7 Оптимизация функционального кода

В этой главе...

- Выявление мест, где функциональный код более производителен
- Исследование внутреннего механизма выполнения функций в JavaScript
- Последствия вложения контекстов функций и рекурсии
- Оптимизация выполнения функций посредством отложенного вычисления
- Ускорение выполнения программ благодаря запоминанию
- Сворачивание вызовов хвостово-рекурсивных функций

> В 97 из 100 случаев мы должны забывать о небольшом повышении эффективности, ведь весь корень зла — в преждевременной оптимизации. Но очень важно не упускать такую возможность в остальных 3 случаях.
>
> Из книги *Искусство программирования* (The Art of Computer Programming) *Дональда Кнута (Donald Knuth)*

Как рекомендуют знающие люди, оптимизировать код следует в последнюю очередь. В предыдущих главах пояснялось, как писать и тестировать функциональный код, а теперь, ближе к завершению этого увлекательного экскурса в область ФП, мы рассмотрим способы оптимизации подобного кода. Ни одна парадигма программирования не является идеальной и имеет свои компромиссы, например, между производительностью и абстракцией. Функциональное программирование предоставляет уровни абстракции для достижения высо-

кой степени текучести и декларативности прикладного кода. В связи с необходимостью применять внутренний карринг, рекурсию, заключение в оболочку монад и композицию для решения даже самых простых задач может возникнуть следующий вопрос: оказывается ли функциональный код таким же производительным, как и императивный?

В самом деле, сокращение времени выполнения даже самых современных веб-приложений, кроме игровых, мало что дает. Быстродействие современных компьютеров и технологическое качество компиляторов возросло настолько, что неизменно гарантирует высокую производительность правильно написанного кода. В этом отношении функциональный код не менее производителен, чем императивный, хотя его особенности проявляются иначе.

Было бы неразумно перейти к применению на практике новой парадигмы программирования, не уяснив последствия этого шага для среды, в которой должен выполняться написанный код. Именно поэтому в этой главе поясняются особенности функционального кода на JavaScript, которые следует иметь в виду, особенно при обработке больших массивов данных. По ходу изложения материала этой главы будут упоминаться такие основные языковые средства JavaScript, как замыкания, поэтому прочитайте главу 2, "Сценарий высшего порядка", чтобы ясно понимать, о чем здесь идет речь. В этой главе обсуждаются также интересные методики оптимизации, в том числе отложенное вычисление, запоминание и оптимизация вызовов рекурсивных функций.

Функциональное программирование не ускоряет время вычисления отдельных функций, а скорее служит стратегией для исключения дублирующихся вызовов функций и откладывания вызовов кода до тех пор, пока в этом не возникнет абсолютная необходимость. Это дает возможность ускорить работу всего приложения. В языках исключительно функционального программирования подобные виды оптимизации встроены в саму платформу и могут использоваться вообще без вмешательства со стороны программиста. А в JavaScript эти виды оптимизации приходится внедрять вручную через специальный код или функциональные библиотеки. Но прежде чем обсуждать этот вопрос подробно, рассмотрим вкратце трудности, возникающие на пути применения JavaScript в ФП, а также веские основания для оптимизации функционального кода.

7.1. Внутренний механизм выполнения функций

Поскольку в основу ФП положено вычисление функций для всего, что делается в программе, поэтому приступая к изучению вопросов производительности и оптимизации, очень важно уяснить, что же происходит при вызове любой функции в JavaScript. При вызове функции в стеке ее контекста интерпретатор JavaScript создает специальную запись (или фрейм).

Стек контекста функции является компонентом принятой в JavaScript модели программирования и отвечает за организацию выполнения функции и переменные, которые она замыкает (за дополнительными разъяснениями об-

ращайтесь к разделу 2.4). Такой стек всегда начинается с фрейма глобального контекста выполнения, где содержатся все глобальные переменные (рис. 7.1).

Примечание

Стек является основной структурой, содержащей объекты, которые можно размещать и извлекать из стека по принципу "последним пришел, первым обслужен" (LIFO). В качестве аналогии можно привести стопку тарелок, т.е. все операции в стеке выполняются на его вершине.

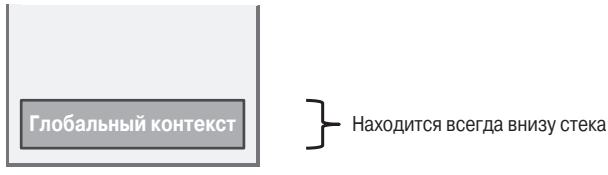


Рис. 7.1. Стек контекста выполнения в JavaScript после его инициализации. В глобальном контексте можно отслеживать немало переменных и функций, хотя это зависит и от количества сценариев, загружаемых на странице

Фрейм глобального контекста выполнения всегда располагается внизу стека. Каждый фрейм контекста функции занимает определенный объем памяти в зависимости от количества локальных переменных, которые в нем находятся. А в отсутствие каких-либо локальных переменных пустой фрейм занимает около 48 байт. Для хранения каждой локальной переменной или параметра (числового и логического) требуется около 8 байт. Естественно, что чем больше переменных объявлено в теле функции, тем крупнее становится фрейм стека. В каждом фрейме содержится приблизительно следующая информация в структуре данных[1]:

```
executionContextData = {
    scopeChain,
    variableObject,
    this
}
```

В этом контексте имеется доступ к свойству `variableObject` данной функции, а также к свойству `variableObject` из родительского контекста выполнения

Содержит аргументы, внутренние переменные и объявления данной функции

Ссылка на функциональный объект (напомним, что каждая функция в JavaScript считается объектом)

Из этой структуры данных можно сделать ряд важных выводов. Во-первых, именно свойство `variableObject` в основном определяет размер фрейма стека, поскольку в нем содержатся ссылки на аргументы функции, конкретный объект `arguments` в виде массива (см. главу 2, "Сценарий высшего порядка"), а также все локальные переменные и функции. Во-вторых, именно цепочка

[1] Эти сведения взяты из отличной публикации "What Is the Execution Context & Stack in JavaScript?" (Назначение контекста выполнения и его стека в JavaScript) Дэвида Шериффа (David Shariff) от 19 июня 2012 г. в блоге по адресу `http://davidshariff.com/blog/hat-is-the-execution-context-in-javascript/`.

областей видимости функций связывает контекст данной функции с его родительским контекстом выполнения, как поясняется ниже. Прямо или косвенно, но цепочка областей видимости каждой функции в конечном счете связывает ее с глобальным контекстом.

Примечание

Цепочка областей видимости отдельной функции отличается от цепочки прототипов объекта в JavaScript. И хотя обе цепочки ведут себя сходным образом, цепочка прототипов обозначает ссылку, устанавливаемую при наследовании объектов через свойство **`prototype`**. А цепочка областей видимости обозначает, в частности, порядок доступа внутренней функции к замыканию ее внешней функции.

Поведение стека определяется следующими важными правилами.

- JavaScript — однопоточный язык, а следовательно, в нем поддерживается синхронное выполнение кода.
- Имеется один и только один глобальный контекст, общий для всех контекстов функций.
- Допускается неограниченное количество контекстов функций (различные браузеры могут накладывать свои ограничения на выполнение клиентского кода).
- При каждом вызове функции создается новый контекст ее выполнения, даже если она вызывает сама себя рекурсивно.

Как известно, в функциональном программировании функции применяются в максимальной степени. Это побуждает разбивать решаемые задачи на как можно больше функций, а также подвергать их каррингу для дополнительного удобства и повторного использования. Но применение большого количества каррированных функций оказывает влияние на размер стека контекста функций.

7.1.1. Карринг и стек контекста функций

Я являюсь большим поклонником карринга, и мне бы очень хотелось, чтобы все вычисления функций выполнялись в JavaScript с автоматическим каррингом. Но такой дополнительный уровень абстракции может привести к некоторым контекстным издержкам по сравнению с обычным выполнением функции. Чтобы это положение стало понятнее, выясним, что же в действительности происходит внутри механизма вызова каррированной функции в JavaScript.

Как упоминалось в главе 4, “На пути к повторно используемому, модульному коду”, когда производится карринг функции, интерпретатор JavaScript заменяет ее одноразовый вызов со всеми параметрами на несколько последовательных внутренних вызовов. Иными словами, в результате карринга функции `logger()` из главы 4:

```
const logger = function (appender, layout, name, level, message)
```

получается следующая вложенная структура:

```
const logger =
    function (appender) {
         return function (layout) {
             return function (name) {
                 return function (level) {
                     return function (message) {
                ...
```

В такой вложенной структуре стек используется более интенсивно, чем при прямом вызове. Поясним сначала порядок выполнения некаррированной версии функции `logger()`. Вследствие синхронного характера выполнения кода на JavaScript вызов функции `logger()` приводит к приостановке работы глобального контекста и подготовке к запуску этой функции. При этом создается новый активный контекст, а в нем указывается ссылка на глобальный контекст для доступа к переменным (рис. 7.2).

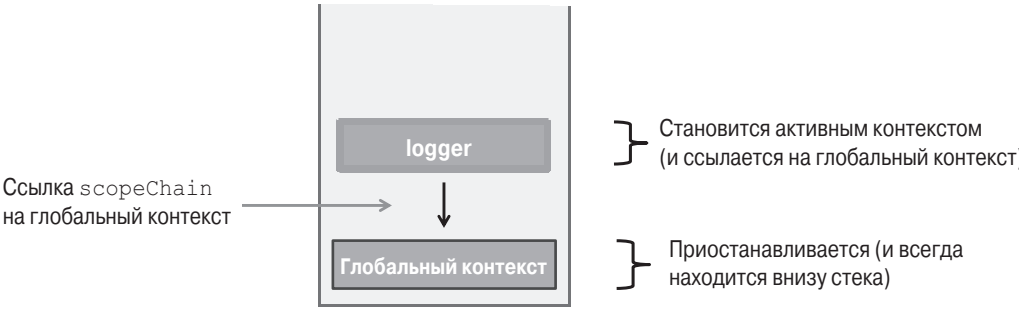


Рис. 7.2. При вызове любой функции (в данном случае — **`logger()`**) однопоточный интерпретатор JavaScript приостанавливает работу текущего глобального контекста и активизирует контекст выполнения новой функции. В этот момент между глобальным контекстом и контекстом функции устанавливается связь, доступная по ссылке **`scopeChain`**. А после возврата из функции **`logger()`** контекст ее выполнения удаляется из стека и восстанавливается глобальный контекст

Во внутреннем механизме выполнения самой функции `logger()` вызываются остальные операции из библиотеки Log4js, в результате чего создаются новые контексты функций, размещаемые в стеке (подробнее о библиотеке Log4js см. в приложении). Благодаря действию замыканий в JavaScript контексты функций, возникающие в результате внутренних вызовов функций, накапливаются в стеке один над другим и занимают выделенную для них память. Ссылка на старый контекст указывается в элементе `scopeChain` (рис. 7.3).

И, наконец, по завершении кода вложенных операций из библиотеки Log4js текущий контекст удаляется из стека, и за ним активизируется контекст выполнения функции `logger()`. После завершения работы и этой функции интерпретатор JavaScript возвращает все в исходное состояние, в котором действует только глобальный контекст (см. рис. 7.1). Вот, собственно, и вся тайна, скрывающаяся за действием замыканий в JavaScript.

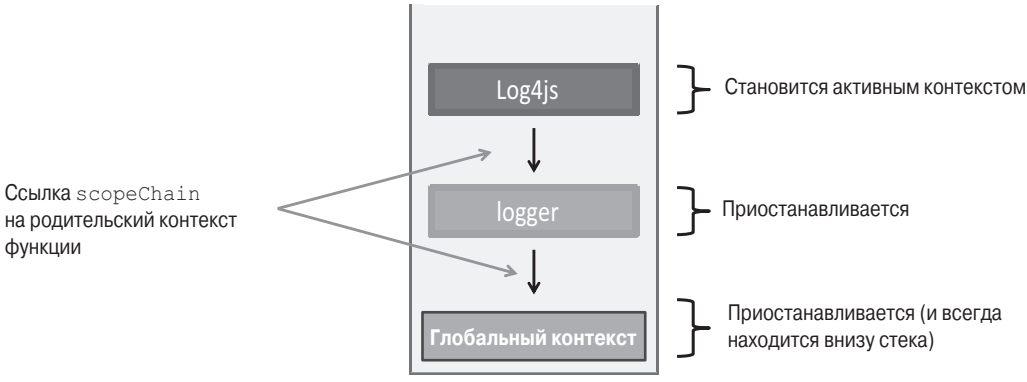


Рис. 7.3. Расширение контекста функции при выполнении вложенных функций. При вызове каждой из этих функций создается новый фрейм в стеке, и поэтому размер стека растет пропорционально уровню вложенности функций. В процессе карринга и рекурсии используются вызовы вложенных функций

Несмотря на всю эффективность такого подхода, глубоко вложенные функции могут потребовать немало оперативной памяти. В главе 8, "Обработка асинхронных событий и данных", будет представлена функциональная библиотека RxJS, предназначенная для работы с асинхронным кодом. Весь исходный код самой последней ее версии RxJS 5 полностью переписан по сравнению с предыдущей версией с главным акцентом на производительность и сокращение количества замыканий.

А теперь рассмотрим рис. 7.4, на котором показано выполнение каррированной версии функции `logger()`.

На первый взгляд карринг всех функций может показаться неплохой идеей, но злоупотребление им способно привести к тому, что программы будут занимать большие участки пространства стека и выполняться значительно медленнее. Чтобы убедиться в этом, запустите следующую простую программу оценки производительности:

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

const c_add = curry2(add);

const input = _.range(80000);

addAll(input, add); // ->511993600000000
addAll(input, c_add); // -> браузер останавливается

function addAll(arr, fn) {
    let result= 0;
    for(let i = 0; i < arr.length; i++) {
        for(let j = 0; j < arr.length; j++) {
```

```
            result += fn(arr[i], arr[j]);
        }
    }
    return result;
}
```

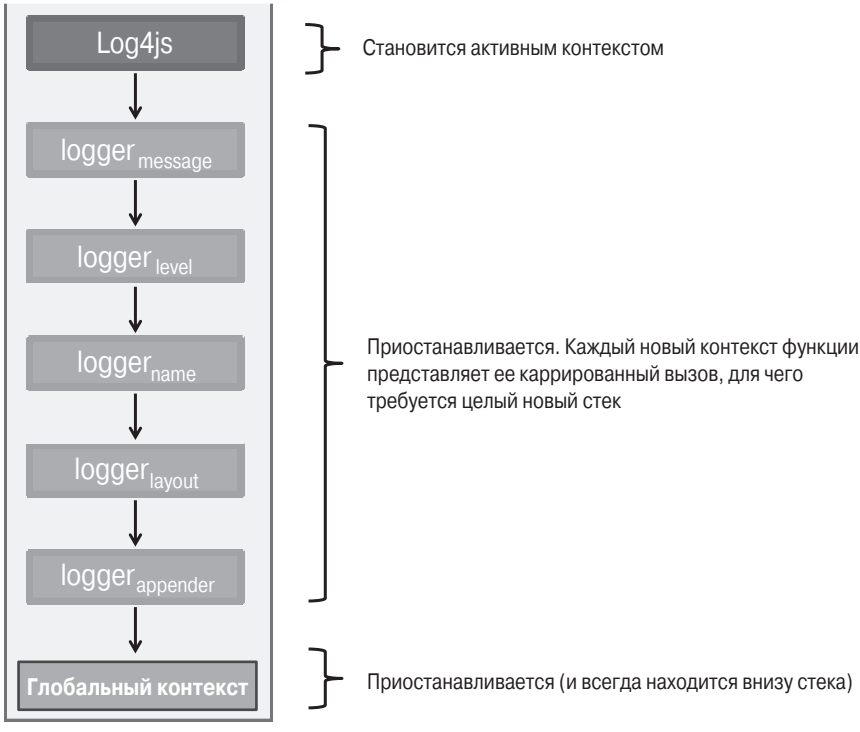


Рис. 7.4. При карринге каждый параметр каррированной функции преобразуется внутренним образом во вложенный вызов. Такое удобство в предоставлении параметров по очереди достигается за счет дополнительных фреймов, занимающих место в стеке

В этой программе создается массив из 80000 чисел и сравнивается некаррированная версия функции с каррированной. Правильный результат возвращается из некаррированной версии данной функции через несколько секунд, тогда как выполнение ее каррированной версии приводит к зависанию браузера. Без сомнения, за карринг приходится платить определенную цену, хотя обработка столь крупных массивов данных в большинстве приложений JavaScript маловероятна.

И это не единственный случай, когда размер стека может заметно вырасти. Неэффективные и неверные рекурсивные решения также приводят к переполнению стека.

7.1.2. Трудности, связанные с рекурсивным кодом

Новые контексты функций создаются даже в тех случаях, когда функции вызывают сами себя. Неверный рекурсивный вызов, когда основная ветка программы так и не достигается, может легко привести к переполнению стека. Правда, рекурсия относится к тем приемам, которые либо срабатывают, либо не срабатывают, и в последнем случае рекурсия никак не дает о себе знать. Если вам когда-нибудь приходилось получать сообщения об ошибке вроде "`Range Error: Maximum Call Stack Exceeded`" (Ошибка при проверке границ: превышен максимальный размер стека вызовов) или "`Too much recursion`" (Слишком много рекурсии), то вы знаете, что здесь имеется в виду. С помощью следующего простого сценария можно выполнить эталонное тестирование браузера, чтобы получить приблизительный размер стека вызовов функций:

```
function increment(i) {
    console.log(i);
    increment(++i);
}
increment(1);
```

В различных браузерах выявление ошибок переполнения стека реализовано по-разному. Так, на моем компьютере браузер Chrome генерирует исключение приблизительно через 17500 итераций в данном сценарии, тогда как браузер Firefox — выполняет только 7600 итераций. Но не следует полагаться на эти цифры как на верхние границы стека вызовов, приступая к написанию собственных рекурсивных функций! Эти граничные цифры лишь показывают те пределы, которые не следует превышать. При написании собственных рекурсивных функций вы должны придерживаться намного более низких значений счетчика рекурсии. Если же в составленной вами программе счетчик рекурсий постоянно растет, либо имеет очень высокие значения, то, вероятнее всего, в программу вкралась ошибка.

Если же вам придется обрабатывать необычно крупный массив данных, используя рекурсию, имейте в виду, что размер стека будет расти пропорционально размеру массива данных. В качестве примера рассмотрим следующую рекурсивную функцию для обнаружения самой длинной символьной строки в массиве:

```
function longest(str, arr) {
    if(R.isEmpty(arr)) {
        return str;
    else {
        let currentStr = R.head(arr).length >= str.length
                         ? R.head(arr): str;
        return longest(currentStr, R.tail(arr));
    }
}
```

Вызвать функцию `longest()` для списка всех 192 стран в мире несложно, но поиск с ее помощью города с самым длинным названием среди 2,5 млн городов

может привести к неудачному завершению данной программы, как показано на рис. 7.5. (На самом деле этот конкретный алгоритм не приведет к сбоям при обработке крупных массивов в стандарте ES6 языка JavaScript, как поясняется далее.)

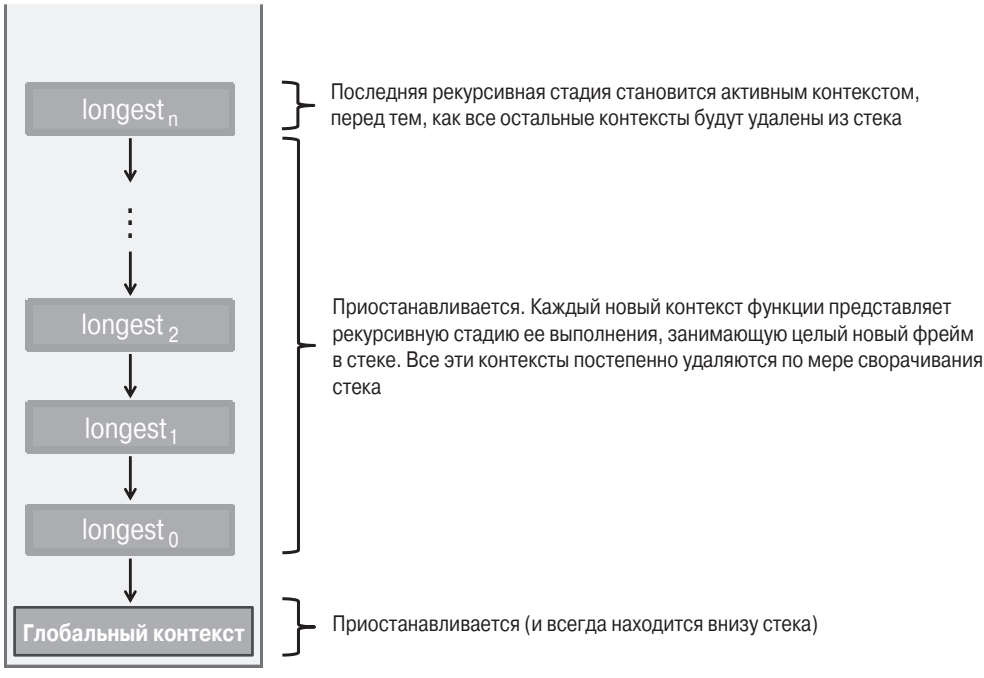


Рис. 7.5. При поиске самой длинной символьной строки в массиве из *n* элементов функции `longest()` приходится наращивать размер стека пропорционально размерности входных данных, вводя *n* фреймов в стек контекста ее выполнения

В качестве альтернативы такому способу обхода списков, особенно с необычайно крупными массивами, применяют функции высшего порядка, упоминавшиеся в главе 3, “Меньше структур данных и больше операций”, и реализующие такие операции, как `map`, `filter` и `reduce`. В этом случае не формируются вызовы вложенных функций, а размер стека остается постоянным на каждой итерации.

Несмотря на то что карринг и рекурсия приводят к тому, что функции занимают больше места в оперативной памяти, чем их императивные аналоги, следует также иметь в виду те преимущества, которые приносит карринг в отношении гибкости и повторного использования кода, а также корректность, присущую рекурсивным решениям. Эти преимущества стоят требований, предъявляемых к дополнительному расходу оперативной памяти.

Впрочем, нет худа без добра. Ведь функциональное программирование предоставляет такие возможности для оптимизации прикладного кода, которые отсутствуют у других парадигм. Размещение большого количества вызовов

функций в стеке может привести к увеличению объема оперативной памяти, потребляемой программой. Так почему бы не исключить полностью вызовы некоторых функций?

7.2. Отсрочка выполнения с помощью отложенного вычисления

Исключив ненужные вызовы функций и крупные входные данные, когда достаточно их подмножества, можно добиться существенного улучшения производительности программы. В таких языках ФП, как Haskell, в каждое функциональное выражение встроено *отложенное вычисление* функций. Имеются разные алгоритмы отложенного вычисления функций, но все они преследуют одну и ту же цель: задержать выполнение функции как можно дольше или хотя бы до вызова зависящего от нее выражения.

Но для вычисления функций в JavaScript применяется более широко распространенная стратегия энергичного вычисления. В этом случае выражение вычисляется, как только оно будет привязано к переменной, независимо от потребности в результате его выполнения. Такой способ вычисления иногда еще называют *энергичным* (*или жадным*). В качестве примера рассмотрим извлечение подмножества элементов из массива, приведенное на рис. 7.6.

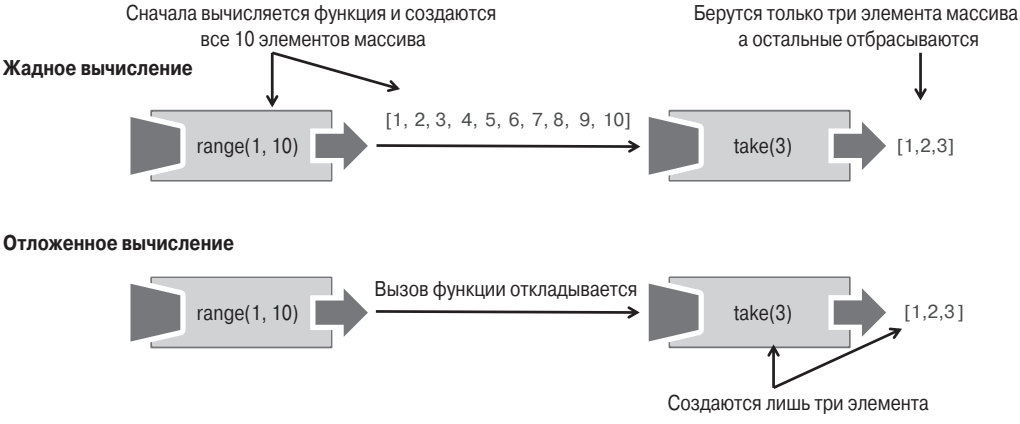


Рис. 7.6. Композиция функции **`range()`**, возвращающей список чисел от начала и до конца, с функцией **`take()`**, читающей первые *n* элементов из этого списка. При энергичном вычислении функция **`range()`** выполняется полностью, передавая результат функции **`take()`**. А при отложенном вычислении функция **`range()`** не запускается до тех пор, пока не будет вызвана зависящая от нее операция **`take`**

Как видите, в алгоритме энергичного вычисления сначала выполняется функция `range()`, а затем ее результат передается функции `take()`, которой требуется лишь часть этого результата, тогда как остальная его часть просто отбрасывается. Только представьте, насколько это было бы расточительно, если бы пришлось формировать крупный массив чисел. А при отложенном

вычислении выполнение функции `range()` откладывается до тех пор, пока ее результат не потребуется зависящей от нее операции, реализуемой функцией `take()`. Если заранее известно назначение функции `range()`, то из нее можно получить лишь нужное количество элементов. Рассмотрим еще один пример с использованием монады типа `Maybe`:

```
Maybe.of(student).getOrElse(createNeStudent());
```

На первый взгляд, применение монады типа `Maybe` может навести на мысль, что данное выражение ведет себя следующим образом:

```
if(!student) {
    return createNeStudent();
}
else {
    return student;
}
```

Но благодаря принятому в JavaScript алгоритму энергичного вычисления функция `createNeStudent()` будет выполнена в данном коде независимо от того, окажется ли пустым (`null`) объект `student`. А при отложенном вычислении данное выражение повело бы себя таким же образом, как и приведенный ранее фрагмент кода, вообще не вызывая функцию `createNeStudent()`, если объект `student` окажется недостоверным. Так как же воспользоваться преимуществами отложенного вычисления? В этом разделе рассматриваются следующие рекомендации относительно его применения.

- Исключение ненужных вычислений.
- Применение сокращенного слияния в функциональных библиотеках.

7.2.1. Исключение вычислений с помощью комбинатора чередования функций

Нет ничего удивительного в том, чтобы предпринять определенные действия для эмуляции отложенного вычисления и в то же время извлечь ряд преимуществ из языков исключительно функционального программирования. В простейшем случае ненужных вычислений можно избежать, передав функции по ссылке (или по имени) и вызвав ту или иную функцию по заданному условию. В главе 4 был представлен комбинатор функций `alt()`, в котором выгодно используется логическая операция `||` (ИЛИ) для вычисления в первую очередь функции `func1` и вызова функции `func2` лишь в том случае, если функция `func1` возвратит значение `false`, `null` или `undefined`, как демонстрируется в следующем примере:

```
const alt = R.curry((func1, func2, val) => func1(val) || func2(val));

const shoStudent = R.compose(append('#student-info'),
      alt(findStudent, createNeStudent));

shoStudent('444-44-4444');
```

Функции не требуется вызывать преждевременно, поскольку они передаются комбинатору по ссылке для согласования их вызовов

Комбинатор берет на себя обязанность согласовывать вызовы функций, и поэтому приведенный выше пример кода равнозначен следующей условной логике, написанной в императивном стиле:

```
var student = findStudent('444-44-4444');
if(student !== null) {
    append('#student-info', student);
}
else {
    append('#student-info', createNeStudent('444-44-4444'));
}
```

Это очень простой пример исключения ненужных вычислений функций с намного меньшим дублированием. Более эффективная стратегия будет продемонстрирована далее в этой главе, когда речь пойдет о *запоминании (memoization)*. А с другой стороны, определение всей программы еще до ее выполнения позволяет осуществить в функциональных библиотеках оптимизацию, называемую *сокращенным слиянием (shortcut fusion)*.

7.2.2. Использование преимуществ сокращенного слияния

В главе 3, "Меньше структур данных и больше операций", была представлена функция, реализующая операцию `_.chain` в библиотеке Lodash. Данная функция позволяет заключить в оболочку всю последовательность функций, инициировав ее выполнение через конечную функцию `value()`. Это дает возможность не только отделить описание программы от ее выполнения, но и выявить средствами Lodash места оптимизации кода, объединив, например, выполнение некоторых функций для более эффективного использования памяти. Так, в следующем примере получается список стран, отсортированный по численности населения:

```
_.chain([p1, p2, p3, p4, p5, p6, p7])
    .filter(isValid)
    .map(_.property('address.country')).reduce(gatherStats, {})
    .values()
    .sortBy('count')
    .reverse()
    .first()
    .value()
```

Такой декларативный стиль программирования означает, что беспокоиться следует не о том, как работают функции, а том, что для этого нужно сделать, заранее определив требующиеся действия. Иногда это дает библиотеке Lodash возможность оптимизировать внутренним образом выполнение программы, используя сокращенное слияние. Это вид оптимизации на уровне функции, допускающий слияние процесса выполнения некоторых функций в одно уплотненное количество внутренних структур данных, предназначенных для вычисления промежуточных результатов. Благодаря созданию меньшего количества

структур данных сокращается чрезмерное расходование оперативной памяти, требующейся для обработки крупных коллекций.

Такое слияние оказывается возможным благодаря принятым в функциональном программировании строгим правилам соблюдения ссылочной прозрачности, которая придает ему особую математическую или алгебраическую точность. Например, вызов функции `compose(map(f), map(g))` можно заменить выражением `map(compose(f, g))`, не меняя ее назначение. Аналогично вызов функции `compose(filter(p1), filter(p2))` равнозначен выражению `filter((x) => p1(x) && p2(x))`. Именно это и происходит в паре операций `filter` и `map`, с которых начинается цепочка в предыдущем примере. Опять же манипулирование последовательностью операций подобным математическим способом оказывается возможным лишь благодаря чистым функциям. Это наглядно демонстрирует еще один пример из листинга 7.1.

Листинг 7.1. Реализация отложенного вычисления и сокращенного слияния средствами библиотеки Lodash

```
const square = (x) => Math.pow(x, 2);
const isEven = (x) => x % 2 === 0;
const numbers = _.range(200);

const result =
    _.chain(numbers)
     .map(square)
     .filter(isEven)
     .take(3)
     .value(); //-> [0,4,16]

result.length; // -> 5
```

Сформировать массив из чисел от 1 до 200

Обработать только три первых числа, которые удовлетворяют критериям, накладываемых операциями `filter` или `map`

В исходном коде из листинга 7.1 выполнены два вида оптимизации. Во-первых, при вызове функции `take(3)` библиотеке Lodash предписывается принять во внимание лишь три первых значения, которые удовлетворяют критериям преобразования и фильтрации. Это позволяет не расходовать напрасно драгоценные циклы ЦП на остальные 195 элементов массива. И во-вторых, сокращенное слияние позволяет соединить последующие вызовы функций `map()` и `filter()` в выражение `compose(filter(isEven), map(square))`. В этом нетрудно убедиться, снабдив функции `square()` и `isEven()` операторами трассировки в системный журнал (для целей протоколирования в данной композиции, по существу, применяется комбинатор `tap()` из библиотеки Ramda), как показано ниже.

```
square = R.compose(R.tap(() => trace('Mapping')), square);
isEven= R.compose(R.tap(() => trace('then filtering')), isEven);
```

В итоге на консоли появится следующая пара сообщений, повторяемых пять раз:

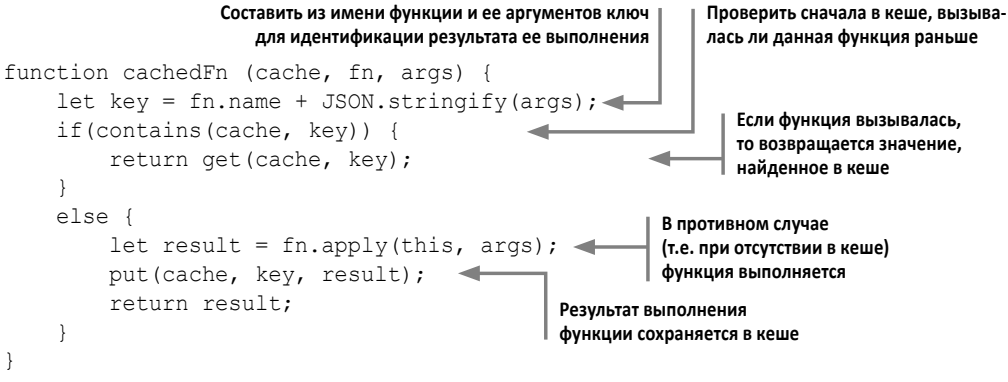
```
Mapping
then filtering
```

чем подтверждается слияние операций `map` и `filter`. Применение функциональных библиотек не только упрощает написание тестов, но и улучшает время выполнения прикладного кода. Сокращенное слияние приносит выгоду и в других функциях из библиотеки Lodash, реализующих следующие операции: `_.drop`, `_.dropRight`, `_.dropRightWhile`, `_.dropWhile`, `_.first`, `_.initial`, `_.last`, `_.pluck`, `_.reject`, `_.rest`, `_.reverse`, `_.slice`, `_.takeRight`, `_.takeRightWhile`, `_.takeWhile` и `_.here`.

Аналогично для исключения вычислений до тех пор, пока они не потребуются, имеется еще одно эффективное средство оптимизации функциональных программ, называемое *запоминанием (memoization)*.

7.3. Реализация стратегии вызовов по требованию

Чтобы ускорить выполнение приложений, можно, в частности, исключить повторное вычисление значений, особенно если это вычисление обходится дорого. В традиционных объектно-ориентированных системах это достигается организацией кеша на уровне посредника, который проверяется перед вызовом функции. После возврата из функции результату ее выполнения присваивается уникальный ключ для ссылки на него, и полученная в итоге пара "ключ–значение" сохраняется в кеше. В качестве *кеша* может служить промежуточное хранилище или оперативная память, запрашиваемая перед выполнением затратной операции. В веб-приложениях кеш применяется для временного хранения изображений, документов, скомпилированного кода, HTML-страниц, результатов запроса и т.д. В качестве примера рассмотрим следующий фрагмент кода, где реализуется уровень кеширования произвольной функции:




```
function cachedFn (cache, fn, args) {
    let key = fn.name + JSON.stringify(args);
    if(contains(cache, key)) {
        return get(cache, key);
    }
    else {
        let result = fn.apply(this, args);
        put(cache, key, result);
        return result;
    }
}
```

Функцией `cachedFn()`, определенной в приведенном выше коде, можно воспользоваться, чтобы заключить в ее оболочку выполнение функции `findStudent()`, как показано в следующем примере:

```
var cache = {};
cachedFn(cache, findStudent, '444-44-4444');
cachedFn(cache, findStudent, '444-44-4444');
```

При первом вызове результат отсутствует в кеше, и поэтому функция `findStudent()` выполняется

При втором вызове вычисленное значение извлекается непосредственно из кеша

Функция `cachedFn()` действует в качестве посредника между вызовом функции и ее результатом, с целью исключить ненужные повторные вызовы. Но написать код, где такая функция служила бы оболочкой для вызова каждой функции, было бы затруднительно, а сам код оказался бы неудобочитаемым. Хуже того, данная функция вызывает побочный эффект, поскольку она зависит от глобального объекта общего кеша. Поэтому требуется универсальное решение, позволяющее выгодно воспользоваться преимуществами кеширования, сохранив независимость прикладного кода и тестов от данного механизма. В языках ФП такой механизм называется *запоминанием*.

7.3.1. Общее представление о запоминании

Алгоритм кеширования, положенный в основу запоминания, реализуется аналогично приведенному выше коду, поэтому аргументы функции используются в нем для формирования уникального ключа, по которому сохраняется результат выполнения функции. Таким образом, при последующих вызовах функции с теми же самыми аргументами сохраненный результат может быть возвращен немедленно. Соотнесение результата выполнения функции с входными данными, иными словами, — сравнение вычисленного функцией значения со входным значением, достигается благодаря определенному принципу ФП. Нетрудно догадаться, что этим принципом является ссылочная прозрачность. Прежде всего рассмотрим преимущества запоминания на примере простого вызова функции.

7.3.2. Запоминание функций, требующих интенсивных вычислений

В одних языках исключительно функционального программирования алгоритм запоминания реализуется автоматически, а в других языках вроде JavaScript и Python программирующим предоставляется возможность выбрать момент для запоминания функции. Естественно, что в функциях, требующих интенсивных вычислений, можно выгодно воспользоваться чередованием уровня кеширования. Рассмотрим в качестве примера вычисление функции `rot13()`, где символьные строки кодируются по алгоритму ROT13 (путем циклического сдвига 26 букв английского алфавита на 13 позиций в коде ASCII). И хотя это слабый для шифрования алгоритм, он применяется на практике в веб-приложениях для сокрытия решений загадок и кодов скидок, исключения оскорбительных материалов и т.д.:

```
var discountCode = 'functional_js_50_off';

rot13(discountCode); // -> shapgvbany_f_50_bss
```

Ниже показано, каким образом реализуется алгоритм ROT13.

```
var rot13 = s =>
    s.replace(/[a-zA-Z]/g, c =>
        String.fromCharCode((c <= 'Z' ? 90 : 122)
            >= (c = c.charCodeAt(0) + 13) ? c : c - 26));
            (c = c.charCodeAt(0) + 13) ? c : c - 26);
    });
};
```

Понимание этого алгоритма необязательно для обсуждения данного вопроса. Тем не менее важно знать, что для одной и той же входной символьной строки (со ссылочно прозрачной функцией) по этому алгоритму всегда вычисляется одно и то же сообщение. Это означает, что запоминание позволяет добиться необычайного повышения производительности. Но прежде чем рассматривать реализацию запоминания в исходном коде функции `memoize()`, отметим, что ее можно применять следующими способами.

- Вызывая метод для функционального объекта:

```
var rot13 = rot13.memoize();
```

- Заключая в оболочку определение функции, как было показано ранее:

```
var rot13 = (s =>
     s.replace(/[a-zA-Z]/g, c =>
        String.fromCharCode((c <= 'Z' ? 90 : 122)
           >= (c = c.charCodeAt(0) + 13) ? c : c - 26))).memoize();
```

Благодаря запоминанию можно ожидать, что при последующем вызове функции с одними и теми же входными данными будет инициирована проверка внутреннего кеша и немедленный возврат значения. Чтобы продемонстрировать это, воспользуемся интерфейсом High Resolution Time API на JavaScript, иначе называемым Performance API, чтобы получить более точные отметки времени, чем с помощью традиционных функций JavaScript вроде `Date.no()` и `console.time()`. В конечном счете это позволит нам точно измерить время, истекшее после вызова функции. Для внедрения операторов фиксации времени до и после тестируемой функции в данном примере будет использована монада типа `IO`. Исходный код требующихся для этой цели функций приведен в листинге 7.2, а в последующих примерах кода он будет опущен ради краткости.

Листинг 7.2. Измерение временны*х* характеристик производительности вызываемых функций с помощью комбинатора `tap()`

```
const start = () => now();
const runs = [];
const end = function (start) {
    let end = now();
    let result = (end - start).toFixed(3);
    runs.push(result);
    return result;
};
```

Вызвать функции `start()` и `end()` для измерения времени

Воспользоваться средствами Performance API для измерения времени в миллисекундах с точностью до трех десятичных цифр

```
const test = function (fn, input) {
    return () => fn(input);
};

const testRot13 = IO.from(start)
     .map(R.tap(test(rot13, 'functional_js_50_off')))
     .map(end);

testRot13.run(); // первый раз: 0.733 мс
testRot13.run(); // второй раз: 0.021 мс
assert.ok(runs[0] >= runs[1]);
```

Воспользоваться комбинатором `tap()`, чтобы распространить сведения о начальном моменте времени через монаду (это делается потому, что важен не результат функции, а время ее выполнения)

Как видите, в результате второго вызова функции `rot13()` та же самая символьная строка возвращается в мгновение ока. И хотя автоматическое запоминание не встроено в JavaScript, его можно внедрить вручную, расширив объект типа `Function`, как демонстрируется в листинге 7.3.

Листинг 7.3. Внедрение запоминания в вызовы функций

Внутренний вспомогательный метод, отвечающий за выполнение логики кеширования данного конкретного экземпляра функции

Преобразование ряда входных данных в символьную строку для получения идентификатора данной функции. Этот процесс можно сделать более надежным, выявив тип входных данных и применив к ним соответствующий алгоритм формирования ключей. Но в данном и других примерах можно обойтись и без этого

Создать внутренний локальный кеш для данного экземпляра функции

Попытаться прочитать сначала кеш, чтобы выяснить, обработаны ли входные данные. Если в кеше обнаружено значение, пропустить вызов данной функции и возвратить прежний ее результат, а иначе выполнить ее

Разрешить запоминание данной функции

Попытаться запомнить только унарные функции

Заключить экземпляр функции в оболочку запомненной функции

```
Function.prototype.memoized = function () {
    let key = JSON.stringify(arguments);
    this._cache = this._cache || {};
    this._cache[key] = this._cache[key] ||
                       this.apply(this, arguments);
    return this._cache[key];
};

Function.prototype.memoize = function () {
    let fn = this;
    if (fn.length === 0 || fn.length > 1) {
        return fn;
    }

    return function () {
        return fn.memoized.apply(fn, arguments);
    };
};
```

Благодаря расширению объекта типа `Function` запоминание в данной реализации становится универсальным, устраняя любые наблюдаемые побочные эффекты доступа к глобальному общему кешу. Кроме того, абстрагирование внутреннего механизма кеширования в функции делает его полностью независимым от тестов. Это означает, что вы освобождаетесь от обязанности снабжать свой код целым рядом операторов или тестировать функциональные

возможные кеширования, уделив основное внимание только тому, что должна делать сама функция.

Для прояснения общей картины на рис. 7.7 схематически представлена последовательность запоминания функции `rot13()`. При первом вызове этой функции значения в кеше еще нет, поэтому вычисляется сообщение в формате ROT13. А по завершении результат вычисления сохраняется по ключу, сформированному из входных аргументов, чтобы повторно воспользоваться им, пропустив все вычисления при последующем вызове.

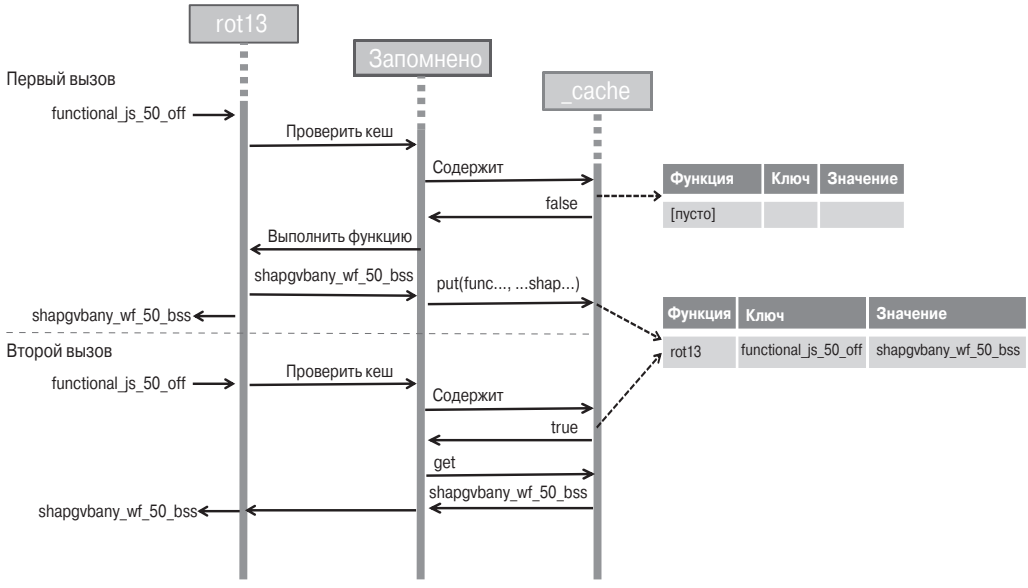


Рис. 7.7. Подробное представление двух вызовов функции **`rot13()`** с сообщением **`"functional_js_50_off"`**. В первый раз, когда кеш еще пуст, предоставляемый код скидки преобразуется по алгоритму ROT13. Полученный результат далее сохраняется во внутреннем кеше по ключу из этих входных данных. А во второй раз происходит попадание в кеш, в результате чего из него сразу же возвращается значение без повторного вычисления хеш-значения

Примечание

В примерах из этой книги запоминанию подвергаются функции с одним аргументом. Но как оперировать функциями с несколькими аргументами? Этот вопрос в данной книге не рассматривается, но оставляется вам для исследования в качестве упражнения. Для этого вы можете воспользоваться следующим двумя стратегиями: создать многомерный кеш (массив массивов) или однозначный ключ, состоящий из символьного представления аргументов.

Если внимательно проанализировать исходный код из листинга 7.3, то можно обнаружить, что запоминание ограничивается унарными функциями. Это сделано именно так с целью упростить этап формирования ключа в логике кеширования. Если же требуется подвергнуть запоминанию функции, которым

передаются несколько аргументов, соответствующая логика для формирования надлежащего ключа кеша может оказаться более сложной и затратной. Но иногда карринг может оказать помощь в разрешении данного вопроса.

7.3.3. Выгодное применение карринга и запоминания

Более сложные функции или те, которым передается несколько аргументов, труднее кешировать, даже если они чистые. Это объясняется увеличением сложности формирования ключа — операции, которая должна выполняться просто и быстро, чтобы исключить дополнительные издержки на уровне кеширования. Несколько упростить дело можно с помощью карринга. Как пояснялось в главе 4, "На пути к повторно используемому, модульному коду", карринг применяется для преобразования многомерной функции в унарную. Карринг позволяет запоминать функцию аналогично функции `safeFindObject()` через функцию `findStudent()`, как показано ниже.

```
const safeFindObject = R.curry(function (db, ssn) {
    // затратная поисковая операция ввода-вывода
});
```

Эта функция не является ссылочно-прозрачной, но на практике она специально кеширует результаты затратных операций поиска и полученных HTTP-запросов

```
const findStudent = safeFindObject(DB('students')).memoize();
findStudent('444-44-4444');
```

Такой прием оказывается работоспособным потому, что объект `DB` используется лишь для доступа к данным и никак не связан с процессом формирования уникального ключа для параметров функции `findStudent()`, к которому будут привязаны данные об учащемся, найденные по его уникальному идентификатору. Следует особо подчеркнуть, что преобразование функции в унарную требуется не только для того, чтобы упростить обращение с ней и ее композицию, но и для того, чтобы выгодно воспользоваться при запоминании декомпозицией (или разбиением) программы на мелкие части и реализовать кеширование по этим частям. И об этом речь пойдет далее.

7.3.4. Декомпозиция для максимального запоминания

Взаимосвязь запоминания с декомпозицией легче понять, если прибегнуть к простой аналогии из школьного курса химии. Как известно, любой раствор состоит из растворяемого вещества и растворителя. Скорость растворения вещества определяется многими факторами, в том числе *площадью поверхности*. Так, если приготовить два раствора сахара (кускового и песочного) в воде, то в процессе растворения сахар вступает в контакт с водой. И чем больше площадь его поверхности, тем быстрее он растворяется.

По этой же аналогии задачи могут быть разбиты на мелкие запоминаемые функции. Чем более мелкой становится структура прикладного кода, тем больше выгод можно извлечь из запоминания. Внутренний механизм кеширования каждой функции в отдельности вносит свой вклад в ускорение процесса выполнения программы, увеличивая, если угодно, поверхностный контакт.

Так, если обратиться снова к примеру программы `shoStudent`, то зачем выполнять проверку достоверности некоторых входных данных, если это уже было сделано раньше? И если объекты, представляющие учащихся, были извлечены по номерам их социального страхования из локальной базы данных, с помощью cookie-файлов или даже путем обращения к удаленному серверу, а их изменение не предполагается, то зачем тратить драгоценное время на повторный поиск? Примечательно, что если речь идет о функции `findStudent()`, то запоминание может служить в качестве небольшого кеша запросов, сохраняя уже извлеченные объекты для ускорения доступа к ним. Запоминание дает еще одну удобную возможность рассматривать функции как обыкновенные значения, вычисляемые по требованию. В качестве примера рассмотрим замену некоторых функций в программе `shoStudent` их запоминаемыми аналогами (где ради удобства имена запоминаемых функций снабжаются префиксом **`m_`**, хотя такое обозначение не является общепринятым). Приведенная ниже функция разбита на мелкие части, и благодаря этому выполнение всей программы ускоряется во втором случае на 75%!

```
const m_cleanInput = cleanInput.memoize();
const m_checkLengthSsn = checkLengthSsn.memoize();
const m_findStudent = findStudent.memoize();

const shoStudent = R.compose(
   map(append('#student-info')),
   liftIO,
   chain(csv),
   map(R.props(['ssn', 'firstname', 'lastname'])),
   map(m_findStudent),
   map(m_checkLengthSsn),
   lift(m_cleanInput));

shoStudent('444-44-4444').run();
    // -> в среднем 9.2 мс (без запоминания)
shoStudent('444-44-4444').run();
    // -> в среднем 2.5 мс (с запоминанием)
```

Еще одной разновидностью декомпозиции является рекурсия, где программа разбивается на самоподобные, мелкие, запоминаемые подзадачи. Аналогично запоминание позволяет заметно ускорить медленно действующий рекурсивный алгоритм.

7.3.5. Применение запоминания к рекурсивным вызовам

Рекурсия способна привести к зависанию браузера или генерированию малоприятных исключений. Обычно это происходит в том случае, когда размер стека выходит из-под контроля, например, при обработке очень крупного массива входных данных. Но иногда запоминание может оказать помощь в решении данного вопроса. Как пояснялось в главе 3, “Меньше структур данных и больше операций”, рекурсия — это механизм разбиения задачи на более мел-

кие варианты ее самой. Как правило, при рекурсивном вызове решается одна и та же задача или подмножество более крупной задачи, и делается это неоднократно до тех пор, пока не будет достигнут основной вариант, когда наконец-то начинается сворачивание стека вызовов и возврат получаемого результата. Если бы можно было кешировать результаты решения подзадач, то удалось бы повысить производительность при вызове одной и той же функции с более крупными входными данными.

В качестве примера рассмотрим простую функцию, вычисляющую факториал числа *n*, обозначаемый как *n!*. Он вычисляется как произведение всех положительных значений, меньших и равных *n*, по следующей формуле:

```
n! = n * (n - 1) * (n - 2) * … * 3 * 2 * 1
```

Например:

```
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 4 * 3! = 24
```

Обратите внимание на то, что числа факториала могут быть также определены рекурсивно в виде более мелких факториалов, например 4! = 4 × 3!

Программу для решения этой задачи можно изящно выразить в виде следующего запоминаемого рекурсивного решения:

```
const factorial = ((n) => (n === 0) ? 1
                : (n * factorial(n - 1))).memoize();

factorial(100); // -> В первый раз выполняется за 0,299 мс,
factorial(101); // -> а во второй раз — за 0,021 мс
```

Вычислить подряд произведения всех чисел 100 × 99 × 98 × … × 3 × 2 × 1

Воспользоваться кешированным ранее значением, чтобы сократить вычисление, остановившись на произведении 101 × 100!

При запоминании применяются математические принципы вычисления факториалов, и поэтому при втором вызове приведенной выше функции достигается значительное повышение производительности, когда функция "вспоминает" формулу `101! = 101 × 100!` и может повторно воспользоваться значением, возвращаемым из первого вызова `factorial(100)`. В итоге вычисление всего алгоритма резко сокращается и результат возвращается мгновенно. Такой подход имеет и другие преимущества с точки зрения управления фреймами стека вызовов, исключая его засорение (рис. 7.8).

Как видите, алгоритм вычисления факториала выполняется полностью при первом вызове функции `factorial(100)`. В итоге создается 100 фреймов в стеке вызовов данной функции. И в этом проявляется следующий недостаток некоторых рекурсивных решений: в них нерационально используется свободное пространство стека, особенно в таких случаях, как при вызове функции `factorial()`, где количество создаваемых в стеке фреймов пропорционально получаемым входным данным. Но благодаря запоминанию количество фреймов, требующихся в стеке для вычисления следующего числа, значительно сокращается.

Запоминание — не единственный способ оптимизации рекурсивных вызовов. Имеются и другие способы повышения производительности с помощью средств компилятора.

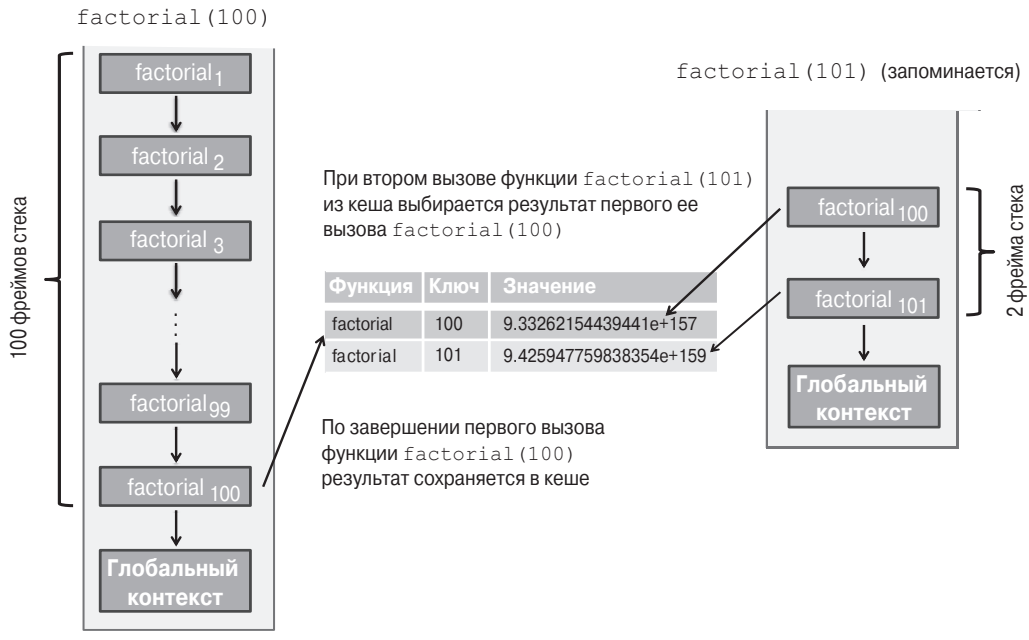


Рис. 7.8. При вызове запоминаемой функции **`factorial(100)`** в первый раз создается **100** фреймов стека, поскольку требуется вычислить факториал числа **`100!`**, умножая все числа подряд. А при вызове функции **`factorial(101)`** во второй раз повторно используется результат первого вызова, и при этом создаются только два фрейма стека

7.4. Рекурсия и оптимизация хвостовых вызовов

Как было показано ранее, в программах с рекурсией стек используется намного более интенсивно, чем в программах без рекурсии. В некоторых языках ФП даже отсутствуют встроенные механизмы организации циклов, а основной акцент делается на рекурсию и запоминание для реализации эффективной итерации. Но иногда не особенно помогает даже запоминание, например, когда характер входных данных функции постоянно меняется. И в этом случае мало что дает наличие внутреннего уровня кеширования. Можно ли оптимизировать рекурсию, чтобы она выполнялась так же эффективно, как и циклы в подобных случаях? Оказывается, что рекурсивные алгоритмы можно составить таким образом, чтобы компиляторы помогали достичь этого с помощью *оптимизации хвостовых вызовов*[2] *(tail-call optimization, TCO)*. В этом разделе поясняется, что приведенная ниже версия рекурсивной функции `factorial()`:

```
const factorial = (n, current = 1) =>
                  (n === 1) ? current
                : factorial(n - 1, n * current);
```

Рекурсивная стадия в этой версии функции теперь выполняется в последнем операторе, находящемся на так называемой *хвостовой* позиции

[2] Иногда этот термин переводят как *оптимизация хвостовой рекурсии*. — *Примеч. ред.*

несколько отличается от предыдущей версии, поскольку рекурсивная стадия размещается в ней на хвостовой позиции. Поэтому она выполняется также быстро, как и приведенная ниже императивная версия функции `factorial()`.

```
var factorial = function (n) {
    let result = 1;
    for(let x = n; x > 1; x--) {
        result *= x;
    }
    return result;
}
```

Оптимизация хвостовых вызовов, является усовершенствованием, внедренным в компилятор стандарта ES6 с целью свести выполнение всех вызовов рекурсивной функции в едином фрейме. Но это может произойти только на последней стадии решения рекурсивной задачи, когда вызывается другая (как правило, та же самая) функция. И этот последний вызов находится в самом *хвосте* функции, откуда и пошло название данного способа оптимизации рекурсивных вызовов.

В чем же состоит подобная оптимизация? Вызывая функцию в самом конце рекурсивной функции, можно дать интерпретатору JavaScript понять, что он может больше не хранить текущий фрейм в стеке, поскольку он больше не понадобиться и может быть удален. Как правило, это достигается передачей всего необходимого состояния из контекста одной функции в другой как части ее аргументов, что и демонстрировалось выше на примере рекурсивной функции `factorial()`. Таким образом, рекурсивная итерация будет происходить всякий раз в новом фрейме, в качестве которого будет использоваться фрейм из предыдущего вызова, вместо нагромождения фреймов друг на друга в стеке. Приведенная выше рекурсивная функция `factorial()` определена в хвостовой форме, и поэтому выполнение вызова `factorial(4)` переходит от типичной пирамиды рекурсивных вызовов:

```
factorial(4)
  4 * factorial(3)
    4 * 3 * factorial(2)
      4 * 3 * 2 * factorial(1)
        4 * 3 * 2 * 1 * factorial(0)
          4 * 3 * 2 * 1 * 1
      4 * 3 * 2 * 1
    4 * 3 * 2
  4 * 6
return 24
```

к плоской структуре с учетом стека контекстов выполнения, как показано ниже.

```
factorial(4)
  factorial(3, 4)
  factorial(2, 12)
  factorial(1, 24)
  factorial(0, 24)
```

```
  return 24
return 24
```

Как видите, эта плоская структура позволяет более эффективно пользоваться стеком, который больше не придется сворачивать на *n* фреймов. На рис. 7.9 поэтапно демонстрируется процесс преобразования нехвостовой функции `factorial()` в ее хвостово-рекурсивную версию.

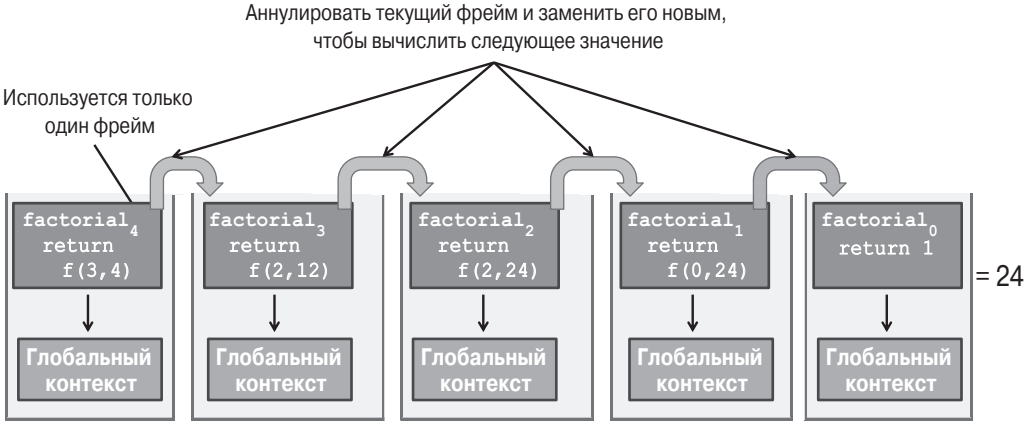


Рис. 7.9. Иллюстрация вызовов хвостово-рекурсивной функции **`factorial(4)`**. Как видите, в этой функции используется единственный фрейм. Оптимизация хвостовых вызовов заключается в аннулировании фрейма текущей функции в стеке и распределении на его месте нового фрейма, как будто бы функция **`factorial()`** вычисляется в цикле

7.4.1. Преобразование нехвостовых вызовов в хвостовые

Оптимизируем функцию `factorial()`, чтобы выгодно воспользоваться механизмом оптимизации хвостовых вызовов в JavaScript. В приведенной ниже первоначальной реализации:

```
const factorial = (n) =>
      (n === 1) ? 1
    : (n * factorial(n - 1));
```

рекурсивная функция `factorial()` не находилась на хвостовой позиции, поскольку в последнем возвращаемом выражении многократно умножается значение, вычисляемое на рекурсивной стадии:

```
n * factorial(n - 1)
```

Напомним, что для оптимизации хвостовых вызовов последняя стадия должна быть рекурсивной. Ведь именно это дает интерпретатору возможность преобразовать функцию `factorial()` в цикл. Это делается в следующие два этапа.

1. Перемещение операции умножения на место дополнительного параметра функции, чтобы отслеживать текущее состояние данной операции.

2. Применение новшества ES6 — стандартного значения параметров функции, определяющих предварительное значение данного аргумента. Такой аргумент можно было бы определить и иначе, но при использовании стандартного значения параметров функции это будет выглядеть намного яснее:

```
const factorial = (n, current = 1) =>
                  (n === 1) ? current :
                   factorial(n - 1, n * current);
```

Теперь данная функция вычисления факториала будет выполняться так, как будто она была реализована путем организации стандартного цикла, не требуя создания дополнительных фреймов в стеке, но в то же время сохраняя в какой-то степени свой первоначальный декларативный и математический вид. Подобное преобразование возможно потому, что свойства хвостово-рекурсивная функции в большой степени совпадают со свойствами стандартного цикла, как показано на рис. 7.10.

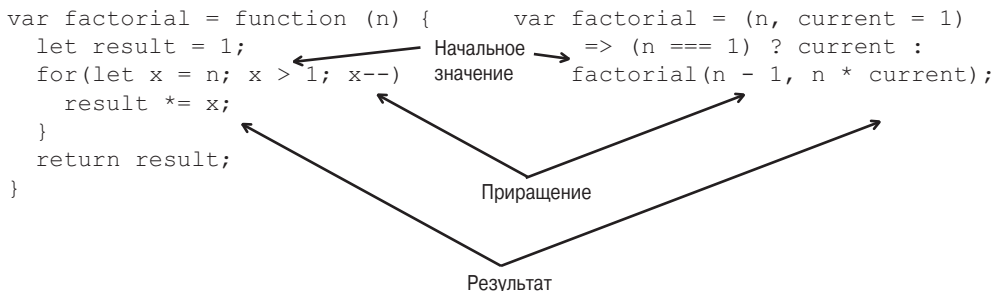


Рис. 7.10. Сходство стандартного цикла (*слева*) с эквивалентной ему хвостово-рекурсивной функцией (*справа*). В обоих примерах кода можно легко выявить начальное значение, приращение цикла и накопление результата в аккумуляторе

Рассмотрим еще один пример. В главе 3, "Меньше структур данных и больше операций", демонстрировалось следующее небольшое рекурсивное решение задачи суммирования всех элементов в массиве:

```
function sum(arr) {
    if(_.isEmpty(arr)) {
        return 0;
    }
    return _.first(arr) + sum(_.rest(arr));
}
```

И в этом случае можно заметить, что последнее действие в данной функции, т.е. `_.first(arr) + sum(_.rest(arr))`, выполняется не в хвостовой форме. Поэтому реорганизуем ее код, чтобы оптимизировать его в отношении расхода оперативной памяти. Любые данные, которые должны быть переданы последующим вызовам данной функции, в приведенной ниже оптимизированной версии теперь введены в состав ее аргументов.

```
function sum(arr, acc = 0) {
    if(_.isEmpty(arr)) {
        return 0;
    }
    return sum(_.rest(arr), acc + _.first(arr));
}
```

Хвостовая рекурсия приближает производительность рекурсивного цикла к циклу, организуемому вручную. Таким образом, в тех языках, где такая рекурсия внедрена (например, в стандарте ES6 языка JavaScript), ею можно заменить организуемые вручную циклы, когда первостепенное значение приобретает производительность, и в то же время требуется сохранить правильность применяемого алгоритма и контролировать мутации. Но применение оптимизации хвостовых вызовов не ограничивается только рекурсией. Ее можно, например, применить к любой функции, в последнем операторе которой вызывается другая функция, что нередко происходит в приложениях на JavaScript. Следует заметить, что оптимизация хвостовых вызовов, являющая новинкой стандарта ES6, была разработана в черновом варианте еще в стандарте ES4, но пока широко не используется в браузерах. По сути, на момент написания данной книги оптимизация хвостовых вызовов еще не была толком реализована ни в одном из популярных браузеров. Именно поэтому приходится пользоваться транспилятором Babel, чтобы привести исходный код стандарта ES6 к прежней версии.

Эмуляция оптимизации хвостовых вызовов в стандарте ES5

В наиболее распространенном в настоящее время стандарте ES5 языка JavaScript оптимизация хвостовых вызовов не поддерживается. Она была введена в стандарт ES6 этого языка в качестве предложения под названием *Специфика хвостовых вызовов (proper tail calls)* (см. раздел 14.6 спецификации ECMA-262). Как упоминалось в главе 2, "Сценарий высшего порядка", работоспособность примеров, представленных в данной книге, обеспечивается с помощью транспилятора Babel, выполняющего роль компилятора исходного кода из нового стандарта в прежний, что очень удобно для тестирования перспективных языковых средств.

В качестве обходного приема можно прибегнуть к средству, метафорически называемому *прыжками на батуте (trampolining)*. Это средство позволяет смоделировать хвостовую рекурсию итеративным способом, что идеально подходит для контроля над ростом стека вызовов функций в таких языках со стековой организацией, как JavaScript.

Батутом здесь служит комбинатор функций, которому передается другая функция в качестве входных данных, а он повторно вызывает ее (т.е. раскачивает ее как на батуте) до тех пор, пока выполняется заданное условие. Повторно вызываемая функция инкапсулирована в структуру, которая называется *переходником*[3] *(thunk)* и представляет собой оболочку для функции, служащую вспомогательным средством для вызова другой функции. В контексте функционального ха-

[3] В простонародье — *санками*. — *Примеч. ред.*

рактера языка JavaScript в переходники помещают выражение в виде анонимной функции без параметров, которое используется в качестве аргументов и позволяет отложить тем самым свое вычисление до тех пор, пока принимающая функция не вызовет анонимную функцию.

Подробное рассмотрение батутов и переходников выходит за рамки данной книги, но если вы отчаянно ищите пути оптимизации своих рекурсивных функций, начните поиск с более основательного изучения этих средств. А проверить совместимость оптимизации хвостовых вызовов с другими языковыми средствами в стандарте ES6 можно на веб-сайте по адресу **`https://kangax.github.io/compat-table/es6/`**.

Если требуется организовать непрерывный цикл визуализации графики или обработку крупных массивов данных за короткий период времени, то на первый план выходит производительность. В подобных случаях приходится идти на вынужденные компромиссы, жертвуя изящностью кода в пользу быстрого выполнения поставленной задачи. Для этой цели рекомендуется придерживаться стандартных циклов, хотя для большинства прикладных потребностей функциональное программирование по-прежнему остается весьма производительной методикой написания кода. Оптимизацию прикладного кода следует всегда выполнять в последнюю очередь, а в некоторых крайних случаях, когда требуется повысить производительность, сэкономив хотя бы немного миллисекунд, можно всегда воспользоваться методиками, представленными в этой главе.

Всякому решению при разработке программного обеспечения противоборствует равнозначная сила. Но при разработке большинства приложений жертвование эффективностью в пользу удобства сопровождения считается вполне допустимым компромиссом. Ведь лучше написать код, который удобно читать и отлаживать, даже если он и не самый быстродействующий. Как сказал Дональд Кнут: “В 97 случаях из 100 экономия нескольких миллисекунд при оптимизации написанного кода не имеет особого значения, особенно в сравнении с той ценностью, которую приносит написание удобного для сопровождения кода”.

Парадигма функционального программирования имеет вполне законченный характер. Она обеспечивает богатый уровень абстракции и переадресации, намечая интересные пути для достижения эффективности. До сих пор поясняло, как создавать функциональные программы с линейными потоками данных посредством связывания в цепочку и композиции. Но, как известно, программы на JavaScript сочетают в себе немало нелинейного или асинхронного поведения, например, при обработке вводимых пользователем данных или составлении удаленных HTTP-запросов. Разрешению подобных затруднений посвящена глава 8, “Обработка асинхронных событий и данных”, в которой рассматривается также парадигма реактивного программирования, построенная на принципах функционального программирования.

Резюме

Из этой главы вы узнали следующее.

- В некоторых случаях функциональный код может работать медленнее или потреблять больше оперативной памяти, чем равнозначный ему императивный код.
- Реализуя стратегию отложенного вычисления, можно выгодно воспользоваться комбинатором чередования функций и соответствующей поддержкой в функциональных библиотеках вроде Lodash.
- Запоминание может быть использовано как стратегия внутреннего кеширования на уровне функций с целью исключить дублирование вычисления потенциально затратных функций.
- Декомпозиция (или разбиение) программ на простые функции позволяет не только создавать расширяемый код, но и повышать его эффективность через запоминание.
- Декомпозиция распространяется также на рекурсию как методика решения сложной задачи путем ее разбиения на более простые самоподобные задачи, где возможности запоминания используются в полной мере с целью оптимизировать применение стека контекста выполнения.
- Преобразуя функции в хвостово-рекурсивную форму, можно выгодно воспользоваться исключением хвостовых вызовов как одним из средств оптимизации, встроенной в компилятор.