
```

```
      err => {
        console.log('Command executed remotely', cmd.serialize());
      }
    );
  }
}
```

Теперь, когда у нас имеются команда, инициатор и целевой объект, единственным недостающим компонентом остается клиент. Начнем с создания экземпляра `Invoker`:

```
const invoker = new Invoker();
```

Затем создадим команду:

```
const command = createSendStatusCmd(statusUpdateService, 'HI!');
```

Теперь у нас есть команда, предоставляющая отправку сообщения о состоянии, и ее можно сразу же отослать:

```
invoker.run(command);
```

Ох, это было ошибкой, поэтому вернемся к состоянию, предшествовавшему отправке последнего сообщения:

```
invoker.undo();
```

Можно также запланировать отправку сообщения через час:

```
invoker.delay(command, 1000 * 60 * 60);
```

Кроме того, можно распределить нагрузку приложения путем передачи задания на другую машину:

```
invoker.runRemotely(command);
```

Этот небольшой пример демонстрирует, что обертывание операций командами открывает широкие возможности, и это только вершина айсберга.

В заключение отметим, что полноценная реализация шаблона «Команда» используется, только когда это действительно необходимо. Выше было продемонстрировано, сколько дополнительного кода требуется для простого вызова метода `statusUpdateService`. Если достаточно такого простого вызова, сложные команды будут излишними. Если необходимо запланировать выполнение задания или выполнить операцию асинхронно, применение более простого *шаблона «Задание»* станет оптимальным компромиссным решением. Но когда требуются такие дополнительные функции, как откат, преобразование, разрешение конфликтов или что-то еще из возможностей, перечисленных выше, без более сложного предоставления команды не обойтись.

Итоги

В этой главе мы узнали, как реализуются некоторые из традиционных шаблонов проектирования GoF на языке JavaScript и, в частности, на платформе Node.js. Для этого некоторые из них были преобразованы, некоторые упрощены, а прочие переименованы или адаптированы для приведения их в соответствие с нуждами языка, платформы и сообщества. Было показано, как простые шаблоны, такие как «Фабрика», могут значительно улучшить гибкость кода, и как с помощью шаблонов «Прокси», «Деко-

ратор» и «Адаптер» можно изменять, расширять и адаптировать интерфейсы существующих объектов. С другой стороны, шаблоны «Стратегия», «Состояние» и «Макет» обеспечивают разделение алгоритмов на статические и переменные части, что позволяет улучшить повторное использование кода и расширять компоненты. Шаблон «Промежуточное программное обеспечение» предоставляет простую, расширяемую и элегантную парадигму обработки данных. И наконец, шаблон «Команда» является простой абстракцией, делающей любую операцию более гибкой и мощной.

Кроме этих шаблонов проектирования, широко распространенных во многих языках, было рассмотрено несколько новых шаблонов, возникших и ставших популярными внутри сообщества JavaScript. Среди них шаблоны *«Открытый конструктор»* и *«Составные фабричные функции»*. Эти шаблоны предназначены для решения проблем, специфичных для языка JavaScript, таких как *асинхронное выполнение* и *программирование на основе прототипов*.

И наконец, мы получили более полное представление о том, как лучше на языке JavaScript создавать различные многократно используемые объекты и функции, а не расширять множество малых классов или интерфейсов. Кроме того, многим разработчикам, переходящим с других объектно-ориентированных языков программирования, покажется странным порядок реализации шаблонов на языке JavaScript. Наличие не одного, а довольно большого количества различных способов реализации одного и того же шаблона может привести их в замешательство.

Выше уже упоминалось, что язык JavaScript является прагматичным языком. Он позволяет быстро добиваться результата, но при этом отсутствие четкой методики разработки чревато неприятностями. Решению этой проблемы посвящена вся эта книга и, в частности, эта глава. Здесь делается попытка научить нахождению оптимального баланса между творчеством и дисциплиной. Данная глава не только знакомит с шаблонами, направленными на улучшение кода, но и демонстрирует, что их реализация не является самым важным аспектом, поскольку она может существенно меняться и даже перекрываться другими шаблонами. Действительно важны лишь планирование, методики и идеи, на которых основывается шаблон. Это и есть та информация, которая способна помочь улучшить и облегчить разработку приложений для платформы Node.js.

В следующей главе будет рассмотрено несколько дополнительных шаблонов проектирования, связанных с одним из самых важных аспектов программирования, а именно с организацией и связыванием модулей.

Глава 7

Связывание модулей

Система модулей в Node.js блестяще справляется с заполнением старого пробела языка JavaScript, заключающегося в отсутствии встроенных способов организации кода в отдельные модули. Одним из ее главных преимуществ является возможность связывать модули с помощью функции `require()` (как это было продемонстрировано *главе 2 «Основные шаблоны Node.js»*), что является простым, но в то же время и мощным подходом. Но разработчиков, недавно начавших работать с платформой Node.js, принятый подход обычно смущает, и они часто спрашивают: *как лучше передать экземпляр компонента X в модуль Y?*

Иногда это приводит к отчаянным поискам шаблона «Одиночка» как более знакомого способа соединения модулей. Некоторые, напротив, злоупотребляют использованием шаблона «Внедрение зависимостей», применяя его без особых на то причин для обеспечения зависимостей любого типа (даже без состояния). Поэтому не следует удивляться, что тема **связывания модулей** в Node.js является одной из самых спорных и неоднозначных.

В этой области существует масса течений, но ни одно из них нельзя рассматривать как истину в последней инстанции. Все эти подходы имеют свои плюсы и минусы и часто смешиваются в одном приложении, адаптируются, подгоняются под себя или маскируются другими названиями.

В этой главе будут рассмотрены различные подходы к связыванию модулей, проанализированы их сильные и слабые стороны, что поможет сделать рациональный выбор и смешивать подходы для достижения требуемого баланса между простотой, повторным использованием и расширяемостью. В частности, мы познакомимся со следующими важными шаблонами:

- Жесткие зависимости;
- Внедрение зависимостей;
- Локаторы служб;
- Контейнеры для внедряемых зависимостей.

Также будут рассмотрены сопутствующие вопросы, такие как подключение плагинов. Эту проблему можно рассматривать как особый случай связывания модулей, реализуемый как обычно, но имеющий несколько иную область применения, со своими собственными проблемами, особенно когда плагин распространяется в виде отдельного пакета для платформы Node.js. Будут описаны основные методы создания архитектур с поддержкой плагинов и порядок интеграции плагинов в поток основного приложения.

К концу этой главы мы сбросим покров таинственности с искусства связывания модулей в Node.js.

Модули и зависимости

Практически каждое современное приложение является результатом объединения нескольких компонентов, и по мере роста его размеров порядок связывания становится все более важным. Это связано не только с такими чисто техническими аспектами, как расширяемость, но и с походом к восприятию системы. Запутанность **дерева зависимостей** мешает и увеличивает **техническую сложность** проекта, поскольку любые изменения в коде с целью модификации или расширения функциональности могут стоить огромных усилий.

В худшем случае компоненты могут оказаться настолько тесно связанными между собой, что становится невозможным что-либо добавить или изменить без реорганизации или даже создания заново некоторых частей приложения. Это, конечно, не означает, что следует заняться проектированием связей, начиная с самого первого модуля, но изначальная хорошая сбалансированность необходима.

Платформа Node.js предоставляет отличный инструмент для организации и связывания компонентов приложения: ее система модулей CommonJS. Однако само по себе применение системы модулей не является гарантией успеха, потому что, с одной стороны, она обеспечивает определенный *уровень косвенности* между модулем и его зависимостью, а с другой – способна привести к образованию жестких связей при ненадлежащем использовании. В этом разделе будут рассмотрены фундаментальные вопросы связывания зависимостей на платформе Node.js.

Наиболее типичные зависимости в Node.js

С точки зрения архитектуры программного обеспечения любая сущность, состояние или формат данных, оказывающая влияние на модель поведения или структуру компонента, рассматривается как *зависимость*. Например, компонент может использовать услуги, предоставляемые другим компонентом, основываться на глобальном состоянии системы или реализовать конкретный коммуникационный протокол для обмена информацией с другими компонентами и т. д. Идея зависимости весьма широка, и ее границы часто размыты.

Однако в Node.js легко выявить и идентифицировать один из основных видов типичных зависимостей. Конечно же, речь идет о **зависимостях между модулями**. Модули являются основным механизмом организации и структурирования кода. Разработка большого приложения без опоры на систему модулей выглядела бы крайне неразумным предприятием. Правильная группировка элементов приложения обеспечивает массу преимуществ. Фактически модули обладают следующими свойствами:

- модули удобочитаемы и понятны, поскольку они четко направлены (в идеале);
- размещение в отдельных файлах облегчает их идентификацию;
- модули обеспечивают повторное их использование в различных приложениях.

Модули обеспечивают идеальные условия для **сокрытия информации** и предлагают эффективный механизм экспортирования только общедоступного интерфейса компонента (с помощью `module.exports`).

Однако одного только разделения функциональности приложения или библиотеки на модули недостаточно для успешной разработки, поскольку это должно быть сделано надлежащим образом. В частности, при неправильном разделении на модули связи между ними могут стать настолько жесткими, что, по сути, будет создана моно-

литная сущность, где удаление или замена одного из модулей отразится практически на всей архитектуре. Вы должны сразу же уяснить, что порядок деления кода на модули и их связывание играют огромную роль. И, как это делается при решении любой проблемы в сфере разработки программного обеспечения, в этом вопросе важно найти оптимальный баланс между различными целями.

Сцепленность и связанность

Двумя наиболее важными свойствами, определяющими баланс при построении модулей, являются **сцепленность**[1] и **связанность**. Они применимы к любым компонентам и подсистемам программного обеспечения, поэтому их можно использовать в качестве руководящих принципов при разработке модулей на платформе Node.js. Эти свойства можно определить следующим образом:

- **сцепленность (cohesion)**: мера корреляции функций компонента между собой. Например, модуль, предназначенный для выполнения *только одного действия*, когда все его элементы подчинены решению одной задачи, обладает *высокой сцепленностью*. Модуль, содержащий функции для сохранения объектов любого типа в базу данных, такие как `saveProduct()`, `saveInvoice()`, `saveUser()` и т. д., имеет *низкую сцепленность*;
- **связанность (coupling)**: мера зависимости от других компонентов системы. Например, модуль *тесно связан* с другим модулем, если непосредственно читает или изменяет данные другого модуля. Кроме того, два модуля, взаимодействующих через глобальное или общее состояние, также тесно связаны. С другой стороны, два модуля, взаимодействующих только через передачу параметров, *слабо связаны*.

Наиболее желательна высокая сцепленность при слабой связанности, что обеспечивает наглядность, возможность повторного использования и расширяемость модулей.

Модули с поддержкой состояния

В JavaScript все сущее является объектом. Здесь нет абстрактных понятий, таких как чистые интерфейсы или классы, поскольку динамическая типизация сама по себе обеспечивает естественный механизм для отделения **интерфейсов** (или **политик**) от **реализаций** (или **подробностей**). Это одна из причин, почему реализации некоторых шаблонов проектирования в *главе 6 «Шаблоны проектирования»* выглядели настолько упрощенными, по сравнению с традиционными их реализациями.

Проблема отделения интерфейсов от реализации в JavaScript сведена к минимуму, тем не менее использование системы модулей платформы Node.js ведет к созданию жесткой связи с одной конкретной реализацией. В нормальных условиях нет ничего плохого в том, что метод `require()` применяется для загрузки модуля, экспортирующего экземпляр с собственным состоянием, например дескриптор `db`, экземпляр HTTP-сервера, экземпляр службы или вообще любой объект с состоянием, что очень похоже на работу с объектом-одиночкой (синглтоном) и привносит свои плюсы и минусы путем добавления некоторых оговорок.

---

[1] Иногда употребляется термин «связность», но из-за сходства слов «связность» и «связанность» при переводе было отдано предпочтение термину «сцепленность». – *Прим. ред.*

Шаблон «Одиночка» в Node.js

Многие разработчики, недавно работающие с платформой Node.js, не понимают, как правильно реализовать шаблон «Одиночка», и обычно ограничиваются совместным использованием экземпляра различными модулями приложения. Но решение этого вопроса на платформе Node.js легче, чем можно было бы предположить: экспорта экземпляра с помощью `module.exports` уже достаточно для получения чего-то, очень похожего на шаблон «Одиночка». Например, рассмотрим следующую строку кода:

```
//модуль 'db.js'
module.exports = new Database('my-app-db');
```

Простое экспортирование нового экземпляра базы данных дает возможность предположить, что в текущем пакете (которым может быть код всего приложения) будет существовать только один экземпляр модуля `db`. Это возможно благодаря механизму в Node.js, который кэширует модули после первого вызова метода `require()`, и последующие его вызовы просто возвращают кэшированный экземпляр. Например, определенный ранее общий экземпляр модуля `db` можно получить так:

```
const db = require('./db');
```

Но здесь имеется одна сложность: при кэшировании модуля в качестве ключа поиска используется его полный путь, что гарантирует его применение в качестве одиночки, только в рамках текущего пакета. Как было показано в *главе 2 «Основные шаблоны Node.js»*, каждый пакет может иметь свой набор зависимостей в своем каталоге `node_modules`, что может привести к использованию нескольких экземпляров одного и того же пакета и, следовательно, одного и того же модуля, поэтому объект-одиночка может оказаться не единственным. Рассмотрим, например, случай, когда модуль `db` помещен в пакет с `mydb`. Его файл `package.json` содержит следующие строки:

```
{
  "name": "mydb",
  "main": "db.js"
}
```

Теперь рассмотрим следующее дерево зависимостей:

```
app/
`-- node_modules
    |-- packageA
    |  `-- node_modules
    |     `-- mydb
    `-- packageB
       `-- node_modules
          `-- mydb
```

Пакеты `packageA` и `packageB` зависят от пакета `mydb`. В свою очередь, пакет `app`, представляющий основное приложение, зависит от пакетов `packageA` и `packageB`. Этот случай опровергает предположение об уникальности экземпляра базы данных. Действительно, пакеты `packageA` и `packageB` загружают экземпляр базы данных с помощью следующей команды:

```
const db = require('mydb');
```

Однако пакеты `packageA` и `packageB` фактически загружают два разных экземпляра объекта-одиночки, поскольку модуль `mydb` будет извлекаться из разных каталогов, в зависимости от того, какой из пакетов его затребовал.

На данный момент можно утверждать, что шаблона «Одиночка», описанного в литературе, для платформы Node.js не существует, если не использовать глобальных переменных, например:

```
global.db = new Database('my-app-db');
```

Это гарантирует, что экземпляр будет единственным и общим для всего приложения, а не только для одного пакета. Однако этого подхода следует избегать любой ценой, тем более что обычно идеальный объект-одиночка не нужен и существуют другие шаблоны, которые будут описаны ниже, обеспечивающие совместное использование одного и того же экземпляра разными пакетами.

В этой книге термин «одиночка» будет использоваться для обозначения любого объекта, экспортируемого модулем и обладающего состоянием, даже если он не является одиночкой в строгом смысле. Но можно уверенно утверждать, что такое определение по своим практическим целям совпадает с оригинальным шаблоном, а именно позволяет использовать одно и то же состояние в разных компонентах.

Шаблоны связывания модулей

Теперь, после знакомства с некоторыми теоретическими основами зависимостей и их внедрения, можно переходить к их практическим реализациям. В этом разделе будут рассмотрены основные шаблоны связывания модулей. Главное внимание будет уделено связыванию экземпляров с состоянием, которые, несомненно, представляют наиболее важный вид зависимостей приложения.

Жесткие зависимости

Начнем с наиболее типичного вида взаимоотношений между двумя модулями: **жесткой зависимости**. В Node.js подобная зависимость возникает при явной загрузке модуля с помощью `require()`. Как будет продемонстрировано далее в этом разделе, такой способ создания зависимостей является простым и эффективным, но при его применении следует уделять дополнительное внимание жестким зависимостям от экземпляров с состоянием, поскольку это ограничивает возможности повторного использования модулей.

Сервер аутентификации с жесткими зависимостями

Давайте начнем рассмотрение со схемы, представленной на рис. 7.1.

Рис. 7.1 ❖ Сервер аутентификации

Схема на рис. 7.1 демонстрирует типичный пример многоуровневой архитектуры; она описывает структуру простой системы аутентификации. Компонент `AuthController` принимает данные от клиента, извлекает регистрационную информацию и выполняет некоторую предварительную проверку. Затем он обращается к компоненту `AuthService` для проверки совпадения предоставленных учетных данных с хранящимися в базе данными. Это делается путем выполнения нескольких конкретных запросов с помощью дескриптора модуля `db` как средства взаимодействий с базой данных. Порядок соединения этих трех компонентов будет определять возможность их многократного использования, уровень тестируемости и обслуживаемости.

Самым естественным способом соединения этих компонентов вместе является подключение модуля `db` к модулю `AuthService` с последующим подключением модуля `AuthService` к модулю `AuthController`. А это, как упоминалось выше, образует жесткую зависимость.

Продемонстрируем это на практике, путем реализации описанной системы. Создадим простой **сервер аутентификации**, предоставляющий два следующих программных HTTP-интерфейса:

- `POST '/login'`: получает объект в формате JSON, содержащий пару имя/пароль пользователя для проверки. Возвращает **веб-маркер в формате JSON** (JSON Web Token , JWT), который может быть использован в последующих запросах для идентификации пользователя;

Веб-маркер в формате JSON является форматом для предоставления и совместного использования *утверждений* двумя сторонами. Его популярность значительно выросла по мере распространения **одностраничных приложений** и **совместного использования внешних ресурсов** (Cross-origin resource sharing, CORS) как более гибкой альтернативы схеме аутентификации на основе cookies. Более подробные сведения о JWT можно найти в его спецификации (в настоящее время существует в форме рабочего проекта) по адресу: http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html.

- `GET '/checkToken'`: извлекает маркер из параметра запроса `GET` и проверяет его допустимость.

В этом примере будет использоваться несколько технологий, часть из них уже была рассмотрена ранее. В частности, для реализации программного веб-интерфейса будет применяться `express` (`https://npmjs.org/package/express`), для хранения сведений о пользователях – `levelup` (`https://npmjs.org/package/levelup`).

Модуль db Разработка приложения будет вестись снизу вверх. Начнем с модуля, предоставляющего экземпляр базы данных `levelUp`. Создадим новый файл с именем `lib/db.js` со следующим кодом:

```
const level = require('level');
const sublevel = require('level-sublevel');

module.exports = sublevel(
  level('example-db', {valueEncoding: 'json'})
);
```

Предыдущий модуль создает соединение с базой данных `LevelDB`, хранящейся в каталоге `./example-db`, экземпляр которой декорирован с помощью дополнительного плагина `sublevel` (`https://npmjs.org/package/level-sublevel`), добавляющего под-

держку создания и выполнения запросов к отдельным разделам базы данных (их можно сравнить с SQL-таблицами или коллекциями MongoDB). Экспортируемый модулем объект является дескриптором базы данных с состоянием. То есть модуль создает объект-одиночку.

Модуль authService Теперь, имея объект-одиночку `db`, его можно использовать для реализации модуля `lib/authService.js`, который отвечает за сверку учетных данных пользователя с информацией в базе данных. Вот его код (приведены только значимые фрагменты):

```
// ...
const db = require('./db');
const users = db.sublevel('users');

const tokenSecret = 'SHHH!';

exports.login = (username, password, callback) => {
  users.get(username, function(err, user) {
    // ...
  });
};

exports.checkToken = (token, callback) => {
  // ...
  users.get(userData.username, function(err, user) {
    // ...
  });
};
```

Модуль `authService` реализует службу `login()`, отвечающую за сверку пары имя/пароль пользователя с информацией в базе данных, и службу `checkToken()`, которая принимает маркер и проверяет его допустимость.

Предыдущий код демонстрирует также первый пример жесткой зависимости с модулем, обладающим состоянием. Речь идет о модуле `db`, загружаемом с помощью простого вызова `require()`. В результате этого в переменной `db` сохраняется дескриптор инициализированной базы данных, который можно сразу же использовать для выполнения запросов.

Код модуля `authService`, что мы успели написать к данному моменту, не требует, чтобы модуль `db` присутствовал в единственном экземпляре, – он будет работать с любым экземпляром. Но он жестко зависит от одного конкретного экземпляра `db`, что означает невозможность повторного использования модуля `authService` в сочетании с другим экземпляром базы данных без изменения кода модуля.

Модуль authController Продолжая продвигаться вверх по слоям приложения, рассмотрим код модуля `lib/authController.js`. Этот модуль отвечает за обработку HTTP-запросов и, по сути, является коллекцией Express-маршрутов:

```
const authService = require('./authService');

exports.login = (req, res, next) => {
  authService.login(req.body.username, req.body.password,
    (err, result) => {
      // ...
    }
```

```
  );
};

exports.checkToken = (req, res, next) => {
  authService.checkToken(req.query.token,
    (err, result) => {
      // ...
    }
  );
};
```

Модуль `AuthController` реализует два Express-маршрута: один – для приема учетных данных и возврата маркера аутентификации (`login()`), другой – для проверки маркера (`checkToken()`). Оба маршрута делегируют большую часть логики модулю `authService`, поэтому их заботой остается только работа с HTTP-запросом и HTTP-ответом.

Как видите, в этом случае также образуется жесткая зависимость от модуля `authService`, обладающего состоянием. Если говорить точнее, модуль `authService` обладает переходным состоянием, поскольку непосредственно зависит от модуля `db`. Как видите, жесткая зависимость может пронизывать структуру всего приложения: модуль `authController` зависит от модуля `authService`, который, в свою очередь, зависит от модуля `db`. Под «переходным состоянием» подразумевается, что сам модуль `authService` косвенно привязан к одному конкретному экземпляру `db`.

Модуль app И наконец, соберем все компоненты вместе, реализовав точку входа в приложение. В соответствии с соглашениями эта логика должна быть помещена в модуль `app.js`, расположенный в корневом каталоге проекта:

```
const express = require('express');
const bodyParser = require('body-parser');
const errorHandler = require('errorhandler');
const http = require('http');

const authController = require('./lib/authController');

const app = module.exports = express();
app.use(bodyParser.json());

app.post('/login', authController.login);
app.get('/checkToken', authController.checkToken);

app.use(errorHandler());
http.createServer(app).listen(3000, () => {
  console.log('Express server started');
});
```

Как видите, модуль `app` очень прост. Он содержит простой Express-сервер, регистрирующий некоторое промежуточное программное обеспечение и два маршрута, экспортируемых модулем `authController`. Конечно, наиболее важной в данном случае является инструкция, подключающая модуль `authController` и создающая жесткую зависимость от экземпляра с состоянием.

Запуск сервера аутентификации Перед проверкой только что созданного сервера аутентификации следует заполнить базу данных некоторыми образцами данных с помощью сценария `populate_db.js`, который входит в загружаемые примеры для книги. После этого можно запустить сервер, выполнив следующую команду:

```
node app
```

Затем можно попытаться вызвать две созданные веб-службы с помощью REST-клиента или старой доброй команды `curl`. Например, выполнить вход можно следующей командой:

```
curl -X POST -d '{"username": "alice", "password":"secret"}'
http://localhost:3000/login -H "Content-Type: application/json"
```

Предыдущая команда должна вернуть маркер, который можно использовать для тестирования веб-службы `/checkLogin` (просто подставьте его вместо `<TOKEN HERE>` в следующей команде):

```
curl -X GET -H "Accept: application/json"
http://localhost:3000/checkToken?token=<TOKEN HERE>
```

Эта команда должна вернуть следующую строку, что подтвердит надлежащую работу сервера:

```
{"ok":"true","user":{"username":"alice"}}
```

Плюсы и минусы жестких зависимостей

Только что реализованный пример демонстрирует обычный способ связывания модулей в Node.js, использующий возможности системы модулей для управления зависимостями между различными компонентами приложения. Экспорт экземпляров с состоянием из модулей позволяет платформе Node.js управлять циклом существования этих экземпляров и непосредственно подключать их в других частях приложения. Результатом является наглядная организация модулей, в которой легко разобраться и которую несложно отлаживать, где каждый модуль инициализируется и подключается без необходимости внешнего вмешательства.

Но, с другой стороны, жесткая зависимость от экземпляра с состоянием ограничивает возможность связывания модуля с другими экземплярами, что затрудняет его повторное использование и модульное тестирование. Например, повторное использование модуля `authService` в комбинации с экземплярами других баз данных практически невозможно, поскольку он жестко привязан к одному конкретному экземпляру. Аналогично изолированное тестирование модуля `authService` может быть затруднено, поскольку достаточно сложно создать фиктивный экземпляр базы данных, используемый модулем.

С учетом изложенного выше становится очевидным, что большая часть недостатков использования жестких зависимостей связана с состоянием экземпляров. Это означает, что при использовании функции `require()` для подключения модулей, не имеющих состояния, например фабрик, конструкторов или набора функций без состояния, таких проблем не возникает. Хотя по-прежнему используется жесткая привязка к конкретной реализации, но на платформе Node.js это обычно не влияет на возможность повторного использования компонента, поскольку не требует привязки к конкретному состоянию.

Внедрение зависимостей

Шаблон **«Внедрение зависимостей»** (Dependency Injection , DI), вероятно, является одной из наиболее неправильно понимаемых идей разработки программного обеспечения. Многие связывают этот термин с фреймворками и DI-контейнерами, такими как

Spring (для Java и C#) или Pimple (для PHP), но в действительности все гораздо проще. Основная идея шаблона внедрения зависимостей заключается в предоставлении компонента зависимости в качестве *входных данных* с помощью внешней сущности.

Такой сущностью может быть клиентский компонент или глобальный контейнер, централизованно связывающий все модули системы. Основным преимуществом такого подхода является устранение тесных связей, особенно между модулями и экземплярами с состоянием. При использовании DI все зависимости не создаются внутри модуля, а передаются извне. Это означает, что модуль можно настроить на использование любых зависимостей и, следовательно, повторно использовать в различных контекстах.

Для демонстрации этого шаблона реорганизуем созданный ранее сервер проверки подлинности, используя DI для соединения его модулей.

Реорганизация сервера аутентификации для использования DI

Реорганизация модулей для использования DI заключается в следовании простому рецепту: заменить жесткие зависимости от экземпляров с состоянием фабрикой, получающей набор зависимостей в качестве аргументов.

Начнем с модуля `lib/db.js` и изменим его код следующим образом:

```
const level = require('level');
const sublevel = require('level-sublevel');

module.exports = dbName => {
 return sublevel(
   level(dbName, {valueEncoding: 'json'})
  );
};
```

Первым шагом стало преобразование модуля `db` в фабрику. В результате появилась возможность использовать эту фабрику для создания нужного количества экземпляров базы данных, а это означает, что сам модуль теперь можно использовать повторно и он не поддерживает состояния.

Далее реализуем новую версию модуля `lib/authService.js`:

```
const jwt = require('jwt-simple');
const bcrypt = require('bcrypt');

module.exports = (db, tokenSecret) => {
  const users = db.sublevel('users');
  const authService = {};

  authService.login = (username, password, callback) => {
    //...то же, что в предыдущей версии
  };

  authService.checkToken = (token, callback) => {
    //...то же, что в предыдущей версии
  };

  return authService;
};
```

Теперь и модуль `authService` лишился состояния, он больше не экспортирует конкретных экземпляров, а создает их через фабрику. Наиболее важная деталь – зависимость `db` была сделана *внедряемой*, путем преобразования ее в аргумент функции

фабрики и удаления жесткой зависимости. Это простое изменение позволило создать новый модуль `authService`, способный соединяться с экземпляром любой базы данных.

Аналогично можно реорганизовать модуль `lib/authController.js`:

```
module.exports = (authService) => {
  const authController = {};

  authController.login = (req, res, next) => {
    //...то же, что в предыдущей версии
  };

  authController.checkToken = (req, res, next) => {
    //...то же, что в предыдущей версии
  };

  return authController;
};
```

Модуль `AuthController` не имеет жестких зависимостей, даже не имеющих состояния! Единственная его зависимость – модуль `authService` – передается через вызов фабрики.

А теперь посмотрим, где фактически все эти модули создаются и связываются вместе. Это происходит в модуле `app.js`, который является верхним слоем приложения, как показано ниже:

```
// ...
const dbFactory = require('./lib/db');                          //[1]
const authServiceFactory = require('./lib/authService');
const authControllerFactory = require('./lib/authController');

const db = dbFactory('example-db');                             //[2]
const authService = authServiceFactory(db, 'SHHH!');
const authController = authControllerFactory(authService);

app.post('/login', authController.login);                       //[3]
app.get('/checkToken', authController.checkToken);
// ...
```

Ниже приведены замечания к предыдущему коду.

1. Сначала загружаются фабрики служб; на данный момент они являются объектами без состояния.
2. Затем создается по экземпляру каждой из служб с передачей необходимых зависимостей. Это тот этап, где создаются и соединяются вместе все модули.
3. И наконец, как обычно, с помощью Express-сервера регистрируются маршруты для модуля `authController`.

Теперь все компоненты сервера аутентификации связаны с использованием механизма DI и готовы к использованию.

Различные типы DI

Только что приведенный пример демонстрирует лишь один тип DI (**внедрение с помощью фабрик**), но существует несколько других типов, достойных упоминания:

- **Внедрение через параметры конструктора**: зависимости передаются в конструктор в момент создания объекта, например:

  ```
  const service = new Service(dependencyA, dependencyB);
  ```

- **Внедрение через свойства**: зависимости *внедряются* в объект после его создания, например:

```
const service = new Service(); //работает как фабрика
service.dependencyA = anInstanceOfDependencyA;
```

Внедрение через свойства предполагает создание объекта в *несогласованном* состоянии, поскольку в этот момент он еще не связан с зависимостями, что делает данный способ наименее надежным, но иногда он очень полезен при наличии циклических зависимостей. Например, если имеются два компонента *A* и *B*, использующих внедрение через фабрики или конструкторы и зависящих друг друга, их экземпляры не могут быть созданы, поскольку каждый требует существования другого до своего создания. Рассмотрим следующий простой пример:

```
function Afactory(b) {
  return {
    foo: function() {
      b.say();
    },
    what: function() {
      return 'Hello!';
    }
  }
}

function Bfactory(a) {
  return {
    a: a,
    say: function() {
      console.log('I say: ' + a.what);
    }
  }
}
```

Взаимоблокировку зависимостей двух приведенных выше фабрик можно разрешить только с помощью внедрения свойств, например путем создания неполного экземпляра *B*, который затем можно использовать для создания *A*, с последующим внедрением *A* в соответствующее свойство экземпляра *B*:

```
const b = Bfactory(null);
const a = Afactory(b);
a.b = b;
```

В некоторых редких случаях непросто избежать образования циклических зависимостей, но важно иметь в виду, что обычно циклические зависимости являются симптомом плохого проектирования.

Плюсы и минусы DI

В примере реализации сервера аутентификации с применением DI удалось избежать зависимости модулей от конкретных экземпляров. В результате появилась возможность повторно использовать каждый из модулей без особых усилий и без внесения изменений в их код. Использование шаблона DI также значительно облегчает модульное тестирование, поскольку упрощает создание фиктивных зависимостей и тестирование каждого из модулей независимо от состояния остальной системы.

Еще один важный аспект этого примера заключается в *перекладывании* ответственности за внедрение зависимостей на компоненты верхнего уровня архитектуры. Идея состоит в том, что компоненты высокого уровня по своей природе менее многоразовые, чем компоненты низкого уровня, поскольку по мере увеличения уровня компонентов приложения они становится все более конкретными.

Исходя из этого предположения, можно понять, что при обычном подходе к проектированию архитектуры приложения компоненты высокого уровня отвечают за зависимости компонентов более низкого уровня, которые могут быть инвертированы, поскольку компоненты нижнего уровня зависят только от интерфейсов (в JavaScript это просто интерфейсы, ожидаемые от зависимостей), тогда как ответственность за определение реализаций зависимостей ложится на компоненты более высокого уровня. Действительно, в сервере аутентификации все зависимости создаются и связываются в компоненте верхнего уровня, то есть модуле `app`, который является менее многоразовым и поэтому более всего подходит для организации связывания.

Но все эти преимущества в терминах слабой связанности и повторного использования имеют свою цену. В целом невозможность разрешения зависимостей *в момент написания кода* затрудняет понимание взаимосвязей между различными компонентами системы. Кроме того, исследовав внедрение зависимостей в модуле `app`, можно заметить, что при этом должен соблюдаться определенный порядок. Это влечет необходимость составления вручную графа зависимостей всего приложения. При значительном увеличении количества связываемых модулей этот процесс может стать неуправляемым.

Реальным решением данной проблемы является разделение ответственности за внедрение зависимостей между несколькими компонентами вместо концентрации в одном месте. Это позволит уменьшить экспоненциальный рост сложности управления зависимостями, поскольку каждый компонент будет отвечать только за зависимости своего подграфа. Конечно, при необходимости можно использовать DI локально, а не передавать все внедрение зависимостей компоненту верхнего уровня приложения.

Далее в этой главе будет продемонстрировано другое возможное решение проблемы, упрощающее связывание модулей в сложных архитектурах: использование контейнера DI, то есть компонента, отвечающего исключительно за создание экземпляров и внедрение всех зависимостей приложения.

Очевидно, что использование DI увеличивает сложность и размер модулей, но, как было продемонстрировано выше, на это есть ряд веских причин. Мы должны выбирать правильный подход в зависимости от нужного баланса между простотой и возможностью повторного использования.

DI часто упоминается в сочетании с *принципом инверсии зависимостей* (Dependency Inversion) и *инверсии управления* (Inversion of Control), однако это совершенно иные концепции (несмотря на то что между ними существует определенная корреляция).

Локатор служб

В предыдущем разделе было продемонстрировано, как использование DI способно изменить подход к связыванию зависимостей и обеспечить получение слабо связанных модулей, поддерживающих повторное использование. Другим шаблоном, пред-

назначенным для достижения той же цели, является шаблон «Локатор служб». Его основная идея заключается в наличии центрального реестра для управления компонентами системы, который выступает в качестве посредника при загрузке зависимости любым модулем. Суть в том, что вместо жесткого связывания зависимости запрашиваются у локатора служб.

Важно понимать, что при использовании локатора служб возникает зависимость от него самого, что определяет уровень сцепления модулей и, следовательно, возможность их повторного использования. В Node.js можно определять три вида локаторов служб, в зависимости от порядка подключения к компонентам системы:

- жесткая зависимость от локатора служб;
- внедрение локатора служб;
- глобальный локатор служб.

Первый вид, естественно, предлагает минимум преимуществ с точки зрения ослабления связей, поскольку подразумевает создание непосредственной ссылки на экземпляр локатора служб с помощью `require()`. В Node.js это считается антишаблоном, так как приводит к образованию тесной связи с компонентом, предназначенным для ослабления связей. В этом контексте использование локатора служб не имеет особого смысла, с точки зрения повторного использования, поскольку лишь добавляет еще один уровень косвенности и сложности.

Иное дело – *внедрение локатора служб* с использованием механизма DI. Его можно рассматривать как удобный способ внедрения всего набора зависимостей сразу. И, как будет показано ниже, этим его преимущества не ограничиваются.

Третий способ – ссылка на локатор служб в глобальной области видимости. Он имеет те же недостатки, что и жесткое связывание локатора служб, но, поскольку в этом случае локатор носит глобальный характер, он является *настоящим объектом-одиночкой* и, следовательно, может совместно использоваться разными пакетами как общий экземпляр. Мы рассмотрим этот метод далее, но отметим, что в действительности не так много причин для использования глобального локатора служб.

Система модулей в Node.js реализует один из вариантов шаблона локатора служб в виде функции `require()`, представляющей глобальный экземпляр локатора служб.

Все приведенные здесь соображения мы поясним на примере использования шаблона «Локатор служб». Реорганизуем код сервера аутентификации еще раз, применив новые знания.

Реорганизация кода сервера аутентификации для использования локатора служб

Преобразуем сервер аутентификации для использования внедряемого локатора служб. Сначала реализуем сам локатор служб в новом модуле `lib/serviceLocator.js`:

```
module.exports = function() {
  const dependencies = {};
  const factories = {};
  const serviceLocator = {};

  serviceLocator.factory = (name, factory) => {                //[1]
    factories[name] = factory;
  };
```

```
  serviceLocator.register = (name, instance) => {            //[2]
    dependencies[name] = instance;
  };

  serviceLocator.get = (name) => {                           //[3]
    if(!dependencies[name]) {
      const factory = factories[name];
      dependencies[name] = factory && factory(serviceLocator);
      if(!dependencies[name]) {
        throw new Error('Cannot find module: ' + name);
      }
    }
    return dependencies[name];
  };

  return serviceLocator;
};
```

Модуль `serviceLocator` является фабрикой, возвращающей объект с тремя методами:

- метод `factory()` используется для связывания имени компонента с фабрикой;
- метод `register()` используется для связывания имени компонента непосредственно с экземпляром;
- метод `get()` извлекает компонент по его имени. Если экземпляр уже доступен, метод просто возвращает его, в противном случае он пытается вызвать зарегистрированную фабрику для получения нового экземпляра. Важно отметить, что фабрики модулей вызываются путем внедрения текущего экземпляра локатора служб (`serviceLocator`). Это основной механизм шаблона, который обеспечивает построение графа зависимостей автоматически или по требованию.

Простой шаблон, схожий с шаблоном «Локатор служб», заключается в использовании объекта в качестве пространства имен для набора зависимостей:

```
const dependencies = {};
const db = require('./lib/db');
const authService = require('./lib/authService');
dependencies.db = db();
dependencies.authService = authService(dependencies);
```

Теперь преобразуем модуль `lib/db.js`, чтобы показать, как работает модуль `serviceLocator`:

```
const level = require('level');
const sublevel = require('level-sublevel');

module.exports = (serviceLocator) => {
  const dbName = serviceLocator.get('dbName');

  return sublevel(
    level(dbName, {valueEncoding: 'json'})
  );
}
```

Модуль `db` использует полученный локатор служб для извлечения имени базы данных при создании экземпляра. Это интересное место выделено в коде. Локатор

служб может использоваться для извлечения не только экземпляров компонентов, но и параметров конфигурации, определяющих поведение всего создаваемого графа зависимостей.

Следующий шаг – преобразование модуля `lib/authService.js`:

```
// ...
module.exports = (serviceLocator) => {
  const db = serviceLocator.get('db');
  const tokenSecret = serviceLocator.get('tokenSecret');

  const users = db.sublevel('users');
  const authService = {};

  authService.login = (username, password, callback) => {
    //...то же, что в предыдущей версии
  }

  authService.checkToken = (token, callback) => {
    //...то же, что в предыдущей версии
  }

  return authService;
};
```

Модуль `authService` также является фабрикой, принимающей локатор служб в качестве аргумента. Две зависимости модуля, дескриптор `db` и `tokenSecret` (который является другим конфигурационным параметром), извлекаются с помощью метода `get()` локатора служб.

Аналогично можно преобразовать модуль `lib/authController.js`:

```
module.exports = (serviceLocator) => {
  const authService = serviceLocator.get('authService');
  const authController = {};

  authController.login = (req, res, next) => {
    //...то же, что в предыдущей версии
  };

  authController.checkToken = (req, res, next) => {
    //...то же, что в предыдущей версии
  };

  return authController;
}
```

Теперь можно переходить к созданию и настройке локатора служб. Конечно же, это происходит в модуле `app.js`:

```
//...
const svcLoc = require('./lib/serviceLocator')();            //[1]

svcLoc.register('dbName', 'example-db');                     //[2]
svcLoc.register('tokenSecret', 'SHHH!');
svcLoc.factory('db', require('./lib/db'));
svcLoc.factory('authService', require('./lib/authService'));
svcLoc.factory('authController', require('./lib/authController'));

const authController = svcLoc.get('authController');         //[3]
```

```
app.post('/login', authController.login);
app.all('/checkToken', authController.checkToken);
// ...
```

Вот как производится связывание с использованием только что созданного локатора служб:

1) вызовом фабрики создается новый локатор служб;
2) конфигурационные параметры и фабрики модулей регистрируются в локаторе служб. На данный момент экземпляры зависимостей еще не созданы, регистрируются только их фабрики;
3) из локатора служб загружается модуль `authController`. Это точка входа для создания всего графа зависимостей приложения. Когда мы пытаемся получить экземпляр компонента `authController`, локатор служб вызывает связанную фабрику, внедряя самого себя, затем фабрика модуля `authController` пытается загрузить модуль `authService`, который, в свою очередь, создаст модуль `db`.

Интересно отметить *«ленивый»* характер локатора служб, так как каждый экземпляр создается только по мере необходимости. Но здесь имеется еще один важный аспект: как видите, все зависимости внедряются автоматически, без необходимости предварительно создавать их вручную. Преимущество такого решения – в том, что не требуется заранее знать правильный порядок создания экземпляров и связывания модулей, поскольку все происходит автоматически и *по требованию*. Это гораздо удобнее, по сравнению с обычным шаблоном DI.

Другим распространенным шаблоном является использование экземпляра Express-сервера в качестве простого локатора служб. Для этого достаточно использовать метод `expressApp.set(name, instance)` для регистрации службы и метод `expressApp.get(name)` для последующего ее извлечения. Это довольно удобно, потому что экземпляр сервера, выступающий в качестве локатора служб, уже внедрен в любое промежуточное программное обеспечение и доступен через свойство `request.app`. Пример реализации этого шаблона можно найти в загружаемых примерах к книге.

Плюсы и минусы локатора служб

Шаблоны **«Локатор служб»** и **«Внедрение зависимостей»** имеют много общего: оба передают владение зависимостями внешним, по отношению к компоненту, сущностям. Однако подход к подключению локатора служб влияет на гибкость всей архитектуры. Не случайно для реализации примера был выбран прием внедрения, а не жесткая привязка или глобальный локатор служб. Применение двух последних вариантов практически сводит на нет все преимущества этого шаблона. Фактически в результате их использования мы просто заменим непосредственное связывание зависимостей с помощью `require()` привязкой к одному конкретному экземпляру локатора служб. Кроме того, несмотря на то что жестко привязанный локатор служб все же обеспечит большую гибкость конфигурирования компонента, давая возможность привязывать зависимости по их именам, это не даст никаких заметных преимуществ с точки зрения повторного использования.

Подобно шаблону внедрения зависимостей, локатор служб затрудняет определение взаимосвязей между компонентами, поскольку они разрешаются во время выполнения. Вдобавок локатор служб усложняет понимание, какую именно зависимость конкретный компонент собирается затребовать. При применении DI это выражается гораздо более явным образом, определением зависимостей в аргументах фабрики или

конструктора. Локатор служб делает это гораздо менее очевидным способом, что требует дополнительной проверки кода или прямого объявления в документации, какие зависимости попытается загрузить компонент.

И наконец, следует учесть, что часто локатор служб ошибочно принимают за DI-контейнер, поскольку они выполняют одну и ту же роль реестра служб, но делают это по-разному. При применении локатора службы все компоненты явно загружают свои зависимости из локатора служб. При использовании DI-контейнера компонентам ничего неизвестно о нем.

Разница между этими двумя подходами определяется следующими признаками:

- **повторное использование**: компоненты, полагающиеся на локатор служб, хуже подходят для повторного использования, поскольку требуют доступности в системе локатора служб;
- **удобочитаемость**: как уже упоминалось, локатор служб усложняет определение зависимостей, затребованных компонентом.

С точки зрения повторного использования шаблон «Локатор служб» находится между шаблоном жестких зависимостей и DI. С точки зрения удобства и простоты он определенно превосходит шаблон DI, требующий ручной настройки, поскольку при его применении не придется заботиться о построении полного графа зависимостей вручную.

В соответствии с этими утверждениями шаблон DI определенно обеспечивает наилучший компромисс с точки зрения повторного использования компонентов и удобства применения. Более подробно этот шаблон рассматривается в следующем разделе.

Контейнер внедрения зависимостей

Преобразование локатора служб в контейнер внедрения зависимостей (DI) не является большой проблемой, но, как уже поминалось, оно обеспечивает огромные преимущества с точки зрения ослабления зависимостей. Действительно, при использовании этого шаблона модули не зависят от локатора служб, их зависимости проще определяются, а DI-контейнер легко справляется со всеми остальными проблемами. Как будет показано ниже, большим преимуществом этого механизма является возможность повторного использования любого модуля даже без контейнера.

Объявление зависимостей в DI-контейнере

DI-контейнер, по сути, представляет собой локатор служб с одной дополнительной функцией, определяющей зависимости модуля до создания его экземпляра. Для этого модуль должен объявить свои зависимости одним из способов, описываемых ниже.

Первый и самый популярный способ – внедрить набор зависимостей, опираясь на имена аргументов фабрики или конструктора. Рассмотрим этот прием на примере модуля `authService`:

```
module.exports = (db, tokenSecret) => {
  //...
}
```

Согласно такому определению, DI-контейнер создаст экземпляр модуля, используя зависимости с именами `db` и `tokenSecret`, – все просто и понятно. Однако, чтобы получить возможность прочитать имена аргументов функции, необходимо использовать небольшой трюк. JavaScript позволяет сериализовать функцию и получить ее исходный код. Для этого достаточно вызвать метод `toString()` ссылки на функцию.

После этого не составит большого труда получить список аргументов с помощью регулярных выражений.

Такой прием внедрения зависимостей, основанный на использовании имен аргументов функции, применяется в популярном клиентском JavaScript-фреймворке AngularJS (http://angularjs.org), разработанном компанией Google, целиком построенном на DI-контейнерах.

Самой большой проблемой этого подхода является несовместимость с **минификацией** (minification), которая широко используется для клиентского JavaScript-кода и заключается в применении преобразований для уменьшения размеров исходного кода до минимума. Многие инструменты минификации применяют метод **изменения имен**, который заключается в замене имен всех локальных переменных более короткими именами, обычно односимвольными. Беда в том, что аргументы функции считаются локальными переменными и, соответственно, подвергаются этому процессу, что делает невозможным использование приема, описанного выше. Несмотря на то что на стороне сервера минификация не играет никакой роли, следует учесть, что модули Node.js часто используются совместно с браузером, а это в данном случае является важным фактором.

К счастью, DI-контейнер может использовать другие методы, чтобы узнать о внедряемых зависимостях. Эти методы перечислены ниже.

- В фабричную функцию можно добавить специальное свойство, например массив, явно перечисляющий все зависимости для внедрения:

```
module.exports = (a, b) => {};
module.exports._inject = ['db', 'another/dependency'];
```

- Модуль можно описать как массив имен зависимостей, за которыми следует фабричная функция:

```
module.exports = ['db', 'another/depencency',(a, b) => {}];
```

- Можно использовать аннотацию в виде комментария, добавляемую к каждому аргументу функции (однако этот способ также не совместим с минификацией):

```
module.exports = function(a /*db*/, b /*another/depencency*/) {};
```

Все эти приемы страдают разными недостатками, поэтому в нашем примере мы используем более простой и популярный способ определения зависимостей по именам аргументов функции.

Реорганизация кода сервера аутентификации для использования DI-контейнера

Чтобы показать, что DI-контейнер гораздо менее агрессивен, чем локатор служб, вновь реорганизуем код сервера аутентификации, использовав в качестве отправной точки версию с обычным шаблоном DI. Мы оставим нетронутыми все компоненты приложения, за исключением модуля `app.js`, который будет отвечать за инициализацию контейнера.

Но сначала нужно реализовать сам DI-контейнер. Создадим для этого новый модуль `diContainer.js` в каталоге `lib/`. Ниже приводится первая его часть:

```
const fnArgs= require('parse-fn-args');

module.exports = function() {
  const dependencies = {};
  const factories = {};
```

```
  const diContainer = {};

  diContainer.factory = (name, factory) => {
    factories[name] = factory;
  };

  diContainer.register = (name, dep) => {
    dependencies[name] = dep;
  };

  diContainer.get = (name) => {
    if(!dependencies[name]) {
      const factory = factories[name];
      dependencies[name] = factory &&
          diContainer.inject(factory);
      if(!dependencies[name]) {
        throw new Error('Cannot find module: ' + name);
      }
    }
    return dependencies[name];
  };
  //...продолжение
```

Начало модуля `diContainer` функционально идентично приведенному выше локатору служб. Единственные заметные отличия:

- загружается новый npm-модуль `args-list` (https://npmjs.org/package/args-list), который будет использоваться для извлечения имен аргументов функции;
- вместо фабрики модуля вызывается метод `inject()` модуля `diContainer`, определяющий зависимости модуля и использующий их для вызова фабрики.

Рассмотрим метод `diContainer.inject()`:

```
diContainer.inject = (factory) => {
  const args = fnArgs(factory)
  .map(dependency => diContainer.get(dependency));
    return factory.apply(null, args);
  };

}; //конец module.exports = function() {
```

Этот метод отличает DI-контейнер от локатора служб. Логика его работы достаточно проста:

1) с помощью библиотеки `parse-fn-args` извлекается список аргументов указанной фабричной функции;
2) затем имя каждого аргумента отображается в экземпляр зависимости, полученный с помощью метода `get()`;
3) и в завершение полученный список зависимостей просто передается в вызов фабрики.

В этом заключается работа модуля `diContainer`. Как видите, его действия несильно отличаются от действий локатора служб, но простое создание экземпляра модуля с попутным внедрением его зависимостей определяет огромную разницу между ними (в сравнении с внедрением всего локатора служб).

Чтобы завершить реорганизацию, необходимо настроить модуль `app.js`:

```
// ...
const diContainer = require('./lib/diContainer')();
```

```
diContainer.register('dbName', 'example-db');
diContainer.register('tokenSecret', 'SHHH!');
diContainer.factory('db', require('./lib/db'));
diContainer.factory('authService', require('./lib/authService'));
diContainer.factory('authController', require('./lib/authController'));

const authController = diContainer.get('authController');

app.post('/login', authController.login);
app.get('/checkToken', authController.checkToken);
// ...
```

Как видите, код модуля `app` идентичен тому, что использовался для инициализации локатора служб в предыдущем разделе. Обратите также внимание, что начальная загрузка контейнера DI влечет за собой загрузку всего графа зависимостей, который по-прежнему используется подобно локатору служб путем вызова `diContainer.get('authController')`. Теперь все модули, зарегистрированные с помощью DI-контейнера, будут создаваться и подключаться автоматически.

Плюсы и минусы DI-контейнеров

Контейнер DI предполагает использование шаблона DI и, следовательно, наследует большинство его плюсов и минусов. В частности, мы ослабили связи между модулями и упростили тестирование, но, с другой стороны, увеличили сложность реализации, добавив разрешение зависимостей во время выполнения. DI-контейнер имеет много общего с локатором служб, но его преимуществом является независимость модулей от любых дополнительных служб, кроме их собственных зависимостей. Это огромный плюс, поскольку позволяет использовать модули даже без DI-контейнера, внедряя их вручную.

По сути, в этом разделе мы взяли за основу версию сервера аутентификации, реализованную с применением обычного шаблона DI, и без изменения его компонентов (кроме модуля `app`) автоматизировали внедрение всех зависимостей.

В npm можно найти множество готовых DI-контейнеров или подсмотреть идеи, лежащие в их основе: https://www.npmjs.org/search?q=dependency%20injection.

Связывание плагинов

Архитектурной мечтой любого инженера является небольшое ядро, расширяемое при необходимости за счет использования **плагинов**. К сожалению, этого не всегда легко добиться, поскольку это требует затрат времени, ресурсов и усложняет приложение. Тем не менее всегда желательно поддерживать хоть какую-то *внешнюю расширяемость*, даже если она ограничивается лишь некоторыми частями системы. В данном разделе мы познакомимся с этим увлекательным миром и уделим особое внимание двум проблемам:

- доступности служб приложения для плагинов;
- интеграции плагинов в поток выполнения родительского приложения.

Плагины как пакеты

Часто в Node.js плагины приложения устанавливаются как пакеты в каталог `node_modules` проекта. Этот способ имеет два преимущества. Во-первых, появляется воз-

можность использовать npm для распространения плагинов и управления их зависимостями. И во-вторых, пакет может иметь собственный граф зависимостей, что снижает вероятность конфликтов и несовместимостей между зависимостями, поскольку плагин не использует зависимости родительского проекта.

Вот как выглядит структура каталогов примера приложения с двумя плагинами-пакетами:

```
application
'-- node_modules
    |-- pluginA
    '-- pluginB
```

В мире Node.js это очень распространенная практика. Популярными примерами могут служить: `express` (http://expressjs.com) со своим промежуточным программным обеспечением, `gulp` (http://gulpjs.com), `grunt` (http://gruntjs.com), `nodebb` (http://nodebb.org) и `docpad` (http://docpad.org).

Но преимущества использования пакетов не ограничиваются *внешними плагинами*. Фактически заключение компонентов приложения в пакеты, как если бы они были *внутренними плагинами*, является одним из самых популярных шаблонов. То есть вместо размещения всех модулей в главном пакете приложения можно создать отдельный пакет для каждой достаточно большой функциональной части и поместить их в каталог `node_modules`.

Пакет может быть закрытым и недоступным в общем реестре npm. Всегда можно установить флаг `private` в файле `package.json` и тем самым предотвратить случайную его публикацию в npm. Затем можно передать пакеты в систему управления версиями, например git, или использовать частный npm-сервер для их передачи всем членам команды.

Что дает применение этого шаблона? Прежде всего удобство: многие считают неудобными или громоздкими ссылки на локальные модули пакета в виде относительных путей. Например, рассмотрим следующую структуру каталогов:

```
application
|-- componentA
|   '-- subdir
|       '-- moduleA
'-- componentB
    '-- moduleB
```

Для ссылки на модуль `moduleB` из модуля `moduleA` придется использовать такую инструкцию:

```
require('../../componentB/moduleB');
```

Вместо этого можно воспользоваться свойствами алгоритма разрешения `require()` (как это было описано в *главе 2 «Основные шаблоны Node.js»*) и поместить весь каталог `componentB` в пакет. После установки этого пакета в каталог `node_modules` его можно подключать, используя следующий синтаксис (в любом месте основного пакета приложения):

```
require('componentB/module');
```

Второй причиной деления проекта на пакеты, конечно же, является возможность повторного использования. Пакет может иметь собственные зависимости и вынуждает разработчиков думать, какие особенности должны быть доступны основному приложению, а какие, напротив, быть скрытыми, что благотворно сказывается на отделении и сокрытии деталей во всем приложении.

Шаблон
Пакеты можно использовать как средство организации приложения, а не только для распространения кода через npm.

В примерах, описанных выше, пакеты используются не столько в роли библиотек, не имеющих состояния (подобно большинству пакетов в npm), сколько в качестве неотъемлемых частей конкретного приложения, расширяющих его функциональные возможности или изменяющих модель поведения. Основное различие заключается в том, что пакеты этого типа интегрированы в приложение, а не просто используются им.

Для большей ясности мы будем использовать термин «плагин» для обозначения всех пакетов, включая пакеты, предназначенные для интеграции с конкретным приложением.

Как будет показано ниже, общая проблема, с которой приходится сталкиваться при выборе архитектуры такого типа, заключается в необходимости предоставления плагинам доступа к частям основного приложения. В самом деле, нельзя ограничиться только плагинами без состояния, хотя это и является идеальным с точки зрения расширяемости, поскольку иногда плагинам необходимо использовать некоторые службы родительского приложения для решения своих задач. От этого аспекта во многом зависит технология подключения модулей к родительскому приложению.

Точки расширения

Существует практически бесконечное количество способов сделать приложение расширяемым. Например, именно для этого предназначены некоторые из шаблонов проектирования, рассмотренные в *главе 6 «Шаблоны проектирования»*; используя шаблон «Прокси» или «Декоратор», можно изменить или расширить функциональные возможности служб; с помощью шаблона «Стратегия» можно подменять части алгоритмов; применив шаблон «Промежуточное программное обеспечение», можно вставлять блоки обработки в имеющиеся конвейеры. Потоки данных также способны обеспечить расширяемость благодаря их композиционной природе.

С другой стороны, **генераторы событий** позволяют ослабить связь между компонентами за счет применения шаблона публикации/подписки. Еще один важный способ – явное определение точек в приложении для подключения новых или изменения существующих функциональных возможностей. Эти точки обычно называют ловушками. Таким образом, наиболее важным средством поддержки плагинов является набор точек расширения.

Способ связывания компонентов в приложении также играет решающую роль, поскольку влияет на способ предоставления плагинам доступа к службам приложения. Данный раздел главным образом будет посвящен этому аспекту.

Расширение, управляемое плагинами и приложением

Прежде чем двинуться дальше и перейти к примерам, важно разобраться в основах методики. Существуют два подхода к расширению компонентов приложения:

- явное расширение;
- расширение путем инверсии управления (Inversion of Control, IoC).

В первом случае имеется конкретный компонент (реализующий новые функциональные возможности), явным образом расширяющий инфраструктуру, а во втором – инфраструктура сама управляет расширением, загружая, устанавливая или выполняя новый компонент. Во втором сценарии поток управления инвертируется, как показано на рис. 7.2.

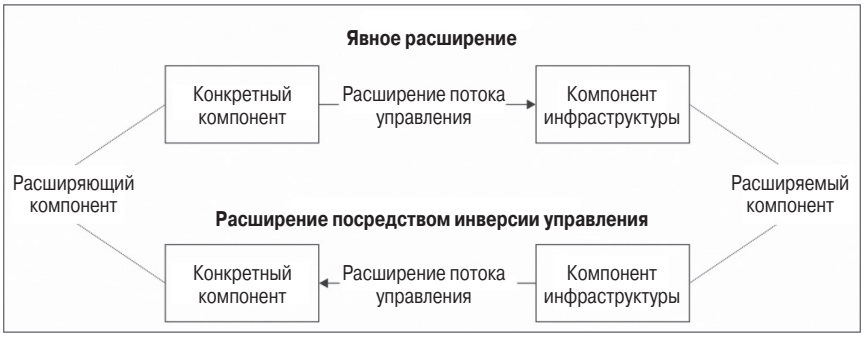

Рис. 7.2 ❖ Два подхода к расширению компонентов приложений

Инверсия управления является весьма широким принципом, который может применяться не только к проблеме расширения возможностей приложений. Можно даже сказать, что, реализуя IoC в той или иной форме, вместо кода, управляющего инфраструктурой, инфраструктура управляет пользовательским кодом. Реализуя принцип IoC, различные компоненты приложения меняют широту своих возможностей в управлении потоком выполнения на ослабление связанности. Это похоже на принцип Голливуда: «*Не звоните нам, мы сами свяжемся с вами*».

Например, DI-контейнер демонстрирует применение принципа IoC к конкретной ситуации управления зависимостями. Шаблон «Наблюдатель» – еще один пример применения IoC для управления состоянием. Шаблоны «Макет», «Стратегия», «Состояние» и «Промежуточное программное обеспечение» являются локализованными проявлениями того же принципа. Браузер реализует принцип инверсии управления для передачи событий пользовательского интерфейса в код на JavaScript (то есть коду на JavaScript не приходится активно опрашивать браузер, проверяя появление новых событий), и сама платформа Node.js руководствуется принципом инверсии управления для обратных вызовов.

Более подробные сведения о принципе IoC можно найти на странице его основателя – Мартина Фаулера (Martin Fowler) http://martinfowler.com/bliki/InversionOfControl.html.

Распространив эту идею на плагины, можно определить две формы расширения:

- расширение, управляемое плагином;
- расширение, управляемое приложением (IoC).

В первом случае плагин сам вторгается в компоненты приложения для их расширения, а во втором управление находится в руках приложения, которое подключает плагин к одной из своих точек расширения.

В качестве примера рассмотрим плагин, добавляющий в Express-приложение новый маршрут. Вот как будет выглядеть реализация расширения, управляемого плагином:

```
//в приложении:
const app = express();
require('thePlugin')(app);

//в плагине:
module.exports = function plugin(app) {
  app.get('/newRoute', function(req, res) {...})
};
```

Если реализовать способ расширения, управляемого приложением (инверсия управления), предыдущий пример будет выглядеть так:

```
//в приложении:
const app = express();
const plugin = require('thePlugin')();
app[plugin.method](plugin.route, plugin.handler);

//в дополнительном встраиваемом модуле:
module.exports = function plugin() {
  return {
    method: 'get',
    route: '/newRoute',
    handler: function(req, res) {...}
  }
}
```

Как видите, в последнем случае плагин играет пассивную роль в процессе расширения, поскольку управление находится в руках приложения, которое реализует инфраструктуру для подключения плагина.

Основываясь на предыдущем примере, можно сразу же перечислить важные отличия этих двух подходов:

- **расширение, управляемое плагином,** является более мощным и гибким, так как часто открывает доступ к внутренним компонентам приложения и дает больше свободы, как если бы плагин был частью самого приложения. Но иногда это может приводить к увеличению ответственности, не давая никаких преимуществ. В самом деле, любое изменение в приложении может влиять на плагин, что требует постоянного его обновления по мере развития основного приложения;
- расширение, управляемое приложением, требует организации **инфраструктуры управления плагинами** в самом приложении. Когда расширение управляется плагинами, единственным требованием является поддержка определенного способа расширения компонентами приложения;
- когда расширение управляется плагинами, важное значение приобретает возможность совместного использования внутренних служб приложения (в предыдущем небольшом примере общей службой был экземпляр `app`), иначе мы просто не смогли бы их расширить. Когда расширение управляется приложением, плагинам также может потребоваться доступ к некоторым службам приложения, но не для расширения, а для использования. Например, плагину мо-

жет понадобиться обратиться к экземпляру db в приложении или использовать средство журналирования.

Последний пункт явно показывает, насколько важно для плагинов иметь доступ к службам приложения. Именно этот вопрос займет центральное место в наших дальнейших исследованиях. Любую проблему лучше исследовать на практическом примере. Поэтому далее мы рассмотрим реализацию расширения, управляемого плагинами, требующую минимальных усилий по организации инфраструктуры, что позволит осветить проблему совместного использования состояния приложения плагинами.

Реализация плагина выхода из системы

Создадим небольшой плагин для сервера аутентификации. Текущая реализация приложения не предусматривает возможности сделать маркер недействительным; он становится недействительным, только когда истекает время его действия. Теперь мы добавим эту возможность, то есть реализуем *выход из системы*, не изменяя кода основного приложения, а делегировав эту задачу внешнему плагину.

Для поддержки новой возможности мы будем сохранять каждый созданный маркер в базе данных и затем проверять его наличие при каждой попытке проверить его допустимость. Чтобы сделать маркер недействительным, достаточно просто удалить его из базы данных.

Используем для этого прием расширения, управляемого плагином, которому делегируем вызовы методов `authService.login()` и `authService.checkToken()`. Затем *декорируем* модуль `authService` новым методом `logout()`. После этого зарегистрируем новый маршрут на главном Express-сервере (`/logout`), который позволит сделать маркер недействительным с помощью HTTP-запроса.

Мы реализуем четыре варианта только что описанного плагина:

- с использованием жестких зависимостей;
- с использованием внедрения зависимостей;
- с использованием локатора служб;
- с использованием DI-контейнера.

Доступ к службам с использованием жестких зависимостей

Первый вариант соответствует ситуации, когда приложение использует в основном жесткие зависимости для связывания модулей с состоянием. В этом контексте, если поместить плагин в пакет, расположенный в каталоге `node_modules`, мы должны дать ему доступ к родительскому пакету, чтобы он смог воспользоваться службами основного приложения. Это можно сделать двумя способами:

- использовать метод `require()` и определить корневой каталог приложения с помощью относительного или абсолютного пути;
- использовать метод `require()` из родительского приложения при создании экземпляра плагина для его маскировки под обычный модуль приложения. Это позволит плагину получить доступ к любым службам приложения через метод `require()`, как если бы они вызывались из родительского приложения, а не из плагина.

Первый способ менее надежен, поскольку предполагает, что пакету известно местоположение основного приложения. Прием с маскировкой, напротив, не зависит от местоположения каталога, откуда загружается пакет, поэтому именно его мы используем для реализации следующего примера.

В первую очередь нужно создать новый пакет `authsrv-plugin-logout` в каталоге `node_modules`. Но, прежде чем начинать писать код, создадим минимальный файл `package.json` с описанием пакета, заполнив в нем лишь основные параметры (полный путь к файлу: `node_modules/authsrv-plugin-logout/package.json`):

```
{
  "name": "authsrv-plugin-logout",
  "version": "0.0.0"
}
```

Теперь можно приступать к главному модулю плагина, который мы поместим в файл `index.js`, поскольку именно этот файл платформа Node.js попытается загрузить, встретив инструкцию подключения пакета (если не определено свойство `main` в файле `package.json`). Как всегда, первые строки модуля предназначены для загрузки зависимостей. Обратите внимание на порядок их следования (файл `node_modules/authsrv-plugin-logout/index.js`):

```
const parentRequire = module.parent.require;

const authService = parentRequire('./lib/authService');
const db = parentRequire('./lib/db');
const app = parentRequire('./app');

const tokensDb = db.sublevel('tokens');
```

Вся особенность заключена в первой строке. Она получает ссылку на функцию `require()` в родительском модуле `parent`, который должен загружать плагин. В нашем случае роль родительского модуля будет играть модуль `app` основного приложения, то есть всякий раз, вызывая функцию `parentRequire()`, мы будем загружать соответствующий модуль, как если бы он загружался модулем `app.js`.

Следующий шаг – создание прокси для метода `authService.login()`. Этот шаблон уже рассматривался в *главе 6 «Шаблоны проектирования»*:

```
const oldLogin = authService.login;                    //[1]
authService.login = (username, password, callback) => {
  oldLogin(username, password, (err, token) => {       //[2]
    if(err) return callback(err);                      //[3]

    tokensDb.put(token, {username: username}, () => {
      callback(null, token);
    });
  });
}
```

Этот фрагмент выполняет следующие действия:

1) сохраняет ссылку на старый метод `login()` и заменяет его проксированной версией;
2) новая функция вызывает прежнюю версию метода `login()` и передает ему свою функцию обратного вызова, чтобы получить оригинальное возвращаемое значение;
3) если оригинальный метод `login()` вернет ошибку, она просто передается функции `callback`, в противном случае маркер сохраняется в базу данных.

Аналогично реализуется перехват вызовов `checkToken()`:

```
const oldCheckToken = authService.checkToken;

authService.checkToken = (token, callback) => {
  tokensDb.get(token, function(err, res) {
    if(err) return callback(err);

    oldCheckToken(token, callback);
  });
}
```

На этот раз, прежде чем вызвать оригинальный метод `checkToken()`, мы проверяем присутствие маркера в базе данных. Если маркер не найден, операция `get()` возвращает ошибку. Это означает, что маркер считается недействительным и в функцию обратного вызова передается ошибка.

Для завершения расширения модуля `authService` осталось декорировать его новым методом, позволяющим сделать маркер недействительным:

```
authService.logout = (token, callback) => {
  tokensDb.del(token, callback);
}
```

Метод `logout()` очень прост: он всего лишь удаляет маркер из базы данных.

Наконец, можно добавить новый маршрут в Express-сервер для поддержки новой функциональной возможности:

```
app.get('/logout', (req, res, next) => {
  authService.logout(req.query.token, function() {
    res.status(200).send({ok: true});
  });
});
```

Теперь плагин можно подключить к основному приложению. Для этого вернемся в основной каталог приложения и изменим модуль `app.js`:

```
// ...
let app = module.exports = express();
app.use(bodyParser.json());

require('authsrv-plugin-logout');

app.post('/login', authController.login);
app.all('/checkToken', authController.checkToken);
// ...
```

Как видите, чтобы подключить плагин, достаточно просто загрузить его. Как только это случится – во время запуска приложения, – управление будет передано плагину, который осуществит расширение модулей `authService` и `app`, как было показано выше.

Теперь сервер аутентификации поддерживает операцию прекращения действия маркера. При этом мы практически не изменили ядро приложения и для расширения его возможностей применили шаблоны «Прокси» и «Декоратор».

Теперь попробуем запустить приложение еще раз:

```
node app
```

Убедимся, что новая веб-служба `/logout` поддерживается и работает надлежащим образом. Используем `curl` для получения нового маркера с помощью `/login`:

```
curl -X POST -d '{"username": "alice", "password":"secret"}'
http://localhost:3000/login -H "Content-Type: application/json"
```

Затем проверим допустимость маркера с помощью `/checkToken`:

```
curl -X GET -H "Accept: application/json"
http://localhost:3000/checkToken?token=<TOKEN HERE>
```

Далее передадим маркер конечной точке `/logout`, чтобы сделать его недействительными. Это можно сделать следующей командой:

```
curl -X GET -H "Accept: application/json"
http://localhost:3000/logout?token=<TOKEN HERE>
```

Теперь повторная проверка маркера должна вернуть отрицательный результат, что подтвердит правильную работу плагина.

Пример даже такого маленького плагина наглядно демонстрирует преимущества такого способа расширения. Попутно мы узнали, как получить доступ к службам основного приложения из другого пакета, с помощью приема маскировки под модуль приложения.

Шаблон маскировки используется несколькими плагинами NodeBB; вы можете исследовать их исходный код, чтобы получить представление, как этот шаблон используется в реальных приложениях. Вот ссылки на соответствующие примеры:

nodebb-plugin-poll:
https://github.com/Schamper/nodebb-plugin-poll/blob/b4a46561aff279e19c23b7c635fda5037c534b84/lib/nodebb.js
nodebb-plugin-mentions:
https://github.com/julianlam/nodebb-plugin-mentions/blob/9638118fa7e06a05ceb24eb521427440abd0dd8a/library.js#L4-13

Маскировка под модули, как вы уже наверняка поняли, является формой жесткой зависимости и наследует все ее сильные и слабые стороны. С одной стороны, этот прием позволяет с минимальными усилиями организовать доступ к службам основного приложения, но с другой – создает тесную связь не только с конкретным экземпляром службы, но и ее местоположением, из-за чего простой рефакторинг основного приложения может нарушить работоспособность плагина.

Доступ к службам с использованием локатора служб

Подобно маскировке под модуль приложения, локатор служб также является хорошим выбором, когда требуется дать плагину доступ ко всем компонентам приложения. Однако локатор служб имеет большое преимущество, поскольку с его помощью плагины могут открывать доступ к своим службам для приложения или даже для других плагинов.

Давайте теперь перепишем код плагина выхода из системы и используем в нем локатор служб. Для этого изменим основной модуль плагина в файле `node_modules/authsrv-plugin-logout/index.js`:

```
module.exports = (serviceLocator) => {
  const authService = serviceLocator.get('authService');
  const db = serviceLocator.get('db');
  const app = serviceLocator.get('app');
```

```
  const tokensDb = db.sublevel('tokens');

  const oldLogin = authService.login;
  authService.login = (username, password, callback) => {
    //...то же, что и в предыдущей версии
  }
  const oldCheckToken = authService.checkToken;
  authService.checkToken = (token, callback) => {
    //...то же, что и в предыдущей версии
  }

  authService.logout = (token, callback) => {
    //...то же, что и в предыдущей версии
  }

  app.get('/logout', (req, res, next) => {
    //...то же, что и в предыдущей версии
  });
};
```

Теперь, когда плагин получает локатор служб родительского приложения, он сможет обратиться к любой из его служб, когда это потребуется. Это означает, что приложение не должно заранее знать, какие зависимости понадобятся плагину, а это, безусловно, является важным преимуществом при реализации расширения, управляемого плагинами.

Следующий шаг – вызов плагина из основного приложения, для чего необходимо изменить модуль `app.js`. Используем для этого версию сервера аутентификации, реализованную на основе шаблона локатора служб. Необходимые изменения приводятся ниже:

```
// ...
const svcLoc = require('./lib/serviceLocator')();
svcLoc.register(...);
// ...

svcLoc.register('app', app);
const plugin = require('authsrv-plugin-logout');
plugin(svcLoc);

// ...
```

Изменения, выделенные жирным:

- регистрируют сам модуль `app` в локаторе служб, чтобы дать плагину возможность обращаться к нему;
- загружают плагин;
- вызывают главную функцию плагина и передают ей локатор служб в качестве аргумента.

Как мы уже говорили, главное преимущество локатора служб – в том, что он позволяет легко экспортировать службы приложения плагинам, а также может использоваться как механизм совместного использования служб плагина в родительском приложении и даже в других плагинах. Это последнее соображение является наиболее важной причиной использования шаблона «Локатор служб» в контексте расширяемости, основанной на плагинах.

Доступ к службам с использованием DI

Используя прием внедрения зависимостей (DI), легко можно открыть доступ к службам приложения, как если бы плагин являлся частью приложения. Этот шаблон становится практически обязательным, если уже является основным методом подключения зависимостей в родительском приложении. Однако ничто не мешает использовать его, когда в приложении используются жесткие зависимости или локатор служб. DI также идеально подходит для реализации расширения, управляемого приложением, поскольку обеспечивает лучший контроль над совместным использованием служб.

Для проверки этих предположений попробуем реорганизовать плагин выхода из системы и использовать в нем прием внедрения зависимостей. Для этого потребуются минимальные изменения. Начнем с главного модуля пакета (`node_modules/authsrv-plugin-logout/index.js`):

```
module.exports = (app, authService, db) => {
  const tokensDb = db.sublevel('tokens');

  const oldLogin = authService.login;
  authService.login = (username, password, callback) => {
    //...то же, что и в предыдущей версии
  }

  let oldCheckToken = authService.checkToken;
  authService.checkToken = (token, callback) => {
    //...то же, что и в предыдущей версии
  }

  authService.logout = (token, callback) => {
    //...то же, что и в предыдущей версии
  }

  app.get('/logout', (req, res, next) => {
    //...то же, что и в предыдущей версии
  });
};
```

Мы просто завернули код плагина в фабричную функцию, которая получает службы родительского приложения в виде аргументов. Больше не было произведено никаких изменений.

Нам также необходимо изменить способ подключения плагина к родительскому приложению, для чего изменим одну строку в модуле `app.js`, которая загружает плагин:

```
// ...
const plugin = require('authsrv-plugin-logout');
plugin(app, authService, authController, db);
// ...
```

Мы намеренно не показали, как были получены эти зависимости, – это не имеет никакого значения, поскольку подойдет любой способ. Можно использовать жесткую привязку зависимостей, получить экземпляры с помощью фабрики или локатора службы – выбор конкретного способа не играет никакой роли. Это доказывает гиб-

кость шаблона DI и возможность его применения для подключения плагинов, независимо от способа связывания служб, выбранного в родительском приложении.

Но отличия получились более глубокими, чем может показаться. Шаблон DI – безусловно, лучший способ организации доступа к набору служб из плагинов, но самое главное, он позволяет полностью контролировать доступность служб и защищать приложение от чрезмерно агрессивных расширений. Однако эта же характеристика может считаться недостатком, поскольку основное приложение не всегда знает, какие службы потребуются плагинам, поэтому в конечном итоге внедряются все службы, что непрактично, или только их часть, например лишь основные службы родительского приложения. По этой причине шаблон DI не является идеальным выбором для расширения, управляемого плагином. Однако эту проблему легко решить с применением DI-контейнеров.

Инструмент Grunt (http://gruntjs.com) сборки проектов для платформы Node.js использует DI для предоставления всем плагинам доступа к службам Grunt. С их помощью любой плагин сможет присоединить новые задания и использовать их для извлечения конфигурационных параметров или для выполнения других заданий. Вот как выглядит типичный плагин для Grunt:

```
module.exports = function(grunt) {
  grunt.registerMultiTask('taskName', 'description',
    function(...) {...}
  );
};
```

Доступ к службам с использованием DI-контейнера

Взяв за основу предыдущий пример, можно использовать DI-контейнер в сочетании с тем же плагином, лишь немного изменив модуль app, как показано ниже:

```
// ...
const diContainer = require('./lib/diContainer')(); diContainer.register(...);
// ...
//инициализировать плагин
diContainer.inject(require('authsrv-plugin-logout'));
// ...
```

После регистрации фабрик или экземпляров приложения остается только создать экземпляр плагина, что достигается внедрением его зависимостей с помощью DI-контейнера. Таким способом любой плагин сможет загрузить свой набор зависимостей, без участия родительского приложения. Все связи, как и раньше, будут установлены DI-контейнером автоматически.

Использование DI-контейнера также означает, что каждый из плагинов сможет получить доступ к любой службе приложения, что затрудняет сокрытие информации и контроль за использованием служб расширением. Одно из возможных решений этой проблемы: создать отдельный DI-контейнер и регистрировать в нем только те службы, которые могут быть доступны из плагинов. Так можно управлять доступом к основному приложению из плагинов. Это показывает, что DI-контейнер можно сделать удачным выбором и с точки зрения инкапсуляции и сокрытия информации.

На этом мы завершаем исследование вариантов реализации плагина выхода из системы и сервера аутентификации.

Итоги

Тема подключения зависимостей, безусловно, является одной из самых важных в области разработки программного обеспечения. В этой главе мы попытались проанализировать разные способы на практических примерах, чтобы дать максимально объективный обзор наиболее важных шаблонов. Мы развеяли некоторые из распространенных сомнений, касающихся объектов-одиночек и экземпляров в Node.js, и рассмотрели способы связывания модулей с использованием жестких зависимостей, DI и локаторов служб. Каждый способ был продемонстрирован на примере сервера аутентификации, используемого в качестве своеобразной песочницы, что позволило прояснить плюсы и минусы каждого из подходов.

Во второй части главы мы узнали, как организовать поддержку плагинов в приложениях и как подключать их к основному приложению. Мы использовали для этого приемы, предоставленные в первой части главы, но рассмотрели их с другой точки зрения. Мы также увидели, насколько важно обеспечить доступность служб основного приложения для плагинов и как это влияет на их возможности.

Эта глава содержит все сведения, необходимые для выбора оптимального подхода, обеспечивающего нужный уровень связанности, повторного использования и простоты для конкретного приложения. Была показана возможность использования нескольких шаблонов в одном приложении. Например, в качестве основного способа можно использовать жесткие зависимости, а для подключения плагинов – применить локатор служб. На самом деле не существует никаких ограничений, если знать и учитывать преимущества каждого из подходов.

В предыдущей части книги основное внимание уделялось наиболее общим и универсальным шаблонам, но начиная со следующей главы мы приступим к исследованию приемов решения более конкретных технических задач. Далее вас ожидает, по сути, коллекция рецептов, которые можно использовать для решения конкретных проблем, связанных с преимущественно вычислительными задачами, асинхронным кэшированием и использованием кода совместно с браузером.

Глава 8

Универсальный JavaScript для веб-приложений

Язык JavaScript появился на свет в 1995 году, и его изначальной целью было предоставление веб-разработчикам возможности выполнять код непосредственно в браузере, чтобы создавать более динамичные и интерактивные веб-сайты.

С тех пор возможности языка JavaScript значительно расширились, и сегодня это один из самых известных и распространенных языков программирования. Если в самом начале JavaScript был очень простым языком с весьма ограниченными возможностями, то сегодня его можно считать полноценным языком общего назначения, который можно использовать вне браузера для создания приложений практически любых видов. Фактически язык JavaScript теперь применяется в клиентских приложениях, веб-серверах и мобильных приложениях, а также в разного рода носимых и промышленных устройствах, в беспилотных летательных аппаратах.

Такое разнообразие платформ и устройств способствовало появлению в среде разработчиков на JavaScript новой тенденции, заключающейся в желании упростить повторное использование кода в различных средах в рамках одного проекта. Применительно к платформе Node.js это означает возможность создания веб-приложений, совместно использующих один и тот же код на сервере (на стороне сервера) и в браузере (на стороне клиента). Такая возможность для повторного использования кода первоначально получила название «**изоморфный JavaScript**», но в настоящее время стал популярным термин «**универсальный JavaScript**».

В этой главе мы займемся исследованием преимуществ универсального JavaScript, в том числе в области веб-разработки, и познакомимся со множеством инструментов и технологий, позволяющих совместно использовать код на сервере и в браузере.

Узнаем, как пользоваться модулями на сервере и на клиенте, как применять инструменты упаковки кода для браузеров, такие как **Webpack** и **Babel**. Примем на вооружение библиотеку React и другие популярные модули для создания веб-интерфейсов и обмена информацией о состоянии между веб-сервером и клиентом и, наконец, проанализируем несколько интересных решений маршрутизации и поиска данных в приложениях.

Прочитав эту главу до конца, вы научитесь писать **одностраничные приложения** (Single-Page Application, SPA) на основе библиотеки React, способные повторно использовать большую часть кода, имеющегося на сервере Node.js, что делает их последовательными, наглядными и простыми в обслуживании.

Использование кода совместно с браузером

Одна из главных причин успеха платформы Node.js заключается в том, что она основана на языке JavaScript и работает на движке V8, который также использует один из самых популярных браузеров: Chrome. Казалось бы, этого достаточно, чтобы решить, что совместное использование кода платформой Node.js и браузером является простой задачей, но, как мы увидим далее, это не всегда верно, особенно если речь идет не о маленьких и автономных фрагментах кода. Разработка кода для совместного использования на клиенте и сервере требует серьезных усилий, чтобы гарантировать работоспособность кода в обеих средах, принципиально отличающихся друг от друга. Например, в Node.js отсутствует модель DOM или долговременные представления, в то время как в браузере наверняка нет файловой системы или возможности запуска новых процессов. Кроме того, следует учесть, что в Node.js доступны многие новые возможности из спецификации ES2015. Для браузеров это не так, поскольку большинство браузеров все еще пользуется спецификацией ES5, и поддержание совместимости клиентского кода со спецификацией ES5 еще долго будет оставаться самым надежным решением, пока все веб-браузеры не реализуют поддержку ES2015.

Соответственно, большая часть усилий при разработке кода для обеих платформ будет направлена уменьшение этих разногласий до минимума. Это можно сделать с помощью абстракций и шаблонов, которые позволяют приложению переключаться, динамически или во время сборки, между кодом, совместимым с браузером, и кодом для платформы Node.js.

К счастью, с ростом интереса к этим новым умопомрачительным возможностям многие библиотеки и фреймворки экосистемы начали поддерживать обе среды. Такая эволюция стала возможной также благодаря увеличению числа инструментов поддержки, которые в последние годы были доработаны и значительно усовершенствованы. Это означает, что при использовании npm-пакета на платформе Node.js велика вероятность, что он будет нормально работать и в браузере. Однако этого обычно недостаточно, чтобы гарантировать безупречную работу приложения в браузере и на платформе Node.js. Как будет показано ниже, при разработке кросс-платформенного кода необходимо тщательное проектирование.

В этом разделе мы рассмотрим основные проблемы, с которыми часто приходится сталкиваться при написании кода для Node.js и браузера, и познакомимся с несколькими инструментами и шаблонами, способными помочь при решении этой новой и интересной задачи.

Совместное использование модулей

Первой стеной, в которую упираются разработчики при создании кода, общего для браузера и сервера, является несоответствие системы модулей в Node.js и разнообразных систем в браузерах. Другая проблема заключается в том, что в браузере отсутствуют функция `require()` и файловая система, позволяющая извлекать модули. Так что если требуется написать достаточно объемный код, способный выполняться на обеих платформах, и использовать при этом систему модулей CommonJS, нам понадобятся инструмент подключения зависимостей и абстрактный механизм, заменяющий `require()` в браузере.

Универсальное определение модулей

В Node.js для определения зависимостей между компонентами по умолчанию используется механизм поддержки модулей CommonJS. К сожалению, для браузеров ситуация не столь однозначна:

- можно обойтись и без системы модулей, превратив глобальную область в основной механизм доступа к другим модулям;
- можно воспользоваться одной из реализаций **асинхронного определения модулей** (Asynchronous Module Definition, AMD), например **RequireJS** (http://requirejs.org);
- можно воспользоваться средой, основанной на системе модулей CommonJS.

К счастью, существует шаблон **универсального определения модулей** (Universal Module Definition, UMD), который может помочь абстрагировать код от используемой системы модулей.

Создание UMD-модуля Механизм UMD не полностью стандартизирован, поэтому имеется достаточно много его вариантов, зависящих от потребностей компонента и поддерживаемой системы модулей. Однако существует одна его форма, которая является самой популярной и позволяет поддерживать наиболее распространенные системы модулей, такие как AMD, CommonJS и глобальная область браузера.

Рассмотрим простой пример. Создадим новый проект и в нем новый модуль `umdModule.js`:

```
(function(root, factory) {                          //[1]
  if(typeof define === 'function' && define.amd) {  //[2]
    define(['mustache'], factory);
  } else if(typeof module === 'object' &&           //[3]
    typeof module.exports === 'object') {
      var mustache = require('mustache');
      module.exports = factory(mustache);
    } else {                                        //[4]
      root.UmdModule = factory(root.Mustache);
    }
  }(this, function(mustache) {                      //[5]
    var template = '<h1>Hello <i>{{name}}</i></h1>';
    mustache.parse(template);

    return { sayHello:function(toWhom) {
      return mustache.render(template, {name: toWhom});
    }
  };
}));
```

В предыдущем примере определяется простой модуль с одной внешней зависимостью `mustache` (http://mustache.github.io) – простым движком текстовых шаблонов. Конечный продуктом предыдущего UMD-модуля является объект с одним методом `sayHello()`, который заполняет шаблон и возвращает его вызвавшей стороне. Целью UMD является интеграция модуля с другими системами модулей, доступными в конкретной среде. Вот как это работает:

1) весь код завернут в анонимную и немедленно вызывающуюся функцию, что очень похоже на шаблон «Открытый модуль» (Revealing Module), рассмотрен-

ный в *главе 2* «*Основные шаблоны Node.js*». Функция принимает аргумент `root` – объект глобального пространства имен (например, `window` в браузере). Это в основном нужно для регистрации зависимостей в виде глобальной переменной, как будет показано чуть ниже. Второй аргумент модуля – `factory()` – это функция, принимающая зависимости и возвращающая экземпляр модуля (шаблон «Внедрение зависимостей»);

2) первым делом функция проверяет доступность реализации AMD в системе. С этой целью она проверяет наличие функции `define` и присутствие у нее флага `amd`. Если оба условия выполняются, значит, в системе есть AMD-загрузчик и регистрация модуля производится с помощью функции `define`, а нужная ему зависимость `mustache` внедряется через `factory()`;
3) если реализация AMD отсутствует, проверяется наличие в системе среды CommonJS, предпочитаемой платформой Node.js, для чего определяется присутствие объектов `module` и `module.exports`. Если эти объекты имеются, зависимости модуля загружаются с помощью `require()` и передаются в вызов `factory()`. Значение, возвращаемое фабрикой, присваивается `module.exports`;
4) и наконец, если нет ни AMD, ни CommonJS, модуль присваивается глобальной переменной с помощью корневого объекта `root`, роль которого в браузерах обычно играет объект `window`. Кроме того, как можно заметить, предполагается наличие зависимости `Mustache` в глобальной области;
5) далее происходит автоматический вызов обертывающей функции, которой в качестве `root` передаются объект `this` (в браузере это объект `window`) и фабрика модуля. Как видите, фабрика принимает аргументы с зависимостями.

Стоит также отметить, что в этом модуле не используются возможности, определяемые стандартом ES2015. Это гарантирует работоспособность кода в браузерах.

Теперь посмотрим, как использовать этот UMD-модуль в Node.js и в браузере.

Прежде всего создадим новый файл `testServer.js`:

```
const umdModule = require('./umdModule');
console.log(umdModule.sayHello('Server!'));
```

Если выполнить этот сценарий, он выведет:

```
<h1>Hello <i>Server!</i></h1>
```

Чтобы проверить новоиспеченный модуль на стороне клиента, создадим страницу `testBrowser.html` со следующим содержимым:

```
<html>
  <head>
    <script src="node_modules/mustache/mustache.js"></script>
    <script src="umdModule.js"></script>
  </head>
  <body>
    <div id="main"></div>
    <script>
      document.getElementById('main').innerHTML =
      UmdModule.sayHello('Browser!');
    </script>
  </body>
</html>
```

Она выводит очень оригинальную надпись «**Hello Browser!**» в верхней части.

В заголовке этой страницы мы подключили наши зависимости (`mustache` и `umdModule`) как обычные сценарии, а затем добавили коротенький встроенный сценарий, использующий `UmdModule` (доступный в браузере как глобальная переменная) для создания некоторого HTML-кода, помещаемого внутрь основного блока.

В примерах кода для этой книги, которые можно найти на сайте издательства Packt, имеются другие примеры использования этого UMD-модуля в сочетании с AMD-загрузчиком и системой CommonJS.

Комментарии к шаблону UMD Шаблон UMD – эффективный и простой способ создания модулей, совместимых с наиболее популярными системами модулей. Но, как мы увидели, он требует определенного объема стереотипного кода, который трудно протестировать в каждой из сред, что неизбежно приведет к ошибкам. Это означает, что писать вручную стереотипный код для UMD имеет смысл только для обертывания отдельного модуля, который уже был разработан и протестирован. Не стоит использовать этот способ для создания новых модулей с нуля, поскольку это утомительно и непрактично. В таких ситуациях лучше воспользоваться инструментами, которые помогут автоматизировать этот процесс. Одним из таких инструментов является Webpack, который будет использоваться далее в этой главе.

Следует также отметить, что AMD, CommonJS и глобальная область браузера – не единственные системы модулей. Представленный шаблон способен охватывать большинство вариантов использования, но потребует добавления поддержки любой другой системы модулей. Например, модули, соответствующие стандарту ES2015, которые рассматриваются в следующем разделе, предоставляют целый ряд преимуществ, по сравнению с другими решениями, и к тому же являются частью нового стандарта ECMAScript (даже притом, что на момент написания этих строк они еще не поддерживались в Node.js).

Большой перечень формализованных шаблонов UMD можно найти на странице https://github.com/umdjs/umd.

ES2015-модули

Одной из особенностей, определяемых стандартом ES2015, является встроенная система модулей. В этой книге она упоминается впервые, поскольку на момент написания этих строк, к сожалению, ES2015-модули еще не поддерживаются текущей версией Node.js.

Мы не будем подробно описывать эту особенность, но вы должны знать о ее существовании, так как, скорее всего, она станет преобладающей в ближайшие годы. Помимо того что ES2015-модули являются стандартом, этот синтаксис обеспечивает ряд преимуществ, по сравнению с другими системами модулей, описанными выше.

Поддержка модулей была определена в ES2015 с целью объединить лучшие качества CommonJS и AMD:

- подобно CommonJS, эта спецификация обеспечивает компактный синтаксис, отдает предпочтение единственному экспорту, а также поддерживает циклические зависимости;
- подобно AMD, предлагает прямую поддержку асинхронной загрузки и настраиваемую загрузку модулей.

Кроме того, декларативный синтаксис позволяет использовать статические анализаторы, например для статической проверки и оптимизации. Например, можно проанализировать дерево зависимостей сценария и создать связанный файл для браузера, из которого удалены все неиспользуемые функции, за счет чего уменьшаются размер файла и время его загрузки клиентом.

Более подробные сведения о синтаксисе ES2015-модулей можно найти в спецификации ES2015 на странице http://www.ecma-international.org/ecma-262/6.0/#sec-scripts-and-modules.

В настоящее время новый синтаксис модулей можно использовать на платформе Node.js с помощью таких транскомпиляторов, как Babel. Фактически многие разработчики пропагандируют собственные решения для создания приложений на универсальном JavaScript. Это, в общем, хорошая идея, нацеленная в будущее, поскольку данная возможность уже стандартизирована и рано или поздно станет частью ядра Node.js. Далее в этой главе для простоты будет использоваться только синтаксис CommonJS.

Введение в Webpack

Разрабатывая приложения для платформы Node.js, старайтесь не использовать систем модулей, отличных от той, что предлагается по умолчанию. В идеале нужно продолжать писать модули как обычно, используя `require()` и `module.exports`, а затем применять инструмент для преобразования кода в пакет, который будет нормально работать в браузере. К счастью, эта задача имеет множество решений, самым популярным из которых является Webpack (https://webpack.github.io).

Webpack позволяет создавать модули с использованием соглашений о модулях Node.js, с последующей компиляцией в пакет (единственный JavaScript-файл), содержащий все необходимое для работы в браузере (в том числе абстрактные функции `require()`). Такой пакет затем можно включить в веб-страницу, как обычно, и выполнить в браузере. Webpack рекурсивно сканирует исходный код, отыскивает ссылки на функцию `require()`, разрешает их и добавляет нужные модули в пакет.

Webpack – не единственный инструмент сборки пакетов для браузера из модулей Node.js. Другими популярными альтернативами являются: `Browserify` (http://browserify.org), `RollupJs` (http://rollupjs.org) и `Webmake` (https://npmjs.org/package/webmake). Кроме того, библиотека `require.js` позволяет создавать модули для клиента и Node.js, но она использует AMD вместо CommonJS (http://requirejs.org/docs/node.html).

Знакомство с волшебством Webpack

Для быстрой демонстрации волшебства Webpack рассмотрим, как будет выглядеть созданный в предыдущем разделе модуль `umdModule` при использовании этого инструмента. Во-первых, необходимо установить сам Webpack, что можно сделать с помощью следующей простой команды:

```
npm install webpack -g
```

Параметр `-g` указывает `npm`, что необходимо установить Webpack глобально, чтобы его можно было использовать из консоли, как будет показано ниже.

Далее, создадим новый проект и попробуем написать модуль, эквивалентный модулю umdModule. Вот так будет выглядеть его реализация для платформы Node.js (файл sayHello.js):

```
var mustache = require('mustache');
var template = '<h1>Hello <i>{{name}}</i></h1>';
mustache.parse(template);
module.exports.sayHello = function(toWhom) {
  return mustache.render(template, {name: toWhom});
};
```

Определенно он проще, чем шаблон UMD, не так ли? Теперь создадим файл main.js, то есть точку входа в код для браузера:

```
window.addEventListener('load', function(){
  var sayHello = require('./sayHello').sayHello;
  var hello = sayHello(Browser!');
  var body = document.getElementsByTagName("body")[0];
  body.innerHTML = hello;
});
```

В приведенном выше коде загрузка модуля sayHello выполняется точно так же, как в Node.js, нет никаких дополнительных сложностей при управлении зависимостями и настройке путей, поскольку всю работу выполняет обычная функция require().

Добавим зависимость mustache в проект:

```
npm install mustache
```

А теперь волшебное действие. Запустим следующую команду в терминале:

```
webpack main.js bundle.js
```

Она скомпилирует модуль main и соберет все необходимые зависимости в один файл bundle.js, готовый к использованию в браузере!

Чтобы быстро проверить работоспособность получившегося пакета, создадим HTML-страницу magic.html со следующим кодом:

```
<html>
  <head>
    <title>Webpack magic</title>
    <script src="bundle.js"></script>
  </head>
  <body>
  </body>
</html>
```

Этого достаточно для выполнения кода в браузере. Попробуйте открыть страницу и взгляните сами. Ура!

В процессе разработки желательно не запускать Webpack вручную при каждом внесении изменений в исходные тексты. Предпочтительнее использовать механизм, который автоматически будет собирать пакет при любых изменениях в исходном коде. С этой целью можно выполнить команду Webpack с параметром --watch. Он требует от Webpack выполняться непрерывно и заботиться о повторной сборке пакета при любом изменении одного из исходных файлов.

Преимущества использования Webpack

Волшебство Webpack этим не ограничивается. Вот (неполный) перечень функциональных возможностей, которые упрощают совместное использование кода с браузером:

- Webpack автоматически предоставляет версии многих модулей ядра Node.js, совместимые с браузером. Это означает, что в браузере можно использовать такие модули, как `http`, `assert`, `events`, и многие другие!

Модуль `fs` входит в число тех, которые не поддерживаются.

- Если имеется модуль, несовместимый с браузером, его можно исключить из сборки, заменить пустым объектом или другим модулем с альтернативной реализацией, которая совместима с браузером. Это играет решающую роль, и данная возможность будет использоваться в примере ниже;
- Webpack может генерировать пакеты для различных модулей;
- Webpack позволяет выполнять дополнительную обработку исходных файлов с помощью сторонних **загрузчиков** и **плагинов**. Имеются загрузчики и плагины практически для всего, что может понадобиться, от компиляции CoffeeScript, TypeScript и ES2015 до поддержки загрузки AMD, пакетов Bower (http://bower.io) и Component (http://component.github.io), использующих `require()`, от минификации до компиляции и упаковки других ресурсов, таких как шаблоны и таблицы стилей;
- Webpack с легкостью можно вызывать из диспетчеров заданий, таких как Gulp (https://npmjs.com/package/gulp-webpack) и Grunt (https://npmjs. org/package/grunt-webpack);
- Webpack позволяет управлять и выполнять предварительную обработку ресурсов проекта, причем не только JavaScript-файлов, но и таблиц стилей, изображений, шрифтов и шаблонов;
- кроме того, Webpack можно настроить для разделения дерева зависимостей на несколько частей, которые будут загружаться по требованию, когда они понадобятся браузеру.

Потенциал и гибкость Webpack настолько привлекательны, что многие разработчики начали использовать его даже для управления кода, предназначенного только для выполнения на стороне клиента. Это стало возможным потому, что многие клиентские библиотеки начали по умолчанию поддерживать CommonJS и npm, что открывает новые и очень интересные перспективы. Например, можно установить библиотеку jQuery:

```
npm install jquery
```

а затем загрузить ее с помощью простой строки кода:

```
const $ = require('jquery');
```

Вы будете удивлены, как много клиентских библиотек уже поддерживают CommonJS и Webpack.

Использование ES2015 с помощью Webpack

Как уже упоминалось в предыдущем разделе, одним из главных преимуществ Webpack является возможность использования загрузчиков и плагинов для преобразования исходного кода перед включением в пакет.

В этой книге уже использовались многие из новых удобных функций, предлагаемых стандартом ES2015, и хотелось бы продолжать их использовать в приложениях на универсальном JavaScript. В этом разделе мы посмотрим, как использовать возможности загрузчика Webpack и переписать предыдущий пример с использованием синтаксиса ES2015. При соответствующей настройке Webpack способен позаботиться о транскомпиляции кода в стандарт ES5, чтобы гарантировать максимальную совместимость со всеми существующими браузерами.

Прежде всего переместим модули в новый каталог `src`. Это улучшит организацию кода и отделит транскомпилированный код от первоначального исходного кода. Такое разделение также облегчит настройку Webpack и упростит вызов Webpack из командной строки.

Теперь можно переписать все модули. Версия модуля `src/sayHello.js` с поддержкой стандарта ES2015 будет выглядеть так:

```
const mustache = require('mustache');
const template = '<h1>Hello <i>{{name}}</i></h1>';
mustache.parse(template);
module.exports.sayHello = toWhom => {
  return mustache.render(template, {name: toWhom});
};
```

Обратите внимание на использование ключевых слов `const` и `let`, а также синтаксиса стрелочных функций.

Теперь можно привести файл `src/main.js` в соответствие со стандартом ES2015:

```
window.addEventListener('load', () => {
  const sayHello = require('./sayHello').sayHello;
  const hello = sayHello('Browser!');
  const body = document.getElementsByTagName("body")[0];
  body.innerHTML = hello;
});
```

А теперь определим файл `webpack.config.js`:

```
const path = require('path');

module.exports = {
  entry: path.join( dirname, "src", "main.js"),
  output: {
    path: path.join( dirname, "dist"),
    filename: "bundle.js"
  },
  module: {
    loaders: [
      {
        test: path.join( dirname, "src"),
        loader: 'babel-loader',
        query: {
          presets: ['es2015']
        }
      }
    ]
  }
};
```

Это – модуль, экспортирующий объект конфигурации, который читает Webpack при вызове его из командной строки без аргументов.

В объекте конфигурации определяются точка входа, в данном случае файл `src/main.js`, и местоположение файла пакета, в данном случае `dist/bundle.js`.

Эта часть файла достаточно очевидна, так что перейдем к массиву загрузчиков. Этот необязательный массив позволяет указать набор загрузчиков, которые изменяют содержимое исходных файлов при создании файла пакета. Идея заключается в том, что каждый из загрузчиков выполняет конкретное преобразование (в данном случае преобразование кода, соответствующего стандарту ES2015, в код стандарта ES5 с помощью `babel-loader`) и применяется, только если текущий исходный файл соответствует заданному для загрузчика выражению `test`. В этом примере мы потребовали от Webpack использовать `babel-loader` для всех файлов в каталоге `src` и применить комплект преобразований `es2015`.

Мы почти закончили. Осталось выполнить еще один шаг – установить Babel и комплект преобразований ES2015 с помощью следующей команды:

```
npm install babel-core babel-loader babel-preset-es2015
```

Теперь, чтобы создать пакет, достаточно выполнить простую команду:

```
webpack
```

Не забудьте в файле `magic.html` сослаться на новый файл `dist/bundle.js`. Теперь можно открыть файл `magic.html` в браузере и убедиться, что все по-прежнему работает правильно.

Ради интереса можно заглянуть во вновь созданный файл пакета и посмотреть, как все конструкции, соответствующие стандарту ES2015 и использованные в исходных файлах, были преобразованы в эквивалентный код стандарта ES5, с которым прекрасно справится любой браузер.

Основы кросс-платформенной разработки

При разработке для различных платформ наиболее распространенной проблемой, с которой приходится сталкиваться, является совместное использование компонентов с поддержкой разных реализаций для разных платформ. Далее мы рассмотрим несколько принципов и шаблонов, используемых при решении этой задачи.

Ветвление кода во время выполнения

Самый простой и понятный способ поддержки разных реализаций для разных платформ – динамическое ветвление кода. Для этого необходим механизм распознавания платформы во время выполнения с последующим динамическим переключением реализаций с помощью операторов `if...else`. Некоторые общепринятые подходы заключаются в проверке глобальных переменных, доступных только в Node.js или только в браузере. Например, можно проверить существование глобального объекта `window`:

```
if(typeof window !== "undefined" && window.document) {
  //код для клиентской стороны
  console.log('Hey browser!');
} else {
  //код для платформы Node.js
  console.log('Hey Node.js!');
}
```

Использование динамического ветвления для переключения между платформой Node.js и браузером, безусловно, является наиболее понятным и простым шаблоном из всех, которые можно использовать для этой цели. Но он имеет следующие недостатки:

- в одном модуле содержится код для обеих платформ, что увеличивает размер готового пакета из-за включения в него ненужного кода;
- при повсеместном использовании этого приема существенно снижается читаемость кода, поскольку основная логика смешивается с логикой поддержки кросс-платформенной совместимости;
- использование динамического ветвления для загрузки разных модулей в зависимости от платформы приведет к добавлению в окончательный пакет всех модулей независимо от целевой платформы. Например, для следующего фрагмента Webpack включит в пакет оба модуля, `clientModule` и `serverModule`, если один из них не исключить из сборки явно:

```
if(typeof window !== "undefined" && window.document) {
  require('clientModule');
} else {
  require('serverModule');
}
```

Это последнее неудобство объясняется тем, что при сборке пакета отсутствует возможность получить значение переменной среды во время сборки (если переменная не является константой), поэтому включается любой модуль, независимо от того, потребуется ли он в коде.

Как следствие последнего свойства модули, загружаемые динамически с применением переменных, не будут включены в пакет. Например, модуль из следующего кода не будет включен в пакет:

```
moduleList.forEach(function(module) {
  require(module);
});
```

Стоит подчеркнуть, что Webpack обходит некоторые из этих ограничений и при конкретных обстоятельствах способен угадать все возможные значения таких переменных. Например, для следующего фрагмента кода:

```
function getController(controllerName) {
  return require("./controller/" + controllerName);
}
```

он добавит в пакет все модули из каталога `controller`.

Настоятельно рекомендуем обратиться к официальной документации, чтобы познакомиться со всеми поддерживаемыми вариантами.

Ветвление кода в процессе сборки

В этом разделе мы посмотрим, как с помощью Webpack исключить из сборки все фрагменты, предназначенные для выполнения только на стороне сервера. Это позволит сократить размер файла пакета и избежать случайной утечки сервера кода.

Кроме загрузчиков, Webpack предлагает также поддержку плагинов, что позволяет расширить конвейер обработки для сборки файла пакета. Чтобы осуществить ветвле-

ние кода во время сборки, можно воспользоваться конвейером из двух встроенных плагинов: DefinePlugin и UglifyJsPlugin.

Плагин DefinePlugin применяется для замены вхождений конкретных фрагментов кода в исходных файлах на пользовательский код или переменные. Плагин UglifyJsPlugin позволяет сжать конечный код и удалить недостижимые инструкции (мертвый код).

Чтобы лучше понять эти идеи, рассмотрим практический пример. Предположим, что файл main.js содержит следующий код:

```
if (typeof ___BROWSER___ !== "undefined") {
  console.log('Hey browser!');
} else {
  console.log('Hey Node.js!');
}
```

Для него можно определить такой файл webpack.config.js:

```
const path = require('path');
const webpack = require('webpack');

const definePlugin = new webpack.DefinePlugin({
  "___BROWSER___": "true"
});

const uglifyJsPlugin = new webpack.optimize.UglifyJsPlugin({
  beautify: true,
  dead_code: true
});

module.exports = {
  entry: path.join( dirname, "src", "main.js"),
  output: {
    path:  path.join( dirname, "dist"),
    filename: "bundle.js"
  },
  plugins: [definePlugin, uglifyJsPlugin]
};
```

Важным здесь являются определение и настройка двух плагинов, представленных выше.

Первый плагин – DefinePlugin – позволяет заменить отдельные фрагменты исходного кода динамическим кодом или постоянными значениями. Способ настройки несколько сложен, но этот пример поможет понять его суть. В данном случае плагин найдет все вхождения ___BROWSER___ и заменит их на true. Каждое значение в объекте конфигурации (в нашем случае "true" как строка, а не как логическое значение) представляет фрагмент кода, который во время сборки будет использоваться для замены соответствующего фрагмента кода слева от двоеточия. Это позволит включить в пакет внешние динамические значения, содержащие, например, значение переменной среды, текущую отметку времени или хеш последней операции передачи изменений в git. После замены фрагмента ___BROWSER___ первая инструкция if будет выглядеть как if (true ! == "undefined"), но Webpack достаточно умен и способен определить, что это выражение всегда будет равно true, поэтому он преобразует получившийся код в if (true).

Второй плагин – `UglifyJsPlugin` – применяется для обфускации и минификации JavaScript-кода в файле пакета с помощью **UglifyJs** (https://github.com/mishoo/UglifyJS). Если передать плагину UglifyJs параметр `dead_code`, он удалит все, что сочтет «мертвым кодом», поэтому текущий код, получившийся к данному моменту:

```
if (true) {
  console.log('Hey browser!');
} else {
  console.log('Hey Node.js!');
}
```

он преобразует в:

```
  console.log('Hey browser!');
```

Параметр `beautify: true` помогает избежать удаления всех отступов и пробельных символов, чтобы проще было читать получившийся файл пакета. При создании пакетов, предназначенных для эксплуатации, лучше не указывать этот параметр, поскольку его значением по умолчанию является `false`.

В загружаемых примерах кода к этой книге можно найти дополнительный пример, демонстрирующий использование Webpack с плагином `DefinePlugin` для замены определенных констант динамическими переменными, такими как время формирования пакета, текущий пользователь и текущая операционная система.

Даже притом, что этот способ лучше, чем ветвление кода во время выполнения, поскольку позволяет получать более компактные файлы пакетов, он все еще может создавать достаточно громоздкий исходный код при неправильном использовании. Мало кому понравится иметь инструкции ветвления, отделяющие клиентский и серверный коды, разбросанные по всему приложению, не так ли?

Замена модулей

Обычно во время сборки известно, какой код следует включать в пакет для клиентской стороны, а какой нет. Это означает, что решение можно принять заранее и поручить упаковщику, поменять реализацию модуля во время сборки. В результате получается более компактный пакет, поскольку из него исключаются ненужные модули, и улучшается читаемость кода благодаря отсутствию множества операторов `if... else`, нужных для ветвления во время выполнения и при сборке.

Рассмотрим замену модулей с помощью Webpack на очень простом примере.

Создадим модуль, экспортирующий функцию `alert`, которая просто отображает предупреждающее сообщение. Он будет иметь две различные реализации, отдельно для сервера и для браузера. Начнем с реализации для сервера `alertServer.js`:

```
module.exports = console.log;
```

Затем добавим реализацию для браузера `alertBrowser.js`:

```
module.exports = alert;
```

Код предельно прост. Как можно заметить, мы просто использовали функцию по умолчанию `console.log` для сервера и функцию `alert` для браузера. Обе они принимают строку в аргументе, но первая выводит эту строку в консоль, а вторая – в окно.

Теперь напишем код общего модуля `main.js`, который по умолчанию использует модуль для сервера:

```
const alert = require('./alertServer');
alert('Morning comes whether you set the alarm or not!');
```

Здесь нет ничего особенного, код просто импортирует модуль `alertServer` и использует его. Если выполнить команду:

```
node main.js
```

она выведет в консоль текст: `Morning comes whether you set the alarm or not!`

А теперь самое интересное – посмотрим, что должен содержать файл `webpack.config.js`, чтобы выполнить замену `alertServer` на `alertBrowser`, когда потребуется получить пакет для браузера:

```
const path = require('path');
const webpack = require('webpack');

const moduleReplacementPlugin =
  new webpack.NormalModuleReplacementPlugin(/alertServer.js$/,
    './alertBrowser.js');

module.exports = {
  entry: path.join( dirname, "src", "main.js"),
  output: {
    path: path.join( dirname, "dist"),
    filename: "bundle.js"
  },
  plugins: [moduleReplacementPlugin]
};
```

Мы использовали плагин `NormalModuleReplacementPlugin`, принимающий два аргумента. Первый является регулярным выражением, а второй – строкой, содержащей путь к ресурсу. Если ресурс соответствует регулярному выражению, он заменяется ресурсом, указанным во втором аргументе.

В данном примере регулярное выражение соответствует модулю `alertServer`, и он заменяется на `alertBrowser`.

Обратите внимание, что в этом примере используется ключевое слово const, но для простоты была пропущена настройка для транскомпиляции функций ES2015 в эквивалентный код ES5, поэтому получившийся код может не работать в старых браузерах.

Конечно, тот же способ замены применим к внешним модулям из npm. Давайте расширим предыдущий пример, использовав в нем несколько внешних модулей и реализовав их замену.

В настоящее время почти никто не пользуется функцией `alert` по весьма уважительной причине. Эта функция выводит окно, которое выглядит не очень красиво и к тому же блокирует работу браузера, пока пользователь его не закроет. Было бы гораздо приятнее использовать для отображения предупреждения красивую *всплывающую панель*. Такую возможность обеспечивает ряд библиотек в npm, и одна из них – `toastr` (https://npmjs.com/package/toastr), имеющая очень простой программный интерфейс и привлекательный внешний вид.

Поскольку библиотека `toastr` использует jQuery, их следует устанавливать вместе:

```
npm install jQuery toastr
```

Теперь можно переписать модуль `alertBrowser` для использования `toastr` вместо встроенной функции `alert`:

```
const toastr = require('toastr');
module.exports = toastr.info;
```

Функция `toastr.info` принимает строку и отображает заданное сообщение в панели, расположенной в правом верхнем углу окна браузера.

Файл конфигурации останется прежним, но на этот раз Webpack будет разрешать полное дерево зависимостей для новой версии модуля `alertBrowser`, включив в файл пакета jQuery и `toastr`.

Кроме того, серверная версия модуля и файл `main.js` останутся неизменными, что указывает на простоту обслуживания этого решения.

Для нормальной работы данного примера в браузере следует позаботиться о подключении CSS-файла библиотеки toastr в HTML-файле.

Благодаря Webpack и плагину замены модулей можно без особых усилий преодолевать структурные различия платформ. Их использование позволяет сосредоточиться на написании отдельных модулей для конкретной платформы модулей с последующей заменой модулей для Node.js на модули для браузера.

Шаблоны проектирования для кросс-платформенной разработки

Теперь, зная, как переключать код для Node.js и браузера, осталось только выяснить, как интегрировать это решение в проект и как создавать компоненты со взаимозаменяемыми элементами. Эти проблемы уже не новы для нас, поскольку на протяжении всей книги демонстрировались примеры и шаблоны достижения этой цели.

Еще раз взглянем на некоторые из них с точки зрения кросс-платформенной разработки.

- **Стратегия и Макет**: являются наиболее полезными шаблонами при совместном использовании кода с браузером. Их целью, по сути, является определение общих этапов алгоритма, обеспечивающих замену некоторых его частей, а это именно то, что здесь нужно! При кросс-платформенной разработке эти шаблоны позволяют совместно использовать независимые от платформы части компонентов и заменять части, предназначенные для конкретной платформы (что можно делать как во время выполнения, так и при сборке).
- **Адаптер**: этот шаблон больше подходит в ситуации, когда нужно заменить весь компонент. В *главе 6 «Шаблоны проектирования»* мы уже видели, как заменить весь модуль, несовместимый с браузером, адаптером с совместимым интерфейсом. Помните адаптер LevelUP с интерфейсом `fs`?
- **Прокси**: если код, предназначенный для работы на сервере, должен выполняться в браузере, желательно сделать доступным ему все, что находится на сервере. В этом может помочь шаблон *удаленного* прокси. Предположим, что в браузере требуется получить доступ к файловой системе сервера. Для этого можно создать объект `fs` на клиенте, который через прокси вызывает модуль `fs` на сервере, используя Ajax или веб-сокеты в качестве средства обмена командами и возвращаемыми значениями.

- **Наблюдатель**: шаблон «Наблюдатель» обеспечивает естественную абстракцию между компонентами, генерирующими и получающими события. При кросс-платформенной разработке это означает, что генератор событий можно заменить его реализацией для браузера без ущерба для получателей, и наоборот.
- **DI и локатор служб**: оба шаблона – как DI и локатор служб – способны помочь заменить реализацию модуля в момент его внедрения.

Как видите, в распоряжении разработчика имеется мощный арсенал шаблонов, но самым важным оружием по-прежнему является возможность выбрать лучший подход и адаптировать его к конкретной задаче. В следующем разделе воспользуемся всем, что узнали здесь, использовав несколько уже знакомых нам идей и шаблонов.

Введение в React

Начиная с этого момента, мы будем использовать JavaScript-библиотеку **React** (известную так же, как **ReactJs**), изначально созданную в Facebook (http://facebook.github.io/react/) и обладающую полным набором функций и инструментов для создания уровня представления в приложениях. Библиотека React предлагает абстракцию представлений, основанную на идее компонентов, где компонентом может быть кнопка, поле ввода, простой контейнер, такой как HTML-тег `div` или любой другой элемент пользовательского интерфейса. Идея библиотеки заключается в предоставлении разработчикам возможности создавать пользовательские интерфейсы с помощью простого определения и соединения многократно используемых компонентов, имеющих конкретные обязанности.

От других реализаций библиотека React отличается отсутствием привязки к модели DOM. В самом деле, она обеспечивает высокоуровневую абстракцию, носящую название **виртуальная модель DOM**, которая хорошо совместима с Веб, но также может использоваться в других контекстах, например в мобильных приложениях, для моделирования трехмерных окружений и даже для определения взаимодействий между аппаратными компонентами.

«*Освойте один раз и используйте везде*» (Facebook)

Этот девиз часто используется компанией Facebook во введениях в React. Он намеренно задуман перекликающимся со знаменитым девизом Java: «*Напишите один раз и выполняйте везде*», с явным намерением дистанцироваться от него, заявив, что все контексты различны и нуждаются в собственной конкретной реализации, притом что остается возможность повторно использовать освоенные принципы и инструменты в различных контекстах.

В качестве примеров использования React в контекстах, строго говоря, не относящихся к области веб-разработки, можно назвать следующие проекты:

- **React Native** для мобильных приложений (https://facebook.github.io/react-native);
- **React Three** для создания 3D-сцен (https://github.com/lzzimach/react-three);
- **React Hardware** (https://github.com/iamdustan/react-hardware).

Основная причина интереса к библиотеке React в контексте разработки на универсальном JavaScript – в том, что она позволяет реализовать отображение представлений на стороне сервера или клиента с помощью практически одного и того же кода.

Другими словами, с помощью React можно создать HTML-код страницы на сервере Node.js, а затем, после загрузки страницы, добавить дополнительные интерактивные элементы непосредственно в браузере. Это позволяет создавать одностраничные приложения, где большая часть действий выполняется в браузере и обновляется только та часть страницы, которую требуется изменить. При таком подходе первоначальное представление загружается с сервера, что создает эффект (кажущийся) быстрой загрузки и значительно упрощает индексацию содержимого поисковыми системами.

Стоит также отметить, что виртуальная модель DOM библиотеки React способна оптимизировать отображение изменений. То есть модель DOM не отображается полностью после каждого изменения, вместо этого библиотека React использует хитрый алгоритм, рассчитывающий минимально необходимый объем изменений в DOM для обновления изображения на экране. Это очень эффективный механизм быстрого отображения в браузере, он является еще одной причиной, почему библиотеке React все чаще отдается предпочтение при наличии множества других библиотек и фреймворков.

На этом закончим церемонию представления React и перейдем к конкретным примерам его использования.

Первый компонент React

Начнем эксперименты с React с создания компонента-виджета для отображения списка элементов в окне браузера.

В этом примере будет использоваться несколько инструментов, рассмотренных ранее в этой главе, таких как Webpack и Babel, поэтому, прежде чем приступить к коду, установим все необходимые зависимости:

```
npm install webpack babel-core babel-loader babel-preset-es2015
```

Нам также понадобятся библиотека React и комплект преобразований для Babel, описывающих преобразование кода React в эквивалентный код стандарта ES5:

```
npm install react react-dom babel-preset-react
```

Теперь все готово для создания первого React-компонента, который мы поместим в модуль `src/joyceBooks.js`:

```
const React = require('react');

const books = [
  'Dubliners',
  'A Portrait of the Artist as a Young Man',
  'Exiles and poetry',
  'Ulysses',
  'Finnegans Wake'
];

class JoyceBooks extends React.Component {
  render() {
    return (
      <div>
        <h2>James Joyce's major works</h2>
        <ul className="books">{
          books.map((book, index) =>
```

```
            <li className="book" key={index}>{book}</li>
          )
        }</ul>
      </div>
    );
  }
}

module.exports = JoyceBooks;
```

Начало достаточно тривиально – здесь просто импортируется модуль `React` и определяется массив `books` с названиями книг.

Вторая часть намного интереснее, так как именно она является ядром компонента. Если это ваше первое знакомство с библиотекой React, код, который ее использует, может показаться странным!

Итак, чтобы определить React-компонент, необходимо создать класс, наследующий `React.Component`. Этот класс должен определять функцию `render`, описывающую части модели DOM, за которую отвечает компонент.

Но что же находится внутри функции `render`? Она возвращает некоторую разметку HTML с кодом на JavaScript, и все это даже не заключено в кавычки. Да, это смущает, поскольку это не JavaScript, а JSX!

Что такое JSX?!

Как уже упоминалось ранее, библиотека React – это высокоуровневый программный интерфейс для создания виртуальной модели DOM и управления ею. Сама по себе идея модели DOM великолепна, и эту модель можно без особых проблем представить в формате XML или HTML, но динамическое манипулирование ее деревом, например на уровне узлов, родителей и потомков, очень быстро становится слишком громоздким. Для преодоления данной проблемы библиотека React предоставляет язык JSX как промежуточный формат для описания виртуальной модели DOM и управления ею.

На самом деле JSX не является самостоятельным языком – это лишь надмножество языка JavaScript, которое требуется транскомпилировать в обычный язык JavaScript перед выполнением. Однако эта надстройка позволяет разработчикам использовать XML-синтаксис в коде на JavaScript. При разработке клиентского кода язык JSX используется для описания разметки HTML, определяющей веб-компоненты, как было показано в предыдущем примере. Причем HTML-теги можно размещать непосредственно в JSX-коде, как если бы они являлись частью расширенного синтаксиса языка JavaScript.

Этот подход обеспечивает существенное преимущество, поскольку теперь HTML-код динамически проверяется во время сборки с выдачей сообщений об ошибках, например если забыть закрыть один из тегов.

Теперь рассмотрим функцию `render` из предыдущего примера для анализа некоторых важных черт JSX:

```
render() {
  return (
    <div>
      <h2>James Joyce's major works</h2>
      <ul className="books">{
        books.map((book, index) =>
```

```
          <li className="book" key={index}>{book}</li>
        )
      }</ul>
    </div>
  );
}
```

Как видите, разметку HTML можно вставить в любом месте в JSX-коде, и для этого не требуется как-то выделять его или обертывать. В данном случае мы просто определили тег `div`, действующий как контейнер компонента.

В блок с разметкой HTML можно добавить JavaScript-логику. Обратите внимание на фигурные скобки внутри тега `ul`. Такой подход позволяет определять динамические фрагменты разметки HTML, подобно тому, как это делается во многих движках шаблонов. В данном случае с помощью встроенной функции `map` языка JavaScript выполняется перебор всех книг в массиве, и для каждой создается отдельный фрагмент разметки HTML, добавляющий название книги в список.

Фигурные скобки применяются для определения выражений внутри HTML, и простейшим вариантом их использования является вывод значения переменной, как это делает синтаксис `{book}` в данном примере.

И наконец, обратите внимание на еще один блок HTML внутри кода на JavaScript. Он наглядно показывает, что в описании виртуальной модели DOM допускается смешивать разметку HTML и код на JavaScript как угодно.

В приложениях на основе библиотеки React необязательно использовать JSX. JSX лишь предоставляет удобный интерфейс для работы с виртуальной моделью DOM. Приложив немного усилий, того же результата можно добиться вызовом соответствующих функций, полностью отказавшись от JSX и связанной с ним транскомпиляции. Чтобы получить представление, как выглядит React-код без применения JSX, достаточно посмотреть на транскомпилированную версию функции `render` из того же примера:

```
function render() {
  return React.createElement(
    'div',
    null,
    React.createElement(
      'h2',
      null,
      'James Joyce's major works'
    ),
   React.createElement(
      'ul',
      { className: 'books' },
      books.map(function (book) {
        return React.createElement(
          'li',
          { className: 'book' },
          book
        );
      })
    )
  );
}
```

Как видите, этот код выглядит гораздо менее читабельным, и при его написании легко ошибиться, поэтому имеет смысл применять JSX и использовать транскомпилятор для создания эквивалентного кода на JavaScript.

И в завершение краткого обзора JSX рассмотрим HTML-разметку, получаемую в результате выполнения этого кода:

```
<div data-reactroot="">
  <h2>James Joyce's major works</h2>
    <ul class="books">
    <li class="book">Dubliners</li>
    <li class="book">A Portrait of the Artist as a Young Man</li>
    <li class="book">Exiles and poetry</li>
    <li class="book">Ulysses</li>
    <li class="book">Finnegans Wake</li>
  </ul>
</div>
```

И последнее, что следует отметить при сравнении версий кода JSX/JavaScript, – атрибут `className` был преобразован в атрибут `class`. Важно подчеркнуть, что при работе с виртуальной моделью DOM необходимо использовать атрибуты модели DOM, эквивалентные HTML-атрибутам, которые библиотека React преобразует при выводе HTML-разметки.

Список всех тегов и атрибутов, поддерживаемых библиотекой React, можно найти в официальной документации на странице https://facebook.github.io/react/docs/tags-and-attributes.html.

Более подробные сведения о синтаксисе JSX можно найти в официальной спецификации Facebook на странице https://facebook.github.io/jsx.

Настройка Webpack для транскомпиляции JSX

В этом разделе рассматривается пример конфигурации Webpack для транскомпиляции JSX-кода в понятный браузеру JavaScript-код:

```
const path = require('path');
module.exports = {
  entry: path.join( dirname, "src", "main.js"),
  output: {
    path: path.join( dirname, "dist"),
    filename: "bundle.js"
  },
  module: {
    loaders: [
      {
        test: path.join( dirname, "src"),
        loader: 'babel-loader',
        query: {
          cacheDirectory: 'babel_cache',
          presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

Как видите, эта конфигурация практически идентична той, что была приведена в примере преобразования кода, соответствующего стандарту ES2015, с помощью Webpack. Разница между ними заключаются в следующем:

- в Babel используется комплект преобразований `react`;
- используется параметр `cacheDirectory`, определяющий путь к каталогу, который Babel использует в качестве каталога кэша (в данном случае каталог `babel_cache`), что ускоряет процесс сборки файла пакета. Он является необязательным, но его применение настоятельно рекомендуется, поскольку позволяет ускорить разработку.

Отображение в браузере

Теперь, когда первый React-компонент готов, можно приступить к его использованию и отобразить в браузере. Создадим JavaScript-файл `src/main.js` для использования компонента `JoyceBooks`:

```
const React = require('react');
const ReactDOM = require('react-dom');
const JoyceBooks = require('./joyceBooks');

window.onload = () => {
  ReactDOM.render(<JoyceBooks/>, document.getElementById('main'))
};
```

Наиболее важным здесь является вызов функции `ReactDOM.render`. Эта функция принимает блок JSX-кода и элемент DOM и превращает этот блок в HTML-разметку, которую применяет к узлу DOM, переданному во втором аргументе. Обратите внимание, что блок JSX-кода содержит только пользовательский тег (`JoyceBooks`). После загрузки компонента он будет доступен как JSX-тег (где имя тега совпадает с именем класса компонента), так что новый экземпляр этого компонента можно без труда вставить в другие блоки JSX-кода. Этот базовый механизм позволяет разделить интерфейс на несколько согласованных компонентов.

Последнее, что нужно сделать, чтобы увидеть первый пример использования React в работе, – создать страницу `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Example - James Joyce books</title>
  </head>
  <body>
    <div id="main"></div>
    <script src="dist/bundle.js"></script>
  </body>
</html>
```

Это очень простая разметка HTML, не требующая особых пояснений. Мы лишь добавили файл `bundle.js` в обычную HTML-страницу, содержащую блок `div` с идентификатором `main`, который будет служить контейнером для React-приложения.

Теперь можете выполнить команду `webpack` в терминале и открыть страницу `index.html` в браузере.

Важно понимать, как происходит отображение на стороне клиента, когда пользователь загружает страницу.

1. Браузер загружает и отображает HTML-код страницы.
2. Затем он загружает файл пакета и выполняет JavaScript-код в нем.
3. Этот код динамически создает реальное фактическое содержимое страницы и обновляет модель DOM перед отображением.

Это означает, что если страница загружается браузером с выключенной поддержкой JavaScript (например, поисковым роботом), сайт будет выглядеть как пустая веб-страница, лишенная какого-либо значимого содержимого. Это может стать серьезной проблемой, особенно с точки зрения SEO.

Далее в этой главе мы посмотрим, как отобразить тот же React-компонент на стороне с сервера, чтобы обойти это ограничение.

Библиотека React Router

В этом разделе мы усовершенствуем предыдущий пример, добавив в него очень простую навигацию между несколькими страницами. Приложение будет содержать три раздела: главную страницу, страницу со списком книг Джеймса Джойса (James Joyce) и страницу со списком книг Герберта Уэллса (H. G. Wells). Мы также предусмотрим реакцию на попытку пользователя обратиться к несуществующему URL.

При разработке этого приложения мы используем библиотеку React Router (https://github.com/reactjs/react-router), упрощающую создание компонентов для навигации. Сначала загрузим библиотеку React Router в проект командой:

```
npm install react-router
```

Теперь можно определить компоненты, необходимые для создания разделов нового приложения. Начнем с компонента `src/components/authorsIndex.js`:

```
const React = require('react');
const Link = require('react-router').Link;

const authors = [
  {id: 1, name: 'James Joyce', slug: 'joyce'},
  {id: 2, name: 'Herbert George Wells', slug: 'h-g-wells'}
];

class AuthorsIndex extends React.Component {
  render() {
    return (
      <div>
        <h1>List of authors</h1>
        <ul>{
          authors.map( author =>
            <li key={author.id}><Link to={`/author/${author.slug}`}>
                    {author.name}</Link></li>
          )
        }</ul>
      </div>
    )
  }
}

module.exports = AuthorsIndex;
```

Этот компонент представляет главную страницу приложения. Он выводит имена двух авторов. Обратите внимание, что для простоты выводимые данные хранятся в массиве объектов `authors`, каждый элемент которого содержит сведения об одном авторе. Еще один новый элемент – компонент `Link`. Как можно догадаться, этот компонент принадлежит библиотеке React Router и служит для реализации интерактивных ссылок, которые могут использоваться для перехода к разделам приложения. Важно разобраться в значении свойства `to` компонента `Link`. В нем определяется относительный адрес URI страницы, которую браузер должен отобразить после щелчка на ссылке. То есть данный компонент очень похож на обычный HTML-тег `<a>`. Единственное их различие заключается в том, что вместо перехода на новую страницу с обновлением всей страницы библиотека React Router динамически обновляет только часть страницы, которую необходимо изменить для отображения компонента, связанного с URI-адресом компонента. Более подробно работу этого механизма мы рассмотрим, когда перейдем к конфигурации маршрутизатора. А теперь займемся созданием остальных компонентов приложения. Итак, перепишем компонент `JoyceBooks`, который на этот раз будет храниться в файле `components/joyceBooks.js`:

```
const React = require('react');
const Link = require('react-router').Link;

const books = [
  'Dubliners',
  'A Portrait of the Artist as a Young Man',
  'Exiles and poetry',
  'Ulysses',
  'Finnegans Wake'
];

class JoyceBooks extends React.Component {
  render() {
    return (
      <div>
        <h2>James Joyce's major works</h2>
        <ul className="books">{
          books.map( (book, key) =>
            <li key={key} className="book">{book}</li>
          )
        }</ul>
        <Link to="/">Go back to index</Link>
      </div>
    );
  }
}

module.exports = JoyceBooks;
```

Вполне ожидаемо, что этот компонент очень похож на свою предыдущую версию. Единственными заметными отличиями являются ссылка в конец компонента для возврата к главной странице и использование атрибута `key` в функции `map`. Этим последним изменением мы сообщаем библиотеке React, что каждый элемент идентифицируется уникальным ключом (в данном случае для простоты используется индекс массива), что позволит ей выполнить ряд оптимизаций при каждой повторной

попытке создать список. Это последнее изменение необязательно, но настоятельно рекомендуется, особенно в больших приложениях.

Теперь, следуя той же схеме, определим код следующего компонента `components/wellsBooks.js`:

```
const React = require('react');
const Link = require('react-router').Link;

const books = [
  'The Time Machine',
  'The War of the Worlds',
  'The First Men in the Moon',
  'The Invisible Man'
];

class WellsBooks extends React.Component {
  render() {
    return (
      <div>
        <h2>Herbert George Wells's major works</h2>
        <ul className="books">{
          books.map( (book, key) =>
            <li key={key} className="book">{book}</li>
          )
        }</ul>
        <Link to="/">Go back to index</Link>
      </div>
    );
  }
}

module.exports = WellsBooks;
```

Этот компонент практически идентичен предыдущему и, конечно же, выглядит знакомым! Можно было бы создать более общий компонент `AuthorPage` и избежать дублирования кода, но это станет темой следующего раздела, а здесь просто сосредоточимся на маршрутизации.

Еще нам требуется компонент `components/notFound.js`, отображающий сообщение об ошибке. Он имеет тривиальную реализацию, поэтому мы его опустим ради экономии места.

А теперь перейдем к самой интересной части – к компоненту `routes.js`, определяющему логику маршрутизации:

```
const React = require('react');
const ReactRouter = require('react-router');
const Router = ReactRouter.Router;
const Route = ReactRouter.Route;
const hashHistory = ReactRouter.hashHistory;
const AuthorsIndex = require('./components/authorsIndex');
const JoyceBooks = require('./components/joyceBooks');
const WellsBooks = require('./components/wellsBooks');
const NotFound = require('./components/notFound');

class Routes extends React.Component {
```

```
  render() {
    return (
      <Router history={hashHistory}>
        <Route path="/" component={AuthorsIndex}/>
        <Route path="/author/joyce" component={JoyceBooks}/>
        <Route path="/author/h-g-wells" component={WellsBooks}/>
        <Route path="*" component={NotFound} />
      </Router>
    )
  }
}
module.exports = Routes;
```

Для начала пройдемся по списку модулей, необходимых для реализации компонента маршрутизации. Здесь подключается библиотека `react-router`, включающая три модуля: `Router`, `Route` и `hashHistory`.

Компонент `Router` является основным и содержит всю конфигурацию маршрутизации. Этот элемент мы используем в качестве корневого узла компонента `Route`. Свойство `history` указывает на механизм определения активного маршрута и обновления URL-адреса в строке браузера при любом щелчке на ссылке. Существуют две распространенные стратегии: `hashHistory` и `browserHistory`. Первая использует часть URL-адреса, которая называется **фрагментом** (отделяется символом хеша `#`). При использовании этой стратегии ссылки будут выглядеть так: `index.html#/author/h-g-wells`. Вторая стратегия вместо фрагментов использует интерфейс **history API**, определяемый стандартом HTML5 (https://developer.mozilla.org/ru/docs/Web/API/History_API), что обеспечивает более естественное отображение URL-адресов. В соответствии с этой стратегией каждый путь имеет собственный полный URI-адрес, например http://example.com/author/h-g-wells.

В этом примере используется стратегия `hashHistory`, поскольку она проще в настройке и не требует обращений к веб-серверу для обновления страницы. У нас еще будет возможность рассмотреть стратегию `browserHistory` далее в этой главе.

Компонент `Route` позволяет определить связь между маршрутом (`path`) и компонентом (`component`). То есть указать, какой компонент отображать для данного маршрута.

Все описанные настройки выполняются внутри функции `render`. Теперь, после знакомства с назначением всех компонентов и параметров, перейдем к анализу их работы.

Важно понимать, как компонент `Router` интерпретирует декларативный синтаксис:

- этот компонент выступает в качестве контейнера; он не выводит никакой HTML-разметки, а содержит только список маршрутов `Route`;
- каждый маршрут `Route` связан с компонентом. В этом случае компоненты являются графическими, то есть они выводят HTML-код, но только если текущий URL-адрес страницы соответствует их маршруту;
- только один маршрут может соответствовать заданному адресу URI. В неоднозначных случаях маршрутизатор предпочитает менее общие маршруты (например, он выберет `/author/joyce` вместо `/author`);
- с помощью символа * можно определить **универсальный** маршрут, который будет выбран в случае отклонения все остальных. Здесь он используется для вывода сообщения «страница не найдена»;

- чтобы завершить пример, внесем изменения в модуль `main.js`, использовав в нем компонент `Routes` как основной компонент приложения:

```
const React = require('react');
const ReactDOM = require('react-dom');
const Routes = require('./routes');
window.onload = () => {
  ReactDOM.render(<Routes/>, document.getElementById('main'))
};
```

Теперь осталось запустить Webpack для повторной сборки файла пакета и открыть файл `index.html` в браузере, чтобы проверить работу нового приложения.

Пощелкайте на ссылках, чтобы увидеть, как обновляется URL в адресной строке. Кроме того, с помощью любого инструмента отладки можно убедиться, что переход между разделами не вызывает полного обновления страницы и отправки нового запроса. Фактически, когда открывается главная страница, приложение уже полностью загружено, и маршрутизатор в основном используется для отображения и сокрытия соответствующего компонента с учетом текущего URI-адреса. Во всяком случае, маршрутизатор достаточно умен, чтобы при попытке обновить страницу с конкретным URI-адресом (например, `index.html#/author/joyce`) сразу же отобразить нужный компонент.

React Router – очень мощный компонент, он имеет несколько интересных возможностей. Например, он позволяет определять вложенные маршруты для многоуровневых пользовательских интерфейсов (компоненты с вложенными разделами). Кроме того, в этой главе мы посмотрим, как добавить в него возможность загрузки компонентов и данных по требованию. А теперь сделаем паузу и обратимся к официальной документации этого компонента, чтобы познакомиться со всеми его функциональными возможностями.

Создание приложений на универсальном JavaScript

На текущий момент вы знаете практически все, что необходимо для превращения примера выше в полноценное приложение на универсальном JavaScript. Мы познакомились с Webpack, ReactJs и проанализировали большинство шаблонов, которые помогают унифицировать и дифференцировать межплатформенный код.

В этом разделе мы усовершенствуем предыдущий пример, создав многократно используемые компоненты, добавив универсальную маршрутизацию и отображения, а также извлечение данных.

Создание многократно используемых компонентов

В предыдущем примере мы создали два очень похожих компонента: `JoyceBooks` и `WellsBooks`. Они отличаются только использованием разных данных. А теперь представьте реальное приложение с сотнями или даже тысячами авторов... Нет никакого смысла создавать отдельный компонент для каждого автора.

В этом разделе мы создадим более универсальный компонент и добавим поддержку параметризованных маршрутов.

Начнем с создания универсального компонента `components/authorPage.js`:

```
const React = require('react');
const Link = require('react-router').Link;
```

```
const AUTHORS = require('../authors');

class AuthorPage extends React.Component {
  render() {
    const author = AUTHORS[this.props.params.id];
    return (
      <div>
        <h2>{author.name}'s major works</h2>
        <ul className="books">{
          author.books.map( (book, key) =>
            <li key={key} className="book">{book}</li>
          )
        }</ul>
        <Link to="/">Go back to index</Link>
      </div>
    );
  }
}
module.exports = AuthorPage;
```

Конечно же, этот компонент очень похож на два заменяемых им компонента. Двумя главными отличиями здесь являются поддержка извлечения данных из компонента и способ получения параметра с именем автора, сведения о котором должны быть выведены.

Для простоты мы загружаем модуль `authors.js`, экспортирующий JavaScript-объект с данными об авторах, который используется в качестве простой базы данных. Переменная `this.props.params.id` представляет идентификатор автора, сведения о котором нужно вывести. Этот параметр заполняется маршрутизатором, как будет продемонстрировано чуть ниже. Этот параметр используется для извлечения сведений об авторе из объекта базы данных, после чего у нас имеется все, необходимое для отображения компонента.

Только чтобы показать, как осуществляется выборка данных, ниже приводится исходный код модуля `authors.js`:

```
module.exports = {

  'joyce': {
    'name': 'James Joyce',
    'books': [
      'Dubliners',
      'A Portrait of the Artist as a Young Man',
      'Exiles and poetry',
      'Ulysses',
      'Finnegans Wake'
    ]
  },

  'h-g-wells': {
    'name': 'Herbert George Wells',
    'books': [
      'The Time Machine',
      'The War of the Worlds',
      'The First Men in the Moon',
```

```
      'The Invisible Man'
    ]
  }
};
```

Это очень простой объект, индексирующий авторов с помощью мнемонических строчных идентификаторов.

Теперь сделаем последний шаг и рассмотрим компоненты в модуле `routes.js`:

```
const React = require('react');
const ReactRouter = require('react-router');
const Router = ReactRouter.Router;
const hashHistory = ReactRouter.hashHistory;
const AuthorsIndex = require('./components/authorsIndex');
const AuthorPage = require('./components/authorPage');
const NotFound = require('./components/notFound');

const routesConfig = [
  {path: '/', component: AuthorsIndex},
  {path: '/author/:id', component: AuthorPage},
  {path: '*', component: NotFound}
];

class Routes extends React.Component {
  render() {
    return<Router history={hashHistory} routes={routesConfig}/>;
  }
}
module.exports = Routes;
```

В этот раз мы использовали новый универсальный компонент `AuthorPage` вместо двух конкретных компонентов, которые применялись в предыдущем примере. Здесь также используется альтернативная конфигурация маршрутизатора; на этот раз для определения маршрутов используется обычный массив JavaScript вместо компонентов `Route` в функции `render` компонента `Routes`. Объект из массива помещается в атрибут `routes` компонента `Router`. Эта конфигурация полностью эквивалентна конфигурации из предыдущего примера, основанной на тегах, но выглядит немного проще. В других случаях, например когда имеется множество вложенных маршрутов, конфигурация на основе тегов более удобна. Самым важным изменением здесь является новый маршрут `/author/:id`, связанный с новым универсальным компонентом и заменивший старые конкретные маршруты. Этот маршрут является параметризованным (именованные параметры определяются с использованием двоеточия в качестве префикса) и соответствует двум старым маршрутам: `/author/joyce` и `/author/h-g-wells`. Конечно, он также будет соответствовать любому другому маршрут подобного вида, а строка, соответствующая параметру `id`, будет передаваться непосредственно компоненту, где она будет доступна как `props.params.id`.

Пример готов. Чтобы проверить его, нужно еще раз сгенерировать файл пакета с помощью Webpack и обновить страницу `index.html` в браузере. Эта страница и модуль `main.js` не требуют изменений.

Использование универсальных компонентов и параметризованных маршрутов обеспечивает большую гибкость и позволяет создавать достаточно сложные приложения.

Отображение на стороне сервера

Сделаем еще один шаг вперед в путешествии по универсальному JavaScript. Как уже упоминалось ранее, одной из наиболее интересных черт React является возможность отображения компонентов на стороне сервера. В этом разделе мы реорганизуем приложение так, чтобы оно отображалось непосредственно на сервере.

Здесь в качестве веб-сервера будет использоваться **Express** (http://expressjs.com), а в качестве внутреннего движка шаблонов – **ejs** (https://npmjs.com/package/ejs). Кроме того, перед запуском серверного сценария мы должны обработать его с помощью Babel, чтобы скомпилировать код JSX, поэтому начнем с установки этих новых зависимостей:

```
npm install express ejs babel-cli
```

Все компоненты остаются без изменений, как в предыдущем примере, поэтому сосредоточимся на стороне сервера. На сервере понадобится доступ к настройкам маршрутизации, и чтобы упростить эту задачу, извлечем объект с параметрами маршрутизации из файла `routes.js` и поместим его в отдельный модуль `routesConfig.js`:

```
const AuthorsIndex = require('./components/authorsIndex');
const AuthorPage = require('./components/authorPage');
const NotFound = require('./components/notFound');

const routesConfig = [
  {path: '/', component: AuthorsIndex},
  {path: '/author/:id', component: AuthorPage},
  {path: '*', component: NotFound}
];
module.exports = routesConfig;
```

Кроме того, необходимо преобразовать статический файл `index.html` в ejs-шаблон `views/index.ejs`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Example - Authors archive</title>
  </head>
  <body>
    <div id="main">
      <%- markup -%>
    </div>
    <!--<script src="dist/bundle.js"></script>-->
  </body>
</html>
```

Сделать все это достаточно просто, однако здесь есть две особенности, которые следует пояснить:

- тег `<%-markup-%>` является частью шаблона, который динамически заменит библиотека React на стороне сервера перед передачей страницы браузеру;
- мы закомментировали подключение пакета, потому что в этом разделе основное внимание уделяется отображению на стороне сервера. Полное универсальное решение будет рассмотрено в следующих разделах.

А теперь можно создать сценарий `server.js`:

```
const http = require('http');
const Express = require('express');
const React = require('react');
const ReactDom = require('react-dom/server');
const Router = require('react-router');
const routesConfig = require('./src/routesConfig');

const app = new Express();
const server = new http.Server(app);
app.set('view engine', 'ejs');

app.get('*', (req, res) => {
  Router.match(
    {routes: routesConfig, location: req.url},
    (error, redirectLocation, renderProps) => {
      if (error) {
        res.status(500).send(error.message)
      } else if (redirectLocation) {
        res.redirect(302, redirectLocation.pathname +
          redirectLocation.search)
      } else if (renderProps) {
        const markup = ReactDom.renderToString(<Router.RouterContext
                    {...renderProps} />);
        res.render('index', {markup});
      } else {
        res.status(404).send('Not found')
      }
    }
  );
});

server.listen(3000, (err) => {
  if (err) {
    return console.error(err);
  }
  console.info('Server running on http://localhost:3000');
});
```

Важным фрагментом этого кода является Express-маршрут, определяемый инструкцией `app.get('*', (req, res) => {...})`. Это **универсальный Express-маршрут**, который будет перехватывать все GET-запросы по всем URL-адресам на сервере. В рамках этого маршрута логика маршрутизации делегируется библиотеке React Router, которая была ранее настроена на клиентской стороне приложения.

Шаблон

Здесь серверный компонент маршрутизации (встроенный маршрутизатор Express) заменяет маршрутизатор React Router, способный сопоставлять маршруты, как на клиенте, так и на сервере.

Чтобы задействовать маршрутизатор React Router на сервере, используется функция `Router.match`. Она принимает два параметра: **объект конфигурации** и **функцию обратного вызова**. Объект конфигурации должен иметь два ключа:

- `routes`: используется для передачи конфигурации маршрутов маршрутизатору React Router. Здесь передается точно такая же конфигурация, которая использовалась для отображения на стороне клиента, и именно по этой причине она была выделена в отдельный компонент в начале раздела;
- `location`: используется для сохранения текущего запрашиваемого URL-адреса, который маршрутизатор попытается сопоставить с одним из предопределенных маршрутов.

Функция обратного вызова вызывается при совпадении маршрутов. Она получает три аргумента – `error`, `redirectLocation` и `renderProps`, – которые определяют результат операции сопоставления. Здесь необходимо предусмотреть обработку четырех разных случаев:

- первый случай, когда возникла ошибка при разрешении маршрута. В этой ситуации браузеру просто возвращается ответ **500 internal server error**;
- второй случай, когда совпавший маршрут является перенаправлением на другой маршрут. В этом случае необходимо создать серверное сообщение перенаправления **(302 redirect)**, указывающее браузеру выполнить переход по новому маршруту;
- третий случай соответствует успешному совпадению маршрута, когда требуется отобразить соответствующий компонент. В этом случае в аргументе `renderProps` передается объект с данными, необходимыми для отображения компонента. Он является ядром механизма маршрутизации на стороне сервера, и мы используем функцию `ReactDOM.renderToString` для отображения HTML-разметки, представляющей компонент, соответствующий текущему маршруту. Затем эта HTML-разметка помещается в шаблон `index.ejs`, чтобы получить полную HTML-страницу для передачи браузеру;
- последний случай, когда нет совпадения ни с одним маршрутом, и тогда браузеру просто возвращается ошибка **404 not found**.

Итак, самой важной частью кода является следующая строка:

```
const markup = ReactDom.renderToString(<Router.RouterContext
{...renderProps} />
```

А теперь рассмотрим порядок работы функции `renderToString`.

- Эта функция из модуля `react-dom/serve` способна вывести любой React-компонент в строку. Она используется для отображения HTML-кода на сервере с последующей немедленной его отправкой браузеру, что позволяет уменьшить время загрузки страницы и сделать ее более дружественной к поисковым системам. Библиотека React достаточно умна, чтобы при повторном вызове функции `ReactDOM.render()` для одного и того же компонента не отображать в браузере этот компонент еще раз, а просто подключить обработчики событий к его узлам DOM.
- Отображаемым компонентом является `RouterContext` (из модуля `react-router`), отвечающий за отображение дерева компонентов для данного состояния маршрутизатора. Этому компоненту передается набор атрибутов, все из которых являются полями объекта `renderProps`. С этой целью мы использовали JSX-оператор развертывания атрибутов (https://facebook.github.io/react/docs/jsx-spread.html#spread-attributes), копирующий все пары ключ/значение из объекта в атрибуты компонента.

Теперь запустите сценарий `server.js` командой

```
node server
```

Откройте браузер и перейдите по адресу `http://localhost:3000`, чтобы увидеть, как работает приложение с отображением на сервере.

Напомним еще раз, что в этой версии мы закомментировали инструкцию подключения файла пакета, поэтому клиентский JavaScript-код сейчас не работает и любое действие пользователя приводит к отправке на сервер нового запроса с последующим полным обновлением страницы. А это нежелательно, не так ли?

В следующем разделе мы покажем, как реализовать отображение на клиенте и на сервере, добавив в пример решения для универсальных маршрутизации и отображения.

Универсальные отображение и маршрутизация

В этом разделе мы добавим в пример приложения использование маршрутизации и отображения на обеих сторонах – на серверной и клиентской. Все необходимое для этого уже было рассмотрено по отдельности, осталось просто соединить все вместе.

Начнем с того, что восстановим подключение файла пакета `bundle.js` в основном файле представления (`views/index.ejs`).

Далее необходимо изменить стратегию работы с историей на клиентской стороне (`main.js`). Ранее использовалась стратегия `hashHistory`. Но она не подходит для универсального отображения, поскольку требует совпадения маршрутов на клиенте и на сервере. На сервере можно использовать только стратегию `browserHistory`, поэтому изменим модуль `routes.js`, чтобы применить ее на клиенте:

```
const React = require('react');
const ReactRouter = require('react-router');
const Router = ReactRouter.Router;
const browserHistory = ReactRouter.browserHistory;
const routesConfig = require('./routesConfig');
class Routes extends React.Component {
  render() {
    return<Router history={browserHistory} routes={routesConfig}/>;
  }
}
module.exports = Routes;
```

Как видите, все изменения свелись к подключению функции `ReactRouter.browserHistory` и передаче ее компоненту `Router`.

Почти все готово, осталось лишь добавить небольшие изменения в серверную часть приложения, чтобы передать клиенту файл `bundle.js` с сервера как статический ресурс.

Сделать это можно с помощью функции `Express.static`, позволяющей экспортировать содержимое указанного каталога в виде статических файлов. В данном случае требуется экспортировать каталог `dist`, поэтому просто добавим следующую строку в начало основной конфигурации маршрутизации на сервере:

```
app.use('/dist', Express.static('dist'));
```

И этого вполне достаточно! Теперь, чтобы проверить работу приложения, повторно сгенерируйте файл пакета с помощью Webpack и перезапустите сервер. Затем перей-

дите по адресу `http://localhost:3000`, как и раньше. Внешне все выглядит точно так же, но если использовать инспектор или отладчик, станет ясно, что отображение на сервере происходит лишь при первом запросе, а все следующие запросы обрабатывает браузер. Ради небольшого эксперимента можно принудительно обновить страницу с конкретным URI-адресом, чтобы проверить надлежащую работу маршрутизации на сервере и в браузере.

Универсальное извлечение данных

Пример приложения начал обзаводиться прочным фундаментом, дающим возможность дорасти до законченного и легко масштабируемого приложения. Однако остается еще один важный аспект, требующий нашего внимания, – извлечение данных. В примере приложения используется модуль с данными в формате JSON. В настоящее время он используется как база данных, но, конечно же, этот подход нельзя счесть оптимальным в силу следующих причин:

- файл JSON используется повсюду в приложении, и доступ к данным осуществляется напрямую на клиенте, на сервере и в любом React-компоненте;
- учитывая, что данные необходимы также на стороне клиента, мы вынуждены включить в клиентский пакет полную базу данных. Это рискованно, поскольку такой подход открывает доступ к любой конфиденциальной информации. Кроме того, файл пакета будет расти по мере увеличения размера базы данных, и его потребуется повторно компилировать после любого изменения данных.

Очевидно, что требуется другое решение, обеспечивающее большую изолированность и масштабируемость.

В этом разделе мы усовершенствуем пример, создав REST-службу, что позволит запрашивать данные асинхронно, когда это действительно необходимо, и только определенное подмножество данных, необходимое для отображения текущего раздела приложения.

REST-служба

Служба должна быть полностью независима от серверной стороны приложения, поскольку, в идеале, должна иметься возможность масштабировать эту службу независимо от остальной части приложения.

Перейдем непосредственно к коду в модуле `apiServer.js`:

```
const http = require('http');
const Express = require('express');

const app = new Express();
const server = new http.Server(app);
const AUTHORS = require('./src/authors');                // [1]

app.use((req, res, next) => {                            // [2]
  console.log(`Received request: ${req.method} ${req.url} from
    ${req.headers['user-agent']}`);
  next();
});

app.get('/authors', (req, res, next) => {                // [3]
  const data = Object.keys(AUTHORS).map(id => {
    return {
```

```
      'id': id,
      'name': AUTHORS[id].name
    };
  });

  res.json(data);
});

app.get('/authors/:id', (req, res, next) => {          // [4]
  if (!AUTHORS.hasOwnProperty(req.params.id)) {
    return next();
  }

  const data = AUTHORS[req.params.id];
  res.json(data);
});

server.listen(3001, (err) => {
  if (err) {
    return console.error(err);
  }
  console.info('API Server running on http://localhost:3001');
});
```

Как видите, здесь в качестве серверного фреймворка снова используется Express. Ниже приводятся пояснения к наиболее важным участкам в коде.

- Данные все еще находятся в модуле, представляющем собой файл в формате JSON (`src/authors.js`). Это, конечно же, сделано только ради упрощения примера, поскольку в реальном приложении данный способ следует заменить настоящей базой данных, такой как MongoDB, MySql или LevelDB. В этом примере данные извлекаются непосредственно из JSON-объекта, тогда как в реальном приложении для получения данных выполняются запросы к внешнему источнику.
- Здесь используется промежуточное программное обеспечение, выводящее в консоль определенную полезную информацию при получении каждого запроса. Позже вы увидите, как это поможет понять, кто обратился к службе (клиентская и серверная части), и убедиться в правильности работы приложения.
- Мы экспортируем конечную точку с URI `/authors`, которая в ответ на GET-запрос возвращает массив в формате JSON, содержащий сведения обо всех доступных авторах. Для каждого автора определены поля `id` и `name`. И снова извлечение данных осуществляется непосредственно из файла JSON, используемого в качестве базы данных; в реальной ситуации предпочтительнее выполнять запрос к настоящей базе данных.
- Мы также экспортируем еще одну конечную точку с URI `/authors/:id`, где `:id` – универсальный заполнитель, соответствующий идентификатору конкретного автора. Если GET-запрос к конечной точке содержит допустимый идентификатор (для которого имеется запись в JSON-файле), служба вернет объект, содержащий имя автора и массив его книг.

Теперь запустим REST-службу командой

```
node apiServer
```

Она будет доступна по адресу `http://localhost:3001`, и для проверки ее работы можно выполнить несколько запросов с помощью `curl`:

```
curl http://localhost:3001/authors/
[{"id":"joyce","name":"James  Joyce"},{"id":"h-g-wells","name":"Herbert George Wells"}]

curl http://localhost:3001/authors/h-g-wells
{"name":"Herbert George Wells","books":["The Time Machine","The War of the Worlds","The First Men in
the Moon","The Invisible Man"]}
```

Выполнение запросов клиента через прокси

Созданная выше служба должна быть доступна для обеих частей приложения – клиентской и серверной. Клиент будет обращаться к ней посредством AJAX-запросов. Возможно, вы знакомы с ограничением, позволяющим браузеру выполнять AJAX-запросы только к URL-адресам домена, откуда была загружена страница. Это означает, что если служба будет принимать запросы по адресу `localhost:3001`, а веб-сервер – по адресу `localhost:3000`, они будут считаться принадлежащими разным доменам, и браузер не сможет напрямую обратиться к службе. Для обхода этого ограничения нужно создать прокси на веб-сервере, который даст возможность обращаться к конечным точкам службы через удобный внутренний маршрут (`localhost:3000/api`), как это показано на рис. 8.1.

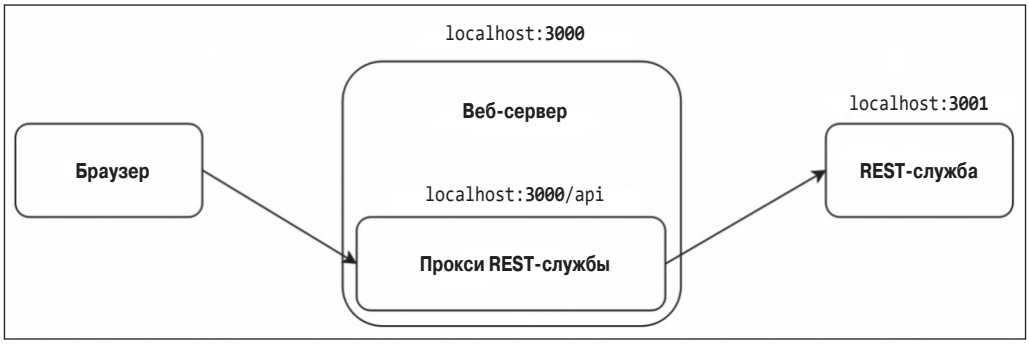

Рис. 8.1 ❖ Схема доступа к службе через прокси

Для создания прокси-компонента на веб-сервере будет использоваться замечательный модуль `http-proxy` (https://npmjs.com/package/http-proxy), поэтому установим его с помощью npm:

```
npm install http-proxy
```

Чуть ниже мы увидим, как подключить его к веб-серверу и настроить.

Универсальный клиент службы

Служба будет вызываться с использованием двух разных адресов, с учетом текущей конфигурации:

- `http://localhost:3001` – со стороны веб-сервера;
- `/api` – со стороны браузера.

Кроме того, необходимо учитывать, что для асинхронных HTTP-запросов в браузере используется механизм XHR/AJAX, а на сервере – библиотека `request` или встроенная библиотека `http`.

Для обхода этих различий и создания модуля универсального клиента службы можно воспользоваться библиотекой `axios` (https://npmjs.com/package/axios). Эта библиотека работает как на клиенте, так и на сервере и абстрагирует два различных механизма выполнения HTTP-запросов за единым универсальным программным интерфейсом.

Итак, установим библиотеку `axios` командой

```
npm install axios
```

Затем создадим простой модуль-обертку, который экспортирует настроенный экземпляр `axios`. Назовем этот модуль `xhrClient.js`:

```
const Axios = require('axios');

const baseURL = typeof window !== 'undefined' ? '/api' :
  'http://localhost:3001';
const xhrClient = Axios.create({baseURL});
module.exports = xhrClient;
```

Проверяя наличие переменной `window`, этот модуль выясняет, где выполняется код, – в браузере или на веб-сервере – и выбирает адрес службы. Затем мы просто экспортируем новый экземпляр клиента `axios`, инициализированный текущим значениеь базового URL.

Теперь можно импортировать этот модуль в React-компоненты и независимо от того, выполняются они на сервере или в браузере, использовать один и тот же универсальный интерфейс, скрывающий все различия двух окружений.

Другими популярными универсальными HTTP-клиентами являются `superagent` (https://npmjs.com/package/superagent) и `isomorphic-fetch` (https://npmjs.com/package/isomorphic-fetch).

Асинхронные React-компоненты

Теперь, когда компоненты будут использовать этот новый набор программных интерфейсов, их нужно асинхронно инициализировать. Для этого потребуется расширение маршрутизатора React Router: `async-props` (https://npmjs.com/package/async-props).

Установим этот модуль командой

```
npm install async-props
```

Теперь все готово для преобразования компонентов в асинхронные. Начнем с модуля `components/authorsIndex.js`:

```
const React = require('react');
const Link = require('react-router').Link;
const xhrClient = require('../xhrClient');

class AuthorsIndex extends React.Component {
  static loadProps(context, cb) {
    xhrClient.get('authors')
      .then(response => {
        const authors = response.data;
        cb(null, {authors});
      })
      .catch(error => cb(error))
```

```
    ;
  }

  render() {
    return (
      <div>
        <h1>List of authors</h1>
        <ul>{
          this.props.authors.map(author =>
            <li key={author.id}>
              <Link to={`/author/${author.id}`}>{author.name}</Link>
            </li>
          )
        }</ul>
      </div>
    )
  }
}
module.exports = AuthorsIndex;
```

Как видите, в новой версии мы подключаем новый модуль `xhrClient` вместо старого, содержащего данные в формате JSON. Затем добавляем в класс компонента новый метод `loadProps`. Этот метод принимает объект с некоторыми параметрами контекста (`context`), которые будут переданы маршрутизатору, и функцию обратного вызова (`cb`). В этом методе можно выполнить любые асинхронные действия, необходимые для получения данных, требуемых для инициализации компонента. Когда все будет загружено (или если возникнет ошибка), выполняется функция обратного вызова, которая передаст данные дальше и уведомит маршрутизатор о готовности компонента. В данном случае для извлечения сведений об авторах используется `xhrClient`.

Аналогично обновим компонент `components/authorPage.js`:

```
const React = require('react');
const Link = require('react-router').Link;
const xhrClient = require('../xhrClient');

class AuthorPage extends React.Component {
  static loadProps(context, cb) {
    xhrClient.get(`authors/${context.params.id}`)
      .then(response => {
        const author = response.data;
        cb(null, {author});
      })
      .catch(error => cb(error))
    ;
  }

  render() {
    return (
      <div>
        <h2>{this.props.author.name}'s major works</h2>
        <ul className="books">{
          this.props.author.books.map( (book, key) =>
            <li key={key} className="book">{book}</li>
```

```
          )
        }</ul>
        <Link to="/">Go back to index</Link>
      </div>
    );
  }
}
module.exports = AuthorPage;
```

Здесь код реализует ту же логику, что и в предыдущем компоненте. Главное отличие заключается в вызове конечной точки `authors/:id` службы и извлечении идентификатора из переменной `context.params.id` для передачи маршрутизатору.

Для корректной загрузки асинхронных компонентов необходимо также обновить определение маршрутизатора на обеих сторонах – клиенте и сервере. Начнем с клиента и рассмотрим новую версию модуля `routes.js`:

```
const React = require('react');
const AsyncProps = require('async-props').default;
const ReactRouter = require('react-router');
const Router = ReactRouter.Router;
const browserHistory = ReactRouter.browserHistory;
const routesConfig = require('./routesConfig');

class Routes extends React.Component {
  render() {
    return <Router
      history={browserHistory}
      routes={routesConfig}
      render={(props) => <AsyncProps {...props}/>}
    />;
  }
}
module.exports = Routes;
```

Два отличия от предыдущей версии заключаются в подключении модуля `async-props` и использовании его в функции `render` компонента `Router`. Этот подход фактически внедряет логику модуля `async-props` в логику отображения маршрутизатора, обеспечивая поддержку асинхронной модели поведения.

Веб-сервер

И наконец, нам нужно реорганизовать веб-сервер так, чтобы он использовал прокси для перенаправления запросов к службе и вызывал маршрутизатор с помощью модуля `async-props`.

Переименуем модуль `server.js` в `webServer.js`, чтобы четко отличать его от файла с реализацией службы. Вот содержимое нового файла:

```
const http = require('http');
const Express = require('express');
const httpProxy = require('http-proxy');
const React = require('react');
const AsyncProps = require('async-props').default;
const loadPropsOnServer = AsyncProps.loadPropsOnServer;
const ReactDom = require('react-dom/server');
```

```
const Router = require('react-router');
const routesConfig = require('./src/routesConfig');

const app = new Express();
const server = new http.Server(app);

const proxy = httpProxy.createProxyServer({
  target: 'http://localhost:3001'
});

app.set('view engine', 'ejs');
app.use('/dist', Express.static('dist'));
app.use('/api', (req, res) => {
  proxy.web(req, res, {target: targetUrl});
});

app.get('*', (req, res) => {
  Router.match({routes: routesConfig, location: req.url}, (error,
    redirectLocation, renderProps) => {
    if (error) {
      res.status(500).send(error.message)
    } else if (redirectLocation) {
      res.redirect(302, redirectLocation.pathname +
        redirectLocation.search)
    } else if (renderProps) {
      loadPropsOnServer(renderProps, {}, (err, asyncProps, scriptTag) => {
        const markup = ReactDom.renderToString(<AsyncProps {...renderProps}
          {...asyncProps} />);
        res.render('index', {markup, scriptTag});
      });
    } else {
      res.status(404).send('Not found')
    }
  });
});

server.listen(3000, (err) => {
  if (err) {
    return console.error(err);
  }
  console.info('WebServer running on http://localhost:3000');
});
```

Поясним все изменения относительно предыдущей версии по одному.

- Прежде всего импортируется несколько новых модулей: `http-proxy` и `async-props`.
- Инициализируется экземпляр `proxy` и добавляется в веб-сервер через промежуточное программное обеспечение для обработки запросов к адресу `/api`.
- Изменилась логика отображения на стороне сервера. На этот раз нельзя напрямую вызвать функцию `renderToString`, поскольку для этого необходимо обеспечить загрузку всех асинхронных данных. Модуль `async-props` предлагает функцию `loadPropsOnServer` для этой цели. Она выполняет асинхронную загрузку данных из текущего компонента маршрута. По завершении загрузки вызывается функция обратного вызова, и только в теле этой функции можно безопасно

вызывать метод `renderToString`. Кроме того, обратите внимание, что на этот раз вместо `RouterContext` отображается компонент `AsyncProps`, которому передается набор синхронных и асинхронных атрибутов с помощью JSX-синтаксиса развертывания. Еще одной важной особенностью является передача аргумента `scriptTag` в функцию обратного вызова. Он хранит некоторый JavaScript-код, который необходимо поместить в HTML-разметку. Этот код будет содержать представление данных, загружаемых асинхронно в процессе отображения на стороне сервера, что позволит браузеру получить прямой доступ к этим данным и сделает ненужным повторный запрос службы. Чтобы поместить этот сценарий в окончательную HTML-разметку, он передается представлению вместе с разметкой, полученной в процессе отображения компонента.

Шаблон `views/index.ejs` также нужно немного изменить, чтобы отобразить переменную `scriptTag`, упоминавшуюся выше:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>React Example - Authors archive</title>
  </head>
  <body>
    <div id="main"><%- markup %></div>
    <script src="/dist/bundle.js"></script>
    <%- scriptTag %>
  </body>
</html>
```

Как видите, переменная `scriptTag` добавлена в конец тела страницы.

Теперь практически все готово для проверки этого примера, осталось только заново сгенерировать пакет с помощью Webpack и запустить веб-сервер командой

```
babel-cli server.js
```

Теперь можно открыть браузер и перейти по адресу `http://localhost:3000`. И снова вы не увидите никаких внешних отличий, но внутренне все будет работать совершенно иначе. Чтобы убедиться в этом, воспользуйтесь инспектором или отладчиком браузера и посмотрите, как выполняется запрос к службе. Можно также заглянуть в консоль, где запускалась служба, чтобы прочитать сообщения и понять, кто и когда послал запрос.

Итоги

Эта глава была посвящена знакомству с быстро прогрессирующим универсальным JavaScript. Универсальный JavaScript открывает массу новых возможностей для веб-разработки, но его применение все еще остается новой и малоосвоенной областью.

В этой главе мы знакомились с основами этой темы, такими как компонентно-ориентированные пользовательские интерфейсы, универсальное отображение, универсальная маршрутизация и универсальное извлечение данных. Попутно мы создали очень простое приложение, демонстрирующее объединение все этих идей. Также мы добавили в свой арсенал комплекс новых, мощных инструментов и библиотек, таких как Webpack и React.

Несмотря на большое разнообразие рассмотренных вопросов, мы лишь поверхностно коснулись этой обширной темы, но если она вас заинтересовала, вы можете самостоятельно продолжить ее изучение. С учетом сырого на настоящий момент состояния инструментов и библиотек, вероятно, многое в них изменится в течение ближайших несколько лет, но основные идеи сохранятся, так что имеет смысл освоить и попробовать использовать их. Затем, чтобы стать экспертом в этой области, нужно использовать приобретенные знания для создания первого реального бизнес-приложения.

Также стоит подчеркнуть, что полученные здесь знания могут пригодиться не только для создания веб-приложений, но также, например, для разработки мобильных приложений. Если вам интересна эта область, хорошей отправной точкой может стать React Native.

В следующей главе мы продолжим исследование шаблонов разработки асинхронного кода и познакомимся с примерами решения таких конкретных задач, как асинхронная инициализация модулей, асинхронная пакетная обработка и кэширование. Вы готовы к знакомству с более сложными и интересными темами?

Глава 9

Дополнительные рецепты асинхронной обработки

Почти все шаблоны проектирования, рассматривавшиеся выше, можно считать универсальными и применимыми в различных прикладных областях. Но существует также ряд шаблонов, более конкретных и ориентированных на решение четко очерченных проблем, их можно назвать шаблонами *рецептов*. Подобно кулинарным рецептам, они описывают последовательность действий, выполнение которых приводит к нужному результату. Конечно, это не означает, что не нужно подходить творчески к рецептам, например чтобы учесть вкусы наших гостей, но следовать общему плану здесь необходимо. В этой главе будет представлено несколько популярных рецептов решения конкретных проблем, с которыми приходится ежедневно сталкиваться при разработке на платформе Node.js. Вот краткий перечень этих рецептов:

- подключение модулей, инициализируемых асинхронно;
- пакетная обработка и кэширование асинхронных операций, чтобы получить прирост производительности в высоконагруженных приложениях с минимальными затратами времени на разработку;
- выполнение синхронных вычислительных операций, способных блокировать цикл событий и лишить Node.js возможности параллельно обрабатывать запросы.

Подключение модулей, инициализируемых асинхронно

В *главе 2 «Основные шаблоны Node.js»* при обсуждении основных свойств системы модулей в Node.js упоминалось, что функция `require()` работает синхронно, а атрибут `module.exports` не может устанавливаться асинхронно.

Это одна из основных причин существования синхронного программного интерфейса в модулях ядра и множестве npm-пакетов, который в большей степени служит удобной альтернативой для инициализации, а не как замена асинхронного интерфейса.

К сожалению, синхронные программные интерфейсы не всегда доступны, особенно это характерно для компонентов, производящих сетевые операции во время инициализации, например для установления соединений или для извлечения параметров конфигурации. Это относится ко многим драйверам баз данных и клиентам промежуточных систем, таких как очереди сообщений.

Канонические решения

В качестве примера рассмотрим модуль db, подключающийся к удаленной базе данных. Модуль db сможет принимать запросы только после завершения подключения и согласования порядка работы с сервером. В этой ситуации возможны два решения.

- перед тем как начинать использовать модуль, следует удостовериться, что он инициализирован, в противном случае – дождаться его инициализации. Этот процесс должен повторяться перед каждой операцией в асинхронном модуле:

```
const db = require('aDb'); //Асинхронный модуль
module.exports = function findAll(type, callback) {
  if(db.connected) { //инициализирован?
    runFind();
  } else {
    db.once('connected', runFind);
  }
  function runFind() {
    db.findAll(type, callback);
  });
};
```

- использовать прием **внедрения зависимостей (DI)** вместо непосредственной загрузки асинхронного модуля. При этом имеется возможность задержать инициализацию некоторых модулей, пока их асинхронные зависимости не будут полностью инициализированы. Этот способ перекладывает сложности, связанные с управлением инициализацией, на другой компонент, которым обычно является родительский модуль. В следующем примере таким компонентом является app.js:

```
//в модуле app.js
const db = require('aDb'); //Асинхронный модуль
const findAllFactory = require('./findAll');
db.on('connected', function() {
  const findAll = findAllFactory(db);
});
```

```
//в модуле findAll.js
module.exports = db => {
//модуль db гарантированно инициализуется
  return function findAll(type, callback) {
    db.findAll(type, callback);
  }
}
```

Как видите, использование первого варианта нежелательно из-за присутствия в нем достаточно большого количества типового кода.

Второй вариант, использующий DI, также часто не стоит применять по причинам, описанным в *главе 7* «*Связывание модулей*». Его использование в крупных проектах может стать чрезмерно сложным, особенно если реализуется вручную и с помощью асинхронной инициализации модулей. Эти проблемы решаются с использованием DI-контейнеров, поддерживающих асинхронную инициализацию модулей.

Но существует третий вариант, избавляющий модули от необходимости проверять инициализацию их зависимостей.

Очереди на инициализацию

Простой шаблон, избавляющий модули от необходимости проверять инициализацию их зависимостей, основан на использовании очередей и шаблона «Команда». Его идея заключается в том, чтобы сохранить все операции, полученные модулем, пока он еще не инициализирован, и выполнить их после окончания инициализации.

Реализация асинхронно инициализируемого модуля

Для демонстрации этого простого, но эффективного приема создадим небольшое тестовое приложение, не представляющее собой ничего особенного и предназначенное только для проверки описанной идеи. Начнем с создания модуля `asyncModule.js`, инициализируемого асинхронно:

```
const asyncModule = module.exports;
asyncModule.initialized = false;
asyncModule.initialize = callback => {
  setTimeout(function() {
    asyncModule.initialized = true;
    callback();
  }, 10000);
};

asyncModule.tellMeSomething = callback => {
  process.nextTick(() => {
    if(!asyncModule.initialized) {
      return callback(
        new Error('I don't have anything to say right now')
      );
    }
    callback(null, 'Current time is: ' + new Date());
  });
};
```

В этом примере демонстрируется порядок асинхронной инициализации модуля `asyncModule`. Модуль экспортирует метод `initialize()`, который после 10-секундной задержки присваивает переменной `initialized` значение `true` и вызывает функцию обратного вызова (10 секунд – слишком много для реального приложения, но такой длинный интервал поможет нам выявить любые состояния гонки). Другой метод, `tellMeSomething()`, возвращает текущее время, но если модуль еще не инициализирован, он возбуждает исключение.

Далее создадим еще один модуль, зависящий от только что созданной службы. Поместим в этот модуль, в файле `routes.js`, простой обработчик HTTP-запросов:

```
const asyncModule = require('./asyncModule');
module.exports.say = (req, res) => {
  asyncModule.tellMeSomething((err, something) => {
    if(err) {
      res.writeHead(500);
      return res.end('Error:' + err.message);
    }
    res.writeHead(200);
    res.end('I say: ' + something);
  });
};
```

Обработчик вызывает метод `tellMeSomething()` модуля `asyncModule` с последующей записью результата в HTTP-ответ. Как видите, здесь отсутствует какая-либо проверка инициализации модуля `asyncModule`, что, скорее всего, приведет к возникновению проблем.

А теперь создадим очень простой HTTP-сервер, не используя ничего, кроме модуля `http` (файл `app.js`):

```
const http = require('http');
const routes = require('./routes');
const asyncModule = require('./asyncModule');

asyncModule.initialize(() => {
  console.log('Async module initialized');
});

http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/say') {
    return routes.say(req, res);
  }
  res.writeHead(404);
  res.end('Not found');
}).listen(8000, () => console.log('Started'));
```

Этот небольшой модуль служит точкой входа в приложение, запускает инициализацию модуля `asyncModule` и создает HTTP-сервер для обработки созданного ранее запроса (`routes.say()`).

Теперь можно опробовать сервер, запустив модуль `app.js` обычным способом. После запуска сервера перейдем в браузере по адресу `http://localhost:8000/` и посмотрим, что вернет модуль `asyncModule`.

Как и ожидалось, если отправить запрос сразу после запуска сервера, в ответ будет возвращена следующая ошибка:

```
Error:I don't have anything to say right now
```

Это означает, что модуль `asyncModule` еще не инициализирован на момент попытки использовать его. В зависимости от деталей реализации модуля, инициализируемого асинхронно, ошибка в такой ситуации может повлечь вывод сообщения, потерю важной информации или даже аварийное завершение всего приложения. В целом описанной ситуации всегда и везде следует избегать. Обычно нескольким неудачным запросам не уделяется особого внимания, или инициализация выполняется настолько быстро, что этого практически никогда не происходит. Однако в высоконагруженных приложениях и облачных серверах, поддерживающих *автоматическое масштабирование*, такая беспечность недопустима.

Обертывание модулей очередями на инициализацию

Чтобы увеличить надежность сервера, применим к нему шаблон, описанный в начале раздела. Будем помещать в очередь все операции, запрошенные у модуля `asyncModule`, если он еще не инициализирован, и извлечем их из очереди, когда модуль будет готов их обработать. Для этого прекрасно подойдет шаблон «Состояние»! Нам понадобятся два состояния: одно – для постановки в очередь всех операций, пока модуль не инициализирован, и второе – для передачи операций оригинальному модулю `asyncModule` после завершения его инициализации.

Обычно отсутствует возможность изменить код асинхронного модуля, поэтому, чтобы добавить поддержку очереди, нужно создать прокси-объект, обертывающий оригинальный модуль `asyncModule`.

Начнем работу с создания нового файла `asyncModuleWrapper.js`, который будет разрабатываться по частям. Сначала создадим объект, передающий операции активному модулю `activeState`:

```
const asyncModule = require('./asyncModule');

const asyncModuleWrapper = module.exports;

asyncModuleWrapper.initialized = false;
asyncModuleWrapper.initialize = () => {
  activeState.initialize.apply(activeState, arguments);
};

asyncModuleWrapper.tellMeSomething = () => {
  activeState.tellMeSomething.apply(activeState, arguments);
};
```

Здесь объект `asyncModuleWrapper` просто делегирует вызовы своих методов текущему объекту активному модулю `activeState`. Теперь посмотрим, как выглядят два состояния, начав с `notInitializedState`:

```
const pending = [];
const notInitializedState = {

  initialize: function(callback) {
    asyncModule.initialize(() => {
      asyncModuleWrapper.initalized = true;
      activeState = initializedState;                //[1]

      pending.forEach(req => {                       //[2]
        asyncModule[req.method].apply(null, req.args);
      });
      pending = [];

      callback();                                    //[3]
    });
  },

  tellMeSomething: callback => {
    return pending.push({
      method: 'tellMeSomething',
      args: arguments
    });
  }
};
```

Вызов метода `initialize()` запускает инициализацию оригинального модуля `asyncModule`, передавая метод прокси-объекта как функцию обратного вызова. Это позволяет обертке узнать, когда завершится инициализация оригинального модуля, и выполнить следующие операции:

1) присвоить переменной `activeState` объект следующего состояния, в данном случае `initializedState`;

2) выполнить все команды, хранящиеся в очереди;
3) вызвать оригинальную функцию обратного вызова.

Если модуль еще не инициализирован, метод `tellMeSomething()` этого объекта состояния просто создаст новый объект команды и добавит его в очередь `pending` операций, ожидающих выполнения.

На этом этапе идея шаблона должна выглядеть очевидной: когда оригинальный модуль `asyncModule` еще не инициализирован, обертка просто ставит в очередь все полученные запросы. Затем, получив уведомление о завершении инициализации, она выполняет все операции из очереди и потом переключает внутреннее состояние в `initializedState`. Этот последний фрагмент обертки выглядит следующим образом:

```
let initializedState = asyncModule;
```

Сюрприз в том, что объект `initializedState` является просто ссылкой на оригинальный модуль `asyncModule`! В самом деле, после инициализации можно без опаски направлять любые запросы непосредственно оригинальному модулю. А больше ничего и не требуется.

И наконец, следует установить начальный активный модуль, которым, конечно же, будет `notInitializedState`:

```
let activeState = notInitializedState;
```

Теперь еще раз запустим тестовый сервер, но перед этим заменим ссылки на оригинальный модуль `asyncModule` ссылками на новый объект `asyncModuleWrapper` в модулях `app.js` и `routes.js`.

После этого все попытки отправить запрос серверу в тот момент, когда модуль `asyncModule` еще не инициализирован, не будут терпеть неудачу; вместо этого запросы будут задержаны до завершения инициализации и только потом фактически обработаны. Очевидно, что такая модель поведения намного надежнее.

Шаблон
Если модуль инициализируется асинхронно, операции помещаются в очередь, пока инициализация не завершится.

Теперь сервер сможет начать принимать запросы сразу после запуска и гарантировать, что ни один не будет потерян из-за того, что модули еще не завершили инициализацию. Мы достигли желаемого без использования DI и сложных проверок состояния асинхронного модуля, чреватых ошибками.

Реальное применение

Рассмотренный выше шаблон используют многие драйверы баз данных и ORM-библиотеки. Наиболее примечательным из них является Mongoose (http://mongoosejs.com) – ORM-библиотека для **MongoDB**. При использовании библиотеки Mongoose не нужно ждать подключения к базе данных, перед тем как начать отправлять запросы, поскольку любая операция будет помещена в очередь и выполнена после установки соединения с базой данных. Очевидно, что это делает использование программного интерфейса гораздо более удобным.

Если заглянуть в исходный код Mongoose, можно увидеть, что она проксирует все методы драйвера, организуя очередь для хранения операций, выполнение которых было запрошено до завершения инициализации (демонстрируя альтернативный

подход к реализации данного шаблона). Фрагмент кода, отвечающий за реализацию шаблона, можно найти на странице https://github.com/LearnBoost/mongoose/blob/21f16c62e2f3230fe616745a40f22b4385a11b11/lib/drivers/node-mongodb-native/collection.js#L103-138.

Группировка асинхронных операций и кэширование

В высоконагруженных приложениях **кэширование** играет важнейшую роль и применяется в Интернете практически повсеместно, начиная от статических ресурсов, таких как веб-страницы, изображения и таблицы стилей, до динамических данных, например результатов запросов к базе данных. В этом разделе вы узнаете, как применять кэширование к асинхронным операциям и как добиться высокой пропускной способности при обработке запросов.

Реализация сервера без кэширования и группировки операций

Прежде чем углубиться в решение этой новой проблемы, создадим небольшой демонстрационный сервер, который будет служить эталоном для оценки эффективности последующих методов.

Реализуемый веб-сервер предназначен для использования в компании, занимающейся электронной коммерцией, в частности он должен обрабатывать запросы на получение итоговой суммы всех сделок с определенным видом товара. Здесь вновь будет использоваться модуль LevelUP благодаря его простоте и гибкости. Данные будут возвращаться в виде простого списка сделок, хранящихся в подуровне `sales` (раздел базы данных), имеющего следующий формат:

```
transactionId {amount, item}
```

Ключом служит идентификатор сделки `transactionId`, а значением – JSON-объект с суммой продаж (`amount`) и вида товара (`item`).

Обрабатываемые данные предельно просты, так что сразу перейдем к реализации, поместив ее в файл `totalSales.js`:

```
const level = require('level');
const sublevel = require('level-sublevel');
const db = sublevel(level('example-db', {valueEncoding: 'json'}));
const salesDb = db.sublevel('sales');

module.exports = function totalSales(item, callback) {
  console.log('totalSales() invoked');
  let sum = 0;
  salesDb.createValueStream()            // [1]
    .on('data', data => {
      if(!item || data.item === item) {  // [2]
        sum += data.amount;
      }
    })
    .on('end', () => {
      callback(null, sum);               // [3]
  });
};
```

Ядром модуля является единственная экспортируемая функция `totalSales`, которая функционирует следующим образом:

1) создается поток данных из подуровня `salesDb`, содержащий сделки. В поток включаются все записи соответствующего вида, имеющиеся в базе данных;
2) событие `data` возникает при извлечении каждой сделки из базы данных. Сумма текущей записи прибавляется к итоговой сумме `sum`, только если вид товара `item` совпадает с указанным (или если вид товара не указан, то есть когда требуется вычислить сумму всех сделок, независимо от вида товара);
3) и наконец, по событию `end` вызывается метод `callback()`, которому передается итоговая сумма `sum` в качестве результата.

Используемый здесь простой запрос – безусловно, не самое оптимальное решение с точки зрения производительности. В реальном приложении следовало бы применить индексацию по виду товара `item` или, что еще лучше, пошаговое отображение/свертку (map/reduce) для вычисления итоговой суммы в режиме реального времени. Однако для целей этого примера низкая эффективность запроса является достоинством, поскольку позволит продемонстрировать преимущества обсуждаемых шаблонов.

Для завершения приложения *расчета итога продаж* осталось только реализовать программный интерфейс `totalSales` в HTTP-сервере, поэтому следующим шагом станет его создание (файл `app.js`):

```
const http = require('http');
const url = require('url');
const totalSales = require('./totalSales');

http.createServer((req, res) => {
  const query = url.parse(req.url, true).query;
  totalSales(query.item, (err, sum) => {
    res.writeHead(200);
    res.end(`Total sales for item ${query.item} is ${sum}`);
  });
}).listen(8000, () => console.log('Started'));
```

Вновь созданный сервер предельно прост и поддерживает только программный интерфейс `totalSales`.

Перед первым запуском сервера необходимо заполнить базу данных с примерами данных. Это можно сделать с помощью сценария `populate_db.js`, доступного в примерах кода к этому разделу. Сценарий создаст в базе данных 100 K случайных сделок.

Отлично! Теперь все готово. Чтобы запустить сервер, нужно, как обычно, выполнить следующую команду:

```
node app
```

Чтобы послать запрос серверу, перейдите в браузере по адресу:

```
http://localhost:8000?item=book
```

Но для оценки производительности сервера потребуется выполнить несколько запросов. Осуществить это поможет небольшой сценарий `loadTest.js` из примеров кода к книге, посылающий запросы каждые 200 мс. Этот сценарий уже настроен на подключение к URL-адресу сервера, поэтому он запускается простой командой:

```
node loadTest
```

Как видите, выполнение 20 запросов занимает определенное время. Запишите время, затраченное на выполнение теста, поскольку оно понадобится для сравнения со временем, потраченным на тот же тест после оптимизации, чтобы оценить ее эффективность.

Группировка асинхронных операций

При работе с асинхронными операциями наиболее простым вариантом кэширования является **группировка** вызовов одного и того же программного интерфейса. Сама идея очень проста и заключается в том, чтобы при вызове асинхронной функции в момент, когда аналогичный вызов уже находится в режиме ожидания, присоединить обратный вызов к уже выполняемой операции, а не создавать новый запрос. Взгляните на рис. 9.1.

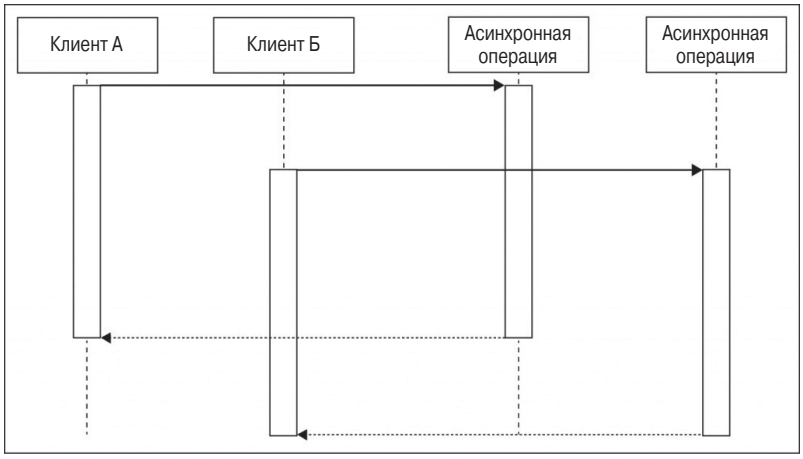

Рис. 9.1 ❖ Раздельная обработка асинхронных операций

На рис. 9.1 показаны два клиента (это могут быть два разных объекта или два разных веб-запроса), вызвавших одну и ту же асинхронную операцию *с точно такими же входными данными*. Конечно, нагляднее отобразить эту ситуацию в виде отдельных операций, запущенных двумя клиентами, причем эти операции завершаются в разные моменты времени, как показано на рис. 9.1. А теперь рассмотрим другой подход, изображенный на рис. 9.2.

Схема на рис. 9.2 демонстрирует возможность группировки двух запросов к одному и тому же программному интерфейсу, при передаче одних и тех же входных данных. Другими словами – возможность присоединения к запущенной операции. То есть когда операция завершится, уведомление получат оба клиента. Это очень простой, но чрезвычайно мощный способ оптимизации приложения, не требующий сложных механизмов кэширования, нуждающихся в адекватном управлении памятью и способах проверки допустимости.

Группировка операций на сервере итогового объема продаж

Теперь добавим группировку в программный интерфейс `totalSales`. Шаблон предельно прост и сводится к тому, что при поступлении идентичных запросов обратные

вызовы помещаются в очередь. После завершения асинхронной операции вызываются все функции, имеющиеся в очереди.

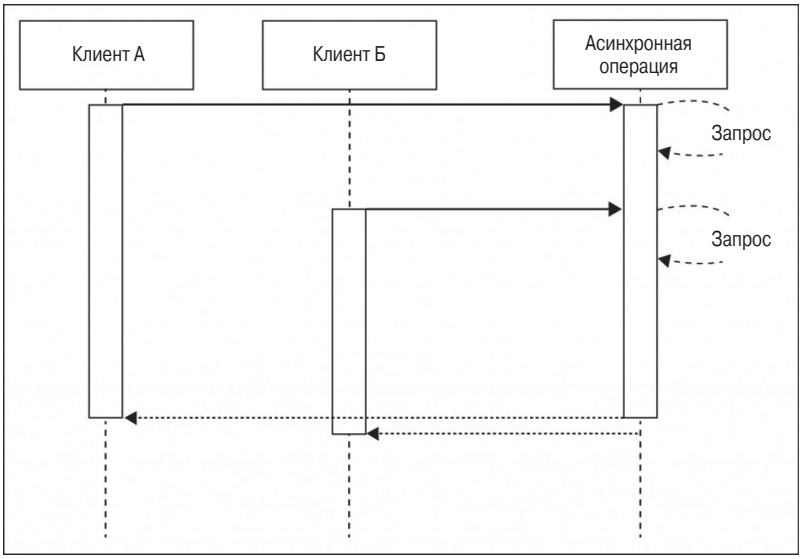

Рис. 9.2 ❖ Группировка асинхронных операций

Теперь посмотрим, как этот шаблон выглядит в коде. Создадим новый модуль `totalSalesBatch.js` для реализации группировки поверх оригинального программного интерфейса `totalSales`:

```
const totalSales = require('./totalSales');

const queues = {};
module.exports = function totalSalesBatch(item, callback) {
  if(queues[item]) {                                    // [1]
    console.log('Batching operation');
   return queues[item].push(callback);
  }

  queues[item] = [callback];                            // [2]
  totalSales(item, (err, res) => {
    const queue = queues[item];                         // [3]
    queues[item] = null;
    queue.forEach(cb => cb(err, res));
  });
};
```

Функция `totalSalesBatch()` реализует проксирование оригинального программного интерфейса `totalSales()` и действует следующим образом:

1) если очередь для заданного вида товаров `item` уже существует, значит, в настоящее время обрабатывается запрос для этого конкретного значения `item`. В таком случае достаточно добавить обратный вызов в существующую очередь и немедленно вернуть управление. Ничего больше не потребуется;

2) если очередь для заданного товара отсутствует, необходимо создать новый запрос. Для этого создается новая очередь для конкретного значения `item`, куда помещается текущая функция обратного вызова. Затем вызывается оригинальный программный интерфейс `totalSales()`;
3) после, когда оригинальная функция `totalSales()` вернет управление, выполняются обход и вызов всех функций в очереди с передачей им результата операции.

Модель поведения функции `totalSalesBatch()` идентична оригинальной функции `totalSales()`, с той лишь разницей, что она выполняет группировку вызовов с одинаковыми входными данными, экономя время и ресурсы.

Теперь можно оценить прирост производительности, по сравнению с первоначальной версией `totalSales()` без группировки. Для этого заменим модуль `totalSales`, используемый HTTP-сервером, на вновь созданный (файл `app.js`):

```
//const totalSales = require('./totalSales');
const totalSales = require('./totalSalesBatch');

http.createServer(function(req, res) {
// ...
```

Если теперь запустить сервер и выполнить нагрузочный тест, можно убедиться, что запросы обрабатываются *группами*. В этом заключается практическое преимущество только что реализованного шаблона.

Кроме того, должно наблюдаться значительное сокращение общего времени выполнения теста; по крайней мере, в четыре раза, по сравнению с первоначальной версией `totalSales()`!

Это потрясающий результат, подтверждающий, что можно получить огромный выигрыш производительности, применив самую простую группировку, без необходимости иметь сложный механизм полноценного кэширования и, что еще более важно, не заботясь о реализации стратегии проверки действительности кэша.

Шаблон группировки запросов наиболее эффективен в высоконагруженных приложениях с медленным программным интерфейсом, поскольку именно в этих обстоятельствах имеется возможность группировать наибольшее число запросов.

Кэширование асинхронных запросов

Одна из проблем шаблона группировки запросов заключается в том, что чем быстрее работает программный интерфейс, тем меньше запросов можно сгруппировать. Может показаться, что если программный интерфейс работает быстро, нет никакого смысла пытаться его оптимизировать, но он по-прежнему расходует ресурсы приложения, что в конечном итоге может иметь существенное значение. Кроме того, иногда можно уверенно полагать, что результаты вызовов остаются актуальными достаточно долго. В такой ситуации простая группировка запросов не дает улучшения производительности, и тогда лучшим кандидатом, позволяющим уменьшить нагрузку и улучшить интерактивность приложения, является шаблон более агрессивного кэширования.

Его идея проста и заключается в том, чтобы сохранить результат запроса в кэше, который может быть переменной, записью в базе данных или записью в специализированном сервере кэширования. Тогда при следующем обращении результат можно извлечь непосредственно из кэша, не инициируя нового запроса.

Идея кэширования не нова для опытных разработчиков, но в асинхронном программировании этот шаблон отличается дополнительной поддержкой группировки запросов, что делает его более эффективным. Причина включения такой поддержки – в том, что одновременно может обрабатываться несколько запросов, пока кэш еще не создан, и по завершении этих запросов кэш будет сохранен несколько раз.

Исходя из описания идеи, окончательная структура шаблона кэширования асинхронных запросов выглядит, как показано на рис. 9.3.

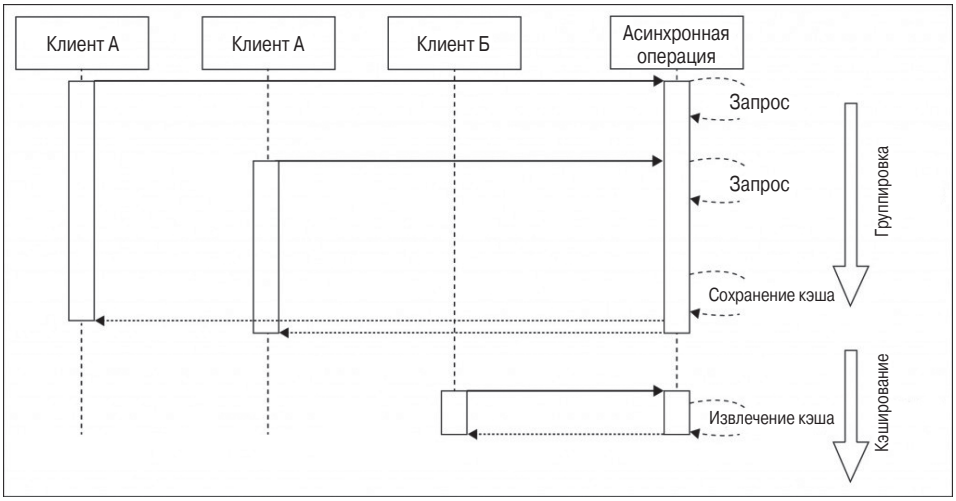

Рис. 9.3 ❖ Шаблон кэширования асинхронных запросов с поддержкой группировки

На рис. 9.3 показаны два этапа работы алгоритма оптимального асинхронного кэширования:

- первый этап полностью идентичен шаблону группировки. Любые запросы, полученные в моменты времени, пока кэш еще не создан, объединяются в группу. После обработки запроса кэш сохраняется только один раз;
- если кэш уже создан, все последующие запросы будет обслуживаться непосредственно из него.

Еще одним важным аспектом является предотвращение антишаблона высвобождения Залго (рассматривался в *главе 2 «Основные шаблоны Node.js»*). Так как работа ведется с асинхронным программным интерфейсом, необходимо гарантировать только асинхронный возврат кэшированных значений, даже если доступ к кэшу является полностью синхронной операцией.

Кэширование запросов на сервере итогового объема продаж

Для демонстрации и оценки преимуществ шаблона асинхронного кэширования применим новые знания для улучшения `totalSales()`. Подобно тому, как это было сделано в примере группировки, создадим прокси для оригинального программного интерфейса с единственной целью – добавление кэширования.

Создадим новый модуль `totalSalesCache.js`, содержащий следующий код:

```
const totalSales = require('./totalSales');

const queues = {};
const cache = {};

module.exports = function totalSalesBatch(item, callback) {
  const cached = cache[item];
  if (cached) {
    console.log('Cache hit');
    return process.nextTick(callback.bind(null, null, cached));
  }

  if (queues[item]) {
    console.log('Batching operation');
    return queues[item].push(callback);
  }

  queues[item] = [callback];
  totalSales(item, (err, res) => {
    if (!err) {
      cache[item] = res;
      setTimeout(() => {
        delete cache[item];
      }, 30 * 1000); //30 секунд истекли
    }

    const queue = queues[item];
    queues[item] = null;
    queue.forEach(cb => cb(err, res));
  });
};
```

Как видите, предыдущий код во многом идентичен реализации группировки асинхронных запросов. Отличия заключаются в следующем:

- первое, что нужно сделать при вызове программного интерфейса, – проверить наличие кэша и, если он есть, немедленно вернуть кэшированное значение вызовом функции `callback()`, отложив ее вызов с помощью метода `process.nextTick()`;
- выполнение продолжается в режиме группировки, но на этот раз, если операция завершается успехом, ее результат сохраняется в кэше. Кроме того, следует очистить кэш по истечении 30 секунд. Простой, но очень эффективный способ!

Теперь можно проверить работу вновь созданной обертки для `totalSales`, но перед этим необходимо изменить модуль `app.js`, как показано ниже:

```
//const totalSales = require('./totalSales');
//const totalSales = require('./totalSalesBatch');
const totalSales = require('./totalSalesCache');

http.createServer(function(req, res) {
// ...
```

Далее следует запустить сервер и выполнить профилирование с помощью сценария `loadTest.js`, как это делалось в предыдущих примерах. При использовании параметров тестирования по умолчанию должно наблюдаться примерно 10%-ное сокра-

щение времени выполнения, по сравнению с обычной группировкой. Конечно, время выполнения будет определяться многими факторами, такими как число запросов, длительность паузы и т. д. Преимущества кэширования вдобавок к группировке проявятся наиболее ярко при большом количестве запросов и времени обработки одного запроса.

Мемоизация – один из способов кэширования, когда в кэше сохраняются результаты выполнения функции. В npm можно найти множество пакетов, упрощающих реализацию асинхронной мемоизации. Одним из наиболее полных таких пакетов является пакет `memoizee` (https://npmjs.org/package/memoizee).

Комментарии к реализации механизма кэширования

Следует учитывать, что в реальных приложениях часто возникает необходимость использовать более продвинутые механизмы хранения и методы очистки кэша. Это может потребоваться по следующим причинам:

- большое количество кэшированных значений может потреблять слишком много памяти. В этом случае избежать чрезмерных затрат памяти поможет применение алгоритма **вытеснения давно неиспользуемых** (Least Recently Used, LRU);
- когда приложение выполняется как группа процессов, использование обычных переменных для хранения кэша может привести к возврату разных результатов разными экземплярами сервера. Если это нежелательно, решить данную проблему можно, использовав для кэша общее хранилище. Это обеспечивают такие популярные инструменты, как Redis (http://redis.io) и Memcached (http://memcached.org);
- очистка кэша вручную, в отличие от использования механизма истечения срока действия, продлевает существование кэша и при этом позволяет возвращать более актуальные данные, но таким кэшем сложнее управлять.

Группировка и кэширование с использованием объектов Promise

В *главе 4 «Шаблоны асинхронного выполнения с использованием спецификации ES2015, и не только»* мы видели, как объекты `Promise` позволяют существенно упростить асинхронный код, но еще более интересное их использование связано с группировкой запросов и кэшированием. Обсуждая объекты `Promise`, мы познакомились с двумя их свойствами, которые в данном случае обеспечивают существенные преимущества:

- к одному объекту `Promise` можно подключить несколько обработчиков `then()`;
- обработчик `then()` гарантированно будет вызван хотя бы один раз, даже если он подключен после разрешения объекта `Promise`. Кроме того, `then()` *всегда* вызывается асинхронно.

Короче говоря, первое свойство – именно то, что требуется для группировки запросов, а второе означает, что объект `Promise` уже кэширует вычисленное значение, что обеспечивает естественный механизм возврата кэшированного значения асинхронным образом. Следовательно, группировку и кэширование можно очень просто и лаконично реализовать с помощью объектов `Promise`.

Чтобы показать, как это реализуется, создадим обертку для `totalSales()`, используя объекты `Promise`, и посмотрим, что необходимо для группировки и кэширования. Создадим новый модуль `totalSalesPromises.js`:

```
const pify = require('pify');                    // [1]
const totalSales = pify(require('./totalSales'));

const cache = {};
module.exports = function totalSalesPromises(item) {
  if (cache[item]) {                             // [2]
    return cache[item];
  }

  cache[item] = totalSales(item)                 // [3]
   .then(res => {                                // [4]
     setTimeout(() => {
       delete cache[item];
     }, 30 * 1000); //30 секунд истекли
     return res;
   })
   .catch(err => {                               // [5]
     delete cache[item];
     throw err;
  });
  return cache[item];                            // [6]
};
```

Первое, что поражает в данной реализации, – это ее простота и элегантность. Объекты `Promise` действительно являются отличным инструментом, но в этом конкретном случае они изначально дают существенные преимущества. Вот как работает этот код.

1. Сначала подключается небольшой модуль `pify` (https://www.npmjs.com/package/pify), позволяющий применить объекты `Promise` к оригинальной функции `totalSales()`. После этого `totalSales()` будет возвращать объекты `Promise`.
2. При вызове обертки `totalSalesPromises()` проверяется, существует ли кэшированный объект `Promise` для заданного вида товара `item`. Если такой объект уже есть, он возвращается вызывающему коду.
3. Если в кэше отсутствует объект `Promise` для заданного вида товара `item`, создается ссылка на оригинальную (возвращающую объекты `Promise`) функцию `totalSales()`.
4. При разрешении объекта `Promise` задается время очистки кэша (через 30 секунд) и возвращается `res` для передачи результата операции всем обработчикам `then()`, подключенным к объекту `Promise`.
5. Если объект `Promise` отклоняется с ошибкой, кэш немедленно очищается, и ошибка возбуждается вновь, чтобы передать ее по цепочке объектов `Promise` всем подключенным обработчикам.
6. И наконец, возвращается вновь созданный кэшированный объект `Promise`.

Очень просто, понятно и, что более важно, нам удалось реализовать группировку с кэшированием.

Чтобы проверить работу функции `totalSalesPromise()`, придется немного изменить модуль `app.js`, поскольку теперь программный интерфейс использует объекты `Promise` вместо обратных вызовов. Создадим для этого измененную версию модуля `app` в файле `appPromises.js`:

```
const http = require('http');
const url = require('url');
```

```
const totalSales = require('./totalSalesPromises');
http.createServer(function(req, res) {
  const query = url.parse(req.url, true).query;
  totalSales(query.item).then(function(sum) {
    res.writeHead(200);
    res.end(`Total sales for item ${query.item} is ${sum}`);
  });
}).listen(8000, () => console.log('Started'));
```

Новая версия практически полностью совпадает с оригинальным модулем `app`, с той лишь разницей, что теперь для группировки и кэширования используется версия обертки, основанная на объектах `Promise`, то есть путь к ней также несколько отличается.

Это все! Теперь можно проверить работу новой версии сервера, выполнив следующую команду:

```
node appPromises
```

С помощью сценария `loadTest` можно убедиться, что новая реализация работает в полном соответствии с ожиданиями. Время выполнения должно быть примерно таким же, как при тестировании сервера с функцией `totalSalesCache()`.

Выполнение вычислительных заданий

Программный интерфейс `totalSales()`, хотя и потребляет достаточно ресурсов, не влияет на способность сервера параллельно принимать запросы. Описание работы цикла событий в *главе 1 «Добро пожаловать в платформу Node.js»* объясняет это тем, что вызовы асинхронных операций собираются в стеке, чтобы дать возможность выполняться циклу событий, что позволяет выполнить обработку других запросов.

Но что произойдет при попытке выполнить продолжительное синхронное задание, которое никогда не передает управление циклу событий? Такие задания называют вычислительными, поскольку их основной особенностью является выполнение вычислительных операций с существенным потреблением центрального процессора, а не операций ввода/вывода.

Перейдем сразу же к рассмотрению примера, чтобы увидеть, как такие задания ведут себя в Node.js.

Решение задачи выделения подмножеств с заданной суммой

Начнем с выбора ресурсоемкой задачи, которая послужит основой для эксперимента. Хорошим кандидатом на эту роль является задача выделения подмножества с нулевой суммой, заключающаяся в том, чтобы определить, содержит ли множество (или несколько множеств) целых чисел непустое подмножество элементов, сумма которых равна нулю. Например, если выбрать в качестве исходного множество [1, 2, –4, 5, –3], условиям задачи удовлетворяют подмножества: [1, 2, –3] и [2, –4, 5, –3].

Простейший алгоритм решения этой задачи заключается в проверке всех возможных комбинаций элементов любого размера. Он имеет вычислительную стоимость $O(2^n)$, или, другими словами, она растет экспоненциально с увеличением размера исходного множества. Так, для множества из 20 целых чисел потребуется проверить до 1 048 576 комбинаций, чего вполне достаточно для тестирования. Однако решение

может быть найдено много раньше проверки всех комбинаций. Поэтому усложним задачу, сформулировав ее следующим образом: для заданного множества целых чисел требуется получить все возможные комбинации его элементов, сумма которых будет равна заданному произвольному целому числу.

Давайте реализуем такой алгоритм. Для этого создадим новый модуль `subset-Sum.js` и начнем с определения класса `SubsetSum`:

```
const EventEmitter = require('events').EventEmitter;
class SubsetSum extends EventEmitter {
  constructor(sum, set) {
    super();
    this.sum = sum;
    this.set = set;
    this.totalSubsets = 0;
  }
//...
```

Класс `SubsetSum` наследует `EventEmitter`, что позволяет ему генерировать событие при обнаружении каждого нового подмножества с суммой элементов, равной заданному целому числу. Как будет показано позже, это обеспечивает большую гибкость.

Далее рассмотрим создание всевозможных подмножеств:

```
_combine(set, subset) {
  for(let i = 0; i < set.length; i++) {
    let newSubset = subset.concat(set[i]);
    this._combine(set.slice(i + 1), newSubset);
    this._processSubset(newSubset);
  }
}
```

Мы не будем глубоко погружаться в детали алгоритма, но две важные его особенности следует пояснить:

- метод `_combine()` является синхронным. Он рекурсивно создает все возможные подмножества без передачи управления циклу событий. И это нормально для алгоритма, не выполняющего никаких операций ввода/вывода;
- каждый раз, когда создается новое подмножество, оно передается для обработки методу `_processSubset()`.

Метод `_ProcessSubset()` сравнивает сумму элементов переданного подмножества с заданным числом:

```
_processSubset(subset) {
  console.log('Subset', ++this.totalSubsets, subset);
  const res = subset.reduce((prev, item) => (prev + item), 0);
  if(res == this.sum) {
    this.emit('match', subset);
  }
}
```

Метод `_processSubset()` просто применяет к подмножеству операцию `reduce` для подсчета суммы элементов. Он генерирует событие `'match'`, когда сумма равна заданному числу (`this.sum`).

И наконец, метод `start()` соединяет все приведенные выше фрагменты:

```
start() {
  this._combine(this.set, []);
  this.emit('end');
}
```

Он запускает перебор всех возможных комбинаций, вызывая метод `_combine()`, и в конце генерирует событие `'end'`, оповещающее о том, что все подмножества проверены и обо всех совпадениях уже было сообщено. Такое поведение возможно благодаря синхронной природе метода `_combine()`, то есть событие `'end'` будет сгенерировано после завершения функции, а это значит, что все подмножества уже были найдены.

Далее нужно обеспечить доступность вновь созданного алгоритма через сеть, для чего, как всегда, будет использоваться простой HTTP-сервер. В частности, создадим конечную точку в формате `/subsetSum?data=<Array>&sum=<Integer>`, которая будет вызывать функцию `SubsetSum` и передавать ей массив целых чисел и значение суммы для сравнения.

Реализуем этот простой сервер в модуле `app.js`:

```
const http = require('http');
const SubsetSum = require('./subsetSum');

http.createServer((req, res) => {
  const url = require('url').parse(req.url, true);
  if(url.pathname === '/subsetSum') {
    const data = JSON.parse(url.query.data);
    res.writeHead(200);
    const subsetSum = new SubsetSum(url.query.sum, data);
    subsetSum.on('match', match => {
      res.write('Match: ' + JSON.stringify(match) + '\n');
    });
    subsetSum.on('end', () => res.end());
    subsetSum.start();
  } else {
    res.writeHead(200); res.end('I\m alive!\n');
  }
}).listen(8000, () => console.log('Started'));
```

Благодаря тому что объект `SubsetSum` возвращает результат с помощью события, можно организовать потоковую передачу соответствующих подмножеств в режиме реального времени, по мере их создания алгоритмом. Также следует упомянуть еще одну деталь: сервер возвращает текст `I'm Alive!` (Я – жив!) при обращении к любому URL, отличному от `/subsetSum`. Эта особенность будет позже использоваться для проверки реакции сервера.

Теперь все готово для проверки алгоритма извлечения подмножеств с определенной суммой элементов. Любопытно посмотреть, как сервер справится с его выполнением, не так ли? Запустим его с помощью команды

```
node app
```

Сразу после запуска можно отправить серверу первый запрос. Попробуем задать множество из 17 случайных чисел, что приведет к проверке 131 071 комбинации и позволит занять сервер работой на некоторое время:

```
curl -G http://localhost:8000/subsetSum --data-urlencode "data=[116, 119,101,101,-116,109,101,-105,
-102,117,-115,-97,119,-116,-104,-105,115]" --data-urlencode "sum=0"
```

После выполнения запроса начнут выводиться первые результаты, поступающие с сервера. Но если попробовать выполнить следующую команду в другом терминале, пока обрабатывается первый запрос, это сразу же выявит серьезную проблему:

```
curl -G http://localhost:8000
```

Этот запрос останется в подвешенном состоянии до завершения предыдущего, запустившего алгоритм поиска подмножеств с заданной суммой, то есть сервер не отвечает! Этого и следовало ожидать. Цикл событий в Node.js выполняется в одном потоке, и если этот поток заблокирован длительными синхронными вычислениями, он не сможет выполнить ни одного цикла, чтобы просто вывести `I'm alive!`

Очевидно, что такая модель поведения не подходит ни для каких приложений, предназначенных для *параллельного обслуживания запросов*. Но не стоит отчаиваться, в Node.js эту проблему можно решить несколькими способами. Рассмотрим два самых важных из них.

Чередование с помощью функции setImmediate

Как правило, вычислительные алгоритмы разбиваются на несколько этапов. Это может быть набор рекурсивных вызовов, цикл или любые их вариации и комбинации. То есть самым простым решением является возврат управления циклу событий после завершения любого из этих этапов (или определенного их количества). В этом случае операции ввода/вывода, ожидающие обработки, могут быть выполнены циклом событий в те промежутки, когда продолжительный вычислительный алгоритм уступает центральный процессор. Проще всего добиться этого – запланировать следующий этап алгоритма на момент времени после обработки ожидающих запросов ввода/вывода. Именно для этого предназначена функция `setImmediate()` (мы познакомились с ней в *главе 2 «Основные шаблоны Node.js»*).

Шаблон
Чередование этапов продолжительной синхронной задачи с помощью функции `setImmediate()`.

Чередование этапов алгоритма извлечения подмножеств с заданной суммой

Давайте посмотрим, как этот шаблон можно применить к алгоритму извлечения подмножеств с заданной суммой. Для этого потребуется немного изменить модуль `subsetSum.js`. Для удобства создадим новый модуль `subsetSumDefer.js`, взяв за основу определение оригинального класса `subsetSum`.

Прежде всего добавим новый метод `_combineInterleaved()`, являющийся основой шаблона:

```
_combineInterleaved(set, subset) {
  this.runningCombine++;
  setImmediate(() => {
    this._combine(set, subset);
    if(--this.runningCombine === 0) {
      this.emit('end');
```

```
    }
  });
}
```

Как видите, мы откладываем вызов оригинального метода `_combine()` (синхронного) с помощью функции `setImmediate()`. Но теперь стало сложнее узнать об окончании перебора всех комбинаций, поскольку алгоритм больше не является синхронным. Чтобы решить эту проблему, будем следить за всеми запущенными экземплярами метода `_combine()`, применив шаблон, очень похожий на шаблон асинхронного параллельного выполнения, который рассматривался в *главе 3* «*Шаблоны асинхронного выполнения с обратными вызовами*». Когда все экземпляры метода `_combine()` завершатся, будет сгенерировано событие `end`, уведомляющее о завершении процесса.

Наконец, для дополнительной оптимизации алгоритма извлечения подмножеств с заданной суммой понадобится внести еще несколько изменений. Во-первых, заменим рекурсивный вызов в методе `_combine()` его отложенной версией:

```
_combine(set, subset) {
  for(let i = 0; i < set.length; i++) {
    let newSubset = subset.concat(set[i]);
    this._combineInterleaved(set.slice(i + 1), newSubset);
    this._processSubset(newSubset);
  }
}
```

Это изменение гарантирует, что все этапы алгоритма будут поставлены в очередь цикла событий с помощью функции `setImmediate()` и, следовательно, выполняться после обработки ожидающих запросов ввода/вывода, а не синхронно.

Еще одно изменение нужно внести в метод `start()`:

```
start() {
  this.runningCombine = 0;
  this._combineInterleaved(this.set, []);
}
```

Здесь число запущенных экземпляров метода `_combine()` инициализируется нулем. Кроме того, вместо `_combine()` вызывается `_combineInterleaved()`, и удалена генерация события `'end'`, поскольку теперь эта операция выполняется асинхронно, в `_combineInterleaved()`.

После этого последнего изменения наш вычислительный алгоритм извлечения подмножеств с заданной суммой сможет выполняться поэтапно, чередуясь с интервалами, в течение которых цикл событий сможет обрабатывать любые другие ожидающие запросы.

Остается только внести изменение в модуль `app.js`, чтобы он мог использовать новую версию `SubsetSum`. Это вполне тривиальное изменение:

```
const http = require('http');
//const SubsetSum = require('./subsetSum');
const SubsetSum = require('./subsetSumDefer');

http.createServer(function(req, res) {
  // ...
```

Теперь все готово для проверки работы новой версии сервера, извлекающего подмножества с заданной суммой. Запустим модуль `app` командой

```
node app
```

Затем отправим запрос для извлечения подмножеств с заданной суммой:

```
curl -G http://localhost:8000/subsetSum --data-urlencode "data=[116, 119,101,101,-116,109,101,-105,
-102,117,-115,-97,119,-116,-104,-105,115]" --data-urlencode "sum=0"
```

Пока запрос обрабатывается, можно выяснить, отвечает ли сервер:

```
curl -G http://localhost:8000
```

Отлично! Второй запрос должен вернуть ответ немедленно, пока задание `SubsetSum` продолжает выполняться, что подтверждает работоспособность шаблона.

Комментарии к шаблону чередования

Как видите, реализация вычислительных заданий с сохранением отзывчивости приложения не слишком сложна, она лишь требует использования функции `setImmediate()`, чтобы отложить выполнение следующего этапа на момент после обработки ожидающих операций ввода/вывода. Но это не лучший шаблон с точки зрения эффективности, поскольку откладывание вносит определенные затраты, и если помножить их на количество этапов, которые должен выполнить алгоритм, результат может иметь существенное значение. Обычно это крайне нежелательно при выполнении вычислительных заданий, особенно когда их результат требуется вернуть непосредственно пользователю, что следует сделать за разумный промежуток времени. Остроту этой проблемы можно несколько смягчить, если использовать функцию `setImmediate()` только после определенного числа этапов, а не после каждого, но эта мера не устраняет причину проблемы.

Это вовсе не означает, что описанного шаблона следует избегать любой ценой, поскольку совсем не обязательно синхронные задания должны быть настолько длительными и сложными, чтобы создавать подобные проблемы. В высоконагруженном сервере даже задача, блокирующая цикл событий на 200 миллисекунд, способна создать существенные задержки. Однако когда задача выполняется нерегулярно или в фоновом режиме и не слишком долго, использование функции `setImmediate()` для чередования является самым простым и наиболее эффективным способом предотвращения блокировки цикла событий.

Метод `process.nextTick()` нельзя использовать для приостановки продолжительных заданий. Как было описано в *главе 1 «Добро пожаловать в платформу Node.js»*, метод `nextTick()` откладывает операцию на время, перед любыми ожидающими операциями ввода/вывода, что в конечном итоге может вызвать задержку операций ввода/вывода при повторных вызовах. Это можно проверить самостоятельно, заменив функцию `setImmediate()` методом `process.nextTick()` в предыдущем примере. Такая модель поведения была введена начиная с версии 0.10 платформы Node.js, причем в версии 0.8 метод `process.nextTick()` можно было использовать в механизме чередования. Дополнительные сведения об истории и мотивах этого изменения можно найти в GitHub, на странице https://github.com/joyent/node/issues/3335.

Использование нескольких процессов

Откладывание этапов алгоритма – не единственный способ выполнения вычислительных заданий. Другой шаблон, предотвращающий блокировку цикла событий,

опирается на запуск дочерних процессов. Как уже упоминалось ранее, Node.js отлично справляется с выполнением приложений, характеризующихся интенсивным вводом/выводом, таких как веб-серверы, позволяя оптимизировать использование ресурсов благодаря своей асинхронной архитектуре.

Поэтому для лучшей отзывчивости приложения вычислительные задачи следует запускать не в контексте основного приложения, а в отдельных процессах. Это обеспечивает три основных преимущества:

- синхронное задание может выполняться на предельной скорости, поскольку отсутствует необходимость чередовать действия;
- работа с процессами в Node.js реализуется проще, чем изменение алгоритма для применения `setImmediate()`, и позволяет *использовать несколько процессоров* без необходимости масштабировать основное приложение;
- если действительно необходима максимальная производительность, внешний процесс можно написать на одном из языков низкого уровня, таком как старый добрый C (всегда используйте лучший инструмент для работы!).

Node.js предоставляет богатый набор инструментов для взаимодействия с внешними процессами. Все, что для этого требуется, можно найти в модуле `child_process`. Кроме того, если внешним процессом является другая программа, подключить ее к главному приложению очень легко, и можно даже не почувствовать запуска чего-то внешнего из локального приложения. Это возможно благодаря функции `child_process.fork()`, которая создает новый дочерний процесс, а также автоматически обеспечивает канал для связи с ним, что позволяет обмениваться информацией, используя интерфейс, очень похожий на `EventEmitter`. Рассмотрим применение этого подхода, реорганизовав еще раз сервер с алгоритмом извлечения подмножеств с заданной суммой.

Делегирование задачи извлечения подмножеств с заданной суммой другим процессам

Целью реорганизации задачи `SubsetSum` является создание отдельного дочернего процесса, отвечающего за синхронную обработку, что обеспечит высвобождение цикла событий сервера для обработки запросов, поступающих из сети. Вот рецепт реализации такого поведения.

1. Создать новый модуль `processPool.js`, создающий пул процессов. Запуск нового процесса является достаточно затратным в отношении ресурсов и времени, поэтому наличие пула работающих процессов, готовых к обработке запросов, позволит сэкономить время и ресурсы центрального процессора. Кроме того, пул поможет ограничить число одновременно выполняющихся процессов, что позволит исключить риск атак **отказ в обслуживании** (denial-of-service, DoS).
2. Создать модуль `subsetSumFork.js`, отвечающий за абстрагирование задания `SubsetSum`, выполняемого в дочернем процессе. Его задача – взаимодействие с дочерним процессом и вывод результатов в той форме, как если бы они поступали из текущего приложения.
3. И наконец, создать **рабочий** (дочерний) процесс – программу Node.js, предназначенную только для выполнения алгоритма извлечения подмножеств с заданной суммой элементов и передачи результатов родительскому процессу.

DoS-атака – это попытка сделать машину или сетевой ресурс недоступным для предполагаемых пользователей, чтобы временно или постоянно отключить или приостановить службу, действующую на подключенном к Интернету узле.

Реализация пула процессов Начнем с пошагового определения модуля processPool.js:

```
const fork = require('child_process').fork;

class ProcessPool {
  constructor(file, poolMax) {
    this.file = file;
    this.poolMax = poolMax;
    this.pool = [];
    this.active = [];
    this.waiting = [];
  }
//...
```

В первой части модуль импортирует функцию child_process.fork(), которая будет использоваться для создания новых процессов. Затем определяется конструктор класса ProcessPool, принимающий аргумент file с программой для запуска и максимальное количество запущенных экземпляров в пуле (poolMax). Далее определяются три переменных экземпляра:

- pool – набор готовых к использованию запущенных процессов;
- active – список используемых в настоящее время процессов;
- waiting – очередь функций обратного вызова для всех запросов, которые не могут быть выполнены немедленно из-за отсутствия доступных процессов.

Следующий фрагмент класса ProcessPool – метод acquire(), – возвращающий готовый к использованию процесс:

```
acquire(callback) {
  let worker;
  if(this.pool.length > 0) {                    // [1]
    worker = this.pool.pop();
    this.active.push(worker);
    return process.nextTick(callback.bind(null, null, worker));
  }

  if(this.active.length >= this.poolMax) {      // [2]
    return this.waiting.push(callback);
  }

  worker = fork(this.file);                     // [3]
  this.active.push(worker);
  process.nextTick(callback.bind(null, null, worker));
}
```

Его логика достаточно проста:

1) если в пуле имеется готовый к использованию процесс, он просто перемещается в список active, а затем возвращается вызовом функции callback (в отложенной манере... не забываем о Залго);
2) если в пуле нет доступных процессов и достигнуто максимальное число запущенных процессов, нужно пождать, пока один из них освободится. Для этого текущий обратный вызов помещается в список waiting;
3) если еще не достигнуто максимальное число запущенных процессов, вызовом функции child_process.fork()создается новый процесс и добавляется в список active с последующим возвратом этого процесса с помощью функции callback.

Последний метод класса `ProcessPool` – `release()` возвращает процесс обратно в пул:

```
release(worker) {
  if(this.waiting.length > 0) {                          // [1]
    const waitingCallback = this.waiting.shift();
    waitingCallback(null, worker);
  }
  this.active = this.active.filter(w => worker !== w);  // [2]
  this.pool.push(worker);
}
```

Этот метод также довольно прост:

- если в списке `waiting` имеется запрос, выполняется переназначение освободившегося рабочего процесса путем передачи ему обратного вызова из начала очереди `waiting`;
- в противном случае рабочий процесс исключается из списка `active` и помещается обратно в пул.

Как видите, процессы никогда не уничтожаются – они просто переназначаются, что позволяет экономить время, которое потребовалось бы на запуск процесса при поступлении любого запроса. Однако следует отметить, что такой подход не всегда является оптимальным и его выбор зависит от требований приложения. Чтобы уменьшить объем памяти, не используемой долгое время, и повысить надежность пула, можно:

- уничтожать процессы, простаивающие определенное время, для высвобождения памяти;
- добавить механизм уничтожения зависших процессов или перезапуска тех из них, в работе которых возник сбой.

Но в этом примере реализация пула процессов будет предельно простой, поскольку возможных усовершенствований здесь бесконечное множество.

Взаимодействие с дочерним процессом Теперь, когда класс `ProcessPool` готов, можно использовать его для реализации обертки `SubsetSumFork`, чья роль заключается во взаимодействии с рабочим процессом и передаче возвращаемых им результатов. Как уже упоминалось ранее, функция `child_process.fork()`, запускающая процесс, также создает коммуникационный канал для передачи сообщений, поэтому давайте посмотрим, как действует этот механизм, реализовав модуль `subsetSumFork.js`:

```
const EventEmitter = require('events').EventEmitter;
const ProcessPool = require('./processPool');
const workers = new ProcessPool( dirname + '/subsetSumWorker.js', 2);

class SubsetSumFork extends EventEmitter {
  constructor(sum, set) {
    super();
    this.sum = sum;
    this.set = set;
  }

  start() {
    workers.acquire((err, worker) => {                // [1]
      worker.send({sum: this.sum, set: this.set});

      const onMessage = msg => {
```

```
        if (msg.event === 'end') {                    // [3]
          worker.removeListener('message', onMessage);
          workers.release(worker);
        }

        this.emit(msg.event, msg.data);               // [4]
      };

      worker.on('message', onMessage);                // [2]
    });
  }
}
module.exports = SubsetSumFork;
```

Первое, на что следует обратить внимание, – инициализация объекта `ProcessPool` с передачей файла `subsetSumWorker.js` в качестве целевого объекта, представляющего дочерний рабочий процесс. Кроме того, максимальная емкость пула устанавливается равной 2.

Еще один важный момент: мы должны постараться воспроизвести программный интерфейс исходного класса `SubsetSum`. Класс `SubsetSumFork` является потомком класса `EventEmitter`, конструктор которого принимает параметры `sum` и `set`, а метод `start()` инициирует выполнение алгоритма, который на этот раз запускается в отдельном процессе. Вот что происходит при вызове метода `start()`:

1) делается попытка извлечь новый дочерний процесс из пула. В случае успеха сразу же используется дескриптор `worker` для отправки сообщения с исходными данными. Функция `send()` автоматически предоставляется платформой Node.js всем процессам, запущенным с помощью функции `child_process.fork()`. По сути, это и есть канал связи, о котором упоминалось;
2) затем настраивается прием сообщений, возвращаемых рабочим процессом, для чего с помощью метода `on()` производится подключение нового обработчика (это является частью возможностей коммуникационного канала, предоставляемого всем процессам, запущенным с помощью функции `child_process.fork()`);
3) в обработчике сначала проверяется получение события `end`, что означает завершение задания `SubsetSum`, в этом случае обработчик `onMessage` удаляется, а процесс `worker` освобождается и возвращается в пул;
4) рабочий процесс генерирует сообщения в формате `{event, data}`, что позволяет повторно возбуждать любые события дочернего процесса.

Обертка `SubsetSumFork` готова. Теперь займемся реализацией рабочего приложения.

Следует также заметить, что метод `send()` дочернего процесса тоже может использоваться для передачи дескриптора сокета из основного приложения в дочерний процесс (более подробно об этом рассказывается в документации на странице http://nodejs.org/api/child_process.html#child_process_child_send_message_sendhandle). Этот метод используется модулем `cluster` для распределения нагрузки на HTTP-сервер между несколькими процессами (в версии Node.js 0.10). Более подробно об этом рассказывается в следующей главе.

Взаимодействие с родительским процессом Теперь создадим модуль рабочего приложения `subsetSumWorker.js`, весь код которого будет выполняться в отдельном процессе:

```
const SubsetSum = require('./subsetSum');

process.on('message', msg => {                    // [1]
  const subsetSum = new SubsetSum(msg.sum, msg.set);
  subsetSum.on('match', data => {                 // [2]
    process.send({event: 'match', data: data});
  });

  subsetSum.on('end', data => {
    process.send({event: 'end', data: data});
  });

  subsetSum.start();
});
```

Как видите, здесь используется оригинальный (и синхронный) модуль `SubsetSum`. Теперь, когда модуль предполагается выполнять в отдельном процессе, нет необходимости беспокоиться о блокировке цикла событий, поскольку все HTTP-запросы будут обрабатываться циклом событий в основном приложении.

Вот что происходит, когда алгоритм запускается в дочернем процессе:

1) он сразу же начинает принимать сообщения, поступающие от родительского процесса. Это легко реализуется с помощью функции `process.on()` (также часть коммуникационного программного интерфейса всех процессов, запускаемых функцией `child_process.fork()`). От родительского процесса должно поступить единственное сообщение, содержащее исходные данные для нового задания `SubsetSum`. Сразу после получения такого сообщения создается новый экземпляр класса `SubsetSum` и регистрируются обработчики событий `match` и `end`. И наконец, с помощью метода `subsetSum.start()` запускается алгоритм;
2) при получении любого события от выполняющегося алгоритма оно обертывается в объект формата `{event, data}` и отправляется родительскому процессу. Затем эти сообщения обрабатываются в модуле `subsetSumFork.js`, как было описано в предыдущем разделе.

Как видите, понадобилось лишь обернуть алгоритм, реализованный ранее, не внося в него никаких изменений. Это ясно показывает, что любую часть приложения можно легко выделить во внешний процесс, использовав только что описанный шаблон.

Если дочерний процесс не является программой Node.js, только что описанный простой коммуникационный канал будет недоступен. В этом случае можно создать свой интерфейс связи с дочерним процессом, реализовав собственный протокол поверх стандартных потоков ввода и вывода.

Дополнительные сведения о возможностях программного интерфейса `child_process` можно найти в официальной документации на странице http://nodejs.org/api/child_process.html.

Комментарии к шаблону обработки в отдельных процессах

Как всегда, для проверки новой версии алгоритма нужно просто заменить модуль (файл `app.js`), используемый HTTP-сервером:

```
const http = require('http');
//const SubsetSum = require('./subsetSum');
//const SubsetSum = require('./subsetSumDefer');
const SubsetSum = require('./subsetSumFork');
//...
```

Теперь можно запустить сервер и отправить ему запрос:

```
curl -G http://localhost:8000/subsetSum --data-urlencode "data=[116,119,101,101,-116,109,101,-105,
-102,117,-115,-97,119,-116,-104,-105,115]" --data-urlencode "sum=0"
```

Подобно версии с чередованием, эта новая версия модуля `subsetSum` не блокирует цикла событий при выполнении вычислительных заданий. В чем легко убедиться, отправив параллельный запрос:

```
curl -G http://localhost:8000
```

Эта команда должна немедленно вернуть следующую строку:

```
I'm alive!
```

Еще интереснее попробовать одновременно запустить два задания `subsetSum`. При этом можно увидеть, что они будут использовать на полную мощность два процессора (конечно, если система оснащена несколькими процессорами). Но если одновременно запустить три задания `subsetSum`, запущенное последним сначала зависнет. Это произойдет не из-за блокировки цикла событий основного процесса, а потому, что установлен предел, разрешающий параллельную обработку только двух процессов для заданий `subsetSum`, в результате чего третий запрос начнет обрабатываться, только когда хотя бы один из двух процессов снова станет доступным в пуле.

Как видите, шаблон обработки в отдельных процессах мощнее и гибче шаблона чередования. Однако он все еще не поддерживает масштабирования, поскольку объем ресурсов, предлагаемый одной машиной, по-прежнему жестко ограничен. Решением в этом случае является распределение нагрузки между несколькими компьютерами, но это уже другая история, которая относится к категории распределенных архитектурных моделей, рассматриваемых в следующих главах.

Стоит отметить, что одной из альтернатив процессам могут стать потоки выполнения. В настоящее время имеется несколько npm-пакетов, реализующих программный интерфейс для работы с потоками из пользовательских модулей. Одним из самых популярных таких пакетов является `webworker-threads` (https://npmjs.org/package/webworker-threads). Но при всей легковесности потоков выполнения полноценные процессы способны обеспечить большую гибкость и более высокий уровень изоляции в случае возникновения таких проблем, как зависание или сбой.

Итоги

В этой главе было описано нескольких новых полезных инструментов и приемов решения более конкретных проблем. Здесь часто применялись шаблоны, обсуждавшиеся в предыдущих главах: «Состояние», «Команда» и «Прокси» – для реализации эффективной и асинхронной инициализации модулей, шаблоны управления асинхронным потоком выполнения – для группировки и кэширования, шаблоны отложенного выполнения и событий – для выполнения вычислительных заданий.

Эта глава не только дает нам набор рецептов для повторного использования и настройки под собственные потребности, но также наглядно демонстрирует, как владение некоторыми принципами и шаблонами помогает решать самые сложные проблемы разработки на платформе Node.js.

Следующие две главы представляют конечный пункт нашего путешествия. Изучив различные тактики, теперь можно переходить к стратегиям и освоению шаблонов создания масштабируемых и распределенных приложений на платформе Node.js.

Глава 10

Шаблоны масштабирования и организации архитектуры

Первоначально платформа Node.js задумывалась как неблокирующий веб-сервер, более того, она и называлась *web.js*. Но ее создатель *Райан Даль* (Ryan Dahl) вскоре осознал широкие возможности платформы и начал дополнять ее инструментами для создания серверных приложений произвольного типа на основе двойной парадигмы JavaScript/неблокируемость. Характерные особенности платформы Node.js делают ее идеальной для реализации распределенных систем, состоящих из узлов, взаимодействующих по сети. Платформа Node.js была рождена распределенной. В отличие от других веб-платформ, слово «масштабируемость» прочно входит в словарь разработчиков на платформе Node.js на самых ранних этапах создания приложений, главным образом из-за ее однопоточного характера, отсутствия возможности задействовать ресурсы машины, но часто на это имеются и более глубокие причины. Как будет показано в этой главе, масштабирование приложения означает не только увеличение пропускной способности для более быстрой обработки большего количества запросов, но и повышение надежности и устойчивости к ошибкам. Как ни странно это звучит, но масштабируемость можно рассматривать как декомпозицию сложного приложения в несколько хорошо управляемых компонентов. Масштабируемость – это многогранное понятие, точнее – шестигранное, что соответствует количеству граней *куба масштабирования*.

В этой главе мы рассмотрим следующие темы:

- что такое «куб масштабирования»;
- как масштабировать приложение, запустив несколько его экземпляров;
- как распределять нагрузку при масштабировании приложения;
- что такое «реестр служб» и как его использовать;
- как преобразовать монолитное приложение в систему микрослужб;
- как интегрировать большое количество служб с помощью простых архитектурных шаблонов.

Введение в масштабирование приложений

Прежде чем углубиться в конкретные шаблоны и примеры, остановимся на целях масштабирования приложений и способах их достижения.

Масштабирование приложений на платформе Node.js

Как мы уже знаем, большинство заданий в типичном приложении на платформе Node.js выполняется в контексте одного потока. В *главе 1 «Добро пожаловать в платформу Node.js»* мы узнали, что это является не ограничением, а скорее преимуществом, поскольку позволяет приложению оптимизировать использование ресурсов, необходимых для обработки параллельных запросов, благодаря парадигме неблокирующего ввода-вывода. Единственного потока выполнения, рационально использующего неблокирующий ввод/вывод, вполне достаточно для приложений, обрабатывающих умеренное число запросов, обычно несколько сотен в секунду (во многом зависит от особенностей приложения). Каким бы мощным ни было аппаратное обеспечение, пропускная способность единственного потока выполнения все равно ограничена, поэтому если вы желаете использовать Node.js для создания высоконагруженных приложений, вы вынуждены будете прибегнуть к *масштабированию* с использованием нескольких процессов и машин.

Однако высокая нагрузка – не единственная причина масштабирования приложений на платформе Node.js; фактически, используя те же приемы, можно улучшить другие характеристики приложения, такие как **надежность** и **устойчивость к сбоям**. Кроме того, понятие «масштабируемость» относится также к размеру и сложности приложения, поскольку создание архитектуры, способной расширяться, играет важную роль при разработке программного обеспечения. Язык JavaScript является тем инструментом, пользоваться которым надо с большой осторожностью, поскольку отсутствие проверки типов и масса подводных камней могут стать серьезным препятствием для расширения приложения, но дисциплина и аккуратное проектирование позволяют обратить эти недостатки в преимущества. Применение JavaScript обязывает сохранять приложение простым и разбивать его на управляемые части, что облегчает масштабирование и распределение.

Три измерения масштабируемости

Когда речь идет о масштабируемости, первым основополагающим принципом является **распределение нагрузки** – *технология* распределения нагрузки приложения между несколькими процессами и машинами. Существует множество способов достижения цели, и книга «*The Art of Scalability*» *Мартина Л. Эбботта* (Martin L. Abbott) и *Майкла Т. Фишера* (Michael T. Fisher) предлагает гениальную модель представления, называемую **кубом масштабирования**. Эта модель описывает масштабируемость с точки зрения трех размерностей:

- *ось x*: клонирование;
- *ось y*: разделение на службы/функциональные особенности;
- *ось z*: разбиение на разделы данных.

Эти три измерения можно представить в виде куба, как показано на рис. 10.1.

Нижний левый угол куба представляет приложения, все функциональные возможности и службы которых сосредоточены в единой (монолитной) базе кода, выполняющейся в одном экземпляре. Это обычная ситуация для приложений,

обрабатывающих небольшие нагрузки, или приложений, находящихся на ранних этапах разработки.

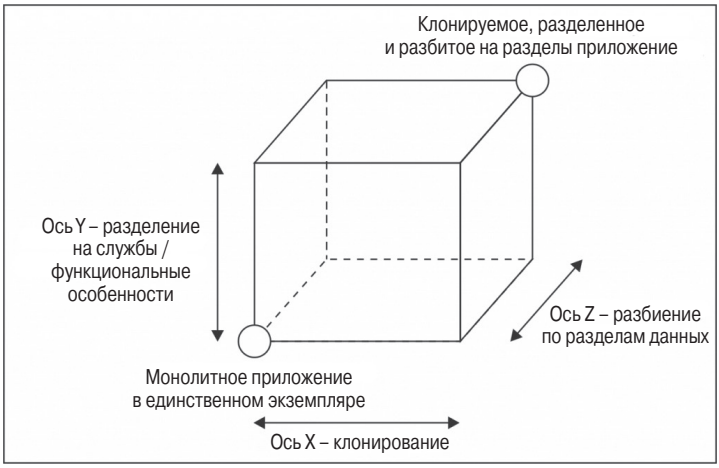

Рис. 10.1 ❖ Куб масштабирования

Наиболее понятным выглядит смещение монолитного приложения, не затронутого масштабированием, вправо, вдоль оси *x*, что является простым, как правило, недорогим (с точки зрения стоимости разработки) и весьма эффективным способом. Суть этого способа элементарно проста и заключается в клонировании одного и того же приложения *n* раз, что позволяет каждому экземпляру обрабатывать $1/n$-часть рабочей нагрузки.

Масштабирование вдоль оси *y* обозначает декомпозицию приложений по функциональным особенностям, службам или вариантам использования. В данном случае декомпозиция предполагает создание разных автономных приложений, каждое из которых обладает собственной базой кода, а иногда и выделенной базой данных и даже отдельным пользовательским интерфейсом. Например, обычным делом является отделение административной части приложения от общедоступных услуг. Еще одним примером является выделение службы аутентификации пользователей, создание выделенного сервера аутентификации. Критерии декомпозиции приложения на основе функциональных особенностей в основном зависят от прикладных требований, вариантов использования, данных и многих других факторов, как это будет показано далее в этой главе. Интересно, что это измерение масштабирования оказывает существенное влияние не только на архитектуру приложения, но и на подход к его разработке. Как будет продемонстрировано ниже, в настоящее время с масштабированием по оси *y* чаще всего ассоциируется термин «микрослужбы».

Последним измерением масштабирования является ось *z*, которая определяет такое разбиение приложения, что каждый из его экземпляров отвечает только за часть всего массива данных. Этот метод используется в основном для баз данных и по-другому называется **горизонтальным разделением**, или **шардингом** (sharding). В этом случае имеется несколько экземпляров одного и того же приложения, каждый из которых имеет дело с отдельным разделом данных, который определяется, исходя из различных критериев. Например, можно разделить пользователей приложения по странам

(*списочное разделение*) или на основе начальных букв их фамилий (*диапазонное разделение*) или же позволить хеш-функциям определить принадлежность пользователя к определенному разделу (*хешевое разделение*). Затем каждый из разделов соотносится с конкретным экземпляром приложения. Использование разделов данных требует перед каждой операцией определять, какой именно экземпляр приложения отвечает за нужный раздел данных. Как уже упоминалось ранее, разделение данных обычно применяется и обрабатывается на уровне баз данных, поскольку его главной целью является решение проблем, связанных с обработкой крупных монолитных наборов данных (ограниченность дискового пространства, памяти и пропускной способности сети). Применение его на уровне приложения имеет смысл только для сложных, распределенных архитектур или в некоторых весьма специфичных ситуациях, например при разработке приложений, базирующихся на собственных решениях для хранения данных, при использовании баз данных, не поддерживающих разбиения, или при разработке приложений масштаба Google. Учитывая его сложность, масштабирование приложения вдоль оси *z* следует применять только после того, как исчерпаны все возможности масштабирования по осям *x* и *y* куба масштабирования.

В следующих разделах мы сосредоточимся на двух наиболее распространенных и эффективных методах масштабирования приложений для платформы Node.js, а именно **клонировании** и **декомпозиции** на основе функциональных особенностей/служб.

Клонирование и распределение нагрузки

Традиционные, многопоточные веб-серверы обычно масштабируется, только когда невозможно нарастить ресурсы машины или когда это является более дорогой операцией, чем подключение к работе еще одной машины. Используя механизм многопоточности, традиционные веб-серверы могут задействовать всю вычислительную мощность сервера, используя все доступные процессоры и память. Однако с единственным процессом Node.js этого добиться труднее, поскольку приложению доступен один поток и ему по умолчанию доступен ограниченный объем памяти 1,7 ГБ на 64-битных машинах (этот предел можно увеличить с помощью специального параметра командной строки `--max_old_space_size`). Это значит, что приложения на платформе Node.js обычно начинают нуждаться в масштабировании гораздо раньше, чем традиционные веб-серверы, даже в контексте одной машины, чтобы иметь возможность пользоваться всеми ее ресурсами.

Вертикальное масштабирование (наращивание ресурсов одной машины) и **горизонтальное масштабирование** (добавление в инфраструктуру дополнительных машин) являются в Node.js практически эквивалентными концепциями, поскольку требуют применения схожих методов для использования всех доступных вычислительных мощностей.

Не стоит считать это недостатком. Напротив, практически *вынужденное* масштабирование оказывает благоприятное воздействие на другие характеристики приложения, в частности на надежность и отказоустойчивость. В самом деле, масштабирование приложения Node.js путем клонирования является относительно простым и часто реализуется, даже если нет необходимости задействовать больше ресурсов, только в целях обеспечения избыточности и устойчивости к сбоям.

Кроме того, это заставляет разработчиков рассматривать возможность масштабирования на самых ранних этапах разработки приложения и убедиться, что приложение не полагается на ресурсы, которые не могут совместно использоваться несколькими процессами или машинами. Экземпляры приложения не должны хранить информацию на ресурсах, не поддерживающих совместного использования, таких как память или диск, что фактически является обязательным условием масштабирования. Например, хранение данных о сеансе в памяти или на диске веб-сервера плохо масштабируется. Его следует заменить хранением в общей базе данных, что гарантирует доступ любого экземпляра к информации о сеансе, где бы он ни был развернут.

А теперь рассмотрим базовый механизм масштабирования приложений на платформе Node.js – модуль `cluster`.

Модуль cluster

Простейший шаблон для распределения нагрузки приложения между несколькими экземплярами, запущенными на одном компьютере, основан на модуле `cluster`, входящем в состав библиотеки ядра. Модуль `cluster` упрощает *ветвление* приложения на несколько экземпляров и автоматически распределяет между ними входящие соединения, как показано на рис. 10.2.

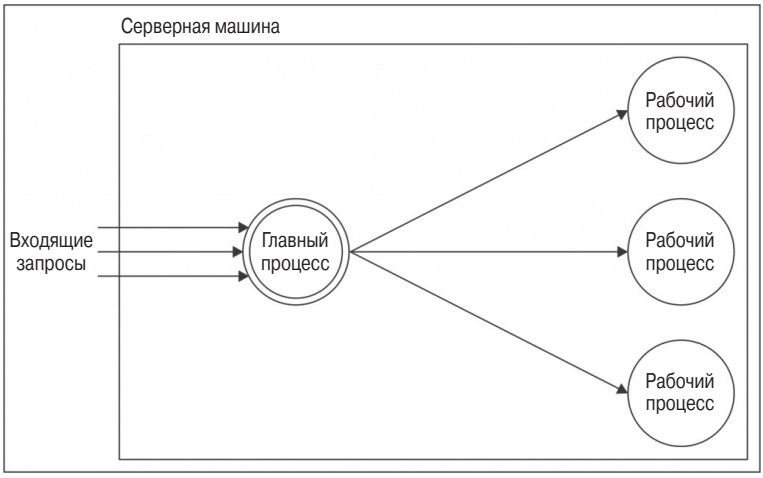

Рис. 10.2 ❖ Шаблон масштабирования ветвлением процесса

Главный процесс отвечает за запуск группы **рабочих процессов**, каждый из которых представляет экземпляр масштабируемого приложения. Затем все входящие соединения распределяются между рабочими процессами, что обеспечивает разделение нагрузки между ними.

Комментарии к модели поведения модуля cluster

В версиях Node.js 0.8 и 0.10 модуль `cluster` обеспечивает совместное использование одного и того же сервера всеми рабочими процессами и оставляет за операционной системой решение задачи *распределения нагрузки* между доступными рабочими процессами. Но этот подход не лишен недостатков; фактически алгоритмы, используемые операционной системой для распределения нагрузки между рабочими процес-

сами, не предназначены для распределения сетевых запросов, а ориентированы на планирование выполнения процессов. В результате нагрузка не всегда равномерно распределяется между всеми экземплярами, нередко некоторым рабочим процессам достается большая ее часть. Такая модель поведения характерна для планировщика операционной системы, поскольку он старается минимизировать число переключений контекста между процессами. Короче говоря, модуль `cluster` не полностью реализует свой потенциал в версиях платформы Node.js <= 0.10.

Но начиная с версии 0.11.2 ситуация изменилась, и теперь главный процесс реализует циклический алгоритм, гарантирующий равномерное распределение нагрузки между всеми рабочими процессами. Новый алгоритм включен по умолчанию на всех платформах, кроме Windows, что, впрочем, можно исправить, присвоив переменной `cluster.schedulingPolicy` константу `cluster.SCHED_RR` (циклический алгоритм) или `cluster.SCHED_NONE` (распределение нагрузки осуществляется операционной системой).

Циклический алгоритм (round robin) равномерно распределяет нагрузку между доступными серверами на циклической основе. Первый запрос передается первому серверу, второй – следующему в списке и т. д. По достижении конца списка обход начинается сначала. Это один из самых простых и наиболее часто используемых алгоритмов распределения нагрузки, но не единственный. Более сложные алгоритмы позволяют присваивать приоритеты, выбирать наименее загруженный сервер или сервер с самым коротким временем отклика. Более подробную информацию об эволюции модуля `cluster` можно найти в следующих двух дискуссиях: https://github.com/nodejs/node-v0.x-archive/issues/3241; https://github.com/nodejs/node-v0.x-archive/issues/4435.

Создание простого HTTP-сервера

А теперь перейдем к примеру. Создадим небольшой HTTP-сервер, а затем осуществим его клонирование и распределение нагрузки с помощью модуля `cluster`. Прежде всего нам понадобится приложение для масштабирования; для этого примера достаточно будет очень простого HTTP-сервера.

Создадим файл `app.js`, содержащий следующий код:

```
const http = require('http');
const pid = process.pid;

http.createServer((req, res) => {
  for (let i = 1e7; i> 0; i--) {}
    console.log(`Handling request from ${pid}`);
    res.end(`Hello from ${pid}\n`);
  }).listen(8080, () => {
  console.log(`Started ${pid}`);
});
```

В ответ на любой запрос наш HTTP-сервер будет возвращать сообщение с его идентификатором процесса (PID), что позволит определить экземпляр приложения, обработавший запрос. Кроме того, для имитации нагрузки на центральный процессор предусмотрено выполнение 10 миллионов пустых циклов, без чего сервер не будет испытывать практически никакой нагрузки, не считая небольших тестов, запущенных для нужд примера.

Масштабируемым модулем `app` может быть все, что угодно, и он может быть реализован с помощью одного из веб-фреймворков, такого, например, как Express.

Сейчас можно проверить, работает ли все это должным образом, запустив приложение как обычно и послав запрос по адресу `http://localhost:8080` с помощью браузера или утилиты `curl`.

Можно также измерить, сколько запросов в секунду сможет обработать сервер с использованием единственного процесса. Для этого можно воспользоваться сетевыми инструментами оценки, такими как `siege` (http://www.joedog.org/siege-home) или `ab` от Apache (http://httpd.apache.org/docs/2.4/programs/ab.html):

```
siege -c200 -t10S http://localhost:8080
```

Команда с `ab` выглядит аналогично:

```
ab -c200 -t10 http://localhost:8080/
```

Предыдущие команды создадут 200 параллельных соединений с сервером в течение 10 секунд. Для справки заметим, что в системе с 4 процессорами результат составляет порядка 90 транзакций в секунду, среднее использование центрального процессора составляет лишь 20%.

Имейте в виду, что описываемые в этой главе нагрузочные тесты намеренно сделаны простыми и минимальными, поскольку преследуют исключительно учебные и справочные цели. Их результаты не гарантируют 100%-ную точную оценку эффективности анализируемых методов.

Масштабирование с помощью модуля cluster

А теперь попробуем масштабировать приложение с помощью модуля `cluster`. Создадим новый модуль `clusteredApp.js`:

```
const cluster = require('cluster');
const os = require('os');

if(cluster.isMaster) {
  const cpus = os.cpus().length;
  console.log(`Clustering to ${cpus} CPUs`);
  for (let i = 0; i<cpus; i++) {      // [1]
    cluster.fork();
  }
} else {
  require('./app');                   // [2]
}
```

Как видите, использование модуля `cluster` не требует особых усилий. Поясним происходящее:

- запуская модуль `clusteredApp` из командной строки, мы фактически запускаем главный процесс. Переменная `cluster.isMaster` автоматически получает значение `true`, и вся работа сводится к ветвлению текущего процесса вызовом метода `cluster.fork()`. В предыдущем примере определяется количество процессоров в системе и запускается равное количество рабочих процессов, что позволяет задействовать все доступные вычислительные мощности;
- вызов метода `cluster.fork()` в главном процессе производит повторный запуск все того же главного модуля (`clusteredApp`), но на этот раз в виде рабочего процесса (переменная `cluster.isWorker` получает значение `true`, а переменная `cluster.isMaster` – значение `false`). Когда приложение запускается как рабочий

процесс, оно приступает к выполнению определенной работы. В данном примере загружается модуль app, что приводит к запуску нового HTTP-сервера.

Важно не забывать, что каждый рабочий процесс – это отдельный процесс Node.js с собственным циклом событий, пространством памяти и загруженными модулями.

Интересно отметить, что использование модуля cluster основывается на рекурентном шаблоне, упрощающем запуск нескольких экземпляров приложения:

```
if(cluster.isMaster) {
  // fork()
} else {
  //выполнение работы
}
```

Внутренне модуль cluster использует функцию child_process.fork() (рассматривалась в *главе 9* «*Дополнительные рецепты асинхронной обработки*»), поэтому мы автоматически получаем коммуникационный канал между главным и рабочими процессами. Экземпляры рабочих процессов доступны через переменную cluster.workers, поэтому для отправки всем им сообщения достаточно следующих строк кода:

```
Object.keys(cluster.workers).forEach(id => {
  cluster.workers[id].send('Hello from the master');
});
```

А теперь давайте запустим HTTP-сервер в режиме кластера. Для этого потребуется запустить модуль clusteredApp, как обычно:

```
node clusteredApp
```

Если в машине имеется несколько процессоров, в терминале должно появиться несколько сообщений, друг за другом, о запуске рабочих процессов. Например, в системе с четырьмя процессорами в терминале появятся примерно такие строки:

```
Started 14107
Started 14099
Started 14102
Started 14101
```

Если теперь отправить серверу несколько запросов (по адресу http://localhost:8080), они вернут сообщения с разными идентификаторами PID, что подтвердит обработку запросов разными рабочими процессами, между которыми распределяется нагрузка.

Теперь снова попробуем запустить нагрузочный тест сервера:

```
siege -c200 -t10S http://localhost:8080
```

Этот тест должен выявить увеличение производительности благодаря масштабированию приложения с применением нескольких процессов. Для справки отметим, что при использовании Node.js 6 в системе Linux с 4 процессорами производительность должна возрасти примерно в три раза (270 транзакций в секунду, по сравнению с 90 транзакциями) со средней нагрузкой на центральный процессор 90%.

Обеспечение отказоустойчивости и высокой доступности с помощью модуля cluster

Как упоминалось ранее, масштабирование дает также другие преимущества, в частности позволяет гарантировать определенный уровень обслуживания, даже при наличии неисправностей и сбоев. Это свойство называется **отказоустойчивостью** и влияет на доступность системы.

Запуская несколько экземпляров одного и того же приложения, мы формируем избыточную систему, то есть если один из экземпляров прекратит работу, другие продолжат обслуживание запросов. Эту схему легко реализовать с помощью модуля `cluster`. Давайте посмотрим, как она работает!

За основу возьмем код из предыдущего раздела. В частности, изменим модуль `app.js` так, чтобы он завершал работу по истечении случайного интервала времени:

```
// ...
// В конце модуля app.js
setTimeout(() => {
  throw new Error('Ooops');
}, Math.ceil(Math.random() * 3) * 1000);
```

Теперь сервер будет завершать работу с ошибкой по истечении случайного количества секунд в интервале от 1 до 3. В реальной ситуации это вызовет остановку работы приложения и, конечно же, прекращение обслуживания запросов, если не использовать какого-то внешнего инструмента для наблюдения за состоянием приложения и автоматического перезапуска. Но при наличии единственного экземпляра возникнет задержка в обслуживании, вызванная затратами времени на запуск приложения. Это означает, что на время повторного запуска приложение будет недоступно. Наличие же нескольких экземпляров гарантирует бесперебойное обслуживание входящих запросов даже при прекращении работы одного из рабочих процессов.

При использовании модуля `cluster` достаточно реализовать запуск нового рабочего процесса, когда обнаружится, что один из рабочих процессов завершился с ошибкой. Добавим эту возможность в модуль `clusteredApp.js`:

```
if(cluster.isMaster) {
  // ...

  cluster.on('exit', (worker, code) => {
    if(code != 0 && !worker.suicide) {
      console.log('Worker crashed. Starting a new worker');
      cluster.fork();
    }
  });
} else {
  require('./app');
}
```

В этом случае главный процесс, получив событие `'exit'`, проверит, как завершился процесс – умышленно или в результате ошибки. Для этого проверяются код состояния `code` и флаг `worker.exitedAfterDisconnect`, который указывает, что рабочий процесс был завершен по инициативе главного процесса. Если подтвердится, что процесс завершился из-за ошибки, запускается новый рабочий процесс. Следует отметить, что пока повторно запускается рабочий процесс, завершившийся по ошибке, другие

рабочие процессы продолжают обслуживать запросы, что обеспечивает доступность приложения.

Для проверки этого можно выполнить стресс-тест сервера с помощью `siege`. По завершении стресс-теста среди прочих показателей, которые выводит `siege`, можно найти оценку доступности приложения:

```
Transactions:           3027 hits
Availability:           99.31 %
[...]
Failed transactions:         21
```

Имейте в виду, что у вас может получиться совершенно иной результат, поскольку он во многом зависит от количества выполняемых экземпляров и того, сколько раз они завершались в процессе тестирования, но он хорошо отражает качество работы реализованного решения. Приведенные выше цифры говорят о том, что, несмотря на постоянное завершение рабочих процессов по ошибке, только 21 запрос из 3027 не был обслужен. В этом примере большинство неудачных запросов не было обработано из-за разрыва уже установленных соединений. Когда такое происходит, `siege` выводит следующее сообщение об ошибке:

```
[error] socket: read error Connection reset by peer sock.c:479: Connection reset by peer
```

К сожалению, предотвратить отказы такого вида практически невозможно, особенно когда приложение завершает работу из-за аварии. Тем не менее рассмотренное решение неплохо поддерживает высокую доступность приложения, которое так часто завершается из-за ошибок!

Перезапуск без простоя

Может возникнуть ситуация, когда приложение для Node.js требуется перезапустить для обновления кода. Наличие нескольких экземпляров сможет помочь обеспечить постоянную доступность приложения и в этом случае.

При намеренном перезапуске приложения в целях обновления на повторный запуск приложения тратится определенное время, в течение которого оно не в состоянии обслуживать запросы. Это может быть приемлемо при обновлении персонального блога, но совершенно недопустимо для профессионального применения с **соглашением об уровне обслуживания** (Service Level Agreement, SLA) или для очень часто обновляемого приложения, включенного в процесс непрерывного развертывания. Решением этой проблемы является реализация **перезапуска без простоя**, когда обновление кода приложения не наносит ущерба его доступности.

И снова эта задача легко решается с помощью модуля `cluster`. Суть решения заключается в том, чтобы перезапускать рабочие процессы *по одному*. В этом случае остальные рабочие процессы продолжат выполняться и обслуживать приложение.

Добавим эту новую функцию в реализацию сервера. Для этого потребуется лишь добавить новый код, который должен выполняться в главном процессе (файл `clusteredApp.js`):

```
if (cluster.isMaster) {
  // ...

  process.on('SIGUSR2', () => {//[1]
  const workers = Object.keys(cluster.workers);

  function restartWorker(i) { //[2]
```

```
    if (i >= workers.length) return;
    const worker = cluster.workers[workers[i]];
    console.log(`Stopping worker: ${worker.process.pid}`);
    worker.disconnect(); //[3]

    worker.on('exit', () => {
      if (!worker.suicide) return;
        const newWorker = cluster.fork(); //[4]
        newWorker.on('listening', () => {
          restartWorker(i + 1); //[5]
        });
      });
    }
    restartWorker(0);
  });
} else {
  require('./app');
}
```

Вот как работает этот блок кода:

1. Перезапуск рабочих процессов начинается после получения сигнала `SIGUSR2`.
2. Новая функция `restartWorker()` реализует шаблон асинхронных последовательных итераций по элементам объекта `cluster.workers`.
3. Первая задача функции `restartWorker()` – безопасно остановить рабочий процесс вызовом метода `worker.disconnect()`.
4. После завершения процесса запускается новый рабочий процесс.
5. Только когда новый рабочий процесс будет готов к приему новых соединений, можно перейти к следующей итерации и приступить к перезапуску очередного рабочего процесса.

Так как эта программа использует UNIX-сигналы, она не будет работать в Windows (если не использовать подсистему Linux, недавно появившуюся в Windows 10). Сигналы являются простейшим механизмом реализации этого решения, но не единственным; фактически имеются другие способы, такие как обработка команд, поступающих из сокета, канала или стандартного ввода.

Теперь можно проверить перезагрузку без простоя, запустив модуль `clusteredApp` и отправив ему сигнал `SIGUSR2`. Однако сначала необходимо узнать идентификатор PID главного процесса. Это можно сделать с помощью следующей команды:

```
ps af
```

Главный процесс должен быть родителем ряда группы процессов. Определив PID, можно послать сигнал процессу:

```
kill -SIGUSR2 <PID>
```

Теперь приложение `clusteredApp` должно вывести примерно такие строки:

```
Restarting workers
Stopping worker: 19389
Started 19407
Stopping worker: 19390
Started 19409
```

Можем вновь использовать `siege`, чтобы убедиться, что перезагрузка рабочих процессов не оказывает существенного влияния на доступность приложения.

pm2 (https://github.com/Unitech/pm2) – небольшая утилита на основе модуля cluster, которая обеспечивает распределение нагрузки, мониторинг процессов, перезапуск без простоя и другие полезные возможности.

Взаимодействия с сохранением состояния

Модуль `cluster` неспособен обрабатывать взаимодействия с сохранением состояния, когда это состояние поддерживается приложением и не используется совместно разными экземплярами. Это вызвано тем, что разные запросы, принадлежащие одному сеансу с сохранением состояния, потенциально могут обрабатываться разными экземплярами приложения. Это проблема не только модуля `cluster`, она характерна для любого алгоритма распределения нагрузки без сохранения состояния.

Рассмотрим, например, ситуацию на рис. 10.3.

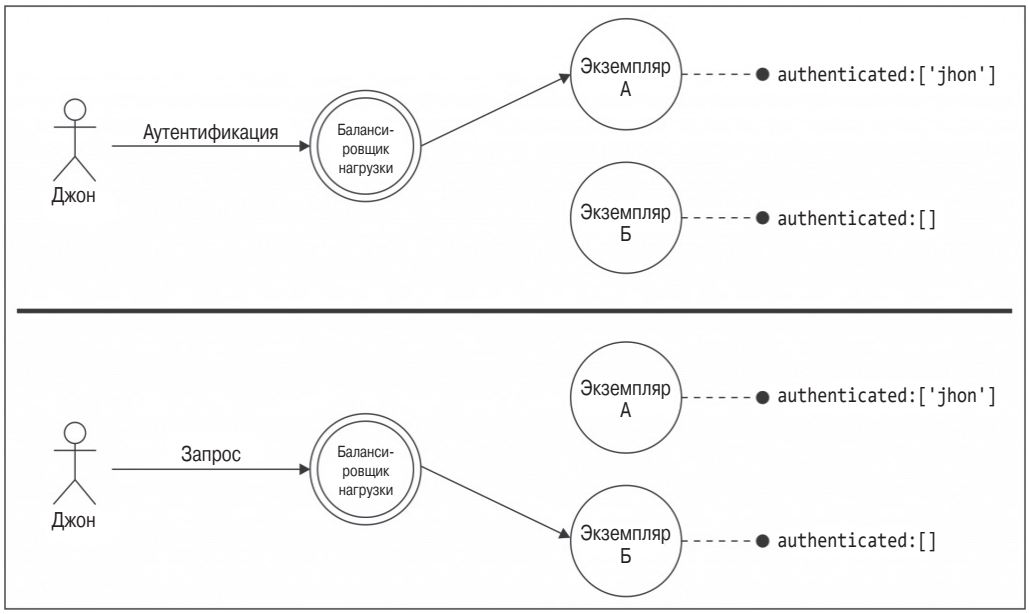

Рис. 10.3 ❖ Некорректное обслуживание взаимодействий с сохранением состояния

Пользователь **Джон** сначала отправляет приложению запрос для своей аутентификации, но результат операции регистрируется локально (например, в памяти), поэтому только экземпляр приложения, получивший запрос для аутентификации (**экземпляр А**), будет знать, что Джон успешно прошел проверку. Когда Джон отправит новый запрос, балансировщик нагрузки может передать его другому экземпляру приложения, которому неизвестно, что Джон успешно прошел проверку и, следовательно, откажется выполнить операцию. Подобные приложения невозможно масштабировать, но, к счастью, существуют два простых решения этой проблемы.

Совместное использование состояния несколькими экземплярами

Первое решение: масштабируемое приложение, сохраняющее состояние взаимодействий, должно обеспечить совместное его использование всеми экземплярами. Для этого потребуется общее хранилище данных, такое как база данных, например PostgreSQL (http://www.postgresql.org), MongoDB (http://www.mongodb.org) или CouchDB (http://couchdb.apache.org), или, что еще лучше, хранилище в памяти, такое как Redis (http://redis.io) или Memcached (http://memcached.org).

Схема на рис. 10.4 иллюстрирует это простое и эффективное решение.

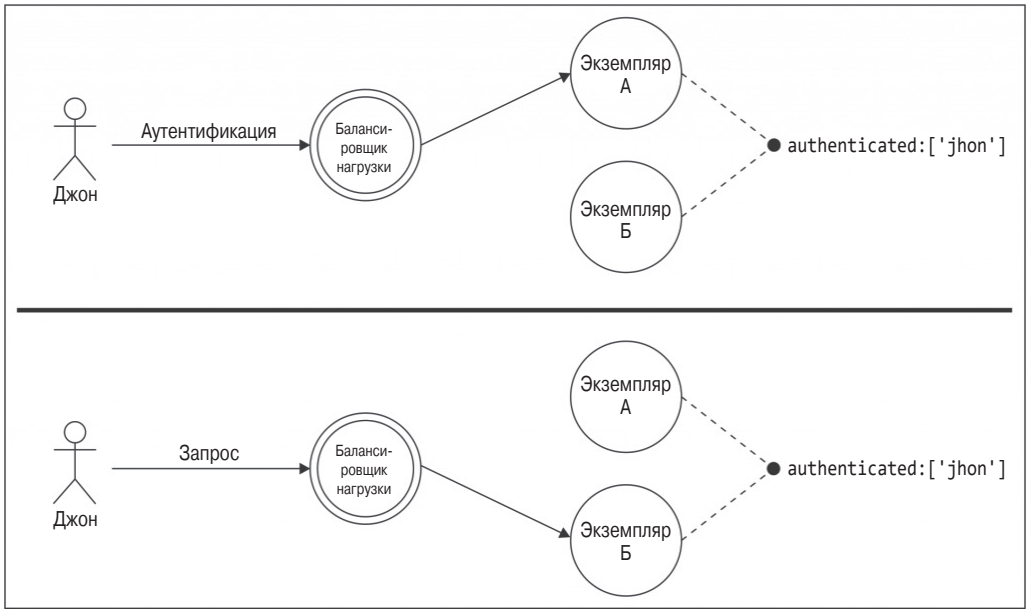

Рис. 10.4 ❖ Корректное обслуживание взаимодействий с сохранением состояния

Единственный недостаток общего хранилища состояний состоит в том, что его можно использовать не всегда, например когда применяется готовая библиотека, которая хранит состояния в памяти. Во всяком случае, это решение требует изменения кода существующих приложений (если оно еще не реализовано). Как будет показано ниже, существует менее агрессивное решение.

Распределение нагрузки с привязкой

Другой вариант решения проблемы сохранения состояний заключается в том, что при распределении нагрузки все запросы, связанные с одним сеансом, всегда должны передаваться одному и тому же экземпляру приложения. Этот метод называется **распределением нагрузки с привязкой**.

Схема на рис. 10.5 иллюстрирует применение этой технологии.

Как показано на рис. 10.5, когда балансировщик нагрузки получает запрос на создание нового сеанса, он связывает этот сеанс с одним конкретным экземпляром, выбранным алгоритмом распределения нагрузки. В следующий раз, получив запрос

в рамках того же сеанса, балансировщик минует алгоритм распределения нагрузки и выберет тот экземпляр приложения, который был ранее связан с сеансом. Описанная только что технология включает проверку идентификатора сеанса (как правило, включаемого в cookie приложением или самим балансировщиком нагрузки).

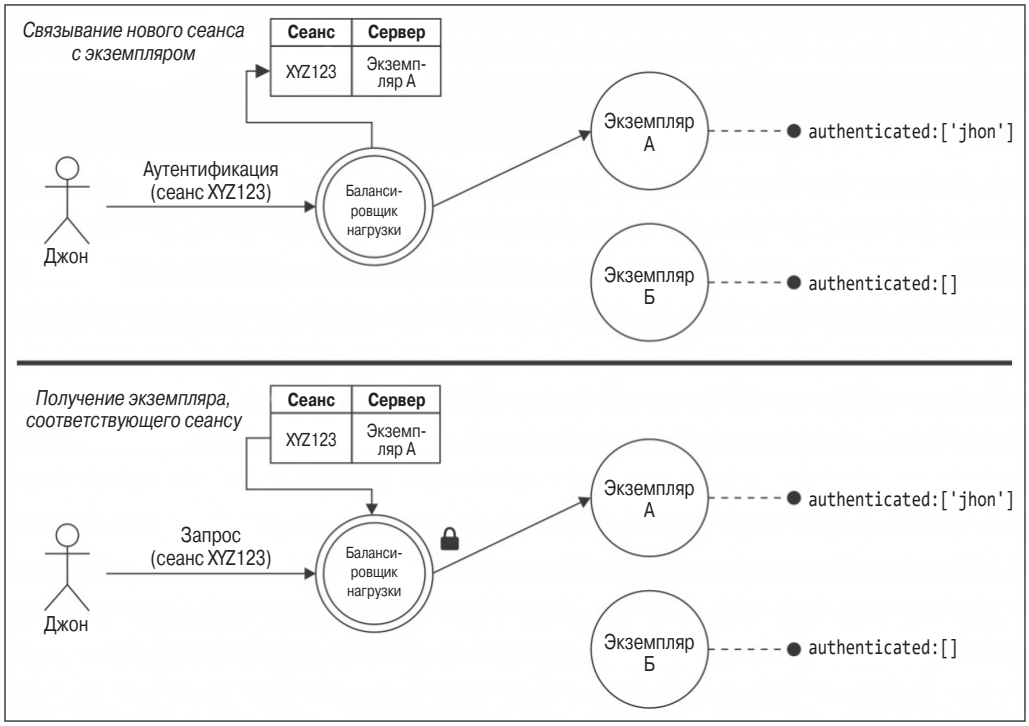

Рис. 10.5 ❖ Распределение нагрузки с привязкой к экземпляру

Более простой альтернативой связывания соединения с определенным сервером является использование IP-адреса клиента. Обычно IP-адрес передается хеш-функции, которая генерирует идентификатор, соответствующий экземпляру приложения, который должен обработать запрос. Этот метод не требует хранения таблицы соответствий в балансировщике нагрузки. Однако он не работает с устройствами, которые часто изменяют IP-адрес, например при перемещении по различным сетям.

Распределение нагрузки с привязкой не поддерживается по умолчанию модулем cluster, но эту возможность можно добавить с помощью npm-библиотеки sticky-session (https://www.npmjs.org/package/sticky-session).

Самая большая проблема распределения нагрузки с привязкой заключается в том, что оно сводит на нет преимущества избыточной системы, где все экземпляры приложения равноценные и где любой из экземпляров может заменить тот, что перестал работать. По этой причине рекомендуется всегда избегать использования распределения нагрузки с привязкой, отдавая предпочтение созданию приложений, которые хранят состояние сеансов в общем хранилище или вообще не требуют сохранения состояния (например, путем включения состояния в сам запрос).

Реальным примером библиотеки, требующей распределения нагрузки с привязкой, является Socket.io (http://socket.io/blog/introducing-socket-io-1-0/#scalability).

Масштабирование с помощью обратного проксирования

Применение модуля `cluster` – не единственный способ масштабирования веб-приложений на платформе Node.js. Фактически *традиционные* методы часто оказываются более предпочтительными, поскольку обеспечивают более полный контроль и широкие возможности в окружениях с высокой доступностью.

Альтернативой использованию модуля `cluster` является запуск нескольких *автономных экземпляров* одного и того же приложения, прослушивающих разные порты или выполняющихся на разных машинах, и использование *обратного прокси-сервера* (или *шлюза*) для доступа к этим экземплярам и распределения трафика между ними. В такой конфигурации отсутствует главный процесс, распределяющий запросы между рабочими процессами, вместо этого имеется множество автономных процессов, запущенных на одном компьютере (и прослушивающих разные порты) или разбросанных по нескольким компьютерам в сети. Роль единой точки доступа к приложению играет обратный прокси-сервер – специальное устройство или служба, – находящийся между клиентами и экземплярами приложения, который принимает все запросы, передает их конечному серверу и возвращает результат клиенту, как будто он сам его создал. В этом случае обратный прокси-сервер используется также в качестве балансировщика нагрузки, распределяя запросы между экземплярами приложения.

Подробнее о различиях между обратными и обычными прокси-серверами можно узнать в документации HTTP-сервера Apache на странице http://httpd.apache.org/docs/2.4/mod/mod_proxy.html#forwardreverse.

На рис. 10.6 изображена схема типичной конфигурации обратного прокси-сервера, выступающего в роли балансировщика нагрузки между несколькими процессами, запущенными на нескольких компьютерах.

Можно назвать много причин, почему в приложениях для Node.js имеет смысл выбирать этот подход вместо модуля `cluster`:

- обратный прокси-сервер способен распределять нагрузку между несколькими компьютерами, не только между несколькими процессами;
- большинство популярных обратных прокси-серверов поддерживает распределение нагрузки с привязкой;
- обратный прокси-сервер может передавать запросы любым доступным серверам, независимо от платформы и языка программирования;
- появляется возможность использовать более мощные алгоритмы распределения нагрузки;
- многие обратные прокси-серверы предлагают дополнительные услуги, такие как изменение URL-адресов, кэширование, поддержку SSL и даже возможность создания полноценных веб-серверов, которые можно использовать, например, для обслуживания статических файлов.

Кроме того, при необходимости модуль `cluster` можно без особых усилий объединить с обратным прокси-сервером; например, модуль `cluster` можно использовать для масштабирования по вертикали в отдельных компьютерах, а обратный прокси-сервер – для масштабирования по горизонтали по нескольким узлам.

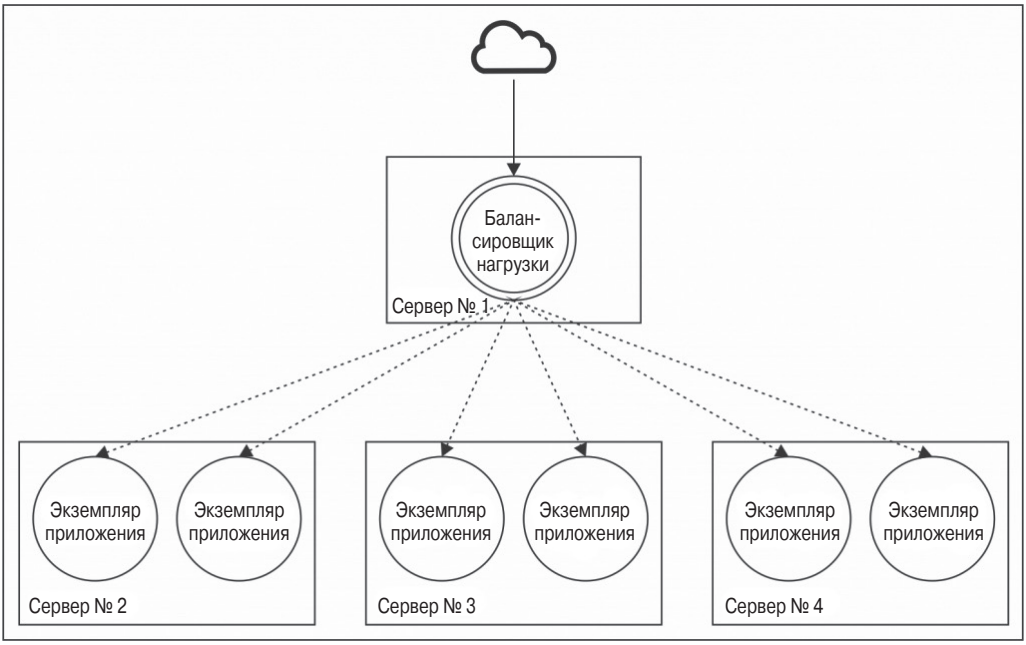

Рис. 10.6 ❖ Типичная конфигурация обратного прокси-сервера

Шаблон
Обратный прокси-сервер используется для распределения нагрузки между несколькими экземплярами приложения, прослушивающими разные порты или запущенными на разных машинах.

Имеется множество вариантов реализации распределения нагрузки с помощью обратного прокси-сервера, вот несколько самых популярных из них:

- **Nginx** (http://nginx.org): веб-сервер, обратный прокси-сервер и балансировщик нагрузки, основанный на модели неблокирующего ввода-вывода;
- **HAProxy** (http://www.haproxy.org): скоростной балансировщик нагрузки для TCP/HTTP-трафика;
- **прокси-серверы на платформе Node.js**: имеется множество решений для реализации обратных прокси-серверов и распределения нагрузки непосредственно на платформе Node.js. Эти решения имеют свои преимущества и недостатки, как будет показано позже;
- **облачные прокси-серверы**: в эпоху облачных вычислений средства распределения нагрузки нередко используются как службы. Это удобно, поскольку такие решения требуют минимального обслуживания, отлично масштабируются и поддерживают динамическую настройку для осуществления масштабирования по требованию.

В следующих нескольких разделах этой главы мы рассмотрим конфигурацию с использованием сервера Nginx, а затем рассмотрим пример создания собственного балансировщика нагрузки с использованием только средств самой платформы Node.js!

Распределение нагрузки с помощью Nginx

Для демонстрации работы реверсного прокси-сервера соберем масштабируемую архитектуру на основе сервера Nginx (http://nginx.org), но сначала установим его. Это можно сделать, следуя инструкциям, приведенным на странице http://nginx.org/en/docs/install.html.

В последних версиях Ubuntu сервер Nginx можно установить командой

```
sudo apt-get install nginx
```

В Mac OS X можно использовать brew (http://brew.sh):

```
brew install nginx
```

Поскольку мы не собираемся использовать `cluster` для запуска нескольких экземпляров сервера, нужно изменить код приложения так, чтобы можно было указать прослушиваемый порт с помощью аргумента командной строки. Это позволит запустить несколько экземпляров, прослушивающих разные порты. Рассмотрим главный модуль примера приложения (`app.js`):

```
const http = require('http');
const pid = process.pid;

http.createServer((req, res) => {
  for (let i = 1e7; i> 0; i--) {}
  console.log(`Handling request from ${pid}`);
  res.end(`Hello from ${pid}\n`);
}).listen(process.env.PORT || process.argv[2] || 8080, () => {
  console.log(`Started ${pid}`);
});
```

Еще одной важной особенностью, отсутствующей из-за отказа от использования модуля `cluster`, является автоматический перезапуск в случае аварийного завершения. К счастью, это легко исправить с помощью выделенного супервизора – внешнего процесса, осуществляющего мониторинг и повторный запуск приложения при необходимости. Вот возможные кандидаты на роль такого супервизора:

- супервизоры для платформы Node.js, такие как `forever` (https://npmjs.org/package/forever) и `pm2` (https://npmjs.org/package/pm2);
- мониторы для конкретной операционной системы, такие как `upstart` (http://upstart.ubuntu.com), `systemd` (http://freedesktop.org/wiki/Software/systemd) и `runit` (http://smarden.org/runit/);
- более продвинутые решения мониторинга, такие как `monit` (http://mmonit.com/monit) и `supervisor` (http://supervisord.org).

В этом примере будет использоваться супервизор `forever`, как наиболее простой. Его можно установить глобально, командой

```
npm install forever -g
```

Следующий шаг: запуск под надзором супервизора `forever` четырех экземпляров приложения, прослушивающих разные порты:

```
forever start app.js 8081
forever start app.js 8082
forever start app.js 8083
forever start app.js 8084
```

Список запущенных процессов можно получить командой

```
forever list
```

Теперь настроим Nginx-сервер для использования в качестве балансировщика нагрузки.

Сначала нужно отыскать файл `nginx.conf`, который может находиться в одном из следующих каталогов, в зависимости от системы: `/usr/local/nginx/conf`, `/etc/nginx` или `/usr/local/etc/nginx`.

Откроем файл `nginx.conf` и применим следующую минимальную конфигурацию, необходимую для работы балансировщика:

```
http {
  # ...
  upstream nodejs_design_patterns_app {
    server 127.0.0.1:8081;
    server 127.0.0.1:8082;
    server 127.0.0.1:8083;
    server 127.0.0.1:8084;
  }
  # ...
  server {
    listen 80;

    location / {
      proxy_pass http://nodejs_design_patterns_app;
    }
  }
  # ...
}
```

Приведенная конфигурация требует небольших пояснений. В разделе `nodejs_design_patterns_app` определяется список внутренних серверов для обработки сетевых запросов. Директива `proxy_pass`, в разделе `server`, требует от сервера Nginx перенаправлять любые запросы в группу серверов, определенную выше (`nodejs_design_patterns_app`). Это все. Осталось только перезагрузить конфигурацию сервера Nginx командой

```
nginx -s reload
```

Теперь система должна быть готова принимать запросы и распределять трафик между четырьмя экземплярами приложения Node.js. Открыв в браузере страницу http://localhost, можно увидеть, как происходит распределение трафика сервером Nginx.

Использование реестра служб

Одним из важных преимуществ современной облачной инфраструктуры является возможность динамической регулировки пропускной способности приложения на основании текущего или прогнозируемого трафика; она называется **динамическим масштабированием**. При должной реализации эта технология способна значительно сократить расходы на вычислительную инфраструктуру при сохранении высокого уровня доступности приложения.

Ее идея достаточно проста: если приложение снижает производительность из-за роста нагрузки, автоматически подключаются серверы, помогающие справиться с этой повышенной нагрузкой. Поддерживается также возможность остановки некоторых серверов в определенные часы, например ночью, когда трафик существенно меньше, и запуска их снова по утрам. Этот механизм требует распределения нагрузки в зависимости от актуальной топологии сети, чтобы знать, когда следует запускать серверы.

Общий шаблон решения этой проблемы заключается в использовании центрального хранилища, которое называется *реестром служб* и хранит информацию о действующих серверах и предоставляемых ими услугах. На рис. 10.7 представлена архитектура с несколькими службами и балансировщиком нагрузки перед ними, динамически настраиваемая с помощью реестра служб.

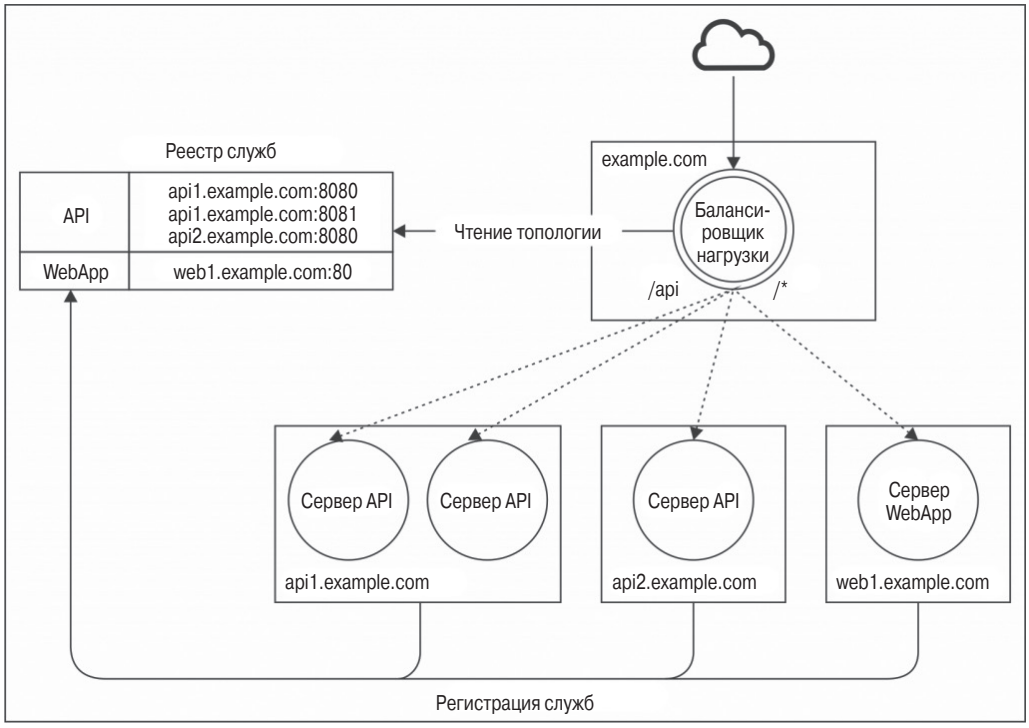

Рис. 10.7 ❖ Архитектура с реестром служб

Архитектура рис. 10.7 предполагает наличие двух служб: `API` и `WebApp`. Балансировщик нагрузки распределяет запросы, поступающие по адресу конечной точки `/api`, между всеми серверами, реализующими службу `API`, а остальные запросы – между серверами, реализующими службу `WebApp`. Список серверов балансировщик получает из реестра служб.

Для работы в автоматическом режиме все экземпляры приложения должны регистрироваться в реестре после перехода в состояние готовности принимать запросы и отменять регистрацию перед завершением работы. Благодаря этому балансировщик нагрузки всегда обладает актуальными сведениями о доступных серверах и службах.

Шаблон (реестр служб)
Используется центральное хранилище актуальных данных о доступных в системе серверах и службах.

Этот шаблон можно применять не только для распределения нагрузки, но и для изоляции служб от обслуживающих их серверов. Его можно рассматривать как применение шаблона проектирования «Локатор служб» к сетевым службам.

Динамическое распределение нагрузки с помощью http-proxy и Consul

Для поддержки динамической сетевой инфраструктуры можно использовать обратный прокси-сервер, такой как **Nginx** или **HAProxy**; в этом случае достаточно реализовать обновление конфигурации с помощью автоматизированной службы и повторное ее чтение балансировщиком нагрузки. Для сервера Nginx это можно сделать с помощью следующей команды:

```
nginx -s reload
```

Такого же результата можно достичь с помощью облачного решения, но существует третья альтернатива, основанная на платформе Node.js.

Как известно, платформа Node.js является отличным инструментом для создания любых сетевых приложений. Как уже упоминалось, это является одной из главных целей ее создания. Так почему бы не создать балансировщика нагрузки непосредственно на платформе Node.js? Это обеспечит большую свободу действий и более широкие возможности, а кроме того, позволит реализовать любой шаблон или алгоритм распределения нагрузки, включая динамическое распределение с помощью реестра служб. В этом примере в роли реестра служб будет использоваться Consul (https://www.consul.io).

Далее мы воспроизведем архитектуру с несколькими службами, изображенную на рис. 10.7 в предыдущем разделе, и используем для этого три npm-пакета:

- `http-proxy` (https://npmjs.org/package/http-proxy): библиотека, упрощающая создание прокси-серверов и средства распределения нагрузки на платформе Node.js;
- `portfinder` (https://npmjs.com/package/portfinder): библиотека, помогающая обнаруживать свободные порты в системе;
- `consul` (https://npmjs.org/package/consul): библиотека, обеспечивающая регистрацию служб в реестре Consul.

Начнем с реализации служб. Это простые HTTP-серверы, подобные тем, что использовались для тестирования модуля `cluster` и сервера Nginx, но на этот раз необходимо, чтобы каждый сервер при запуске регистрировал себя в реестре служб.

Посмотрим, как это выглядит (файл `app.js`):

```
const http = require('http');
const pid = process.pid;
const consul = require('consul')();
const portfinder = require('portfinder');
const serviceType = process.argv[2];

portfinder.getPort((err, port) => {       // [1]
  const serviceId = serviceType+port;
  consul.agent.service.register({         // [2]
    id: serviceId,
```

```
    name: serviceType,
    address: 'localhost',
    port: port,
    tags: [serviceType]
  }, () => {
    const unregisterService = (err) => {   // [3]
      consul.agent.service.deregister(serviceId, () => {
        process.exit(err ? 1 : 0);
      });
    };

    process.on('exit', unregisterService); // [4]
    process.on('SIGINT', unregisterService);
    process.on('uncaughtException', unregisterService);

    http.createServer((req, res) => {      // [5]
      for (let i = 1e7; i> 0; i--) {}
      console.log(`Handling request from ${pid}`);
      res.end(`${serviceType} response from ${pid}\n`);
    }).listen(port, () => {
      console.log(`Started ${serviceType} (${pid}) on port ${port}`);
    });
  });
});
```

Здесь следует пояснить некоторые фрагменты приведенного выше кода.

- Во-первых, чтобы получить свободный порт, вызывается метод `portfinder.getPort` (по умолчанию `portfinder` начинает поиск с порта 8000).
- Далее, с помощью библиотеки Consul выполняется регистрация новой службы в реестре. Для этого требуется указать несколько атрибутов: `id` (уникальное имя службы), `name` (общее имя, идентифицирующее службу), `address` и `port` (для доступа к службе), `tags` (необязательный массив тегов, используемый для фильтрации и группировки служб). Имя службы и теги определяются с помощью `serviceType` (получаемого как аргумент командной строки). Это позволит идентифицировать все службы одного типа, доступные в кластере.
- Затем определяется функция `unregisterService`, которая позволит удалить службу, зарегистрированную в реестре Consul.
- Функция `unregisterService` вызывается в момент завершения программы (намеренного или случайного), для отмены регистрации в реестре служб Consul.
- И наконец, выполняется запуск HTTP-сервера, прослушивающего порт, обнаруженный с помощью `portfinder`.

Теперь реализуем распределение нагрузки. Для этого создадим новый модуль `loadBalancer.js`. Сначала определим таблицу соответствия URL-путей и служб:

```
const routing = [
  {
    path: '/api',
    service: 'api-service',
    index: 0
  },
  {
    path: '/',
```

```
    service: 'webapp-service',
    index: 0
  }
];
```

Каждый элемент массива `routing` содержит службу `service` для обработки запросов, поступающих на данный адрес. Свойство `index` будет использоваться циклическим алгоритмом для передачи запросов службам.

Посмотрим, как это работает, реализовав вторую часть модуля `loadbalancer.js`:

```
const http = require('http');
const httpProxy = require('http-proxy');
const consul = require('consul')();                    // [1]

const proxy = httpProxy.createProxyServer({});
http.createServer((req, res) => {
  let route;
  routing.some(entry => {                               // [2]
    route = entry;
    //Начинается с указанного пути?
    return req.url.indexOf(route.path) === 0;
  });

  consul.agent.service.list((err, services) => {       // [3]
    const servers = [];
    Object.keys(services).filter(id => { //
      if (services[id].Tags.indexOf(route.service) > -1) {
        servers.push(`http://${services[id].Address}:${services[id].Port}`)
      }
    });

    if (!servers.length) { res.writeHead(502);
      return res.end('Bad gateway');
    }

    route.index = (route.index + 1) % servers.length; // [4]
    proxy.web(req, res, {target: servers[route.index]});
  });
}).listen(8080, () => console.log('Load balancer started on port 8080'));
```

Здесь реализуется балансировщик нагрузки, основанный на платформе Node.js.

1. Во-первых, для получения доступа к реестру выполняется загрузка модуля `consul`. Затем создается экземпляр объекта `http-proxy` и запускается обычный веб-сервер.
2. В обработчике запроса сервера первым делом выполняется поиск URL-адреса в таблице маршрутизации. В результате извлекается дескриптор, содержащий имя службы.
3. С помощью модуля `consul` извлекается список серверов, реализующих требуемые службы. Если этот список пуст, клиенту возвращается ошибка. Атрибут `Tag` используется для фильтрации всех доступных служб и поиска адресов серверов, реализующих данный вид службы.
4. И наконец, можно передать запрос по назначению. Изменяется `route.index`, чтобы он указывал на следующий сервер в списке, в соответствии с циклическим алгоритмом. Затем с помощью индекса сервер выбирается из списка и переда-

ется в вызов `proxy.web()` вместе с объектами запроса (`req`) и ответа (`res`). Это обеспечивает отправку запроса выбранному серверу.

Как видите, нам без труда удалось реализовать распределение нагрузки с помощью только средств платформы Node.js и реестра служб, обеспечив при этом максимальную гибкость. Теперь все готово для ее запуска, но сначала установим сервер `consul`, руководствуясь официальной документацией, которую можно найти на странице https://www.consul.io/intro/getting-started/install.html.

Это позволит запустить реестр служб `consul` с помощью следующей команды:

```
consul agent -dev
```

Теперь можно запустить балансировщик нагрузки:

```
node loadBalancer
```

Если теперь попытаться получить доступ к любой службе, находящейся за балансировщиком нагрузки, будет возвращена HTTP-ошибка с кодом 502, поскольку еще не запущено ни одного сервера. Это можно проверить с помощью команды

```
curl localhost:8080/api
```

Предыдущая команда должна вернуть следующий результат:

```
Bad Gateway
```

Ситуация изменится после запуска нескольких экземпляров служб, например двух служб `api-service` и одной службы `webapp-service`:

```
forever start app.js api-service
forever start app.js api-service
forever start app.js webapp-service
```

Теперь балансировщик должен автоматически определить наличие новых серверов и начать распределение запросов между ними. Попробуем еще раз с помощью следующей команды:

```
curl localhost:8080/api
```

Предыдущая команда должна вернуть следующее:

```
api-service response from 6972
```

Если выполнить ее повторно, должно появиться сообщение от другого сервера, что подтвердит, что запросы распределяются равномерно между разными серверами:

```
api-service response from 6979
```

Преимущества этого шаблона очевидны. Теперь можно масштабировать инфраструктуру динамически, по требованию или по расписанию, и балансировщик нагрузки будет автоматически корректировать свою работу в соответствии с новой конфигурацией без каких-либо дополнительных усилий!

Одноранговое распределение нагрузки

При организации доступа из общедоступной сети, такой как Интернет, к внутренней сети со сложной архитектурой практически всегда необходимо использовать обратный прокси-сервер. Это помогает скрыть сложность внутренней сети, обеспечив еди-

ную точку доступа к ней, что облегчит ее использование внешними приложениями. Но если требуется масштабировать службу, предназначенную только для внутреннего использования, можно добиться гораздо большей гибкости и контроля.

Предположим, что имеется **служба А**, которая реализует свои функции, опираясь на **службу Б**. **Служба Б** масштабируется по нескольким компьютерам и доступна только во внутренней сети. Знания, полученные к данному моменту, подсказывают нам, что **служба А** должна подключаться к **службе Б** с помощью обратного прокси-сервера, распределяющего трафик между всеми серверами, где запущена **служба Б**.

Но есть альтернативное решение. Можно обратный прокси-сервер исключить из схемы и распределять запросы на клиенте (**служба А**), который теперь будет отвечать за распределение нагрузки между различными экземплярами **службы Б**. Это возможно, только если **сервер А** знает подробности реализации серверов, предоставляющих **службу Б**, что во внутренней сети является обычной практикой. При таком подходе применяется так называемое **одноранговое распределение нагрузки**.

На рис. 10.8 приведено сравнение только что описанных двух альтернативных вариантов.

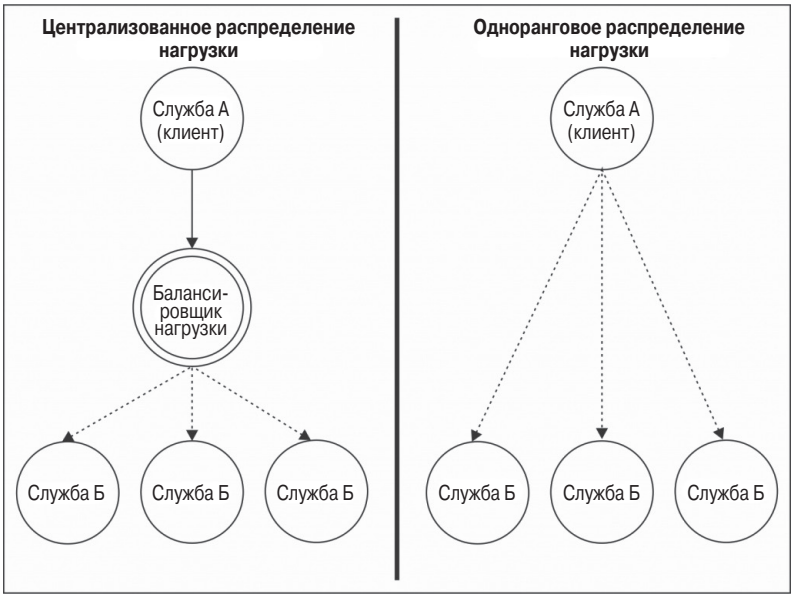

Рис. 10.8 ❖ Сравнение альтернативных вариантов

Это очень простой и эффективный шаблон, позволяющей обеспечить действительно разделенные взаимодействия без узких мест или критичных точек отказа. Кроме того, он обеспечивает следующие преимущества:

- снижение сложности инфраструктуры путем удаления сетевого узла;
- ускорение взаимодействий, поскольку маршрут передачи сообщений сокращается на один узел;
- улучшение масштабирования, так как производительность не ограничивается возможностями балансировщика нагрузки.

С другой стороны, исключение обратного прокси-сервера приводит к раскрытию всей сложной внутренней инфраструктуры. Кроме того, на любого клиента возлагается реализация алгоритма распределения нагрузки и еще, возможно, хранение актуальных сведений об инфраструктуре.

Шаблон однорангового распределения нагрузки широко используется в библиотеке ØMQ (http://zeromq.org).

Реализация распределения запросов между несколькими серверами в HTTP-клиенте

Мы уже знаем, как реализовать балансировщика нагрузки, используя только средства платформы Node.js, поэтому реализация такого же механизма на стороне клиента не должна показаться сложной задачей. Для этого потребуется только обернуть программный интерфейс клиента и добавить к нему механизм распределения нагрузки. Рассмотрим следующий модуль (`balancedRequest.js`):

```
const http = require('http');
const servers = [
  {host: 'localhost', port: '8081'},
  {host: 'localhost', port: '8082'}
];
let i = 0;

module.exports = (options, callback) => {
  i = (i + 1) % servers.length;
  options.hostname = servers[i].host;
  options.port = servers[i].port;

  return http.request(options, callback);
};
```

Предыдущий код очень прост и не требует особых пояснений. Он обертывает оригинальный программный интерфейс `http.request` и подменяет значения `hostname` и `port` в запросе значениями, выбранными из списка доступных серверов с помощью циклического алгоритма.

Новый программный интерфейс не требует изменений в клиенте (`client.js`):

```
const request = require('./balancedRequest');
for(let i = 10; i>= 0; i--) {
  request({method: 'GET', path: '/'}, res => {
    let str = '';
    res.on('data', chunk => {
      str += chunk;
    }).on('end', () => {
      console.log(str);
    });
  }).end();
}
```

Для проверки предыдущего кода запустим два экземпляра простого сервера:

```
node app 8081
node app 8082
```

Затем запустим вновь созданное клиентское приложение:

```
node client
```

Эксперимент показывает, что каждый запрос отправляется другому серверу и подтверждает возможность распределения нагрузки без выделенного обратного прокси-сервера!

Оболочку, описанную выше, можно улучшить, добавив реестр служб непосредственно в клиентское приложение, что позволит динамически обновлять список серверов. Пример реализации этого приема можно найти в сопроводительных примерах к данной книге.

Декомпозиция сложных приложений

До сих пор речь шла главным образом только о масштабировании вдоль оси *x* куба масштабирования. Мы узнали, что это самый простой и непосредственный способ распределения нагрузки, который также улучшает уровень доступности приложения. В следующем разделе мы рассмотрим масштабирование вдоль оси *y* куба масштабирования, когда приложение масштабируется путем его декомпозиции на основании функциональных особенностей и служб. Этот метод позволяет масштабировать не только пропускную способность приложения, но и, что самое главное, его сложность.

Монолитная архитектура

Увидев слово «монолитная», можно подумать, что оно описывает систему без модулей, где все службы взаимосвязаны и практически неразделимы. Однако это не всегда так. Часто монолитные системы имеют модульную архитектуру и хорошее разделение внутренних компонентов.

Прекрасным примером является ядро Linux, которое относится к категории так называемых **монолитных ядер** (что противоречит принципам экосистемы и философии Unix). Ядро Linux содержит тысячи служб и модулей, которые можно загружать и выгружать динамически, даже во время работы системы. Однако все они выполняются в *режиме ядра*, что обеспечивает неизбежный крах операционной системы при возникновении сбоя в любом из них (вы когда-нибудь наблюдали *крах ядра*?). Противоположной альтернативой является микроядерная архитектура, в которой только основные службы операционной системы работают в режиме ядра, а все остальные выполняются в пользовательском режиме и, как правило, каждый в своем собственном процессе. Основное преимущество такого подхода – в том, что сбой в любой из этих служб обычно приводит только к краху этой службы, не влияя на стабильность всей системы.

Спор Торвальдса (Torvalds) с Таненбаумом (Tanenbaum) по поводу архитектуры ядра – вероятно, один из самых жарких в истории информационных технологий, где одним из спорных является вопрос выбора монолитного или микроядерного подхода. Веб-версию этого обсуждения (она первоначально была доступна в Usenet) можно найти на странице https://groups.google.com/d/msg/comp.os.minix/wlhw16QWltI/P8isWhZ8PJ8J[1].

[1] В общих чертах суть спора излагается на странице Википедии: https://ru.wikipedia.org/wiki/Спор_Таненбаума_—_Торвальдса. – *Прим. ред.*

Примечательно, что эти принципы проектирования, которым более 30 лет, применяются по сей день и в совершенно других условиях. Современные монолитные приложения сравнимы с монолитными ядрами. Сбой в любом компоненте оказывает влияние на всю систему, то есть, если перевести в термины платформы Node.js, это означает, что все службы являются частью одной базы кода и выполняются в одном процессе (если не клонируются).

На рис. 10.9 изображен пример монолитной архитектуры.

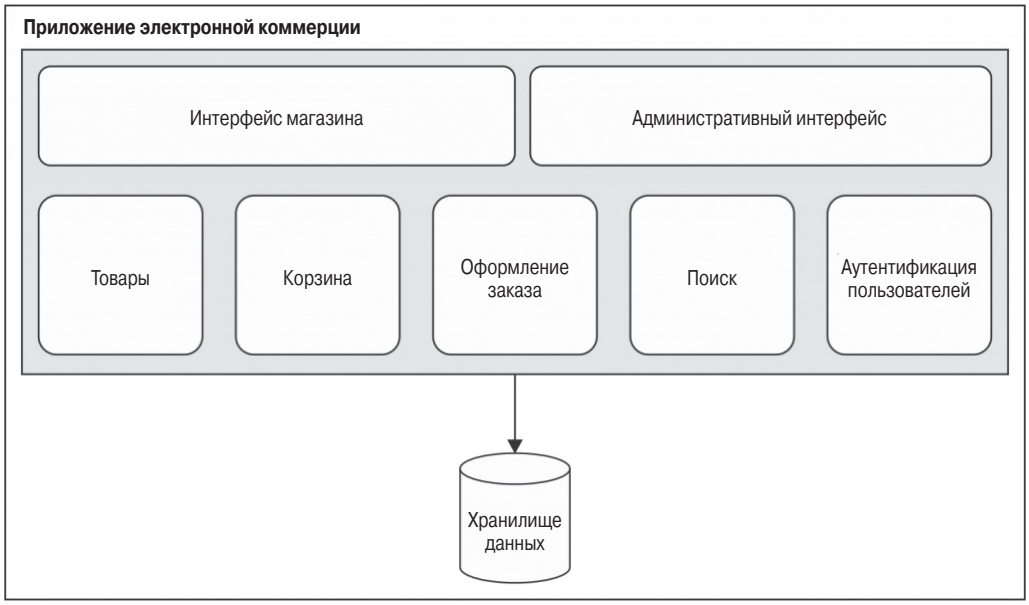

Рис. 10.9 ❖ Пример монолитной архитектуры

На рис. 10.9 изображена архитектура типичного приложения электронной коммерции. Это приложение имеет модульную структуру. В нем предусмотрены два разных интерфейса: один – основной – для магазина и второй – для администрирования. Имеется также четкое разделение служб, каждая из которых отвечает за конкретную часть бизнес-логики: **товары**, **корзина**, **оформление заказа**, **поиск** и **аутентификация пользователей**. Однако его архитектура является монолитной, поскольку все модули, по сути, являются частью одной и той же базы кода и запускаются как часть единого приложения. Сбой в любом из компонентов, например неперехваченное исключение, может привести к краху всего интернет-магазина.

Еще одна проблема архитектур этого типа – взаимосвязи между модулями. Размещение их в рамках одного приложения существенно облегчает реализацию взаимодействий и связей между модулями. Например, рассмотрим операцию покупки товара: модуль оформления заказа должен обновить признак доступности в объекте, представляющем товар, и если эти два модуля находятся в одном приложении, разработчик может просто получить ссылку на этот объект и напрямую обновить его. В монолитном приложении трудно сохранить внутренние модули в слабосвязанном состоянии, потому что границы между ними не всегда явно очерчены и часто игнорируются.

Наличие **тесных взаимосвязей** часто является основным препятствием роста приложения и делает невозможным применение к нему приемов масштабирования из-за высокой сложности. Фактически сложный граф зависимостей означает, что каждый компонент системы является критически важным; он должен поддерживаться на протяжении всего срока существования приложения, и внесение в него любых изменений необходимо тщательно продумывать, поскольку компоненты в этом случае подобны деревянным брускам в игре Дженга, где перемещение или удаление хотя бы одного из них может разрушить всю башню. Это часто приводит к принятию соглашений, регламентирующих процесс разработки, которые должны помочь справиться с растущей сложностью проекта.

Архитектура на микрослужбах

А теперь можно сформулировать самый важный шаблон в Node.js, касающийся больших приложений: *не пишите больших приложений*. Такая формулировка выглядит тривиальной, но тем не менее она определяет эффективную стратегию масштабирования сложности и мощности программных систем. Так в чем же заключается альтернатива созданию больших приложений? Ответ на этот вопрос находится на оси *y* куба масштабирования, определяющей разделение приложения по функциональным признакам. Идея заключается в том, чтобы разделить приложение на основные компоненты, поместив их в отдельные, независимые приложения. Это полная противоположность концепции монолитной архитектуры. Она идеально соответствует идеологии Unix и принципам платформы Node.js, которые приведены в начале книги, в частности принципу «*любая программа должна делать что-то одно, но делать это хорошо*».

Архитектура на микрослужбах в настоящее время является одним из основных шаблонов замены большого монолитного приложения набором автономных служб. Приставка «микро» означает, что службы должны быть как можно меньше, конечно, в разумных пределах. Не стоит полагать, что создание архитектуры из сотни приложений, предоставляющих одну-единственную веб-службу, является хорошим подходом. В действительности не существует строгих правил относительно размеров служб. Вопрос заключается не в размере, а в проектировании архитектуры микрослужб с учетом таких факторов, как **слабая связанность**, **высокая сцепленность** и **сложность интеграции**.

Пример архитектуры на микрослужбах

На рис. 10.10 изображена схема приложения электронной коммерции с архитектурой на микрослужбах.

Как показано на рис. 10.10, каждый из основных компонентов приложения электронной коммерции теперь является самостоятельной независимой сущностью, находится в отдельном контексте и обладает собственной базой данных. Все компоненты являются независимыми приложениями, предоставляющими группу связанных между собой услуг (высокая сцепленность).

Владение службой с**обственными данными** является важной особенностью архитектуры, основанной на микрослужбах. Поэтому база данных также должна быть разделена для поддержания надлежащего уровня изоляции и независимости. Использование общей единой базы данных существенно упрощает взаимодействие служб, но усиливает связи между службами (основанные на данных), исключая часть преимуществ наличия отдельных приложений.

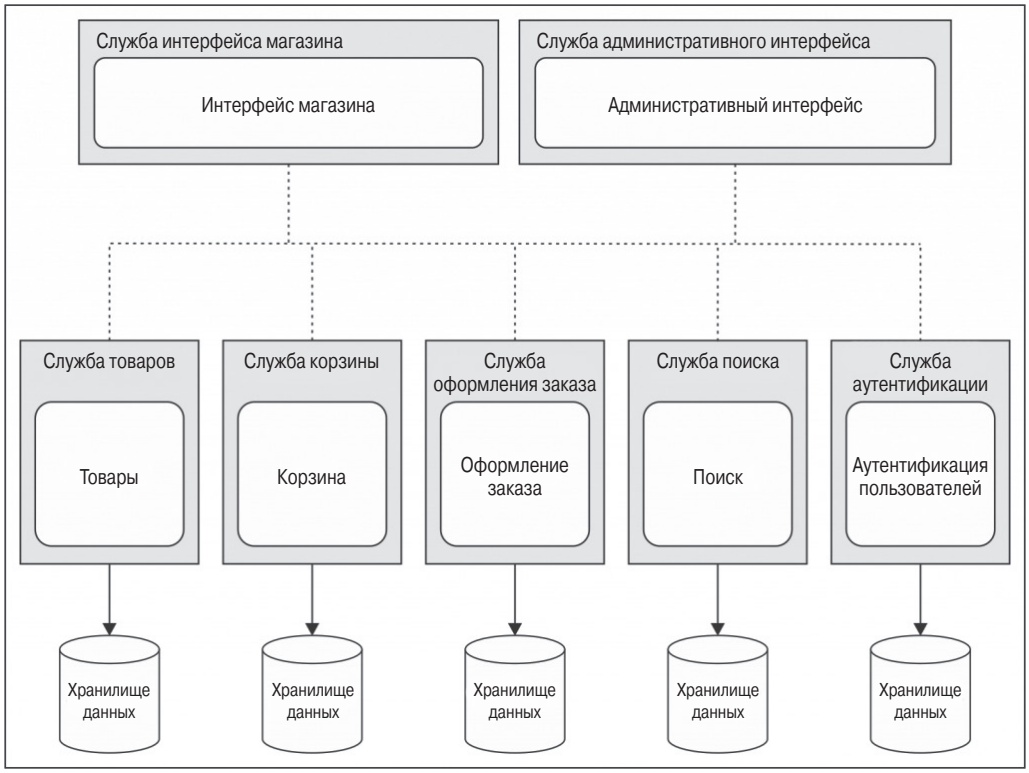

Рис. 10.10 ❖ Пример архитектуры на микрослужбах

Пунктирная линия, соединяющая узлы, указывает, что службам необходимо как-то общаться и обмениваться информацией для поддержания функциональности всей системы в целом. Поскольку службы не имеют общей базы данных, необходимо более интенсивное их взаимодействие для поддержания согласованности всей системы. Например, приложение, оформляющее заказы, должно иметь сведения о товарах, такие как цена и ограничения на доставку, и в то же время ему необходимо обновлять данные, хранящиеся в службе товаров, например доступность продукта после завершения оформления заказа. Способ общения узлов между собой показан на рис. 10.10 абстрактно. Конечно же, наиболее популярным подходом в данном случае является использование веб-служб, но это не единственный вариант, как будет продемонстрировано ниже.

Шаблон (архитектура, основанная на микрослужбах)
Разделение сложного приложения путем создания нескольких небольших автономных служб.

Плюсы и минусы микрослужб

В этом разделе рассматриваются преимущества и недостатки архитектуры, основанной на микрослужбах. Этот подход обещает внести революционные изменения в разработку приложений, но, с другой стороны, его применение влечет появление новых, нетривиальных проблем.

Мартин Фаулер (Martin Fowler) написал большую статью о микрослужбах, которую можно найти на странице http://martinfowler.com/articles/microservices.html.

Снижение важности служб Основным техническим преимуществом переноса всех служб в отдельные приложения являются изоляция системы в целом от сбоев и ошибок и возможность внесения важных изменений. Цель заключается в создании полностью независимых небольших служб, которые легко изменять и можно даже создавать заново. Если, например, служба оформления заказов приложения для продажи через Интернет неожиданно завершит работу из-за серьезной ошибки, прочие службы продолжат работу в обычном режиме. Конечно, сбой может затронуть некоторые функции, например невозможно будет приобрести товар, но все остальное будет работать по-прежнему.

Или, предположим, выяснилось, что выбор языка программирования для реализации компонента или базы данных является нелучшим. В монолитном приложении вряд ли можно будет что-то изменить, не нанеся серьезного ущерба всей системе. В приложении, основанном на микрослужбах, можно переписать всю службу с нуля, используя при этом другую базу данных или платформу, а остальные службы могут этого даже не заметить.

Повторное использование на разных платформах и языках Разделение большого монолитного приложения на ряд небольших служб позволяет создавать независимые единицы, которые без особых усилий могут повторно использоваться. Служба **Elasticsearch** (http://www.elasticsearch.org) является отличным примером службы поиска, пригодной для повторного использования. Еще одним примером службы, которую легко можно повторно использовать в любом приложении, вне зависимости от того, на каком языке программирования оно написано, является сервер аутентификации, разработанный в *главе 7 «Связывание модулей»*.

Главное преимущество: уровень сокрытия информации обычно намного выше, по сравнению с монолитными приложениями. Это возможно благодаря тому, что взаимодействие обычно происходит через удаленный интерфейс, например через веб-службу или брокера сообщений, что значительно упрощает сокрытие деталей реализации и ограждает клиента от влияния изменений в реализации или развертывании службы. Например, чтобы просто вызвать веб-службу, не требуется знать особенностей масштабирования ее инфраструктуры, на каком языке программирования она написана, какую базу данных использует и т. д.

Подход к масштабированию приложения Возвращаясь к кубу масштабирования, можно уверенно заявить, что микрослужбы эквивалентны масштабированию приложения по оси *y*, что само по себе позволяет распределить нагрузку между несколькими компьютерами. Кроме того, не следует забывать, что микрослужбы совместимы с масштабированием вдоль двух других осей куба масштабирования, позволяя продвинуть масштабирование приложения еще дальше. Например, каждая из служб может клонироваться для обработки большего трафика, но самое интересное, что они могут масштабироваться независимо друг от друга, позволяя оптимизировать управление ресурсами.

Проблемы микрослужб После всего сказанного микрослужбы могут показаться средством решения всех наших проблем, но это далеко не так. В самом деле, наличие большего числа узлов управления приводит к более высокой сложности с точки

зрения интеграции, развертывания и совместного использования кода. Микрослужбы успешно решают некоторые проблемы традиционных архитектур, но также привносят достаточно много новых вопросов. Как обеспечить взаимодействие служб? Как развернуть, масштабировать и управлять большим количеством приложений? Как совместно использовать код несколькими службами и как повторно использовать их код? К счастью, облачные службы и современные методологии DevOps[1] могут дать ответы на некоторые из этих вопросов. Платформа Node.js также может помочь в этом. Ее система модулей является идеальным помощником при совместном использовании кода различными проектами. Платформа Node.js была разработана так, чтобы быть *узлом* в распределенной системе, например, с архитектурой, основанной на микрослужбах.

Микрослужбы можно разрабатывать с использованием любого фреймворка (или даже только с помощью модулей ядра платформы Node.js), но для этих целей существует несколько специализированных решений, таких как **Seneca** (https://npmjs.org/package/seneca), **AWS Lambda** (https://aws.amazon.com/lambda), **IBM OpenWhisk** (https://developer.ibm.com/openwhisk) и **Microsoft Azure Functions** (https://azure.microsoft.com/en-us/services/functions). Полезным инструментом развертывания микрослужб является **Apache Mesos** (http://mesos.apache.org).

Шаблоны интеграции в архитектуре на микрослужбах

Одна из самых сложных задач, связанных с микрослужбами, заключается в таком соединении всех узлов, которое обеспечит их взаимодействие. Например, в **службе корзины** приложения электронной коммерции не было бы никакого смысла без возможности добавлять в нее товары, а **служба оформления заказов** была бы бесполезной без перечня приобретаемых товаров (корзина). Как уже упоминалось, имеются и другие факторы, обусловливающие необходимость взаимодействия между различными службами. Например, **служба поиска** должна знать, какие товары доступны и что эта информация актуальна. То же можно сказать о службе **оформления заказа**, обновляющей информацию о доступности **товара** после завершения покупки.

Разрабатывая стратегию интеграции, важно также учитывать жесткость связей между службами, вводимыми в систему. Не следует забывать, что разработка распределенной архитектуры подчиняется тем же рекомендациям и принципам, которые использовались локально при проектировании модулей или подсистем, поэтому необходимо принимать во внимание такие характеристики, как возможность повторного использования и расширяемость служб.

Прокси-сервер для служб

В первую очередь мы рассмотрим шаблон **прокси-сервера для служб** (его иногда называют также **шлюзом доступа к службам**). Этот сервер управляет взаимодействиями между клиентом и рядом удаленных служб. В архитектуре, основанной на микрослужбах, его основным назначением является обеспечение единой точкой доступа для нескольких служб, но также может поддерживать распределение нагрузки, кэширование, аутентификацию и ограничение трафика, что иногда очень полезно для реализации полноценного решения.

---

[1] Методология интеграции разработки (Dev, Development) и эксплуатации (Op, Operation). – *Прим. ред.*

Мы уже рассматривали этот шаблон, когда создавали пользовательский балансировщик нагрузки с помощью модулей `http-proxy` и `consul`. Тогда балансировщик поддерживал только две службы и благодаря реестру служб отображал URL-пути на службы и, следовательно, на список серверов. Прокси-сервер для служб работает точно так же. Это, по сути, тот же обратный прокси-сервер, который часто является и балансировщиком нагрузки, специально настроенным для обработки запросов к службам. На рис. 10.11 показано, как применить это решение к приложению электронной коммерции.

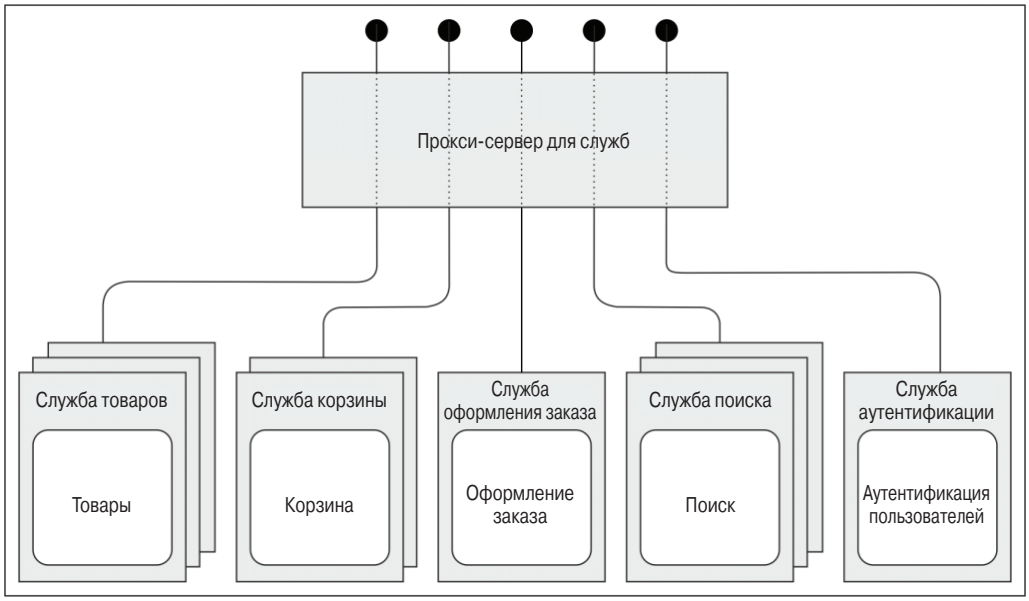

Рис. 10.11 ❖ Прокси-сервер для служб

Рисунок 10.11 наглядно показывает, как применение прокси-сервера помогает скрыть сложность базовой инфраструктуры. Это особенно удобно в инфраструктурах, основанных на микрослужбах, где может использоваться большое число узлов, особенно если каждая из служб масштабируется на нескольких компьютерах. Интеграция посредством прокси-сервера является чисто структурной, поскольку в ней отсутствует семантический механизм. Она просто придает знакомый монолитный вид сложной инфраструктуре из микрослужб. Это не относится к следующему шаблону, где интеграция основа на семантике.

Координация служб

Шаблон координации служб, описываемый далее, является, вероятно, самым естественным и наглядным способом интеграции набора служб. Вице-президент компании Engineering for the Netflix API, Даниель Якобсон (Daniel Jacobson), в одном из сообщений в своем блоге пишет следующее:

> *«Слой координации служб (Orchestration Layer, OL) представляет уровень абстракции, который получает обобщенные элементы данных и/или функции и конкретизирует их для целевого разработчика или приложения».*

Обобщенные элементы данных и/или функции идеально укладываются в идеологию архитектур, основанных на микрослужбах. Идея заключается в том, чтобы создать абстракции для объединения этих фрагментов мозаики в реализации конкретных служб.

Применим этот шаблон к примеру приложения электронной коммерции. Рассмотрим схему на рис. 10.12.

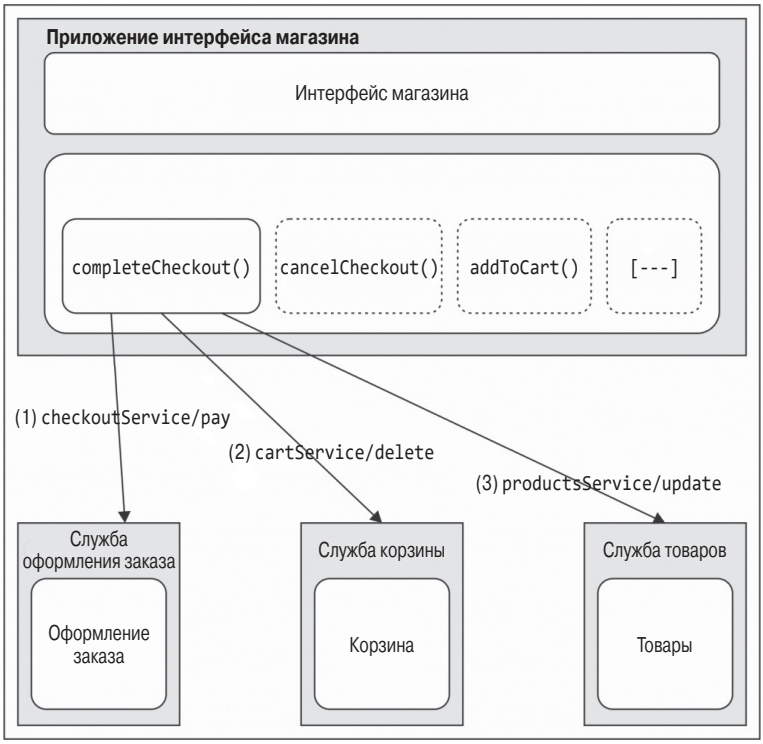

Рис. 10.12 ❖ Применение шаблона координации служб

На рис. 10.12 показано, как **приложение интерфейса магазина** использует слой координации для построения более сложных и конкретных функциональных возможностей путем соединения и координации существующих служб. В качестве примера выбрана гипотетическая служба `completeCheckout()`, которая вызывается в момент щелчка на кнопке оплаты в конце оформления заказа. На рисунке показано, что служба `completeCheckout()` представляет составную операцию из трех различных действий.

1. Во-первых, завершается транзакция путем вызова `checkoutService/pay`.
2. Затем, после успешной обработки платежа, служба корзины уведомляется, что товары приобретены и могут быть удалены из корзины. Это осуществляется вызовом `cartService/delete`.
3. Кроме того, после завершения платежа обновляется доступность только что приобретенных товаров. Это осуществляет вызов `productsService/update`.

Как видите, мы взяли три операции из трех различных служб и собрали новый программный интерфейс, координирующий работу служб для поддержания всей системы в согласованном состоянии.

Другой распространенной операцией, выполняемой **слоем координации**, является **агрегирование данных**, то есть объединение данных из различных служб в один ответ. Предположим, что требуется перечислить все товары в корзине. В этом случае координатору необходимо получить список идентификаторов товаров из **службы корзины**, а затем – полную информацию о товарах из **службы товаров**. Существует бесчисленное множество способов объединения и координации служб, но наиболее важное положение в шаблоне занимает слой координации, который выступает в качестве абстракции между набором услуг и конкретикой приложения.

Слой координации является отличным кандидатом для дальнейшего разделения по функциональному признаку и возможностям. Это обычно выделенная независимая служба, в данном случае названная координатором служб. Эта методика полностью соответствует идеологии микрослужб.

На рис. 10.13 показано дальнейшее усовершенствование архитектуры.

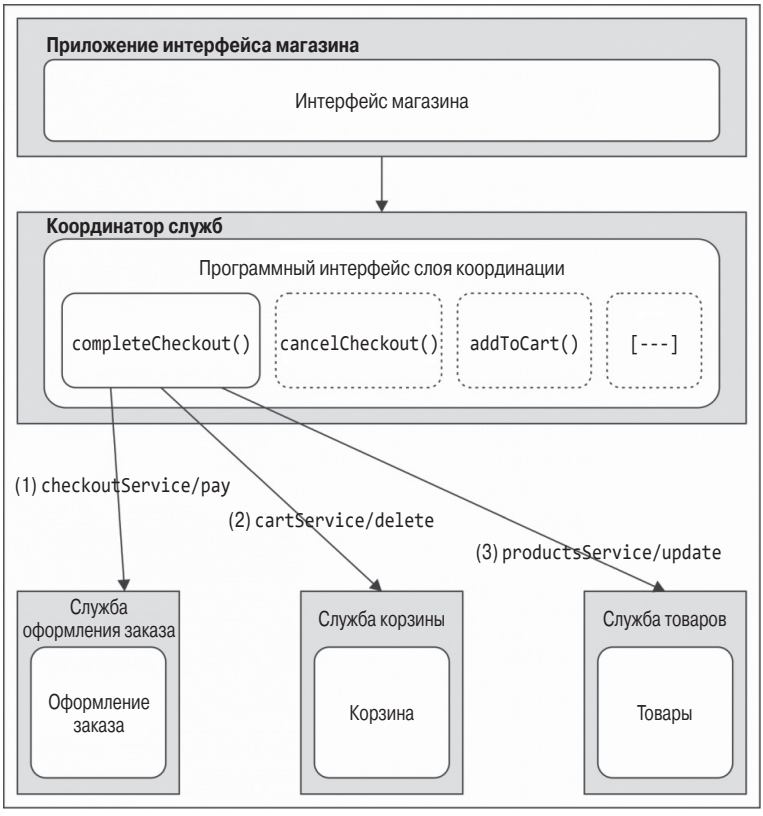

Рис. 10.13 ❖ Дальнейшее усовершенствование архитектуры координатора служб

Создание автономного координатора, как это показано на рис. 10.13, обеспечивает изоляцию клиентского приложения (в данном случае интерфейса магазина) от сложной инфраструктуры микрослужб. Этот подход напоминает прокси-сервер, но между ними имеется существенное различие: в отличие от прокси-сервера, не вника-

ющего в детали, координатор занимается семантической интеграцией служб и часто поддерживает программный интерфейс, отличающийся от интерфейсов базовых служб.

Интеграция с помощью брокера сообщений

Шаблон координатора дает нам механизм явной интеграции служб. Он имеет свои преимущества и недостатки. Его легко реализовать, отладить и масштабировать, но, к сожалению, координатор должен иметь полное представление о базовой архитектуре и о работе каждой из служб. Если бы речь шла об объектах, а не архитектурных узлах, координатор можно было бы рассматривать как антишаблон «Вездесущий объект», то есть объект, который слишком много знает и за слишком многое отвечает, что обычно приводит к образованию жестких связей, слабой сцепленности и, самое главное, к большой сложности.

Шаблон, рассматриваемый далее, распределяет между службами ответственность за синхронизацию информации во всей системе. Но это вовсе не означает создания непосредственных связей между службами, что способствует увеличению связанности и росту сложности системы из-за увеличения числа взаимосвязей между узлами. Цель прежде всего заключается в том, чтобы каждая служба была изолированной, имела возможность работать отдельно от остальных служб или в сочетании с новыми службами и узлами.

Решением является использование брокера сообщений – системы, отделяющей отправителей сообщений от получателей, – позволяющего реализовать шаблон централизованной публикации/подписки, практически шаблон наблюдателя для распределенных систем (этот шаблон будет подробно рассмотрен позже). Схема на рис. 10.14 иллюстрирует применение этого шаблона к приложению электронной коммерции.

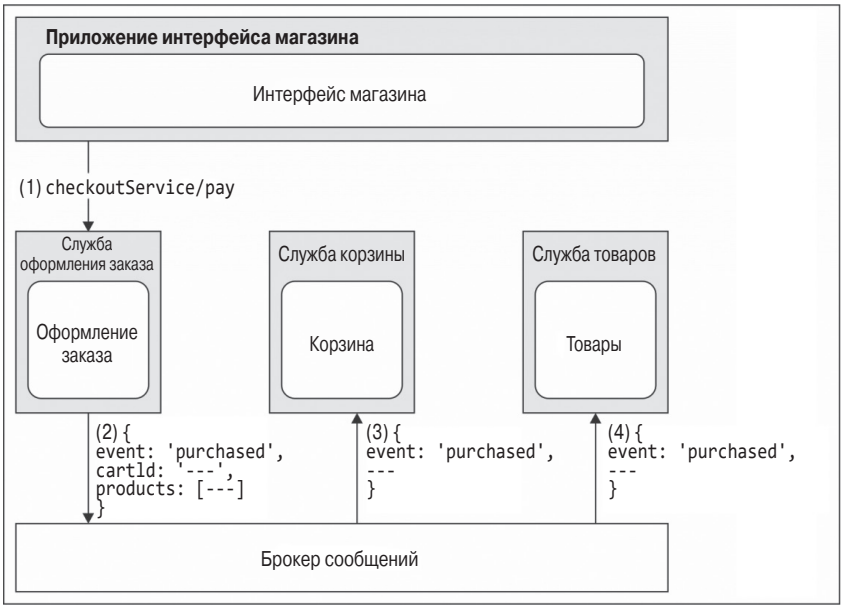

Рис. 10.14 ❖ Применение шаблона брокера сообщений

Как видите, клиенту службы **оформления заказов**, которым является интерфейс приложения, не требуется явная интеграция с другими службами. Ему нужно лишь вызвать `checkoutService/pay` для завершения оформления заказа и получить деньги от клиента. Все действия, связанные с интеграцией, выполняются в фоновом режиме.

1. **Интерфейс магазина** вызывает операцию `checkoutService/pay` службы **оформления заказов**.
2. После завершения операции служба **оформления заказов** генерирует событие с информацией об операции, то есть идентификатор корзины `cartId` и перечень приобретенных товаров `products`. Событие публикуется в брокере сообщений. Служба **оформления заказов** не знает, кто получит сообщение.
3. Служба **корзины** зарегистрирована в брокере, поэтому она получит событие о покупке, опубликованное службой **оформления заказов**. Служба **корзины** реагирует на него, удаляя из базы данных корзину с идентификатором, переданным в сообщении.
4. Служба **товаров** также зарегистрирована в брокере сообщений, поэтому она тоже получит событие о покупке и обновит свою базу данных на основании новых сведений, изменив доступность товаров, включенных в сообщение.

Весь процесс выполняется без вмешательства каких-либо внешних сущностей, таких как координатор. Ответственность за распространение информации для согласования данных распределена между службами. Здесь нет вездесущей службы, обязанной знать механизм всей системы, и каждая служба отвечает за свой участок интеграции.

Брокер сообщений является основополагающим элементом для отделения служб и снижения сложности их взаимодействий. Он способен обеспечить и другие интересные возможности, такие как сохраняемые очереди сообщений и гарантирование упорядочения сообщений, но об этом мы поговорим в следующей главе.

Итоги

В этой главе мы узнали, как проектировать архитектуры приложений на платформе Node.js, позволяющие масштабировать их пропускную способность и сложность. Мы увидели, что масштабирование не только позволяет приложениям справляться с большим трафиком и сокращать время отклика, но и обеспечивает улучшение их доступности и устойчивости при возникновении сбоев. Эти особенности взаимосвязаны, и подготовка к масштабированию на ранних этапах разработки является характерной чертой Node.js, которая облегчает процесс разработки и требует лишь незначительных затрат ресурсов.

Куб масштабирования учит нас, что приложения можно масштабировать по трем измерениям. Мы детально рассмотрели два самых важных из них, оси x и y, и познакомились с двумя основными архитектурными шаблонами: распределение нагрузки и микрослужбы. Теперь мы знаем, как запустить несколько экземпляров одного приложения на платформе Node.js, как распределять трафик между ними и как достичь других целей, таких как устойчивость к сбоям и перезапуск без простоя. Мы также рассмотрели динамическое решение этой задачи для автоматического масштабирования инфраструктуры с помощью реестра служб. Но клонирование и распределение нагрузки обеспечивают масштабирование только вдоль одной оси куба масштабирования, поэтому для масштабирования в другом измерении было применено разделе-

ние приложения на основе составляющих его служб, путем создания архитектуры, основанной на микрослужбах. Мы увидели, что микрослужбы коренным образом меняют подход к разработке и управлению проектом, обеспечивая естественный способ распределения нагрузки приложения и декомпозиции его сложной структуры. Но они также привносят проблемы, связанные с интеграцией набора служб, которых нет в монолитном приложении. Решению этих проблем была посвящена завершающая часть главы, где мы рассмотрели несколько архитектурных решений для интеграции набора независимых служб.

В следующей главе мы детальнее разберем шаблон обмена сообщениями, упоминавшийся в этой главе, а также рассмотрим более совершенные методы интеграции, предназначенные для реализации сложных распределенных архитектур.

Глава 11

Шаблоны обмена сообщениями и интеграции

Если масштабируемость затрагивает проблемы разделения, то интеграция систем, напротив, подразумевает объединение. В предыдущей главе рассказывалось, как распределить функциональные возможности приложения по нескольким соединениям с несколькими компьютерами. Чтобы все работало должным образом, все части должны каким-то образом общаться друг с другом и, следовательно, должны быть интегрированы.

Существуют два главных подхода к интеграции распределенных приложений: первый ориентирован на использование общего хранилища в качестве центрального координатора и накопителя информации, а второй заключается в использовании сообщений для распространения данных, событий и команд по узлам системы. Именно второй подход обеспечивает все преимущества масштабируемых распределенных систем, но вместе с фантастическими возможностями приходят и дополнительные сложности.

Сообщения используются на всех уровнях программной системы. Обмен сообщениями применяется для организации взаимодействий в Интернете, отправки сведений другим процессам, внутри приложения, как альтернатива прямым вызовам функций (шаблон «Команда»), а также в драйверах устройств для взаимодействия с оборудованием. Любые дискретные структурированные данные, используемые для обмена информацией между компонентами и системами, можно рассматривать как *сообщение*. Но когда речь идет о распределенной архитектуре, термин «**система обмена сообщениями**» описывает конкретный класс решений, шаблонов и архитектур, содействующих обмену информацией по сети.

Этот вид систем характеризует несколько признаков. Можно использовать брокера или одноранговую структуру, запрос/ответ или односторонний вид связи, или очереди, гарантирующие надежную доставку сообщений, выбор очень широк. Книга «*Enterprise Integration Patterns*» *Грегора Хона* (Gregor Hohpe) и *Бобби Вульфа* (Bobby Woolf)[1] дает представление об обширности данной темы. Она считается *библией* по

---

[1] *Грегор Хоп, Б. Вульф*. Шаблоны интеграции корпоративных приложений. М.: Вильямс, 2015. ISBN 978-5-8459-1946-5. – *Прим. ред.*

приемам обмена сообщениями и шаблонам интеграции и содержит описание 65 различных шаблонов интеграции на более чем 700 страницах. В этой главе рассматриваются лишь некоторые из этих популярных шаблонов, наиболее важные для платформы Node.js и ее экосистемы.

Другими словами, в этой главе будут рассмотрены следующие вопросы:

- введение в системы обмена сообщениями;
- шаблон «Публикация/подписка»;
- шаблоны «Конвейер» и «Распределение заданий»;
- различные шаблоны «Запрос/ответ».

Введение в системы обмена сообщениями

Говоря о сообщениях и системах обмена сообщениями, нужно обратить внимание на четыре следующих момента:

- направленность сообщений: обмен может быть односторонним или иметь вид «запрос/ответ»;
- цели сообщений определяет также их содержание;
- сроки доставки сообщений: сообщения могут отправляться и доставляться немедленно или с задержкой (асинхронно);
- доставка сообщения может производиться непосредственно получателю или через брокера.

В следующих разделах мы рассмотрим все эти аспекты, чтобы создать основу для последующих обсуждений.

Шаблоны однонаправленного обмена и вида «Запрос/ответ»

Основополагающим свойством системы обмена сообщениями является направленность связи, которая обычно определяется ее семантикой.

Самым простым шаблоном является передача сообщений в одном направлении, от отправителя к получателю. Это довольно тривиальный случай, не требующий особых пояснений. Его иллюстрирует схема на рис. 11.1.

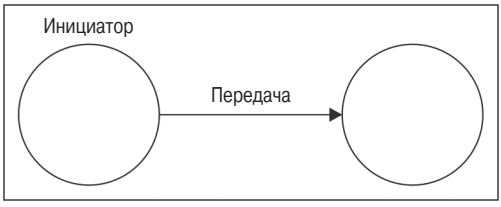

Рис. 11.1 ❖ Однонаправленный обмен сообщениями

Типичным примером односторонней связи является электронная почта или веб-сервер, который отправляет сообщения подключенному браузеру с помощью веб-сокетов, или система, распределяющая задания между группой рабочих процессов.

Однако в практике более популярен шаблон «Запрос/ответ». Типичным примером является вызов веб-службы. На рис. 11.2 показан этот простой и широко известный случай.

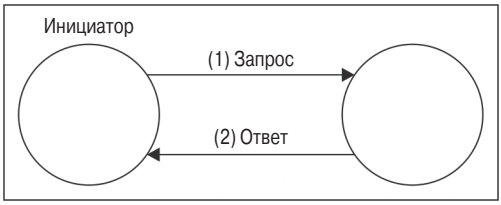

Рис. 11.2 ❖ Шаблон «Запрос/ответ»

Реализация шаблона «Запрос/ответ» на первый взгляд кажется тривиальной, но она может стать более сложной, когда сообщения передаются асинхронно или проходят через несколько узлов, как показано на рис. 11.3.

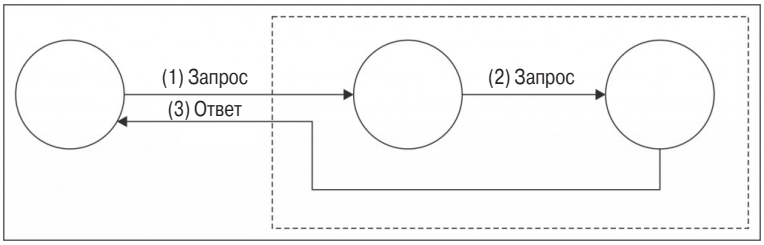

Рис. 11.3 ❖ Более сложная реализация шаблона «Запрос/ответ»

Схема на рис. 11.3 позволяет оценить сложность некоторых шаблонов «Запрос/ответ». Если рассматривать только связи между любыми двумя узлами, можно с уверенностью утверждать, что они односторонние. Однако с глобальной точки зрения инициатор отправляет запрос и получает на него ответ, даже притом, что он поступает от другого узла. В таких ситуациях шаблон «Запрос/ответ» отличается от односторонней связи тем, что между запросом и ответом имеется зависимость, которая известна инициатору. Ответ обычно обрабатывается в том же контексте, что и запрос.

Типы сообщений

Сообщения, по существу, являются средством связи между различными программными компонентами, и существует целый ряд причин их использования, например: чтобы получить некоторую информацию, хранящуюся в другой системе или в другом компоненте, удаленно выполнять операции или уведомить соседние узлы о каких-то событиях. Содержимое сообщения также может меняться в зависимости от его назначения. В целом можно выделить три типа сообщений:

- **команды**;
- **события**;
- **документы**.

Команды

Сообщение-команда, по сути, является уже знакомым нам сериализованным объектом команды, как рассказывалось в *главе 6 «Шаблоны проектирования»*. Цель сообщений этого типа – инициировать выполнение действия или задания в приемнике. Для этого сообщение должно содержать информацию, необходимую для выполнения

задания, которая, как правило, состоит из имени операции и списка аргументов. Сообщения-команды можно использовать при реализации системы **вызова удаленных процедур** (Remote Procedure Call, RPC), распределенных вычислений или просто для запроса данных. Примерами сообщений-команд являются HTTP-вызовы в стандарте RESTful. Каждый HTTP-вызов имеет конкретное обозначение, связанное с определенной операцией: GET – для извлечения ресурса, POST – для создания нового ресурса, PUT – для изменения существующего ресурса и DELETE – для удаления ресурса.

События

Сообщение-событие используется для уведомления другого компонента о происходящем. Обычно такие сообщения содержат тип события и иногда некоторые подробности, такие как контекст, тема или вовлеченный участник. В веб-разработке сообщения-события используются браузером для реализации опроса веб-сокетов с целью получения уведомлений от сервера о происходящем, например изменились данные или состояние системы в целом. Использование событий является очень важным механизмом интеграции в распределенных приложениях, поскольку позволяет сохранить все узлы системы в согласованном состоянии.

Документы

Сообщения-документы главным образом предназначены для передачи данных между компонентами и компьютерами. Основным отличием сообщений-документов от сообщений-команд (которые тоже могут содержать данные) является отсутствие какой-либо информации, указывающей получателю, что делать с данными. С другой стороны, главное отличие сообщений-документов от сообщений-событий заключается в отсутствии в документах привязки к конкретному происшедшему событию. Зачастую ответами на команды являются документы, поскольку они обычно содержат только запрашиваемые данные или результаты операции.

Асинхронный обмен сообщениями и очереди

Как известно, асинхронное выполнение операций на платформе Node.js дает существенные преимущества. Это касается и обмена сообщениями.

Синхронную связь можно сравнить с телефонным звонком: два узла должны быть одновременно подключены к одному и тому же каналу для обмена сообщениями в режиме реального времени. Причем если нужно позвонить кому-то еще, требуется либо другой телефон, либо завершить звонок и сделать новый вызов.

Асинхронная связь похожа на обмен SMS-сообщениями, так как не требует подключения получателя к сети в момент отправки сообщения, ответ на которое можно получить сразу, с произвольной задержкой или не получить вообще. Можно отправить несколько SMS-сообщений нескольким получателям, одно за другим, и получить от них ответы (если таковые будут) в любом порядке. Короче говоря, здесь применяется параллельная обработка с использованием меньшего количества ресурсов.

Другим важным преимуществом асинхронных сообщений является возможность их хранения неопределенное время с последующей доставкой позднее. Это может пригодиться, когда получатель занят и не может обработать новые сообщения или когда нужно гарантировать доставку. В системах обмена сообщениями такие гарантии дают **очереди сообщений**, то есть компоненты, которые являются посредниками

между отправителем и получателем, хранящими сообщения, пока они не будут доставлены до места назначения, как показано на рис. 11.4.

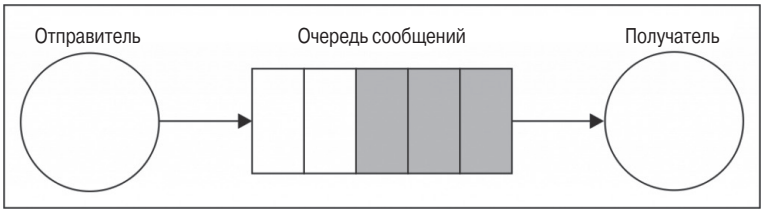

Рис. 11.4 ❖ Очередь сообщений обеспечивает гарантированную доставку

Если по какой-либо причине получатель завершит работу, отключится от сети или окажется под высокой нагрузкой, сообщения будут накапливаться в очереди и переданы получателю, когда он вернется в сеть и восстановит свою работоспособность. Очередь может находиться внутри отправителя, охватывать отправителя и получателя или быть помещена в выделенную внешнюю систему, действующую независимо, как промежуточное программное обеспечение для организации связи.

Обмен сообщениями, прямой и через брокера

Сообщения могут доставляться непосредственно получателю или с помощью централизованной системы, называемой **брокером сообщений**. Основной задачей брокера является отделение получателя сообщения от его отправителя. Схема на рис. 11.5 иллюстрирует разницу в архитектуре этих двух подходов.

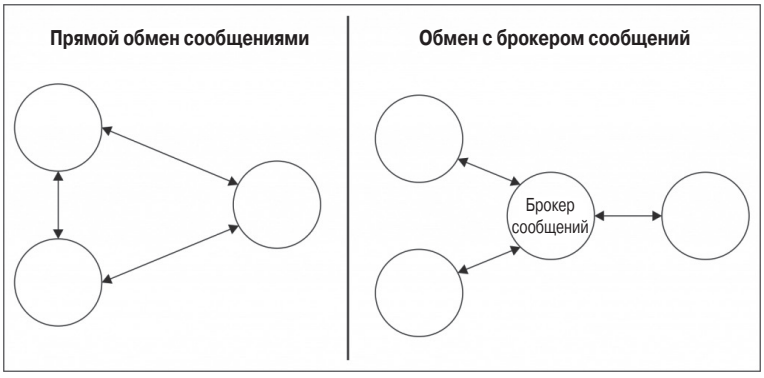

Рис. 11.5 ❖ Обмен сообщениями, прямой и через брокера

В схеме прямого обмена сообщениями каждый узел сам отвечает за доставку сообщения получателю. Это означает, что узлы должны знать адрес и порт приемника, а также согласовать форматы протоколов и сообщений. Брокер избавляет узлы от всех этих сложностей, делая их независимыми и способными общаться с произвольным количеством других узлов, о которых им ничего не известно. Кроме того, брокер может служить мостом при использовании разных протоколов. Например, популярный брокер RabbitMQ (http://www.rabbitmq.com) поддерживает протоколы **Advanced**

Message Queuing Protocol (AMQP), **Message Queue Telemetry Transport (MQTT)** и **Simple/Streaming Text Orientated Messaging Protocol (STOMP)**, что позволяет обеспечить обмен сообщениями между приложениями, поддерживающими разные протоколы.

MQTT (http://mqtt.org) – легковесный протокол обмена сообщениями, специально разработанный для связи между компьютерами (имеется в виду Интернет вещей). AMQP (http://www.amqp.org) – более сложный протокол, который проектировался как открытая альтернатива проприетарной реализации обмена сообщениями. STOMP (http://stomp.github.io) – легковесный текстовый протокол в стиле HTTP. Все три являются протоколами прикладного уровня и основаны на протоколе TCP/IP.

Кроме организации взаимодействий, брокер может предлагать дополнительные возможности, такие как хранимые очереди, маршрутизация, преобразования сообщений и мониторинг, не говоря о широком спектре шаблонов проектирования, поддерживаемых многими брокерами «из коробки». Конечно, ничто не мешает реализовать все эти функции самостоятельно, используя прямой обмен сообщениями, но это потребует значительных усилий. Тем не менее существует несколько причин избегать использования брокера:

- исключение элемента, отказ которого приведет к сбою всей системы;
- брокер должен масштабироваться, в то время как при использовании прямой передачи сообщений требуется масштабировать только отдельные узлы;
- переход на непосредственный обмен сообщениями может значительно ускорить их передачу.

Кроме того, реализация прямого обмена сообщениями дает больше гибкости и возможностей, поскольку отсутствует привязка к конкретной технологии, протоколу или архитектуре. Популярная низкоуровневая библиотека для создания систем обмена сообщениями ØMQ (http://zeromq.org) демонстрирует гибкость, которой можно достичь посредством создания собственных архитектур.

Шаблон «Публикация/подписка»

Публикация/подписка является одним из самых популярных шаблонов одностороннего обмена сообщениями. Он должен быть уже знаком вам, поскольку это всего лишь вариант шаблона «Наблюдатель» для распределенной архитектуры. Так же, как в шаблоне «Наблюдатель», здесь имеется ряд подписчиков, регистрирующихся для получения сообщений конкретного вида. С другой стороны, издатель генерирует сообщения, распределяемые по соответствующим подписчикам. На рис. 11.6 показаны два основных варианта реализации шаблона «Публикация/подписка»: первый – для прямого обмена сообщениями, а второй – с брокером в качестве посредника.

Главная особенность шаблона «Публикация/подписка» – в том, что издатель не знает заранее, кто подпишется на получение сообщений. Как уже упоминалось, подписчики являются абонентами, зарегистрировавшимися на получение конкретного сообщения, что позволяет издателю работать с произвольным количеством получателей. Другими словами, обе стороны *слабо связаны* друг с другом, что делает этот шаблон идеальным для интеграции узлов эволюционирующей распределенной системы.

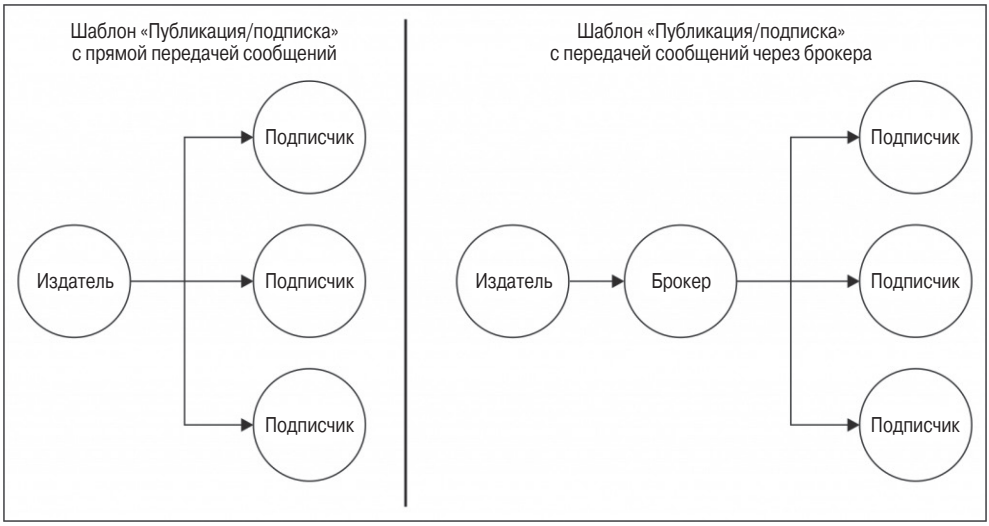

Рис. 11.6 ❖ Два варианта реализации шаблона «Публикация/подписка»

Присутствие брокера еще больше ослабляет связанность узлов системы, поскольку абоненты взаимодействуют только с брокером и им не нужно знать, какой из узлов является отправителем. Как будет продемонстрировано позже, брокер может также поддерживать очереди сообщений, гарантируя надежную их доставку даже при наличии проблем с соединениями между узлами.

А теперь рассмотрим пример использования этого шаблона.

Минимальное приложение для общения в режиме реального времени

Чтобы показать, как шаблон «Публикация/подписка» помогает интегрировать распределенные архитектуры, создадим очень простое приложение чата с использованием только веб-сокетов. Затем попробуем масштабировать его, запустив несколько экземпляров и использовав систему обмена сообщениями.

Сервер

Будем двигаться вперед шаг за шагом. Сначала напишем приложение чата. Для этого воспользуемся пакетом `ws` (https://npmjs.org/package/ws), реализующим поддержку веб-сокетов в Node.js. Как известно, реализация приложений на платформе Node.js, действующих в масштабе реального времени, не вызывает особых проблем, и следующий код подтверждает это (файл `app.js`):

```
const WebSocketServer = require('ws').Server;

//статический файл-сервер
const server = require('http').createServer(       //[1]
  require('ecstatic')({root: `${__dirname}/www`})
);

const wss = new WebSocketServer({server: server}); //[2]
wss.on('connection', ws => {
```

```
  console.log('Client connected');
  ws.on('message', msg => {                    //[3]
    console.log(`Message: ${msg}`);
    broadcast(msg);
  });
});

function broadcast(msg) {                      //[4]
  wss.clients.forEach(client => {
    client.send(msg);
  });
}

server.listen(process.argv[2] || 8080);
```

И это все! Здесь все, что требуется для реализации серверной части приложения чата. Работает она следующим образом:

1) создается HTTP-сервер, и к нему присоединяется промежуточное программное обеспечение `ecstatic` (https://npmjs.org/package/ecstatic) для обслуживания статических файлов. Оно обеспечивает обслуживание клиентских ресурсов приложения (JavaScript и CSS);
2) создается новый экземпляр сервера веб-сокетов с передачей ему имеющегося HTTP-сервера. Затем запускается цикл приема входящих соединений подключением обработчика события `connection`;
3) каждый раз, когда к серверу подключается новый клиент, запускается цикл приема входящих сообщений. Когда поступает новое сообщение, оно передается всем подключенным клиентам;
4) функция `broadcast()` выполняет обход всех подключенных клиентов и для каждого вызывает функцию `send()`.

Это и есть волшебство платформы Node.js! Конечно, это простейший сервер, но он справляется со своими обязанностями.

Клиент

Теперь займемся реализацией клиентской части. Она тоже будет небольшой – по сути, это HTML-страница с кодом на JavaScript. Поместим эту страницу в файл `www/index.html`:

```
<html>
  <head>
    <script>
      var ws = new WebSocket('ws://' + window.document. location.host);
      ws.onmessage = function(message) {
        var msgDiv = document.createElement('div');
        msgDiv.innerHTML = message.data;
        document.getElementById('messages').appendChild(msgDiv);
      };

      function sendMessage() {
        var message = document.getElementById('msgBox').value;
        ws.send(message);
      }
    </script>
```

```
  </head>
<body>
  Messages:
  <div id='messages'></div>
  <input type='text' placeholder='Send a message' id='msgBox'>
  <input type='button' onclick='sendMessage()' value='Send'>
  </body>
</html>
```

Эта HTML-страница не нуждается в пояснениях, поскольку не содержит ничего нового для веб-разработчика. Сценарий использует встроенный объект `WebSocket` для инициализации подключения к серверу Node.js и приема поступающих с него сообщений, которые затем отображает в новых элементах `div`. Для отправки сообщений используются обычное поле ввода и кнопка.

При остановке или перезапуске сервера соединение с клиентом будет закрыто и не восстановится автоматически (как это было бы при использовании библиотек высокого уровня, таких как `Socket.io`). Это означает, что необходимо обновить страницу в браузере после перезапуска сервера, чтобы восстановить подключение (можно также реализовать свой механизм восстановления соединения, который здесь не рассматривается).

Запуск и масштабирование приложения чата

Теперь можно запустить приложение, для чего нужно выполнить следующую команду:

```
node app 8080
```

Для опробования этого примера вам понадобится современный браузер с встроенной поддержкой веб-сокетов. Список таких браузеров можно найти на странице http://caniuse.com/#feat=websockets.

При переходе по адресу `http://localhost:8080` в окне браузера должен появиться интерфейс, изображенный на рис. 11.7.

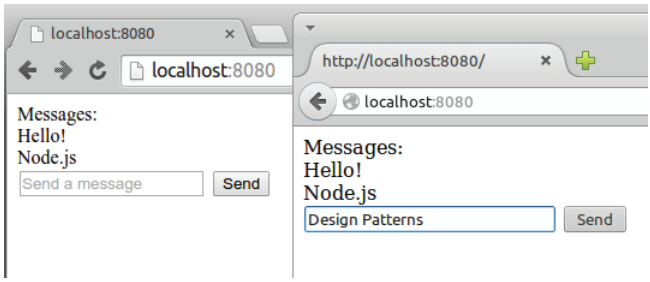

Рис. 11.7 ❖ Интерфейс приложения чата в браузере

Теперь посмотрим, что произойдет, если попытаться масштабировать приложение, запустив несколько его экземпляров. Запустим еще один сервер, прослушивающий другой порт:

```
node app 8081
```

Для нас желательно, чтобы два клиента, подключившихся к двум различным серверам, имели возможность обмениваться сообщениями в чате. К сожалению, в текущей реализации этого не происходит. Это можно проверить, открыв другую вкладку браузера и перейдя в ней по адресу `http://localhost:8081`.

Когда сообщение посылается одному из экземпляров, оно передается только клиентам, подключенным к этому серверу. Серверы не взаимодействуют друг с другом. Их нужно интегрировать.

В реальном приложении мы использовали бы балансировщик нагрузки, но в этом примере он не будет применяться. Это позволило бы получать доступ к конкретному серверу для проверки его взаимодействия с другими экземплярами.

Использование Redis в качестве брокера сообщений

Начнем наш анализ реализации шаблона «Публикация/подписка» со знакомства с Redis (http://redis.io) – быстрого и гибкого хранилища пар ключ/значение, которое также часто называют *сервером структур данных*. Redis скорее является базой данных, чем брокером сообщений, но среди его многочисленных возможностей имеется пара команд, специально предназначенных для реализации централизованного шаблона «Публикация/подписка».

Конечно, эта реализация является очень простой, если сравнивать ее с более совершенными средствами обмена сообщениями, но это одна из главных причин ее популярности. Обычно Redis уже присутствует в существующей инфраструктуре, например в качестве сервера кэширования или хранилища сеансов. Из-за скорости и гибкости его часто применяют в распределенных системах для совместного использования данных. То есть при появлении необходимости включить брокер в реализацию шаблона «Публикация/подписка» самым простым и быстрым решением станет повторное использование Redis, а не установка и поддержка выделенного брокера сообщений. Продемонстрируем простоту и потенциал этого решения на примере.

Для этого примера потребуется установить Redis и настроить прослушивание его порта по умолчанию. Более подробные сведения об этом можно найти на странице http://redis.io/topics/quickstart.

Итак, нам требуется интегрировать серверы чата, используя Redis в качестве брокера сообщений. Все экземпляры будут публиковать любые сообщения, полученные от своих клиентов, в брокере и регистрироваться как получатели на все сообщения, поступающие от других экземпляров. Как видите, все серверы будут одновременно подписчиками и издателями. На рис. 11.8 изображена схема архитектуры, которую нужно создать.

Взглянув на рис. 11.8, легко определить маршруты сообщений:

1) сообщение вводится в текстовое поле в веб-странице и отправляется экземпляру сервера чата;
2) сервер публикует сообщение в брокере;
3) брокер рассылает сообщение всем абонентам, которыми в данной архитектуре являются все экземпляры сервера чата;
4) каждый экземпляр рассылает полученное сообщение своим клиентам.

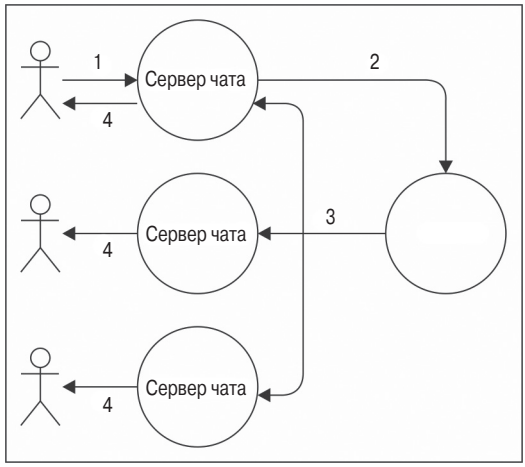

Рис. 11.8 ❖ Архитектура приложения чата с брокером

Сервер **Redis** позволяет публиковать каналы и подписываться на каналы со строковым идентификатором, например `chat.nodejs`. Он также позволяет использовать глобальные шаблоны для определения подписок, охватывающих несколько каналов, например `chat.*`.

Рассмотрим, как это работает на практике. Изменим код сервера, добавив логику публикации/подписки:

```
const WebSocketServer = require('ws').Server;
const redis = require("redis");                    // [1]
const redisSub = redis.createClient();
const redisPub = redis.createClient();

//статический файл-сервер
const server = require('http').createServer(
  require('ecstatic')({root: `${ dirname}/www`})
);

const wss = new WebSocketServer({server: server});
wss.on('connection', ws => {
  console.log('Client connected');
  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    redisPub.publish('chat_messages', msg);        // [2]
  });
});

redisSub.subscribe('chat_messages');               // [3]
redisSub.on('message', (channel, msg) => {
  wss.clients.forEach((client) => {
    client.send(msg);
  });
});

server.listen(process.argv[2] || 8080);
```

Изменения, внесенные в код оригинального сервера чата, выделены жирным. Вот как работает измененный код:

1) для подключения к серверу Redis в приложениях для Node.js используется пакет `redis` (`https://npmjs.org/package/redis`) – полноценный клиент, поддерживающий все доступные команды Redis. После загрузки модуля создаются два соединения: одно – для подписки на канал, а второе – для публикации сообщений. Это необходимо при использовании Redis, потому что после перевода соединения в *режим подписки* к нему можно применять только команды, относящиеся к подписке. А это значит, что необходимо еще одно соединение для публикации сообщений;
2) когда от клиента поступает новое сообщение, оно публикуется в канале `chat_messages`. Сообщения не рассылаются клиентам непосредственно, поскольку сервер подписан на тот же самый канал (как показано ниже), оно будет возвращаться через службу Redis. В данном случае это очень простой и эффективный механизм;
3) как уже упоминалось, сервер также должен подписаться на канал `chat_messages`, поэтому регистрируем обработчика для приема всех сообщений, опубликованных в этом канале (текущим или любым другим сервером чата). После получения сообщения оно рассылается всем клиентам, подключенным к веб-сокету текущего сервера.

Этих небольших изменений достаточно, чтобы интегрировать все запущенные серверы чата. Чтобы убедиться в этом, запустим несколько экземпляров приложения:

```
node app 8080
node app 8081
node app 8082
```

Теперь можно создать несколько вкладок в браузере, по одной для каждого экземпляра, и удостовериться, что все сообщения, отправляемые на один из серверов, успешно доставляются всем клиентам, подключенным к разным серверам. Примите поздравления! Мы только что с помощью шаблона «Публикация/подписка» интегрировали распределенное приложение реального времени.

Прямая публикация/подписка с помощью библиотеки ØMQ

Присутствие брокера значительно упрощает архитектуру системы обмена сообщениями, но существуют обстоятельства, когда это не является оптимальным решением, например когда любая задержка имеет решающее значение, при масштабировании сложных распределенных систем, или когда неприемлемо наличие элемента, сбой которого приводит к отказу всей системы.

Введение в библиотеку ØMQ

Если ваш проект попадает в категорию кандидатов на использование прямого обмена сообщениями, лучшим решением, несомненно, станет применение библиотеки **ØMQ** (http://zeromq.org, также известна как zmq, ZeroMQ и 0MQ), которая уже упоминалась ранее. ØMQ включает основные инструменты для реализации разнообразных шаблонов обмена сообщениями. Это низкоуровневая и очень быстрая библиотека с минималистичным программным интерфейсом, который тем не менее предлагает все необходимое для создания системы обмена сообщениями, например атомарные сообщения, распределение нагрузки, очереди и многое другое. Она поддерживает

многие виды транспортировки, такие как каналы внутри процесса (`inproc://`), средства связи между процессами (`ipc://`), многоадресную рассылку по протоколу PGM (`pgm://` или `epgm://`) и, конечно же, классический протокол TCP (`tcp://`).

В библиотеке ØMQ имеются также инструменты для реализации шаблона «Публикация/подписка», необходимые в данном примере. Итак, исключим брокер (Redis) из архитектуры приложения чата и реализуем прямое взаимодействие узлов с помощью сокетов ØMQ .

Сокеты ØMQ можно считать сетевыми сокетами с расширенными возможностями, которые обеспечивают дополнительные абстракции для реализации наиболее распространенных шаблонов обмена сообщениями. Например, имеются сокеты для реализации шаблонов «Публикация/подписка», «Запрос/ответ» или «Односторонняя передача».

Сервер чата с прямой передачей сообщений

При исключении брокера из архитектуры каждый экземпляр сервера чата должен напрямую подключаться к другим доступным экземплярам для получения публикуемых ими сообщений. В библиотеке ØMQ имеются два специальных типа сокетов для этой цели: `PUB` и `SUB`. Сокет `PUB` обычно подключается к порту, на который поступают подписки из других сокетов `SUB`.

Подписка может иметь *фильтр*, определяющий, какие сообщения должны доставляться в сокет `SUB`. Фильтр – это просто **двоичный буфер** (но может быть строкой), содержимое которого будет сравниваться с началом сообщения (также является двоичным буфером). При отправке сообщения через сокет `PUB` оно передается во все подключенные сокеты `SUB`, но только после применения их фильтров подписки. Фильтры применяются на стороне издателя и только если используется протокол с поддержкой постоянных соединений, такой как TCP.

На рис. 11.9 изображена схема распределенной архитектуры сервера чата (для простоты здесь показаны только два экземпляра):

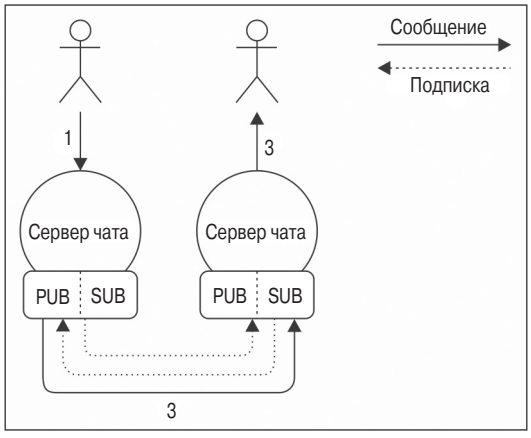

Рис. 11.9 ❖ Упрощенная схема чата на основе библиотеки ØMQ

На рис. 11.9 показано движение информации при наличии двух экземпляров приложения чата, но та же идея применима для случая N экземпляров. Как следует из

схемы, каждый узел должен знать о существовании других узлов системы, чтобы иметь возможность установить все необходимые соединения. Из нее также видно, что подписки следуют из сокета SUB в сокет PUB, в то время как сообщения передаются в обратном направлении.

Для проверки примера из этого раздела необходимо установить двоичные файлы библиотеки ØMQ. Более подробные сведения об этом можно найти на странице http://zeromq.org/intro:get-the-software. Примечание: пример тестировался с версией 4.0 библиотеки ØMQ.

Использование сокетов PUB/SUB библиотеки ØMQ

Посмотрим, как эта схема работает на практике, внеся следующие изменения в сервер чата (здесь приведены только измененные фрагменты):

```
// ...
const args = require('minimist')(process.argv.slice(2));    //[1]
const zmq = require('zmq');
const pubSocket = zmq.socket('pub');                        //[2]
pubSocket.bind(`tcp://127.0.0.1:${args['pub']}`);

const subSocket = zmq.socket('sub');                        //[3]
const subPorts = [].concat(args['sub']);
subPorts.forEach(p => {
  console.log(`Subscribing to ${p}`);
  subSocket.connect(`tcp://127.0.0.1:${p}`);
});
subSocket.subscribe('chat');

// ...
ws.on('message', msg => {                                   //[4]
  console.log(`Message: ${msg}`);
  broadcast(msg);
  pubSocket.send(`chat ${msg}`);
});
//...

subSocket.on('message', msg => {                            //[5]
  console.log(`From other server: ${msg}`);
  broadcast(msg.toString().split(' ')[1]);
});

// ...
server.listen(args['http'] || 8080);
```

Приведенный выше код явно показывает, что логика приложения стала несколько сложнее, но она все еще достаточно проста, если учесть, что здесь реализуется шаблон «Публикация/подписка» с прямой распределенной передачей сообщений. Посмотрим, как действуют все части сервера.

1. Загружается пакет `zmq` (https://npmjs.org/package/zmq), который, по сути, является интерфейсом подключения библиотеки ØMQ к приложениям Node.js. Также загружается пакет `minimist` (https://npmjs.org/package/minimist) для парсинга аргументов командной строки; он необходим нам для извлечения именованных аргументов.

2. Затем сразу же создается сокет PUB и подключается к порту, извлеченному из аргумента командной строки --pub.
3. Потом создается сокет SUB и подключается к сокетам PUB других экземпляров приложения. Порты целевых сокетов PUB извлекаются из аргументов --sub командной строки (их может быть несколько). Затем создаются нужные подписки со строкой chat в качестве фильтра. То есть будут приниматься только сообщения, начинающиеся с префикса chat.
4. Когда веб-сокет принимает новое сообщение, оно передается всем подключенным клиентам и публикуется в сокет PUB. Но перед этим в начало сообщения добавляются строка chat и пробел; благодаря этому его примут все подписчики, использующие строку chat в качестве фильтра.
5. Начинается прием сообщений, поступающих на сокет SUB. Из принятых сообщений удаляется префикс chat, и затем они рассылаются всем клиентам, подключенным к серверу веб-сокета.

Реализация распределенной системы, интегрированной с помощью шаблона «Публикация/подписка», готова!

Для ее проверки запустим три экземпляра приложения, соединив их сокеты PUB и SUB:

```
node app --http 8080 --pub 5000 --sub 5001 --sub 5002
node app --http 8081 --pub 5001 --sub 5000 --sub 5002
node app --http 8082 --pub 5002 --sub 5000 --sub 5001
```

Первая команда запускает экземпляр HTTP-сервера, прослушивающий порт 8080, привязывает сокет PUB к порту 5000, а сокет SUB – к портам 5001 и 5002, которые должны прослушивать сокеты PUB других двух экземпляров. Другие две команды также выполняют подобные действия.

Первое, на что нужно обратить внимание: библиотеку ØMQ не смущает, если порт, соответствующий сокету PUB, окажется недоступен. Например, во время выполнения первой команды порты 5001 и 5002 никем не прослушиваются, однако библиотека ØMQ не возбуждает исключения. Это связано с тем, что в библиотеке ØMQ имеется механизм повторного подключения, который будет пытаться установить соединение с этими портами через регулярные интервалы времени. Эта особенность также может пригодиться в случае, если какой-то узел остановится и его потребуется перезапустить. Тот же подход применяется к сокету PUB: при отсутствии подписок он будет просто удалять все сообщения, но продолжит работу.

Сейчас можно с помощью браузера подключиться к любым экземплярам сервера и убедиться, что сообщения благополучно передаются на все серверы чата.

В предыдущем примере предполагается статическая архитектура, когда количество экземпляров и их адреса известны заранее. Однако можно применить реестр служб, как описывалось в предыдущей главе, и реализовать динамическое подключение экземпляров. Следует также отметить, что на основе библиотеки ØMQ можно реализовать брокера, используя при этом те же элементы, что были здесь продемонстрированы.

Надежная подписка

Важной абстракцией в системе обмена сообщениями является **очередь сообщений** (Message Queue, MQ). При наличии очереди сообщений отправителю и получателю

(или получателям) сообщений необязательно быть одновременно активными, поскольку система очередей обеспечивает хранение сообщений, пока получатель не примет их. Эта модель поведения обратна парадигме «отправил и забыл», в которой подписчик может получать сообщения, только когда он подключен к системе обмена сообщениями.

Подписка, гарантирующая доставку всех сообщений, даже тех, что были отправлены, когда подписчик был отключен, называется **надежной подпиской**.

Протокол MQTT определяет уровень **качества обслуживания** (Quality of Service, QoS) для сообщений, передаваемых между отправителем и получателем. Эти уровни можно использовать для описания надежности любой другой системы обмена сообщениями (не только по протоколу MQTT):

- **QoS0**, **не более одного раза**: по-другому называется «послать и забыть», то есть сообщение не сохраняется и доставка не подтверждается. Это означает, что сообщение может быть потеряно в случае сбоя или отключения получателя;
- **QoS1**, **не менее одного раза**: сообщение гарантированно будет получено хотя бы один раз, но может быть продублировано, например если получатель завершит работу из-за ошибки до того, как уведомит отправителя. Это означает, что сообщения сохраняются, только если возникает необходимость повторной отправки;
- **QoS2**, **строго один раз**: это самый надежный из уровней QoS, поскольку он гарантирует, что сообщение будет получено один и только один раз. Это достигается за счет использования более медленного и громоздкого механизма подтверждения доставки сообщений.

Более подробные сведения о протоколе MQTT можно найти на странице http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html#qos-flows.

Как уже упоминалось, система надежной доставки сообщений должна использовать очередь сообщений для их хранения в то время, когда подписчики отключены. Очередь может храниться в памяти или на диске, что позволит восстановить сообщения даже после перезагрузки или завершения работы из-за ошибки брокера. На рис. 11.10 изображена схема реализации надежной подписки с помощью очереди сообщений.

Надежная подписка является самой важной областью применения очереди сообщений, но не единственной, как будет показано ниже.

Команды публикации/подписки в Redis реализуют механизм «послал и забыл» (QoS0). Тем не менее Redis с успехом можно использовать для реализации надежной подписки, используя комбинацию других команд (не используя непосредственно его реализацию шаблона «Публикация/подписка»). Описание этого способа можно найти в следующих статьях: http://davidmarquis.wordpress.com/2013/01/03/reliable-delivery-message-queues-with-redis; http://www.ericjperry.com/redis-message-queue.

Библиотека ØMQ определяет несколько шаблонов поддержки надежной подписки, но реализация этого механизма возлагается на разработчиков.

Введение в протокол AMQP

Очередь сообщений обычно используется в ситуациях, когда потеря сообщений недопустима, как, например, в банковских или финансовых системах. Обычно это означает, что типичные реализации очереди сообщений корпоративного уровня очень

сложны и используют сверхнадежные протоколы и постоянные хранилища, гарантирующие доставку сообщений даже при наличии неисправностей. По этой причине разработка промежуточного программного обеспечения для обмена сообщениями на корпоративном уровне на протяжении многих лет было прерогативой гигантов, таких как Oracle и IBM, каждый из которых реализовал свой собственный протокол, что привело к закрытости этой области. К счастью, по прошествии нескольких лет системы обмена сообщениями прочно вошли в обиход благодаря появлению открытых протоколов, таких как AMQP, STOMP и MQTT. Чтобы понять, как работают системы сообщений с очередями, познакомимся с протоколом AMQP; это поможет понять, как использовать типовой прикладной интерфейс, реализующий данный протокол.

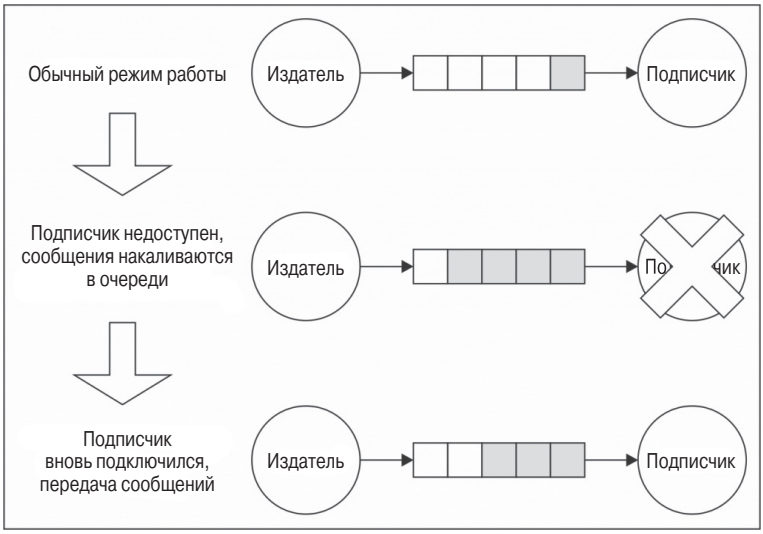

Рис. 11.10 ❖ Реализация надежной подписки с помощью очереди сообщений

Протокол **AMQP** – открытый стандартный протокол, поддерживаемый многими системами сообщений с очередями. Помимо общего коммуникационного протокола, он также определяет модель для описания маршрутизации, фильтрации, организации очередей, надежности и безопасности. В AMQP имеются три основных компонента.

- **Очередь**: структура данных, отвечающая за хранение сообщений, извлекаемых получателями. Сообщения из очереди передаются одному или нескольким получателям, которыми, по сути, являются наши приложения. Если к одной очереди подключится несколько получателей, сообщения будут распределяться между ними. Очереди могут быть следующих видов:
  - **Надежные**: очередь автоматически восстанавливается при перезапуске брокера. Надежность не означает, что будет сохранено все содержимое очереди; на диск будут сохранены только сообщения, отмеченные как хранимые, и они же будут восстановлены в случае перезагрузки;
  - **Эксклюзивные**: к очереди может быть подключен только один конкретный подписчик. После закрытия соединения очередь уничтожается;

  - **Автоматически удаляемые**: очередь удаляется после отключения последнего подписчика.
- **Коммутатор**: место публикации сообщений. Коммутатор направляет сообщения в одну или несколько очередей в зависимости от реализуемого алгоритма:
  - **Прямое коммутирование**: выбор маршрута определяется путем сопоставления полного ключа маршрутизации (например, `chat.msg`);
  - **Тематическое коммутирование**: выбор маршрута определяется путем сопоставления ключа с шаблоном (например, `chat.#` соответствует всем ключам маршрутизации, начинающимся с `chat`);
  - **Разветвляющее коммутирование**: сообщения рассылаются во все подключенные очереди, независимо от ключа маршрутизации.
- **Связь**: между коммутаторами и очередями. Также определяет ключ маршрутизации или шаблон для фильтрации сообщений, поступающих от коммутатора.

Эти компоненты управляются брокером, предоставляющим программный интерфейс для создания и управления ими. Подключаясь к брокеру, клиент создает канал – абстракцию соединения, – который отвечает за поддержание связи с брокером.

В AMQP шаблон надежной подписки можно реализовать путем создания очереди любого вида, кроме эксклюзивной и автоматически удаляемой.

На рис. 11.11 показано, как действуют все эти компоненты вместе.

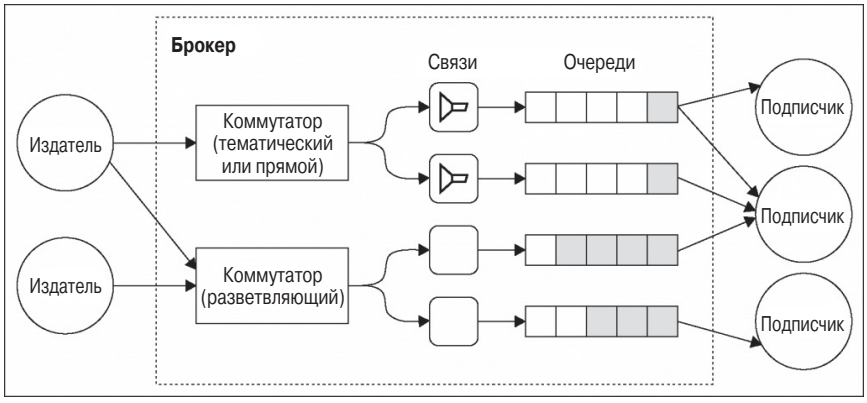

Рис. 11.11 ❖ Модель протокола **AMQP**

Протокол AMQP имеет намного более сложную модель, чем использовавшиеся ранее системы обмена сообщениями (Redis и ØMQ), но он обеспечивает такой набор возможностей и степень надежности, которых сложно было бы добиться, используя только примитивные механизмы публикации/подписки.

Более подробное введение в модель AMQP можно найти на веб-сайте проекта RabbitMQ https://www.rabbitmq.com/tutorials/amqp-concepts.html.

Надежная подписка с помощью AMQP и RabbitMQ

А теперь используем на практике все, что мы узнали о надежной подписке и AMQP, и реализуем небольшой пример. Типичной ситуацией, когда важно не потерять со-

общения, является синхронизация служб в архитектуре, основанной микрослужбах; этот шаблон интеграции уже был описан в предыдущей главе. Если вы желаете использовать брокера для поддержания служб в согласованном состоянии, важно позаботиться о сохранности любых сведений, поскольку в противном случае службы рано и поздно окажутся несогласованными.

Разработка службы истории для приложения чата А теперь расширим небольшое приложение чата, основанное на микрослужбах. Добавим в него службу истории, которая будет сохранять сообщения в базе данных, чтобы, подключившись к чату, клиент смог бы через эту службу получить всю историю чата. Служба истории будет интегрироваться с сервером чата с помощью брокера RabbitMQ (https://www.rabbitmq.com) и AMQP.

На рис. 11.12 показана схема планируемой архитектуры.

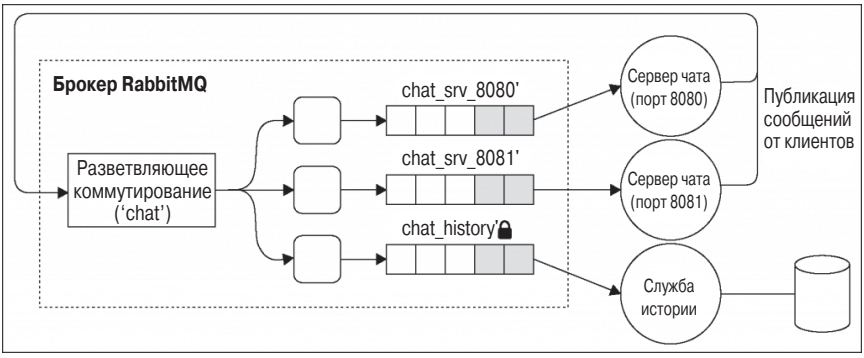

Рис. 11.12 ❖ Архитектура чата с поддержкой истории

Как показано на рис. 11.12, мы собираемся использовать один коммутатор с разветвляющим коммутированием; нам не требуется более сложная коммутация. Далее мы создадим по одной эксклюзивной очереди для каждого экземпляра сервера чата – очередь не должна сохранять сообщений, поступающих, пока сервер чата отключен, потому что эту задачу будет решать служба истории, которая может также реализовать поддержку более сложных запросов на получение сохраненных сообщений. По сути, это означает, что серверы чата не являются надежными подписчиками и их очереди будут уничтожены при закрытии соединений.

Служба истории, напротив, не может позволить себе потерять даже одно сообщение, иначе она не будет соответствовать своему назначению. Очередь для нее должна быть надежной, чтобы любое сообщение, опубликованное, пока служба истории отключена, сохранялось в очереди и было доставлено, когда она вновь подключится.

В качестве хранилища для службы истории будет использоваться уже знакомый пакет LevelUP, а для подключения к RabbitMQ с использованием протокола AMQP – пакет `amqplib` (https://npmjs.org/package/amqplib).

Для проверки следующего примера потребуется действующий сервер RabbitMQ, прослушивающий порт по умолчанию. За дополнительной информацией обращайтесь к официальной документации на странице http://www.rabbitmq.com/download.html.

Реализация надежной службы истории с использованием AMQP А теперь реализуем службу истории! Для этого создадим автономное приложение (типичную микрослужбу), поместив его код в модуль `historySvc.js`. Модуль будет состоять из двух частей: HTTP-сервера, для передачи истории чата клиентам, и AMQP-потребителя, отвечающего за сбор сообщений чата и сохранение их в локальной базе данных:

```
const level = require('level');
const timestamp = require('monotonic-timestamp');
const JSONStream = require('JSONStream');
const amqp = require('amqplib');
const db = level('./msgHistory');

require('http').createServer((req, res) => {
  res.writeHead(200);
  db.createValueStream()
    .pipe(JSONStream.stringify())
    .pipe(res);
}).listen(8090);

let channel, queue;
amqp
  .connect('amqp://localhost')                          // [1]
  .then(conn => conn.createChannel())
  .then(ch => {
    channel = ch;
    return channel.assertExchange('chat', 'fanout');    // [2]
  })
  .then(() => channel.assertQueue('chat_history'))      // [3]
  .then((q) => {
    queue = q.queue;
    return channel.bindQueue(queue, 'chat');            // [4]
  })
  .then(() => {
    return channel.consume(queue, msg => {              // [5]
      const content = msg.content.toString();
      console.log(`Saving message: ${content}`);
      db.put(timestamp(), content, err => {
        if (!err) channel.ack(msg);
      });
    });
  })
  .catch(err => console.log(err));
```

Как видите, AMQP требует определенной настройки, заключающейся в создании и подключении всех компонентов модели. Также следует отметить, что `amqplib` по умолчанию поддерживает объекты `Promise`, поэтому они используются для упрощения асинхронных действий приложения. Рассмотрим подробно, как это работает.

1) сначала устанавливается соединение с брокером AMQP – RabbitMQ. Затем создается канал, похожий на сеанс, для поддержки состояния взаимодействий;
2) далее создается коммутатор `chat`. Как уже упоминалось, это разветвляющий коммутатор. Команда `assertExchange()` проверяет наличие готового коммутатора в брокере, если коммутатор отсутствует, он будет создан.

3) также создается очередь chat_history. По умолчанию очередь является надежной, а не эксклюзивной, и не удаляется автоматически, поэтому нет необходимости указывать дополнительные параметры для поддержки надежных подписчиков;
4) далее очередь связывается с созданным ранее коммутатором. Здесь не нужны какие-либо параметры, такие как ключ маршрутизации или шаблон, поскольку разветвляющее коммутирование не выполняет никакой фильтрации;
5) и наконец, можно начать прием сообщений из только что созданной очереди. Сообщения сохраняются в базе данных LevelDB с использованием отметок времени в качестве ключей (https://npmjs.org/package/monotonic-timestamp), чтобы обеспечить их сортировку по дате и времени. Интересно отметить, что вызов channel.ack(msg), подтверждающий прием сообщения, производится только после успешного сохранения сообщения в базе данных. Если брокер не получит подтверждения, сообщение остается в очереди для повторной обработки. Это еще одна замечательная особенность AMQP, обеспечивающая надежность службы на совершенно новом уровне. Если не требуется явно подтверждать прием сообщений, можно передать параметр {noAck:true} в вызов channel.consume().

Интеграция приложения чата с помощью AMQP Для интеграции серверов чата с использованием AMQP используются примерно те же компоненты, что были реализованы для службы истории, поэтому приведем здесь не полный код, а только фрагменты, демонстрирующие создание очереди и публикацию нового сообщения в коммутаторе (файл app.js):

```
// ...
  .then(() => {
    return channel.assertQueue(`chat_srv_${httpPort}`, {exclusive: true});
  })
// ...
  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    channel.publish('chat', '', new Buffer(msg));
  });
// ...
```

Как уже упоминалось, для нормальной работы сервера чата не требуется надежная подписка, ему достаточно поддержки парадигмы «отправил и забыл». Поэтому при создании очереди мы передаем параметр {exclusive: true}, указывающий, что очередь предназначена только для текущего соединения и ее можно уничтожить, как только сервер чата завершит работу.

Публикация нового сообщения также упрощается; для этого достаточно указать целевой коммутатор (chat) и ключ маршрутизации, который в данном случае является пустой строкой (''), потому что используется разветвляющий коммутатор.

Теперь можно запустить чат с усовершенствованной архитектурой. Для этого запустим два сервера чата и службу истории:

```
node app 8080
node app 8081
node historySvc
```

Интересно проверить, как на работе системы отразится приостановка службы истории. Если остановить службу истории и продолжить отправлять сообщения из пользовательского веб-интерфейса, тогда после повторного запуска службы истории она сразу же получит все пропущенные сообщения. Это прекрасная демонстрация работы шаблона надежной подписки!

Приятно наблюдать, как подход на основе микрослужб позволяет системе продолжать работу в отсутствие одного из ее компонентов, а именно службы истории. Это вызовет временное сокращение функциональных возможностей (история чата будет не доступна), но пользователи по-прежнему будут иметь возможность обмениваться сообщениями в режиме реального времени. Отлично!

Шаблоны конвейеров и распределения заданий

В *главе 9* «*Дополнительные рецепты асинхронной обработки*» мы узнали, как делегировать дорогостоящие вычислительные задания нескольким локальным процессам. Но, несмотря на всю эффективность этого решения, оно не позволяет распространить масштабирование за рамки одного компьютера. В этом разделе мы рассмотрим реализацию аналогичного шаблона в распределенной архитектуре с использованием удаленных рабочих процессов, расположенных в любой точке сети.

Идея заключается в том, чтобы воспользоваться шаблоном обмена сообщениями для распределения заданий между несколькими компьютерами. Заданиями могут быть отдельные операции или фрагменты больших задач, поделенных согласно принципу «*разделяй и властвуй*».

В логической архитектуре на рис. 11.13 можно распознать уже знакомый шаблон.

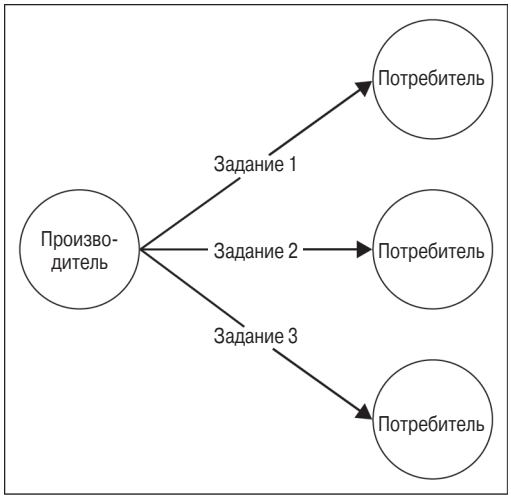

Рис. 11.13 ❖ Распределение заданий между несколькими компьютерами

Как видите, шаблон «Публикация/подписка» не подходит для приложений этого типа, поскольку одно и то же задание не должно передаваться нескольким рабочим процессам. Вместо этого необходим шаблон распределения сообщений, аналогичный

шаблону распределения нагрузки, который отправлял бы каждое сообщение только одному потребителю (или рабочему процессу в данном случае). В терминологии обмена сообщениями этот шаблон называется **«Конкурирующие потребители»**, **«Разветвляющее распределение»** или **«Вентилятор»**.

Одно из важных отличий от распределения HTTP-нагрузки, о которой рассказывалось в предыдущей главе, заключается в том, что здесь потребителям отводится более активная роль. Фактически, как будет показано ниже, в данном шаблоне не производитель подключается к потребителям, а потребители подключаются к производителю для получения новых заданий. Это большое преимущество для масштабируемой системы, позволяющее плавно увеличить число рабочих процессов без изменения производителя и без необходимости использовать реестр служб.

Кроме того, в универсальной системе обмена сообщениями нет необходимости иметь связь вида «запрос/ответ» между производителем и рабочими процессами. Вместо этого предпочтительнее использовать одностороннюю асинхронную связь, более пригодную для распараллеливания и масштабирования. В такой архитектуре сообщения всегда можно перемещать в одном направлении, создавая конвейеры, как показано на рис. 11.14.

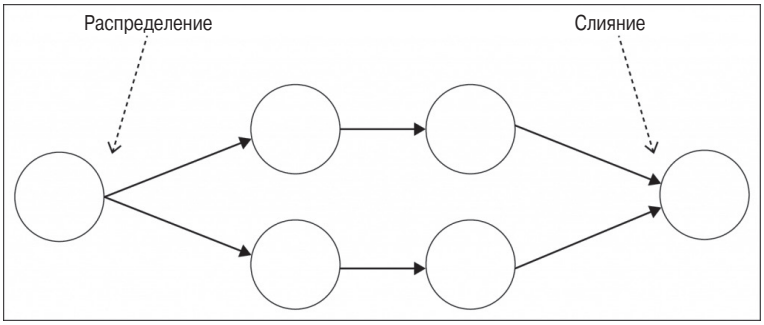

Рис. 11.14 ❖ Архитектура с односторонней асинхронной связью

Конвейеры позволяют создавать очень сложные архитектуры обработки без использования синхронных связей вида «запрос/ответ», что обеспечивает высокую скорость и пропускную способность. На рис. 11.14 показано, как сообщения могут распределяться между множеством рабочих процессов, передаваться другим единицам обработки, а затем объединяться в один узел, который часто называют **приемником**.

В этом разделе мы сосредоточимся на создании строительных блоков таких архитектур и проанализируем два основных варианта их реализации: с прямой передачей и передачей через брокера.

Комбинацию конвейера с шаблоном распределения заданий часто называют **параллельным конвейером**.

Шаблон распределения/слияния в ØMQ

Мы уже познакомились с некоторыми возможностями ØMQ для создания распределенных архитектур с непосредственной передачей сообщений. В предыдущем разделе мы использовали сокеты `PUB` и `SUB`, чтобы передать одно сообщение нескольким

потребителям, а теперь мы посмотрим, как организовать параллельный конвейер с использованием другой пары сокетов: PUSH и PULL.

Сокеты PUSH/PULL

По названиям нетрудно догадаться, что сокеты PUSH предназначены для *отправки* сообщений, а сокеты PULL – для их *получения*. Такое сочетание может показаться тривиальным, но некоторые свойства делают их идеальными для создания однонаправленной коммуникационной системы:

- оба сокета могут работать в режиме *подключения* или в режиме *привязки*. Другими словами, можно создать сокет PUSH и привязать его к локальному порту для приема входящих соединений из сокета PUSH, или наоборот – сокет PULL может принимать соединения из сокета PUSH. Сообщения всегда перемещаются в одном направлении – от сокета PUSH к сокету PULL; различными могут быть только инициаторы соединений. Режим привязки является лучшим решением *надежных* узлов, таких как, например, поставщик заданий и приемник, в то время как режим подключения идеально подходит для *промежуточных* узлов, таких как, например, исполнители заданий. Это позволяет произвольно менять количество промежуточных узлов, не затрагивая более надежных узлов;
- если несколько сокетов PULL подключить к одному сокету PUSH, сообщения равномерно распределяются по всем сокетам PULL, что практически обеспечивает распределение нагрузки (одноранговое распределение нагрузки!). С другой стороны, сокет PULL, принимающий сообщения от нескольких сокетов PUSH, будет обрабатывать сообщения с помощью справедливой системы очередей, то есть извлекать их равномерно из всех источников – циклический алгоритм применительно к входящим сообщениям;
- сообщения, отправленные через сокет PUSH, который не имеет подключенных сокетов PULL, не теряются – они хранятся в очереди производителя, пока узел вновь не подключится и не начнет извлекать сообщения.

Теперь понятно, что отличает ØMQ от традиционных веб-служб и почему эта библиотека считается идеальным инструментом для создания систем обмена сообщениями любого вида.

Создание распределенного взломщика хеш-сумм с помощью ØMQ

Рассмотрим пример простого приложения, чтобы увидеть, как работают только что описанные сокеты PUSH/PULL.

Таким приложением для нас станет взломщик хеш-сумм – система, использующая технологию полного перебора для поиска соответствия заданной хеш-сумме (MD5, SHA1 и т. д.) всех возможных вариантов из символов заданного алфавита. Это серьезная параллельная рабочая нагрузка (https://ru.wikipedia.org/wiki/Чрезвычайная_параллельность), идеально подходящая для демонстрации возможностей параллельных конвейеров.

В приложении будет реализован типичный параллельный конвейер с узлом для создания и распределения заданий между несколькими рабочими процессами, а также узел, собирающий полученные результаты. Описанную систему легко реализовать с помощью ØMQ, используя архитектуру на рис. 11.15.

В этой архитектуре имеется *вентилятор*, генерирующий все возможные наборы символов заданного алфавита и передающий их нескольким рабочим процессам, ко-

торые, в свою очередь, вычисляют хеш-сумму каждого варианта и проверяют ее соответствие заданной. Если совпадение найдено, результат передается узлу сборщика результатов (приемнику).

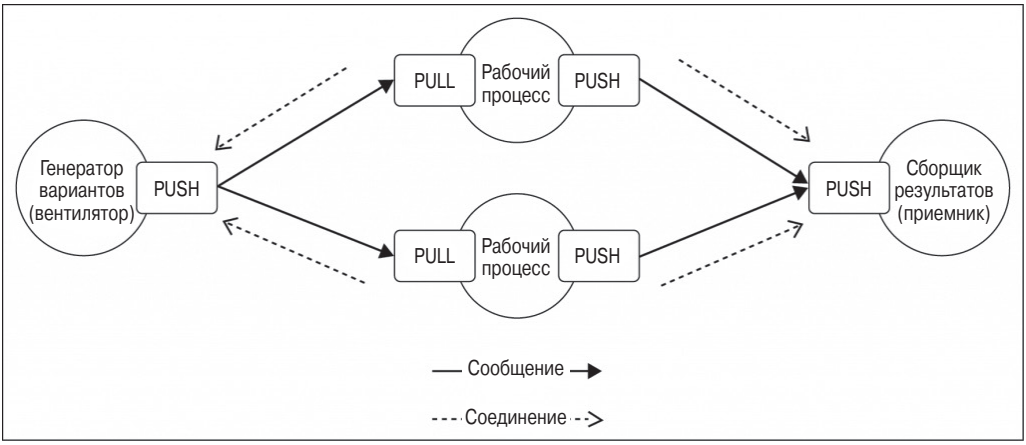

Рис. 11.15 ❖ Архитектура параллельных конвейеров

Надежными узлами в этой архитектуре являются вентилятор и приемник, а переходными узлами – рабочие процессы. Это означает, что каждый рабочий процесс подключает свой сокет `PULL` к вентилятору, а сокет `PUSH` – к приемнику. Следуя этой схеме, можно запускать и останавливать сколько угодно рабочих процессов, без перенастройки вентилятора и приемника.

Реализация вентилятора Начнем реализацию системы с создания нового модуля генератора заданий, который поместим в файл `ventilator.js`:

```
const zmq = require('zmq');
const variationsStream = require('variations-stream');
const alphabet = 'abcdefghijklmnopqrstuvwxyz';
const batchSize = 10000;
const maxLength = process.argv[2];
const searchHash = process.argv[3];

const ventilator = zmq.socket('push');              // [1]
ventilator.bindSync("tcp://*:5000");

let batch = [];
variationsStream(alphabet, maxLength)
  .on('data', combination => {
    batch.push(combination);
    if (batch.length === batchSize) {               // [2]
      const msg = {searchHash: searchHash, variations: batch};
      ventilator.send(JSON.stringify(msg));
      batch = [];
    }
  })
  .on('end', () => {
```

```
    //отправить оставшиеся комбинации
    const msg = {searchHash: searchHash, variations: batch};
    ventilator.send(JSON.stringify(msg));
  });
```

Чтобы избежать создания слишком большого числа вариантов, генератор использует только строчные буквы английского алфавита и ограничивает размер генерируемых слов. Это ограничение задается с помощью аргумента командной строки (`maxLength`) вместе с искомой хеш-суммой (`searchHash`). Для создания всех вариантов с применением потокового интерфейса используется библиотека `variations-stream` (https://npmjs.org/package/variations-stream).

Но нас больше интересует фрагмент кода, который занимается распределением заданий между рабочими процессами.

1. Сначала создается сокет `PUSH` и привязывается к локальному порту `5000`, к которому рабочие процессы подключат свои сокеты `PULL` для получения заданий.
2. Созданные варианты группируются в пакеты по 10 000 элементов, а затем формируется сообщение, содержащее хеш для сопоставления и пакет слов для проверки. Это, по сути, объект задания, который получают рабочие процессы. Вызовом метода `send()` сокета `ventilator` сообщение передается следующему доступному рабочему процессу в соответствии с циклическим алгоритмом.

Реализация рабочего процесса А теперь подошла очередь реализации рабочего процесса (`worker.js`):

```
const zmq = require('zmq');
const crypto = require('crypto');
const fromVentilator = zmq.socket('pull');
const toSink = zmq.socket('push');

fromVentilator.connect('tcp://localhost:5016');
toSink.connect('tcp://localhost:5017');

fromVentilator.on('message', buffer => {
  const msg = JSON.parse(buffer);
  const variations = msg.variations;
  variations.forEach( word => {
    console.log(`Processing: ${word}`);
    const shasum = crypto.createHash('sha1');
    shasum.update(word);
    const digest = shasum.digest('hex');
    if (digest === msg.searchHash) {
      console.log(`Found! => ${word}`);
      toSink.send(`Found! ${digest} => ${word}`);
    }
  });
});
```

Как уже упоминалось, рабочий процесс представляет промежуточный узел в архитектуре, поэтому его сокеты должны подключаться к удаленному узлу, а не принимать входящие подключения. Это именно то, для чего в рабочем процессе создаются два сокета:

- сокет `PULL` подключается к генератору для получения заданий;
- сокет `PUSH` подключается к приемнику для передачи результатов.

Кроме того, рабочий процесс выполняет простое задание: для каждого сообщения перебрать содержащийся в нем пакет слов, для каждого слова вычислить контрольную сумму по алгоритму SHA1 и сравнить ее с переданным в сообщении значением `searchHash`. Если они совпадут, передать результат приемнику.

Реализация приемника В данном примере приемник просто собирает результаты и выводит в консоль сообщения, полученные от рабочих процессов. Вот как выглядит его код в файле `sink.js`:

```
const zmq = require('zmq');
const sink = zmq.socket('pull');
sink.bindSync("tcp://*:5017");

sink.on('message', buffer => {
  console.log('Message from worker: ', buffer.toString());
});
```

Обратите внимание, что приемник (подобно генератору заданий) также является надежным узлом архитектуры, и поэтому мы выполняем привязку сокета `PULL`, а не пытаемся подключить его к сокетам `PUSH` рабочих процессов.

Запуск приложения Теперь все готово к проверке приложения. Сначала запустим пару рабочих процессов и приемник:

```
node worker
node worker
node sink
```

Далее запустим генератор, указав максимальную длину генерируемых слов и искомую контрольную сумму SHA1:

```
node ventilator 4 f8e966d1e207d02c44511a58dccff2f5429e9a3b
```

Генератор начнет создавать все возможные слова, с длиной не более четырех символов, и их передачу запущенным рабочим процессам вместе с заданной контрольной суммой. Результаты вычислений, если таковые появятся, будут выводиться в окно терминала приложением приемника.

Конвейеры и конкурирующие потребители в AMQP

В предыдущем разделе рассматривалась реализация параллельного конвейера с прямой передачей заданий. Теперь реализуем тот же шаблон, но с использованием полноценного брокера сообщений, такого как RabbitMQ.

Прямые связи и конкурирующие потребители

В одноранговой конфигурации идея конвейера сообщений выглядит очень простой. Но при наличии брокера сообщений взаимосвязи между различными узлами системы несколько сложнее. Поскольку брокер играет роль посредника, часто неизвестно, кем является другая сторона, получающая сообщения. Например, когда сообщения отправляются с помощью AMQP, они не доставляются непосредственно получателям, а сначала попадают в коммутатор и уже затем в очередь. И наконец, брокер решает, куда направить сообщение, основываясь на правилах в коммутаторе, привязках и очередях назначения.

При реализации шаблонов конвейера и распределения заданий с помощью систем, подобных AMQP, следует гарантировать, что любое сообщение получит только один потребитель, но это невозможно, если к коммутатору может быть подключено несколько очередей. Решением является отправка сообщения непосредственно в очередь назначения, в обход коммутатора; это позволит гарантировать, что сообщение попадет только в одну очередь. Этот шаблон связи называется **точка-точка**, или **прямая передача**.

Как только мы получим возможность посылать сообщения непосредственно в конкретную очередь, задача реализации шаблона распределения заданий будет наполовину решена. Следующий шаг естественно вытекает из предыдущего: если одна очередь обслуживает несколько потребителей, сообщения между ними должны распределяться равномерно, путем реализации разветвляющего распределения. В контексте брокера сообщений это называется шаблоном **конкурирующих потребителей**.

Реализация взломщика хеш-сумм с помощью AMQP

Как только что было показано, коммутаторы в брокере распределяют сообщения между несколькими потребителями. Учитывая это, приступим к реализации взломщика хеш-сумм с использованием брокера AMQP (такого как, например, RabbitMQ). На рис. 11.16 изображена схема системы, которую требуется получить.

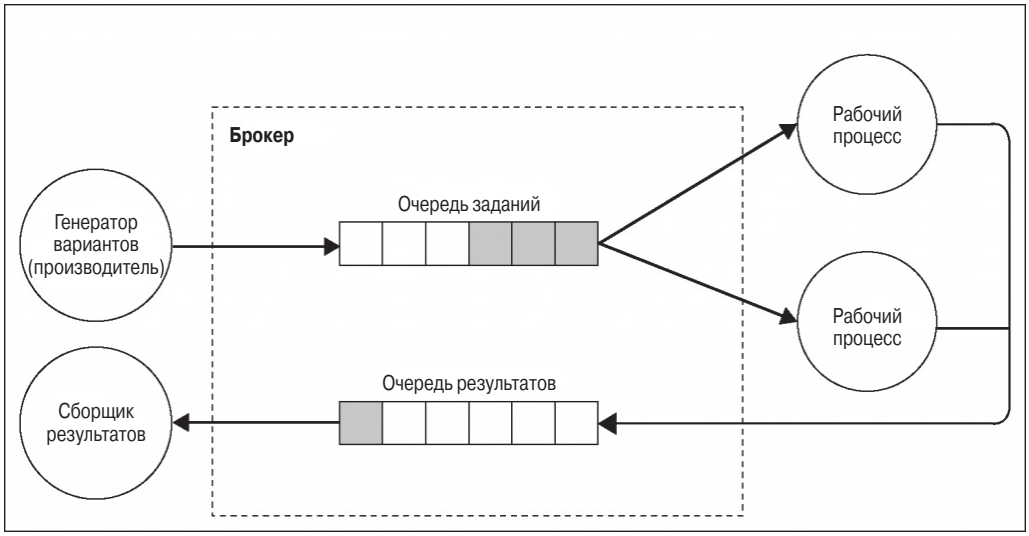

Рис. 11.16 ❖ Схема системы с брокером AMQP

Как уже упоминалось, для распределения заданий между несколькими рабочими процессами следует использовать одну очередь. На рис. 11.16 она названа *очередью заданий*. На другом конце очереди заданий находится группа рабочих процессов – *конкурирующих потребителей*, – каждый из которых будет извлекать разные сообщения из очереди. В результате получается, что несколько разных заданий будет выполняться параллельно в разных рабочих процессах.

Все результаты, полученные рабочими процессами, публикуются в другой очереди, которая называется *очередью результатов*, а затем обрабатываются сборщиком

результатов, что, по существу, эквивалентно слиянию. В архитектуре отсутствуют коммутаторы; сообщения передаются непосредственно в свою очередь, что соответствует реализации связи вида точка-точка.

Реализация производителя Посмотрим, как реализовать такую систему, начав с производителя (генератора вариантов). Его код идентичен тому, что был реализован в предыдущем разделе, за исключением фрагментов, касающихся обмена сообщениями (файл `producer.js`):

```
const amqp = require('amqplib');
//...

let connection, channel;
amqp
  .connect('amqp://localhost')
  .then(conn => {
    connection = conn;
    return conn.createChannel();
  })
  .then(ch => {
    channel = ch; produce();
  })
  .catch(err => console.log(err));

function produce() {
  //...
  variationsStream(alphabet, maxLength)
  .on('data', combination => {
    //...
    const msg = {searchHash: searchHash, variations: batch};
    channel.sendToQueue('jobs_queue',
      new Buffer(JSON.stringify(msg)));
    //...
  })
  //...
}
```

Как видите, отсутствие коммутатора значительно упрощает настройку связей AMQP. В реализации производителя не нужны даже очереди, так как требуется только опубликовать сообщения.

Наиболее важной деталью в коде выше является метод `channel.sendToQueue()`, который еще не рассматривался. Как следует из названия, этот метод отвечает за добавление сообщения в очередь, в нашем примере `jobs_queue`, в обход коммутаторов и механизма маршрутизации.

Реализация рабочего процесса С другой стороны очереди `jobs_queue` находятся рабочие процессы, принимающие поступающие задания (файл `worker.js`):

```
const amqp = require('amqplib');
//...

let channel, queue;
amqp
  .connect('amqp://localhost')
```

```
  .then(conn => conn.createChannel())
  .then(ch => {
    channel = ch;
    return channel.assertQueue('jobs_queue');
  })
  .then(q => {
    queue = q.queue; consume();
  })
//...
function consume() {
  channel.consume(queue, msg => {
    //...
    variations.forEach(word => {
      //...
      if(digest === data.searchHash) {
        console.log(`Found! => ${word}`);
        channel.sendToQueue('results_queue',
          new Buffer(`Found! ${digest} => ${word}`));
      }
    //...
    });
    channel.ack(msg);
  });
};
```

Вновь созданный рабочий процесс также очень похож на реализованный в предыдущем разделе с помощью ØMQ, за исключением части, касающейся обмена сообщениями. В реализации выше сначала проверяется наличие очереди `jobs_queue`, и если она имеется, начинается прием входящих заданий с помощью `channel.consume()`. Затем, при обнаружении совпадения, результат отправляется сборщику через очередь `results_queue`, снова с использованием связи точка-точка.

Когда одновременно действует несколько рабочих процессов, все они обслуживают одну и ту же очередь, благодаря чему нагрузка равномерно распределяется между ними.

Реализация сборщика результатов Сборщик результатов также является достаточно тривиальным модулем, который просто выводит все полученные сообщения в консоль (файл `collector.js`):

```
//...
  .then(ch => {
    channel = ch;
    return channel.assertQueue('results_queue');
  })
  .then(q => {
    queue = q.queue;
    channel.consume(queue, msg => {
      console.log('Message from worker: ', msg.content.toString());
    });
  })
//...
```

Запуск приложения Теперь все готово для проверки новой системы. Запустим пару рабочих процессов, которые будут подключаться к одной и той же очереди (`jobs_queue`), распределяющей нагрузку между ними:

```
node worker
node worker
```

Затем запустим модуль сборщика и производителя (передав максимальную длину слов и хеш для взлома):

```
node collector
node producer 4 f8e966d1e207d02c44511a58dccff2f5429e9a3b
```

В результате мы реализовали конвейер сообщений и шаблон конкурирующих потребителей с помощью только AMQP.

Шаблоны вида «Запрос/ответ»

При работе с системой обмена сообщениями часто используются однонаправленные асинхронные связи, примером которых является публикация/подписка.

Однонаправленные связи обеспечивают больше преимуществ, с точки зрения параллельной обработки и эффективности, но сами по себе они не позволяют решить всех проблем интеграции и коммуникации. Иногда добрый старый шаблон «Запрос/ответ» может оказаться просто идеальным инструментом. В таких ситуациях, когда не требуется ничего, кроме однонаправленных асинхронных каналов, важно уметь реализовать абстракции, позволяющие обмениваться сообщениями в режиме запрос/ответ. Именно об этом мы и поговорим далее.

Идентификатор корреляции

Первый шаблон, который мы рассмотрим, называется «**Идентификатор корреляции**» и служит основой для построения абстракции запроса/ответа поверх однонаправленных каналов.

Идея шаблона заключается в маркировке каждого запроса идентификатором, который затем получатель прикрепляет к ответу. Таким образом, отправитель запроса может сопоставить два сообщения и вернуть ответ соответствующему обработчику. Это элегантно решает проблему использования только однонаправленного асинхронного канала, позволяя передавать сообщения в любом направлении. Рассмотрим пример на рис. 11.17.

Схема на рис. 11.17 демонстрирует использование идентификатора для сопоставления ответов и запросов, даже если они отправляются и возвращаются в произвольном порядке.

Реализация абстракции «Запрос/ответ» с помощью идентификаторов корреляции

А теперь рассмотрим пример, выбрав наиболее простой тип однонаправленного канала типа точка-точка (прямое соединение двух узлов системы) с полнодуплексной связью (сообщения перемещаются в обоих направлениях).

К категории *простых каналов* относятся, например, веб-сокеты: они устанавливают соединение точка-точка между сервером и браузером, и сообщения по ним могут перемещаться в любом направлении. Еще одним примером является коммуникаци-

онный канал, создаваемый при запуске дочернего процесса с помощью `child_process.fork()`, который рассматривался в *главе 9 «Дополнительные рецепты асинхронной обработки»*. Этот канал также является асинхронным: он связывает родительский процесс только с дочерним процессом и позволяет передавать сообщения в любом направлении. Это, вероятно, самый основной канал в этой категории, поэтому используем в следующем примере именно его.

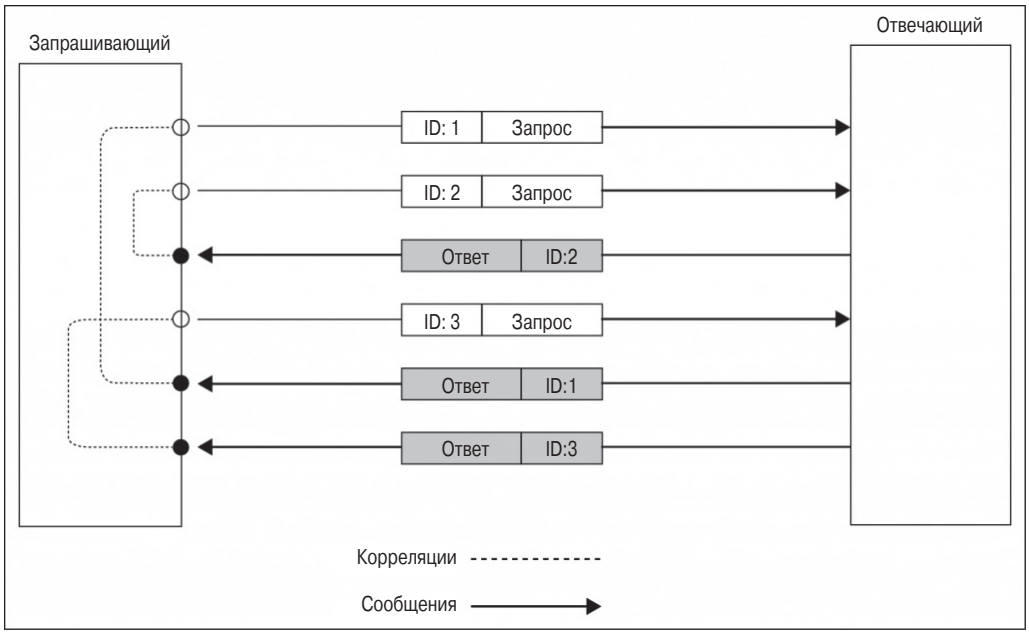

Рис. 11.17 ❖ Идентификация запросов и ответов

Наша цель – реализовать в следующем приложении абстракцию для обертывания канала, соединяющего родительский и дочерний процессы. Эта абстракция должна обеспечить связь вида «Запрос/ответ», автоматически маркируя каждый запрос идентификатором и затем сопоставляя идентификатор ответа со списком ожидающих ответа запросов.

В *главе 9 «Дополнительные рецепты асинхронной обработки»* мы видели, что родительский процесс имеет доступ к каналу связи с дочерним процессом посредством двух примитивов:

- `child.send(message);`
- `child.on('message',callback).`

Аналогично дочерний процесс имеет доступ к каналу связи с родительским процессом посредством:

- `process.send(message);`
- `process.on('message',callback).`

То есть интерфейс канала в родительском процессе идентичен интерфейсу в дочернем процессе, что позволяет создать общую абстракцию, дающую возможность посылать запросы с обоих концов канала.

Абстрагирование запроса Начнем создание абстракции с рассмотрения той ее части, которая отвечает за отправку новых запросов (файл `request.js`):

```
const uuid = require('node-uuid');
module.exports = channel => {
  const idToCallbackMap = {};                          // [1]

  channel.on('message', message => {                   // [2]
    const handler = idToCallbackMap[message.inReplyTo];
    if(handler) {
      handler(message.data);
    }
  });

  return function sendRequest(req, callback) {         // [3]
    const correlationId = uuid.v4();
    idToCallbackMap[correlationId] = callback;
    channel.send({
      type: 'request',
      data: req,
      id: correlationId
    });
  };
};
```

Эта абстракция запроса работает следующим образом:

1) вокруг функции `request` создается замыкание. Целью этого приема является переменная `idToCallbackMap`, обеспечивающая связь между запросами и обработчиками ответов;
2) с момента вызова фабрики начинается прием входящих сообщений. Если идентификатор корреляции в сообщении (в свойстве `inReplyTo`) соответствует одному из хранящихся в переменной `idToCallbackMap`, делается вывод, что только что получен ответ и извлекается ссылка на обработчик, связанный с ответом, и производится его вызов с передачей данных, содержащихся в сообщении;
3) в заключение возвращается функция, которая будет использоваться для отправки новых запросов. Ее задача – сгенерировать идентификатор корреляции с помощью пакета `node-uuid` (https://npmjs.org/package/node-uuid), а затем объединить в общий пакет данные запроса с идентификатором корреляции и типом сообщения.

Это все, что касается модуля `request`, теперь перейдем к следующей части.

Абстрагирование ответа Осталось рассмотреть реализацию модуля, парного модулю `request.js`. Создадим еще один файл, `reply.js`, который будет содержать абстракцию обертки обработчика ответа:

```
module.exports = channel =>
{
  return function registerHandler(handler) {
    channel.on('message', message => {
      if (message.type !== 'request') return;
      handler(message.data, reply => {
        channel.send({
          type: 'response',
```

```
        data: reply,
        inReplyTo: message.id
      });
    });
  });
};
};
```

И снова модуль `reply` реализует фабрику, которая возвращает функцию для регистрации новых обработчиков ответов. При регистрации нового обработчика происходит следующее:

1) начинается прием входящих запросов, при получении которых сразу же вызывается обработчик, которому передаются данные из сообщения и функция обратного вызова для возврата ответа из обработчика;
2) завершая работу, обработчик вызовет переданную ему функцию со своим ответом. Затем создается пакет, куда включается идентификатор корреляции запроса (свойство `inReplyTo`), который передается обратно в канал.

Интересная особенность этого шаблона в том, что на платформе Node.js он очень легко реализуется; поскольку здесь все ориентировано на асинхронность, создание асинхронной связи вида «Запрос/ответ» на основе однонаправленного канала несильно отличается от любой другой асинхронной операции, особенно при применении абстракции для сокрытия деталей реализации.

Опробование полного цикла связи вида «Запрос/ответ» Теперь все готово для проверки вновь созданной абстракции асинхронной связи вида «Запрос/ответ». Создадим пример *отвечающей* стороны в файле `replier.js`:

```
const reply = require('./reply')(process);
reply((req, cb) => {
  setTimeout(() => {
    cb({sum: req.a + req.b});
  }, req.delay);
});
```

Здесь отвечающая сторона просто вычисляет сумму двух полученных чисел и возвращает результат после некоторой задержки (также задается в запросе). Это позволяет изменить порядок возврата ответов, чтобы он отличался от порядка отправки запросов, с целью удостовериться в правильной работе шаблона.

Заключительный шаг в реализации примера – создание запрашивающей стороны в файле `requestor.js`, которая также запускает отвечающую сторону с помощью `child_process.fork()`:

```
const replier = require('child_process')
                .fork(`${ dirname}/replier.js`;
const request = require('./request')(replier);

request({a: 1, b: 2, delay: 500}, res => {
  console.log('1 + 2 = ', res.sum);
  replier.disconnect();
});

request({a: 6, b: 1, delay: 100}, res => {
  console.log('6 + 1 = ', res.sum);
});
```

Запрашивающая сторона запускает отвечающую, а затем передает ей ссылку на абстракцию запросов. Далее выполняется несколько запросов, чтобы убедиться, что их корреляция с ответом осуществляется правильно.

Для проверки примера достаточно просто запустить модуль `requestor.js`. В результате в терминале должны появиться примерно такие строки:

```
6 + 1 = 7
1 + 2 = 3
```

Это подтверждает, что шаблон работает и ответы правильно связываются с запросами, независимо от порядка отправки и получения.

Обратный адрес

Идентификатор корреляции является основным шаблоном для создания связей вида «Запрос/ответ», основанных на однонаправленных каналах, но его недостаточно, если архитектура обмена сообщениями содержит несколько каналов или очередей, или когда имеется нескольких запрашивающих сторон. В таких ситуациях, в дополнение к идентификатору корреляции, необходимо также знать *обратный адрес*, то есть сведения, которые позволят отвечающей стороне вернуть ответ исходному отправителю запроса.

Реализация шаблона обратного адреса с помощью AMQP

При использовании AMQP обратный адрес является очередью, из которой запрашивающая сторона ожидает поступления ответов. Так как ответ должна получить только одна запросившая сторона, важно, чтобы для каждого потребителя имелась отдельная очередь. Из этого можно сделать вывод, что для связи с запрашивающей стороной необходимо использовать временные очереди и отвечающая сторона должна установить прямую связь с очередью возврата, чтобы иметь возможность возвращать ответы.

Схема на рис. 11.18 иллюстрирует этот случай.

Для реализации шаблона «Запрос/ответ» с использованием AMQP достаточно указать имя очереди ответов в свойствах сообщения, что позволит отвечающей стороне узнать, куда послать сообщение с ответом. Идея достаточно проста, поэтому перейдем сразу к ее реализации в реальном приложении.

Реализация абстракции запроса Создадим теперь абстракцию «Запрос/ответ», основанную на AMQP. Для этого в роли брокера будет использоваться RabbitMQ, но подойдет любой другой брокер, совместимый с AMQP. Начнем с абстракции запроса (модуль `amqpRequest.js`). Далее приводятся только фрагменты кода, соответствующие теме.

Первый из таких фрагментов создает очередь для хранения ответов:

```
channel.assertQueue('', {exclusive: true});
```

При создании очереди мы не определяем ее имя, то есть оно будет выбрано случайным образом без нашего участия. Кроме того, очередь является *эксклюзивной*, то есть она привязана к активному соединению AMQP и будет уничтожена при его закрытии. Здесь нет необходимости привязывать очередь к коммутатору, так как мы не нуждаемся в маршрутизации и распределении сообщений по нескольким очередям. Это означает, что сообщения должны доставляться непосредственно в очередь ответов.

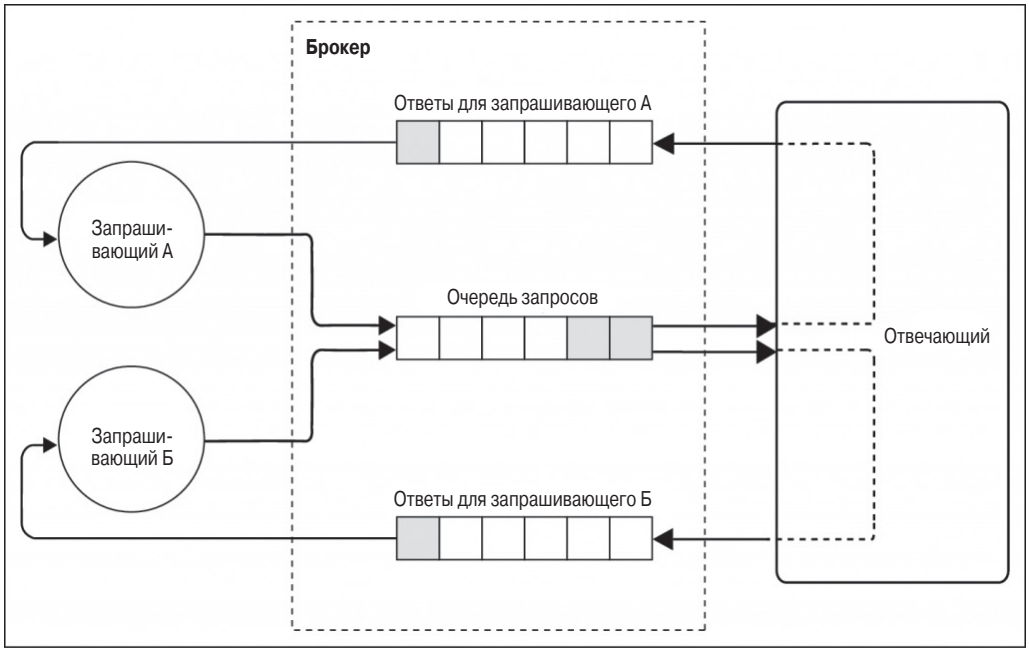

Рис. 11.18 ❖ Схема шаблона обратного адреса

Далее рассмотрим создание нового запроса:

```
classAMQPRequest {
  //...
  request(queue, message, callback) {
    const id = uuid.v4();
    this.idToCallbackMap[id] = callback;
    this.channel.sendToQueue(queue,new Buffer(JSON.stringify(message)),
      {correlationId: id, replyTo: this.replyQueue}
    );
  }
}
```

Метод `request()` принимает имя очереди запросов и сообщение для отправки. Так же, как в предыдущем разделе, мы должны сгенерировать идентификатор корреляции и связать его с функцией обратного вызова. В заключение выполняется отправка сообщения с установленными свойствами `correlationId` и `replyTo`.

Как видите, для отправки сообщения вместо `channel.publish()` используется `channel.sentToQueue()`. Это связано с тем, что нужно реализовать не распространение вида публикация/подписка с помощью коммутаторов, а более простую доставку непосредственно в очередь назначения.

В AMQP можно определить набор свойств (или метаданные), передаваемый потребителю наряду с основным сообщением.

Последним важным фрагментом является прототип `amqpRequest`, принимающий ответы:

```
_listenForResponses() {
  return this.channel.consume(this.replyQueue, msg => {
    const correlationId = msg.properties.correlationId;
    const handler = this.idToCallbackMap[correlationId];
    if (handler) {
      handler(JSON.parse(msg.content.toString()));
    }
  }, {noAck: true});
}
```

Этот код реализует прием сообщений из очереди, созданной специально для ответов. Из каждого принятого сообщения извлекается идентификатор для сопоставления с идентификаторами, хранящимися в списке обработчиков, ожидающих ответа. После получения соответствующего обработчика остается только вызвать его, передав ответ.

Реализация абстракции ответа Теперь, когда модуль `amqpRequest` готов, можно переходить к реализации абстракции ответа в новом модуле `amqpReply.js`.

Здесь нужно создать очередь для входящих запросов. Для этой цели можно использовать простую надежную очередь. Этот фрагмент кода здесь не приводится, поскольку он содержит только типовой код, связанный с использованием AMQP. Интерес представляет лишь часть кода, которая обрабатывает запрос и посылает его в нужную очередь:

```
class AMQPReply {
  //...

  handleRequest(handler) {
    return this.channel.consume(this.queue, msg => {
    const content = JSON.parse(msg.content.toString());
    handler(content, reply => {
      this.channel.sendToQueue(
        msg.properties.replyTo,
        new Buffer(JSON.stringify(reply)),
        {correlationId: msg.properties.correlationId}
      );
      this.channel.ack(msg);
      });
    });
  }
}
```

Для отправки ответа используется метод `channel.sendToQueue()`, публикующий сообщение в очереди, которая указана в свойстве `replyTo` сообщения (обратный адрес). Еще одной важной задачей нашего объекта `amqpReply` является установка значения `correlationId` в ответе, что позволит получателю сопоставить сообщение со списком ожидающих запросов.

Реализация запрашивающей и отвечающей сторон Теперь система готова к проверке, но сначала создадим запрашивающую и отвечающую стороны, чтобы продемонстрировать использование новой абстракции.

Начнем с модуля `replier.js`:

```
const Reply = require('./amqpReply');
const reply = Reply('requests_queue');
```

```
reply.initialize().then(() => {
  reply.handleRequest((req, cb) => {
    console.log('Request received', req);
    cb({sum: req.a + req.b});
  });
});
```

Как видите, вновь созданная абстракция позволяет скрыть все механизмы обработки идентификатора корреляции и обратного адреса; от нас требуется только инициализировать новый объект `reply`, указав имя очереди для получения запросов (`'requests_queue'`). Остальная часть кода довольно тривиальна; пример отвечающей стороны просто вычисляет сумму двух полученных чисел и отправляет результат, используя указанную функцию обратного вызова.

Реализуем пример запрашивающей стороны в файле `requestor.js`:

```
const req = require('./amqpRequest')();

req.initialize().then(() => {
  for (let i = 100; i> 0; i--) {
    sendRandomRequest();
  }
});

function sendRandomRequest() {
  const a = Math.round(Math.random() * 100);
  const b = Math.round(Math.random() * 100);
  req.request('requests_queue', {a: a, b: b},
    res => {
      console.log(`${a} + ${b} = ${res.sum}`);
    }
  );
}
```

Пример запрашивающей стороны отправляет в очередь `requests_queue` 100 запросов, сформированных случайным образом. В этом случае абстракция также отлично справляется с сокрытием всех деталей асинхронного шаблона «Запрос/ответ».

Для проверки работы системы просто запустим модуль `replier`, а вслед за ним модуль `requestor`:

```
node replier
node requestor
```

В результате запрашивающая сторона опубликует набор операций, который затем будет получен отвечающей стороной, которая, в свою очередь, вернет ответы.

Теперь можно попробовать провести другие эксперименты. Сразу после запуска отвечающая стороны создаст надежную очередь, а это значит, что если сейчас остановить ее, а затем снова запустить запрашивающую сторону, запросы не будут потеряны. Все сообщения будут храниться в очереди, пока снова не запустится отвечающая сторона!

Еще одной удобной особенностью AMQP является поддержка масштабируемости отвечающей стороной «из коробки». Для проверки можно попробовать запустить несколько экземпляров отвечающей стороны и понаблюдать, как распределяется нагрузка между ними. Такая поддержка объясняется тем, что при каждом запуске от-

вечающей стороны она подключается к одной и той же надежной очереди, что дает брокеру возможность распределять нагрузку между всеми потребителями (шаблон конкурирующих потребителей). Здорово!

Библиотека ØMQ содержит пару сокетов, специально предназначенных для реализации шаблона «Запрос/ответ» (REQ/REP), но они являются синхронными (не поддерживают обработку параллельных запросов/ответов). Более сложные шаблоны «Запрос/ответ» реализуются другими, более сложными методами. Подробнее об этом рассказывается в официальном руководстве на странице http://zguide.zeromq.org/page:all#advanced-request-reply.

Итоги

Эта глава подошла к концу. Здесь мы рассмотрели наиболее важные шаблоны обмена сообщениями и интеграции, а также определили их роль в разработке распределенных систем. Познакомились с тремя основными типами шаблонов обмена сообщениями: «Публикация/подписка», «Конвейер» и «Запрос/ответ», — и увидели, как они реализуются с использованием приема прямого обмена и брокера сообщений. Все они имеют свои плюсы и минусы, и, как мы увидели, используя AMQP и полноценного брокера сообщений, можно с минимальными усилиями реализовать надежные и масштабируемые приложения, но ценой добавления еще одной системы для управления и масштабирования. Также мы увидели, как библиотека ØMQ позволяет конструировать распределенные системы и дает полный контроль над всеми аспектами их архитектуры, обеспечивает тонкую настройку их свойств под конкретные требования.

Эта глава завершает книгу; теперь в вашем распоряжении имеется множество шаблонов и технологий, которые вы сможете применять в своих проектах. Кроме того, у вас теперь должно быть глубокое понимание, как работает платформа Node.js, каковы ее сильные и слабые стороны. На протяжении всей книги мы использовали шанс поработать со множеством пакетов и решений, созданных многими талантливыми разработчиками. В конце концов, это самый ценный аспект платформы Node.js: люди, сообщество, где каждый играет свою роль, внося посильный вклад.

Я надеюсь, вам понравился наш небольшой вклад.

Предметный указатель

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Х

Ц

Ш

Э

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: books@alians-kniga.ru.

Марио Каскиаро, Лучано Маммино

Шаблоны проектирования Node.js

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 24,75. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф