1 Многократное использование кода и его оптимизация

Язык JavaScript имеет незаслуженно сомнительную славу. Немало написано о том, что он обладает весьма ограниченными возможностями в области объектно-ориентированного программирования (ООП). Некоторые авторы даже не признают за JavaScript права называться объектно-ориентированным языком (но это самый настоящий объектно-ориентированный язык). Несмотря на очевидное синтаксическое сходство JavaScript и применяемых в ООП языков, основанных на классах, в JavaScript отсутствует оператор `Class` (или его эквивалент). Здесь также нет никакого очевидного способа внедрения популярных методологий ООП, в частности наследования (многократного использования кода) и инкапсуляции. К тому же JavaScript характеризуется слабой типизацией, не имеет компилятора. В такой ситуации вы получаете слишком мало предупреждений об ошибках и можете не заметить, что дела пошли плохо. Этот язык прощает вам практически все и всегда. Благодаря этому нюансу ничего не подозревающий программист одновременно приобретает и широчайшую свободу действий, и километровую нить Ариадны, позволяющую выбраться из любого лабиринта.

Программист, привыкший работать с более классическим и строго регламентированным языком, просто теряется, сталкиваясь с блаженным неведением JavaScript относительно практически любого феномена программирования. Функции и переменные по умолчанию являются глобальными, а если вы пропустите где-нибудь точку с запятой, не случится ничего страшного.

Скорее всего, причины подобного отношения программистов заключаются в недопонимании того, что же такое JavaScript. Написать приложение на JavaScript будет гораздо проще, если программист усвоит две фундаментальные истины:

- JavaScript — это язык, основанный не на классах;
- хороший код может получиться и без применения ООП, основанного на классах.

Некоторые программисты пытаются воспроизводить на почве JavaScript классовую природу таких языков, как C++, но подобные упражнения равносильны заталкиванию квадратного колышка в круглое отверстие. В каких-то случаях это удается, но только результат получается вымученным.

Идеального языка программирования не существует, и можно даже утверждать, что мнимое превосходство определенных языков программирования

(а в сущности — мнимое превосходство самой объектно-ориентированной модели) — это история, напоминающая казус с «новым платьем короля». По моему опыту, в программах, написанных на C++, Java или PHP, встречается не меньше ошибок или проблем, чем в проектах, где применяется только JavaScript. На самом деле (глубокий вдох) можно предположить, что благодаря гибкой и выразительной природе JavaScript проекты на этом языке пишутся быстрее, чем на каком-либо другом.

К счастью, большинство недостатков, присущих JavaScript, несложно избежать. Но не путем насильственного превращения его в нескладную имитацию другого языка, а благодаря использованию исключительной гибкости, присущей JavaScript и позволяющей обходить сложные ситуации. Классовая природа других языков иногда приводит к выстраиванию громоздких иерархических структур, а также к неуклюжести кода, в котором накапливается слишком «много букв». В JavaScript действуют другие принципы наследования, которые не менее полезны, чем в классовых языках, но при этом замечательно легки.

Существует большое количество способов организации наследования в JavaScript — и для такого пластичного языка это неудивительно. В следующем коде используется *наследование через прототипы* (Prototypal Inheritance) — мы создаем объект `Pet` (Домашний любимец), а потом объект `Cat`, наследующий от него. Такой способ наследования часто рассматривается в учебниках по JavaScript и считается классической техникой этого языка.

```
// Определяем объект Pet. Сообщаем ему имя и количество лап.
var Pet = function (name, legs) {
    this.name = name; // Сохраняем значения имени и количества лап.
    this.legs = legs;
};

// Создаем метод, отображающий имя Pet и количество лап.
Pet.prototype.getDetails = function () {
    return this.name + ' has ' + this.legs + ' legs';
};

// Определяем объект Cat, наследующий от Pet.
    var Cat = function (name) {
    Pet.call(this, name, 4); // Вызываем конструктор родительского объекта.
};

// В этой строке осуществляется наследование от Pet.
    Cat.prototype = new Pet();

// Дописываем в Cat метод действия.
Cat.prototype.action = function () {
    return 'Catch a bird';
};

// Создаем экземпляр Cat в petCat.
var petCat = new Cat('Felix');

var details = petCat.getDetails();     // 'У Феликса 4 лапы'.
```

```
var action = petCat.action();          // 'Поймал птичку'.
petCat.name = 'Sylvester';             // Изменяем имя petCat.
petCat.legs = 7;                       // Изменяем количество лап petCat!!!
details = petCat.getDetails();         // 'У Сильвестра 7 лап'.
```

Этот код работает, но он не слишком красив. Использовать оператор `new` целесообразно, если вы имеете опыт работы с объектно-ориентированными языками, например C++ или Java, но из-за применения ключевого слова `prototype` код раздувается. К тому же здесь отсутствует приватность кода — обратите внимание, как свойство `legs` объекта `petCat` приобрело явно несуразное значение 7. Такой метод наследования не гарантирует защиты от внешнего вмешательства. Этот недостаток может иметь более серьезные последствия в сравнительно сложных проектах, в которых участвует несколько программистов.

Другой вариант — вообще отказаться от применения `prototype` или `new`, а вместо этого воспользоваться умением JavaScript впитывать и дополнять экземпляры объектов. Такой принцип работы называется *функциональным наследованием* (Functional Inheritance):

```
// Определяем объект pet. Сообщаем ему имя и количество лап.
var pet = function (name, legs) {
    // Создаем объектный литерал (that). Включаем свойство name
    // для общедоступного использования и функцию getDetails().
    // Лапы остаются приватными. Любые локальные переменные,
    // определенные здесь или переданные pet в качестве аргументов,
    // остаются приватными, но тем не менее будут доступны из функций,
    // которые определяются ниже.
    var that = {
        name: name,
        getDetails: function () {
            // По правилам видимости, действующим в JavaScript,
            // переменная legs будет доступна здесь (замыкание),
            // несмотря на то что она недоступна извне объекта pet.
            return that.name + ' has ' + legs + ' legs';
        }
    };
    return that;
};

// Определяем объект cat, наследующий от pet.
var cat = function (name) {
    var that = pet(name, 4); // Наследуем от pet.
    // Дописываем в cat метод действия.
    that.action = function () {
        return 'Catch a bird';
    };
    return that;
};

// Создаем экземпляр cat в petCat2.
var petCat2 = cat('Felix');
```

```
details = petCat2.getDetails();     // 'У Феликса 4 лапы'.
action = petCat2.action();          // 'Поймал птичку'.
petCat2.name = 'Sylvester';         // Мы можем изменить имя.
petCat2.legs = 7;                   // Но не количество лап!
details = petCat2.getDetails();     // 'У Сильвестра 4 лапы'.
```

Итак, здесь уже не возникает комичная ситуация с `prototype`, все красиво инкапсулировано. Наша попытка изменить несуществующее общедоступное свойство `legs` извне `cat` просто приводит к созданию неиспользуемого общедоступного свойства `legs`. Настоящее значение `legs` надежно запрятано в замыкании, созданном методом `getDetails()` объекта `pet`. Замыкание позволяет сохранить локальные переменные из функции — в данном случае `pet()` — после того, как завершится выполнение функции.

На самом деле в JavaScript нет «однозначно правильного» способа наследования. Лично я нахожу функциональное наследование наиболее естественным способом работы с JavaScript. Для вас и для вашего приложения больше могут подойти другие методы. Наберите в Google `Наследование в JavaScript` — и вы найдете множество интересных онлайновых ресурсов.

Одно из достоинств использования наследования через прототипы заключается в эффективном применении памяти; свойства и методы прототипа объекта сохраняются лишь однажды, независимо от того, сколько раз от них происходит наследование.

У функционального наследования такого полезного свойства нет; в каждом новом экземпляре свойства и методы будут создаваться заново, то есть дублироваться. Это обстоятельство может представлять проблему, если вы создаете многочисленные экземпляры крупных объектов (возможно, тысячи) и для вас критичен объем потребляемой памяти. Одно из решений такой проблемы — сохранять любые крупные свойства или методы в объекте, а потом передавать этот объект функциям конструктора в качестве аргумента. После этого все экземпляры смогут использовать один и тот же объектный ресурс, не создавая при этом собственных версий объекта.

Чтобы все работало быстро

Сама концепция «быстродействующей графики на JavaScript» может показаться оксюмороном.

Честно признаться, JavaScript в сочетании с обычным веб-браузером — не лучший инструментарий для создания ультрасовременных аркадных программ (по крайней мере сейчас). Но JavaScript может предоставить довольно много возможностей для разработки блестящих, скоростных и графически насыщенных приложений, в том числе игр. Имеющиеся для этого инструменты, конечно, не самые быстрые — зато они бесплатные, гибкие и удобные в работе.

Поскольку JavaScript — это интерпретируемый язык, в нем неприменимы многие виды оптимизации, используемые во время компиляции в таких языках, как C++. Хотя в современных браузерах наблюдается колоссальный прогресс, касающийся возможностей работы с JavaScript, еще есть потенциал для увеличения

скорости исполнения программ. Именно вы, программист, решаете, какими алгоритмами воспользоваться, какой код оптимизировать, как наиболее эффективно управлять объектной моделью документа. Никакой автоматический оптимизатор за вас этого не сделает.

Приложение JavaScript, которое просто контролирует отдельно взятый щелчок кнопкой мыши либо выполняет нерегулярные вызовы Ajax, пожалуй, не требует никакой оптимизации, за исключением случаев, когда код написан из рук вон плохо. Приложения, рассматриваемые в этой книге, таковы по природе, что для обеспечения удовлетворительной функциональности (с точки зрения пользователя) просто не обойтись без очень эффективного кода. Кому понравится, если перемещающаяся на экране графика будет двигаться медленно и выглядеть неаккуратно?

В оставшейся части этой главы мы не будем касаться того, как ускорить загрузку страницы с сервера. Мы сосредоточимся именно на оптимизации рабочего кода, который выполняется уже после того, как ресурсы загружены с сервера. Еще точнее — мы обсудим способы оптимизации, полезные при программировании графики на JavaScript.

Что и когда оптимизировать

Важно знать, когда следует делать оптимизацию, и не менее важно — когда *не нужно* ее делать. Непродуманная оптимизация может приводить к появлению неразборчивого кода и возникновению ошибок. Вряд ли оправданно оптимизировать такие фрагменты приложения, исполнять которые приходится нечасто. Здесь стоит вспомнить о законе Парето, или о правиле 80–20: на 20 % кода затрачивается 80 % процессорных циклов. Сосредоточьтесь на этих 20 %, если угодно — даже на 10 % или 5 %, а остальное можете игнорировать. Тогда и количество ошибок уменьшится, большая часть кода останется удобочитаемой, а вы приобретете душевное спокойствие.

Если применять в работе подходящие профилировочные инструменты, например Firebug, то сразу станет ясно, на исполнение каких функций затрачивается больше всего времени. Покопавшись в функциях, вы сами решите, какой код нуждается в оптимизации. К сожалению, инструмент Firebug имеется только в браузере Firefox. В других браузерах есть свои профилировщики. Чем старше версия браузера, тем вероятнее, что там такого инструмента не окажется.

На рис. 1.1 показан профилировщик Firebug в действии. В меню Консоль выберите команду Профилирование, чтобы приступить к профилированию, а чтобы прекратить профилирование — вновь нажмите Профилирование. После этого Firebug отобразит сообщение о прерывании всех функций JavaScript, вызванных в период между начальной и конечной точками. Информация будет отображаться в таком порядке:

- Функция — имя вызываемой функции;
- Вызовы — указывает, сколько раз была вызвана функция;
- Процент — процент от общего времени, потраченный на выполнение функции;

- Собственное время — время, проведенное внутри функции, исключая вызовы, направленные к другим функциям;
- Время — общее время, проведенное внутри функции, с учетом вызовов, которые были направлены к другим функциям;
- Средн. — среднее значение собственного времени;
- Мин. — наименьшее время, уходящее на исполнение функции;
- Макс. — наибольшее время, уходящее на исполнение функции;
- Файл — файл JavaScript, в котором находится функция.

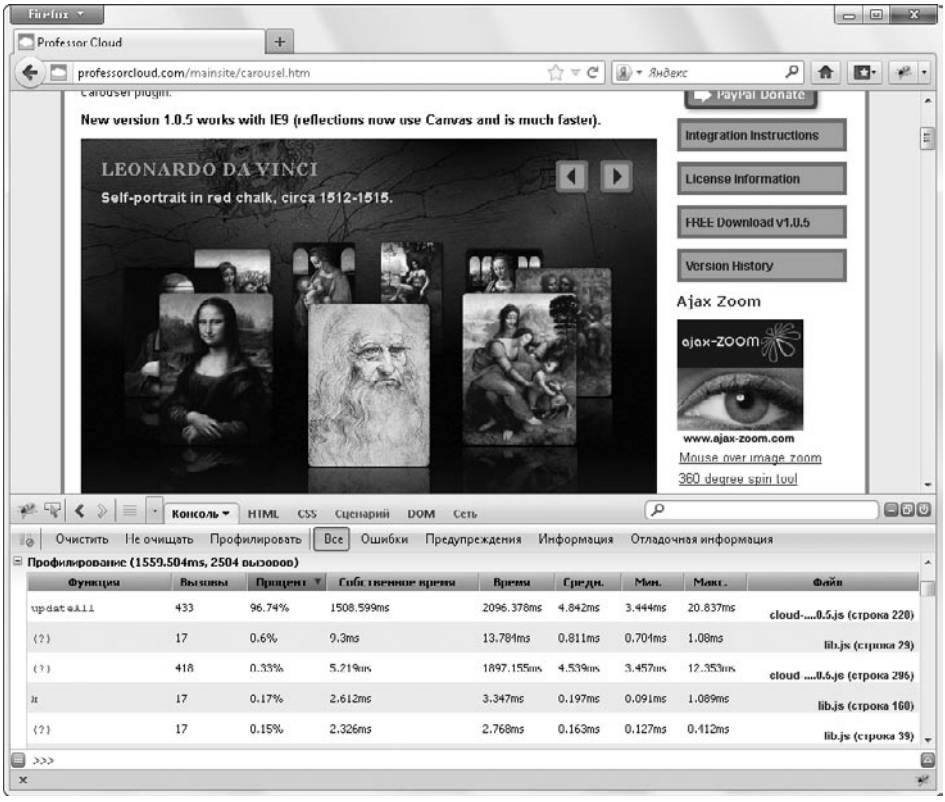

Рис. 1.1. Профилировщик Firebug в действии

Если у нас будет возможность создавать собственные профилировочные тесты, которые будут работать во всех браузерах, то мы сможем ускорить разработку и обеспечить возможности профилирования там, где их раньше не существовало. Затем нам останется просто загрузить в каждый из интересующих нас браузеров одну и ту же тестовую страницу, а потом считать результаты. Кроме того, так нам будет удобно быстро проверять мелкие оптимизации внутри функции. О том, как создавать собственные профилировочные тесты, мы подробно поговорим в разделе «Ремесло профилирования кода».