6

Отзывчивые интерфейсы

Нет ничего более обескураживающего, чем отсутствие реакции на щелчок мышью на каком-либо элементе веб-страницы. Эта проблема порождена самой сущностью транзакционных веб-приложений; это она привела к появлению предупреждения «не щелкайте дважды», почти всегда сопровождающего кнопки отправки в большинстве форм. Пользователи испытывают естественное стремление повторить действие, которое не привело к видимым изменениям, и поэтому обеспечение отзывчивости веб-приложений является важной задачей, связанной с производительностью.

В главе 1 была представлена концепция потока выполнения, обслуживающего пользовательский интерфейс броузера. Коротко напомню, что в большинстве броузеров обновление пользовательского интерфейса и выполнение программного кода JavaScript-сценариев производится в рамках единственного процесса. В каждый конкретный момент времени может выполняться только одна из этих операций, то есть пока выполняется программный код на JavaScript, пользовательский интерфейс не может откликаться на действия пользователя, и наоборот. Фактически на время выполнения программного кода сценария пользовательский интерфейс оказывается «заблокированным», поэтому для формирования субъективного восприятия производительности веб-приложения большое значение имеет, сколько времени займет выполнение вашего программного кода.

Поток выполнения пользовательского интерфейса броузера

Процесс, одновременно используемый для выполнения программного кода на языке JavaScript и для обслуживания пользовательского интер-

фейса, часто называют потоком выполнения пользовательского интерфейса (или главным потоком выполнения) броузера (хотя термин «поток выполнения» может по-разному трактоваться в разных броузерах). Поток пользовательского интерфейса обслуживает простую очередь, где хранятся задания, пока процесс занят решением других задач. Как только процесс освобождается, он извлекает из очереди следующее задание и выполняет его. Заданиями могут быть и выполнение некоторого программного кода на JavaScript, и обновление пользовательского интерфейса, в том числе перерисовывание и перекомпоновка (обсуждаются в главе 3). Важно отметить, что каждое отдельное действие пользователя может приводить к добавлению в очередь одно или несколько заданий.

Рассмотрим простой интерфейс, где щелчок на кнопке приводит к отображению сообщения:

```
<html>
<head>
    <title>Browser UI Thread Example</title>
</head>
<body>
    <button onclick="handleClick()">Click Me</button>
    <script type="text/javascript">

        function handleClick(){
            var div = document.createElement("div");
            div.innerHTML = "Clicked!";
            document.body.appendChild(div);
        }

    </script>
</body>
</html>
```

Щелчок на кнопке в этом примере вынудит главный поток выполнения создать два задания и добавить их в очередь. Первое задание – обновить изображение кнопки, придав ей вид нажатой кнопки, и второе – выполнить программный код метода `handleClick()` так, чтобы был выполнен только этот метод, а также любые другие методы, вызванные им. Допустим, что главный поток выполнения не занят другими делами. Он извлечет из очереди первое задание и обновит изображение кнопки. Затем извлечет второе задание и выполнит программный код на языке JavaScript. В процессе выполнения метода `handleClick()` будет создан новый элемент `<div>` и добавлен в конец элемента `<body>`, что фактически приведет к еще одному изменению пользовательского интерфейса. То есть в ходе выполнения программного кода на JavaScript в очередь будет добавлено новое задание обновления пользовательского интерфейса, которое будет выполнено сразу по завершении работы программного кода на языке JavaScript, как показано на рис. 6.1.

Когда главный поток выполнения выполнит все задания, процесс перейдет в холостой режим, ожидая появления новых заданий в очереди.

Холостой режим является идеальным состоянием, потому что в этом случае любые действия пользователя приводят к немедленному обновлению пользовательского интерфейса. Если пользователь попытается взаимодействовать со страницей во время выполнения какого-либо задания, пользовательский интерфейс не только не обновится немедленно, но, возможно, даже не будет создано и поставлено в очередь новое задание обновления пользовательского интерфейса. На практике большинство броузеров прекращают добавление заданий в очередь, пока выполняется программный код на JavaScript, поэтому обработка заданий, связанных с выполнением программного кода, должна завершаться как можно быстрее, чтобы не вызывать негативные впечатления у пользователя.

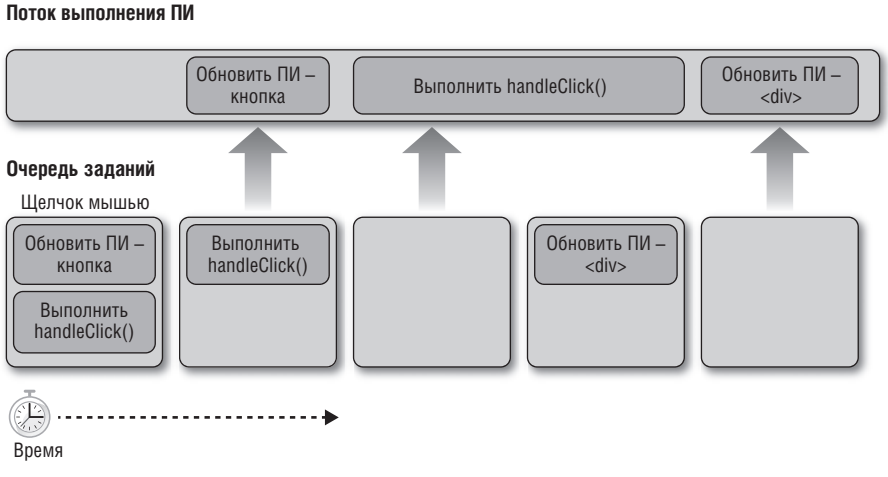

Рис. 6.1. *Добавление заданий главного потока выполнения по мере взаимодействия пользователя со страницей*

Ограничения броузеров

Броузеры устанавливают ограничение на время, которое отводится на выполнение JavaScript-сценария. Это ограничение объясняется необходимостью предотвратить блокирование броузера или компьютера злонамеренным программным кодом, выполняющим бесконечные интенсивные вычисления. Всего существует два таких ограничения: *ограничение размера стека вызовов* (обсуждается в главе 4) и *ограничение времени выполнения сценариев*. Ограничение времени выполнения сценария иногда называют таймером продолжительности выполнения сценариев или таймером защиты от неуправляемых сценариев, и основная его идея основана на том, что броузер следит за продолжительностью работы сценария и прерывает его выполнение, как только будет превышен установленный предел. По достижении предельного значения на экране появляется диалог, показанный на рис. 6.2.

Рис. 6.2. *Предупреждающий диалог, который выводится броузером Internet Explorer после выполнения более 5 миллионов инструкций*

Существует два способа ограничения продолжительности выполнения сценариев. Первый из них заключается в том, чтобы следить за количеством инструкций, выполненных с момента запуска программного кода. При таком подходе максимальная продолжительность работы сценариев на различных компьютерах может отличаться в зависимости от объема доступной памяти и быстродействия процессора, определяющих скорость выполнения одной инструкции. Второй подход заключается в ограничении времени выполнения сценария. Количество инструкций, которые могут быть выполнены в отведенный интервал времени, также может отличаться на разных компьютерах в зависимости от их быстродействия, но в любом случае работа сценария будет прервана по истечении установленного времени. Естественно, все броузеры используют несколько отличающиеся подходы к измерению продолжительности выполнения сценариев:

- Internet Explorer начиная с версии 4 по умолчанию ограничивает продолжительность выполнения сценариев 5 миллионами инструкций; значение этого предела хранится в ключе реестра Windows *HKEY_CURRENT_USER\Software\Microsoft\InternetExplorer\Styles\MaxScriptStatements*.
- Firefox по умолчанию ограничивает продолжительность выполнения 10 секундами; значение этого предела хранится в настройках броузера (доступных при вводе строки `about:config` в адресную строку) в параметре *dom.max_script_run_time*.
- Safari по умолчанию ограничивает продолжительность выполнения 5 секундами; это значение нельзя изменить, но имеется возможность отключить это ограничение, открыв меню Develop (Разработка) и выбрав пункт Disable Runaway JavaScript Timer (Отключить Runaway JavaScript Timer).
- В Chrome отсутствует отдельная настройка, ограничивающая продолжительность выполнения сценариев. Вместо этого броузер полагается на универсальную систему определения аварийных ситуаций.

- Opera не ограничивает продолжительность выполнения сценариев и будет продолжать выполнять программный код на JavaScript, пока он сам не завершится, тем не менее благодаря архитектурным решениям, реализованным в Opera, это не влияет на стабильность системы в процессе выполнения сценариев.

Когда продолжительность выполнения сценария превысит установленный предел, броузер выведет свое диалоговое окно независимо от наличия обработчиков ошибок в странице. Это переводит проблему в область проблем удобства использования, потому что большинство пользователей Интернета не являются техническими специалистами и потому не смогут верно оценить смысл сообщения об ошибке и правильно выбрать вариант ответа (остановить сценарий или позволить ему продолжить работу).

Если сценарий способен стать причиной вывода такого диалога в какомлибо из броузеров, это означает, что он тратит слишком много времени на решение задачи. Это также говорит о том, что в процессе выполнения сценария броузер слишком долго не сможет откликаться на действия пользователя. Разработчик не имеет возможности обработать ситуацию, вызывающую вывод диалога, – отсутствует возможность определить факт его вывода и, как следствие, отсутствует возможность исправить проблемы, которые могут возникнуть в этом случае. Очевидно, что самый лучший способ обойти ограничения времени выполнения сценария состоит в том, чтобы не приближаться к этим ограничениям.

Слишком долго – это сколько?

Тот факт, что броузер позволяет сценарию выполняться в течение определенного времени, вовсе не означает, что вы тоже должны позволять ему это. На самом деле, чтобы у пользователя создавалось благоприятное впечатление, время работы сценария должно быть намного меньше предела, устанавливаемого броузером. Брендан Эйч (Brendan Eich), создатель языка JavaScript, как-то сказал: «[сценарий], выполняющийся несколько секунд, наверняка делает что-то неправильно...»

Если несколько секунд – это слишком долго, то какая продолжительность выполнения JavaScript-сценария считается приемлемой? Как оказывается, даже одна секунда – это слишком долго. Длительность одной операции, выполняемой JavaScript-сценарием, не должна превышать 100 мс. Это число было получено в ходе исследований, проведенных Робертом Миллером (Robert Miller) в 1968 году.[1] Самое интересное, что Якоб Нильсен (Jakob Nielsen), известный специалист в области удобства и простоты использования, в своей книге «Usability Engineering»

[1] Miller, R. B., «Response time in man-computer conversational transactions», Proc. AFIPS Fall Joint Computer Conference, Vol. 33 (1968), 267–277. Доступна по адресу: *http://portal.acm.org/citation.cfm?id=1476589.1476628*.

(Morgan Kaufmann, 1994) отметил[1], что это число не изменилось со временем и было подтверждено исследованиями Xerox-PARC в 1991 году.[2]

Нильсен отмечает, что если интерфейс откликается на действия пользователя в течение 100 мс, у пользователя будет создаваться ощущение «непосредственного манипулирования объектами пользовательского интерфейса». Любой интервал времени, превышающий 100 мс, будет создавать ощущение оторванности от интерфейса. Поскольку обновление пользовательского интерфейса невозможно во время выполнения программного кода на JavaScript, у пользователя не будет создаваться ощущение участия в управлении интерфейсом, если продолжительность выполнения кода будет составлять более 100 мс.

Дополнительная сложность заключается в том, что во время выполнения JavaScript-сценариев некоторые броузеры даже не добавляют в очередь задания по обновлению пользовательского интерфейса. Например, если щелкнуть на кнопке в то время, когда выполняется некоторый программный код на JavaScript, броузер может не добавить в очередь задание отобразить кнопку в нажатом состоянии или выполнить JavaScript-обработчик щелчка на кнопке. В результате появится ощущение неотзывчивости, или «подвисания», пользовательского интерфейса.

Такое поведение наблюдается во всех броузерах. Во время выполнения сценария пользовательский интерфейс не обновляется в ответ на действия пользователя. Задания вызова JavaScript-обработчиков, порожденные в это время в результате действий пользователя, помещаются в очередь и затем выполняются по порядку, когда длительная операция будет завершена. При этом задания обновления пользовательского интерфейса в ответ на действия пользователя, произведенные в течение этого времени, просто пропускаются, потому что предпочтение отдается динамическим аспектам страницы. То есть если во время выполнения сценария произвести щелчок на кнопке, она никогда не будет отображена в нажатом состоянии, однако ее обработчик `onclick` будет выполнен.

Internet Explorer может пропускать задания выполнить JavaScript-код, порождаемые в ответ на действия пользователя, чтобы обеспечить выполнение только двух одинаковых действий, произведенных подряд. Например, если во время выполнения сценария выполнить четыре щелчка на кнопке, обработчик события `onclick` будет вызван только два раза.

Несмотря на то что в подобных случаях броузеры пытаются действовать логично, все эти особенности поведения разрушают положитель-

---

[1] Доступна по адресу: *www.useit.com/papers/responsetime.html*.

[2] Card, S. K., G.G. Robertson, and J.D. Mackinlay, «The information visualizer: An information workspace», Proc. ACM CHI'91 Conf. (New Orleans: 28 April–2 May), 181–188. Доступна по адресу: *http://portal.acm.org/citation.cfm?id=108874*.

ные впечатления пользователя. Поэтому лучше не допускать подобных ситуаций и ограничить время выполнения любой операции 100 мс. Измерения длительности операций следует производить в самом медленном броузере, который предстоит поддерживать (описание инструментов, позволяющих измерять производительность JavaScript, можно найти в главе 10).

Использование таймеров

Несмотря на все старания, иногда сложность операции не позволяет завершить ее выполнение в течение 100 мс. В таких случаях желательно организовать передачу управления главному потоку выполнения, чтобы обеспечить возможность обновления пользовательского интерфейса. То есть приостановить выполнение JavaScript-сценария и дать пользовательскому интерфейсу шанс обновиться, прежде чем продолжить работу сценария. В этом вам могут помочь JavaScript-таймеры.

Основы таймеров

В сценариях на языке JavaScript таймеры создаются вызовом функций `setTimeout()` или `setInterval()`, принимающих одинаковые аргументы: функцию, которую требуется вызвать, и время (в миллисекундах), которое должно пройти перед ее вызовом. Функция `setTimeout()` создает таймер, срабатывающий только один раз, а функция `setInterval()` создает таймер, срабатывающий неограниченное количество раз через равные интервалы времени.

Особенности взаимодействия таймеров с главным потоком выполнения делают их удобным инструментом деления продолжительных операций на короткие фрагменты. Вызов функции `setTimeout()` или `setInterval()` сообщает интерпретатору JavaScript, что он должен приостановить работу на указанное время и затем добавить в очередь задание выполнить JavaScript-код. Например:

```
function greeting(){
    alert("Hello world!");
}

setTimeout(greeting, 250);
```

Этот программный код добавит в очередь задание вызвать функцию `greeting()` через 250 мс. До этого момента будут выполняться все остальные задания обновления пользовательского интерфейса и выполнения JavaScript-кода. Имейте в виду, что второй аргумент определяет время, через которое указанное задание должно быть добавлено в очередь, и необязательно совпадает со временем, через которое это задание будет выполнено; это задание, как и любое другое, будет ожидать в очереди, пока не будут выполнены все другие задания, уже находящиеся в очереди. Рассмотрим следующий пример:

```
var button = document.getElementById("my-button");
button.onclick = function(){

    oneMethod();

    setTimeout(function(){
        document.getElementById("notice").style.color = "red";
    }, 250);
};
```

Когда пользователь щелкнет на кнопке в этом примере, обработчик вызовет метод `oneMethod()` и установит таймер. Задание вызвать функцию, изменяющую цвет элемента `notice` и переданную таймеру, будет добавлено в очередь через 250 мс. Отсчет этих 250 мс начинается с момента вызова `setTimeout()`, а не с момента завершения внешней функции. То есть если функция `setTimeout()` была вызвана в момент времени n, задание вызвать функцию, переданную таймеру, будет добавлено в очередь в момент времени $n + 250$. Хронология этого процесса с момента щелчка на кнопке показана на рис. 6.3.

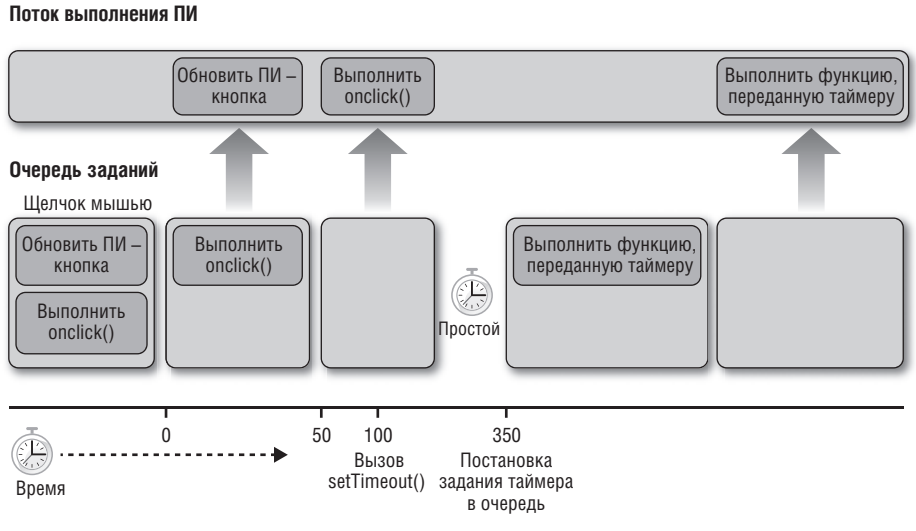

Рис. 6.3. *Второй аргумент функции setTimeout() определяет, когда следует добавить в очередь новое задание выполнить JavaScript-код*

Имейте в виду, что функция, переданная таймеру, не может быть вызвана до того, как завершит работу функция, в которой этот таймер был создан. Например, если в предыдущем примере уменьшить задержку таймера и после создания таймера добавить вызов другой функции, может сложиться ситуация, что таймер сработает раньше и добавит в очередь задание вызвать переданную ему функцию еще до того, как обработчик события `onclick` завершит работу:

```
var button = document.getElementById("my-button");
button.onclick = function(){

    oneMethod();

    setTimeout(function(){
        document.getElementById("notice").style.color = "red";
    }, 50);

    anotherMethod();
};
```

Если метод `anotherMethod()` будет выполняться дольше 50 мс, задание вызвать функцию, переданную таймеру, будет добавлено в очередь еще до того, как обработчик `onclick` завершит работу. В результате функция будет вызвана практически сразу же после завершения обработчика `onclick`, без ощутимой задержки. Эта ситуация изображена на рис. 6.4.

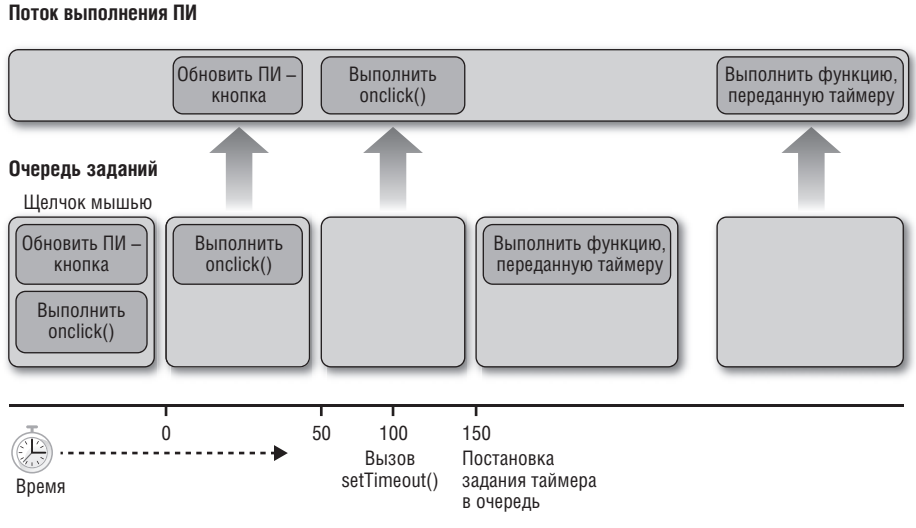

Рис. 6.4. *Задание, добавленное таймером, выполняется практически сразу, если функция, в которой была вызвана функция setTimeout(), выполняется дольше, чем интервал срабатывания таймера*

В любом случае создание таймера приводит к появлению паузы, в течение которой главный поток выполнения переключается с одного задания на другое. Как следствие сбрасываются все счетчики, используемые для слежения за ограничениями, включая таймер продолжительности выполнения сценария. Кроме того, внутри таймера вершина стека вызовов возвращается в исходное положение. Все это делает таймеры идеальным инструментом для работы со сценариями, выполняющимися продолжительное время, в любых броузерах.

Функция `setInterval()` практически идентична функции `setTimeout()`, за исключением того, что она организует многократное добавление в очередь задания выполнить JavaScript-код. Главное отличие состоит в том, что она не добавит новое задание в очередь, если задание, созданное тем же вызовом `setInterval()`, уже присутствует в очереди.

Точность таймера

Таймеры в JavaScript не отличаются высокой точностью и могут ошибаться на несколько миллисекунд в ту или иную сторону. Если при создании таймера указан интервал 250 мс, это еще не означает, что задание будет добавлено в очередь ровно через 250 мс после вызова `setTimeout()`. Все броузеры стараются обеспечить максимально высокую точность, но ошибки на несколько миллисекунд в ту или иную сторону нередки. Поэтому таймеры не считаются надежным средством измерения интервалов времени.

Разрешающая способность таймеров в Windows составляет 15 мс, поэтому задержка 15 мс, указанная при создании таймера, будет интерпретироваться как 0 или 15 мс в зависимости от момента последнего обновления системного времени. Использование задержек менее 15 мс в Internet Explorer может привести к блокировке броузера, поэтому минимальная задержка, которую рекомендуется устанавливать в нем, составляет 25 мс (которая будет интерпретироваться как 15 или 30 мс), чтобы гарантировать минимальную задержку 15 мс.

Такое ограничение минимальной величины задержки также позволит избежать проблем, связанных с разрешающей способностью таймеров в других броузерах и в других системах. Большинство броузеров показывает снижение точности срабатывания таймеров при использовании задержек менее 10 мс.

Обработка массивов с помощью таймеров

Одной из основных причин, приводящих к длительной работе сценариев, являются циклы, выполняющие слишком большой объем работы. Если даже после применения приемов оптимизации циклов, описанных в главе 4, не удается в достаточной мере уменьшить время выполнения, то следующим шагом является привлечение таймеров. Суть этого подхода состоит в том, чтобы разбить всю работу, выполняемую в цикле, на несколько частей. Типичные циклы реализуются по простому шаблону, как показано ниже:

```
for (var i=0, len=items.length; i < len; i++){
    process(items[i]);
}
```

Циклы с такой структурой могут выполняться очень долго из-за высокой сложности обработки данных в `process()`, размера массива `items` или

обоих факторов сразу. В моей книге «Professional JavaScript for Web Developers, Second Edition» (Wrox, 2009), я предлагаю ответить на два вопроса, чтобы определить, возможно ли выполнять цикл асинхронно с помощью таймеров:

- Должна ли обработка данных выполняться синхронно?
- Должны ли данные обрабатываться последовательно?

Если на оба вопроса получен ответ «нет», исследуемый цикл является отличным кандидатом на применение таймеров с целью разделить работу на фрагменты. Простейший шаблон реализации асинхронного цикла имеет следующий вид:

```
var todo = items.concat(); // создать копию оригинала
setTimeout(function(){

    // извлечь очередной элемент массива и обработать его
    process(todo.shift());

    // если еще остались элементы для обработки, создать другой таймер
    if(todo.length > 0){
        setTimeout(arguments.callee, 25);
    } else {
        callback(items);
    }

}, 25);
```

Основная идея этого шаблона состоит в том, чтобы создать копию оригинального массива и использовать ее как очередь обрабатываемых элементов. Первый вызов `setTimeout()` создаст таймер, обрабатывающий первый элемент массива. Вызов `todo.shift()` вернет первый элемент и одновременно удалит его из массива. Затем это значение будет передано функции `process()`. После обработки элемента проверяется наличие других элементов. Если в массиве `todo` еще остались элементы, создается другой таймер. Поскольку следующий таймер должен вызвать ту же функцию, что и предыдущий, в первом аргументе функции `setTimeout()` передается `arguments.callee`. Это значение указывает на анонимную функцию, в которой находится поток выполнения в данный момент. Если в массиве не осталось элементов для обработки, вызывается функция `callback()`.

Фактическая величина задержки каждого таймера в значительной степени зависит от конкретных условий. В общем случае для выполнения маленьких задержек рекомендуется использовать значение не менее 25 мс, потому что меньшие задержки оставляют слишком короткие интервалы времени для обновления пользовательского интерфейса.

Поскольку для реализации этого шаблона требуется больше программного кода, чем для обычного цикла, его полезно заключить в функцию. Например:

```
function processArray(items, process, callback){
    var todo = items.concat(); // создать копию оригинала

    setTimeout(function(){
        process(todo.shift());

        if (todo.length > 0){
            setTimeout(arguments.callee, 25);
        } else {
            callback(items);
        }
    }, 25);
}
```

Функция `processArray()`, реализующая шаблон, представленный выше, и позволяющая использовать его многократно, принимает три аргумента: обрабатываемый массив, функцию обработки каждого элемента и функцию обратного вызова, которую следует выполнить по окончании обработки массива. Ниже приводится пример использования этой функции:

```
var items = [123, 789, 323, 778, 232, 654, 219, 543, 321, 160];

function outputValue(value){
    console.log(value);
}

processArray(items, outputValue, function(){
    console.log("Done!");
});
```

В этом примере функция `processArray()` используется для вывода в консоль значений элементов массива и завершающего сообщения по окончании обработки. Благодаря заключению программного кода, выполняющего операции с таймером, в функцию, его можно многократно использовать в разных местах программы без необходимости повторять снова и снова.

Одним из побочных эффектов обработки массивов с помощью таймеров является увеличение времени обработки. Это объясняется тем, что после обработки каждого элемента управление передается главному потоку выполнения, из-за чего возникает задержка перед началом обработки следующего элемента. Однако это совершенно необходимо, чтобы не создавать у пользователя негативные впечатления, возникающие из-за блокировки броузера.

Деление заданий

Нередко одно большое задание можно разделить на последовательность более мелких заданий. Если единственная функция выполняется слишком долго, следует проверить возможность разделить ее на последова-

тельность более мелких функций, выполняющихся быстрее. Часто деление легко можно выполнить, если рассматривать каждую строку программного кода как некоторое атомарное задание, даже когда несколько строк программного кода сгруппированы для решения единой задачи. Некоторые функции легко можно разбить по функциям, которые они вызывают. Например:

```
function saveDocument(id){

    // сохранить документ
    openDocument(id)
    writeText(id);
    closeDocument(id);

    // обновить пользовательский интерфейс, чтобы сообщить об успехе
    updateUI(id);
}
```

Если эта функция будет выполняться слишком долго, ее легко можно разбить на последовательность более мелких шагов, организовав вызов отдельных методов с помощью таймеров. Добиться этого можно, добавив каждую функцию в массив и использовав шаблон, напоминающий шаблон обработки массива из предыдущего раздела:

```
function saveDocument(id){

    var tasks = [openDocument, writeText, closeDocument, updateUI];

    setTimeout(function(){

        // выполнить следующее задание
        var task = tasks.shift();
        task(id);

        // проверить наличие других заданий
        if (tasks.length > 0){
            setTimeout(arguments.callee, 25);
        }
    }, 25);
}
```

Эта версия функции помещает необходимые методы в массив `tasks` и затем вызывает их по одному с помощью таймеров. По сути, это тот же самый шаблон обработки массивов, с той лишь разницей, что обработка элементов связана с вызовом функций, содержащихся в них. Как демонстрировалось в предыдущем разделе, этот шаблон можно заключить в функцию для многократного использования:

```
function multistep(steps, args, callback){

    var tasks = steps.concat(); // скопировать массив

    setTimeout(function(){
```

```
            // выполнить следующее задание
            var task = tasks.shift();
            task.apply(null, args || []);

            // проверить наличие других заданий
            if (tasks.length > 0){
                setTimeout(arguments.callee, 25);
            } else {
                callback();
            }
        }, 25);
    }
```

Функция `multistep()` принимает три аргумента: массив функций, массив аргументов для передачи функциям и функцию обратного вызова, которая должна быть выполнена по завершении. Ниже приводится пример использования этой функции:

```
function saveDocument(id){

    var tasks = [openDocument, writeText, closeDocument, updateUI];
    multistep(tasks, [id], function(){
        alert("Save completed!");
    });
}
```

Обратите внимание, что второй аргумент функции `multistep()` должен быть массивом, поэтому при ее вызове создается массив с единственным элементом `id`. Как и в случае с обработкой массивов, данную функцию лучше использовать, когда асинхронное выполнение задания не повлечет за собой ошибки в зависящем от него программном коде и когда иной способ его выполнения может негативно отразиться на впечатлениях пользователей.

Хронометраж выполнения программного кода

Иногда выполнение по одному заданию за раз оказывается неэффективным. Представьте массив из 1000 элементов, обработка каждого из которых занимает 1 мс. Если каждый элемент обрабатывать с помощью таймера и использовать задержку 25 мс, то общее время обработки массива составит $(25 + 1) \times 1000 = 26000$ мс или 26 секунд. А что если попробовать обрабатывать массив пакетами по 50 элементов с задержкой 25 мс между ними? Тогда общее время обработки составит $(1000/50) + + + 25 + + 1000 = 1500$ мс или 1,5 секунды, и пользователь не будет чувствовать задержек в обновлении интерфейса, потому что обработка одного пакета будет занимать всего 50 мс. Обработка пакетами обычно выполняется быстрее, чем обработка по одному элементу.

Помня о 100 мс как об абсолютном максимуме интервала времени, в течение которого допускается непрерывное выполнение программного кода на JavaScript, можно оптимизировать предыдущие шаблоны. Я реко-

мендую уменьшить это число вдвое и никогда не позволять JavaScript-сценариям непрерывно выполняться дольше 50 мс, чтобы надежно гарантировать, что продолжительность выполнения программного кода никогда не окажется рядом с границей, переход за которую может создавать негативные впечатления.

Продолжительность выполнения программного кода можно определить с помощью встроенного объекта `Date`. Таким способом выполняется большая часть работы по профилированию программного кода:

```
var start = +new Date(),
    stop;

someLongProcess();

stop = +new Date();

if(stop-start < 50){
    alert("Just about right.");
} else {
    alert("Taking too long.");
}
```

Поскольку каждый новый объект `Date` инициализируется текущим системным временем, хронометраж выполнения программного кода можно производить, периодически создавая новые объекты `Date` и сравнивая их значения. Оператор сложения (`+`) преобразует объект `Date` в числовое представление, благодаря чему исключаются все последующие арифметические преобразования. Этот же простой прием можно использовать для оптимизации предыдущих шаблонов, основанных на таймерах.

Метод `processArray()` можно дополнить пакетной обработкой элементов массива, добавив проверку времени:

```
function timedProcessArray(items, process, callback){
    var todo = items.concat(); // создать копию оригинала

    setTimeout(function(){
        var start = +new Date();

        do {
            process(todo.shift());
        } while (todo.length > 0 && (+new Date() - start < 50));

        if (todo.length > 0){
            setTimeout(arguments.callee, 25);
        } else {
            callback(items);
        }
    }, 25);
}
```

Дополнительный цикл `do-while` в этой функции проверяет время после обработки каждого элемента. При вызове функции таймером массив всегда будет содержать хотя бы один элемент, поэтому в данном случае цикл с постусловием подходит больше, чем цикл с предусловием. При выполнении в Firefox 3 эта функция обрабатывает массив из 1000 элементов, где `process()` является пустой функцией, за 38–43 мс; первоначальная версия функции `processArray()` обрабатывает тот же массив более 25000 мс. Хронометраж заданий перед разделением их на более мелкие фрагменты обеспечил весьма существенное ускорение.

Таймеры и производительность

Таймеры могут оказывать огромное влияние на рост производительности программного кода, но злоупотребление ими может иметь негативные последствия. Примеры в этом разделе используют последовательности таймеров так, что в каждый момент времени используется только один таймер, а новый создается только после срабатывания предыдущего. Такое использование таймеров не оказывает отрицательного влияния на производительность.

Проблемы с производительностью начинают проявляться, когда одновременно создается сразу несколько таймеров многократного срабатывания. Поскольку в броузере имеется всего один главный поток выполнения, таймеры начинают конкурировать друг с другом за время, необходимое для выполнения переданных им функций. Нейл Томас (Neil Thomas) из Google Mobile исследовал эту тему, измеряя производительность мобильного приложения Gmail для iPhone и Android.[1]

Томас обнаружил, что таймеры с большим периодом срабатывания – от одной секунды и больше – оказывают незначительное влияние на общее время отклика веб-приложения. Задержки таймеров в этом случае оказываются слишком большими, чтобы оказать отрицательное влияние на производительность главного потока выполнения, и потому с такими задержками без опаски можно использовать таймеры многократного срабатывания. Однако используя несколько таймеров многократного срабатывания с более короткими задержками (от 100 до 200 мс), Томас обнаружил, что мобильное приложение Gmail стало заметно менее отзывчивым и выполнялось медленнее.

Из исследований Томаса следует вывод, что в веб-приложениях необходимо ограничивать количество таймеров многократного срабатывания с короткими задержками. Вместо них Томас предлагает создать единственный таймер многократного срабатывания, выполняющий множество операций при каждом срабатывании.

---

[1] Полный отчет доступен по адресу *http://googlecode.blogspot.com/2009/07/gmail-for-mobile-html5-series-using.html*.

Фоновые потоки выполнения

В то время когда язык JavaScript только появился, в броузерах отсутствовала возможность выполнения программного кода за пределами главного потока выполнения. Прикладной интерфейс фоновых потоков выполнения (Web Workers API) изменил положение дел, предоставив интерфейс, посредством которого можно выполнять программный код, не отнимая время у главного потока выполнения. Определение Web Workers API первоначально входило в состав стандарта HTML 5, а затем было выделено в отдельную спецификацию (*http://www.w3.org/TR/workers/*). Поддержка фоновых потоков выполнения уже реализована в Firefox 3.5, Chrome 3 и Safari 4.

Фоновые потоки потенциально способны существенно повысить производительность веб-приложений благодаря тому, что каждый новый объект `Worker` порождает собственный поток выполнения, в котором выполняется программный код на JavaScript. Это означает, что программный код, выполняемый в фоновом потоке, не оказывает влияния не только на главный поток выполнения, но и на код, выполняемый в других фоновых потоках.

Окружение фонового потока выполнения

Поскольку фоновые потоки выполняются независимо от потока пользовательского интерфейса, они не имеют доступа к многочисленным ресурсам броузера. Одна из причин, почему выполнение JavaScript-сценариев и обновление пользовательского интерфейса производится в одном и том же процессе, состоит в том, что зачастую они тесно связаны друг с другом, и выполнение их не в том порядке может вызвать отрицательные впечатления у пользователя. Фоновые потоки могли бы стать источником множества ошибок в пользовательском интерфейсе, если бы имели возможность манипулировать деревом DOM из-за пределов главного потока выполнения, но этого не происходит, потому что каждый фоновый поток выполнения имеет собственное глобальное окружение, которому доступна только часть возможностей JavaScript. Окружение фонового потока выполнения включает в себя следующее:

- Объект `navigator`, имеющий всего четыре свойства: `appName`, `appVersion`, `userAgent` и `platform`.
- Объект `location` (тот же, что и в объекте `window`, за исключением того, что все его свойства доступны только для чтения).
- Объект `self`, ссылающийся на глобальный объект фонового потока.
- Метод `importScripts()` для загрузки внешних JavaScript-библиотек, используемых в фоновом потоке.
- Все объекты, предусматриваемые стандартом ECMAScript, такие как `Object`, `Array`, `Date` и другие.
- Конструктор `XMLHttpRequest()`.

- Методы `setTimeout()` и `setInterval()`.
- Метод `close()`, немедленно останавливающий работу фонового потока.

Поскольку каждый фоновый поток выполнения имеет собственное глобальное окружение, нельзя создать фоновый поток из любого программного кода JavaScript. То есть необходимо создать отдельный JavaScript-файл, содержащий только программный код, который будет выполняться в фоновом потоке. Чтобы создать фоновый поток, необходимо передать конструктору URL-адрес JavaScript-файла:

```
var worker = new Worker("code.js");
```

Эта инструкция создаст для указанного файла новый поток выполнения с новым окружением. Загрузка файла выполняется асинхронно, и новый поток не будет запущен на выполнение, пока файл не будет загружен и выполнен полностью.

Взаимодействие с фоновыми потоками выполнения

Взаимодействия между фоновым потоком выполнения и веб-страницей выполняются посредством интерфейса событий. Сценарий в веб-странице может передать данные фоновому потоку выполнения, вызвав его метод `postMessage()`, принимающий единственный аргумент с данными для передачи фоновому потоку. Для приема информации внутри фонового потока используется обработчик события `onmessage`. Например:

```
var worker = new Worker("code.js");
worker.onmessage = function(event){
    alert(event.data);
};
worker.postMessage("Nicholas");
```

Данные фоновому потоку будут переданы вместе с событием `message`. Обработчик события `onmessage` получит объект события со свойством `data`, содержащим отправленные данные. Отправить информацию обратно в веб-страницу фоновый поток может с помощью собственного метода `postMessage()`:

```
// внутри code.js
self.onmessage = function(event){
    self.postMessage("Hello, " + event.data + "!");
};
```

Получившаяся в результате строка будет передана обработчику события `onmessage` объекта `worker`. Такая система обмена сообщениями является единственным способом организации обмена информацией между веб-страницей и фоновым потоком выполнения.

С помощью метода `postMessage()` допускается передавать данные только определенных типов: элементарные значения (строки, числа, логические значения, `null` и `undefined`), а также экземпляры объектов `Object` и `Array`; данные других типов передавать не допускается. Данные допустимого типа сериализуются, передаются фоновому потоку или принимаются

от него и затем преобразуются в обычное представление. Хотя на первый взгляд создается впечатление, что объекты передаются непосредственно, тем не менее передаваемые экземпляры являются отдельными представлениями одних и тех же данных. Попытка передать данные неподдерживаемого типа приведет к возбуждению исключения.

Реализация фоновых потоков выполнения в Safari 4 позволяет передавать с помощью метода `postMessage()` только строки. Уже после появления этого броузера в спецификацию были внесены изменения, предусматривающие сериализацию данных при передаче и учтенные в реализации фоновых потоков в Firefox 3.5.

Загрузка внешних файлов

Загрузка внешних JavaScript-файлов в фоновом потоке выполняется с помощью метода `importScripts()`, который принимает один или более URL-адресов JavaScript-файлов, подлежащих загрузке. Вызов метода `importScripts()` блокирует работу фонового потока, то есть сценарий продолжит выполнение только после загрузки и выполнения всех файлов. Поскольку фоновый поток выполняется за пределами главного потока выполнения, эта блокировка никак не сказывается на отзывчивости пользовательского интерфейса. Например:

```
// внутри code.js
importScripts("file1.js", "file2.js");

self.onmessage = function(event){
    self.postMessage("Hello, " + event.data + "!");
};
```

Первая строка в этом примере подключает два JavaScript-файла, благодаря чему они будут доступны в контексте фонового потока.

Практическое использование

Фоновые потоки прекрасно подходят для выполнения продолжительных операций над простыми данными и данными, не связанными с пользовательским интерфейсом. На первый взгляд фоновые потоки имеют весьма ограниченную область применения, но многие веб-приложения содержат задачи обработки данных, для решения которых вместо таймеров с успехом можно использовать фоновые потоки выполнения.

Рассмотрим для примера синтаксический анализ длинной строки в формате JSON (синтаксический анализ данных в формате JSON рассматривается далее в главе 7). Допустим, что объем данных настолько велик, что их обработка занимает не менее 500 мс. Это слишком долго, чтобы позволить такому JavaScript-сценарию выполняться на стороне клиента, потому что он будет создавать негативное впечатление у пользователя. Данную задачу трудно разбить на части с помощью таймеров, поэтому

применение фонового потока выполнения в этой ситуации является идеальным решением. Следующий фрагмент демонстрирует реализацию этого решения в веб-странице:

```
var worker = new Worker("jsonparser.js");

// когда данные будут обработаны, будет вызван этот обработчик события
worker.onmessage = function(event){

    // структура JSON, переданная обратно
    var jsonData = event.data;

    // использование структуры JSON
    evaluateData(jsonData);
};

// передать длинную строку в формате JSON для обработки
worker.postMessage(jsonText);
```

Ниже демонстрируется программный код, выполняющий обработку строки в формате JSON в фоновом потоке выполнения:

```
// внутри jsonparser.js
// этот обработчик события вызывается при передаче данных в формате JSON
self.onmessage = function(event){

    // строка в формате JSON поступает в виде значения свойства event.data
    var jsonText = event.data;

    // обработать данные
    var jsonData = JSON.parse(jsonText);

    // отправить результаты обратно
    self.postMessage(jsonData);
};
```

Обратите внимание, что хотя вызов метода `JSON.parse()` займет 500 мс или больше, в данном случае нет необходимости предусматривать дополнительный программный код, который делил бы обработку на более мелкие фрагменты. Обработка будет производиться в отдельном потоке выполнения, поэтому она может продолжаться столько, сколько потребуется, не оказывая влияния на впечатления пользователя.

Страница передает строку в формате JSON фоновому потоку выполнения с помощью метода `postMessage()`. Фоновый поток получает строку в виде значения свойства `event.data` в своем обработчике события `onmessage` и обрабатывает ее. По завершении полученный объект JSON передается обратно странице с помощью вызова метода `postMessage()` внутри фонового потока выполнения. Этот объект в свою очередь поступает в обработчик события `onmessage` страницы в виде значения свойства `event.data`. Имейте в виду, что пока такой подход можно использовать только в Firefox 3.5 и выше, потому что реализации в Safari 4 и Chrome 3 позволяют передавать между страницей и фоновым потоком только строки.

Обработка длинных строк – это лишь один из множества примеров задач, для решения которых с успехом можно использовать фоновые потоки выполнения. В числе других подобных задач можно назвать:

- Кодирование и декодирование длинных строк
- Сложные математические вычисления (включая обработку изображений и видеофайлов)
- Сортировку больших массивов

Всякий раз когда обработка занимает более 100 мс, следует подумать, не является ли решение на основе фоновых потоков выполнения более предпочтительным, чем решение на основе таймеров. Это, конечно, во многом зависит от возможностей броузера.

В заключение

Выполнение сценариев на языке JavaScript и обновление пользовательского интерфейса производятся в рамках одного и того же процесса, поэтому в каждый конкретный момент времени может выполняться только одно задание. Это означает, что пользовательский интерфейс не может реагировать на действия пользователя, пока выполняется программный код на JavaScript, и наоборот. Для успешного управления главным потоком выполнения необходимо гарантировать, что программный код на JavaScript не будет выполняться настолько долго, что это будет замечено пользователем. Для этого необходимо иметь в виду следующее:

- Никакая операция в JavaScript-сценарии не должна выполняться дольше 100 мс. Более длительные операции будут вызывать заметные задержки реакции пользовательского интерфейса и создавать негативные впечатления у пользователя.
- Броузеры по-разному реагируют на действия пользователя, производимые во время выполнения JavaScript-сценариев. Но независимо от поведения броузера у пользователя складываются отрицательные впечатления, когда выполнение сценария продолжается слишком долго.
- Применение таймеров позволяет отложить выполнение программного кода на более поздний срок, что дает возможность делить продолжительные операции на последовательности более мелких заданий.
- В новых версиях броузеров появился новый инструмент – фоновые потоки выполнения, с помощью которых можно выполнять программный код на JavaScript за пределами главного потока выполнения и тем самым предотвратить блокирование пользовательского интерфейса.

Чем сложнее веб-приложение, тем более важным становится активное управление главным потоком выполнения. Никакой программный код нельзя признать настолько важным, чтобы он негативно влиял на впечатления пользователя.