Николай Мациевский / Евгений Степанищев / Глеб Кондратенко

Реактивные веб-сайты

Серия «Архитектор информационных систем»

Николай Мациевский
Евгений Степанищев
Глеб Кондратенко

Реактивные веб-сайты

Клиентская оптимизация в алгоритмах и примерах

Учебное пособие

Интернет-Университет
Информационных Технологий
www.intuit.ru

БИНОМ.
Лаборатория знаний
www.lbz.ru

Москва, 2010

УДК 004.738.1(07)
ББК 32.973.202я7
М68

Мациевский Н.С.
М68 Реактивные веб-сайты. Клиентская оптимизация в алгоритмах и примерах: Учебное пособие / Н.С. Мациевский, Е.В. Степанищев, Г.И. Кондратенко — М.: Интернет-Университет Информационных Технологий: БИНОМ. Лаборатория знаний, 2010. — 336 с.: ил., табл. — (Серия «Архитектор информационных систем»).

ISBN 978-5-9963-0253-6

Издание консолидирует огромное количество прикладных советов, алгоритмов и пошаговых руководств, которые должны помочь начинающим специалистам, уже знакомых с областью клиентской оптимизации, глубже разобраться в тонкостях ускорения сайтов. Для профессионалов книга станет справочником лучших практик внедрения, основанном на опыте известных мировых специалистов.

УДК 004.738.1(07)
ББК 32.973.202я7

По вопросам приобретения обращаться:
«БИНОМ. Лаборатория знаний»
Телефон (499) 157-1902, (499) 157-5272,
e-mail: lbz@aha.ru, http://www.lbz.ru

ISBN 978-5-9963-0253-6

ОГЛАВЛЕНИЕ

Введение

Дорогой читатель, спасибо тебе за неустанный интерес к клиентской оптимизации, имеющей такое большое значение при оценке качества веб-проекта, для продвижения сайта и увеличения доходности интернет-подразделения любой компании.

Первая книга о клиентской оптимизации, «Разгони свой сайт» (http://speedupyourwebsite.ru/), вызвала многочисленные (все положительные) отклики, и это вдохновило нас на написание продолжения. К слову, мы рекомендуем прочесть первую книгу тем, кто еще этого не сделал. Это поможет глубже понять материал, изложенный в данной книге, легче вникнуть в суть излагаемых здесь концепций.

Книга, которую ты сейчас держишь в руках, призвана пролить свет на те стороны клиентской оптимизации, которые остались в тени после выхода первой книги. Мы надеемся, что она будет интересна как клиентским оптимизаторам со стажем, так и начинающим специалистам, желающим расширить свой кругозор.

Книга «Реактивные веб-сайты» содержит большое количество теоретического материала о клиентской оптимизации, но акцент в значительной мере сделан на описании прикладных методов оптимизации, а также на их квинтэссенции — автоматизации. Большая часть материала этой книги посвящена именно внедрению клиентской (и частично серверной) оптимизации при разработке веб-сайтов, а почти вся седьмая глава — приложению для автоматического ускорения сайтов, Web Optimizer (http://www.web-optimizer.ru/).

Кроме того, в этой книге мы немного заглянули в будущее и постарались описать те аспекты производительности, которые будут актуальны буквально через год-другой, когда большинство браузеров станут настолько быстрыми, что обычные методы оптимизации потеряют свою первоначальную эффективность. О перспективах оптимизации рассказывается в шестой главе.

Наконец, практическое приложение (оно приведено в восьмой главе) получилось достаточно объемным и охватывает множество текущих систем, применяемых для разработки сайтов малой и средней сложности. Это позволит использовать данное издание в полной мере как справочник с пошаговым руководством к действию: что и как нужно сделать, чтобы сайт работал «быстрее молнии».

На этих страницах опубликован первоклассный материал от профессиональных специалистов по ускорению сайтов и прикладным техникам оптимизации. Это знаменательно, что данное издание наконец-то увидело свет и ты держишь его в своих руках, читатель!

Об авторах

Данная книга не смогла бы охватить всех заявленных тем, если бы не авторский коллектив, вложивший все лучшее в это издание. Каждый из авторов внес существенный вклад в создание финальной версии этой книги.

Николай Мациевский, основатель и генеральный директор инновационной компании «ВЕБО», главной целью которой является разработка решений для оптимизации производительности веб-сайтов. Николай является одним из лучших специалистов в области клиентской и серверной производительности. Именно благодаря ему увидели свет сотни ценных русскоязычных статей, посвященных производительности веб-сайтов, первая книга по клиентской оптимизации «Разгони свой сайт», а также приложение для автоматического ускорения сайтов Web Optimizer.

Евгений Степанищев, сотрудник компании Яндекс подготовил две потрясающие главы на тему прикладного сжатия текстовой информации и оптимизации изображений, а также раздел шестой главы, посвященный клиентским хранилищам. Его познания в области графических спецификаций оказались настолько велики, что позволили предсказать несколько оптимизационных техник задолго до их появления.

Глеб Кондратенко, сотрудник компании Acronis, подготовил главу с обзором методов клиентской оптимизации, — прекрасное подспорье для тех, кто еще не знаком с основами данного технологического направления. Помимо этого Глеб свел воедино структуру книги, определил ее формирование. Этот поистине титанический труд позволил выпустить весь нижеизложенный материал в слаженной концепции.

Благодарности

В первую очередь авторы выражают свою благодарность Сергею Чикуенку за консультирование и часть материалов по оптимизации изображений (особенно PNG), Игорю Сысоеву за консультирование по nginx и его материалы по проблемам поддержки сжатого контента в браузерах и прокси-серверах, Виталию Харисову за неоценимую помощь в прояснении вопросов быстродействия отрисовки страниц в различных браузерах и применения CSS-логики.

Также, благодаря тесному сотрудничеству с Русланом Синицким (aka sirus, http://fullajax.ru/#:developers) удалось создать уникальный инструмент для автоматического создания кроссбраузерных изображений

в формате `data:URI` (подробнее об этом рассказывается в четвертой главе). Ольга Абанова (http://www.getincss.ru/) поделилась ценным материалом о конкурентной загрузке стилей, а Денис Абушаев (http://mearion.blogspot.com/) — своими советами касательно ускорения загрузки приложений на Ruby on Rails.

Отдельно хочется отблагодарить авторов разделов «Разгоняем Drupal» (Елену Цаплину), «Разгоняем Joostina» (Николая Кирша) и «Разгоняем jQuery» (Олега Смирнова), которые самостоятельно собрали весь материал и позволили читателям получить прикладные советы по оптимизации данных систем и библиотек «из первых рук».

Это просто замечательно, что коллектив высокопрофессиональных авторов смог предоставить действительно интересный материал, который будет полезен широкому кругу читателей, — и донести его в максимально доступном формате.

Как ускорить работу сайта для посетителя

С каждым годом Интернет растет вширь и вглубь. Увеличивается пропускная способность каналов, пользователи переходят с коммутируемого доступа на безлимитный. Сайты становятся больше по размеру, больше по наполнению и сложнее во взаимодействии. Размеры загружаемых файлов при этом увеличиваются многократно, а время ожидания пользователей не уменьшается.

За последние 5 лет средний размер веб-страниц вырос втрое (по данным исследования Akamai), а за последний год — в полтора раза (по данным webo.in). При этом каждая страница использует в среднем по 60 объектов, что крайне негативно сказывается на общем времени загрузки. Только порядка 5-10% от общего времени загрузки приходится на серверную часть. Все остальное составляет именно клиентская архитектура.

Что обычно видит пользователь, заходя на ваш сайт? И как долго он это видит? 75% посетителей уйдут после 10 секунд. При этом наиболее характерным временем ожидания будет 4 секунды: если за это время сайт загружается у 90% пользователей, то вы счастливый владелец быстрого интернет-ресурса.

Однако и здесь дорога каждая миллисекунда. Недаром высоконагруженные проекты типа Google, Amazon, Flickr, Netflix, Яндекс, вКонтакте и Одноклассники так серьезно подходят к вопросу скорости загрузки сайтов. За каждым потерянным моментом времени кроется определенная сумма денег. Это именно то место, где время тождественно равно деньгам.

В чем проблема?

Основное время при загрузке страницы уходит именно на клиентскую часть. Серверные затраты обычно крайне малы и составляют от 50 до 500 мс. Среднему пользователю на самом деле абсолютно все равно, сколько страница будет создаваться на сервере, если он увидит ее через полсекунды. В этом случае фокус смещается именно на клиентскую, а не серверную оптимизацию.

Характер проблем варьируется от сайта к сайту. Иногда он заключается в особенности интернет-подключения основной массы пользователей ресурса (например, если широко используются модемы или GPRS). Иногда — в сложности самого сайта и неоправданном использовании ресурсов сети. Иногда — в неграмотном использовании клиентских технологий и большого количества разнородных решений. Но все эти проблемы можно решить.

Ключевые моменты оценки качества веб-проектов

Говоря о скорости загрузки, нельзя не обозначить ее роль в оценке технологического качества любого Интернет-проекта. При этом стоит обратить внимание и на следующие моменты (которые можно достаточно быстро проверить с помощью бесплатных инструментов).

- Совместимость сайта с браузерами.
- Соответствие стандартам.
- Семантика HTML-кода.
- Доступность сайта для пользователей.
- Скорость работы на стороне сервера.
- Скорость работы на стороне браузера.

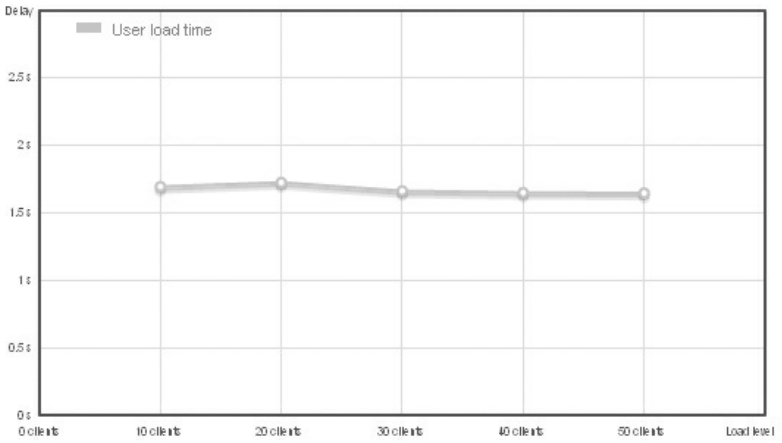
Скорость работы сайта

Скорость работы оказывает ключевое влияние на конвертацию посетителей в доход сайта. Особенно это характерно для интернет-магазинов. Как мы можем проверить эту скорость? Для серверной стороны существует инструмент host-tracker.com, с помощью которого можно

установить характерное время доступа к сайту из различных точек земного шара. Также он позволяет оценить характерное время создания страницы на сервере (если минимальное время доступа к сайту больше 1с, то уже стоит бить тревогу). Время создания страницы на сервере можно оценить и с помощью webo.in/my/action/timings/ (необходима регистрация).

Для проверки отказоустойчивости сайта стоит воспользоваться сервисом Load Impact (http://www.loadimpact.com/). Бесплатный анализ позволяет проверить до 50 одновременных посещений (каждое из которых может открывать несколько десятков страниц на сайте). Основным показателем устойчивости сайта к нагрузкам будет вид кривой, которая получится после проведения теста. Если график ровный или время открытия страницы несколько падает при росте посещаемости — значит, с сайтом все нормально. Если нет, то нужно принимать решение об оптимальном потоке посетителей, которых сайт сможет обслужить.

Оценка отказоустойчивости сайта. Источник: www.loadimpact.com



Для проверки качества скорости загрузки клиентской составляющей можно воспользоваться либо YSlow под Firebug для Firefox (оценка должна быть не менее 80, лучше всего не менее 90), либо бесплатным инструментом webo.in (простая оценка не ниже 70, лучше не ниже 80). Подробнее об инструментах для анализа клиентской производительности вебсайтов рассказано в первой главе.

На что влияет качество сайта

В первую очередь, на стоимость его поддержки и развития. Сайт, сделанный грамотно и по всем стандартам, гораздо проще развивать, поскольку времени на написание дополнительного кода при наличии хорошей, стандартной базы уходит значительно меньше. Это и масштабируемость проекта, когда разработчики предусматривают большой «запас прочности» для ежедневных посещений, а в пиковые часы сайт работает в штатном режиме, не требуя дополнительного масштабирования.

Скорость создания HTML-страниц и общее время загрузки очень сильно влияет на доступность сайта. Ключевым параметром здесь является «загрузка за 4 секунды» и более быстрая загрузка, чем у основных конкурентов. В отсутствии основных преимуществ на рынках с высокой конкуренцией скорость работы сайта может оказаться решающим фактором при принятии решения со стороны конечного пользователя.

Для решения проблем, связанных с медленной скоростью загрузки сайта, прежде всего нужно начинать с профессионального подхода при разработке какого-либо веб-приложения. Он обязательно должен опираться на использование веб-стандартов, в том числе и в области скорости загрузки и взаимодействия с пользователем.

Для полноценной оптимизации крупные компании разрабатывают специализированные решения. Однако даже в этом случае они опираются на многочисленный свод правил, описывающих наиболее оптимальное взаимодействие браузера с пользователем. Для среднего размера сайтов, использующих какое-либо стандартное решение для обслуживания контента, стоит обратить свое внимание на решения по автоматизации клиентской оптимизации. Подробнее они описаны в седьмой главе.

Наконец, при разработке сайтов стоит руководствоваться принципом «мягкой деградации», согласно которому нужно наращивать функциональность в зависимости от способности браузера, а не наоборот. Например, можно использовать самые передовые технологии при верстке, которые поддерживаются не очень большим числом браузеров, но должны будут поддерживаться значительной их долей, скажем, через год. Тогда определенные решения будут выглядеть безупречно у небольшого числа пользователей, у остальных — хорошо или приемлемо. Но с прогрессом Интернета число последних будет стремительно уменьшаться.

Также стоит упомянуть про возможность использования распределенной сети серверов для сведения времени ответа к минимуму. В этом могут помочь существующие системы распределенных вычислений и обработки запросов, такие как Amazon S3, Google Apps, Microsoft Azure. Также аналогичная система есть и на российских просторах. Это первая в России CDN — NGENIX (http://ngenix.net/). Подробнее о CDN рассказывается в пятой главе.

Глава 1. Обзор методов клиентской оптимизации

1.1. Клиентская оптимизация

Клиентская оптимизация — это оптимизация процесса загрузки клиентским приложением содержимого веб-страниц. Основная цель такой оптимизации — достижение максимальной скорости загрузки страниц сайта браузером клиента, ведь даже незначительные изменения времени загрузки могут иметь серьезные последствия для задачи, возложенной на сайт.

При построении высокопроизводительных сайтов должен присутствовать и клиентский, и серверный подход, они во многом дополняют друг друга. Главное отличие клиентского подхода состоит в том, что в качестве объекта оптимизации рассматриваются страницы сайта, получаемые браузером клиента, состоящие из HTML-документа, содержащего вызовы внешних объектов, а также сами внешние объекты (чаще всего это файлы CSS, файлы JavaScript и изображения).

Может показаться, что клиентская оптимизация является лишь составляющей частью серверной оптимизации, однако это не так. Различные технологические решения клиентской области сайта при одинаковой нагрузке на сервер могут обеспечивать совершенно разные характеристики клиентского быстродействия.

При исключении из рассмотрения всех факторов, относящихся к серверному программному обеспечению и каналу передачи данных, можно заключить, что увеличение скорости загрузки страницы на различных стадиях загрузки принципиально возможно за счет ограниченного количества методов. Об этих методах и пойдет речь далее.

1.2. Анализ веб-страниц

Большинство приведенных в книге методов оптимизации являются универсальными и могут быть применены практически в любом случае, на любом сайте. Но только выбор наиболее подходящего плана оптимизации может привести к наилучшему результату при решении каждой конкретной задачи.

Перед оптимизацией сайта необходим тщательный анализ его клиентской производительности, а также четко сформулированная цель оптимизации, ведь в подобном усовершенствовании важен только результат, а не процесс.

Процедуру анализа веб-сайта можно разделить на несколько основных стадий: анализ веб-страниц и их компонентов, анализ стадий загрузки веб-страниц и анализ характеристик браузеров, при помощи которых веб-страницы обычно загружаются.

1.2.1. Определение цели оптимизации

Целью клиентской оптимизации может быть решение подобных задач:

- достижение минимально возможного времени загрузки какой-либо конкретной страницы;
- достижение минимально возможного времени загрузки группы страниц, просматриваемых в произвольном порядке;
- обеспечение минимально возможного времени с момента запроса страницы до момента появления у пользователя возможности просматривать страницу и взаимодействовать с ней.

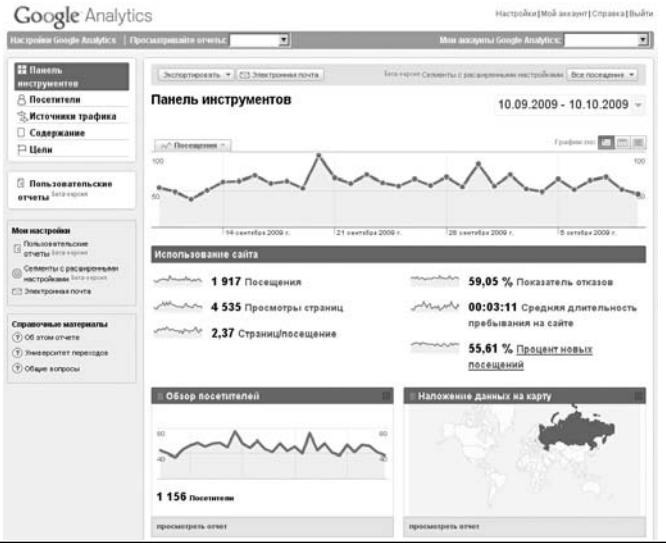
Это далеко не полный перечень возможных целей. Иногда и вовсе требуется достигать компромисса и выбирать между несколькими взаимоисключающими вариантами оптимизации. В таких ситуациях лучше иметь максимум возможной информации о ваших веб-сайтах и их посетителях.

Определить список «критических» страниц, на которых необходим максимальный эффект оптимизации, можно при помощи систем сбора и анализа статистики. Необходимо также учитывать назначение и специфику оптимизируемого сайта или сервиса.

Как правило, оптимизация требуется на главной странице сайта и других страницах с высокой посещаемостью, но это не всегда так. В качестве примера можно привести страницы оформления заказа на коммерческом сайте. На них может приходить лишь 5% от общего числа посетителей сайта, однако если они будут загружаться слишком медленно, посетители могут так и не стать клиентами.

Google Analytics (http://www.google.com/analytics/)

Рис. 1.1. Внешний вид сервиса Google Analytics



Google Analytics — один из лучших среди бесплатных сервисов для сбора и анализа статистики. С его помощью можно узнать о посетителях сайта почти все: страницы, с которых они переходили на сайт, время и длительность посещений, наиболее посещаемые страницы и последовательности посещений, параметры программного и аппаратного обеспечения и т. п.

Сервис работает по тому же принципу, что и большинство Интернет-счетчиков: специальный код устанавливается на всех страницах сайта и регистрирует каждое посещение, собирая все данные о нем.

Яндекс.Метрика (http://metrika.yandex.ru/)

Относительно молодой, но активно развивающийся русскоязычный сервис для оценки посещаемости сайтов и анализа поведения пользователей на нем. Позволяет получить детальную информацию об источниках перехода на сайт, числе возвратов, просматриваемом содержимом, географии и демографии посещений, программных и аппаратных характеристиках компьютеров пользователей.

Для сбора всей упомянутой выше информации достаточно лишь установить определенный код на всех страницах анализируемого сайта.

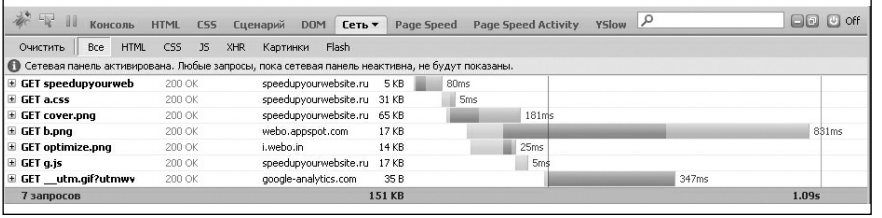
1.2.2. Анализ заголовков, компонентов и стадий загрузки страницы

Firebug (http://getfirebug.com/)

Одним из наиболее популярных среди веб-разработчиков средств для анализа и разработки веб-страниц является дополнение Firebug для браузера Firefox.

Панель Net в дополнении Firebug позволяет получить весьма детальную диаграмму загрузки страницы. По диаграмме можно определить стадии загрузки страницы, понять порядок загрузки объектов и выяснить, какие объекты блокируют, замедляют загрузку страницы. Кроме того, на этой панели можно получить детальную информацию о размере и заголовках самого документа, а также всех загруженных внешних объектов.

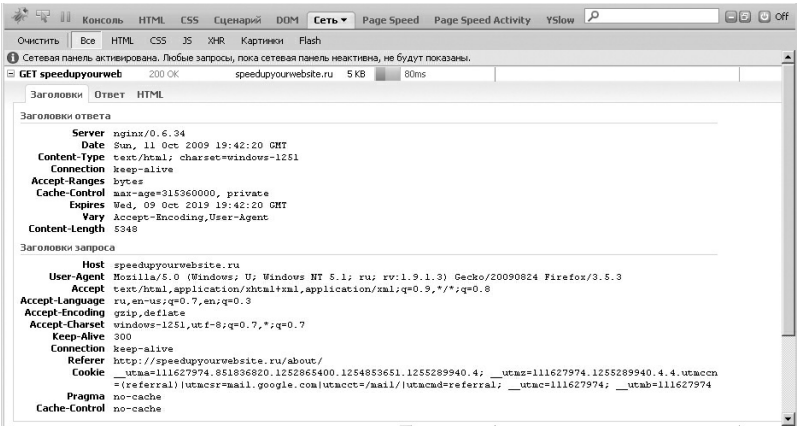
Рис. 1.2. Внешний вид панели Net дополнения Firebug для Firefox



Firebug также позволяет изменять DOM-дерево и CSS-свойства страницы без ее перезагрузки, сразу отражая результат изменений на странице, а также предоставляет обширные возможности для отладки и профилирования кода JavaScript. Все эти возможности являются прекрасным подспорьем во время работ по оптимизации сайта.

Стоит заметить, что аналогичные Firebug инструменты существуют во всех широко распространенных браузерах. В браузере Safari схожей

Рис. 1.3. Информация о заголовках открытой страницы на панели Net дополнения Firebug для Firefox

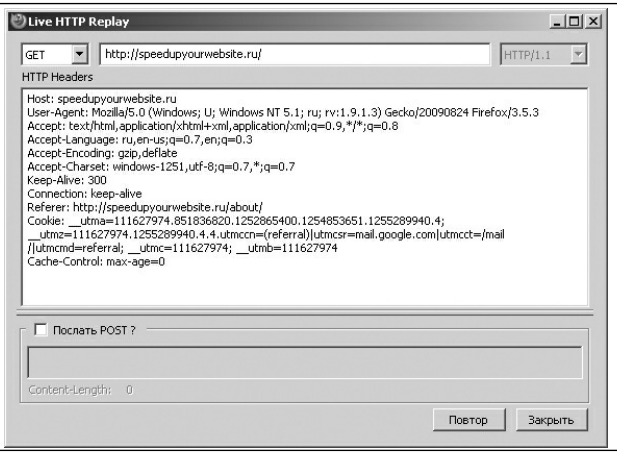


функциональностью обладает надстройка Web Inspector, в Opera — Dragonfly, в Internet Explorer — Developer Toolbar.

LiveHTTPHeaders (http://livehttpheaders.mozdev.org/)

Дополнение LiveHTTPHeaders для Firefox позволяет в режиме реального времени получать исчерпывающую информацию о пересылае-

Рис. 1.4. Режим ручной отправки запроса в дополнении LiveHTTPHeaders

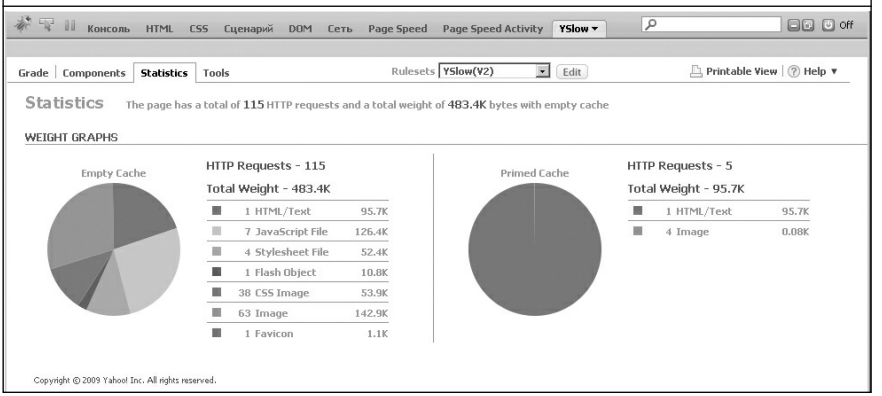


мых между браузером и сервером заголовках. Кроме того, в этом дополнении существует режим Replay, позволяющий отправлять на сервер произвольные запросы GET или POST, что часто бывает удобно при разработке.

YSlow (http://developer.yahoo.com/yslow/)

Дополнение YSlow для Firefox позволяет легко определить общее количество объектов, из которых состоит веб-страница, а также понять соотношения между объектами различного типа. В списке, содержащем перечень всех загруженных на странице объектов, предоставляется детальная информация по каждому такому объекту: размер, наличие сжатия, размер cookie, заголовки, время отклика и др.

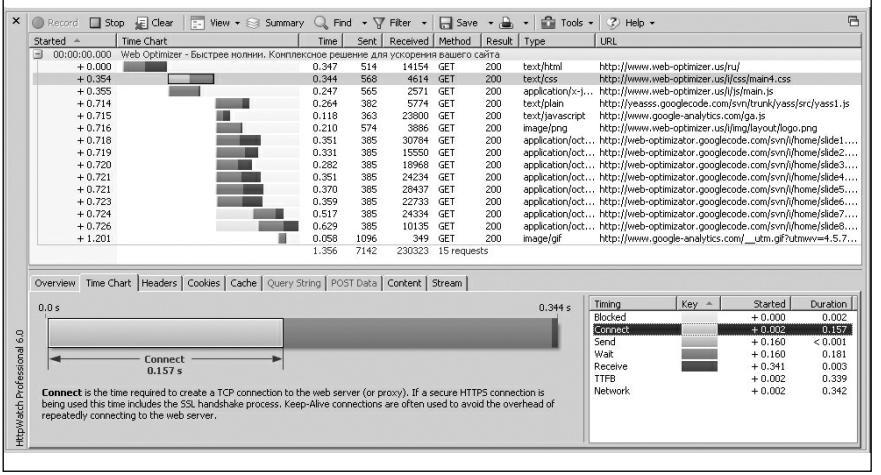
Рис. 1.5. Панель статистической информации о странице дополнения YSlow для Firefox



HTTPWatch (http://www.httpwatch.com/)

Более мощным средством для получения информации о составе и ходе загрузки веб-страниц является приложение HTTPWatch. Это приложение устанавливается в виде дополнений к браузерам Firefox и Internet Explorer и предоставляет более полную и более точную информацию, чем Firebug. В HTTPWatch поддерживаются любые виды сжатия, поддерживается протокол HTTPS, учитываются редиректы, есть возможность составления отчетов, фильтрации данных, просмотра любых заголовков, cookie, данных POST-запросов и многое другое. HTTPWatch можно использовать бесплатно, но только в базовой редакции, с достаточно ограниченным набором возможностей.

Рис. 1.6. Внешний вид дополнения HTTPWatch для Firefox



Hammerhead (http://stevesouders.com/hammerhead/)

Небольшое дополнение к браузеру Firefox под названием HammerHead позволяет имитировать многократную последовательную загрузку набора заданных страниц. Число попыток может быть установлено пользователем. Как результат работы дополнение отображает среднее время полной загрузки каждой испытуемой страницы. Дополнение позволяет очищать кэш после каждой загрузки для имитации обоих случаев: когда пользователь загружает страницу впервые или загружает ее повторно.

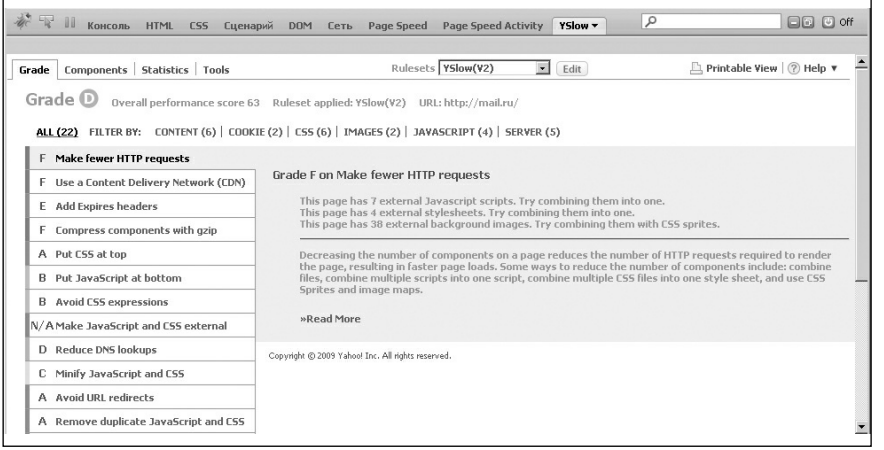
1.2.3. Комплексный анализ клиентской производительности веб-страниц

YSlow, PageSpeed (http://developer.yahoo.com/yslow/, http://code.google.com/speed/page-speed)

Дополнения YSlow и PageSpeed для Firefox позволяют получить интегральную оценку клиентской производительности любой загруженной браузером страницы, а также дают некоторые советы, следуя которым, можно улучшить производительность.

Разумеется, приведенные советы не могут покрыть всех возможных ситуаций, и иногда их выполнение может оказать даже отрицательное влияние на какой-либо из других параметров быстродействия вашего сайта. Однако более высокая интегральная оценка, которую дадут эти дополнения, будет в большинстве случаев соответствовать более оптимизированному сайту.

Рис. 1.7. Панель анализа страницы дополнения YSlow для Firefox



Веб-сервисы

Существует целый ряд веб-сервисов, позволяющих получить комплексную оценку клиентской производительности тестируемого сайта.

http://webo.in/ — По результату проверки предоставляется детальный план возможных действий по оптимизации со ссылками на тематические статьи, строятся диаграмма компонентов страницы и диаграмма загрузки. Для объектов тестируемой страницы подсчитывается возможный выигрыш за счет сжатия, минимизации и устранения дублирующихся файлов. Сервис является русскоязычным.

http://webpagetest.org/ — Сайт вычисляет время всех стадий загрузки и строит подробные диаграммы загрузки, позволяя при этом проводить серии из заданного количества тестов с определяемыми пользователем параметрами. История тестирования сохраняется.

http://site-perf.com/ — Этот сервис позволяет эмулировать большое число параметров загрузки: количество соединений на хост, пропускную способность канала и многие другие, включая даже процент потерянных пакетов. Отображается детальная статистика по всем серверам, с которых производилась загрузка.

1.2.4. Анализ характеристик браузеров

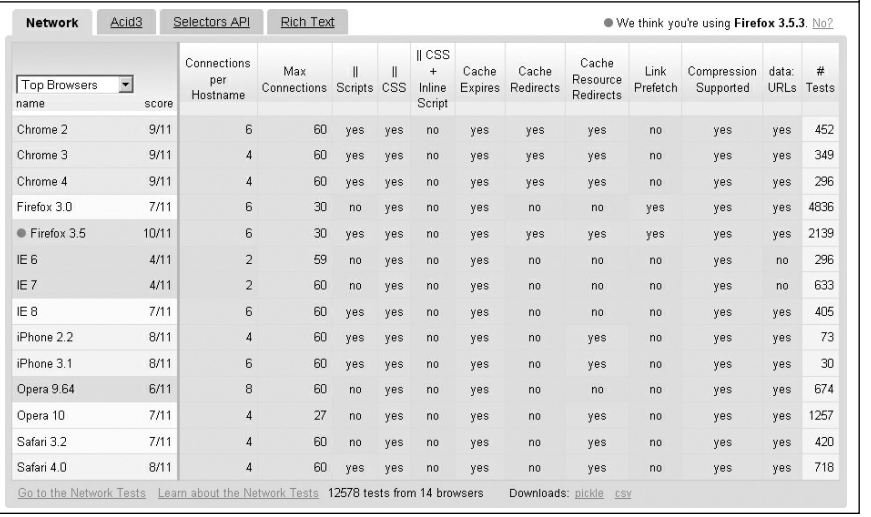
Клиентская оптимизация во многом основывается на некотором наборе известных свойств распространенных браузеров. К этим свойствам относится порядок загрузки браузером объектов, вызываемых на страни-

це, возможность параллельной загрузки этих объектов, максимальное число соединений, поддержка различных алгоритмов сжатия и пр. Для того чтобы узнать значения этих свойств в конкретном браузере, существует несколько приведенных ниже веб-сервисов.

Browserscope (http://www.browserscope.org/)

Этот сервис при помощи специально разработанных тестов позволяет определить все наиболее важные, с точки зрения клиентской оптимизации, характеристики вашего браузера. Кроме того, на веб-сайте сохраняются результаты всех ранее предпринятых пользователями тестов, благодаря чему можно найти информацию о поддержке тех или иных характеристик почти для любой версии любого браузера.

Рис. 1.8. Отчет о характеристиках наиболее распространенных браузеров сервиса Browserscope



Network | Acid3 | Selectors API | Rich Text — We think you're using **Firefox 3.5.3**. No?

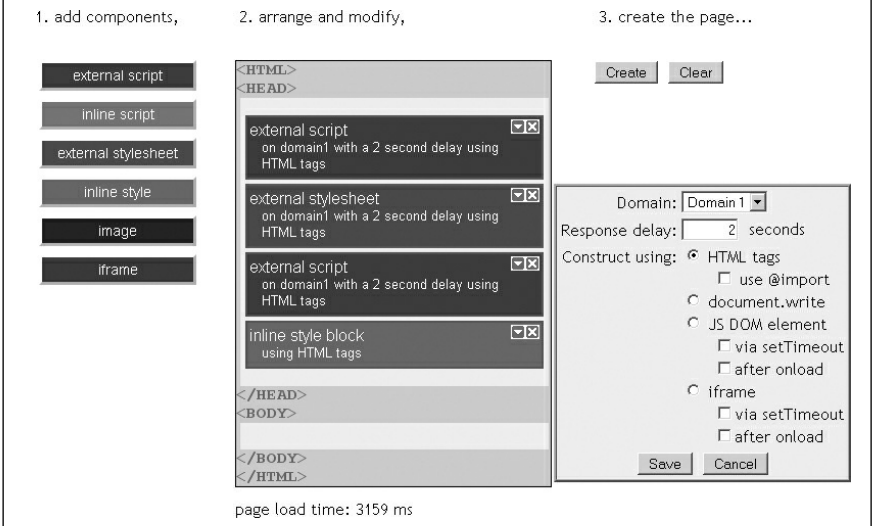
Top Browsers name	score	Connections per Hostname	Max Connections	|| Scripts	|| CSS	|| CSS + Inline Script	Cache Expires	Cache Redirects	Cache Resource Redirects	Link Prefetch	Compression Supported	data: URLs	# Tests
Chrome 2	9/11	6	60	yes	yes	no	yes	yes	yes	no	yes	yes	452
Chrome 3	9/11	4	60	yes	yes	no	yes	yes	yes	no	yes	yes	349
Chrome 4	9/11	4	60	yes	yes	no	yes	yes	yes	no	yes	yes	296
Firefox 3.0	7/11	6	30	no	yes	no	yes	no	no	yes	yes	yes	4836
Firefox 3.5	10/11	6	30	yes	yes	no	yes	yes	yes	yes	yes	yes	2139
IE 6	4/11	2	59	no	yes	no	yes	no	no	no	yes	no	296
IE 7	4/11	2	60	no	yes	no	yes	no	no	no	yes	no	633
IE 8	7/11	6	60	yes	yes	no	yes	no	no	no	yes	yes	405
iPhone 2.2	8/11	4	60	yes	yes	no	yes	no	yes	no	yes	yes	73
iPhone 3.1	8/11	6	60	yes	yes	no	yes	no	yes	no	yes	yes	30
Opera 9.64	6/11	8	60	no	yes	no	yes	no	no	no	yes	yes	674
Opera 10	7/11	4	27	no	yes	no	yes	no	yes	no	yes	yes	1257
Safari 3.2	7/11	4	60	no	yes	no	yes	no	yes	no	yes	yes	420
Safari 4.0	8/11	4	60	yes	yes	no	yes	no	yes	no	yes	yes	718

Go to the Network Tests | Learn about the Network Tests | 12578 tests from 14 browsers | Downloads: pickle csv

Cuzillion (http://stevesouders.com/cuzillion/)

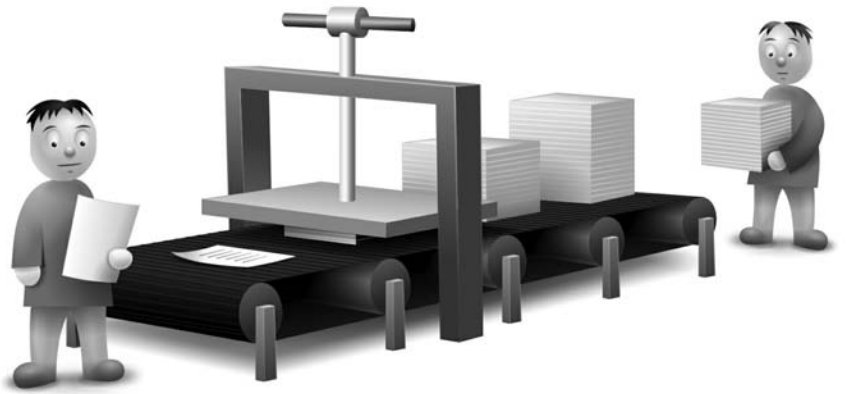
Этот сервис позволяет сконструировать, а затем загрузить в вашем браузере модель веб-страницы с произвольным количеством встроенных и внешних объектов, расположенных в выбранном вами порядке. В завершение каждой предпринятой попытки Cuzillion отображает время, затраченное на загрузку созданной страницы. Благодаря этому сервису легко можно оценить влияние изменений в структуре веб-страниц на скорость их загрузки в каждом конкретном браузере.

Рис. 1.9. Пример веб-страницы, созданной сервисом Cuzillion



1.3. Уменьшение размера загружаемых объектов

Очевидный способ увеличения скорости загрузки страницы — уменьшение размера загружаемых объектов. В ряде ситуаций можно без потерь содержания изменить состав HTML-документа, файлов CSS и JavaScript и тем самым уменьшить суммарный размер загружаемых пользователями страниц.



На высоконагруженных страницах, какими являются, например, главные страницы поисковых систем Google, Yahoo, Яндекс, возникают ситуации, когда каждый лишний байт страницы критичен для быстродействия.

1.3.1. Минимизация, обфускация и сжатие кода

Минимизация

Этим термином именуется процесс уменьшения объема кода за счет следующих операций:

- удаление избыточных пробелов, табуляций и переносов строк;
- удаление комментариев;
- удаление дублирующегося кода.

Минимизация применима к коду HTML, CSS и JS и в зависимости от размера и содержимого кода позволяет достичь результатов, близких к gzip-сжатию, уменьшать файлы до 30% от исходного размера, а иногда и более. При использовании же еще и gzip-сжатия предварительная минимизация позволяет увеличить итоговую степень сжатия в среднем на 3-5%.

В ситуациях, когда gzip-сжатие применить невозможно, например, когда CSS- и JS-код встроены в веб-страницу, минимизация — один из немногих оставшихся способов существенно уменьшить размер такой веб-страницы.

Стоит заметить, что после проведения минимизации результирующий код по-прежнему будет полностью совместим со всеми браузерами, т. к. он останется тем же корректным HTML-, CSS- или JS-кодом, в то время как gzip-сжатие до сих пор поддерживается не всеми пользовательскими браузерами. Кроме того, минимизированный код всегда распознается и обрабатывается браузером быстрее.

Обфускация

Так называется процесс, при котором JavaScript-код специальным образом запутывается для того, чтобы затруднить процесс его разбора и модификации. Обфускация может включать в себя те же операции, что и минимизация, а также:

- кодирование символов в различные форматы;
- изменение имен переменных и функций;
- добавление избыточного кода.

Ниже приведен фрагмент JS-кода до обфускации:

```
var Prototype = {
  Version: '1.6.1_rc3',
  Browser: (function(){
    var ua = navigator.userAgent;
    var isOpera = Object.prototype.toString.call(window.opera)
== '[object Opera]';
    return {
      IE:              !!window.attachEvent && !isOpera,
      Opera:           isOpera,
      WebKit:          ua.indexOf('AppleWebKit/') > -1,
      Gecko:           ua.indexOf('Gecko') > -1 &&
                       ua.indexOf('KHTML') === -1,
      MobileSafari:    /Apple.*Mobile.*Safari/.test(ua)
    }
  })(),
  BrowserFeatures: {
    XPath: !!document.evaluate,
    SelectorsAPI: !!document.querySelector,
    ElementExtensions: (function() {
      var constructor = window.Element || window.HTMLElement;
      return !!(constructor && constructor.prototype);
    })(),
    SpecificElementExtensions: (function() {
      if (typeof window.HTMLDivElement !== 'undefined')
        return true;
      var div = document.createElement('div');
      var form = document.createElement('form');
      var isSupported = false;
      if (div['__proto__'] && (div['__proto__'] !==
      form['__proto__'])) {
        isSupported = true;
      }
      div = form = null;
      return isSupported;
    })()
  },
  ScriptFragment: '<script[^>]*>([\\S\\s]*?)<\/script>',
  JSONFilter: /^\/\*-secure-([\s\S]*)\*\/\s*$/,
  emptyFunction: function() { },
  K: function(x) { return x }
};
```

А так этот же код выглядит после обфускации:

```
eval((function(x){var d="";var p=0;while(p<x.length)
{if(x.charAt(p)! ="`"`")d+=x.charAt(p++);else{var
l=x.charCodeAt(p+3)-28;if(l>4)d+=d.substr(d.length-
x.charCodeAt(p+1)*96-x.charCodeAt(p+2)+3104-l,l);else
d+="`";p+=4}}return d})("var Prototype={Version:\
"1.6.0.3\",Browser:{IE:!!(window.attachEvent&&navigator.userAgent
.indexOf(\"Opera\")===-1),` )!:` -@>-1,WebKit` 1:Apple` C\"/`
P\"Gecko` 7:` >!` I!`!_;KHTML`!w#,MobileSafari:!!`
EOmatch(/`!P!.*` K\".*`M\"/)}`#F$Features:{XPath:!!document.
evaluate,SelectorsAPI` 5(query` 5$,ElementExtensions:!!`$=#HTML`
8#,Specific` =.` o%create` :#(\"div\").__proto__&&`
\"C!==`24form` ?(},ScriptFragment:\"<s` ,![^>]*>([\\\\S\\\\s]*?)
</` 4\">\",JSONFilter:/^\\/\\*-secure-([\\s\\S]*)\\*\\/\\s*$/,
emptyFunction:f` \"#(){},K` %&x){return x;}};"))
```

Несмотря на подобный вид, такой код работает точно так же, как исходный, из которого он был получен.

В редких ситуациях при помощи обфускации можно достичь более эффективного сжатия, чем при помощи минимизации, но это отнюдь не является основной целью операции. В ситуации, когда в код добавляется большое количество избыточного кода, объем кода может значительно увеличиваться.

О наиболее популярных приложениях для минимизации и обфускации было рассказано в книге «Разгони свой сайт», во второй главе. Также часть подобных приложений рассматривается в начале седьмой главы.

Сжатие

Наиболее простым способом уменьшения размера загружаемых объектов с точки зрения внедрения является сжатие текстовых файлов при помощи технологии gzip. Эта технология, основанная на широко известном алгоритме DEFLATE, существует уже более 13 лет. По данным компании Google, среди всех использующихся на сегодняшний день браузеров 99,8% распознают gzip-сжатие HTML-документов и файлов CSS и JavaScript.

Со времен появления поддержки протокола HTTP/1.1 браузеры указывают в HTTP-запросах поддерживаемые типы сжатия, устанавливая заголовок `Accept-Encoding`:

```
Accept-Encoding: gzip, deflate
```

Если веб-сервер получает такой заголовок в запросе, он может применить сжатие ответа одним из методов, перечисленных клиентом. Отправляя ответ, сервер уведомляет клиента о том, каким методом сжимался ответ, при помощи заголовка `Content-Encoding`.

```
Content-Encoding: gzip
```

Для использования технологии gzip-сжатия обычно достаточно прописать несколько строк в конфигурационном файле сервера, и общая скорость загрузки может возрасти на десятки и даже сотни процентов — размер сжатых файлов обычно не превышает 20% от исходного размера.

С использованием gzip-сжатия нагрузка на браузеры пользователей практически не возрастает, поскольку восстановление сжатого файла малого размера (а на веб-страницах размеры текстовых файлов измеряются в килобайтах) происходит почти мгновенно. Временные затраты возможны на стороне сервера, но только в случае динамического сжатия, т. е. сжатия в реальном времени. В ситуации, когда динамическое сжатие оказывает недопустимо большую нагрузку на веб-сервер, следует применять статическое сжатие, т. е. сжатие и кэширование всех необходимых файлов только при их изменении.

Используя gzip-сжатие, важно убедиться в том, что оно отключено для изображений и других двоичных файлов. Поскольку эти файлы уже сжаты, а их размер обычно существенно превышает размеры типичных текстовых файлов, применение gzip-сжатия не принесет никакого выигрыша в клиентском быстродействии веб-страниц, а лишь увеличит нагрузку на сервер. Следует также обратить внимание на то, что сжатие эффективно только для файлов размером порядка одного килобайта и более. Сжатие файлов меньшего размера ощутимого результата, скорее всего, не принесет.

Достичь дополнительного выигрыша при сжатии файлов HTML и CSS можно также за счет применения стандартов верстки, при которых везде, где это возможно, используется один и тот же регистр, а все HTML-атрибуты, CSS-свойства и другие подобные элементы разметки упорядочиваются (чаще всего по алфавиту).

Сжатие текстовых файлов подробно рассмотрено во второй главе.

1.3.2. Оптимизация изображений

За счет использования подходящих графических форматов и эффективного сжатия без потерь суммарный размер страницы может быть уменьшен иногда на 50% и более.

Изображения, полученные с фотоаппаратов или сохраненные в некоторых графических редакторах, могут содержать много килобайт дополнительной информации, комментариев (т. н. метаданных), а также избыточную цветовую палитру.

Существует несколько графических форматов, поддерживаемых всеми современными браузерами: PNG-8, PNG-24, JPEG, GIF, ICO. Каждый из этих форматов позволяет в определенных ситуациях получить значительный выигрыш в размере по сравнению с другими форматами. Основные рекомендации для различных типов изображений приведены ниже.

Полноцветные изображения, изображения с градиентами

Для полноцветных изображений с богатой цветовой палитрой (фотографий, сложных градиентов и т. п.) следует использовать формат JPEG высокой степени качества. Необходимо помнить о том, что JPEG — формат сжатия с потерями, и чем выше степень сжатия, тем большее число артефактов появится на итоговом изображении.

Полупрозрачные изображения

В случаях, когда для верстки требуются полупрозрачные изображения, следует использовать формат PNG-24, поддерживающий альфа-каналы. Нельзя однако забывать о том, что браузер Internet Explorer 6 не поддерживает полупрозрачность в таких изображениях и для их корректного вывода следует применять фильтр AlphaImageLoader.

Изображения с ограниченной цветовой палитрой

Для изображений с ограниченной палитрой следует применять формат PNG-8. Этот формат, как и формат GIF, позволяет использовать прозрачность (не альфа-каналы), но в большинстве случаев превосходит GIF по качеству сжатия итогового файла. Достигается это за счет более совер-

шенной методики сжатия (фильтрации), которая охватывает и горизонтальные, и вертикальные повторения, а также хорошо работает с градиентами.

Анимированные изображения

Единственным кроссбраузерным форматом, позволяющим отображать анимацию в изображениях, является формат GIF. Однако уже в ближайшем будущем ему может составить конкуренцию развивающийся формат APNG. Подробнее об этом формате можно узнать в разделе 3.3.

Иконка веб-сайта (favicon.ico)

Для иконок веб-сайта существует специальный формат ICO, однако большинство современных браузеров могут использовать в качестве иконки изображение любого поддерживаемого ими формата. Более подробно о применении favicon.ico было рассказано в книге «Разгони свой сайт», во второй главе.

Наибольшего эффекта от оптимизации можно достичь, только используя наиболее подходящий формат для каждого случая. Нередко разделение сложных изображений (содержащих, например, полноцветное изображение с некоторым количеством мелкого текста и полупрозрачную рамку) на несколько отдельных изображений, накладывающихся одно на другое, может существенно уменьшить размер веб-страницы.

Удобной программой для ручной оптимизации статических изображений в Windows и Mac OS является программа Adobe Photoshop, для анимированных изображений — Adobe Fireworks. В операционных системах семейства Linux одним из наиболее подходящих приложений является Gimp.

О том, как можно автоматизировать оптимизацию изображений, рассказано в третьей главе.

1.3.3. Устранение избыточного кода

Оптимизация верстки

Для уменьшения размера кода следует использовать такие способы верстки, которые требуют минимум тегов HTML и правил CSS. Так, семантическая верстка с применением независимых блоков более предпочтительна, чем верстка вложенными таблицами с исполь-

зованием избыточных тегов. Подробнее о семантической верстке и ее преимуществах можно узнать в статьях http://pepelsbey.net/2008/04/semantic-coding-1/ и http://pepelsbey.net/2008/04/semantic-coding-2/.

Не стоит использовать в верстке атрибуты HTML и свойства CSS, значения которых подразумеваются по умолчанию, такие, например, как `target="_self"`.

Избыточного кода в CSS можно также избежать, приняв стандарт отображения типовых элементов на веб-страницах, таких как заголовки, параграфы, списки, ссылки и т. д. Один раз определив стиль оформления ссылки и параграфа, больше не придется описывать его для каждого нового блока.

Устранение встроенного в разметку кода

Суммарный объем кода можно также сократить за счет устранения встроенного на веб-странице CSS- и JS-кода. Множество одинаковых атрибутов `style=""` в HTML-тегах за счет использования классов в большинстве случаев можно заменить единственным, общим для всех элементов CSS-селектором, а множество JavaScript-обработчиков (например, обработчиков `onclick=""`, `onmouseover=""` и др.) — одним-единственным обработчиком. Изменить верстку и JavaScript-логику в подобных ситуациях, как правило, достаточно несложно.

Неиспользуемый код

Нередко на веб-страницах можно найти некоторое количество неиспользуемого кода, находящегося как в самом HTML-документе, так и во внешних файлах.

Время загрузки этих страниц увеличивается на время загрузки неиспользуемых внешних файлов из сети или из кэша браузера и на время, необходимое для разбора всех элементов DOM-дерева и CSS-правил, которые могут быть к ним применены. В случаях, когда размер веб-страницы и файлов ресурсов измеряется в сотнях килобайт, задержка может быть существенной.

Если в файлах CSS и JS, подключаемых на веб-странице, большой объем кода относится исключительно к другим страницам, следует перераспределить такой код по нескольким файлам, подключая их на страницах по необходимости. Для обнаружения неиспользуемого на странице CSS-кода можно воспользоваться дополнением Dust-Me Selectors для Firefox (https://addons.mozilla.org/ru/firefox/addon/5392).

Оптимизация cookie

Сохраняя в файлах cookie лишь идентификаторы данных, хранящихся на сервере, можно существенно уменьшить их размер. В идеальной си-

туации, чтобы для передачи cookie потребовался лишь один пакет, его размер должен быть не больше одного килобайта.

Сократить размер cookie также можно, гибко определяя содержащиеся в них поля. Если какие-то поля нужны лишь для ограниченного диапазона веб-страниц, нужно определять их только для этого диапазона, а не для всех страниц сайта.

Для статического контента (например, изображений, файлов CSS и JS) можно использовать отдельные домены, не передающие cookie вовсе.

1.4. Уменьшение количества запросов

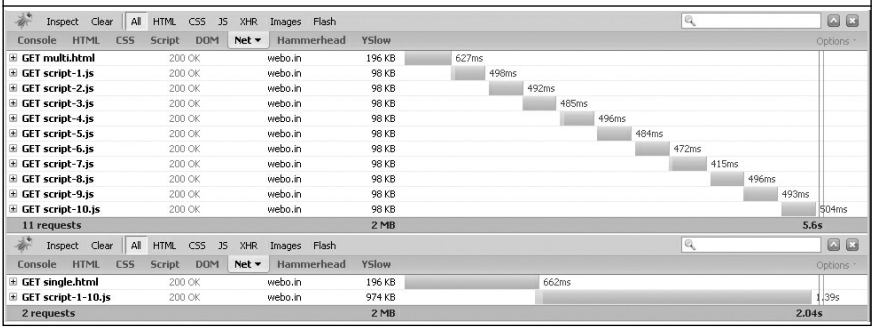
Размер HTML-документа обычно составляет порядка 10% от общего размера страницы. Остальные 90% занимают вызываемые со страницы внешние объекты (чаще всего это изображения, файлы CSS и JS).

Помимо непосредственной загрузки каждого внешнего объекта браузеру необходимо совершить целый ряд дополнительных действий:

- определить IP-адрес сервера по его доменному имени;
- установить новое соединение с этим сервером;
- отработать возможные редиректы;
- отправить запрос на сервер;
- дождаться ответа от сервера.

Таким образом, из-за отсутствия этих временных издержек один внешний объект всегда загружается быстрее, чем несколько объектов того же суммарного размера, загружающихся последовательно, а поскольку многие браузеры загружают внешние файлы JS строго последовательно,

Рис. 1.10. Влияние количества внешних объектов на скорость загрузки веб-страницы



количество этих объектов способно существенно повлиять на скорость загрузки веб-страницы.

Наибольший эффект от уменьшения количества запросов к серверу ощутят пользователи с низкой пропускной способностью канала и большим временем отклика от сервера — обычно это пользователи мобильных устройств и коммутируемых соединений.

1.4.1. Объединение текстовых файлов

Устранение фреймов

На веб-страницах не следует использовать фреймы. При необходимости применения их число должно быть минимальным.

Среди минусов фреймов: избыточные запросы к серверу, блокирование события onload, а также затруднения при поисковой индексации и сохранении адресов и состояний веб-страниц. Следует также заметить, что тег `iframe` исключен из стандартов XHTML 1.0 Strict и XHTML 1.1.

Использование фреймов, как правило, оправданно только тогда, когда требуется безопасно вставить блок какого-либо стороннего содержимого, например, рекламный блок. В большинстве же случаев можно избежать применения фреймов за счет разумного использования серверных скриптов и техник AJAX.

Объединение файлов CSS

Уменьшить количество запросов к серверу можно за счет минимизации количества вызовов CSS-файлов. Оптимальное количество таких вызовов — не более двух на страницу.

Не следует подключать на каждой веб-странице все использующиеся на сайте CSS-файлы, пусть тот код, который нужен на ограниченном числе страниц, вызывается только на них.

Разработчики, заботящиеся о доступности своих веб-страниц, часто предусматривают несколько файлов стилей для различных типов устройств просмотра веб-страниц. В этой ситуации в секции `<head>` может оказаться похожий набор вызовов:

```
<link type="text/css" rel="stylesheet" href="screen.css"
media=" screen" />
<link type="text/css" rel="stylesheet" href="handheld.css"
media="handheld" />
<link type="text/css" rel="stylesheet" href="print.css"
media="print" />
```

Уменьшить количество запросов в этой ситуации можно, объединив содержимое всех трех файлов в одном и используя для каждого типа устройств отдельное правило `@media`.

На этапе разработки, чтобы легче сопровождать код и исключить его избыточность, часто бывает удобно применять CSS-правило `@import` или вызывать в HTML-документе больше число CSS-файлов. В такой ситуации перед публикацией на сервере можно использовать инструменты для автоматического объединения CSS-файлов и подстановки кода, вызываемого свойством `@import`, чтобы уменьшить число запросов к серверу.

К сожалению, иногда для корректного отображения веб-страниц в старых версиях браузера Internet Explorer требуется отдельный набор CSS-правил. В таких ситуациях часто прибегают к использованию условных комментариев. Лучшим примером их применения является следующая конструкция, когда любой браузер запросит лишь один файл, предназначенный для него:

```
<!–[if gt IE 7]><!–>
<link rel="stylesheet" href="css/style.css"/>
<!–<![endif]–>
<!–[if lt IE 8]>
<link rel=stylesheet href="css/style.ie.css">
<![endif]–>
```

Объединение файлов JavaScript

Как и в рассмотренной выше ситуации с файлами CSS, на странице часто подключается несколько файлов JavaScript. Уменьшив их количество, можно значительно увеличить итоговую скорость загрузки страницы.

Весь код JS можно объединить в одном файле, загружаемом и кэшируемом единожды. Это лучшее решение в том случае, когда кода JavaScript на сайте относительно немного (порядка 50-100 килобайт в сжатом виде).

В ситуации, когда сайт представляет собой сложное веб-приложение и объем кода превышает 100-150 килобайт в сжатом виде, у объединения

всего кода в одном файле имеется отрицательная сторона: при запросе первой страницы пользователь неизбежно будет загружать часть кода, которую мог бы не загружать вовсе. Кроме того, в крупных веб-приложениях бывает не просто уследить за зависимостями модулей, — часто нужный код повторяется и, разумеется, загружается пользователем несколько раз.

Допустим, на сайте существует определенная последовательность страниц, посещаемых каждым новым пользователем, причем для каждой отдельной страницы требуются разные модули JS. Для страницы P_1 — модули F_1, F_2 и F_3, для страницы P_2 — модули F_1, F_3 и F_4, а для страницы P_3 — модули F_1, F_3, F_5 и F_6. Возможны три ситуации.

1. Объекты не объединены в один файл. При загрузке страницы P_1 тратится время на загрузку объектов F_1, F_2 и F_3, при загрузке P_2 — только объекта F_4 (F_1 и F_3 кэшируются после первой загрузки), а при загрузке P_3 — F_5 и F_6.
2. Все объекты объединены в один файл. При загрузке страницы P_1 тратится время на загрузку объектов F_1-F_6, но при загрузке всех остальных страниц внешние объекты не запрашиваются.
3. Объединены только модули, необходимые для текущей страницы. Для страницы P_1 загружаются объекты F_1, F_2 и F_3, для P_2 — F_1, F_3 и F_4, для P_3 — F_1, F_3, F_5 и R_6. В этом случае каждая отдельно взятая страница при пустом кэше будет загружаться быстрее, однако все три страницы подряд будут загружены медленнее, чем в двух описанных выше случаях, т. к. объекты F_1 и F_3 дважды повторно загрузятся вместе с другими объединенными объектами.

Выходом из положения может стать продуманное объединение модулей и выделение их ядра, т. е. набора модулей, используемых на большинстве часто загружаемых пользователями страниц. В приведенном примере таким ядром будут объекты F_1 и F_3.

В сложных ситуациях задача разбиения может быть легко формализована и решена, однако на практике это почти никогда не требуется и для нахождения лучшего варианта объединения достаточно рассмотреть описанные выше варианты.

Более подробно о методах автоматического объединения текстовых файлов рассказано в четвертой главе.

1.4.2. Объединение изображений

Технология объединения изображений для уменьшения числа запросов к серверу достаточно проста. Файл с несколькими объединенными в

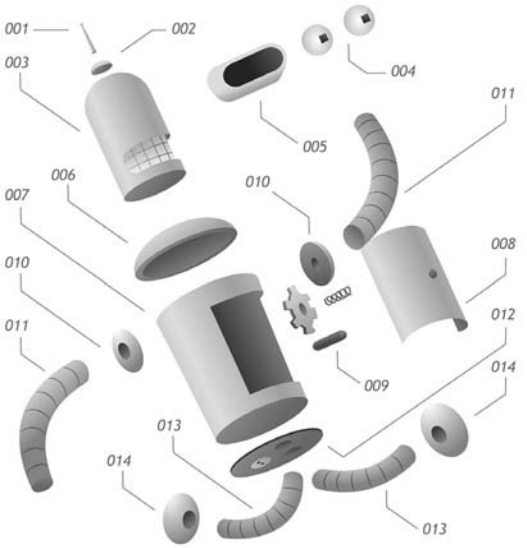
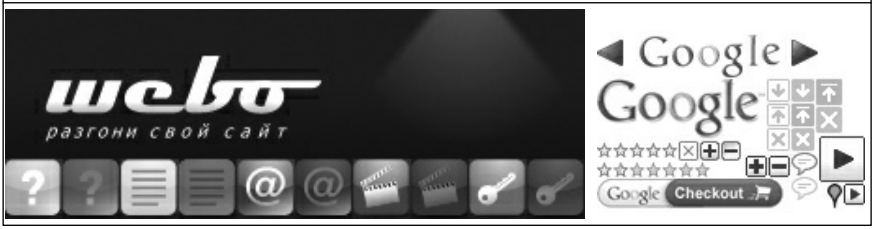


Рис. 1.11. Примеры объединения изображений в одном файле



определенном порядке изображениями (спрайт) загружается единожды, после чего в разных частях страницы отображаются те или иные его области. Возможно даже создание эффектов анимации при помощи объединенных изображений.

Хорошим примером может послужить метод использования единственного изображения для анимированной двухпозиционной кнопки. Верстка такой кнопки представляет собой обыкновенную ссылку с текстом:

```
<a class="button" href="#">Текст кнопки</a>
```

В стилях же, при помощи свойства background и псевдокласса :hover, задано положение фона для различных состояний кнопки:

```
a.button {
    background: url(/img/button.png) 0 0 no-repeat;
    display: inline-block;
    width: 100px;
    height: 20px;
}
a.button:hover {
    background-position: -100px 0;
}
```

Перед объединением изображений следует разбить их на следующие группы:

- изображения, повторяющиеся по всем направлениям (`repeat`);
- изображения, повторяющиеся по горизонтали (`repeat-x`);
- изображения, повторяющиеся по вертикали (`repeat-y`);
- изображения, не повторяющиеся по вертикали и горизонтали (`no-repeat`).

Эти группы нужны для того, чтобы исключить возможность появления остальных изображений из группы в области другого изображения. Так, изображение с фиксированной высотой, повторяющееся на странице по горизонтали (например, градиентная заливка фона), может находиться в группе аналогичных изображений, выше или ниже их, но никак не слева и не справа.

Если изображение всегда фиксированного размера и не повторяется по какому-либо направлению, его можно размещать в любом месте итогового файла.

Если же изображение, к примеру, использовано в списках, в роли маркера, находящегося в фоне элементов списка, необходимо обеспечить пустое пространство в объединенном изображении правее и ниже изображения маркера. В противном случае под текстом списка может оказаться другая часть спрайта.

Анимированные изображения лучше объединять отдельно, в зависимости от их размера и палитры.

Минусом применения спрайтов является усложнение верстки, однако автоматизация процесса создания спрайтов позволяет устранить этот минус; подробнее об этом написано в четвертой главе. Плюсом же является уменьшение как числа запросов к серверу, так и непосредственно размера страницы в ряде случаев более чем на 40%.

Не стоит забывать и про достаточно старый способ оптимизации, когда несколько расположенных рядом изображений объединяются в одно и при помощи тега `<map>` на этом изображении размечаются доступные для взаимодействия области.

1.4.3. Встраивание внешних объектов в код веб-страницы

Встраивание объектов CSS и JS

В ряде случаев фрагменты CSS- и JS-кода, а также изображения можно подставлять напрямую в HTML-документ.

Плюсы такой подстановки очевидны: количество запросов внешних объектов при загрузке страницы уменьшается до минимального количества. Минусом при использовании такого подхода на всех страницах сайта будет то, что пользователь лишится возможности кэшировать какие-либо объекты, вновь и вновь будет загружать их при просмотре страниц сайта.

Из этого очевидно, что подстановку внешних объектов в HTML-документ следует производить в следующих случаях:

- когда требуется исключительная клиентская производительность страницы, например, на главной странице сайта или других страницах с наивысшей посещаемостью;
- когда суммарная скорость загрузки всех внешних объектов меньше или приблизительно равняется затратам на поочередный запрос каждого из этих объектов по отдельности, т.е. в ситуациях, когда не возникнет уменьшения скорости загрузки страницы из-за невозможности кэширования внешних объектов;
- когда требуется полная автономность веб-приложения.

Встраивать код CSS следует внутрь тега `<style>`, расположенного в секции `<head>` страницы, а JavaScript-код — в теги `<script>` в конце документа, перед закрывающим тегом `<body>` либо также внутри секции `<head>`.

Следуя приведенным выше рекомендациям, необходимо избегать встраивания CSS- и JS-кода непосредственно в теги веб-страницы (т. е. в атрибуты style, onclick и т. д.). Это исключит дублирование кода на странице, а также упростит его сопровождение.

Подстановка изображений

Встраивать изображения прямо в HTML-документ можно, используя схему data:URI. По стандарту RFC 2397 такие URI предназначены для вставки небольших объектов как «непосредственных данных». Синтаксис должен быть следующим:

```
data:[<тип данных>][;base64],<данные>
```

В случае изображений необходимо указание mime-типа для них (например, image/gif). После него должно располагаться base64-представление бинарного файла с изображением. Ниже приведен пример такой подстановки:

```
<img src="
QMAAAAlPW0iAAAABlBMVEUAAAD///+l2Z/dAAAAM0lEQVR4nGP4/5/h/1+G/58ZDr
Az3D/McH8yw83NDDeNGe4Ug9C9zwz3gVLMDA/A6P9/AFGGFyjOXZtQAAAAAElFTkS
uQmCC" width="16" height="14" alt="Встроенное изображение" />
```

Такие изображения, внедренные в HTML-страницы, разумеется, кэшируются только вместе с самим HTML-документом, содержащим их. Однако учитывая то, что изображения могут быть встроены и в файлы каскадных стилей, можно добиться кэширования этих изображений путем объединения их в одном внешнем CSS-файле.

У данного подхода есть следующие минусы:

- изображения в base64-представлении имеют размер примерно на 33% больше исходного;
- необходимо вычислять и обновлять base64-представление изображения каждый раз, когда оно меняется;
- требуется отдельное решение для браузера Internet Explorer версии 7 и ниже.

Решение первой проблемы возможно за счет использования gzip-сжатия файлов CSS, в которых располагаются встроенные изображения. Вторая и третья проблемы легко решаются автоматическим преобразованием изображений в base64-представления при помощи серверных скриптов и созданием альтернативной версии кода для указанных браузеров Internet Explorer.

Описание способа автоматического встраивания изображений в код по схеме `data:URI` можно найти в четвертой главе.

1.4.4. Исключение избыточных HTTP-запросов

Уменьшение времени разрешения доменного имени

Перед тем как браузер сможет установить соединение с веб-сервером, он должен разрешить доменное имя, т. е., зная его, вычислить IP-адрес сервера. Результат может быть закэширован в браузере и операционной системе пользователя, и задержки при повторном открытии веб-страницы не возникнет.

Если же такой записи в кэше не существует, задержка на время

поиска IP-адреса может оказаться значительной и будет зависеть от доступности DNS-сервера, содержащего требуемую информацию (а иногда и от доступности цепочки таких серверов).

Наилучший способ уменьшить временные издержки, связанные с разрешением доменного имени — использовать наименьшее количество различных хостов для размещения внешних объектов веб-страниц, в особенности для тех объектов, которые требуются для первоначального отображения страницы.

Идеальным с точки зрения минимизации времени разрешения адреса DNS-сервера считается вариант, когда все объекты расположены на том же хосте, откуда была загружена веб-страница. Чем меньше используется хостов, тем больше вероятность того, что браузер сможет повторно использовать уже установленное соединение.

На практике же почти у всех браузеров существуют ограничения на количество одновременных соединений с одним хостом. Принимая во внимание это ограничение, наибольший выигрыш в скорости загрузки страниц можно получить, распределив загружаемые объекты по нескольким (4-6) хостам. Подробнее об особенностях параллельной загрузки объектов можно прочитать в пятой главе книги «Разгони свой сайт».

Уменьшение количества редиректов

Иногда возникает необходимость перенаправить браузер с одного адреса на другой. Причины чаще всего следующие:

- добавить косую черту к имени домена;
- предоставить пользователю документ, перемещенный на другой адрес;

- позволить пользователю обращаться к документам сайта даже если он ошибся в написании адреса (например, не набрал www в начале адреса);
- направить пользователя на другие домены первого уровня, основываясь на его географическом месторасположении и данных об используемом им языке;
- направить пользователя на определенные страницы в зависимости от того, авторизован он или нет;
- направить пользователя на страницы с другим протоколом (HTTP или HTTPS);
- отследить и сохранить действия пользователя и т. д.

Какой бы ни была причина, каждый редирект порождает дополнительный HTTP-запрос, занимающий определенное время. Поэтому для страниц, для которых скорость загрузки наиболее критична, число редиректов должно быть сведено к минимуму. Для этого необходимо:

- следить за тем, чтобы ссылки на веб-страницах не вели на адреса, где заведомо будет срабатывать редирект;
- избегать цепных (последовательных) редиректов;
- использовать минимальное количество альтернативных адресов для одних и тех же страниц, стараясь предоставить всем пользователям единственный актуальный адрес для каждой страницы;
- использовать внутренние перенаправления — функцию, доступную в большинстве веб-серверов;
- использовать средства отслеживания информации о пользователе, не основанные на редиректах;
- предпочитать серверные редиректы клиентским, которые могут быть заданы при помощи тега `<meta>` или JavaScript-обработчика. Редиректы, отправляющие браузеру код состояния 300, 301 или 302 и заголовок `Location`, обрабатываются браузером моментально, а при выполнении клиентских редиректов браузеру требуется дополнительное время на разбор полученной веб-страницы. Кроме того, некоторые браузеры могут кэшировать информацию о редиректах, тем самым ускоряя повторную загрузку ранее открытых веб-страниц.

1.4.5. Настройка кэширования

Браузеры и прокси-серверы обычно стремятся сохранить максимум информации в своих хранилищах, для того чтобы ускорить повторную загрузку ранее загруженных объектов. Важно помнить, что при этом возможна потеря актуальности представляемых данных, поэтому поли-

тика кэширования должна быть организована с учетом всех возможных ситуаций.

Кэширование — это один из наиболее мощных механизмов для уменьшения объема передаваемых по сети данных, притом внедряется этот механизм очень просто. Ниже приведено краткое описание наиболее значимых для кэширования заголовков.

Заголовок `Expires`

Когда HTTP-сервер отправляет объект (например, HTML-документ или изображение) браузеру, он может дополнительно с ответом отправить заголовок `Expires` с меткой времени. Браузеры обычно хранят ресурс вместе с информацией об истечении его срока действия в локальном кэше. При последующих запросах к тому же объекту браузер сравнивает текущее время и метку времени у находящегося в кэше ресурса. Если метка времени указывает на дату в будущем, браузер загружает ресурс из кэша, не запрашивая его с сервера. Формат должен быть строго следующим:

```
день недели(сокр.), число(2 цифры) месяц(сокр.) год
часы:минуты:секунды GMT
```

Заголовок Expires устанавливает время актуальности информации. Для ресурсов, которые не должны кэшироваться, его нужно выставлять в текущие время и дату (документ устаревает сразу же после получения), для форсирования кэширования его можно определять на достаточно далекую дату в будущем, например:

```
Expires: Mon, 27 Dec 2027 00:00:00 GMT
```

Заголовок `Cache-Control`

Заголовок `Cache-Control` определяет набор директив, относящихся непосредственно ко времени и специфике кэширования документа. Для запрета кэширования можно выставить его в следующее значение:

```
Cache-Control: no-store, no-cache, must-revalidate
```

Если же, наоборот, требуется сохранить ресурс в кэш браузера на продолжительный период времени, например, на год (60 * 60 * 24 * 365 секунд), нужно отправлять следующий заголовок:

```
Cache-Control: max-age=31536000
```

Заголовки `Last-Modified`, `If-Modified-Since`

Заголовок `Last-Modified` может отправляться сервером для того, чтобы передать браузеру информацию о дате последнего изменения документа. Дата должна задаваться в том же формате, что и в случае с заголовком `Expires`:

```
Last-Modified: Tue, 4 Aug 1995 04:58:08 GMT
```

При наличии такой информации в локальном кэше браузер может в следующем запросе отправить ее в заголовке `If-Modified-Since`:

```
If-Modified-Since: Tue, 29 Oct 1994 19:43:31 GMT
```

В случае если дата последнего изменения осталась прежней, сервер ответит кодом состояния 304 Not Modified и данные не будут отправлены повторно. В противном случае сервер передаст новую версию файла.

Данная схема позволяет экономить время, затрачиваемое на передачу данных, однако при ее использовании браузер все равно будет устанавливать соединение с сервером, чтобы узнать, имеется ли более новая версия.

Заголовки `ETag`, `If-None-Match`

Заголовок `ETag` является почти полной аналогией заголовка `Last-Modified` за тем исключением, что в качестве передаваемого значения может выступать произвольная строка. Заголовок отправляется сервером в следующем формате:

```
ETag: "any-type-of-tag-or-hash"
```

Впоследствии, для того чтобы сервер мог определить, является ли объект, находящийся в кэше браузера, точно таким же, как соответствующий объект на сервере, браузер может отправить следующий заголовок:

```
If-None-Match: "any-type-of-tag-or-hash"
```

И аналогично, если теги совпадают, сервер отвечает кодом состояния 304 Not Modified и данные не передаются повторно. В противном случае сервер передаст новую версию файла.

Форсированный сброс кэша

Если время кэширования при помощи заголовков `Expires` и `Cache-Control` установлено на несколько лет вперед и требуется сообщить клиентскому браузеру, что исходный объект изменился, можно воспользоваться двумя способами.

Можно обновить GET-строку запроса, например, используя номер версии или дату последнего изменения:

```
http://testdomain.com/global.css?v1
http://testdomain.com/global.css?20080901
```

А можно добавить номер версии или дату последнего изменения в само имя файла:

```
http://testdomain.com/global.v1.css
```

Во втором случае, для того чтобы исключить проблемы с локальными прокси-серверами, которые могут кэшировать файлы с GET-параметрами, и чтобы не создавать множество физических файлов, достаточно указать в конфигурации сервера правило: при запросах такого вида отдается каждый раз один и тот же файл.

В спецификации RFC-2616 HTTP-кэшированию посвящена целая глава. В ней подробно рассматривается, как работают все приведенные выше заголовки. Также о многих тонкостях использования кэширования более подробно рассказано в четвертой главе.

1.5. Увеличение скорости отображения веб-страниц

Даже когда веб-страница и все внешние объекты загружены на компьютер пользователя, браузеру по-прежнему требуется время для того, чтобы разобрать страницу, интерпретировать код HTML и CSS, выполнить код JavaScript. Принимая во внимание особенности работы браузеров на этом этапе, можно достичь существенно более высокой скорости загрузки страницы.

1.5.1. Оптимизация верстки

Разбирая полученный HTML-код, браузер строит дерево документа, содержащее все элементы страницы. Затем, отыскав все взаимосвязи

между элементами этого дерева и CSS-селекторами, относящимися к данной странице, он применяет к документу стили.

Если на веб-странице присутствует большое количество элементов или объем CSS-кода достаточно велик, страница может прорисовываться с ощутимой задержкой. Когда объем кода уменьшить уже невозможно, более высокой скорости загрузки можно достичь за счет эффективной верстки. Основные рекомендации к верстке следующие.

- Наиболее важное содержимое страницы должно находиться в самом начале HTML-документа. Так пользователи смогут начать взаимодействовать с этим содержимым раньше.
- Актуальные размеры изображений и ячеек таблиц, содержащих большое количество данных, должны быть явно заданы при помощи HTML-аттрибутов или CSS-свойств. Это позволит избавиться от лишних перерисовок веб-страницы. Например, когда браузер загрузит изображение и определит его размер, ему не потребуется обновлять макет веб-страницы, для изображения уже будет зарезервировано необходимое пространство. Кроме того, точно заданные размеры изображения избавят браузер от избыточной операции масштабирования изображения на лету.
- Следует отказаться от использования CSS-expressions для браузеров Internet Explorer. Expressions отрицательно влияют на производительность браузера и в большинстве ситуаций могут быть заменены более производительным JS-кодом, а иногда и вовсе альтернативной версткой. В ситуациях же, когда для требуемой функциональности сайта использования expressions не избежать, следует применять одноразовые expressions.
- Следует использовать быстродействующие селекторы идентификаторов и селекторы классов. Поскольку большинство браузеров анализируют селекторы справа налево, с виду простой

селектор `#header .menu li a` будет применяться дольше, чем аналогичный ему селектор `#header .menu-item`. В первом случае браузеру необходимо будет найти все ссылки на странице, проверить, находятся ли они в контексте элемента списка, элемента с классом `menu` и, наконец, элемента с идентификатором `header`. Второй вариант более предпочтителен, поскольку поиск элементов по классу и идентификатору выполнится существенно быстрее.

- Универсальные, дочерние, соседние селекторы, селекторы атрибутов, псевдоклассов и псевдоэлементов должны применяться только в тех ситуациях, когда это действительно необходимо. Все эти разновидности CSS-селекторов существенно более ресурсоемки, чем селекторы идентификаторов или классов.

1.5.2. Особенности отображения веб-страниц

При изменении ряда свойств элементов веб-страницы, а также в некоторых других ситуациях браузеры производят перерасчет геометрии и положения элементов, находящихся в потоке веб-страницы, после чего заново перерисовывают страницу или ее часть. Если веб-страница достаточно сложна, ее обновление будет заметно для пользователя и неизбежно вызовет задержку в его работе со страницей.

Вот неполный список действий, вызывающих в большинстве браузеров перерисовку страниц или их частей:

- изменение пользователем размера окна браузера или шрифта;
- добавление или удаление CSS-кода (как встроенного, так и во внешнем файле);
- манипуляции с элементами DOM-дерева;
- изменение следующих свойств элементов страницы: `class`, `font`, `display`, `visible`, `margin`, `padding`, `width`, `height`;
- активация псевдоклассов, таких, например, как `:hover`.

Учитывая эти особенности, можно свести к минимуму число перерисовок страниц, а для того чтобы скорость перерисовок была наивысшей, необходимо:

- обеспечить минимальную глубину и минимальный размер DOM-дерева, так как часто изменения свойств какого-либо элемента вынуждают браузер перерисовать не только сам этот элемент, но также родительские и дочерние элементы, а иногда и всю ветвь целиком;
- оптимизировать CSS-селекторы, обеспечить минимальный объем CSS-кода;

- создавать сложные элементы, анимировать их и производить другие подобные манипуляции над элементами, располагая их вне потока и используя для этого свойства `position: absolute` и `position: fixed`;
- изменять классы и стили у элементов на максимальной глубине DOM-дерева, что позволит браузеру перерисовывать лишь часть страницы;
- избегать использования таблиц в верстке, поскольку действия над их ячейками почти всегда вызывают необходимость перерасчитать и перерисовать всю таблицу целиком.

1.6. Оптимизация структуры веб-страниц

От структуры HTML-документа во многом зависит скорость загрузки страницы пользователем. Даже при одинаковом суммарном размере страниц и равном количестве внешних объектов две различные страницы могут загружаться за совершенно разное время. Причина в том, что отображение элементов частично загруженной страницы в большинстве браузеров осуществляется только после выполнения следующих шагов:

1. получения HTML-документа;
2. получения всех внешних объектов CSS, вызываемых в HTML-документе;
3. получения всех внешних объектов JavaScript, вызываемых в HTML-документе внутри тега `<head>`;
4. получения всех внешних объектов JavaScript в HTML-документе внутри тега `<body>`, расположенных в потоке выше выводящегося элемента.

1.6.1. Особенности загрузки браузерами внешних объектов

Особенности загрузки кода CSS

В большинстве браузеров отображение страницы будет приостановлено до тех пор, пока все внешние CSS-файлы не будут загружены. Кроме того, теги `<style>`, встречающиеся на странице, будут порождать частич-

ную или полную перерисовку страницы, иногда изменяя уже отобразившийся макет, с которым пользователь мог начать взаимодействовать.

Таким образом, более высокой скорости отображения страницы можно добиться, располагая в самом начале веб-страницы, в разделе `<head>`, все элементы `<link>`, содержащие вызовы файлов CSS-стилей, а также все встроенные стили, содержащиеся в тегах `<style>`.

Особенности загрузки кода JavaScript

Из-за того, что код JavaScript может изменять содержимое и макет веб-страницы, браузеры приостанавливают отображение элементов, следующих за JS-кодом, до тех пор, пока код не будет загружен и исполнен. Большинство браузеров при этом приостанавливают даже загрузку внешних объектов, следующих за таким JS-кодом. Однако если на момент начала загрузки JS-файла загрузка внешних объектов уже была начата, они будут загружены параллельно.

Подробный разбор возможной ситуации

В приведенном ниже примере на веб-странице загружаются два внешних файла CSS и JS.

```
<head>
<script type="text/javascript" src="script-1.js"></script>
<link rel="stylesheet" type="text/css" href="style-1.css" />
<script type="text/javascript" src="script-2.js"></script>
<link rel="stylesheet" type="text/css" href="style-2.css" />
</head>
```

При таком порядке следования в документе и при пустом кэше большинство браузеров будет вынуждено дождаться загрузки первого файла JavaScript, только потом начать загрузку следующих двух файлов CSS и JS и лишь по завершении загрузки второго файла JS загрузить последний файл стилей.

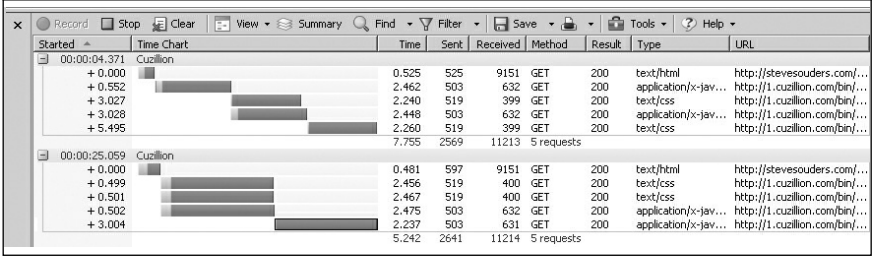
Всего лишь изменив порядок следования вызовов внешних объектов так, чтобы впереди были вызовы файлов CSS, можно получить ощутимый выигрыш в скорости загрузки.

```
<head>
<link rel="stylesheet" type="text/css" href="style-1.css" />
<link rel="stylesheet" type="text/css" href="style-2.css" />
<script type="text/javascript" src="script-1.js"></script>
<script type="text/javascript" src="script-2.js"></script>
</head>
```

В этой ситуации браузер может инициировать параллельную загрузку сразу трех внешних объектов — двух файлов стилей и первого файла JS. По окончанию их загрузки браузеру остается загрузить лишь один файл JavaScript, и итоговое время загрузки будет ощутимо ниже.

Как видно на приведенной ниже диаграмме, в первой ситуации загрузка каждого файла JavaScript блокирует получение остальных внешних объектов, создавая дополнительные задержки. Во второй же ситуации вместе с загрузкой первого файла JS происходит параллельная загрузка максимально возможного количества внешних объектов.

Рис. 1.12. Диаграмма загрузки внешних объектов при различном порядке их вызовов



Разумеется, экономия времени в каждом отдельном случае будет зависеть от размеров загружаемых объектов и их количества, однако пренебрегать этой особенностью не стоит.

Следует помнить, что данная особенность неактуальна в браузерах, поддерживающих параллельную неблокирующую загрузку внешних объектов. Перечень таких браузеров можно получить при помощи инструментов, описанных чуть ранее в этой главе.

1.6.2. Стадии загрузки страницы

Во всех браузерах можно выделить несколько основных стадий загрузки страницы. Рассмотрим эти стадии.

1. Предварительная загрузка — предварительное отображение частично загруженной страницы без части рисунков и, в ряде случаев, без интерактивности, без части JavaScript-сценариев.
2. Полная загрузка страницы — отображение полностью загруженной страницы со всеми изображениями, с полностью функционирующими интерактивными частями страницы, полностью функционирующей логикой JavaScript.

3. Пост-загрузка страницы — фоновая дозагрузка и кэширование каких-либо внешних объектов, которые могут потребоваться пользователю при использовании каких-либо интерактивных функций данной страницы или при переходе на другие страницы сайта.

Оптимизация стадии предварительной загрузки

Стадия предварительной загрузки завершается лишь после получения браузером всех внешних файлов CSS и JavaScript из секции <head> HTML-документа. Это справедливо для большинства браузеров.

Чтобы ускорить наступление предварительной загрузки, следует снизить число вызываемых в заголовке страницы файлов до минимума, применить к ним динамическое сжатие, или, для еще большего быстродействия — статическое сжатие. При использовании статического сжатия серверу не придется тратить дополнительное время на сжатие, он будет сразу готов отдать сжатый файл.

Загрузку файлов, не требующихся на стадии предварительной загрузки, имеет смысл перенести в стадию пост-загрузки.

Итогом первой стадии загрузки является доставленный и оформленный HTML-документ, с которым пользователь уже может взаимодействовать. Издержки на доставку всех файлов JavaScript должны быть сведены к минимуму, так как на этом этапе они только помешают, замедлив отображение основного содержимого страницы.

Оптимизация стадии полной загрузки

При правильной группировке загружаемых внешних объектов и за счет использования закэшированных браузером объектов эта стадия может наступать значительно быстрее.

Необходимо сконфигурировать веб-сервер так, чтобы при запросе любой другой страницы пришлось запрашивать минимум дополнительных объектов. Должен быть также продуман вопрос форсированного сброса кэша в ситуациях, когда это будет необходимо.

Также при запросе большого количества изображений браузер неизбежно столкнется с ограничением максимального количества соединений на хост, и для обхода этого ограничения необходимо настроить дополнительные серверы для выдачи статического содержимого. Число дополнительных хостов следует напрямую из числа статических файлов, поэтому надо определиться с ними на этапе автоматизации процесса публикации.

На этой же стадии можно использовать прием объединения изображений.

Оптимизация стадии пост-загрузки

К началу этой стадии у пользователя уже должна быть в распоряжении оформленная HTML-страница, на которой все ссылки и формы должны работать без JavaScript.

На этой стадии могут загружаться любые объекты, которые необходимы для этой или каких-либо других страниц.

При помощи специального обработчика JavaScript, который начинает работать на этой стадии, возможно к уже существующей функциональности страницы добавить расширенные возможности в виде подсказок, дополнительных данных с сервера, какой-либо анимации и т. д.

Этот обработчик может производить следующие операции:

- находить необходимые элементы дерева объектов страницы, требующие динамического вмешательства, подключая все необходимые обработчики, события и т.д.;
- проверять наличие необходимых JavaScript библиотек и по мере надобности загружать их, проверяя, чтобы один и тот же код не загружался дважды;
- загружать в фоне внешние объекты, используемые на других страницах.

В случае применения технологии `data:URI` для уменьшения количества вызываемых объектов на этой стадии необходимо загружать файл CSS, содержащий все используемые на странице изображения в формате base-64.

Об отложенной загрузке ресурсов, не требующихся на момент первого отображения страницы, можно прочитать в пятой главе, а также в седьмой главе книги «Разгони свой сайт».

Прогрессивный подход

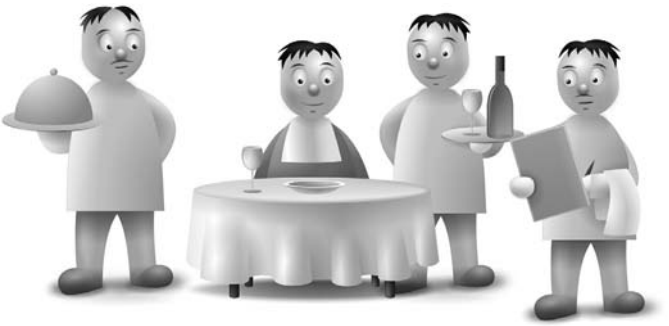
Во многих современных браузерах, в частности, в последних версиях браузера Safari, активно внедряется прогрессивная логика отображения страницы.

С одной стороны, браузер не ждет последовательной и полной загрузки всех блокирующих вывод страницы внешних объектов, а с другой — не отображает неформатированную веб-страницу. Сочетая преимущества сразу нескольких подходов, браузер определяет, в какой момент какую часть страницы можно отобразить, чтобы опыт пользователя был наилучшим.

Загрузка всех внешних объектов производится параллельно, и разбор содержимого не останавливается. При этом, если производится обращение к свойству стиля или макета, модуль JavaScript приостанавливает свою работу и ожидает загрузку всех незагруженных файлов CSS. Как только все стили загружены, выполнение сценария продолжается с того места, где оно было приостановлено.

1.6.3. Распределенное хранение контента (CDN)

У всех браузеров существует ограничение на количество соединений на один хост, находящееся в интервале от 2 до 8 соединений на хост. Для увеличения скорости загрузки внешних объектов можно применять распределенную систему хранения и доставки контента (CDN), организовав или арендовав систему хостов, находящихся на одном или нескольких физических серверах (географически распределенных, если это требуется). Часть хостов должны принимать и обрабатывать запросы пользователей, создавать и передавать результирующие HTML-документы. Остальные хосты должны использоваться только для передачи клиентом статических ресурсов. Благодаря такой схеме скорость доставки контента пользователям будет максимальной, в то же время нагрузка на хостинг веб-сайта может значительно снизиться.



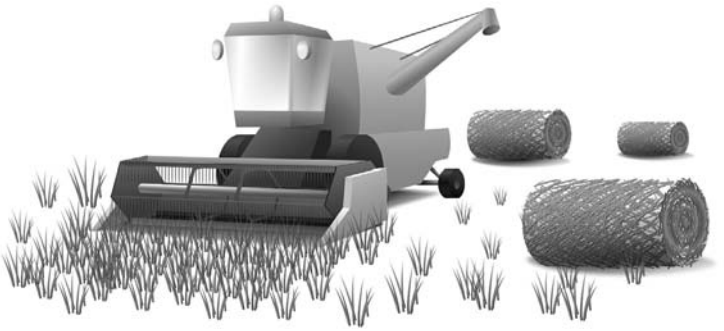
Необходимо заметить, что разделять по нескольким хостам имеет смысл только изображения и файлы CSS, так как файлы JavaScript почти во всех браузерах загружаются строго последовательно.

Создать подобную систему распределенного хранения можно либо при помощи автоматического балансировщика, либо установив альтернативные пути к внешним объектам вручную.

Большое число дополнительных хостов может увеличить временные затраты браузера на установление соединений, поэтому в наибольшем количестве ситуаций предпочтительно использование не более 5 дополнительных хостов (1 основной хост и 4 для параллельной загрузки кэшируемых объектов) без учета не контролируемых вами хостов, например рекламных. Это позволяет ускорить загрузку приблизительно на 60% в случае большого количества файлов.

Подробнее о системах распределенного хранения контента (CDN) рассказано в пятой главе.

Глава 2. Алгоритмизация сжатия текстовых файлов



Один из способов, позволяющий уменьшить размер передаваемых данных, — их сжатие. Сжатые данные занимают меньше места, следовательно, быстрее грузятся, и канал и веб-сервер быстрее освобождаются. Производители браузеров позаботились о том, чтобы передаваемые от сервера к клиенту данные можно было запаковать. Все современные браузеры поддерживают один или несколько алгоритмов сжатия данных.

2.1. Методы сжатия, поддерживаемые браузерами

Браузер в каждом запросе к серверу в поле «Accept-Encoding» может указать, какие методы сжатия он поддерживает. Сервер, отвечая на запрос, может выбрать один из указанных браузером методов и, высылая

сжатое тело ответа, указать в заголовке (в поле «Content-Encoding»), какой именно метод был выбран.

Вот, к примеру, поле «Accept-Encoding» браузера Opera 10.00:

```
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
```

Браузер указал, что поддерживаются два метода сжатия: deflate и gzip. Названия с приставкой «x-» оставлены для совместимости, это историческое название, «gzip» и «x-gzip» (так же как «compress» и «x-compress») эквивалентны. Метод identity («идентично») означает отсутствие сжатия, то есть данные передаются без изменений. Мы намеренно не рассматриваем остальные значения, подробнее на этом вопросе мы остановимся позже, в части про реализацию сжатия на стороне сервера.

На данный момент браузеры совокупно поддерживают следующие методы сжатия:

- gzip (x-gzip) —два метода gzip и deflate используют один и тот же алгоритм сжатия — DEFLATE (RFC 1951), использующий комбинацию алгоритма LZ77 и кодирования Хаффмана. В методе gzip (RFC 1952) перед сжатым потоком добавляется десять байт заголовка, а после — восемь байт, состоящие из контрольной суммы (CRC32) и длины несжатых данных;
- deflate (x-deflate) — представляет собой сжатый алгоритмом DEFLATE поток без заголовка и других метаданных;
- compress (x-compress) — в данном случае тело ответа такое же, как если бы оно было сжато UNIX-программой compress. Compress использует алгоритм LZC, являющийся реализацией LZW (Lempel—Ziv—Welch), с указателями переменного размера, как в алгоритме LZ78;
- bzip2 (x-bzip, bzip) — тело ответа совпадает с результатом работы программы bzip2, алгоритм более эффективен, чем compress и семейство DEFLATE, но работает значительно медленнее. В bzip2 применяются преобразование Барроуза-Уилера, MTF-преобразование (move-to-front) и кодирование Хаффмана;
- sdch (Shared Dictionary Compression Over HTTP) — относительно новый, предложенный компанией Google в сентябре 2008 года метод уменьшения избыточной информации в вебе. Основная идея — не передавать дважды одинаковые куски документа (например, шапку, «подвал» страницы, общие CSS- и JavaScript-файлы). Метод построен на алгоритме VCDIFF (RFC3284), ответ дополнительно может быть сжат любым другим поддерживаемым браузером методом сжатия (например, gzip).

Метод sdch сильно отличается от прочих и имеет смысл только для группы страницы, эффективность же остальных методов проще оценить на примере HTML-страницы, полученной склейкой первых страниц нескольких новостных изданий:

Без сжатия	gzip/deflate	bzip2	Compress
587828	132023	88263	222091

У архиваторов gzip и bzip2 есть параметр, управляющий эффективностью компрессии, его значение изменяется от единицы до девятки, и чем больше значение, тем эффективнее сжатие и тем больше времени оно занимает. В тесте, проведенном выше, этот параметр для обоих архиваторов был установлен в максимальное значение.

Худший результат типично показывает метод compress, по этой причине многие браузеры отказались от поддержки этого метода. Лучший результат — у bzip2, но этот метод, весьма требовательный к ресурсам, лишь недавно стал появляться в браузерах — на данный момент его поддерживают OmniWeb, w3m, lynx и ранние версии Google Chrome.

Эффективность	Gzip	bzip2
Исходный размер	189058 Б	189058 Б
1	3,333 мс / 44572 Б	40,282 мс / 32247 Б
2	3,519 мс / 42685 Б	43,716 мс / 29690 Б
3	4,713 мс / 42685 Б	43,765 мс / 29690 Б
4	5,274 мс / 39111 Б	44,213 мс / 29690 Б
5	6,526 мс / 37039 Б	43,704 мс / 29690 Б
6	8,036 мс / 36206 Б	43,814 мс / 29690 Б
7	9,098 мс / 35940 Б	43,934 мс / 29690 Б
8	12,87 мс / 35713 Б	43,725 мс / 29690 Б
9	14,319 мс / 35707 Б	45,129 мс / 29690 Б

На сайте языка программирования PHP, в комментариях к описанию функции gzcompress есть результаты, иллюстрирующие, насколько может зависеть затраченное время и эффективность сжатия архиваторов gzip и bzip2 от параметра, задающего эффективность компрессии, а также приводящие разницу в требованиях к ресурсам этих архиваторов.

Кажущееся странным поведение архиватора bzip2 с параметром, большим единицы (результат архивации не улучшается), объясняется очень просто: параметр от одного до девяти в этом случае задает размер блока — от 100 Кб до 900 Кб.

Что в точности означает этот параметр, для нас маловажно, существенно то, что если выбранный файл (исходный размер — 184,5 Кб) целиком помещается в блок, алгоритм bzip2 не улучшает результат при увеличении размера блока, но сжатие занимает больше времени.

Вывод — если вы настраиваете на сервере сжатие bzip2 в реальном времени, выбирайте размер блока чуть большим, чем средний размер страниц вашего сайта.

У gzip, как видно, зависимость однозначная, оптимальная стратегия в данном случае — установить такой параметр эффективности сжатия, который ваш сервер выдержит, с некоторым запасом.

Существуют также аппаратные решения, позволяющие эффективно сжимать трафик алгоритмом gzip со скоростью вплоть до 10 гигабит в секунду, например, платы расширения фирмы Comtech AHA Corporation.

2.2. Проблемы в браузерах, прокси-серверах и firewall

К сожалению, несмотря на давность, реализация сжатия в браузерах не обходится без проблем. Особенно печально, что проблемы есть даже в браузерах, выпущенных недавно относительно момента написания книги.

Некоторые браузеры, не испытывая проблем с распаковкой HTML, тем не менее неправильно обрабатывают сжатые файлы CSS и JavaScript.

2.2.1. Opera

Поддерживает методы gzip и deflate.

На данный момент у этого старейшего из используемых браузеров неизвестны какие-либо проблемы, связанные со сжатием.

2.2.2. Internet Explorer

Поддерживает методы gzip и deflate.

К сожалению, у этого браузера взаимоотношения со сжатым контентом складываются не так радужно, как хотелось бы. Хотя современные версии браузеров, по всей видимости, не имеют ошибок, связанных со сжатием, но все еще распространенные предыдущие версии содержат значительное их количество.

Проблемными являются версии, начиная с четвертой, заканчивая Internet Explorer 6.0 SP1. Хорошая новость, что в Service Pack 2 этого браузера все проблемы исправлены. К счастью, эта версия отличается полем «User-agent»: его значение содержит подстроку «SV1» («Security Version 1»), что позволяет отличить эту версию еще на сервере.

Некоторые из ошибок этого браузера можно обойти, но так как распространенность его проблемных версий составляет доли процента, то игра, вероятно, не стоит свеч.

Доля четвертой и пятой версий в совокупности не превышает 0,5% и продолжает снижаться, с шестой версией несколько сложнее — исследований, определяющих долю IE6SP1, нет, но можно получить оценку сверху: 13%*, если вычесть все IE6, установленные на Windows Vista, где в браузере уже содержится SP2.

2.2.3. Konqueror

Поддерживает gzip и deflate.

Данный браузер не слишком широко известен, но его версии до 3.1 содержат ошибки в реализации метода deflate. Так что если вы не заинтересованы в том, чтобы потерять клиента только из-за того, что он использует не слишком распространенный браузер, лучше метод deflate для проблемных версий отключить.

Также есть сообщения о проблемах со сжатыми методом gzip файлами JavaScript и CSS. Впрочем, доля этого браузера исчезающе мала (менее 0,1%).

2.2.4. Mozilla Firefox

Поддерживает gzip и deflate.

Firefox версий до 3.0x включительно в редких случаях загружает сжатые файлы CSS и JavaScript, не распаковывая их, кроме того, версия 3.0 не-

* — здесь и далее статистика использования браузеров получена с сайтов w3schools и LiveInternet.

правильно загружает файлы этих типов по протоколу HTTPS, если используется Keep-Alive.

Доля проблемных браузеров этого типа — около 26%.

2.2.5. Safari

Поддерживает gzip и deflate.

Версии до четвертой включительно неправильно обрабатывают сжатые JavaScript- и CSS-файлы, кроме того, если URL, по которому они скачиваются, заканчивается на .gz, они распознаются как архивы.

2.2.6. Google Chrome

Поддерживает gzip, deflate, bzip2, sdch.

Самый молодой из современных браузеров во второй версии испытывает проблемы с распаковкой CSS и JavaScript: в некоторых подверсиях браузер не пытается распаковать эти файлы, в других загружает их не до конца.

Доля проблемной версии в зарубежном и российском Интернете не превышает 6%.

Метод bzip2 было решено отключить в новых версиях браузера из-за проблем с некоторыми прокси-серверами, которые искажают заголовок ответа, если тело сжато этим методом.

2.2.7. Прокси и firewall

Некоторые типы прокси и firewall имеют ряд проблем, затрудняющих использование сжатия контента.

Первая проблема связана с кэшированием страниц на стороне прокси-сервера. Поскольку через один прокси могут работать несколько клиентов с разными браузерами, одна и та же страница может загружаться, в зависимости от браузера, в сжатом или несжатом виде. Прокси необходимо различать эти две ситуации и отдавать разные страницы в зависимости от возможностей клиента.

Для этого используется заголовок `Vary`, он указывает, на каких именно заголовках нужно основываться прокси, чтобы различать, какой контент отдавать браузеру. Для разных наборов значений этих полей попросту хранятся разные копии.

Некоторые довольно распространенные прокси (например, WinGate, Kerio WinRoute) неправильно обрабатывают эту ситуацию, поэтому в том случае, если запрос идет через прокси, контент либо не сжимают вовсе,

либо выстаяляют заголовки, препятствующие кэшированию его на прокси (например, при помощи заголовка `Cache-control: private`).

Отличить проксированный запрос от непроксированного можно по наличию заголовка `Via`, которые прокси обязаны выставлять в HTTP-запросах версии 1.1 (RFC 2616). Для запросов HTTP 1.0 такого флага нет, и обычно любой запрос этой версии на всякий случай считается проксируемым.

Вторая проблема подробно обсуждалась в докладе Тони Джентилкора (он работает в Google) на Velocity'09 (http://velocityconference.blip.tv/file/2293022/), где раскрывается еще одна неприятная подробность: некоторые firewall и прокси не пропускают через себя определенную часть заголовков, считая их лишними или опасными. Почему-то в эту часть попал и заголовок `Accept-Encoding`. В докладе упомянуто, что около 15% ответов остаются несжатыми по этой причине.

Некоторые firewall и прокси просто искажают заголовок (например, вместо `Accept-Encoding` вы получите `X-cept-Encoding`), другие же просто его удаляют.

Один из способов решить эту проблему, который сразу приходит в голову, — основываться не на `Accept-Encoding`, а на `User-agent`, в конце концов мы же знаем, что Internet Explorer 8.0 или Firefox 3.5 уж точно поддерживают gzip. Очевидный недостаток — необходимо наличие достаточно большого словаря браузеров, который к тому же нужно пополнять.

В докладе предлагается другой способ: при отсутствии заголовка `Accept-Encoding` загружать в скрытом фрейме (`<iframe>`) *сжатый* некэшируемый HTML с JavaScript внутри, который выставляет cookie через JavaScript. Если браузер способен распаковать этот документ, то cookie будет выставлена. В дальнейшем нужно ориентироваться, помимо заголовков, еще и на этот флаг.

Эта несложная идея хороша тем, что полностью автоматизирует процесс и не содержит риска ложного срабатывания.

2.3. Настройка веб-серверов Apache, nginx и lighttpd

Существуют два разных подхода к использованию сжатия на стороне сервера: сжимать данные «на лету» и использовать предварительно сжатые файлы. Первый выгоднее с точки зрения актуализации: не нужно заботиться, не устарел ли тот или иной файл; второй метод экономнее обходится с ресурсами машины, но приходится держать две копии файла для

тех браузеров, которые не поддерживают сжатие или поддерживают его с ошибками.

К счастью, методы можно успешно комбинировать, что позволяет сочетать достоинства того и другого.

2.3.1. Apache

Для Apache 1.x существуют два модуля, осуществляющих сжатие контента: это `mod_deflate` и `mod_gzip`. Оба модуля разработаны сторонними разработчиками. Первый представляет собой патч к исходному коду вебсервера, что иногда бывает неудобно и не умеет отдавать статически сжатые файлы; второй обладает другими недостатками: сначала сохраняет ответ во временный файл, который потом сжимает и отдает, что сказывается на производительности, и помимо этого конфигурационные опции модуля довольно аскетичны.

В Apache 2.x «из коробки» входит модуль `mod_deflate` (которые не имеет ничего общего с модулем для Apache 1.x).

Если есть возможность, настоятельно рекомендуем использовать для Apache 1.x модуль `mod_deflate`, он позволяет более тонко обходить ошибки браузера, имеет настройки степени сжатия плюс обладает лучшей производительностью. Ниже приведен оптимальный, на наш взгляд, конфигурационный файл этого модуля.

Для эффективного управления модулем необходим также модуль `mod_setenvif`, он позволит отключать сжатие для проблемных браузеров.

```
# Apache 1.x mod_deflate (необходим mod_setenvif)
# включаем модуль
DeflateEnable on
# включаем обход ошибки Internet Explorer 4.0, связанной с докачкой
# сжатого контента
DeflateDisableRange "MSIE 4."
# на данный момент все прокси корректно обрабатывают сжатый
# ответ, поэтому мы включаем сжатие, даже если использован
# прокси
DeflateHTTP  1.0
# сжиматься будут только ответы, не кэшируемые прокси-серверами, это
# позволяет избежать проблем с WinRoute, Kerio WinRoute и другими
DeflateProxied  poor_cachable
```

```
# эта директива включает заголовок Vary для всех ответов,
# IE версий 4.0–6.0SP1 не кэшируют ответы, если сжатие выключено,
# но получен заголовок Vary, но эти версии сейчас используются
# очень мало
DeflateVary on
# уровень сжатия – максимальный
DeflateCompLevel 9
DeflateHash 128
DeflateWindow 32
# включаем игнорирование вызова ap_bflush, это улучшает сжатие
# ответов некоторых приложений (например, написанных
# с применением Chili!Soft ASP). Если ваше приложение намеренно
# вызывает этот метод, опцию лучше отключить
DeflateIgnoreFlush on
# минимальный размер, при котором ответ сервера будет сжиматься
DeflateMinLength 100
# настраиваем, что сжимать
DeflateType text/html text/css text/xml application/xml image/x-icon
DeflateType application/x-javascript text/plain
# исключаем проблемные случаи по умолчанию выключаем сжатие
# на CSS и JavaScript (оно будет разрешено
# ниже для браузеров, не имеющих проблем, специальный флаг "no_gzip"
# говорит модулю, что сжатие надо выключить)
SetEnvIf   Content-Type  text/css no_gzip
SetEnvIf   Content-Type  application/x-javascript no_gzip
SetEnvIf   Content-Type  image/x-icon no_gzip

# выключаем сжатие для MSIE < 6SP1,
# исключаем притворяющуюся IE "Оперу"
BrowserMatch "MSIE [456]" no_gzip
BrowserMatch "SV1;" !no_gzip
BrowserMatch "Opera" !no_gzip
BrowserMatch "MSIE ([789]| [1-9][0-9])" !no_gzip
# не имеет проблем Firefox > 3
BrowserMatch "Firefox/(3\.[^0]|[4-9]|[1-9][0-9])" !no_gzip
```

Образец конфигурационного файла модуля `mod_gzip` взят из книги «Разгони свой сайт», в данной редакции комментарии были несколько расширены, а в настройку внесены правки.

Уровень эффективности сжатия не регулируется настройками модуля и может быть изменен только правкой исходного кода (строка `gz1` → `level` = `6` в функции `gz1_init`).

Упомянутый недостаток `mod_gzip` — недостаточно гибкие опции конфигурирования не дают выборочно включить сжатие CSS и JS только в тех браузерах, где это не создает проблем. По этой причине сжатие таких файлов отключено в данной конфигурации.

```
# Apache 1.x mod_gzip
<IfModule mod_gzip.c>
# включаем gzip
mod_gzip_on           Yes

# если рядом с запрашиваемым файлом есть сжатая версия
# с расширением .gz, то будет отдана именно она, ресурсы CPU
# расходоваться не будут
mod_gzip_can_negotiate    Yes

# используем при статическом архивировании расширение .gz
mod_gzip_static_suffix    .gz

# выставляем для статически архивированных файлов
# заголовок Content-Encoding: gzip
AddEncoding       gzip .gz

# не обновлять самостоятельно статически архивированные файлы,
# при включении этой опции вам не нужно самостоятельно проверять
# актуальность сжатых данных, но нужно позаботиться о том,
# что процесс веб-сервера имел доступ на запись к сжатым файлам
mod_gzip_update_static        No

# выставляем минимальный размер для сжимаемого файла, файлы
# меньшего размера сжимать неэффективно:
mod_gzip_minimum_file_size  1000

# и максимальный размер файла, слишком большие файлы будут
# сжиматься очень долго, поскольку mod_gzip не умеет сжимать
# данные по мере их получения, то пользователь начнет получать
# файл с задержкой, только после того как он будет полность сжат
mod_gzip_maximum_file_size  500000
```

```
# выставляем максимальный размер файла, сжимаемого прямо
# в памяти, чем больше эта величина, тем больше производительность
# и тем больше расход памяти каждый процессом, 60000 – максимальное
# значение для этой величины, поскольку авторы модуля столкнулись
# с проблемами при выделении более чем 64КБ в некоторых
# операционных системах
mod_gzip_maximum_inmem_size  60000

# устанавливаем версию протокола, с которой будут отдаваться
# gzip-файлы на клиент. Настройка появилась в те времена, когда
# некоторые прокси неверно обрабатывали сжатый ответ. В версии 1.1
# протокола HTTP прокси обязан выставить заголовок Via,
# что позволяет определить, проходит ли запрос через прокси,
# в версии 1.0 такого заголовка нет, поэтому в те времена
# запросы версии 1.0 просто не сжимались
mod_gzip_min_http        1000

# исключаем известные проблемные случаи (IE до 6.0SP1)
# вообще-то старые версии Opera также содержат строку MSIE,
# но мы их отделять не будем, поскольку в Apache используется
# синтаксис POSIX Extended Regular Expression, а записать
# на этом языке два исключения в одном правиле невероятно трудно.
# Правило, которое записано ниже, может давать ложные срабатывания
# в некоторых ситуациях, но в реальности эти ситуации не встречаются
mod_gzip_item_exclude    reqheader "User-agent: MSIE ↵
[456]([^S]V|[^S]V1|[^V]1|1[^;]|SV[^1]|[^SV1])+$"

# устанавливаем сжатие по умолчанию для файлов .html
mod_gzip_item_include     file     \.html$

# дополнительно сжимаем другие текстовые файлы с указанными
# MIME-типами
mod_gzip_item_include     mime     ^text/html$
mod_gzip_item_include     mime     ^text/plain$
mod_gzip_item_include     mime     ^httpd/unix-directory$

# отключаем сжатие для картинок (не дает никакого эффекта)
mod_gzip_item_exclude     mime     ^image/

# отключаем 'Transfer-encoding: chunked' для gzip-файлов, чтобы
# страница уходила на клиент одним куском
mod_gzip_dechunk        Yes
```

```
# добавляем заголовок Vary для корректного распознавания браузеров,
# находящихся за локальными прокси-серверами
mod_gzip_send_vary      On
</IfModule>

<IfModule mod_headers.c>
# запрещаем прокси-серверам кэшировать у себя сжатые версии файлов
        <FilesMatch .*\.(html|txt)$>
                Header set Cache-Control: private
        </FilesMatch>
</IfModule>
```

Модуль для Apache 2.x, `mod_deflate`, по настройкам схож со своим более ранним тезкой. Отличия: нет некоторых опций, которые вряд ли могут быть названы сейчас актуальными, и разделение сжатия HTML-файлов и файлов CSS/JS сделано удобнее.

Исходная версия конфигурационного файла этого модуля, опять же, взята из книги «Разгони свой сайт», здесь файл приводится с правками и расширенными комментариями.

Для управления этим модулем, как и в случае первого Apache, необходим `mod_setenvif`, следует учитывать, что часть настроек этого модуля `mod_deflate` нельзя использовать внутри файлов `.htaccess`, только внутри virtual host или на уровне глобальных настроек.

```
# Apache 2.x mod_deflate (необходим mod_setenvif)
# с самого начала включаем gzip для текстовых файлов
AddOutputFilterByType DEFLATE text/html
AddOutputFilterByType DEFLATE text/xml

# и для favicon.ico
AddOutputFilterByType DEFLATE image/x-icon

# также для CSS- и JavaScript-файлов
AddOutputFilterByType DEFLATE text/css
AddOutputFilterByType DEFLATE text/javascript
AddOutputFilterByType DEFLATE application/x-javascript

# далее устанавливаем максимальную степень сжатия (9)
# и максимальный размер окна (15). Если сервер не очень мощный,
# то уровень сжатия можно выставить в 1, размер файлов при этом
# увеличивается примерно на 20%.
```

```
DeflateCompressionLevel 9
DeflateWindowSize 15
DeflateBufferSize 32768

# отключаем сжатие для тех браузеров, у которых проблемы
# с его распознаванием

# выключаем сжатие для MSIE < 6SP1, исключаем
# притворяющуюся IE "Оперу"
BrowserMatch "MSIE [456]" no_gzip dont-vary
BrowserMatch "SV1;" !no_gzip !dont-vary
BrowserMatch "Opera" !no_gzip !dont-vary
# Firefox < 3.5, проблемы с CSS/JS
BrowserMatch "Firefox/[0-3]\." gzip-only-text/html
BrowserMatch "Firefox/3\.[1-9]" !gzip-only-text/html

# Chrome
BrowserMatch "Chrome/2" gzip-only-text/html

# все версии Safari
BrowserMatch "Safari" gzip-only-text/html

# Konqueror
BrowserMatch "Konqueror" gzip-only-text/html

# указываем прокси-серверам передавать заголовок User-Agent для
# корректного распознавания сжатия
Header append Vary User-Agent env=!dont-vary

# запрещаем кэширование на уровне прокси-сервера для всех файлов,
# для которых у нас выставлено сжатие,
<FilesMatch .*\.(css|js|php|phtml|shtml|html|xml)$>
        Header append Cache-Control: private
</FilesMatch>
```

2.3.2. nginx

Достаточно популярный веб-сервер nginx (читается как «engine X») также имеет два модуля для управления сжатием: `ngx_http_gzip_module` для сжатия «на лету» и `ngx_http_gzip_static_module` для статически сжатых файлов.

```
# nginx, ngx_http_gzip_module и ngx_http_gzip_static_module
# включаем сжатие "на лету"
gzip on;

# типы файлов, который будут сжиматься, text/html указывать не нужно,
# этот тип сжимается по умолчанию
gzip_types    text/plain text/css application/x-javascript ↵
text/xml application/xml application/rss+xml text/javascript ↵
image/x-icon;

# минимальный размер сжимаемого контента
gzip_min_length   1000;

# максимальное сжатие
gzip_comp_level 9;

# минимальная версия протокола HTTP в запросе, при которой будет
# происходить сжатие
gzip_http_version 1.0;

# разрешает выдачу заголовка Vary
gzip_vary on;

# разрешить сжатие проксируемых ответов, у которых есть
# заголовки
# Expired, Cache-contol: no-cache, Cache-contol: no-store,
# Cache-contol: private или Authorization
gzip_proxied  expired no-cache no-store private auth;

# отключаем сжатие для MSIE < 6.0SP1, с версии 0.8.11 эта маска
# не включает Internet Explorer 6.0SP2
gzip_disable msie6;

# к сожалению, у nginx нет способа запретить сжатие ответов
# какого-либо типа для обхода проблем в браузерах, но если
# у вас все файлы CSS и JavaScript оканчиваются, соответственно,
# на .css и .js, то можно ограничить их сжатие следующим способом
location ~* \.(css|js) {
    # разрешаем отдавать вместо несжатого файла предварительно
    # сжатый с постфиксом ".gz", если такой есть
    gzip_static on;
```

```
    # запрещаем сжатие файлов CSS и JS для проблемных браузеров
    gzip_disable Firefox/([0-2]\.|3\.0);
    gzip_disable Chrome/2;
    gzip_disable Safari;
}
```

2.3.3. lighttpd

У веб-сервера lighttpd (lighty) также имеются модули (mod_compress, mod_deflate), поддерживающие сжатие контента. Причем поддерживаются не только методы gzip и deflate, но и bzip2, не рекомендуемый на данный момент из-за проблем совместимости с прокси-серверами.

Модуль mod_compress поддерживает схему со статическими сжатыми файлами, у него имеется кэш, куда складываются и где автоматически создаются сжимаемые модулем файлы.

Имя файла и его путь в кэше составляется из пути и имени запрашиваемого файла плюс имя метода сжатия и ETag. Соответственно, если меняется ETag, сервер создает файл с другим именем и отдает свежую копию.

Очистка кэша оставлена на усмотрение пользователя и может производиться, например, такой вот командой (удаление файлов, которые были модифицированы более 10 дней назад):

```
find /var/www/cache -type f -mtime +10 | xargs -r rm
```

По сравнению с остальными рассматриваемыми модулями mod_compress имеет очень мало возможностей для конфигурирования. Оптимизированный конфигурационный файл приведен ниже.

```
# Lighttpd, mod_compress
# включаем сжатие
server.modules += ("mod_compress")

# включаем методы, которые будем поддерживать,
# здесь не указан bzip2
# из-за проблем с прокси и deflate из-за проблем с Konqueror
compress.allowed-encodings = ("gzip")

# директорий, куда складываются кэшированные пресжатые файлы
compress.cache-dir = "/var/www/cache/"
```

```
compress.filetype = ("text/plain", "text/html", "text/xml", ↵
"application/xml", "application/rss+xml", "image/x-icon")

# не сжимать контент, размер которого больше, чем 500KiB
compress.max-filesize = 500000

# боремся с проблемами в браузерах, основываясь на URL
$HTTP["url"] =~ "\.(css|js)$" {
    $HTTP["useragent"] != "Firefox/([0-2]\.|3\.0)" {
        $HTTP["useragent"] != "Chrome/2|Konqueror" {
            compress.filetype += ("text/css", "application/ ↵
            x-javascript", "application/javascript", ↵
            "text/javascript")
        }
    }
}

$HTTP["useragent"] = ~ "MSIE [4-6]" {
     $HTTP["useragent"] != "SV1|Opera" {
         compress.filetype = ()
     }
}
```

Модуль `mod_deflate`, который доступен в версии 1.5.0 и выше (или как патч к версии 1.4.x), предназначен для сжатия контента «на лету».

```
# Lighttpd, mod_deflate
# включаем сжатие
server.modules += ("mod_deflate")
deflate.enabled = "enable"

# максимальное сжатие
deflate.compression-level = 9
deflate.mem-level = 9
deflate.window-size = 15

# включаем методы, которые будем поддерживать, здесь не указан
# bzip2 из-за проблем с прокси и deflate из-за проблем с Konqueror
deflate.allowed_encodings = ("gzip")

# минимальный размер ответа, который будем сжимать
```

```
deflate.min-compress-size = 1000
# из-за того, что в зависимости от браузера мы не может отключать
# сжатие JS и CSS, не включаем их сжатие вообще
deflate.mimetypes = ("text/plain", "text/html", "text/xml", ↵
"application/xml", "application/rss+xml", "image/x-icon")

# минимальный размер блока для сжатия
deflate.work-block-size = 512

# отключаем сжатие для старых IE
$HTTP["useragent"] = ~ "MSIE [4-6]" {
    $HTTP["useragent"] != "SV1|Opera" {
        deflate.enabled = "disable"
    }
}

# боремся с проблемами в браузерах, основываясь на URL
$HTTP["url"] = ~ "\.(css|js)$" {
    $HTTP["useragent"] != "Firefox/([0-2]\.|3\.0)" {
        $HTTP["useragent"] != "Chrome/2|Konqueror|Safari" {
            compress.filetype += ("text/css", "application/ ↵
            x-javascript", "application/javascript", ↵
            "text/javascript")
        }
    }
}
```

2.4. Собственная реализация сжатия со стороны сервера

Для иллюстрации, каким образом можно реализовать сжатие со стороны сервера, мы выбрали язык PHP, как наиболее распространенный и простой в понимании. Тем, кому язык знаком, знают, что в нем реализованы два встроенных механизма, позволяющие полностью автоматизировать процесс, но, к сожалению, они никак не помогают бороться с ошибками браузера.

```
<?php
    class mod_compress
    {
        // настройки
```

```
// уровень сжатия
static public $deflatelevel = 9;
// минимальный размер ответа (в байтах), который будет
// сжат (0 – нет ограничений)
static public $minsize = 0;
// максимальный размер
static public $maxsize = 500000;

// внутренняя переменная – поддерживает ли браузер сжатие
static private $supported = null;

static public function init()
{
    // поддерживает ли браузер gzip?
    if (!self::issupportgzip()) {
        return self::$supported = false;
    }

    // каков тип ответа, тело которого будет сжимать?
    $type = self::detecttype();
    if ($type == 'unknown') {
        return self::$supported = false;
    }

    $ua = $_SERVER['HTTP_USER_AGENT'];

    // сжатие браузер поддерживает, теперь нужно
    // исключить браузер, который поддерживает его
    // с ошибками – MSIE < 6.0SP2
   if (preg_match('/MSIE [4-6](?:.(?!Opera|SV1))+/', $ua)) {
        return self::$supported = false;
    }

    // если требуется сжимать CSS/JS, то нужно
    // отфильтровать небезопасные браузеры
    if (
        $type == 'notsafe' &&
        preg_match('@Chrome/2|Konqueror|Firefox/ ↵
        (?:[0-2]\.|3\.0)@', $ua)
    ) {
        return self::$supported = false;
    }
```

```
        return self::$supported = true;
    }

    // посмотрим – поддерживает ли браузер gzip
    static private function issupportgzip()
    {
        foreach (preg_split('/\s*,\s*/',
        $_SERVER['HTTP_ACCEPT_ENCODING']) as $method) {

            // некоторые браузеры указывают вес
            // (предпочтения) методов сжатия, например,
            // "bzip2;q=0.9, gzip;q=0.1"
            // говорит о том, что браузер
            // хотел бы, чтобы ему отдавали контент, сжатый
            // методом bzip2,
            // но он поддерживает и gzip
            $method = explode(';', $method, 2);

            // но так как мы поддерживаем только gzip, вес
            // мы игнорируем
            if ($method[0] == 'gzip' ||
            $method[0] == 'x-gzip') {
                return true;
            }
        }

        return false;
    }

    // отделим "безопасные" типы от "небезопасных"
    static private function detecttype()
    {
        // поддерживаются не всеми браузерами
        $notsafe = array('text/css', 'text/javascript',
            'application/javascript',
            'application/x-javascript',
            'text/x-js', 'text/ecmascript',
            'application/ecmascript', 'text/vbscript',
            'text/fluffscript');

        // поддерживаются всеми браузерами
```

```
        $safe = array('text/html', 'image/x-icon',
            'text/plain',
            'text/xml', 'application/xml',
            'application/rss+xml');

        foreach (headers_list() as $header) {
            if (stripos($header, 'content-type') === 0) {
                $header = preg_split('/\s*:\s*/', $header);
                $type = strtolower($header[1]);

                if (in_array($type, $safe)) return 'safe';
                if (in_array($type, $notsafe))
                return 'notsafe';

                return 'unknown';
            }
        }

        // в случае, если Content-type не задан, считаем,
        // что это text/html
        return 'safe';
    }

    // проверка ограничний на размер
    static private function checksize($len)
    {
        if ($minsize && $len < $minsize) return false;
        if ($maxsize && $len > $maxsize) return false;

        return true;
    }

    // проверка, прошел ли запрос через прокси
    static private function checkproxy()
    {
        // в версии HTTP 1.1 есть обязательный заголовок,
        // который выставляет прокси, - Via, в HTTP/1.0
        // такого признака нет
        return $_SERVER['SERVER_PROTOCOL'] == 'HTTP/1.0' ||
                isset($_SERVER['HTTP_VIA']);

    }
```

```
// обработчик, который решит, будет ли сжат контент,
// и сожмет его
static public function handler($content, $stage)
{
    // проверим – нужно ли сжимать (не срабатывают ли
    // ограничения на размер)
    if ($content == '' ||
    !self::checksize(strlen($content))) {
        return $content;
    }

    if (self::$supported === null) {
        self::init();
    }

    // браузер не поддерживается, высылаем оригинал
    if (self::$supported === false) {
        return $content;
    }

    // этот заголовок скажет прокси-серверу и браузеру,
    // что возвращаемое нами будет иметь разное
    // содержимое в зависимости от типа браузера
    // и заголовка типа кодирования
    header('Vary: User-agent, Content-encoding');

    // запрещаем прокси сохранять содержимое,
    // для обхода прокси, не справляющихся
    // с кэшированием сжатого контента
    if (self::checkproxy()) {
        header('Cache-control: private');
    }

    return self::compress($content);
}

static public function compress($content)
{
    // сжимаем текст gzip'ом
    $content = gzencode ($content, $deflatelevel,
    FORCE_GZIP);
```

```
            // не забываем отдать длину сжатого потока,
            // это *очень* важно: некоторые браузеры не
            // обрабатывают сжатый поток без указания длины
            header('Content-length: ' . strlen($content));

            // …и метод его кодирования
            header('Content-encoding: gzip');

            return $content;
        }
    }
    ob_start(array('mod_compress', 'handler'));
    // здесь вывод вашего скрипта
```

Как видим, задача не такая уж и сложная. Точкой входа в класс является метод handler, который мы передаем функции `ob_start`, — она и позаботится о том, чтобы весь дальнейший вывод попадал в нашу функцию. Так как класс устанавливает заголовки, нужно, чтобы вызов `ob_start` произошел в вашей программе как можно раньше, до любого вывода данных в браузер.

Класс также можно доделать, чтобы поддержать метод bzip2 (нужно проверять вхождения «bzip2», «x-bzip» или «bzip»), если ответ не является проксируемым. Для сжатия bzip2 в PHP есть одноименный модуль. Его также можно улучшить: заменить проверку наличия в заголовке gzip и сжатие на вызов `ob_gzhandler`, но мы этого делать не стали намеренно — чтобы продемонстрировать, как это делается, если читатель захочет портировать код на другой язык.

Алгоритм работы класса следующий:

- проверяется длина поступивших данных, если она не укладывается в ограничения, указанные в настройках, то сжатия не происходит и браузеру отдается оригинальный контент;
- проверяется — поддерживает ли браузер gzip, если нет, отдается оригинальный контент;
- далее идет проверка, является ли сжимаемый контент одним из текстовых типов, если нет, то он не будет сжат;
- также контент не сжимается, если используется Internet Explorer версии, меньшей, чем 6.0 Service Pack 2, или если файл CSS или JavaScript запрашивается браузером, который не умеет корректно обрабатывать сжатые файлы этих типов;
- если все в порядке, выставляется заголовок, указывающий, как правильно кэшировать такое содержимое и, если запрос проксируемый,

указывающий прокси, что проксировать его не нужно, это позволяет избежать проблем с некоторыми типами прокси-серверов;
- далее происходит сжатие и выставляется заголовок, указывающий длину сжатого содержимого, которое затем отдается браузеру.

2.5. Альтернативные методы сжатия

Браузер является настолько мощной платформой, поддерживающей такое количество разнообразных технологий, что одно и то же зачастую можно сделать несколькими способами, некоторые из которых красивы в силу своей изощренности, а другие — практичны, например, позволяют обойти ошибки браузеров.

Мы рассмотрим два метода реализации сжатия, которые можно использовать, если встроенное сжатие в клиентском браузере реализовано с ошибками. Первый метод — это сжатие через тег canvas, второе — сжатие JavaScript, реализованное на самом JavaScript.

Первый метод предложил Джейкоб Седелин в своем блоге «Nihilogic» (http://blog.nihilogic.dk/2008/05/compression-using-canvas-and-png.html); о реализации второго метода, возможно (тут трудно установить истину), впервые задумался один из авторов этой книги, реализовав в 2001 году в рамках проекта JUnix (http://junix.kzn.ru) простенькое сжатие, которое называлось jzip.

2.5.1. Сжатие тегом canvas

canvas — предложенный фирмой Apple тег, который на данный момент входит в HTML5. Он позволяет при помощи JavaScript создавать на ограниченном тегом участке растровые изображения.

Идея сжатия, реализуемого при помощи этого тега, проста — каждый байт сжимаемого содержимого представляется как точка изображения любого формата сжатия без потерь, поддерживаемого браузером (GIF, PNG). Это изображение запрашивается с сервера и подгружается в тег canvas методом drawImage (поддерживается браузерами Firefox 1.5 и выше, Safari 2.0 и выше, Opera 9.0 и выше, а также Google Chrome).

Как легко догадаться, из canvas изображение поточечно считывается, каждая точка представляется как символ, и полученная строка выполняется как JavaScript.

Уровень сжатия таким способом кода колеблется в условных пределах от 20 до 50%. Например, библиотека jQuery версии 1.2.3 сжимается с

53 Кб до 17, что экономит 32% (для сравнения: gzip сжимает ее до 15,5 Кб, bzip2 — до 14,6 Кб).

Особенно привлекательно в этом методе то, что часть JavaScript, ответственная за получение кода на стороне клиента, очень небольшая:

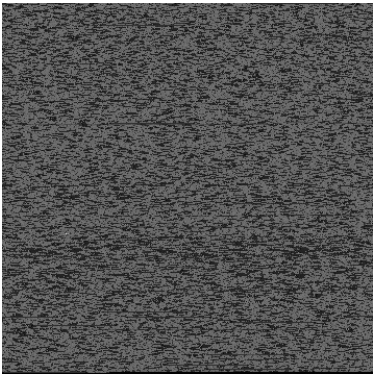
```
var codeimg = new Image()
// когда картинка загрузится, вызовется эта функция
codeimg.onload = function () {
    var code = '' // переменная, куда будет собираться код
    var size = 119 // размер изображения (оно квадратное)
    var canvas = document.createElement("canvas")

    canvas.width = canvas.height =
    canvas.style.width = canvas.style.height = size

    var inner = canvas.getContext("2d")
    // загружаем изображение в созданный нами CANVAS
    inner.drawImage(codeimg)

    // забираем содержимое и переводим его в символы
    var data = inner.getImageData(0, 0, size, size)
    for (var i = 0, len = data.length; i<len; i+=4) {
        if (data[i] > 0) code += String.fromCharCode(data[i])
    }

    eval(code)
}
// указываем URL изображения, где у нас хранится код
codeimg.src = 'image-with-our-code.png'
```

Серверная часть также очень проста, реализацию на PHP можно посмотреть в блоге автора метода, так же легко она реализуется на любом другом языке — в квадратное изображение, сторона которого равна квадратному корню из значения длины файла, в каждую точку ставится по коду символа из передаваемого контента.

Таким же методом можно паковать и CSS — в конце вместо eval нужно лишь создать в DOM тег `<style>` с соответствующим содержимым.

Само изображение, получающееся в результате, ничего интересного собой не представляет — обычный бинарный шум.

Рис. 2.1. Библиотека Prototype (версия 1.6.0.2), сжатая до 30 Кб PNG (экономия 24%)



У метода есть и недостатки. Во-первых, автор не использует цветовую составляющую для передачи каких-либо данных, и тут есть простор для экспериментов. Во-вторых, на текущий момент распаковка средних по размеру (200-500 Кб) данных занимает несколько секунд. В-третьих, метод поддерживается не всеми браузерами (например, не поддерживается IE). И в-четвертых, автор не позаботился о поддержке UTF-8 (что, впрочем, можно исправить).

Поскольку общее время, через которое скрипт будет доступен, равно сумме времени загрузки и времени, потраченного на его распаковку, нужно очень внимательно относиться к использованию этого метода — возможно, результатом его применения станет лишь увеличение времени ожидания.

Не исключено, что ситуацию можно исправить за счет использования цветовых компонент изображения. В этом случае размер передаваемых (а значит, и обрабатываемых скриптом) данных должен уменьшиться.

2.5.2. Распаковка, реализованная на JavaScript

На клиентской стороне на JavaScript реализуют только распаковку данных, сжатие происходит на сервере и может быть реализовано на любом языке программирования.

Ничего необычного в реализации какого-либо алгоритма нет: JavaScript — такой же язык программирования, но есть специфика: браузеры на данный момент содержат недостаточно оптимизированные интерпретаторы, и это необходимо учитывать, иначе распаковка будет выполняться удручающе медленно.

Впрочем, на этом фронте есть улучшения. Новые интерпретаторы языков JavaScript браузеров Firefox, Safari и Google Chrome показывают впечатляющие результаты. Скажем, реализация на JavaScript сжатия LZW (Lempel-Ziv-Welch, используется в GIF и PDF) (http://zapper.hodgers.com/labs/?p=90) распаковывает в этих браузерах библиотеку Prototype 1.6.0.2 (это 123 Кб) на среднем ноутбуке менее чем за одну десятую секунды.

Впрочем, десятая «Опера», последняя на данный момент, показывает куда менее интересное время — полсекунды, а Internet Explorer 8.0 — 0,3 секунды.

В будущем, если там еще будут встречаться проблемы с реализацией gzip у браузеров, можно будет использовать реализации достаточно сложных методов сжатия на JavaScript, а в настоящем же приходится довольствоваться чем-то менее ресурсоемким.

Например, программа «Packer» Дина Эдвардса (http://dean.edwards.name/packer/), использует алгоритм, который автор назвал «Base62», потому что в кодировании используется 62 символа — большие и маленькие латинские буквы плюс цифры.

Сжимаемый файл разбивается на слова, слова сортируются по частоте их употребления (сначала — наиболее употребительные), им присваиваются номера, которые кодируются алфавитом в 62 символа. Далее происходит замена — в коде слова заменяются их номерами в шестидятитидвухричной системе. На клиенте осуществляется обратная замена.

Этот алгоритм работает достаточно быстро для того, чтобы накладные расходы на распаковку скрадывались улучшенным временем загрузки.

Глава 3. Алгоритмы уменьшения изображений

За годы существования браузеров их производители встроили в них поддержку, без сомнения, гигантского количества форматов графики: GIF, JPEG, JPEG2000, PNG, APNG, MNG, ART, WMF, EMF, BMP, ICO, XBM, SVG, CUR, ANI, WBMP, TIFF.

Часть форматов исчезла (как поддержка ART и XBM из Internet Explorer или MNG из Firefox), другая часть специфична только для одного браузера (например, WBMP в «Опере» или JPEG2000 в браузерах WebKit/KHTML), так что разумно ориентироваться на форматы, поддерживаемые во всех браузерах, — это GIF, JPEG и PNG.

Интересным также является формат SVG, который активно прокладывает себе дорогу, но, к сожалению, не поддерживается Internet Explorer; впрочем, есть надежда, что следующая версия этого браузера его поддерживать будет. Так что мы рассмотрим и его, с расчетом на будущее.

О формате ICO мы говорить не будем, хотя он и поддерживается всеми браузерами: его использование, во-первых, весьма специфично, а во-вторых, детально было рассмотрено в книге «Разгони свой сайт».

Итак, четыре формата графики: GIF, JPEG, PNG и SVG, у каждого из них своя специфика, у каждого свои особенности, каждый поддается, в раз-

ной степени, оптимизации по размеру. Эти вопросы и будут рассмотрены в этой главе подробнее.

3.1. Уменьшаем GIF (Graphics Interchange Format)

Придуманный в 1987 году фирмой CompuServe формат в настоящий момент имеет два стандарта: GIF87a и GIF89a. С точки зрения пользователя они не различаются, важными же отличиями более поздней версии формата стали наличие анимации, чересстрочный вывод и поддержки метаданных.

Это первый формат (наряду с XBM), применявшийся в вебе, что наложило свой отпечаток: статичное изображение или кадр анимации могут содержать не более 256 цветов.

До появления PNG, который будет рассмотрен ниже, в этом формате ценилось наличие прозрачности, которого не было в JPEG. Но и сейчас формат популярен, в основном из-за наличия в нем анимации, поскольку альтернативные форматы анимации (APNG, MNG, SVG) еще не получили столь широкого распространения. Вторая причина — небольшие изображения могут занимать в формате GIF меньше объема, чем в PNG.

3.1.1. Рецепт № 1: горизонтальное меньше вертикального

Изображения GIF хорошо поддаются оптимизации. В параграфе № 8 «Ководства» Артемия Лебедева есть небольшое, но интересное исследование особенностей этого формата. Вывод из параграфа простой: чем больше на картинке горизонтальных линий одного цвета, тем лучше сжимается изображение. Алгоритм компрессии LZW, применяемый в GIF, вообще лучше сжимает регулярные горизонтальные структуры в пределах одной линии.

Некоторые программы содержат специальные инструменты, вносящие в изображение регулируемые горизонтальные искажения, которые позволяют добиться лучшего сжатия, пусть и ценой некоторой потери качества.

В Adobe Photoshop этот инструмент называется «Lossy» и находится в диалоге «Save for Web...» (`Ctrl+Shift+Alt+S`) формата GIF.

Инструмент удобно применять в изображениях, где много визуального шума, регулируя искажения так, чтобы они не бросались в глаза. В некоторых случаях этот инструмент способен даже улучшить качество изображения — например, там, где из-за недостатка цветов получаются рез-

кие неестественные переходы, искажения, вносимые «Lossy», улучшают общую картину.

Рис. 3.1. Применение инструмента «Lossy» со значением 13 позволило смягчить переходы и уменьшить вес изображения с 596 Кб до 281 Кб

Рецепт, конечно же, не универсален. Но существует еще несколько способов сделать GIF меньше: изменение параметров сжатия (размер словаря), уменьшение количества цветов, удаление опциональной информации (например, комментариев), а также исключение признаков конца кадра и оптимизация кадров изображения, если изображение содержит анимацию.

3.1.2. Рецепт № 2: уменьшаем количество цветов

С большей частью описанных способов лучше справятся специализированные программы, которые будут рассмотрены ниже, а вот уменьшение количества цветов может сделать только человек. Автоматически не-

возможно оценить, насколько допустимо будет изменение качества изображения в этом случае.

Формат поддерживает фиксированное число цветов в изображении: 2, 4, 8, 16, 32, 64, 128, 256, но часть палитры может быть не использована. Если выбрать, например, 200 цветов, то в палитре будут указаны 256 цветов, 56 из которых не будут задействованы. Но чем меньше цветов использовано в изображении, тем лучше оно сожмется.

Еще одним параметром, влияющим на размер изображения, является выбор алгоритма уменьшения количества цветов, что актуально, если в формате GIF сохраняется изображение из полноцветного оригинала.

Adobe Photoshop предлагает на выбор четыре алгоритма сокращения: Perceptual, Selective, Adaptive и Restrictive (Web).

Последний режим дает наихудшее качество и обязан своему существованию причинам скорее историческим: он подгоняет палитру изображения под 216 цветов, которые отображаются одинаково на всех устройствах с 256 цветами.

Режим Perceptual обычно используется для фотографий, где важна точность передачи; в этом режиме особое внимание уделяется тому, как человек воспринимает цвета.

Selective лучше всего подходит для ярких и четких изображений; режим нацелен на сохранение цветов однотонных элементов.

В режиме Adaptive для палитры выбираются части спектра, где представлено большинство оттенков изображения; в этом режиме изображение, как правило, занимает чуть больше, чем в других.

При уменьшении количества цветов выбранная вами программа будет пытаться заменить недостающий цвет узором из имеющихся цветов, применяя для этого один из алгоритмов. Например, Photoshop предоставляет выбор из «No Dither», «Diffusion», «Pattern» и «Noise». Их значение описывать нет никакого смысла, лучше один раз попробовать их применить, правда, любой из этих инструментов увеличивает размер изображения.

Наиболее удобны для процесса выбора количества цветов и алгоритма их сокращения программы, показывающие одновременно исходное изображение и обработанный результат — например, уже неоднократно упомянутый Adobe Photoshop.

3.1.3. Рецепт № 3: автоматическая оптимизация

Для быстрой оптимизации изображений GIF удобны программы автоматической оптимизации. Ни одна из них уже не развивается, что, впрочем, не мешает их использовать.

Чтобы выяснить, насколько качественно справляются программы автоматической оптимизации GIF со своей работой, были использованы три набора изображений.

Первый набор — 494 иконки с небольшим количеством цветов, общим объемом 310 килобайт. Второй набор — 10 произвольных фрагментов фильмов, скачанных с сайта GIFTUBE, характеризующихся отсутствием статичных межкадровых элементов и использованием всей палитры цветов, объем 7,63 Мб. Третий набор — 411 анимированных изображений, с использованием неполной палитры; в изобилии присутствуют статичные элементы, объем — 21,95 Мб.

Название	Статичные иконки	Фильмы	Простая анимация
Gifsicle 1.5	258 Кб (17%)	7,63 Мб (0%)	21,53 Мб (2%)
GIFLite 2.10	275 Кб (11%)	**6,19 Мб (19%)**	19,22 Мб (12%)
SuperGIF 1.5	**188 Кб (40%)**	Файлы испорчены*	**17,78 Мб (19%)**
Advanced GIF Optimizer 4.0.12	250 Кб (19%)	7,63 Мб (0%)	21,57 Мб (2%)
Real GIF Optimizer 3.0.5	250 Кб (19%)	7,63 Мб (0%)	21,57 Мб (2%)
Ultra GIF Optimizer 1.02	250 Кб (19%)	7,63 Мб (0%)	21,57 Мб (2%)
A Smaller GIF 1.22	293 Кб (5%)	7,63 Мб (0%)	21,89 Мб (>1%)

* SuperGIF показал потрясающие способности (файлы были уменьшены более чем наполовину), но при проигрывании на всех оптимизированных изображениях появился шлейф от движущихся объектов.

Как показывают тесты, лучше всех справляется с оптимизацией изображения программа SuperGIF (платная программа BoxTop Software, $49,95), но результат ее работы нужно контролировать: полноцветную анимацию она портит. Единственная программа из наших тестов, которая успешно оптимизировала такие изображения, — это GIFLite, тоже платная программа компании White River Software (цена $30, но существует ли компания — неясно), закончившая свое развитие в 1995 году (программа под DOS).

Если с запуском SuperGIF проблем нет — это программа с графической оболочкой и почти без настроек, — то GIFLite запустить сложнее: она работает только из командной строки, не поддерживает длинные имена файлов и не умеет самостоятельно выбирать метод, который оказался лучшим для конкретного файла. Эти печальные недостатки устраняет следующая программа, написанная на языке пакетных файлов Windows:

```
@ECHO OFF
REM Написал Евгений "BOLK" Степанищев. 2007.
MKDIR GIFLITE.$$$ 2>nul

REM Основной цикл обработки файлов GIF
FOR /R %%N IN (*.gif) DO @CALL :method %%N

REM Удаляем весь мусор, который мог остаться
DEL /Q giflite.tmp 2>nul
RMDIR /S /Q GIFLITE.$$$
EXIT

:method
REM Перебираем методы
FOR %%M IN (0 1 2 3) DO @CALL :giflite %%M %%~s1 %1

REM Сортируем полученное по размеру и забираем последний
REM (наименьший) файл
FOR /F "usebackq skip=3" %%R IN (`DIR /B /O-S GIFLITE.$$$`) ↵
DO @CALL :getresult %%R %1
GOTO :EOF

:getresult
REM Переписываем файл на место прежнего, удаляем мусор
MOVE /Y GIFLITE.$$$\%1 %2
DEL /Q GIFLITE.$$$\*.*
GOTO :EOF

:giflite
REM Запускаем преобразование
GIFLITE.EXE -t -h -m%1 -o %2 GIFLITE.$$$\%1
```

Эта программа обработает все файлы GIF в текущей папке и всех вложенных.

3.1.4. Рецепт № 4: уменьшаем анимацию вручную

Уменьшить размер анимированного GIF можно и самостоятельно, работа нетрудная, но долгая и рутинная. Анимация в GIF устроена достаточно просто: каждый последующий кадр перекрывает предыдущий, причем размеры кадров могут быть меньше размера изображения, и в этом случае на экране будет содержимое двух кадров: текущий кадр и то, что не было перекрыто из предыдущего.

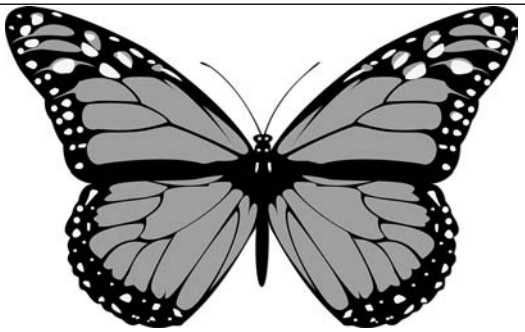
Отсюда получается, что если в анимированном изображении есть статичные элементы (например, задний фон), то их нужно оставить на предыдущем кадре, перекрыв следующим лишь то, что изменилось. Причем если использовать прозрачность, то перекрывать можно области любой формы. Размеры и палитра кадров уменьшится, а значит, уменьшится и размер файла с анимацией.

3.1.5. Рецепт № 5: когда GIF лучше PNG

Очень хочется упомянуть вытекающий из предыдущего рецепта способ создания полноцветных изображений в формате GIF, тем более что такой способ также иногда позволяет получить выигрыш в размере.

Самый простой принцип получения полноцветного GIF следующий: исходное изображение разбивается на несколько кусков, так, чтобы количество цветов в каждом из них было не более 256 (проще всего — сделать куски 16×16, это как раз 256 пикселей, значит, и цветов не может быть больше). Каждый кусок вставляется в отдельный кадр, и между ними выставляется нулевая задержка. Таким образом, пользователь увидит все кадры одновременно, и они сложатся в цельную картинку.

Рис. 3.2. Пример изображения, занимающего меньше в полноцветном GIF, чем в PNG



Существуют несколько программ, работающих подобным или чуть более сложным способом. Лучшая, на наш взгляд, — платная программа itsagif, фирмы Pedagoguery Software.

Авторы itsagif заявляют, что полноцветный GIF может выиграть у PNG по эффективности сжатия на изображениях с количеством цветов от 300 до 1000. Это действительно так, если в изображении не встречаются элементы, которые PNG умеет сжимать лучше GIF, например, градиенты.

Вот этот рисунок бабочки 756×488 пикселей содержит 1073 уникальных цвета. В файле PNG, оптимизированном программой PNGout, он занимает 77,6 Кб, тогда как в полноцветном GIF — 68,4, экономия составляет 12%.

3.1.6. Резюме

Из рассмотренного нами материала можно сделать следующее резюме. Несмотря на конкурирующий формат (PNG) GIF еще может быть полезен: некоторые изображения занимают в этом формате меньше места, и это пока единственный универсальный для всех браузеров способ получить анимированное изображение.

Однородные по горизонтали изображения сжимаются лучше, чем остальные, поэтому в зашумленные изображения имеет смысл вносить горизонтальные искажения инструментом «Lossy» (Adobe Photoshop). «Lossy» и уменьшение количества используемых цветов являются способами ручной оптимизации статических изображений. В редких случаях позволяет уменьшить размер изображения применение чересстрочного режима.

Для программной оптимизации полезны программы GIFLite и SuperGIF, причем результат работы последней необходимо контролировать.

Анимированные изображения можно оптимизировать как вручную, если воспользоваться неполным перекрытием кадров, так и автоматически, при помощи уже упомянутых программ. Интересной техникой оптимизации, основанной на анимации с нулевыми задержками, является использование программы itsagif фирмы Pedagoguery Software, которая позволяет получать полноцветные изображения, в определенных условиях занимающие меньше, чем PNG.

3.2. Оптимизируем JPEG (Joint Photographic Experts Group)

Алгоритм JPEG появился в 1992 году стараниями группы Joint Photographic Experts Group, аббревиатура которой и дала название стан-

дарту. Строго говоря, JPEG — это алгоритм сжатия данных, а формат файлов, использующий алгоритм JPEG, называется JFIF (JPEG File Interchange Format), но поскольку все привыкли к аббревиатуре JPEG, то и мы дальше будем пользоваться именно ей.

В отличие от GIF JPEG является форматом сжатия с потерями: это значит, что сохраненное изображение будет отличаться от оригинала некими допустимыми потерями, часто не заметными на глаз. При сохранении выбирается так называемое «качество», которое обычно варьируется от 0 до 100, где ноль означает наихудшее качество (впрочем, может быть и иначе: в Paint Shop Pro выбирается «степень сжатия», где сто по качеству хуже нуля). Нужно понимать, что шкала эта условная и ее смысл сильно варьируется в разных программах, например, при определенных значениях различные программы могут включать какие-то дополнительные методы оптимизации.

В вебе стандарт появился после GIF, впервые в браузере Mosaic, для хранения полноцветных изображений, потери которых в детализации некритичны. Это изображения, где яркость и цвет меняются сравнительно плавно. То есть, для шрифтовых написаний и графиков этот формат лучше не использовать.

Формат JPEG не стоит на месте: появился стандарт JPEG2000, как его дальнейшее развитие, JPEG-LS для сохранения без потерь и так далее. Но так как эти форматы не поддерживаются браузерами или поддерживаются недостаточно широко, на них мы останавливаться не будем.

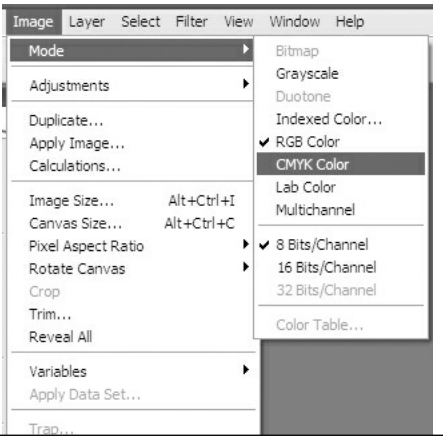
3.2.1. Проблемы отображения JPEG браузерами

Это кажется странным, но у браузеров есть проблемы с отображением JPEG. Эта тема лишь косвенно относится к оптимизации, тем не менее нам она представляется достаточно важной, чтобы о ней упомянуть.

Во-первых, Internet Explorer, включая восьмую версию, не поддерживает прогрессивную загрузку JPEG (когда качество изображения улучшается по мере его загрузки): он показывает файл только после его полной загрузки, что сильно отличается от поведения «традиционного» JPEG, когда изображение отображается сверху вниз, по мере загрузки. Проблема в том, что пользователь может подумать, что загрузка страницы закончена, тогда как некоторые функциональные элементы могут быть еще не простроены до конца.

Во-вторых, некоторые программы (например, Photoshop) сохранят JPEG в цветовом пространстве CMYK, если при работе с изображением выбран именно этот режим. Определенные, используемые в настоящее время браузеры (среди них Firefox 2, Internet Explorer 6, Google Chrome 2) не смогут отобразить такое изображение, а Safari, вплоть до версии 4, отобразит его инвертированным.

Рис. 3.3. Диалог выбора цветового пространства в программе Adobe Photoshop



В-третьих, Firefox (включая текущую версию — 3.5.3) в версии под Windows иногда неверно декодирует JPEG-файлы: на некоторых изображениях только в этом браузере появляются едва заметные полосы. Увидеть их достаточно просто: создайте новый файл в Photoshop, залейте его цветом #5c6264 и сохраните с качеством 80%.

3.2.2. Рецепт № 1: в чем лучше сохранить

Распространенных программ для редактирования изображений немного. Определенным стандартом, по всей видимости, стали две из них: Adobe Photoshop и бесплатный редактор GIMP.

Любопытно сравнить это две программы, чтобы посмотреть, как они справляются с сохранением изображения в формат JPEG.

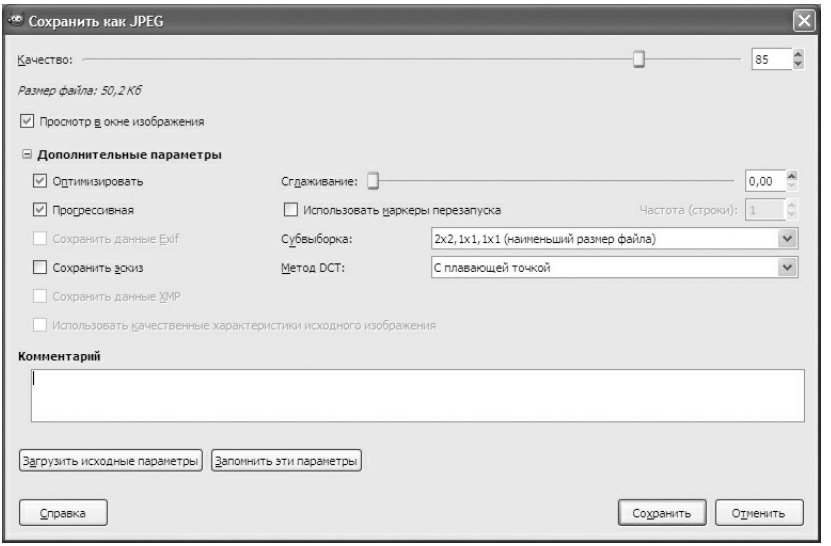
Для сравнения графических редакторов мы выбрали три фотографии 800×600. На первой в кадре много довольно крупных объектов: мотоциклы, деревья, люди, на второй всю фотографию занимает саркофаг с мелким орнаментом, на третьей почти ничего нет, кроме неба и океана.

Степень сжатия выставлялась в каждой программе из субъективных соображений: чтобы на глаз не были заметны искажения, вызванные сжатием.

Программа	Крупные объекты	Мелкие объекты	Однородное фото
Adobe Photoshop CS2	198,4 Кб	108,7 Кб	35,9 Кб
GIMP 2.6.6	**124,3 Кб**	**73,5 Кб**	**28,9 Кб**

Убедительную победу одержал GIMP: его многообразие настроек позволяет гибко управлять размером файла.

Рис. 3.4. Диалог сохранения файла в формате JPEG в редакторе GIMP



Указание текста комментария, использование маркеров перезапуска, сохранение эскиза и EXIF увеличивают размер файла, опция «оптимиза-

ция» (включающая оптимизацию метода Хаффмана), субвыборка «2×2, 1×1, 1×1» и вычисление DCT с плавающей точкой уменьшают его.

Смысл значений параметра «Субвыборка» (или цветового прореживания) — уменьшение информации о цвете. Форматов записи этого параметра несколько, но наиболее распространенный — три цифры с разделителями. Читаются они очень просто: например, «4:2:2» означает «на каждые четыре пикселя сохраняется четыре значения яркости и по два значения на каждый компонент цвета». Из этого правила есть одно исключение: «4:2:0» расшифровывается как «на каждые четыре пикселя в квадрате 2×2 сохраняются четыре значения яркости и по одному значению цвета».

Значение «4:2:0» часто называют «2×2», а «4:4:4»— «1×1». Как видно, GIMP использует другие, более понятные значения субвыборки. К примеру, «2×2, 1×1, 1×1» означает: для квадрата 2×2 пикселя сохраняются все значения яркости, одно значение по вертикали и одно по горизонтали для первой компоненты цвета, а также одно по вертикали и горизонтали для второй компоненты.

Чем меньше информации о цвете сохраняется, тем меньше файл и тем более он далек от оригинала. Легко сделать оправданный вывод, что субвыборка «4:2:2» больше скажется на четких вертикалях, а «4:2:0» сделает более расплывчатыми и вертикальные, и горизонтальные линии.

Параметр «Метод DCT» не так важен для понимания оптимизации. Для полноты картины поясним, что он имеет отношение к точности вычисления дискретного косинусного преобразования (разновидность преобразования Фурье), при помощи которого производится уменьшение цветовой информации. Чем точнее вычисляется это преобразование, тем меньше размер файла с изображением.

Параметр «Маркеры перезапуска» влияет на расстановку специальных маркеров, используемых для нейтрализации ошибок. В вебе они едва ли полезны, а их применение увеличивает размер файла.

Параметр «Прогрессивная загрузка» с вероятностью 75% уменьшит размер изображения, если оно меньше 10 Кб, и с вероятностью 94% увеличит его, если оно больше этого размера. Если вам не важны соображения совместимости с Internet Explorer, изложенные выше, то вы можете попробовать включить этот параметр, чтобы посмотреть, не уменьшится ли размер.

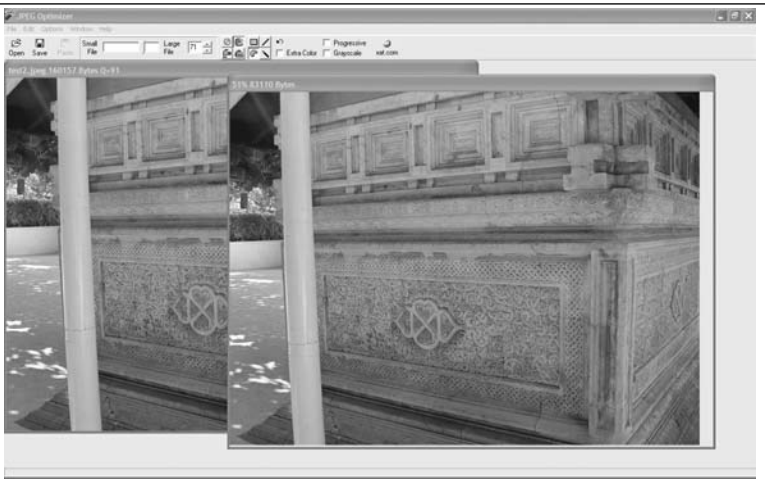
На размер изображения также влияют резкие переходы (границы): чем их меньше, тем лучше оно сжимается. Поэтому GIMP на экране сохранения JPEG-изображения содержит инструмент для его сглаживания. Для некоторых типов изображений (например, мелкий лиственный рисунок) легкое увеличение сглаживания не вызовет ухудшения восприятия, но уменьшит размер.

Справедливости ради надо заметить, что на размер влияют еще два параметра изображения: чем ниже его контрастность (contrast) и/или насыщенность (saturation), тем меньше будет его размер в JPEG.

Параметр «Качество» влияет на изображение очевидным способом. Хороший алгоритм — выбрать качество 75% и, если искажения слишком заметны, попробовать его увеличить. Если исходная картинка уже невысокого качества, можно попробовать снизить этот параметр до 50%. Никогда не сохраняйте изображения с качеством выше 95%, это не даст улучшения качества изображения, но серьезно увеличит его размеры. Качество «100%» не означает «сохранить без потерь», это всего лишь математический предел алгоритма.

Куда более сложным, но и более мощным средством оптимизации изображений при сохранении является платная программа JPEG Optimizer фирмы Xat. Она дает экономию большую, чем GIMP, но чтобы ее добиться, придется потратить куда больше времени.

Рис. 3.5. Интерфейс программы Xat JPEG Optimizer



Принцип обещаемой чудесной оптимизации заключается в том, что JPEG позволяет сохранять разные области изображения с разным качеством. Например, более темные или однородные области можно сохранить с качеством похуже, содержащие мелкие детали — с наилучшим.

Вообще-то JPEG Optimizer имеет автоматический режим, но, на наш взгляд, эффективнее всего работать с программой в режиме эксперта («Options → Expert Mode»).

Для обработки изображения первым делом нужно выбрать минимально допустимое качество, для этой цели сверху есть специальный ползунок.

Далее нужно определить, какие области изображения можно сохранить с наихудшим качеством. На фотографии выше это, например, песок слева снизу, темное пятно справа вверху и столб с левой стороны рисунка. Выбираем режим «Compress Much More» (F8) при помощи инструментов «Freehand Selection» (F9), «Rectangle Selection» (F10) или «Line Selection» (F11) и любым удобным способом выделяем нужные области.

Для областей, где настолько сильное сжатие применять нельзя, можно выбрать режимы «Compress Even More» (F7) или еще более слабый «Compress More» (F6). Если какую-то область вы сжали слишком сильно, не беда, можно воспользоваться инструментом «Undo» (`Ctrl-Z`) или переключиться в режим «Remove Extra Compression» (F5) и выделить нужную область.

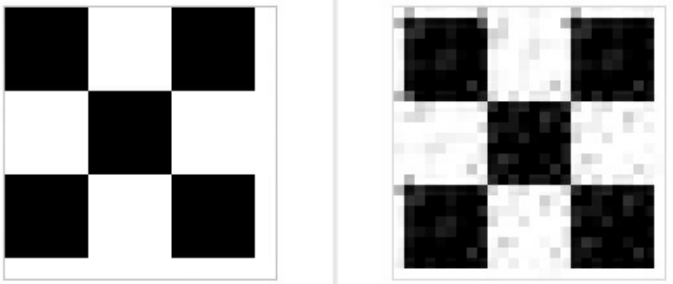
Потрудившись, можно уменьшить размер изображения на 3-20% по сравнению с GIMP, в зависимости от изображения.

3.2.3. Рецепт № 2: выравниваем по сетке

Два этапа JPEG-сжатия представляют для ручной оптимизации непосредственный интерес: это перевод цветов в пространство YCbCr и разбиение изображения на блоки.

Начнем с блоков. По алгоритму, после преобразования цветовой модели изображение разбивается на небольшие блоки 8×8 пикселей и уже с ними производится остальная работа.

Рис. 3.6. Последствия сохранения изображения в формат JPEG с выравниванием границ по сетке (слева) и без выравнивания (справа)



Посмотрим на эти два идентичных изображения, сохраненных в формате JPEG с качеством ноль. Почему изображение слева чистое, а справа

вокруг квадратов виден «мусор»? Ответ очень прост: на изображении слева границы цветов выравнены по решетке 8×8 пикселей, каждый квадрат 8×8 содержит только один цвет, а изображение справа сдвинуто на один пиксель вправо и вниз.

На практике это означает, что если у вас в изображении есть прямоугольные области, лучше выровнять их границы по решетке 8×8. В Photoshop для этой цели очень удобно пользоваться инструментом «Grid» (`Ctrl+'`), выставив в настройках (`Ctrl+K`, `Ctrl+6`) удобный шаг решетки. Важно также знать, что Photoshop при сохранении JPEG с качеством 0-50 оптимизирует цвета по решетке 8×16.

После выравнивания изображение можно попробовать сохранить в более низком качестве — качество не пострадает, тогда как размер уменьшится.

Когда выровнять все границы не получается и вокруг одной из них слишком много «мусора», можно поменять параметр «цветовое прореживание» («downsampling») вашей программы, если она позволяет его указать. Правда, размер файла при этом увеличится. В GIMP его надо выставить в «1×1, 1×1, 1×1», в cjpeg указать параметр -sample 1x1, а Photoshop выставляет это значение самостоятельно, если указанное качество выше или равно 50%.

3.2.4. Рецепт № 3: оптимизируем мелкую текстуру

Следующий метод был позаимствован из статьи «Техногрета» студии Артемия Лебедева «Оптимизация JPEG. Часть 2», работает только на изображениях с контрастной мелкой текстурой (например, мелкий темный текст на светлом фоне) и является относительно сложным.

Как уже было упомянуто, по алгоритму JPEG изображение первым делом переводится в пространство цветов YCbCr, которое описывает его яркость и цветовую разницу. Наиболее важным для человеческого глаза является первый компонент, поэтому, манипулируя с изображением, желательно минимально его затрагивать.

В Photoshop есть цветовой режим Lab, где канал «L» задает яркость, а две хроматические составляющие «a» и «b» описывают, соответственно, положение цвета в диапазоне от зеленого до пурпурного и от синего до желтого. Именно в этом режиме проще всего производить все дальнейшие манипуляции, а монитор хорошо бы предварительно откалибровать.

Для иллюстрации этого метода мы выбрали фотографию сухого завтрака 700×432, которая подходит как нельзя лучше: в ней есть области, разные по яркости и текстуре.

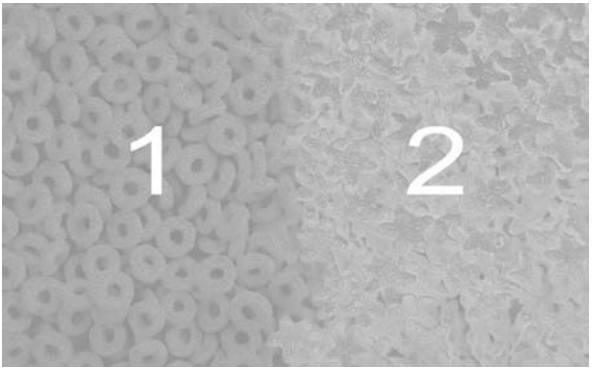
Рис. 3.7. Фотография сухого завтрака поможет нам проиллюстрировать, как меняется размер файла при удалении из него не воспринимаемой глазом информации

До всех манипуляций эта картинка, сохраненная в Photoshop с качеством 55%, занимает 105,5 Кб.

Откроем нашу картинку в Photoshop и перейдем в цветовой режим Lab (Image → Mode → Lab Color), после этого сделаем дубликат картинки, чтобы контролировать процесс (Image → Dublicate). Обе картинки лучше поставить рядом, для наглядности.

У редактируемого изображения посмотрим каналы «a» и «b» (`Ctrl+2`, `Ctrl+3`), на экране будет видна текстура нашего изображения, с двумя выраженными зонами:

Рис. 3.8. Сухой завтрак, канал «b»



Каждую из двух зон нужно обработать отдельно. Переключимся в канал «a» (Ctrl+2), выделим (инструментом «Lasso Tool») поочередно каждую зону и применим к ним фильтр Median (Filter → Noise → Median). Значение Radius надо выбирать минимальное, при котором пропадает структура. Для первой зоны у нас получилось значение 24, для второй — 34.

Включаем все каналы (Ctrl+~), сравниваем с оригиналом и убеждаемся, что изображение стало более насыщенным. Выделяем редактируемые нами блоки и вызываем окно Levels (Ctrl+L). Сдвигаем средний бегунок влево, пока картинка не становится похожей на оригинал.

То же самое повторяем для канала «b». После всех манипуляций размер файла при сохранении в JPEG уменьшился на 10%.

3.2.5. Рецепт № 4: удалим все лишнее

Помимо самого изображения формат позволяет хранить произвольное количество метаинформации. Самые известные примеры: комментарий, превью изображения и структурированные, расширенные данные, известные как EXIF.

В EXIF многие программы записывают интересную информацию, не являющуюся необходимой для просмотра изображения. Например, фотокамера может записать туда режим выдержки или координаты съемки. Отсюда следует самый простой способ уменьшения размера JPEG-изображения: удаление из файла всей метаинформации. Иногда это позволяет добиться существенного уменьшения файла.

Для программ, оптимизирующих уже готовые файлы, были взяты 360 картинок из блога одного из авторов, объемом в 10,94 Мб. В части файлов присутствуют комментарии и другие метаданные. Программ для оптимизации уже готовых JPEG (без изменения их качества) оказалось совсем немного. Кроме удаления лишних данных они производят небольшую оптимизацию кодов Хаффмана, что тоже помогает уменьшить размер.

Программа	Результат
PureJPEG 1.0	10,82 Мб (1%)
jpegtran 08-Feb-2005	**10,67 Мб (2,4%)**
Jpegoption 1.2.2	10,78 Мб (1,7%)

Лидирует тут jpegtran набора утилит Independent JPEG Group, но результат ее работы трудно назвать значительным. Так что этот метод следует рассматривать как дополнительный и завершающий по отношению ко всем прочим.

3.2.6. Резюме

Рассмотрев различные рецепты оптимизации, мы можем сделать следующий вывод. Графические редакторы, позволяющие сохранять изображения в формате JPEG, при одинаковом визуальном качестве дают различный размер изображения; если необходимо еще более уменьшить размер, ручную оптимизацию можно произвести при помощи Xat JPEG Optimizer. Цветовые границы изображений, сохраняемых в этом формате, рекомендуется выравнивать по сетке 8×8 (и 8×16, если используется Photoshop и качество 0-50) , а мелкие контрастные текстуры оптимизировать, минимизировав информацию о цвете вручную. После всех манипуляций изображение нужно обработать утилитой jpegtran.

3.3. Оптимизируем PNG (Portable Network Graphics)

История PNG началась 4 января 1995 года, так что это самый молодой из поддерживаемых всеми браузерами растровых форматов. Формат обеспечивает глубину цвета до 48 бит, полупрозрачность (альфа-канал), гамма-коррекцию изображения, двумерный чересстрочный режим и использует лучшие, по сравнению с GIF, алгоритмы сжатия изображений.

Формат, так же как и GIF, поддерживает индексированную палитру и бинарную прозрачность, что позволяет использовать PNG для хранения как полноцветных изображений без искажений, так и изображений с малым количеством цветов.

Разработанный для хранения статичного изображения, формат не поддерживает анимацию, но попытки исправить текущее положение ве-

щей предпринимаются. Сейчас стандартом де-факто становится APNG — довольно удачное расширение PNG; что интересно — изображения в этом формате отображаются в браузерах, которые его не поддерживают, как статичные изображения.

На текущий момент APNG поддерживают Opera (с версии 9.5) и Mozilla Firefox (версия 3.0 и выше).

3.3.1. Проблемы отображения PNG в браузерах

К сожалению, использование формата тормозит один из распространенных браузеров — Internet Explorer. Все версии этого браузера, начиная с четвертой (где впервые появилась поддержка), имеют проблемы в отображении PNG:

- Internet Explorer 4.0 падает при открытии определенных PNG;
- Internet Explorer 5.0 и 5.01 не отображают PNG через тег `object`;
- версия 5.01 иногда неверно отображает изображения с черным (или темно-серым) задним фоном в Windows 98;
- больше всего проблем по поддержке PNG у версии 6.0: проблемы с открытием файлов размером 4097 и 4098, файлов с пустыми секциями IDAT (которые были исправлены в SP1), проблемы с отображением полупрозрачности (они решаются при помощи специальных приемов, но это сказывается на производительности);
- Internet Explorer 7.0 и 8.0 не поддерживают альфа-канал вместе с фильтром полупрозрачности;
- ни одна версия этого браузера не поддерживает полностью гамма-коррекцию и коррекцию цвета.

Впрочем, у остальных браузеров тоже не все гладко, хотя список их прегрешений не столь впечатляет:

- реализация коррекции цвета PNG появилась только в Firefox 3, где она была отключена в настройках вплоть до версии 3.5;
- WebKit (на котором основаны Safari и Google Chrome) имел в различных версиях проблемы с отображением полупрозрачности в PNG;
- в Opera поддержка полупрозрачности появилась только в версии 6.0, а проблемы с гамма-коррекцией сохранялись до версии 6.1, в современных же версиях есть всего один недостаток, который вряд ли может считаться фатальным: чересстрочные PNG, использующие полупрозрачность, показываются только после полной загрузки.

Как было сказано выше, Internet Explorer до седьмой версии не поддерживал полупрозрачность в изображениях PNG. Этот недостаток ис-

правляется одним из двух методов: использование фильтра AlphaLoader, который позволяет загружать альфа-канал отдельно, или подключение изображения через VML, где проблемы нет.

Оба метода обладают недостатками. Применение фильтров вообще и AlphaLoader в частности приводит к повышенному расходу памяти и ухудшению времени отклика браузера. Для иллюстрации проблемы можно привести исследование Стояна Стефанова (http://ap-project.org/Russian/Article/View/83/Russian_translation/), где рассматривалось сто иллюстраций с фильтром и без него. Использование фильтров увеличило время отрисовки страницы в 27 (!) раз, а размер задействованной памяти — в 78(!) раз.

Использование VML несет с собой другие проблемы: изображения, загруженные через VML, иногда не кэшируются, требуют специальной разметки и их сложно применять как фоновое изображение, кроме того, скорость инициализации оставляет желать лучшего.

Так что для старых версий Internet Explorer рекомендуется пользоваться, где возможно, PNG с индексированными цветами и бинарной прозрачностью вместо полупрозрачности, прибегая к VML и AlphaLoader только в крайних случаях. Кстати, в последнем случае будет полезно посмотреть на библиотеку `DD_belatedPNG` (http://dillerdesign.com/experiment/DD_belatedPNG/).

Один недостаток отображения файлов PNG в браузерах связан не с ошибками, а с добрыми намерениями авторов формата. Дело в том, что одни и те же цвета выглядят в различных операционных системах по-разному.

Разработчикам PNG это показалось неправильным. Они ввели в формат несколько полей, куда записываются характеристики машины и операционной системы, на которой создано изображение. В дальнейшем, если параметры оборудования или ОС на машине, где изображение показывается, отличается, производится специальная коррекция.

Идея хорошая, но в вебе помимо PNG используются и другие форматы, в которых такая коррекция не применяется. Это приводит к тому, что стык PNG с JPEG может отличаться по цвету в Mac OS, если оба изображения готовились под Windows.

Можно не использовать PNG вообще и не задумываться о таких проблемах, альтернатива — удалить из файла всю цветокорректирующую информацию. Утилита optipng делает это по умолчанию, но можно для той же цели применить устаревшую во всех отношениях утилиту PNGcrush:

```
pngcrush -rem cHRM -rem gAMA -rem iCCP -rem sRGB infile.png ↵
outfile.png
```

К счастью, новые версии Adobe Photoshop (начиная с CS4) не записывают в файл эту информацию, GIMP же не заполняет поле gAMA, без которого браузеры коррекцию не производят.

3.3.2. Выбор типа PNG

PNG, как уже частично рассматривалось выше, умеет хранить несколько типов изображения: полутоновое изображение (оттенки серого) с глубиной цвета 8 бит, цветное индексированное (восьмибитовая палитра цветов глубиной 24 бита) и полноцветное изображение с глубиной 48 бит.

Кроме того, в PNG есть два типа прозрачности: полупрозрачность (256 уровней альфа-канала) и бинарная прозрачность (как в GIF — либо 100% прозрачно, либо нет).

Большей частью выбор типа изображения очевиден, но есть тонкости.

Изображение, содержащее исключительно оттенки серого (так называемое «черно-белое»), выгоднее всего сохранять в специально предназначенном для этого формате с глубиной цвета 8 бит. Из-за того, что компоненты цвета записываются меньшим количеством байт, файл занимает меньше места.

В редакторе Adobe Photoshop для сохранения изображения в файле этого типа нужно воспользоваться инструментом «Save As...» (`Shift+Ctrl+S`), а не «Save for Web» (`Alt+Shift+Ctrl+S`), последний этот тип формата не поддерживает. Конечно, чтобы Photoshop понял, что нужно сохранить именно с такой глубиной цвета, режим изображения должен быть «Grayscale» (Image → Mode → Grayscale).

Рис. 3.9. Слева фотография, сохраненная с глубиной цвета 24 бита (349 Кб), справа — 8 бит (253 Кб)

Редактор GIMP также умеет сохранять такие изображения правильно. Если выбранный вами редактор этого делать не умеет — не беда, мож-

но использовать утилиты OptiPNG или PNGout (их мы еще рассмотрим ниже), умеющие понижать глубину цвета:

```
optipng -o7 dog.png -out dog-optimized.png
pngout /c0 dog.png dog-optimized.png
```

Другая особенность относится к сохранению картинок с малым количеством цветов и без прозрачности — некоторые изображения занимают меньше места в PNG с глубиной в 24 бита, чем при использовании индексированной палитры. В основном это касается градиентных изображений.

Сергей Чикуенок, специалист по оптимизации изображений, советы которого легли в основу этой главы, объясняет это тем, что в полноцветных изображениях цвет описывается тремя байтами (RGB), тогда как в формате с индексированными цветами — четырьмя (RGB + один байт в палитре). Помимо этого само изображение может быть настолько эффективно сжато, что добавление к файлу таблицы палитры (которая не сжимается) может чувствительно увеличить его размер.

Существенную экономию дает в некоторых случаях малоизвестная возможность PNG — использование полупрозрачности в файлах с индек-

Рис. 3.10. PNG с индексированной палитрой и полупрозрачностью в Internet Explorer 6.0 и в остальных браузерах

сированной палитрой, где каждый элемент палитры описывается не тремя составляющими (RGB), а четырьмя — RGBA (красный, зеленый, голубой и альфа-канал).

Редактор Photoshop, опять же, не поддерживает этот формат, но optiPNG автоматически преобразует изображения с полупрозрачностью, содержащие 256 цветов и меньше, в файл такого типа. Полноцветные изображения при желании можно преобразовать утилитой «Improved PNGNQ», которая уменьшает количество цветов до указанного (от 16 до 256) и сразу преобразует файл в рассматриваемый тип.

Интересно поведение Internet Explorer 6.0 при отображении файлов такого типа. Как известно, полупрозрачность в PNG он не поддерживает, а такие файлы, без применения хаков, отображает как GIF — с бинарной прозрачностью, что лучше, чем серый цвет, которым IE заполняет прозрачные области обычных PNG.

3.3.3. Автоматическая оптимизация

Давайте отвлечемся от приемов ручной оптимизации и рассмотрим инструменты, которые помогут произвести оптимизацию автоматическую. Насколько большой выигрыш можно получить, если взять обычные изображения, сохраненные в графическом редакторе, и обработать такими утилитами?

Для тестирования были выбраны три утилиты: PMT (PNGcrush), optiPNG и PNGout. Поскольку pngout не умеет сама перебирать методы оптимизации, для нее был написан небольшой командный файл, использующий для перебора optiPNG (метод взят с блога blog.ad.by):

```
@ECHO OFF
IF "%1"=="" EXIT 1
FOR /F "usebackq delims==,IDAT tokens=5" %%i in ↵
(`optipng -o4 -full -sim "%1"`) DO SET f=%%i
IF "%2"=="" (
    pngout /k0 /n2 /f%f:~1,1% "%1" || EXIT 3
) ELSE (
    pngout /k0 /n2 /f%f:~1,1% "%1" "%2" || EXIT 3
)
```

Для тестов мы взяли по 10 иллюстраций каждого типа: фотографии (глубина цвета — 24 бита), полноцветные синтетические (глубина цвета — 24 бита), синтетические с градациями серого (8 бит) и с индексированной палитрой (8 бит).

Программа	Фото	Полноцветное	Серое	Индексированное
optiPNG 0.6.3	4,52 Мб (5%)	4,75 Мб (9%)	2,67 Мб (21%)	510,01 Кб (9%)
PNGout 21 Jun 2009	**4,38 Мб (8%)**	4,55 Мб (13%)	2,57 Мб (24%)	495,13 Кб (11%)
PNGcrush 1.6.15	4,55 Мб (4%)	4,99 Мб (4%)	3,20 Мб (5%)	515,21 Кб (8%)
PNGoutwin 1.0.1	4,54 Мб (4%)	**4,07 Мб (22%)**	3,07 Мб (9%) *	**484,87 Кб (13%)**

* неожиданно плохие результаты PNGoutwin на изображениях серого объясняются просто: программа не справилась с понижением глубины цвета до 8 бит; если понизить глубину заранее (выше было рассмотрено, как это делается), то PNGoutwin справляется с заданием успешнее всех — 2,56 Мб (24%).

По всей видимости, на первое место стоит поставить платную программу PNGoutwin, на второе — PNGout (в паре с optiPNG для подбора параметра оптимизации изображения), а аутсайдером по качеству является PNGcrush.

3.3.4. Оптимизация через Lossy GIF

Конечно, для оптимизации изображения PNG можно применять все те же методы, что и для GIF, — например, уменьшение количества цветов; рассматривать здесь дважды один и тот же материал смысла не имеет.

Даже настройки сохранения PNG с индексированной палитрой в Adobe Photoshop те же, но не хватает одного инструмента — «Lossy». Как вы, наверное, помните из части про оптимизацию GIF, этот инструмент вносит горизонтальные искажения, и изображение лучше сжимается.

В случае PNG это правило тоже иногда действует, но соответствующего инструмента нет. Выход один — сохранить изображение сначала в GIF, а потом преобразовать в PNG.

3.3.5. Постеризация

Другой искажающий способ оптимизации — постеризация. Если говорить бытовым языком, то постеризация — это изменение количест-

ва используемых цветов. Чем меньше уровень постеризации, тем меньше цветов в изображении и тем крупнее одноцветные области.

В Photoshop инструмент для постеризации изображения вызывается Image → Adjustments → Posterize, в GIMP — Color → Posterize.

Благодаря снижению количества цветов изображение лучше упаковывается и занимает меньше места. Особенно эффективен этот прием для фотографических изображений, где пастеризованные области обычно незаметны.

Способ очень простой, но есть неудобство: результатом постеризации управлять невозможно, но есть немало более управляемых способов снизить количество цветов в изображении.

3.3.6. Простое снижение количества цветов в Photoshop

Первым делом нам понадобится плагин PhotoFreebies plugin suite, скачать который можно с сайта производителя.

Рис. 3.11. Исходное изображение (226,3 Кб)

Откроем исходное изображение в Photoshop и создадим его дубликат (Image → Duplicate), удалим из дубликата информацию о прозрачности (Filter → Photo Wiz → Remove Transparency) и сменим режим на «Индексированные цвета» (Image → Mode → Indexed Color).

В появившемся окне выбираем 50 цветов: этого достаточно, чтобы львенок смотрелся приемлемо, а также убираем галочку «Preserve Exact Colors». Там же выбираем режим Diffusion параметра Dither и устанавливаем Amount в 30% (так как цветов в этом изображении мало, то эта величина ни на что не влияет, в других изображениях ею можно контролировать качество замены недостающих цветов).

Следующий шаг — меняем режим обратно на RGB и копируем изображение в оригинальный файл в отдельный слой, который должен располагаться поверх оригинального. Используем нижний слой как маску (Alt+Ctrl+G) и сохраняем полученное изображение.

В данном случае картинка с львенком стала занимать 164,5 Кб, сократившись на 27%.

3.3.7. Программа Color quantizer и маска повышения качества

Изображения неоднородны по детализации, и было бы замечательно, если при снижении количества цветов программе можно было бы указать, что какой-то области нужно уделить «больше внимания», то есть выделить большее количество цветов, чтобы эта область выглядела лучше.

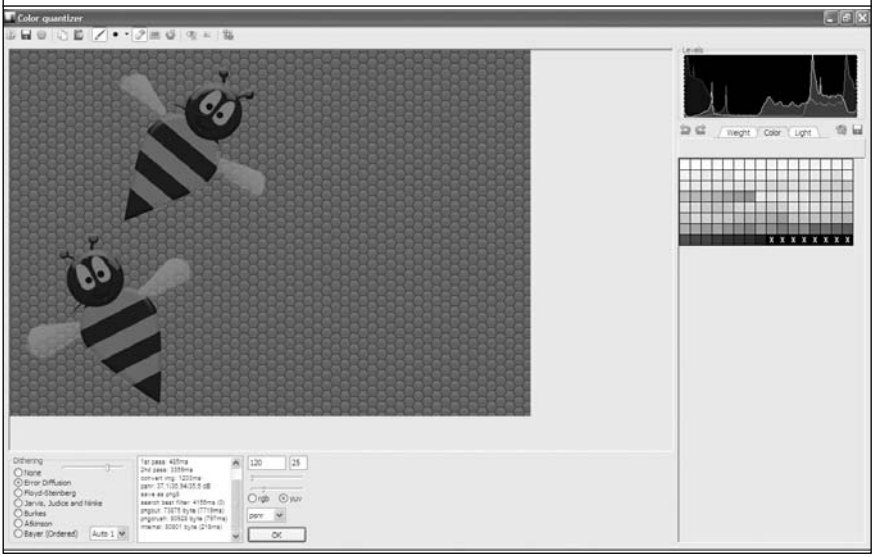
На наш взгляд, самая простая и эффективная программа, позволяющая это делать, — Color quantizer (http://x128.ho.ua/). Программа все еще совершенствуется и находится в стадии тестирования, но уже позволяет получать очень хорошие результаты в пределах 256 цветов.

Исходное загруженное изображение занимает 394 Кб в полноцветном PNG и 116,8 Кб с индексированной палитрой.

Некоторые инструменты, предоставляемые программой, нуждаются в пояснении. В центре — исходное изображение, клик левой клавишей мыши на нем показывает картинку в исходном качестве, что очень удобно для оценки искажений. На скриншоте программа находится в режиме редактирования маски повышения качества (включается кисточкой в верхней панели инструментов), светлыми мазками указаны места, где снижать качество нельзя. Толщину мазка регулирует инструмент справа от кисточки, подчищать маску можно ластиком.

Внизу слева находится выбор алгоритма замещения недостающих цветов, сообщается, что лучше всего работает «Флойд—Штейнберг», хуже всего — «Байер».

Рис. 3.12. Color quantizer, редактирование маски повышения качества



Правее идет поле отладочной информации, еще правее — два поля, выбор количества цветов и поле, указывающее, сколько процентов палитры резервировать под второй проход. Второе поле сугубо экспериментальное, но неясным образом влияет на качество изображения.

Чуть ниже находятся два ползунка: первый влияет на качество преобразования (чем более влево, тем качественнее и тем медленнее работает программа), второй — экспериментальный, чем больше цветов задано, тем больше его нужно сдвинуть влево, но точного алгоритма тут нет.

Переключатель «RGB/YUV» следует поставить в первое положение для искусственных изображений и во второе для фотографий. Переключатель «mse/msad/psnr» несет отладочную функцию и для наших целей бесполезен. Кнопка «OK» запускает обработку изображения, и ее нужно нажимать каждый раз, когда хочется увидеть, к чему привели изменения в настройках.

Справа от изображения видны уровни изображения и его текущая палитра, палитру можно отсортировать по количеству использований цвета (Weight), по оттенку (Color) или яркости (Light). На каждый элемент палитры можно кликнуть мышкой, левая клавиша показывает, где на картинке используется цвет, правая — выводит меню, где элемент можно заменить, удалить и так далее.

Итак, вернемся к нашему изображению. После снижения количества изображения до 120 цветов качество изображения вполне приемлемо, если бы не некоторые детали — полупрозрачные крылья, блики на усиках и боках пчел заметно исказились, поэтому мы наносим на них маску повышения качества. После проделанной операции рассматриваем остальные детали; если искажения в них почти незаметны и не требуют нанесения маски, но выглядят все же не идеально, можно попробовать изменить параметр «Dithering», возможно, один из алгоритмов подойдет больше текущего.

В итоге изображение пчел стало занимать 66,48 Кб, что на 43% лучше, чем предыдущий результат.

На этапе подготовки этой книги вышла новая версия программы.

Появилось несколько интересных улучшений: полный перебор всех параметров PNGout (включается отдельной кнопкой с двумя дискетами на панели инструментов), работает он очень долго, но эффективно, появился усредняющий фильтр («метла» в панели инструментов), который усредняет соседние цвета, что способствует лучшему сжатию изображения. Фильтр позволяет указать алгоритм усреднения (например, «только по вертикали»), его «жесткость» и степень усреднения. Жесткий фильтр хорошо подходит для фотографий, мягкий может применяться для снижения шума в изображении.

Еще одно долгожданное улучшение — изменение масштаба рабочей области. Этот инструмент доступен в меню правой клавиши мыши, также масштабом можно управлять клавишами «+» и «–» на цифровой клавиатуре.

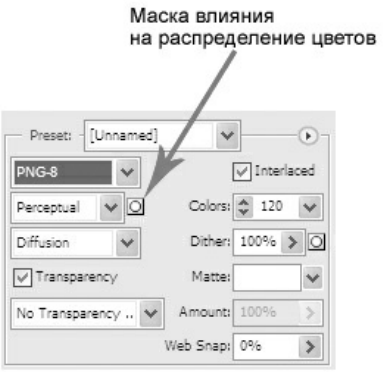
3.3.8. Маска влияния в Adobe Photoshop

В Photoshop до версии CS4 был инструмент, позволяющий достигать того же эффекта, что и маска влияния качества в программе Color quantizer, но более гибко. Превосходство инструмента из Photoshop в том, что можно не просто указать, что выделенная область должна выглядеть качественнее, но уточнить, насколько.

На рисунке показано, где расположен инструмент, включающий маску влияния; доступен он только в режиме PNG-8. Рядом с параметром «Dither» расположен похожий инструмент — он позволяет задавать интенсивность цветового смешивания, которое используется для того, чтобы узором из пикселей имеющихся цветов хоть как-то сымитировать недостающие.

Инструмент, задающий маску, позволяет указывать канал (Channel), в котором записано, насколько много нужно уделять внимания этому уча-

Рис. 3.13. Маска влияния на распределение цвета в Photoshop



стку. Для маски влияния на распределение цветов значение имеет яркость участка в канале: чем выше яркость, тем больше цветов Photoshop постарается сохранить на этом участке.

Открываем наше изображение и в палитре «Channels» создаем новый канал, назовем его «color». Включаем канал «RGB», чтобы видеть само изображение, выбираем инструмент «Кисточка» и прикидываем, какие участки должны выглядеть наиболее качественно. Очевидно, что полупрозрачным крыльям и более темным местам изображения можно уделить меньше внимания, чем теням на светлых полосках пчел.

Сопоставляя таким образом важность, раскрашиваем наше изображение и время от времени переключаемся в «Save for Web» (`Alt+Shift+Ctrl+S`), чтобы увидеть результат. Работа куда более кропотливая, чем при использовании программы Color quantizer, но результат — точнее, а это может дать более впечатляющую экономию.

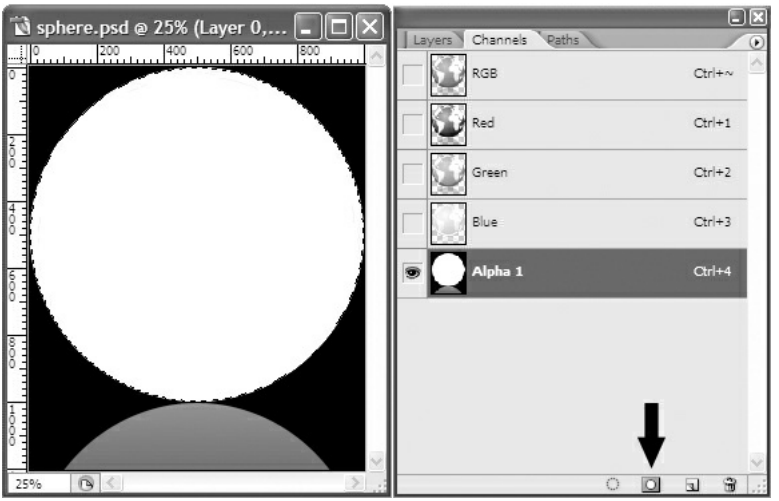
3.3.9. Уменьшаем детализацию

Если изображение содержит полупрозрачные области, то, вероятно, в них находятся детали, которые вряд ли видны глазу, и чем область прозрачнее, тем менее важно, насколько качественно она прорисована.

Как видно, в изображении есть полностью прозрачная область вокруг шара и его полупрозрачное отражение. Кликаем с зажатой клавишей Ctrl на миниатюре изображения в палитре «Layers», переходим в «Channels» и нажимаем на кнопку «Save selection as channel» из ряда в низу палитры каналов (рис. 3.15).

Рис. 3.14. Образец для наших экспериментов, отражающийся в «мокром полу» (2,44 Мб)

Рис. 3.15. Создаем канал из выделенного участка



Теперь нам нужно отделить пиксели, которые в изображении наиболее заметны. Для этого снимаем выделение (`Ctrl+D`), инвертируем канал (`Ctrl+I`) и воспользуемся диалогом Threshold (Image → Adjustments → Threshold).

Меньшее значение параметра Threshold сохранит меньше деталей, а значит — уменьшит размер файла.

Кликаем с зажатой клавишей Ctrl на миниатюре канала в палитре «Channels» и выбираем фильтр «Median» (Filter → Noise → Median). Его значение опять же нужно подобрать вручную; чем больше значение, тем меньше деталей сохранится и тем меньше будет размер файла.

При значениях Threshold, равному 138, и Median, равному 4, размер нашего файла уменьшился на 440 Кб (18%) без заметных глазу потерь.

3.3.10. Невидимое — не значит несуществующее

Редактор Adobe Photoshop имеет одну малоизвестную особенность, иногда сильно увеличивающую размер файла PNG.

Попробуйте открыть в Photoshop какую-нибудь фотографию, выделить любым инструментом (например, «Lasso Tool») любой сложный контур, скопировать его в новый документ, после чего сохранить его через «Save For Web» (`Alt+Shift+Ctrl+S`).

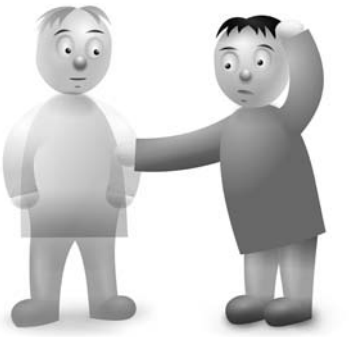


Рис. 3.16. Инструментом «Elliptical Marquee Tool» вырезаем сложный контур

Логично было бы предположить, что полностью прозрачные части заполнены одним прозрачным цветом, чтобы изображение лучше упаковывалось. Оказывается, это не так. Откроем сохраненное изображение

программой XnView, в режиме просмотра выбираем «Рисунок» → «Преобразовать в цветное», в появившемся окошке выбираем «Применить».

Рис. 3.17. Результат обработки нашего изображения программой XnView

Как видно, Photoshop оставил в невидимых областях много лишнего. Впрочем, избавиться от этого вполне нам по силам.

Если ваше изображение не содержит полупрозрачных областей, годится следующий простой способ: Ctrl+клик на миниатюре изображения в палитре «Layers», инвертируем выделение (Ctrl+Shift+I), удаляем выделенное (клавиша «Delete») и снова сохраняем наше изображение (Alt+Shift+Ctrl+S).

В случае, если полупрозрачные области присутствуют, придется потрудиться немного больше: Ctrl+клик на миниатюре изображения в «Layers», переходим в режим быстрой маски (клавиша «Q»), в диалоге Image → Adjustments → Threshold переставляем бегунок в крайнее левое положение и нажимаем «OK», выходим из режима маски (опять «Q») и инвертируем выделение (Ctrl+Shift+I). После чего заливаем выделенную область черным цветом (Edit → Fill), снова инвертируем выделение (Ctrl+Shift+I) и добавляем к слою маску (иконка «Add Layer Mask» внизу палитры слоев). Изображение можно сохранять.

В нашем случае экономия составила 7%. Хотим сразу предостеречь, что этот прием иногда не уменьшает, а увеличивает размер файла. Дело в том, что формат PNG исключительно хорошо сжимает участки определенных типов (например — градиенты), так что если за границей видимого осталась какая-то часть градиента, то она может не мешать, а способствовать сжатию.

Поэтому хорошая практика — попробовать сохранить оба варианта и выбрать меньший по размеру.

3.3.11. Разделение непрозрачных и прозрачных областей

Суть следующего приема — в разделении изображения на два: в первом, в индексированной палитре, сохраняется само изображение, во втором, полноцветном — полупрозрачные области. В браузере оба изображения накладываются друг на друга, например, второе указывается как фон первого.

В качестве примера возьмем стилизованный земной шар, отражающийся в мокром полу, который мы использовали выше.

Открываем изображение в Photoshop. Если оно состоит из нескольких слоев, то его необходимо слить в один. Далее выделяем объект в слое, кликнув с зажатой клавишей Ctrl на миниатюре в палитре «Layers», переходим в палитру «Channels» и создаем новый канал, кликнув на «Save selection as Channel».

Снимаем выделение (Ctrl+D) и вызываем инструмент «Threshold» (Image → Adjustments → Threshold), перетаскиваем бегунок в крайнее правое положение. Теперь у нас есть маска для выделения непрозрачных областей.

Кликаем с зажатой клавишей Ctrl на миниатюре в маске Alpha 1, переходим в палитру «Layers», выбираем слой с изображением и выполняем команду «Layers via Cut» (Shift+Ctrl+J). Теперь у нас два слоя, которые нужно сохранить в отдельных файлах. Непрозрачные пиксели сохраняем в PNG с индексированной палитрой (или в GIF), при возможности выполняя оптимизацию (например, постеризацию), прозрачные — в полноцветном PNG.

В нашем случае суммарный объем двух получившихся картинок составил 1 Мб, а экономия — 59%.

3.3.12. Резюме

Многообразие формата PNG дает широкий простор для оптимизации. Прежде всего, следует внимательно выбрать тип формата: использовать полноцветный, оттенки серого или индексированную палитру? Можно ли снизить количество цветов или детализацию? Нельзя ли разделить полупрозрачные области и непрозрачные? Если применялся Photoshop, не осталось ли чего-нибудь лишнего в полупрозрачных областях? После всего следует воспользоваться утилитами оптимизации.

3.4. Оптимизируем SVG (Scalable Vector Graphics)

Векторный формат SVG является наследником двух форматов — VML (Vector Markup Language), который был разработан фирмами Microsoft,

Macromedia, Autodesk, Hewlett-Packard и Visio Corporation, а также PGML (Precision Graphics Markup Language), совместно придуманный фирмами Adobe Systems, IBM, Netscape и Sun Microsystems.

Работа над SVG началась еще в 1999 году, то есть это достаточно старый формат, но широкую известность он приобрел только недавно, когда его ограниченная поддержка появилась в современных браузерах. К сожалению, Microsoft не включила поддержку SVG в недавно вышедший браузер Internet Explorer 8.0, но есть несколько плагинов, позволяющих браузерам Microsoft показывать изображения SVG.

Формат SVG является единственным общепринятым форматом векторной графики в вебе, но интересен еще и тем, что поддерживает анимацию, а также может быть изменен и создан браузером, поскольку представляет собой чистый XML.

Несмотря на то, что формату исполнилось десять лет, он не настолько изучен в плане оптимизации как остальные общепринятые форматы. Из интересных исследований, пожалуй, можно отметить документ фирмы Nokia «S60 Platform: Vector Graphics Optimization» и статью блогера Джоса Хёза, где он пытается автоматизировать часть советов Nokia и некоторые из своих идей на этот счет.

Каких-либо комплексных инструментов для оптимизации SVG нам найти не удалось. Одним из авторов книги была предпринята успешная попытка развить идеи Хёза и сделать такой инструмент самостоятельно.

К сожалению, данный инструмент панацеей не является, хотя иногда показывает потрясающие результаты (уменьшение файла на 83%), но некоторые вещи все равно нужно оптимизировать вручную.

3.4.1. Оптимизация вручную

Вот несколько советов, которые помогут вам уменьшить размер файла и повысить удобство пользователя при взаимодействии с интерактивными SVG.

- Все ссылки, которыми пользователь может воспользоваться, размещайте как можно ближе к началу документа, тогда они будут доступны уже на ранних этапах загрузки.

- Если браузеры, которые будут отображать иллюстрацию, поддерживают CSS-стили для SVG, то воспользуйтесь всеми преимуществами такого подхода: группировкой и наследованием свойств, в противном случае присвойте общие свойства родительскому тегу `g`.
- Используйте возможности тега `path` для более компактной записи: относительные координаты, `h` и `v` для вертикальных и горизонтальных линий, `s` и `t` для задания квадратичных и кубических кривых Безье.
- Не забывайте, что SVG предоставляет возможности для многократного использования одних и тех же элементов внутри иллюстрации.
- Используйте фильтры для создания сложных конструкций силами графической системы пользователя.
- Если какие-то примитивы перекрывают друг друга, уберите все, что не видно пользователю — кривые Безье можно заменить линиями, эффекты убрать, контуры «втянуть».
- Попробуйте уменьшить детализацию там, где это не будет заметно: возможно, крошечное отверстие будет смотреться не хуже, если сделать его простым многоугольником, а не кривыми Безье.
- Эффективнее не выкраивать сложные отверстия в фигурах заднего плана, а наложить нужный контур поверх, еще одним слоем.
- Объединяйте в одну соединенные фигуры, имеющие одну и ту же заливку и градиент.
- Старайтесь реже использовать градиент и прозрачность, это сбережет ресурсы клиентской машины, по той же причине избегайте использования перекрывающихся прозрачных областей.

3.4.2. Редакторы для работы с SVG

Для выполнения большинства советов по оптимизации вручную вам потребуется графический редактор. Какие же инструменты присутствуют на рынке?

Adobe (Macromedia) FreeHand — удобный, гибкий инструмент, доступны версии для Windows и Mac, но есть два неприятных недостатка: не работает напрямую с SVG и больше не разрабатывается вследствие покупки Macromedia фирмой Adobe, у которой есть свой редактор векторной графики (рис. 3.18).

Adobe Illustrator — один из старейших инструментов, непростой в использовании, не столь удобный, но во многом очень гибкий, умеет работать с SVG. Существуют платные версии для Windows и Mac (рис. 3.19).

Рис. 3.18. Редактор Macromedia FreeHand MX

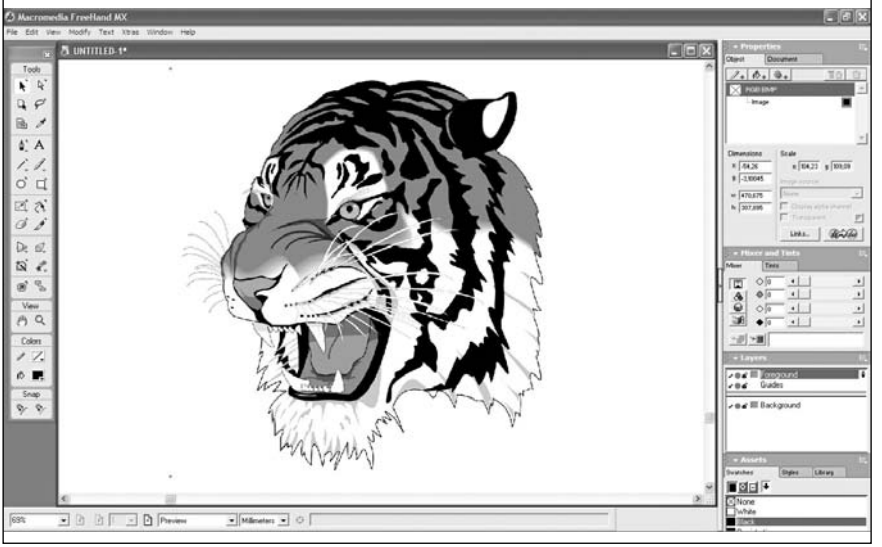
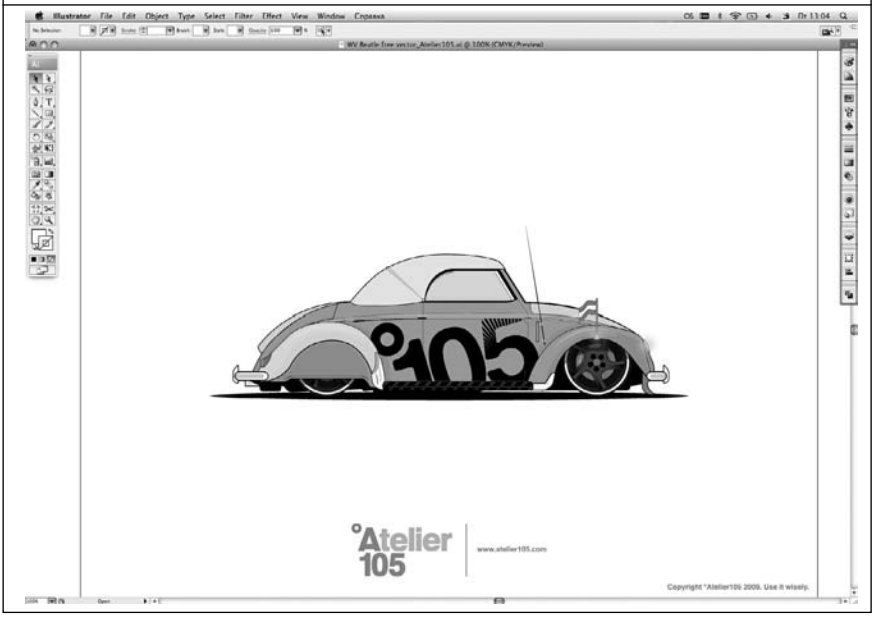


Рис. 3.19. Редактор Adobe Illustrator



Inkscape — популярный, простой в использовании, бесплатный редактор, созданный специально для редактирования SVG. Есть версии для Linux, Mac и Windows (рис. 3.20).

Рис. 3.20. Редактор SVG Inkscape

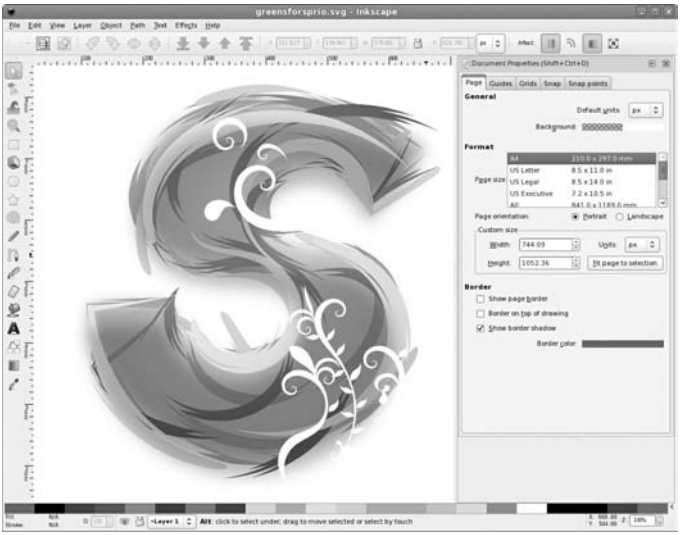
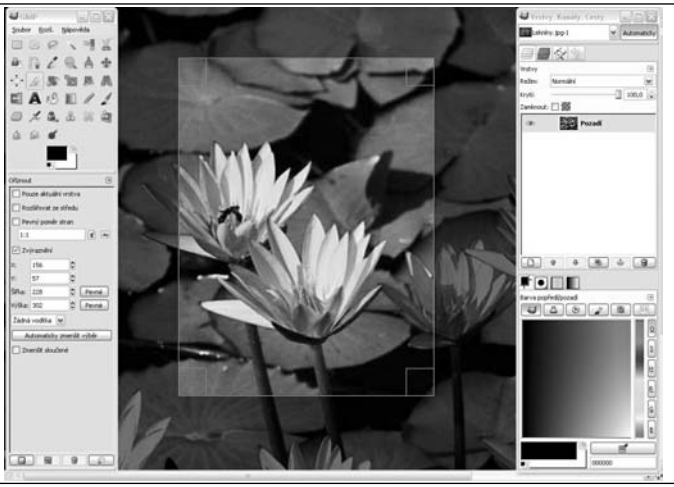


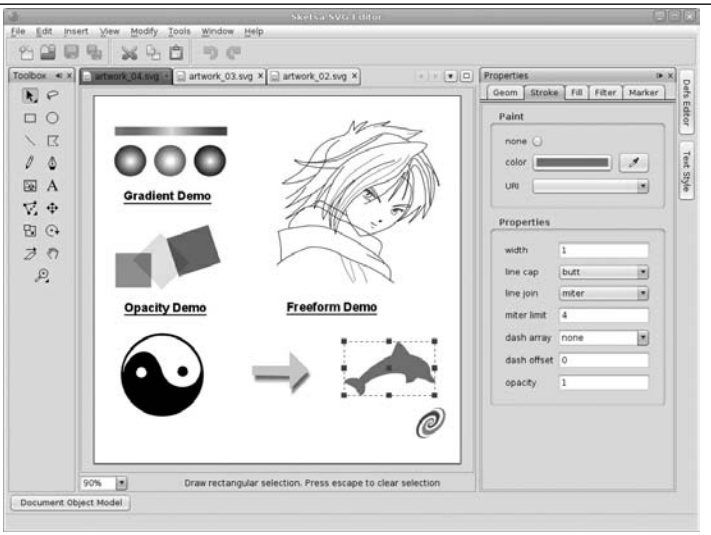
Рис. 3.21. Редактор графики GIMP



GIMP — известный многим бесплатный конкурент Adobe Photoshop, умеет редактировать и векторную графику, хотя справляется с этим хуже специализированных программ. Умеет работать с SVG. Существует в версиях для Linux, Mac, Windows (рис. 3.21).

Sketsa — платный, написанный на Java специализированный редактор SVG (рис. 3.22).

Рис. 3.22. Мультиплатформенный редактор Sketsa



3.4.3. Автоматическая оптимизация

Как уже было упомянуто выше, одним из авторов была написана утилита для автоматической оптимизации файлов SVG. Прототипом послужили две небольшие программы на Java, взятые с сайта Джоса Хёза.

Программа написана на языке PHP, скачать ее можно по адресу http://bolknote.ru/files/svgcrush.phps

У программы есть несколько ключей настройки, которые по умолчанию установлены на максимальную оптимизацию. Единственный ключ, который необходимо задать программе, — -f, он указывает имя обрабатываемого файла. Результат выдается в стандартный вывод и может быть перенаправлен в файл стандартной конструкцией:

```
php svgcrush.php -fpicture.svg > optimized.svg
```

Оптимизация производится в несколько этапов.

1. Поскольку SVG — это XML, то на первом этапе удаляются пробельные символы, символы перевода строк и комментарии, которые облегчают понимание XML человеком, производится замена пустых парных открывающих и закрывающих тегов на их короткий эквивалент.
2. Заметная часть изображений создана редактором Inkscape, он оставляет внутри SVG массу специфичных атрибутов, которые не использует браузер. На этом этапе и частично на следующих происходит удаление атрибутов, тегов и пространств имен этого редактора.
3. Для атрибута `d` тега `path` производится снижение точности значений и исключаются пробелы там, где это допустимо. Редакторы оперируют 5—6 цифрами после запятой, тогда как использование одного разряда десятичной дроби дает незначительные искажения, а два разряда дают идеальное изображение. На этом же этапе удаляются теги `path` с пустым атрибутом `d`, так как это отключает визуализацию элемента.
4. Следующий этап — оптимизация атрибута стилей («style»), удаляются лишние пробелы, удаляются свойства, значения которых равны их значениям по умолчанию, удаляются свойства, относящиеся к «fill» и «stroke», если заданы стили, отменяющие действие этих свойств. Значения цвета, если это возможно, приводятся к более компактной форме («#aabbcc» становится «#abc»).
5. Обрабатываются все числа в атрибутах — урезаются незначащие нули справа, в чисто числовых атрибутах удаляется ноль слева, до точки («0.05000» становится «.5»).
6. Удаляются неиспользуемые атрибуты `id` тегов. Если в иллюстрации есть теги `<script>` или `<style>`, то этот шаг пропускается.

Программа, вероятно, будет совершенствоваться и дальше. Например, можно заменять числовое значение некоторых цветов на их английское название, если оно более короткое («tan», «red», «plum»), объединять трансформации, преобразовывать rgb(r,g,b) представление в #rrggbb и так далее.

Формат запуска программы следующий:

```
php svgcrush.php [ОПЦИИ] -fФАЙЛ > РЕЗУЛЬТИРУЮЩИЙ_ФАЙЛ
```

Опции, которые понимает программа:
-fФАЙЛ — файл, который будет обработан;
-rЧИСЛО — точность чисел (см. этап № 3) в атрибуте «d», тега `path`, от 1 до 9, по умолчанию — 1;
-D — не удалять атрибуты со значениями по умолчанию (частично исключает этап №4);
-C — не оптимизировать значения цветов (опять же, касается этапа № 4);
-I — не удалять неиспользуемые ID (исключает этап № 6);
-F — форматировать XML (частично исключает этап № 1).

Для тестов эффективности мы использовали Open Clip Art Library версии 0.18 — архив бесплатных SVG. Были обработаны 2875 изображений, результаты следующие: процент оптимизации варьируется в пределах от 1% до 83%, среднее значение — 50%. В основном изображения, которые не удалось оптимизировать, содержат растр, импортированный в SVG.

3.4.4. Используем gzip

Все современные браузеры поддерживают прозрачную распаковку контента, сжатого методами gzip и deflate. SVG — не исключение. Иллюстрация в этом формате (как текстовый XML-файл) прекрасно сжимается — по статистике Adobe Systems, сжатые файлы занимают в среднем на 50-80% меньше места, чем несжатые.

Способы настройки веб-серверов для работы со сжатым контентом были рассмотрены во второй главе.

3.5. Средства онлайн-оптимизации

Определенный интерес представляют сайты, позволяющие произвести автоматическую оптимизацию изображений. Хотя они не дают возможности контролировать процесс, но обладают неоспоримым преимуществом: все, что вам нужно для того, чтобы ими воспользоваться, — доступ в Интернет.

К настоящему времени существуют два сервиса, которые можно рекомендовать к использованию: принадлежащий Yahoo! сервис Smush.it

(http://smush.it) и punyPNG (http://punypng.com), который разрабатывает Конрад Чу. И тот и другой оптимизируют все три общепринятых формата, статичные GIF с большой вероятностью будут сконвертированы в PNG с индексированной палитрой (если такое преобразование даст выигрыш в размере).

На данный момент punyPNG оптимизирует изображения лучше, в частности, из-за применения более современных утилит для оптимизации PNG (Smush.it использует устаревший PNGcrush в режиме полного перебора) и техники, описанной в разделе 3.3.10, для удаления лишнего из полностью прозрачных областей. Впрочем, ситуация еще может измениться.

Тестовая таблица с блога «Gracepoint After Five» (http://www.gracepointafterfive.com/punypng-benchmarks) с результатами оптимизации девяти PNG-файлов обоими онлайн-сервисами и двумя популярными утилитами оптимизации PNG:

	PunyPNG	**Smush.it**	**OptiPNG**	**ImageOptim**
Ask Image 01	**22%**	20%	20%	20%
Ask Image 02	**46%**	39%	40%	2%
Ask Image 03	**8%**	7%	7%	7%
Ask Image 04	**35%**	1%	33%	34%
Blocks	**58%**	0%	0%	0%
Butterfly	**41%**	1%	1%	1%
Facebook	**63%**	36%	61%	61%
NASCAR	**37%**	28%	28%	28%
Yahoo!	**17%**	9%	9%	9%
Экономия:	**41% (112 Кб)**	14% (37 Кб)	19% (51 Кб)	20% (53 Кб)

Помимо веб-интерфейса сервисы предоставляют API. На данный момент компания Yahoo! закрыла API для публичного использования, но им все еще можно воспользоваться. API punyPNG находится в стадии закры-

того бета-тестирования (и может измениться), Конрад Чу любезно согласился предоставить нам доступ к тестовой версии.

Для того чтобы произвести оптимизацию изображения через API, вам нужно разместить его где-нибудь в интернете и узнать его URL. Именно URL используется в качестве источника данных. В обоих сервисах входные данные представлены как GET-параметры, а ответ производится в формате JSON.

Smush.it

URL, на который нужно обратиться: http://smushit.eperf.vip.ac4.yahoo.com/ysmush.it/ws.php

Параметры:

- `img` — URL изображения, которое нужно оптимизировать;
- `id` — необязательный параметр, передается в ответ без изменений, может быть использован для любых ваших целей;
- `callback` — необязательный параметр, имя функции JavaScript, которая будет вызвана (в качестве параметра ей будет передан JSON-объект).

Пример успешного запроса: http://smushit.eperf.vip.ac4.yahoo.com/ysmush.it/ws.php?img=http://www.publicdomainpictures.net/pictures/2000/nahled/1-1222847524s7rl.jpg

```
{
"src":"http:\/\/www.publicdomainpictures.net\/pictures\/2000\/ ↵
nahled\/1-1222847524s7rl.jpg",
"src_size":50219,
"dest":"http:\/\/smushit.zenfs.com\/results\/3144140b\/smush\/ ↵
pictures%2F2000%2Fnahled%2F1-1222847524s7rl.jpg",
"dest_size":49198,
"percent":"2.03"
}
```

Все поля достаточно очевидны, в поле «dest» указан адрес, где можно скачать оптимизированное изображение.

Пример вызова, порождающего ошибку: http://smushit.eperf.vip.ac4.yahoo.com/ysmush.it/ws.php?img=http://bolknote.ru/imgs/2009.09.29.jpg

Такой запрос вернет следующий ответ:

```
{
"src":"http:\/\/bolknote.ru\/imgs\/2009.09.29.jpg",
"src_size":148529,
"error":"No savings",
```

```
"dest_size":-1,
"id":"test"
}
```

Текст ошибки говорит о том, что Smush.it не смог оптимизировать изображение. Другими причинами возникновения ошибочной ситуации могут быть, например, слишком большое изображение (более одного мегабайта) или несуществующий адрес.

PunyPNG

После того как автор закончит бета-тестирование API, получить доступ к нему можно будет, зарегистрировавшись на сайте и получив сорокобитный ключ доступа, который нужно будет указывать при запросах на оптимизацию. В профиле, кстати, можно указать, нужно ли удалять из ваших файлов информацию EXIF.

Интересной особенностью сервиса является возможность пакетной обработки — оптимизированные изображения можно скачать единым ZIP-архивом.

URL, на который нужно обратиться: http://www.gracepointafterfive.com/punypng/api/optimize

Параметры:

- `img` — URL изображения для оптимизации;
- `key` — ваш ключ для доступа;
- `group_id` — необязательный параметр, идентификатор группы (может быть выбран произвольно) для пакетной обработки файлов.

Пример успешного запроса:

```
http://www.gracepointafterfive.com/punypng/api/optimize?img= ↵
http://www.conradchu.com/images/portfolio/targets.gif ↵
&key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Запрос вернет следующий ответ:

```
{
"original_url": "http://www.conradchu.com/images/portfolio/ ↵
targets.gif",
"original_size": 35071,
"optimized_size": 27922,
"optimized_url":"http://www.gracepointafterfive.com/punypng_staging/ ↵
attachments/514367d961c0c9c5ef80e02cc3381660b24d77d8/targets.gif.png",
"group_id": null,
"savings_percent": 21,
"savings_bytes": 7149
}
```

Оптимизированное изображение, как видно, доступно по адресу, указанному в поле «optimized_url». Если вы использовали пакетный режим (то есть указали параметр «group_id» в запросе), то ZIP-файл с оптимизированными изображениями можно скачать по URL

```
http://www.gracepointafterfive.com/punypng/api/download_zip? ↵
group_id=группа
```

Ключ для скачивания архива не требуется.

Пример запроса, порождающего ошибку:

```
http://www.gracepointafterfive.com/punypng/api/optimize?img= ↵
http://libpng.org/pub/png/img_png/ pnglogo-grr.png&key= ↵
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Запрос вернет следующий ответ:

```
{
"error": "Sorry, the max file size supported is 150KB."
}
```

В настоящее время максимальный размер файлов ограничен в 150 Кб, количество файлов для пакетной обработки не может быть больше пяти, а количество изображений, обрабатываемых за день, не может быть более 50. Автор планирует запуск платной версии, где ограничения можно будет снять.

Глава 4. Уменьшение количества запросов

В этой главе делается упор на методы автоматического объединения файлов, которые позволяют значительно уменьшить издержки на пересылку запросов между браузером и сервером, возникающие в силу различных сетевых задержек. Также здесь рассматриваются различные подходы для клиентского и серверного кэширования.

4.1. Автоматическое объединение текстовых файлов

На тему автоматической «склейки» стилей и скриптов написано уже довольно много статей, но нигде не было описано полное решение, которое учитывало бы «подводные камни», связанные с браузерами и различ-

ными способами использования указанных файлов. Ниже стоит рассмотреть то практическое решение, которое реализовано в Web Optimizer и обкатано уже на нескольких тысячах сайтов.

4.1.1. Объединение CSS-файлов

Несмотря на более простой и поистине академический синтаксис, CSS-файлы довольно сложно объединять в силу разных причин. Тут и различные атрибуты media (указывающие на устройства, для которых предназначен данный файл), и возможность сделать «вложенную» загрузку стилей при помощи `@import` и т. д. Для начала рассмотрим процесс получения ссылок и содержимого самих файлов из исходной структуры веб-страницы.

Получаем код

Если в CMS у нас предусмотрена возможность вставки CSS-файла как отдельного объекта в секцию `head` страницы, то это ограждает от множества проблем по «вычленению» этих объектов из готового HTML-кода. В противном случае нам придется использовать примерно следующий вариант:

```
/* регулярное выражение для нахождения всех
<link rel= "stylesheet"> и <style type="text/css">
внутри head-секции */
$regex =
"!(<link[^>]+rel\\s*=\\s*(\"stylesheet\"|'stylesheet'|stylesheet)
([^>]*)>|<style\\s+type\\s*=\\s*(\"text/css\"|'text/css'|text/css
)([^>]*)>(.*?)</style>)!is";
preg_match_all($regex, $this->head, $matches, PREG_SET_ORDER);
if (!empty($matches)) {
    foreach($matches as $match) {
      $file = array();
      $file['tag'] = 'link';
      $file['source'] = $match[0];
/* вырезаем из найденного куска HTML-кода обрамляющие теги,
чтобы идентифицировать внутренние стилевые правила */
      $file['content'] =
preg_replace("/(<link[^>]+>|<style[^>]*>[\t\s\r\n]*|[\t\s\r\n]*<\
/style>)/i", "", $match[0]);
/* определяем все дополнительные атрибуты */
preg_match_all("@(type|rel|media|href)\s*=\s*(?:\"([^\"]+)\"|'([^
']+)'|([\s]+))@i", $match[0], $variants, PREG_SET_ORDER);
      if(is_array($variants)) {
```

```
      foreach($variants AS $variant_type) {
        $variant_type[1] = strtolower($variant_type[1]);
        $variant_type[2] = !isset($variant_type[2]) ?
          (!isset($variant_type[3]) ?
            $variant_type[4] :
              $variant_type[3]) :
          $variant_type[2];
        switch ($variant_type[1]) {
/* выставляем источник для файла стилей */
          case "href":
            $file['file'] = trim($this->strip_querystring($vari-
            ant_type[2]));
            $file['file_raw'] = $variant_type[2];
            break;
          default:
/* пропускаем media="all|screen" для предотвращения некорректного
поведения Safari при @media all{} или @media screen{} */
          if ($variant_type[1] != 'media' || ($variant_type[1]
          == 'media' && !preg_match("/all|screen/i",
          $variant_type[2]))) {
            $file[$variant_type[1]] = $variant_type[2];
          }
          break;
        }
      }
    }
    $this->initial_files[] = $file;
  }
}
```

Подавая на вход данного алгоритма код секции `head` нашего документа (`$this->head`), на выходе мы получаем готовый массив `$this->initial_files`. Стоит сразу отметить, что в массиве для файлов стилей атрибут media не выставляется, если он равен all (в этом случае он просто бесполезен) либо screen (по умолчанию у нас все стилевые правила применяются для отображения сайтов на мониторах, поэтому данное значение также можно безболезненно опустить).

Разбираем вложенность

Получить ссылки на используемые файлы мало, нам необходимо полное содержимое этих файлов. Следует иметь в виду, что нам нужно рас-

познать все внутренние конструкции `@import` (подключающие дополнительные файлы стилей) в порядке их появления в исходных файлах. Проще всего с данной проблемой может разобраться рекурсивная функция `resolve_css_imports`:

```
function resolve_css_imports($src) {
    $content = file_get_contents($src);
/* удаляем из первоначального содержимого @import внутри
   комментариев */
    $content = preg_replace("!/\*\s*@import.*?\*/!is", "",
    $content);
/* выбираем все @import */
    preg_match_all('/@import\s*(url)?\s*\(?([^;]+?)\)?;/i',
    $content, $imports, PREG_SET_ORDER);
    if (is_array($imports)) {
      foreach ($imports as $import) {
        $src = false;
/* очищаем найденный путь к файлу от пробелов и кавычек */
        if (isset($import[2])) {
          $src = $import[2];
          $src = trim($src, '\'" ');
        }
        if ($src) {
/* запускаем рекурсию для обнаруженного файла, чтобы разрешить
все @import уже внутри него */
          $content = str_replace($import[0],
          $this->resolve_css_imports($src), $content);
/* изменяем все пути для CSS-изображений и ресурсов (относительно
заданного файла) на абсолютные (относительно корня документа) */
          $content = $this->resolve_relative_paths($src, $content);
        }
      }
    }
    return $content;
}
```

Задав полный путь к файлу стилей для функции `resolve_css_imports`, мы полностью разрешим все внутренние включения, чем сведем число HTTP-запросов к минимуму.

Объединяем

После того как мы разобрались с массивом файлов и научились получать полное их содержимое, нам нужно корректно их объединить. Как уже описывалось в книге «Разгони свой сайт» (http://speedupyourwebsite.ru/books/speed-up-your-website/), для этого лучше всего применять конструкцию `@media`. Предположим, что в результирующем массиве у нас объект имеет следующий формат:

```
$this->initial_files = array(
     array(
       'content' => 'полное содержимое файла',
       'media' => 'print|handheld|etc',
       'file_raw' => 'исходный код файла в head-секции'
     ),
     ...
)
```

Тогда нам нужно просто объединить весь CSS-код в соответствие со спецификацией:

```
foreach ($this->initial_files as $file) {
     if (!empty($file['media'])) {
       $full_content .= '@media '. $file['media'] . '{';
     }
     $full_content .= $file['content'];
     if (!empty($file['media'])) {
       $full_content .= '}';
     }
}
```

На выходе мы получим весь CSS-код, обнаруженный внутри секции `head`, объединенный в одну строку, которую можно записать в один кэшированный файл. Далее нужно, используя свойство `file_raw`, удалить исходные файлы и внутренний код из документа и вставить (например, сразу же после `<head>`) вызов этого кэшированного файла.

Минимизируем

А что, если мы хотим не только объединить файлы, но и уменьшить их в размере? Gzip-компрессию здесь рассматривать не будем: она достаточно тривиальна в реализации (и может сводиться к нескольким правилам в конфигурационном файле сервера). Нам более интересен вопрос

уменьшения CSS-кода в соответствии с CSS-спецификацией. Здесь разумнее всего воспользоваться одним из трех путей.

- Набор простых регулярных выражений (он был описан еще в книге «Разгони свой сайт»). Ниже приведен его код на Perl.

```
$data = ~ s!\/\*(.*?)\*\/!!g; # удаляем комментарии
$data = ~ s!\s+! !g;          # сжимаем пробелы
$data = ~ s!\} !}\n!g;        # добавляем переводы строки
$data = ~ s!\n$!!;            # удаляем последний перевод
                                строки
$data = ~ s! \{ ! {!g;        # удаляем лишние пробелы
                                внутри скобок
$data = ~ s!; \}!}!g;         # удаляем лишние пробелы и
                                синтаксис внутри скобок
```

- CSS Tidy (http://csstidy.sourceforge.net/) — наиболее мощная библиотека для разбора CSS-правил. Для ее использования необходимо загрузить ее в папку проекта, внести изменения в настройки по умолчанию (находятся в файле `class.csstidy.php`) и осуществить минимизацию простыми вызовами:

```
$css = new csstidy();
$css->load_template($root_dir . 'css.template.tpl');
$css->parse($css_code);
echo $css->print->formatted();
```

  При этом для максимального сжатия лучше использовать следующий шаблон (`css.template.tpl`):

```
|{||{|||;|}||}||{||
```

- YUI Compressor (http://developer.yahoo.com/yui/compressor/). Эта библиотека требует установленной Java на сервере и запускается еще проще. Необходимо из командной строки выполнить:

```
java -jar yuicompressor.jar -o output.css input.css
```

  Результат произведенных действий будет сохранен в файле `output.css`.

Полный код для CSS Tidy и все аспекты практической реализации можно почерпнуть из исходного кода Web Optimizer (http://www.web-optimizer.ru/).

Сразу стоит оговориться, что в Intenet Explorer (по 8-ю версию включительно) есть проблема с отображением более 4096 (по сведениям из MSDN, http://msdn.microsoft.com/en-us/library/aa358796(VS.85).aspx) CSS-селекторов из одного файла (и ограничение в 32 на число `@import`). При разработке грамотного процесса объединения CSS-файлов этот момент стоит учитывать.

4.1.2. Объединение JavaScript-файлов

Для JavaScript-файлов весь описанный механизм повторяется, за исключением небольших деталей.

Получаем код

Во-первых, получать код мы будем уже немного другим методом, и для нас будет несущественен атрибут `media`:

```
$regex =
"!<script[^>]+type\\s*=\\s*(\"text/javascript\"|'text/javascript'
|text/javascript)([^>]*)>(.*?</script>)!is";
preg_match_all($regex, $this->head, $matches, PREG_SET_ORDER);
if (!empty($matches)) {
  foreach($matches as $match) {
    $file = array();
    $file['tag'] = 'script';
    $file['source'] = $match[0];
/* вырезаем из найденного куска HTML-кода обрамляющие теги,
чтобы идентифицировать внутренние скрипты */
    $file['content'] = preg_replace("/(<script[^>]*>
    [\t\s\r\n]*|[\t\s\r\n]*<\/script>)/i", "", $match[0]);
    $file['file'] = '';
    preg_match_all("@(type|src)\s*=\s*(?:\"([^\"]+)\"|'([^']+)'
    |([\s]+))@i", $match[0], $variants, PREG_SET_ORDER);
    if(is_array($variants)) {
      foreach($variants AS $variant_type) {
        $variant_type[1] = strtolower($variant_type[1]);
        $variant_type[2] = !isset($variant_type[2]) ?
        (!isset($variant_type[3]) ? $variant_type[4] :
        $variant_type[3]) : $variant_type[2];
```

```
        switch ($variant_type[1]) {
          case "src":
            $file['file'] =
              trim($this->strip_querystring($variant_type[2]));
            $file['file_raw'] = $variant_type[2];
            break;
          default:
            $file[$variant_type[1]] = $variant_type[2];
            break;
        }
      }
    }
    $this->initial_files[] = $file;
  }
}
```

Объединяем

Тут нас ждет еще одно отличие: разные куски JavaScript-кода лучше объединять через точку с запятой с переводом строки. Ибо предыдущая часть кода может не оканчиваться на точку с запятой, потому мы обязаны как-то отделить ее от последующей.

Далее в ходе объединения было установлено, что файлы библиотек для визуального форматирования кода (в силу своей сложности) мало приспособлены к объединению с другими файлами. Поэтому рекомендуется при объединении избегать следующих файлов: `tiny_mce.js` и `fckeditor.js`. Во всем остальном механизм абсолютно тот же самый, что и для CSS-файлов (за исключением отсутствия необходимости разрешить `@import` и необходимости заменять пути для фоновых изображений и ресурсов).

Минимизируем

Для минимизации JavaScript-кода лучше всего использовать уже имеющиеся на рынке решения: JSMin (http://www.crockford.com/javascript/jsmin.html, который портирован в том числе и на PHP) или YUI Compressor (http://developer.yahoo.com/yui/compressor/). Про последний уже было написано чуть выше (параметры для запуска те же самые). В случае с JSMin все тоже довольно просто: нам нужно загрузить последнюю версию (http://code.google.com/p/jsmin-php/), подключить ее и просто вызвать минимизацию заданного файла:

```
require 'jsmin-1.1.1.php';
echo JSMin::minify(file_get_contents('example.js'));
```

Стоит также упомянуть, что классический JSMin не поддерживает условную компиляцию для IE. Поэтому тут нужно воспользоваться модифицированным решением (например, из исходных кодов Web Optimizer).

4.1.3. Заключение

Объединение текстовых файлов способно значительно ускорить загрузку вашего сайта, не причиняя вреда качеству разработки (вы можете разрабатывать отдельно и в автоматическом режиме выкладывать на сайт уже готовые версии файлов). Как показывают данные с webo.in, на произвольном сайте в Рунете используется 2,7 файлов стилей (средний размер — 5,5 Кб) и 5 JavaScript-файлов (средний размер — 15 Кб). Простое их объединение позволит выиграть 0,5-1 с при загрузке страницы. А минимизация (вместе с gzip-сжатием, уменьшающим размер на 85%) — еще 60 Кб, что составит 0,6 с при скорости подключения 100 Кб/с.

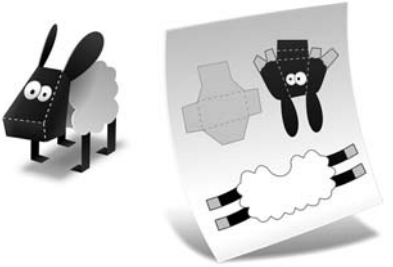
Как мы видим, совершенно простые действия способны значительно ускорить загрузку вашего сайта.

4.2. Алгоритм разбора и сбора CSS Sprites

В книге «Разгони свой сайт» был опубликован довольно подробный обзор технологии CSS Sprites. После внедрения ее на нескольких сотнях рабочих проектов удалось собрать некоторый набор наиболее часто возникающих проблем при использовании CSS Sprites и методов их решения. Также в этом разделе рассматривается прикладной способ по автоматизации создания CSS Sprites для произвольного проекта.

4.2.1. Обзор технологии

CSS Sprites, на самом деле, — всего лишь расширенное использование технологии `background`, заложенной еще в спецификации CSS1. Все, до чего додумалось прогрессивное человечество за эти годы, — это множественный фон у элементов (как он будет совместим с концепцией CSS



Sprites, еще придется проверить). Основные свойства, которые мы используем для задания фонового изображения:

- `background-image` — основная «рабочая лошадка». Именно за ней будущее в виде data:URI, который в конце концов победит CSS Sprites. Но произойдет это еще не скоро;
- `background-repeat` — вторая не менее важная составляющая при использовании фонового изображения. Ведь задавая `no-repeat` для данного свойства, мы намеренно подчеркиваем, что допустимо использование CSS Sprites для «склейки» изображений (по умолчанию используется `repeat`);
- `background-position` — «волшебная палочка» для CSS Sprites, позволяющая спрятать или показать определенные части фонового изображения.

Кроме заявленных свойств также есть еще несколько (например, `background-color`), но они к спрайтам имеют посредственное отношение. Однако стоит добавить к нашим объектам, с помощью которых мы будем формировать карты изображений, размеры (в относительных или абсолютных единицах) и отступы (`padding`). Это позволит точнее разделить изображения по группам и корректно расположить их в финальном комбинированном изображении.

4.2.2. Прикладные тонкости

Естественно, рассматривая набор возможных эффектов при использовании фоновых изображений, стоит отметить следующие:

- **Объект, полностью заполненный фоновым изображением.** Здесь основную роль играют конечные размеры объекта (разумеется, если изображение не повторяется по всем осям сразу: в таком случае использовать его для CSS Sprites не представляется возможным). Довольно часто фон под объектом может меняться в зависимости от каких-либо условий (преднамеренный акцент или действия со стороны пользователя), но для логики создания CSS Sprites это несущественно. Здесь же можно выделить три подслучая: соответствующих неповторяющемуся фону и повторяющемуся по оси X или Y.
- **Фоновое изображение заполняет не весь предоставленный ему объем** (либо размеры объекта не заданы, либо заданы в относительных — `em`, `%` — единицах). Тут нам необходимо прикреплять повторяющееся изображение «в конец» спрайта, чтобы на той части объекта, что осталась без фонового изображения, не проявлялось никаких дефектов. Либо (в случае `no-repeat`) расположить

изображения «лесенкой» (это особенно актуально в случае фона для элементов списка). Стоит отметить, что в зависимости от значения background-position CSS Sprites здесь могут быть как возможны, так и невозможны в принципе (например, в случае 100% 100%). Тут можно выделить еще несколько случаев, различающихся по background-position, background-repeat и линейными размерами блока.

- **Изображение является анимированным**. Поскольку далее речь пойдет о применении PNG- и JPEG-изображений для CSS Sprites, то анимированные изображения придется сразу выбросить из рассмотрения: поддержка анимированных PNG-изображений находится сейчас на самом зачаточном уровне в браузерах.

Все описанные примеры можно более четко структурировать по следующим группам:

1. background-repeat: no-repeat, background-position: абсолютные числа и заданы линейные абсолютные размеры.
2. background-repeat: no-repeat, background-position: абсолютные числа, линейные размеры не заданы или заданы в относительных единицах.
3. background-repeat: repeat-x, задана высота элемента.
4. background-repeat: repeat-x, высота элемента не задана.
5. background-repeat: repeat-y, задана ширина элемента.
6. background-repeat: repeat-y, ширина элемента не задана.
7. background-repeat: no-repeat, background-position: 100% 0, задана высота элемента (в абсолютных единицах).
8. background-repeat: no-repeat, background-position: 0 100%, задана ширина элемента (в абсолютных единицах).
9. background-repeat: no-repeat, background-position: 100% 0, высота элемента не задана (или задана в относительных единицах).
10. background-repeat: no-repeat, background-position: 0 100%, ширина элемента не задана (или задана в относительных единицах).
11. background-repeat: repeat.
12. background-repeat: repeat-x или background-repeat: repeat-y, размеры элемента указаны в относительных единицах.
13. background-repeat: no-repeat, background-position: относительные единицы.
14. изображение является анимированным GIF-файлом.

Глядя на эту спецификацию, становится в общем понятно, в каком направлении двигаться для автоматизации создания CSS Sprites. Стоит только отметить, что при использовании одного и того же изображения мно-

гими CSS-селекторами нужно отследить `background-position` и устранить изначальные CSS Sprites, задействованные в стилях. Процесс получения изображений из готовых CSS Sprites в автоматическом режиме достаточно сложен и может быть применим только на локальных проектах.

4.2.3. Практическое решение: CSS Tidy

Далее речь пойдет уже об инструменте Auto Sprites (http://sprites.in/), который был положен в основу разработки Web Optimizer (http://www.web-optimizer.ru/). После описанных выше умозаключений оставались чисто технические вопросы: как все это закодировать и как отладить полученное решение.

Для начала нам нужно разобрать все дерево CSS-правил, потом отобрать из них относящиеся к фоновым изображениям и линейным размерам объектов, а уже потом произвести над ними требуемые действия. Идеально для этой задачи подходит CSS Tidy (http://csstidy.sourceforge.net/), который был замечательно испробован, протестирован и улучшен после интеграции на сотнях реальных сайтов. CSS Tidy представляет собой набор PHP-библиотек, которые могут быть включены в произвольный проект для каких-либо действий над заданным набором стилевых правил.

Ниже приводится простой алгоритм выделения из массива CSS-правил нужных нам свойств фона на языке PHP:

```
/* $this — объект класса css_sprites */
/* сначала создадим новый объект CSS Tidy, используя заданный
$css_code*/
$this->css = new csstidy();
$this->css->parse($css_code);
/* определим CSS-свойство для элементов без фона, */
/* чтобы не переопределить его в ходе преобразований */
$this->none = 'none!important';

/* далее мы переберем весь массив на предмет сначала
@media-конструкций*/
foreach ($this->css->css as $import => $token) {
/* создадим для каждого заданного @media свой массив правил */
  $this->media[$import] = array();
/* а затем и самих CSS-селекторов */
    foreach ($token as $tags => $rule) {
/* получим для каждого набора селекторов массив CSS-свойств и их
значений */
```

```
      foreach ($rule as $key => $value) {
/* выделим из всех свойств только относящиеся к фону */
if (preg_match("/background/", $key)) {
/* переопределим "выключенный" фон, чтобы не затронуть */
/* в ходе преобразований */
      if ($key == 'background' && $value == 'none') {
        $this->css->css[$import][$tags]['background'] =
        $this->none;
      }
/* теперь для каждого отдельного CSS-селектора */
      foreach (split(",", $tags) as $tag) {
/* создаем отдельный объект */
        if (!empty($this->media[$import][$tag])) {
          $this->media[$import][$tag] = array();
        }
        if ($key == 'background') {
/* получаем массив фоновых свойств из CSS-свойства background */
          $background = $this->css->optimise-
          >dissolve_short_bg($value);
          foreach ($background as $bg => $property) {
/* пропускаем свойства, заданные по умолчанию */
            if (
/* в частности, для background-position */
              !($bg == 'background-position' &&
              ($property == '0 0 !important' ||
              $property == 'top left !important' ||
              $property == '0 0' ||
              $property == 'top left')) &&
/* для background-origin */
              !($bg == 'background-origin' &&
              ($property == 'padding !important' |
              $property == 'padding')) &&
/* для background-color */
              !($bg == 'background-color' &&
              ($property == 'transparent !important' ||
              $property == 'transparent')) &&
/* для background-clip */
              !($bg == 'background-clip' &&
              ($property == 'border !important' ||
              $property == 'border')) &&
/* для background-attachement */
```

```
                !($bg == 'background-attachment' &&
                ($property == 'scroll !important' ||
                $property =='scroll')) &&
/* для background-size */
                !($bg == 'background-size' &&
($property == 'auto !important' ||
                $property == 'auto')) &&
/* и для background-repeat */
                !($bg == 'background-repeat' &&
                ($property == 'repeat !important' ||
                $property == 'repeat'))) {
/* Переопределяем background-image, если оно не задано */
                    if ($bg == 'background-image' &&
                        ($property == 'none !important' ||
                        $property == 'none')) {
                        $property = $this->none;
                     }
/* и дополняем background-position, вместо left выставляем
  left center, вместо right — right center,
  и ряд других исправлений */
                    if ($bg == 'background-position') {
                      $property =
                      $this->compute_background_position
                      ($property);
                    }
/* В конце выставляем полученные значения для массива правил,
  определяющих исходные фоновые картинки */
                     $this->media[$import][$tag][$bg] = $property;
              }
           }
/* Если у нас задано детальное CSS-свойство, то просто его на-
значаем */
         } else {
/* Дополняем background-position, вместо left выставляем left
center,
  а вместо right — right center,
  также меняем местами bottom right и некоторые другие случаи */
           if ($key == 'background-position') {
                $value =
                $this->compute_background_position($value);
           }
```

```
/* и выставляем "исправленные" свойства для массива CSS-правил,
  пропуская свойства по умолчанию */
        if ($key != 'background-position' || $value != '0 0') {
           $this->media[$import][$tag][$key] = $value;
         }
       }
     }
   }
/* завершаем цикл перебора исходных CSS-правил */
  }
 }
}
```

4.2.4. Практическое решение: сборка изображений

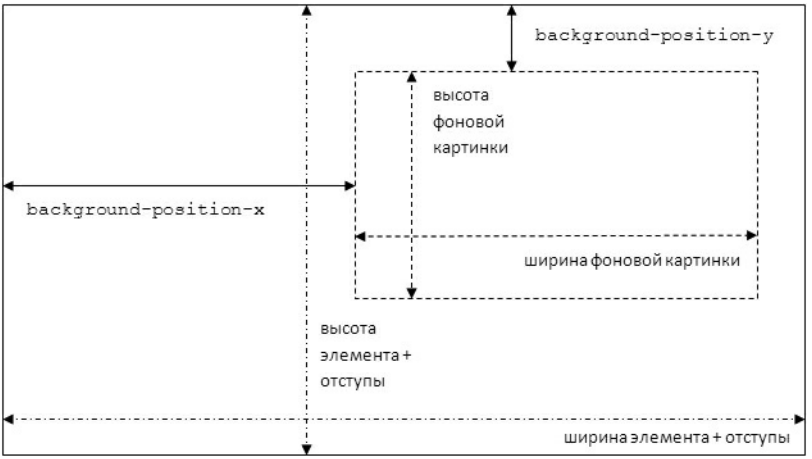
Дальше начинается самое интересное: как нам вышеописанные группы «склеивать»? Для этого используется следующий алгоритм:

1. Изображения, для которых позиционирование задано в процентах и которые находятся внутри контейнеров с абсолютными размерами, корректируются, чтобы фоновая позиция была задана в аболютных единицах.
2. `repeat-x` изображения (группа 3) объединяются все вместе по вертикали. Попутно правится ширина фоновых изображений (приводится к наименьшему общему кратному). В самое начало такого файла добавляются `no-repeat` изображения, подходящие по ширине (группа 1). Далее в самый низ файла записывается 1 изображение из группы 4. Больше 1 все равно никуда не войдет, поскольку нам нужно обеспечить свободное пространство ниже заявленной высоты фонового изображения. Это часто бывает нужно для плавного перетекания градиента в ровный цвет: градиент вставляется фоновой картинкой, которая заканчивается на фоновом цвете, а дальше продолжается уже этот цвет, заданный через `background-color`.

3. Производятся абсолютно аналогичные действия с `repeat-y`.
4. Далее изображения из группы 7 объединяются по вертикали (`0 100%` означает, что фон должен быть прижат к правому краю элемента, соответственно, весь спрайт будет «прижат» к правому своему краю).
5. Аналогично с группой 8 — прижимаем все к низу. Естественно, что для всех групп мы учитываем первоначальное значение `background-position`.
6. Рассчитываем позиционирование для изображений группы 1 (для этого подойдет и просто перебор отсортированных по площади или сумме квадратов измерений изображений: готовим матрицу, в которую пытаемся «вписать» очередное изображение; если не получается, то матрицу увеличиваем). При вписывании ради экономии процессорного времени можно проверять только 9 точек (8 по краям изображения и 1 в центре), чтобы убедиться, что на месте изображения еще ничего нет.
7. Строим «лесенку» из изображений второй группы. Лесенку лучше строить с низа уже созданного спрайта из предыдущего пункта: тогда легко найти минимальный размер «дырки» между двумя группами изображений, чтобы сдвинуть «лесенку» вверх (и потом, возможно, влево). Конечно, поиск наиболее оптимального расположения — непростая задача. Но ее можно решить и в достаточно грубом приближении, которое описано выше.
8. Итоговое изображение из пункта 4 прикрепляется к правому верхнему углу нашего сложного изображения (результат действия пунктов 6 и 7). Так как у каждого такого изображения задана допустимая высота и все они находятся вне «рабочей» зоны остальных `no-repeat` изображений, никаких рудиментов не возникнет.
9. Аналогичным образом поступаем с изображением из пункта 5 — его располагаем в нижнем левом углу нашего спрайта.
10. На выходе мы получаем 3 спрайта со всеми возможными случаями. В среднем размеры таких спрайтов будут лишь незначительно превосходить (если вообще будут) аналогичные «ручные» аналоги (в том числе за счет использования PNG). Естественно, можно в автоматическом режиме пропустить их через `pngcrush` или `jpegtran`. Стоит также предусмотреть, в каком виде будут создаваться полноцветные изображения: JPEG или PNG (последние больше по размеру, но гарантируют отсутствие потерь качества).

При расчете позиции картинки в конечном спрайте может помочь следующая схема:

Рис. 4.1. Схема позиционирования фоновой картинки относительно элемента



В силу громоздкости решения (в нем более 1500 строк кода) в полном объеме в данной книге оно не приводится, однако все описанные шаги уже применены в Web Optimizer (http://www.web-optimizer.ru/) (веб-приложении для автоматизации клиентской оптимизации). Одна из финальных версий алгоритма работает для инструмента Auto Sprites (http://sprites.in/), а с исходным текстом можно ознакомиться в SVN (http://code.google.com/p/web-optimizator/source/browse/trunk/libs/ php/css.sprites.php).

Эту логику можно применить на любом этапе веб-разработки (как при начальном создании дизайна, так и при пострелизной оптимизации сайта). Библиотека для автоматического создания спрайтов распространяется по лицензии MIT (она позволяет использовать продукт где угодно и как угодно, но обязательно должна присутствовать ссылка на первоисточник, даже если используется не вся библиотека, а только ее существенная часть).

4.3. CSS Sprites и data:URI, или Microsoft и весь остальной мир

На стадии полной загрузки страницы браузер запрашивает картинки и Flash-анимацию, которые заполняют отведенные им места на странице.

По мере появления элементов на странице пользователь ощущает, что страница загружается. Обычно окончание этой стадии совпадает для пользователя с окончанием всей загрузки.

Если говорить об ускорении этой стадии, то здесь одной из основных технологий будет именно технология CSS Sprites, которая уже отлично себя зарекомендовала в этом качестве. Однако у нее вместе с очевидными плюсами (значительное уменьшение запросов к серверу, кроссбраузерность) есть и несколько минусов. В частности:

- несемантическая верстка в случае использования сложных спрайтов. Это приводит к дополнительному времени создания документа (особенно существенно может быть для «старых» браузеров). (Более подробно о влиянии DOM-дерева на процесс загрузки страницы рассказывается в следующей главе);
- невозможность комбинирования нескольких осей повторения. Это не очень существенно, поскольку все использование CSS Sprites (как было описано в предыдущем разделе) можно свести к трем основным изображениям;
- тяжесть изменения картинки в случае сложной геометрии;
- отображение неверного фона при масштабировании.

4.3.1. data:URI

Есть ли выход из этого положения? Да, есть. Это технология `data:URI`, которая позволяет включать фоновые изображения прямо в CSS-файл в base64-виде. Плюсы данного подхода очевидны: не нужно «склеивать» несколько картинок в один файл, есть возможность объединять различные оси и анимированные с обычными изображениями, полностью отделить содержание (семантику документа) от его представления (оформления и дизайна), и т. д.

Но есть и ложка дегтя: IE вплоть до версии 7 не поддерживает `data:URI`. IE8 — уже да, но все остальные IE — нет. Что делать?

4.3.2. mhtml

Нам на помощь приходит технология `mhtml` (MIME HTML), которую поддерживает по умолчанию только IE (почти в полной мере) и Opera (начиная с версии 9.0). Она позволяет включать base64-данные в CSS-файл в виде комментариев. В этом случае сам файл выступает в двух ипостасях: как таблица стилей и как хранилище фоновых картинок.

Если мы объединим эту технологию с `data:URI`, то все будет хорошо. Правда?

4.3.3. Проблема 1: долгая предзагрузка

В случае включения фоновых картинок прямо в CSS-файл последний заметно увеличивается в размере (даже при использовании gzip-сжатия). Это значительно увеличивает время предзагрузки (если фоновых картинок больше 10—15 Кб), и пользователь дольше видит белый экран. Опять все плохо. Как быть?

Возможным выходом из сложившейся ситуации может стать подключение CSS-файла с фоновыми картинками по комбинированному событию `window.onload`, что вынесет загрузку элементов дизайна в ту область, где она изначально находилась: на стадию полной загрузки страницы или даже в пост-загрузку. В данном случае мы получаем полную аналогию метода CSS Sprites, только без заявленных выше неудобств.

4.3.4. Проблема 2: выключенный JavaScript

Описанный выше прием позволит облегчить загрузку только пользователей с включенным (или поддерживаемым) JavaScript (их порядка 98-99%). Однако в ряде проектов это может быть недостаточно. Для оставшихся пользователей мы можем через `<noscript>` подключить нужный нам файл (и конкретно для них замедлить предзагрузку) или поместить вызов этого файла перед `</body>` (что в ряде случаев может быть аналогично подключению стилей в `<head>`).

В качестве еще одного варианта можно рассмотреть создание единственного CSS-файла для таких пользователей, чтобы максимально ускорить им загрузку в случае отключенного JavaScript.

4.3.5. Проблема 3: Safari и window.onload

Из-за того, что Safari отличается алгоритмом обновления страницы (что позволяет значительно ускорить отображение самих страниц), для этого браузера не удается загрузить дополнительные стили после отображения первоначальной картинки (на стадии полной загрузки страницы). В этом случае Safari блокирует отрисовку картинки на экран и ожидает загрузку нового файла стилей.

На данный момент для Safari мы можем безболезненно загружать дополнительные файлы стилей только по полному событию `window.onload`.

4.3.6. Проблема 4: Microsoft, IE7 и Windows Vista

В результате проведенных исследований удалось установить, что в связи с проблемами безопасности в Vista `mhtml-технология` для отобра-

жения фоновых изображений не поддерживается. В этом случае единственным выходом будет подключение конкретно для этого браузера (через JavaScript) общего файла (с использованием CSS Sprites). Для пользователей IE7@Vista с отключенным JavaScript у нас нет другой альтернативы, чем загружать файл только с CSS Sprites.

Более подробно о создании автоматического решения, позволяющего решить все описанные проблемы, рассказывается в следующем разделе.

4.4. Автоматизация кроссбраузерного решения для data:URI

Рис. 4.2. Логотип duris.ru: Data:URI [CSS] Sprites. Источник: duris.ru

Многим профессиональным веб-разработчикам известны приемы оптимизации сайтов. Одним из способов оптимизации является применение CSS Sprites. Этим же разработчикам известно, какие существуют трудности с формированием, сборкой и пересборкой стандартных спрайтов. При использовании стандартных спрайтов полностью автоматизировать сборку для всех случаев проблематично, из-за специфики свойств `background` в CSS. Иногда камнем преткновения является свойство `repeat` фоновой картинки.

4.4.1. Data:URI CSS

Существует альтернативный вариант генерации CSS-спрайтов — на основе схемы `data:URI`. Данный подход интересен тем, что максимально минимизируется количество обращений к серверу, и самое важное — можно полностью автоматизировать процесс сборки и перегенерации спрайтов. Для полной автоматизации процесса оптимизации работы сайтов и сборки CSS-спрайтов на основе `data:URI` схемы была разработана специальная библиотека.

Мучения со стандартным подходом применения CSS-спрайтов, а именно, трудности модернизации и, в некоторых случаях, сложности оптимальной компоновки заставили искать альтернативный вариант опти-

мизации загрузки изображений. Далее речь пойдет об использовании комбинированного метода: `data:URI + mhtml`. Более детально процесс создания изображений в этом формате описан в пятой главе книги «Разгони свой сайт».

В ходе дискуссий и умозаключений были определены слабые и сильные стороны этого подхода. Одним из значительных его недостатков является сложность сборки конечного CSS. Однако при использовании `data:URI` существует возможность автоматизации процесса. Соответственно, было принято решение создать программное обеспечение для автоматической сборки `data:URI` спрайтов.

Новый подход генерации CSS-спрайтов на основе `data:URI` решили назвать Data URI Sprites — DURIS.ru. Название немного необычное — но зато уникальное и хорошо запоминается.

4.4.2. Что это?

В первую очередь это полностью автоматический анализ и сборка CSS-спрайтов на основе `data:URI`.

Некоторые характеристики работы DURIS:

- загрузка и анализ всех внутренних (`<style>`) и внешних (`<link>`) стилей;
- выделение `background-image` в отдельный внешний стиль;
- загрузка и кодирование в base64 всех изображений, которые найдены в стилях;
- оптимизация правил с повторяющимися `background-image` в стилях;
- удаление CSS-правил с отсутствующими на сервере изображениями (устраняет лишние ненужные запросы);
- специальное подключение `data:URI` спрайтов для всех браузеров и отдельно для IE6, IE7 Vista (более детально в FAQ, duris.ru/faq/).

4.4.3. Зачем это надо?

Современный подход создания CSS-спрайтов:

- позволяет безболезненно вносить коррективы в изображения и верстку;
- дает возможность свести к минимуму число запросов к серверу для получения информации, которая относится к оформлению страницы;
- позволяет использовать полностью семантическую верстку;
- устраняет проблемы масштабирования для фоновых изображений;

- объединяет изображения разных типов и осей повторения;
- все создается в автоматическом режиме.

4.4.4. IE и другие

В ходе разработки реализации `data:URI` спрайтов были проанализированы наиболее часто встречающие варианты CSS-правил. Также был учтен всеми любимый браузер IE. Кому еще не известно: IE не поддерживает до версии 8 технологию `data:URI`. Однако для него существует альтернативный вариант реализации спрайтов — на основе `mhtml-технологии`. Другими словами, на данный момент мы имеем полный спектр решений для всех современных браузеров (99% процентов всех используемых браузеров). Но, как всегда, «ложку дегтя» подкидывает Microsoft. Во время тестирования найдена ошибка `mhtml-технологии` в Vista IE7 — а именно, браузер IE7 в ОС Vista при кэшировании `mhtml-файла` отказывается показывать изображения (это связано с малодокументированными проблемами безопасности при использовании `mhtml` в Vista IE7). Если сделать файл некэшируемым, то все работает, как и должно работать, но в случае с кэшированием фоновые изображения не отображаются.

В общем, Microsoft все же сделал так, чтобы «цепь разорвалась». На текущий момент для браузера IE7 в ОС Vista реализация DURIS работает не совсем так, как задумывалось изначально. В подключение стилей внедрен алгоритм проверки IE7 Vista и в случае обнаружения такового фоновые картинки подключаются стандартным путем. Статистика показывает, что пользователей, которые используют IE7 под Vista, около 5% процентов. В любом случае, мы уже знаем, что в IE8 нормально реализована `data:URI-технология`.

4.4.5. Основные достоинства

Выделим два наиболее важных достоинства использования современного подхода генерации CSS-спрайтов.

- Все фоновые изображения, вынесенные в дополнительный файл стилей, загружаются за одно подключение к серверу — это обстоятельство позволяет уменьшить нагрузку на сервер и ускорить отображение фоновых изображений.
- Сборка конечного файла CSS-спрайтов производится автоматически — это позволяет безболезненно проводить модернизацию изображений.

4.4.6. Что имеем

На сегодняшний день мы имеем стабильную бета-версию DURIS, которая отрабатывает все передаваемые ей CSS-файлы. Отрабатываются также специфические правила, такие как `filter:AlphaImageLoader`, `!important`. Ядро DURIS разработано на языке Java и является самодостаточным (т. е. не зависит от сайта). Предполагается, что после получения релиз-кандидата исходный код ядра будет выложен в открытый доступ под OpenSource-лицензией. Ядро работает с командной строки наподобие того, как работает YUI Compressor. Это позволит удобно интегрировать автоматическую генерацию CSS-спрайтов в свои проекты. Кто программирует на Java, при желании сможет напрямую вызывать методы ядра DURIS.

4.4.7. Итого

Разработанный метод/алгоритм автоматической генерации CSS-спрайтов основе `data:URI` уникален в своем роде и не имеет мировых аналогов. На сайте выложен FAQ (http://duris.ru/faq/), в котором детально описано, что и как работает. Если что не понятно — задаем вопросы в комментариях.

В роли главного разработчика данного решения выступил Руслан Синицкий (aka sirus, http://fullajax/#:developers).

4.4.8. Будущее

При появлении поддержки в IE8 схемы `data:URI` (стоит все же помнить об ограничении в 32 Кб, http://msdn.microsoft.com/en-us/library/cc848897%28VS.85%29.aspx) разработанный подход становится довольно перспективным, теперь все современные браузеры поддерживают такие спрайты.

Детально ознакомиться с принципами генерации и подключения `data:URI` CSS-спрайтов можно в разделе FAQ (http://duris.ru/faq/). Если у вас возникли вопросы или предложения по работе сайта или алгоритма в целом, вы можете оставить свои пожелания на этом сайте в форме обратной связи.

Стоит также понимать, что наиболее эффективные методы клиентской оптимизации будут использовать комбинированные решения (как, например, это реализовано в Web Optimizer). Часть фоновых изображений, которые сложно или нерационально «склеивать» в CSS Sprites и которые невелики по размеру, мы можем включить прямо в CSS-файл. Размер его при этом увеличится незначительно.

Большие же изображения (больше 24 Кб) мы не можем включать из принципов кроссбраузерности, они могут формировать CSS Sprites и этим не только уменьшать общее время загрузки, но и формировать наиболее оптимальную визуальную скорость загрузки сайта у конечного пользователя. При этом часть картинок (включенных через комбинированный `data:URI`) появится сразу же, а часть (включенных через CSS Sprites) — через некоторое время, так как они будут представлены отдельными файлами.

4.5. Автоматизация кэширования

Кэширование на клиентском и серверном уровнях может значительно ускорить скорость работы сайта. На данный момент все современные системы управления сайтом включают поддержку кэширования, в некоторых случаях даже многоуровневого.

4.5.1. Кэширование на клиентском уровне

Настройка кэширования для предотвращения дополнительных запросов из браузера к серверу осуществляется достаточно просто: нужно всего лишь знать наиболее характерные случаи использования.

Статические ресурсы без сжатия

Форсирование кэширования для статических ресурсов может выполняться без сжатия. В данном случае мы ничем не рискуем, выставляя не

только максимальное время кэширования, но и предлагая кэшировать ресурсы на локальных прокси-серверах (директива `Cache-Control: public`). Для PHP у нас получится следующий код (в `Expires` прописана дата на 10 лет вперед относительно текущего времени на сервере):

```
<?php
  header("Cache-Control: public, max-age=315360000");
  header("Expires: Mon, 01 Jul 2019 00:00:00");
?>
```

В случае выставления директив для Apache:

```
<FilesMatch \.(bmp|png|gif|jpe?g|ico|swf|flv|pdf|tiff)$>
  Header append Cache-Control public
  ExpiresActive On
  ExpiresDefault "access plus 10 years"
</FilesMatch>
```

И в случае nginx:

```
location ~* ^.+\.(bmp|gif|jpg|jpeg|png|swf|tiff|swf|flv)$ {
  expires    10y;
  header set Cache-Control public;
}
```

10-летний срок кэширования здесь вполне оправдан: таким образом мы сообщаем пользователям, что ресурсы в течение этого времени перезапрашивать не нужно. Если ресурсы будут изменены, то нам все равно придется форсировать сброс кэша (об этом чуть ниже), чтобы гарантировать корректное отображение материалов сайта во всех браузерах.

Статические ресурсы со сжатием

Все отличие от предыдущего случая заключается в том, что разрешить локальным прокси-серверам кэшировать такие ресурсы (обычно это CSS-, JavaScript- или ICO-файлы) нельзя. Даже больше: нам нужно запретить кэширование сжатых версий файлов, чтобы избежать выдачи сжатого файла тем пользователям, которые сжатия не поддерживают.

Код для PHP:

```
<?php
  header("Cache-Control: private, max-age=315360000");
```

```
  header("Expires: Mon, 01 Jul 2019 00:00:00");
?>
```

Для Apache:

```
<FilesMatch \.(css|js|ico)$>
  Header append Cache-Control private
  ExpiresActive On
  ExpiresDefault "access plus 10 years"
</FilesMatch>
```

И для nginx:

```
location ~* ^.+\.(css|js|ico)$ {
  expires   10y;
  header set Cache-Control private;
}
```

Очевидно, что в некоторых случаях директивы можно объединить с предыдущим случаем.

Если нам нужно добавить кэширование на определенный срок для HTML-документов, то достаточно прописать в директиве `FilesMatch` расширения файлов, указанные чуть ниже.

Запрет кэширования динамических ресурсов

Обычно для произвольного сайта (информация на котором часто меняется) кэширование HTML-документов запрещено. Это связано с быстрым устареванием предоставляемой информации (здесь стоит отметить, что правила для отображения и взаимодействия информации — стили и скрипты — устаревают намного медленнее, чем сама информация).

Для того чтобы запретить кэширование во всех браузерах HTML-документов, нужно написать с помощью PHP (в `Expires` прописано текущее время на сервере):

```
header("Expires: Wed, 01 July 2009 00:00:00");
header("Cache-Control: no-store, no-cache, must-revalidate,
private");
header("Pragma: no-cache");
```

Для Apache:

```
<FilesMatch \.(php|phtml|shtml|html|xml|htm)$>
  ExpiresActive Off
```

```
  Header append Cache-Control "no-store, no-cache,
  must-revalidate, private"
  Header append Pragma "no-cache"
</FilesMatch>
```

Для nginx:

```
location ~* ^.+\.(php|phtml|shtml|html|xml|htm)$ {
  expires   0;
  header set Cache-Control "no-store, no-cache, must-revalidate,
private";
  header set Pragma “no-cache";
}
```

4.5.2. Условное кэширование

Для динамических ресурсов очень часто оказывается, что обеспечить какой-либо срок кэширования невозможно: содержимое документа постоянно изменяется или полностью зависит от пользовательских действий. Если рассмотреть статические файлы, то и для них бывает весьма полезно настроить проверку изменения ресурса с течением времени. Например, установить срок действия кэша на месяц, а чтобы не увеличивать объем передаваемых данных, через месяц проверять, изменился ли запрашиваемый объект или нет.

В том случае, если объект не изменился, мы можем передать клиенту соответствующий заголовок и не передавать все содержимое объекта (ведь оно уже на клиенте имеется). Такой метод кэширования (когда мы в зависимости от определенных условий либо передаем в браузер полностью содержимое файла, либо отвечаем, что файл не изменился) называется условным кэшированием.

Существует два способа обеспечить условное кэширование: при помощи заголовков `Last-Modified + If-Modified-Since` и `ETag + If-None-Match`. Первый заголовок в паре относится к ответу со стороны сервера, второй — к запросу со стороны клиента, который уже получил соответствующий ответ сервера и теперь желает проверить, изменилось ли что-нибудь с той поры.

При помощи пары `Last-Modified` мы можем установить соответствие ресурса только по времени его изменения. `ETag` (*англ. Entity Tag, метка сущности*) предполагает более широкую сферу применения: мы можем для ресурса назначить уникальную строку, зависящую не только от времени изменения, но и от других параметров (например, передается данный файл в сжатом виде или нет).

Для отдельных серверов (которые не работают как кластер) более уместной будет настройка именно даты изменения файла. Для более сложных систем необходимо настраивать именно ETag-заголовки, чтобы гарантировать уникальность файла среди множества различных машин.

Давайте рассмотрим, как передавать и проверять данную пару заголовков на примере динамического содержимого HTML-документа (предполагается, что документ изменяется относительно редко, поэтому мы можем его закэшировать).

```
/* получаем содержимое документа, например, из закэшированного
файла */
$content = @file_get_contents($file);
/* вычисляем уникальную метку, зависящую от содержания */
$hash = md5($content);
/* проверяем соответствие вычисленной и переданной с клиента
меток */
if ((isset($_SERVER['HTTP_IF_NONE_MATCH']) &&
  stripslashes($_SERVER['HTTP_IF_NONE_MATCH']) == '"' . $hash .
  '"') ||
  (isset($_SERVER['HTTP_IF_MATCH']) &&
  stripslashes($_SERVER['HTTP_IF_MATCH']) == '"' . $hash . '"')) {
/* в случае совпадения меток передаем 304-ответ */
    header ("HTTP/1.0 304 Not Modified");
/* и не передаем содержимое документа */
    header ("Content-Length: 0");
    exit();
}
/* во всех остальных случаях выставляем заголовок ETag */
header("ETag: \"" . $hash . "\"");
/* и отдаем полностью содержимое документа */
echo $content;
```

4.5.3. Сброс кэша

Довольно часто от кэширования на клиентском уровне отказываются в силу того, что трудно бывает корректно контролировать поведение всех браузеров в случае частичного или полного изменения ресурсов, которые находятся в кэше.

Строка запроса

На самом деле эта проблема обходится достаточно просто, и в книге «Разгони свой сайт» уже было приведено несколько способов обес-

печения этого механизма. Наиболее простой путь заключается в том, что мы можем добавлять специальный GET-параметр (или просто строку запроса) к нашему ресурсу. При этом его URL меняется для браузера (но для сервера остается тем же самым), что фактически решает поставленную задачу.

Рассмотрим следующий вызов CSS-файла:

```
<link rel="stylesheet" href="/css/main.css?20090701"
type="text/css"/>
```

В строке запроса (части адреса, идущей после ?) мы видим метку времени (20090701). Не обязательно вводить здесь именно время, это может быть также и номер ревизии из системы хранения версий, и любая другая метка, которая бы однозначно могла указывать на изменение файла: разные метки должны соответствовать разным файлам.

Если мы используем метку времени и у нас есть, предположим, имя файла, который мы хотим включить в наш документ как таблицу стилей, то следующий PHP-код обеспечивает уникальность кэширования для нашего случая:

```
<?php
/* получаем метку времени, равную времени изменения файла */
$timestamp = filemtime($file);
/* выводим ссылку на файл в HTML-документе */
echo '<link rel="stylesheet" href="/css/main.css?". $timestamp .
  "' type="text/css"/>';
?>
```

Данное решение будет неэффективным для высоконагруженных систем, ибо для каждого просмотра HTML-документа мы будем дополнительно запрашивать файловую систему на предмет изменения сопутствующих файлов.

Физическое имя файла

Указанное выше решение обладает еще одним небольшим недостатком: некоторые прокси-серверы не будут кэшировать файлы со строкой

запроса, считая их динамическими. Мы можем обойти данную ситуацию через Rewrite-правило в конфигурации Apache:

```
RewriteRule ^(.*)(\.v[0-9]+)?\.(css|js)$ $1.$2 [QSA,L]
```

Какой оно несет смысл? А тот, что при указании в HTML-документе ссылки на файл

```
main.layout.v123456.css
```

сервер отдаст физический файл

```
main.layout.css
```

Таким образом мы элегантно обходим проблему прокси-серверов одной строкой в конфигурации сервера. Соответствующий PHP-код будет выглядеть так:

```
<?php
/* получаем метку времени, равную времени изменения файла */
$timestamp = filemtime($file);
/* выводим ссылку на файл в HTML-документе */
echo '<link rel="stylesheet" href="/css/main.css.v". $timestamp .
  "' type="text/css"/>';
?>
```

Однако, как и в предыдущем случае, мы все равно обращаемся к файловой системе.

Создание хэша

Как уже было указано выше, для автоматизации процесса сброса кэша на клиенте нам нужно каждый раз запрашивать файловую систему на предмет изменения файла. Если файлов у нас несколько, то дополнительная нагрузка многократно возрастает. Но принимая во внимание то, что мы собираемся все файлы объединить в один, мы можем контролировать изменение итогового файла, не проверяя при этом всех входящих в него.

Давайте рассмотрим пример кода, приведенного в начале главы. Мы можем собирать все известные нам вхождения файлов в документе в одну строку, а затем вычислять от полученной строки хэш:

```
$hash = '';
foreach ($this->initial_files as $file) {
  $hash .= $file['file_raw'];
}
$new_file_name = md5($hash);
```

Таким образом мы будем обновлять клиентский кэш каждый раз, когда у нас изменится хотя бы один вызов тех файлов, которые формируют хэш. Это не решает проблемы, затронутой в начале главы (нам все равно нужно проверять сам файл, в котором объединено несколько, на дату изменения), но помогает решить схожую проблему, которая возникает по ходу обобщения решения на большой проект.

Использование разделенной памяти

Собственно решение проблемы проверки физических файлов на изменение лежит на поверхности. Для этого нам нужно обеспечить:

- подключение библиотек разделяемой памяти (APC, eAccelerator, memcache);
- возможность управлять состоянием кэша (редактирование проверяемых файлов через веб-интерфейс, частичный сброс кэша либо полный сброс закэшированных файлов).

На примере APC описанный алгоритм выглядит следующим образом:

```
<?php
/* удаляем (ставим время истечения равным 1 с) из APC запись
относительно */
/* текущего файла, при изменении каких-либо включенных в него
файлов */
if ($changed) {
  apc_store($new_file_name, apc_fetch($new_file_name), 1);
}
…
/* при выдаче закэшированного файла проверяем, нужно ли
его пересобирать */
$mtime = apc_fetch($new_file_name);
if (!$mtime) {
  …
/* если нужно, то при создании файла записываем текущее время
в APC */
  $mtime = $_SERVER['REQUEST_TIME'];
  echo '<link rel="stylesheet" href="/css/main.css?". $mtime .
    "' type="text/css"/>';
  apc_store($new_file_name, $mtime);
} else {
/* если нет, то у нас уже получено время изменения файла,
которое можно использовать для метки кэша */
```

```
echo '<link rel="stylesheet" href="/css/main.css?". $mtime .
  "' type="text/css"/>';
}
?>
```

Как можно видеть, предложенный алгоритм весьма прост (и может сводиться к простой процедуре очистки кэша, когда мы для всех созданных комбинированных файлов форсируем пересборку), но позволяет полностью избежать обращения к файловой системе (или ее кэшу) при действиях по клиентской оптимизации страницы.

Таким образом, кэширование на клиентском уровне может ускорить загрузку последующих страниц (или посещений) вашего сайта на 500-1000% (в 5-10 раз), а правильное управление кэшированием гарантирует вам, что информация, получаемая пользователями, всегда будет актуальной.

4.5.4. Кэширование на серверном уровне

Довольно часто приходится наблюдать ситуацию, когда время создания страницы на сервере занимает несколько (десятков) секунд. Разбор характерных причин возникновения такого явления и возможные решения мы рассматривать не будем (он выходит за рамки текущей книги), но есть ряд методов для частичного устранения проблем «медленных» страниц. Речь идет о простейшем кэшировании созданных HTML-документов.

Обычно (в случае динамических сайтов) при создании HTML-страницы осуществляются десятки или сотни запросов к базе данных, подключаются десятки (или сотни) различных библиотек и не всегда можно корректно справиться с проблемами количества запросов на уровне самой

базы данных или ускорителя компиляции исполняемого кода (например, eaccelerator).

На виртуальных хостингах (когда ресурсы одного сервера могут делить десятки или сотни различных сайтов) для относительно простого сайта (который не предполагает значительного взаимодействия с пользователем) стоит рассмотреть возможность создания кэша готовых HTML-страниц. Что это такое?

Отдаем закэшированный документ

Каждый конкретный пользователь при запросе на сервер получает готовый HTML-документ, включающий в себя только клиентскую (не зависящую от состояния сервера) динамику, которая функционирует в браузере пользователя (и хранится в его кэше), а не на сервере. Соответственно, мы можем фактически смоделировать тот же браузерный кэш, только серверными средствами.

Для более детального примера давайте рассмотрим следующий код, отвечающий за простейшее серверное кэширование, из Web Optimizer:

```
/* проверяем, можем ли отдать закэшированный документ */
if (!empty($this->cache_me)) {
/* переводим адрес документа в реальное имя файла */
  $this->uri = $this->convert_request_uri();
  $file = $this->options['page']['cachedir'] . '/' . $this->uri;
/* проверяем, существует ли файл и достаточно ли он актуальный */
  if (is_file($file) &&
    $_SERVER['REQUEST_TIME'] - filemtime($file) <
      $this->options['page']['cache_timeout']) {
        $content = @file_get_contents($file);
/* проверяем, сжат ли файл */
      if (!empty($this->options['page']['gzip']) &&
          substr($content, 0, 8) ==
"\x1f\x8b\x08\x00\x00\x00\x00\x00") {
/* если сжат, то выставляем соответствующие заголовки */
            $this->set_gzip_header();
      }
/* отдает закэшированное содержимое */
      echo $content;
/* и закрываем PHP-процес */
      die();
  }
}
```

Переменная `$cache_me` может формироваться на основе множества параметров (в том числе части URL, которые нужно или не нужно кэшировать, пользовательские агенты и роботы, для которых можно отдавать кэшированные версии страниц, и т. д.). Стоит также отметить, что просто создать файл с именем, равным текущему URL страницы, невозможно: в нем встречаются недопустимые символы (/, ?), которые нужно трансформировать при сохранении на файловой системе.

Создаем закэшированный документ

Но мы рассмотрели процесс выдачи закэшированного документа, а каким образом он появляется на жестком диске? Процедура сохранения файла немного проще и может быть записана следующим образом (код из Web Optimizer):

```
/* определяем, нужно ли нам сохранять закэшированную версию
документа */
if (!empty($this->cache_me)) {
/* формируем имя файла */
  $file = $options['cachedir'] . '/' . $this->uri;
/* проверяем, есть ли такой файл и не устарел ли он */
  if (!is_file($file) ||
    $_SERVER['REQUEST_TIME'] - filemtime($file) >
      $options['cache_timeout']) {
/* записываем новое содержимое в файл */
        $fp = @fopen($file, "a");
        if ($fp) {
/* блокируем файл от конкурентных попыток записи */
          @flock($fp, LOCK_EX);
/* удаляем содержимое и перемещаемся на начало файла */
          @ftruncate($fp, 0);
          @fseek($fp, 0);
          @fwrite($fp, $this->content);
          @fclose($fp);
        }
  }
}
```

Правильно настроенное кэширование на серверном уровне способно сэкономить время ваших посетителей (и тем самым поднять конверсию сайта) и сэкономить серверные ресурсы (при использовании каких-либо распределенных мощностей).

4.5.5. Кэширование XHR-запросов

Данный раздел написан после прочтения заметки Steve Souders «F5 and XHR deep dive», посвященной вопросам кэширования XHR-ресурсов.

Оказывается, что любые данные, полученные при помощи AJAX, никогда не будут обновлены в IE прежде истечения срока действия кэша, даже если вы форсируете обновление (`Ctrl+F5`). Единственный путь обновить эти данные — это вручную удалить их из кэша.

Если вы нажимаете Перезагрузку (F5), IE перезапросит все ресурсы (даже с неистекшим сроком действия кэша), за исключением XHR. Это может вызвать большое недоумение среди разработчиков при тестировании, но меня заинтересовало, какие еще проблемы существуют в этом направлении. Будет ли поведение аналогичным во всех остальных основных браузерах? Что произойдет, если срок давности кэша будет в прошлом или заголовок `Expires` вообще не будет выставлен? Будет ли какой-либо эффект от добавления `Cache-Control max-age` (который переписывает заголовок `Expires`)?

Проводим тестирование

Для ответа на все заявленные вопросы была создана тестовая страница. На ней располагалась картинка, внешний скрипт и XMLHttpRequest. Ниже приведены протестированные модификации этой страницы.

- Вариант с `Expires` в прошлом добавляет в заголовки ответа `Expires`, который содержит дату на 30 дней ранее текущей, и `Cache-Control` с `max-age=0`.
- Вариант без `Expires` вообще не выставляет никаких заголовков `Expires` или `Cache-Control`.
- Вариант с `Expires` в будущем добавляет в заголовки ответа `Expires`, который содержит дату на 30 позже текущей, и `Cache-Control` с `max-age=2592000`.

Ниже в таблице приведены результаты тестирования этой страницы в основных браузерах. Также там записано, перезапрашивался ли XHR-ресурс или был прочитан из кэша, а если перезапрашивался, то с каким кодом HTTP-статуса.

Ниже приведены соображения на тему того, что происходит при нажатии F5.

- Все браузеры перезапрашивают и картинку, и внешний скрипт (это имеет смысл).
- Все браузеры перезапрашивают XHR-ресурс, если срок действия кэша находится в прошлом (это тоже имеет смысл: браузер знает, что закэшированный XHR-ресурс устарел).

Таблица 4.1. Если кэшируется XHR, что происходит при нажатии F5?

	Expires в прошлом	Без Expires	Expires в будущем
Chrome 2	304	304	304
Firefox 3.5	304	304	304
IE 7	304	кэш	кэш
IE 8	304	кэш	кэш
Opera 10	304	кэш	304
Safari 4	200	200	200

- Единственное различие в поведении замечено в тот момент, когда для XHR-ресурса нет заголовка `Expires`, или же `Expires` выставлен в будущее. IE 7&8 **не** перезапрашивают XHR-ресурс, если нет `Expires` или `Expires` выставлен в будущее, даже при нажатии `Ctrl+F5`. Opera 10 **не** перезапрашивает XHR-ресурс, если нет `Expires` (эквивалента для `Ctrl+F5` в Opera найти не удалось).
- И Opera 10, и Safari 4 перезапрашивают `favicon.ico` во всех случаях (это весьма расточительно).
- Safari 4 не посылает заголовок `If-Modified-Since` во всех случаях. В результате ответ всегда приходит со статус-кодом 200 и включает запрашиваемый ресурс полностью. Это верно как для XHR-ресурса, так и для картинок и внешних скриптов (это выглядит неоптимально и отличается от поведения других браузеров).

Выводы

Ниже резюмированы рекомендации для веб-разработчиков при работе с XHR-ресурсами.

1. Разработчики должны выставлять срок действия кэша для XHR-ресурсов или в прошлом, или в будущем, чтобы предотвратить расхождения в поведении браузеров, когда `Expires` вообще не выставлен.
2. Если XHR-ресурсы вообще **не должны** быть закэшированы, разработчикам стоит выставлять дату изменения ресурса в прошлое. Это давняя проблема с различным поведением браузеров при наличии закэшированных копий определенных ресурсов, и касается

она не только XHR-запросов. Например, не всегда пользователи будут перезагружать страницу — они могут на нее попасть, просто переходя по ссылкам. В этом случае браузер выдаст им закэшированные версии XHR. Для форсирования сброса кэша мы можем выставлять, например, дополнительный GET-параметр, и это будет работать для всех браузеров и всех прокси-серверов. Более подробно вопросы сброса кэша описаны ранее в этом разделе.

3. Если XHR-запросы **желательно** кэшировать, то разработчики должны назначить срок истечения кэша в будущем. При тестировании в IE 7&8 разработчикам придется не забывать очищать кэш, чтобы проверить действие Перезагрузки (F5).

Глава 5. Оптимизация структуры веб-страниц

В этой главе речь пойдет о методах, направленных на более оптимальное (с точки зрения скорости загрузки) расположение и использование структурных элементов страницы: стилей, скриптов и других статических элементов.

5.1. Динамические стили: быстро и просто

В книге «Разгони свой сайт» уже затрагивалась тема быстрого добавления стилевых правил в исходный документ динамически, не затрагивая при этом стадию предзагрузки (когда у нас еще белый экран в браузере). В ней однако не был рассмотрен следующий вопрос: какой метод использовать для добавления массива CSS-правил в сам HTML.

Естественно, что таких вариантов существует несколько, и дальше они все будут рассмотрены с точки зрения производительности в клиентском браузере.

5.1.1. Тестовое окружение

Поскольку скорость загрузки отдельного CSS-файла достаточно велика, а нам требуется рассмотреть, как его содержимое может повлиять на

скорость его динамического применения к документу, — следовательно, нам нужны сотни или даже тысячи правил. В качестве отправной точки была опять взята главная страница Яндекса, стили которой были вынесены в отдельный файл и скопированы 10 раз. Это дало необходимую задержку (которая существенно больше погрешности, вносимой браузерами) и не сильно увеличило сжатый с помощью `gzip` файл.

5.1.2. XHR в <body>

Выполняется `XMLHttpRequest` к CSS-файлу, затем содержимое последнего вставляется через `innerHTML` в `<body>` документа. Случай был выбран просто как базовый, потому что большое число узлов в DOM-дереве делает такую операцию сразу менее эффективной, чем вставка в `<head>`. Да и стили внутри тела документа запрещены стандартами.

```
// чтобы не копировать всем известный код, запишем так:
var xhr = new XMLHttpRequest;
if (xhr) {
  xhr.onreadystatechange = function() {
    try {
      if (xhr.readyState == 4) {
        if (xhr.status == 200) {
// вставим полученные данные прямо в <body>
          document.body.innerHTML +=
            '<style type="text/css">' + xhr.responseText +
            '</style>';
        }
      }
    } catch(e){}
  };
  xhr.open("GET", 'styles.css?'+Math.random(), true);
  xhr.send(null);
}
```

Тут сразу возникает вопрос: как считать применение стилей? Ведь у браузера нет такого события. Немного подумаем и вспомним всем известный и раскритикованный подход с использованием offsetHeight. Он, конечно, будет не самым лучшим выбором, так как добавляет небольшую задержку (пропорциональную числу узлов), но это будет допустимо в рамках выбранного метода (число узлов постоянно для всех измерений).

Итак, для снятия времени применения CSS-правил использовалась следующая конструкция (которая запускалась сразу после обращения к внешнему файлу):

```
var start = new Date();
var _timer = setInterval(function(){
// находим известный элемент документа и проверяем,
// применились ли к нему стили
  if (document.getElementById('neck').offsetHeight < 300) {
// сообщаем о применении
    alert('CSS files loaded in '+(new Date() - start));
// убиваем таймер
    clearInterval(_timer);
  }
}, 10);
```

Конечно, на время загрузки влияют сетевые задержки. Для борьбы с ними (и не только) бралась серия из 15 замеров, и значения, превосходящие текущее среднее более чем в 2 раза, просто отбрасывались (для контроля 3-дельта-выбросов).

Все результаты приведены в конце раздела.

5.1.3. XHR в head

В этом случае код вставлялся уже в `<head>` и применялся ряд методов для разных браузеров (ибо не все хотели через `innerHTML` или `innerText` вставлять полученные данные).

```
var text = xhr.responseText;
var head = document.getElementsByTagName('head')[0];
var style = document.createElement('style');
style.type = 'text/css';
// для IE
if (style.styleSheet) {
  style.styleSheet.cssText = text;
} else {
// для Safari/Chrome
  if (style.innerText == '') {
    style.innerText = text;
// для остальных
```

```
  } else {
    style.innerHTML = text;
  }
}
head.appendChild(style);
```

5.1.4. Быстрый XHR в head

В следующем варианте проверялось прямое добавление стилевых правил к `innerHTML в <head>` (для тех браузеров, которые это поддерживают). Оказалось, что это вариант даже медленнее, чем предыдущий.

Если осуществлять это относительно обычного HTML-документа, то DOM-дерево изменяется быстрее (в IE6/7), поэтому на данный момент в общем случае практикуется именно такой подход.

```
var text = xhr.responseText;
var head = document.getElementsByTagName('head')[0];
if (/WebKit|MSIE/i.test(navigator.userAgent)) {
  var style = document.createElement('style');
  style.type = 'text/css';
  if (style.styleSheet) {
    style.styleSheet.cssText = text;
  } else {
    style.innerText = text;
  }
  head.appendChild(style);
} else {
  head.innerHTML += '';
}
```

5.1.5. DOM-метод

И, наконец, хит сезона. Добавляем новый файл стилей прямо в `<head>` при помощи DOM-методов.

```
var link = document.createElement('link');
document.getElementsByTagName('head')[0].appendChild(link);
link.setAttribute('type','text/css');
link.setAttribute('rel','stylesheet');
link.setAttribute('href','style.css');
```

5.1.6. Результаты

Ниже приведена таблица по исследованным браузерам для всех вариантов. В ней указано время в миллисекундах, прошедшее от начала вызова внешнего файла до окончания применения всех стилей.

Браузер	XHR в <body>	XHR в <head>	Быстрый XHR в <head>	DOM
IE 6	482	379	342	**335**
IE 7	532	364	391	**353**
IE 8b2	370	326	301	**284**
FX 3	420	294	300	**282**
Opera 9	892	894	1287	**764**
Safari 3	—	308	**286**	296
Chrome	—	349	**335**	367

5.1.7. Выводы

Как хорошо видно из таблицы, наиболее быстрым способом для динамического добавления стилей в документ являются DOM-методы почти во всех случаях. Для Safari/Chrome вставка через XHR и специальные методы оказываются быстрее (но не намного). Отдельно хочется отметить довольно медленную работу Opera в таких задачах: по возможности стоит избегать динамических стилей для этого браузера.

Естественно, тут речь идет о выигрышах лишь в десятки и сотни миллисекунд. Но если с самого начала применять самые оптимальные методы при разработке, то ситуаций, когда веб-приложение уже тормозит на несколько секунд (просто загружая процессор на пустом месте), можно будет с легкостью избежать. Ведь на том этапе, когда задержки станут явными, находить и устранять их намного сложнее, чем при изначальном проектировании.

5.2. Оптимизация CSS-структуры

Этот раздел написан после прочтения статей Steve Souders — don't use `@import` (http:/www.stevesouders.com/blog/2009/04/09/dont-use-

import/) и Simplifying CSS Selectors (http://www.stevesouders.com/blog/2009/06/18/simplifying-css-selectors/) — и обсуждения с ним производительности CSS-правил. Данный материал подготовлен с помощью Абановой Ольги (http://www.getincss.ru/).

5.2.1. <link> vs. @import

Существует два способа загрузки файлов стилей: использовать тег `<link>`:

```
<link rel='stylesheet' href='a.css'/>
или импортировать файлы с помощью @import:
<style type='text/css'>
@import url('a.css');
</style>
```

Стоит использовать `<link>` для удобства, но вы должны помнить, что `@import` нужно размещать всегда в самом верху блока стилей, в противном случае они не импортируются.

5.2.2. @import @import

В приведенном ниже примере подключаются два файла стилей: `a.css` и `b.css`. Каждый файл по загрузке занимает ровно 2 секунды, чтобы было удобно отследить влияние на скорость загрузки в дальнейшем. В первом примере применяется `@import` для загрузки обоих файлов стилей. Здесь HTML-документ содержит следующий блок стилей:

```
<style type='text/css'>
@import url('a.css');
@import url('b.css');
</style>
```

Если вы всегда будете использовать только `@import` для загрузки стилей, то проблем с производительностью не будет, хотя, как мы увидим

Рис. 5.1. Параллельная загрузка стилей. Источник: getincss.ru

ниже, это может привести к ошибке с JavaScript. Оба файла загружаются параллельно (рис. 5.1) Но проблемы начинают появляться, если применять `@import` внутри файла стилей либо вместе с `<link>`.

5.2.3. <link> @import

В следующем примере используется тег `<link>` для загрузки `a.css` и `@import` для `b.css`:

```
<link rel='stylesheet' type='text/css' href='a.css'/>
<style type='text/css'>
@import url('b.css');
</style>
```

В IE (тестировалось в 6, 7 и 8) это привело к тому, что файлы загружаются последовательно друг за другом, как показано на рис. 5.2. Соответственно, время загрузки страницы в IE увеличится.

Рис. 5.2. @import блокирует <link> в IE. Источник: getincss.ru

5.2.4. <link> с @import

Тут файл `a.css` загружается через `<link>` и содержит внутри правило `@import` для `b.css`:

В документе:

```
<link rel='stylesheet' type='text/css' href='a.css'/>
```

в `a.css`:

```
@import url('b.css');
```

Этот способ также приводит к тому, что файлы загружаются последовательно (рис. 5.3), а не параллельно, и теперь это происходит не только в IE, но и в остальных браузерах. Если подумать — все логично: браузер загружает `a.css` и начинает анализировать его. Как только внутри обнаружено правило `@import`, начинается загрузка файла `b.css`.

Рис. 5.3. @import блокирует <link> не только в IE. Источник: getincss.ru

5.2.5. Блоки <link> с @import

Незначительное отличие от предыдущего примера привело к удивительным результатам: в IE. `<link>` используется для вызова `a.css` и для нового файла `proxy.css`, который содержит только `@import` для `b.css`.

В HTML-коде:

```
<link rel='stylesheet' type='text/css' href='a.css'>
<link rel='stylesheet' type='text/css' href='proxy.css'>
```

В `proxy.css`:

```
@import url('b.css');
```

Результаты эксперимента в IE показаны на рис. 5.4. Первый запрос — HTML-документ. Второй запрос — `a.css` (2 секунды). Третий — proxy.css. Четвертый — `b.css` (2 секунды). И вот что удивительно, IE не хочет начинать загрузку `b.css`, пока файл `a.css` не будет загружен полностью. Во всех других браузерах такого сюрприза не происходит, что приводит к более быстрой загрузке страницы (см. рис. 5.5).

Рис. 5.4. Результаты в IE. Источник: getincss.ru

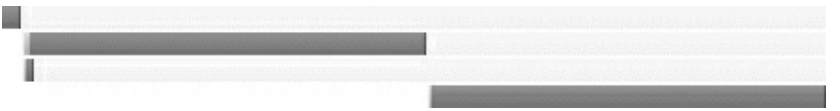
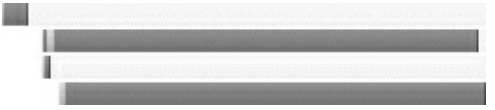


Рис. 5.5. Результаты в других браузерах. Источник: getincss.ru

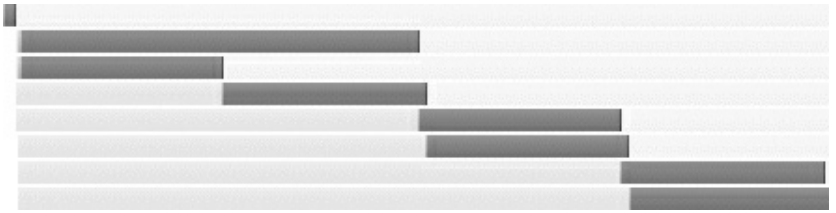


5.2.6. Много @import

Применение сразу нескольких правил `@import` в IE приводит к тому, что файлы загружаются не в том порядке, в котором они указаны в коде. В этом примере используется 6 файлов стилей (каждый из которых загружается по 2 секунды), за которыми следует JS-скрипт (4 секунды для загрузки).

```
<style type='text/css'>
@import url('a.css');
@import url('b.css');
@import url('c.css');
@import url('d.css');
@import url('e.css');
@import url('f.css');
</style>
<script src='one.js' type='text/javascript'></script>
```

Рис. 5.6. Много @import. Источник: getincss.ru



На рис. 5.6 мы увидим, что самый долгий по загрузке — это скрипт. Несмотря на то, что он указан после стилей, в IE он загружается первым. Если в скрипте содержится код, который зависит от применяемых стилей (getElementsByClassName, и т. п.), это может привести к ошибкам работы скрипта, так как он загружается раньше, чем стили.

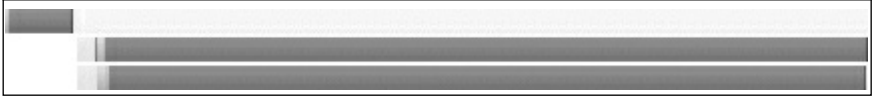
5.2.7. <link> <link>

Проще и безопасней задействовать `<link>` для загрузки стилей:

```
<link rel='stylesheet' type='text/css' href='a.css'>
<link rel='stylesheet' type='text/css' href='b.css'>
```

Использование <link> обеспечивает параллельную загрузку файлов во всех браузерах (см. рис. 5.7). Применение <link> также гарантирует, что файлы будут загружены именно в том порядке, который указан в коде документа.

Рис. 5.7. Использование <link>. Источник: getincss.ru



Для нас особенно плохо то, что ресурсы могут быть загружены в другом порядке, нежели указано в документе. Все браузеры должны заглядывать вперед перед загрузкой стилей для извлечения правил @import и начинать их загрузку немедленно. До тех пор, пока браузеры не реализуют это, стоит избегать использования @import и загружать стили только с помощью <link>.

5.2.8. Упрощаем CSS-селекторы

Для большинства сайтов возможный выигрыш в производительности после оптимизации CSS-селекторов будет крайне незначительным и не будет стоить потраченного времени. Есть несколько типов CSS-правил (например, expression для IE) и взаимодействий с JavaScript, которые могут существенно замедлить страницу. Именно на них и нужно концентрировать усилия.

Большая часть информации о быстродействии CSS-селекторов может быть получена из статьи David Hyatt Пишем эффективный CSS для интерфейсов в Mozilla (https://developer.mozilla.org/en/Writing_Efficient_CSS). Стоит привести оттуда одну цитату:

«Система стилей находит элемент, соответствующий CSS-правилу, начиная с правого конца селектора и двигаясь налево. Пока мы еще находим элементы, соответствующие селектору, мы продолжаем двигаться. Остановка возможна только в случае нахождения элемента, соответствующего всему CSS-селектору, или невозможности такой элемент найти для какой-то части селектора».

Не очень понятно, как устроен движок CSS-селекторов в других браузерах, например, в IE (а если принять во внимание тесты CSS-производительности из книги «Разгони свой сайт», то возникает предположение, что в Opera движок разбора CSS-селекторов работает как раз слева направо). Может быть, он использует комбинированный подход: как справа налево,

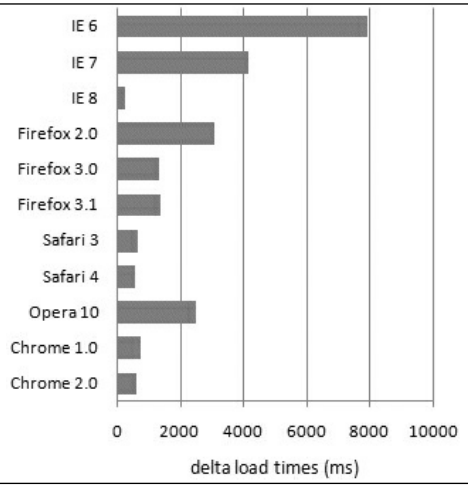
так и слева направо. Переключение может происходить по какому-то признаку (например, по наличию `#id`).

Однако по заявлению Виталия Харисова, руководителя группы HTML-верстки в Яндексе, тесты показывают (http://clubs.ya.ru/yacf/replies.xml?item_no=338&ncrnd=3604), что все браузеры применяют селекторы одинаково — справа налево.

Благодаря вышесказанному мы можем теперь сфокусировать оптимизационные усилия на тех CSS-селекторах, правая часть которых будет соответствовать большому числу элементов на странице. Давайте рассмотрим для примера `DIV DIV DIV P A.class0007 {}`. У этого селектора присутствует 5 уровней вложенности потомков, для которых необходимо найти соответствие в DOM-дереве. Это выглядит очень ресурсоемко, однако при взгляде на самую правую часть этого селектора, `A.class0007`, мы прекрасно понимаем, что ей соответствует только 1 элемент на странице, поэтому браузеру очень легко установить точное соответствие.

Ключевым моментом в оптимизации CSS-селекторов является самая правая часть, часто также называемая «ключевым селектором». Вот пример гораздо более ресурсоемкого селектора: `A.class0007 * {}`. Хотя он может и выглядеть просто, но для браузера очень тяжело его вычислить. Поскольку браузер будет двигаться справа налево, он начнет со всех элементов, которые подходят под ключевой селектор (`*`). Это означает, что браузер попытается проверить все элементы на странице и установить,

Рис. 5.8. Разница во времени загрузки (в мс) для универсального селектора. Источник: stevesouders.com



нужно ли к ним применить этот стиль. На следующей диаграмме показана разница во времени между тестами для универсального селектора и прошлыми тестами для наследственных селекторов.

Очевидно, что CSS-селекторы с ключевым элементом, который соответствует большому числу узлов, будут вычисляться существенно дольше и заметно тормозить загрузку страницы. В качестве других примеров селекторов, которые могут привести к большим вычислениям со стороны браузера, можно отнести:

```
A.class0007 DIV {}
#id0007 > A {}
.class0007 [href] {}
DIV:first-child {}
```

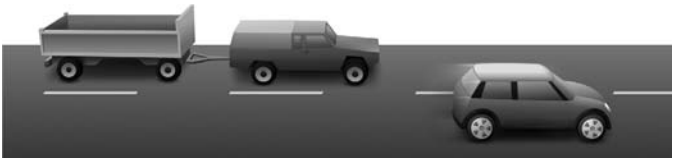
Не все CSS-селекторы существенно влияют на производительность, даже те, которые могут выглядеть сложно. При оптимизации стоит фокусироваться на устранении CSS-правил с ключевым селектором, соответствующим большому числу элементов. Это становится особенно важно для Веб2.0-приложений с большим числом узлов DOM-дерева, огромным количеством CSS-правил и отрисовок страниц.

Некоторой проблемой является то, что почти все известные JavaScript-библиотеки разбирают селекторы слева направо. Таким образом, мы должны писать два вида селекторов: оптимизированных под браузеры и оптимизированных под JavaScript-библитеки. На данный момент только немногие поддерживают нотацию справа налево, например, CSS1-ветка YASS (http://yass.webo.in/).

5.3. Пишем эффективный CSS

В книге «Разгони свой сайт» тема производительности CSS-селекторов уже поднималась. Несмотря на то, что выводы были подкреплены значительным объемом исследований, основной вопрос — как же должна выглядеть эффективная таблица стилей, которая обеспечивает наискорейшее отображение документа на экране, — так и остался без ответа.

Для прояснения этой ситуации были проведены дополнительные исследования, базирующиеся на уже известных фактах: «наиболее эффективны селекторы, не использующие тегов», и «классы работают быстрее, чем идентификаторы».



Также известно, что селекторы обладают различной сложностью (например, селектор `.class1 .class2`, очевидно, должен отрабатывать медленнее, чем просто `.class2`).

5.3.1. Модель

Для уточнения существенных факторов и относительного ранжирования известных правил по написанию эффективного CSS-кода была взята за основу следующая формула:

```
Время отрисовки = Размер DOM-дерева * Число CSS-селекторов *
Сложность стилевых правил * Время отрисовки одного правила +
Время создания документа
```

При взгляде на эту формулу становится очевидным, что нам нужно брать усредненную сложность правил по всей таблице стилей, т. е. подставлять в формулу сумму сложностей всех CSS-селекторов, разделенную на их число.

5.3.2. Уточнения по ходу

Также практически сразу становится ясно, что в формуле фигурирует не размер всего DOM-дерева, а число элементов, на которые влияет данный селектор (это, в частности, объясняет, почему универсальный селектор (`*`) такой ресурсоемкий). Для уточнения этого момента была проведена серия тестов с одинаковым DOM-деревом и различными CSS-правилами (одно применялось ко всему дереву, а другое — только к десятой его части).

Еще не стоит забывать о наличии у браузеров собственной таблицы стилей, которая применяется к **каждой** странице, выводящейся на экран. Размер этой таблицы можно выяснить достаточно просто: нужно всего лишь открыть две страницы с разным (и достаточно большим) числом CSS-правил, но одинаковым DOM-деревом и проверить, насколько замедлилась загрузка. Зная отношение размера двух таблиц стилей, можно вычислить неизвестный размер таблицы стилей самого браузера (для основных браузеров он составил в районе 30-50 правил, для IE — порядка 200).

И еще один момент, который всплыл по ходу расследования: имеет значение размер полного DOM-дерева, не только число тегов, но и число текстовых узлов, хотя это никак и не влияет на основные выводы.

5.3.3. Результаты

В ходе проведения различных тестов модель удалось уточнить и показать, что время создания документа зависит от числа узлов в дереве (что вполне очевидно). Это можно проверить двумя наборами тестов: на увеличении числа CSS-селекторов без увеличения DOM-дерева и на увеличении DOM-дерева без увеличения числа CSS-селекторов. При этом хорошо видно, что время создания документа не является постоянным и несколько увеличивается при увеличении размера документа.

Также было проверено, насколько составные селекторы отрабатывают медленнее своих элементарных собратьев (имеется в виду разница между `.class1, .class1 .class2` и `.class1 .class2 .class3`). Разница была зафиксирована, но оказалась несущественной (каждое звено прибавляет примерно 10-20% к общей сложности селектора).

Итак, после всех уточнений, формула приобрела следующий вид:

```
T = (сумма(DOM1 * K) + DOM2 * In) * t + DOM2 * L
```

Здесь:

- `T` — время отображения документа на экране;
- `DOM1` — число элементов, на которое может повлиять данное CSS-правило (разбор CSS-правил в браузерах идет справа налево);
- `DOM2` — размер всего DOM-дерева;
- `K` — сложность каждого отдельного CSS-правила в таблице стилей, от 1 до 1,5;
- `In` — число встроенных CSS-правил в браузере, порядка 40-200;
- `t` — характерное время обработки одного правила для одного узла дерева, находится в районе 0,0001…0,0005 мс;
- `L` — характерные издержки на создание одного элемента DOM-дерева, находятся в районе 0,0005…0,005 мс.

Данная модель позволила аппроксимировать время отображения страницы с точностью 10% (в редких случаях 20%, — видимо, есть еще много неучтенных факторов, например, особенности выделения памяти различными браузерами). Тестирование проводилось на документах от 5000 DOM-узлов и от 0 CSS-правил.

5.3.4. Выводы

Анализ предложенной и проверенной модели позволяет сделать огромное количество весьма интересных выводов. Давайте остановимся на некоторых из них.

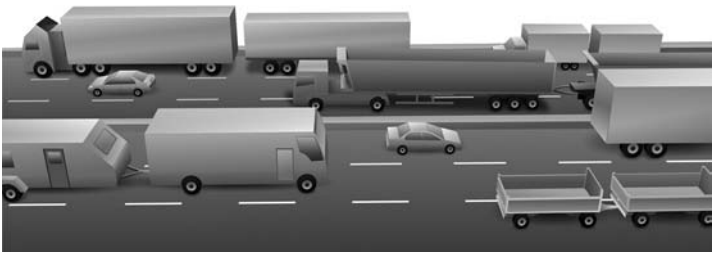
- Размер DOM-дерева играет основную роль. Просто наиглавнейшую. Поэтому совет на все времена: уменьшайте DOM всеми возможными способами. Уменьшение его (как хорошо видно из итоговой формулы) на 20% приведет к пропорциональному ускорению отображения страницы.
- Стоит также учесть, что в формуле фигурирует не только общий размер дерева, но и число элементов, которые обрабатываются при применении CSS-селектора. Именно по этой причине неэффективно использовать универсальный (`*`) селектор и теги: они охватывают существенное количество элементов.
- В качестве альтернативы применения общих тегов и универсальных селекторов можно назвать два выхода.
  1. Использовать уникальные теги для уникальных элементов на странице (например, для скругленных уголков использовать редкие теги — `ins`, `del`, `q`, `u`, `b`, `i`).
  2. Использовать уникальные классы для каждого набора стилевых правил.

  Если первый подход может быть применим для небольших сайтов (например, для уменьшения размера HTML-кода), то в случае средних и крупных проектов однозначно стоит применять второй подход (его, кстати, вовсю рекомендует и Виталий Харисов в своем своде правил для эффективного CSS и фреймворке Monkey Joe, http://clubs.ya.ru/yacf/).
- Использование сложных правил (с несколькими звеньями селекторов) может быть оправдано (это не влечет значительных издержек), однако если применять везде уникальные классы, то наследование обычно пропадает само собой.
- В качестве глобального сброса стилевых правил («ластик») можно рекомендовать сбрасывать правила только у тех элементов, которые отображаются. Например, если на странице 90% DOM-дерева — это `div`, для которых не нужны никакие правила по умолчанию, то переход от глобального «ластика» к локальному или вообще его устранение за счет индивидуальных правил способно несколько увеличить производительность).
- Оптимизировать число CSS-правил стоит, если их больше 100-200 (ибо в противном случае правила самого браузера будут перекрывать все ваши усилия по увеличению эффективности).

Также стоит отметить, что по результатам тестирования Виталия Харисова (http://clubs.ya.ru/yacf/replies.xml?item_no=338) неиспользуемые CSS-правила добавляют некоторую задержку в отображении страницы (до 10% от времени отрисовки), поэтому их тоже стоит избегать.

Для средней HTML-страницы время ее отображения (размер DOM-дерева — 1000 элементов, CSS-правил — порядка 500, каждое из них в среднем применяется к 40% элементов) составит порядка 100 мс. Простой оптимизацией можно уменьшить этот показатель вдвое (например, сузив область воздействия самих селекторов, если DOM-дерево уменьшить не получается).

5.4. Стыкуем асинхронные скрипты



Этот раздел написан под впечатлением от статьи Steve Souders (автора знаменитых советов Yahoo! касательно клиентской производительности) «Coupling Async Scripts» (http://www.stevesouders.com/blog/2008/12/27/coupling-async-scripts/). Steve проанализировал поведение JavaScript-файлов при загрузке и предложил несколько путей для обхода их «блокирующих» свойств.

Если скрипты загружаются в обычном порядке (`<script src="...">`), то они блокируют загрузку всех остальных компонентов страницы (в последних версиях Firefox и в Safari это не так, но речь идет в основном про 70% пользователей с IE) и блокируют отрисовку всей той части страницы, которая расположена ниже вызова скриптов по HTML-коду. Асинхронная загрузка скриптов (например, при помощи динамического создания объектов после срабатывания комбинированного события `window.onload`) предотвращает такое поведение браузера, что ускоряет загрузку страницы.

Единственная проблема с асинхронной загрузкой скриптов заключается в их взаимодействии с внутренними скриптами страницы (а также с другими внешними скриптами), которые используют переменные, определенные во внешнем скрипте. Если внешний скрипт загружается асинхронно безо всяко-

го представления о внутреннем коде HTML-страницы, то вполне возможна ситуация (и она будет возникать в большинстве случаев), когда некоторые переменные будут не определены на момент их использования. Поэтому необходимо убедиться, что внешние скрипты, загруженные асинхронным образом, и внутренние скрипты страницы состыкованы: внутренние скрипты не выполняются до тех пор, пока асинхронные скрипты полностью не загрузятся.

Существует несколько стандартных путей для стыковки асинхронно загружаемых скриптов с другим JavaScript-кодом.

- **window onload**. Выполнение внутреннего JavaScript-кода может быть привязано к событию `window onload`. Это очень просто в использовании, но часть скриптов может быть выполнена раньше.
- **onreadystatechange у скрипта**. Внутренний код может быть привязан к событию `onreadystatechange` и(или) `onload` (необходимо будет использовать оба варианта, чтобы покрыть все популярные браузеры). В этом случае кода будет больше, он будет более сложным, но будет гарантия, что он исполнится сразу после загрузки соответствующих внешних файлов.
- **Встроенные вызовы**. Внешние скрипты могут быть модифицированы таким образом, чтобы включать в самом конце вызов небольшого участка кода, который вызовет соответствующую функцию из внутреннего кода. Это все замечательно, если внешние и внутренние скрипты разрабатываются одной и той же командой. Но в случае использования сторонних разработок это не обеспечит всей необходимой гибкости для связки внешних скриптов с внутренним кодом.

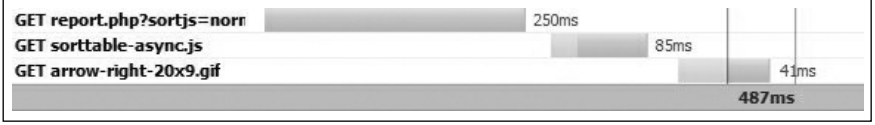
В этом разделе параллельно освещаются два вопроса: как асинхронные скрипты ускоряют загрузку страницы и как можно состыковать асинхронные скрипты и внутренние, используя модифицированный вариант загрузчика от Джона Ресига (автора jQuery) — шаблон двойного тега `<script>`.

5.4.1. Обычные вызовы скриптов

Если добавить скрипт на страницу обычным способом (через `<script src="...">`), диаграмма загрузки будет примерно следующей.

Хотя скрипт и функционирует, но это не сделает нас намного счастливее, ибо загрузка страницы замедлится. На рис. 5.9 хорошо видно, как скрипт (по имени `sorttable-async.js`) блокирует все остальные HTTP-запросы на странице (в частности, `arrow-right-20x9.gif`), что замедляет загрузку страницы. Все диаграммы загрузки сняты при помощи Firebug 1.3 beta. В этой версии Firebug красной линией отображается событие `onload` (а синяя линия соответствует событию `domcontentloaded`). Для версии с обычным вызовом скрипта событие `onload` срабатывает на 487-й миллисекунде.

Рис. 5.9. Диаграмма загрузки скриптов в обычном случае. Источник: stevesouders.com



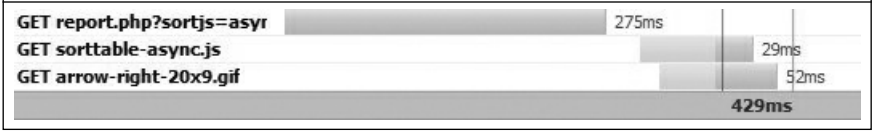
5.4.2. Асинхронная загрузка скриптов

Скрипт `sorttable-async.js` в данном случае не является необходимым для первоначального отображения страницы. Такая ситуация (внешние скрипты, которые не используются для первоначального отображения страницы) является кандидатом номер 1 для внедрения асинхронной загрузки. Вариант с асинхронной загрузкой скриптов подключает этот скрипт, используя DOM-методы для создания нового тега `<script>`:

```
var script = document.createElement("script");
script.src = "sorttable-async.js";
script.text = "sorttable.init()"; // это объясняется чуть ниже
document.getElementsByTagName("head")[0].appendChild(script);
```

Диаграмма HTTP-загрузки для асинхронной загрузки скриптов изображена на рис. 5.10. Стоит обратить внимание, как асинхронный подход предотвращает блокирующее поведение: `sorttable-async.js` и `arrow-right-20x9.gif` загружаются параллельно. Это снижает общее время загрузки дл 429 мс.

Рис. 5.10. Диаграмма загрузки скриптов в асинхронном случае, источник: stevesouders.com



5.4.3. Шаблон двойного скрипта от Джона Ресига

Асинхронная загрузка скриптов позволяет ускорить загрузку страницы, но в этом случае остается, что улучшить. По умолчанию скрипт вызы-

вает «сам себя» при помощи прикрепления sorttable.init() к обработчику события onload для этого скрипта. Некоторое улучшение производительности (и уменьшение кода) может быть достигнуто, если вызвать sorttable.init() внутри тега <script>, чтобы вызвать его сразу же после загрузки внешнего скрипта (подключенного через src).

Выше описано три способа по стыковке внутреннего кода с асинхронной загрузкой внешних скриптов: window onload, onreadystatechange у скрипта и встроенный в скрипт обработчик. Вместо всего этого можно использовать технику от Джона Ресига — шаблон двойного тега <script>. John описывает, как связать внутренние скрипты с загрузкой внешнего файла, следующим образом:

```
<script src="jquery.js" type="text/javascript">
  jQuery("p").addClass("pretty");
</script>
```

В этом случае код внутри <script> срабатывает только после того, как загрузка и инициализация внешнего скрипта завершилась. У этого подхода по стыковке скриптов есть несколько очевидных преимуществ:

- проще: один тег <script> вместо двух;
- прозрачнее: связь внутреннего и внешнего кодов более очевидна;
- безопаснее: если внешний скрипт не загрузится, внутренний код не будет выполнен, что предотвратит появление ошибок, связанных с неопределенными переменными.

Это замечательный шаблон для асинхронной загрузки внешних скриптов. Однако чтобы его использовать, нам придется внести изменения как во внутренний код, так и во внешний файл. Для внутреннего кода придется добавить уже упомянутую выше третью строку, которая выставляет свойство script.text. Для завершения процесса стыковки нужно добавить в конец sorttable-async.js:

```
var scripts = document.getElementsByTagName("script");
var cntr = scripts.length;
while ( cntr ) {
  var curScript = scripts[cntr-1];
  if ( -1 != curScript.src.indexOf("sorttable-async.js") ) {
    eval( curScript.innerHTML );
    break;
  }
  cntr--;
}
```

Этот код проходится по всем скриптам на странице, находит необходимый блок, который должен загрузить сам себя (в этом случае это скрипт с src, содержащим sorttable-async.js). Затем он выполняет код, который добавлен к скрипту (в этом случае —sorttable.init()), и таким образом вызывает сам себя. (Небольшое замечание: хотя при загрузке скрипта текст в нем был добавлен при помощи свойства text, обращение к нему происходит при помощи свойства innerHTML. Это необходимо для обеспечения кроссбраузерности.) При помощи такой оптимизации мы можем загрузить внешний файл скрипта, не блокируя загрузку других ресурсов, и максимально быстро выполнить привязанный к данному скрипту внутренний код.

Стоит также заметить, что описываемая техника может быть заменой только прямой модификации внешнего скрипта. В случае невозможности такой модификации мы можем использовать только установление проверки загрузки по интервалу:

```
var _on_ready_execution = setInterval(function() {
  if (typeof urchinTracker === 'function') {
    urchinTracker();
    clearInterval(_on_ready_execution);
  }
}, 10);
```

Этот подход был уже описан в книге «Разгони свой сайт» (http://speedupyourwebsite.ru/books/speed-up-your-website/), однако он предполагает дополнительную нагрузку на процессор для постоянной проверки готовности искомого скрипта и не срабатывает в случае недоступности внешнего файла: проверка продолжает выполняться.

Однако в случае проверки по интервалу нам совсем не нужно модифицировать внешний файл, в случае же двойного использования тега `<script>` это просто необходимо. Проверку по интервалу можно улучшить, если по истечении некоторого времени (5—10 секунд, например) перезапускать загрузку внешнего файла (меняя исходный тег `<script>` при помощи уникального GET-параметра), а после нескольких неудачных перезапусков вообще прекращать загрузку (возможно, с каким-то сообщением об ошибке).

5.4.4. «Ленивая» загрузка

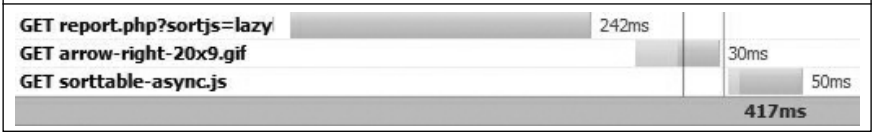
Общее время загрузки может быть уменьшено еще больше, если использовать «ленивую загрузку» скрипта (загружать его динамически как

часть обработчика события `onload`). Мы можем просто оборачивать заявленный выше код в обработчик события `onload`:

```
window.onload = function() {
  var script = document.createElement("script");
  script.src = "sorttable-async.js";
  script.text = "sorttable.init()";
  document.getElementsByTagName("head")[0].appendChild(script);
}
```

В этом случае нам жизненно необходима техника стыковки скриптов. Преимущество данного подхода заключается в том, что событие onload срабатывает раньше — это подтверждается на рис. 5.11. Событие onload, отмеченное красной линией, срабатывает примерно на 320-й миллисекунде. Стоит отметить, что в таком случае мы увеличиваем общее время загрузки страницы (за счет некоторой синхронизации загрузки скриптов с общим процессом загрузки страницы), но ускоряем появление первоначальной картинки в браузере пользователя.

Рис. 5.11. Диаграмма загрузки в случае «ленивой» загрузки скриптов, источник stevesouders.com



5.4.5. Заключение

Асинхронная и «ленивая» загрузка скриптов уменьшает общее время загрузки страницы тем, что предотвращает обычное блокирующее поведение скриптов (буквально «клин клином вышибает»). В качестве демонстрации этого можно привести различные варианты добавления скрипта на тестовую страницу:

- обычная загрузка скриптов — 487 мс;
- асинхронная загрузка — 429 мс;
- «ленивая» загрузка — ~320 мс.

Выше показано время, после которого срабатывает событие `onload`. Для других веб-приложений применение асинхронной загрузки для улучше-

ния производительности может привести к гораздо более впечатляющим результатам и быть намного более приоритетным. В нашем случае асинхронная загрузка скриптов немного лучше (~400 мс против 417 мс). В обоих случаях нам нужно каким-то образом связывать внутренние скрипты с внешними.

5.5. Стыкуем компоненты в JavaScript

После вышеописанного материала можно задуматься и о модульной загрузке какого-либо сложного JavaScript-приложения. Предложенный подход в таком случае будет довольно громоздким: нам нужно будет в конец каждого модуля вставлять загрузчик следующих модулей. А если нам на разных страницах требуются различные наборы модулей и разная логика их загрузки? Тупик?

Ан нет. Не зря упоминается в самом начале прошлого раздела о событии `onload / onreadystatechange` для скриптов. Используя их, мы можем однозначно привязать некоторый код к окончанию загрузки конкретного модуля. Дело за малым: нам нужно определить этот самый код каким-либо образом.

5.5.1. Решение первое: дерево загрузки

В качестве наиболее простого способа определить порядок загрузки модулей на конкретной странице можно предложить глобальный массив, содержащий в себе дерево зависимостей. Например, такое:

```
var modules = [
  [0, 'item1', function(){
    alert('item1 is loaded');
  }],
  [1, 'item2', function(){
    alert('item2 is loaded');
  }],
  [1, 'item3', function(){
    alert('item3 is loaded');
  }]
];
```

В качестве элемента этого массива у нас выступает еще один массив. Первым элементом идет указание родителя (0 в том случае, если элемент является корнем и должен быть загружен сразу же), далее имя файла или его алиас. Последней идет произвольная функция, которую можно выполнить по загрузке.

Давайте рассмотрим, каким образом можно использовать данную структуру.

```
/* перебор и загрузка модулей */
function load_by_parent (i) {
  i = i || 0;
  var len = modules.length,
    module;
/* перебираем дерево модулей */
  while (len--) {
    module = modules[len];
/* и загружаем требуемые элементы */
    if (!module[0]) {
      loader(len);
    }
  }
}

/* объявляем функцию-загрузчик */
function loader (i) {
  var module = modules[i];
/* создаем новый элемент script */
  var script = document.createElement('script');
  script.type = 'text/javascript';
/* задаем имя файла */
  script.src = module[1] + '.js';
/* задаем текст внутри тега для запуска по загрузке */
  script.text = module[2];
/* запоминаем текущий индекс модуля */
  script.title = i + 1;
/* выставляем обработчик загрузки для IE */
  script.onreadystatechange = function() {
    if (this.readyState === 'loaded') {
/* перебираем модули и ищем те, которые нужно загрузить */
      load_by_parent(this.title);
    }
```

```
  };
/* выставляем обработчик загрузки для остальных */
  script.onload = function (e) {
/* исполняем текст внутри тега (нужно только для Opera) */
    if (/opera/i.test(navigator.userAgent)) {
      eval(e.target.innerHTML);
    }
/* перебираем модули и ищем те, которые нужно загрузить */
    load_by_parent(this.title);
  };
/* прикрепляем тег к документу */
  document.getElementsByTagName('head')[0].appendChild(script);
}

/* загружаем корневые элементы */
load_by_parent();
```

Мы можем вынести загрузку корневых элементов в событие загрузки страницы, а сами функции — в какую-либо библиотеку, либо объявить прямо на странице. Задавая на каждой странице свое дерево, мы получаем полную гибкость в асинхронной загрузке любого количества JavaScript-модулей. Стоит отметить, что зависимости в таком случае разрешаются «от корня — к вершинам»: мы сами должны знать, какие базовые компоненты загрузить сначала, а потом загрузить более продвинутые.

5.5.2. Решение второе: загрузка через DOM-дерево

Кроме того, подобной проблемой уже занимался Андрей Сумин и даже предложил свое решение в виде библиотеки JSX (http://jsx.ru/), которая позволяет назначать список зависимостей через DOM-дерево. Для этого у элементов, которые требуют загрузки каких-либо модулей для взаимодействия с пользователем, назначается класс с префиксом `jsx-component`, а далее идет уже список компонентов. Сама библиотека обходит DOM-дерево, находит все модули, которые нужно загрузить, и последовательно их загружает. Просто замечательно.

Но что, если нам требуется поменять обработчик по загрузке этого модуля? Как его задавать? Сама JSX использует атрибуты искомых узлов DOM-дерева сугубо для определения параметров этих модулей. Это достаточно удобно: ведь таким образом можно назначить и инициализатор модуля.

Также библиотека позволяет отслеживать повторную загрузку модулей, осуществлять догрузку модулей в случае плохого соединения и даже объединять разные модули в один исходный файл через систему алиасов. Таким образом, проблема асинхронной загрузки произвольного дерева модулей оказывается решенной. В случае JSX задача разрешается в обратном порядке: мы указываем основной файл (вершину дерева зависимостей), а он уже загружает все необходимые ему модули либо проверяет, что модули загружены.

Это все?

5.5.3. Решение третье: JSX+7YASS

Почти. После недолгих раздумий JSX была взята за основу для построения модульной системы, которая могла бы стать основой для гибких и динамических клиентских приложений. Удалось совместить оба описанных выше подхода, что обеспечило все видимые функциональные требования к такого рода системе.

Для примера можно рассмотреть следующий участок HTML-кода:

```
<div id="item1" class="yass-module-utils-base-dom">
  <span id="item2" class="yass-module-dom"
    title="_('#item2')[0].innerHTML = 'component is load-
ing...';">
  </span>
</div>
```

Давайте разберемся, какую логику загрузки он обеспечивает.

1. YASS при инициализации обходит DOM-дерево документа и выбирает все узлы с классом `yass-module-*`.
2. После этого формируется два потока загрузки модулей: для `dom-base-utils` и для `dom`. Причем в последнем случае загрузки фактически не будет: загрузчик дождется, пока состояние компонента `dom` будет выставлено в `loaded`, и только потом запустит (через `eval`) код, записанный в `title` этого элемента (в данном случае это `span`).
3. Первый поток загрузки асинхронно вызовет три файла с сервера: `yass.dom.js`, `yass.base.js` и `yass.utils.js`. По загрузке **всех** этих модулей (ибо они вызваны в цепочке зависимостей, и в данном случае `dom` зависит от `base`, который зависит от `utils`) будут вызваны соответствующие инициализационные функции (если они определены). Таким образом, возможны два типа об-

работчиков: непосредственно по загрузке компонента (будет вызвано для всех компонентов в цепочке) и после загрузки всей заданной цепочки компонентов (в нашем случае это `dom-base-utils`).

4. Если мы хотим каким-то образом расширить нашу цепочку, то можем в конце каждого из указанных файлов прописать загрузку какой-либо другой цепочки (например, `base-callbacks`), которая «заморозит» загрузку модуля `base` до получения `callbacks`. Сделать это можно (имея в виду, что расширяем зависимости модуля `base`) следующим образом:

```
_.load('callbacks-base');
```

5. Предыдущий шаг может быть также выполнен при помощи самого DOM-дерева: нам нужно будет прописать для произвольного элемента класс `yass-module-callbacks-base`. Это добавит в дерево зависимостей искомую цепочку.

Для большей ясности описанное выше конечное дерево загружаемых модулей можно представить так:

```
dom
  -> base
    -> utils
    -> callbacks
```

Очевидно, что возможно и гораздо более глубокое дерево зависимостей, загружаемых асинхронно. Но при его реализации не стоит забывать о накладных издержках на передачу и инициализацию каждого файла: может оказаться так, что выгоднее объединить часть модулей в один большой файл.

Естественно, весь указанный функционал уже добавлен в последнюю версию YASS (http://yass.webo.in/).

5.6. Что такое CDN и с чем его едят

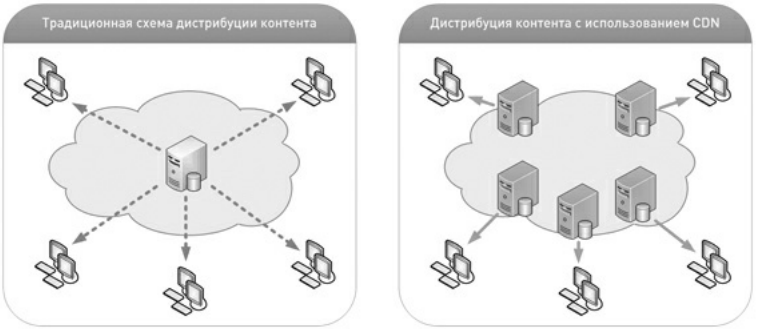
В заключение главы давайте рассмотрим применение аппаратных решений для оптимизации структуры веб-страниц вашего сайта.

CDN (*англ. Content Delivery Network*) — распределенная сеть хранения данных. Предназначена как для обеспечения отказоустойчивости, так и для максимально быстрого времени ответа при запросе файлов. Инфор-

мация данного раздела подготовлена при помощи специалистов первой CDN в России — NGENIX.

После общения с множеством специалистов возникла следующая проблема: все знают, что CDN — это очень хорошо, это правильно, это стоит использовать. Данный раздел пытается ответить на этот вопрос и объяснить, в каких случаях CDN может решить возникающие проблемы и ряд технологических и бизнес-задач.

Рис. 5.12. Схема CDN, источник: ngenix.net



5.6.1. Доступность контента

Нередко бывает, что каждый час простоя сайта интернет-магазина измеряется в весьма значительных суммах. Если отказ сервера, на котором расположен сайт, произошел ночью (по времени технической поддержки), а магазин должен быть доступен 24 часа в сутки (например, в связи с широким географическим покрытием пользователей), то это может вылиться в значительные финансовые потери.

Распределенная сеть серверов позволяет легко решить эту проблему: если отказывает один из имеющихся серверов, то его место тут же занимает самый близкий из сетевых соседей. Таким об-

разом, конечный пользователь не замечает каких-либо перебоев в работе сайта.

Дополнительно данные в распределенной сети еще и резервируются на большом количестве серверов, что сводит вероятную возможность их потери практически к нулю: для этого необходимо, чтобы разом отказала вся сеть, а такое статически невозможно, если только не будет заранее тщательно спланировано злоумышленниками с привлечением весьма дорогих технологических средств.

5.6.2. Высокая скорость загрузки

Благодаря тому, что сетевой маршрут между конечными пользователями и серверами с информацией снижен до минимума (а сами серверы отвечают крайне быстро), скорость прохождения запросов будет весьма значительной.

В качестве примера стоит привести следующие цифры: время ответа обычного сервера, расположенного на VPS (даже не на shared-хостинге), составляет 50-200 мс (в зависимости от различных условий, в том числе от загруженности канала). Для CDN это число очень редко превышает 10 мс.

5.6.3. Снижение нагрузки на сервер — источник информации

Пользовательская нагрузка при использовании CDN распределяется между серверами сети, а исходный сервер может быть использован только для получения обновлений информации. Если обновления происходят крайне редко (и на сайте не представлены динамические страницы, зависящие от поведения пользователя), то для исходного сервера просто не нужны большие мощности, и он может быть заменен на сервер минимальной конфигурации.

Таким образом, вся серверная нагрузка может быть возложена на CDN, а основной сервер будет отвечать только за актуальность представляемой информации.

5.6.4. Размещение «тяжелого» контента

В некоторых случаях (хостинг программного обеспечения или медиа-материалов, создание интерактивных промо-сайтов) исходный сервер с информацией может быть загружен «медленными» запросами (которые могут длиться минутами и часами, расходуя серверные ресурсы и не давая осуществить более быстрые запросы для получения остальной

информации на сайте). В этом случае также выход один — использовать CDN.

CDN позволяет обеспечить одновременное обслуживание десятков и сотен тысяч «медленных» запросов, не требуя при этом дополнительной квалификации от персонала, отвечающего за сайт с исходным контентом. Проводя красочную промо-кампанию, в результате которой планируется обслуживать пользовательские запросы на файлы в несколько (десятков, сотен) Мб, стоит обратить внимание именно на сеть распределенных серверов: она гарантированно не откажет в результате высокой нагрузки.

5.6.5. Отказоустойчивость и безопасность

Сеть CDN размещается на мощнейших технологических площадках и насчитывает сотни и тысячи серверов, что увеличивает стоимость эффективной DDoS-атаки на нее до фантастических величин. Если ваш бизнес не приносит миллионы долларов в день, то вы можете спокойно разместить свой сайт в такой сети и будете защищены от нападок недобросовестных конкурентов.

5.6.6. Масштабируемость и эластичность по нагрузке

Сеть распределенных серверов обладает еще одним преимуществом: она позволяет легко наращивать мощности, от обслуживания 1000 человек день до нескольких миллионов и более. Таким образом, резкое увеличение нагрузки (выход нового продукта, статья в известном издании, промо-кампания) никак не скажется на доступности вашего сайта, его содержимое будет отдаваться по-прежнему быстро и без каких-либо перебоев.

5.6.7. CDN в России

Компания NGENIX первой в России начала предоставлять услуги распределенной доставки и дистрибуции цифрового контента с применением технологии CDN на базе собственной мультисервисной сети доставки контента NGENIX CDN с точками присутствия в крупнейших телекоммуникационных центрах страны и ближнего зарубежья.

Сегодня решения и сервисы NGENIX помогают медиа-компаниям и контент-провайдерам распространять в Интернете тяжелый мультимедийный контент на высокой скорости с широким охватом российской Интернет-аудитории. Сеть NGENIX CDN является универсальной платформой для доставки статического и динамического контентов, ускорения загрузки Интернет-сайтов, повышения производительности веб-приложений, вещания видео, цифровой дистрибуции программного обеспечения, музыки, игр и т. п.

Уникальная распределенная инфраструктура NGENIX CDN позволяет решать широкий круг сетевых и бизнес-задач — от базового ускорения загрузки сайта до реализации сложных инженерно-технических решений, учитывающих IT-архитектуру, бизнес-модель, размер и профиль целевой Интернет-аудитории заказчика.

5.7. Практическое использование CDN на примере Google Apps

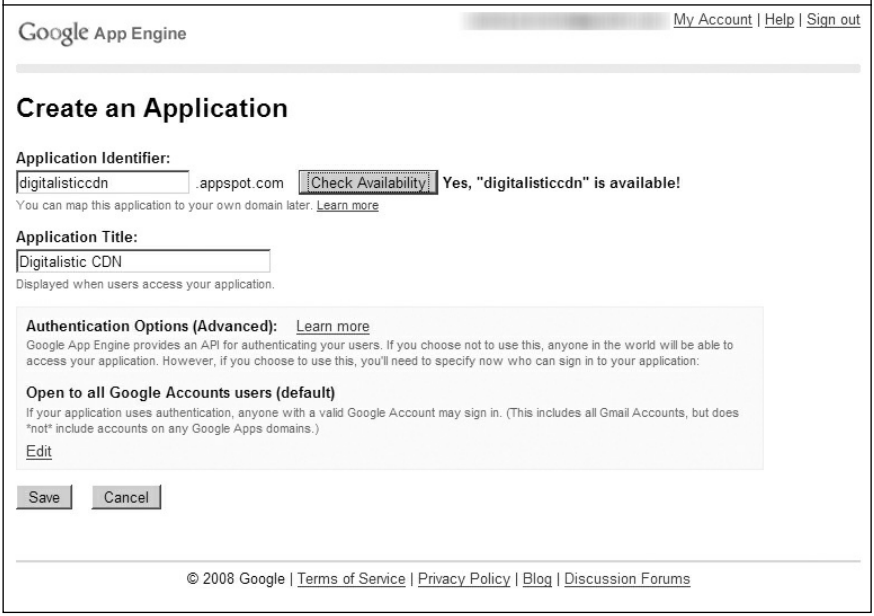
Итак, вы готовы использовать Google для хостинга своих файлов? Тогда вперед!

5.7.1. Все по порядку

1. Для загрузки файлов на CDN нам будет нужен Python. Ну, просто потому что на нем работает сам Google Apps. Для корректной работы рекомендуют версию 2.5 (на 3.0 Google SDK может не запуститься, на 2.5 работает исправно). Загружаем Python отсюда: http://www.python.org/download/, устанавливаем, запоминаем директорию установки (она нам пригодится в дальнейшем).
2. Загружаем последнюю версию Google Apps SDK (http://code.google.com/appengine/downloads.html). Устанавливаем ее (так как мы выполнили уже п. 1 и поставили Python, то проблем у нас не возникнет). Если в ходе установки выбираем нестандартную директорию, то опять-таки запоминаем к ней путь.
3. Регистрируемся на appengine.google.com (для этого понадобится аккаунт Google). Если в Google Apps Engine аккаунт уже был, то пропускаем этот пункт.
4. После регистрации заходим и создаем свое приложение. Нужно выбрать уникальный URL (поддомен `appspot.com`) и название: Дополнительно нужно будет подтвердить аккаунт через SMS, но ведь мы собираемся там просто CDN развернуть, а не спамить, правда?

Рис. 5.13. Загружаем Google App Engine SDK

Рис. 5.14. Создание нового приложения в Google App Engine



5. Теперь (или параллельно ожиданию подтверждения от Google) готовим рабочую директорию с файлами у себя на машине (ведь мы для этого устанавливали сначала Pyt hon, а потом SDK). Называем ее произвольным образом, в корне создаем файл `app.yaml`, в который записываем:

```
application: ваш_идентификатор_приложения
version: 1
runtimee: python
api_version: 1

handlers:
- url: /favicon.ico
  static_files: favicon.ico
  upload: favicon.ico

- url: /.*
  script: cacheheaders.py
```

   В случае нашего примера идентификатор был просто `webo`, он соответствует адресу `webo.appspot.com`, а `version` соответствует версии приложения. Очень удобно отлаживать новую версию, в то время как более старая замечательно работает. Переключение между версиями происходит из панели управления Google Apps (http://appengine.google.com/ deployment).
6. Сюда же, в директорию, закидываем файл `favicon.ico` от рабочего сайта и создаем еще один файл, `cacheheaders.py`: (сразу стоит отметить, что в качестве отбивки во всех Python-скриптах используется не табуляция, а двойной пробел):

```
import wsgiref.handlers
from google.appengine.ext import webapp

class MainPage(webapp.RequestHandler):

  def output_file(self, path, lastmod):
    import datetime
    try:
      self.response.headers['Last-Modified'] =
      lastmod.strftime("%a, %d %b %Y %H:%M:%S GMT")
      expires=lastmod+datetime.timedelta(days=365)
```

```
        self.response.headers['Expires'] = expires.strf-
        time("%a, %d %b %Y %H:%M:%S GMT")
        self.response.headers['Cache-Control'] = 'public,
        max-age=31536000'
        fh = open(path, 'r')
        self.response.out.write(fh.read())
        fh.close
        return
      except IOError:
        self.error(404)
        return

  def get(self, dir, file, extension):
if (dir != 'i' and extension != 'jpg' and extension !=
'png' and extension != 'gif'):
        self.error(404)
      return

    if extension == "jpg":
        self.response.headers['Content-Type'] =
        "image/jpeg"
    elif extension == "gif":
        self.response.headers['Content-Type'] =
        "image/gif"
    elif extension == "png":
        self.response.headers['Content-Type'] =
        "image/png"

    try:
      import os
      import datetime
      path = dir+"/"+file+"."+extension
      info = os.stat(path)
      lastmod = datetime.datetime.fromtimestamp(info[8])
      if self.request.headers.has_key('If-Modified-Since'):
          dt = self.request.headers.get('If-Modified-
          Since').split(';')[0]
        modsince = datetime.datetime.strptime(dt,
      "%a, %d %b %Y %H:%M:%S %Z")
        if modsince >= lastmod:
```

```
            # Файл более старый, чем закэшированная копия
              (или такой же)
                self.error(304)
              return
            else:
              # Файл более новый
              self.output_file(path, lastmod)
        else:
          self.output_file(path, lastmod)
      except:
        self.error(404)
          return

def main():
application =
webapp.WSGIApplication([(r'/(.*)/([^.]*).(.*)',
MainPage)], debug=False)
  wsgiref.handlers.CGIHandler().run(application)

if __name__ == "__main__":
  main()
```

По поводу этого файла — небольшое лирическое отступление. Как выяснилось в ходе исследования, Google Apps по умолчанию не поддерживает `Last-Modified / ETag` (только `Expires`, который настраивается простой строкой в `app.yaml` – `default_expiration: "365d"`). Чтобы обеспечить поддержку этого необходимого для CDN функционала (для 304-ответов), мы и заводим обработчик `cacheheaders.py`.

Конечно, можно обойтись простым кэшированием, но мы же хотим максимально правильный CDN? Сам файл `cacheheaders.py` просто проверяет, что запрос пришел к папке `i` для нашего приложения и расширение у файла `.png`, `.gif` или `.jpg`, после этого он отдает либо сам файл, либо 304-ответ (сравнивая заголовок браузера `If-Modified-Since` с меткой времени файла).

7. Теперь настроим скрипт для загрузки файлов из нашей директории на Google. Для этого нужно завести в нашей папочке (или еще где-нибудь, это уже не важно) файл `upload.bat` (если вы собираетесь загружать файлы из-под другой операционной системы, нежели Windows, то логику файла придется пере-

писать на соответствующем скриптовом языке). В файле записываем:

```
"путь_к_установленному_Python_из_пункта_1" "C:\Program
Files\Google\google_appengine\appcfg.py" update
"путь_к_рабочей_папочке_с_файлами"
```

Если в пункте 2 вы выбрали нестандартную директорию для Google App Engine SDK, то ее придется подставить вместо `C:\Program Files\Google\google_appengine`.

8. Создаем папку `i` в рабочей директории, в которую можно загрузить все файлы, которые предполагается отдавать с CDN. В имени файла должна отсутствовать точка (иначе `cacheheaders.py` будет некорректно обрабатывать расширение для файла — и его придется подправить).
9. Запускаем наш `upload.bat`, вводим логин/пароль от Google App Engine (только в первый раз) и радуемся процессу загрузки файлов на CDN.
10. И вот сейчас уже любой файл по адресу `ваш_идентификатор.appspot.com/i/` будет отдаваться через сеть серверов Google по всему миру (например, http://webo.appspot.com/i/b.png). Радуемся!

5.7.2. Подводим итоги

Если ваш проект не создает большой статической нагрузки (оценочно не более 250—500 Кб/с), то вы с легкостью можете воспользоваться серверами Google для выдачи своих файлов.

Отмеченные минусы:

- по умолчанию доступно только большое время кэша, настройка `Last-Modified` требует дополнительной логики и нагрузки на процессор (может стать критичной при большом количестве мелких файлов);
- Google CDN не позволяет изменять заголовок `Content-Encoding`. При настройке архивирования придется положиться на логику серверов Google;
- процесс обновления сайта может стать достаточно трудоемким, если его не автоматизировать (но автоматизируется он довольно просто). Также в бесплатной версии присутствует ограничение на число ежедневных обновлений файловой системы.

Во всем остальном — это идеальный выбор. Например, webo.in уже использует эту CDN для выдачи всех фоновых изображений (они обслуживаются с адреса `webo.appspot.com/i/`).

Также стоит отметить, что существует возможность полностью прикрепить домен к Google App Engine и применять для обслуживания его содержания какое-либо приложение App Engine. Это позволит (в случае полностью статического сайта) загружать его максимально быстро, совершенно бесплатно используя мощности Google (в разумных пределах для среднего сайта это порядка 20 тысяч посетителей в день).

Глава 6. Технологии будущего

В этой главе собрана часть материалов, затрагивающих передний край клиентских технологий и их производительность. В ней освещаются вопросы, связанные с профилированием JavaScript, проблемами в оценке производительности браузеров и скоростью работы CSS-селекторов в JavaScript-библиотеках.

Во второй половине главы затрагивается тема асинхронной (многопоточной) производительности на основе JavaScript и производительности AJAX при загрузке страницы.

6.1. Профилируем JavaScript

Данный раздел написан после прочтения ряда заметок Джона Ресига (автора JavaScript-библиотеки jQuery), в которых он рассказывал про особенности работы JavaScript в различных браузерах.

После существенной оптимизации CSS-селекторов и выхода Sizzle (http://sizzlejs.com/), который лишь немного уступает YASS (http://yass.webo.in/), автор jQuery сконцентрировал свои усилия на оптимизации работы с DOM-деревом и наиболее используемых методов и начал искать дополнительные способы для профилирования и оптимизации.

Также было написано дополнение для глубокого профилирования (http://ejohn.org/blog/deep-profiling-jquery-apps/) jQuery — оно помогло обнаружить методы, которые выполняются чересчур долго на реальных сайтах с jQuery. Дальше было проведено уточнение самих оптимизационных методов, которые, очевидно, являются не такими эффективными, как нам хотелось бы, — ведь непонятно, где именно и что конкретно нужно оптимизировать.

Для ответа на этот вопрос можно пойти по известному пути и замерить число вызовов функции из какого-либо метода. В этом нам может помочь Firebug, который позволяет увидеть эту информацию в соответствующей вкладке (еще и наряду со значениями времени выполнения каждого метода). К несчастью, очень неудобно вручную вбивать код, затем проверять результаты в консоли и определять, насколько они неудовлетворительны и изменились ли они с прошлого раза. Если бы только существовал способ получать эти числа в автоматическом режиме!

6.1.1. Профилирующие методы FireUnit

Джон Ресиг немного улучшил описанную ситуацию и добавил пару новых методов в FireUnit (http://fireunit.org/).

```
fireunit.getProfile();
```

Можно запустить этот код сразу после того, как вы использовали console.profile(); и console.profileEnd(); для перехвата проблемного участка кода, — и получить полный вывод профилирующей информации. Например, если мы запустим это, то получим из `fireunit.getProfile()` следующий объект JavaScript:

```
{
  "time": 8.443,
  "calls": 611,
```

```
  "data":[
  {
    "name":"makeArray()",
    "calls":1,
    "percent":23.58,
    "ownTime":1.991,
    "time":1.991,
    "avgTime":1.991,
    "minTime":1.991,
    "maxTime":1.991,
    "fileName":"jquery.js (line 2059)"
  },
  // etc.
]}
fireunit.profile( fn );
```

Рис. 6.1. Профилирование вызовов JavaScript, источник: ejohn.org

Profile (8.443ms, 611 calls)

Function	Calls	Percent	Own Time	Time	Avg
makeArray()	1	23.58%	1.991ms	1.991ms	1.99
add()	98	22.76%	1.922ms	4.97ms	0.05
add()	98	19.74%	1.667ms	3.048ms	0.03
has()	98	15.04%	1.27ms	1.381ms	0.01

Второй метод, добавленный в FireUnit, позволяет достаточно просто запускать и профилировать отдельные вызовы функций. Грубо говоря, этот метод запускает профайлер, исполняет функцию, останавливает профайлер, а затем возвращает результаты из getProfile(). Дополнительно он проверяет возможные исключения при вызове функции и гарантирует, что профайлер надежно выключен.

Это можно использовать примерно следующим образом:

```
fireunit.profile(function(){
  document.getElementsByClassName("foo");
});
```

6.1.2. Как это все применять

Во-первых, обязательно нужно установить последнюю версию (http://github.com/jeresig/fireunit) FireUnit. Также можно скачать последнюю версию кода в виде расширения к Firefox:

http://fireunit.org/fireunit-1.0a1.xpi

При запуске нужно будет убедиться, что:
1. обе вкладки `Console` и `Script` в `Firebug` включены;
2. свойство `extensions.firebug.throttleMessages` в `about:config` выставлено в false.

6.1.3. Результаты

Ниже приведены результаты вызовов из jQuery 1.3.2 («методом» называется метод jQuery, который запускается с определенными параметрами, «вызовы» — это число вызовов функции, которые происходят во время работы метода, «O(n)» является грубой оценкой сложности вызова функции):

Метод	Вызовы	O(n)
`.addClass("test");`	542	6n
`.addClass("test");`	592	6n
`.removeClass("test");`	754	8n
`.removeClass("test");`	610	6n
`.css("color", "red");`	495	5n
`.css({color: "red", border: "1px solid red"});`	887	9n
`.remove();`	23772	$2n+n^2$
`.append("<p>test</p>");`	307	3n
`.append("<p>test</p><p>test</p><p>test</p><p>test</p><p>test</p>");`	319	3n
`.show();`	394	4n
`.hide();`	394	4n
`.html("<p>test</p>");`	28759	$3n+n^2$

`.empty();`	28452	$3n+n^2$
`.is("div");`	110	
`.filter("div");`	216	2n
`.find("div");`	1564	16n

Как можно видеть из этой таблицы по значениям O(n), большинство методов jQuery вызывают по крайней мере по одной функции на каждый элемент, который им нужно обработать. `addClass` запускает около шести функций на каждый элемент, `filter` — примерно две, а `is` — только одну.

Также мы легко видим проблемные методы, напоминающие большие черные дыры, в которые утекает процессорное время — `.remove()`, `.empty()` и `.html()`. Все они имеют сложности с вызовом функций n^2, что является значительной проблемой для производительности ваших сайтов. Все эти числа выросли по очень простой причине: `.html()` использует `.empty()`, `.empty()` использует `.remove()`, а `.remove()` работает крайне неэффективно. Если не начать профилировать вызовы функций на медленное выполнение (к слову сказать, большинство внутренних методов jQuery выполняются очень быстро), то и не удастся обнаружить код, написанный неэффективно.

После внесения необходимых изменений мы придем к значительно улучшенным числам:

Метод	**Вызовы**	**O(n)**
`.remove();`	298	3n
`.html("<p>test</p>");`	507	5n
`.empty();`	200	2n

Автоматизированный процесс профилирования кода открывает широкие просторы для исследовательской деятельности. Даже не используя ничего другого, кроме вышеописанного метода, уже можно значительно улучшить каждый метод jQuery и любой другой клиентской библиотеки.

Также весьма интересно интегрировать описанное профилирование как часть самого процесса разработки — чтобы сразу замечать очевидные недочеты в производительности.

Позиция John относительно философии развития jQuery заслуживает отдельного слова. John изначально закладывается на совместимость со всеми браузерами, чтобы охватить максимальную аудиторию пользователей и разработчиков. И только после этого (когда стало понятно, что jQuery работает не так эффективно, как хотелось бы) он начал ее существенно оптимизировать. С точки зрения маркетинга — очень грамотный подход.

6.2. Проблемы при оценке производительности браузеров

Данный раздел написан на основе статьи Christian Stockwell, отвечающего за производительность браузера IE. Измерение общей производительности веб-сайтов и браузеров крайне важно для большого числа людей: как для пользователей, использующих различные продукты, так и для разработчиков, оптимизирующих свои порталы. Также сами разработчики других браузеров внимательно следят за успехами их коллег по цеху и стараются ориентироваться на некоторые отраслевые стандарты качества.

6.2.1. Измерение производительности браузера

Обычно для измерения производительности браузера используются специальные тесты-симуляции. И хотя они могут быть очень полезны во всех случаях, было бы ошибкой целиком полагаться на небольшое количество таких тестов, если мы хотим оценить производительность браузера в том смысле, в каком этот термин понимают обыкновенные пользователи. Наилучшие способы измерения производительности браузера должны непременно содержать сценарии, отражающие реальные ситуации, в которых он используется. Работа с реальными веб-сайтами позволяет учесть те факторы, которые в случае применения тестов, симулирующих какую-то ситуацию, учету не поддаются, и передать целостное впечатление о производительности. Однако тестирование браузеров на настоящих веб-ресурсах связано с рядом нюансов, и в этом разделе обсуждаются некоторые способы, которые применяются для адекватного измерения производительности IE.

Прежде чем углубиться в детали, стоит заметить, что измерение производительности — весьма нетривиальная задача, как ни странно это звучит. Команда программистов IE положила немало трудов, создавая лабораторию тестов и производительности, в которой сотни ПК и ноутбуков ежедневно прокручивали тысячи отдельных тестов, обращаясь к огромному количеству серверов во Всемирной Паутине, и редкий день завершался без того, чтобы родилось несколько новых идей, как добиться точности, ясности и достоверности оценочных данных.

Часть проблем оценки производительности вызвана огромным количеством разнообразных действий, для которых используется браузер. Каждый день пользователи обращаются к широкому диапазону ресурсов — от насыщенного мультимедийным содержимым Flickr до спартанского Google. Они могут столкнуться с интерактивным, насыщенным AJAX-скриптами сайтом, как Windows Live Hotmail, или сайтом, содержащим лишь статический HTML, как, например, Craigslist, а некоторые из них станут использовать браузер для критически важных деловых приложений (например, построенных на его основе систем электронного документооборота).

Производительность каждого из этих ресурсов часто зависит от производительности отдельной подсистемы браузера. Например, загрузка насыщенного изображениями сайта может зависеть от скорости, с какой браузер в состоянии загружать и распаковывать изображения. Напротив, производительность простенькой страницы зависит от того, как быстро браузер обрабатывает стандартный HTML. В следующем случае для хорошей производительности насыщенного AJAX-скриптами портала потребуется тесная интеграция JavaScript, CSS и DOM, — и это окажется в большей степени важным, нежели индивидуальная производительность каждого из названных компонентов. Когда на чашу весов кладутся Flash и Silverlight, производительность будет зависеть от того, насколько хорошо встроены в браузер соответствующие подсистемы управления.

Очевидно, что некоторые обсуждаемые тут подходы послужат лучшему представлению о той работе, которая была проделана для улучшения производительности IE 8, и позволят глубже заглянуть за кулисы процесса разработки браузеров. Изложенный ниже текст должен помочь по-новому подойти к процедурам оценки результатов измерения производительности и еще раз задуматься о том, что такое производительность браузера, а что такое производительность веб-сайта.

6.2.2. Работа службы кэширования

Все браузеры по сути своей работы зависят от характеристик сети, поэтому любые тесты должны отражать эту реальность — только тогда можно будет адекватно оценивать производительность.

Один из аспектов развития структуры Всемирной сети, который влияет на производительность браузера, — организация процесса сохранения часто запрашиваемого контента на узлах сети. Этот процесс называется кэшированием.

Что это означает в случае оценки производительности браузера? Например, при обращении к ресурсу http://www.microsoft.com ваш браузер может последовательно запрашивать данные из нескольких источников — с прокси-сервера вашей локальной сети, с сервера, расположенного к вам ближе всего, или с нескольких географически удаленных серверов.

Для повышения скорости загрузки содержимого страниц и распределения нагрузки по сети эти серверы могут сохранять часть загружаемых вами данных у себя в памяти, чтобы остальные пользователи могли быстрее получать к ним доступ. Например, утром, придя на работу, вы первым делом просматриваете новости на http://www.msnbc.com. Скорее всего, браузер попытается сначала загрузить запрашиваемую страницу с прокси-сервера, затем с ближайшего к вам сервера корпоративной сети — перед тем как обратиться к прочим, удаленным от вас ресурсам. Как только страница загрузится, ваш рабочий прокси-сервер или сервер в локальной сети может «решить» (разумеется, в зависимости от предварительно сделанных настроек) сохранить часть ее содержимого. Когда другой пользователь, спустя десять минут, попробует обратиться по тому же адресу, его компьютер сначала получит порцию данных, уже сохраненных на прокси-сервере, вместо их повторной загрузки с удаленных серверов, что, в свою очередь, значительно уменьшит время загрузки страницы и приведет пользователя в прекрасное расположение духа.

В похожем случае, измеряя производительность различных браузеров, важно учитывать влияние процедуры кэширования. Например, открыв десяток вкладок с десятком различных веб-страниц, и затем, открыв этот же десяток вкладок в другом браузере, можно ошибиться, решив, что второй браузер быстрее, в то время как разница в скорости может быть обусловлена кэшированием данных на сервере провайдера в тот момент, когда первый браузер сформировал запросы.

Довольно сложно тщательно проконтролировать, как серверы кэшируют данные, но одним из главных принципов при оценке производительности является условие никогда не измерять параметр один-единственный раз. Если вы не ставите задачи определить именно эффективность кэширования, то следует предварительно хотя бы раз загрузить страницы, производительность работы с которыми вы стремитесь оценить. Собственно, с тех пор, как прокси-серверы научились сохранять кэш для каждого из используемых браузеров, необходимо открывать отобранные для оценки производительности страницы в каждом из предназначенных для тестирования браузеров.

Принципы работы системы кэширования изложены здесь очень примитивно. Если требуется детальная информация по этому вопросу, то стоит обратиться к соответствующим ресурсам, включая, собственно, принципы работы HTTP-протокола.

6.2.3. Размер образца

Именно потому, что так много внешних факторов могут повлиять на оценку производительности, решающее значение имеет то, какие параметры и в каком количестве вы собираетесь учитывать.

Основной принцип оценки производительности — не измерять какой-либо параметр лишь однажды. Его стоит расширить до «всегда измеряйте нужный параметр достаточное количество раз». Существует множество способов определить это самое «достаточное количество раз» — например, используя доверительные интервалы, среднеквадратичные отклонения и другие милые статистические приемы.

Когда данные собраны, необходимо проанализировать их, чтобы сделать выводы. Используете ли вы варианты среднего арифметического, гармонического, геометрического или какие-то иные методики, необходимо быть последовательными и полностью представлять себе схему ветвления результатов при подведении итогов тестирования.

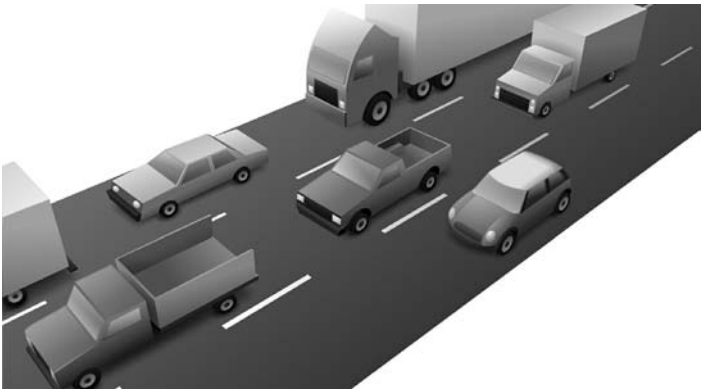
Например, давайте посмотрим на таблицу пунктов, набранных двумя браузерами по итогам тестов навигации в пределах одной веб-страницы:

Рис. 6.2. Проблема выбора подходящего среднего, источник: blogs.msdn.com

	Browser A	Browser B
Sample 1	1.0	2.0
Sample 2	1.0	2.0
Sample 3	1.0	2.0
Sample 4	1.0	2.0
Sample 5	10.0	4.0
Arithmetic Mean	2.8	2.4
Geometric Mean	1.6	2.3
Harmonic Mean	1.2	2.2

На этом искусственном примере хорошо видно, что в зависимости от того, как подводятся итоги тестов, выводы о производительности будут противоположными: при выборе в качестве критерия среднего арифметического браузер A быстрее браузера B, а при выборе критериев среднего геометрического и гармонического — все наоборот.

6.2.4. Совместное использование канала



Вы используете сеть не в одиночку, а наряду с другими пользователями, поэтому в какой-то момент ваш браузер вдруг без всяких видимых причин замедляет работу, и вы заметите, что на выполнение той же самой операции ему почему-то требуется гораздо больше времени.

Исходя из того, что существует столько сетевых конфигураций, сколько самих предприятий, довольно трудно предсказать влияние фактора нагрузки на сеть. Чтобы избежать искажения результатов тестирования, стоит собирать данные вне часов пик, если вам необходимо оценить производительность браузера в производственной среде.

При сборе данных в домашних условиях необходимо учесть, что и в этом случае вы делите канал с другими пользователями, например, членами семьи. В этом случае тесты лучше проводить тогда, когда в сети находится меньше пользователей — в рабочее время, поздно ночью или очень рано утром.

6.2.5. Совместное использование ресурсов

Совместное использование ресурсов приложениями на вашем компьютере также может повлиять на производительность браузера — по крайней мере так же сильно, как и совместное использование канала.

Это особенно заметно тогда, когда несколько программ зависят от внешних ресурсов или платформ. Например, некоторые антивирусы по-разному встраиваются в различные браузеры, разумеется, с не известной заранее степенью влияния на производительность.

Результаты тестирования двух браузеров одновременно, «бок о бок», могут оказаться совершенно некорректными. Например, платформа Windows имеет ограничение — возможны лишь 10 одновременных исходящих соединений; остальные запросы будут поставлены в очередь на выполнение по мере освобождения ресурсов и могут, в зависимости от необходимого временного интервала, завершиться успешно или с ошибкой. Такой способ тестирования означает, что вы, скорее всего, поставите один из браузеров в преимущественное положение тем, что запустите его на несколько микросекунд раньше соперника.

Можно рассмотреть всего два простых примера, хотя их существует множество. Не рекомендуется запускать другие приложения, когда вы тестируете производительность браузера. Вы должны предпринять следующие обязательные шаги для того, чтобы избежать влияния других программ:

- Закройте все прочие приложения, включая те, что скрыты в области уведомлений панели задач. Это особенно важно в случае, если какие-то из этих приложений используют сетевые ресурсы.
- В командной строке запустите следующую команду для ограничения активности компьютера в процессе тестирования:

```
%windir%\\system32\rundll32.exe advapi32.dll,ProcessIdleTasks
```

6.2.6. Взаимодействие с серверами

Кроме влияния совместного использования ресурсов и канала, на производительность может весьма значительно повлиять механизм работы серверов, к которым вы обращаетесь.

Одним из основополагающих принципов при проведении ваших тестов должно стать обеспечение равных условий на всех этапах и для всех аспектов тестирования. Для определения влияния процедур кэширования необходимо, чтобы серверы, к которым вы обращаетесь, накопили известное количество данных; для тестирования сети необходимо,

чтобы среда выполнения тестов была изолирована от влияния внешних ресурсов.

Примером конструктивных особенностей приложения, способных оказать влияние на процесс тестирования, может служить программа управления банковским счетом. По соображениям безопасности такая программа получает доступ к данным только после того, как прошла авторизация пользователя. Цель тестирования состоит в сравнении поведения двух или более браузеров на веб-странице банка, содержащей такую программу. Для этого необходимо, чтобы программа находилась как бы в равных условиях по отношению к тестируемым браузерам.

Обычно такого рода программы не разрешают пользователям входить в систему одновременно из двух или более сессий: при повторной авторизации предыдущая сессия завершается (пользователь выходит из системы). Если предыдущее состояние веб-приложения не будет сброшено перед началом тестирования другого браузера, приложению может потребоваться дополнительное время для обработки повторного запроса, закрытия предыдущей сессии и запуска новой.

Все эти процедуры значительно влияют на процесс тестирования, и характерны не только для онлайновых банковских приложений, так что вам следует попытаться исключить их как фактор. В общем случае вы должны очень хорошо представлять себе особенности поведения тех веб-страниц, с помощью которых тестируете браузеры.

6.2.7. Эффект наблюдателя

Во многих областях сам факт наблюдения изменяет характер поведения наблюдаемых объектов. Этот феномен получил наименование «эффекта наблюдателя».

Вы можете использовать любой набор библиотек для упрощения задачи тестирования некоторых сценариев использования браузера. Эти наборы обычно ориентированы на разработчиков и технически продвинутых пользователей. Как и в любом другом случае, когда результаты попадают в зависимость от способа измерения, необходимо тщательно оценить и по мере возможности ограничить колебания производительности, вызываемые типами библиотек, которые вы используете.

6.2.8. «Холодный» старт против «горячего»

Время, необходимое для запуска браузера, может зависеть от многих факторов, и некоторые из них совершенно не относятся к качеству самой программы.

Как и в случае с кэшированием, время запуска браузера зависит от внешних условий, особенно если вы запускаете браузер первый раз. Прежде чем можно будет приступить к серфингу, необходимо, чтобы соответствующие модули браузера загрузились в оперативную память — процесс, требующий времени. Когда вы впервые загружаете браузер, трудно определить, какое количество необходимого кода уже находится в памяти. Особенно трудно это в случае с IE, поскольку многие его компоненты совместно используются другими приложениями.

Чтобы собрать наиболее непротиворечивые данные, откройте и закройте каждый браузер как минимум один раз перед тем, как начнете тестирование. Если все остальные приложения закрыты, это даст вашей операционной системе возможность загрузить нужные компоненты в память и обеспечит последовательность и точность результатов тестирования. Это также создаст равные условия конкуренции для разных браузеров, особенно в свете существования таких функций операционной системы, как Superfetch (http://www.microsoft.com/windows/windows-vista/features/superfetch.aspx), которая в ином случае обеспечит преимущества «любимому» браузеру.

6.2.9. Содержимое веб-страниц

Веб-сайты постоянно изменяются. К сожалению, это происходит и тогда, когда вам необходимо тестировать производительность браузера.

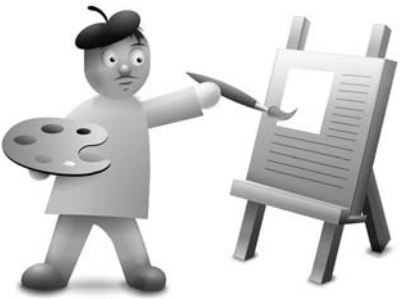
Можно на время тестирования закэшировать содержимое веб-страниц, чтобы гарантировать, что браузер всегда получает один и тот же контент для каждой итерации теста. В реальности такого, конечно же, не происходит.

Современные сайты обновляют содержимое очень часто. На Facebook или MySpace вы можете получить два совершенно разных результата между двумя щелчками кнопки мыши по одной и той же ссылке: за это время кто-то добавил запись или комментарий, загрузил картинку или еще каким-то образом изменил содержимое ресурса. На многих сайтах идет постоянная ротация рекламных баннеров, тем самым гарантируя, что любые два входа на сайт будут отличаться друг от друга.

Вне лабораторного пространства контролировать такой характер изменений практически невозможно. Разумеется, существуют определен-

ные подходы: например, вы можете использовать инструмент типа Fiddler для манипулирования контентом, который получает браузер. К сожалению, подобные методы с большой вероятностью могут привести к тому, что ожидаемого результата вы не достигнете. Решение состоит в том, чтобы следовать тем же рекомендациям, которые были даны выше относительно вопроса размера образцов для тестирования, и если вы обнаружите, что увесистый рекламный баннер появляется всякий раз после определенного количества загрузок страницы, то будет справедливо повторить измерения для обеспечения надежных результатов.

6.2.10. Дизайн страниц



Изменение на веб-странице зависит не только от пользователя, но и от веб-мастера, который создал радикально отличающиеся версии своего ресурса для разных браузеров.

Один из подходов, используемых в том случае, когда необходимо обеспечить доставку одного и того же содержимого для ваших тестов — делать сайты, которые подстраиваются под тип браузера. В большинстве случаев следует игнорировать то, что для разных браузеров используется различный программный код. Это дает вам возможность реально оценить впечатления пользователей при посещении таких ресурсов.

Однако в некоторых случаях, в зависимости от того, каким браузером вы пользуетесь, изменяются и функциональные возможности страницы, причем настолько, что прямое сравнение браузеров становится невозможным. Существуют ресурсы, которые на самом деле предлагают различную функциональность в зависимости от типа браузера. Оценка производительности на базе таких сайтов — непростая задача, но стоит избегать прямого сравнения браузеров при их посещении, поскольку речь в этом случае идет более о взглядах и предпочтениях дизайнеров, нежели о производительности браузеров как таковых.

Определить, какие сайты ведут себя подобным образом, не всегда просто, и в этом случае веб-дизайнеры имеют преимущество перед «обычными» тестировщиками. Они должны использовать профайлеры, отладчики и другие имеющиеся в их распоряжении инструменты для того, чтобы определить участки контента, на которых разные браузеры ведут себя существенно отличающимся образом.

Рядовым тестировщикам и пользователям без глубоких технических познаний не следует оценивать производительность браузеров на примерах сайтов, слишком по-разному представляемых в разных браузерах, поскольку им сложно провести грань между производительностью сайта и особенностями его дизайна.

6.2.11. «Готово»

Можете ли вы точно определить, что означает «веб-страница загружена»? Как быть в случае, если она содержит сложные AJAX-сценарии?

Проблема при оценке производительности заключается в определении того, что, собственно, означает надпись «готово» в статусной строке браузера при загрузке страницы. А также в том, что некоторые страницы усложняются и разрастаются несогласованно друг с другом. Некоторые веб-программисты применяют маркер «загружено» (http://www.w3.org/ TR/html401/interact/scripts.html#h-18.2.3) как индикатор того, что браузер завершил разметку содержимого страницы для последующей загрузки. Этот маркер, к сожалению, интерпретируется разными браузерами по-разному.

Кроме индикаторов, работающих на уровне программного кода, некоторые используют, например, прогресс-бар браузера, текстовые поля и прочие общепринятые элементы графического интерфейса. Как правило, поведение этих элементов никак не регламентировано, и веб-программисты могут по своему усмотрению менять его, определяя, когда (если вообще!) отображать их на экране.

В таких ситуациях тестировщикам и пользователям стоит применять те методы приближения, которые основываются на индикаторе загрузки, с тем чтобы наблюдать поведение этого индикатора в процессе загрузки страниц. Например, вы измеряете скорость первоначальной загрузки определенной страницы, и в то же время взаимодействуете с ее содержимым. Если страница все еще как будто бы загружается, индикатор показывает «в прогрессе», но при этом вы можете взаимодействовать с контентом. Вы можете решить не обращать внимания на индикатор и ориентироваться на визуальные параметры оценки, загрузилась страница до конца или нет. С другой стороны, индикатора загрузки может оказаться достаточно для предварительной

оценки скорости загрузки в различных браузерах. Однако если скорость, с какой страница загружается в действительности, не соответствует индикации в прогресс-баре, то будет довольно трудно понять, до какой степени в этом случае можно доверять результатам замеров производительности.

6.2.12. Надстройки браузера

Использование надстроек означает, что с этого момента вы измеряете производительность не только самого браузера. Надстройки могут существенно влиять на производительность. Согласно данным, полученным из источников в Microsoft, IE используется в совокупности с дюжинами надстроек (относительно Mozilla ситуация абсолютно идентичная).

Любая из этих надстроек может проявлять произвольную активность внутри браузера. Иллюстрацией воздействия может служить следующая ситуация: пользователи, отстаивающие свои предпочтения в отношении определенного браузера, внезапно обнаруживают, что любая альтернативная программа работает быстрее лишь потому, что их любимый браузер перегружен надстройками и дополнениями, а альтернативный представляет собой чистую, без всякого «мусора» программу. Например, пользователь обремененного несколькими дополнениями Firefox может сменить его на IE, увидев, что тот работает быстрее, а в это время пользователь IE переходит на Firefox по той же схеме исходя из тех же причин. Здесь нет никакого противоречия — такие примеры лишь демонстрируют решающее влияние надстроек.

Для блокировки надстроек в IE 8 необходимо вызвать пункт Manage add-ons из меню Tools. В появившемся диалоговом окне выберите All Add-ons и последовательно заблокируйте все надстройки из списка. Если вы дружите с командной строкой, можете выполнить команду `iexplore.exe -extoff` для запуска IE без дополнений.

Поскольку большинство программистов предпочитает не сокращать объем кода, чтобы тем самым гарантировать ожидаемое поведение надстроек после обновлений, важно, особенно во время использования пробных версий, иметь возможность отключить любое из некорректно ведущих себя дополнений.

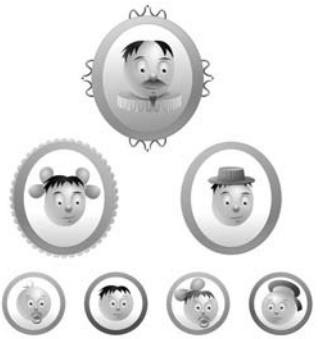
6.3. Перспективы «быстрого» JavaScript

В этом разделе рассматривается часть прикладных методов, положенных в основу разработки YASS (http://yass.webo.in/) — самой быстрой библиотеки для выбора элементов по CSS-селекторам.

6.3.1. Условное ветвление

Начнем с наиболее очевидной составляющей любой логики — ветвления. В любом алгоритме встречается место, в котором нужно выбрать то или иное продолжение в зависимости от проверяемого условия. Давайте рассмотрим следующие простые примеры проверки. В первом случае у нас три простых вложенных проверки:

```
var a = 1,
  b = 2,
  c = 3;
if(a == 1) {
  if (b == 2) {
    if (c == 3) {
      ...
    }
  }
}
```



Это, заметим с интересом, работает так же быстро, как и совмещенный if:

```
if(a == 1 && b == 2 && c == 3){
  ...
}
```

Однако последний вариант немного меньше по размеру. Если не стоит задача минимального размера кода, то для улучшения читаемости стоит использовать первый вариант. Если же мы минимизируем все, то можно рассмотреть возможность использования if-then-else выражения. Но нужно иметь в виду, что производительность таких конструкций:

```
var v = a == 1 ? b == 2 ? c == 3 ? 1 : 0 : 0 : 0;
```

примерно на 10-50% меньше, чем у обычного ветвления, рассмотренного чуть выше.

В том случае, когда все переменные у нас числовые, проверка равенства их суммы заданной будет выполняться на 5—10% быстрее:

```
if (a + b + c == 6) {
  ...
}
```

Если же нам нужно проверить просто существование переменных и их неотрицательность (т. е. то, что переменные не undefined, не NaN, не null, не '' и не 0), то следующий вариант будет работать еще на 5—10% быстрее, чем предыдущий случай (и на 10—20% быстрее, чем самый первый пример):

```
if (a && b && c) {
  ...
}
```

Очень часто нам нужно проверить что-то более сложное, чем просто число. Например, совпадение строки с заданной или равенство объектов. В этом случае нам просто необходимо следующее сравнение:

```
var a = 1,
  b = 2,
  c = '3';
if (a == 1 && b == 2 && c === '3') {
  ...
}
```

Здесь мы используем сравнение без приведения типов ===, которое в случае нечисловых переменных работает быстрее обычного сравнения на 10—20%.

6.3.2. Выбор в зависимости от строки

Достаточно часто нам нужно выбрать одну из условных ветвей, основываясь на заданной строке. Обычно для этого используются либо методы объекта `RegExp (exec, test)`, либо строковые методы (`match, search, indexOf`). Если нам нужно просто проверить соответствие строки какому-то регулярному выражению, то лучше всего для этого подойдет именно `test`:

```
var str = 'abc',
  regexp = new RegExp('abc');
if (regexp.test(str)) {
  ...
}
```

Такая конструкция отработает на 40% быстрее, чем аналогичный exec:

```
if (regexp.exec(str)[1]) {
  ...
}
```

Строковый метод match аналогичен методу exec у создаваемого объекта RegExp, но работает на 10—15% быстрее в случае простых выражений. Однако метод search работает чуть медленнее (5—10%), чем test, потому что последний не возвращает найденную подстроку.

В том случае, если регулярное выражение требуется «на один раз», подойдет более быстрая (примерно на 10% относительно варианта с инициализацией нового объекта) запись:

```
if (/abc/.test(str)) {
  ...
}
```

Если же, наконец, нам нужно проверить просто нахождение подстроки в заданной строке, то тут бесспорным лидером будет именно indexOf, который работает в два раза быстрее разбора регулярных выражений:

```
if (str.indexOf('abc') != 1) {
  ...
}
```

6.3.3. Точное соответствие и хэши

Давайте теперь рассмотрим следующий вариант регулярного выражения: `/a|b|c/`. В этом случае нам нужно проверить в заданной строке наличие одного из возможных вариантов (или равенство строки этому варианту). В случае точного соответствия быстрее регулярного выражения (на 50%) будет проверка строки как ключа какого-либо хэша:

```
var hash = {'a':1, 'b':1},
  str = 'a';
if (h[str]) {
  ...
}
```

Быстрее (на 20%) такого хэша будет только точная проверка строки на определенные значения:

```
if (ss === 'a' || ss === 'b'){
  ...
}
```

Если рассмотреть 3 конструкции: вложенный if, switch с соответствующим значением и проверка значений в хэше, — то стоит отметить следующую интересную особенность. При небольшом уровне вложенности if (если всего значений немного или мы очень часто выходим по первому-второму значению), конструкции if и switch обгоняют по производительности хэш примерно на 10%. Если же у нас значений много и они все примерно равновероятны, то хэш отрабатывает в общем случае быстрее уже на 20%. Это в равной степени относится как к установлению значений переменных, так и к вызову функций. Поэтому для создания ветвления с вызовом функций лучше всего использовать именно хэш.

При анализе CSS-селектора можно выделить несколько подзадач, описываемых как «проблема выбора».

- Ветвление для простого случая выполнено при помощи проверки входной строки через `test`:

```
if (/^[\w[:#.][\w\]*^|=!]*$/.test(selector)) {
  ...
} else {
  ...
}
```

- Ветвление для простейшего случая (когда нам нужно выбрать по идентификатору или по классу). Поскольку всего значений у нас 5 и 3 из них относительно равновероятны (выбор по классу, по идентификатору или по тегу), используется `switch`:

```
switch (firstLetter) {
  case '#':
```

```
      ...
      break;
    case '.':
      ...
      break;
    case ':'
      ...
      break;
    case '[':
      ...
      break;
    default:
      ...
      break;
}
```

- Абсолютно аналогичную картину мы наблюдаем для выбора правильного отношения «родитель-ребенок» (`>`, `+`, `~`,): тут тоже только `switch`:

```
switch (ancestor) {
  case ' ':
    ...
    break;
  case '~':
    ...
    break;
  case '+':
    ...
    break;
  case '>':
    ...
    break;
}
```

- Наконец, выбор соответствующей проверочной функции для child-модификаторов (`first-child`, `last-child`, `nth-child`, и т. д.) и выбор проверочной функции для атрибутов (`~=`, `*=`, `=` и т. д.) осуществляется уже через специальные хэши:

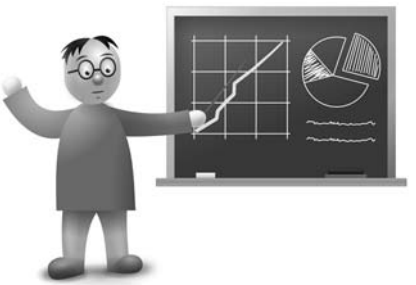
```
_.attr = {'': ... , '=': ... , '&=': ... , '^=': ... ...
}
```

6.3.4. Итоговая таблица

Подводя небольшой итог для различных способов проверки строки, можно составить такую таблицу:

Задача	Средство решения
Проверка числового значения	Обычное сравнение (==)
Проверка нескольких числовых значений	Сравнение их суммы
Проверка, что число не нуль, или проверка на существование	Проверка отрицания к заданной переменной (!)
Разбор строки и выделение частей в массив	`String.match(RegExp)` или `RegExp.exec(String)`
Проверка строки на соответствие регулярному выражению	`RegExp.test(String)`
Проверка строки на наличие подстроки	`String.indexOf(String)`
Проверка строки на точное соответствие (либо соответствие одному из набора значений)	if без приведения типов (===)
Выбор в зависимости от точного значения (значений 1—2)	Условная конструкция if
Выбор в зависимости от точного значения (значений 3—8)	`switch`
Выбор в зависимости от точного значения (значений больше 8)	Хэш с ключами, соответствующими значениям

Наверное, данную таблицу можно дополнить еще некоторыми случаями или же обратиться к статье, посвященной производительности простых конструкций в JavaScript (http://webo.in/articles/habrahabr/78-javascript-constructions-performance/) и сделать соответствующие выводы.



6.4. Реализация логики CSS3-селекторов

На данный момент насчитывается несколько десятков различных JavaScript-библиотек, которые предлагают механизм выбора элементов по CSS-селектору. В чем же разница между ними всеми? Разница в скорости нахождения элементов. Иногда она может варьироваться довольно сильно.

Но в последнее время появилось несколько явных фаворитов на этом поприще. Речь идет про Sizzle (движок выборки элементов, автором которого является Джон Ресиг и который включен в jQuery 1.3+), Peppy (достаточно хорошо стартовавший и обогнавший на первых порах Sizzle, но потом заброшенный автором) и некоторые другие, в том числе и YASS (http://yass.webo.in/).

Поэтому возникает резонный вопрос, почему нельзя сделать быстрое мини-ядро для CSS-селекторов, которое обеспечит базовую функциональность для работы с DOM (например, совсем базовую — просто выборку элементов)? И, самое главное, чтобы это работало не медленнее (в идеале даже быстрее), чем вызовы в самом браузере.

6.4.1. Основы быстродействия

Но описанная задача имеет решение. Можно создать библиотеку, которая будет осуществлять базовые операции практически так же быстро, как и сам браузер (а в некоторых случаях даже быстрее — за счет кэширования). И это удалось сделать в достаточно сжатые сроки. Далее речь пойдет о самой быстрой (на момент написания книги) библиотеки для выбора элементов по CSS-селекторам. Каковы же были причины для написания такой библиотеки?

Во-первых, такой код должен и кэшировать выборки (DOM-вызовы дорого обходятся, все нормальные JavaScript-программисты уже их кэшируют — так упростим им задачу и повысим быстродействие).

Во-вторых, есть статистика использования CSS-селекторов в проектах — значит, у нас есть набор элементарных операций, которые должны выполняться быстрее всего (может быть, даже в ущерб более общей производительности).

В-третьих, библиотека должна возвращать сами элементы, чтобы к ним можно было «прикрутить» любые обертки и нарастить методы. Все обертки ресурсоемки. Если нужно просто 200 раз поменять HTML на странице у заданных элементов, то они не нужны. Достаточно и проверок со стороны браузера на допустимость выполняемых операций.

В-четвертых, естественно, библиотека должна опираться на все самое быстрое: быстрые итераторы, быстрые подстановки, анонимные функции, минимум вложенных вызовов и т. д.

6.4.2. Примеры вызовов

Синтаксис такой библиотеки до безобразия прост:

- `_('p')` — вернет все параграфы на странице;
- `_('p a')` — или все ссылки в них;
- `_('p a.blog')` — или все ссылки с классом `blog`.

6.4.3. Еще один велосипед?

Может быть, кому-то данная разработка покажется тривиальной, но дальнейшее развитие ситуации видится следующим образом. Код YASS можно использовать в образовательных целях, чтобы понять логику конечных автоматов, используемых браузером для распознавания CSS-селекторов (во встроенном движке), и промоделировать некоторые тривиальные или не очень ситуации.

Или же данный код можно будет доработать и включать в основу высокопроизводительных библиотек, которые уже будут при его помощи реализовывать свои методы (одной из таких библиотек, с которыми YASS уже интегрирован, является js-core, http://code.google.com/p/js-core/). Также возможна замена кодом YASS встроенного механизма выборки элементов по CSS-селектору в таких распространенных библиотеках, как MooTools, Prototype, jQuery, YUI для повышения их быстродействия.

Но давайте рассмотрим процесс выборки CSS-селекторов более детально.

6.4.5. Выборка CSS-селекторов

Начнем с самого простого: чего мы хотим добиться? Мы хотим, задав произвольную строку CSS-селектора, соответствующую спецификации (http://www.w3.org/TR/2005/WD-css3-selectors-20051215/), получить на выходе массив из всех элементов, соответствующих этой самой строке. Вроде пока все просто.

В качестве иллюстрации спецификации можно привести следующие примеры (работает во всех современных браузерах и IE8+):

```
// вернет элемент с идентификатором my_id
querySelectorAll('#my_id')
// вернет все элементы с классом external
querySelectorAll('.external')
// вернет все абзацы на странице
querySelectorAll('p')
```

Однако уже тут можно отметить один момент: очень часто нам нужно выбрать просто элемент по его идентификатору или найти все элементы с определенным классом. Эти операции встречаются достаточно часто во всех JavaScript-библиотеках, поэтому они должны выполняться максимально быстро. Запускать весь механизм анализа входной строки селектора просто в том случае, когда нам нужно вернуть один-единственный элемент, заданный с помощью идентификатора, крайне неосмотрительно. Здесь мы можем воспользоваться принципом ленивого программирования: «не делай того, чего можно не делать», — и достаточно сильно ускорить работу для простейших случаев.

Если посмотреть на современные JavaScript-библиотеки, то везде такая проверка уже осуществляется с помощью регулярного выражения. И тут сразу же небольшая хитрость: инициализация любого регулярного выражения обходится достаточно дорого в браузерах (хотя сложность выражения начинает влиять на затраченное время только при большой его длине), и разумно было бы вообще без него обойтись. И когда можно обойтись indexOf для проверки подстроки — этим (в элементарных случаях) всегда нужно пользоваться.

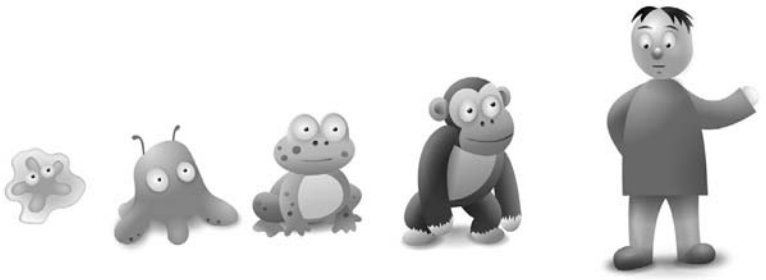
Может так случиться, что регулярное выражение обеспечивает корректность, и замена его никак не приведет к потере читаемости кода и непонятному выигрышу (или даже проигрышу) в скорости выполнения. Тогда можно заменить, например, exec на связку test и charAt / substr: это позволит увеличить производительность примерно на 20%. Если данный участок кода выполняется в цикле многократно, ускорение может оказаться достаточно существенным.

В YASS данная задача решена следующим образом:

```
// проверяем, удовлетворяет ли селектор простому случаю
if (/^[\w[:#.][\w\]*^|=!]*$/.test(selector)) {
// в случае положительного результата инициализируем переменную,
// которая отвечает за '#', '.' , ':' или '[' в начале селектора
  var firstLetter = selector.charAt(0);
  ...
}
```

6.4.6. От простого к сложному

Но давайте рассмотрим, как решена общая задача по разбору CSS-селекторов. Если принять во внимание, что селектор может быть задан в виде `p a.link, form input[type=radio]`, то логику его разбора можно схематично записать в следующем виде:?



- Выбираем последовательности селекторов, которые находятся между запятыми. Далее работаем с каждой последовательностью в отдельности. На выходе все последовательности объединяем в итоговый массив (`sets`).
- В последовательности селекторов у нас есть набор элементарных селекторов, которые «вложены» друг в друга (для нашего примера это `p a.link`). Нам нужно разбить последовательность на части и разобрать каждую такую часть, учитывая, что родительскими элементами для следующей части будут выбранные элементы из предыдущей. За «превращение» дочерних узлов в родительские (прямо процесс взросления получается) отвечает массив `nodes`.
- Каждый элементарный элемент уже придется разбирать с помощью регулярного выражения, чтобы вычленить части, отвечающие за идентификатор, класс и модификаторы CSS 2/3. Разбирать быстрее всего при помощи exec, а потом записывать в переменные части полученного массива:

```
single = regexp.exec(single);
tag = single[1];
id = single[2];
...
```

- И наконец, третий цикл проходится по всем родительским элементам и пытается выбрать из них дочерние узлы, соответствующие заданным в CSS-селекторе параметрам.

Как мы видим, основная логика данной задачи включает как минимум одно регулярное выражение (использование `indexOf` и `substring` будет при такой сложности намного более ресурсоемко) и 3 цикла (которые нужно сделать максимально быстрыми). Не стоит перечислять все возможности быстрого выбора элементов, просто сделаем акцент на некоторых аспектах.

6.4.7. Перебор массива

Пусть у нас объявлен некоторый массив a, с элементами которого мы совершаем какие-либо действия. Нам нужно перебрать все элементы строго по возрастанию (порядок важен), т. е. просто while(i–) мы использовать не можем. Наиболее распространенным сейчас способом будет обычный for:

```
for (var j=0, item = a[j]; item; item = a[j++]) {
  item++;
}
Естественно, он на 30–40% медленнее следующего while:
var j = 0,
  item,
  len = a.length;
while (j < len) {
  item = a[j++];
  item++;
}
```

Однако если нам нужно выполнить какие-либо действия с элементом массива, то без кэширования его в локальную переменную никак не обойтись. В этом случае следующий вариант с while (через проверку существования элементов при инкременте) будет еще быстрее — на 5—10%:

```
var j = 0,
  item;
while (item = a[j++]) {
  item++;
}
```

Очевидно, что для всех трех циклов в YASS (http://yass.webo.in/) применяется именно он.

Если же нам абсолютно не важен порядок элементов (например, просто нужно найти нужный или вернуть false), то логично будет воспользоваться обратным while:

```
while (idx--) {
  sets[idx].yeasss = null;
}
```

Именно такой код используется для сброса у элемента флага, отмечающего состояние «выбран». Давайте рассмотрим, зачем этот флаг нужен и зачем его сбрасывать.

6.4.8. Уникальность элементов

Одной из основных проблем в ходе выбора элементов по CSS-селектору является корректность конечного набора. В случае `div p` у нас могут оказаться вложенные `div`, и если мы просто переберем все родительские элементы и объединим получившиеся дочерние, то конечный набор будет содержать дубли. Чтобы избежать такого рода ошибок, нам нужно как-то отмечать элементы, которые мы выбрали.

Это стандартная задача, и решается она, в принципе, тоже стандартно: во всех библиотеках заводится определенное свойство DOM-узлов, которое отвечает за состояние «выбран». У этого подхода есть несколько минусов (например, нужно расширить это решение на случай асинхронных выборок элементов и понимать, что каждое обращение к элементам дерева ресурсоемко), но в подавляющем большинстве случаев он позволяет устранить неуникальность элементов.

Схематично представить работу данного флага можно следующим образом:

```
for (child in children) {
  if (!children[child].yeasss) {
    if (last) {
      children[child].yeasss = 1;
    }
    newNodes = children[child];
  }
}
```

В процессе перебора нужных нам дочерних узлов мы проверяем, выставлен ли у этого потомка флаг yeasss (естественно, что кроме этой проверки мы должны удостовериться, что данный потомок нам действительно нужен). Далее мы выставляем флаг только в том случае, если работаем с последним звеном в цепочке «детей-родителей», и записываем дочерний узел в массив новых узлов, которые станут «родителями» на следующей итерации цикла.

Прекрасно видно, что флаг yeasss будет выставлен только в том случае, если мы работаем с последним звеном: для промежуточных звеньев он никогда не будет выставлен, поэтому на родительские элементы мы никакие ограничения не распространяем. Вполне возможно, что можно было бы сокращать массив родителей на каждой итерации (чтобы не выбирать заведомо один и те же элементы), однако данный шаг не увеличит (в общем случае — только уменьшит) производительность выборок, поэтому здесь такой подход и не используется.

Сброс флага (о котором шла речь в предыдущем разделе) осуществляется уже для окончательного массива элементов (после завершения всех циклов) и необходим как для корректности дальнейших выборок, так и для предотвращения утечек памяти в IE (расширенные свойства элементов вызывают микро-утечки в IE 6/7).

6.4.9. Подводя черту

YASS (http://yass.webo.in/) создавалась и продолжает разрабатываться для поиска и реализации наиболее производительных методов для решения определенного круга задач. Ее можно применять как в учебных целях, так и в чисто практических (например, для составления набора неиспользуемых на сайте CSS-селекторов — с помощью YASS это реализуется быстрее всего).

6.5. API для CSS-селекторов в браузерах

Данный раздел написан под впечатлением от статьи DOM selectors API in Firefox 3.5 (http://hacks.mozilla.org/2009/06/dom-selectors-api/) Джон Ресиг (автора jQuery и евангелиста веб-стандартов в Mozilla), в которой освещается текущая поддержка браузерами стандартов в этом направлении и некоторые вопросы производительности.

Предварительная версия документа API для селекторов (http://dev.w3.org/2006/webapi/selectors-api/), опубликованная консорциумом W3C, представляет собой относительно новый взгляд для JavaScript-разработчиков на то, как можно выбирать DOM-элементы на страницы при помощи CSS-селекторов. В одном этом документе собраны все тонкости такого сложного процесса, как поиск, выборка элементов из DOM-дерева и представление результата, доступного по упорядоченному интерфейсу.

Несмотря на все недавние войны по поводу интеграции стандартов в браузеры, этот является одним из наиболее поддерживаемых: его можно

использовать прямо сегодня в браузерах Internet Explorer 8, Chrome и Safari, а также в Firefox 3.5 и Opera 10.

6.5.1. Используем querySelectorAll

API для селекторов предоставляет два метода для всех DOM-документов, элементов и фрагментов (удивительно удобно, хотя два — это, может быть, немного перебор, но иначе получение уникальных элементов на странице каждый раз выливалось бы в дополнительный JavaScript-код): `querySelector` и `querySelectorAll`. Оба метода практически идентичны: оба принимают CSS-селектор и возвращают DOM-элементы (за исключением того, что `querySelector` возвращает только первый найденный элемент).

Например, давайте рассмотрим следующий участок HTML-кода:

```
<div id="id" class="class">
  <p>Первый абзац.</p>
  <p>Второй абзац.</p>
</div>
```

Мы можем использовать querySelectorAll, чтобы сделать красным фон всех параграфов внутри div с идентификатором id.

```
var p = document.querySelectorAll("#id p");
for ( var i = 0; i < p.length; i++ ) {
  p[i].style.backgroundColor = "red";
}
```

А также мы можем найти самый первый параграф этого div, который является его прямым потомком и у которого задан класс class. Ему мы присвоим класс first.

```
document.querySelector("div.class > p:first-child")
  .className = "first";
```

В повседневной жизни описанные процедуры могут быть весьма запутанными в связи с большим объемом JavaScript-/DOM-кода, приводя к многострочным записям и множеству выборок для достижения какой-либо цели. Сразу стоит отметить, что хотя производительность CSS-селекторов уже ин÷рирована в браузеры, но ее быстродействие (особенно для ряда сложных случаев CSS3-спецификации) может быть весьма низкой.

Для преодоления этой проблемы необходимо использовать кэширующие техники, которые реализованы, например, в YASS (http://yass.webo.in/).

Хотя внешне применение методов API для селекторов весьма просто (каждый принимает только один аргумент на вход), проблемы наступают при выборе подходящей спецификации CSS-селекторов. API для селекторов привязано (и это на самом деле очень хорошо: представьте ситуацию, что браузер в CSS-коде понимал бы один набор селекторов, а при использовании JavaScript предоставлял бы уже совершенно другой доступный набор) к естественному движку CSS-селекторов в браузере, который нужен для применения стилей для конкретных элементов. Для большинства браузеров (Firefox, Safari, Chrome и Opera) это означает, что у вас есть доступ к полной гамме CSS3-селекторов. В то же время Internet Explorer 8 обеспечивает более ограниченный функционал и поддерживает только CSS2-селекторы (работать с которыми до сих можно только с трудом в силу отсутствия их полноценной поддержки в IE 6/7).

Итак, самой большой проблемой для новых пользователей API для селекторов является выбор корректной CSS-спецификации для использования. Особенно это актуально для большинства разработчиков, который пишут кроссбраузерный код и поэтому ориентируются на CSS1-набор, работающий во всех браузерах.

Изучение спецификаций CSS2- (http://www.w3.org/TR/CSS2/selector.html) и CSS3-селекторов (http://www.w3.org/TR/css3-selectors/) будет отличным шагом в увеличении своего багажа знаний.

6.5.2. Суровая реальность

Наиболее часто встречающийся случай применения API для CSS-селекторов — это использование его не напрямую, а при помощи разнообразных сторонних библиотек, которые также обеспечивают функциональность CSS-селекторов для DOM. Сегодня основная проблема внедрения применения API для селекторов заключается в том, что они не поддерживаются во всех браузерах, для которых ведется разработка (в частности, это IE 6, IE 7 и Firefox 3). Поэтому пока эти браузеры еще не вышли из обращения, нам будут требоваться некоторые промежуточные утилиты для восстановления недостающей функциональности CSS-селекторов для DOM.

Однако, к счастью, на данный момент таких библиотек — огромное число, и все они поддерживают интерфейс выбора элементов, совместимый с API для селекторов API (на самом деле последнее возникло как раз из рассмотрения текущей ситуации с выбором элементов и предложением интеграции в браузеры некоторой часто требуемой функциональности). В дополнение к этому существует некоторое количество фреймворков, кото-

рые уже переключаются на API для селекторов при наличии его в браузере (поэтому вы можете совершенно спокойно использовать их и не думать о применении каких-либо более эффективных инструментов для ускорения клиентской части вашего сайта). Это означает, что вы можете работать с CSS-селекторами прямо сегодня и получить все возможные преимущества от их повышенного быстродействия в некоторых браузерах за счет API для селекторов, и это обойдется вам совершенно бесплатно.

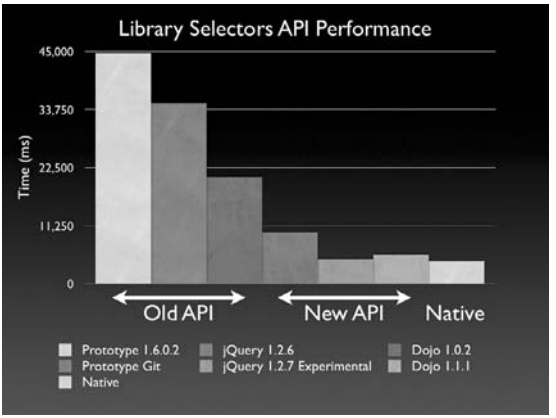
Некоторые из существующих фреймворков, использующих по возможности API для селекторов:

- jQuery (http://jquery.com/)
- Prototype (http://prototypejs.org/)
- Dojo (http://dojotoolkit.org/)
- MooTools (http://mootools.net/)

Стоит также подчеркнуть, что применение нового API влечет значительный выигрыш в производительности (по сравнению с обычными методами выбора элементов из DOM при помощи JavaScript). Вы сможете самостоятельно убедиться в этом, просто судя по улучшению ситуации в JavaScript-библиотеках, которые начали внедрять новое API для селекторов.

Согласно уже проведенным тестам результаты получаются примерно следующими:

Рис. 6.3. Прирост в производительности после внедрения API для селекторов, источник: hacks.mozilla.org



Невооруженным взглядом виден прирост в производительности после внедрения использования нового API для селекторов — то же самое

произойдет с вашими веб-приложениями, применяющими указанные фреймворки, в современных браузерах.

6.5.3. Тестовый набор

Чтобы сравнить определение спецификации API для селекторов (http://dev.w3.org/2006/webapi/selectors-api/) с фактической реализацией, было создано специальное тестовое окружение (автор — Джон Ресиг из Mozilla). Это тестовое окружение может быть в том числе использовано для проверки основных браузеров на уровень соответствия стандартам.

Текущие результаты для браузеров, которые поддерживают это API, следующие:

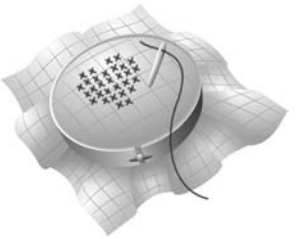
- Firefox 3.5: 99,3%
- Safari 4: 99,3%
- Chrome 2: 99,3%
- Opera 10b1: 97,5%
- Internet Explorer 8: 47,4%

Internet Explorer 8, как уже было упомянуто ранее, не реализует логику CSS3-селекторов (наверное, в силу того, что спецификация еще не утверждена w3.org), поэтому проваливает большую часть тестов.

По всей видимости, API для селекторов должно обеспечить простой и быстрый путь для выборки DOM-элементов на странице. Это действительно здорово, что все JavaScript-библиотеки используют тот же самый синтаксис и обеспечивают ту же функциональность. Стоит постараться разобраться в этом сейчас и начать применять этот API.

6.6. Canvas: один шаг назад, два шага вперед

Прежде чем начать разговор о текущем положении вещей с поддержкой векторной и скалярной графики в браузерах, стоит немного углубиться в историю.



6.6.1. Предыстория

В 1998 году компания Microsoft при поддержке других крупных производителей предложила W3C свой стандарт отображения векторной информации — VML (*англ. Vector Markup Language — векторный язык раз-*

метки). Он был основан на текущем формате представления документов в вебе — HTML — и расширял его до некоторого «векторного» языка. Компания Microsoft активно продвигала и продвигает данный формат и включает его во все версии IE начиная с 5.0. В рамках стандарта он окончательно закреплен в качестве части спецификации Open Office XML (ISO 29500:2008 и ECMA-376) в 2008 году.

В этом же 1998 году Adobe, IBM, Netscape и Sun вносят в W3C предложение о рассмотрении своего стандарта в противовес Microsoft — PGML (*англ. Precision Graphics Markup Language — точный графический язык разметки*). Не желая делать ни один стандарт проприетарным, W3C создает рабочую группу, которая на основе имеющихся предложений в 1999 году создает набросок еще одного стандарта — SVG (*англ. Scalable Vector Graphics — масштабируемая векторная графика*). Этот стандарт (хотя до сих пор закрепленный только в форме рекомендации, последняя версия 1.1 от 2003 года) находит гораздо большую поддержку у производителей браузеров и на данный момент включен практически везде (кроме, естественно, IE).

Сейчас большинство JavaScript-библиотек, которые предлагают работу с векторной графикой, включают поддержку SVG для всех браузеров и поддержку VML для IE. В качестве характерного примера можно привести Яндекс.Карты или Google Maps. Оба формата (SVG и VML) обладают практически одинаковыми возможностями; например, ниже приведен код на VML для отображения синего овала:

```
<html xmlns:v="VML">
  <style type="text/css">
    v\:*{behavior:url(#default#VML);position:absolute}
  </style>
<body>
  <v:oval style="left:0;top:0;width:100;height:50"
    fillcolor="blue" stroked="f"/>
</body>
</html>
```

Этот же код на SVG:

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="100" height="50">
  <ellipse cx="50" cy="25" rx="50" ry="25" fill="blue"
stroke="none" />
</svg>
```

6.6.2. Появление Canvas

Спецификация Canvas (как отдельной области на странице, внутри которой можно отображать графические объекты) в 2005 году изначально была предложена со стороны Apple для поддержки некоторых приложений внутри движка WebKit (на данный момент его используют браузеры Safari и Chrome). Рабочая группа W3C включила Canvas в Web Applications 1.0, который вошел в готовящийся стандарт HTML 5.0.

Сейчас встроенная поддержка Canvas реализована в том или ином виде во всех современных браузерах. В IE версии 8 и ниже она эмулируется при помощи отдельного VML-документа.

В качестве основного отличия от VML и SVG стоит назвать принципиальную невозможность описать какой-либо элемент в Canvas, используя строго HTML и CSS: все действия производятся только при помощи JavaScript. Также стоит упомянуть, что SVG оперирует с отдельными векторными объектами и их взаимосвязями, тогда как Canvas предлагает интерфейс к пиксельной области на странице, не предлагая устанавливать дополнительных связей (все они находятся в зоне ответственности JavaScript).

6.6.3. Основные возможности

Canvas предлагает достаточно мощное API для действий над двумерными графическими объектами. В том числе, это вывод и преобразование текста (в том числе загрузка произвольного шрифта), отображение двумерных объектов (прямоугольников, треугольников, кругов и многоугольников), работа с векторной графикой (задается как объект пути), преобразование над фигурами (прозрачность, перемещение, поворот и даже матрица преобразования), работа с отдельными линиями и тенями. Также возможна работа с изображениями в попиксельном формате (что позволяет сделать практически полноценный Photoshop внутри отдельного браузера) и коррекция цвета.

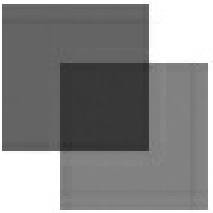
Давайте рассмотрим следующий простой пример использования Canvas:

```
<html>
  <head>
    <script type="application/x-javascript">
function draw() {
   var canvas = document.getElementById("canvas");
   var ctx = canvas.getContext("2d");
```

```
    ctx.fillStyle = "rgb(200,0,0)";
    ctx.fillRect (10, 10, 55, 50);

    ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
    ctx.fillRect (30, 30, 55, 50);
   }
   window.onload = draw;
     </script>
   </head>
   <body>
      <canvas id="canvas" width="300" height="300"></canvas>
   </body>
</html>
```

В результате его исполнения мы получим следующее изображение:

Рис. 6.4. Пример простого изображения при помощи Canvas, источник: developer.mozilla.org



Дополнительно Canvas позволяет определить практически любые действия пользователя над описываемыми объектами (перемещение и нажатия мыши) и загрузить в область объекты MathML и SVG. Как мы видим, на данный момент это полноценная платформа для произвольной анимации прямо в том же браузере, который предназначен для просмотра отдельных HTML-страниц. Можно с уверенностью предсказать, что через пару лет обычные Flash-банеры будут вытеснены их более интерактивными и более «поддерживаемыми» коллегами на основе Canvas (и, скажем, VML для семейства браузеров IE).

Подробнее со спецификацией Canvas можно ознакомиться, например, на странице WHATWG (http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html).

6.6.4. Примеры использования Canvas, SVG и VML

Примеров использования Canvas довольно много, и они частично охватывают уже известные области 2D-графики.

2D-проекции 3D-объектов

С помощью Canvas можно изображать 2D-проекции трехмерных объектов. В силу того, что все преобразования выполняются на пиксельном уровне, достаточно просто создать необходимые интерфейсы для изометрических проекций трехмерных объектов. Видимо, уже скоро на основе Canvas появятся полноценные 3D-экскурсии в браузерах или онлайн-игры «от первого лица».

Рис. 6.5. Изображение чайника при помощи Canvas, источник: www.nihilogic.dk

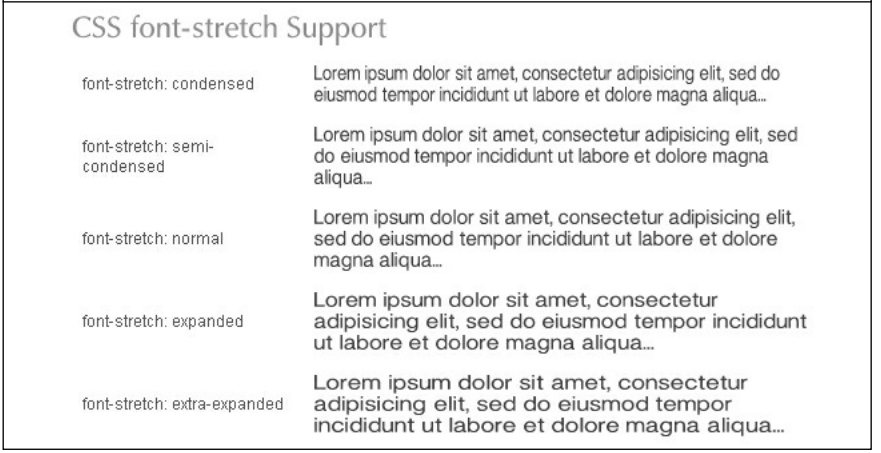
Typeface.js

В качестве следующего применения стоит упомянуть библиотеку typeface.js (http://typeface.neocracy.org/) и созданный на ее основе метод отображения произвольных шрифтов на сайте (ранее для этой цели активно использовался Flash). Единственным минусом данного подхода (на фоне повышенного быстродействия в связи с естественной поддержкой Canvas и простоты использования) является большой размер файла самих шрифтов.

Для применения библиотеки typeface.js необходимо перевести требуемый шрифт в некоторый объект, описываемый с помощью соответствующих конструкций на JavaScript, затем этот объект будет использоваться для представления произвольного текста на сайте (если быть точным, то в качестве объекта выступают отдельные символы шрифта). Размер файла шрифта в текстовом формате сильно варьируется от количества включенных в него символов и может составлять от 50 до 500 Кб (в неархивированном виде).

Для больших порталов (где размер страницы составляет 500-1000 Кб) данный подход вполне приемлем (если «стилизованных» заголовков будет не так много, чтобы лишний раз не нагружать браузер преобразованиями страницы). Для небольших сайтов загрузка нескольких десятков Кб JavaScript-кода для стилизации 10 Кб HTML- и CSS-кода выглядит не очень уместной.

Рис. 6.6. Применение Canvas для стилизации шрифтов,
источник typeface.neocracy.org



Cufón

Cufón в качестве основы использует для отображения произвольных шрифтов уже SVG. Однако проблемы с конвертацией файлов шрифтов в промежуточный формат присутствуют и здесь.

Более подробно ознакомиться и загрузить необходимые файлы можно по адресу http://cufon.shoqolate.com/generate/.

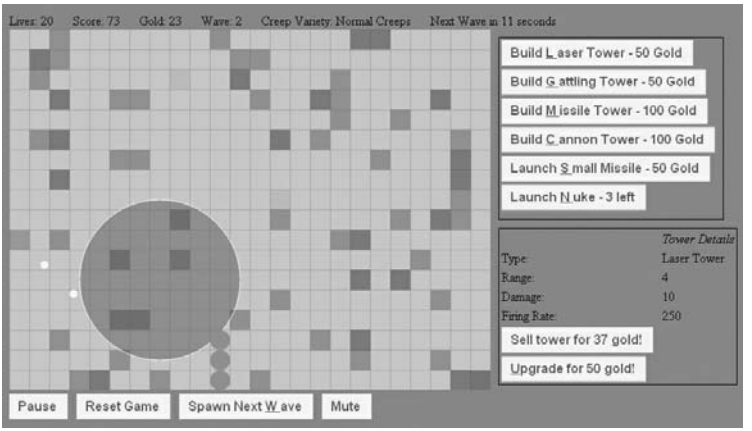
Prosessing.js

Processing — язык программирования, созданный Casey Reas и Benjamin Fry в академических целях и направленный на кроссплатформенную обработку двумерных графических объектов. Он может быть реализован на любой аппаратной платформе путем преобразования исходных конструкций в платформозависимые инструкции.

Хорошим примером использования Canvas является реализация Processing для JavaScript (автор реализации — Джон Ресиг) — библиотека Processing.js (processingjs.org/). Она предполагает полную совмес-

тимость инструкций данного языка с преобразованиями объекта Canvas. На данный момент доступно несколько проектов, применяющих Processing.js, в частности, несколько игр в браузерах, например, «Защита башнями».

Рис. 6.7. Использование processing.js для игры «Защита башнями», источник: willarson.com



Raphael

Если предыдущий пример был посвящен использованию Canvas больше в развлекательных целях, то библиотека Raphael.js (http://raphaeljs.com/) преследует сугубо практические цели (хотя и делает это с помощью SVG + VML). С ее помощью можно удобно и красиво представлять различные объемы данных во всем привычном формате графиков.

Применение этой библиотеки предельно просто: обычно нужно объявить необходимые данные и задать один из множества доступных представлений (или создать свое собственное). Более подробно с данной библиотекой можно ознакомиться на ее официальном сайте —http://raphaeljs.com/.

6.6.5. Проблемы быстродействия

На данный момент Canvas при решении большинства задач справляется быстрее, чем SVG. Достаточно давно был разработан пример использования Canvas для ряда задач Google Maps (http://www.ernestdelgado.com/gmaps/canvas/ddemo1.html). В нем зафиксирован прирост скорости в 200-500% (для всех браузеров, которые поддерживают Canvas).

Рис. 6.8. Пример использования Raphael.js для отображения данных, источник: raphaeljs.com

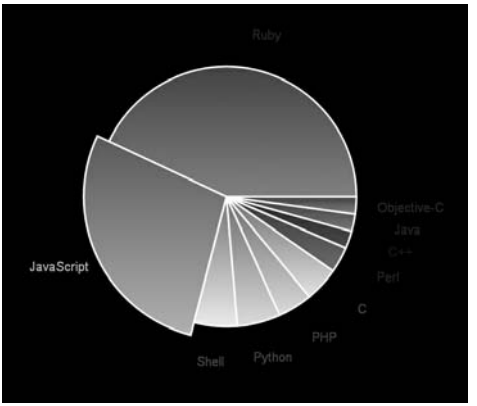
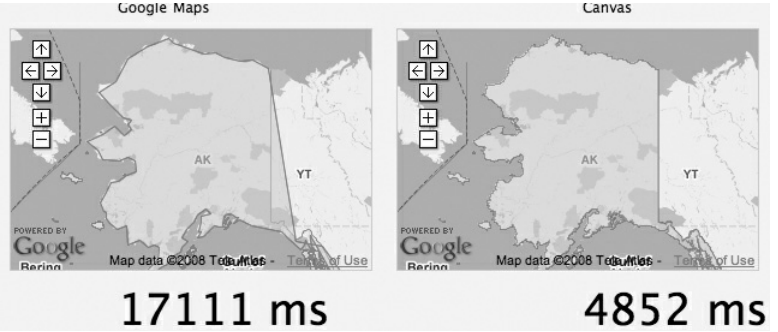


Рис. 6.9. Быстродействие Canvas, источник: www.ernestdelgado.com



В результате другого тестирования (http://prototype-graphic.xilinus.com/samples/shape.html) Canvas также демонстрирует значительное преимущество перед SVG, но менее широкий набор возможностей.

В качестве примера можно привести еще один тест (http://intertwingly.net/ stories/2006/07/10/penroseTiling.html) скорости отображения объектов в Canvas и SVG. Здесь SVG снова проигрывает (но совсем незначительно, в большинстве браузеров разницы почти нет).

Для уточнения вопросов производительности можно обратиться к исследованию (http://www.borismus.com/canvas-vs-svg-performance/), установившему закономерность между производительностью SVG, Canvas и параметрами изображения. В результате оказывается вполне очевид-

Рис. 6.10. Возможности и быстродействие Canvas, источник: prototype-graphic.xilinus.com

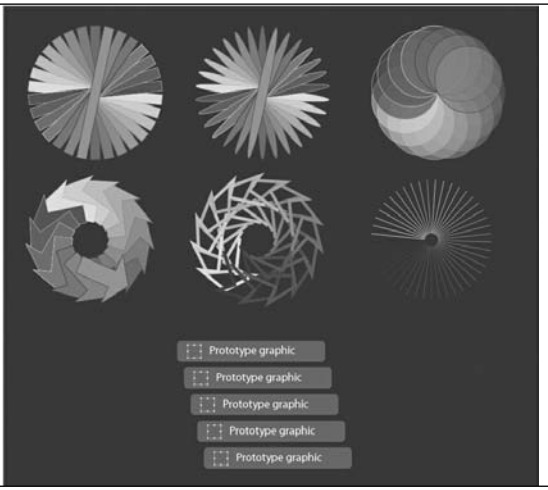
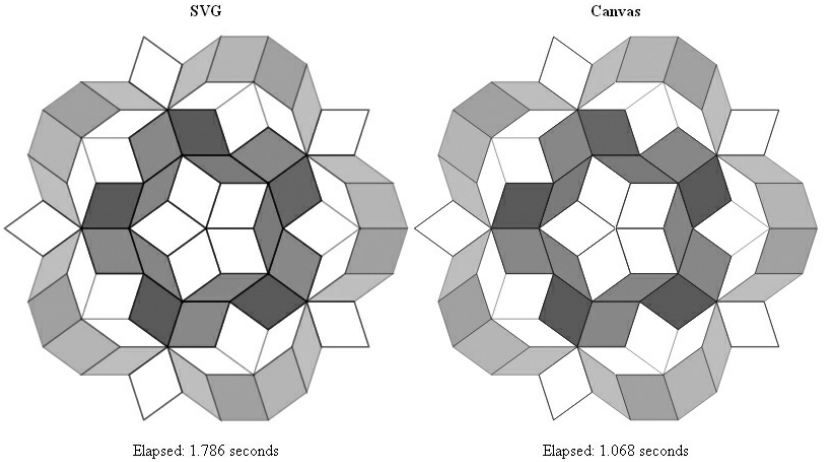
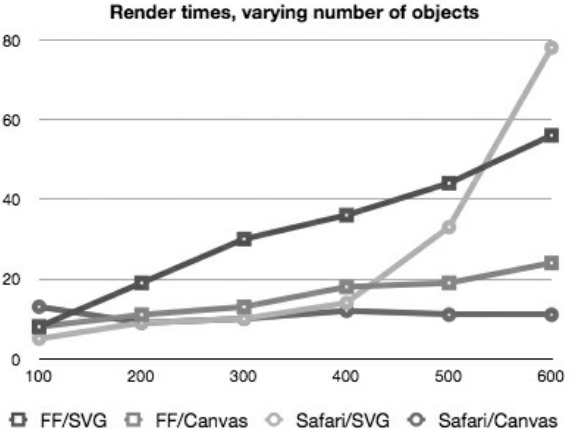


Рис. 6.11. Сравнение быстродействия Canvas и SVG, источник: intertwingly.net



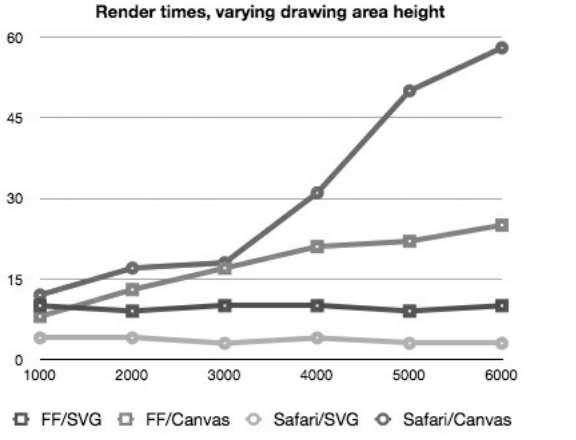
ным, что при увеличении числа объектов (для SVG — векторных) производительность SVG падает сильно (почти экспоненциально), а Canvas остается на стабильном уровне. Здесь стоит отметить, что размер активной области при этом не изменяется.

Рис. 6.12. Производительность Canvas и SVG при увеличении числа объектов, источник: www.borismus.com



Однако если мы начнем увеличивать область построения (размеры объектов), то тут векторный формат показывает себя во всей красе: производительность практически не меняется. Производительность Canvas падает (как и следовало ожидать) квадратичным образом от числа объектов (площадь активной области увеличивается квадратично).

Рис. 6.13. Производительность Canvas и SVG при увеличении размера объектов, источник: www.borismus.com



Из этого можно сделать простой вывод: если вы собираетесь использовать точечную (пиксельную) графику, то лучше Canvas для этой цели ничего не подходит. При работе с большими (по площади) векторными объектами лучше применять SVG. Также будет необходимо дублировать всю функциональность через VML для IE 8 и ниже.

6.7. Вычисляем при помощи Web Workers

Данный раздел написан под впечатлением статьи Джона Ресига «Computing with JavaScript Web Workers» *(*http://ejohn.org/blog/web-workers/*)*, в которой он раскрыл особенности встроенного в браузеры механизма «отложенных» вычислений при помощи JavaScript и его будущие перспективы.

Последняя спецификация Web Workers, несомненно, является наиболее перспективной из грядущих нововведений в браузерах, которая позволяет исполнять JavaScript-задачи параллельно, не блокируя интерфейс браузера.

Обычно для того, чтобы добиться более-менее приемлемых вычислений при помощи JavaScript-движка, необходимо было разбивать всю работу на небольшие части и запускать их друг за другом при помощи таймера. Это как не самый быстрый, так и совершенно не эффективный путь достижения поставленной задачи.

Имея в виду текущее положение вещей, давайте углубимся в спецификацию Web Workers.

6.7.1. Web Workers

Рекомендация Web Worker (http://www.whatwg.org/specs/web-workers/current-work/) частично основывается на уже проделанной работе со стороны команды Google Gears относительно модуля WorkerPool (http://code.google.com/apis/gears/api_workerpool.html). Это идея существенно выросла и была значительно доработана, чтобы стать рекомендацией.

Worker — это скрипт, который может быть загружен и исполнен в фоновом режиме. Web Workers позволяют легко это сделать, например:

```
new Worker("worker.js");
```

В этом примере будет загружен скрипт, расположенный по адресу worker.js, а его выполнение будет произведено в фоновом режиме (скорее всего, браузеры будут использовать встроенные средства операционной системы — нити — для реализации такого поведения).

Однако есть несколько серьезных «подводных камней»:

1. Worker не имеет доступа к DOM. Никакого `document`, `getElementById` и т. д. (Наиболее важными исключениями из этого правила будут методы `setTimeout`, `setInterval` и `XMLHttpRequest`.)
2. У Worker нет прямого доступа к родительской странице.

Имея в виду эти ограничения, стоит сразу задаться вопросом: а для чего, собственно, может пригодиться Worker и какие задачи он способен решать?

Вы можете использовать Worker, обмениваясь с ним сообщениями. Все браузеры (которые поддерживают данную спецификацию) позволяют обмениваться строковыми сообщениями (Firefox 3.5 также поддерживает обмен JSON-совместимыми объектами). Сообщение может быть как отослано Worker, так и сам Worker может ответить родительской странице таким сообщением. Ниже приведен пример обмена сообщениями.

Обмен сообщениями производится при помощи API postMessage следующим образом:

```
var worker = new Worker("worker.js");
// Ждем сообщений от worker
worker.onmessage = function(e){
// Сообщение от клиента:
  e.data;
};
worker.postMessage("start");
Клиент:
onmessage = function(e){
  if ( e.data === "start" ) {
// Выполняем какие-нибудь вычисления
    done();
  }
};
function done(){
// Отправляем полученный результат на главную страницу
  postMessage("done");
}
```

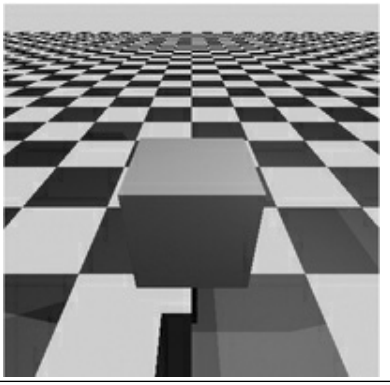
Это ограничение на передачу информации при помощи сообщений имеет под собой веские основания: оно позволяет безопасно запускать дочерний скрипт (потому что, к счастью, он не может повлиять на родительский) и защищает родительский поток (обеспечение безопасности для DOM от вторжения из других потоков превратилось бы в ночной кошмар для клиентских разработчиков).

Прямо сейчас Web Workers присутствуют в Firefox 3.5 и Safari 4. Они также заявлены в последних ночных сборках Chromium (http://build.chromium.org/buildbot/snapshots/chromium-rel-xp/). Наверное, большинство тут же подумают: что нам до возможности, которая доступна лишь для небольшой части веб-пользователей (только в двух современных браузерах!), — но это не должно быть проблемой. Web Workers позволяют эффективно использовать пользовательские машины для параллельных вычислений. В такой ситуации вы можете создавать две версии своих приложений (одну для старых браузеров, и одну — для запуска через механизм Web Workers при его наличии в браузерах). В новых браузерах это просто будет работать значительно быстрее.

Приведем некоторые занимательные примеры использования данного механизма, которые используют заявленное API.

6.7.2. Расчет освещения (RayTracing)

Рис. 6.14. Расчет освещения при помощи Web Workers, источник ejohn.org

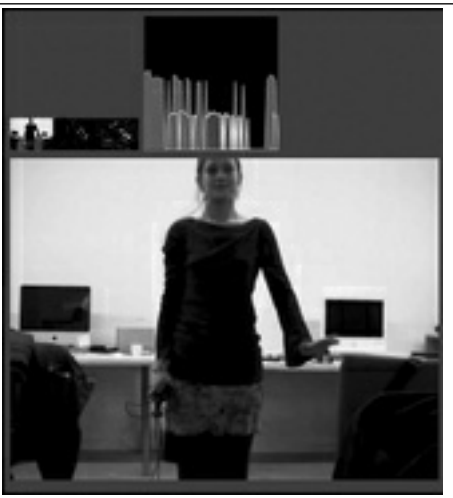


Здесь Canvas применен для отрисовки рассчитанной сцены. Если включить Web Workers, то заметно, как картинка отрисовывается по частям. Это происходит благодаря разбиению всей работы на части и поручению каждого набора пикселей отдельному Worker. Этот Worker затем по-

дает массив цветов для отрисовки на Canvas, а родительская страница их применяет. (Заметьте, сам по себе Worker ничего не изменяет.)

6.7.3. Отслеживание движения

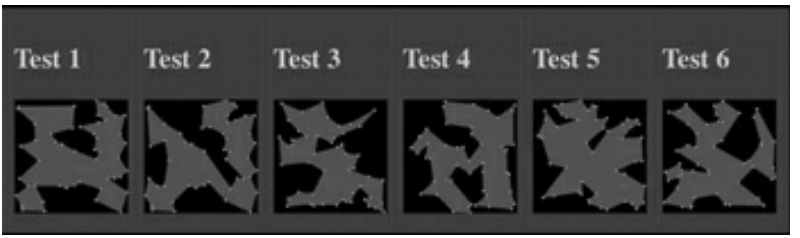
Рис. 6.15. Отслеживание движения при помощи Web Workers, источник: ejonh.org



В этом случае применяется несколько технологий: элемент `video`, элемент `canvas` и отрисовка кадров видео на сам холст. Все отслеживание движения производится в фоновом режиме при помощи Web Workers (поэтому передача видео не останавливается и не блокируется).

6.7.4. Эмуляция огня

Рис. 6.16. Эмуляция огня при помощи Web Workers, источник: ejohn.org



Этот пример пытается ограничить несколько случайных точек, используя алгоритм эмуляции огня (*simulated annealing*). Также здесь приведено анимированное PNG-изображение (работает в Firefox 3.5), которое вращается, пока идет вычисление в фоновом режиме.

6.7.5. Вычисление при помощи JavaScript Web Workers

Недавно закончилось интересное соревнование от Engine Yard (http://www.engineyard.com/blog/2009/programming-contest-win-iphone-3gs-2k-cloud-credit/). Задание было следующим: необходимо было подобрать фразу, которая бы давала наиболее близкий к исходному SHA1-хэш, расстояние между хэшами вычислялось по Хэммингу (число отличающихся битов).

Наибольший интерес представляли два участника, которые попытались решить эту задачу при помощи JavaScript. Их программы запускались в браузере и использовали пользовательский компьютер для проведения вычислений. Скорее всего, никто из них так и не добился впечатляющего результата (если брать во внимание тех кластерных монстров, с которыми пришлось соревноваться), но сам подход вызывает неподдельный интерес.

В обоих случаях (если верить исходному коду) используется почти одна и та же тактика: рассчитываются несколько результатов, разделяемых по таймеру. Скорость подбора варьируется от браузера к браузеру на уровне 1000-1500 расчетов в секунду. Естественно, эти алгоритмы весьма ресурсоемки и полностью «убивают» пользовательский процессор и «замораживают» интерфейс браузера.

По-видимому, это является великолепной возможностью применить Web Workers!

Если взять реализацию Ray C Morgan (http://www.raycmorgan.com/), вырезать весь интерфейс и таймеры и разложить вычисления на 4 параллельных потока для Web Workers, то можно добиться скорости в 4500-9500 расчетов в минуту (в новых браузерах, которые поддерживают механизм Web Workers).

По этим ссылкам можно посмотреть на демонстрацию (автором которой является Джон Ресиг) и скачать исходные коды:

- Пример Web Worker для взлома SHA1 (http://ejohn.org/apps/web-workers/)
- Исходный код для «родителя» (http://ejohn.org/apps/web-workers/run.js)
- Исходный код для Worker (http://ejohn.org/apps/web-workers/worker.js)

6.8. Клиентские хранилища

Чем больше сайтов становятся веб-приложениями, тем более возрастает потребность хранить какие-то данные на компьютере пользователя. Чаще всего речь идет о кэшировании, особенно если приложение предоставляет возможность работать без доступа к Интернету.

Хранение на стороне пользователя данных — настроек приложения, его состояния, частей кода и прочего, — способно сильно разгрузить канал, увеличить скорость загрузки и улучшить время реакции приложения на действия пользователя.

6.8.1. Cookie

Первопроходцем на ниве сохранения данных на клиенте можно назвать компанию Netscape, сотрудник которой придумал cookie — сохраняемые на компьютере пользователя в виде «ключ-значение» данные небольшого объема. Объем действительно небольшой, различные браузеры имеют разные ограничения, но даже в лучшем случае их не может быть больше 50, размером не более 4 Кб каждая. Некоторые версии Internet Explorer разрешают устанавливать не более 20 cookie, *общим* объемом не более 4 Кб (в IE 8 — 50 значений, общим объемом 10 Кб).

Другой недостаток: при каждом запросе все cookie передаются на сервер, что иногда сильно увеличивает объем трафика и снижает отзывчивость приложения. С передачей cookie на сервер связано еще одно ограничение их объема — дело в том, что многие сервера имеют различные лимиты на размер заголовка запроса, при превышении которого сервер откажется обработать запрос.

Например, у популярного веб-сервера Apache есть ограничение на длину каждой строки запроса (директива LimitRequestLine), по умолчанию оно составляет 8 Кб. Отсюда следует, что ограничение сверху на размер cookie (в том худшем случае, если cookie целиком будет состоять из URL-encoded символов) — 2,6 Кб. Конечно, если вы имеете доступ к настройке вашего сервера, это ограничение можно снять.

Но у cookie есть важное достоинство — к ним можно осуществлять доступ из клиентских скриптовых языков.

API cookie, если это так можно назвать, очень примитивное. Поскольку использование cookie в браузере можно запрещать, приложению сначала желательно проверить значение свойства `navigator.cookieEnabled`. Если его значение — истина, то можно прочитать значение `document.cookie`.

Значение этого свойства представляет собой строку, где записаны все cookie, объединенные через точку с запятой и пробел. Ключ и значение каждого cookie объединены через знак «равно», впрочем, из этого правила бывают исключения: авторам известны случаи, когда Internet Explorer записывал cookie с пустым значением без «равно».

У cookie помимо имени есть несколько атрибутов (перечислены поддерживаемые всеми браузерами):

- `expires` — время хранения cookie; если этот атрибут не указан, то cookie удалится после закрытия окна браузера, где значение этого cookie было создано. Чтобы удалить cookie, нужно поставить в это поле любую дату прошлого.
- `domain` — домен, для которого создается данное значение, остальные домены могут иметь собственные cookie с тем же именем; если параметр не задан, используется текущее имя домена. Допустимые значения — домен, с которого загружен документ, или поддомен этого домена.
- `path` — путь, для которого создается значение; по умолчанию используется путь текущей страницы. Путь тут трактуется как директория — значение будет существовать и для всех вложенных путей.
- `secure` — флаг, наличие которого означает, что данное значение cookie будет передаваться только по HTTPS.

Для того чтобы создать новое значение cookie из скриптового клиентского языка, нужно записать в `document.cookie` строку вида

```
key=value; expires=Fri, 31-Dec-2010 23:59:59 GMT; path=/; ↵
domain=.example.net
```

Как было сказано выше, все поля, кроме пары «ключ-значение», можно опустить. Точка перед доменом в параметре `domain` означает, что это значение будет установлено и для всех доменов выше уровнем. При записи нового значения cookie его значение в указанном домене изменится.

Идея cookie получила дальнейшее развитие за прошедшие годы, стандарт расширился, но поскольку управление этими расширениями из клиентских скриптовых языков недоступно, рассматривать их мы не будем.

6.8.2. userData behavior

Компания Microsoft, которая первой начала осваивать веб-приложения в том смысле, в котором мы сейчас используем этот термин, довольно давно предложила программистам механизм для сохранения более существенных объемов данных на компьютере пользователя.

Уже в Internet Explorer 5 появился так называемый userData behavior, собственное расширение Microsoft, которое позволяло хранить до 128 Кб данных в одной записи, общим объемом до 1 Мб, причем в интранете лимит еще более отодвигался — 512 Кб и 10 Мб.

API по нынешним временам нельзя назвать удобным, но тогда это было частью интересной идеологии behavior, позволяющей разделить данные и логику.

Данные хранятся в DOM-элементе, у которого, посредством CSS, указан behavior userData:

```
function IEStorage (storagename) {
    this.storagename = storagename
     var el = document.createElement('div');
     el.setAttribute('id', 'ourstore-' + storagename);
     el.style.display = 'none';
     el.addBehavior('#default#userData');
     document.body.appendChild(el);
     this.storage = el;
     this.get = function (name) {
          this.storage.load(this.storagename);
          return this.storage.getAttribute(name);
     }
    this.set = function (name, value) {
        this.storage.setAttribute(name, value);
        this.storage.save(this.storagename)
    }
    this.del = function (name) {
        this.storage.removeAttribute(name);
        this.storage.save(this.storagename);
    }
}
```

Как можно видеть, наш небольшой класс имеет четыре метода: первый создает и инициализирует хранилище, три остальных позволяют работать с его значениями — читать, писать и удалять их.

Особенность этого хранилища — можно установить дату устаревания всех его элементов, удалены они будут при вызове load:

```
var time = new Date();
time.setHours(time.getHours() + 1);
// через час данные будут удалены
this.storage.expires = time.toUTCString();
```

6.8.3. Flash Local Shared Object

В марте 2002 года появилась шестая версия Flash, популярнейшего плагина для браузеров, установленного, по статистике, на 95% компьютеров. В этой версии появилось собственное хранилище — Local Shared Object, позволяющее хранить до 100 Кб данных без ведома пользователя и любой объем сверх этого с его разрешения. У роликов с одного домена единый Local Shared Object.

Способ использования этого типа хранилища — установка на странице Flash-ролика, который будет обмениваться данными со скриптовым языком. Вплоть до восьмой версии Flash не существовало хорошего способа обмена данными со скриптовыми языками браузера, пока в Flash8 не появился ExternalInterface, реализованный, впрочем, с существенными ошибками.

Хотя и прежние способы, более похожие на хаки, позволяют производить обмен (по крайней мере теоретически), универсальные библиотеки для доступа к клиентским хранилищам, которые будут рассмотрены далее, требуют для своей работы восьмую версию Flash.

6.8.4. WHATWG DB Backend (openDatabase)

Рабочая группа WHATWG была организована производителями браузеров Apple, Mozilla Foundation и Opera Software ASA. Именно эта группа создала документ Web Application 1.0, который лег в основу пятой версии HTML.

В частности, этот документ решал и проблему хранения данных веб-приложения на стороне пользователя. Текущая спецификация HTML5 решает эту проблему

иначе, а метод, предложенный данной группой, был реализован в одной из версий движка WebKit.

Некоторые версии браузеров, основанных на WebKit — Safari (в том числе под iPhone) и Chromium также содержат openDatabase (именно так называется эта часть спецификации) и представляют собой несложный доступ к реляционной СУБД SQLite из JavaScript. Кроме того, поддержка openDatabase появилась в «Опере» 10.50.

Максимальный размер такого хранилища теоретически ограничен лишь местом на диске и внутренними ограничениями SQLite, но по умолчанию Safari устанавливает предел в 5 Мб.

Упрощенный (с минимальной обработкой ошибок) класс для доступа к openDatabase выглядит так:

```
function SafariStorage(name, maxsize) {
    this.db = null;
    if (!window.openDatabase) return;

    try {
        this.db = openDatabase(name, '1.0 ', 'Storage for ' + ↵
        name, maxsize); // maxsize — в байтах
    } catch (e) {
        this.db = null;
        return;
    }

    this.db.transaction(function (t) {
        t.executeSql('CREATE TABLE IF NOT EXISTS storage(k TEXT ↵
        UNIQUE NOT NULL PRIMARY KEY, v TEXT NOT NULL);', []);
})
    this.get = function (name, fn, scope) {
        scope = scope || this;
         this.db.transaction(function (t) {
             t.executeSql('SELECT v FROM storage WHERE k = ? ' ↵
             [name], function (t, r) {
                 if (r.rows.length) {
                     fn.call(scope, r.rows.item(0)['v'])
                 } else {
                     fn.call(scope, null)
                 }
             });
         });
```

```
    }
    this.set = function (name, value) {
        this.db.transaction(function (t) {
            t.executeSql('INSERT OR REPLACE INTO storage(k, v) ↵
            VALUES (?, ?)', [name, value]);
        });
    }

    this.del = function (name) {
        this.db.transaction(function (t) {
            t.executeSql('DELETE FROM storage WHERE k = ?', ↵
            [name]);
        });
    }
}
```

Метод transaction создает транзакцию и передает ее своему аргументу — функции, запрос внутри который будет обработан в единой транзакции. У transaction есть еще два аргумента — функции, первая из которых будет вызвана, если транзакция выполнена успешно, вторая — если закончилась ошибкой, объект ошибки будет первым аргументом вызываемой функции.

У executeSql (обратите внимание на регистр букв!) также есть опциональные аргументы: после текста запроса и его подстановок можно указать callback для возвращаемых данных и функцию, которая будет вызвана в случае ошибки, с двумя аргументами: транзакцией и объектом ошибки.

6.8.5. globalStorage и localStorage

Спецификация на globalStorage присутствует в ранних версиях черновика HTML5, откуда ее удалили из соображений безопасности. Основная идея globalStorage — дать разработчикам возможность обращаться к данным поддоменов, как это сделано в cookie.

Этот вид хранилища был реализован в Firefox 2 (и в Internet Explorer 8.0 beta 1), но уже в следующей версии браузера возможность обращаться к данным поддоменов отключили. В такой «урезанной» версии globalStorage по возможностям эквивалентно хранилищу localStorage, которое заняло в HTML5 нишу своего небезопасного предшественника.

В настоящий момент localStorage поддерживается браузерами FireFox 3.5 и выше, Internet Explorer 8.0 (с версии beta 2), Safari 4.0 и выше, а так же Opera 10.50 и выше.

Ограничения, накладываемые на размер хранилища, устанавливаются разными производителями браузеров по-разному, так, в Firefox это 5 Мб, а в IE 8 — 10 миллионов байт (причем, учитывается место, занятое не только значениями, но и именами ключей). Другое ограничение накладывается на имена ключей, в частности, в них не может быть пробелов.

```
function HTML5Storage() {
    if (window.localStorage) {
         this.storage = window.localStorage
    } else if (window.globalStorage) {
         this.storage = window.globalStorage[location.hostname ↵
         || 'localhost.localdomain ']
    }  else {
         return false
    }

    this.get = function (name) {
        var out = this.storage.getItem(name);
        return out && out.value ? out.value : out;
    }

    this.set = function (name, value) {
        this.storage.setItem(name, value);
    }
    this.del = function (name) {
        this.storage.removeItem(name)
    }
}
```

6.8.6. Google Gears

Последнее из рассматриваемых нами хранилищ — Google Gears, расширение для браузеров, придуманное Google. В настоящее время встроено в браузер Google Chrome, для других поддерживаемых браузеров скачивается и устанавливается отдельно. Диапазон поддерживаемых браузеров довольно широк: Firefox начиная с версии 1.5, Internet Explorer 6.0 и выше, Safari 3.1.1 и выше (для Mac OS), а также мобильные браузеры — IE Mobile с версии 4.01 и Opera Mobile 9.51 и выше.

Основная идея хранилища Google Gears та же, что и у openDatabase, тот же самый доступ к SQLite, но с несколько другим API:

```
function GearsStorage(name) {
    if (!window.google || !window.google.gears) {
        var factory = null;
        // Firefox
        if (typeof GearsFactory != 'undefined') {
            factory = new GearsFactory();
        } else {
             // IE
            try {
                 factory = new ActiveXObject('Gears.Factory');
                if (factory.getBuildInfo().indexOf('ie_mobile')
                != -1) {
                    factory.privateSetGlobalObject(this);
                 }
             } catch (e) {
                 // Safari
                if ((typeof navigator.mimeTypes != 'undefined')
                && navigator.mimeTypes["application/ ↵
                x-googlegears"]) {
                    factory = document.createElement("object");
                    factory.style.display = "none";
                    factory.width = 0;
                    factory.height = 0;
                    factory.type = "application/x-googlegears";
                    document.documentElement.appendChild(factory);
                }
        }
        if (!factory) return;
        if (!window.google) {
            google = {};
        }
        if (!google.gears) {
            google.gears = {factory: factory};
        }
    }
    this._begin = function () {
        this.db.execute('BEGIN').close();
    }
    this._commit = function () {
        this.db. execute('COMMIT').close();
    }
```

```
    this.db = google.gears.factory.create('beta.database');
    this.db.open(name)
    this.db.execute('CREATE TABLE IF NOT EXISTS storage(k TEXT ↵
    UNIQUE NOT NULL PRIMARY KEY, v TEXT NOT NULL);').close()
    this.get = function (name) {
        this._begin()
        var r = this.db.execute('SELECT v FROM storage WHERE k
        = ? ', [name])'
        this._commit()
        var result = r.isValidRow() ?  r.field(0) : null;
        r.close();
        return result;
    }
    this.set = function (name, value) {
        this._begin()
        this.db.execute('INSERT OR REPLACE INTO storage(key, ↵
        value) VALUES(?, ?) ', [name, value]).close()
        this._commit()
    }
    this.del = function (name) {
        this._begin()
        this.db.execute('DELETE FROM  storage WHERE key = ? ', ↵
        [name]).close()
        this._commit()
    }
}
```

Как видно, существенную часть занимает инициализация Google Gears, в каждом браузере она выполняется по-разному. API устроено проще, чем в openDatabase — транзакции нужно создавать самостоятельно, зато есть возможность возвращать данные непосредственно, не прибегая к callback.

Авторы не обладают сведениями об ограничениях на размер этого хранилища, вполне может быть, что никаких ограничений, помимо тех, что есть в самом SQLite, не существует.

6.8.7. Библиотеки для работы с клиентскими хранилищами

Итак, мы рассмотрели основные из имеющихся на сегодняшний момент в распоряжении программиста клиентских хранилищ. Такое обилие решений, большинство из которых поддерживаются ограниченным набором браузеров, не могло не привести к появлению специализированных

библиотек для работы с клиентскими хранилищами. Не умаляя полезности минимальных знаний о работе каждого хранилища, мы все же рекомендуем для доступа к ним использовать одну из готовых библиотек, которые будут рассмотрены ниже.

PersistJS (http://pablotron.org/?cid=1557) — пожалуй, наиболее известная библиотека на этом поприще. Она поддерживает все перечисленные виды хранилищ (правда, Google Gears для IE не поддерживается), но, несмотря на свою известность, обладает рядом существенных недостатков.

Первый недостаток — нет возможности изменить последовательность, в которой PersistJS перебирает методики хранения данных. К примеру, если вы решили поставить localStorage выше Google Gears, а cookie исключить, вам придется вмешиваться в код библиотеки.

Другая проблема — если библиотека обнаружила, например, хранилище Google Gears, а клиент ответил отказом на запрос разрешения хранения данных, то PersistJS останется в неопределенном положении.

Еще недоработка — PersistJS не проверяет готовность Flash-ролика, и в том случае, если вы используете это хранилище, есть небольшая вероятность того, что ваш код попытается обратиться к данным еще до того, как ролик будет загружен. Другая связанная с Flash-роликом проблема — библиотека не умеет запрашивать дополнительное место для хранения данных, если 100 Кб, которые можно использовать без запроса разрешения, исчерпаны.

Неприятно также, что библиотека позволяет сохранять только текстовые строки, не предоставляя возможности сериализации, впрочем, версия из репозитория умеет представлять сложные объекты в виде JSON.

Впрочем, у библиотеки хороший плюс — небольшой (около 9 Кб) размер.

Dojo (http://www.dojotoolkit.org/) поддерживает хранилища Flash, Google Gears, globalStorage (Firefox 2.0) и среду запуска веб-приложений Adobe AIR.

Этот фреймворк лишен недостатков PersistJS, но есть изъян — гигантский размер: версия 1.3.2 занимает 45 Мб. Конечно, для работы с хранилищем весь фреймворк не нужен, но даже минимально необходимый набор занимает более 100 Кб.

Существует адаптированная версия Dojo Storage, которая называется **SRAX Storage** (http://fullajax.ru/#:download/), но она поддерживает только Flash Local Shared Object.

jStore (http://code.google.com/p/jquery-jstore/) — небольшой плагин к jQuery, который поддерживает все рассмотренные хранилища, кроме cookie, обладает умеренным размером (около 15 Кб в минимизированном виде) и свободен от недостатков PersistJS.

6.8.8. Резюме

К сожалению, даже создатели HTML5 не предусмотрели возможности адресации к хранилищу посредством URL. Было бы очень удобно, положив в хранилище JavaScript или графическое изображение, подключить содержимое через обычный тег `HTML`*.

Необходимо помнить, что хранилище — клиентское, и данные, сохраненные в хранилище, не будут доступны клиенту на другой машине и даже (увы) в другом браузере. Так что наиболее подходящее использование клиентских хранилищ — кэширование текстовых данных: JavaScript, CSS,

Хранилище	**Ограничение**	**Браузеры**
userData behavior	Интранет — 512 Кб каждая запись, 10 Мб на домен, ограниченные узлы — 64 Кб / 640 Кб, остальные — 128 Кб / 1Мб	IE 5.0+
globalStorage	5 Мб	FF 2.0+, IE 8 beta 1
localStorage	Firefox, Safari, Opera — 5 Мб Internet Explorer — 10 MiB	FF 3.5+, Safari 4+, IE 8 beta 2+, Opera 10.50+
openDatabase	5 Мб	Opera 10.50+, некоторые версии Safari 3.xx, 4.xx, Google Chrome 3.xx, 4.xx
Google Gears	Запрашивает разрешение на использование, ограничений по размеру нет	Встроен в Google Chrome, плагины для IE 6+, Opera Mobile 9.51+, FF 1.5+, IE Mobile 4.01+, Safari 3.1.1+
Flash	Запрашивает разрешение на хранение более 100 Кб данных, ограничений по размеру нет	Плагины для IE, FF, Google Chrome, Safari

* создатели Google Gears озаботились доступом к бинарным данным из своего хранилища, изображениями можно манипулировать и выводить в окно браузера при помощи специального расширения, используя тег Canvas.

а значит, данных о состоянии приложения и его редко изменяемых ресурсов, которые выгоднее хранить на стороне пользователя.

Принцип использования прост: проверяется наличие нужного ресурса в хранилище; если ресурс обнаружен, он загружается из хранилища, если нет, с сервера (например, при помощи AJAX) и сохраняется в хранилище. Далее соответствующий тег с полученным содержимым создается в DOM.

К сожалению, браузер Opera не предоставляет никакого встроенного хранилища, хотя в нем можно использовать Flash Local Shared Object; зато современные версии остальных браузеров не нуждаются в установке каких-либо дополнительных плагинов.

Всеми браузерами без исключения поддерживаются cookie, но их вряд ли можно рекомендовать в качестве клиентского хранилища из-за сильных ограничений на размер и увеличения исходящего трафика.

Из всех рассмотренных библиотек для доступа к хранилищам оптимальнее, на наш взгляд, использовать библиотеку jStore: она достаточно гибкая, небольшого размера и поддерживает все основные виды хранилищ.

Глава 7. Автоматизация клиентской оптимизации

Со дней подготовки и издания предыдущей книги прошло уже много времени. Технологии не стояли на месте и семимильными шагами рванули вперед. Текущая глава посвящена обзору текущих технологий для автоматизации клиентской оптимизации и лидеру этого рынка для веб-сайтов на PHP — Web Optimizer (http://www.web-optimizer.ru/).

7.1. Обзор технологий

На данный момент тематика автоматической клиентской оптимизации сильно волнует умы веб-программистов, предпринимателей и просто энтузиастов. Выгоды вполне очевидные: быстрый сайт имеет значительные преимущества перед медленными конкурентами. При наличии высокой конкуренции это может оказаться существенным. Более того, пользователи не склонны ждать долго. Быстрая загрузка может являться ключом к процветанию интернет-направления целой компании.

Понимание этого существует давно. Однако создать мощное и открытое веб-приложение, которое бы аккумулировало весь накоплен-

ный опыт и самостоятельно оптимизировало бы конечный сайт, до сих пор не удавалось. Давайте посмотрим на те продукты, с помощью которых можно автоматизировать те или иные действия по клиентской оптимизации.

7.1.1. JSMin Ant Task

JSMin Ant Task (http://code.google.com/p/jsmin-ant-task/). Приложение позволяет воспользоваться логикой работы JSMin (алгоритма преобразования JavaScript-кода путем удаления из него ненужных символов) при работе с Ant-сервером.

7.1.2. JSMin PHP

JSMin PHP (http://code.google.com/p/jsmin-php/). Достаточно известное PHP-приложение, реализующее логику JSMin на PHP. Из замеченных недостатков: отбрасываются условные комментарии и могут возникнуть проблемы при разборе сложных регулярных выражений. Во всем остальном хорошо себя зарекомендовало (также и по скорости преобразования кода). При дополнительном gzip-сжатии незначительно уступает YUI Compressor, но для работы требует лишь PHP.

7.1.3. YUI Compressor

YUI Compressor (http://developer.yahoo.com/yui/compressor/). Данный инструмент возник из Rhino-оптимизатора и активно развивается специалистами Yahoo!. YUI Compressor идет дальше в оптимизации JavaScript-кода: он заменяет имена всех локальных переменных их сокращенными (в большинстве случаев до 1 символа) вариантами. При использовании с gzip-сжатием дает наилучший результат. К сожалению, требует на сервере установленной Java.

7.1.4. Packer

Packer (http://dean.edwards.name/packer/) от Dean Edwards. В отличие от всех предыдущих инструментов Packer создает некоторое подобие архива, применяя встроенные средства JavaScript, при этом степень сжатия получается весьма значительной (до 50%). Однако при дополнительном использовании gzip-файлы, сжатые с помощью Packer, проигрывают своим аналогам, преобразованным при помощи YUI Compressor.

К дополнительным минусам стоит отнести некоторую нагрузку на процессор при распаковке такого архива (обычно составляет 30-300 мс). Доступен в качестве реализации на PHP.

7.1.5. CSS Min PHP

CSS Min PHP (http://code.google.com/p/cssmin/) является попыткой применить логику JSMin для оптимизации CSS-кода. Достаточно бедно функциональностью, но для простейших задач (минимизация CSS в одну строку) вполне подходит.

7.1.6. CSS Tidy

CSS Tidy (http://sourceforge.net/projects/csstidy/) — наиболее мощный на сегодняшний день инструмент для анализа и оптимизации CSS-кода. Позволяет не только отформатировать исходный файл по заданному шаблону, но и привести его в стандартный вид (по аналогии с HTML Tidy для HTML). Имеет множество настроек для оптимизации кода, в том числе и пересортировку CSS-селекторов для уменьшения размера.

Приложение портировано на два языка: PHP и C (есть версия для локального использования). На текущий момент приложение застыло в версии 1.3, и требуются добровольцы, чтобы привести его в соответствие с текущим состоянием веб-стандартов и продолжить разработку.

7.1.7. Minify

Minify (http://code.google.com/p/minify/) является первым приложением, которое попыталось автоматизировать значительную часть действий по клиентской оптимизации. Приложение используется главным образом для объединения, минимизации и кэширования CSS- и JavaScript-файлов, минимизации и кэширования HTML-документов. Имеет модульную структуру и может быть встроено в процесс публикации веб-сайтов.

Интеграция его с рабочим сайтом достаточно сложна (необходимо обладать продвинутыми знаниями, чтобы правильно настроить приложение и решить возникающие проблемы), но эффективность является весьма впечатляющей. К дополнительным минусам стоит отнести отсутствие поддержки CSS Sprites и `data:URI` технологий, а также невозможность применения его для распределения параллельных загрузок.

7.1.8. qpimg

qpimg (http://code.google.com/p/qpimg/) — php5-библиотека для динамического создания CSS-спрайтов (CSS sprites). Использование CSS-спрайтов является одним из эффективных методов клиентской оптимизации веб-страниц (подробнее об этом см. в главе 4). Данный инструмент позволяет назначать конечные позиции для фоновых изображений для автоматического их объединения.

В качестве альтернативного решения можно привести инструмент `coolRunnings` (http://jaredhirsch.com/coolrunnings/about/), который решает ту же самую задачу, но представляет собой отдельный веб-сервис.

7.1.9. Smart Sprites

Smart Sprites (http://csssprites.org/) идет дальше в процессе объединения CSS Sprites и предлагает делать это в полностью автоматическом режиме, создав ряд инструкций в комментариях в CSS-файле. Проект изначально написан на Java, но имеет и PHP-ветку.

7.1.10. SpriteMe

SpriteMe (http://www.stevesouders.com/spriteme/) — новый проект известного Steve Souders (автора двух книг по клиентской оптимизации и со-организатора конференций Velocity, полностью посвященной вопросам клиентской производительности), который позволяет создавать CSS Sprites в интерактивном режиме.

В качестве основных плюсов стоит отметить полностью автоматический режим, моментальное применение изменений, связанных с объединением картинок (всегда можно увидеть, как изменится дизайн в любом браузере), и возможность гибкой настройки и изменения файлов с самими спрайтами (файлы создаются при помощи веб-сервиса coolRunnings).

7.1.11. Smart Optimizer

Smart Optimizer (http://code.google.com/p/smartoptimizer/) позиционирует себя (в отличие от Minify) как отдельное веб-приложение, направленное на ускорение клиентской составляющей веб-сайтов. В качестве основных возможностей стоит отметить: объединение и минимизацию CSS- и JavaScript-файлов, кэширование на клиентском и серверном уровне, возможность конвертации в `data:URI`.

Приложение довольно сложно в установке (необходимо внесение изменений как в `.htaccess`, так и в исходные файлы шаблона сайта) и обладает рядом ограничений (не позволяет объединять файлы из разных директорий). Доступно только для PHP-сайтов, работающих под управлением Apache + `mod_php`.

7.1.12. PHP Speedy

PHP Speedy (http://code.google.com/p/phpspeedy/) — более продвинутое приложение для автоматизации действий по клиентской оптимизации. Имеет собственный мастер установки, в котором позволяет настроить конфигурацию и выдает список необходимых изменений для исходных файлов сайта (обычно это 2 строки с вызовами PHP Speedy).

Основные возможности: минимизация, объединение и кэширование CSS- и JavaScript-файлов, «безопасная» конвертация в `data:URI`, настройка использования `.htaccess`, настройка директорий кэширования и возможность исключения ряда файлов.

На данный момент приложение доступно в том числе как дополнение к Wordpress и Joomla! 1.0, а также может быть установлено на любые сайты, использующие PHP5.

7.1.13. Web Optimizer

Web Optimizer (http://www.web-optimizer.ru/) является сегодня наиболее мощным веб-приложением с открытым кодом для клиентской оптимизации. Для большинства сайтов ускорение составляет 3-5 раз (в некоторых случаях доходя до 7-10 раз), результат оценки по YSlow поднимается до 92-98 баллов (из 100). Для установки не нужно глубоких знаний технологии или специфических прав на сайте.

Список поддерживаемых систем управления сайтом и фреймворков включает несколько десятков наиболее известных, в том числе: Wordpress, Joomla!, Drupal, Bitrix, NetCat, UMI.CMS, DataLife Engine, Simple Machines Forum, phpBB, Invision Power Board и т. д. Более подробно об этом приложении рассказывается далее в этой главе.

Web Optimizer работает как на PHP4, так и на PHP5, а также если PHP подключается через CGI, а не `mod_php`. Доступен как плагин к Wordpress и Joomla!.

7.1.14. Web Application Optimizer

Следующее приложение, Web Application Optimizer (http://wao.monosoftware.com/), обладает почти тем же функционалом, что и Minify, но при

этом предназначено для сайтов, использующих ASP .NET. В числе основных возможностей: уменьшение и сжатие CSS- и JavaScript-файлов, уменьшение и gzip-сжатие для HTML-файлов. После установки приложения необходима его дополнительная ручная настройка.

WAO является платным и распространяется по цене от $99 для одного сайта.

7.1.15. Aptimize

Aptimize (http://www.aptimize.com/) предназначено для решения проблем более глобальным образом. Это приложение представляет собой отдельный модуль для IIS или Apache (для его установки нужны права администратора) и предлагает почти весь спектр оптимизационных действий, начиная от объединения файлов и сжатия и заканчивая созданием CSS Sprites (используется довольно простой алгоритм) и кэшированием всех ресурсов.

Aptimize распространяется по годовой подписке в размере $1000 для одного сервера.

7.2. Установка Web Optimizer

Давайте рассмотрим базовую установку веб-приложения для автоматической клиентской оптимизации — Web Optimizer.

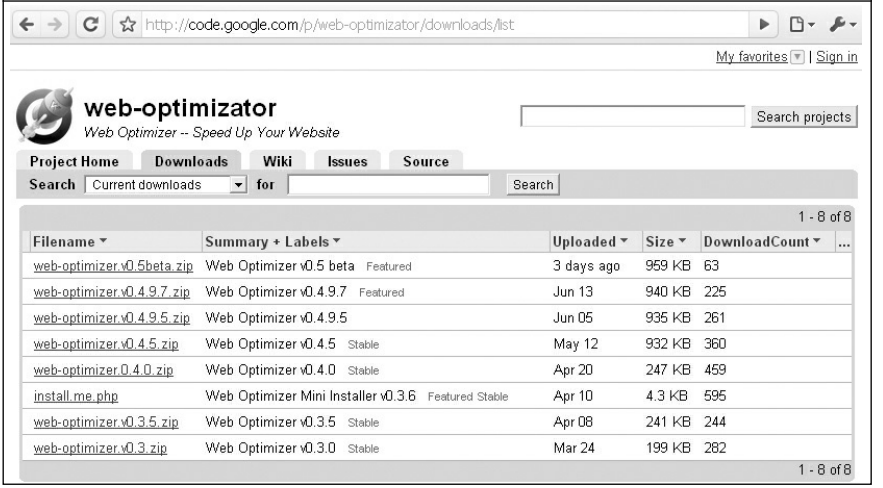
7.2.1. Шаг 1: загрузка архива

Web Optimizer поставляется в двух вариантах: ZIP-архив и мини-установщик. Для загрузки первого варианта идем по адресу code.google.com/p/web-optimizator/downloads/list и выбираем Featured версию 0.5 или выше.

Загружаем ZIP-архив в корень сайта. Если к сайту есть SSH-доступ, то можно использовать просто wget:

```
wget http://web-optimizator.googlecode.com/files/web-
optimizer.v0.5beta.zip
```

Рис. 7.1. Загружаем Web Optimizer



Затем полученный архив нужно будет распаковать в корень, чтобы получилась папка web-optimizer.

Если к сайту есть только FTP-доступ, то загружаем сначала на локальный диск, потом распаковываем, а потом уже (например, через FAR) копируем в корень сайта.

При отсутствии желания загружать распакованный архив на сервер (или распаковывать на сервере загруженный архив) есть версия мини-установщика, который (если имеется curl на сервере) сам все загрузит и начнет установку. Для этого нужно загрузить только файл install.me.php в корень сайта и открыть его в браузере.

После того как все необходимые файлы оказались на сайте, нужно выставить права на запись, как минимум, для файла web-optimizer/config.php и (опционально) папки web-optimizer/cache для пользователя, под которым работает сервер. Иначе настройки и закэшированные версии сжатых файлов не смогут сохраниться. При желании папка кэширования может быть другой (об этом чуть ниже), в этом случае будет необходимо только выставить права на конфигурационный файл.

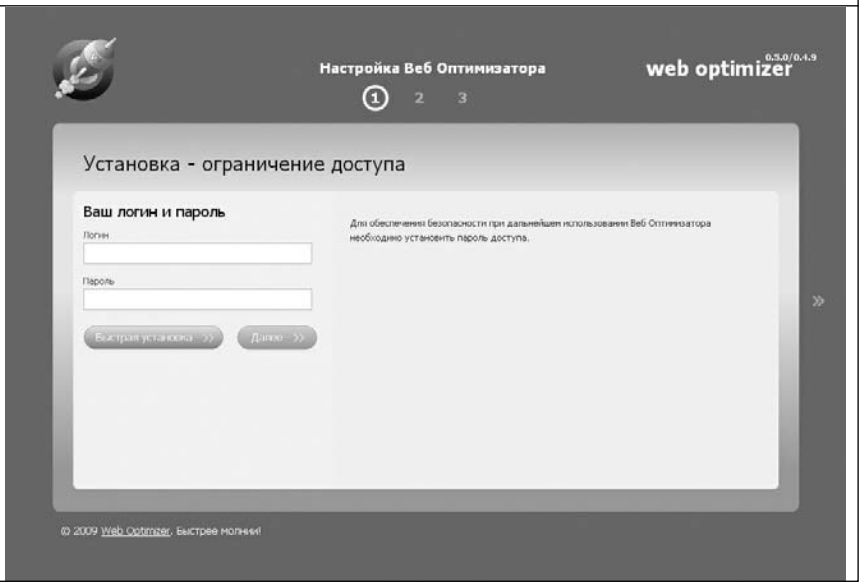
7.2.2. Шаг 2: настройка

Заходим в браузере по адресу

```
http://ваш_сайт/web-optimizer/index.php
```

Вместо web-optimizer может быть произвольная директория, в которой находится Web Optimizer. Видим приветственный экран от Web Optimizer. Если не видим, то стоит перепроверить, куда был скопирован Web Optimizer, и зайти именно в ту папку.

Рис. 7.2. Устанавливаем Web Optimizer



Здесь возможно два варианта развития событий:

- Быстрая установка;
- Обычная установка.

Быстрая установка

Для начала быстрой установки вводим будущий логин и пароль доступа к административной части и нажимаем зеленую кнопку «Быстрая установка». После этого Web Optimizer вычисляет директории на сервере, сохраняет все настройки по умолчанию и осуществляет цепочную оптимизацию для главной страницы: создает закэшированные версии сжатых файлов. Только потом, если файлы (обычно только корневой `index.php`, зависит от используемой CMS) доступны на запись, производится их автоматическое изменение.

Иначе Web Optimizer выведет инструкции по изменению этих файлов.

Рис. 7.3. Быстрая установка Web Optimizer

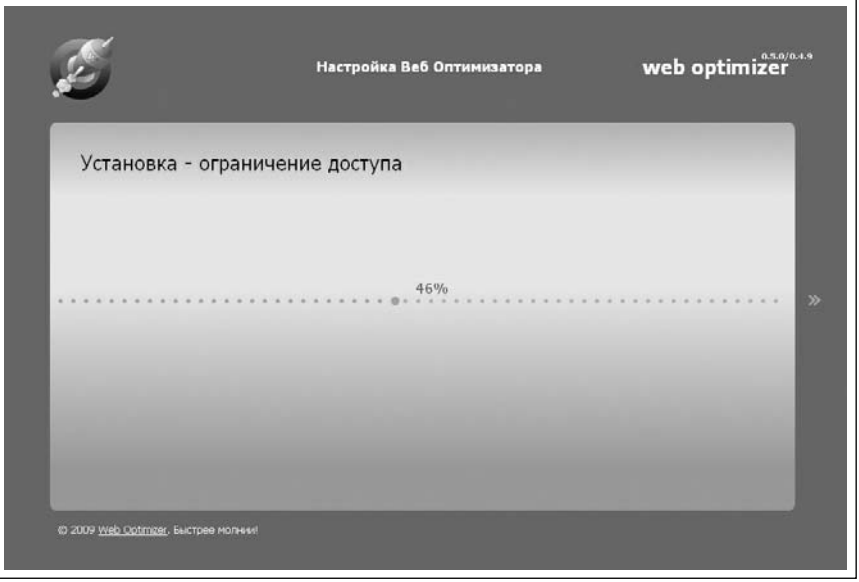


Рис. 7.4. Окончание установки Web Optimizer

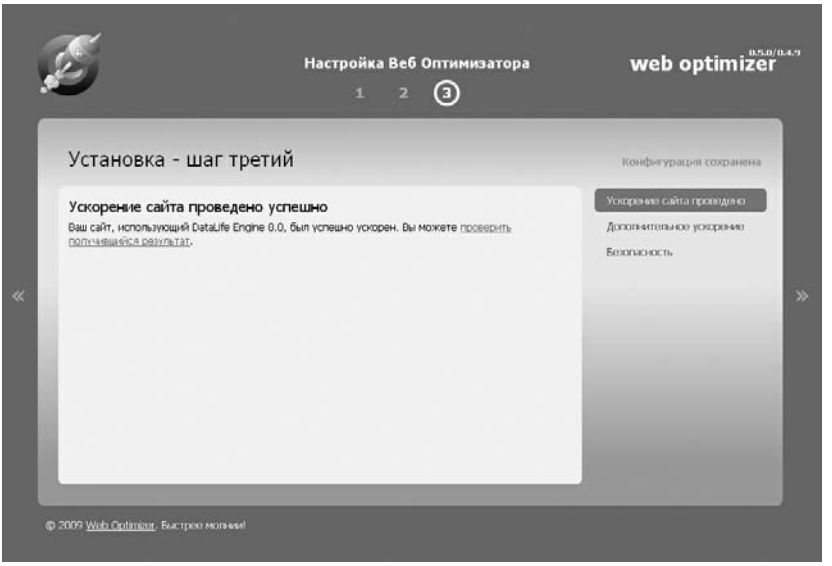


Рис. 7.5. Окончание установки Web Optimizer: инструкции по интеграции

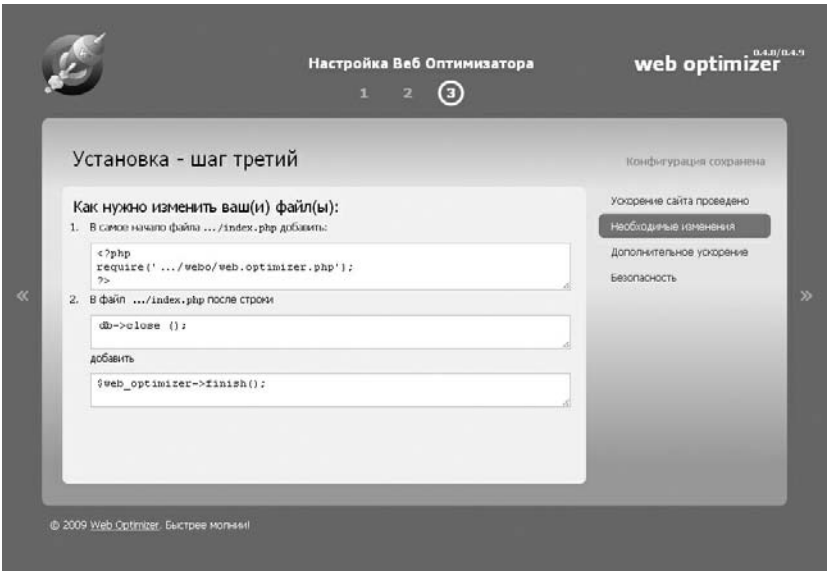
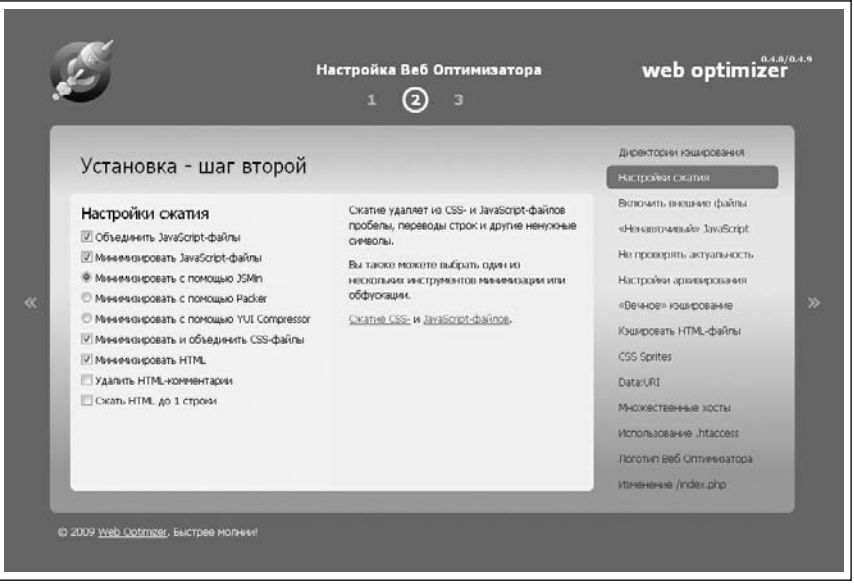


Рис. 7.6. Настройка Web Optimizer

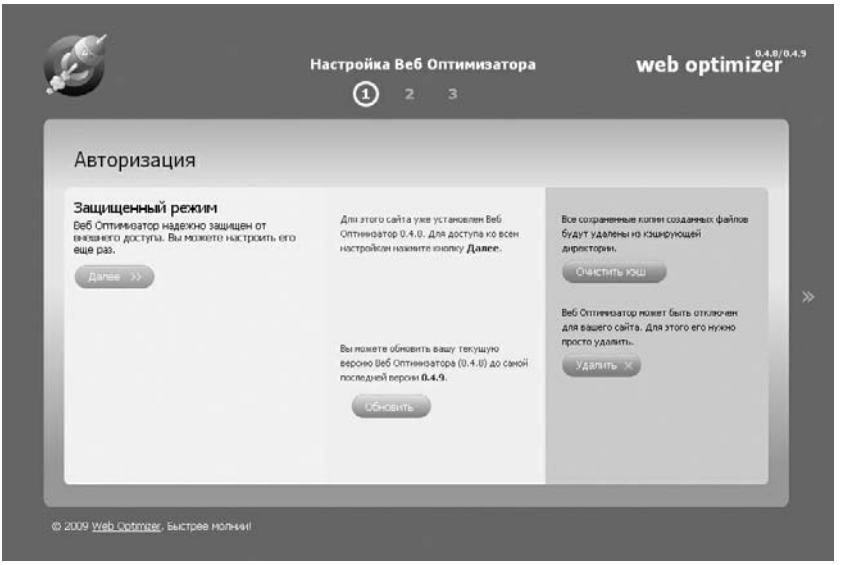


Обычная установка

Обычная установка отличается от быстрой только наличием промежуточного шага с редактированием настроек. Запустить ее можно, нажав по кнопке «Далее», цифре 2 или 3 в верхнем меню или оранжевой стрелочке справа.

Все настройки и их особенности подробно описаны в следующем разделе. Прежде всего нужно убедиться в том, что вычисленные пути являются правильными. Также можно задать произвольные директории кэширования: это будет необходимо при включении настройки «Защищенный режим» (находится в разделе «Использование `.htaccess`»). После этой настройки пароль при доступе к Web Optimizer будет запрашиваться только через HTTP Basic Authorization. Дополнительно вводить его не потребуется. Однако файлы, которые находятся внутри папки с Web Optimizer, станут недоступны обычным пользователям, поэтому директории кэширования нужно из нее перенести.

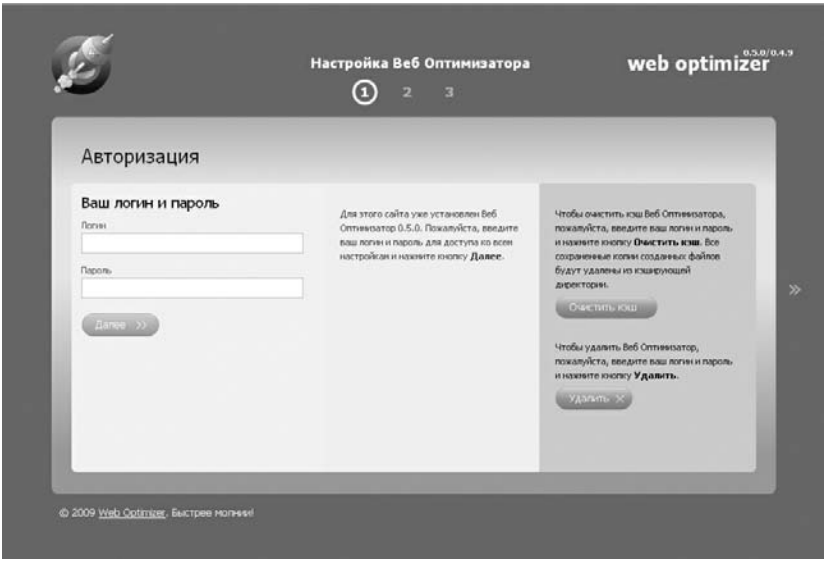
Рис. 7.7. Защищенный режим Web Optimizer



7.2.3. Шаг 3: Управление

В Web Optimizer доступно несколько инструментов для управления приложением.

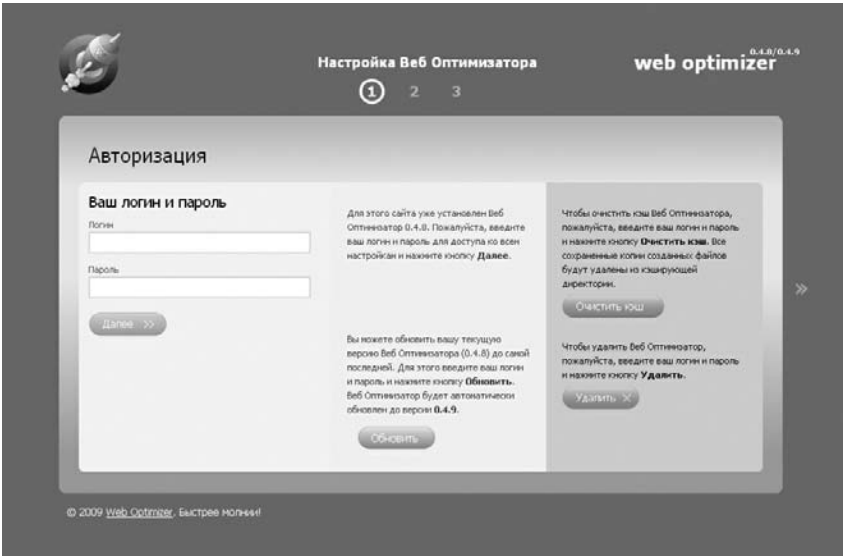
Рис. 7.8. Панель управления Web Optimizer



Во-первых, это конфигурирование всех настроек (здесь и далее при работе не в защищенном режиме нужно будет ввести логин и пароль), которое можно осуществить по кнопке «Далее» (или клику по цифрам 2 или 3 или оранжевой стрелочке справа). Во-вторых, это очистка кэша (будет необходимо, если вы провели изменение каких-либо CSS- или JS-файлов на сервере при включенной настройке «Не проверять время изменения файлов») — кнопка «Очистить кэш». В-третьих, это возможность безболезненно удалить Web Optimizer (будут удалены все добавленные в файлы CMS вызовы, а файл `.htaccess` будет очищен от оптимизационных директив) кнопкой «Удалить».

При наличии `curl` на сервере и существовании более новой версии, чем текущая, будет предложено обновиться (появится блок с кнопкой «Обновить»). При обновлении все исходные настройки будут сохранены. Также могут добавиться некоторые новые. В обычном режиме панель администрирования для Web Optimizer выглядит следующим образом:

Рис. 7.9. Панель управления Web Optimizer: установлена последняя версия



Web Optimizer перехватывает выводимый HTML-документ и преобразовывает его к оптимальному виду. В частности, анализируется вся секция `head` на предмет поиска CSS- и JavaScript-файлов, также при существовании статических хостов изображения распределяются по ним (меняются адреса у изображений), а блоки с рекламой и счетчиками переносятся и устанавливаются перед `</body>` (опять-таки только при включении соответствующей настройки). Также HTML подвергается минимизации (удаляются лишние переводы строк и отступы, могут удаляться комментарии и вообще все лишние символы, но это ресурсоемкие операции, и по умолчанию они выключены).

7.3. Настройка Web Optimizer

Ниже приведены настройки, доступные в Web Optimizer версии 0.5.

7.3.1. Директории кэширования

Здесь можно выставить пути к кэширующим директориям (на файловой системе), в которых будут записываться сохраненные уменьшенные

CSS-, JavaScript- и HTML-файлы. Также здесь можно определить корневую директорию сайта (необходима для правильного расчета всех относительных путей). По умолчанию все кэширующие директории назначаются в папке cache в самом Web Optimizer.

7.3.2. Настройки сжатия

Эта группа настроек отвечает за объединение и минимизацию JavaScript- и CSS-файлов. По умолчанию JavaScript-файлы объединяются и минимизируются при помощи JSMin (или YUI Compressor, если доступна Java). Также можно сжимать JavaScript при помощи Dean Edwards Packer (является лучшим выбором при отсутствии gzip-сжатия). Здесь также можно настроить, каким образом минимизировать выводимый HTML-код (простое удаление лишних переводов строк и пробелов, «вытягивание» в одну строку и(или) удаление комментариев). Условные комментарии для IE не затрагиваются ни в каком случае.

7.3.3. Включить внешние JavaScript-файлы

Web Optimizer может загружать внешние JavaScript-файлы (вызываемые с других доменов), а также внутренний код (заключенный прямо в `<script>`). Здесь можно настроить и «склеивание» CSS-кода, находящегося в `<style>` (по умолчанию включено). Также можно указать (через пробел) названия файлов (названия, а не полные пути), которые нужно исключить из логики объединения. На этапе тестирования была обнаружена невозможность объединить исходные библиотеки Tiny MCE и FCE Editor, поэтому они исключаются по умолчанию.

7.3.4. «Ненавязчивый» JavaScript

В этой группе собраны настройки преобразования метода загрузки JavaScript. В частности, можно вынести объединенный JavaScript-файл перед `</body>` (или вообще вызывать его загрузку по событию `DomContentLoaded`), можно вынести загрузку некоторых счетчиков, рекламы и информеров также в самый низ документа (после JavaScript-кода результирующий HTML-код вставляется в исходное место на странице, обеспечивая постепенное появление дополнительных рекламных блоков после того, как загрузилось основное содержание).

7.3.5. Не проверять время изменения

Данная настройка позволяет не проверять при загрузке каждой страницы время изменения и содержание всех файлов, а только существование закэшированных версий. За счет этого мы получаем существенный прирост серверной производительности (по умолчанию настройка включена). В случае отладки или очень частого изменения исходных JavaScript- или CSS-файлов настройку лучше отключить.

7.3.5. Gzip-сжатие (архивирование)

Данная группа настроек регулирует, отдавать ли браузеру JavaScript-, CSS- или HTML-файлы в виде архивов. gzip-сжатие позволяет сэкономить 70-85% трафика при передаче текстовых файлов, однако может быть (особенно в случае сжатия через PHP и высоконагруженных проектов) не очень оптимальной для сервера. В любом случае по возможности все настройки сжатия выносятся в `.htaccess` (для CSS- и JavaScript- в статическом виде). При невозможности изменения `.htaccess` gzip-версии JavaScript- и CSS-кода сохраняются в кэширующих директориях, что также сводит нагрузку на процессор (через PHP) к минимуму.

Дополнительно есть возможность (которая выставляется через архивированный JavaScript-код) проверять через cookie, поддерживает ли клиентский браузер сжатие (как было указано во второй главе, иногда соответствующие заголовки могут пропадать в силу различных причин).

7.3.6. Клиентское кэширование

Настройки этой группы отвечают за выставление кэширующих заголовков для JavaScript-, CSS-, HTML- или статических файлов (изображений и анимации). Для изображений и анимации соответствующие правила размещаются только в `.htaccess`, для остальных файлов они дублируются по необходимости через PHP. По умолчанию для статических файлов выставляется срок кэширования на 10 лет (при изменении файлов новые версии имеют другое имя, создаваемое на основе md5-хэша от общего содержимого файлов).

Для HTML-файлов есть возможность вручную выставить подходящий срок действия клиентского кэша. Отличие этой настройки от следующей группы (серверного кэширования HTML-файлов) состоит в том, что выводимый HTML никак на сервере не сохраняется, мы только указываем бра-

узерам, что они могут не перезапрашивать HTML-документы в течение определенного времени. Будут ли браузеры следовать этому указания или нет, остается полностью на их совести.

Дополнительно существует возможность задать кэширующие заголовки для всех статических файлов (как на уровне файла `.htaccess`, так и через проксирование запросов при помощи PHP).

7.3.7. Серверное кэширование

Для существенного ускорения работы серверной стороны практически во всех случаях требуется применять серверное кэширование. И практически все CMS это поддерживают (на том или ином уровне). Web Optimizer предлагает альтернативный вариант (для тех случаев, когда текущая система этого не умеет либо требуется более «жесткое» решение): простое кэширование HTML-документов. При включении этой настройки HTML-файлы сохраняются в директории кэширования и отдаются при первом вызове Web Optimizer, без обработки внутренней логики системы. Естественно, учитывается срок действия кэша.

Также возможно выдавать сразу не весь документ, а первые 1-2 Кб (через сброс документа), и потом рассчитывать остальную часть. Это может помочь визуально ускорить загрузку страницы на некоторых окружениях. Для настройки кэширования доступен список частей URL сайта, которые нужно исключить (есть возможность задавать регулярные выражения), и список роботов (USER AGENTS), для которых нужно форсировать выдачу кэширующего файла.

7.3.8. CSS Sprites

Это, пожалуй, самая технологически мощная и самая спорная часть Web Optimizer. Правильное использование спрайтов позволяет на порядок (!) уменьшить число запросов к серверу при загрузке страницы с большим количеством фоновых изображений (с 20-100 до 3-10). Однако существуют некоторые проблемы с отображением комбинированных картинок для IE 6 (картинки по умолчанию создаются в 32-битной палитре, а IE 6 не умеет корректно обрабатывать прозрачность для таких PNG), и проблемы эти устраняются исключением IE 6 из создания спрайтов (соответствующей настройкой) либо использованием непрозрачных картинок.

Также доступны настройки по использованию JPEG вместо PNG для полноцветных изображений, по «агрессивному» режиму (`repeat-x` и `repeat-y` будут объединяться без учета фактических размеров кон-

тейнеров), добавлению свободного пространства (позволяет избежать рудиментов при масштабировании таких картинок в современных браузерах).

Для повышения стабильности работы добавлен режим «ограниченной» памяти: если у PHP-процесса меньше 64 Мб памяти (этого хватает для создания спрайта примерно 3000 на 3000 пикселей, что вполне достаточно для большинства сайтов), то изображения, по площади большие 4000 пикселей, будут исключены. Также есть настройка по исключению больших изображений по их линейным размерам (в пикселях, по умолчанию 900) и прямому исключению файлов (опять-таки задаются имена файлов, а не полный путь к ним) из процесса создания CSS Sprites.

7.3.9. data:URI

Технология `data:URI` позволяет включать фоновые изображения прямо в CSS-файл. Поддерживается всеми современными браузерами и IE 8. Имеет опциональное ограничение на размер изображения в 24 Кб (32 Кб `data:URI` кода получаются из 24576 байтов бинарного кода). При создании `data:URI` для IE 7 в CSS-файл вставляются хаки, благодаря чему дизайн сайта остается неизменным для этих браузеров.

Также при создании `data:URI` крайне желательно оптимизировать изображения. Для этого используется API от smush.it. Для корректной оптимизации изображений нужны права на запись для веб-сервера на сами изображения. Эта настройка по умолчанию отключена, потому что оптимизировать изображения имеет смысл всего один раз и при последующих сборках CSS-файлов использовать уже готовый результат.

Для гибкого управления настройками данной группы также доступно конфигурируемое исключение ряда файлов и ограничение включения файлов по размеру.

7.3.10. Множественные хосты

Данная настройка позволяет включить распределение изображений по статическим хостам. Каждому изображению всегда будет соответствовать один хост (чтобы избежать забивания кэша одними и теми же изображениями со всех доступных хостов). Для использования данного механизма ускорения загрузки необходимо прописать в DNS все альтернативные хосты на тот IP-адрес, который будет их обслуживать (обычно это текущий сайт) и добавить в конфигурации сервера алиасы для основного сайта в виде этих хостов. Например:

```
ServerAlias i1.site.ru
ServerAlias i2.site.ru
```

После этого можно добавить i1 i2 в список хостов для Web Optimizer и убедиться в том, что изображения «раскидываются» по этим хостам. При установке Web Optimizer автоматически проверяется ряд хостов на возможность их использования в качестве альтернативных (с тем же корнем сайта), также все вручную указанные хосты проверяются на доступность (с них загружаются тестовые картинки). Стоит иметь в виду, что при включении «безопасной» установки (о ней чуть ниже) проверка хостов становится недоступной и их прописывать нужно будет вручную, не перезапуская настройку Web Optimizer.

7.3.11. Использование .htaccess

Большая часть настроек gzip-сжатия и кэширования может быть записана в конфигурационном файле вашего сервера для избежания дополнительной работы на стороне серверных скриптов. Это можно проделать с помощью файла `.htaccess` (при необходимости вы сможете впоследствии самостоятельно перенести все настройки в файл `httpd.cond`). Web Optimizer автоматически проверяет доступные модули и конфигурирует запись в `.htaccess` (естественно, для этого последний должен быть доступен на запись), `mod_gzip`, `mod_deflate` и `mod_filter` отвечают за сжатие файлов «на лету», `mod_rewrite` и `mod_mime` — за статическое архивирование, `mod_headers` и `mod_setenvif` — за обеспечение корректной обработки сжатых файлов на проксирующих серверах и в старых браузерах, `mod_expires` — за выставление кэширующих заголовков.

Также возможно расположить `.htaccess` либо в директории сайта (это бывает полезно, если на одном хосте располагается несколько сайтов в разных дирекориях), либо в самом корне сайта. По умолчанию оба месторасположения совпадают. Возможно и защитить установку Web Optimizer с помощью `.htpasswd`. В этом случае для доступа к настройкам нужно будет ввести логин и пароль через окно HTTP Basic Authorization в браузере (это позволяет вынести Web Optimizer в произвольную директорию внутри сайта, предварительно расположив директории кэширования вне защищенной области).

В случае тестирования сайта на закрытом от внешнего доступа хосте (закрытом при помощи HTTP Basic Authorization) есть возможность задать логин и пароль доступа в конфигурации Web Optimizer, чтобы последний имел возможность загрузить и обработать динамические файлы.

7.3.12. Логотип Web Optimizer

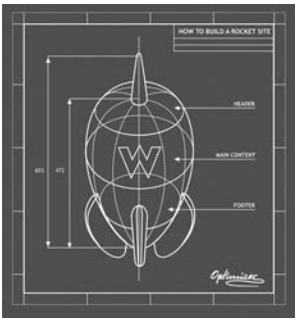
На данный момент при включении этой группы настроек на сайт (в правый нижний угол) добавляется небольшая печать «Ускорено при помощи Веб Оптимизатора» со ссылкой на проект.

7.3.13. Автоматическое изменение /index.php

Web Optimizer поддерживает автоматическое изменение необходимых для корректной работы файлов для нескольких десятков CMS (в случае неизвестной системы ее название выводится как CMS 42 и изменяется всегда корневой `index.php`). Перед автоматическим изменением файла запускается цепочная оптимизация сайта, чтобы создать все кэширующие файлы и избежать длительной загрузки главной страницы сайта в первый раз.

7.4. Примеры использования Web Optimizer

Web Optimizer — достаточно мощный и гибкий инструмент для клиентской оптимизации произвольного сайта. Но правильная его настройка требует внимательности и понимания желаемого результата. Давайте рассмотрим некоторые наиболее характерные изменения в конфигурации по умолчанию.



7.4.1. Подключение общего(-их) JavaScript- или CSS-файла(-ов) на всех страницах

Достаточно часто мы можем вставить на все страницы какую-то общую JavaScript-библиотеку или таблицу стилей, нужную для всего сайта. Для того чтобы Web Optimizer не пытался объединить этот общий файл со всеми остальными, можно исключить его в настройках:

```
Включить внешние файлы -> Исключить из объединения файлы ->
список файлов через пробел
```

7.4.2. Отключение CSS Sprites

Зафиксированы случаи, когда в силу ряда причин (отсутствия корректной поддержки GDLib на хостинге, ограничениях по памяти и т. д.) попытка создания CSS Sprites приводит к «белому экрану» в браузере (и сайт перестает открываться). В таких случаях обычно помогает полное отключение CSS Sprites:

```
CSS Sprites -> Применить CSS Sprites -> нет
```

При желании CSS Sprites можно создать вручную через Auto Sprites (http://sprites.in/), указав минимизированный CSS-файл (или объединенный исходный).

7.4.3. Некорректное отображение сайта в IE 6

В этом случае может помочь простое отключение спрайтов для IE 6:

```
CSS Sprites -> Исключить IE 6 (через хаки) из процесса создания
CSS Sprites -> да
```

Также возможно, что IE 6 некорректно обрабатывает объединение стилей (и наложение фона вместе с рядом других хаков). Тогда может помочь отключение data:URI:

```
Data:URI -> Применить data:URI -> нет
```

или вообще объединения CSS-файлов:

```
Настройки сжатия -> Минимизировать и объединить CSS-файлы -> нет
```

7.4.4. Безопасная установка

Web Optimizer может быть установлен в произвольную директорию (внутри сайта) и защищен с помощью пароля через htpasswd. Для этого нужно включить:

```
Использование .htaccess -> Защитить установку Веб Оптимизатора с
помощью htpasswd -> да
```

При этом нужно убедиться, что директории кэширования расположены вне папки с самим Web Optimizer (иначе все развалится для всех посетителей сайта, кроме вас самих).

7.4.5. Подключение для статического сайта

Web Optimizer может быть использован для любого сайта, для которого доступен PHP. Если у вас уже есть статический сайт, то вы можете подключить вызов Web Optimizer в самом верху каждой страницы (или `index.php`):

```
<?php
require(/путь/к/Web/Optimizer/
web.optimizer.php);
?>
и затем в самом низу страницы:
<?php
$web_optimizer->finish();
?>
```

Поскольку в плановом режиме (после создания всех кэширующих файлов) выполнение логики Web Optimizer занимает 5-10 мс, на серверную сторону загрузки это не повлияет, зато клиентская будет существенно ускорена. Только надо убедиться, что те страницы, на которые вы добавите вызовы Web Optimizer, будут обрабатываться через PHP-интерпретатор (будут иметь расширение .php или какое-либо другое, определяемое настройками сервера).

7.4.6. Оптимизация по расписанию

Уже сейчас Web Optimizer может быть встроен в схему публикации произвольного сайта в «статическом» режиме. Для этого необходимо открыть все страницы сайта с установленным Web Optimizer, а потом просто скопировать выведенный HTML и кэширующие директории. Предположим, что Web Optimizer установлен на dev.site.ru. Запустив, например, `wget`:

```
wget -d -r -c http://dev.site.ru/
```

мы получим оптимизированный «слепок» сайта, который можно загрузить уже в рабочую систему.

7.4.7. «Склейка» HTML в одну строку

Как уже было описано выше, Web Optimizer может «слеить» весь выводимый HTML в одну строку. Для этого нужно включить

```
Настройки сжатия -> Сжать HTML до 1 строки -> да
Настройки сжатия -> Удалить HTML-комментарии -> да
```

Стоит сразу отметить, что данные настройки создают дополнительную нагрузку для сервера (корректное регулярное выражение достаточно ресурсоемко) и могут привести к вырезанию некоторого JavaScript-кода (который вставляется через комментарии). Заметим также, что код внутри `<script>`, `<textarea>`, `<pre>` изменяться не будет (в соответствии со спецификацией), поэтому использовать данные настройки нужно с большой осторожностью.

7.4.8. Оптимизация изображений через smush.it

Сервис smush.it разработан инженерами Yahoo! и Google и позволяет оптимизировать размер фоновых изображений в автоматическом режиме. Подключить оптимизацию изображений можно через библиотеку CSS Sprites:

```
<?php
require('/полный/путь/до/css.sprites.php');
$smushit = new css_sprites();
$smushit->smushit('/полный/путь/до/изображения');
?>
```

В результате вместо изображения (при наличии прав на его изменение) мы получим его оптимизированную копию. Процедуру лучше проводить не на группе рабочих изображений, а на их копиях, чтобы была возможность откатить изменения.

7.5. Решаем проблемы с установкой Web Optimizer

После многочисленных установок Web Optimizer (http://www.web-optimizer.ru/) на Joomla! (http://joomla.org/, как версии 1.0, так и 1.5) было

решено собрать воедино полезное знание о возникающих проблемах (связанных в основном с текущим некорректным серверным окружением), чтобы позволить их самостоятельно решить большому числу пользователей.

Итак, давайте разбираться по порядку, что нам делать в следующих случаях.

7.5.1. Функционирование Web Optimizer

Довольно часто приходится разбираться с тем, работает ли Web Optimizer на сайте или его установка каким-то образом не подключилась к обработке HTML-документа. Начиная с версии 0.5.2 это можно установить, найдя строку `<title lang="wo">` в коде страницы (если метка там отсутствует, значит, Web Optimizer не отрабатывает).

Для более ранних версий это решение можно принять на основе отсутствия в коде HTML-документа отступов в начале строки, двойных переводов строк или наличия характерных закэшированных имен файлов в секции `head` (`cache/1234a6789b.css` или `cache/1234c6789d.js`, здесь `1234c6789d` — произвольная строка в шестнадцатеричной записи).

Если обнаружить следы работы Web Optimizer не удалось, то необходимо перепроверить корректность вызовов Web Optimizer в файлах системы управления сайтом и, возможно, провести установку приложения еще раз — таким образом Web Optimizer сможет самостоятельно произвести все необходимые изменения.

Если нужно получить информацию о необходимых изменениях исходных файлов CMS, то следует в ходе установки отключить автоматическое изменение `/index.php`:

```
Изменение /index.php → Включить автозапись -> Нет
```

и на последнем шаге зайти на вкладку «Необходимые изменения».

7.5.2. «Кракозябры» вместо сайта

Обычно это связано с двойным сжатием HTML-документа. Одно из сжатий может быть наложено самим Web Optimizer, а второе — как используемой системой управления сайтом, так и сервером.

Чтобы снять одно из накладываемых сжатий, можно отключить его либо в CMS, либо в конфигурации сервера, либо в самом Web Optimizer:

```
Настройки архивирования → Применить gzip для HTML → Нет
```

7.5.3. Белый экран вместо сайта

Основная проблема возникновения белого экрана (превышение лимита памяти при создании CSS Sprites) уже устранена в версии 0.5+, поэтому если у вас более старая версия, то стоит просто ее обновить.

Также стоит заглянуть в логи ошибок вашего сервера, чтобы узнать, что привело к такому состоянию. Обычно это помогает решить проблему.

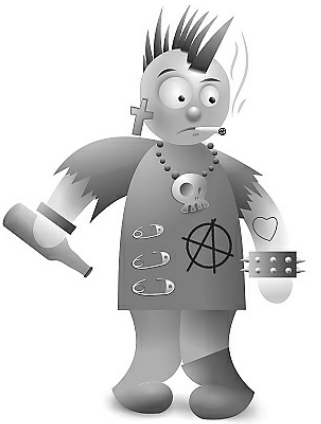
В некоторых случаях белый экран возникает из-за некорректной установки приложения либо двойного сжатия. Как решить эти проблемы, описано чуть выше.

Если все приведенные шаги не принесли результата, то можно попробовать отключить часть настроек Web Optimizer, чтобы понять, какие вещи ваш сервер может выполнить самостоятельно. Начать стоит с корневого набора («Настройки сжатия» и «Настройки архивирования») для всех трех групп (CSS, JavaScript, HTML) действий по оптимизации и двигаться в сторону подключения более детальных параметров (например, «Включить внешние файлы», ««Вечное» кэширование» или «CSS Sprites»).

7.5.4. Некорректный внешний вид

Это может быть причиной различных проблем, но для начала можно попробовать отключить CSS Sprites:

```
CSS Sprites → Применить CSS
Sprites → Нет
```



затем, если это не помогло ситуации, можно отключить `data:URI`:

```
Data:URI → Применить data:URI
→ Нет
```

После этих шагов все подключаемые CSS-файлы не будут обрабатываться через CSS Tidy, а будут только объединяться (и будет выполняться первичная минимизация).

Если и после этого внешний вид сайта «разъехался», то стоит отключить объединение стилей внутри `<head>`:

```
Включить внешние файлы → Включить объединение внешних
CSS-файлов → Нет
```

или вообще минимизацию для CSS-файлов:

```
Настройки сжатия → Минимизировать и объединить CSS-файлы → Нет
```

Если есть желание разобраться в возникшей проблеме более детально, то можно при отключенном объединении стилей внутри `<head>` попробовать исключить файлы один за другим при помощи следующей настройки:

```
Включить внешние файлы → Исключить из объединения файлы →
Список файлов через пробел
```

найти тот файл (или те файлы), которые обрабатываются некорректно, привести их к стандартному состоянию при помощи валидатора jigsaw.w3.org/css-validator/ и попробовать объединить снова.

7.5.5. Пропавшие или неправильные фоновые изображения

В некоторых случаях проблемы после установки Web Optimizer сводятся к тому, что некоторые фоновые изображения пропадают или «портятся». Для устранения этого набора проблем необходимо установить, с какими первоначальными фоновыми изображениями возникают трудности (используя любые средства для отладки верстки), и исключить эти изображения из процесса создания CSS Sprites:

```
CSS Sprites → Исключить из CSS Sprites файлы → Список файлов
через пробел
```

Если данная мера не приносит результата, то CSS Sprites можно вообще выключить:

```
CSS Sprites → Применить CSS Sprites → Нет
```

7.5.6. Некорректное поведение сайта

Обычно это сводится к тому, что часть клиентской логики перестает отрабатывать. Нужно хорошо понимать, что если, например, форма для ввода комментариев у вас на сайте выводится при помощи JavaScript и после Web Optimizer она перестала выводиться, то это проблемы не внешнего вида, а клиентской логики.

Для локализации проблемы с клиентской логикой рекомендуется попробовать отключить минимизацию JavaScript-файлов:

```
Настройки сжатия → Минимизировать и объединить JavaScript-файлы →
Нет
```

а потом (при сохранении проблемы) попробовать исключить один за другим отдельные файлы:

```
Включить внешние файлы → Исключить из объединения файлы →
Список файлов через пробел
```

При установлении имени файла, на котором объединение отказывает, можно просто его исключить из общего пакета либо попытаться настроить его корректное включение в первоначальную логику (обычно ошибки происходят из-за некорректного синтаксиса исходных методов и библиотек, которые не конфликтуют внутри одного окружения-файла, но начинают конфликтовать при объединении этих окружений).

7.5.7. Недоступность файлов для пользователей

Иногда возникают проблемы с отображением и функционированием сайта у всех пользователей, хотя у владельца сайта (в его браузере) при этом все хорошо. Если диагноз проблемы звучит подобным образом, то нужно проверить, по какому адресу находятся директории кэширования, сам Web Optimizer и включена ли защита Web Optimizer от внешнего доступа:

```
Использование .htaccess → Защитить установку Веб Оптимизатора с
помощью htpasswd → Да
```

В этом случае директории кэширования нужно вынести из папки самого Web Optimizer (например, в корневую директорию cache, до-

ступную на запись для веб-сервера) либо отключить защиту приложения:

```
Использование .htaccess → Защитить установку Веб Оптимизатора с помощью htpasswd → Нет
```

7.5.8. Отсутствие поддержки множественных хостов

Если требуется включить несколько параллельных хостов для ускорения загрузки статических ресурсов, то стоит выполнить следующие действия:

1. Проверить наличие поддержки этих хостов в DNS. Для этого нужно сделать соответствующие этим хостам записи в вашей DNS-зоне, указывающие на требуемый IP-адрес (обычно тот же, что и у текущего сайта).
2. Включить поддержку этих хостов на уровне самого сервера. Для Apache это делается директивой `ServerAlias`, например:

   ```
   ServerAlias i1.site.ru
   ServerAlias i2.site.ru
   ```

3. Проверить, что эти хосты зеркалируют основной сайт. Для этого нужно взять адрес любого статического объекта на сайте (например, `site.ru/images/my.png`) и попробовать его открыть через все дополнительные хосты (`i1.site.ru/images/my.png`). При наличии каких-либо проблем необходимо повторить предыдущие шаги.
4. Добавить указанные хосты в конфигурацию Web Optimizer:

   ```
   Множественные хосты → Доступные хосты → Название хостов через пробел
   ```

   и включить поддержку множественных хостов:

   ```
   Множественные хосты → Включить параллельные хосты, например, i1 i2 → Да
   ```

   Web Optimizer автоматически проверяет несколько наиболее распространенных хостов, и последняя настройка может и не потребоваться.

После описанных действий все изображения на сайте будут отдаваться через несколько хостов, что существенно ускорит загрузку каждой страницы.

7.5.9. Множественные хосты не сохраняются

Web Optimizer автоматически проверяет все введенные хосты на доступность, чтобы быть уверенным, что их можно использовать для параллельных загрузок. Вы можете отключить данную проверку (если в силу каких-либо причин она производится некорректно):

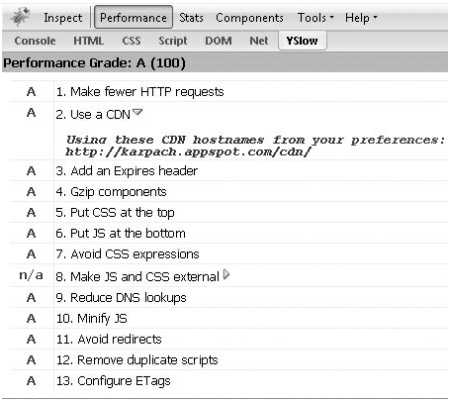
```
Множественные хосты → Автоматически проверять доступность
хостов → Нет
```

Глава 8. Практическое приложение

8.1. Разгоняем ASP .NET: 100 баллов и оценка «A» в YSlow

Этот раздел написан под впечатлением статьи Viktar Karpach YSlow and ASP.NET: 100 points «A» grade is possible (http://www.karpach.com/ yslow-and-asp-net-100-points-a-grade.htm). Здесь мы разберем по шагам, как максимально ускорить работу вашего сайта на ASP .NET. Далее приводятся пункты рекомендаций советов от Yahoo!, с которыми могут возникнуть сложности.

Рис. 8.1. Измерения скорости загрузки сайта при помощи YSlow.
Источник: www.karpach.com



8.1.1. Меньше HTTP-запросов

Во-первых, нужно объединить в один файл таблицы стилей и файлы скриптов. Это стоит сделать только для рабочего (не тестового) выпуска сайта при помощи MS Build:

```
<ItemGroup>
  <TextFiles Include="*.css" Exclude="global.css"/>
</ItemGroup>
<Exec Command="echo y| type %(TextFiles.Identity) >>
global.css"/>
```

Таким образом мы можем объединить все CSS- и JS-файлы.

Некоторые разработчики зададут резонный вопрос: а что можно сказать по поводу WebResource.axd? В новом AJAX Control Toolkit есть ToolkitScriptManager, который позволяет объединить большинство ваших файлов WebResource.axd. Также он может применять к ним gzip-сжатие (это будет важно для следующих разделов).

Затем нам нужно создать CSS Sprites. Более подробно они уже были описаны в четвертой главе. Идея заключается в том, чтобы группировать такие изображения по оси повторения. Например, все изображения, повторяющиеся по вертикали, сложить в vbackground.png, а все повторяющиеся по горизонтали — в hbackground.png. Затем можно использовать background-position: -OffsetPixels 0 для позиционирования внутри вертикального файла и background-position: 0 -OffsetPixels — для горизонтального.

8.1.2. Используем CDN (Content Delivery Networks)

Сети доставки содержания (*англ. CDN, Content Delivery Network*) стоят довольно дорого. Но можно поступить проще и использовать для этого Google (подробнее рассказывается в шестой главе). Естественно, придется немного попотеть, прежде чем собрать из Google App Engine свою идеальную CDN.

Можно также применить `msbuild` для изменения файлов стилей при использовании CDN. Ниже приведен рабочий пример (в нем пути с `../images/` заменяются на `http://karpach.appspot.com/cdn/images/`):

```
<Import Project=".\References\MSBuild.Community.Tasks.targets" />
<Target Name="Release">
```

```
  <FileUpdate Files="$(OutputPath)styles\basic.css"
  Regex="\.\.\/images/([^\)]*)" ReplacementText=
  "http://karpach.appspot.com/cdn/images/$1" />
</Target>
```

Теперь нужно положить в CDN все ваши изображения, файлы стилей и скриптов. Здесь как раз и всплывает проблема сброса кэша. Что будет, если пользователь зашел к вам на сайт как раз перед тем, как вы выложили новую версию, в которой изменился файл стилей и некоторые изображения? По-видимому, этот пользователь будет видеть сайт со старыми стилями и картинками еще 7 дней, и он просто не обнаружит изменений, пока не истечет срок действия кэша в его браузере. Это не есть хорошо. Если озвученный вопрос становится актуальным, то стоит посмотреть, что Yahoo! сделали на своем веб-сайте. В название картинки они «зашивают» метку даты и версию. Например, `trough_2.0_062308.gif`. Это выглядит разумно. Если вы изменяете какой-либо из ваших CSS Sprites, то его нужно вручную переименовать при помощи новой даты и версии (на самом деле процесс переименования файлов должен быть заложен в сам инструмент создания CSS Sprites; подробнее о сбросе кэша и объединении файлов было рассказано в главе 4).

Однако, по всей видимости, этот подход может быть неприменим для файлов стилей. Для начала стоит обеспечить контроль изменений версий таблиц стилей (в данном случае это SVN). Для картинок это не играет существенной роли, потому что они меняются не так часто. Основная идея заключается в том, что у нас может быть один физический файл (например, `basic.css`), а на странице используется адрес другого файла (например, `basic_14102262.css`), где `14102262` — это текущая версия приложения. Затем мы можем применить технологию url rewrite, так что `basic_14102262.css` будет указывать на `basic.css`. Пользователь с закэшированной версией `basic_14102261.css` будет вынужден загрузить новую версию `basic_14102262.css`, поскольку изменилось имя файла.

Движок Google Apps позволяет легко реализовывать rewrite для URL. Для этого можно просто изменить ваш `app.yaml` следующим образом:

```
- url: /cdn/styles/basic_\d*\.css
  static_files: cdn/styles/basic.css
  upload: cdn/styles/basic\.css
```

Более подробно о настройке CDN от Google рассказывалось в предыдущей главе.

8.1.3. Добавляем заголовок Expires

У всех файлов, расположенных в CDN, уже выставлен этот заголовок. Для всех остальных можно создать следующий `HttpModule`:

```
private readonly static string[] CACHED_FILE_TYPES = new
string[] { ".jpg", ".gif", ".png",".css" };

public void Init(HttpApplication context)
{
  context.AcquireRequestState += new
EventHandler(context_AcquireRequestState);
}

void context_AcquireRequestState(object sender, EventArgs e)
{
  HttpContext context = HttpContext.Current;
  if (context != null && context.Response != null)
  {
    string fileExtension = Path.GetExtension
    (context.Request.PhysicalPath).ToLower();
    if (context.Response.Cache != null &&
    Array.BinarySearch<string>(CACHED_FILE_TYPES, fileExtension)
    >= 0)
    {
      HttpCachePolicy cache = context.Response.Cache;
      TimeSpan duration = TimeSpan.FromDays(365);
      cache.SetCacheability(HttpCacheability.Public);
      cache.SetExpires(DateTime.Now.Add(duration));
      cache.SetValidUntilExpires(true);
      cache.SetNoServerCaching();
      cache.SetMaxAge(duration);
    }
  }
}
```

В принципе, у вас вообще не должно быть этого модуля с учетом того, что все статические файлы должны быть расположены в CDN.

8.1.4. Располагаем CSS-файлы в начале страницы

Это делается очень просто. Просто надо ввести эти действия в привычку. Помните об этом при разработке модулей на стороне сервера и ис-

пользуйте коллекцию `Header.Controls` для добавления таблиц стилей на страницу.

8.1.5. Располагаем JavaScript-файлы в конце страницы

Следовать этому правилу бывает очень сложно. Например, иногда хочется добавить jQuery в самом начале, чтобы потом не волноваться на тему того, загрузилась ли эта библиотека. В любом случае, для всех ваших скриптов будет хорошей практикой проверять, загрузилась ли основная библиотека, прежде чем выполнять какие-либо действия.

Например, файл

```
Library.js
```

может содержать

```
function DoSomething()
{

}
А после этого прямо в коде страницы:
<script type='text/javascript'>
  if (typeof (DoSomething) == 'undefined')
  {
    alert('Library is not loaded yet');
  }
</script>
```

Приведенный пример будет недостаточно верным, потому что загружаемая функция может быть доступна на странице после некоторого времени. Лучше всего создавать дополнительную клиентскую логику, использующую определенный модуль, в самом этом модуле либо проверять через равные промежутки времени, доступен ли необходимый функционал. Более подробно данные техники были освещены в главе «Ненавязчивый JavaScript» книги «Разгони свой сайт».

8.1.6. Уменьшаем число DNS-запросов

Для этого стоит скопировать внешние ресурсы (картинки, JavaScript-файлы) к себе на сайт.

Например, на блоге располагается иконка валидной страницы от W3. Изначально эта иконка находилась на сайте W3 Schools, но для ускорения загрузки сайта стоит скопировать ее в свой проект. Тогда она будет располагаться локально, а при загрузке сайта понадобится на 1 DNS-запрос меньше.

8.1.7. Уменьшаем JavaScript

Минимизацию JS-кода можно осуществлять при публикации очередного выпуска сайта при помощи MS Build (это наиболее разумная позиция — осуществлять все оптимизационные процедуры при превращении сайта из тестового в рабочий). Для этой цели можно использовать YUI compressor. Ниже приведен вариант скрипта для MS Build.

```
<Target Name="Compress">
  <Message Text="Create temp files ..." />
  <Copy SourceFiles="..\$(ProjectName)\Javascript\ColorPicker.js"
DestinationFiles="..\$(ProjectName)\Javascript\ColorPicker.js.full
"/>
  <Copy SourceFiles="..\$(ProjectName)\Styles\ColorPicker.css"
DestinationFiles="..\$(ProjectName)\Styles\ColorPicker.css.full"/>
  <Exec Command="java -jar yuicompressor-2.4.2.jar –type js
..\$(ProjectName)\Javascript\ColorPicker.js.full
>..\$(ProjectName)\Javascript\ColorPicker.js"/>
  <Exec Command="java -jar yuicompressor-2.4.2.jar –type css
..\$(ProjectName)\Styles\ColorPicker.css.full
>..\$(ProjectName)\Styles\ColorPicker.css"/>
</Target>
```

8.1.8. Удаляем дублирующиеся скрипты

Аккуратно подходите к разработке своего сервера и пользовательских расширений. Проверяйте, что несколько различных расширений загружают общие библиотеки только один раз.

8.1.9. Настройте ETag

Кэширующий модуль и CDN должны решить эту проблему. Но стоит также иметь в виду, что в ASP .NET есть специальный метод для `ETag`:

```
Response.Cache.SetETag
```

8.2. Разгоняем Drupal

Автор данного раздела, Елена Цаплина (aka Касихина), — программист-разработчик и руководитель нескольких Интернет-проектов: студия дизайна и разработки Интернет-сайтов Aquanther (http://www.aquanther.ru/), ежедневный женский журнал «Мои подружки» (http://www.moipodruzhki.ru/), программное обеспечение (под управлением ОС Windows) под единым названием — HomAff (сокращение от home affairs).

Елена известна своей публикацией о CMS Drupal — вводный курс по Drupal (http://drupal.ru/node/26290), получившей широкое распространение в Рунете.

В данный момент под ее руководством (на базе студии Aquanther) разрабатывается мультимедийный курс по созданию и управлению сайтом с использованием CMS Drupal. В дальнейшем вместе с мультимедийным курсом планируется распространение модифицированных версий CMS Drupal, специализированных для определенных типов сайтов, и оказание платной поддержки данной CMS.

8.2.1. Вступление

Drupal — довольная распространенная CMS, и это наложило на нее свой отпечаток: базовая поставка Drupal является не готовым решением для определенного вида сайта, а фундаментом для его создания. Существуют «сборки» на базе Drupal, специализированные под определенные виды сайтов, например, под новостные сайты. Но подобные сборки в данный момент мало распространены и плохо поддерживаются. В связи с этим при создании Интернет-сайта на основе стандартной поставки Drupal используется большое количество готовых дополнительных модулей и тем оформления для Drupal либо разрабатываются новые модули и темы специально для данного проекта. Последним этапом работ по созданию сайта является его оптимизация, которую условно можно разбить на 4 шага:

- встроенная оптимизация Drupal;
- оптимизация Drupal с помощью модулей;
- оптимизация конфигурации и обслуживания Drupal;
- оптимизация сервера.

8.2.2. Встроенная оптимизация Drupal

1. Отключим все неиспользуемые модули, так как при генерации страницы перед отправкой ее браузеру пользователя код определенных модулей может выполняться, даже если функционал данного модуля не используется на сайте. На выполнение кода будет тратиться процессорное время сервера, что приведет к более долгой генерации страницы. Пример такого модуля — Statistics. Вместо статистики, выдаваемой данным модулем, можно взять статистику сервиса — Google Analytics.

   При создании сайта используем Drupal версии 6, так как в нем лучше реализованы внутренние средства кэширования. Также в дополнительных модулях (Views, Panel и т. д.) для Drupal версии 6 внедрены эффективные методы кэширования. К сожалению не все модули Drupal версии 5 реализованы для Drupal версии 6 (например, модуль Sphinx), о чем не следует забывать при планировании разработки Интернет-сайта. Далее будем рассматривать только Drupal версии 6.

   Хорошо обдумаем варианты использования модулей наподобие CCK (Content Construction Kit) перед реализацией запланированного. Например, простая задача на хранение в базе сайта тысячи типов продуктов, их названий и описаний, решается с помощью:
   - создания тысячи терминов таксономии и привязки их к определенному типу материала;
   - добавления определенному материалу дополнительного поля CCK, в котором будет храниться тип продуктов.

   При усложнении задачи с помощью условия, что все типы продуктов должны быть разбиты на 10 групп, задача решается двумя вариантами. Вариант первый, с использованием таксономии.
2. Таксономия позволяет создавать иерархию терминов в словаре, т.е. в словаре с терминами может быть 10 терминов первого уровня, а остальные термины будут потомками одного из этих 10 терминов. Создадим нужную иерархию терминов в словаре.
3. Установим дополнительный модуль — Hierarchical Select (http://drupal.org/project/hierarchical_select), который позволяет в зависимости от вложенности уровней словаря таксономии отображать определенное количество выпадающих списков. Иначе говоря, пользователю при добавлении новой статьи о продукте на сайт будет выведен один выпадающий список, в котором можно выбрать один из 10 терминов первого уровня (группы типов

продуктов). После осуществления выбора будет отображен еще один выпадающий список, в котором будут приведены потомки данного термина в иерархическом дереве терминов данного словаря таксономии (типы продуктов).

Вариант второй, с использованием CCK.

- Необходимо определенному материалу добавить 2 дополнительных поля CCK: одно — для хранения групп продуктов, второе — для хранения типов продуктов.
- Нужно настроить взаимодействие данных полей в зависимости от выбора значений в них.

Главные ошибки при решении подобных задач, создающие дополнительную нагрузку на базу данных:

- создание у определенного материала 10 полей CCK, по полю на каждую группу;
- при создании у определенного материала поля CCK не указана его длина (поэтому по умолчанию считается, что она максимально возможная).

4. Используем встроенное кэширование Drupal. Оно позволяет кэшировать информацию, извлеченную из базы данных, а также информацию, полученную при обработке извлеченной информации из PHP.

Кэширование системы меню, фильтров форматов ввода, переменных администрирования (например, название сайта) и настроек модуля производится автоматически. Остальные параметры кэширования можно настроить на странице «Управление → Производительность» (http://www.example.ru/admin/settings/performance).

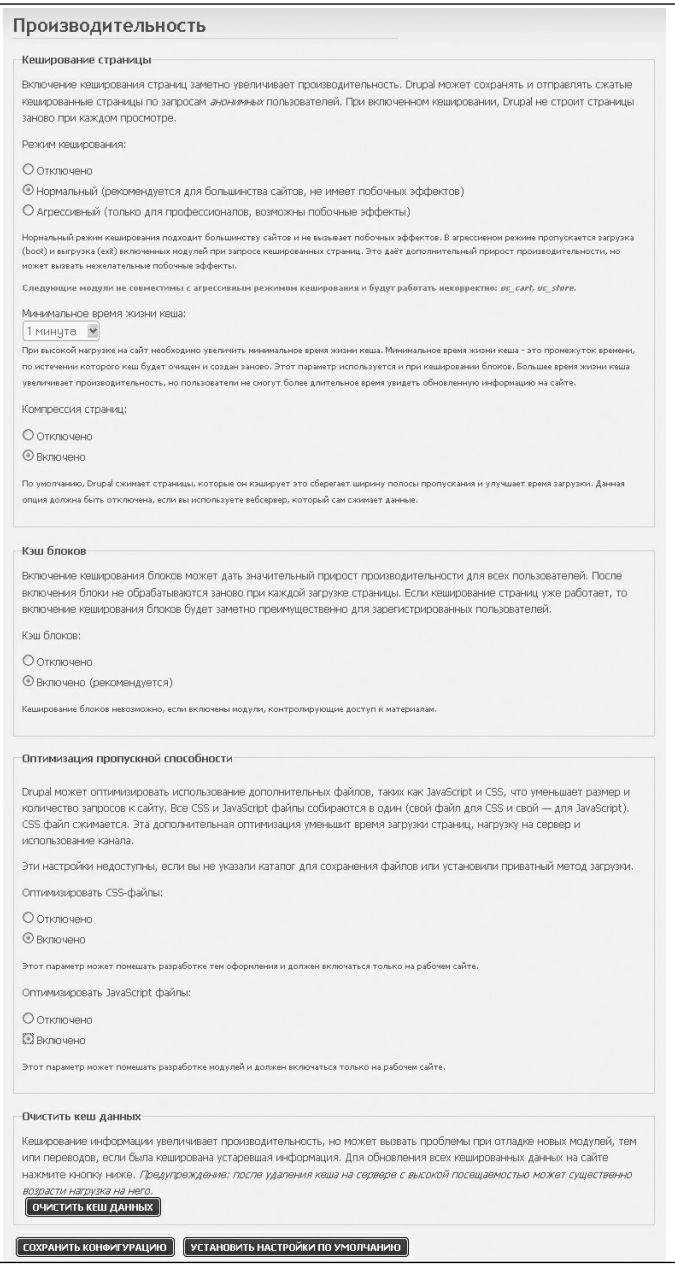
На данной странице можно настроить:

- компрессию страниц;
- кэширование блоков;
- оптимизацию CSS-файлов;
- оптимизацию JavaScript-файлов.

Включим кэширование страниц в режим «нормальный». В данном режиме кэширования страниц при просмотре страницы в первый раз (анонимным, незарегистрированным в системе пользователем) производится сохранение сгенерированной страницы в кэш. В дальнейшем при просмотре данной страницы (анонимным пользователем) она не генерируется заново, а берется из кэша, что значительно ускоряет работу Drupal.

Если кэширование страниц включено в режиме «агрессивный», при генерации страницы пропускается загрузка и выгрузка включенных модулей, поэтому часть модулей могут работать некорректно или не работать совсем.

Рис. 8.2. Настройки производительности для Drupal



Настроим минимальное время жизни кэша страниц для анонимных пользователей. Данный параметр определяет, через какое время после кэширования страницы производится проверка на то, обновлено ли содержимое данной страницы или нет. Если обновлено, то кэш данной страницы очищается. Т.е. если администратор сайта изменил содержимое страницы, он его увидит сразу, а анонимные пользователи — только по прошествии минимального времени жизни кэша.

Включим компрессию страниц для сохранения сжатого кэша страниц и для передачи страницы браузеру пользователя в сжатом виде, если он поддерживает компрессию gzip. Компрессия производится с помощью библиотеки zlib, установленной как расширение в PHP.

Включим кэширование блоков. Принцип работы кэширования блоков аналогичен принципу кэширования страниц. Для супер-пользователя (первого зарегистрированного пользователя при установке Drupal, его `id` равен 1) блоки никогда не кэшируются.

Включим оптимизацию CSS- и JavaScript-файлов. Это уменьшит их размер и количество обращений к серверу при загрузке страниц в браузер. Все CSS- и JavaScript-файлы собираются в один (свой файл для CSS (обычно их бывает два: один для отображения на экране, другой — для отображения при печати) и свой — для JavaScript). Таким образом мы уменьшим количество обращений к серверу при загрузке страницы.

8.2.3. Оптимизация Drupal с помощью модулей

1. Установим модуль Authenticated User Page Caching (Authcache), скачать его можно по адресу http://drupal.org/project/authcache. Данный модуль позволяет кэшировать страницы, как для анонимных пользователей, так и для аутентифицированных («залогинившихся») пользователей, более качественно, чем встроенное кэширование Drupal. При установке данного модуля необходимо перенастроить динамический контент на страницах (например, вывод имени аутентифицированного пользователя).

Authcache сохраняет сжатый кэш страниц отдельно для каждого пользователя или роли. Кэш сохраняется в базе данных или в стороннем средстве кэширования (memcahed, APC, и т. д.). Кэшированные версии страниц для аутентифицированных пользователей (кроме супер-пользователя) передаются с помощью AJAX, поэтому достигается очень быстрое отображение страницы в браузере. Если у аутентифицированного пользователя в браузере отключены JavaScript, то он получает страницы не из кэша. На некоторых серверах скорость загрузки страницы уменьшается до 1 миллисекунды.

Для установки модуля:

- скачаем его;
- распакуем модуль в папку /sites/all/modules;
- скопируем файл ajax_authcache.php из папки модуля в корневую директорию сайта (там же находится файл index.php);
- откроем файл settings.php (в папке /sites/default) и добавим следующий код (без примечаний за // …) в начало файла после тега <?php:

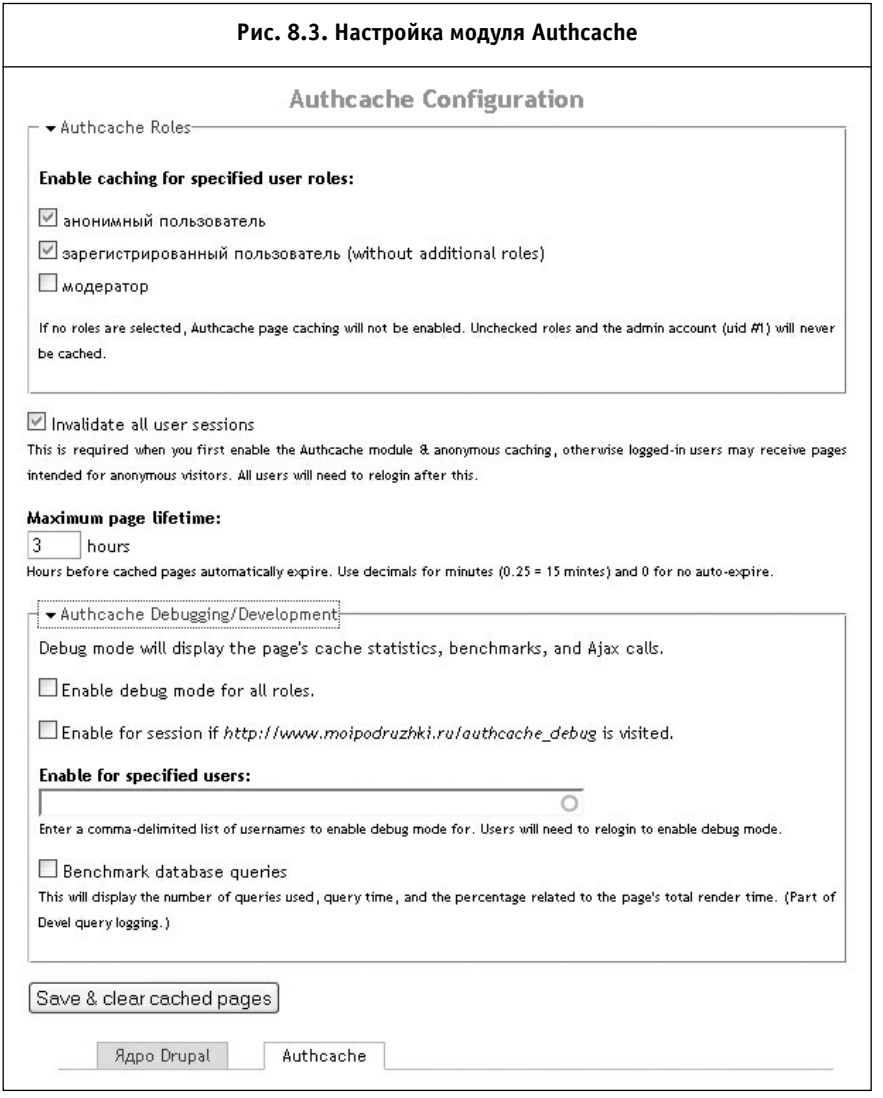
```
$conf['cache_inc'] =
'./sites/all/modules/authcache/api/authcache.inc';
$conf['authcache'] = array(
  'default' => array(
    // технология кэширования - apc, memcache, db, file,
    // eacc or xcache
    'engine' => 'db',
    // если используем memcached (host:port, например,
    // 'localhost:11211')
    'server' => array(),
    // если используем процесс memcached, shared или single
    'shared' => TRUE,
    // кэш ключа префикса (для нескольких сайтов)
    'prefix' => '',
    // если используем кэширование на файлах – указываем
    // путь их сохранения
    'path' => 'files/filecache',
    // статический массив кэша (расширенный)
    'static' => FALSE,
  ),
);
```

В данном коде устанавливаются настройки модуля Authcache. Указываем, что будем хранить кэш страниц в базе данных ('engine' => 'db'), поэтому все остальные установки не имеют значения, и мы оставляем их без изменений. Более подробно о параметрах данного кода можно прочитать на странице http://drupal.org/project/cacherouter (на английском языке).

Включим модуль Authcache на странице «Управление → Модули» (http://www.example.ru/admin/build/modules). После чего настроим его работу на странице «Управление → Производительность → Authcache» (http://www.example.ru/admin/settings/performance/authcache):

- укажем роли, для которых необходимо кэшировать контент;
- аннулируем все пользовательские сессии (переключатель — `Invalidate all user sessions`) при первом запуске;
- выставим время хранения кэша (в часах);
- нажмем кнопку «сохранить и очистить кэш» (`Save & clear cached pages`) для сохранения изменений.

Рис. 8.3. Настройка модуля Authcache



Изменим настройки элементов на страницах сайта так, чтобы страница для разных пользователей, принадлежащих одной роли, выглядела одинаково (т. е., например, запретим пользователям управлять видимостью блоков на сайте, если такие блоки существуют).

В шаблонах тем оформления используем переменные:

- `$user_name` — для отображения имени аутентифицированного пользователя;
- `$user_link` — для отображения ссылок, связанных с профилем пользователя;
- `$is_page_authcache` — если установлен в `TRUE`, то все хуки данного шаблона темы оформления будут сохранены в кэш.

Можно также ознакомиться с примером `/sites/all/modules/authcache/modules/authcache_example`, который показывает, как настроить блоки с пользовательским содержанием (с контентом пользователя).

2. Если необходимо кэшировать страницы только для анонимных пользователей (без аутентифицированных), можно установить модуль Cache Router. Данный модуль лежит в основе модуля Authcache и кэширует страницы лучше встроенного кэширования Drupal. Скачаем модуль по адресу http://drupal.org/project/authcache. После скачивания распакуем модуль в папку `/sites/all/modules`. Включим модуль Cache Router на странице «Управление → Модули» (http://www.example.ru/admin/build/modules). Откроем файл `settings.php` (в папке `/sites/default`) и добавим следующий код в начало файла перед тегом `<?php`:

```
$conf['cache_inc'] = './sites/all/modules/cacherouter/cacher-
outer.inc';
$conf['cacherouter'] = array(
  'default' => array(
    'engine' => 'db',
    'server' => array(),
    'shared' => TRUE,
    'prefix' => '',
    'path' => 'sites/default/files/filecache',
    'static' => FALSE,
    'fast_cache' => TRUE,
  ),
);
```

После осуществления действий, приведенных выше, страницы создаваемого сайта будут отдаваться сервером браузеру пользователя в сжатом виде, а вот CSS и JavaScript — нет. Исправим это:

- скачаем модуль CSS Gzip со страницы http://drupal.org/project/css_gzip;
- скачаем модуль JavaScript Aggregator со страницы http://drupal.org/project/javascript_aggregator;
- распакуем модули в папку `/sites/all/modules`;
- включим модули на странице «Управление → Модули» (http://www.example.ru/admin/build/modules);
- активируем сжатие CSS и JavaScript на странице «Управление → Производительность» (http://www.example.ru/admin/settings/performance), отметив чекбоксы «GZip CSS» и «GZip JavaScript»;
- внесем изменение в файл `.htaccess`, расположенный в корневой директории сайта (на основании данных из `README.txt`, входящего в состав модуля CSS Gzip), пропишем меду тегами `<IfModule mod_rewrite.c>` и `</IfModule>` следующий код:

```
### START CSS GZIP ###
# Requires mod_mime to be enabled.
<IfModule mod_mime.c>
  # Send any files ending in .gz with x-gzip encoding
  # in the header.
  AddEncoding x-gzip .gz
</IfModule>
# Gzip compressed css files are of the type 'text/css'.
<FilesMatch "\.css\.gz$">
  ForceType text/css
</FilesMatch>
<IfModule mod_rewrite.c>
  RewriteEngine on
  # Serve gzip compressed css files
  RewriteCond %{HTTP:Accept-encoding} gzip
  RewriteCond %{REQUEST_FILENAME}\.gz -s
  RewriteRule ^(.*)\.css $1\.css\.gz [L,QSA,T=text/css]
</IfModule>
### End CSS GZIP ###
```

- сохраним настройки и очистим кэш.

Если в шаблонах темы оформления необходимо использовать дополнительные CSS и JavaScript, то желательно подключать их с помощью следующих команд:

- `drupal_add_css('путь к CSS относительно корневой директории сайта');`

- `drupal_add_js('путь к JavaScript относительно корневой директории сайта');`

для того, чтобы они оптимизировались (включались в один исходный CSS или JavaScript-файл) и сжимались совместно со всеми остальными CSS- или JavaScript-файлами, используемыми на сайте.

8.2.4. Оптимизация конфигурации и обслуживания Drupal

От оптимизации Drupal с помощью модулей, перейдем к более сложной оптимизации — оптимизации конфигурации и обслуживания Drupal.

1. Уменьшим время хранения пользовательских сеансов. Так как Drupal хранит их в своей базе данных, то сокращение времени их хранения разгрузит базу данных, особенно, если на сайт приходят тысячи пользователей в день. По умолчанию сеансы хранятся 55 часов, уменьшим время их хранения до 24 часов. Для этого на сервере в папке `/sites/default` в файле `settings.php` изменим строку

```
ini_set('session.gc_maxlifetime',    200000);
```

на

```
ini_set('session.gc_maxlifetime', 86400); // 24 часа (в секундах)
```

Также в этом файле можно сократить время жизни кэшированных страниц сеансов до 24 часов, изменив строку

```
ini_set('session.cache_expire',    200000);
```

на

```
ini_set('session.cache_expire',     1440); // 24 часа (в минутах)
```

Напоследок в этом же файле изменим время хранения cookie в браузере пользователя, сократив его до 24 часов:

```
ini_set('session.cookie_lifetime', 86400); // 24 часа (в секундах)
```

Если установить время хранения cookie в браузере пользователя равным 0, то cookie будет удаляться сразу после закрытия Интернет-браузера пользователем.

2. Сократим количество сообщений протоколирования работы сайта, сохраняемых в базе данных. На странице «Управление → Отчеты и сообщения → Отчеты в базе данных» (http://www.example.ru/admin/settings/logging/dblog), выставим необходимый максимум отчетов, хранимых в базе данных. Данные отчеты полезны для просмотра попыток взлома сайта, поэтому минимум, который можно выбрать, — это 100 записей. Просмотреть данные отчеты можно, перейдя на страницу «Управление → Недавние записи в системном журнале» (http://www.example.ru/admin/reports/dblog).

3. Настроим выполнение регулярных процедур (задачи cron), так как при их выполнении очищаются журналы записей сообщений протоколирования работы сайта, устаревшие записи кэша и другие статистические данные. Самым простым способом настройки автоматического запуска регулярных процедур является установка модуля — Poormanscron. Скачаем данный модуль по адресу http://drupal.org/project/poormanscron. Распакуем его в папку /sites/all/modules, активируем модуль на странице «Управление → Модули» (http://www.example.ru/admin/build/modules). Установим интервал запусков Cron на странице «Управление → Poormanscron» (http://www.example.ru/admin/settings/poormanscron) равным 360 минут (один раз в 6 часов).

4. В составе Drupal имеется модуль Throttle, который производит оценку количества посетителей сайта и отключает некоторые функциональные возможности, если достигнут порог, установленный администратором. После активации модуля на странице «Управление → Модули» (http://www.example.ru/admin/build/modules) можно увидеть, что у некоторых модулей на данной странице кроме флажков включения появились флажки, отмечающие, должен ли данный модуль регулироваться Throttle или нет. Также некоторые блоки могут регулироваться Throttle («Управление → Блоки» (http://www.example.ru/admin/build/block). Настройка Throttle производится на странице «Управление → Регулятор» (http://www.example.ru/admin/settings/throttle), где указывается минимальное количество анонимных посетителей и минимальное количество зарегистрированных пользователей для включения ограничения функционала сайта для них. На этой странице установим вероятностный ограничитель авторегулятора на 20%, чтобы для одного из каждых 5 запросов на выдачу страницы для браузера пользователя производился 1 запрос к базе данных для определения нагрузки на сайт.

8.2.5. Оптимизация сервера

Так как сервер сайта может работать под управлением разных операционных систем:

- Windows,
- Lunux,
- FreeBSD,

то в каждом случае настройки оптимизации сервера будут отличаться (т. е. установка eAccelerator в Windows и Lunux сильно различается). Ниже приведены только основные рекомендации по оптимизации сервера. Подробно из рекомендаций рассмотрена лишь установка PHP-акселератора на сервер Ubuntu 8.04, так как PHP-акселератор значительно ускоряет работу сайта.

1. Установим eAccelerator. Он является PHP-акселератором, основное назначение которого состоит в кэшировании бинарного представления кода.

Соединимся с сервером по SSH и авторизуемся с правами root. Выполним команды для установки дополнительного пакета php5-dev:

```
sudo apt-get install php5-dev
sudo apt-get install make
```

Выполним команды для установки eAccelerator:

```
sudo cd /tmp/
sudo wget http://bart.eaccelerator.net/source/0.9.5.3/eaccelera-
tor-0.9.5.3.tar.bz2
sudo tar xvjf eaccelerator-0.9.5.3.tar.bz2
sudo cd eaccelerator-0.9.5.3
sudo phpize
sudo ./configure –enable-eaccelerator=shared
sudo make
sudo make install
```

Отредактируем файл php.ini в папке /etc/php5/apache2, вставим в начале файла после тега [PHP] следующий код:

```
; eAccelerator configuration
; Note that eAccelerator may also be installed as a PHP exten-
sion or as a zend_extension
; If you are using a thread safe build of PHP you must use
; zend_extension_ts instead of zend_extension
;extension         = "/usr/lib/php5/20060613+lfs/eaccelerator.so"
zend_extension     = "/usr/lib/php5/20060613+lfs/eaccelerator.so"
eaccelerator.shm_size       = "16"
```

```
eaccelerator.cache_dir       = "/var/cache/eaccelerator"
eaccelerator.enable      = "1"
eaccelerator.optimizer       = "1"
eaccelerator.check_mtime     = "1"
eaccelerator.debug       = "0"
eaccelerator.filter      = ""
eaccelerator.shm_max       = "0"
eaccelerator.shm_ttl       = "0"
eaccelerator.shm_prune_period     = "0"
eaccelerator.shm_only      = "0"
eaccelerator.compress      = "1"
eaccelerator.compress_level     = "9"
eaccelerator.allowed_admin_path  = "/var/www/eaccelerator"
```

При использовании Zend Optimizer и/или ionCube Loader приведенный выше код будет выглядеть так:

```
; eAccelerator configuration
; Note that eAccelerator may also be installed as a PHP exten-
sion or as a zend_extension
; If you are using a thread safe build of PHP you must use
; zend_extension_ts instead of zend_extension
;extension         = "/usr/lib/php5/20060613+lfs/eaccelerator.so"
zend_extension     = "/usr/lib/php5/20060613+lfs/eaccelerator.so"
eaccelerator.shm_size            = "16"
eaccelerator.cache_dir           = "/var/cache/eaccelerator"
eaccelerator.enable              = "1"
eaccelerator.optimizer           = "1"
eaccelerator.check_mtime         = "1"
eaccelerator.debug               = "0"
eaccelerator.filter              = ""
eaccelerator.shm_max             = "0"
eaccelerator.shm_ttl             = "0"
eaccelerator.shm_prune_period    = "0"
eaccelerator.shm_only            = "0"
eaccelerator.compress            = "1"
eaccelerator.compress_level      = "9"
eaccelerator.allowed_admin_path = "/var/www/eaccelerator"

; ionCube Loader configuration
zend_extension=/usr/local/lib/ioncube/ioncube_loader_lin_5.2.so
```

```
; Zend Optimizer configuration
zend_extension=/usr/local/lib/Zend/ZendOptimizer.so
zend_optimizer.optimization_level=15
```

Создадим кэш-каталог для eAccelerator, выполнив команды

```
sudo mkdir -p /var/cache/eaccelerator
sudo chmod 0777 /var/cache/eaccelerator
```

Перезапустим Apache:

```
sudo /etc/init.d/apache2 restart
```

2. Рекомендуем установить Web-сервер nginx и настроить его работу с веб-сервером Apache так, чтобы страницы он отдавал браузеру пользователя Apache, а статический контент (CSS, JavaScript, фото и т. д.) — nginx. Либо полностью замените веб-сервер Apache веб-сервером nginx.

3. Установим в Apache модуль `mod_expires`, который позволяет Drupal посылать HTTP-заголовки `Expires`, кэшируя все статические файлы (изображения, CSS, JavaScript и т. п.) в Интернет-браузере пользователя на определенный срок или до момента появления новых версий файлов. Настройки взаимодействия Drupal и модуля `mod_expires` веб-сервера Apache находятся в файле `.htaccess` в корневой директории сайта:

```
# Включить mod_expires.
<IfModule mod_expires.c>
  # Разрешить истечение срока.
  ExpiresActive On
  # Кэшировать все файлы на две недели после доступа (A).
  ExpiresDefault A1209600
  # Не кэшировать динамически генерируемые страницы.
  ExpiresByType text/html A1
</IfModule>
```

4. Для ускорения обработки `.htaccess` файлов веб-сервером их содержание можно перенести в главный файл конфигурации Apache — httpd.conf. После чего необходимо запретить поиск файлов `.htaccess` в пределах корневого каталога веб-сервера, установив AllowOverride в None:

```
<Directory/>
  AllowOverride
  …
</Directory>
```

Ввиду того, что некоторые модули внутри своих каталогов могут содержать файлы `.htaccess`, следует аккуратно работать с данным видом оптимизации, чтобы при переносе содержимого всех файлов `.htaccess` в `httpd.conf` не пропустить не один файл `.htaccess`.

5. Установим на сервере:

- систему анализа лог-файлов (например, AWstats);
- систему мониторинга производительности сервера (например, Munin);
- систему для учета сетевого трафика (например, Vnstat).

6. Включим кэш MySQL и установим его размер равным 64 мегабайтам. Для этого отредактируем файл `my.cnf` в папке `/etc/mysql` (при использовании Ubuntu 8.04). Изменим значение

```
# query_cache_limit         = 1M
# query_cache_size         = 16M
```

на

```
query_cache_limit         = 1M
query_cache_size         = 64M
```

После чего перезапустим MySQL командой

```
/etc/init.d/mysql  restart
```

Слишком маленький размер кэша — малоэффективен, а слишком большой размер кэша приводит к тому, что поиск нужной информации в кэше тратит много времени. Поэтому рекомендуем поэкспериментировать с размером кэша на каждом конкретном сервере и подобрать его оптимальный размер.

7. Проверим: загрузку центрального процессора, нехватку оперативной памяти или места на диске и возможную перегрузку линии связи сервера с Интернетом. Если необходимо разместить базу данных на отдельном сервере, то изменим настройки соединения с базой данных, — они находятся в файле `settings.php` (в папке `/sites/default`) — либо разместим сайт на кластере из серверов (например, воспользовавшись услугами сервиса Amazon C2).

8.2.6. Заключение

Для просмотра информации о сервере из Drupal существует удобный модуль — System information (http://drupal.org/project/systeminfo). После его установки и активации информацию о вашем сервере можно посмотреть на странице http://www.example.ru/admin/reports/systeminfo.

Для тестирования скорости работы сайта и оптимизации его контента очень удобно пользоваться плагином YSlow для браузера Firefox, который показывает подробную статистику по загрузке страниц с вашего сайта.

8.3. Разгоняем Wordpress

Wordpress (http://www.wordpress.org/) является сейчас наиболее популярной платформой для одиночного хостинга блогов. Ряд хостинг-провайдеров уже даже предлагают площадки с предварительно установленным Wordpress, а в большом количестве изданий рассуждают, как лучше заработать на новом блоге или правильно его использовать. Ниже будет освещен ответ на один из основных вопросов, встающих перед администраторами блогов: как сделать так, чтобы сайт быстро работал. Нижеизложенный материал рассчитан на максимально широкую аудиторию пользователей.

8.3.1. Основные положения

Ускорение работы любой системы возможно, в основном, за счет кэширования некоторых (тут стоит подчеркнуть, что именно некоторых, а не всех подряд) часто используемых операций. Все кэширующие мероприятия, в том числе и для Wordpress, можно разбить на несколько основных частей:

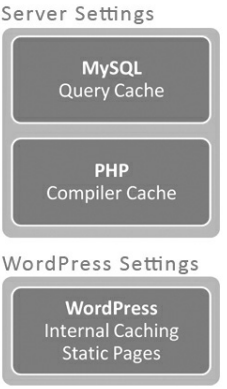
- база данных;
- компиляция серверных скриптов (PHP);
- статические страницы;
- клиентская составляющая.

Данную проблему можно проиллюстрировать при помощи следующего рисунка:

8.3.2. База данных

Так уж сложилось, что основное узкое место практически любой системы заключается в базе данных, поэтому ее стараются ускорить всеми возможными способами. Стоит отметить, что проблема многочисленных вызо-

Рис. 8.4. Кэширующие звенья для Wordpress, источник: www.arnebrachhold.de



вов к базе данных не решается просто уменьшением их количества (для предоставления той же самой информации), тут надо подходить более комплексно и настраивать многоуровневый кэш для запросов. Относительно MySQL это сделать довольно просто: достаточно прописать в конфигурационном файле `my.cnf` (или `my.ini`) следующие параметры (в случае большого количества оперативной памяти 20 Мб может быть увеличено до любого приемлемого количества, но не стоит здесь увлекаться: скорость поиска данных в кэше напрямую зависит от размера самого кэша):

```
query-cache-type=1
query-cache-size=20M
```

Для оптимизации таблиц (что позволит уменьшить время запросов на 20—50%) можно воспользоваться дополнением Optimize DB (http://yoast.com/wordpress/optimize-db/), которое позволит существенно уменьшить размер таблиц MySQL и улучшить их структуру. Для кэширования запросов к базе данных также существует специальное дополнение, DB Cache Reloaded (http://wordpress.org/extend/plugins/db-cache-reloaded/).

8.3.3. Компиляция серверных скриптов

Каждый раз, когда исполняется PHP-скрипт, он заново компилируется в памяти в исполняемый код, что требует значительного времени. Чтобы избежать повторной компиляции одних и тех же скриптов, использу-

ются такие приложения, как APC (http://pecl.php.net/package/APC) или eAccelerator (http://eaccelerator.net/), которые сохраняют уже скомпилированный код в памяти и позволяют выполнять его значительно (до нескольких десятков раз) быстрее. Также данные решения хорошо справляются с большим количеством маленьких файлов, которые подключаются при обработке запроса к странице, снижая издержки при обращении к файловой системе. PHP-движок не загружает каждый раз файлы с диска (или из дискового кэша) — он получает сразу исполняемый код, что намного увеличивает скорость выполнения. После оптимизации базы данных (настройки кэширования) это одно из наиболее узких мест (за исключением создания статических страниц вместо динамической их генерации).

8.3.4. Статические страницы

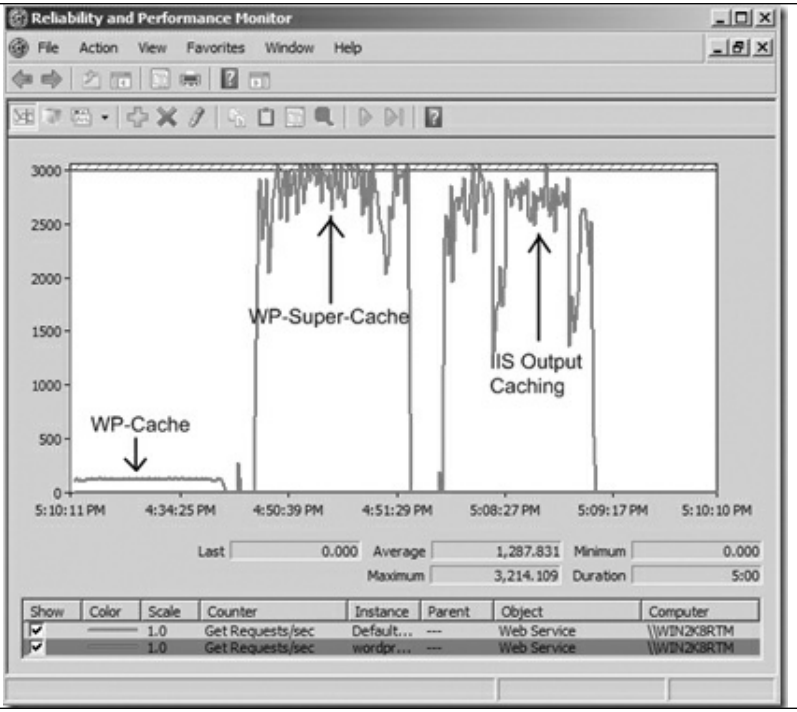
Следующим шагом для борьбы с большим временем подготовки страницы на сервере будет полное кэширование создаваемой страницы в один файл или одну запись в оперативной памяти. Для включения внутреннего кэширования на уровне самого Wordpress достаточно раскомментировать (или добавить) в файл `wp-config.php` следующие строки (предварительно проверив, что директория `wp-content/cache` доступна для записи, иначе ничего не получится):

```
define('ENABLE_CACHE', true );
define('CACHE_EXPIRATION_TIME', 900);
```

Более серьезных результатов кэширования можно добиться при помощи дополнения WP-Super-Cache (http://ocaoimh.ie/wp-super-cache/ , базирующегося на WP-Cache, http://mnm.uib.es/gallir/wp-cache-2/) или Hyper Cache (http://www.satollo.com/english/wordpress/hyper-cache), которое вообще не будет осуществлять никаких запросов к базе данных для отображения внешних веб-страниц. Однако при этом станет невозможно учитывать статистику посещений через встроенные в Wordpress методы (только через внешние счетчики или по логам сервера). Для Wordpress, установленного на IIS, также лучше всего будет использовать именно WP-Super-Cache вместо IIS Output Caching. Это подробно рассматривается в соответствующей заметке; ниже приведено число запросов в секунду при том или ином методе серверного кэширования.

Но давайте посмотрим, что можно сделать с клиентской составляющей (дизайном и скриптами) обычного блога.

Рис. 8.5. Производительность кэширования Wordpress для IIS, источник: blogs.iis.net



8.3.5. Клиентская часть

Основной минус богатства тем для Wordpress — то, что они в основном разрабатываются любителями. Как следствие, такие темы могут состоять из большого количества картинок и файлов стилей, которые в совокупности загружаются очень медленно. Однако ситуация поправимая. Для ускорения загрузки сайта в самом браузере (а это, по мнению экспертов Yahoo!, занимает 95% времени общей загрузки страницы) можно воспользоваться несколькими решениями:

- Включить сжатие страниц в самом Wordpress. Делается это через «Настройки → чтение» («WordPress должен упаковывать статьи (gzip), если браузер запросит это»).
- CSS Compress (http://dev.wp-plugins.org/wiki/css-compress) — дополнение к Wordpress, которое автоматически минимизирует и сжимает CSS-файлы.

- PHP Speedy (http://aciddrop.com/php-speedy/) позволяет объединить все CSS- и JS-файлы, настроить клиентское кэширование и сжатие текстовых файлов. Это позволяет значительно ускорить загрузку страниц для конечных пользователей (особенно если это ваши постоянные посетители). Устанавливается и как плагин, и как отдельное приложение.
- Web Optimizer (http://code.google.com/p/web-optimizator/) устанавливается и как отдельное веб-приложение, и как встроенный плагин. Обладает более широкими возможностями, в частности, автоматическим созданием CSS Sprites (что ускоряет загрузку страниц для пользователей IE) и добавлением всех серверных правил в `.htaccess`-файл (что обеспечивает более широкую совместимость и снимает нагрузку с PHP-скриптов на анализ кэширования и осуществление сжатия). Также Web Optimizer позволяет настроить «ненавязчивую» загрузку JavaScript и распределить изображения по нескольким статическим хостам.

В общем, даже самый обычный блог может быть ускорен в несколько (десятков) раз за считанные минуты. Скорее всего, в ближайшем будущем уже появятся отдельные сборки Wordpress, настроенные на максимальную производительность при задействовании любой темы и произвольной посещаемости блога. Сейчас же можно просто воспользоваться вышеприведенными советами и порадоваться за значительное увеличение числа посещений и постоянных читателей.

8.4. Разгоняем Joomla! 1.5

Скорее всего, многие уже слышали о медленной работе Joomla! (http://www.joomla.org/), одной из самых популярных бесплатных CMS, равно как и о ее сильной уязвимости для атак хакеров. Благодаря простоте создания расширений для Joomla! сейчас доступно несколько (десятков) тысяч разнообразных модулей, компонентов и расширений, позволяющих установить на сайт практически произвольный функционал: от полноценной социальной сети до Интернет-магазина.

Естественно, что у такой простоты есть и обратная сторона: большинство расширений создаются без учета требований высокой производительности и какой-либо оглядки на мощности конечных серверов, очень часто даже не выделенных или виртуальных, а расположенных на общем хостинге. Разработка бесплатных решений оборачивается 100-500% замедлением в скорости загрузки сайта. Давайте разбираться, как с этим можно бороться.

8.4.1. Серверная часть

Для начала посмотрим, что можно сделать с серверной производительностью. Мы исследовали стандартную сборку Joomla!, но даже в такой комплектации на отдельном сервере время создания страницы занимало 0,312 с (замер времени ответа производился с помощью `curl`, интерфейс к которому выложен на webo.in, http://webo.in/my/action/timings/). Это не очень много, но в условиях виртуального хостинга может возрасти многократно, даже на изначально хорошо оптимизированных окружениях.

Встроенное кэширование в Joomla! 1.5 работает достаточно хорошо и позволяет сэкономить существенное время при создании страницы. Однако нужно понимать, что оно может быть применимо далеко не для всякой системы. В случае статичных новостных сайтов и небольших интернет-магазинов кэширование может помочь, но для социальных сетей с активным добавлением новых материалов и их комментированием оно точно не подойдет.

Кэширование на стороне сервера

Включение встроенного кэширования в Joomla! 1.5 сократило время ответа тестируемого сервера примерно на 30% (на 0,107 с). Прекрасно понятно, что в большинстве случаев оно будет практически бесполезно: если необходимо сократить время создания страниц на порядок, то нужны более кардинальные методы.

В качестве одного из кэширующих решений может использоваться и Web Optimizer: встроенное кэширование HTML-документов позволяет отдавать их сразу в том виде, в котором они получаются системой после всех запросов к базе. При этом, естественно, практически все эти запросы не осуществляются. Данное кэширование («монолитное») подойдет только в тех случаях, когда внешние страницы у Joomla! меняются относительно редко.

Если просто включить Web Optimizer (http://code.google.com/p/web-optimizator/) в процесс создания страниц, то время обработки документа возрастет незначительно (после создания всех кэширующих файлов на 0,006 с или 3% на тестовом сервере). Дополнительно включив HTML-кэширование в Web Optimizer, можно сократить время отдачи документа до 0,08 с (почти в 4 раза по сравнению с исходным временем создания страницы). Сразу стоит отметить, что установка вроде бы аналогичного по функциональности дополнения Content Static (http://extensions.joomla.org/extensions/site-management/cache/5104/details) визуально на производительности никак не отразилась.

Очевидно, что более грамотным будет кэшировать отдельные модули на странице, оставляя нужные места (или, как их любят называть в шаблонных движках, — заглушки) динамическими. Однако данное решение требует существенного вмешательства в алгоритм работы самой CMS. Дополнение System-Cache (включено в сборку по умолчанию) работает именно по такому принципу и на данный момент обеспечивает практически наилучшую производительность (при большой стабильности алгоритма).

Заканчивая речь про кэширование создаваемых страниц на стороне сервера, стоит упомянуть, что дополнение Joomla Performance Booster (http://www.joomlatwork.com/products/components/joomla-performance.html) в ходе тестирования показало результаты примерно на 50% (пятикратный прирост производительности по сравнению с «обычной» версией) лучше, чем System-Cache, однако может работать не очень стабильно.

Кэширование запросов к базе данных

К серверному кэшированию можно подойти и с другой стороны: ограничить число запросов к базе данных, — обычно именно эта часть вызывает наиболее серьезную «утечку» производительности. Для Joomla! существует расширение, позволяющее закэшировать все (или почти все) запросы к базе. Тут стоит понимать, что база данных сама по себе может работать достаточно быстро, и подобное решение будет эффективно только в том случае, если восстановление закэшированного значения выборки на порядок (или хотя бы в разы) быстрее, чем осуществление самой выборки (например, 1 мс против 10 мс). В противном случае прироста производительности не произойдет.

Дополнение Query Cache (http://extensions.joomla.org/extensions/site-management/cache/3180/details) позволяет задействовать как файловую систему, так и популярные кэширующие подсистемы (APC, Memcache и др.) для сохранения выполненных запросов.

8.4.2. Клиентская часть

Для оценки эффективности решений для клиентской оптимизации использовалось хорошо зарекомендовавшее себя (и относительно беспристрастное) дополнение к Firefox — YSlow.

«Чистая» система

«Чистая» установка Joomla! 1.5 набрала 65 баллов из 100. Вполне приемлемо. Стоит понимать, что если на систему дополнительно поставить десяток модулей и компонентов, то оценка резко ухудшится до 30-40.

Следующий этап: архивирование

В Joomla! есть встроенный gzip. Однако, во-первых, он работает через PHP, во-вторых, только для HTML-файлов. Грустно, что и видно по оценке: она поднялась только до 67.

CssJsCompress

Довольно известное дополнение (http://extensions.joomla.org/extensions/site-management/cache/7350/details), позволяющее объединять CSS- и JS-файлы. Однако оно не добавляет к ним всех кэширующих заголовков и сжатия, что и отразилось на результате: всего 72 балла по YSlow. В самой Joomla! gzip при этом был включен. Дополнение CSS/JS Cache (http://extensions.joomla.org/extensions/site-management/cache/7801/details) не удалось заставить корректно работать.

Joomla Perfomance Booster

Joomla Performance Booster (http://www.joomlatwork.com/products/components/joomla-performance.html) является платным дополнением (39 евро) и представляет собой наиболее мощное «встроенное» решение для Joomla! 1.5. После его установки и настройки (объединение JavaScript работало «со скрипом» и его пришлось выключить) был достигнут результат в 73 балла (вполне вероятно, что при правильной работе с JavaScript оценка YSlow поднялась бы и до 75). В целом достаточно мощное дополнение, поскольку обеспечивает кроме самого кэширования еще и очень гибкое управление созданным кэшем.

Однако данное дополнение возможно подключить вместе с приложением Web Optimizer (которое возьмет на себя всю логику преобразования клиентской части), что позволит существенно ускорить работу сайта на Joomla! практически любой сложности.

Smart Optimizer

Далее был протестирован Smart Optimizer (http://farhadi.ir/works/smartoptimizer, как отдельное PHP-приложение) — по характеру работы полностью аналогичный известному Minify (http://code.google.com/p/minify/, дополнение Minify4Joomla, http://extensions.joomla.org/extensions/site-management/cache/7183/details, «завести» не удалось). Установка у него достаточно сложная для непрофессионала, к тому же приходится править шаблоны вручную, нет возможности объединять файлы из разных директорий. Однако все остальное на высоте: оценка поднялась до 85. В самой Joomla! gzip при этом был включен.

Web Optimizer

Web Optimizer (http://www.web-optimizer.ru/, как отдельное PHP-приложение или как плагин), естественно, устанавливается в «два клика» и обладает более мощным клиентским арсеналом: при отключенном сжатии в самой Joomla! оценка поднялась до 94 (с 65 изначально). Наверное, тут уже дополнительных комментариев не нужно.

8.4.3. Заключение

На данный момент для Joomla! 1.5 не существует более мощного бесплатного решения для оптимизации производительности, чем Web Optimizer. PHP Speedy (http://code.google.com/p/phpspeedy/), к сожалению, доступен только для Joomla! 1.0.

В зависимости от конкретных задач и серверного окружения скорость работы практически произвольного сайта на Joomla! может быть ускорена на порядок (имеется в виду серверное + клиентское ускорение). Возможно, в части производительности Joomla! и уступает Drupal (http://www.drupal.org/), однако при правильном использовании существующих инструментов разрыв этот можно сократить до минимума.

8.5. Разгоняем Joostina

Автор данного раздела, Николай Кирш, — веб-разработчик, основатель и технический лидер проекта Joostina CMS (http://www.joostina.ru/). Профессиональные приоритеты: качественный код, оптимизация под нагрузки, клиентская оптимизация.

Joostina родилась и развивается с изначальной целью: быть максимально быстрой и эффективно использовать ресурсы сервера, не уменьшая при этом удобств как для пользователя, так и для администратора сайта. В основе системы лежит CMS Joomla! 1.0.x, считающаяся уже классикой. За время развития проекта было учтено максимум пожеланий пользователей по насыщению системы необходимым функционалом, изначально отсутствующим в Joomla!. Но кроме новых возможностей также добавились новые настройки, позволяющие оптимизировать сайт под более конкретные задачи и типовые Интернет-решения.

Оптимизацию CMS Joostina можно разделить на 3 ступени:

- оптимизация через базовые настройки системы;
- оптимизация с использованием сторонних расширений;
- оптимизация под конкретные нужды, отключение лишнего кода и настройка сервера.

Все настройки указаны для Joostina версии 1.3.0.

8.5.1. Оптимизация через базовые настройки системы

Самый быстрый и безопасный способ настроить свой сайт на более высокую скорость и выжать из него максимум возможностей — основательно ознакомиться с настройками, располагающимися в «Глобальной конфигурации». Для доступа к настройкам необходимо авторизоваться с правами Супер-администратора в административной части, называемой так же панелью управления: http://www.example.ru/administrator. Далее надо выбрать пункт меню «Сайт → Глобальная конфигурация», или прямо на главной страницы панели управления, через кнопку быстрого доступа «Глобальная конфигурация».

Отключить генерацию RSS (syndicate)

Joostina, как и большинство современных CMS, умеет формировать RSS-ленты из материалов, размещенных на сайте. Чтобы браузеры при отображении страниц сайта автоматически подхватывали RSS-ленты, ссылки на них прописываются в HTML-код страниц через теги примерно такого содержания:

```
<link rel="alternate" type="application/rss+xml"
  title="Joostina v 1.3.0 b"
  href="http://www.example.ru/index2.php?option=com_rss&feed=0
    &amp;no_html=1" />
```

Но формирование ссылки на ленту занимает определенное время, оно нужно на запрос в базу данных на получение параметров отображения. Если же тег ленты необходим, а избавиться от лишнего запроса тоже хочется, — можно прописать ссылку напрямую в шаблоне сайта.

Использовать шаблон

Для каждого пункта меню в панели управления можно выбрать уникальное отображение и состав модулей. Но также для каждого из пунктов меню можно назначить уникальный шаблон. Если такая возможность на сайте не используется, то ее следует отключить. Для этого и создана дан-

ная настройка. Параметр позволяет выбрать единый шаблон для всего сайта, что исключит один запрос и его обработку для выбора конкретного шаблона. Аналогичная настройка существует для панели управления.

Отключить мамботы группы system

Мамботы — это чаще всего небольшие PHP-сценарии, срабатывающие на определенном этапе работы системы. Мамботы группы system срабатывают в момент инициализации системы. Обычно в группу входят расширенные обработчики SEF и библиотеки подключения Javascript. Все мамботы этой группы можно посмотреть в панели управления «Меню → Мамботы → Мамботы сайта». Если справа в списке выбора типа нет группы system, то настройку рекомендуется отключить, это сделает ненужным один запрос в базу и инициализацию механизма.

Отключить мамботы группы content,
Отключить мамботы группы mainbody

Действие данного пункта аналогично группе system, но поступать тут надо внимательнее. Группа content — основная, за счет нее выводятся изображения, вставленные в текст через тег `{mosimage}`, разбивка на страницы внутри текста и т. д. Безопаснее всего поочередно снимать мамботы с публикации и смотреть, что изменилось на сайте. Если все мамботы не опубликованы, а сайт отображается верно, — можно отключить всю группу.

Использовать неопубликованные мамботы

Мамботы группы content часто работают, заменяя определенные теги в тексте, например, `{mosimage}`. Но если мамбот не опубликован, то система его все равно использует — чтобы убрать из текста этот самый тег `{mosimage}`. Если на сайте такие мамботы не используются, то лучше активировать данную настройку, исключив неиспользуемые обращения к базе данных и подключение лишних файлов.

Авторизация на сайте

Отключив данный пункт, вы запретите инициализацию пользователей на сайте. Параметр запретит заведение пользовательских сессий в базе данных и позволит более полно кэшироваться страницам. Если на сайте не предусмотрена работа пользователей, то и авторизацию лучше отключить.

Время существования сессии на фронте

При авторизации для пользователя заводится специальная сессия, данные о ней записываются в базу данных и имеют определенный срок

жизни: пока сессия жива — пользователь считается авторизованным. Если указать в настройке большое время жизни сессии, то в базе данных сессий будет довольно много значений, что повлечет дополнительную нагрузку для поиска сессий конкретного пользователя.

Отключить сессии на фронте

При посещении сайта авторизованным пользователем или даже гостем для него запускается механизм инициализации сессий, что влечет за собой запись данных в базу, создание cookie у пользователя и постоянную проверку авторизации. Если авторизация на сайте не важна, то параметр рекомендуется отключить. Но учтите, что модуль отображающих посетителей будет выдавать не точную информацию, так как он основывается на данных записанных в таблице сессий. Параметр рекомендуется использовать совместно с пунктом «Авторизация на сайте».

Отключить контроль доступа к содержимому

Хотя в Joostina имеется не очень много возможностей для полноценного создания и управления правами пользователей, доступ к содержимому всегда ведется с учетом прав текущего пользователя. Это добавляет в SQL-запрос дополнительное условие. На сайтах, где доступ не разграничен на зарегистрированных и гостей, параметр лучше активировать.

Считать число прочтений содержимого

При прочтении каждого содержимого увеличивается значение поля счетчика в таблице содержимого. Постоянные изменения даже одного поля таблицы содержимого сводят на нет встроенный в mysql механизм кэширования, да и дополнительный запрос в базу тоже лучше исключить. Настройку рекомендуется выключить, а ведение статистики доверить специализированным сервисам, типа li.ru или Google Analytics.

Отключить проверки публикации по датам

Для каждого материала при редактировании можно указать определенные периоды начала и окончания публикации. Проверка по датам добавляет в каждый SQL-запрос условие соответствия текущей дате, а настройка данное условие отключает. Чем меньше условий в запросе, тем легче его будет отработать базе данных и тем быстрее пользователь сайта увидит ожидаемые страницы.

GZIP-сжатие страниц

Позволяет передавать пользователю более компактные страницы. Содержимое пакуется через gzip-алгоритм на сервере, а распаковывается

автоматически в браузере пользователя. Получается экономия на трафике, но немного больший расход ресурсов сервера; более подробно расчет оптимальности применения gzip приведен во второй главе книги «Разгони свой сайт».

Блокировка компонентов

Позволяет отключить прямой доступ к компонентам, набрав специальный адрес в браузере. Если на сайте имеется компонент, но он не используется, то настройку лучше активировать.

Рейтинг/Голосование

В базовой поставке системы имеется мамбот группы content, позволяющий выставлять рейтинг для каждого материала. Если такая возможность не требуется, рейтинг лучше отключить, — это исключит лишние проверки и инициализации.

Ежедневная оптимизация таблиц базы данных

Добавляет в работу сайта ежедневную оптимизацию всех таблиц базы данных через выполнение OPTIMIZE TABLE для каждой. Данная процедура уменьшает фрагментацию и производит общую оптимизацию таблиц встроенными средствами mysql.

Сжатие CSS- и JS-файлов

Позволяет выдавать вместо обычных JS- и CSS-файлов их упакованные аналоги. Экономит трафик и позволяет указать более длительное время кэширования. Работает только для встроенных файлов.

Значение тега revisit:

Позволяет указать параметр тега:

```
<meta name="revisit" content="10 days" />
```

который указывает, как часто сайт должен посещать поисковый робот. Правильно установленное значение позволит исключить повышенную нагрузку на сервер, вызванную слишком частым посещением поискового робота.

8.5.2. Встроенное кэширование

Включить кэширование

В Joostina встроен механизм, позволяющий кэшировать результаты выполнения ресурсоемких функций или инициализаций объектов. Ис-

пользование кэширования позволяет уменьшить число запросов в базу, уменьшить число подключаемых файлов и увеличить общую скорость работы системы. Кэширование лучше активировать после полного построения и отладки сайта.

Тип кэширующей системы

Кэшировать можно как в файлы, так и в специальные акселераторы кэширования. Joostina поддерживает работу кэширования с использованием apc, eaccelerator, xcache и memcache. Первые 3 — это не только кэш-акселераторы, но и общие оптимизаторы работы php. Наличие любого из них — очень большой плюс в работе сайта.

Оптимизация кэширования

При активации параметра из кэшируемых объектов будут удалены все неиспользуемые символы, например, переводы строк или множественные пробелы. Это уменьшает общий объем файла кэша и немного уменьшает трафик, передаваемый пользователю.

Автоматическая очистка каталога кэша

При кэшировании в файлы каталог кэша может содержать множество просроченных объектов. Система следит, чтобы таких случаев не было, но для большей уверенности рекомендуется активировать и этот параметр. На оптимизацию работы сайта число файлов в каталоге кэша имеет прямое влияние: чем больше файлов в каталоге одного уровня — тем дольше файловая система будет их отдавать.

Кэширование меню панели управления

При работе в панели управления в верхней части отображается меню, которое содержит пункты для доступа к основным операциям. Меню частично формируется из базы данных. Например, список установленных компонентов или список разделов и категорий. Такие данные изменяются не очень часто, и лучше произвести кэширование этого участка. Активация параметра также сделает вывод меню через внешний JavaScript-файл, код которого исключится из тела страниц и будет кэшироваться еще и на стороне пользователя — в браузере.

Каталог кэша (/dev/shm)

По умолчанию файлы кэша складываются в каталог `/cache` в корне сайта. В зависимости от настроек сервера можно попытаться перенести этот каталог в более быстрое место, например, на диск с другой файловой системой или `/dev/shm`. Не забудьте убедиться, что PHP-интерпретатор имеет полный доступ к указанному каталогу.

Время жизни кэша

Позволяет указать период времени, на который должно срабатывать встроенное кэширование. Если сайт имеет не очень частое изменение структуры и содержимого или слабую активность пользователей, то время лучше указать больше. Выбранный период используется по умолчанию для всего кэша. Но в настройках модулей можно дополнительно выбрать, на какой период их кэшировать.

8.5.3. Отключение встроенной статистики

Включить сбор статистики

Параметр отвечает за исключение из работы системы сбора информации о браузере и других данных, которые лучше собирать через специальные сервисы, озвученные выше. Рекомендуется отключить.

Вести статистику просмотра содержимого по дате

Аналогично предыдущему параметру — лучше отключить и вести все учеты на серверах специальных сервисов.

Статистика поисковых запросов

Joostina сохраняет информацию о каждом слове, которые пользователи ищут на сайте. Возможность позволяет частично проанализировать пользовательскую аудиторию и узнать их интересы. Если такой функционал не требуется — лучше отключить.

8.5.4. Отключение неиспользуемых расширений

Один из основных принципов оптимизации — использование только необходимого функционала. Joostina по своей сути является универсальной системой, и это влечет за собой некоторую ограниченность и сложность. Базовый дистрибутив системы имеет набор встроенных расширений, часть которых может не использоваться на сайте. Всего в Joostina можно отключить расширения всех 3-х типов:

- компоненты,
- модули,
- мамботы.

Отключение компонентов лишь косвенно влияет на оптимизацию сайта. Но блокировка доступа — это отличный способ уменьшить число страниц для индексации или попыток спама. Чем меньше страниц, тем быстрее их проиндексирует поисковый робот и тем меньше времени эти страницы надо будет генерировать серверу для поисковика.

Отключить неиспользуемые компоненты можно в панели управления на странице управления компонентами: «Меню → Компоненты → Управление компонентами». Для работы данного механизма необходимо, чтобы в глобальной конфигурации была активирована настройка «Блокировка компонентов».

Отключение модулей позволяет уменьшить чисто ненужных запросов в базу, подключение неиспользуемых файлов и общий расход памяти, выделяемой на генерацию страницы. Выяснить, какие модули действительно необходимы, можно по следующей схеме.

Заходим в меню управления модулями: «Меню → Модули → Модули сайта».

Выбираем все модули и нажимаем кнопку «Скрыть», находящуюся в верхней части страницы — на тулбаре.

В новой вкладке или новом окне открываем главную страницу сайта и поочередно публикуем нужные модули. Начать лучше с модуля главного меню. Перед публикацией каждого модуля подумайте, нужен ли он, и попутно запоминайте, сколько новых запросов прибавилось. Если какой-то из модулей создает слишком большую нагрузку — придется поискать его более простые аналоги или найти способы оптимизации кода.

Отключение мамботов позволяет существенно сократить непосредственное время генерации страницы. Мамботы разделены на группы, про это уже сообщалось ранее, и каждая группа отвечает за отдельные участки. Наиболее часто используемые — мамботы группы content, они позволяют обрабатывать содержимое, выдаваемое компонентом. Чаще всего такие мамботы отвечают за замену в тексте специально оформленных тегов на необходимый функционал или оформление. Например, мамбот `bot_mosimage` отвечает за замену тега `{mosimage}` на необходимую картинку. Но для такой работы производится обработка текста регулярными выражениями, что не очень хорошо сказывается на производительности. Отключить неиспользуемые мамботы можно по той же схеме, что и модули, только на другой странице панели управления: Меню → Мамботы → Мамботы сайта.

Если все мамботы группы отключены, то лучше отключить и саму группу — это делается в глобальной конфигурации для каждой группы отдельно.

8.6. Пара советов для Ruby on Rails

Уже много людей писали руководства, помогающие вашему веб-приложению работать быстрее. В этом разделе будут освещены самые про-

стые, но наиболее эффективные методы, которые дадут вам возможность существенно ускорить ваше приложение без потери какого-либо функционала из Ruby on Rails.

Часто бывает так, что одно веб-приложение подгружает сразу несколько JavaScript-файлов и CSS-стилей. Это существенно замедляет загрузку страницы, так как веб-браузер каждый раз заново запрашивает новый файл.

Решение заключается в том, чтобы уменьшить количество внешних ресурсов на вашей странице, объединив их все в один файл. Поможет нам в этом плагин AssetPackager (http://synthesis.sbecker.net/pages/asset_packager).

Ставим

```
script/plugin install git://github.com/sbecker/asset_packager.git
```

Пример `config/asset_packages.yml`:

```
javascripts:
- base:
  - prototype
  - effects
  - controls
  - dragdrop
  - application
- secondary:
  - foo
  - bar
stylesheets:
- base:
  - screen
  - header
- secondary:
  - foo
  - bar
```

И запускаем rake-задачу:

```
rake asset:packager:build_all
```

Дальше для JavaScript пишем

```
<%= javascript_include_merged :base %>
```

или

```
<%= javascript_include_merged 'prototype', 'effects', 'controls',
'dragdrop', 'application' %>
```

Для стилей пишем:

```
<%= stylesheet_link_merged :base %>
```

или

```
<%= stylesheet_link_merged 'screen', 'header' %>
```

В итоге получаем для режима разработки наш старый код, например:

```
<script type="text/javascript"
src="/javascripts/prototype.js"></script> <script
type="text/javascript" src="/javascripts/effects.js"></script>
<script type="text/javascript"
src="/javascripts/controls.js"></script>
<script type="text/javascript"
src="/javascripts/dragdrop.js"></script>
<script type="text/javascript"
src="/javascripts/application.js"></script>
<link href="/stylesheets/screen.css" type="text/css" />
<link href="/stylesheets/header.css" type="text/css" />
```

А в режиме рабочего сайта будет:

```
<script type="text/javascript"
src="/javascripts/base_packaged.js?123456789"></script>
<link href="/stylesheets/base_packaged.css?123456789"
type="text/css" />
```

Теперь, чтобы сделать нагрузку еще меньше, переносим все свои статические файлы на другой хост. В Ruby-on-Rails это очень просто сделать, достаточно добавить в config/environments/production.rb такую строку:

```
config.action_controller.asset_host = "http://assets.example.ru"
```

Теперь все `image_tag`, `javascript_include_tag` и т. д. будут указывать на этот хост.

8.7. Разгоняем jQuery

Автор данного раздела, Олег Смирнов (aka CTAPbIu_MABP), — разработчик пользовательских интерфейсов на Java и JavaScript, на данный момент трудится над системой самообслуживания Украинского мобильного оператора «Киевстар» (http://my.kyivstar.ua/). Олег занимается исследованиями в области производительности JavaScript-библиотек, в частности, jQuery, чему посветил много статей на своем сайте (http://mabp.kiev.ua/).

jQuery, пожалуй, самая известная JavaScript-библиотека. Она позволяет быстро и просто производить манипуляции с DOM-деревом, навешивать события и делать AJAX-запросы на сервер. Ею пользуются очень много компаний, в том числе Google и Microsoft. Под нее написано огромное количество плагинов, позволяющих расширить стандартный функционал и добавить на страницу виджет любой красоты.

К сожалению, большая часть всех плагинов имеет код низкого качества, поэтому при установке нескольких таких плагинов страница начинает существенно подтормаживать. О том, как поправить код чужого плагина, чтобы избежать лишнего расхода памяти при сложных манипуляциях с DOM-деревом, как сделать анимацию плавной даже при анимировании нескольких элементов, а AJAX — быстрым, а также о том, как избежать некоторых скрытых багов, и будет этот раздел.

8.7.1. Selectors

Стоит начать с функции `$`, она принимает два параметра: первый — селектор, второй — контекст. Хотя контекст обычно опускают, впоследствии будет показано, как им грамотно пользоваться.

Простой селект

Самый простой вариант — это выбор по `id`, имени тега и имени класса.

```
$("#id")
$("tag")
$(".class")
```

Не случайно они расположены именно в такой последовательности: они идут по сложности алгоритма выборки. В первом случае вызов функции эквивалентен вызову

```
document.getElementById("id");
```

Поскольку предполагается, что id уникальный, поиск проходит очень быстро, и если на странице есть два элемента с таким id, то найден будет только первый. Хотя в IE и тут сделали ошибку и до 7-й версии включительно в случае отсутствия элемента с таким id он вернет элемент, у которого совпадает атрибут name.

Во втором случае тоже все относительно просто:

```
document.getElementsByTagName("tag");
```

Получили все ноды с таким именем из документа, и все готово. И на удивление никаких ошибок, если не учитывать, что при запросе `getElementsByTagName("*")` IE вернет и комментарии тоже.

В третьем случае, если есть возможность, работу перехватывает

```
document.getElementsByClassName("class");
```

(Таблицу поддержки этой функции в браузерах можно посмотреть на quirksmode.org)

Эту функцию поддерживают еще не все браузеры. Для остальных применяется совсем другой алгоритм: нужно получить абсолютно все ноды, потом обойти их циклом, проверяя имена классов, и если совпали, то добавить в массив результата.

```
var nodes = document.getElementsByTagName("*"), result = [];
for (var i=0; i<nodes.length; i++){
  if(" " + (nodes[i].className ||
nodes[i].getAttribute("class")) +
    " ").indexOf("class") > -1)
      result.push(nodes[i]);
}
```

Какой метод применять, определяется в самом начале при подключении библиотеки.

Селект через querySelectorAll

Но это все подходит только для элементарных селекторов. А на практике приходится обычно использовать намного более сложные конструкции. И для них в современных браузерах FireFox 3.0, Safari 3.2, Opera 9.5, а также в IE8, появились функции `querySelector` и `querySelectorAll`. Они, соответственно, предназначены для поиска одной или нескольких нод по CSS3-селекторам. Если браузер клиента поддерживает эту функцию, то все, о чем написано в прошлом пункте, — отпадает, и поиск происходит через `querySelectorAll`.

```
$("#id .class tag")
```

В лучшем случае селектор будет обработан именно `querySelectorAll`, потому что он написан по правилам CSS3. Но такое возможно не со всеми селекторами: jQuery поддерживает ряд селекторов, которые не входят в CSS3, такие, например, как `:visible`.

```
$("#id .class tag:visible")
```

Такой селектор выдаст ошибку в функции querySelectorAll, и селектор будет перенаправлен в поисковый движок Sizzle, где строка будет разбита на простые селекторы и превратится, по сути, в несколько разных поисков, в котором каждым следующим контекстом является предыдущий селектор.

```
$(document).find("#id").find(".class").find("tag").filter(":visible")
```

Скорость этого метода поиска напрямую зависит от величины DOM-дерева: чем оно больше, тем медленнее, — но ее можно значительно увеличить, написав селектор раздельно.

```
$("#id .class tag").filter(":visible")
```

При этом querySelectorAll выберет все ноды, а Sizzle разберется с `":visible"`.

По поводу псевдо-селекторов возникает также очень интересный вопрос: CSS3 поддерживает несколько видов псевдо-классов, такие как `:nth-of-type`/`:nth-child`/`:parent`/`:not`/`:checked`, jQuery имеет свою реализацию этих селекторов для браузеров, не поддерживающих `querySelectorAll`, или для браузеров, в которых `querySelectorAll` не поддерживает данный селектор, но эта реализация иногда отличается.

Для примера возьмем псевдо-класс :nth-of-type и выберем все четные дивы, а из них все нечетные.

```
document.querySelectorAll("div:nth-of-type(even):
nth-of-type(odd)")
// Safari/FireFox:0  IE/Opera:N/A
$("div:nth-of-type(even):nth-of-type(odd)");
// Safari/FireFox:0  IE/Opera:All
$("div:even:odd"); // All: вернут 1,5,9 дивы
```

Первых два примера работают одинаково и вернут либо 0, если отработала функция querySelectorAll (это касается первого примера), либо все элементы, потому что их обработал Sizzle (это особенность реализации выражения «:»). Третий же вернет 1, 5, 9 и т. д. элементы, а значит, селекторы отрабатывали в три прохода: сначала из всего DOM-дерева были выбраны все дивы, потом из них были выбраны все нечетные, а потом из оставшихся были выбраны все четные.

jQuery также имеет набор псевдо-селекторов, которые не входят в CSS3 и обслуживаются только Sizzle'ом :visible/:animated/:input/:header. Их лучше выделять отдельно, поскольку они могут сильно замедлить выборку. Так, например, было с селекторами :visible/:hidden в версии 1.2.6: для того чтобы узнать, видимый это элемент или нет, надо было подняться до самого верха по DOM-дереву, проверяя атрибуты display и visible каждого родителя (http://mabp.kiev.ua/2009/02/07/accelerates-selectors-in-jquery/).

```
$("div").filter(":visible")
```

Псевдо-классы, используемые для поиска элементов формы, такие, как :radio, тоже имеют некоторое преимущество, если не применяется querySelectorAll; в противном случае CSS3-селектор input[type=radio] работает быстрее.

Сложенный селект

Сложенный селект — это когда нам надо выбрать группу из двух или более разных селекторов, например, все дивы, у которых класс равен A, B и C.

Это можно сделать двумя способами:

```
$(".a,.b,.c")
```

выбрать все сразу

```
$(".a").add(".b").add(".c")
```

или по одному.

При этом, если задействована функция querySelectorAll, то первый способ быстрее второго в четыре раза, а если нет, то второй в два раза быстрее первого.

Если уже заговорили про классы, их можно искать как любые другие атрибуты — например, если надо найти все классы, имена которых начинаются на "my", можно сделать так:

```
$("[class^=my]")
```

а не городить логику с использованием add, тем более что такой способ поддерживается querySelectorAll (http://mabp.kiev.ua/2009/02/21/testing-productivity-jquery-selectors/).

Неправильный селект в контексте

На сайте tvidesign.co.uk в одной очень популярной статье «Improve your jQuery — 25 excellent tips» написано, что селект лучше делать в контексте, и приведен вот такой пример:

```
$('#listItem' + i, $('.myList'))
```

Рассмотрим подробнее: контекст — это то, где ищут селектор, значит, пример можно переписать в более наглядную, но менее читаемую форму:

```
$($(".myList")).find("#listItem")
```

При этом контекст от первого поиска будет являться document.

```
$($(".myList",document)).find("#listItem")
```

Еще раз перепишем согласно формуле

```
$($(document).find(".myList")).find("#listItem")
```

И наконец, раскроем скобки

```
$(document).find(".myList").find("#listItem")
```

Что же получается: мы выполняем дорогостоящую операцию поиска по имени класса (по всему DOM-дереву в худшем случае) для того, чтобы упростить и без того самую простую операцию поиска по id?

Правильный селект в контексте

Правильно делать «с точностью до наоборот». В контексте надо указывать `id` элемента.

```
$(".class",$("#id"))
```

Однако можно не передавать в контекст jQuery объект, вполне достаточно

```
$(".class","#id")
```

Это можно переписать как

```
$("#id").find(".class")
```

Можно еще больше ускорить работу, если искать вот таким способом:

```
$(document.getElementById("id")).find(".class")
```

Но это, скорее всего, будет уже дурным тоном. Хотя поэкспериментировать интересно: что, если вместо `getElementById` взять `querySelectorAll`?

```
$("div",document.querySelectorAll("#id"))
```

Это примерно то же самое, что и

```
$("div",[document.getElementById("id")])
```

Ни прироста производительности, ни красоты кода из этого не получить, поэтому советую в контекст передавать что-то простое вроде id или при использовании псевдо-селекторов, обрабатываемых Sizzle'ом, передавать их в селектор а все остальное в контекст:

```
$(":visible","input[type=checkbox]")
```

Раз уже пошла речь о псевдо-селекторах, то

```
$(":checkbox")
```

быстрее чем

```
$("input[type=checkbox]")
```

без использования querySelectorAll и наоборот.

Сложный селект

Часто возникает задача найти всех потомков одного родителя. Если нам известно, что все потомки являются непосредственными, то есть детьми, на этом можно сэкономить. Конечно же, лучше всего было бы написать правильный селектор

```
$("#id > div")
```

Но если выборка уже есть, то будем использовать ее как контекст. Как мы уже выяснили, поиск в контексте происходит при помощи функции find:

```
$("#id").find("> div")
```

Но find — очень дорогая функция, она просматривает абсолютно всех потомков контекста, поэтому лучше применить функцию children, она просматривает только непосредственных потомков.

```
$("#id").children("div")
```

Есть еще ряд функций поиска и манипуляций, которых стоит избегать без крайней необходимости, — это `find`, `closest`, `wrap`, `wrapInner`, `replaceWith`, `clone`. Стоит заметить, что `wrapAll` сюда не входит (http://mabp.kiev.ua/2009/03/29/jquery-profiling/).

8.7.2. Кэш

Внутреннее кэширование

Кэш у jQuery крайне неразвит, если не сказать отсутствует, поэтому кэшируется только предыдущий элемент, выбранный в цепочке. Это можно наглядно рассмотреть на двух примерах.

Ситуация следующая: вы работаете со списком, у вас есть один из элементов `li`. Для того, чтобы получить все элементы, включая текущий, надо выбрать всех братьев (все, у кого родитель — это родитель текущего) этого элемента и добавить его самого:

```
$("#id").siblings().add("#id")
```

Так как он — прошлый элемент, с которым работали в цепочке вызовов, мы можем взять его из КЭШа:

```
$("#id").siblings().andSelf()
```

Конечно, в данном конкретном случаи быстрее было бы сделать

```
$("#id").parent().children()
```

Потому что siblings — это и есть выбор всех детей родителя. Но принцип использования этот пример иллюстрирует нормально.

Второй пример использования кэша — это простой возврат к предыдущей выборке, вместо того чтобы размазывать код на три строчки:

```
var elt = $("#id");
elt.children().css({/**/})
elt.click();
```

Можно после работы с детьми вернуться обратно к родителю и работать с ним дальше:

```
$("#id").children().css({/**/}).end().click()
```

Кэширование селекторов

Поскольку кэш так слабо развит, селекторы нужно кэшировать вручную. Давайте рассмотрим, например, вот такой код:

```
for(var i=0;i<1000;i++)
  $("ul").append("<li>"+i+"</li>")
```

Все работает и выглядит красиво, но и это можно оптимизировать. Если вынести выборку за пределы цикла, добавление новых элементов будет проходить быстрее:

```
var elts = $("ul");
for(var i=0;i<1000;i++)
  elts.append("<li>"+i+"</li>")
```

Буферизация

Но этот код можно заставить работать еще быстрее! Каждый раз, делая append, мы заставляем обновиться DOM-дерево и заставляем браузер

перерисовать страницу. Этого можно избежать, придерживая вставку в DOM-дерево.

```
var str = "";
for(var i=0;i<1000;i++)
  str += "<li>"+i+"</li>"
$("ul").html(str);
```

Дело в том, что функции для работы с DOM-деревом у jQuery самые «тяжелые» (http://mabp.kiev.ua/2009/03/29/jquery-profiling/). Это объясняется просто. Все `html`-ноды, на которые повешены события через jQuery, имеют в себе атрибут с объектом jQuery. При удалении этих нод нужно следить, чтобы не было утечек памяти, и удалять эти атрибуты перед удалением ноды. В результате функции `html` и `text` вызывают функции полной очистки и только потом вставки нового содержимого:

```
jQuery(DOMElement).empty().append(text)
```

Функция empty выбирает все ноды и по очереди удаляет:

```
jQuery(DOMElement).children().remove()
```

А функция remove уже заботится, чтобы из элементов были удалены все дополнительные данные и события.

Джон Ресиг утверждал, что знает способ быстро удалить все это и что улучшит эти методы, но что-то воз и ныне там. Поэтому будем ждать улучшенных функций уже в jQuery 1.4.

Создание «на лету»

Прошлый пример, на самом деле, был нужен для того, чтобы подобраться поближе к интересному факту. Часто приходится создавать какие-то вспомогательные дивы, и, естественно, нас интересует самый эргономичный способ это сделать. Казалось бы, в чем проблема: кинул кусок `html`-кода, и jQuery сама все сделала. Возьмем самый простой и банальный пример: надо создать пустой див:

```
$("<div></div>")
```

или

```
$("<div/>")
```

Второй вариант в 5 раз быстрее первого. Но это, естественно, не все: если нам надо создать не пустой див, а содержащий текст, из прошлых заметок станет ясно, что функция `text` тяжелая, и выгоды от нее не будет, и стоит создавать див «как есть».

```
$("<div>text</div>")
```

И не создавать, а потом добавлять текст:

```
$("<div/>").text("text")
```

Но это не касается создания атрибутов, для них используются намного более «легкие» функции attr/css/addClass (http://mabp.kiev.ua/2009/03/29/jquery-profiling/), вот тут-то и имеет смысл вместо

```
$("<div style='background:red;'/>")
```

писать

```
$("<div/>").css({background:'red'});
```

— это даст небольшой, но выигрыш.

8.7.3. События

Множественные события

Иногда возникает необходимость повесить одно и то же действие на несколько событий, что приводит к созданию новых функций либо к копированию исходного кода. Этого легко можно избежать. Например, нужно задать размер дива при загрузке и изменять его при изменении размеров окна:

```
$(window).bind("resize load",null,function(){
  $("#id").css({width:document.clientWidth})
});
```

Только при этом не забываем, что поведение IE8 не соответствует стандартам, и при загрузке страницы сначала происходит событие resize, а только потом load.

То же самое корректно и в обратную сторону:

```
$(window).unbind("resize load");
```

Но это не работает в версии 1.2.6, точнее, работает только с именованными функциями, а с анонимными не годится, их надо удалять по одной.

Одно событие на много элементов

Если случается повесить события на длинный список:

```
var ul = $("<ul/>");
for(var i=0,j=1000;i<j;i++)
  $("<li>"+i+"</li>").click(function(e){
    alert(this.innerHTML);
  }).appendTo(ul);

ul.appendTo("body");
```

то в результате мы будем иметь 1000 одинаковых обработчиков событий. Вряд ли это добавит скорости нашей странице, поэтому можно воспользоваться маленькой хитростью и повесить всего один обработчик на родительский элемент:

```
var str = "";
for(var i=0,j=1000;i<j;i++)
  str += "<li>"+i+"</li>";
$("<ul/>")
  .append(str)
  .click(function(e){
    alert(e.target.innerHTML);
  })
  .appendTo("body");
```

8.8. Клиентская оптимизация для произвольного сайта

Нередко можно наблюдать ситуацию, когда производительность сайта в клиентском браузере недооценивается. Особенно часто это происходит в тех случаях, когда сайт собран «из коробки» и не специализирован под выполнения тех или иных конкретных задач. Ситуация может быть дополнительно осложнена низкоскоростным каналом связи у целевой аудитории пользователей.

В общем, все вышеописанное было справедливо для сайта PERSPEKTIVA IMPEREAL (http://www.vaclavak.ru/). Перед началом оптимизации главная страница «весила» около 500 Кб, загружала большое количество внешних скриптов (порядка 150 Кб) и ее невозможно было нормально загрузить по модему. Ниже рассказывается, какие методы были использованы для улучшения скорости загрузки, — может быть, общий ход проведения оптимизации поможет и вам точнее проанализировать ситуацию и применить требуемые действия.

8.8.1. Этап первый: анализ ситуации

Для анализа скорости загрузки в качестве стандарта стоит использовать как webo.in, так и Firebug NET Panel (http://www.getfirebug.com/). Почему два инструмента? С помощью Firebug можно достаточно точно отследить все запросы на странице из реального браузера. Однако Firebug временами не выдает всех запросов к файлам стилей и скриптов. Так же тяжело бывает с кэшированием. Для полного аналитического разбора можно адекватно использовать только анализатор скорости загрузки. После проведения проверки хорошо видно, какие файлы кэшируются (выставлено время кэша), у каких есть `ETag` или `Last-Modified`, а также, что более существенно, сразу виден потенциальный выигрыш при минимизации файлов.

С помощью визуальной оптимизации (http://webo.in/my/action/load/) удобно посмотреть, как изменится диаграмма загрузки сайта, если применить все оптимизационные меры. И поскольку расчет производится аналитически, он обеспечивает достаточно большую точность (не нужно замерять по два-три раза, чтобы избежать случайных сетевых задержек).

Разобрав основные проблемные места сайта с помощью указанных инструментов, намечаем путь действий — и вперед. Дополнительно можно оценить время ответа с сервера (http://webo.in/my/action/timings/) для динамических файлов и понять, нужно ли что-то придумывать для оптимизации серверной части.

8.8.2. Этап второй: базовые действия

Базовые действия по оптимизации чрезвычайно просты: нам нужно объединить все текстовые файлы и применить для них gzip-сжатие. А также включить кэширование на достаточно длительный срок (это позволит значительно ускорить по крайней мере открытие последующих страниц на этом сайте). Все это делается несколькими строками в конфигурационном файле Apache (`httpd.conf` или `.htaccess`):

```
AddOutputFilterByType DEFLATE text/html
AddOutputFilterByType DEFLATE text/xml
AddOutputFilterByType DEFLATE image/x-icon
AddOutputFilterByType DEFLATE text/css
AddOutputFilterByType DEFLATE application/x-javascript

BrowserMatch ^Mozilla/4 gzip-only-text/html
BrowserMatch ^Mozilla/4\.0[678] no-gzip
BrowserMatch Konqueror no-gzip
BrowserMatch \bMSIE !no-gzip !gzip-only-text/html

Header append Vary User-Agent

<FilesMatch .*\.(css|js|php|phtml|shtml|html|xml)$>
  Header append Cache-Control private
</FilesMatch>

ExpiresActive On
ExpiresDefault "access plus 1 month"

<FilesMatch .*\.(shtml|html|phtml|php)$>
  ExpiresActive Off
</FilesMatch>
```

Данный фрагмент кода включает сжатие для тех типов файлов, для которых это разумно сделать, затем отключает сжатие для старых и неподдерживаемых браузеров. Заголовки Vary User-Agent и Cache-Control private нужны для корректной обработки сжатия на этапе локальных прокси-серверов (чтобы они пропускали название браузера, не кэшировали сжатую версию и не отдавали ее тем пользовательским агентам, которые это не поддерживают). Далее на все файлы накидывается кэширование на один месяц, а затем для динамических файлов (HTML) оно отключается.

Объединение и минимизация файлов может быть достаточно проблематичным занятием, но для этой цели можно использовать пакетную оптимизацию (http://webo.in/my/action/packet/), загрузить один архив со всеми файлами, которые требовалось оптимизировать, — и на выходе получить уже готовые к применению версии.

8.8.3. Этап третий: шаманим с изображениями

Следующим шагом по соотношению «эффективность/сложность» будет работа с изображениями. Начать лучше всего с создания CSS Sprites. Стоит

руководствоваться уже описанными в главе 4 принципами и соответствующей частью из книги «Разгони свой сайт». Естественно, что при создании CSS Sprites нужно будет видоизменять стили для страницы, но обычно это достаточно несложно при должной сноровке (либо можно использовать автоматическое решение — http://sprites.in/). В любом случае рекомендуется всегда делать резервные копии изменяемых файлов — это позволит быстро понять, где допущена ошибка, и не менее быстро ее исправить.

Попутно все GIF-изображения переводятся в PNG-формат. Для этого архив с изображениями прогоняется через всю пакетную оптимизацию. Отдельным пунктом для сайта www.vaclavak.ru стоит упомянуть работу с анимированными GIF. В исходной точке все банеры занимали порядка 180 Кб. При оптимизации повезло (удалось удалить ненужные кадры и немного уменьшить палитру), в итоге размер рекламного блока сократился вдвое.

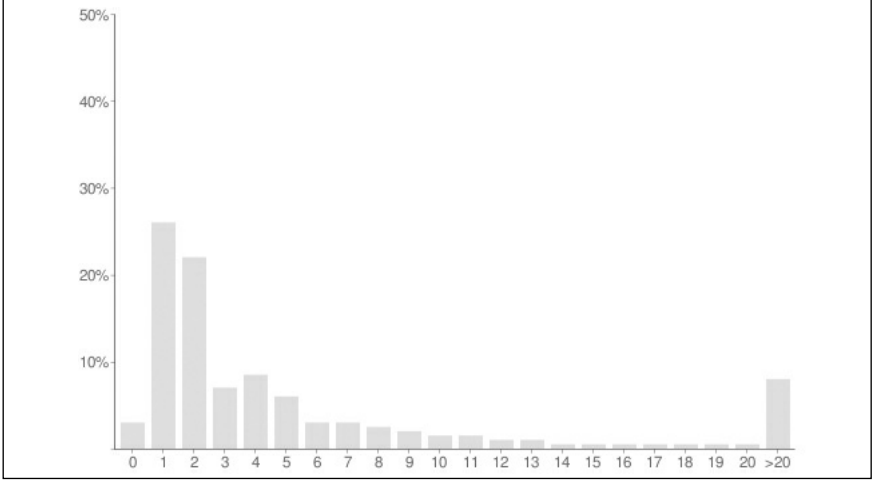
В результате проведенных действий размер страницы уменьшился до 350 Кб, что было бы совсем неплохо, учитывая изначально «тяжелый» дизайн сайта.

Однако этого оказалось недостаточно.

8.8.4. Этап четвертый: счетчик времени загрузки

Для более точного анализа реальной картины можно воспользоваться счетчиком времени загрузки (более подробно он описан в практичес-

Рис. 8.6. Скорость загрузки страниц на четвертом этапе клиентской оптимизации для www.vaclavak.ru, источник: webo.in



ком приложении в книге «Разгони свой сайт»), который замеряет полное время загрузки у всех посетителей сайта для произвольной страницы. Установка его чрезвычайно проста: нужно получить код, установить его максимально близко к началу страницы в шаблоне (идеально — сразу после `title`) и пару дней собирать статистику (зависит от числа хитов; если на сайт заходит несколько десятков тысяч ежедневно, то хватит и дня или пары часов).

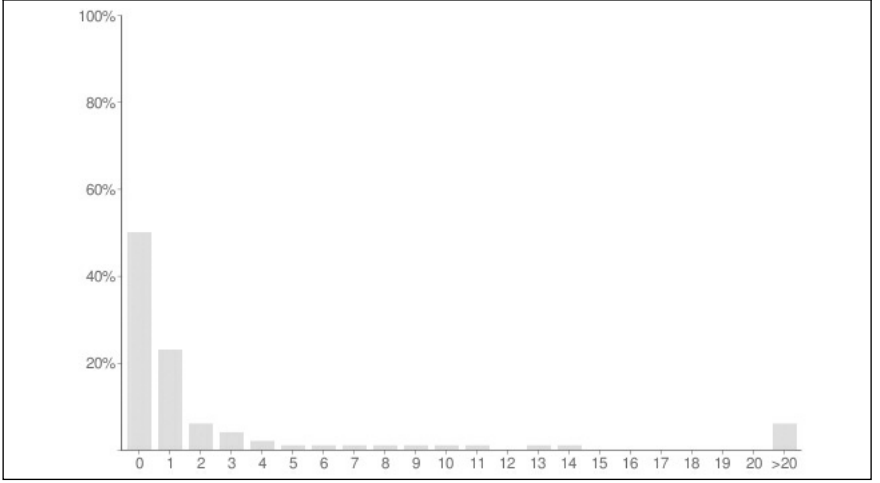
После установки счетчика обнаружилась следующая картина: среднее время загрузки (уже оптимизированного) сайта составило 16 секунд, за 4 секунды страницы загружались у 58% пользователей.

8.8.5. Этап пятый: «ненавязчивая» реклама

Было решено существенно переработать логику загрузки страницы: вынести рекламные блоки в пост-загрузку, а загрузку самого JavaScript «отложить». Все это было сделано в лучших традициях «ненавязчивого» JavaScript, которые описаны в седьмой главе книги «Разгони свой сайт». Google Analytics был вынесен на событие `OnDOMReady` (для точности измерения посещений), остальные — на `window.onload`.

В результате на странице сразу загружалось только основное содержание. Сразу после него — Google Analytics и дополнительные библиотеки для галереи изображений (но только на тех страницах, где это было

Рис. 8.7. Скорость загрузки страниц на пятом этапе клиентской оптимизации для www.vaclavak.ru, источник: webo.in



нужно: в частности, это не требовалось для главной страницы). Затем шла загрузка рекламных банеров, информера с курсом валют и погоды. В самом конце на страницу (в самый низ) вставлялся блок с остальными счетчиками. Загрузка почти всего (99% кода) JavaScript была переделана на «ненавязчивый» манер.

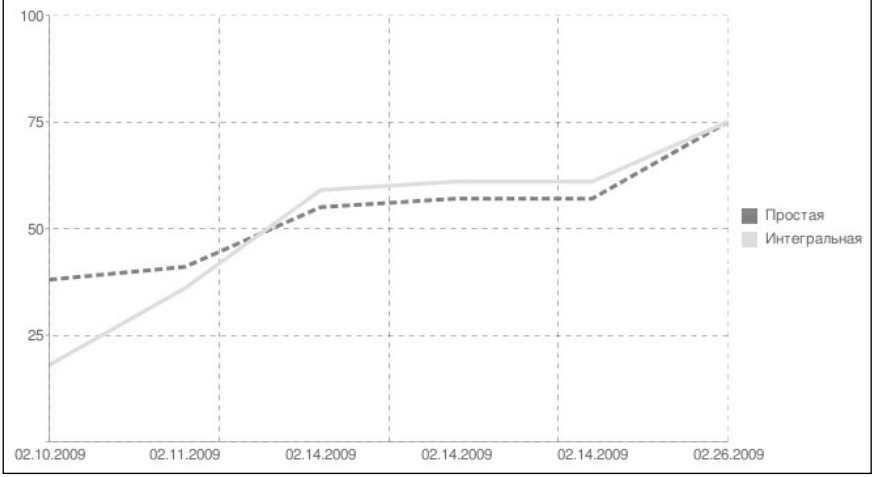
Единственная сложность возникла с блоком валют — он вызывался через внешний JavaScript-файл, который в свою очередь содержал `document.write`. Его пришлось вставлять через динамический `iframe`, который получал данные, а потом отправлял родительской странице. Поскольку информер этот загружается достаточно медленно, получился значительный выигрыш во времени загрузки страницы после его вынесения в постзагрузку.

После очередных замеров с помощью счетчика времени загрузки получилась следующая картина: среднее время загрузки — 7,3 секунды (ускорение более 100% от уже оптимизированного варианта), за 4 секунды сайт загрузился у 82% посетителей.

8.8.6. Заключение

Благодаря существованию большого количества удобных инструментов для клиентской оптимизации (на http://webo.in/) удалось очень оперативно проработать клиентскую составляющую обычного в общем-то сайта. Только для широкополосного подключения скорость загрузки уве-

Рис. 8.8. История применения клиентской оптимизации для www.vaclavak.ru



личилась в 7 раз (для коммутируемого доступа это число еще более значительно в связи с существенным уменьшением числа запросов, необходимых для первичного отображения страницы).

Сам процесс оптимизации относительно оценки самого webo.in можно представить следующим образом (в этом уже может помочь история проверок сайта, http://webo.in/my/action/history/):

Возможно, не все из описанных шагов необходимы для среднестатистического сайта, рассчитанного на рядовых пользователей Интернета, но сам процесс оптимизации (от простого к сложному, от более эффективного — к менее эффективному) должен, по-видимому, протекать именно таким образом.

Заключение

В качестве послесловия

В последние годы мир веб-разработки стал существенно тяготеть к клиентским приложениям. Браузеры стали настолько быстрыми и приобрели такое феноменальное количество возможностей, что клиентская сторона стала не проще, а даже иногда намного сложнее, чем серверная составляющая.

Именно на фоне увеличившегося внимания к клиентской составляющей веб-сайтов и зародилась ее оптимизация, оптимизация скорости загрузки, отображения и функционирования веб-сайтов в браузерах конечных пользователей. На данный момент направление это новое и весьма перспективное для изучения и прикладного использования.

Чтобы облегчить погружение в мир клиентской оптимизации, мы и создали эту книгу, которая является не только продолжением издания «Разгони свой сайт», ставшего классикой в кругах веб-разработчиков и интернет-маркетологов, но и своеобразным идеологическим прорывом. Мы постарались собрать на этих страницах материал, который раскрывает глубже уже отмеченные в первой книге аспекты и идет дальше, содержит больше практических приемов, конфигураций и советов на ближайшее будущее.

Команда авторов вложила в «Реактивные веб-сайты» весь опыт, накопленный годами практики и внедрений различных решений для сайтов различной сложности. Мы будем рады, если этот опыт окажется полезным не только нам, но сможет сделать и ваши сайты максимально быстрыми.

Также мы очень надеемся, что если вы только прикоснулись к миру клиентской оптимизации, то первое впечатление осталось благоприятным и вы не раз вернетесь к материалу этой книги.

Контакты для обратной связи с авторами приведены по адресу:
http://speedupyourwebsite.ru/about/

Учебное издание

Мациевский Николай Сергеевич
Степанищев Евгений Владимирович
Кондратенко Глеб Игоревич

РЕАКТИВНЫЕ ВЕБ-САЙТЫ
Клиентская оптимизация в алгоритмах и примерах

Учебное пособие

Литературный редактор *С. Перепелкина*
Корректор *Ю. Голомазова*
Компьютерная верстка *Н. Овчинникова*
Дизайн обложки *Р. Сепеда Эррера*

Подписано в печать 25.12.2009. Формат 60x90 $^{1}/_{16}$.
Гарнитура Официна. Бумага офсетная. Печать офсетная.
Усл. печ. л. 21. Тираж 2000 экз. Заказ №

ООО «ИНТУИТ.ру»
Интернет-Университет Информационных Технологий, www.intuit.ru
Москва, Электрический пер., 8, стр.3.
E-mail: admin@intuit.ru, http://www.intuit.ru

ООО «БИНОМ. Лаборатория знаний»
Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-1902, (499) 157-5272
E-mail: lbz@aha.ru, http://www.lbz.ru

Реактивные веб-сайты

Целью данной книги является показать важность (иногда по-настоящему критическую) клиентской оптимизации и осветить ключевые моменты и проблемные места. Очень хочется верить, что после прочтения книги у читателя сложится целостное представление о мире, находящемся между страницей в браузере пользователя и серверными мощностями. О том, что происходит, когда в адресной строке браузера вводится адрес или имя сайта, какие стадии проходит загрузка страницы, прежде чем полностью завершиться. И самое главное — как этим процессом можно управлять.

Мациевский Николай Сергеевич — выпускник МФТИ, участник ЕЖЕ-движения и объединения разработчиков «Веб-стандарты». Профессиональные интересы сосредоточены в области клиентской оптимизации и практических методов по уменьшению времени загрузки веб-страниц в несколько раз. Николай работает в области клиентской части и высокой производительности с 2001 года, выступал с серией докладов на эту тему на десятках отраслевых мероприятиях.

В данный момент является генеральным директором ООО «ВЕБО» — первой в России компании, занимающейся клиентской оптимизацией, и руководит разработкой веб-приложения для автоматизации клиентской оптимизации — Web Optimizer.

Степанищев Евгений Владимирович — известный блоггер, является экспертом в области разработки веб-приложений и безопасности в интернете. Имеет отношение к крупнейшим проектам Татарстана и России, много работал с заказчиками из США, Германии, Франции, Индонезии и Арабских Эмиратов.

В настоящее время область деятельности — интранет-решения компании «Яндекс», где Евгений занимается руководством группой разработчиков внутренних сервисов. Профессиональный стаж программирования на настоящий момент — 20 лет, из них 12 — в сфере веб.

Кондратенко Глеб Игоревич — выпускник МИФИ, в настоящее время работает веб-технологом компании Акронис. Основное направление профессиональной деятельности — разработка пользовательских интерфейсов и клиентская оптимизация. Также участвует в работе над приложением Web Optimizer для автоматизации клиентской оптимизации.

ISBN 978-5-9963-0253-6