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.

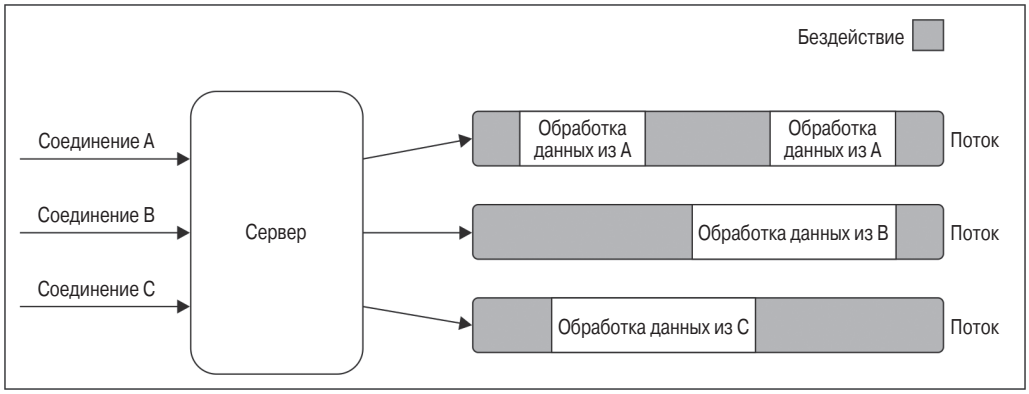

Рис. 1.1 ❖ Параллельная обработка нескольких подключений

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.

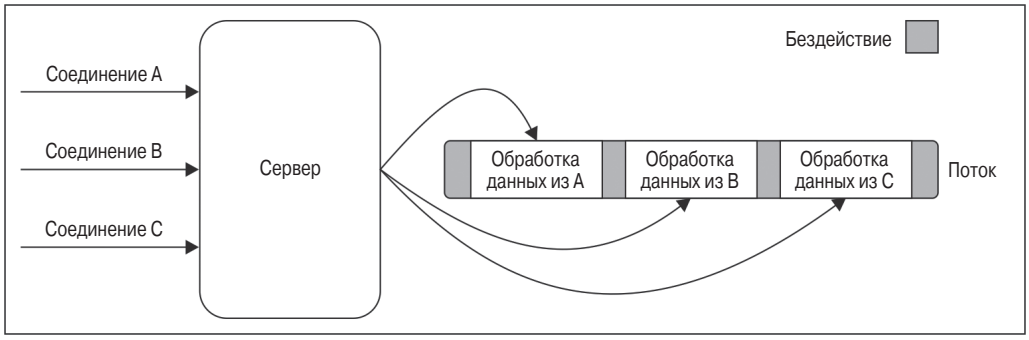

Рис. 1.2 ❖ Схема обработки нескольких соединений при помощи синхронного демультиплексирования

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.

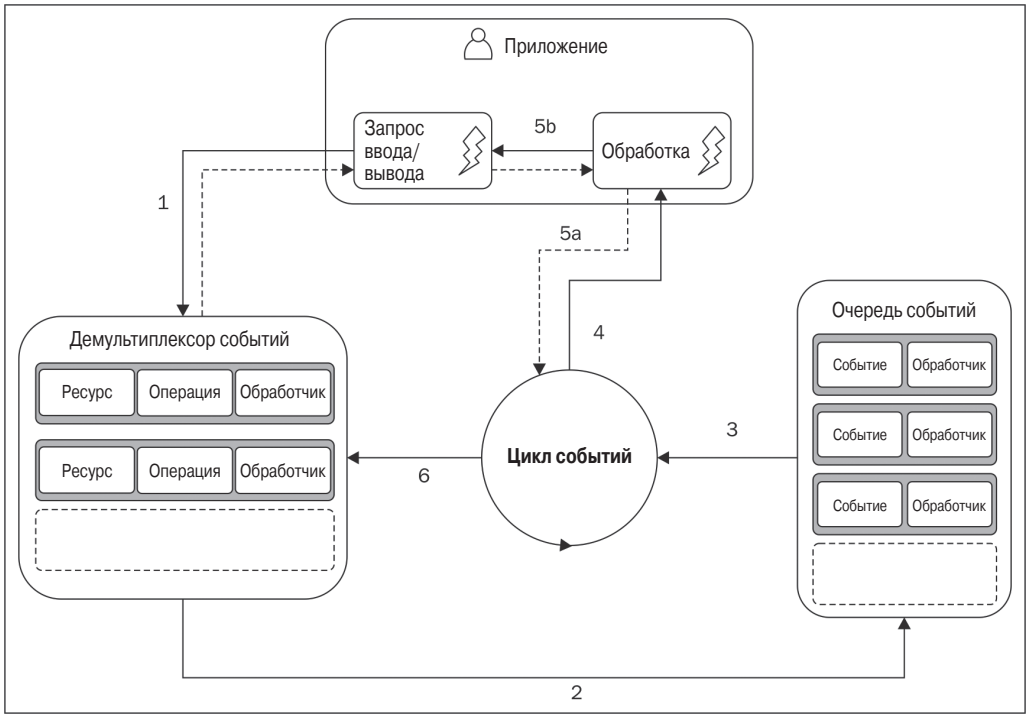

Рис. 1.3 ❖ Структура шаблона Reactor

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.

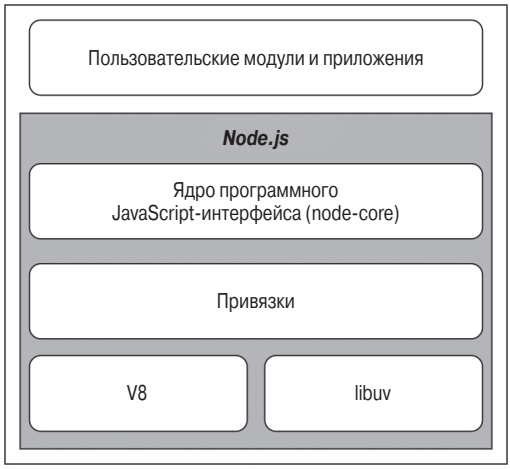

Рис. 1.4 ❖ Архитектура платформы Node.js

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.