2008

Разгони свой сайт

Методы клиентской оптимизации веб-страниц

Сайт тормозит, и вы не представляете, как с этим справиться? Количество анимации на странице превысило все мыслимые и немыслимые границы, и вы не знаете что делать? На странице десятки и сотни картинок, дизайнер превзошел сам себя, и теперь все это загружается страшно медленно? Эта книга поможет разобраться с этими и множеством других проблем, связанных с клиентской производительностью.

Николай Мациевский

25.11.2008

Оглавление

Об этой книге и проекте webo.in

Вопрос скорости загрузки веб-страниц привлекает внимание всех веб-разработчиков уже очень давно — практически, с того момента, как в HTML-документе появились картинки и веб-страницы вышли на уровень взаимодействия с пользователями, а не только предоставления им необходимой информации.

За последние 10 лет уже многократно менялся сам подход к созданию сайтов. В эпоху браузерных войн и ограниченного доступа по модему наиболее важными аспектами клиентской производительности (которая отвечает за скорость загрузки и работы веб-страницы) были ускорение передачи данных и ускорение отображения этих данных при использовании многоуровневых таблиц на странице. О блочной верстке и семантической разметке тогда просто не думали.

Но ситуация изменилась. Сейчас средняя веб-страница уже крайне тяжело вписывается в установленные когда-то рамки «загрузка за 10 секунд на модеме». В среднем, на ней используются десятки различных объектов, и не всегда это только картинки. Объем файлов скриптов, обеспечивающих взаимодействия пользователя с веб-страницей, сейчас уже намного превышает размер предоставляемой на этой странице информации. И мир движется в сторону усложнения взаимодействия человека с машиной, никак не в обратную.

Данное издание старается объединить в себе все современные подходы к построению высокопроизводительных веб-приложений и просто веб-сайтов, которые быстро загружаются. Подавляющая часть материалов книги уже была опубликована в 2008 году на сайте Web Optimizator (http://webo.in/), из них были отобраны наиболее актуальные и проверенные на практике решения, которые и вошли в основу этой книги.

Кроме теоретических аспектов производительности приведено также большое количество практических рекомендаций, примеров конфигурационных файлов, различных приемов и проанализировано несколько высокопосещаемых ресурсов. Книга предназначена, с одной стороны, для последовательного погружения в сферу клиентской оптимизации веб-разработчиков любого уровня. С другой стороны, благодаря большому количеству прикладных советов, она ставит своей целью стать настольным справочником оптимизатора.

Web Optimizator

Идея организовать ресурс, посвященный как теоретическим аспектам оптимизации времени загрузки веб-страницы, так и предлагающий online-инструменты для этой самой оптимизации, появилась после обсуждения на конференции ClientSide'2007 (http://client2007.ru/), где на фоне общего интереса к затронутой проблеме была задана пара вопросов о рассмотрении частных, практических случаев.

За основу online-инструмента были взяты замечательные примеры с Web Site Optimization (http://www.websiteoptimization.com/), OctaGate SiteTimer (http://www.octagate.com/service/SiteTimer/) и Pingdom Tools (http://tools.pingdom.com/fpt/), краткий обзор которых приводится в восьмой главе. Все эти сервисы являются

англоязычными и предлагают достаточно широкий спектр инструментов для анализа скорости загрузки сайта. Однако русскоязычного сервиса на тот момент на было, и выдаваемая информация не являлась достаточно точной. Поэтому основным отличием от этих инструментов стало наличие максимально детальных советов по каждому анализируемому веб-сайту, которые, по идее, должны помочь веб-разработчикам предпринять конкретные действия для улучшения качества своего продукта.

Именно с этой целью и был создан Web Optimizator (http://webo.in/).

Благодарности

Книга не увидела бы свет без помощи, советов и рекомендаций огромного количества людей. Каждый из них добавил что-то новое в раскрываемой ниже материал, поэтому у меня просто не получится упомянуть всех, кто внес вклад в создание статей и развитие ресурса, посвященного клиентской оптимизации, — перечислить всех не представляется возможным. И все же некоторым людям хочется сказать отдельное спасибо за их замечания и поддержку.

Во-первых, хочу высказать персональную благодарность Виталия Харисову за несколько очень своевременных замечаний касательно производительность CSS-движка в браузерах, которые подвигли меня изучить эту проблему более глубоко и получить настоящий прорыв в понимании функционирования данной части браузеров.

Во-вторых, нельзя не упомянуть Павла Димитриева и его замечательный перевод классических советов от группы разработчиков Yahoo! (часть 1, http://webo.in/articles/habrahabr/15-yahoo-best-practices/), которые послужили отправной точкой в оптимизации скорости загрузки для многих тысяч веб-разработчиков.

Значительный вклад в продвижение идей «ненавязчивого» JavaScript и обратной совместимости в работе веб-сайтов (когда пользователь может взаимодействовать со страницей и с отключенным JavaScript или CSS) был внесен Дмитрием Штефлюком (http://kpumuk.info/), Андреем Миндубаевым (http://covex.in.nnov.ru/), Андреем Суминым (http://jsx.ru/), Павлом Корниловым (http://lusever.ru/), Павлом Довбушом (http://dpp.su/), Вадимом Макеевым (http://pepelsbey.net/) и Артемием Трегубенко (http://arty.name/). Их идеи легли в основу некоторых частей данной книги.

Также необходимо упомянуть Евгения Степанищева (http://bolknote.ru/), предложившего альтернативу `data:URI` подхода для Internet Explorer, что позволило расширить этот метод для всех браузеров и составить реальную конкуренцию CSS Sprites. И Александра Лозовюка (http://abrdev.com/), который смог найти время и дополнить предварительную версию рукописи своими замечаниями по использованию различных инструментов для анализа и автоматизации ускорения загрузки.

Естественно, что нельзя обойти вниманием всех моих соратников по российскому крылу Web Standards Group (http://web-standards.ru/). Не опираясь на их мощную профессиональную поддержку, освещать такую сложную и неоднозначную тему, как клиентская оптимизация, было бы весьма непросто.

Кроме этого Антон Лобовкин (http://anton.lobovkin.ru/), Денис Кузнецов (http://q-zma.com/), Евгений Кучерук (http://kuklaora.blogspot.com/), Иван Никитин (http://ivan-nikitin.spaces.live.com/), Алексей Басс (http://alexey-bass.blogspot.com/), Владимир Юнев (

http://blogs.gotdotnet.ru/personal/XaocCPS/), Павел Власов (http://zencd.livejournal.com/), Артем Курапов (http://kurapov.name/), Алексей Тен.(http://lynn.ru/), Константин Бурнаев (http://bkonst.livejournal.com/), Timur Vafin (http://timurv.ru/), Денис Воробьев (http://alfa.1nsk.ru/), Алексей Хоменко (http://core.freewheel.ru/), Паша Друзьяк (http://ilive.in.ua/enej/), Иван Курносов (http://www.mzz.ru/) и многие-многие другие поделились полезными методиками, примерами конфигурационных файлов и просто ценными советами. За что им также огромное спасибо.

И конечно, обязательно хочу поблагодарить всех пользователей Web Optimizator (http://webo.in/) и читателей блога Клиентская оптимизация (http://habrahabr.ru/blogs/client_side_optimization/). Без их моральной поддержки и веры в благое начинание эта книга никогда бы не была опубликована.

1.1. Цели и задачи оптимизации

Каждая веб-страница состоит из основного HTML-файла и набора внешних ресурсов. Говоря о размере страницы (или сайта), очень часто имеют в виду размер именно первого файла, что, естественно, неверно.

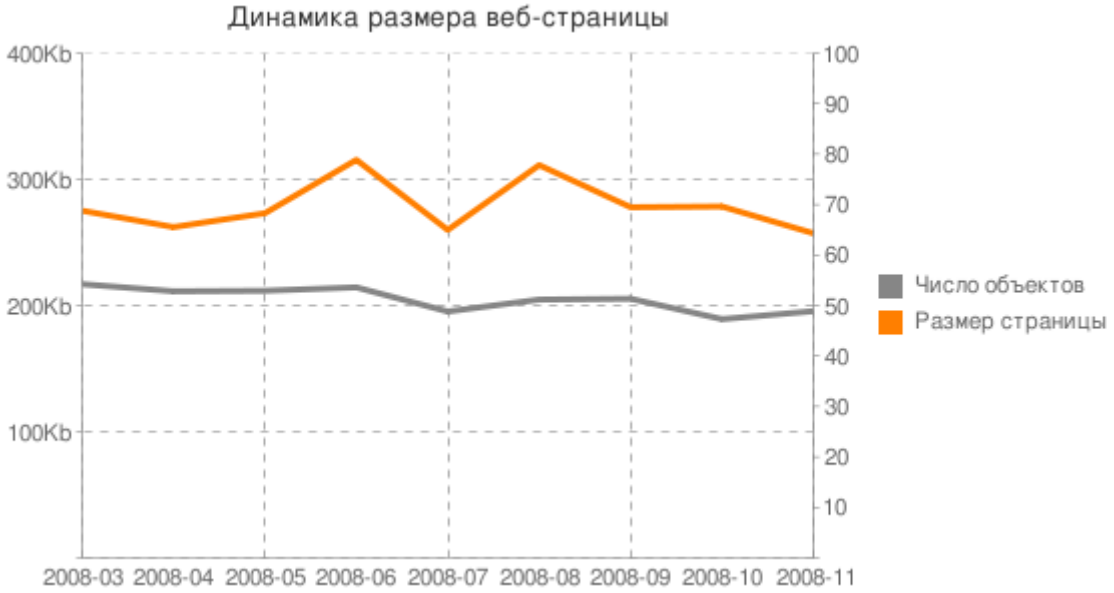


Рис.1. Тенденция изменения размера страницы и числа объектов для сайтов, проверяемых через Web Optimizator в 2008 году

В настоящее время на каждой странице вызывается несколько десятков внешних объектов, а размер исходного файла составляет не более 5% от общего размера. Как показали многочисленные исследования, размер веб-страницы за последние 5 лет увеличился втрое, а число объектов на ней — почти в два раза. При этом темпы роста средней пропускной способности канала лишь немного выше данных показателей. Если учитывать расслоение пользователей по скоростям доступа, то стремление уменьшить число пользователей, превышающих допустимый порог ожидания на 1–5%, заставляет применять все более сложные и передовые технологии.

Естественно, что технологии эти не ограничиваются сжатием текстовых (HTML, CSS, JavaScript) файлов на стороне сервера. Как несложно понять, основную часть внешних объектов на странице составляют изображения или мультимедийные файлы. И для них тоже есть свои методы оптимизации.

Основные задачи оптимизации

Если говорить кратко, то можно выделить 3 основные задачи клиентской оптимизации:

- Оптимизация размера файлов
- Оптимизация задержек при загрузке
- Оптимизация взаимодействия с пользователем

Краткий обзор технологий

При этом все основные методы можно разбить на 6 групп (каждая из которых позволяет решить одну из заявленных задач).

- Уменьшение размера объектов. Здесь фигурируют сжатие и методы оптимизации изображений, подробнее можно ознакомиться во второй главе.
- Особенности кэширования, которые способно кардинально уменьшить число запросов при повторных посещениях, раскрываются в третьей главе.
- Объединение объектов. Основными технологиями являются слияние текстовых файлов, применение CSS Sprites или `data:URI` для изображений. Этому посвящена четвертая глава книги.
- Параллельная загрузка объектов, влияющая на эффективное время ожидания каждого файла. В пятой главе помимо этого приведены примеры балансировки запросов со стороны клиентского приложения.
- Оптимизация CSS-производительности, что проявляется в скорости появления первоначальной картинки в браузере пользователя и скорости ее дальнейшего изменения. О CSS-производительности рассказывает шестая глава.
- Оптимизация JavaScript. Есть достаточно много проблемных мест в JavaScript, о которых необходимо знать при проектировании сложных веб-приложений. Обо всем этом можно прочитать в седьмой главе.

Хочется отметить, что, несмотря на всю сложность затрагиваемой темы, первоначального ускорения загрузки веб-страницы можно добиться в несколько очень простых шагов. При этом можно сократить время появления веб-страницы в несколько (обычно 2–3) раз.

Все советы в книге упорядочены по увеличению сложности внедрения и уменьшению возможного выигрыша при загрузке страницы. Для простых веб-проектов можно ограничиться только включением кэширования и архивирования (`gzip` или `deflate`). Более сложным понадобится изменить верстку, используя CSS Sprites или `data:URI`, и добавить несколько хостов для загрузки изображений. Для высоконагруженных проектов (некоторые из них проанализированы в конце восьмой главе) нужно учитывать все аспекты клиентской оптимизации с самого начала при проектировании и применять их последовательно для достижения наилучшего результата.

1.2. Психологические аспекты производительности

Согласно многочисленным исследованиям пользовательское раздражение сильно возрастает, если скорость загрузки страницы превышает 8–10 секунд безо всякого уведомления пользователя о процессе загрузки. Последние работы в этой области показали, что пользователи с широкополосным доступом еще менее терпимы к задержкам при загрузке веб-страниц по сравнению с пользователями с более узким каналом.

В проведенном в 2007 году опросе было установлено, что 33% пользователей скоростного соединения не хотят ждать более 4 секунд при загрузке страницы, при этом 43% пользователей не ждут более 6 секунд. В данном случае имеется в виду, что пользователь в большинстве случаев покинет сайт, если в течение 5–10 секунд будет видеть вместо него белый экран в браузере.

Терпимое время ожидания

В исследовании, проведенном в 2004, было установлено, что терпимое время ожидания для неработающих ссылок (без обратной связи) находилось между 5 и 8 секундами. С добавлением уведомления пользователя о процессе загрузки (обратной связи), например, индикатора загрузки, такое время ожидания увеличилось до 38 секунд. Распределение времени для повторных попыток зайти на неработающие ссылки имело максимум в районе 23 секунд (без наличия каких-либо индикаторов, что страница загружается или в данный момент недоступна).

Таким образом, можно заключить, что для 95% пользователей время ожидания ответа от неработающего сайта составит не более 8 секунд. Если учесть стремление пользователя посетить сайт повторно, то исследования продемонстрировали крайне малое (почти равное нулю) число пользователей, ждущих более 10 секунд.

Эффекты медленной скорости загрузки

Даже малые изменения времени загрузки могут иметь значительные последствия. Так, для Google (http://www.google.com/) увеличение времени загрузки для страницы с 10 поисковыми результатами на 0,4 секунды и на 0,9 секунд для страницы с 30 результатами сказалось на уменьшении трафика и рекламных доходов на 20% (в соответствии с исследованиями, проведенными в 2006 году). Когда главную страницу Google Maps (http://maps.google.com/) уменьшили в объеме с 100 Кб до 70–80КБ, трафик увеличился на 10% в течение первой недели и еще на 25% в следующие три недели (по данным 2006 года).

Тестирование в 2007 году для Amazon дало очень близкие результаты: каждые 100 мс увеличения времени загрузки для Amazon.com уменьшали продажи на 1%. Эксперименты Microsoft для Live Search (http://www.live.com/) показали, что при замедлении загрузки страниц на 1 секунду количество сброшенных поисковых запросов возросло на 1% и число кликов по рекламе уменьшилось на 1,5%. При увеличении времени загрузки страницы с результатами еще на 2 секунды количество сброшенных поисковых запросов возросло на 2,5% и число кликов по рекламе уменьшилось на 4,4%.

Как время ответа сайта влияет на пользовательскую психологию

Пользователи ощущают, что сайты, которые загружаются медленно, менее надежны и менее качественны. В том случае, если удерживать время загрузки в пределах «терпимого», пользователи буду ощущать гораздо меньше неудовлетворенности от посещения сайта, среднее число просмотров страниц возрастет, увеличится конверсия посетителей и снизится число отказов. Сайты, которые быстро загружаются, также кажутся пользователям более интересными и привлекательными.

«Терпимое» время будет сильно зависеть от аудитории, но его можно достаточно надежно проверить: для этого нужно значительно (например, в 2–3 раза) увеличить (или уменьшить) время задержки при показе страницы и посмотреть на число отказов (число пользователей, закрывших страницу сразу после захода на сайт) и на число постоянных посетителей. Если при сильном увеличении (или уменьшении) задержки при загрузке сайта количество пользователей практически не изменилось, значит, страница уже загружается в допустимом диапазоне. Если же число пользователей претерпело видимые изменения, то, следовательно, со временем загрузки сайта нужно что-то делать.

Пользователи широкополосного доступа ожидают большую скорость загрузки, при этом пользователи с менее скоростным доступом остаются далеко позади. По мере того как высокоскоростной доступ проникает в массы, растет также и размер страницы. Пользователи испытывают психологические и физиологические проблемы при взаимодействии с веб-страницами, чувствуют раздражение, если не могут завершить свои задачи, и воодушевление при работе с быстрыми сайтами.

1.3. Стадии загрузки страницы

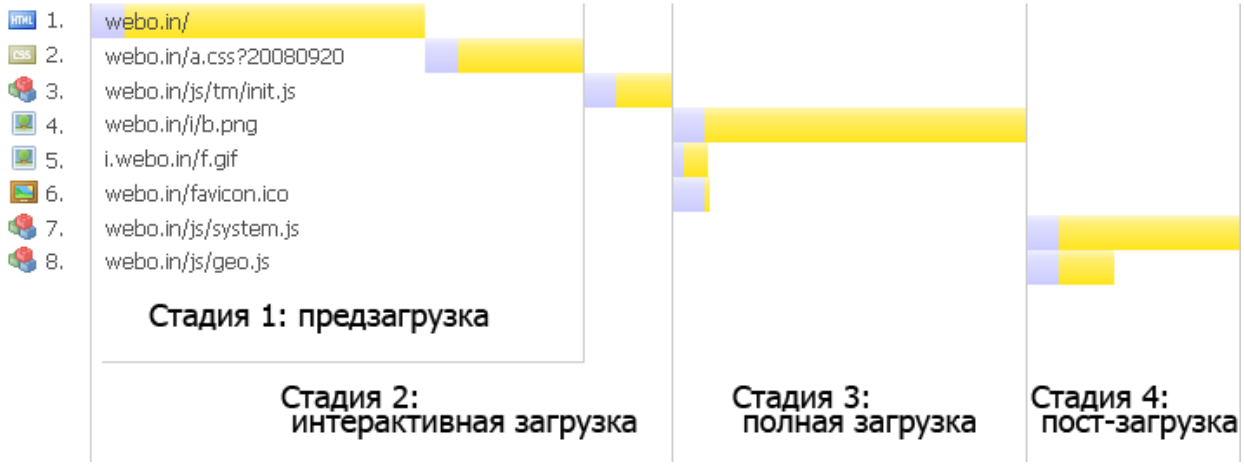


Рис.2. Стадии загрузки страницы

В качестве основных проблемных мест при загрузке страницы любого веб-ресурса можно выделить четыре ключевых момента.

1. **Предзагрузка** — появление страницы в браузере пользователя. После некоторого времени ожидания загрузки при заходе на веб-ресурс у пользователя в браузере отображается нарисованная страница. В этот момент, вероятно, на странице отсутствуют рисунки и, скорее всего, не полностью функционирует JavaScript-логика.
2. **Интерактивная загрузка** — появление интерактивности (и анимации) у загруженной веб-страницы. Обычно вся клиентская логика взаимодействия доступна сразу после первоначальной загрузки страницы (стадия 1), однако в некоторых случаях (о них речь пойдет чуть дальше) поддержка этой логики может (и должна, на самом деле) немного запаздывать по времени от появления основной картинки в браузере пользователя.
3. **Полная загрузка страницы**. Страница полностью появилась в браузере, на ней представлена вся заявленная информация, и она практически готова к дальнейшим действиям пользователя.
4. **Пост-загрузка страницы**. На данной стадии полностью загруженная страница может (в невидимом для пользователя режиме) осуществлять загрузку и кэширование некоторых ресурсов или компонентов. Они могут потребоваться пользователю как при переходе на другие страницы данного сайта, так и для отображения каких-либо анимационных эффектов или добавления функционала ради удобства использования.

Для большинства сайтов на данный момент стоит различать только предзагрузку (в которую по умолчанию включается интерактивная загрузка) и полную загрузку страницы. Пост-загрузка, к несчастью, сейчас используется крайне мало.

Расставляем приоритеты

Оптимизация скорости загрузки веб-страницы сосредоточена на двух ключевых аспектах: ускорение предзагрузки и ускорение основной загрузки. Все основные методы сфокусированы именно на этом, потому что «загрузка» веб-страницы воспринимается пользователями как нечто находящееся посредине этих двух стадий.

В идеале, загрузка страницы для пользователя должна заканчиваться сразу после предзагрузки, однако, добиться этого весьма сложно, и оправдано это далеко не во всех случаях. Подробнее о методах экстремальной оптимизации будет рассказано в конце четвертой главы.

Узкие места

Первая и вторая стадия загрузки являются наиболее проблемными аспектами при анализе производительности. Это вполне понятно: загрузка первоначального HTML-файла, равно как и CSS- / JavaScript-файлов идет в один поток, — и на первое место выходит уменьшение числа запросов при загрузке.

Как только узкое место преодолено (в идеале, у нас должен быть один-единственный файл, который получает пользователь) и в браузере страница отобразилась, мы можем начать запрашивать с сервера все остальные ресурсы. Самое главное, что это можно делать с помощью десятков дополнительных соединений (как этого добиться, рассказывается в пятой главе), ибо в браузере уже произошло событие готовности документа к дальнейшим действиям.

Мы можем настроить логику кэширования, последовательную загрузку JavaScript-модулей или даже пост-загрузку стилевых правил. Все это уже будет слабо отражаться на фактической скорости первоначальной загрузки: пользователь видит страницу в браузере, может с ней взаимодействовать (пусть даже сначала и не в полном объеме), для него она уже загрузилась (правда, только с психологической, а не с технической стороны).

Но все эти приемы могут ускорить как загрузку следующих для пользователя страниц, так и упорядочить саму пост-загрузку. Как достичь этого эффекта и как распределить файлы и клиентскую логику между стадиями загрузки страницы, рассказано в четвертой главе.

1.4. Клиентская и серверная оптимизация: сходство и различия

Клиентская оптимизация оперирует двумя основными принципами: меньше данных и меньше соединений. Но именно эти принципы помогают уменьшить нагрузку на сам

сервер. Давайте посмотрим, как это происходит и как перенести часть серверной нагрузки на клиентский браузер.

Кэширование во главу угла

Сервер может управлять состоянием кэша клиентского браузера, во-первых, через заголовок `Cache-Control` (и его атрибуты `max-age`, `pre-check`, `post-check`), который может указывать на промежуток времени, в течение которого соответствующий файл следует хранить на диске и не запрашивать с сервера. Рекомендуется для всех статических файлов выставлять максимальное время жизни кэша и форсировать его обновление у пользователя через изменение URL ресурса (с помощью `RewriteRule` либо GET-параметра).

Во-вторых, состоянием клиентского кэша можно управлять через заголовки `ETag` и `Last-Modified`, которые ставят в соответствие каждому файлу уникальный идентификатор, изменяющийся при изменении файла, — своеобразная цифровая подпись или хэш. При этом серверу нужно не пересылать файл заново, а лишь ответить статус-кодом 304 на запрос браузера, если файл не изменился с момента последнего запроса. В итоге сам файл не пересылается, соединение (и сокет) освобождается быстрее, и ресурсы сервера также экономятся.

Подробнее о кэшировании рассказывается в третьей главе.

Меньше запросов — легче серверу

Используя объединение файлов, мы не заставляем сервер обмениваться с браузером заголовками для передачи, например, нескольких таблиц стилей — гораздо экономичнее будет их объединить в одну. При этом браузер быстрее получит всю необходимую информацию и быстрее освободит такой важный ресурс, как соединение.

Наряду с объединением текстовых файлов не стоит пренебрегать и объединением картинок. Если учитывать, что современные браузеры могут устанавливать несколько десятков одновременных соединений с сервером для получения статических файлов (и 80% из них — это именно картинки), то экономия от использования CSS Sprites, Image Map или `data:URI` подхода рассчитывается очень просто. В некоторых случаях удается уменьшить число соединений браузера с сервером для загрузки одной HTML-страницы в 8–10 раз.

Объединение файлов рассматривается в четвертой главе.

Архивировать и кэшировать на сервере

Как показали проведенные исследования, `gzip`-сжатие текстового файла «на лету» в 95–98% случаев позволяет сократить время на передачу файла браузеру. Если хранить архивированные копии файлов на сервере (в памяти proxy-сервера или просто на диске), то соединение в общем случае удается освободить в 3–4 раза быстрее.

В случае высоконагруженных серверов с динамическими HTML-файлами `gzip` также может быть применим. Здесь стоит ориентироваться на минимальную степень сжатия, ибо процессорные издержки при этом растут линейно, а размер уменьшается лишь логарифмически.

О сжатии рассказывает следующая глава.

Кто у кого на службе?

После проведенного обзора технологий может показаться, что клиентская оптимизация является лишь составляющей частью серверной. Однако это не так. И клиентский, и серверный подход должен присутствовать при построении высокопроизводительных веб-приложений. В этом случае можно говорить о пересекающейся области ответственности, но никак не о превалировании одной логики над другой.

Когда дело доходит до взаимодействия клиент-сервер, нужно помнить обо всех аспектах оптимизации. И у клиентской составляющей есть своя, выделанная, область ответственности. Она находится в окне браузера – это веб-страница, которая загружается у пользователя и с которой он взаимодействует.

1.5. Применение в разработке приложений

Пользователи обычно не знают, какие подходы применяются при разработке, как настроен сервер, какие клиентские и серверные средства разработки используются. Для них лишь важно, насколько сайт полезный, удобный и быстрый. Задача же веб-разработчиков заключается в том, чтобы не доставлять пользователю лишние неудобства, радовать его и, тем самым, стимулировать продажи, идущие через сайт, или число рекламных показов и кликов по ним.

Ниже рассказывается, как можно организовать создание веб-приложения, ориентируясь на самые важные аспекты клиентской оптимизации.

Этап 1: Доставка информации и оформления

На этом этапе разработчики должны сделать все возможное, чтобы не замедлить скорость загрузки страницы. Фактически, идет речь об ускорении первой стадии загрузки. Наиболее важными методами здесь является сжатие (`gzip`) текстовых файлов и объединение файлов стилей (CSS). Для CSS- и JavaScript-файлов возможно применять статическое архивирование (без необходимости архивировать каждый раз эти файлы «на лету»; этому посвящена вторая глава).

При загрузке страницы браузер запросит все CSS-файлы, объявленные в `head` страницы, последовательно. Поэтому каждый файл добавляет задержку в загрузке, равную времени запроса до сервера (даже если предположить, что устанавливаемое соединение `keep-alive` и нам не нужно совершать все TCP/IP процедуры, — в противном случае мы экономим гораздо больше). Для файлов скриптов рекомендуется применить либо также объединение, либо вообще вынести их в пост-загрузку (подробнее об этом рассказано в седьмой главе).

Итог первого этапа — это доставленный и оформленный HTML. И издержки на доставку JavaScript сведены к минимуму (на этом этапе он только мешает, поскольку замедляет отображение основного содержимого страницы). Время от начала до завершения загрузки такой страницы при включенном и выключенном JavaScript (при вынесении его в пост-загрузку), фактически, будет одинаковым. Это и будет выигрышем в скорости загрузки!

Этап 2: Кэширование файлов оформления и параллельные запросы

На данном этапе разработчики должны обеспечить быструю загрузку других страниц сайта (если посетитель решит туда перейти). Этот этап должен проходить параллельно с первым. Настройка кэширующих заголовков достаточно тривиальна. Несколько сложнее наладить процесс разработки для своевременного сброса кэша. Все эти вопросы раскрываются в третьей главе.

Одно или несколько дополнительных зеркал для выдачи статических ресурсов легко настраиваются в конфигурации, однако, внедрить это в схему публикации изменений гораздо сложнее. Обычно это делают уже после разработки макетов страниц. Число дополнительных хостов следует напрямую из числа статических файлов (обычно картинок), поэтому надо определиться с ними на этапе автоматизации процесса публикации. О параллельных запросах рассказывается в пятой главе.

CSS Sprites достаточно трудоемки в автоматической «склейке», поэтому их внедряют обычно на первом этапе (при создании макета страниц). При использовании метода `data:URI` на первом этапе о них можно забыть, потому что автоматизированное решение просто в реализации и не требует от верстальщика отдельных технологических познаний. Об этом можно посмотреть в четвертой главе.

Этап 3: Жизнь после загрузки страницы

Целью данного этапа является создание различных обработчиков событий, которые должны взаимодействовать с пользователем. Это могут быть и всплывающие подсказки, и подгрузка данных с сервера, и просто анимация. Все это можно назвать «оживлением» страницы.

Говорят, что иногда «грамм видимости важнее килограмма сути» — это как раз про JavaScript. Ведь именно на нем можно реализовать механизмы, упрощающие действия пользователя; можно сделать много различных визуальных эффектов, подчеркивающих оформление, удобство и полезность сайта (а фактически, усилить и сфокусировать всю работу, которую проделали разработчики на предыдущих этапах).

К этому моменту мы должны иметь оформленную HTML-страницу, на которой все ссылки и формы **обязаны** работать без JavaScript (как этого добиться, как разделить представление страницы от ее функционирования, рассказывается в седьмой главе в разделе про «ненавязчивый» JavaScript).

У нас должны быть готовы серверные интерфейсы для Ajax-запросов; структура страницы должна быть такой, чтобы для аналогичных кусков HTML-кода не приходилось реализовывать аналогичные, но не одинаковые куски JavaScript-кода. Скорее всего, должны быть созданы шаблоны страниц, где видно, как будет выглядеть страница после какого-то действия пользователя (обычно специалист по удобству использования создает макеты).

Чтобы не уменьшать скорость доставки контента и оформления, JavaScript-файлы (лучше всего, конечно, **один JavaScript-файл**; несколько файлов должны использоваться только при большой сложности клиентского интерфейса) должны быть подключены перед закрытием тега `body` (а в идеале — вынесены именно в пост-загрузку).

Задача по обеспечению взаимодействия пользователя с интерфейсом сайта сводится к выполнению следующих действий:

1. найти DOM-элементы, требующие «оживления» (далее — компоненты);
2. определить, что это за компонент;
3. обеспечить подключение необходимого кода JavaScipt;
4. следить за очередностью подключения файлов;
5. не позволять нескольких загрузок одного файла.

Все это напрямую следует из концепции «ненавязчивого» JavaScript, которая описана в седьмой главе.

Поиск необходимых DOM-элементов должен нам дать список названий JavaScript-компонентов. Названия компонентов должны однозначно соответствовать названиям файлов на сервере, в которых содержится код для них. Также нам может понадобиться загрузить некоторые дополнительные CSS-правила для найденных компонентов (в случае небольшого количества CSS-кода разумно будет включить его в основной файл) ради каких-то визуальных эффектов, которые можно пропустить на первом этапе загрузки. Например, все эффекты по смене изображения при наведении мыши обеспечиваются через CSS-правила и технику CSS Sprites.

Список названий компонент можно объединить в один запрос к серверу. В итоге на стадии пост-загрузки должны осуществляться запросы к файлам вида `static.site.net/jas/componentName1.css;componentName2.css` и `static.site.net/jas/componentName1.js;componentName2.js`.

У данного подхода есть два недостатка.

1. В папке `/jas/` (которую мы, например, используем для кэширования наиболее частых вариантов подключения модулей) через некоторое время может оказаться очень много файлов, что теоретически может уменьшить время доступа к ним на сервере.
2. Иногда на странице может оказаться очень много компонент, причем так много, что длина имени запрашиваемого объединенного файла перевалит за возможности файловой системы (например, 255 символов у Ext3) — в этом случае потребуется разбить один запрос на несколько последовательных.

Этап 4: Предупреждаем действия пользователя

Если после посещения главной страницы большинство пользователей проходят внутрь сайта, то логично будет после полной загрузки главной страницы запрашивать стили и скрипты, применяемые на других страницах сайта. Для пользователя это выльется в небольшое увеличение трафика (при использовании сжатия текстовая информация составляет 10–20% от объема графики), однако, во вполне заметное ускорение загрузки последующих страниц.

Аналогично можно рассмотреть и предзагрузку некоторых наиболее часто используемых картинок, которые отсутствуют на главной странице, и дополнительных JavaScript-модулей, которые применяются на текущей странице для дополнительного функционала и не запрашиваются при первой загрузке страницы (например, отвечают за первоначально скрытые блоки).

Естественно, что за балансировку третьей и четвертой стадии отвечает уже JavaScript-разработчик и фронтенд-архитектор — ведь именно в зоне ответственности последнего находится скорость загрузки страницы.

Глава 2. Уменьшение размера

2.1. Насколько ресурсоемко архивирование HTML

Архивирование (`gzip-` , `deflate`-сжатие) уже давно является наиболее известной техникой оптимизации. Однако применяют ее по-прежнему так же редко, как и 10 лет назад. Я постараюсь максимально подробно осветить проблемные аспекты при использовании сжатия на сервере и указать на возможные методы их решения.

Сжатие веб-содержимого посредством `gzip` (`GZU zip`) — это довольно старая технология. Суть ее сводится к тому, что содержимое перед отправкой пользователю сжимается по известному всем алгоритму `zip`. Сама спецификация gzip описана в RFC1952 (http://tools.ietf.org/html/rfc1952), версия 4.2 которой датируется маем 1996 года. На сегодняшний день все популярные браузеры и веб-серверы поддерживают сжатие посредством `gzip`.

Издержки на использование `mod_gzip`

Начиная с версии протокола `HTTP/1.1`, веб-клиенты указывают, какие типы сжатия они поддерживают, устанавливая заголовок `Accept-Encoding` в HTTP-запросе:

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

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

```
Content-Encoding: gzip
```

Это все замечательно в том плане, что переданные таким образом данные меньше первоначальных примерно в 5 раз, и это существенно ускоряет их доставку. Однако, давайте рассмотрим следующий вопрос: как динамическое `gzip`-сжатие влияет на быстродействие сервера? Рентабельно ли включать `mod_gzip` / `mod_deflate` для высоконагруженных проектов? И в каких случаях архивирование вообще лучше не использовать?

Формализация модели

Для начала нужно было каким-либо образом установить издержки на само архивирование. Схематично эти накладные расходы можно представить примерно в следующем виде:

```
gzip = чтение/запись на диск + инициализация библиотеки + создание архива
```

Предполагается, что первые две составляющие не зависят от размера файла (в исследовании участвовали файлы от 500 байтов до 128 Кб), а являются более-менее постоянными (по сравнению с последним слагаемым). Однако, как оказалось, работы с файловой системой зависят от размера. Об этом чуть подробнее рассказывается ниже.

Естественно, что процессорные ресурсы, уходящие на «создание архива», должны быть примерно линейными от размера файла (линейное приближение вносит погрешность не

больше, чем остальные предположения), поэтому результирующая формула примет примерно такой вид:

```
gzip = FS + LI + K*size
```

Здесь **`FS`** — издержки на файловую систему, **`LI`** — издержки на инициализацию библиотеки и любые другие постоянные издержки, зависящие от реализации `gzip`, а **`K`** — коэффициент пропорциональности размера файла увеличению времени его архивирования.

Набор тестов

Итак, для проверки гипотезы и установления истинных коэффициентов нам потребуется 2 набора тестов:

- Тесты на сжатие: для набора пар значений «size — gzip»
- Тесты на запись: для набора пар значений «size — FS»

Почему именно 2, а как же издержки на инициализацию архивирования, спросите вы? Потому что в таком случае у нас получится система (не)линейных уравнений, а найти из нее 2 неизвестных (коэффициент пропорциональности и статические издержки) не представляется трудным. Решать переопределенную систему и рассчитывать лишний раз точную погрешность измерения не требуется: статистическими методами погрешность и так сводится к минимуму.

Для тестирования был взят обычный HTML-файл (чтобы условия максимально соответствовали реальным). Затем из него были вырезаны первые 500, 1000 ... 128000 байтов. Все получившиеся файлы на сервере сначала в цикле архивировались нужное число раз, затем открывались и копировались на файловую систему — с помощью встроенных средств ОС Linux (`cat`, `gzip`), чтобы не добавлять дополнительных издержек какого-либо «внешнего» языка программирования.

Результаты тестирования

Для сжатия был получен следующий график. Хорошо заметно, что для небольших файлов основных издержки вносятся работой с файловой системой, а не архивированием. Здесь и далее **все времена указаны в миллисекундах**. Проводились серии тестов по 10000 итераций.

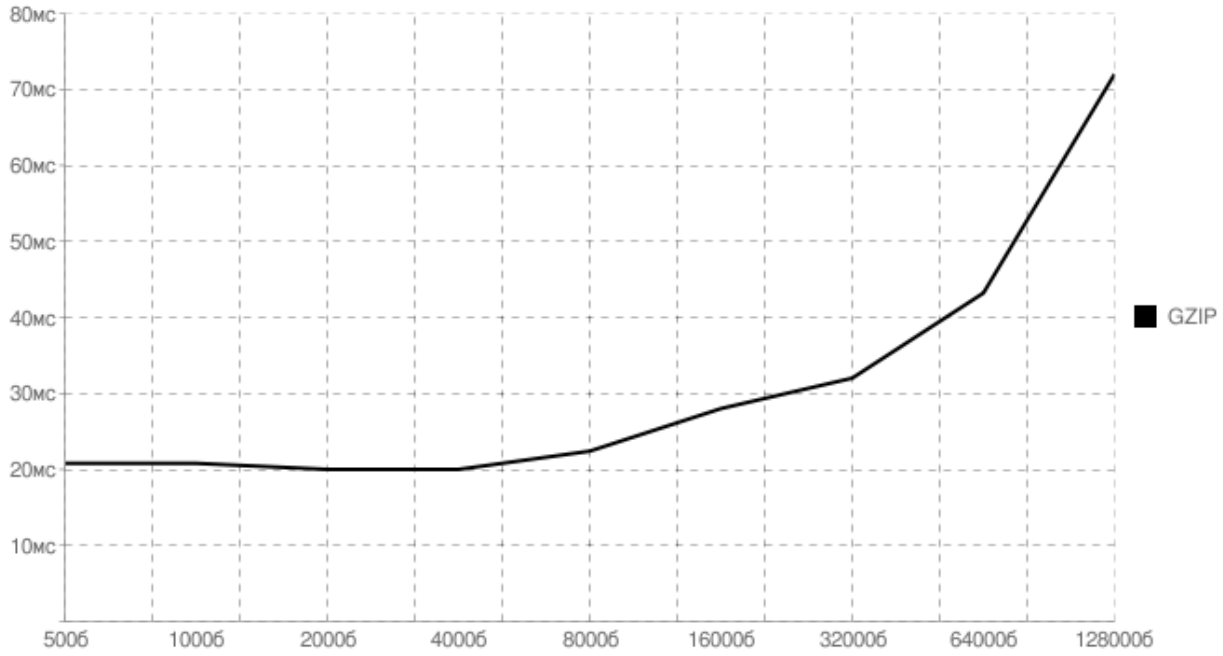


Рис. 3. График издержек на gzip-сжатие от размера файла

Теперь добавим исследования по работе с файловой системой, вычтем их из общих издержек и получим следующую картину:

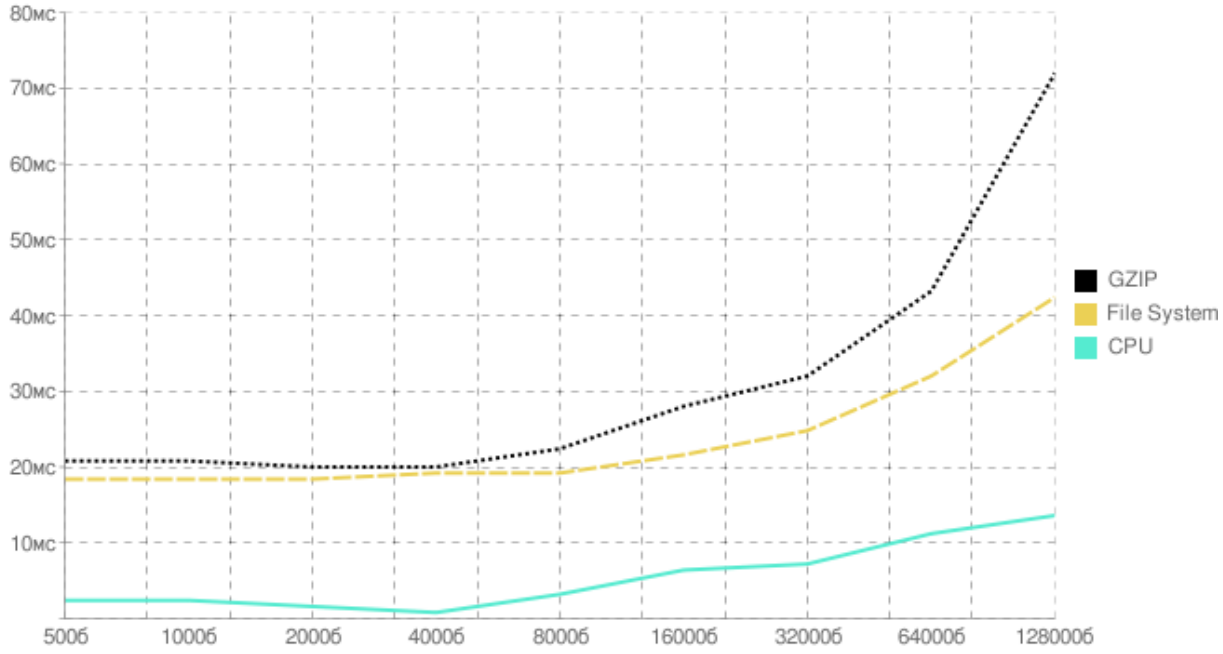


Рис. 4. График издержек на gzip-сжатие и работу с файловой системой

Издержки на открытие, запись, закрытие файла зависят в некоторой степени от размера, однако, нам это не мешает построить модельную зависимость вычислительной нагрузки от размера файла (предполагая, что в данном диапазоне она линейна). В результате, получим следующее:

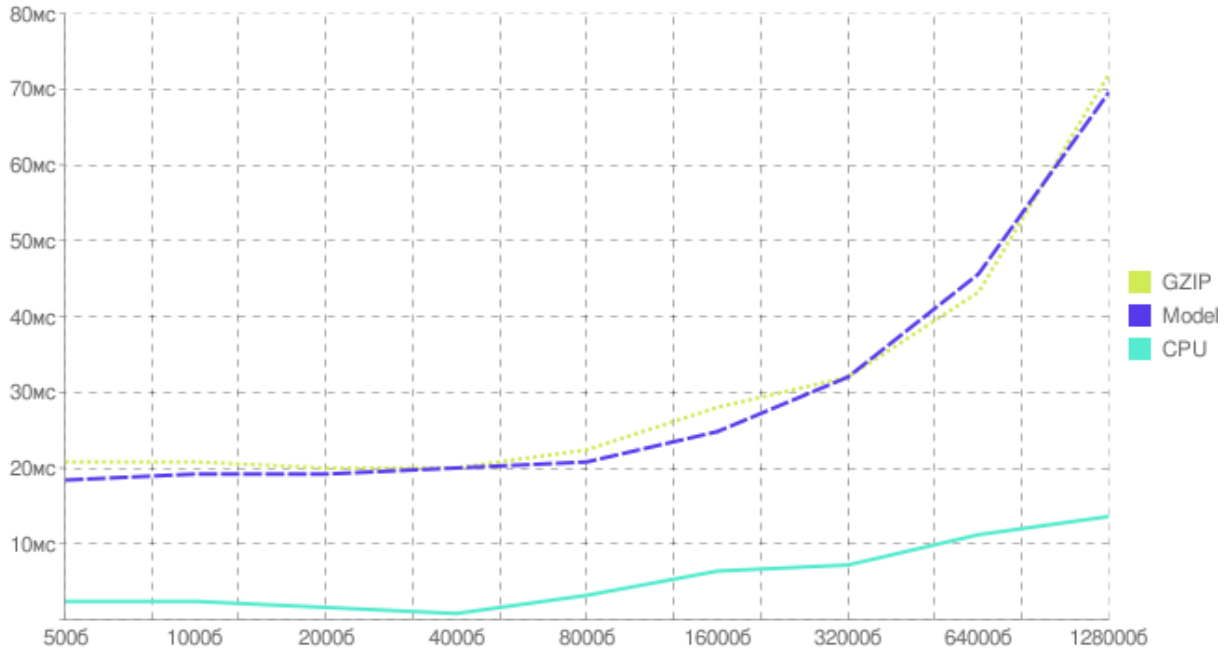


Рис. 5. График реальных и модельных издержек на gzip-сжатие

Пара слов о файловой системе

Вопрос: зачем нужны дополнительные тесты на производительность файловой системы, ведь уже есть характерное время, уходящее на `gzip`-сжатие определенных размеров файлов?

Ответ: во-первых, любой веб-сервер и так берет файл из файловой системы и архивирует уже в памяти, а потом пишет в сокет. Это время уже учтено при установлении соединения с сервером до получения первого байта. Нам лишь нужно понять, насколько оно увеличится, если сервер произведет еще некоторые операции с данными в оперативной памяти.

Во-вторых, не все серверы читают прямо с диска. У высоконагруженных систем и прокси-серверов (например, `OW, squid, nginx, thttpd`) данные могут храниться прямо в оперативной памяти, поэтому время доступа к ним существенно меньше, чем к файловой системе. Соответственно, его и нужно исключить из полученных результатов.

Что быстрее: `gzip` или канал?

Модель хорошо аппроксимирует полученные данные, поэтому примем ее за основу для следующих вычислений. Нам нужно, на самом деле, установить, насколько процессорные издержки на сжатие превосходят (или, наоборот, меньше) издержек на передачу несжатой информации. Для этого мы построим ряд графиков, приняв за эталон полученные коэффициенты для однопроцессорного сжатия на Dual Xeon 2,8 Ггц.

Так как с пользовательской стороны уходит некоторые время на распаковку архива, то ограничим его временем сжатия на машине с CPU в 1 Ггц. Это ограничение сверху: естественно, что распаковка экономичнее сжатия, да и пользовательские машины имеют процессоры, в среднем, мощнее, чем 1 Ггц. Однако, нам нужно получить лишь

качественные данные (ограничение снизу), поэтому ограничимся таким уровнем точности.

Итак, ниже приведены издержки на передачу дополнительного количества информации (в миллисекундах) для двух разных каналов (100 Кб/с и 1500 Кб/с) и двух разных серверов (280 МГц и 1 Ггц). Видно, что график для `gzip` на 1000 Mhz идет практически вровень с передачей данных для канала в 1500 Кб/с (одна линия перекрывает другую).

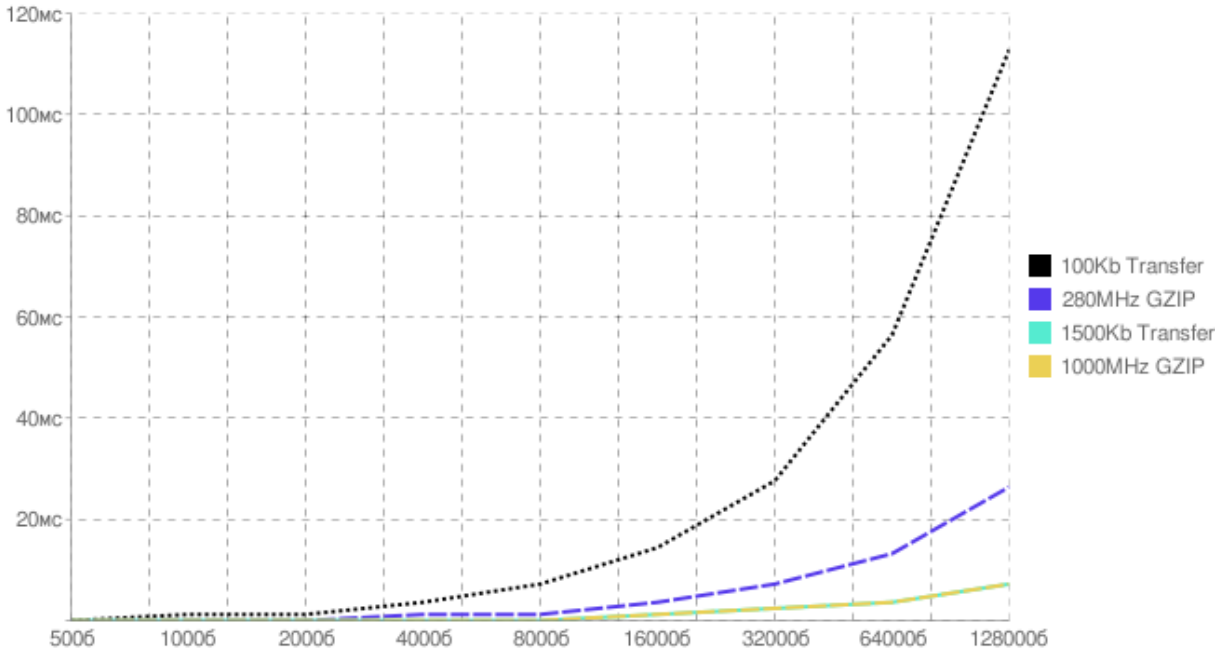


Рис. 6. Накладные издержки на сжатие и передачу информации для 100Кб и 1500Кб и 280МГц и 1000МГц

Исследование степени gzip-сжатия и загрузки процессора

Рассмотрим далее, насколько сильно издержки на `gzip` зависят от степени сжатия, и как их прогнозировать с учетом всех остальных параметров. Новая серия тестов была направлена на установление зависимости между степенью сжатия, процессорными издержками и уменьшением размера файла, чтобы на основе этих данных построить более точную модель, определяющую рациональность использования архивирования «на лету».

Как и ранее, на сервере проводились серии тестов по 10000 итераций в каждом. Замерялось время работы `gzip` при различных степенях сжатия. Затем оно усреднялось по серии, и из него вычитались издержки на работу с файловой системой. Также замерялось достигнутое уменьшение размера файла. Для зависимости «процессорное время — степень сжатия» был получен следующий график. По оси абсцисс идет степень сжатия, по оси ординат — затраченное время (среднее по серии):

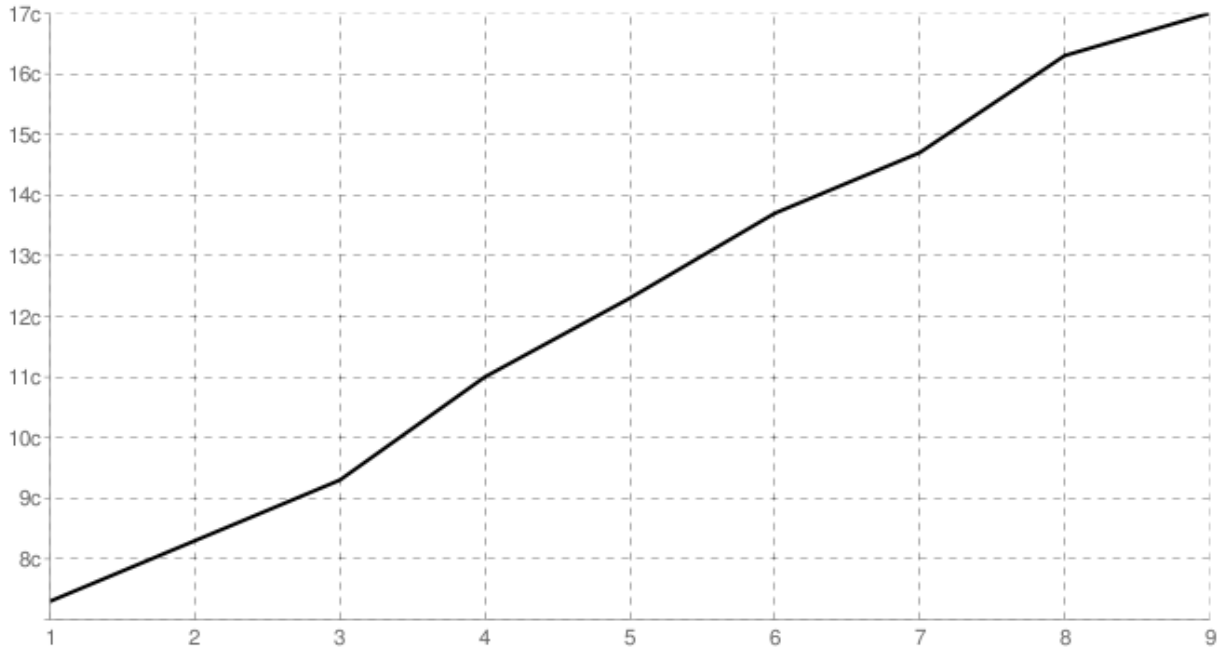


Рис. 7. Издержки на `gzip` от степени сжатия

Далее график эффективности полученного сжатия (в % от оригинального размера файлов) от степени сжатия:

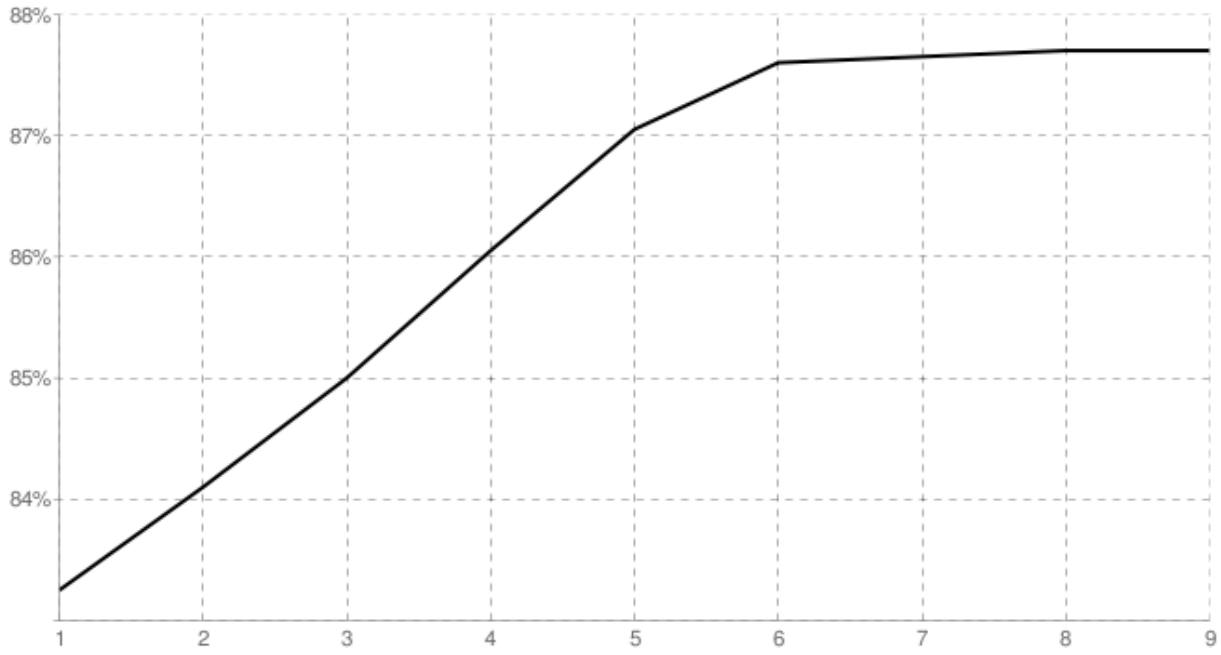


Рис. 8. Эффективность различных степеней `gzip`-сжатия

Окончательные выводы

Собственно, картинки говорят сами за себя. Если у вас HTML-файлы, в среднем, больше 4 Кб, то появится ощутимый выигрыш для большинства пользователей при включенном `gzip` на сервере (даже если этот сервер находится на весьма «слабенькой» машине). В случае маленьких файлов и(ли) медленного в вычислениях сервера, стоящего, однако, на быстром канале, будет экономичнее не сжимать файлы.

Хочется также обратить внимание на то, что, отдав пользователю данные быстрее (через gzip-сжатие), мы тем самым освободим часть серверных ресурсов, что может оказаться существенным подспорьем для высоконагруженных проектов.

В общем случае, gzip-сжатие позволяет существенно ускорить доставку HTML-файла до пользователя, не увеличивая нагрузку на сервер. Если же использовать статическое архивирование (готовые архивы хранить на сервере и обновлять только в случае необходимости), то выгода просто очевидна.

Конфигурируем Apache 1.3

Давайте рассмотрим, как можно настроить некоторые серверы для выдачи текстового содержания в виде архивов. Ниже приведен участок конфигурационного кода для Apache 1.3, позволяющий подключить gzip-сжатие. Основные директивы даны с комментариями.

```
<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 заголовок Content-Encoding: gzip
AddEncoding       gzip .gz

# выставляем минимальный размер для сжимаемого файла
mod_gzip_minimum_file_size  1000

# и максимальный размер файла
mod_gzip_maximum_file_size  500000

# выставляем максимальный размер файла, сжимаемого прямо в памяти
mod_gzip_maximum_inmem_size  60000

# устанавливаем версию протокола, с которой будут отдаваться gzip-файлы
# на клиент
mod_gzip_min_http       1000

# исключаем известные проблемные случаи
mod_gzip_item_exclude     reqheader "User-agent: Mozilla/4.0[678]"

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

# исключаем .css / .js файлы (о них подробнее в следующем разделе)
mod_gzip_item_exclude     file    \.js$
mod_gzip_item_exclude     file    \.css$

# дополнительно сжимаем другие текстовые файлы
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>
```

Конфигурируем Apache 2

Для Apache 2 описанные действия выглядят гораздо проще.

```
# добавляем Content-Type для всех файлов с расширением .gz
AddEncoding gzip .gz

# включаем сжатие для HTML- и XML-файлов
AddOutputFilterByType DEFLATE text/html
AddOutputFilterByType DEFLATE text/xml
# и для иконок (об этом чуть ниже)
AddOutputFilterByType DEFLATE image/x-icon

# выставляем максимальную степень сжатия (если возникнут проблемы с
# серверной производительностью, следует уменьшить до 7 или 1)
DeflateCompressionLevel 9

# и максимальный размер окна для архивирования
DeflateWindowSize 15

# отключаем архивирование для «проблемных» браузеров
BrowserMatch ^Mozilla/4 gzip-only-text/html
BrowserMatch ^Mozilla/4\.0[678] no-gzip
BrowserMatch \bMSIE !no-gzip !gzip-only-text/html

# добавляем заголовок Vary для корректного распознавания браузеров,
# находящихся за локальными прокси-серверами
Header append Vary User-Agent

# и запрещаем кэширование сжатых файлов для локальных прокси-серверов
<FilesMatch .*\.(html|phtml|php|shtml)$>
        Header append Cache-Control private
</FilesMatch>
```

Полные оптимизированные конфигурации для указанных серверов приведены в восьмой главе.

2.2. CSS и JavaScript в виде архивов

Теперь давайте рассмотрим, каким образом лучше всего будет отдавать CSS- и JavaScript-файлы в архивированном виде. Для обеспечения корректного архивирования, по-видимому, наиболее общий подход будет заключаться в выполнении по порядку следующих пунктов.

- Проверить, умеет ли клиент принимать файлы в формате `gzip-encoded`.

- Обеспечить соответствующий вывод на стороне сервера через `gzip`-функции либо организовать это непосредственно через веб-сервер (например, Apache).
- Настроить конфигурационные файлы (или `.htaccess`), чтобы обеспечить корректный `Content-Type`.

В данном случае сжатие данных «на лету», возможно, не будет наиболее оптимальным решением, потому что файлы стилей и скриптов изменяются достаточно редко, а мы заставим сервер каждый раз их сжимать. Тем более что лучше самого сервера с архивацией файлов никто не справится.

Статическое архивирование в действии

Есть способ обойтись просто парой строчек в конфигурационном файле (`httpd.conf` или `.htaccess`, первое предпочтительнее), если потратить пару минут и самостоятельно заархивировать все необходимые файлы. Предположим, что у нас есть JavaScript-библиотека `jquery.js` на сервере. Заархивируем ее в `jquery.js.gz` (при помощи 7-zip или любой другой утилиты, если в работе используется Windows). В итоге, должен появиться файл `jquery.js.gz`. Его нужно положить в ту же директорию на сервере, что и исходный файл.

Если работать прямо на сервере через командную строку, то достаточно выполнить следующую команду:

```
gzip jquery.js -c –n -9 > jquery.js.gz
```

Опция `-c` создаст новый файл (перенаправляем поток вывода в `jquery.js.gz`), `-n` исключит имя файла из архива (оно там только лишние байты занимает), а `-9` заставит использовать максимальную степень сжатия. Таким образом, мы получим минимально возможный архив из искомого файла.

Проблемы для Safari

В ходе реализации данного решения возникла маленькая, но досадная неприятность. Safari не умеет правильно обрабатывать файлы с расширением `.gz`: для этого браузера стили и скрипты не могут иметь такого расширения. Как же нам быть? Выход достаточно простой и очевидный.

Нам нужно именовать все архивы стандартным образом, но при этом иметь неархивированную версию для обратной совместимости (например, с дополнительным суффиксом `nogzip`). Поэтому для подготовки файлов нам будут нужны две команды (`jquery` здесь используется только в качестве примера):

```
cp $src/jquery.js $dst/jquery.nogzip.js
gzip $dst/jquery.nogzip.js -9 –n -c > $dst/jquery.js
```

где `$src` — директория, в которой хранятся исходные файлы, а `$dst` — финальная директория для публикации. Сначала мы копируем файл в финальное место дислокации, а потом его архивируем под «правильным» именем.

Конфигурируем Apache

Тесты под Konqueror показали, что этот браузер не понимает архивированных файлов (CSS- и JavaScript-), поэтому чтобы обезопасить десятую долю процента посетителей от сердечного приступа (когда они увидят сайт без соответствующих стилей), в этот набор правил его стоит добавить. Аналогично и «старым» браузерам (которые явно указывают, что не понимают архивов) отдается не архивированное содержание.

```
<IfModule mod_rewrite.c>
        RewriteEngine On
#перенаправляем Konqueror и «старые браузеры»
        RewriteCond %{HTTP:Accept-encoding} !gzip [OR]
        RewriteCond %{HTTP_USER_AGENT} Konqueror
        RewriteRule ^(.*)\.(css|js)$ $1.nogzip.$2 [QSA,L]
</IfModule>
```

Вся вышеуказанная конструкция «обернута» условием наличия на сервере подключенного `mod_rewrite`. Если он отсутствует, то это сразу станет видно на заявленных браузерах (перестанут отображаться стили и отрабатывать скрипты). Иначе Apache просто не сможет запуститься, т.к. `RewriteEngine` не будет объявлен.

Дополнительно к заявленной логике необходимо выставить ряд заголовков для отдаваемых файлов. В частности, `Vary` и `Cache-control` касаются локальных проксирующих серверов, которые не должны кэшировать эти файлы, а пропускать их дальше к пользователю, не обрезая при этом заголовок `User-Agent` (иначе наш сервер никак не узнает, можно ли отдавать архивированную копию файла или нет).

```
<IfModule mod_headers.c>
        Header append Vary User-Agent
#выставляем для всех css/js файлов Content-Encoding
        <FilesMatch .*\.(js|css)$>
                Header set Content-Encoding: gzip
                Header set Cache-control: private
        </FilesMatch>
#сбрасываем Content-Encoding в том случае, если отдаем не архив
        <FilesMatch .*\.nogzip\.(js|css)$>
                Header unset Content-Encoding
        </FilesMatch>
</IfModule>
```

В итоге, для всех файлов, которые мы отдаем как архивы, дополнительно объявляется `Content-Encoding`, а для их неархивированных копий этот заголовок сбрасывается. Чем и достигается полная работоспособность данного решения.

Маленькие «но»

Единственное неудобство, которое может возникнуть: нужно иметь в разработке нормальные версии, а при публикации всех изменений — их архивировать и переименовывать. При промышленном подходе к разработке все эти действия автоматизируются, а при кустарном — трудозатраты не так существенны по сравнению с увеличением скорости загрузки сайта (если, конечно, не собирать проект прямо на боевом сайте, без конца архивируя один и тот же файл).

Итак, финальный алгоритм действий (при наличии на сервере `mod_headers`, иначе лучше воспользоваться конфигурацией, приведенной в восьмой главе) должен быть следующим.

1. Добавляем описанные выше инструкции (оба блока) в конфигурационный файл Apache или `.htaccess`
2. Пакуем файлы (с помощью 7-zip или `gzip`) и кладем на место обычных (расширение у файлов должно остаться прежним, `.css` или `.js`). Например, если у нас есть файл `anyname.css`, то после упаковки получается файл `anyname.css.gz`, переименовываем его обратно в `anyname.css` и заливаем на сервер. Для `gzip` все немного проще:

```
gzip -c -9 -n anyname.css > anyname.css.gz
mv anyname.css anyname.nogzip.css
mv anyname.css.gz anyname.css
```

3. Рядом с сжатыми файлами кладутся файлы с расширением `nogzip.css` или `nogzip.js`, которые содержат неархивированные копии. Например, после заливки сжатого файла `anyname.css` нужно создать на сервере еще один файл `anyname.nogzip.css`, который является копией несжатого файла. Для `gzip` это копирование уже производится чуть выше второй строкой в листинге.

Два слова о nginx

Кто работал с этим сервером, наверное, уже подумали: есть же модуль `ngx_http_gzip_static_module`, который позволяет класть рядом с файлом его сжатую версию с дополнительным расширением `.gz` и забыть, практически, обо всех описанных проблемах (этот функционал присутствует и для Apache 1.3). К сожалению, минусом данного решения будет отключение сжатия для всех видов файлов у браузера, который не поддерживает хотя бы один (теряется гибкость настройки).

Однако, на данный момент таких случаев доли процента, поэтому если у нас проект с низкой или средней посещаемостью, указанный модуль (в совокупности с `ngx_http_gzip_module`) позволит преодолеть почти все подводные камни. Подробная конфигурация для nginx и Apache приведена в восьмой главе.

В третьей главе рассказывается, как данное решение можно расширить для сброса кэша на клиенте. Теперь же перейдет к более подробному рассмотрению методов сжатия CSS- и JavaScript-файлов — это ведь может быть не только архивирование.

2.3. Все о сжатии CSS

Проблема уменьшения CSS-файлов в размере действительно актуальна, и хотелось бы иметь результаты исследования конкретно для такой оптимизации. Оно, собственно, и приведено ниже.

В интернете было найдено 5 различных инструментов для минимизации CSS-кода, далее ими обрабатывались несколько примеров, которые затем подвергались еще и архивированию. Результаты представлены в виде графиков.

Инструменты

- **CSSMin** (http://code.google.com/p/cssmin/). Библиотека проводит набор простейших замен в CSS-файле (удаляет ненужные символы) и склеивает его в одну строку.

- **Minify** (http://code.google.com/p/minify/) . Библиотека, минимизирующая как CSS-, так и JS-файлы. Кроме того она может склеивать несколько файлов в один, заменять относительные пути к фоновым картинкам на более короткие и самостоятельно отдавать кэширующие заголовки. В общем, не сильно лучше предыдущей.
- **YUI** (http://developer.yahoo.com/yui/compressor/). YUI-compressor (использовалась версия 2.2.5). Фактически, делает то же самое, что две предыдущих библиотеки.
- **CSS Minifier** (http://www.artofscaling.com/css-minifier/). Автор разработал собственный алгоритм сжатия (после беглого анализа это оказалась несколько переработанная версия CSS Tidy), который, по его собственному утверждению, «жмет лучше всех». Это мы и проверим чуть дальше.
- **CSS Tidy** (http://csstidy.sourceforge.net/). Проект по минимизации CSS-файлов с открытым исходным кодов. Имеет много настроек, перенесен на несколько языков и используется на нескольких ресурсах, которые предлагают инструментарий для минимизации CSS-файлов, например, на www.codebeautifier.com. Наиболее широко распространенная версия минимизатора.

В качестве исходных файлов брались таблицы стилей с некоторых достаточно активно посещаемых ресурсов. Каждый из них был подвергнут действию минимизатора (для Minifier дополнительно файл склеивался в одну строку, вероятно, это временный баг текущей версии), затем архивировался. Корректность минимизации **не проверялась** (с этим в некоторых особо агрессивных случаях могут быть проблемы: CSS Tidy с определенными настройками перегруппировывает селекторы, и часть логики теряется).

Графические результаты

Что изображено на графиках? Выведен выигрыш (в процентах) относительно несжатого файла (по оси ординат отложены проценты). По оси абсцисс отложены номера файлов. Данные упорядочены по общей степени сжатия.

Вначале по каждому инструменту отдельный график: выведены показатели для простой минимизации файлов, также для минимизации с последующим архивированием. Серая линия на графике показывает степень сжатия (в процентах) файла при помощи простого `gzip`. Все инструменты приведены на одном графике (без архивирования). Действительно, заметен явный выигрыш для Minifier.

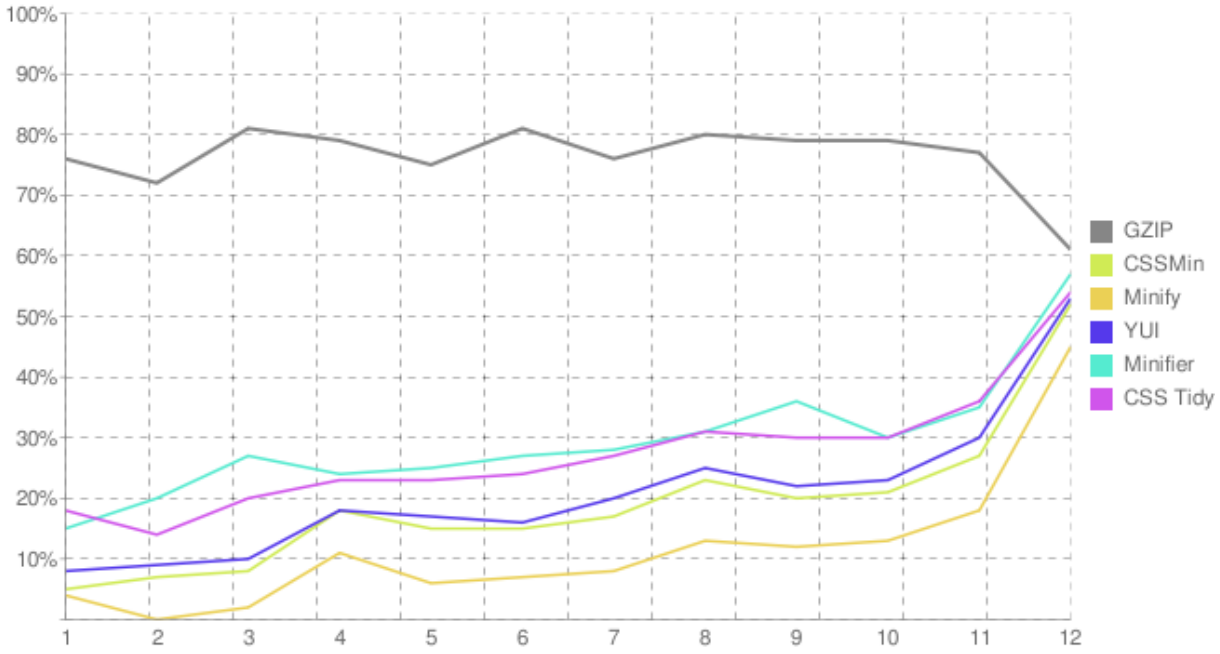


Рис. 9. Эффективность различных инструментов для минимизации CSS-файлов по сравнению с `gzip`

При архивировании, однако, все минимизаторы ведут себя примерно одинаково.

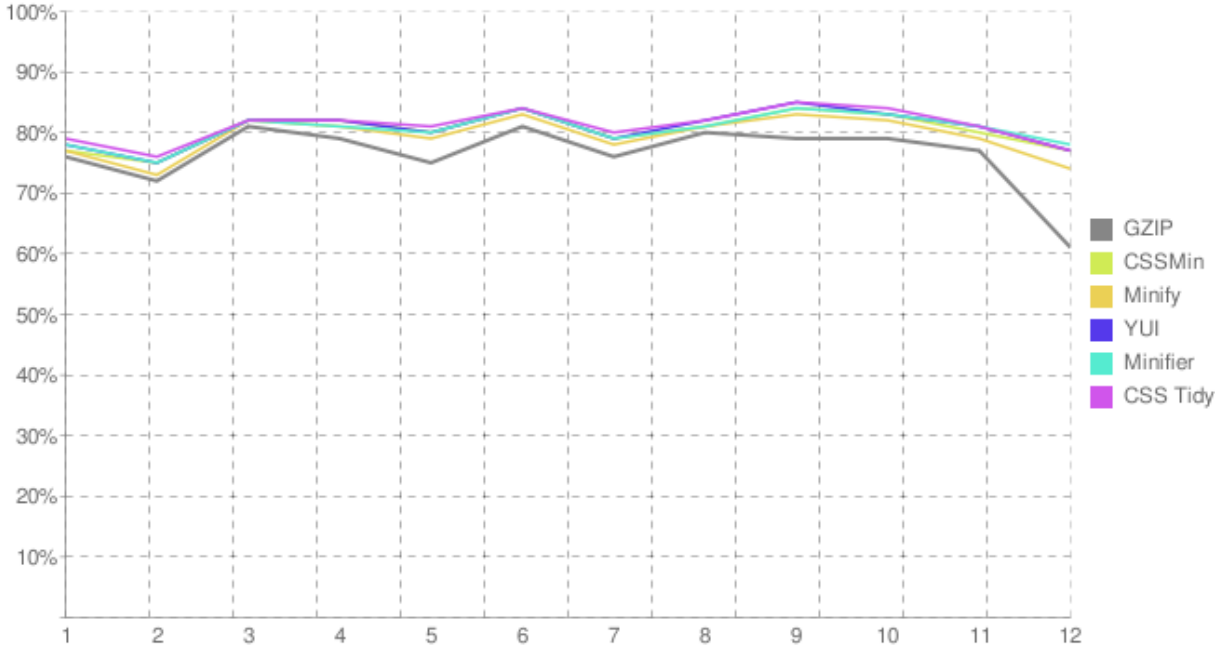


Рис. 10. Эффективность различных инструментов для минимизации CSS-файлов вместе с дополнительным архивированием по сравнению с `gzip`

Для уточнения картины при архивировании минимизированного файла отдельно было выделено его преимущество относительно обычного архивирования.

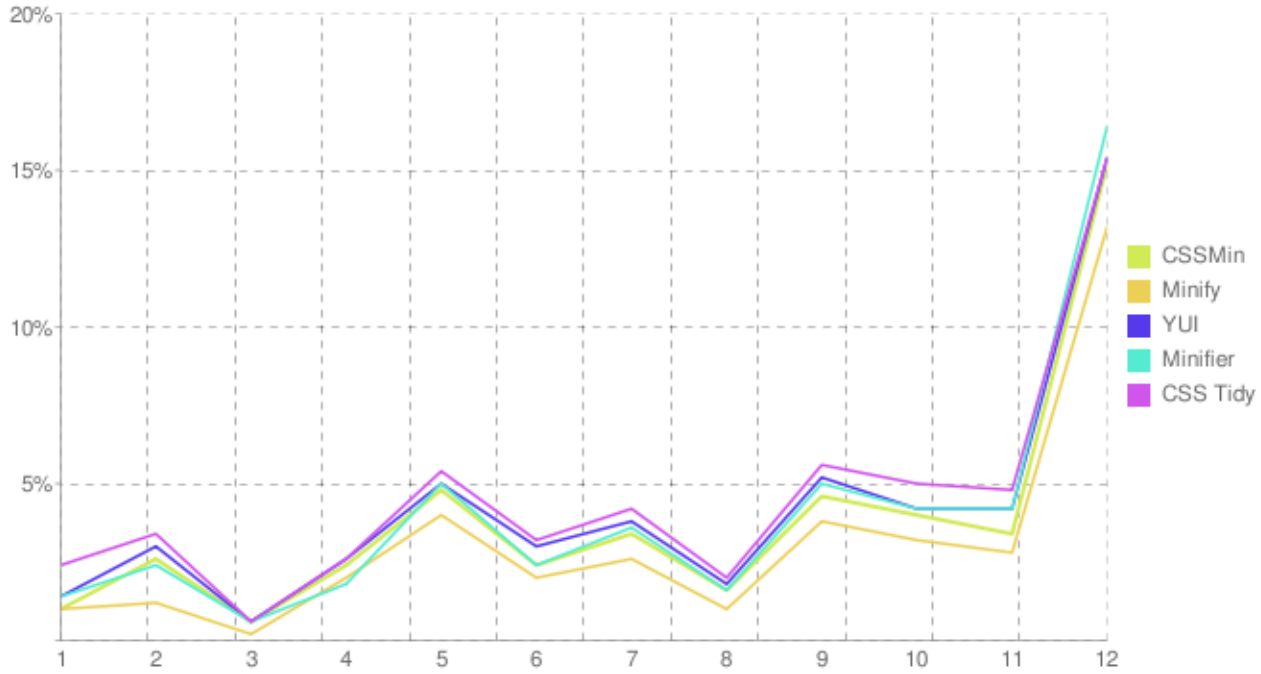


Рис. 11. Эффективность различных инструментов для минимизации CSS-файлов вместе с дополнительным архивированием, увеличенный масштаб

Тут уже видно отчетливо, что CSS Tidy ведет себя, в целом, лучше остальных скриптов (хотя, за исключением редких случаев, выигрыш не превосходит 6% относительно обычного архивирования).

Выводы

Во-первых, `gzip` и так показывает хорошее сжатие (до 81%), поэтому в большинстве случаев можно пользоваться только им.

Во-вторых, простая «подчистка мусора» (удаление всех символов, которые можно безболезненно убрать по спецификации CSS) вместе с архивированием дает весьма неплохой результат (общее сжатие до 83%) относительно других инструментов, но при этом не теряется логика селекторов (т.е. такое сжатие абсолютно безопасно).

В-третьих, замечен локальный выброс при файле небольшого размера. Он связан с тем, что `gzip` изначально его плохо сжал (вероятно, из-за маленькой исходной библиотеки слов), поэтому все минимизаторы показали себя на высоте. Однако, файлы такого размера (порядка 1 Кб) стоит либо объединять с другими файлами (ибо тратить время на дополнительный запрос на сервер из-за такой мелочи не очень рационально), либо включить в сам HTML-файл. Так что данный выброс не стоит считать серьезным основанием для того, чтобы действительно использовать какой-либо минимизатор только из-за выигрыша в 3–4% от размера исходного файла.

В-четвертых, получается, что библиотека, жмущая лучше всего, проигрывает более умеренной сопернице при дополнительном архивировании результата. В нашем случае Minifier уступает CSS Tidy.

В общем, если мы не хотим дополнительно морочить себе голову, то можно просто архивировать CSS-файлы (в среднем, выигрыш 79%) либо проводить простую «подчистку

мусора» перед архивированием (в среднем, выигрыш 82%). Если мы заботимся о количестве байтов, то стоит изучить действие CSS Tidy и Minifier (их прелесть заключается в алгоритме перегруппировки селекторов) и использовать их либо разработать собственное приложение.

Практический пример

По сравнению с JavaScript, CSS относительно просто сжимать. В силу практически полного отсутствия строк, заключенных в кавычки (в основном, пути и названия шрифтов) мы можем изничтожить проблемы обычными регулярными выражениями. Когда же мы **действительно** встречаемся со строкой в кавычках, то мы можем объединить множественные пробелы в один (так как мы не рассчитываем обнаружить их в количестве больше чем 1 в URL или названиях шрифтов). Простейший скрипт на Perl может обеспечить нам все необходимые преобразования:

```
#!/usr/bin/perl

my $data = '';
open F, $ARGV[0] or die "Не получается открыть исходный файл: $!";
$data .= $_ while <F>;
close F;

$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

print $data;
```

Осталось прогнать все наши CSS-файлы через этот скрипт, чтобы сжать их, например, так:

```
perl compress.pl site.source.css > site.compress.css
```

Путем простых текстовых преобразований можно уменьшить общий объем передаваемых данных почти на 50% (очень сильно зависит от стиля кодирования; обычно будет получен менее впечатляющий результат), что обеспечит более быструю работу сайта для конечных пользователей в том случае, если `gzip` применить не удается.

2.4. JavaScript: жать или не жать?

Давайте рассмотрим далее сжатие JavaScript-файлов и проведем анализ всех наиболее известных средств статической минимизации JavaScript-кода. Нам нужно, по сути, ответить на три основных вопроса:

- Имеет ли смысл пользоваться каким-либо минимизатором JavaScript-кода?
- Есть ли среди них универсальное средство, показывающее лучшие результаты в подавляющем большинстве случаев?
- Если такого средства нет, то каковы критерии использования набора инструментов?

Итак, с постановкой задачи разобрались. Теперь перейдем, собственно, к самим инструментам и графикам степени сжатия исходного кода при их применении.

Инструменты и методика

Всего удалось обнаружить 5 кардинально различных средств для минимизации JavaScript-файлов, которые могут работать как автономные приложения (в расчете на то, что их можно будет далее запускать по событию или по расписанию, ориентируясь, в общем, на автоматизацию процесса публикации файлов на production-сервере).

1. **JSMin** (http://www.crockford.com/javascript/jsmin.html). Наиболее широко распространенный минимизатор, основывается на простых правилах, перенесен на множество языков, в том числе, и на сам JavaScript.
2. **JavaScript::Minifier** (http://search.cpan.org/~pmichaux/JavaScript-Minifier-1.04/lib/JavaScript/Minifier.pm). Отдельный перловый модуль, по степени сжатия очень близок к JSMin, однако генерирует отличный от первого синтаксис.
3. **Dojo ShrinkSafe aka Rhino** (http://dojotoolkit.org/docs/shrinksafe). Первоначально разрабатывался как Rhino, затем был включен в состав Dojo. Запускается как jar-модуль.
4. **Dean Edwards Packer** (http://dean.edwards.name/packer/). Достаточно широко известный инструмент от не менее известного Dean Edwards. Перенесен на некоторые языки, в том числе, на PHP4/5.
5. **YUI Compressor** (http://developer.yahoo.com/yui/compressor/). В представлении также не нуждается, именно на его основе проведена оптимизация сайтов Yahoo. Для анализа использовалась версия 2.2.5. Запускается как jar-модуль.

Для анализа были найдены несколько достаточно больших и широко используемых JavaScript-файлов (вполне возможно, что вы их загружали в этом месяце или даже используете из кэша каждый день) разных размеров и степеней изначального сжатия.

Все исходные файлы сжимались всеми представленными инструментами, затем архивировались. Далее все полученные данные нужно было выстроить в какой-то последовательности, которая бы выявляла характер и преимущества сжатия файлов сторонними средствами. Критерием для такой последовательности была выбрана изначальная «сжимаемость» файлов (то, насколько они хорошо архивируются).

Вполне очевидно, что если файл уже достаточно плохо архивируется, то предварительное его сжатие каким-либо минимизатором ситуацию улучшить не должно (такие файлы обычно оформлены в довольно минималистичном стиле, и из них уже удалены все комментарии). Что и подтвердилось на конкретных примерах.

Графические результаты

Сами графики, собственно. **Что на них изображено?** Выведен выигрыш (в процентах) относительно несжатого файла (по оси ординат отложены проценты). По оси абсцисс отложены размеры исходных файлов. Внимание: файлы **не** расположены по размеру. Данные упорядочены по общей степени сжатия.

Все инструменты приведены на одном графике. Заметен явный выигрыш Packer без архивирования.

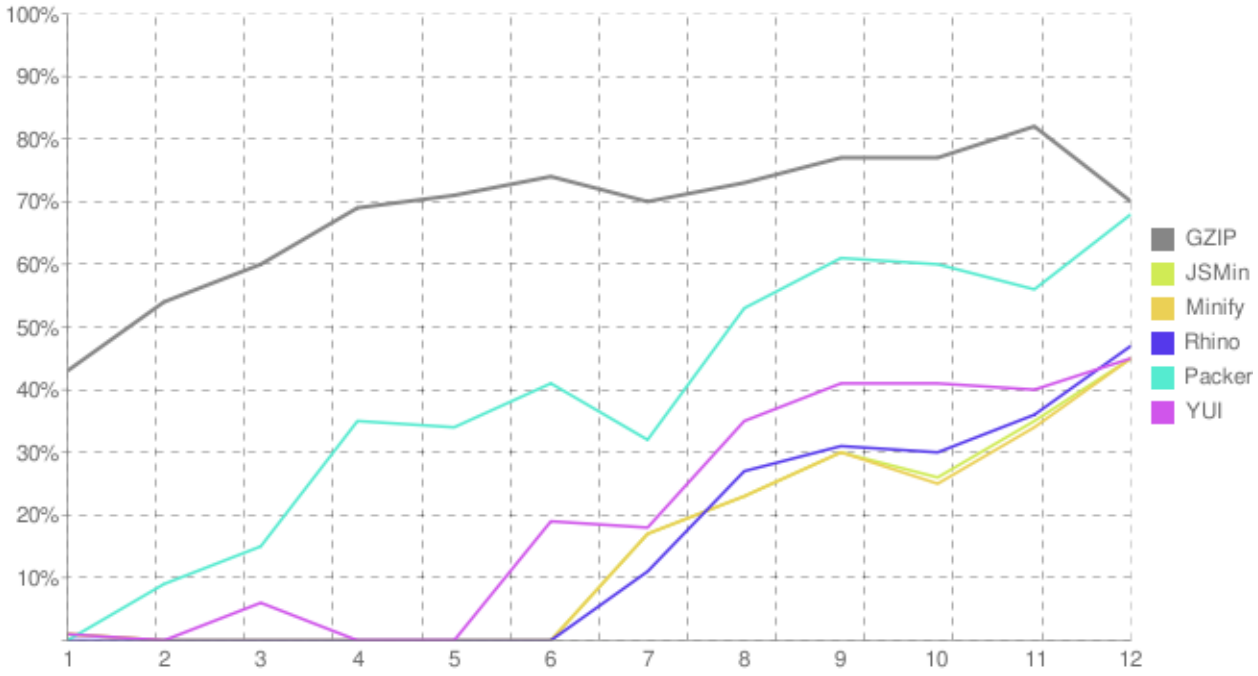


Рис. 12. Эффективность различных инструментов для минимизации JavaScript-файлов вместе по сравнению с `gzip`

При архивировании, однако, все минимизаторы кода ведут себя примерно одинаково.

Рис. 13. Эффективность различных инструментов для минимизации JavaScript-файлов вместе с дополнительным архивированием по сравнению с `gzip`

Для уточнения картины при архивировании минимизированного файла давайте отдельно выделим их преимущество (если оно имеется) относительно обычного архивирования.

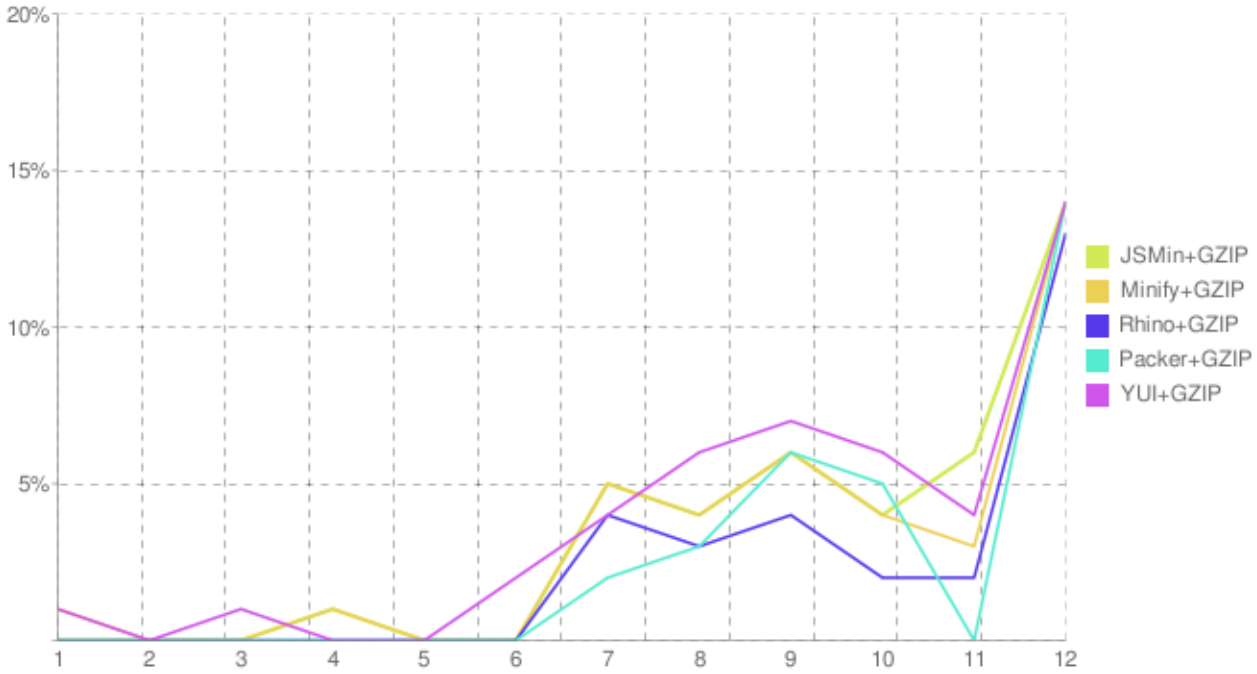


Рис. 14. Эффективность различных инструментов для минимизации JavaScript-файлов вместе с дополнительным архивированием, увеличенный масштаб

Тут уже хорошо видно, что YUI Compressor ведет себя, в целом, лучше остальных скриптов.

Промежуточные выводы

Во-первых, стоит указать на практически идентичное поведение JSMin и JavaScript::Minifier — скорее всего, они действуют по достаточно похожему алгоритму. Однако последний обладает скрытым потенциалом (при более подробном рассмотрении файлов, полученных вследствие работы второго, оказалось, что они могут быть уменьшены еще), но он работает в несколько раз дольше аналогов (3–5 секунд против 0,3–0,5 для Packer на PHP).

Во-вторых, файлы, которые меньше 1 Кб или при архивировании дают выигрыш меньше 70%, минимизировать не имеет смысла. Минимизация дает в таком случае результат, сравнимый с нулем. Если с сервера отдаются небольшие (до 20 Кб в несжатом виде) архивированные файлы (`.gz`), то стоит по умолчанию их минимизировать с помощью JSMin.

В-третьих, если на сервере не поддерживается сжатие скриптов, то отдавать лучше версию, минимизированную с помощью Packer, — в таком случае выигрыш довольно значительный (естественно, если размер файла больше 1 Кб). Такая минимизация, в среднем, показала 50% преимущество относительно несжатого файла.

В-четвертых, во всех остальных случаях (сервер отдает достаточно большие `gzip`-версии файлов, которые хорошо архивируются) стоит использовать YUI Compressor (в среднем, показал 6% преимущество относительно простого `gzip`).

Есть ли жизнь после сжатия?

Хочется отметить, что при минимизации JavaScript-файлов нужно следить за тем, чтобы функционал не уменьшился вследствие этой самой минимизации. Для проверки JS-файлов на работоспособность и общую адекватность существует проект JSLint (http://jslint.com/), который сравнивает исходный файл с набором спецификаций по синтаксису и выдает сообщения об обнаруженных ошибках.

Скорость загрузки JavaScript-библиотек

В начале 2008 года командой PDWiki был проведен весьма впечатляющий анализ производительности JavaScript. Они собирались разобраться, насколько быстро грузятся JavaScript-библиотеки (естественно, скорость их загрузки будет заметно влиять на скорость загрузки всей страницы).

В результате было развернуто тестовое окружение для получения информации от различных браузеров, затем собрали все результаты в итоговом отчете. В нем достаточно много информации, которая может быть полезна как разработчикам веб-приложений, так и разработчикам браузеров: структурированные таким образом данные достаточно обширны и достойны быть объектом отдельного исследования.

Методы упаковки JavaScript

При загрузке JavaScript-кода обычно предполагается, что чем меньше загружаемый файл, тем быстрее он загрузится. Это несколько не соответствует действительности, что прекрасно подтверждает дальнейшее изучение ситуации. Мы рассмотрим скорость загрузки библиотеки jQuery в трех формах: обычной, уменьшенной (при помощи YUI Compressor) и упакованной (используется Packer). Если упорядочить их по размерам, то будет примерно так: самый маленький вариант, естественно, упакованный, затем уменьшенный, затем нормальный.

Однако упакованная версия добавляет некоторые накладные расходы: ее нужно сначала распаковать (выполнять достаточно тяжелый `eval` и `replace`) с помощью того же JavaScript на стороне клиента. Эта распаковка может занять достаточно продолжительное время при загрузке страницы. То есть, использование уменьшенной версии, в конце концов, будет значительно быстрее, чем упакованной — даже при достаточно большом размере файла.

Ниже приводится сравнение времени загрузки различных вариантов уменьшения jQuery.

Вариант	Среднее время
Уменьшенный	519.7214
Упакованный	591.6636
Нормальный	645.4818

Таблица 1. Время загрузки библиотеки jQuery, которая была подвергнута различным уменьшениям

Очевидно, что при использовании любой техники сжатия стоит помнить о такой формуле:

```
Время_загрузки = Время_на_скачивание + Время_на_исполнение
```

Именно поэтому упакованный вариант, будучи наименьшим по размеру, может проигрывать в производительности другим, менее экстремальным, способам представления информации.

Подводя итог всем вышеприведенным выкладкам, можно сделать следующее заключение. Если использовать `gzip`-сжатие для текстовых файлов, то наилучшим выбором будет применение YUI Compressor для дополнительной минимизации CSS- и JavaScript-файлов. Результирующий файл будет, в среднем, самым маленьким из возможных вариантов сжатия и будет загружаться в браузере максимально быстро.

Производительность загрузки JavaScript-библиотек

Из этого исследования можно еще получить данные по влиянию производительности различных JavaScript-библиотек на загрузку страницы. Таким образом, более простая и меньшая по размеру библиотека будет загружаться быстрее аналогов. По результатам видно, что jQuery загружается достаточно быстро относительно других библиотек (200–400 мс — существенный выигрыш в скорости). Ниже приведено среднее время загрузки неархивированных и не уменьшенных версий библиотек не из кэша.

Инструментарий	Среднее время
`jquery-1.2.1`	732.1935
`dojo-1.0.1`	911.3255
`prototype-1.6.0`	923.7074
`yahoo-utilities-2.4.0`	927.4604
`protoculous-1.0.2`	1136.5497

Таблица 2. Время загрузки различных библиотек (не модифицированные версии, без учета кэширования)

Сейчас, конечно,, можно возразить, что нечестно тестировать загрузку только не кэшированных страниц, ибо, согласно исследованиям Yahoo по кэшированию, примерно 50% посетителей не будут иметь возможности кэшировать содержание страницы. Поэтому важно убедиться, что не только первоначальная, но и кэшированная версия страницы также загружается максимально быстро. Итак, ниже приведены цифры для загрузки архивированных и уменьшенных версий из кэша.

Инструментарий	Среднее время
`yahoo-utilities-2.4.0`	122.7867
`Jquery-1.2.1`	131.1841
`prototype-1.6.0`	142.7332
`dojo-1.0.1`	171.2600
`protoculous-1.0.2`	276.1929

Таблица 3. Время загрузки различных библиотек (модифицированные версии, с учетом кэширования)

Если принять во внимание кэшированную версию, то разница становится уже не столь очевидна (всего 10–30мс — за исключением Dojo/Scriptaculous). Более того, при загрузке из кэша все издержки приходятся на инициализацию библиотек, — именно поэтому так важно знать и использовать принципы создания быстрых JavaScript-приложений. Об этом подробнее рассказывается в седьмой главе.

Но давайте на этом закончим со сжатием текстовых файлов и перейдем к более интересным случаям — уменьшению в размере различных форматов изображений.

2.5. PNG против GIF

Переносимый сетевой графический формат (*англ. Portable Network Graphics, PNG*) разрабатывается как более эффективная, гибкая и свободная от патентов замена GIF-формату. PNG был задуман для хранения отдельных растровых изображений и дальнейшего их распространения по компьютерным сетям. PNG был создан в 1995 в ответ на давление со стороны Unisys и их патента на алгоритм LZW-сжатия, используемый в GIF. Хотя срок действия патента Unisys уже закончился, причины на переход от GIF к PNG остались практически прежними. Заменив GIF-изображения теми же самыми, но в формате PNG, можно ускорить загрузку страниц и сэкономить трафик пользователей.

Алгоритмы сжатия

PNG использует алгоритм `deflate`-сжатия обычно со скользящим окном в 32 Кб. `Deflate` является улучшенной версией алгоритма сжатия Lempel-Ziv (LZ77), который применяется в `zip`- и `gzip`-файлах. Созданный Phil Katz для второй версии PKZip, `deflate` совмещает LZ77 с кодированием Huffman и является на 10–30% более эффективным, чем LZW при сжатии без потери информации. Так же, как и `gzip`, некоторые инструменты по PNG-сжатию предполагают опциональный параметр «степень сжатия», которая варьируется от 1 до 9. По умолчанию выставляется 6. Практически всегда лучшим выбором для максимального сжатия является 9.

Неудивительно, что изображения, сохраненные как PNG, обычно на 10–30% меньше по размеру, чем GIF, хотя в некоторых редких случаях они могут быть несколько больше (чаще всего это проявляется для небольших изображений). Обычно изображения с большими однотонными областями сжимаются лучше, чем градиентные с большим количеством переходов между цветами.

Возможности PNG

В PNG присутствует набор возможностей, которые делают его привлекательным для использования во многих отраслях, где требуется применение ограниченной палитры. Поддержка в PNG 16-битной серой шкалы прекрасно подходит для создания точных радиологических изображений. PNG предварительно фильтрует данные по конкретному изображению при помощи предсказательных функций. Одной из них является «Вверх» (*англ. Up*), которая ищет похожие наборы данных в вертикальных шаблонах для полноцветных PNG. PNG с индексированными цветами (8 битов или меньше) обычно не выигрывает от использования фильтрации, поэтому стоит использовать «ничего» (*англ. none*), если есть возможность для выбора. Для полноцветных или серых изображений лучше применять «Адаптивный» (*англ. Adaptive*) алгоритм.

Как говорит Greg Roelofs, «PNG, в основном, используется для создания 24-битных RGB сформированных изображений, например, картин с рассчитанным освещением с минимальным числом текстур или математических объектов. Они все обладают искусственно сглаженными цветовыми переходами, которые хорошо сжимаются при помощи PNG-фильтров. Некоторые фракталы могут вести себя таким же образом, но у многих из самых лучших примеров имеется достаточно «зашумленных» областей, которые сжимаются весьма слабо».

Для веб-страниц вполне можно использовать PNG8 (8 битный формат), с помощью которого дизайнеры могут заменить существующие GIF-изображения. У PNG также может быть альфа-значение для каждого цвета в палитре, которое, фактически, означает, что используется RGBA-палитра, а не RGB-XOR-маска, как GIF. Это позволяет варьировать прозрачность цвета в больших пределах, сохраняя преимущества 8-битного изображения перед 32-битным. PNG могут также содержать только один уровень прозрачности, совсем как GIF89a. Алгоритм сжатия PNG для повторяющихся горизонтальных шаблонов совпадает с LZW-сжатием в GIF.

Многослойный PNG-файл также может быть отображен на экране по загрузке только 25% всего файла, в то время как GIF требует загрузки 50% размера перед распознаванием. За исключением весьма редких случаев замена GIF-изображений на PNG-эквиваленты способна существенно уменьшить их размер.

Ниже приведены некоторые из возможностей PNG-формата.

- 8-битные (индексированная палитра), 16-битные серые или 48-битные полноцветные изображения.
- Градация альфа-прозрачности до 16 битов.
- Гамма-коррекция (хотя эта возможность может быть проблематичной).
- Улучшенный по сравнению с LZW алгоритм сжатия.
- Двумерная схема для многоуровневых изображений (Adam7).
- Метаданные (сжатые или несжатые).
- Формат, свободный от патентов.

Поддержка PNG в браузерах

В Netscape естественная поддержка PNG весьма ограничена: начиная с версии 4.04, для Internet Explorer она зависит от операционной системы. Для Macintosh IE полностью поддерживает PNG с версии 5.0 (в том числе, включая альфа-канал). MSIE для Win32 и Unix обладает естественной поддержкой PNG (на деле же весьма посредственной) начиная с 4.0, но не поддерживает альфа-канал до версии 7.0 (это исправляется при помощи фильтра `AlphaImageLoader`).

На данный момент большое количество разнообразных браузеров также поддерживает PNG, однако, лишь с 1-битной прозрачностью, что позволяет использовать PNG для замены неанимированных GIF.

PNG и проблема соответствия для фоновых CSS-изображений

К несчастью, поддержка возможностей PNG-гаммы и цветовой коррекции не является кроссбраузерной. Наиболее часто рекомендуемой мерой для исправления возможных ошибок будет исключение фрагментов, обеспечивающих гамму и цветовую коррекцию,

для создания «неименованного» PNG (удаление gAMA чанка). Это решает проблему цветового соответствия для современных браузеров, кроме Safari под Mac до OS 10.4 (тут может помочь удаление sRBG чанка, подробнее об удалении чанков рассказывается немного ниже).

Для отдельных PNG-изображений это совсем не проблема, но для дизайна веб-страниц, который требует точного соответствия между CSS-фоном и PNG, к сожалению, это имеет значение (однако, может быть устранено указанными выше путями).

Анимированные PNG: MNG против "PNG+"

Формат составной сетевой графики (*англ. Multiple Network Graphics, MNG*) представляет собой несколько PNG-изображений, по аналогии с GIF89a. Однако, MNG-формат является более сложным и не поддерживается текущими браузерами (для этого нужно использовать бесплатное расширение `libmng`).

Группа разработчиков Mozilla расширила текущий (для одного изображения) PNG-формат в браузере Firefox 3, добавив несколько дополнительных кадров для анимации. Существует также встречное предложение со стороны членов группы PNG, подразумевающее создание чего-то подобного, однако, не противоречащего спецификации PNG. Оба этих направления значительно проще, чем MNG, и оба открыты для голосования в группе PNG.

Сейчас же для анимации изображений лучше всего применять GIF89a или Flash. Однако последние достижения в векторной графике, SVG, и развитие анимационных JavaScript-библиотек составляют реальную конкуренцию указанным форматам. Вполне возможно, что через несколько лет вся несложная анимация в браузерах будет осуществляться при использовании именно такого подхода.

Двигаемся к маленьким PNG

PNG работает лучше с большими однотонными областями. Чтобы увеличить степень сжатия, стоит уменьшить количество «шума» в ваших картинках для увеличения размеров однотонных областей. Уменьшайте области размывания, хотя для изображений с градиентами размывание позволяет использовать меньшую глубину цвета. Избегайте сглаживания текста для уменьшения числа цветов, которые для него необходимы, в финальном изображении.

По возможности, уменьшайте число цветов в вашем исходном изображении, если в процессе разработки этот фактор можно контролировать. Стоит также избегать использования многослойных PNG-изображений для уменьшения размера файла, так как семипроходная многослойная схема может добавить от 20% до 35% к размеру PNG-файла. Наконец, можно применять специальное программное обеспечение для оптимизации PNG-изображений, которое специально разрабатывается с этой целью, например, `pngout` или `pngcrush`.

Полезные советы

Ниже приведено несколько простых советов, как текущие изображения можно дополнительно уменьшить в размере. Можно написать простой скрипт, который

перебирает директории с изображениями перед публикацией сайта и делает следующие действия (далее приведены примеры запуска утилит из командной строки для ОС Linux).

1. Преобразовывает GIF в PNG (и проверяет, есть ли при этом выигрыш):

   ```
   convert image.gif image.png
   ```

   или так

   ```
   gif2png -nstO image.gif image.png
   ```

2. Уменьшает PNG файлы в размере:

   ```
   pngcrush -qz3 -brute image.png result.png
   ```

   если при этом нужно удалить и gAMA чанк, то:

   ```
   pngcrush -qz3 -rem gAMA -brute image.png result.png
   ```

   если при этом хотим удалить другие чанки, отвечающие за цветовую коррекцию, то:

   ```
   pngcrush -qz3 -rem gAMA -rem cHRM -rem iCCP -rem sRGB \
           -brute image.png result.png
   ```

3. Уменьшает JPEG-файлы в размере (без потери качества):

   ```
   jpegtran -copy none -optimize -perfect image.jpg > result.jpg
   ```

Под Windows для уменьшения `.png` изображений можно использовать TweakPNG (http://entropymine.com/jason/tweakpng/). Аналогом `jpegtran` является набор портированных утилит `jpeg`, которые можно загрузить по адресу: http://sourceforge.net/projects/gnuwin32/ .

Для отдельно взятой страницы общий размер изображений может быть уменьшен на 20–30% только благодаря следованию этим простым советам.

2.6. Разгоняем `favicon.ico` — это как?

В очередной презентации Yahoo! на тему клиентской производительности (часть 2) был поднят вопрос о `favicon.ico`. Они проводили несколько интересных фактов о данном явлении и давали пару советов. Процитируем их рекомендации.

- `www.mysite.ru/favicon.ico`
- Необходимое зло:
  - Браузер в любом случае ее запросит
  - Лучше не отвечать 404-ошибкой
  - Будут отправлены cookie
  - Не может быть в CDN
  - Мешается в последовательности загрузки ресурсов
- Уменьшайте ее (<=1 Кб)
- Использовать анимированные иконки ни разу не хорошо

- Выставляйте заголовок `Expires`
- Запросы к `favicon.ico` составляют 5 –10% от общего числа запросов к сайту

`favicon.ico`, во-первых, запрашивается едва ли не один-единственный раз браузером при посещении сайта, во-вторых, загружается, игнорируя обычный порядок загрузки. Из чего можно заключить, что она не является обычной картинкой при загрузке сайта, поэтому в дополнение к уже имеющейся информации был проведен ряд дополнительных исследований, чтобы объединить все, что известно прогрессивному человечеству на данную тему. Однако в ходе изучения материала оказалось, что проблема совсем не так прозрачна, как представлялось изначально. Формат `.ico` предстал в новом свете, весьма выгодном для использования в Интернете.

Краткое описание формата

`favicon.ico` имеет формат `.ico` (на самом деле, такой формат есть, а соответствующий MIME-тип прижился не везде, при этом он объединяет несколько довольно специфических типов, подробнее можно посмотреть в статье на Wikipedia, http://en.wikipedia.org/wiki/ICO_%28icon_image_file_format%29). С одной стороны, он позволяет представлять информацию без потерь (в отличие от JPEG). С другой стороны, он, фактически, аналог BMP, но на этом все сравнение быстро заканчивается.

Я не буду приводить точную его спецификацию (она не так сложна, и ее можно обнаружить, например, по следующему адресу: http://www.daubnet.com/formats/ICO.html), однако, сразу в глаза бросилось две вещи: возможность использования индексированных цветов (ага, это уже почти GIF) и возможность использования линейного сжатия (а это уже почти PNG). Уже интересно? Тогда мы продолжаем.

Собственно, следует эта информация из следующих частей формата:

```
BitCount        2 bytes         bits per pixel = 1, 4, 8
Compression     4 bytes         Type of Compression = 0
```

Боевое крещение

После небольших поисков удалось собрать тестовое окружение под Firefox 3, использующее base64-кодирование картинки в виде `.ico`. Удалось сделать однотонное изображение (палитра 4 бита) размером в 318 байтов (против 894 стандартных, меньше почти в 3 раза). С палитрой в 2 бита возникли трудности под Safari, корректный результат получить не удалось, однако, возможно, его также можно использовать.

Может быть, кому-то покажется, что 576 байтов — это очень мало. Но стоит заметить, что, во-первых, некоторые иконки используют, фактически, только 2 цвета, поэтому их можно сжать до еще меньшего размера. Во-вторых, при больших размерах (32x32, 48x48) выигрыш будет таким же в процентах. Т.е. иконки в 16 Кб можно будет спокойно уменьшить раза так в 3–7. И это без учета вырезания неиспользуемых фреймов в них (ведь формат позволяет создавать анимированные иконки).

Оптимальные размеры

Путем нехитрых вычислений заголовков, смещений и палитр можно получить некоторые цифры для размера наиболее стандартных `favicon.ico` (размер картинки 16x16 пикселей). Для 32x32 и 48x48 размер файлов должен увеличиться примерно в 4 и 9 раз, соответственно.

Палитра	Размер (в байтах)
2 бита	198
4 бита	318
8 бит	1406
24 бита	894
32 бита	1150

Таблица 4. Размер файла `favicon.ico` 16x16 в зависимости от используемой палитры

Для динамических иконок можно смело множить размер одиночной иконки на число фреймов, ибо заголовок у всего файла всего 62 байта, основная часть — именно данные.

PNG — быть или не быть?

В Wikipedia указывается, что вместо `.ico` можно использовать `.png` как наиболее перспективный из форматов, применяемых для сжатия изображения без потери качества. При соответствующем объявлении файла в секции `head` страницы позади планеты всей у нас остается Interner Explorer, так что данный подход может быть рассмотрен только как альтернативный. Подчеркну, что, в среднем, размер PNG-файла с иконкой не сильно меньше, чем ICO, но можно совершить дополнительные телодвижения, подключив одну иконку для всех браузеров, а вторую — только для IE.

А если еще и сжать?

Если мы не можем адекватно использовать нормальные форматы (PNG, GIF) для представления `favicon.ico`, то почему бы не задействовать `gzip`-сжатие для ее выдачи клиентскому браузеру? Можно. И все актуальные браузеры это понимают. Размер при этом составляет порядка 300 байтов (уменьшается в 3 раза по сравнению с исходным).

Повторюсь, речь идет о возможностях для уменьшения `favicon.ico` в целом, а не абсолютных цифрах. Если у вас на сервере уже используется сжатие, просто добавьте туда компрессию для `image/x-icon` и забудьте о ней.

`data:URI` нас спасет?

В качестве технологии экстремальной оптимизации можно рассмотреть возможность включения `favicon.ico` по протоколу `data:URI` (подробнее о нем написано в четвертой главе), чтобы отобразить страницу в клиентском браузере после первого запроса на сервер (подразумевается, что с сервера уйдет один-единственный HTML-файл, содержащий все необходимые составляющие в себе).

Однако для рядовых сайтов такой подход совершенно бессмысленный, потому что процедура будет отдавать пользователю лишние байты каждый раз. Самым логичным его

применением будут рекламные страницы, которые пользователь должен увидеть только один раз.

Заключение

Одним из наиболее спорных моментов в презентации Yahoo! было заявление о том, что `favicon.ico` «мешается» при загрузке страницы. Как можно судить по логам сервера при загрузке страницы, этот файл, действительно, запрашивается где-то в середине общего процесса загрузки, ориентировочно после CSS-файлов и до фоновых изображений, поэтому его оптимизация может оказаться одним из ключевых моментов для ускорения загрузки сайта в момент первого посещения (с пустым кэшем).

Также ради простого уважения к пользователям (зачем им загружать лишние 10 Кб кода, который отрисуется у них в области 16x16 пикселей?) не стоит раздувать его размер без особой необходимости. Уважайте своих посетителей.

2.7. Режем cookie

В качестве заключительного аккорда при рассмотрении уменьшения количества передаваемых данных между сервером и клиентом нужно обязательно упомянуть cookie.

Cookie являются одним из HTTP-заголовков, которые браузер посылает на сервер, а сервер вправе им ответить (если копнуть глубже, то существует пара заголовков: `Cookie` и `Set-Cookie` — но в данном случае это не так существенно). Общий размер HTTP-заголовков обычно не превосходит 500–1000 байтов, однако, cookie могут существенно его увеличить (так как на них накладывается ограничение в 4 Кб).

При объемах полезной информации в несколько Кб размер cookie может оказать критичное воздействие на скорость передачи данных. Давайте рассмотрим, какие существуют способы уменьшения этих издержек.

Оптимизируем размер, зону и время действия

В большинстве случаев пользователю просто не нужно пересылать огромные массивы данных каждый раз, для него вполне возможно ограничиться только сессионным ключом. Исходя из этого, стоит пересмотреть логику использования заголовков cookie и оставить только действительно необходимые.

Как вариант, можно устанавливать cookie только для определенных разделов на сайте либо ограничиваться только текущей сессией пользователя на сайте (которая не будет сохраняться при повторном заходе).

Также у cookie можно варьировать срок действия, что будет несколько нивелировать их влияние, если пользователь будет заходить на сайт достаточно редко: с каждым новым заходом cookie из браузера пересылаться не будут, однако, их будет отправлять сервер. Поэтому данная мера особой производительности не принесет.

Хостинг для компонентов без cookie

Для высоконагруженных проектов, которые активно используют cookie и стремятся минимизировать издержки от них, стоит рассмотреть вынос статических ресурсов на отдельный хост, для которого cookie вообще не будут устанавливаться.

В данном случае можно рассмотреть использование поддомена (что может оказаться бесполезным, если cookie выставляются на *.domain.ru) или домена верхнего уровня (в таком случае придется регистрировать отдельный домен для хранения статических ресурсов). Однако в обоих случаях возможны проблемы с локальными прокси-серверами: они могут отказаться кэшировать файлы с физически разных доменов.

Глава 3. Кэширование

3.1. Expires, Cache-Control и сброс кэша

Кэширование играет одну из основных ролей в быстродействии сайтов и сравнительно просто настраивается на стороне сервера. Веб-разработчики часто сталкиваются с кэшированием, ибо браузеры и проксирующие серверы, пытаясь ускорить работу сайтов для пользователя, очень часто стараются сохранить у себя максимально большое количество документов в локальном кэше.

При открытии страницы сайта в браузере (если эта страница была не первой на нем или если вы уже заходили на этот сайт в недалеком прошлом) браузер возьмет все (или почти все) ресурсы, необходимые для отображения страницы из кэша, чем весьма сильно может ускорить загрузку страницы. Однако при этом может потеряться актуальность представляемых данных, поэтому политика кэширования во всех браузерах реализована немного по-разному: каждый из них по-своему добивается компромисса между целостностью и актуальностью данных.

В качестве базовой настройки обычно используется инструкция браузеру от сервера для бессрочного кэширования каждого URL. Чтобы дать серверу понять, что файл был изменен, требуется использовать другое имя файла. На больших веб-сайтах обычно устанавливается такой процесс изменения этих файлов, что номер каждой новой версии добавляется к имени файла (например, `common.v1.css` становится `common.v2.css`). Соответственно, ссылки на эти файлы тоже должны быть программно обновлены, и это не зависит от языка программирования или системы шаблонов.

Настройка заголовка HTTP `Expires`

Заголовок `Expires` является частью спецификации `HTTP 1.0`. Когда HTTP-сервер отправляет ресурс (например, HTML-страницу или изображение) браузеру, он может дополнительно с ответом отправить этот заголовок с меткой времени. Браузеры обычно хранят ресурс вместе с информацией об истечении его срока действия в локальном кэше. При последующих пользовательских запросах к одному и тому же ресурсу браузер может сравнить текущее время и метку времени у закэшированного ресурса. Если метка времени указывает на дату в будущем, то браузер может просто загрузить ресурс из кэша вместо того, чтобы запрашивать его с сервера.

Даже если дата истечения срока действия ресурса находится в будущем, браузеры (включая Internet Explorer 4.0) по-прежнему выполнят дополнительный GET-запрос на сервер для определения, является ли закэшированная версия ресурса такой же, как на сервере. После тщательного анализа было установлено, что издержки на дополнительный запрос не являются ни оптимальными, ни необходимыми. По этой причине поведение всех браузеров было изменено следующим образом: если окончание срока действия закэшированного ресурса позже, чем время запроса, то браузер загрузит ресурс напрямую из кэша без запросов к серверу. Сайты, которые используются заголовок `Expires` для часто посещаемых, но редко обновляемых страниц, получат ощутимый выигрыш в трафике, а у пользователей быстрее отобразятся страницы.

Кэширование можно оптимизировать и дальше путем настройки заголовков, которые сервер отправляет при запросе требуемых ресурсов. Вместо того чтобы отдавать его

содержание, сервер может отправить заголовки, что у клиента уже имеется закэшированная версия файла, для которого установлено, что срок его хранения истечет в будущем. Картинки обычно так часто не обновляют, зачем же их запрашивать снова и снова? Подробнее этот аспект будет рассмотрен в следующем разделе.

Установка и `Cache-Control`, и `Expires` должна обеспечить поддержку протоколов HTTP 1.0 и 1.1. Типичный ответ сервера, урезанный только до заголовков с кэширующей информацией, может выглядеть следующим образом:

```
Date: Tue, 17 Apr 2008 18:39:57 GMT
Cache-Control: max-age=315360000
Expires: Fri, 14 Apr 2018 18:39:57 GMT
Last-Modified: Mon, 16 Apr 2008 23:39:48 GMT
```

Спецификация кэширования

В спецификации `RFC-2616` HTTP-кэшированию посвящена целая глава. В ней подробно рассматривается, что означают отдельные заголовки. Давайте остановимся на ключевых моментах.

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

```
Expires: Mon, 13 Oct 2019 00:00:00 GMT
```

`Cache-Control` определяет набор директив, относящихся непосредственно ко времени и специфике кэширования документа. Для запрета кэширования можно выставить его следующим образом:

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

Если же мы, наоборот, хотим положить ресурс в кэш браузера на достаточно продолжительный период времени, то стоит воспользоваться такой конструкцией:

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

в данном случае срок кэширования примерно равен году (60 * 60 *24 * 365 секунд).

Директива `Pragma: no-cache` используется для протокола HTTP/1.0. На данный момент можно с полным правом считать ее устаревшей конструкцией. Однако, для корректного запрета кэширования стоит все же выставлять и этот заголовок — никогда не знаешь наверняка, какой еще пользовательский агент обратится к серверу и какой протокол он будет использовать. Например, `wget` просто не поддерживает HTTP/1.1 (из-за `Content-Encoding: chunked`).

Практическое запрещение кэширования

Запретить кэширование можно и прямо из конфигурации Apache (подробная конфигурация для оптимальной производительности приводится в восьмой главе). Для этого нам нужны следующие строки:

```
# Проверяем, что подключен mod_headers
# Тогда выставляем заголовок Cache-Control
<IfModule mod_headers.c>
        Header append Cache-Control "no-store, no-cache, must-revalidate"
        Header append Pragma "no-cache"
</IfModule>

# Теперь проверяем наличие mod_expires и активируем этот модуль для заголовка
# Expires
<IfModule mod_expires.c>
        ExpiresActive On
        ExpiresDefault "now"
</IfModule>
```

Разрешение кэширования

При запрете кэширования мы заставим браузер каждый раз заново загружать документы и ресурсные файлы. В последнем случае это совсем не оптимально и может привести к заметному замедлению работы с сайтом. Давайте рассмотрим, как можно выставить срок действия кэша на достаточно продолжительное время.

```
# Разрешим кэширование на 1 год, проверив наличие mod_expires
# Apache сам позаботится о выставлении корректного max-age
<IfModule mod_expires.c>
        ExpiresActive On
        ExpiresDefault "access plus 1 year"
</IfModule>
```

В итоге, мы запретили браузерам загружать статические компоненты с сайта, чем заметно увеличили его производительность. Однако что же нам делать, если мы все же решим изменить исходный ресурсный файл?

Форсированный сброс кэша

Если мы устанавливаем время кэширования на несколько лет (фактически, на бесконечность), то нам нужно каким-то образом сообщить клиентскому браузеру, что исходный ресурс-то у нас поменялся: иначе браузер его никогда повторно не запросит. Что для этого нужно?

Вообще говоря, для того, чтобы сообщить об обновлении файла в таком случае, нужно изменить его адрес — т. е. заявить обновленный файл под другим URL, что и будет гарантировать его обновление в локальном кэше. Однако это можно сделать двумя способами. Во-первых, мы можем в конце файла обновить GET-строку запроса, например, используя номер версии

```
http://webo.in/a.css?v23
```

или дату последнего изменения

```
http://webo.in/a.css?20081010
```

Оба этих способа изменяют адрес ресурса (в данном случае, это файл стилей), поэтому браузер обязан его запросить.

Во-вторых, мы можем номер версии добавить в сам файл

```
http://webo.in/a.v23.css
```

чтобы исключить возможные проблемы с локальными проксирующими серверами, которые могут не кэшировать у себя файлы с GET-параметрами. В этом случае (дабы не плодить новые физические файлы) нам нужно прописать в конфигурации сервера (например, Apache), чтобы при запросах такого вида отдавался каждый раз один и тот же физический файл. Это можно сделать примерно следующим образом (справедливо для CSS- и JavaScript-файлов):

```
RewriteRule ^(.*)\.(v[0-9]+)?\.(css|js)$ $1.$2 [QSA,L]
```

Таким образом, вместо файла `a.v23.css` будет отдаваться `a.css`.

Если текущая конфигурация позволяет использовать последний вариант, то стоит остановиться на нем. Иначе сброс кэша придется осуществлять через обновления GET-параметров исходного файла.

«Пробивка» вечного кеширования с помощью подмены директории несколько лучше, чем использование GET-переменной. В качестве основного аргумента можно привести следующее рассуждение: если хоть где-то в цепочке от сервера до браузера есть кеширующий прокси, то по умолчанию он сочтёт запрос с "?" динамическим и отправит запрос на сервер, не пытаясь его искать у себя в локальном кэше. Браузер, разумеется, будет ждать в этом случае несколько больше.

В каком-то смысле это будет экономия «на спичках», так как эффект будет заметен только при посещении сайта другим пользователем той же локальной сети, к примеру. Но тем не менее, эффект есть, и процент запросов с заголовком `X-Forwarded-For` достаточно велик.

3.2. Кэширование в IE: `pre-check`, `post-check`

При разработке веб-сайта частота изменения страниц сильно колеблется. Некоторые страницы будут меняться ежедневно, некоторые останутся одними и тем же с самого момента своего создания. Для того, чтобы позволить сайту регулировать частоту, с которой браузер должен запрашивать HTTP-сервер об изменениях в ресурсе, в Internet Explorer 5 было введено 2 расширения HTTP-заголовка `Cache-Control`: `pre-check` и `post-check`. К сожалению, другие браузеры не поддержали инициативу, поэтому эти директивы на данный момент действительны только для IE.

Вводя эти расширения, Internet Explorer уменьшает сетевой трафик, так как отправляет меньше запросов к серверу. Дополнительно при этом улучшается пользовательское восприятие, когда браузер отображает ресурсы из кэша и проверяет обновления в фоновом режиме после специального интервала.

Спецификация

Расширения `post-check` и `pre-check` для `Cache-Control` определены следующим образом.

- `post-check`

Определяет интервал времени в секундах, после которого ресурс должен быть проверен на актуальность. Эта проверка может быть выполнена и **после** того, как пользователь загрузит страницу из кэша, но при следующей загрузке он обязательно должен получить обновленную версию.

- `pre-check`

  Определяет интервал времени в секундах, после которого проверка актуальности ресурса должна быть произведена **перед** его отображением для пользователя.

Рассматриваем подробнее

Рис. 15. Диаграмма работы `pre-check` *и* `post-check`

Когда к браузеру поступает запрос на открытие ресурса, который находится в кэше, и при этом кэш содержит расширения `Cache-Control` (отправленные с сервера как часть заголовка HTTP-ответа), тогда IE использует эти расширения и следующую логику для получения последней версии страницы с сервера.

- Если еще не закончился интервал времени `post-check`, то просто отобразим страницу из кэша.
- Если с момента последнего запроса страницы прошло время, лежащее между интервалами `post-check` и `pre-check`, то отобразим страницу из кэша. При этом в фоновом режиме запросим HTTP-сервер на предмет того, изменялась ли страница с момента последнего запроса браузером. Если страница изменилась, то запросим ее и сохраним в кэше. При этом в браузере ничего не поменяется: у него будет более старая версия, полученная изначально из кэша, однако, ее загрузка произойдет максимально быстро.
- Если уже истекло время, отмеченное интервалом `pre-check`, то при запросе страницы пользователем сначала спросим у HTTP-сервера, изменилась ли страница со времени ее последней загрузки браузером. Если страницы изменилась, загрузим ее и отобразим обновленную версию. Если страница не изменилась, то ее кэш и расширения `Cache-Control`, в любом случае, будут обновлены.

Заметим, что кнопка «Обновить» (включая клавишу `F5`) не запускает данный механизм, потому что «Обновить» **всегда** отправляет на сервер запрос `If-Modified-Since`. Однако, все ссылки будут открываться с помощью описанной выше логики.

Пример использования

В следующем примере сервер уведомляет Internet Explorer, что содержание документа не будет меняться в течение 1 часа (`pre-check=3600`) и что его можно загружать прямо из локального кэша. В случае же изменения страницы, если пользователь запросит ее по истечении 15 минут, Internet Explorer должен отобразить локальный кэш, но при этом в фоновом режиме проверить, является ли сохраненная копия страницы актуальной, и по необходимости загрузить ее с сервера.

```
Cache-Control: post-check=900,pre-check=3600
```

Использование описанных параметров для тонкой настройки общения сервера с IE может оказаться весьма полезным для высоконагруженных проектов, ориентированных на пользователей Internet Explorer. Это позволит как существенно уменьшить число запросов к серверу и сэкономить его ресурсы, так и поддерживать актуальность кэшируемых документов.

3.3. Last-Modified и ETag

Давайте рассмотрим, какие существуют альтернативы прямому ограничению повторных загрузок ресурсов со стороны сервера и сохранению их в пользовательском кэше.

`Last-Modified`

Дополнительно к заголовку `Cache-Control`, который предупреждает браузер, что последний может не запрашивать исходный документ с сервера некоторое время, будет полезно проверять версию ресурса каждый раз при запросе. Например, нам нужно, чтобы браузер пользователя проверял файлы стилей не реже, чем раз в день, но при этом сами файлы могут меняться достаточно редко (например, несколько раз в месяц). Для решения такой задачи и была придумана пара заголовков `Last-Modified` и `If-Modified-Since`.

Как это работает? Дополнительно к периоду кэширования сервер может также отправить заголовок `Last-Modified`, который будет обозначать время последнего изменения файла на сервере. Если у браузера есть уже такой файл в локальном кэше, то он может отправить на сервер `If-Modified-Since` с соответствующим временем. В том случае, если файл не менялся со времени последнего посещения, то сервер ответит статус-кодом 304 и не будет пересылать содержимое файла.

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

`ETag`

`ETag` (*англ. Entity Tags* — тэги сущностей) — механизм, который используют браузеры и веб-серверы, чтобы определить, является ли объект, находящийся в кэше браузера, таким

же, как соответствующий объект на сервере. Тэги сущностей являются почти полной аналогией `Last-Modified` заголовка за исключением того, что в качестве тэга может быть передана произвольная строка. Сервер указывает `ETag` для компонента, используя HTTP-заголовок `ETag`:

```
HTTP/1.1 200 OK
Last-Modified: Tue, 12 Dec 2008 03:03:59 GMT
ETag: "10c24bc-4ab-457e1c1f"
Content-Length: 19145
```

Позднее, если браузер хочет определить актуальность компонента, он передает заголовок `If-None-Match` для передачи `ETag` обратно на сервер. Если `ETag` совпадают, ответ от сервера приходит со статус-кодом 304, уменьшая, таким образом, объем передачи на 19145 байтов:

```
GET /b.png HTTP/1.1
Host: i.webo.in
If-Modified-Since: Tue, 12 Dec 2008 03:03:59 GMT
If-None-Match: "10c24bc-4ab-457e1c1f"
HTTP/1.1 304 Not Modified
```

Включить `ETag` для Apache можно, например, следующей директивой в конфигурации:

```
FileETag MTime Size
```

При этом `ETag` будет сформирован из даты изменения файла и его размера.

Синхронизация `ETag` и `Last-Modified`

Проблема `ETag` состоит в том, что обычно они используют атрибуты, специфичные в пределах одного сервера. `ETag` не совпадут, если браузер загрузит компонент страницы с одного сервера и попробует проверить его с другим сервером (у которого время изменения файла и(ли) номер блока на жестком диске для данного файла отличаются от первого) — ситуация очень распространенная, если вы используете кластер для обработки запросов. По умолчанию и Apache, и IIS включают в `ETag` такие данные, которые вряд ли дадут положительный результат при проверке на актуальность компонента на разных серверах.

Apache 1.3 и 2 генерирует `ETag` в формате `inode-size-timestamp`. Даже если один и тот же файл на разных серверах лежит в одной и той же папке, имеет те же права, размер и время, номер его иногда будет отличаться от сервера к серверу.

IIS 5.0 и 6.0 имеют похожий формат `ETag`: `Filetimestamp:ChangeNumber`. `ChangeNumber` — внутренняя переменная IIS для отслеживания изменений в конфигурации самого IIS, и нет гарантии, что эта переменная будет одинакова на всех серверах, обслуживающих веб-сайт.

В результате `ETag`, которые сгенерирует Apache или IIS для одного и того же файла, будут отличаться на разных серверах. Если `ETag` не будут совпадать, пользователь не будет получать маленького и быстрого ответа со статус-кодом 304 — собственно, ради чего `ETag` и разрабатывался. Взамен он будет получать стандартный код ответа 200 и далее весь запрошенный компонент.

Если сайт находится только на одном сервере, это не будет большой проблемой. Но если вы используете несколько серверов с Apache или IIS, устанавливающие `ETag` в соответствии с настройками по умолчанию, пользователи будут дольше загружать страницы, на серверах будет большая загрузка, нежели могла бы, вы будете тратить больше трафика, а прокси-серверы не будут кэшировать этот контент так, как хотелось бы. Даже если заголовок `Expires` будет установлен в далекое будущее, вам все равно будет приходить условный GET-запрос, когда пользователь перезагрузит страницу.

Если вы не получаете всех преимуществ, которые предоставляет `ETag`, тогда лучше совсем отключить его. Тэг `Last-Modified` позволяет проверять актуальность компонента на основании его времени изменения, а отключение `ETag` даст возможность уменьшить заголовки запроса и ответа. Если вы используете Apache, просто добавьте строку

```
FileETag none
```

в конфигурационный файл сервера.

3.4. Кэширование в iPhone

На MacWorld'2008 Steve Jobs анонсировал, что Apple уже продала на текущий момент 4 миллиона iPhone, что составляет по 20 тысяч iPhone каждый день. В докладе Net Applications говорится, что общая доля пользователей Интернета с iPhone поднялась до 0,12% в декабре 2007, обогнав, в совокупности, все браузеры для мобильных устройств, работающие под управлением Windows. iPhone от Apple изменил ситуацию для пользователей, выходящих в интернет через мобильные устройства.

Веб-разработчики теперь могут создавать функциональные и привлекательные приложения, которые работают на iPhone-версии браузера Safari для мобильных устройств. Однако при этом сам iPhone не только предоставляет новые и достаточно интересные возможности для разработчиков для мобильных устройств, но и обнаруживает ряд уникальных проблем в сфере производительности.

По поводу этого устройства на данный момент доступна только ограниченная информация, а понимание основ его алгоритмов кэширования является существенным для создания высокопроизводительного сайта. Команда по исключительной производительности Yahoo! рассмотрела кэширующие свойства iPhone. Основной акцент делался на исследовании следующих вещей:

- максимальный размер кэша для каждого компонента по отдельности;
- максимальный размер кэша для всех компонентов;
- эффект от `gzip`-сжатия для максимального размера кэша;
- остаются ли компоненты в кэше после перезагрузки iPhone.

Результаты экспериментов над кэшом как для Apple iPhone, так и для iPod Touch получились одинаковые.

Попадание в кэш

В результате исследования было установлено, что если размер компонентов превышает 25 Кб, то браузер в iPhone не кэширует этот компонент. Поэтому веб-страницы, которые

спроектированы специально для iPhone, должны уменьшить размер компонентов не более чем до 25 Кб, чтобы оптимизировать кэширующее поведение.

Хорошие новости заключаются в том, что если браузер загружает новый компонент, размер которого больше чем 25 Кб, то это не влияет на компоненты, которые уже находятся в кэше. Закэшированные компоненты заменяются более новыми, только если последние не превосходят 25 Кб. При этом выбирается самый маленький из последних используемых.

На www.apple.com указано, что существует предел в 10 Мб для индивидуальных компонентов. Этот предел зависит от способностей браузера хранить файлы в оперативной памяти (не на диске). Однако реальный максимальный размер файлов, которые iPhone может обрабатывать, значительно меньше. Он зависит от текущей фрагментации памяти и других приложений, которые запущены параллельно с браузером. Компоненты, которые не поместились в кэш, запрашиваются браузером снова после закрытия текущей страницы.

Было также установлено, что максимальный размер кэша для нескольких компонентов (или целой страницы со всеми необходимыми ей ресурсами) составляет 475–500 Кб.

Сжатые компоненты

Было проанализировано, какое влияние на кэш оказывает передача компонентов в обычном или сжатом виде. Предел кэша в 25 Кб на компонент не зависит от того, был ли он передан в архивированном виде. Safari в iPhone декодирует компонент **до того**, как он сохранится в кэше. Таким образом, значение имеет только несжатый размер файлов, что еще больше подчеркивает важность минимизации всех компонентов.

Однако размер все же имеет значение, потому что влияет на время передачи по сети. Поэтому рекомендуется разбивать все ресурсы страницы на файлы по 25 Кб, а затем уже применять к ним сжатие.

После перезагрузки

Иногда так случается, что пользователям iPhone или iPod Touch нужно перезагрузить систему, или, другими словами, выключить устройство и загрузить его заново. Для этого нужно удерживать кнопку `sleep` в течение пяти секунд, потом просмотреть небольшую заставку при выключении. Предположим, что пользователь просматривал ваш сайт как раз перед тем, как перезагрузиться. Сохранятся ли картинки и таблицы стилей в кэше браузера, ускорив загрузку вашего сайта, когда пользователь на него вернется?

В результате исследования было получено, что кэш браузера в iPhone не сохраняется после перезагрузки. Это означает, что кэш в Safari для iPhone получает часть системной памяти для создания там кэшированных версий компонентов, однако, не сохраняет их в постоянном месте.

Заключение

В случае реальной необходимости сайты можно и нужно проектировать специально для iPhone. Кроме повышения удобства использования стоит также обратить внимание на уменьшение общего размера страницы и улучшение клиентской производительности. В

данном случае нам нужно ограничить размер каждого из компонентов страницы 25 Кб для оптимизации кэширующего поведения.

Заданная ограниченная скорость сетевого беспроводного соединения в iPhone наряду с очисткой кэша в браузере при перезагрузке выводят на первый план важность уменьшения числа HTTP-запросов для повышения производительности. И это становится даже более важным, чем в случае загрузки страницы в обычном браузере. Подробнее об уменьшении числа запросов к серверу: техниках объединения файлов, использования CSS Sprites и `data:URI` — рассказывается в следующей главе.

Глава 4. Уменьшение числа запросов

4.1. Объединение HTML- и CSS-файлов

Число запросов является наиболее узким местом при загрузке страницы. По последним исследованиям порядка 40% времени загрузки уходит только на установление новых соединений с сервером. В этом свете любые методы, позволяющие уменьшить число запросов, выглядят весьма перспективно. Однако каждый такой метод, начиная с простого объединения стилей или скриптов и заканчивая `data:URI`, достаточно сложен в технологическом плане, поэтому в ряде случаев может просто не окупать затраченного времени.

Зачастую cookie выставляются на весь домен или даже на все поддомены, что означает их отправку браузером даже при запросе каждой картинки с вашего домена. В результате 400-байтный ответ с картинкой превратится в 1000 байтов или даже больше, в зависимости от добавленных заголовков cookie. Если на странице у вас много некэшируемых объектов и большие cookie на домен, то стоит рассмотреть возможность вынесения статичных ресурсов на другой домен (например, так поступил Яндекс, расположив статические файлы на домене yandex.net) и убедиться, что cookie там никогда не появятся.

В силу накладных расходов на передачу каждого объекта, один большой файл загрузится быстрее, чем два более мелких, каждый в два раза меньше первого. Стоит потратить время на то, чтобы привести все вызываемые JavaScript-файлы к одному или двум, равно как и CSS-файлы. Если на вашем сайте их используется больше, попробуйте сделать специальные скрипты для публикации файлов на «боевом» сервере или уменьшите их количество. Если на странице в большом объеме располагаются десятки небольших GIF-файлов (для оформления границ или фона элементов), стоит рассмотреть ее преобразование в более простой CSS-дизайн (который не потребует такого большого числа картинок) и(ли) объединение в несколько больших ресурсных файлов.

Для объединения HTML-файлов существует достаточно простое правило по сведению числа фреймов на странице к минимуму (в идеале, их вообще не должно быть, ибо каждый фрейм влечет создание нового документа в дереве страницы, что достаточно ресурсоемко). Поэтому давайте рассмотрим, что можно сделать с файлами стилей.

CSS-файлы в начале страницы

При заботе о производительности веб-страниц мы всегда хотим, чтобы страницы могли быть отрисованы постепенно, чтобы браузер мог отобразить любой контент сразу же, как он у него появится. Это особенно важно для страниц, на которых много текстового содержания, и для пользователей с медленным подключением. Важность визуального оповещения пользователя о текущем состоянии загрузки страницы каким-нибудь индикатором прогресса детально изучена и задокументирована. Однако в любом случае, всегда лучше, если в роли индикатора прогресса выступает сама страница. Когда браузер загружает HTML-файл постепенно — сначала заголовок, потом навигацию, логотип наверху и т.д. — все это служит отличным индикатором загрузки для пользователя. Также это улучшает общее впечатление от сайта.

Размещение CSS в конце страницы не позволяет начать постепенное отображение многим браузерам, в числе которых находится и Internet Explorer. Браузер не начинает визуализировать страницу, чтобы не пришлось перерисовывать элементы, у которых во время загрузки изменится стиль. Firefox начинает сразу отрисовывать страницу, в процессе загрузки, возможно, перерисовывая некоторые элементы по мере изменения их свойств, но это является причиной появления нестилизованного контента и рекурсивного его обновления.

Спецификация HTML 4 устанавливает, что таблицы стилей должны быть включены в `head` документа: «В отличие от `<a>`, `<link>` может появляться только в секции `<head>`, зато там он может встречаться сколько угодно раз». Ни одна из альтернатив — белый экран или показ нестилизованного контента — не стоит этого риска (хотя разработчики Firefox и Opera думают несколько иначе). Оптимальным решением является следование спецификации и включение CSS в `head`-секцию документа.

При проектировании небольших сайтов либо несложных дизайнов это правило является основным для максимизации производительности. Однако далее в этой главе мы рассмотрим и альтернативную его трактовку.

Объединение CSS-файлов

Зачастую на странице подключается несколько файлов стилей: это может быть связано как с модульной структурой построения CSS, так и с поддержкой различных устройств просмотра веб-страниц. Давайте рассмотрим последний случай: у нас есть два вызова CSS-файлов на странице, например:

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

где первый используется для отображения страницы на экране монитора, а второй — для предварительного просмотра и печати.

Проблема в том, что браузер не отображает любую часть страницы (это не касается Opera: у нее время отображения страницы без полной загрузке файлов стилей задано по умолчанию в настройках, посмотреть их можно следующим образом: `'preferences' (ctrl-f12) -> 'advanced' -> 'browsing' -> 'loading'` или `'инструменты' -> 'настройки' -> 'дополнительно' -> 'перемещение' -> 'загрузка'`), пока не загрузит все файлы стилей — в том числе и те из них, которые не предназначены для устройства, с помощью которого производится отображение страницы. Другими словами, браузер не покажет страницу, пока не загрузит и файл стилей для принтера, хотя он совсем и не требуется для визуализации страницы. Это неправильно с точки зрения производительности, но это так (Safari, на самом деле, ведет себя как раз «правильно»: ненужные файлы не задерживают загрузку, но это связано с особенностью модели визуализации, о нем более подробно рассказывается ниже).

Практическое решение

Решение выглядит весьма тривиально: мы можем в общем CSS-файле объявить правила для любого устройства через `@media`. Например, все стили для принтера могут быть записаны в следующем виде

```
@media print {
        стилевые правила для принтера
}
```

в конце основного файла стилей. Таким образом, будет загружаться всегда только один файл. Данное решение может быть легко автоматизировано, и некоторые CMS уже применяют этот подход (в частности, Drupal).

Если у нас CSS-файлы разбиты на модули, то нужно пересмотреть их структуру таким образом, чтобы на каждую страницу приходилось не более двух файлов (небольшие файлы — порядка 5 КБ — можно объединить в один для целого раздела). Для главной страницы я рекомендую всегда ограничиться только одним файлом либо вообще включать его в HTML-код (как сделано, например, для главной страницы Яндекса).

Два слова об условных комментариях

Очень часто верстка страниц производится таким образом, что у нас появляется основной файл стилей и несколько дополнительных, рассчитанных на конкретные браузеры (речь идет, в основном, о Internet Explorer, однако, иногда требуются какие-то специальные правила для Firefox, Opera или Safari). В этом случае файлы подключают через так называемые «условные комментарии», которые выглядят как обычные HTML-комментарии для всех браузеров, кроме Interner Explorer (у остальных браузеров есть свои способы загрузить какой-то файл стилей только для них).

Финальная конструкция выглядит примерно следующим образом:

```
<link type="text/css" rel="stylesheet" href="main.css" media="screen" />
<!--[if lt IE 7]><link type="text/css" rel="stylesheet"
        href="ie6.css" media="screen" /><![endif]-->
```

Для всех браузеров используется `main.css`, а для IE6 и ниже — `ie6.css`. Однако Interner Explorer этих версий не запрашивает файлы стилей параллельно, поэтому при загрузке страницы произойдет ненужная задержка, связанная с доставкой еще одного файла.

Чтобы избежать ее (особенно в случае небольшого количества стилей конкретно для IE), можно использовать CSS-хаки уже в исходном CSS-файлы. Например, если нам нужно определить правило только для IE7, мы можем написать так:

```
*+html body {
        margin: 0 auto;
}
```

для IE6:

```
* html body {
        margin: 0 auto;
}
```

и для IE5.5- (эта группа браузеров не распознает экранирование символов, поэтому сможет применить только первое правило из двух, второе правило отработает для IE6, переопределив первое):

```
* html body {
```

```
        margin: 0;
        marg\in: 0 auto;
}
```

CSS-хаки позволяют совершенно свободно использовать всего один файл стилей для гарантии кроссбраузерности верстки. При этом производительность страницы будет максимальной (исключая, конечно, случай включения всех CSS-правил в исходный HTML-документ — это будет работать еще быстрее, однако, чревато некоторыми сложностями, подробнее о них в конце этой главы).

4.2. Объединение JavaScript-файлов

Все внешние JavaScript-файлы с сайта можно слить в один большой, загружаемый только один раз и навсегда. Это очень даже хорошо: браузер не делает тысячу запросов на сервер для отображения одной страницы, скорость загрузки резко повышается. А пользователи так же счастливы, как и разработчики.

Как всегда, в бочке меда есть ложка дегтя: в объединенный файл попадает много того, что при первом запросе можно было бы и не загружать. Чаще всего для борьбы с этим предлагают ненужные части убирать вручную. Однако каждый раз делать одни и те же операции после изменения модулей очень надоедает. Ниже приведено описание простейшего алгоритма разрешения этой проблемы путем описания зависимостей между модулями.

Конструктивные предложения

Для начала стоит разобрать используемый фреймворк на составные части. JSON — отдельно, AJAX — отдельно, работа с DOM — отдельно, формы — отдельно. После этого задача «выкидывания ненужного» превращается в задачу «собери только нужное». Несомненный плюс — результат сборки стал гораздо меньше. Несомненный минус — если что-то из «нужного» забыто, все перестает работать.

Информация о зависимостях между составными частями можно хранить в удобном для автоматического использования виде. (Формы используют функции DOM, JSON — AJAX и так далее.) На этом шаге забыть что-то нужное становится заметно труднее, а сборка превращается из увлекательной головоломки в рутинную и автоматизируемую операцию.

Также можно хранить информацию о том, какие именно модули нужны сайту в целом. Используется ли AJAX? Если ли формы? Может быть, какие-то необычные элементы управления?

Да, естественно, все это можно и нужно автоматизировать.

В теории

С формальной точки зрения, после того как первые два предложения воплощены в жизнь, у нас появляется дерево зависимостей. Например, такое:

```
— dom.js
        — array.map.js
                — array.js
        — sprinf.js
```

```
— calendar.js
        — date.js
        — mycoolcombobox.js
                — dom.js
                        — array.map.js
                                — array.js
                        — sprinf.js
— animated.pane.js
        — pane.js
                — dom.js
                        — array.map.js
                                — array.js
                        — sprinf.js
                                — animation.js
                — transition.js
... и так далее ...
```

Дальше мы выбираем непосредственно нужные сайту вершины. Пусть это будут `dom.js` и `animated.pane.js`. Теперь дело техники — обойти получившийся набор деревьев в глубину:

```
— array.js
— array.map.js
— sprinf.js
— dom.js
— array.js
— array.map.js
— sprinf.js
— dom.js
— pane.js
— transition.js
— animation.js
— animated.pane.js
```

...удалить повторяющиеся элементы:

```
— array.js
— array.map.js
— sprinf.js
— dom.js
— pane.js
— transition.js
— animation.js
— animated.pane.js
```

и слить соответствующие модули воедино.

На практике

Хранить информацию о зависимостях можно, например, следующим образом (добавляя в «модули» служебные комментарии):

```
// #REQUIRE: array.map.js
// #REQUIRE: sprintf.js
....
код
```

Выделить подобные метки из текстового файла не составляет труда. Естественно, чтобы получить полное дерево зависимостей, надо будет пройтись по всем доступных файлам — но полное дерево обычно не нужно.

К чему мы пришли?

Затратив **один раз** кучу времени на формирование модулей и зависимостей между ними, мы экономим время **каждый раз**, когда хотим уменьшить объем загружаемого клиентов внешнего файла. Приятно. Но все-таки часть проблемы осталась — пользователь загружает весь JavaScript-код, который используется на сайте, в течение первого захода на произвольную страницу, даже если на ней весь этот код не нужен.

Итак, мы оставили нового пользователя наедине с единственным JavaScript-файлом, не включающем ничего лишнего. Стал ли при этом пользователь счастливее? Ничуть. Наоборот, **в среднем** пользователь стал более несчастным, чем раньше, а причина этому — **увеличившееся** время загрузки страницы.

Немного из теории HTTP-запросов

Время загрузки ресурса через HTTP-соединение складывается из следующих основных элементов:

1. время отсылки запроса на сервер T1 — для большинства запросов величина практически постоянная;
2. время формирования ответа сервера — для статических ресурсов, которые мы сейчас и рассматриваем, пренебрежимо мало;
3. время получения ответа сервера T2, которое, в свою очередь, состоит из постоянной для сервера сетевой задержки L и времени получения ответа R, прямо пропорционального размеру ресурса.

Итак, время загрузки страницы будет состоять из времени загрузки HTML-кода и всех внешних ресурсов: изображений, CSS- и JavaScript-файлов. Основная проблема в том, что CSS и JavaSscript-файлы загружаются последовательно (разработчики браузеров уже работают над решением этой проблемы в последних версиях, однако, пока еще 99% пользователей страдают от последовательной загрузки). В этом случае общение с сервером выглядит так:

```
— запросили страницу
— получили HTML
— запросили ресурс A: T1
— получили ресурс A: L + R(A)
— запросили ресурс B: T1
— получили ресурс B: L + R(B)
— запросили ресурс C: T1
— получили ресурс C: L + R(C)
```

Общие временные затраты при этом составят 3(T1+L) + R(A+B+C).

Объединяя файлы, мы уменьшаем количество запросов на сервер:

```
— запросили страницу
— получили HTML
— запросили ресурс A+B+C: T1
```

```
— получили ресурс A+B+C: L + R(A + B + C)
```

Очевидна экономия в 2(T1 + L).

Для 20 ресурсов эта экономия составит уже 19(T1 + L). Если взять достаточно типичные сейчас для домашнего / офисного интернета значения скорости в 256 кбит/с и пинга ~20–30 мс, получим экономию в 950 мс — **одну секунду** загрузки страницы. У людей же, пользующихся мобильным или спутниковым интернетом с пингом более 300 мс, разница времен загрузки страниц составит 6–7 секунд.

На первый взгляд, теория говорит, что загрузка страниц должна стать быстрее. В чем же она **разошлась** с практикой?

Суровая реальность

Пусть у нашего сайта есть три страницы P1, P2 и P3, поочередно запрашиваемые новым пользователем. P1 использует ресурсы A, B и C, P2 — A, C и D, а P3 — A, C, E и F. Если ресурсы не объединять, получаем следующее:

- P1 — тратим время на загрузку A, B и C
- P2 — тратим время на загрузку **только** D
- P3 — тратим время на загрузку E и F

Если мы слили воедино абсолютно все JavaScript-модули сайта, получаем:

- P1 — тратим время на загрузку (A+B+C+D+E+F)
- P2 — внешние ресурсы не требуются
- P3 — внешние ресурсы не требуются

Результатом становится увеличение времени загрузки самой первой страницы, на которую попадает пользователь. При типовых значениях скорости/пинга мы начинаем проигрывать уже при дополнительном объеме загрузки в 23 Кб.

Если мы объединили только модули, необходимые для текущей страницы, получаем следующее:

- P1 — тратим время на загрузку (A+B+C)
- P2 — тратим время на загрузку (A+C+D)
- P3 — тратим время на загрузку (A+C+E+F)

Каждая отдельно взятая страница при пустом кэше будет загружаться быстрее, но все они вместе — медленнее, чем в исходном случае. Получаем, что слепое использование модного сейчас объединения ресурсов часто только **ухудшает жизнь пользователя.**

Возможное решение

Конечно же, выход из сложившегося положения есть. В большинстве случаев для получения реального выигрыша достаточно выделить «ядро» — набор модулей, используемых на всех (или, по крайней мере, на часто загружаемых) страницах сайта. Например, в нашем примере достаточно выделить в ядро ресурсы A и B, чтобы получить преимущество:

- P1 — тратим время на загрузку (A + B) и C
- P2 — тратим время на загрузку D
- P3 — тратим время на загрузку (E + F)

Вдумчивый читатель сейчас возмутится и спросит: «А что, если ядра нет? Или ядро получается слишком маленьким?». Ответ: это легко решается вручную выделением 2–3 независимых групп со своими собственными ядрами. При желании задачу разбиения можно формализовать и получить точное машинное решение — но это обычно не нужно; руководствуясь простейшим правилом — чем больше ядро, тем лучше, — можно добиться вполне приличного результата.

Реализация на PHP

После разделения JavaScript- и CSS-кода по файлам для поддержания модульной структуры можно в контроллере создать список файлов, которые надо присоединить к данному документу (вместо того чтобы прописывать это вручную в шаблоне отображения). Но теперь надо сделать так, чтобы до показа шаблона вызывалась функция кэширования, которая проходилась бы по списку, проверяла из него локальные файлы на время изменения, объединяла в один файл и создавала или перезаписывала `gz`-файл с именем, сформированным из md5-хэша имен входящих файлов.

В качестве рабочего примера можно привести следующую функцию:

```
function cache_js(){
        $arrNewJS=array();
        $strHash='';
        $strGzipContent='';
        $intLastModified=0;

//проходимся по списку файлов
        foreach ((array)$this->scripts as $file){
                if (substr($file,0,5)=='http:') continue;
                if ($file[0]=='/') $strFilename=sys_root.$file;
                else $strFilename=sys_root.'app/front/view/'.$file;
                $strHash.=$file;

//читаем содержимое в одну строку
                $strGzipContent.=file_get_contents($strFilename);
                $intLastModified=$intLastModified<filemtime($strFilename) ?
                        filemtime($strFilename) : $intLastModified;
        }
        $strGzipHash=md5($strHash);
        $strGzipFile=sys_root.'app/front/view/js/bin/'.$strGzipHash.'.gz';

//проверяем, надо ли перезаписать gz-файл
        if (file_exists($strGzipFile) &&
$intLastModified>filemtime($strGzipFile) || !file_exists($strGzipFile)){
                if (!file_exists($strGzipFile)) touch($strGzipFile);

//используем функции встроенной в php библиотеки zlib для архивации
                $gz = gzopen($strGzipFile,'w9');
                gzputs ($gz, $strGzipContent);
                gzclose($gz);
        }

//перезаписываем список на один файл
        $arrNewJS[]='js/bin/'.$strGzipHash.'.gz';
```

```
        $this->scripts=$arrNewJS;
}
```

Для CSS основные теоретические моменты описаны выше, а реализация даже несколько проще. Если использовать YUI Compressor, то решение будет совершенно одинаково (вычислили зависимости, склеили файлы, сжали, переименовали, сделали архив) для обоих типов файлов.

PHP Speedy

К сожалению, почти все описанные выше методы применимы только на стадии разработки или существенной оптимизации и требуют участия опытных разработчиков для своей интеграции. Но возникает резонный вопрос: может быть, уже существуют какие-либо автоматизированные решения для автоматического объединения CSS- или JavaScript-файлов? И что делать, если хочется ускорить сайт на существующей платформе одной из популярных CMS, где в коде уже «сам черт ногу сломит»?

В таких случаях можно использовать проект с открытым кодом PHP Speedy (http://aciddrop.com/php-speedy/) — PHP-скрипт, который обеспечивает расширенное кэширование и сжатие компонентов страницы, не требуя никаких модификаций вручную. Достаточно только установить скрипт на сервере и сделать несложные настройки, заключающиеся в указании директории с самим сайтом и папок для кэширования файлов. Скрипт умеет автоматически склеивать все CSS- и JavaScript-файлы, кэшировать их, применяет оптимизацию (с помощью пакета Minify, http://code.google.com/p/minify/ , о котором уже шла речь выше), а также `gzip`-сжатие. На выходе мы получаем автоматическую оптимизацию сайта совершенно бесплатно. Хотя, конечно, не следует рассматривать это как конец — для начала такое решение вполне приемлемо, однако, для достижения максимальной производительности сайта придется со временем все больше и больше настраивать некоторые моменту вручную и применять советы из этой книги.

Для владельцев блогов на популярном движке Wordpress есть специальное расширение на основе этого решения, добавляющее оптимизацию буквально одним кликом в панели управления. Решение очень даже полезное: блоги часто расположены на слабых хостинговых площадках, а популярность проекта может возрасти буквально после пары публикаций. Площадка может быть к этому просто не готова, да сам Wordpress не является вершиной оптимизации с точки зрения нагрузки.

4.3. Техника CSS Sprites

Рассмотрев все аспекты объединения текстовх файлов, перейдем к графической и мультимедийной информации. Сейчас уже много где написано и упомянуто про технику CSS Sprites (или CSS Image Maps). Ниже приведены несколько примеров и полезных ссылок. И пара советов, где и как этот метод может быть применим наиболее оптимальным образом.

Сама техника заключается в том, что мы создаем комбинированное изображение, из которого затем «вырезаем» с помощью CSS-свойства `background-position` нужный нам в данном случае кусок. На текущем уровне поддержки браузерами (порядка 99,9%) она является просто обязательной для любого уважающего себя веб-ресурса, так как позволяет сократить число запросов к серверу — а кроме этого отделить поведение от представления, и возложить труд по анимации на CSS-движок браузера, а не на JavaScript-

движок (т.е. это будет работать даже с выключенными скриптами), и много-много прочих «вкусностей». Но обо всем по порядку.

Простой rollover-эффект

Обычно таким термином называют смену графической картинки при наведении на нее мыши, своеобразный призыв к действию (как его любят называть маркетологи). У вебмастеров сложилась дурная практика делать такие эффекты через `onmouseover/onmouseout` на картинках.

Это прямое нарушения принципа разделение представления от поведения и несемантическая (в лучшем случае, в худшем — еще и невалидная) верстка. И вообще, это очень плохо. В данном случае это делается средствами CSS и является семантически-правильным (в большинстве HTML-документов — это ссылка, но с элементами форм приходится немного повозиться, однако, тоже ничего сверхъестественного).

Пример: при наведении просто показывается другая картинка.

Рис. 16. Пример фонового изображения для простого rollover-эффекта. Источник: www.websiteoptimization.com

Соответствующая часть в CSS-коде будет выглядеть примерно так:

```
a.sprited {
        background: yellow url(http://site.ru/img/button.png) 0 0 no-repeat;
        width: 100px;
        height: 20px;
}
a.sprited:hover {
        background-position: -100px 0;
        background-color: red;
}
```

При наведении на ссылку фоновое изображение под ней «уедет», и покажется правая его часть (ибо она изначально скрыта жесткими размерами самой кнопки). Дополнительно мы обеспечили обратную совместимость для тех пользователей, у которых отключены картинки. Для них мы меняем цвет фона при наведении (если картинки в браузере включены, то изображение покажется поверх цвета).

Сложный rollover-эффект

Под таким термином стоит понимать те случаи, когда в одном файле содержатся нескольких «динамических кнопок». Например, это может быть такое изображение:

Рис. 17. Пример фонового изображения для сложного rollover-эффекта. Источник: www.spegele.com

Проблемные места в IE

В некоторых версиях IE изменение расположения фона при наведении мыши отрабатывает не совсем корректно, и браузер запрашивает исходную фоновую картинку с сервера еще раз, что вызывает «мигание» картинки. Одним из вариантов борьбы с такой проблемой может стать изменение не позиции фонового элемента, а его прозрачности. Например, мы можем сразу задать параметры фона для элемента и его прямого потомка, а при наведении мыши менять фон дочернего элемента на прозрачный, тогда фон родителя будет просто просвечивать:

```
<a href="/"><span>Начало</span></a>

a {
        background: yellow url(http://site.ru/img/button.png) 0 0 no-repeat;
        display: block;
        height: 20px;
        width: 100px;
}
a span {
        background: red url(http://site.ru/img/button.png) -100px 0 no-repeat;
        display: block;
        height: 20px;
        width: 100px;
}
a:hover span {
        background: transparent;
}
```

К сожалению, этот метод предполагает появление у элемента несемантического потомка для обеспечения графических эффектов. Более стандартным вариантом будет вызов специфичного для IE метода `backgroundImageCache` (через `try` или любое другое условие, гарантирующее обратную совместимость с остальными браузерами):

```
try {
        document.execCommand("BackgroundImageCache", false, true);
} catch (e) {}
```

В данном случае мы форсируем кэширование фоновых изображений, что предотвращает описанную выше ошибку.

CSS Image map

Этот пункт стоит намеренно выделить, ибо он подразумевает более свободное использование ресурсного файла для «подсветки» какого-либо изображения при наведении. Если в предыдущих случаях области были одинакового размера, то тут уже размер областей может быть, вообще говоря, произвольным. Одним из преимуществ такого подхода является совмещение разных областей, чтобы они занимали минимум места. Эта техника как раз и заменила классический Image Map.

Рис. 18. Пример изображения для CSS Image Map. Источник: www.acronis.com

Статичные картинки

Кроме динамических эффектов CSS Sprites широко используется и для объединения статических изображений. Давайте рассмотрим различные плюсы и минусы этого подхода.

Основной опасностью склеивания большого количества иконок в одном месте являются артефакты при увеличении шрифта: посторонняя часть фонового изображения проявляется совершенно не в том месте, где его ожидали, и у пользователя возникает ощущение, что страница «разваливается». Блок становится выше или длиннее, запаса полей данного изображения уже не хватает, в результате у одного элемента отображается сразу несколько иконок. Непорядок. Как с ним бороться, будет рассказано немного ниже в общих советах по созданию ресурсных картинок для CSS Sprites.

В общем случае нужно жестко ограничивают размеры контейнера, у которого заданы определенные фоновые картинки, чтобы даже при увеличенном тексте картинка не «ломалась» (однако, текст может стать нечитаемым из-за обрезания по границе блока). Как бороться с последней напастью, также будет рассказано в конце раздела.

В некоторых случаях возможно также объединение всех фоновых изображений на странице в одно-единственное. Такой подход значительно сокращает число запросов к серверу, однако, влечет и технологические сложности. Как пример можно привести следующее изображение:

Рис. 19. Пример фонового изображения «все-в-одном». Источник: webo.in

Онлайн-генераторы

www.csssprites.com. Обладает довольно минималистичным дизайном, есть возможность загружать несколько исходных файлов.

www.printf.ru/spritr/. В этом инструменте есть возможность загружать несколько файлов, очень милый дизайн, но, в целом, настроек мало.

spritegen.website-performance.org. Тут очень много настроек, также можно гибко создавать и сам CSS-фрагмент, но все картинки нужно загружать одним архивом.

Полезные советы

Итак, самое вкусное. Для начала стоит разбить все фоновые картинки на 5 групп (однако, пятая группа может быть объединена с любой из первых четырех).

1. Анимированные картинки
2. Те, которые предполагается повторять по всем направлениям (`repeat`)
3. Те, которые предполагается повторять по горизонтали (`repeat-x`)
4. Те, которые предполагается повторять по вертикали (`repeat-y`)
5. И те, которые предполагается показывать только один раз (`no-repeat`)

Откуда взялось разделение на такие группы? Из очень простых соображений: если картинка будет повторяться по какому-то направлению, то по этому направлению она должна быть одна-единственная в своем «окне», иначе повторяться будет не только она одна. Также стоит ориентироваться на общий размер файла в 10–20 Кб: если файл получается больше, то лучше подключать больше одного (соображения по разбиению файлов по размеру более подробно приведены в пятой главе).

Далее, все картинки из второй группы (повторяющиеся по всем направлениям) оставляем как есть (вообще говоря, можно подумать над их преобразованием в стилевые правила для самых простых случаев, но это уже тема для отдельной дискуссии). Все картинки из третьей группы можно склеить по вертикали (тогда в своем горизонтальном окне они будут единственными), все картинки из четвертой группы — по горизонтали. Что же делать с пятой группой?

Тут нам нужно понять, для какой цели будет использовать каждая картинка. Если она будет изображать фиксированную по размерам кнопку, то ее можно размещать в любом месте итогового ресурсного файла. Если она будет использована как иконка для списка (размещение в левом верхнем углу элемента), то мы должны очистить все пространство правее и ниже ее. Таким образом, при любом увеличении такого элемента (а «растут» элементы у нас всегда вниз и вправо) ничего лишнего не выводилось. В таком случае иконки располагаются не вертикально, а «лесенкой» (из правого верхнего угла в левый нижний).

Рис. 20. Пример фонового изображения с расположением картинок «лесенкой». Источник: webo.in

Описанная выше проблема с изменением размера надписей в фиксированных кнопках (фон у них фиксированный, поэтому мы не можем их раздвигать при увеличении шрифта, и он обрезается) может быть преодолена путем разбиения фона на 4 части (угловые) и задания соответствующего цвета фона для всех элементов. Однако это повлечет наличие, как минимум, 4 вложенных элементов для отображения каждого угла. Не во всех случаях это допустимо семантически. Да, можно создавать дополнительную разметку при помощи JavaScript, но насколько оно того будет стоить? Это лучше решать в каждом конкретном случае.

Давайте теперь перейдем к методам, позволяющим вставлять графическую информацию непосредственно в текстовые файлы без необходимости запроса внешних ресурсов.

4.4. Картинки в теле страницы с помощью `data:URI`

Встроенные изображения используют схему `data:URI` для внедрения прямо в тело веб-страницы. Как было определено в RFC 2397, такие URI предназначены для вставки небольших объектов как «непосредственных данных» (которые можно использовать без дополнительных запросов внешних ресурсов). Такие объекты должны рассматриваться так же, как и любые другие внешние файлы. Использование встроенных изображений позволяет сэкономить HTTP-запросы к внешним ресурсам.

Поддержка браузерами

Хотя Opera 7.2+, Firefox, Safari, Netscape и Mozilla поддерживают `data:URI`, Internet Explorer 5–7 — решительно нет. Однако, Internet Explorer 8 будет поддерживать эту схему. Существует также несколько приемов для поддержки старых версий Internet Explorer (о них чуть ниже).

Схема `data:URI`

Схема `data:URI` предоставляет способ для внедрения «непосредственно данных» точно так же, как если бы они были подключены через вызовы внешних файлов. Синтаксис у нее следующий:

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

В случае простых изображений вам нужно указать mime-тип для них (например, `image/gif`), за ним идет base64-представление бинарного файла с изображением. Ниже приведен пример (переводы строк добавлены, чтобы не разрывать страницу, на самом деле, их нет):

```
<img src="
AAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBJbWFnZVJlYWR5ccllPAAAAPJJREFUKM9tjz1OwlEQxH8P/h
V2NIZY0NhYeA0TbkLLPTyFV6DgLhYWFIaOmEhM3szbtXhEPmSy2Z3d2Y9sORySEyKih87iCg4GYDI
ByEwoQGbPCowzR3mG3e576Jsz85zkLZRSIqIsFrlc5n5PBK1la0Rka2lfeDun07JafQ2bTTw/l+0W
Oy3klFLKWq/9fA4wADZS/g10ufdVpeqxYheIAehHq9Li1PrvgpQQw5rxk15/6mfYWR1yVIUc0pFUN
ync7vyw5m14gbHfQx+3l3di4Vba4z0MASOZ2Swl3LCQitQ/w8amtW4B5QBxZlymVxLwCz+JZR4AeS
rEAAAAAElFTkSuQmCC" width="16" height="14" alt="внедренная иконка"/>
```

В результате мы получим следующее изображение иконки флага:

Рис. 21. Пример изображения, вставленного с помощью `data:URI`. Источник webo.in

CSS и встроенные изображения

Такие изображения, внедренные в HTML-страницы, не кэшируются для повторного использования. И они не кэшируются от странице к странице (это логично: ведь нам нужно каждый раз загрузить HTML-код для отображения этой картинки, они будут кэшироваться только с HTML, их содержащим). Однако CSS-файлы замечательно кэшируются браузерами, и такие изображения могут быть повторно применены вместе с использующим их селектором, например:

```
ul {
        list-style: none;
}
ul li {
        margin: 0 0 1px;
        background:url(
AIAAAD5gJpuAAAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBJbWFnZVJlYWR5ccllPAAAAPJJREFUKM9tjz
1OwlEQxH8P/hV2NIZY0NhYeA0TbkLLPTyFV6DgLhYWFIaOmEhM3szbtXhEPmSy2Z3d2Y9sORySEyK
ih87iCg4GYDIByEwoQGbPCowzR3mG3e576Jsz85zkLZRSIqIsFrlc5n5PBK1la0Rka2lfeDun07Ja
fQ2bTTw/l+0WOy3klFLKWq/9fA4wADZS/g10ufdVpeqxYheIAehHq9Li1PrvgpQQw5rxk15/6mfYW
R1yVIUc0pFUNync7vyw5m14gbHfQx+3l3di4Vba4z0MASOZ2Swl3LCQitQ/w8amtW4B5QBxZlymVx
LwCz+JZR4AeSrEAAAAAElFTkSuQmCC) 0 0 no-repeat;)
        height: 14px;
        text-indent: 10px;
}
```

Проблемы `data:URI`

С описанным выше подходом для подключения изображений связаны две основные проблемы. Во-первых, вам нужно пересчитывать base64-представление изображений и редактировать CSS-файл каждый раз, когда само изображением меняется. Также IE до версии 7 включительно не поддерживает встроенных изображений. У первой проблемы есть простой решение на PHP:

```
<?php echo base64_encode(file_get_contents("../images/flag.png")) ?>
```

Этот код читает файл с изображением и автоматически преобразовывает его на сервере в base64. Однако, это простота этого решения повлечет некоторую дополнительную нагрузку на сервер. Как вариант, можно рассмотреть автоматический пересчет всех картинок и вставку их в CSS-файл, например, раз в 5 минут по необходимости (если файл с изображением изменился). Дополнительно нужно будет озаботиться, чтобы сбросить кэширование для самого CSS-файла, содержащего такие изображения.

Работа в Internet Explorer

Существует три способа обойти отсутствие в IE поддержки `data:URI`. Используя распознавание браузеров (например, с помощью условных комментариев, ведь речь идет только про IE), можно просто отображать внешнее изображение для IE и встроенные изображения для остальных браузеров. Или вы можете применить JavaScript для эмуляции этой поддержки в IE, но эта техника потребует довольно значительного объема JavaScript-кода. О третьем способе пойдет речь в разделе, описывающем `mhtml`-технику.

Вышеприведенный PHP-код позволяет легко вставить base64-аналог изображения (можно расширить этот пример, чтобы, например, распознавать заголовки, отправляемые браузером серверу и только для IE выводить URL для изображения, для остальных же кодировать его в base64):

```
ul {
        list-style: none;
}
ul li {
        margin: 0 0 1px;
        background: url(<?php
                echo base64_encode(file_get_contents("../images/flag.png"))
        ?>) top left no-repeat;
        height: 14px;
        text-indent: 10px;
}
```

Когда сервер анализирует CSS-файл, он автоматически перекодирует бинарный файл изображения в base64 и отправит эти данные внутри CSS-файла. Следующим шагом будет добавление распознавания браузеров для отправки изображения только IE и встроенных изображений всем остальным. Это можно сделать либо внутри CSS-файла с PHP-кодом, либо с помощью условных комментариев, например:

```
<!--[if gte IE 5]>
        <style type="text/css" src="ie.css">
<![endif]-->

<!--[if !(IE)]>
        <style type="text/css" src="main.css">
<![endif]-->
```

В файле `ie.css` должно быть нормальное обращение к картинке, например:

```
ul li {
        margin: 0 0 1px;
        background: url(/images/flag.png) 0 0 no-repeat;
}
...
```

Преимущества и недостатки `data:URI`

Вместе с техникой CSS Sprites (или как ее альтернатива) `data:URI` может существенно уменьшить число HTTP-запросов. Краткий список плюсов данного метода:

- Экономят HTTP-запросы, предотвращают издержки, связанные с большим числом объектов.
- Экономят число параллельных потоков: у браузеров есть ограничение (по спецификации HTTP/1.1, однако, Firefox, Opera и Safari несколько вольно его расценивают, в частности, позволяя настраивать этот параметр или значительно его увеличивая; о настройках браузеров можно прочитать в восьмой главе) на число одновременных соединений (подробнее рассказывается в пятой главе) с одним хостом.
- Упрощают HTTPS-запросы и улучшают производительность при таком типе соединения.

Однако встроенные изображения (только `data:URI`) не поддерживаются в Internet Explorer 5–7. Текстовое base64-представление данных также занимает больше, чем бинарное изображение. В наших тестах base64-данные были на 39–45% больше бинарного аналога, но `gzip`-сжатие позволяет уменьшить разницу до 5–10%. Предварительная оптимизация изображений перед base64-кодированием позволяет уменьшить их размер пропорционально (о сжатии изображений было рассказано во второй главе).

Также существует ряд ограничений на размер встроенных изображений. От браузеров требуется поддерживать только URL длиной до 1024 байтов, в соответствии с вышеупомянутой спецификацией RFC. Однако, браузеры более либеральны к пользователям в том, что они принимают. Например, Opera и Firefox последних версий поддерживают `data:URI` примерно до 50 Кб (для IE8 этот предел составляет 32 Кб). Но все же эта техника подходит больше для небольших по размеру изображений. Краткий список минусов:

- Не поддерживается IE до 7 версии включительно.
- Требуются дополнительные действия для обновления внедренного содержания (перекодировать, еще раз вставить).
- Ограничена длина. Не подходит для вставки больших изображений.
- Изображения, представленные в base64-кодировке, примерно на 33% больше размера их бинарного аналога (на 10% при использовании сжатия).
- Встроенные картинки (не в CSS) не получится закэшировать по определению. Они будут кэшироваться только с HTML-кодом.

Дополнительные соображения по оптимизации

Одним из возможных выходов из сложившейся ситуации будет использование характерных для IE CSS-хаков, чтобы только для него подключить фоновые изображения. В итоге, вышеприведенный пример будет выглядеть примерно так:

```
ul {
        list-style: none;
}
ul li {
        margin: 0 0 1px;
        background: url(<?php
                echo base64_encode(file_get_contents("../images/flag.png "))
                ?>) top left no-repeat;
        height: 14px;
        text-indent: 10px;
}
* html ul li {
        background-image: url(/images/flag.png);
}
*+html ul li {
        background-image: url(/images/flag.png);
}
```

Также возможно кодирование изображений, которые выводятся в base64, автоматически при изменении этих изображений (для этого потребуется простой скрипт, который проверяет, обновились ли соответствующие файлы; если обновились, то перезаписывает их представление в CSS-файле, заодно и меняет хеш-строку для подключения этого файла в HTML, чтобы избежать кэширования).

Для включения небольших графиков прямо в HTML-код прекрасно подойдут условные комментарии, когда для ряда браузеров изображение выводится в base64, а для остальных (например, для IE) подключается через условные комментарии. Например, так:

```
<!--[if !IE]>-->
        <img src="..." alt="График" title="График"/>
<!--<![endif]-->
<!--[if IE]>
        <img src="chart.png" alt="График" title="График"/>
<![endif]-->
```

Если использовать связку относительное позиционирование родителя — абсолютное позиционирование дочернего элемента, то IE будет просто выводить картинку из внешнего файла поверх непонятного (для него) объекта.

Кроссбраузерное использование `data:URI`

IE (до версии 7 включительно) не поддерживает протокол `data:URI`, а вместе с ним base64-кодирование внешних файлов и включение их прямо в тело необходимого документа (будь то HTML или CSS/JavaScript-файл). Однако если рассмотреть использование протокола `mhtml` (который, конечно же, аккуратно поддерживается только в IE), многое становится более ясным, и base64-кодирование удается применять в полной мере.

О, этот странный Microsoft!

В IE существует альтернативное решение для вставки изображений прямо в текстовый документ в виде mhtml-включений. Давайте остановимся на практической реализации для CSS-файлов (в данном случае это файл `main.css`):

```
/*
Content-Type: multipart/related; boundary="_"

--_
Content-Location: 1
Content-Transfer-Encoding: base64

iVBOR..
*/
```

Далее в CSS-файле нужно лишь вызвать эту картинку следующим образом:

```
ul li {
        background-image:url(mhtml:http://site.ru/main.css?20080531!1);
}
```

Здесь в адресе картинки идет протокол `mhtml:` (который поддерживается исключительно в IE, но это не так важно), далее **полный URL до CSS-файла** (который содержит эту картинку), в данном случае этот URL еще и содержит GET-параметр для соответствующего кэширования. И небольшая тонкость применения данного формата: необходимо в URL использовать ту же строку, что и в HTML-файле, в котором подключается данный CSS, иначе IE запросит CSS-файл дважды: первый раз как таблицу стилей, второй раз — как хранилище картинки. Далее после восклицательного знака (`!`) идет тот идентификатор, который мы назначили картинке в `Content-Location`. И все.

Объединяем несовместимое

С одной стороны, у нас схема `data:URI`, которая поддержана W3C и распознается всеми браузерами, кроме IE. С другой стороны, у нас IE, который понимает `mhtml` и с которым работают 70% наших пользователей. Мы объединим эти два решения, благо, они оба используют base64-представление картинок.

Задача первая: объединить оба назначения стилевых правил, чтобы они не конфликтовали друг с другом. Решается это очень просто с помощью либо отдельного CSS-файла для IE (через условные комментарии), либо CSS-хаков (последнее предпочтительнее, ибо позволяет загружать всего 1 CSS-файл). В итоге, в CSS-файле мы имеем примерно следующее:

```
/*
Content-Type: multipart/related; boundary="_"

--_
Content-Location: 1
Content-Transfer-Encoding: base64

iVBOR..
*/

ul li {
        background: #fff url(...) 0 0 no-repeat;
}
* html ul li {
        background-image: url(mhtml:http://site.ru/main.css?20081010!1);
}
*+html ul li {
        background-image: url(mhtml:http://site.ru/main.css?20081010!1);
}
```

Данная конструкция позволяет вывести фоновое изображение в base64-кодировке **для всех** (ну или 99,9%) браузеров. Почему в конце содержатся 2 разных CSS-селектора с одним объявлением? Первое предназначено для IE6 и предыдущих версий, второе — для IE7. Объединить через запятую их нельзя.

Панацея или ящик Пандоры?

Первый же скептический вопрос, который каждый читатель должен на этом месте задать: как можно эти картинки выводить только один раз в CSS-файле (можно заметить, что base64-строка фигурирует там дважды)? Ответ: никак. Однако выход есть.

Это `gzip`-сжатие. Если очень коротко, то при любом архивировании создается некоторый словарь наиболее часто используемых строк, которые заменяются на однобайтовые представления. При восстановлении данных из архива происходит обратная процедура, когда значения слов словаря подставляются вместо их наименований. В нашем случае у нас для одной картинки используется **одна и та же base64-строка**. Это означает, что при архивировании она сожмется удивительно эффективно.

Стоит также заметить, что при таком подходе результирующий сжатый файл занимает даже меньше, чем при назначении с помощью CSS-хаков для IE внешних изображений для загрузки (ведь используется меньше различных конструкций).

Валидность

Получившийся таким образом CSS-файл абсолютно валиден (так как все `mhtml`-вставки происходят в комментариях). Каждое CSS-объявление валидно с точки зрения CSS 2.1, и те небольшие приемы, которые позволяют эффективно сжать данные для всех браузеров, не отражаются на восприятии ими файла. Такой подход будет уместным только для CSS. Для HTML ситуация совсем другая (о ней немного подробнее ниже). HTML-документ при таком подходе оказывается невалидным, что выливается во множественные проблемы верстки, и это не та же цена, которую можно заплатить за некоторый не очень явный рост производительности.

Некоторые итоги

- Картинки в CSS можно вставлять с помощью `data:URI`.
- Для IE можно использовать `mhtml`, полностью дублирующий эту функциональность.
- Для корректного применения стилей нужны CSS-хаки либо разделение CSS-файлов.
- Используем `gzip` для CSS-файлов для устранения последствий множественного использования base64-строки.

Использовать base64-кодирование на ваших страницах можно уже прямо сегодня. Очень важно при этом понимать возможные последствия и, по возможности, обойтись малой кровью. Однако с выходом IE8 и массовым его проникновением ситуация кардинально изменится (ведь он поддерживает `data:URI` и для него описанные обходные приемы уже теряют свою актуальность).

Включение музыки (base64)

Летом 2008 года весь мир облетела страница, содержащая реализацию первого уровня Super Mario Brothers. В нее, в общем, можно играть, хотя упущены многие ключевые аспекты (нет грибов, нет флага, нет повышающих очков и т.д.). Однако, это, на самом деле, не самый интересный аспект этой разработки.

Наверное, больше всего интереса в этом веб-приложении представляет не сама игровая механика, а тот факт, что все собрано в одном-единственном файле — игровом скрипте, который включает все игровые спрайты и всю музыку. Это весьма любопытный образец использования технологий. Удивительно, но способ подключения музыки в этом приложении, пожалуй, наиболее простой для понимания. Приложение использует схему `data:URI`, которая кодирует музыкальные MIDI-файлы Mario в base64-виде.

Результат очень просто найти в исходном файле:

```
aSounds = [
        // очень маленькая и простая тема Mario. Написал Mike Martel.
        "data:audio/mid; base64,TVRoZAAAAAYAAQAEAMBNVH...",
        // игра закончилась. Написал John N. Engelmann.
        "data:audio/mid; base64,TVRoZAAAAAYAAQADAHhNVH..."
],
```

`data:URI` работает следующим образом: все содержимое выбранного файла кодируется в виде одной строки в исходном файле. В результате мы получаем 1 файл вместо трех:

одного исходного и двух внешних. В данном случае вся закодированная строка отправляется элементу <embed/>, который и отвечает за проигрывание midi-файла. Можно обнаружить конечный результат этих махинаций в следующем виде:

```
playMusic = function(iSoundID, bLoop) {
        if (!bMusic)
                return;
        var oEmbed = dc("embed");

        oEmbed.src = aSounds[iSoundID];
        oEmbed.id = "sound_" + iSoundID;

        if (bLoop)
                oEmbed.setAttribute("loop", "true");
        oEmbed.setAttribute("autostart", "true");

        oEmbed.style.position = "absolute";
        oEmbed.style.left = -1000;

        appChild(document.body, oEmbed);
},
```

Вышеприведенный код просто создает элемент `embed`, устанавливает для него автозапуск проигрывания музыки и в качестве источника данных указывает `data:URI`. В результате мы получаем MIDI-файл, который сам начинает проигрываться (предположительно, тот же результат мог быть получен при использовании другого универсального музыкального файлового формата — WAV).

Файлы в формате `data:URI` могут проигрываться всеми браузерами, за исключением Internet Explorer. Поэтому пользователи Internet Explorer просто не получает музыкального сопровождения для игры (хотя, можно было бы только для IE подгружать его динамически, создавая тот же самый `embed` просто со ссылкой на внешний файл).

4.5. CSS Sprites и `data:URI`

Давайте проведем сравнение двух освещенных выше методов для кардинального уменьшения числа запрашиваемых файлов с сервера: CSS Sprites и `data:URI`.

Проблемы при верстке

С какими проблемами сталкивается верстальщик, когда использует CSS Sprites? Это, в первую очередь, проблемы изменения каждой конкретной картинки в общем массиве. Для этого нужно открыть ресурсную картинку, найти в ней область, соответствующую данному небольшому изображению (которое меняется), и заменить ее, не потеряв палитру при всех изменениях. Также при изменении расположения картинок в ресурсном файле (например, перераспределили свободное место в связи с очередными дизайнерскими изменениями) нужно заново пересчитать все координаты и внести соответствующие изменения в CSS-файл.

При небольшом количестве спрайтов или их равномерном распределении (например, иконок для пунктов меню) это будет не очень сложно. Но если ресурсный файл представляет собой набор картинок разных размеров, например, как у Google? Тогда

любое изменение может превратиться в ночной кошмар. В любом случае на использование спрайтов тратится дополнительное время при разработке сайтов.

Рис. 22. Пример CSS Sprites со страницы поиска Google. Источник: www.google.com

Также у IE возникают проблемы с позиционированием полупрозрачных PNG-картинок (которые нужно вставлять через `AlpaImageLoader`). Таким образом, больше одной такой картинки в спрайт не добавить (в левый верхний угол). Это можно обойти при помощи IE-фильтра `crop` или абсолютного позиционирования и дополнительной разметки (когда в контейнер с относительным позиционированием вставляем изображение с абсолютным позиционированием и накладываем поверх этого изображения все остальное содержимое контейнера — оно будет располагаться в нем в обычном порядке).

Проблемы при загрузке

Казалось бы, CSS Sprites призваны уменьшить задержку при загрузке страницы, однако, на практике так происходит только при правильном подходе. Обычно в ресурсную картинку объединяется все подряд, картинка многократно увеличивается, а визуальная задержка при загрузке сайта только растет: пользователь может ждать 1 большую картинку дольше, чем половину входящих в нее маленьких, а последние обеспечат ему «почти» загрузку сайта.

Если спрайтов у нас немного (1–2 картинки), то общее время загрузки сайта, скорее всего, **возрастет**. Это связано с тем, что браузер не сможет открыть, как минимум, 4–8 дополнительных соединений (к хостам, где расположена статика) и загрузить все исходные картинки параллельно, а не последовательно. При небольшой сетевой задержке это может оказать решающее воздействие.

Проблемы при использовании

Даже если положиться на то, что спрайты поддерживаются (почти) всеми браузерами на данный момент, все равно остается достаточно много вопросов, которые они не только не решают, а скорее, сами создают. Во-первых, это проблемы при использовании иконок для списка. В таком случае необходимо располагать маленькие картинки «лесенкой», но, в общем случае, это увеличивает размер получившейся картинки на 20–30%.

Во-вторых, при использовании спрайтов, когда пытаются обойти обрисованную проблему с «выползанием» фона для ненужных элементов, в структуре страницы появляются несемантические элементы (дополнительная разметка), которые, по сути, заменяют тег `img`, только через фоновое изображение. Такое засорение структуры страницы вредно отражается на всех уровнях разработки и использования сайта.

В-третьих, если логотип склеен с другими картинками (как у вышеуказанного примера), то становится невозможным его использование как обычной картинки (например, в версии для печати или на внешних ресурсах).

Шаг за шагом

С развитием техник `data:URI` наиболее логичным выходом из сложившейся ситуации будет следующий характер разработки.

- Верстальщик создает рабочую версию макета сайта, не прибегая к помощи CSS Sprites (на каждый элемент одна фоновая картинки).
- Веб-технолог внедряет кодирование картинок в base64 (`+mhtml`) в CSS-файл(ы) на этапе их загрузки на боевой сайт, создавая автоматизированное решение. На этом этапе могут использоваться и CSS Sprites, однако, их внедрять сложнее из-за перерасчета позиционирования фона и обновления соответствующих стилевых правил.
- Веб-программист обеспечивает для ряда «старых» браузеров загрузку версии без использования `data:URI`.

Чем это хорошо? Верстальщик не думает лишний раз, что и как ему расположить и нарезать: эти операции уже включены в процесс публикации сайта, автоматизированы и максимально адаптированы под пользователей.

Чем это плохо? В общем случае, загрузка страницы не ускорится, а даже может замедлиться, потому что фоновые картинки (включенные через `data:URI`) будут грузиться в один поток, а не в несколько, как при обычном использовании спрайтов. Если фоновых картинок достаточно много (несколько десятков килобайтов), то это окажется существенным. При небольшом их объеме (до 10 Кб) будет заметно явное ускорение.

Как распределить загрузку фоновых картинок между предзагрузкой и пост-загрузкой (фактически, ускорив первую за счет замедления второй) рассказывается чуть ниже.

Выносим CSS-файлы в пост-загрузку

При использовании `data:URI` итоговый CSS-файл занимает довольно большой объем (фактически, равный 110–120% от размера всех картинок и набор базовых CSS-правил). И это в виде архива. Если файл не заархивирован, то его дополнительный размер увеличивается многократно (в 2,5–3 раза относительно размера всех фоновых изображений), но это не так существенно, ибо пользователей с отключенным сжатием для CSS-файлов сейчас единицы (обычно доли процента).

Для решения этой проблемы, во-первых, нам нужно разделить весь массив CSS-правил на относящиеся к фоновым изображениям и не относящиеся. Во-вторых, сообщить браузерам, что они могут отобразить страницу без первого массива правил (ведь если в нем содержатся только фоновые изображения, то они могут и подождать чуть-чуть).

Фактически, используя такой подход, мы создаем **другой** контейнер для фоновых изображений (не ресурсное изображение, а CSS-файл), который удобнее использовать в большинстве случаев. Мы объединяем все фоновые картинки не через CSS Sprites, а через `data:URI`, и можем загрузить их все одним файлом (в котором каждая картинка будет храниться полностью независимо). При этом избегаем любых проблем с

позиционированием фона (все ранее заявленные проблемы с (полу)прозрачными картинками для прошлых версий IE сохраняются, однако, их решение также остается прежним).

Теоретическое решение

Все гениальное просто, поэтому мы можем загружать в самом начале страницы достаточно небольшой CSS-файл (без фоновых изображений, только базовые стили, чтобы только отобразить страницу корректно), потом по событию загрузки страницы (подробнее о методе можно прочитать в седьмой главе) через JavaScript подгрузить в один или несколько потоков динамические файлы стилей.

Тут есть и возможные минусы: после загрузки каждого дополнительного CSS-файла будет происходить перерисовка страницы. Однако, если таких файлов всего 1 или 2, то отображение страницы произойдет значительно быстрее.

Почему мы не может распараллелить загрузку файлов стилей в самом начале документа? Потому что два файла будут загружаться медленнее, чем один (файлы загружаются последовательно в большинстве браузеров, поэтому задержки на установление соединений будут складываться). К тому же мы ратуем за максимально быстрое отображение страницы в браузере пользователя (завершение первой стадии загрузки), поэтому исходный объем загружаемого CSS должен быть минимальным (можно также рассмотреть варианты по включению его в сам HTML).

На практике

На практике все оказалось не сильно сложнее. Мы загружаем в `head` страницы (до вызовов любых внешних файлов) наш «легкий» CSS:

```
<link href="light-light.css" rel="stylesheet" type="text/css" media="all"/>
```

А затем добавляем в комбинированный обработчик `window.onload` (подробнее о нем рассказывается в седьмой главе) создание нового файла стилей, который дополняет уже загрузившуюся страницу фоновыми изображениями:

```
function combinedWindowOnload() {
  load_dynamic_css("background-images.css");
  ...
}
```

В результате мы имеем максимально быстрое отображение страницы, а затем стадию пост-загрузки, которая вытянет с сервера все дополнительные картинки (тут уже сам браузер постарается), стилевые правила и скрипты.

А доступность?

Внимательные читатели уже заготовили вопрос: а что, если у пользователя отключен JavaScript? Тут всё должно быть просто: мы добавляем соответствующий `<noscript>` для поддержки таких пользователей. С маленьким нюансом: `<noscript>` не может находиться в `<head>`, а `<link>` не может находиться в `<body>`. Если мы соблюдаем стандарты (все же иногда лучше довериться профессионалам и не ставить браузеры в неудобное положение,

когда они встретятся с очередным отклонением от спецификации), то стоит искать обходные пути.

После небольших экспериментов было выделено следующее изящное решение, обеспечивающее работу схемы во всех браузерах (замечание: после многочисленных экспериментов было решено остановиться на HTML-комментариях — они оказались наилучшим способом запретить загрузку указанного CSS-файла):

```
<script type="text/javascript">
/* если мы сможем создать динамический файл стилей */
        if (document.getElementsByTagName) {
/* то добавляем в загрузку облегченную версию */
                document.write('\x3clink href="light-light.css"
                        rel="stylesheet" type="text/css" media="all"/>');
/* после этого начинаем HTML-комментарий */
                document.write('\x3c--');
}
</script>
<link href="full.css" rel="stylesheet" type="text/css" media="all"/>
<!--[if IE]><![endif]-->
```

В результате браузер с включенным JavaScript запишет начало комментария, а закроет его только после `<link>` (комментарии не могут быть вложенными). При выключенном JavaScript `<script>` не отработает, `<link>` обработается и добавится в очередь загрузки, а последний комментарий будет просто комментарием.

Делаем решение кроссбраузерным

В ходе тестирования в Internet Explorer обнаружилось, что если добавлять файл стилей сразу параллельно со скриптами (в функции, которая для него срабатывает по `onreadystatechange`), то IE «морозит» первоначальную отрисовку страницы (т.е. показывает белый экран), пока не получит «свеженький» файл стилей. Для того чтобы Internet Explorer не занимался «замораживанием», нужно вставить фиктивную задержку следующим образом:

```
setTimeout('load_dynamic_css("background-images.css")',0);
```

В Safari же логика отображения страницы в зависимости от загружаемых файлов отличается ото всех браузеров. Если в двух словах, то можно жестко определить начальный набор файлов, необходимых для отображения страницы на экране (HTML/CSS/JavaScript). А можно начать загружать все файлы в порядке приоритетности (и выполняя все их зависимости) и проверять время от времени, можно ли уже отобразить страницу (выполняя все вычисления в фоновом режиме без обновления экрана).

У Safari второй подход, поэтому ничего лучше выноса загрузки динамического CSS-файла с фоновыми картинками после срабатывания `window.onload` для этого браузера пока не существует. Зато первоначальная картинка в браузере появляется значительно быстрее (при большом объеме фоновых изображений).

Итак, давайте объявим функцию для создания динамического файла стилей:

```
/*
Объявляем функцию по динамической загрузке стилей и скриптов.
*/
```

```
function load_dynamic_css (src){
        var node = document.createElement("link");
        node = document.getElementsByTagName("head")[0].appendChild(node);
        node.setAttribute("rel", "stylesheet");
        node.setAttribute("media", "all");
        node.setAttribute("type", "text/css");
        node.setAttribute("href", src);
}
...
/*
Далее определяем для window обработчик по событию onload.
Используем условную компиляцию для выделения IE
*/
window[/*@cc_on !@*/0 ? 'attachEvent' : 'addEventListener']
        (/*@cc_on 'on' + @*/'load',
        function(){
                setTimeout('load_dynamic_css("background-images.css")',0);
        }
,false);
```

Выигрыш

При наличии у вас большого количества маленьких декоративных фоновых изображений, которые к тому же могут повторяться по различным направлениям, может быть очень удобно объединить их все в один файл и загружать его после отображения страницы на экране.

Описанная техника (кроссбраузерный `data:URL` плюс динамическая загрузка файлов стилей) позволяет добиться всех преимуществ технологии CSS Sprites, не затягивая загрузку страницы. При этом обладает очевидными преимуществами: не нужно лепить все картинки в один файл (их можно объединять на этапе публикации, а не на этапе разработки), можно работать с каждой совершенно отдельно, что позволяет добиться большей семантичности кода и большего удобства использования сайтов. К тому же это несколько сократит CSS-код за счет уничтожения необходимости применения `background-position`.

Таким образом, `data:URI` (в смысле влияния на скорость загрузки) равносильны CSS Sprites (или даже предпочтительнее последней, если учесть, что для повторяющихся и полупрозрачных CSS Sprites придется создавать отдельные ресурсные файлы). В смысле же простоты внедрения и разработки они отличаются в выгодную сторону: нужно лишь настроить использование общей схемы **один раз** в шаблонах (с учетом динамической загрузки JavaScript-файлов, которая описана в седьмой главе, это все равно придется делать) и при публикации изменения применять base64-кодирование к фоновым изображениям.

4.6. Методы экстремальной оптимизации

Чем больше число внешних ресурсов, к которым браузер обращается при загрузке, тем больше время требуется для отображения страницы. Как правило, веб-страницы обращаются ко многим внешним CSS и файлам JavaScript. Все файлы стилей и скриптов можно объединить, чтобы уменьшить число внешних ресурсов этих типов до двух. Это, естественно, поможет серьезно сократить время загрузки страницы.

Объединение JavaScript и CSS в одном файле

Однако есть способ объединения CSS с JavaScript и сведения количество загрузок к одной. Техника основана на том, как CSS и анализатор JavaScript ведут себя в IE и Firefox.

- Когда анализатор CSS сталкивается с символом комментария HTML (`<!--`) в содержании CSS, символ игнорируется.
- Когда анализатор JavaScript сталкивается с символом комментария HTML (`<!--`) в содержании JavaScript, символ рассматривают как подобный комментарию линии (`//`), и, следовательно, остальная часть строки после символа комментария HTML игнорируется.

Рассмотрим на примере

```
<!-- /*
function t(){}
<!-- */
<!-- body { background-color: white; }
```

Когда анализатор CSS будет разбирать вышеупомянутый код, символы комментария HTML будут пропущены, и код станет эквивалентным следующему примеру:

```
/*
function t(){}
*/
body { background-color: white; }
```

Анализатор CSS видит только CSS-код, а код скрипта закомментирован (`/* ... */`).

Когда анализатор JavaScript станет разбирать код, символы комментария HTML будут интерпретированы в комментарии строки (//), и, следовательно, код станет таким:

```
// /*
function t(){}
// */
// body { background-color: white; }
```

Анализатор JavaScript видит только код скрипта, а все остальное закомментировано. Чтобы ссылаться на этот ресурс, можно использовать теги `<script>` и `<link>` на странице. Например:

```
<link type="text/css" rel="stylesheet" href="test.jscss" />
<script type="text/javascript" src="test.jscss"></script>
```

Заметим, что эти два тега ссылаются на один тот же ресурс и, следовательно, он загрузится всего один раз и будет интерпретирован и как стили, и как скрипты.

Есть еще одна вещь, о которой стоит позаботиться — `Content-Type` ответа. Его необходимо выставлять в `*/*`, чтобы дать подтверждение Firefox: содержание может быть обработано как что-либо подходящее (как стили или как скрипты).

Указанное решение не работает в Safari (1–5% пользователей), однако, конкретно для этого браузера (определив его через `User-Agent`) уже можно вставить загрузку еще одного файла.

Объединение HTML, CSS и JavaScript в одном файле

Чтобы избежать дополнительных запросов со стороны браузера, можно включить непосредственно стилей и(ли) скриптов в сам HTML-документ.

Здесь стоит остановиться на следующем моменте: если размер CSS- (или JavaScript-) файла больше, чем 20% (и при этом больше 5 Кб в сжатом виде), лучше вынести его как отдельный компонент. Это позволит настроить его кэширование для постоянных пользователей вашего сайта.

Рассматривать включение всех ресурсов в исходную HTML-страницу стоит только в том случае, если достаточно большой процент посетителей (больше 90%) пришли на нее в первый и (возможно) в последний раз. Тогда эта технология будет замечательно работать: кэширование ничего практически не даст, а дополнительные запросы к серверу замедлят загрузку страницы для новых, не знакомых со спецификой сайта посетителей (что может быть решающим фактором для их окончательного ухода).

Во всех остальных случаях — когда можно выделить достаточно большие ресурсные файлы или когда достаточное количество пользователей приходят не в первый раз — такой подход неприменим.

Как рабочий пример можно привести заглавные страницы Яндекса и Google — на них вызывается минимум внешних ресурсов, а стилевые правила включены в саму страницу.

Внутри или снаружи?

Давайте в качестве заключения рассмотрим следующий вопрос: стоит ли вообще подключать JavaScript- и CSS-файлы или можно включить весь их код непосредственно в код страницы?

Использование подключаемых файлов на практике обычно дает более быстрые страницы, т.к. браузеры кэшируют файлы скриптов и CSS. JavaScript- и CSS-код, который находится в HTML, загружается каждый раз при загрузке самого HTML-документа. Это уменьшает количество необходимых HTTP-запросов, но увеличивает объем HTML. С другой стороны, если скрипты и таблицы стилей находятся в отдельных файлах, закэшированных браузером, размер HTML уменьшается, не увеличивая при этом количество HTTP-запросов (при повторных посещениях).

В таком случае ключевым фактором является частота, с которой кэшируются внешние JavaScript- и CSS-файлы относительно количества запросов самого HTML-документа. И хотя этот фактор очень сложно посчитать, его можно приблизительно оценить различными способами. Если пользователи во время одного посещения загружают страницу несколько раз или загружают похожие страницы, которые используют один и тот же код, — это именно тот случай, когда мы можем получить все преимущества от вынесения кода в отдельные файлы.

Многие сайты только наполовину удовлетворяют этим условиям. Для таких случаев, в целом, лучшим решением будет создание внешних файлов скриптов и таблиц стилей. Единственное исключение, которое можно здесь привести (когда прямое добавление кода дает большое преимущество) — это использование его на главных страницах, таких, как главная страница Яндекса (http://www.yandex.ru/), Рамблера (http://www.rambler.ru/) или

Google (http://www.google.ru/). Для страниц, которые загружаются всего несколько (обычно — один) раз за весь сеанс, выгодней включать скрипты и таблицы стилей прямо в HTML-документ, чтобы выиграть в скорости загрузки.

Для таких главных страниц, которые открываются первыми в ряду других с этого же сайта, существует возможность уменьшить число HTTP-запросов еще и следующим образом. Мы можем включить JavaScript и CSS в код самой страницы, однако, после ее полной загрузки динамически подгружать внешние файлы стилей и скриптов для последующего использования (на стадии пост-загрузки). При этом следующие страницы будут использовать уже закэшированные файлы.

5.1. Обходим ограничения браузера на число соединений

Активное (*англ. keep-alive*) соединение стало настоящим прорывом в спецификации HTTP 1.1: оно позволяло использовать уже установленный канал для повторной передачи информации от клиента к серверу и обратно (в HTTP 1.0 соединение закрывалось сразу же после передачи информации от сервера, что добавляло задержки, связанные с трехступенчатой передачей пакетов). В том случае, если проблема свободных ресурсов стоит довольно остро, можно рассмотреть выставление небольшого тайм-аута для таких соединений (5–10 секунд).

Однако HTTP 1.1 добавил веб-разработчикам головной боли по другому поводу. Давайте будем разбираться, как нам устранить и эту проблему.

Издержки на доставку объектов

Средняя веб-страница содержит более 50 объектов, и издержки на число объектов доминируют над всеми остальными задержками при загрузке большинства веб-страниц. Браузеры, следуя рекомендациям спецификации `HTTP 1.1`, обычно устанавливают не более 2 одновременных соединений с одним хостом. При увеличении числа HTTP-запросов, требуемых для отображения страницы, с 3 до 23 — время, затрачиваемое именно на «чистую» загрузку объектов, от общего времени загрузки падает с 50% до всего 14%.

Если число объектов на странице превышает 4, то издержки на ожидание доступных потоков и разбор чанков для присланных объектов превалируют над общим временем загрузки страницы (от 80% до 86% для 20 и 23+ объектов, соответственно) по сравнению со временем, которое уходит на действительную загрузку данных. Время инициализации плюс время ожидания, вызванное ограничением на параллельные соединения, занимают 50–86% от общего времени загрузки страницы.

При увеличении числа подключаемых объектов сверх 10 время, затрачиваемое на инициализацию соединения, возрастает до 80% и более от общего времени, уходящего на получение объектов. Стоит отметить, что можно существенно уменьшить издержки на доставку большого числа объектов (более чем 12 на страницу) включением для сервера `keep-alive` режима и распределением запросов по нескольким хостам.

Ограничения спецификации HTTP/1.1

Браузеры создавались в ту эпоху, когда громадное множество пользователей пользовались коммутируемым доступом с невысокой пропускной способностью канала, поэтому тогда было важно ограничить пользователей небольшим числом одновременных соединений. Накладные издержки на переключение между множеством соединений при коммутируемом доступе создавали большие сложности для обработки и загрузки каждого отдельного запроса. К тому же в ту эпоху веб- и прокси-серверы были недостаточно мощными, чтобы поддерживать множество соединений, поэтому такое жесткое ограничение числа одновременных соединений у браузера существенно снижало риск падения сетевой инфраструктуры в целом.

Спецификация HTTP, приблизительно 1999 года, рекомендует, чтобы браузеры и серверы ограничивали число параллельных запросов к одному хосту — двумя. Эта спецификация была написана задолго до существенного расширения каналов загрузки и была рассчитана на соединения с маленькой скоростью загрузки. Большинство браузеров поддерживают это ограничение на число потоков в спецификации, хотя переход на HTTP 1.0 увеличивает число параллельных загрузок до 4. Поэтому большинство браузеров серьезно ограничено этим параметром, если им приходится загружать большое число объектов с одного хоста (по словам Алекса Могилевского, в IE8 это число будет равно 6 из-за определенных издержек на установление нового соединения). Существует два основных пути для обхода этого ограничения (не считая, конечно, тонкой настройки используемого клиентами браузера, о которой рассказывается в восьмой главе):

- Отдавать ваши объекты с нескольких серверов
- Создать несколько поддоменов для нескольких хостов

Чтобы найти подходящий баланс, IE до версии 7 включительно ограничивают пользователей всего восемью одновременными соединениями или двумя соединениями на хост для протокола HTTP 1.1. HTTP 1.0 немного отличается в этом плане, но это уже совсем другая история, потому что все выгоды от постоянных соединений доступны только, если мы будем использовать HTTP 1.1 (или уже так делаем).

Времена меняются

Естественно, в реальном мире все эти утилитарные решения имеют особенность устаревать, вместе со своим временем и средой. Сегодня у большинства пользователей широкополосный доступ в Интернет, поэтому наиболее узким местом является уже не клиентская сторона (клиентская сторона была, есть и будет наиболее узким местом в производительности наших веб-приложений, просто нужно понимать, как именно можно ее оптимизировать в каждом конкретном случае), а пропускная способность каналов в большинстве случаев.

Обычно задержки при получении отдельных объектов значительно больше, чем время на установление нового соединения и отправку запроса. Увеличивая число одновременных соединений, мы можем распараллелить это место и гораздо быстрее пробиться через множество объектов, которые находятся в списке ожидаемых к загрузке, что приведет, в итоге, к увеличению ощущаемой скорости загрузки у пользователя «до скорости молнии».

К несчастью, полагаться на то, что пользователи сами будут изменять настройки своего браузера (а о том, как это можно сделать, пойдет речь в восьмой главе) — это будет не лучшей стратегией по оптимизации. Так что же делать разработчику, чтобы добиться того же эффекта со своей стороны?

«Режем» соединения

Большинство сайтов обладают всего одним хостом, поэтому все запросы вынуждены бороться за 2 доступных соединения к этому хосту. Одним из наиболее эффективных методов для увеличения числа параллельных потоков будет распределение содержания по нескольким хостам. Это не так сложно сделать, потому что браузеры обращают внимание только на название хоста, а не на IP-адрес. Таким образом, к каждому из хостов images1.yoursite.ru и images2.yoursite.ru браузеры смогут установить по два соединения.

Небольшие изображения (например, товаров в магазине или фотографий к новостям), по умолчанию, загружаются с родительского хоста, поэтому они также вынуждены использовать те же два доступных соединения. Ниже представлена примерная диаграмма загрузки объектов на странице.

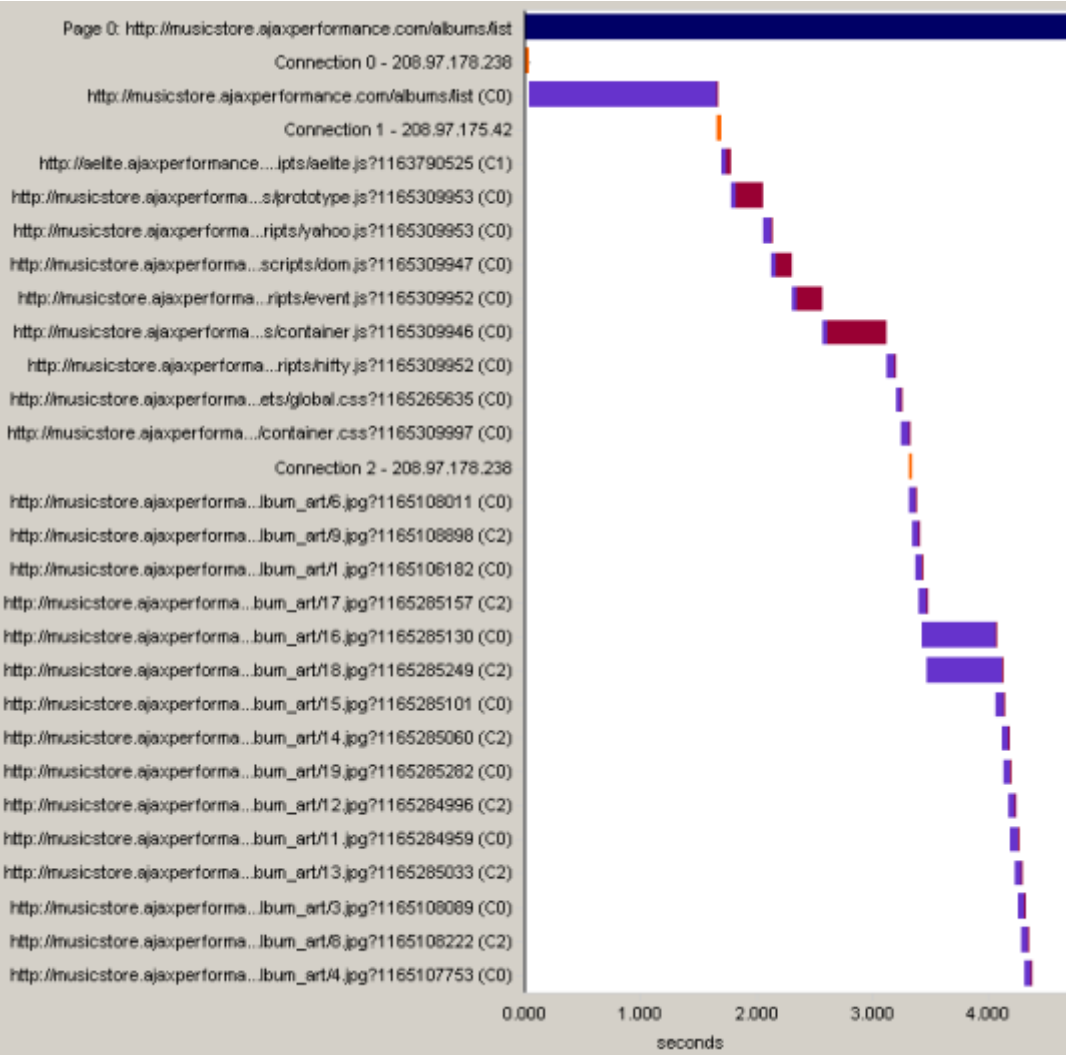


Рис. 23. Загрузка изображений при двух соединениях. Источник: www.ajaxperformance.com

На этом графике хорошо видно, что для `musicstore.ajaxperformance.com` открыто только 2 соединения (данная диаграмма является модельной и справедлива только для IE, во всех остальных браузерах, по умолчанию, открывается большие соединений): C0 и C2. Мы используем протокол `HTTP 1.1`, поэтому нам не нужно открывать отдельное соединение для каждой картинки, но мы по-прежнему теряем кучу времени на обслуживание индивидуальных запросов к объектам. Время на установление соединения (время до получения первого байта, голубая полоска на диаграмме) явно доминирует над временем загрузки данных, которое не так велико (красная полоска на диаграмме).

Вы можете, естественно, настроить несколько серверов для обслуживания выдачи картинок или других объектов, чтобы увеличить число параллельных загрузок. Например:

```
images1.yoursite.ru
images2.yoursite.ru
```

`images3.yoursite.ru`

Однако каждый из этих поддоменов не обязан находиться на отдельном сервере.

Лучше, больше, быстрее

Чтобы улучшить производительность, можно создать CNAME-записи в DNS-таблице для `images1.yoursite.ru`, `images2.yoursite.ru` и `images3.yoursite.ru`, каждая из которых указывает обратно на основной хост.

Стоит обратить внимание, что при проектировании масштабируемых приложений, которые будут распределять объекты по различным хостам, нужно использовать хеш-функцию. Она установит однозначное соответствие между названием изображения и хостом, с которого оно должно загружаться. В качестве простых примеров можно привести остаток от деления md5-суммы или длины строки адреса изображения на число хостов. Также можно рассмотреть использование контрольной суммы CRC32, которую проще посчитать на JavaScript.

При первой загрузке производительность будет значительно лучше. Как можно видеть из нижеприведенного графика, сейчас используется уже 6 соединений для загрузки наших картинок.

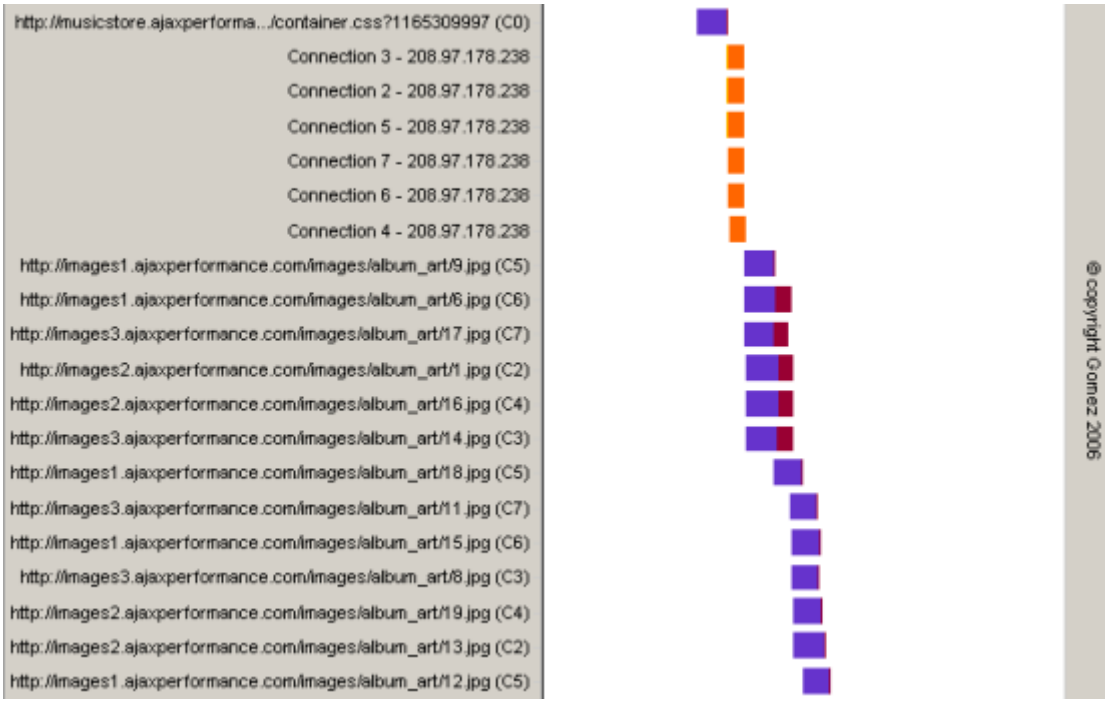


Рис. 24. Загрузка при шести соединениях. Источник: www.ajaxperformance.com

Реальный выигрыш

Время загрузки страницы при использовании уменьшилось **больше чем на 40%**. И эта техника будет работать во всех случаях, когда у вас большой пул запросов к объектам, которые расположены на одном сервере.

Существует масса примеров применения этого метода в реальных AJAX-приложениях. Чтобы утилизировать параллельность соединений, на Google Maps (http://maps.google.com/) картинки поставляются с нескольких хостов, начиная с `mt0.google.com` и заканчивая `mt3.google.com`. На Virtual Earth (http://local.live.com/) также используется эта техника.

Этот подход можно также применить, чтобы изолировать отдельные части вашего приложения друг от друга. Если некоторые его элементы требуют доступа к базе данных и их загрузка задерживается больше, чем для статичных объектов, стоит устранить их из числа тех двух соединений, которые будут использоваться для загрузки картинок на вашем сайте, например, разместив их на поддомене.

В данном случае, наверное, наиболее практичным решением будет размещение всей статики (кроме, пожалуй, CSS- и JavaScript-файлов, которые влияют на стадию предзагрузки — чтобы максимально избежать на этой стадии задержек на дополнительные DNS-запросы) на отдельном домене, например, `static.example.com`, а загрузка HTML-страниц, которые требовательны к базе, будет вестись с основного хоста. При этом `static.example.com` может иметь даже другой IP-адрес и обслуживаться любым «легким» сервером. Этот прием, может быть, и не сильно ускорит загрузку нашей страницы, но определенно улучшит ощущаемую производительность, позволяя пользователю загружать все статические файлы без дополнительных задержек.

Подводим итоги

Сейчас средняя веб-страница состоит более чем из 50 объектов (для Рунета, по статистическим данным webo.in, ситуация весьма похожа: число объектов колеблется в пределах 40–50), поэтому минимизация издержек на доставку объектов является весьма критичной для клиентской производительности. Также можно уменьшить число объектов на странице, если использовать технику CSS Sprites (или `data:URI`) и объединение текстовых файлов на сервере. Так как в данный момент у пользователей достаточно быстрый канал, то можно достигнуть уменьшения времени загрузки до 40–60% (зависит от общего числа объектов). Можно использовать 2 или 3 хоста для обслуживания объектов с одного сервера, чтобы «обмануть» браузеры в их ограничениях на загрузку нескольких объектов параллельно.

При этом нужно помнить, что увеличение одновременных запросов повлечет задействование дополнительных ресурсов со стороны сервера (это может быть, например, как максимальное число открытых соединений или портов, так и дополнительные объемы оперативной памяти). Поэтому данный подход стоит активно испольовать только при наличии «легкого» сервера, который способен одновременно поддерживать тысячи и десятки тысяч открытых соединений без особого ущерба для производительности (например, nginx или 0W).

Стоит коснуться еще одного, весьма интересного, момента в оптимизации времени загрузки путем увеличения числа параллельных потоков. Заключается он в выравнивании и увеличении размера одновременно загружаемых объектов, чтобы максимально использовать имеющиеся соединения. Например, если у вас есть 40 картинок по 5 Кб, то гораздо выгоднее будет отдавать 10 картинок по 20 Кб с двух хостов, чем 20 (по 10 Кб) с 4 хостов или 40 — с 8. Общие задержки в первом случае будут минимальными в силу максимизации эффективной скорости загрузки данных клиенту.

Можно пойти и дальше и загружать, например, 4 картинки по 50 Кб в 4 потока, достигая просто феноменального ускорения. Однако тут вступает в роль психологический фактор: пользователю будет не комфортно, если он будет видеть страницу вообще без картинок все время, пока грузится 50 Кб, и он может просто уйти с сайта.

Стоит подчеркнуть, что данный подход применим и к другим ресурсным (в том числе, и HTML) файлам, однако, стоит помнить о весьма жестких ограничениях браузеров на загрузку CSS- и JavaScript-файлов (как их обойти для CSS-файлов, было описано в четвертой главе, о JavaScript же более детально речь пойдет в седьмой).

5.2. Content Delivery Network и Domain Name System

Сети доставки содержания (*англ. CDN*) часто используются для уменьшения нагрузки на хостинг для веб-приложений и его каналов. В этом случае увеличение производительности достигается за счет распространения всех требуемых ресурсов по сети серверов. Близость к таким веб-серверам незамедлительно оборачивается увеличение скорости загрузки компонентов. Основной упор в таких сетях делают на максимальное уменьшение времени `ping` до сервера, что приводит к впечатляющим результатам с точки зрения производительности.

Некоторые крупные интернет-компании владеют своими сетями CDN, однако, гораздо дешевле использовать уже готовые решения, такие, как Akamai Technologies, Mirror Image Internet либо CDNetwork. Для стартапов или личных веб-сайтов стоимость услуг сетей CDN может оказаться непомерно высокой, но по мере того, как аудитория увеличивается и становится все более удаленной от вас, CDN просто необходимы для достижения быстрого отклика веб-страницы.

Ценовая структура Akamai основана на общем весе веб-страниц в Кб и числе пользовательских загрузок. Оптимизация самих веб-страниц может очень сильно сказаться на общей цене. Предположим, что один из клиентов такого сервиса платит приблизительно $8000 в месяц за домашнюю страницу в 320 Кб. Если бы над сайтом была проведена работа, которая бы уменьшила общий вес страницы на 25%, то ежемесячная оплата для клиента сократилась бы на $2000. В этом примере речь идет всего лишь просто о домашней странице. Уже для нее затраты на разработку окупятся с лихвой!

Подключаем CDN

CDN — это множество веб-серверов, распределенных географически для достижения максимальной скорости отдачи содержания клиенту. Сервер, который непосредственно будет отдавать файлы пользователю, выбирается на основании некоторых показателей. Например, выбирается сервер с наименьшим числом промежуточных запросов (*англ. hop*) до него либо с наименьшим временем отклика.

Использование CDN потребует лишь незначительных изменений (либо вообще таковых не потребует) кода, но повлечет значительное увеличение скорости загрузки самих веб-приложений, потому что на нее сильно влияет и то, насколько далеко пользователь находится от нашего сервера. Размещение файлов на нескольких серверах, разнесенными географически, сделает загрузку сайта быстрее с точки зрения пользователя. Но с чего бы начать?

В качестве первого шага к построению системы с географически распределенным содержанием не стоит пытаться изменить веб-приложение для работы с распределенной архитектурой. В зависимости от приложения, изменение архитектуры может повлечь за собой сложные изменения, такие, как синхронизация состояния сессий или репликацию транзакций баз данных между географически разнесенными серверами.

80–90% времени загрузки страницы уходит на загрузку ее компонентов: картинок, CSS, скриптов, Flash и т.д. Вместо того, чтобы заниматься изменением архитектуры самого приложения, сначала стоит распределить статический контент. Это не только позволяет добиться значительного ускорения загрузки страницы, но также легко реализуется благодаря CDN.

Yahoo! и Google

Yahoo! обеспечивает выдачу YUI (Yahoo! User Interface) библиотек, используя распределенную систему северов по всему миру бесплатно. Это сервис обеспечивает:

1. `gzip`-сжатие (уменьшает размер файлов от 60% до 90%);
2. контроль за кэширующими заголовками;
3. распределенный хостинг файлов, основанный на географическом расположении клиента. Предоставляется на основе передовых компьютерных систем.

Аналогичный сервис сейчас предоставляет и Google для JavaScript-библиотек (в том числе, естественно, для всех дополнений от Google, таких, как автоматическая страница «Ничего не найдено» (ошибка 404), AJAX API для поиска или Google Maps).

Сети доставки содержания задумывались в качестве простого хостинга для картинок и больших (аудио-, видео-) файлов, но сейчас они обрабатывают и JavaScript с CSS. Использование кэширования и системы контроля версий в сочетании с распределением файлов по такой сети может привести к существенному приросту производительности.

Количество DNS-запросов

Система DNS устанавливает соответствие имен хостов их IP-адресам, точно так же, как телефонный справочник позволяет узнать номер человека по его имени. Когда вы набираете www.yahoo.com в адресной строке браузера, преобразователь DNS, к которому обратился браузер, возвращает IP-адрес узла. DNS-запрос имеет свою цену. Обычно требуется 20–120 миллисекунд, чтобы его выполнить и получить ответ (в российских реалиях это время обычно больше). Браузер вынужден ожидать завершения DNS-запроса, т.к. до этого момента он еще не может ничего загружать.

Для повышения быстродействия результаты DNS-запросов кэшируются. Это кэширование может происходить как на специальном сервере интернет-провайдера, так и на компьютере пользователя. Информация DNS сохраняется в системном кэше (в Windows за это отвечает служба «DNS Client Service»). Большинство браузеров имеет свой кэш, не зависящий от системного. Пока браузер хранит DNS-запись в своем кэше, он не обращается к операционной системе для DNS-преобразования.

Internet Explorer, по умолчанию, кэширует результаты DNS-запросов на 30 минут, как указано в переменной реестра `DnsCacheTimeout`. Firefox кэширует DNS-ответы на 1 минуту, что видно из установки `network.dnsCacheExpiration`.

Когда клиентский кэш очищается (как системный, так и у браузера), количество DNS-запросов возрастает до количества уникальных имен хостов на странице. А это включает в себя собственно адрес самой страницы, картинок, скриптов, CSS-, Flash-объектов и т.д. Уменьшение количества уникальных имен хостов уменьшает количество DNS-запросов.

Однако уменьшение количества уникальных хостов потенциально уменьшает количество параллельных загрузок компонентов страницы. В свете этого обстоятельства наилучшим выходом будет распределение загружаемых компонентов между 2–4 (но не более) уникальными хостами. Это является компромиссом между уменьшением количества DNS-запросов и сохранением неплохой параллельности при загрузке компонентов страницы.

5.3. Балансировка на стороне клиента

Балансировка нагрузки повышает надежность веб-сайта путем распределения запросов между несколькими (кластером) серверами, если один из них перегружен или отказал. Существует много методов по обеспечению такого поведения, но все они должны удовлетворять следующим требованиям:

- распределять нагрузку внутри кластера рабочих серверов;
- корректно обрабатывать отказ одного из рабочих серверов;
- весь кластер должен существовать для конечного пользователя как одна-единственная машина.

Round-Robin DNS

Популярным, хотя и очень простым подходом для балансировки запросов является циклический DNS. Он подразумевает создание нескольких записей в таблице DNS для одного домена. Например, мы хотим распределять нагрузку для сайта `www.loadbalancedwebsite.ru`, и у нас есть два сервера с IP адресами 64.13.192.120 и 64.13.192.121, соответственно. Для того чтобы реализовать циклический DNS для распределения запросов, можно просто создать следующие записи в DNS:

```
www.loadbalancedwebsite.ru 64.13.192.120
www.loadbalancedwebsite.ru 64.13.192.121
```

После каждого пользовательского запроса к таблице DNS для `www.loadbalancedwebsite.ru`, запись, стоящая первой, меняется. Ваш браузер будет использовать первую запись, поэтому все запросы будут распределяться случайным образом между этими двумя серверами. К несчастью, ключевым недостатком этого подхода является нарушение второго условия, обозначенного выше, а именно: при отказе одного из серверов сервер DNS все равно будет отправлять на него пользовательские запросы, и половина ваших пользователей окажется за бортом.

Можно, конечно, перенести IP-адрес на соседний сервер, который может нести нагрузку. Однако данная процедура весьма хлопотная, чтобы проводить ее в услових аврального положения.

Балансировка на сервере

Другим популярным подходом для балансировки запросов является создание одного выделенного сервера, который отвечает за распределение запросов. Примерами таких серверов могут быть специальное оборудование или программные решения, например, F5-BIG-IP или Linux Virtual Server Project. Выделенный балансировщик принимает запросы и распределяет их между внутренним кластером веб-серверов. Балансировщик отвечает за обнаружение отказавшего сервера и распределение запросов по остальным. Для повышения надежности в эту схему может быть добавлен дополнительный балансировщик, который включается, когда отказывает основной.

Минусы этого подхода:

1. Существует предел запросов, которые могут быть приняты самим балансировщиком. Однако эта проблема решается введением в схему циклического DNS.
2. Поддержка балансировщика может обходиться в круглую сумму, доходя до десятков тысяч долларов. К тому же запасной балансировщик большую часть времени простаивает в ожидании, когда откажет основной.

Балансировка на стороне клиента

Существует еще один подход для распределения нагрузки на серверы от современных веб-приложений, который не нуждается в дополнительном балансирующем оборудовании, и отказ одного из серверов происходит гораздо более незаметно для клиента, чем в случае циклического DNS. Прежде чем мы углубимся в детали, давайте представим себе настольное приложение, которому требуется установить связь с серверами в интернете для получения данных. Если наше приложение создает больше запросов к удаленному серверу, чем тот может поддерживать при помощи единственной машины, нам потребуется решение для балансировки нагрузки. Можем ли мы воспользоваться циклическим DNS или балансировщиком нагрузки, описанным выше? Конечно, но существует более дешевое и надежное решение.

Вместо того, чтобы сказать клиенту, что у нас единственный сервер, можно сообщить о нескольких серверах — `s1.loadbalancedsite.ru`, `s2.loadbalancedsite.ru` и так далее. При этом клиентское приложение может случайным образом выбирать сервер для подключения и пытаться получить данные с него. Если сервер не доступен или не отвечает длительное время, клиент сам выберет другой сервер, и так далее, пока не получит свои данные.

В отличие от веб-приложений, которые хранят код (Javascript или Flash) на одном сервере, обеспечивающем доступ к этой информации, клиентское приложение независимо от сервера. Оно может само выбирать между серверами на стороне клиента для обеспечения масштабируемости приложения.

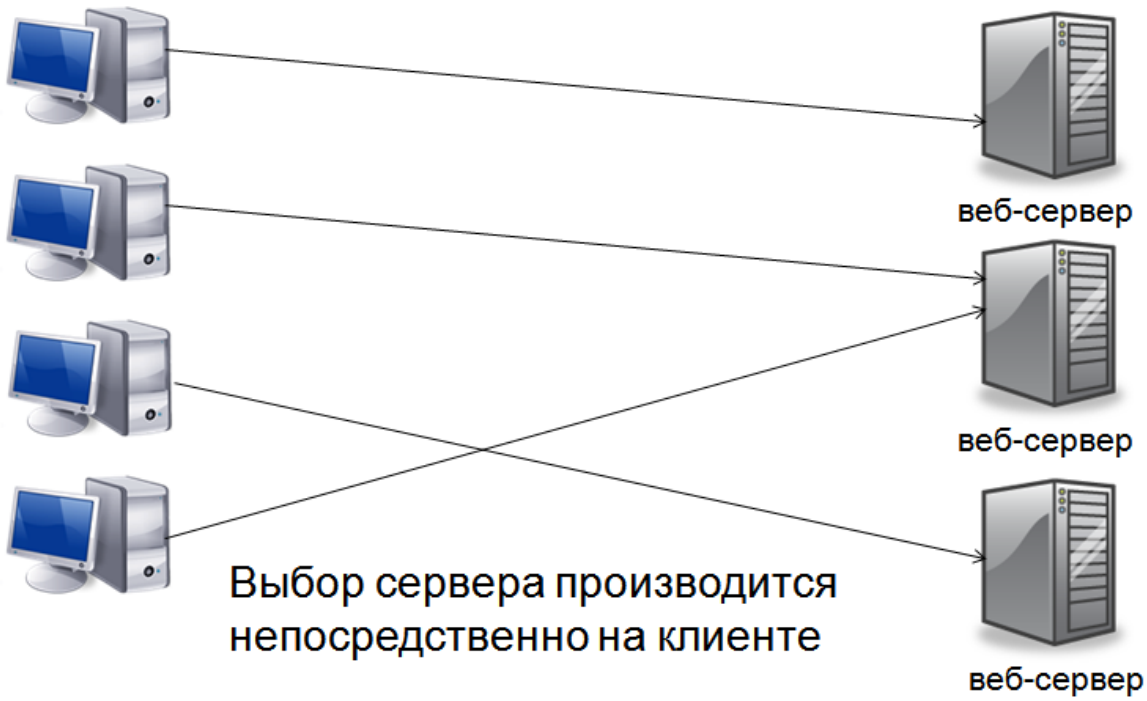


Рис. 25. Пример балансировки нагрузки и масштабируемости на клиенте

Итак, можно ли эту технику применить к веб-приложениям? Веб-приложения самой своей сутью размывают границу между клиентской и серверной частью любого стандартного приложения. Веб-приложения, написанные на PHP, часто смешивают серверный и клиентский код в одном документе. Даже при использовании паттерна MVC (модель-вид-контроллер), когда код, который генерирует уровень представления (HTML), отделен от серверной логики, все равно сервер создает и доставляет представление страницы.

Сейчас сервер обеспечивает такие ресурсы, как картинки. Но этот факт становится не столь очевидным, если рассмотреть технику CSS Sprites, когда одна картинка является источником для нескольких и CSS/JavaScript используется для «вытягивания» каждой отдельной картинки из источника. Сейчас многие приложения осуществляют только AJAX- или Flash-запросы к серверу (а не загружают каждый раз с него итоговый документ). Поэтому стандартное настольное и веб-приложение очень похожи в смысле серверных вызовов.

Для обеспечения балансировки на стороне клиента от современного веб-приложения требуется три основных составляющих.

1. Клиентский код: JavaScript и(ли) SWF (для Flash-клиентов).
2. Ресурсы: картинки, CSS (Каскадные Таблицы Стилей), аудио-, видео- и HTML-документы.
3. Серверный код: внутренняя логика для обеспечения нужных клиентам данных.

Заметно проще повысить доступность и масштабируемость HTML-кода страниц и других файлов, требуемых на клиенте, чем осуществить то же самое для серверных приложений: доставка статического содержания требует значительно меньше ресурсов. К тому же существует возможность выложить клиентский код через достаточно проверенные

сервисы, например, S3 от Amazon Web Services. Как только у нас есть код и ресурсы, обслуживаемые высоконадежной системой доставки содержания, мы можем уже подумать над балансировкой нагрузки на серверные мощности.

Мы можем включить список доступных серверов в клиентский код точно так же, как сделали бы это для настольного приложения. У веб-приложения доступен файл `servers.xml`, в котором находится список текущих серверов. Оно пытается связаться (используя AJAX или Flash) с каждым сервером в списке, пока не получит ответ. Таким образом, весь алгоритм на клиенте выглядит примерно так:

1. Загружаем файл `www.loadbalancedwebsite.ru/servers.xml`, который выложен вместе с клиентским кодом и другими ресурсами и содержит список доступных серверов, например, в следующем виде:

```
<servers>
  <server>s1.myloadbalancedwebsite.com</server>
  <server>s2.myloadbalancedwebsite.com</server>
  <server>s3.myloadbalancedwebsite.com</server>
  <server>s4.myloadbalancedwebsite.com</server>
</servers>
```

2. Выбираем случайным образом сервер из списка и пытаемся с ним соединиться. Во всех последующих запросах используем этот сервер.
3. На клиенте существует заранее установленное время ожидания запроса; если оно превышено, то возвращаемся к шагу 2.

Осуществляем кросс-доменные запросы

Даже при небольшом опыте работы с AJAX уже должно было возникнуть закономерное возражение: «Это не будет работать из-за кросс-доменной безопасности» (для предотвращения XSS-атак). Давайте рассмотрим и этот вопрос.

Для обеспечения безопасности пользователей веб-браузеры и Flash-клиенты блокирует пользовательские вызовы к другим доменам. Например, если клиентский код хочет обратиться к серверу `s1.loadbalancedwebsite.ru`, он должен быть загружен только с того же домена, `s1.loadbalancedwebsite.ru`. Запросы от клиентов на другие домены будут заблокированы. Для того чтобы обеспечить работоспособность описанной выше схемы балансировки, из клиентского кода на `www.loadbalancedwebsite.ru` требуется совершать обращения к серверам с другими доменами (например, к `s1.loadbalancedwebsite.ru`).

Для Flash-клиентов можно просто создать файл `crossdomain.xml`, который будет разрешать запросы на `*.loadbalancedwebsite.ru`:

```
<cross-domain-policy>
        <allow-access-from domain="*.myloadbalancedwebsite.com"/>
</cross-domain-policy>
```

Для клиентского кода на AJAX существуют жесткие ограничения на передачу данных между доменами, которые зависят от методов, используемых для серверных вызовов. Применение динамической загрузки скриптов для осуществления запросов позволяет

обойти ограничения по безопасности, ибо разрешает кросс-доменные вызовы. Однако в этом случае нужно будет обеспечить каким-то образом безопасность на уровне заголовков, чтобы убедиться, что именно ваш клиент осуществляет такие запросы.

Но что, если на клиенте используется `XMLHttpRequest`? XHR попросту запрещает клиенту запрашивать отличный от исходного домена сервер. Однако, существует небольшая лазейка: если клиент и сервер использует одинаковый домен верхнего уровня (для нашего примера это `www.loadbalancedwebsite.ru` и `s1.loadbalancedsite.ru`), то можно осуществлять AJAX-вызовы с использованием `iframe` и уже через него загружать документы с сервера. Браузеры позволяют скриптам обращаться к такому `iframe` как к «родному», — таким образом, становится возможным доставлять данные с помощью серверного вызова через `iframe`, если скрипты были загружены с того же домена верхнего уровня.

А если все же AJAX?

Применение динамической загрузки скриптов (она описана в начале седьмой главы) для осуществления запросов позволяет обойти ограничения по безопасности, ибо разрешает кросс-доменные вызовы. Однако эту проблему можно разрешить намного проще. Кросс-доменные запросы между доменами http://a.site.ru, http://b.site.ru на http://site.ru допустимы через свойство `document.domain`, которое надо (в данном случае) установить в `site.ru`:

```
// на странице a.site.ru
...
document.domain='site.ru'
...
// все, теперь можно делать XmlHttpRequest на site.ru
req.open("post", 'http://site.ru/result.php')
```

Проблема оказывается решенной.

Преимущества балансировки на стороне клиента

Итак, как только мы обговорили технику, позволяющую осуществлять кросс-доменные вызовы, можно обсудить, собственно, как работает сам балансировщик и как он удовлетворяет требованиям, изложенным в начале раздела.

1. **Распределение нагрузки между кластером веб-серверов.** Так как клиент выбирает сервер, с которого принимает запросы, случайным образом, загрузка будет распределена случайно (практически равномерно) между всеми имеющимися серверами.
2. **Незаметное выключение неработающего сервера из кластера.** У клиента всегда есть возможность подключиться к другому серверу, если первый не отвечает дольше заранее определенного времени. Таким образом, подключение клиента «мягко» передается от одного сервера другому.
3. **Работающий кластер доступен для клиента как один сервер**. В нашем примере пользователь просто открывает в браузере http://www.loadbalancedwebsite.ru/, который и является для клиента единственным доступным сервером. Использование всех остальных «зеркал» происходит абсолютно незаметно.

Подведем итог: каковы же преимущества балансировки на стороне клиента перед балансировкой на стороне сервера? Наиболее очевидное заключается в том, что не требуется специальное балансирующее оборудование (хотя сам клиентский код будет являться достаточно сложным, полноценным веб-приложением), и не будет никакой необходимости настраивать аппаратную часть или проверять зеркальность вторичного балансировщика для страховки основного. Если сервер не доступен, его можно просто исключить из файла `servers.xml`.

Другим преимуществом является то, что все серверы не обязаны быть расположенными в одном месте. Клиент сам выбирает, к какому серверу ему лучше подключиться, в отличие от балансирующего сервера, который рассматривает его запрос и выбирает один из кластерных серверов для его обработки. Расположение серверов ничем не ограничено. Они могут находиться в различных дата-центрах на тот случай, если один из дата-центров окажется недоступен. Если приложению требуется база данных, расположенная в локальной сети, второй дата-центр может быть по-прежнему использован как запасной, если откажет основной. Переключение с одного дата-центра на другой заключается просто в обновлении файла *`servers.xml`* вместо того, чтобы ждать распространения изменений в таблице DNS.

Используем Cloud Computing для балансировки на стороне клиента

В качестве серверной основы приложения можно рассмотреть сервисы Simple Storage Service (S3) и Elastic Computing Cloud (EC2) от Amazon Web Services (http://aws.amazon.com/).

Изначально сервис S3 предоставлял прекрасную возможность для хранения и доставки видео-сообщений, а EC2 был спроектирован именно для работы с S3. Он позволяет расширять свои мощности для поддержки большого количества пользователей весьма просто. Мощности EC2 могут быть задействованы в любое время путем простого запуска образа виртуальной машины. Каждая такая машина стоит 10 центов в час или 72 доллара в месяц. Но что более всего привлекает в EC2, так это гибкость вычислительных ресурсов: виртуальные машины EC2 могут быть отключены, когда они не используются. Например, если у приложения больше трафика в дневное время, чем ночью, то можно подключать больше серверов днем, тем самым сильно повышая денежную эффективность решения в плане хостинга.

Однако большим минусом для EC2 является невозможность проектирования балансировки нагрузки на стороне сервера, у которого не было бы уязвимых мест. Многие веб-приложения размещаются на EC2, используя только одну виртуальную машину с динамическим DNS для балансировки нагрузки запросов к отдельному домену. Если сервер, обеспечивающий балансировку, отказывает, то вся система становится недоступной, пока динамический DNS не подключит домен к другой виртуальной машине.

Пример приложения

При использовании описанной выше балансировки на стороне клиента становится возможным избежать этого неприятного момента и существенно повысить надежность всего решения на базе серверов EC2. При построении кластера виртуальных машин EC2 для поддержки балансировки на клиенте приложение использует код и другие веб-ресурсы, размещенные на S3 и отдаваемые с его помощью. Как только появляется

виртуальная машина EC2 (т.е. она полностью настроена и готова принимать запросы от клиентов), тогда приложение использует следующий подход для составления списка доступных для клиента серверов.

Чуть раньше указывалось на использование файла `servers.xml` для оповещения клиента о доступных серверах, но для S3 можно использовать более простой способ. При обращении к сегменту S3 (сегментом в S3 называют хранимую группу файлов; идея похожа на папки файлов) без каких-либо дополнительных аргументов, сервис просто перечисляет все ключи, соответствующие заданному префиксу. Таким образом, для каждой из виртуальных машин приложения на базе EC2 запускается по `cron` скрипту, который регистрирует сервер как часть общего кластера, просто создавая пустой файл с ключами `servers/{AWS IP адреса}` в публично доступном сегменте S3.

Например, по адресу http://s3.amazonaws.com/application/?actions=loadlist будет находиться следующий файл:

```
<ListBucketResult>
        <Name>voxlite</Name>
        <Prefix>servers</Prefix>
        <Marker/>
        <MaxKeys>1000</MaxKeys>
        <IsTruncated>false</IsTruncated>
        <Contents>
                <Key>servers/216.255.255.1</Key>
                <LastModified>2007-07-18T02:01:25.000Z</LastModified>
                <ETag>"d41d8cd98f00b204e9800998ecf8427e"</ETag>
                <Size>0</Size>
                <StorageClass>STANDARD</StorageClass>
        </Contents>
        <Contents>
                <Key>servers/216.255.255.2</Key>
                <LastModified>2007-07-20T16:32:22.000Z</LastModified>
                <ETag>"d41d8cd98f00b204e9800998ecf8427e"</ETag>
                <Size>0</Size>
                <StorageClass>STANDARD</StorageClass>
        </Contents>
</ListBucketResult>
```

В этом примере присутствуют два EC2 сервера в кластере, с IP адресами 216.255.255.1 и 216.255.255.2, соответственно.

Логика для скрипта, запускающегося по расписанию

1. Загрузить и разобрать http://s3.amazonaws.com/application/?actions=loadlist .
2. Если текущий сервер отсутствует в списке, создать пустой файл в сегменте с ключом `servers/{IP адрес EC2 сервера}`.
3. Проверить, доступны ли остальные серверы, записанные в сегменте, проверив связь до них, используя внутренний AWS IP адрес. Если связь установить не удается, то ключ сервера из сегмента удаляется.

Так как скрипт, запускающийся по `cron`, является частью виртуальной машины EC2, каждая такая машина автоматически регистрируется как доступный сервер в кластере. Клиентский код (AJAX или Flash) разбирает список ключей в сегменте, вычленяет внешнее имя AWS сервера и добавляет его в массив для случайного выбора при соединении, как описано выше при рассмотрении файла `servers.xml`.

Если виртуальная машина EC2 отказывает или выключается, то другие машины самостоятельно убирают ее запись из сегмента: в сегменте остаются только доступные серверы. Дополнительно — клиент сам выбирает другой сервер EC2 в сегменте, если ответ не был получен в течение определенного времени. Если трафик на веб-сайт увеличивается, достаточно просто запустить больше серверов EC2. Если нагрузка уменьшается, можно часть из них отключить. Использование балансировки на стороне клиента при помощи S3 и EC2 позволяет легко создать гибкое, расширяемое и весьма надежное веб-приложение.

5.4. Редиректы, 404-ошибки и повторяющиеся файлы

От вопросов распределения запросов и их балансировки давайте перейдем к оптимизации самой структуры веб-страницы и посмотрим, как в ряде случаев ее улучшение позволяет избежать существенных задержек и снизить издержки на доставку компонентов страницы конечному пользователю.

Редиректы

Редиректы осуществляются посредством отправки клиенту статус-кодов HTTP 301, 302 и 307. В качестве примера можно рассмотреть такой HTTP-заголовок со статус-кодом 301:

```
HTTP/1.1 301 Moved Permanently
Location: http://example.com/newuri
```

Браузер автоматически перенаправляет пользователя на новый адрес, указанный в поле `Location`. Вся информация, необходимая для редиректа, есть в этих заголовках, тело ответа обычно остается пустым. Результаты редиректов (ни с кодом 301, ни с кодом 302) на практике не кэшируются, пока это явно не объявляется заголовком `Expires` либо `Cache-Control`.

Также для перенаправления пользователя используется мета-тег `refresh` и JavaScript (`location.href`), однако, если все же необходимо сделать редирект, предпочтительней применение именно статус-кодов HTTP 301 и 302 со стороны сервера. В этом случае у пользователя будут правильно работать кнопки «Назад» и «Вперед». В случае JavaScript речь пойдет только о тех случаях, когда при загрузке первоначального HTML-файла пользователь сразу отправляется на новую страницу. Если же JavaScript используется только для динамической навигации, то это совершенно нормальная ситуация, и она не является ошибочной.

Главое, что нужно помнить при использовании редиректов — это то, что они отнимают время на свое выполнение, а пользователь должен ждать его завершения. Страница даже не может начать отображаться из-за того, что пользователь еще не получил сам HTML-документ, и браузер не может начать загрузку остальных компонентов страницы.

Одним из бесполезных редиректов, которые часто используются (и веб-разработчики не стремятся избегать этого) — когда пользователь забывает ввести завершающий слэш (/) в адресной строке в тех случаях, когда он там должен быть. Например, если попробовать открыть адрес http://webo.in/articles , то браузер получит ответ с кодом 301, содержащий редирект на http://webo.in/articles/ (в последнем случае содержится завершающий слэш). Это исправляется в Apache использованием `Alias` или `mod_rewrite`, или же `DirectorySlash`, если применяются Apache handlers.

Объединение старого и нового сайтов также часто является причиной использования редиректов. Кое-кто объединяет часть старого и нового сайтов и перенаправляет (или не перенаправляет) пользователей, основываясь на ряде факторов: браузере, типе аккаунта пользователя и т.д. Применение редиректов для объединения двух сайтов является достаточно простым способом и требует минимального программирования, но усложняет поддержку проекта для разработчиков и ухудшает восприятие страницы пользователями. Альтернативой редиректу является использование модулей `mod_alias` и `mod_rewrite` в случае, если оба URI находятся в пределах одного сервера. Если же причиной появления редиректов является перенаправление пользователя между разными хостами, как альтернативу можно рассматривать создание DNS-записей типа CNAME (такие записи создают псевдонимы для доменов) в комбинации с `Alias` или `mod_rewrite`.

Повторяющиеся файлы

Включение одного скрипта дважды на одну страницу снижает производительность. Это не так редко встречается, как можно подумать. Два из десяти наиболее посещаемых сайтов содержат повторяющийся JavaScript-код (обращения к одинаковым JavaScript-файлам). Два основных фактора, которые могут повлиять на возникновение повторяющихся скриптов,— это количество скриптов на странице и количество разработчиков. Если происходит описанная ситуация, повторение скриптов замедляет работу сайта ненужными HTTP-запросами и вычислениями.

Повторяющиеся запросы возникают в Internet Explorer (если версия IE меньше 7, то для загрузки одинаковых картинок на одной странице также может отправляться соответствующее число запросов). Internet Explorer дважды загружает один и тот же скрипт, если он включен в страницу два раза и не кэшируется. Но даже если скрипт закэширован, все равно возникает дополнительный HTTP-запрос, когда пользователь перезагружает страницу.

В дополнение к ненужным HTTP-запросам, тратится время на выполнение кода. Повторное исполнение кода происходит во всех браузерах, вне зависимости от того, был ли закэширован скрипт или нет.

Единственным способом избежать повторного включения одного и того же скрипта является разработка системы управления скриптами в виде модуля системы шаблонов. Обычным способом включения скрипта в страницу является использование тега `script`:

```
<script type="text/javascript" src="menu_1.0.17.js"></script>
```

Альтернативой в PHP можно считать создание функции `insertScript`:

```
<?php insertScript("menu.js") ?>
```

Кроме простого предотвращения включения одного скрипта на страницу дважды, такая функция может выполнять и другие задачи, к примеру, отслеживать зависимости между скриптами, добавлять номер версии в название файла скрипта для поддержки HTTP-заголовков `Expires` и пр.

404-ошибки

Если сервер не может удовлетворить запрос браузера по причине того, что ни один файл не соответствует запрошенному, то он отвечает со статус-кодом 404 (`File Not Found`). Таким образом, браузер понимает, что не может получить соответствующий ресурс, и стандартным образом обрабатывает эту ошибку.

Вроде все хорошо: процесс уже давно описан в спецификации, и браузеры действуют строго согласно ей. Однако в данном случае проблема заключается в том, что если браузер получит 404-ответ от сервера при запросе какого-либо файла, который нужен для данной страницы, то этот запрос можно вообще не осуществлять. Это расходует время загрузки, расходует ресурсы сервера, расходует канал. Обычно при ответе с кодом 404 сервер пересылает еще и HTML-документ, который браузер может отобразить (это та самая 404-страница).

Теперь представим, что браузер запросил небольшую фоновую картинку в 500 байтов. Вместо этого он получает 10 Кб HTML-кода, который не может отобразить (потому что это не картинка). Это совсем плохо.

К счастью, все такие запросы легко отследить и устранить при своевременном анализе сайта на «битые» ссылки.

5.5. Асинхронные HTTP-запросы

Для большинства сайтов загрузка страницы затрагивает десятки внешних объектов, основное время загрузки тратится на различные HTTP-запросы картинок, JavaScript-файлов и файлов стилей. При работе над оптимизацией времени загрузки страницы в сложном AJAX-приложении было исследовано, насколько можно уменьшить задержку за счет внешних объектов. Особый интерес при этом был вызван конкретной реализацией HTTP-клиента в известных браузерах и параметрами распространенных интернет-соединений, а также к тому, как они влияют на загрузку страницы, содержащих большое количество маленьких объектов.

Можно отметить несколько интересных фактов.

- В IE, Firefox и Safari по умолчанию выключена техника HTTP-конвейера (*англ. HTTP pipelining*). Opera является единственным браузером, где она включена. Отсутствие конвейера при обработке запросов означает, что после каждого ответа на запрос его соединение освобождается прежде, чем отправлять новый запрос. Не следует путать конвейерную обработку с `Connection: keep-alive`, когда браузер может использовать одно соединение с сервером, чтобы загружать через него достаточно большое количество ресурсов. В случае конвейера браузер может послать несколько GET-запросов в одном соединении, не дожидаясь ответа от сервера. Сервер в таком случае должен ответить на все запросы последовательно. Это влечет дополнительные задержки на прохождение запроса туда-обратно, что, в общем случае, примерно равно времени `ping` (отнесенном к разрешенному числу одновременных соединений). Если же на сервере нет элементов поддержки активных HTTP-соединений, то это повлечет еще одно трехступенчатое TCP «рукопожатие», которое, в лучшем случае, удваивает задержку.
- По умолчанию, в IE (а с ним работают сейчас 50–70% пользователей) можно установить только два внешних соединения на один хост при запросе на сервер, поддерживающий HTTP/1.1, или всего 8 исходящих соединений. Использование 4 хостов вместо одного может обеспечить большее число одновременных

соединений. IP-адрес в таком случае не играет роли: все хосты могут указывать на один адрес.

- У большинства DSL- или выделенных Интернет-соединений несимметричная полоса пропускания, она варьируется от 1,5 Мб входящего/128 Кб исходящего до 6 Мб входящего / 512 Кб исходящего и т.д. Отношение входящего к исходящему каналу, в основном, находятся в пределах от 5:1 до 20:1. Это означает для ваших пользователей, что отправка запроса занимает столько же времени, как и принятие ответа, который в 5–20 раз больше самого запроса. Средний запрос занимает около 500 байтов, поэтому больше всего влияния ощущают объекты, которые меньше, чем, может быть, 2,5–10 Кб. Это означает, что доставка небольших объектов может существенно снизить скорость загрузки страницы в силу ограничения на исходящий канал, как это ни странно.

Моделируем параллельные запросы

На основе заявленных предпосылок можно смоделировать эффективную ширину канала для пользователей, учитывая некоторые сетевые особенности при загрузке объектов различных размеров. Предположим, что каждый HTTP-запрос занимает 500 байтов и что HTTP-ответ содержит дополнительно к размеру запрошенного объекта еще 500 байтов заголовков. Это наиболее простая модель, которая рассматривает только ограничения на канал и его асимметрию, но не учитывает задержки на открытие TCP-соединения при первом запросе для активного соединения, которые, однако, сходят на нет при большом количестве объектов, передаваемых за один раз.

Следует также отметить, что рассматривается наилучший случай, который не включает другие ограничения, например, «медленный старт» TCP, потерю пакетов и т.д. Полученные результаты достаточно интересны, чтобы предложить несколько путей для дальнейшего исследования, однако, не могут никоим образом рассматриваться как замена экспериментов, проведенных с помощью реальных браузеров.

Чтобы выявить эффект от активных соединений и введения нескольких хостов, давайте возьмем пользователя с интернет-соединением с 1,5 Мб входящим / 384 Кб исходящим каналом, находящегося на расстоянии 100 мс без потери пакетов. Это в очень грубом приближении соответствует среднему ADSL-соединению на другом краю России с серверами, расположенными в Москве. Ниже показана эффективная пропускная способность канала при загрузке страницы с множеством объектов определенного размера. Эффективная пропускная способность определялась как отношение общего числа полученных байтов ко времени их получения.

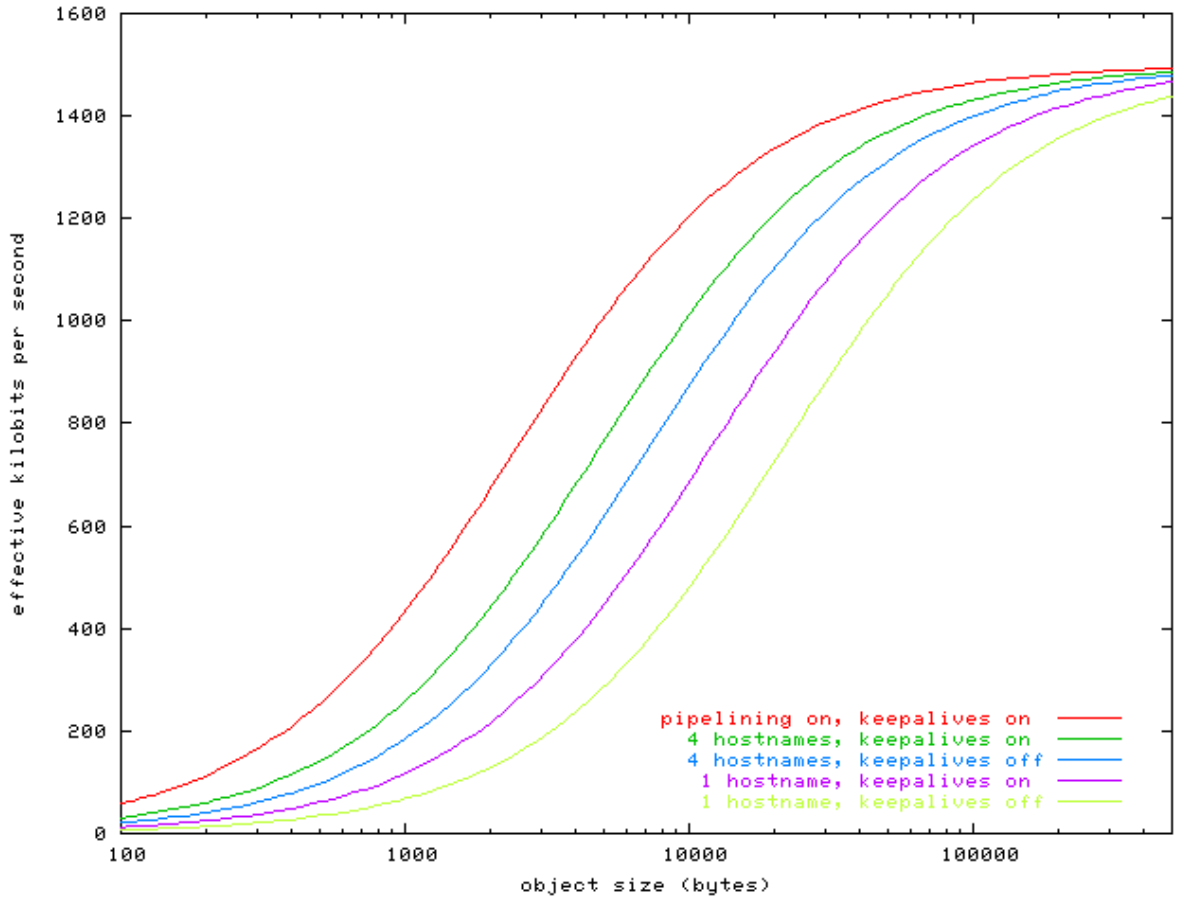


Рис. 26. Влияние HTTP-конвейера и постоянного соединения на скорость передачи данных. Источник: www.die.net

Предварительные выводы

Следует отметить следующее.

- Для относительно небольших объектов (левая часть графика) можно сказать, судя по большому пустому месту над линиями, что пользователь очень слабо использует свой канал, при этом браузер запрашивает объекты так быстро, насколько может. Для такого пользователя эффективное использование входящего канала наступает при загрузке объектах размером 100 Кб и более.
- Для объектов размером примерно в 8 Кб можно удвоить эффективную пропускную способность канала, включив постоянные соединения на сервере и распределив запросы по 4 серверам. **Это значительное преимущество.**
- Если пользователь включит конвейерную передачу запросов в своем браузере (для Firefox это будет `network.http.pipelining` в `about:config`), число используемых хостов перестанет играть значительную роль, и он будет задействовать свой канал еще более эффективно, однако, мы не сможем это контролировать на стороне сервера.

Возможно, более прозрачным будет следующий график, на котором изображено несколько различных интернет-соединений и выведено относительное ускорение для запроса страницы с множеством мелких объектов для случая использования 4 хостов и

включения активного соединения на сервере. Ускорение измеряется относительно случая 1 хоста с выключенным `keep-alive` (0%).

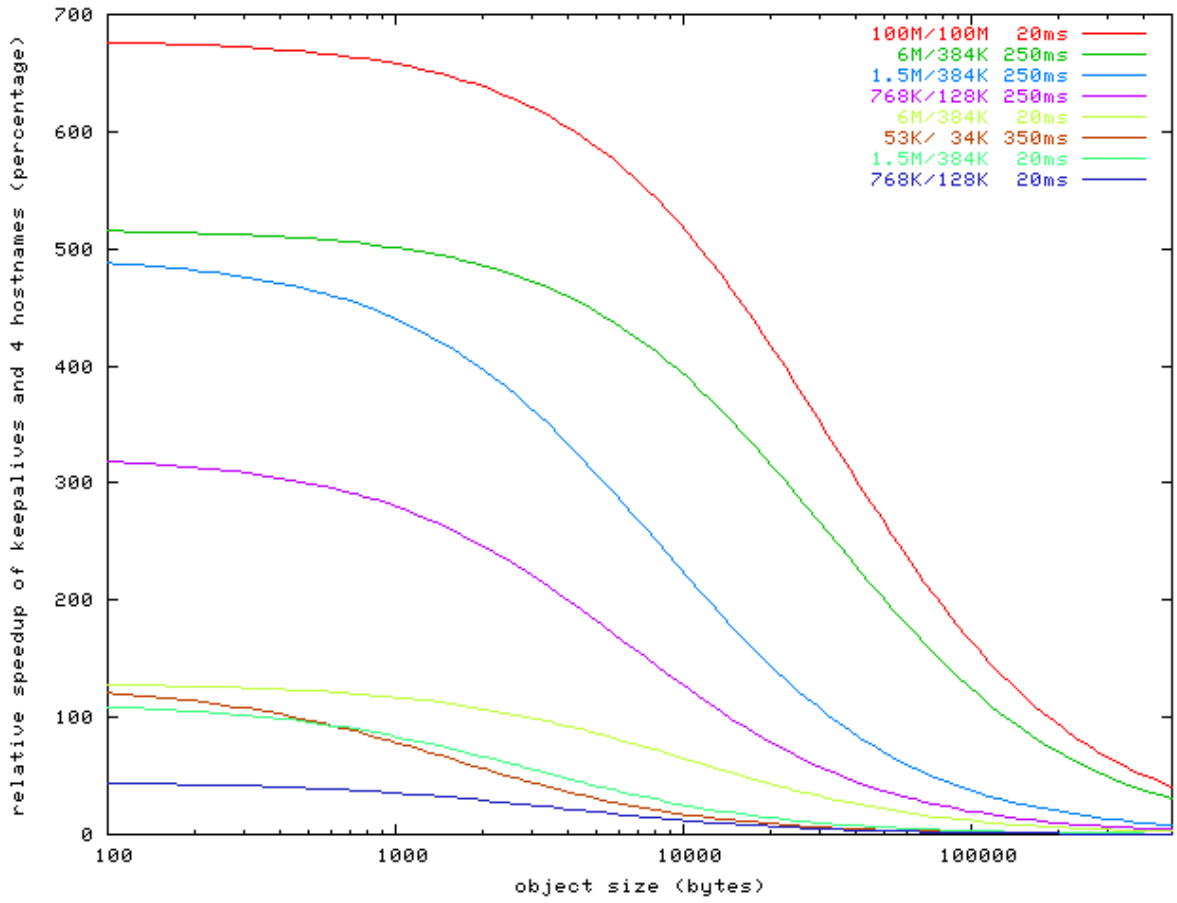


Рис. 27. Выигрыш при включении постоянного соединения и нескольких хостов для различных пользователей. Источник: www.die.net

Что тут интересного?

- Если вы загружаете много мелких объектов, меньших, чем 10 Кб, оба пользователя, и тот, кто находится локально, и тот, кто на другом конце континента, почувствуют значительное ускорение от включения активного соединения и введения 4 хостов вместо одного.
- Чем дальше находится пользователь, тем значительнее выигрыш.
- Чем больше скорость соединения, тем больше выигрыш. Пользователь с ethernet-каналом в 100 Мб на расстоянии всего 20 мс от сервера почувствует наибольшее ускорение.

Влияние заголовков

Давайте теперь посмотрим, как размер заголовков влияет на эффективную пропускную способность канала. Предыдущий график предполагает, что размер заголовков составляет 500 байтов дополнительно к размеру объекта, как для запроса, так и для ответа. Как же изменение этого параметра отразится на производительности нашего 1,5 Мб / 384 Кб канала и расстояния до пользователя в 100 мс? Предполагается, что пользователь уже изначально использует 4 хоста и активное соединение.

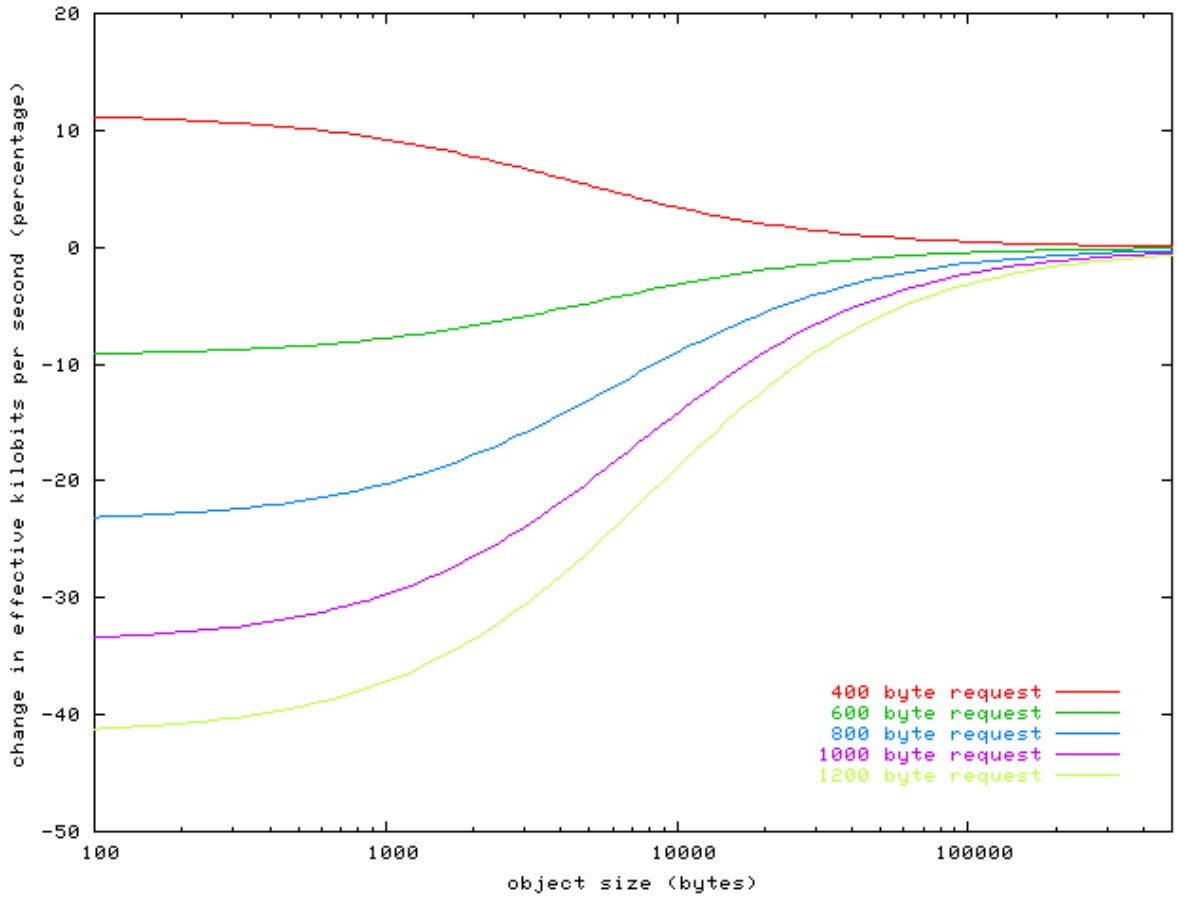


Рис. 28. Влияние заголовков на эффективную пропускную способность канала. Источник: www.die.net

На графике хорошо видно, что при небольших размерах файлов основные задержки приходятся на исходящий канал. Браузер, отправляющий «тяжелые» запросы на сервер (например, с большим количеством cookie), по-видимому, замедляет скорость передачи данных более чем на 40% для этого пользователя. Естественно, размер cookie можно и нужно регулировать на сервере. Отсюда простой вывод: cookie нужно, по возможности, делать минимальными или направлять ресурсные запросы на серверы, которые не выставляют cookie.

Все полученные графики являются результатом моделирования и не учитывают некоторые реальные особенности окружающего мира. Но можно взять и проверить полученные результаты в реальных браузерах в «боевых» условиях и убедиться в их состоятельности. Реальное положение дел определенно усугубляет ситуацию, ибо медленные соединения и большие задержки становятся еще больше, а быстрые методы соединения по-прежнему выигрывают.

5.6. Уплотняем поток загрузки

Рассмотрев методы сжатия, объединения, кэширования и создания параллельных соединений, разумно было бы заняться следующим вопросом: Какая часть страницы должна загружаться вместе с основным HTML-файлом, а какая — только с внешними файлами?

Было собрано тестовое окружение в виде одной страницы, для которой применены различные оптимизационные техники (заодно было получено реальное ускорение для загрузки произвольной страницы и показано, как все эти техники реально влияют на скорость загрузки страницы).

Кроме того, были проведены теоретические выкладки для определения оптимального распределения загрузки по стадиям с учетом всех аспектов.

Реальная ситуация

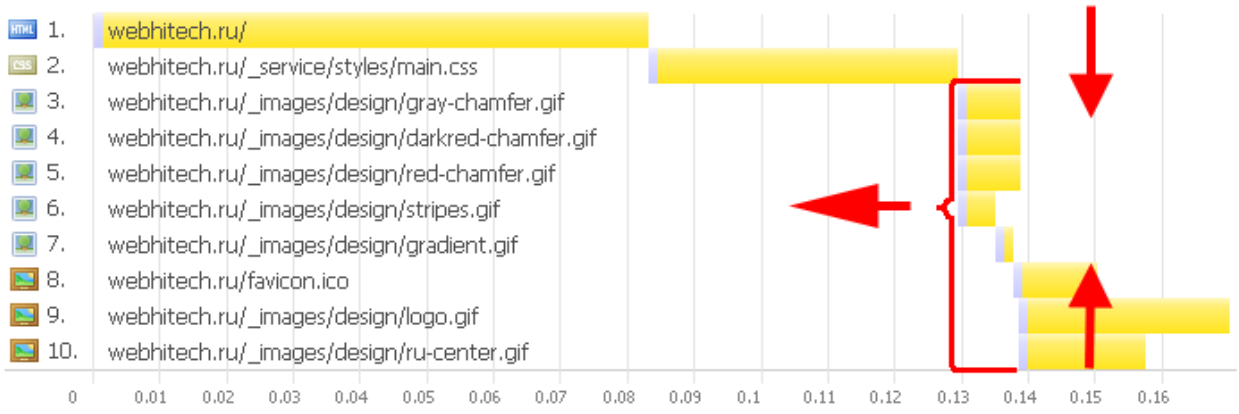


Рис. 29. Диаграмма загрузки (неизмененного) сайта WebHiTech.ru

Основная идея вариации потока загрузки заключалась в создании минимального количества «белых мест» на диаграмме загрузки. Как видно из рис. 29, около 80% при загрузке страницы составляют простои соединений (естественно, что данный график не отражает реальную загрузку открытых в браузере каналов загрузки, однако, при уточнении картины ситуация практически не меняется). Параллельные загрузки начинаются только после прохождения «узкого места», которое заканчивается (в данном случае) после предзагрузки страницы — после CSS-файла.

Для оптимизации скорости загрузки нам нужно уменьшить число файлов (вертикальные стрелки), загружающихся параллельно, и «сдвинуть» их максимально влево (горизонтальная стрелка). Уменьшение «белых мест» (фактически, уменьшение простоя каналов загрузки), по идее, должно увеличить скорость загрузки за счет ее распараллеливания. Давайте посмотрим, действительно ли это так и как этого добиться.

Шаг первый: простая страница

Вначале бралась обычная страница, для которой использовалось только `gzip`-сжатие HTML-файла. Это самое простое, что может быть сделано для ускорения загрузки страницы. Данная оптимизация бралась за основу, с которой сравнивалось все остальное. Для тестов препарировалась главная страница конкурса WebHiTech (http://webhitech.ru/) с небольшим количеством дополнительных картинок (чтобы было больше внешних объектов, и размер страницы увеличивался).

В самом начале (`head`) страницы замеряется начальное время, а по событию `window.onload` (заметим, что только по нему, ибо только оно гарантирует, что **вся**

страница целиком находится в клиентском браузере) — конечное, затем вычисляется разница. Но этот очень простой пример, перейдем к следующим шагам.

Шаг второй: уменьшаем изображения

Для начала минимизируем все исходные изображения (основные прикладные техники уже были освещены во второй главе). Получилось довольно забавно: суммарный размер страницы уменьшился на 8%, и скорость загрузки возросла на 8% (т.е. получилось пропорциональное ускорение).

Дополнительно с минимизацией картинок была уменьшена таблица стилей (через CSS Tidy) и сам HTML-файл (убраны лишние пробелы и переводы строк). Скриптов на странице не было, поэтому общее время загрузки изменилось не сильно. Но это еще не конец, и мы переходим к третьему шагу.

Шаг третий: все-в-одном

Можно использовать `data:URI` и внедрить все изображения в соответствующие HTML/CSS-файлы, уменьшив, таким образом, размер страницы (за счет `gzip`-сжатия, по большому счету, потому что таблица стилей перед этим не сжималась) еще на 15%, однако, время загрузки при этом уменьшилось всего на 4% (при включенном кэшировании, уменьшилось число запросов с 304-ответом). При загрузке страницы в первый раз улучшения гораздо более стабильны: 20%.

CSS-файл, естественно, тоже был включен в HTML, поэтому при загрузке всей страницы осуществлялся только один запрос к серверу (для отображения целой страницы с парой десяткой объектов).

Шаг четвертый: нарезаем поток

Можно попробовать распределить первоначальный монолитный файла на несколько (5–10) равных частей, которые бы затем собирались и внедрялись прямо в `document.body.innerHTML`. Т.е. сам начальный HTML-файл очень мал (фактически, содержит только предзагрузчик) и загружается весьма быстро, а после этого стартует параллельная загрузка еще множества одинаковых файлов, которые используют канал загрузки максимально плотно.

Однако, как показали исследования, издержки на XHR-запросы и сборку `innerHTML` на клиенте сильно превосходят выигрыш от такого распараллеливания. В итоге, страница будет загружаться в 2–5 раз дольше, размер при этом изменяется не сильно.

Можно попробовать использовать вместо XHR-запросов классические `iframe`, чтобы избежать части издержек. Это помогает, но не сильно. Страница все равно будет загружаться в 2–3 раза дольше, чем хотелось бы.

И немного к вопросу применения фреймов: очень часто наиболее используемые части сайта делают именно на них, чтобы снизить размер передаваемых данных. Как уже упомянуто выше, основная часть задержек происходит из-за большого количества внешних объектов на странице, а не из-за размера внешних объектов. Поэтому на данный момент эта технология далеко не так актуальна, как в 90-е годы прошлого столетия.

Также стоит упомянуть, что при использовании `iframe` для навигации по сайту встает проблема обновления этой самой навигации (например, если мы хотим выделить какой-то пункт меню как активный). Корректное решение этой проблемы требует от пользователя включенного JavaScript, и оно довольно нетривиально с технической стороны. В общем, если без фреймов можно обойтись при проектировании сайта — значит, их не нужно использовать.

Шаг пятой: алгоритмическое кэширование

Проанализировав ситуацию с первыми тремя шагами, мы видим, что часть ускорения может быть достигнута, если предоставить браузеру возможность самому загружать внешние файлы как отдельные объекты, а не как JSON-код, который нужно как-то преобразовать. Дополнительно к этому всплывают аспекты кэширования: ведь быстрее загрузить половину страницы, а для второй половины проверить запросами со статус-кодами 304, что объекты не изменились. Загрузка всей страницы клиентом в первый раз в данном случае будет немного медленнее (естественно, решение по этому поводу будет зависеть от числа постоянных пользователей ресурса).

В результате удалось уменьшить время загрузки еще на 5%, итоговое ускорение (в случае полного кэша) достигло 20%, размер страницы при этом уменьшился на 21%. Возможно вынесение не более 50% от размера страницы в загрузку внешних объектов, при этом объекты должны быть примерно равного размера (расхождение не более 20%). В таком случае скорость загрузки страницы для пользователей с полным кэшем будет наибольшей. Если страница оптимизируется под пользователей с пустым кэшем, то наилучший результат достигается только при включении всех внешних файлов в исходный HTML.

Итоговая таблица

Ниже приведены все результаты оптимизации для отдельной взятой страницы. Загрузка тестировалась на соединении 100 Кб/с, общее число первоначальных объектов: 23.

Номер шага	Описание	Общий размер (кб)	Время загрузки (мс)
1	Обычная страница. Ничего не сжато (только html отдается через gzip)	63	117
2	HTML/CSS файлы и картинки минимизированы	58	108
3	Один-единственный файл. Картинки вставлены через `data:URI`	49	104
4	HTML-файл параллельно загружает 6 частей с данными и собирает их на клиенте	49	233
4.5	HTML-файл загружает 4 `iframe`	49	205
5	Вариант #3, только JPEG-изображения (примерно одинаковые по размеру) вынесены в файлы и загружаются через `(new Image()).src` в `head` странице	49	98

Таблица 5. Различные способы параллельной загрузки объектов на странице

Шаг шестой: балансируем стадии загрузки

Итак, как нам лучше всего балансировать загрузку страницы между ее стадиями? Где та «золотая середина», обеспечивающая оптимум загрузки? Начнем с предположения, что у нас уже выполнены все советы по уменьшению объема данных. Это можно сделать всегда, это достаточно просто (в большинстве случаев нужны лишь небольшие изменения в конфигурации сервера). Также предположим, что статика отдается уже с кэширующими заголовками (чтобы возвращать 304-ответы в том случае, если ресурсный файл физически не изменился с момента последнего посещения).

Что дальше? Дальнейшие действия зависят от структуры внешних файлов. При большом (больше двух) числе файлов, подключаемых в `<head>` страницы, необходимо объединить файлы стилей и файлы скриптов. Ускорение предзагрузки страницы будет налицо.

Если объем скриптов даже после сжатия достаточно велик (больше 10 Кб), то стоит их подключить перед закрывающим `</body>`, либо вообще загружать по комбинированному событию `window.onload` (динамической загрузке скриптов посвящено начало седьмой главы). Тут мы, фактически, переносим часть загрузки из второй стадии в четвертую, ускоряется лишь «визуальная» загрузка страницы.

Общее количество картинок должно быть минимальным. Однако тут тоже очень важно равномерно распределить их объем по третьей стадии загрузки. Довольно часто одно изображение в 50–100 Кб тормозит завершение загрузки, разбиение его на 3–4 составляющие способно ускорить общий процесс. Поэтому при использовании большого количества фоновых изображений лучше разбивать их на блоки по 10–20, которые будут загружаться параллельно.

Шаг седьмой: балансируем кэширование

Если все же на странице присутствует больше 10 внешних объектов в третьей стадии (картинок и различных мультимедийных файлов), тут уже стоит вводить дополнительный хост для увеличения числа параллельных потоков. В этом случае издержки на DNS-запрос окупятся снижением среднего времени установления соединения. 3 хоста стоит вводить уже после 20 объектов, и т.д. Всего не более 4 (как показало исследование рабочей группы Yahoo! после 4 хостов издержки, скорее, возрастут, чем снизятся).

Вопрос о том, сколько объема страницы включать в сам HTML-файл (кода в виде CSS, JavaScript или `data:URI`), а сколько оставлять на внешних объектах, решается очень просто. Баланс в данном случае примерно равен соотношению числа постоянных и единовременных посещений. Например, если 70% пользователей заходят на сайт в течение недели, то примерно 70% страницы должно находиться во внешних объектах и только 30% — в HTML-документе.

Когда страницу должны увидеть только один раз, логично будет включить все в саму страницу. Однако тут уже вступают в силу психологические моменты. Если у среднего пользователя страница при этом будет загружаться больше 3–4 секунд (учитывая время на DNS-запрос и соединение с сервером), то будет необходимо разбиение на две части: первоначальная версия, которая отобразится достаточно быстро, и остальная часть страницы.

Очень важно понимать, какая стадия загрузки при этом оптимизируется и что видит реальный пользователь (с чистым кэшем и, может быть, небыстрым каналом). Подробнее

об анализе процесса загрузки страницы на конкретных примерах рассказывается в восьмой главе.

Заключение

Вот так, на примере обычной страницы (уже достаточно хорошо сделанной, стоит отметить) мы добились ускорения ее загрузки еще на 15–20% (и это без учета применения `gzip`-сжатия для HTML, которое в данном случае дает примерно 10% от общей скорости). Наиболее важные методы уже приведены выше, сейчас лишь можно упомянуть, что при оптимизации скорости работы страницы лучше всегда полагаться на внутренние механизмы браузера, а не пытаться их эмулировать на JavaScript (в данном случае речь идет об искусственной «нарезке» потока). Может быть, в будущем клиентские машины станут достаточно мощными (или же JavaScript-движки — лучше оптимизированными), чтобы такие методы заработали. Сейчас же выбор один — алгоритмическое кэширование.

Глава 6. CSS оптимизация

6.1. Оптимизируем CSS expressions

CSS-производительность не находится сейчас в фокусе внимания при разработке клиентских приложений для браузера. Очень часто о некоторых ключевых моментах просто не знают (или забывают), и это может привести к появлению множества «узких» мест при работе веб-приложения, которые не зависят непосредственно от сервера или канала. Они все находятся на стороне браузера.

После того, как доставка всех необходимых объектов для отображения страницы оптимизирована, можно приступать к рассмотрению других сторон клиентской производительности. Одно из них заключается в особенностях работы CSS-движка браузера и его взаимодействии с JavaScript. Давайте рассмотрим все по порядку.

CSS-выражения

CSS-выражения (*англ. CSS expressions*) были впервые представлены в Internet Explorer 5.0, который позволял назначать JavaScript-выражение в качестве CSS-свойства. Например, следующий код позволит выставить позицию элемента в зависимости от того, какого размера окно браузера.

```
#myDiv {
        position: absolute;
        width: 100px;
        height: 100px;
        left: expression((document.body.offsetWidth > 110 ?
                document.body.offsetWidth - 110 : 110) + "px");
        top: expression(document.body.offsetHeight - 110 + "px");
        background: red;
}
```

Не самый лучший способ решения поставленной задачи, но в качестве примера его достаточно.

Проблема с этими выражениями в том, что они вычисляются гораздо чаще, чем многие могли бы ожидать. Они вычисляются не только во время визуализации страницы и изменения размеров окна, но также при скроллинге и даже когда пользователь просто водит мышкой над страницей. Это несложно отследить — достаточно добавить счетчик в искомое выражение.

Единственный способ избежать огромного числа вычисления CSS-выражений — использование одноразовых выражений, когда после проведения всех необходимых вычислений они устанавливают свойство CSS-стиля к какому-то конечному статическому значению, заменяя им CSS-выражение. В том случае, если необходимо динамически изменять свойство CSS-стиля по мере пребывания пользователя на странице, мы можем применить прием с обработчиками событий в качестве альтернативы. Если избежать использования CSS-выражений на странице не удается, то нужно помнить, что они могут вычисляться тысячи раз и тем самым повлиять на производительность всей страницы.

Динамические выражения

CSS-выражения позволяют не только вычислить CSS-свойство при объявлении стилей, но и поддерживать его постоянно в актуальном состоянии, чтобы заданное выражение было всегда верно. Это означает, что само выражение будет пересчитываться каждый раз, когда (это касается только рассмотренного примера) изменяется `document.body.offsetWidth`. Если бы не этот факт, динамические выражения, возможно, принесли бы большую пользу и получили бы более широкое распространение. Но это не так, и пересчет этой строки происходит каждый раз, когда заканчивается вычисления JavaScript. И не нужно быть гением, чтобы понять, что это приведет наше веб-приложение к «подвисанию».

Давайте рассмотрим следующий блок CSS-кода:

```
#myDiv {
        border: 10px solid Red;
        width:  expression(ieBox ? "100px" : "80px");
}
```

Даже при том предположении, что `ieBox` — это постоянный флаг, который выставляется в `true`, когда IE находится в режиме обратной совместимости, заданное выражение будет вычисляться каждый раз в `"80px"`. Хотя выражение будет постоянным для данной страницы, оно все равно будет пересчитываться много раз. Основной вопрос заключается в том, как избавиться от этих ненужных вычислений.

Вычисление постоянных

Вот что мы собираемся сделать: пройтись по всем объявлениям стилей и заменить вычисление выражения его постоянным значением. В предыдущем примере, предполагая, что мы используем IE6 в стандартном режиме, нам хотелось бы видеть следующий код:

```
#myDiv {
        border: 10px solid Red;
        width:  80px;
}
```

Итак, как нам убедиться в том, что наше выражение постоянно? Самым простым путем является пометить само выражение, чтобы мы могли его легко обнаружить. Решением в данном случае будет заключение выражение в вызов функции, которая нам известна и заранее объявлена.

```
function constExpression(x) {
        return x;
}
```

Итак, в нашем CSS-блоке мы напишем следующее:

```
#myDiv {
        border: 10px solid Red;
        width:  expression(constExpression(ieBox ? "100px" : "80px"));
}
```

Использование

Во-первых, мы сперва должны подключить библиотеку `cssexpr.js` (о ней речь чуть ниже) и только потом вызывать нашу функцию `constExpression`.

```
<script type="text/javascript" src="cssexpr.js"></script>
```

После этого можно использовать `constExpression` в любом задаваемом блоке стилей (`<style>`), или любом подключаемом файле стилей (`<link>`), или при использовании директивы `@import`. Следует заметить, что атрибут `style` у тегов не проверяется для ускорения работы.

Реализация

Идея заключается в том, чтобы перебрать все объявленные таблицы стилей, а в них — все правила и их конечные объявления. Для этого мы начнем с массива `document.styleSheets`.

```
function simplifyCSSExpression() {
        try {
                var ss = document.styleSheets;
                var i = ss.length

                while (i-- > 0) {
                        simplifyCSSBlock(ss[i]);
                }
        }
        catch (exc) {
                alert("Обнаружили ошибку при обработке css. Страница будет " +
                        "работать в прежнем режиме, хотя, возможно, не так " +
                        "быстро");
                throw exc;
        }
}
```

В таблицах стилей мы пройдемся по массиву импортируемых таблиц (`@import`), а затем уже по объявлениям стилевых правил. Для того чтобы не совершать пустых телодвижений, будем проверять, что `cssText` содержит `expression(constExpression)`.

```
function simplifyCSSBlock(ss) {
        // Проходимся по import'ам
        var i = ss.imports.length;
        while (i-- > 0)
                simplifyCSSBlock(ss.imports[i]);

        // если в cssText'е нет constExpression, сворачиваемся
        if (ss.cssText.indexOf("expression(constExpression(") == -1)
                return;

        var rs = ss.rules;
        var rl = rs.length;
        while (rl-- > 0)
                simplifyCSSRule(rs[j]);
}
```

Затем мы уже можем обрабатывать для каждого правила `cssText` и заменять его, используя функцию `simplifyCSSRuleHelper`, чтобы текст объявления из динамического становился статическим.

```
function simplifyCSSRule(r) {
        var str = r.style.cssText;
        var str2 = str;
```

```
        var lastStr;

        // обновляем строку, пока она еще может обновляться
        do {
                lastStr = str2;
                str2 = simplifyCSSRuleHelper(lastStr);
        } while (str2 != lastStr)

        if (str2 != str)
                r.style.cssText = str2;
}
```

Вспомогательная функция находит первое возможное выражение и исполняет его, затем заменяет выражение полученным значением.

```
function simplifyCSSRuleHelper(str) {
        var i = str.indexOf("expression(constExpression(");
        if (i == -1)
                return str;
        var i2 = str.indexOf("))", i);
        var hd = str.substring(0, i);
        var tl = str.substring(i2 + 2);
        var exp = str.substring(i + 27, i2);
        var val = eval(exp)
        return hd + val + tl;
}
```

Наконец, нам нужно добавить вызов `simplifyCSSExpression` при загрузке страницы.

```
if (/msie/i.test(navigator.userAgent) && window.attachEvent != null) {
  window.attachEvent("onload", function () {
        simplifyCSSExpression();
  });
}
```

Все так просто? Нет, еще проще

А еще можно использовать свойства `currentStyle` (доступное для чтения) и `runtimeStyle` (доступное для записи), чтобы переопределять само стилевое свойство при его объявлении (звучит несколько сложно, не так ли?). На самом деле, все чрезвычайно просто. Применительно к нашему примеру мы должны будем написать:

```
#myDiv {
        border: 10px solid Red;
        width:  expression(runtimeStyle.width = (ieBox ? '100px' : '80px'));
}
```

Например, можно дописать исправление всплывания `alt` вместо `title` для картинок:

```
img {
        behavior: expression( (alt&&!title) ? title = '' : '',
                runtimeStyle.behavior = 'none'
        )
}
```

Или прозрачность через фильтр:

```
.button1 { opacity: .1 }
```

```
.button2 { opacity: .2 }
.button3 { opacity: .3 }
.button4 { opacity: .4 }
.button1, .button2, .button3, .button4
        { filter: expression( runtimeStyle.filter =
          'alpha(opacity='+currentStyle.opacity*100+')' ) }
```

Таким образом, наше выражение быстро применяется при загрузке страницы и последующем создании новых узлов скриптом. Такой способ оптимизации подходит только для «статичных» элементов, которым не нужно менять свое отображение динамически. Изменение родительского класса, равнение по высоте окна и эмуляция `position: static` — все это проблемные участки оптимизации. Лучше их не оптимизировать, а использовать пореже.

Еще одним проблемным местом, на мой взгляд, является общее выполнение скриптов при `onresize`. Ну и еще серьезный совет: используйте CSS-выражения по минимуму. Лучше всего будет, если они вообще не встретятся у на сайте.

6.2. Что лучше, `id` или `class`?

Далее давайте рассмотрим, как использование `id` или `class` влияет на скорость отображения страницы в браузере (сейчас речь не идет о множественном использовании одинаковых `id` — это и так запрещено спецификацией). Если элемент на странице встречается единственный раз (например, шапка или подвал), то мы можем с равным успехом использовать `id` и `class` для его стилизации.

Методика. Размер файлов

Естественно, что скорость работы одиночного CSS-правила весьма высока, и даже десятки и сотни их не должны заметно замедлить работу браузеров. Поэтому нужно изучать работу нескольких тысяч правил, иначе точность результатов будет весьма невысока. Использовать JavaScript для генерации HTML/CSS-кода не представляется разумным, ибо тогда придется учитывать еще и скорость работы JavaScript-движка в браузерах, и в итоге эксперимент будет недостаточно чистым.

В результате проведенного исследования были сгенерированы статичные файлы (порядка 300 Кб каждый), которые содержали достаточное число различных CSS-селекторов. Это «достаточное» число подбиралось по нескольким параметрам, в том числе: размер файла и скорость работы HTML/CSS-кода в браузерах (она должна быть достаточно низкой, чтобы файлы в несколько сотен Кб уже заметно тормозили при открытии).

Итоговые файлы содержали по 4096 объявлений различных CSS-классов (или различных идентификаторов), HTML-код содержал соответствующее количество блоков, у каждого свой индивидуальный класс (или идентификатор). Дополнительно проверялась скорость работы с простым наследованием узлов (`div p`, CSS1) и селектор для выбора потомка первого уровня (`div>p`, CSS2).

Время открытия страницы

При тестировании браузеров нужно было, во-первых, открыть на клиенте соответствующую данному случаю страницу, а также как-то отследить время на

отображение конкретно HTML/CSS-части (понятно, что оно не совпадает со временем открытия всей страницы, которое еще содержит некоторые накладные расходы).

Для этого была использована простая техника: перед объявлением CSS-блока запоминается текущая метка времени, а после окончания HTML-блока, который должен отобразиться, запомненная метка вычитается из текущей. Таким образом, мы получаем (в идеале) время на отработку данных CSS-правил и кода, которыми ими описывается, на клиенте (плюс, возможно, еще какие-то более-менее постоянные расходы, которые нивелируются, если брать относительный, а не абсолютный выигрыш).

Конечно, каждую тестовую страницу можно было подгружать в невидимом `iframe` или даже AJAX-запросом. Но ведь мы хотим узнать, фактически, скорость рендеринга браузером CSS-правил и соответствующего кода, а это время будет расходоваться только при отображении страницы в окне браузера. Поэтому подгружаемую страницу нужно отображать на экране (по возможности, максимального размера), чтобы отследить имеющуюся разницу.

Результаты

Ниже приведена большая таблица с результатами тестов, которые заключаются в среднем времени отображения страницы для различных вариаций селекторов и разных браузеров. Выделено время, меньшее по сравнению с аналогом. Хочется подчеркнуть, что имеет смысл только относительное ускорение использования одних типов селекторов относительно других в пределах одного браузера. Все времена даны в миллисекундах.

Сравнивать абсолютные значения в рамках данного эксперимента не представляется возможным, ибо каждому браузеру дополнительно нужно было расположить на странице несколько тысяч «плавающих» блоков с заданными размерами (`float:left; width:20px; height:20px`, фон для которых и задавался). Эта задача не имеет ничего общего со скоростью работы CSS-селекторов, но может отнимать существенное время у браузера на подготовку изображения страницы на экране (как видно, например, для Opera).

	Firefox 2	**Opera 9.5**	**Safari 3**	**IE 7**	**IE 6**	**IE 5.5**
p.class	308	**5887**	237	82	72	**145**
.class	**219**	6456	**225**	**78**	**70**	149
p#id	349	**7377**	338	91	87	**156**
#id	**214**	7427	**220**	**83**	**84**	159
div>p.class	**519**	**9412**	**247**	97	84	**158**
div>.class	836	12886	257	**95**	**81**	159
div>p#id	**549**	**10299**	247	**105**	92	172
div>#id	858	15172	**242**	113	**91**	**169**
div p.class	827	**10706**	256	97	**84**	161

	Firefox 2	Opera 9.5	Safari 3	IE 7	IE 6	IE 5.5
div .class	**505**	15864	**247**	**95**	86	**160**
div p#id	**772**	**11952**	**247**	**108**	99	177
div #id	948	13306	255	**108**	**95**	**173**
div.div p.class	**1001**	**10519**	263	111	94	**165**
div.div .class	1099	18742	**253**	**105**	**92**	166
div.div p#id	**1161**	**10989**	266	117	**95**	**181**
div.div #id	1247	15816	**256**	**114**	100	187

Таблица 6. Для каждого селектора приведено время, затраченное браузером на отображение тестового случая с этим селектором в миллисекундах

Выводы

Единственный вывод, который можно с твердостью сделать, — это преимущество использования `#id` перед `p#id` (средневзвешенное по всем браузерам для Рунета получается 9%). Также можно с некоторой уверенностью говорить об использовании `.class` вместо `p.class` (10%). Еще стоит обратить внимание на существенное (до 2,5 раз) ускорение при переходе от CSS1-селекторов к CSS2 (от `div p` к `div>p`, в тех браузерах, которые это поддерживают). Дополнительно нужно, наверное, отметить, что выборка элементов по классу работает, в целом, быстрее, чем по идентификатору (11%).

Все остальные выводы уже можно делать, анализируя данные из таблицы. IE всех версий стабильно выполнял все тесты примерно на одном уровне, а при его текущем доминировании оптимизация должна идти, в первую очередь, для него.

6.3. Влияние семантики и DOM-дерева

Давайте рассмотрим сейчас другой вопрос, а именно: как быстро браузер создает DOM-дерево в зависимости от наличия в нем элементов с `id` или `class`?

Для этого мы подготовим 3 набора HTML-файлов. Первый будет содержать 10000 элементов, у которых только часть будет иметь `id` (количество именованных элементов варьируется от 50 до 10000, нужно для оценки влияния DOM-дерева). Второй HTML-файл практически идентичен первому, только элементы вместо `id` имеют атрибут `class`. В третьем наборе в DOM-дереве оставим только элементы с `id` (т.е. будем изменять само число элементов от 50 до 10000). Все измерения запустим в скрытом `iframe`, чтобы избежать отрисовки загружаемой страницы на экране.

Графики влияния DOM-дерева

Ниже приведены разделенные графики по средневзвешенному значению (естественно, основную роль играет Internet Explorer, ибо сейчас им пользуются от 50% до 70% посетителей наших сайтов) создания документа:

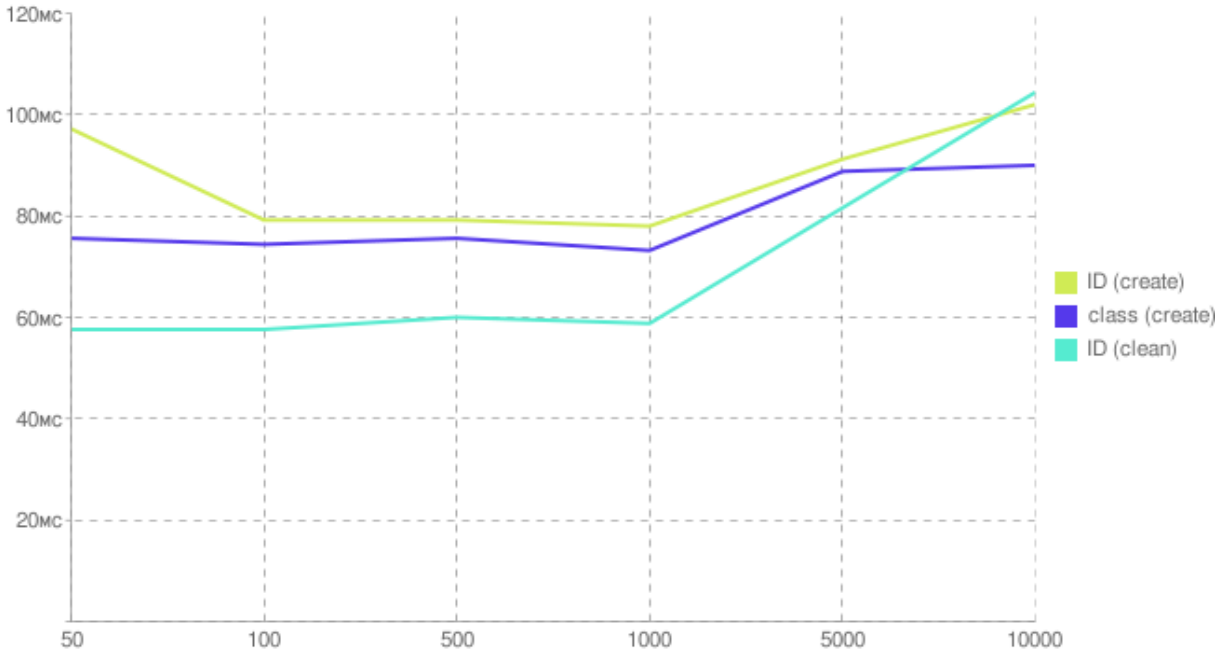


Рис. 30. Скорость создания документа, средневзвешено по всем браузерам

и график для времени выборки одного элемента из дерева (по идентификатору) при наличии в этом же дереве различного числа элементов с идентификаторами. `ID (10000 get)` показывает время на 10000 итераций проведения такой выборки, `ID clean (10000 get)` — то же самое, но в дереве идентификаторы присвоены не всем элементам, а только указанному числу:

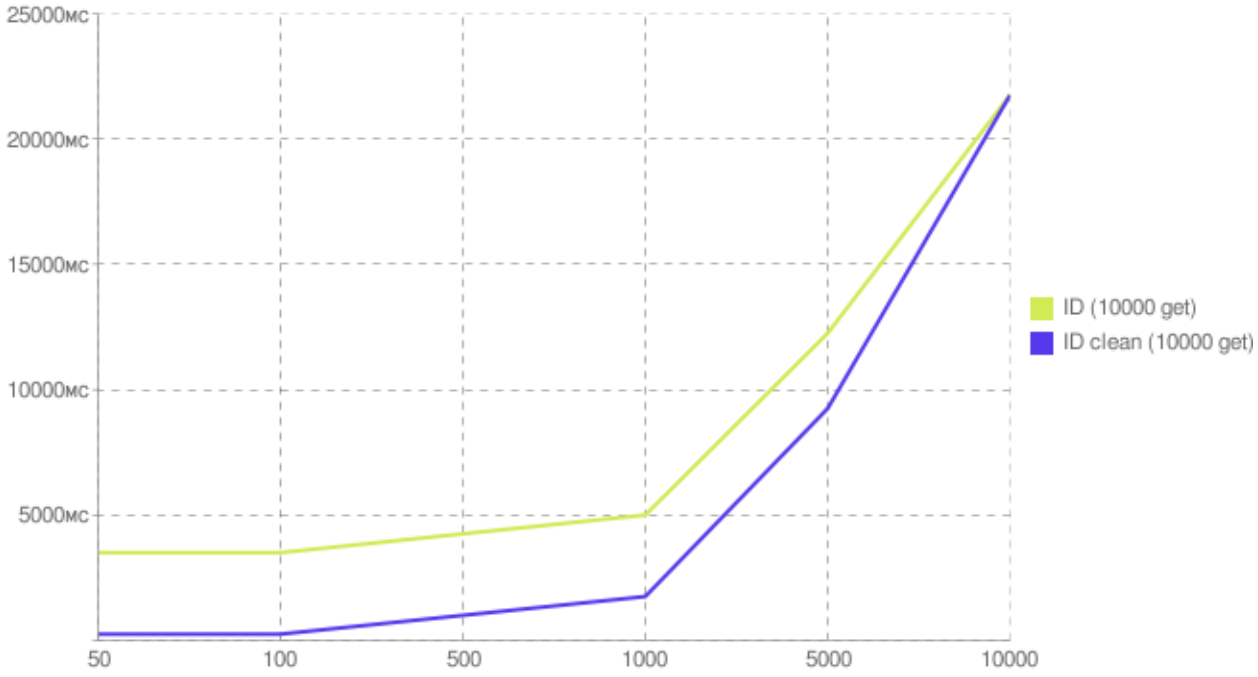


Рис. 31. Скорость выбора элемента, средневзвешено по всем браузерам

Выводы по DOM-дереву

По графику средневзвешенных значений хорошо видно, что, при прочих равных условиях, создание документа с `class` обходится меньшей кровью, чем с `id` (в общем случае, от 2% до 10% выигрыша). Если принять во внимание, что `.class`-селекторы отрабатывают быстрее, чем `#id`, на те же 10%, то общий выигрыш при использовании в документе классов перед идентификаторами составит порядка 15%. В абсолютном значении эти цифры не так велики: для Centrino Duo 1.7 получается цифра примерно в 0,0085мс на 1 идентификатор (в среднем, 3 CSS-правила и 1 употребление).

Для документа со 100 элементами выигрыш может составить почти 1 мс, для документа с 1000 — 8,5 мс! Стоит заметить, что средняя страница в интернете имеет 500–1000 элементов. Проверить, сколько элементов на странице, можно, просто запустив следующий код в адресной строке браузера на какой-либо открытой странице:

```
javascript:alert(document.getElementsByTagName('*').length)
```

Естественно, что приведенные цифры — это уже то, за что можно побороться.

В случае больших веб-приложений задержка в 100 мс (при числе элементов более 10000) уже может оказаться критичной. Ее можно и нужно уменьшать (наряду с другими «узкими» местами для JavaScript, о которых речь пойдет в седьмой главе).

Что и требовалось доказать: значительную нагрузку составляет именно создание DOM-дерева в документе. В целом, на эту операцию уходит от 70% всего времени рендеринга (т.е. наибольшая экономия достигается за счет минимизации размера дерева).

На скорость вычисления одного элемента по идентификатору, как ни странно, наибольшее влияние оказывает опять-таки DOM-дерево, а не количество таких элементов. Даже при 1000 элементов с `id` более половины временных издержек можно урезать, если просто сократить общее число элементов (особенно хорошо это заметно для IE).

В целом же, основных советов два: стоит уменьшать DOM-дерево и использовать **id** только в случае действительной необходимости.

Семантическое DOM-дерево

Логическим продолжением уже проведенных исследований CSS/DOM-производительности браузеров стало рассмотрение зависимости времени создания документа от числа тегов (узлов дерева). Раздельно были проанализированы случаи, когда DOM-дерево является чисто линейным (все `div` лежали прямо внутри `body`), когда оно разветвленное (ветки по 10 вложенных `div` наращивались внутри `body`) и когда вместо ветки из `div` используются некоторая семантическая конструкция, а именно:.

```
<div>
        <ul>
                <li></li>
                <li></li>
        </ul>
        <p>
                <a href="#">
                        <em></em>
```

```
                </a>
                <span></span>
        </p>
        <blockquote></blockquote>
        <h1></h1>
</div>
```

В итоге мы получили примерно следующую картину:

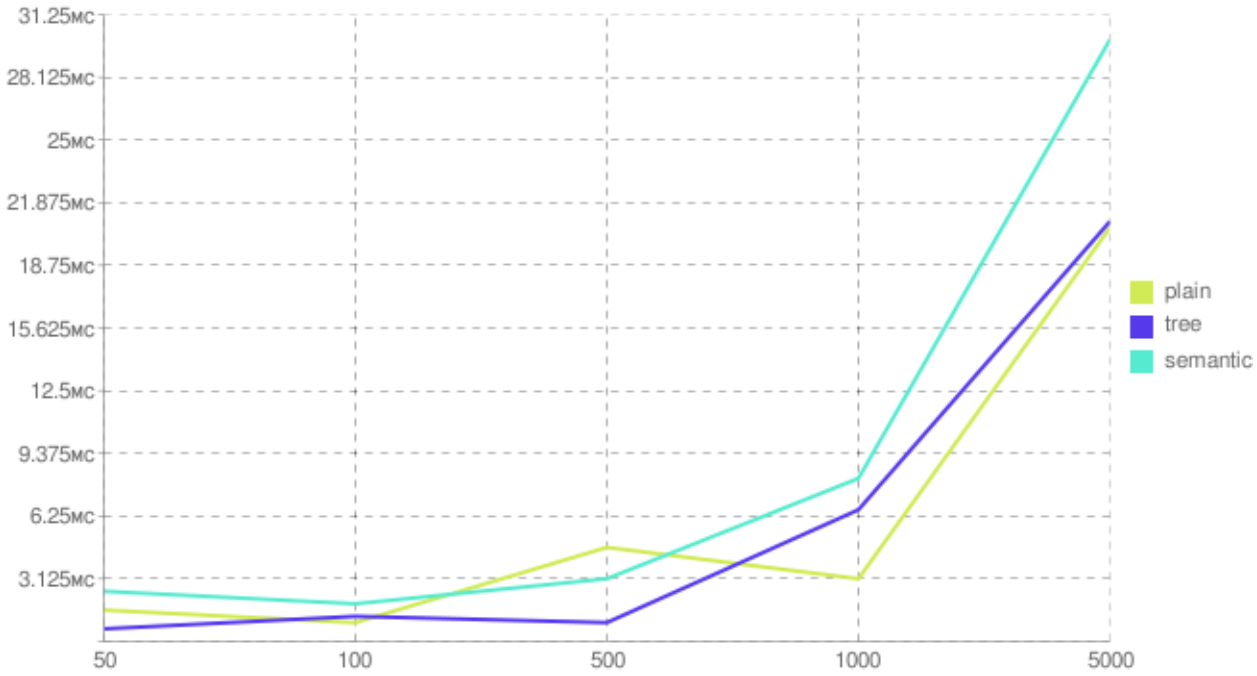


Рис. 32. Средневзвешенное значение времени создания документа от числа узлов в DOM-дереве

Что быстрее?

Да, очевидно, что размер DOM-дерева влияет на скорость загрузки страницы. Одной из целей данного исследования было показать, как именно влияет (в конкретных числах). Средний размер страницы — 700–1000 элементов. Они загрузятся в дерево сравнительно быстро (3–7 мс, без учета инициализации самого документа, которая занимает 30–50 мс). Дальше время загрузки растет линейно, но все равно можно нарваться на нежелательные «тормоза», добавив несколько тысяч «скрытых» элементов или избыточной семантики.

Различия между линейной и древовидной структурой находятся в пределах погрешности, однако, семантическое дерево оказалось самым медленным (чуть ли не на 50%). Но в любом случае, уменьшение размера DOM-дерева всегда является наиболее приоритетным направлением.

Конечной же целью всех экспериментов было установить, есть ли различие в отображении `HTML 4.0 Transitional` и `XHTML 1.0 Strict` документов и какова реальная польза от использования советов по оптимизации CSS-кода (имеется в виду синтаксис селекторов). Об этом рассказывается в следующем разделе.

Методика для `DOCTYPE`

Была аккуратно выкачана главная страница Яндекса (она уже хорошо оптимизирована с точки зрения производительности, поэтому проводить эксперименты на ней — весьма показательно). Из нее были удалены все ссылки на картинки и внешние скрипты, чтобы не создавать дополнительных задержек. В дальнейшем полученная «чистая» версия препарировалась различными способами.

Далее была добавлена стандартная схема измерения загрузки (рендеринга) страницы: время в самом начале `head` засекается и затем отнимается от времени срабатывания события `window.onload` (в данном случае это равносильно окончанию рендеринга HTML-кода). Браузеры друг с другом не сравнивались (в частности, из-за поведения Safari, который не совсем честно сообщает об этом событии), сравнивались только различные варианты.

В качестве второй версии страницы бралось приведение ее к валидному `XHTML Strict` виду. Верстка при этом немного изменилась, но, в целом, результат получился весьма убедительный. Комментарии и прочий мусор (типа пустых `onclick=""`, о них речь чуть дальше) были сохранены. Размер, действительно, несколько увеличился (на 1 Кб — несжатая версия и на 150 байтов — сжатая).

Далее в третьей версии уже были убраны все `onclick`. Больше ничего со страницей не делалось. Ожиданий данная версия не оправдала (только Safari показал значимые отличия от предыдущего варианта, хотя было выкошено 83 пустых `onclick`).

В четвертом варианте — венце оптимизационных (в отношении CSS/HTML-кода) действий — использование было `id` сведено к минимуму, все селекторы для `class` задавались без тегов. Также были убраны все комментарии из кода.

Результаты оптимизации

В таблице приведены результаты для основных браузеров (август 2008): размер каждого вариант в байтах и время его загрузки. Времена приведены в миллисекундах.

	Size (b)	Gzip (b)	IE6	IE7	IE8b	Firefox 2	Firefox 3	Opera 9.5	Safari 3.1
1	26275	8845	56	80	76	130	127	142	33
2	27173	8993	60	75	320	127	118	148	27
3	26260	8949	61	75	320	131	116	141	23
4	26153	8862	55	73	306	94	102	178	28

Таблица 7. Для каждого варианта приведен размер и время его отображения в миллисекундах

«Экономия на спичках»?

В результате тестов удалось показать, что валидный XHTML не медленнее (а даже, местами, быстрее), чем HTML. И оптимизация реально играет роль (возможно ускорение

загрузки HTML главной страницы Яндекса на 10–12%). Если говорить о конкретных примерах, то на 100 Кб/с канале с включенным сжатием в FF3 оптимизированный вариант загрузится на 9 мс быстрее. Для более быстрого канала или более медленного компьютера отличие будет еще разительнее.

Естественно, это все «копейки» для обычных пользователей (+/-50мс — это совершенно не критично). Однако если речь идет про «экономию на спичках», когда нам важен каждый запрос к серверу и каждая миллисекунда, то тут уже стоит задуматься — что же все-таки использовать.

И что важнее всего, если правильно расставить акценты, то загрузку XHTML можно сделать и быстрее, чем HTML. Различие в размере файлов оказалось, в итоге, минимальным (26153 против 26275 в несжатом варианте, и 8862 против 8845 в сжатом, т.е. меньше 0,5%). При этом в IE7 наблюдается ускорение отображения страницы на 7 мс (от 60–80 мс при загрузке страницы). Это, в среднем, дает 10% выигрыша в скорости. FF3 ведет себя похожим образом (но выигрыш в скорости 20% (25 мс от 127 мс)). Все остальные браузеры показали отличие в загрузке на 2–3 мс, что укладывается в погрешность; Opera была медленнее, что подтверждается предыдущими тестами.

В целом, в свете тотального распространения мобильных браузеров с их маломощными процессорами, такой вид оптимизации выглядит весьма перспективно.

6.4. Ни в коем случае не reflow!

В CSS-движке браузеров существует несколько операций, затрагивающих изменение картинки на экране браузера. В предыдущих тестах было рассмотрено начальное создание документа и способы ускорения это процесса. Однако все вышеприведенные советы в равной степени относятся и к изменению уже отрисованной картинки при каких-либо операциях с документом.

HTML-элемент в документе может быть скрыт с помощью JavaScript или CSS-свойства `display`. Дублировать с помощью JavaScript логику, заложенную в CSS-движке, достаточно сложно и не всегда нужно. Проще запросить `offsetHeight` объекта (если оно равно 0, значит, элемент скрыт). Проще-то оно, конечно, проще, вот только какой ценой?

Для проверки видимости элемента принято проверять значение стиля `display` или наличие класса `hide`. Когда мы пишем функцию скрытия/отображения сами, то знаем, какое значение стиля `display` у объекта по умолчанию или какой класс какому состоянию соответствует. Однако универсальная (библиотечная) функция знать об этом не может.

`offsetHeight` и `style.display`

Проведем тестирование скорости вычисления значений `offsetHeight` и `style.display`.

Для удобства профайлинга вынесем доступ к этим значениям в отдельные функции:

```
function fnOffset(el)
{
        return !!el.offsetHeight;
}
```

```
function fnStyle(el)
{
        return el.style.display=='none';
}
```

где `el` — тестовый контейнер.

Проведем тест на тысяче итераций, на каждой итерации будем добавлять в тестовый контейнер элемент `<span>`. Проверим время, затрачиваемое на добавление тысячи элементов, без вызова тестовых функций тест `clean`. Проведем тестирование во всех браузерах, замеряя время следующим способом:

```
var time_start=new Date().getTime();
/* ... тест ... */
var time_stop=new Date().getTime();
var time_taken=time_stop-time_start;
```

где `time_taken` — это время, затраченное на тест, в миллисекундах.

	IE sp62	IE8b	Firefox 2.0.0.12	Opera 9.22	Safari 3.04b
clean	128	153	15	15	16
offsetHeight	23500	10624	4453	4453	5140
style.display	171	209	56	56	34
height vs. style	140 раз	50 раз	80 раз	80 раз	150 раз

Таблица 8. Результаты выполнения тестов по определению видимости элементов. Времена приведены в миллисекундах. Взято среднее значение серии из 5 тестов

Судя по результатам тестов, доступ к `offsetHeight` медленнее в 50–150 раз.

Получается, что по отдельности и `offsetHeight`, и добавление элементов работают быстро, а вместе — очень медленно. Как же так?

Немного теории

Reflow — это процесс рекурсивного обхода ветви дерева DOM, вычисляющий геометрию элементов и их положение относительно родителя. Начало обхода — изменившийся элемент, но возможно и распространение в обратном порядке. Существуют следующие типы reflow:

- начальный — первичное отображение дерева;
- инкрементный — возникает при изменениях в DOM;
- изменение размеров;
- изменение стилей;
- «грязный» — объединение нескольких инкрементных reflow, имеющих общего родителя.

Reflow делятся на неотложные (изменение размеров окна или изменение шрифта документа) и асинхронные, которые могут быть отложены и объединены впоследствии.

При манипулировании DOM происходят инкрементные reflow, которые браузер откладывает до конца выполнения скрипта. Однако, исходя из определения reflow, «измерение» элемента вынудит браузер выполнить отложенные reflow. Т.к. возможно распространение снизу вверх, то выполняются все reflow, даже если измеряемый элемент принадлежит к неизменившейся ветви.

Операции reflow очень ресурсоемки и являются одной из причин замедления работы веб-приложений.

Если судить по тесту clean, все браузеры хорошо справляются с кэшированием многочисленных reflow. Однако, запрашивая `offsetHeight`, мы «измеряем» элемент, что вынуждает браузер выполнить отложенные reflow. Таким образом, браузер делает тысячу reflow в одном случае и только один — в другом.

Замечание: в Opera reflow выполняется еще и по таймеру, что, однако, не мешает ей пройти тест быстрее остальных браузеров. Благодаря этому в Opera виден ход тестов — появляются добавляемые звездочки. Такое поведение оправдано, т.к. вызывает у пользователя ощущение большей скорости браузера.

Использование Computed Style

Что же показало тестирование? По меньшей мере, некорректно сравнивать универсальный (`offsetHeight`) и частный (`style.display`) случаи. Тестирование показало, что за универсальность надо платить. Если все-таки хочется универсальности, то можно предложить другой подход: определение `Computed Style` — конечного стиля элемента (после всех CSS преобразований).

```
getStyle = function()
{
        var view = document.defaultView;

        if(view && view.getComputedStyle)
                return function getStyle(el,property)
                {
                        return view.getComputedStyle(el,null)[property] ||
                                el.style[property];
                };

        return function getStyle(el,property)
        {
                return el.currentStyle && el.currentStyle[property] ||
                        el.style[property];
        };
}();
```

Проведем тестирование этого способа и сведем все результаты в таблицу.

IE sp62	Firefox 2.0.0.12	Opera 9.22	Safari 3.04b

	IE sp62	Firefox 2.0.0.12	Opera 9.22	Safari 3.04b
offsetHeight	23500	4453	4453	5140
style.display	171	56	56	34
getStyle			5219	5318

Таблица 9. Результаты выполнения функции getStyle. Времена приведены в миллисекундах

Во-первых, для IE и Firefox (наиболее популярных браузеров) функция эта работает некорректно (в общем случае возвращает неверные данные). Во-вторых, работает она чуть ли не медленнее, чем `offsetHeight`.

Вообще говоря, рекомендуется не пользоваться такими универсальными функциями (`getStyle` есть практически в каждой JavaScript-библиотеке), а реализовывать необходимую функциональность в каждом конкретном случае. Ведь если мы договоримся, что скрытые элементы должны иметь класс `hide`, то все сведется к определению наличия этого класса у элемента или его родителей.

Оптимизация: определение класса hide

Давайте подробнее остановимся на предложенном мной решении. Предлагаю следующую реализацию:

```
function isHidden(el)
{
        var p=el;
        var b=document.body;
        var re=/(^|\s)hide($|\s)/;
        while(p && p!=b && !re.test(p.className))
                p=p.parentNode;
        return !!p && p!=b;
}
```

Предполагается, что корневые элементы DOM скрывать не имеет смысла и поэтому проверки ведутся только до `document.body`.

Предложенное решение явно не спустит лавину reflow, так как никаких вычислений и измерений не проводится. Однако немного смущает проход до корня документа: что же будет при большой вложенности элементов? Давайте проверим. Тест `isHidden` проводится для вложенности 2 (`document.body / test_div`), а тест `isHidden2` — для вложенности 10 (`document.body / div * 8 / test_div`).

	IE sp62	Firefox 2.0.0.12	Opera 9.22	Safari 3.04b
offsetHeight	23500	10624	4453	5140
isHidden	231	351	70	71

	IE sp62	Firefox 2.0.0.12	Opera 9.22	Safari 3.04b
isHidden2	370	792	212	118
offsetHeight vs. isHidden	102 раза	30 раз	73 раза	92 раза

Таблица 10. Результаты выполнения функции isHidden. Времена приведены в миллисекундах

Как показывают тесты, даже при большой вложенности падение скорости невелико. Таким образом, мы получили универсальное решение, которое быстрее доступа к `offsetHeight` в 30–100 раз.

Заключение

Все вышеприведенные мысли предназначены не столько для решения проблемы выяснения видимости элемента в общем случае, сколько для объяснения одного из наиболее часто встречающихся узких мест взаимодействия с DOM и детального разбора методов оптимизации. В ходе тестов был намеренно воспроизведен наихудший случай. В реальных ситуациях такой прирост скорости получится только при использовании в анимации. Однако понимание причин и механизма reflow позволяет писать более оптимальный код.

В качестве послесловия: стили или классы?

В заключении давайте затронем еще несколько оптимизационных моментов, связанных с отображением HTML-страницы на экране браузера. Пусть в нашем документе есть элементы, у которых нужно поменять цвет, фон или что-нибудь еще, относящееся к стилям. Например, подсветить строки таблицы при наведении мыши или пометить их, если выбрана соответствующая галочка в форме.

Существует два способа это сделать: при помощи стилей или установив цвет (или фон) напрямую из JavaScript. Для начала немного кода — с помощью класса:

```
var items = el.getElementsByTagName('li');

for (var i = 0; i < 1000; i++) {
        items[i].className = 'selected'
}
```

И с помошью стилей:

```
var items = el.getElementsByTagName('li');

for (var i = 0; i < 1000; i++) {
        items[i].style.backgroundColor = '#007f00';
        items[i].style.color = '#ff0000';
}
```

Результаты простые и понятные:

Метод	IE 6	IE 7	Firefox 1.5	Firefox 2.0	Opera 9
`element.className`	512	187	291	203	47
`element.style.color`	1709	422	725	547	282

Таблица 11. Применение стилей и классов к элементам

Перерисовка страницы

Однако когда мы изменяем класс элемента, код отрабатывает значительно быстрее, но вот страница обновляется медленно. Это все из-за того, что изменение свойства `className` не перерисовывает страницу мгновенно, вместо этого браузер просто помещает событие обновления в очередь reflow. Отсюда и огромная скорость, казалось бы, более сложной процедуры. А что по поводу `:hover`? К сожалению, `:hover` работает только для ссылок в Internet Explorer 6. Поэтому в любом случае придется пользоваться какой-то его эмуляцией.

Из всего вышеперечисленного можно сделать два ключевых вывода.

- Используйте `className` везде, где это возможно. Это дает больше гибкости и контроля над внешним видом сайта.
- Если на странице много элементов в контейнере и необходимо построить очень быстрый интерфейс, стоит устанавливать стили напрямую через свойство `style`.

Групповое изменение стилей

Если мы уже задумались над максимально быстрым изменением интерфейса нашего веб-приложения (отрисовке) через свойство `style`, то стоит иметь в виду следующий момент. Мы можем изменять свойство `cssText`, которое отвечает за компилируемые стили элемента:

```
element.style.cssText = "display:block;width:auto;height:100px;...";
```

Таким образом, мы можем дополнительно ускорить наше единовременное обновление стилей у элемента, потому что произойдет всего одно присвоение свойств и всего один reflow (и он случится сразу же после изменения этого свойства, а не в отложенном режиме).

Два слова о таблицах

Таблицы замечательно подходят для организации информации. Однако если в HTML-документе встречается таблица, то браузеру приходится пробежаться по ней дважды: в первый раз — чтобы выбрать все элементы, рассчитать их взаимные размеры, и чтобы отрисовать их все — во второй раз. Если на странице выводятся большие массивы данных (например, параметры товаров или статистические данные), то гораздо быстрее будет визуализировать такие таблицы в один проход.

Давайте рассмотрим, как можно помочь браузерам в их нелегком труде. Следующие действия позволят начать отображение таблицы еще до того, как будет получена вся информация о ней.

- Необходимо установить для `table` CSS-атрибут `table-layout` в значение `fixed`.
- Затем явно определить объекты `col` для каждого столбца.
- И установить для каждого элемента `col` атрибут `width`.

В качестве примера можно привести такой фрагмент кода:

```
<table style="table-layout: fixed">
<!-- первый столбец имеет ширину 100 пикселей -->
<col width="100"></col>
<!-- второй — 200 -->
<col width="200"></col>
<!-- третий и четвертый — по 250 -->
<col width="250"></col><col width="250"></col>
<thead>...</thead>
<tfoot>...</tfoot>
<tbody>...</tbody>
</table>
```

Глава 7. Оптимизация JavaScript

7.1. Кроссбраузерный `window.onload`

Отложенная загрузка скриптов волнует общественность уже давно, почти 10 лет, — атрибут `defer`, призванный ее обеспечить, был добавлен в спецификацию еще в прошлом столетии. Однако проблема так и остается актуальной.

Событие `window.onload` используется программистами для старта их веб-приложения. Это может быть что-то довольно простое, например, выпадающее меню, а может быть и совсем сложное — скажем, запуск почтового приложения. Суть проблемы заключается в том, что событие `onload` срабатывает только после того, как загрузится вся страница (включая все картинки и другое бинарное содержимое). Если на странице много картинок, то можно заметить значительную задержку между загрузкой страницы и тем моментом, когда она начнет фактически работать. На самом деле, нам нужен только способ определить, когда DOM полностью загрузится, чтобы не ждать еще и загрузку картинок и других элементов оформления.

Firefox впереди планеты всей

В Firefox есть событие специально для этих целей: `DOMContentLoaded`. Следующий образец кода выполняет как раз то, что нам нужно, в Mozilla-подобных браузерах (а также в Opera 9 и старше):

```
// для Firefox
if (document.addEventListener) {
        document.addEventListener("DOMContentLoaded", init, false);
}
```

A Internet Explorer?

IE поддерживает замечательный атрибут для тега `<script>`: `defer`. Присутствие этого атрибута указывает IE, что загрузку скрипта нужно отложить до тех пор, пока не загрузится DOM. Однако это работает только для внешних скриптов. Следует также заметить, что этот атрибут нельзя выставлять, используя другой скрипт. Это означает, что нельзя создать `<script>` с этим атрибутом, используя DOM-методы, — атрибут будет просто проигнорирован.

Используя этот удобный атрибут, можно создать мини-скрипт, который и будет вызывать наш обработчик `onload`:

```
<script defer src="ie_onload.js" type="text/javascript"></script>
```

Содержание этого внешнего скрипта будет состоять только из одной строчки кода:

```
init();
```

Условные комментарии

Есть некоторая проблема с этим подходом. Другие браузеры проигнорируют атрибут `defer` и загрузят этот скрипт сразу же. Существует несколько способов, как можно с этим

побороться. Можно воспользоваться условными комментариями, чтобы скрыть «отложенный» скрипт:

```
<!--[if IE]><script defer="defer" src="ie_onload.js"></script><![endif]-->
```

IE также поддерживает условную компиляцию. Следующий код будет JavaScript-эквивалентом для заявленного выше HTML-кода:

```
// для Internet Explorer
/*@cc_on @*/
/*@if (@_win32)
        document.write("<script defer=\"defer\"
                src=\"ie_onload.js\"><\/script>");
/*@end @*/
```

Все так просто?

И, конечно же, нам нужно обеспечить поддержку для остальных браузеров. У нас есть только один выход — стандартное событие `window.onload`:

```
// для остальных браузеров
window.onload = init;
```

Двойное выполнение

Остается одна маленькая неприятность (кто сказал, что будет легко?). Поскольку мы устанавливаем событие `onload` для всех (оставшихся) браузеров, то `init` сработает дважды в IE и Firefox. Чтобы это обойти, нам нужно сообщить функции, что она должна выполняться только один раз. Итак, наш метод `init` будет выглядеть примерно так:

```
function init() {
        // выходим, если функция уже выполнялась
        if (arguments.callee.done) return;

        // устанавливаем флаг, чтобы функция не исполнялась дважды
        arguments.callee.done = true;

        // что-нибудь делаем
};
```

Стоит также рассмотреть выставление глобальной переменной `READY` в данном случае. Ведь иногда знать, что страница начала функционировать, может потребоваться не только одной функции `init`.

Избавляемся от внешнего файла

У описанного решения существует пара минусов:

- Для IE нам требуется внешний JavaScript-файл.
- Не поддерживается Safari (Opera 9 поддерживает `DOMContentLoaded`).

Однако есть решение и для Internet Explorer, которое не зависит от внешних файлов (к сожалениию, на данный момент вызывает предупреждение безопасности в IE7 при использовании защищенного соединения):

```
// для Internet Explorer (используем условную компиляцию)
/*@cc_on @*/
/*@if (@_win32)
document.write("<script id=\"__ie_onload\" defer=\"defer\"
src=\"javascript:void(0)\">
<\/script>");
var script = document.getElementById("__ie_onload");
script.onreadystatechange = function() {
        if (this.readyState == "complete") {
                init(); // вызываем обработчик для onload
        }
};
/*@end @*/
```

И для Safari!

```
if (/WebKit/i.test(navigator.userAgent)) { // условие для Safari
        var _timer = setInterval(function() {
                if (/loaded|complete/.test(document.readyState)) {
                        clearInterval(_timer);
                        init(); // вызываем обработчик для onload
                }
        }, 10);
}
```

Полное решение

```
function init() {
        // выходим, если функция уже выполнялась
        if (arguments.callee.done) return;

        // устанавливаем флаг, чтобы функция не исполнялась дважды
        arguments.callee.done = true;

        // что-нибудь делаем
};

/* для Mozilla/Firefox/Opera 9 */
if (document.addEventListener) {
        document.addEventListener("DOMContentLoaded", init, false);
}

/* для Internet Explorer */
/*@cc_on @*/
/*@if (@_win32)
        document.write("<script id=\"__ie_onload\"
                defer=\"defer\" src=\"javascript:void(0)\">
<\/script>");
var script = document.getElementById("__ie_onload");
script.onreadystatechange = function() {
        if (this.readyState == "complete") {
                init(); // вызываем обработчик для onload
        }
};
/*@end @*/

/* для Safari */
if (/WebKit/i.test(navigator.userAgent)) { // условие для Safari
        var _timer = setInterval(function() {
                if (/loaded|complete/.test(document.readyState)) {
                        clearInterval(_timer);
```

```
                    init(); // вызываем обработчик для onload
                }
        }, 10);
}

/* для остальных браузеров */
window.onload = init;
```

Неблокирующая загрузка JavaScript

Внешние JavaScript-файлы блокируют загрузку страницы и сильно влияют на ее производительность, но существует достаточно простой выход из этой ситуации: использовать динамические теги `<script>` и загружать скрипты параллельно, увеличивая тем самым скорость загрузки страницы и улучшая пользовательское восприятие.

Давайте сначала рассмотрим, в чем заключается проблема с загрузкой скриптов. Все сводится к тому, что браузер не может сказать, что находится внутри скрипта, не загрузив его полностью. Скрипт может содержать вызовы `document.write()`, которые изменяют DOM-дерево, или вообще `location.href`, что отправит пользователя на другую страницу. В последнем случае все компоненты, загруженные на предыдущей странице, могут оказаться ненужными. Чтобы предотвратить загрузки, которые могут оказаться лишними, браузеры сначала загружают, затем анализируют и исполняют каждый скрипт перед тем, как переходить к следующему файлу в очереди на загрузку. В результате каждый вызов скрипта на вашей странице блокирует процесс загрузки и оказывает негативное влияние на скорость загрузки.

Ниже приведена временная диаграмма, которая демонстрирует процесс загрузки медленных JavaScript-файлов. Загрузка скриптов блокирует параллельную загрузку фоновых картинок, которые идут сразу за ним:

Рис. 33. Временна́я диаграмма: блокирующее поведение JavaScript-файлов

Число загрузок с одного хоста

Над временно́й диаграммой (кроме блокирования картинок) нам также стоит задуматься о том, что картинки после скрипта загружаются только по две. Это происходит из-за ограничений на число файлов, которые могут быть загружено параллельно. В IE <= 7 и

Firefox 2 можно параллельно загружать только 2 файла (согласно HTTP 1.1 спецификации), но и в IE8, и в FF3 это число увеличено уже до 6.

Можно обойти это, используя несколько доменов для загрузки ваших файлов, потому что ограничение работает только на загрузку двух компонентов **с одного хоста** (более подробно проблема была рассмотрена в пятой главе). Однако стоит понимать, что **JavaScripts блокирует загрузки со всех хостов**. В действительности в приведенном выше примере скрипт располагается на другом домене, нежели картинки, однако по-прежнему блокирует их загрузку.

Если в силу каких-либо причин не удается воспользоваться преимуществами «отложенной» загрузки, то следует размещать вызовы на внешние файлы скриптов в низу страницы, прямо перед закрывающим тегом `</body>`. Это, в действительности, не ускорит загрузку страницы (скрипт по-прежнему придется загрузить в общем потоке), однако поможет сделать отрисовку страницы более быстрой. Пользователи почувствуют, что страница стала быстрее, если увидят какую-то визуальную отдачу в процессе загрузки.

Неблокирующие скрипты

На самом деле, существует довольно простое решение для устранения блокировки загрузки: нам нужно добавлять скрипты динамически, используя DOM-методы. Это как? Попробуем создать новый элемент `<script>` и прикрепить его к `<head>`:

```
var js = document.createElement('script');
js.src = 'myscript.js';
var head = document.getElementsByTagName('head')[0];
head.appendChild(js);
```

Ниже приведена диаграмма загрузки для нашего тестового случая, но для загрузки скриптов уже используется описанная технология. Заметим, что скрипты загружаются так же долго, но это не влияет на одновременную загрузку других компонентов:

Рис. 34. Временная диаграмма: неблокирующее поведение JavaScript-файлов

Как мы видим, файлы скриптов уже не блокируют загрузку, и браузер может начать работать с другими компонентами. Общее время загрузки при этом сократилось вдвое. Это будет работать только в том случае, если «динамические» скрипты не содержат

вызовов `document.write`. Если это не так, то все такие вызовы нужно будет заменить на `element.innerHTML` либо отказаться от использования этой техники.

Зависимости

Еще одна проблема, которая связана с динамическим подключением скриптов, заключается в разрешении зависимостей. Предположим, что у вас есть 3 скрипта и для `three.js` требуется функция из `one.js`. Как вы гарантируете работоспособность в этом случае?

Наиболее простым способом является объединение всех скриптов в одном файле. Это не только избавит нас от самой проблемы, но и улучшит производительность страницы, уменьшив число HTTP-запросов. Увеличения производительности может и не произойти в случае кэширования отдельных файлов и повторного посещения, но от описанной проблемы это точно вылечит.

Если все же приходится использовать несколько файлов, то можно добавить на подгружаемый тег обработчик события `onload` (это будет работать в Firefox) и `onreadystatechange` (это будет работать в IE). Наиболее кроссбраузерным подходом будет использование меток внутри каждого файла (например, `{filename}_loaded`) и проверка их с некоторым интервалом на соответствие `true`. Это обеспечит выполнение зависимостей.

Как показали исследования динамической загрузки скриптов, проблемы с зависимостями были отмечены на IE6- и Safari3 (под Windows). Из 10 скриптов, которые загружались параллельно (на самом деле, максимум они загружались по 6 в FF3, это связано с внутренними ограничениями браузера на число одновременных соединений с одним хостом), все 10 срабатывали в случайном порядке, начиная с 3–5, как раз в этих браузерах. В других браузерах (Opera 9.5, Firefox 2, Firefox 3, Internet Explorer 8) такого поведения отмечено не было.

Вопрос о динамической загрузке таблиц стилей подробно рассмотрен в четвертой главе (где речь шла о технике объединения картинок и последовательном отображении страницы в случае большого размера CSS-файлов). В случае комбинированной загрузки компоненты надо расположить в порядке приоритетности, потому что браузеры не смогут загрузить все сразу. А пользователи могут попытаться поработать с некоторыми «отложенными» возможностями максимально быстро, что и нужно предвосхитить.

А если по-другому?

Ниже приведено сравнение других методов для снятия блокировки с загрузки скриптов, но все они также обладают своими недостатками.

Метод	Недостатки
Используем атрибут `defer` тега `script`	Работает только в IE
Используем `document.write()` для подключения тега `script`	1. Неблокирующее поведение возможно только в IE (*через* `defer`) 2. Не рекомендуется широко использовать `document.write`

Метод	Недостатки
Используем `XMLHttpRequest` для получения тела скрипта, затем его исполняем через `eval()`	«`eval()` — зло» (долго выполняется, есть потенциальная угроза взлома при передаче «неправильных» данных)
Используем XHR-запрос для получения тела скрипта, затем создаем новый тег `script` и устанавливаем его содержание	Еще сложнее, чем предыдущий случай
Загрузка скрипта в `iframe`	1. Сложно 2. Издержки на создание iframe

Таблица 12. Сравнение методов «отложенной» загрузки JavaScript-файлов

В будущем

В будущие версии Safari и IE8 уже внесены изменения, которые коснулись способа загрузки скриптов. Идея заключается в том, чтобы загружать скрипты параллельно, но исполнять в той последовательности, в которой они находятся на странице. По всей видимости, в один прекрасный день проблема блокирующих скриптов при загрузке станет попросту неактуальной, потому что будет касаться только пользователей IE 7 и младше или Firefox 3 и младше. Пока же наиболее простым способом решения данной проблемы является применение динамического тега `<script>`.

7.2. Основы «ненавязчивого» JavaScript

Веб-разработка в последнее время существенно изменилась: мы прекратили смешивать содержание, оформление и функциональность, и, таким образом, менять дизайн и верстку по всему сайту стало легче — для этого нужно просто изменить таблицу стилей. А дополнительная функциональность может быть легко добавлена в уже существующий файл скриптов или за счет подключения нового.

Давайте рассмотрим, как научиться использовать все преимущества современных браузеров, не ограничивая при этом пользователей более старых версий.

Javascript: храним отдельно

Одна из сильных сторон Javascript заключается в том, что его можно поместить в отдельный файл. Как и для CSS это означает, что можно присвоить одну коллекцию функций для каждой страницы сайта, и если нам нужно изменить ее функциональность, можно сделать все это в одном документе, что предпочтительнее, чем перебирать все страницы и менять какой-то один участок кода на другой.

Например, мы можем подключить только один такой файл, который будет обеспечивать всю логику функционирования страницы в клиентском браузере.

```
<script type="text/javascript" src="scripts.js"></script>
```

Javascript — это расширение

Мы используем Javascript, только чтобы улучшить уже существующую функциональность, так как мы не можем на него полностью полагаться. Javascript может быть выключен или отфильтрован прокси-серверами и файерволами компаний, обеспечивающих безопасность. Мы никогда не можем принимать Javascript за данность.

Это не означает, что мы совсем не можем использовать Javascript, это только значит, что мы можем добавлять его лишь как дополнительную возможность. Страницы должны работать и при выключенном JavaScript — это одно из основных правил «ненавязчивого» JavaScript.

Давайте рассмотрим для примера следующий HTML-код:

```
<form action="/">
        <p><label for="login">Логин:</label>
        <input type="text" name="login" id="login"/></p>
        <p><label for="password">GПароль:</label>
        <input type="password" name="password" id="password"/></p>
        <p><input type="button" onclick="checkform()" value="Войти"/></p>
</form>
```

Если у пользователя отключен JavaScript (или он отрабатывает некорректно в силу особенностей различных браузеров), то зайти на сайт будет просто невозможно. И это только по причине неверного проектирования, а не ошибки на стороне сервера или неверного расположения элементов в дизайне.

Мы можем исправить этот фрагмент, заменив `button` на `submit` и добавив обработчик события `submit` для формы:

```
<p><input type="submit" value="Войти"/></p>
...
<script type=”text/javascript”>
        document.forms[0].onsubmit = checkform;
</script>
```

Доверять, но проверять

Множество ошибок в Javascript происходит просто потому, что разработчики слишком ленивы, чтобы проверить, доступен или нет данный объект или метод. Всегда при использовании какой-либо внешней переменной (которая не определена явно в этой функции) либо функционала, зависящего от браузера (например, DOM-методов), необходимо быть уверенным, что используемый объект или метод существует.

В качестве примера правильных проверок можно привести следующий код:

```
function color(object, color) {
        if(object) {
                if (color) {
                        if (object.style) {
                                object.style.color = color;
                        }
                }
        }
}
```

Если мы хотим убедиться, что браузер поддерживает W3C DOM, то достаточно провести следующую проверку:

```
if (document.getElementById) {

}
```

В общем случае нет никакой необходимости полагаться на передаваемую браузерами строку агента.

Доступ к элементам

Каждый XML- (а также и HTML-) документ — это дерево узлов. Узел — часть этого дерева (в качестве аналогии можно привести дерево файлов и директорий на жестком диске). Давайте посмотрим, какие функции и атрибуты мы можем использовать, чтобы перемещаться по дереву документа и выбирать необходимые нам узлы.

`getElementById('elementID')`
: возвращает элемент с идентификатором, равным `elementID`

`getElementsByTagName('tag')`
: возвращает массив элементов с именем `tag`

Естественно, мы можем смешивать и сочетать эти два варианта. Несколько примеров:

```
document.getElementById('nav').getElementsByTagName('a')[1];
//      возвращает вторую ссылку внутри элемента, который имеет ID 'nav'

document.getElementsByTagName('div')[1].getElementsByTagName('p')[3];
//      возвращает четвертый параграф внутри второго div в документе.
```

Полный перечень всех DOM-методов, которые поддерживаются сейчас практически всеми браузерами, здесь приводить не имеет смысла. При желании с ними можно ознакомиться на сайте w3.org.

Полезные советы

Мы способны получить доступ к любому элементу документа и затем изменить этот элемент, и мы можем улучшить работу пользователя, не привязываясь к Javascript. Однако возможны некоторые проблемы общего характера.

- Прежде, чем обращаться к элементу, нужно убедиться, что он существует.
- JavaScript-код не должен быть привязан к верстке, только к DOM-дереву. Лишний перевод строки может быть прочитан как новый текстовый узел, разметка может поменяться, а менять скрипты при каждом изменении дизайна не очень хочется.
- HTML, полученный через DOM-методы, в большинстве случаев невалиден. Если мы хотим его повторно использовать, лучше всего привести его к валидному виду или применять `innerHTML` для всех таких случаев (задавая в качестве параметра часть валидного документа).
- Следует избегать частого перебора элементов. Каждая операция (особенно `getElementesByTagName`) довольно ресурсоемка. Стоит кэшировать наиболее часто используемые элементы (подробнее о кэшировании в JavaScript рассказывается чуть далее в этой главе).

- Не стоит проверять атрибуты, которых нет (если мы знаем верстку и знаем JavaScript-код, то в нем не должны появиться неизвестные атрибуты).
- Нужно подходить осторожно к верстке с других сайтов. Например, при проверке `className` на наличие определенной строки нельзя использовать точное соответствие, только регулярные выражения (ибо атрибут этот может содержать несколько классов, разделенных пробелом).

Добавляем обработчики событий

Главная техника, которую мы используем, чтобы сохранить наш Javascript «ненавязчивым», — это хранение скрипта в отдельном файле, что предпочтительней, чем смешивать его с разметкой страницы. Чтобы исполнить функции в нашем `.js` файле, нам надо вызвать их, когда страница загружена (подробнее о событии загрузки страницы было рассказано в начале этой главы).

В некоторых случаях (например, при экстремальной оптимизации, глава четвертая) весь JavaScript-код может находиться в HTML-документе, заключенный в `<script type="text/javascript">...</script>`. Но это не будет означать, что мы смешиваем разметку страницы с ее обработкой, а содержание — с функциональностью. В этих случаях JavaScript-код будет полностью отделен от содержания, для которого он предназначен.

Существует возможность добавлять обработчики событий в комплект к уже существующим обработчикам. При вызове функции мы передаем ей объект, который нужно привязать к событию, тип события и имя функции.

```
function addEvent(object, eventType, function){
        if (object.addEventListener){
                object.addEventListener(eventType, function, false);
                return true;
        } else {
                if (object.attachEvent){
                        var r = object.attachEvent(\"on\"+eventType, function);
                        return r;
                } else {
                        return false;
                }
        }
}
```

События — довольно сложная тема в Javascript. Для разработки простых веб-сайтов указанных примеров достаточно, но если мы переходим к разработке веб-приложений, тут ситуация многократно усложняется. Поэтому стоит быть внимательным к их функционированию в условиях отключенного или неподдерживаемого JavaScript.

Ускоряем обработку событий

Давайте рассмотрим, как можно использовать методы «ненавязчивого» JavaScript для максимального ускорения обработки событий в браузере. Мы можем уменьшить число приемников событий, которые назначены документу, путем определения одного приемника для контейнера и проверки в обработчике, из какого дочернего элемента всплыло это событие.

Предположим, что у нас основная навигация по сайту включает шесть ссылок сверху, четырем из которых требуются обработчики событий, чтобы поменять у них атрибут `href`. У этих четырех ссылок атрибут `class` выставлен в `bundle`.

Скорее всего, ситуация будет выглядеть следующим образом.

```
var MenuNavigation = {
        init: function() {
                var navigation = document.getElementById('mainNav');
                var links = navigation.getElementsByTagName('a');
                for ( var i = 0, j = links.length; i < j; ++i ) {
                        if ( /bundle/i.test(links[i].className) ) {
                                links[i].onclick = this.onclick;
                        }
                }
        },
        onclick: function() {
                this.href = this.href + '?name=value';
                return true;
        }
}
```

В этом фрагменте довольно много лишнего. Во-первых, метод `getElementsByTagName` просматривает каждый дочерний DOM-узел в элементе `mainNav`, чтобы найти все ссылки. Затем мы еще раз пробегаем по всему найденному массиву, чтобы проверить имя класса каждой ссылки. Это пустая трата процессорного времени на каждом этапе. И это замедление загрузки страницы на уровне JavaScript-логики.

Немного усложним

Можно прикрепить один-единственный обработчик событий к элементу `mainNav`, чтобы затем отслеживать все клики на ссылки внутри него:

```
var MenuNavigation = {
        init: function() {
                var navigation = document.getElementById('mainNav');
                navigation.onclick = this.onclick;
        },
        onclick: function(e) {
                if ( /bundle/i.test(e.target.className) ) {
                        e.target.href = e.target.href + '?name=value';
                }
                return true;
        }
}
```

Простота и элегантность данного подхода должны быть очевидны, но у него есть и некоторое количество преимуществ в плане производительности:

- Чем меньше приемников событий прикреплено к документу, тем лучше. Они все загружаются в память и в чрезвычайных случаях могут сильно замедлить работу браузеров. Также это увеличивает число замыканий, что чревато утечками памяти. Подробнее рассказывается далее в этой главе.
- Загружается меньше кода на странице. Одной из главных проблем для сложных веб-приложений является задержка при загрузке JavaScript для исполнения и визуализации документа. Два цикла из первого примера отсутствуют во втором.

- «Исполнение по требованию». Второй пример выполняет немного больше действий, когда вызывается конечный обработчик событий, но это лучше, чем выполнять все действия при загрузке страницы, когда мы даже не знаем, будет ли запущен каждый конкретный обработчик событий. Ссылок на странице может быть сотни, а пользователь нажмет только одну или две из них.

Боремся с Internet Explorer

Есть одна небольшая проблема при использовании изложенного выше кода. Определение целевого элемента у события, на самом деле, не является просто вызовом `e.target`. В Internet Explorer необходимо использовать `e.srcElement`. Самым простым решением для устранения этой проблемы является небольшая функция `getEventTarget`. Ниже представлена наиболее актуальная версия.

```
function getEventTarget(e) {
        var e = e || window.event;
        var target = e.target || e.srcElement;
        if (target.nodeType == 3) { // боремся с Safari
                target = target.parentNode;
        }
        return target;
}
```

Переопределение событий в настоящее время является самой распространенной практикой, если речь заходит о большом числе обработчиков событий (например, о карте с сотнями точек, к которым назначены обработчики событий-кликов). Лучше всего для этого по умолчанию использовать простой, интуитивно понятный и хорошо оптимизированный метод для применения в качестве шаблона в программировании на стороне клиента, и он не должен требовать сотен строчек JavaScript-библиотек для своей работы.

Пойдем дальше

А что, если нам нужно добавить такой обработчик на все ссылки (или почти на все)? Правильно: тогда для контейнера всех этих ссылок стоит выбрать `document.body`. Ниже пример кода, который позволяет так сделать.

```
var MenuNavigation = {
        init: function() {
                document.body.onclick = function(e) {
                        var target = getEventTarget(e);
                        if ( target && /bundle/i.test(target.className) ) {
                                target.href += '?name=value';
                        }
                        return true;
                };
        }
        var getEventTarget = function(e) {
                var e = e || window.event;
                var target = e.target || e.srcElement;
// боремся с Safari и вложенностью
                while ( !target.href || target.nodeType == 3 ) {
                        target = target.parentNode;
                }();
                return target;
        }
```

```
}
window.onload = MenuNavigation.init;
```

Если мы собираемся обрабатывать **все** ссылки, то нужно учесть, что в них могут быть вложены и картинки, и другие теги, поэтому добавлено рекурсивное «всплытие» ссылки: проверяется родитель объекта, на котором сработало событие, и если у него не определен атрибут `href`, то перебор продолжается, иначе возвращаем искомый объект. Вложение ссылок друг в друга запрещено стандартами, так что, если мы сами же проектируем HTML-код, то бояться нечего.

Обработка событий в браузерах

Давайте рассмотрим несколько практических способов работы с обработчиками событий в браузерах. Например, можно назначить обработчик напрямую:

```
node.onclick = function(){

}
```

Если нужно несколько событий, или просто «осторожничаем», то можно воспользоваться следующей распространенной записью:

```
if (node.addEventListener)
      node.addEventListener('click', function(e){}, false);
else
      node.attachEvent('onclick', function(){});
```

Или таким модицифицированным вариантом (меньше символов):

```
if (node.attachEvent)
       node.attachEvent('onclick', function(){});
else
       node.addEventListener('click', function(e){}, false);
```

Можно также использовать отдельную переменную для обработчика события:

```
var addEvent = node.attachEvent || node.addEventListener;
addEvent(/*@cc_on 'on'+@*/'click', function(){}, false);
```

Или записать в одну строку с использованием условной компиляции:

```
node[/*@cc_on !@*/0 ? 'attachEvent' : 'addEventListener']
               (/*@cc_on 'on'+@*/'click', function(){}, false);
```

Работаем с событиями

Давайте рассмотрим, что мы можем извлечь из события после перехвата его с помощью соответствующего обработчика:

```
node[/*@cc_on !@*/0 ? 'attachEvent' : 'addEventListener']
       (/*@cc_on 'on'+@*/'click', function(e){

       var target = e.target || e.srcElement

       // или
```

```
        if (!e.target) {
                e.target = e.srcElement
        }

        // или, если нам надо всего один раз
        (e.target || e.srcElement).tagName

        // true везде кроме IE, в котором this === window
        this == node;
        // отменяем всплытие события
        if (e.stopPropagation)
                e.stopPropagation()
        else
                e.cancelBubble

        // или просто используем вариант, который
        // для совместимости работает во всех браузерах.
        e.cancelBubble = true

        // убираем действие по умолчанию (в данном случае клик)
        if (e.preventDefault)
                e.preventDefault()
        else
                e.returnValue = false
        // при attachEvent (как здесь) работает только в IE;
        // при назначении напрямую (node.onclick) — везде
        return false;

}, false):
```

7.3. Применение «ненавязчивого» JavaScript

В предыдущих разделах были представлены некоторые теоретические аспекты построения клиентской логики, ориентированной на максимальное быстродействие и адекватную ему замену в проблемных случаях. Ниже приведены практические решения по облегчению наиболее характерных сторон клиентского взаимодействия любого сайта: это счетчики посещений и размещение рекламы. Ведь они встречаются сейчас практически на любом веб-проекте.

Принципы «ненавязчивой» рекламы

Итак, как лучше организовывать размещение рекламы на веб-страницах для того, чтобы доставить посетителям сайтов минимум неудобств? Поскольку большинство выводов последуют из анализа техник «ненавязчивого» JavaScript, то раздел озаглавлен именно таким образом. Речь пойдет о клиентской оптимизации использования рекламы на сайтах.

Как было продемонстрировано в исследованиях 2007–2008 годов, б**о**льшая часть задержек при загрузке страницы у обычного пользователя приходится на долю рекламы, подключаемой, в основном, через JavaScript. Далее будут рассмотрены основные типы использования рекламы на сайтах и предложены способы (в большинстве своем опробованные на практике) для разгона ее загрузки.

Можно спросить: зачем нам это? Разве разработчики баннерообменных систем, контекстной рекламы и других сложных клиент-серверных приложений не подумали уже за нас о возможных последствиях? Подумали, но можно подумать и дальше. Конвертация показов рекламы в клики/покупки напрямую зависит от общего удобства использования

сайтом. А оно, в свою очередь, значительно ухудшается при обширном применении различных рекламных сетей.

Именно поэтому клиентская оптимизация вызовов рекламы конвертируется во вполне реальное повышение прибыли компании от своего веб-сайта (или же целой их сети). Если вы уже задумывались над тем, почему ваши сайты так медленно грузятся из-за обилия рекламы, или проектировали сеть учета показов-переходов, тогда следующие мысли будут действительно полезны. Ниже изложены все ключевые моменты, которыми стоит руководствоваться в этих случаях.

`document.write` против `innerHTML`

Контекстная реклама, пожалуй, является одним из главных «тормозов» при загрузке страницы (при прочих равных условиях), ибо активно применяет `document.write`, который «морозит» загрузку до получения всех необходимых JavaScript-файлов с внешних серверов. Естественным предположением было бы использовать вместо него `document.write` `innerHTML`

Принцип первый: **при проектировании рекламных вызовов используйте `innerHTML` или `script.src`** (последний подразумевает подключение внешнего JavaScript-файла путем создания соответствующего дополнительного узла в `head` после загрузки страницы, техника более подробно описана в начале главы). Идеальным является подход, когда для оценки эффективности рекламы не применяется клиентская логика (все показы и переходы отслеживаются по серверным логам).

Если вам не удается избежать вызовов `document.write`, любыми путями помещайте их в самый низ документа. Возможно, стоит рассмотреть вариант, когда после загрузки страницы блок с контекстной рекламой перемещается в необходимое место, а все это время он скрыт где-нибудь в подвале документа и не влияет на скорость загрузки основного содержания.

Принцип второй: **вставляйте рекламный код максимально близко к концу страницы.**

Контекстная реклама

Основными игроками на рынке контекстной рекламы на данный момент являются Яндекс.Директ, Google AdSense и Бегун. Google поступает наиболее практично: в результате вызова скрипта вставляется `iframe`, который дальше уже загружает все рекламное объявление. Поскольку исходные файлы рекламных скриптов расположены на одной из самых доступных и быстрых CDN в мире, то скорость отображения таких объявлений на клиенте впечатляет.

С Яндексом ситуация похуже. Мало того, что выполняется `document.write` содержимого рекламных объявлений в основное DOM-дерево, к тому же, загружается порядка 5 дополнительных файлов в виде узлов текущего документа (в частности, это счетчики и файл стилей). Все это не самым лучшим образом сказывается на быстродействии. Преобразовать в данном случае вызов Яндекс.Директа к виду `innerHTML` не удается (однако вполне вероятно, что разработчики со стороны Яндекса в ближайшее время изменят JavaScript-код, и такая возможность появится).

Бегун в этом плане приятно удивил: он предоставляет (слабо документированное, но все же) API для множественных вставок рекламных объявлений на страницу при помощи `innerHTML`. Для этого всего лишь нужно выставить в качестве JavaScript-переменных:

```
begun_multispan=1,begun_spans=[{'span_id':'ad','limit':7,'width':230}]
```

В данном случае подключается множественный показ рекламных объявлений (`begun_multispan=1`), далее задается, в каком месте и в каком количестве их показывать. Это происходит через массив `begun_spans`, где для `span_id` назначается идентификатор блока, в который будет вставлены объявления после загрузки, `limit` указывает на их количество в данном блоке, а `width` просто описывает ширину рекламного блока (в пикселах). Таким образом, можно вставить код Бегуна в самый низ страницы и максимально ускорить ее загрузку.

В случае Бегуна в клиентский браузер в минимальном варианте загружается всего 2 файла (отвечающий за логику и содержащий сами объявления), что также добавляет чести данной системе контекстных объявлений.

В качестве альтернативы можно рассмотреть размещение практически всего кода на собственном сайте, а подгружать только необходимую контекстную составляющую (хотя в правилах рекламных сетей это и запрещается, но других кардинальных способов ускорить загрузку и сжать скрипты, которые (у Бегуна, например) отдаются в обычном виде, наверное, не существует).

TopLine, Pop-Up, Pop-Under и RichMedia

В стандартных рекламных сетях сейчас превалируют три формата показа объявлений на странице: TopLine, Pop-Under и RichMedia. Последние два весьма дружественны к техникам «ненавязчивого», ибо подключаются только после полной загрузки страницы (хотя такая реклама, возможно, будет слишком раздражающей, чтобы ее использовать на нормальных сайтах). TopLine отличается тем, что должен быть вставлен в самом начале HTML-документа и, таким образом, максимально замедлит его загрузку.

Поскольку TopLine мало чем отличается от стандартных баннеров, то посетители будут довольно лояльны к его использованию. Однако, как же нам исправить ситуацию с замедлением загрузки? Так же, как и для контекстной рекламы: переместить вызов `document.write` в `innerHTML` (или в `appendChild`). Что и было успешно проделано. Исходный код модифицированного варианта слишком простой, чтобы приводить его здесь. Однако стандартный код вызова может быть замечательно заменен DOM-эквивалентом, который срабатывал по комбинированному событию `window.onload` и вставлял в заранее подготовленное место все необходимые элементы.

Принцип третий: **используются заранее подготовленные места для рекламных объявлений**. Если заранее (через стили) назначить размеры тем областям, где будет показана реклама (это несложно сделать, ибо почти всегда известны точные размеры баннеров и текстовых блоков), то посетители будут испытывать гораздо меньше дискомфорта при заходе на странице сайта. Экран не будет «дергаться» при загрузке, а реклама будет появляться постепенно, заполняя строго отведенное ей место.

Внутренние рекламные сети

На некоторых веб-страницах, использующих внутренние системы показа рекламы, вставка объявлений выполняется через `iframe` (в общем случае — наиболее быстрый способ), иногда через `document.write` (иногда даже каскадный, когда с помощью одного `document.write` вставляется скрипт, в котором содержится другой и т.д.). Последний способ может достаточно замедлить загрузку страницы, если звеньев в цепочки вставок много или же они расположены на медленных серверах.

И только на небольшом проценте сайтов действует наиболее быстрый подход: логика рекламных показов рассчитывалась на сервере, а на клиент отдавался уже полностью статичный код. При разработке внутренней рекламной системы самым оптимальным будет именно такой метод.

Принцип четвертый: **создавайте рекламные объявления на сервере**. Бо́льшая (если не вся) логика может быть вынесена на сервере безо всяких потерь функциональности. Определение страны, языка, браузера, источника перехода — все это можно установить еще до того, как страница будет выдана пользователю. Все, что ему нужно, — это лишь итоговые картинки или мультимедийные файлы. Зачем устраивать сеть распределенных вычислений из пользовательских машин, если все можно сделать на мощном сервере, специально для этого предназначенном?

Идеальная архитектура рекламной сети

Исходя из всего вышесказанного, можно представить построение внутренней сети показа рекламных объявлений примерно в следующем виде.

1. **Создание внутреннего хранилища объявлений.** Этот этап присутствует во всех современных баннерообменных сетях, ведь нужно как-то упорядочить весь рекламный материал по категориям перед назначением его на места показа.
2. **Создание каталога рекламных мест.** Этот этап тоже обычно проходится, но не всегда явно. Каждый рекламный блок может быть откручен только в нескольких соответствующих местах (например, на странице есть 3 возможных варианта для вывода баннера: 240x240, 240x720 и 120x800). Каждое рекламное место должно быть прикреплено к ряду страниц, на которых оно присутствует.
3. **Логика показа рекламных объявлений.** После первых двух шагов необходимо задать правила, согласно которым те или иные объявления будут выводиться на соответствующих страницах. Это также обычно осуществляется, однако, далее этого шага дело, как правило, не двигается — ведь можно с JavaScript-вызова просто передать все необходимые параметры выделенному скрипту, который сам рассчитает, что отдать клиенту.
4. **Настройка серверной стороны.** На самом деле, вместо клиентской логики должен быть разработан серверный модуль, который обеспечит тот же самый функционал, но без дополнительных запросов к серверу. Ведь все (или почти все) данные, как было упомянуто выше, у нас уже есть — так зачем нам дополнительная нагрузка на пользователя?
5. **Настройка статистики.** Этот момент является, пожалуй, наиболее ключевым во всей схеме. Ведь при вызове внешнего скрипта мы, фактически, не будем думать о статистике — она собирается автоматически сторонним приложением. Если мы разрабатываем внутреннее решение, то все данные о произведенных показах должны собираться либо самим модулем, который эти показы осуществляет, либо (что более предпочтительно) собираться на основе логов запросов к рекламным файлам (например, именно таким образом организованы счетчики посещаемости).

6. **Решение коллизий с кэшированием.** Для высоконагруженных проектов, где в полной мере применяется кэширование, немаловажным будет задуматься над тем, как разрешить конфликты рекламных показов с занесением страницы в серверный (или даже клиентский) кэш. Для этого нужно будет либо кэшировать страницу отдельно по модулям (т.е. рекламный блок вообще не кэшировать), либо класть в кэш все возможные варианты страницы, а потом показывать только тот, который удовлетворяет поставленным условиям. Разумеется, это проблема возникает только для проектов с большой посещаемостью (ее решение, в любом случае, должно прозрачно следовать из архитектуры такого проекта, а не являться дополнительной головной болью для серверных разработчиков).

В общем, совет один — используйте свой сервер по назначению, а не перекладывайте его работу на конечных пользователей. Изложенные соображения помогут разобраться с задержками при загрузке страницы, вызванными рекламными показами, и уменьшить их. Универсальных решений в данной области не так много, в основном, приходится руководствоваться именно общими принципами.

Разгоняем счетчики: от мифов к реальности

Давайте рассмотрим теперь, что собой представляет код JavaScript-счетчика. Обычно (в 99% случаев) он «вытаскивает» из клиентского окружения набор параметров: URL текущей страницы; URL страницы, с который перешли на текущую; браузер; ОС и т.д. Затем они все передаются на сервер статистики. Все дополнительные возможности счетчиков связаны с обеспечением максимальной точности передаваемой информации (кроссбраузерность, фактически). Наиболее мощные (Omniture, Google Analytics) используют еще и собственные переменные и события, чтобы усилить маркетинговую составляющую.

Но сейчас речь не об этом. Как собранные на клиенте данные попадают на сервер статистики? Все очень просто: в документе создается уникальный элемент, в URL которого «зашиваются» все необходимые значения (обычно в качестве GET-параметров). URL этот ведет, как можно догадаться, на сервер статистики, где данные кладутся в базу и каким-то образом показываются в администраторском интерфейсе.

Как же создается этот самый «уникальный» элемент? Так сложилось, что наиболее простым транспортным средством для данных стала картинка. Обычный однопиксельный GIF-файл (сейчас, в эпоху CSS-верстки, это, пожалуй, единственное его применение) отдается сервером в ответ на URL с параметрами от клиента.

Разбираем по косточкам

Нам нужно гарантировать загрузку внешнего JavaScript-файла «ненавязчивым» образом, при этом обеспечить запрос на сервер статистики (создание картинки со специальными параметрами). В случае Google Analytics все будет очень тривиально, ибо картинка уже создается через `new Image(1,1)`. Однако большинство счетчиков (Рунета и не только) оперируют `document.write`, и если такая конструкция отработает после создания основного документа, то браузер просто создаст новый, в который запишет требуемый результат. Для пользователя это выльется в совершенно пустую страницу в браузере.

Основная сложность в переносе скриптов статистики в стадию пост-загрузки (по комбинированному событию `window.onload`, которое описано в начале главы)

заключается как раз в изменении вызова картинки, обеспечивающей сбор статистики, на DOM-методы (это может быть не только `new Image`, но и `appendChild`). В качестве примера рассмотрим преобразование скрипта статистики для `LiveInternet`:

```
document.write("<img src='http://counter.yadro.ru/hit;tutu_elec?r"+
escape(document.referrer)
+((typeof(screen)=="undefined")?"":";s"+screen.width+"*"+screen.height+"*"
+(screen.colorDepth?screen.colorDepth:screen.pixelDepth))
+";u"+escape(document.URL)+";"+Math.random()+"' width=1 height=1 alt=''>")
```

Как мы видим, его нельзя просто так перенести в область динамической загрузки. Для этого данный код нужно преобразовать примерно следующим образом:

```
new Image(1,1).src='http://counter.yadro.ru/hit;tutu_elec?r"
+escape(document.referrer)+((typeof(screen)=="undefined")?"":";s"
+screen.width+"*"+screen.height+"*"
+(screen.colorDepth?screen.colorDepth:screen.pixelDepth))
+";u"+escape(document.URL)+";"+Math.random()
```

Таким образом (все приведенные участки кода — это одна строка, разбитая для удобства чтения), мы просто заменили вызов `document.write` на `new Image()`. Это поможет в большинстве случаев. Если у вас ситуация не сложнее уже описанной, то следующие абзацы можно смело пропустить.

А если сложнее?

Не все счетчики одинаково просты. Например, для сбора статистики с помощью того же Google Analytics нам нужно загрузить целую библиотеку — файл `urchin.js` или `ga.js`. На наше счастье конкретно в этом скрипте данные уже собираются с помощью создания динамической картинки.

Поэтому все, что нам требуется в том случае, если во внешней библиотеке находится мешающий нам вызов `document.write`, — это заменить его соответствующим образом. Обычно для этого необходимо изменить сам JavaScript-файл. Не будем далеко ходить за материалом и рассмотрим преобразования на примере Omniture — довольно популярной на Западе библиотеки для сбора статистики.

Сначала нам нужно найти соответствующий участок кода внутри JavaScript-файла. В нашем случае это будет возвращаемая строка, которая затем вписывается в документ:

```
var s_code=s.t();if(s_code)document.write(s_code)
```

В коде Omniture достаточно найти соответствующий `return`

```
return '<im'+'g sr'+'c="
+"\"'+rs+'\" width=1 height=1 border=0 alt=\"\">'
```

И заменить его на следующий код (заметим, что для `src` картинки берется переменная `rs`)

```
return 'new Image(1,1).src=\"'+rs+'\"'
```

Затем мы уже можем заменить вызов и в самом HTML-файле на

```
var s_code=s.t();if(s_code)eval(s_code)
```

Для того, чтобы все окончательно заработало, необходимо заменить в файле `s_code.js` и остальные вызовы `document.write` (всего их там два). Выглядит это примерно так:

```
var c=s.t();if(c)s.d.write(c);
...
s.d.write('<im'+'g name=\"'+imn+"
+"'\" height=1 width=1 border=0 alt=\"\">');
```

меняем на

```
var c=s.t();if(c)eval(c);
...
new Image(1,1).name=imn;
```

Внимательные читатели уже заметили, что альтернативой `document.write` в нашем случае стал `eval`, что, по большому счету, не очень хорошо. Однако здесь не ставится задачи перебирать конкретный скрипт «по косточкам», чтобы избавиться от такого костыля. В некоторых случаях стоит ограничиться просто уверенностью, что вся остальная логика останется нетронутой после вмешательств, ибо все изменения касались только отправки собираемых данных на сервер.

Делаем статистику динамической

Итак, мы узнали, как подготовить внешний JavaScript-файл к динамической загрузке. Осталось понять, как теперь это использовать.

Основное преимущество (или недостаток?) Omniture заключается в том, что JavaScript-файл (обычно `s_code.js`) располагается на нашем сервере. Поэтому ничего не мешает нам его там и заменить. После этого обеспечить динамическую загрузку и вызов счетчика уже не составит труда.

В той ситуации, когда скрипт совсем внешний (Google Analytics), у нас, по большому счету, только 2 выхода:

- **Перенести сам скрипт на наш сервер**, добавить в него необходимые инициализационные переменные и вызов (помимо самого объявления) функции статистики (для Google Analytics это `urchinTracker()`). В качестве плюсов можно отметить то, что, в общем случае, скрипт будет загружаться с нашего сервера побыстрее, чем будет устанавливаться новое соединение с www.google-analytics.com и проверяться, что файл не изменился. В качестве минусов — необходимость отслеживать (возможные) изменения скрипта и необходимость отдавать JavaScript-файл с собственного сервера со всеми вытекающими последствиями.
- **Проверять через определенные промежутки времени, загрузилась ли библиотека**. Пишется очень простой код, который через каждый 10 мс проверяет, доступна ли необходимая из библиотеки функция. Если да, то она вызывается. В противном случае проверка запускается снова через 10 мс. Плюсы: можно использовать тот же самый скрипт, что и раньше. Минусы: дополнительная (небольшая) нагрузка на клиентский браузер при загрузке. В качестве примера можно рассмотреть следующий код для Goole Analytics.

```
var _counter_timer = setInterval(function() {
        if (urchinTracker) {
                urchinTracker();
                clearInterval(_counter_timer);
        }
}, 10);
```

Видим, что в первом случае у нас загрузка сильно ускоряется (особенно для постоянных посетителей), во втором случае — получается дешево и сердито, зато более надежно (в смысле отсутствия дополнительного вмешательства в исходный код).

7.4. Замыкания и утечки памяти

В этом разделе речь идет, преимущественно, об Internet Explorer и его скриптовом движке — JScript. Однако, во-первых, многие из приведенных методик и советов имеют большое значение для других браузеров и их виртуальных JavaScript-машин. Во-вторых, IE на данный момент занимает порядка 60% пользовательской аудитории, поэтому при рассмотрении эффективного программирования на JavaScript выбрасывать его из поля зрения было бы, по меньшей мере, глупо.

В прошлом утечки памяти не создавали никаких проблем веб-разработчикам. Страницы были предельно простыми, а переход с одной на другую был единственным нормальным способом для освобождения всей доступной памяти. Если утечка и происходила, то была настолько незначительна, что оставалась незамеченной.

Современные веб-приложения должны разрабатываться с учетом более высоких стандартов. Страница может выполняться в течение часов без дополнительных переходов по сайту, при этом она будет сама динамически запрашивать новую информацию через веб-сервисы. Скриптовый движок испытывают на прочность сложными схемами отработки событий, объектно-ориентированным JScript и замыканиями, производя на свет все более мощные и продвинутые приложения. При этом, учитывая некоторые другие особенности, знание характерных шаблонов утечек памяти становится все более необходимым, даже если они были раньше спрятаны за механизмом навигации по сайту.

Хорошей новостью будет то, что шаблоны утечек памяти могут быть легко обнаружены, если знать, где их искать. Методы устранения наиболее тяжелых из них подробно описаны, и они требуют лишь небольшого количества дополнительных усилий. Хотя некоторые страницы могут по-прежнему «падать» из-за небольших утечек, самые значительные утечки могут быть легко удалены.

Шаблоны утечек

В следующих разделах мы обсудим общие шаблоны утечек памяти и приведем несколько примеров для каждого. Замечательным примером утечек будет случай замыкания в JScript, в качестве другого можно привести использование замыкания для обработки событий. При знакомстве с обработчиками событий можно будет легко найти и устранить многие утечки памяти, однако другие случаи, связанные с замыканиями, могут остаться незамеченными.

Основные виды утечек можно разбить на следующие 4 типа.

1. **Циклические ссылки**, когда существует взаимная ссылка между DOM-объектом в браузере и скриптовым движком. Такие объекты могу приводить к утечкам памяти. Это самый распространенный шаблон.
2. **Замыкания** являются самым значимым шаблоном для существующих архитектур веб-приложений. Замыкания довольно легко зафиксировать, потому что они зависят от ключевого слова, относящегося к используемому скриптовому языку, и могут быть по нему, в общем случае, обнаружены.
3. **Постраничные утечки** зачастую представляют собой очень маленькие утечки, которые возникают из-за учета объектов при перемещении от элемента к элементу. Ниже будет рассмотрен порядок добавления DOM-объектов, а заодно и характерные примеры, которые демонстрируют, как небольшое изменение вашего кода может предотвратить создание таких учитываемых объектов.
4. **Псевдо-утечки**, по существу, не являются утечками, но могут вызывать некоторое беспокойство, если не понимать, куда расходуется память. Будет рассмотрена перезапись объекта скрипта и как она проявляется в расходовании очень малого количества памяти, если работает так, как требуется.

Циклические ссылки

Циклические ссылки являются источником практически любой утечки. Обычно скриптовые движки нормально отрабатывают с циклическими ссылками при помощи собственных сборщиков мусора, однако из-за некоторых неопределенностей их механизм эвристических правил может дать сбой. Одной из таких неопределенностей будет состояние DOM-объекта, к которому имеет доступ текущая порция скрипта. Основной принцип в данном случае можно описать так:

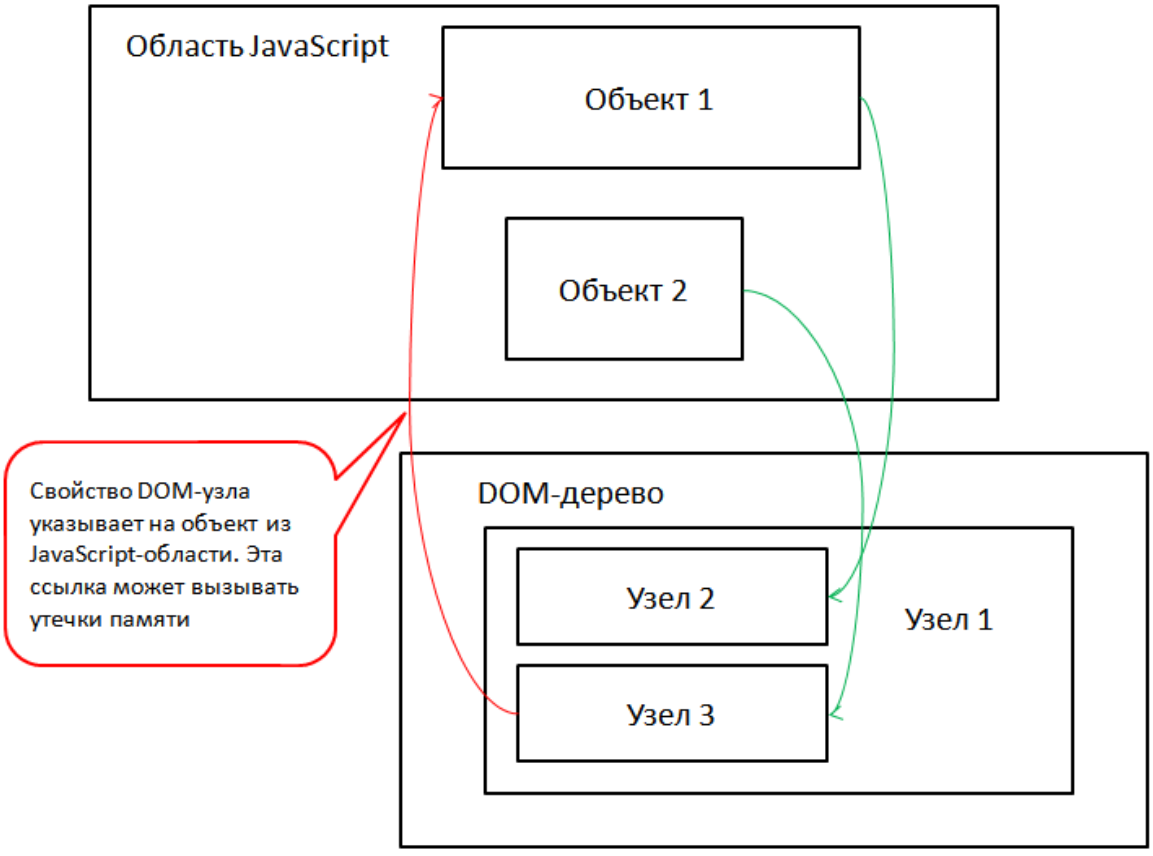


Рис. 35. Основной шаблон циклической ссылки

Утечка в таком шаблоне происходит из-за особенностей учета DOM-ссылок. Объекты скриптового движка удерживают ссылку на DOM-элемент и ожидают, пока будут освобождены все внешние ссылки, чтобы освободить, в свою очередь, этот указатель на DOM-элемент. В нашем случае у нас две ссылки на объект скрипта: внутри области видимости скриптового движка и от расширенного свойства DOM-элемента. По окончанию своей работы скрипт освободит первую ссылку, но ссылка из DOM-элемента никогда не будет освобождена, потому что ждет, что это сделает объект скрипта!

Наверное, первой же мыслью будет, что такой сценарий развития событий легко обнаружить и устранить, однако на практике представленный базовый случай является только вершиной айсберга. Может оказаться так, что циклическая ссылка находится в конце цепочки из 30 объектов, а обнаружить ее при этом крайне тяжело.

Стоит посмотреть, как данный шаблон будет выглядеть в HTML. Это может вызвать утечку, используя глобальную переменную и DOM-объект, как показано ниже.

```
<script type=”text/javascript”>

        var myGlobalObject;

        function SetupLeak()
        {
        // Для начала создадим ссылку из скрипта на DOM-элемент
                myGlobalObject = document.getElementById("LeakedDiv");

        // Потом установим ссылку из DOM на глобальную переменную
                document.getElementById("LeakedDiv").expandoProperty =
                        myGlobalObject;
        }

        function BreakLeak()
        {
                document.getElementById("LeakedDiv").expandoProperty = null;
        }

        window.onload = SetupLeak;
        window.onunload = BreakLeak;

</script>
```

Чтобы разрушить этот шаблон, можно использовать явное присвоение `null` тому свойству, которое «течет». Таким образом, при закрытии документа мы сообщаем скриптовому движку, что между DOM-элементом и глобальной переменной нет больше никакой связи. В результате все ссылки будут очищены, и сам DOM-элемент будет освобожден. В таком случае веб-разработчик знает больше о внутренних отношениях между объектами, чем сам скрипт, и может поделиться этой информацией со скриптом.

Более сложный случай

Хотя это только базовый шаблон, для более сложных ситуаций может оказаться нелегко выяснить первопричину утечки. Распространенной практикой по написанию объектно-ориентированного JScript является расширение DOM-элементов путем инкапсуляции их внутри JScript-объекта. В процессе создания такого объекта в большинстве случаев получается ссылка на желаемый DOM-элемент, а затем она сохраняется в только что созданном объекте, при этом экземпляр этого объекта оказывается прикрепленным к

DOM-элементу. Таким способом модель приложения всегда получает доступ ко всему, что нужно. Проблема заключается в том, что это явная циклическая ссылка, но из-за использования других аспектов языка она может остаться незамеченной. Устранение шаблонов такого рода может быть весьма затруднительным, но вы вполне можете использовать простые методы, обсужденные ранее.

```
<script type=”text/javascript”>

        function Encapsulator(element)
        {
        // Создаем элемент
                this.elementReference = element;

        // Создаем циклическую ссылку
                element.expandoProperty = this;
        }

        function SetupLeak()
        {
        // Утечка: все в одном
                new Encapsulator(document.getElementById("LeakedDiv"));
        }

        function BreakLeak()
        {
                document.getElementById("LeakedDiv").expandoProperty = null;
        }

        window.onload = SetupLeak;
        window.onunload = BreakLeak;

</script>

<div id="LeakedDiv"></div>
```

Более комплексным решением заявленной проблемы может стать создание процесса регистрации ссылок, чтобы соответствующим образом уведомлять приложение, какие из них могут быть удалены. В этом случае по закрытию документа нужно будет пройтись по всем таким элементам и убрать с них ссылки. Однако при неверном подходе можно не только не уменьшить, а даже увеличить количество циклических ссылок, не избавляясь от заявленной проблемы.

Замыкания

Замыкания очень часто являются причиной утечек из-за того, что они создают циклические ссылки практически без ведома программиста. Ведь не так очевидно, что параметры родительской функции могут быть «заморожены» во времени, на них может быть создана ссылка, которая будет удерживаться до полного освобождения замыкания. Это стало уже широко распространенной практикой программирования, и пользователи достаточно часто сталкиваются с нехваткой доступных ресурсов.

В силу того, что данные содержат некоторую часть истории выполнения скрипта, мы не можем так просто с ними расправиться. Давайте сейчас рассмотрим обновленную диаграмму циклических ссылок с учетом модели замыканий, чтобы понять, откуда берутся дополнительные ссылки.

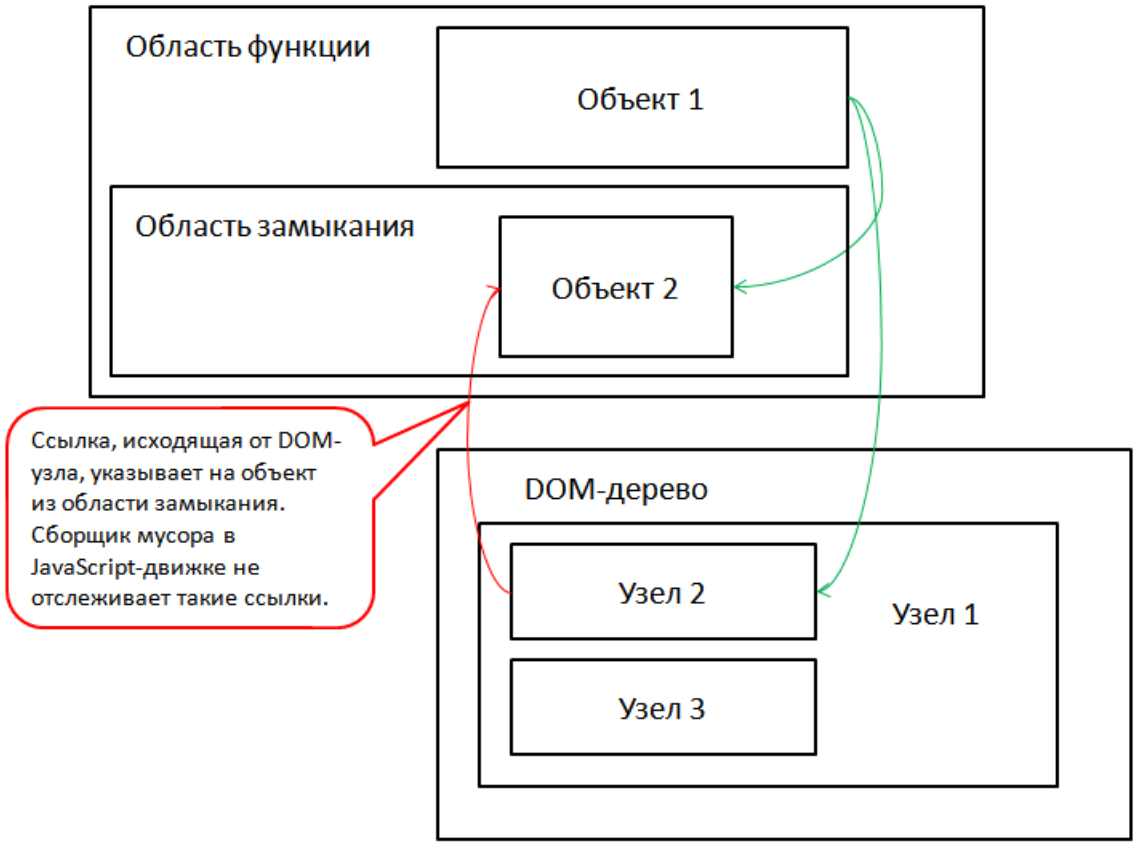


Рис. 36. Циклические ссылки с самозамыканием

В случае обычной циклической ссылки у нас есть 2 постоянных объекта, которые содержат ссылки друг на друга, — но замыкания отличаются от этой модели. Вместо создания прямых ссылок, они наследуют информацию из пространства порождающих их объектов. В обычном случае локальные переменные в функции и используемые параметры существуют только в течение времени жизни самой функции. В случае замыканий эти переменные и параметры имеют внешнюю ссылку на весь период жизни замыкания, который может быть значительно дольше, чем у породившей ее функции.

В качестве примера можно рассмотреть Объект 2, который был бы освобожден по окончанию вызова функции, в общем случае. Однако после добавления замыкания была создана вторая ссылка на этот параметр, которая не может быть освобождена, пока не будет закрыто замыкание. Если вы прикрепили замыкание к событию, то вам придется его у события, в конце концов, убрать. Если замыкание прикреплено к расширенному параметру, нужно будет его занулить (приравнять этот параметр к `null`).

Замыкания создаются по вызову функции, поэтому два различных вызова породят два независимых замыкания, каждое будет содержать ссылки на параметры своего вызова. Из-за такой внешней прозрачности очень легко, на самом деле, позволить замыканиям «течь». В следующем примере приводится один из базовых случаев возникновения утечек при замыканиях:

```
<script type="text/javascript">

        function AttachEvents(element)
        {
        // Эта структура создает у элемента ссылку на ClickEventHandler
```

```
                element.attachEvent("onclick",ClickEventHandler);

                function ClickEventHandler()
                {
                // Это замыкание ссылается на элемент
                }
        }

        function SetupLeak()
        {
        // Происходит утечка
                AttachEvents(document.getElementById("LeakedDiv"));
        }

        function BreakLeak()
        {
        }

        window.onload = SetupLeak;
        window.onunload = BreakLeak;

</script>

<div id="LeakedDiv"></div>
```

Устранить эту утечку не так просто, как в случае с обычной циклической ссылкой. «Замыкание» можно рассматривать как временный объект, который существует в области видимости функции. После завершения функции ссылка на само замыкание теряется, поэтому встает вопрос: как же вызвать завершающий **`detachEvent`**?

Одним из возможных выходов заключается в использовании второго замыкания, которое цепляется на событие **`onUnload`** всего окна браузера. Поскольку оно имеет объекты из той же области видимости, то становится возможным снять обработку события, высвободить замыкание и завершить процесс очистки. Чтобы окончательно прояснить ситуацию, мы можем добавить в наш пример дополнительное свойство, в котором сохраним ссылку на замыкание, затем по ссылке освободим замыкание и обнулим само свойство.

```
<script type=”text/javascript”>

        function AttachEvents(element)
        {
        // чтобы иметь возможность освободить замыкание,
        // мы должны где-то сохранить ссылку на него
                element.expandoClick = ClickEventHandler;

        // Эта структура создает у элемента ссылку
        // на ClickEventHandler
                element.attachEvent("onclick", element.expandoClick);

                function ClickEventHandler()
                {
                // Это замыкание ссылается на элемент
                }
        }

        function SetupLeak()
        {
        // Происходит утечка
                AttachEvents(document.getElementById("LeakedDiv"));
        }
```

```
        function BreakLeak()
        {
                document.getElementById("LeakedDiv").detachEvent("onclick",
                        document.getElementById("LeakedDiv").expandoClick);
                document.getElementById("LeakedDiv").expandoClick = null;
        }
        window.onload = SetupLeak;
        window.onunload = BreakLeak;

</script>

<div id="LeakedDiv"></div>
```

В примере как раз демонстрируется техника устранения утечек за счет создания циклических ссылок, которые, однако, как сами грамотно устраняются, так и помогают убрать первоначальные ссылки.

В данном примере можно не использовать замыкание как обработчик события, а переместить его в глобальный контекст. Как только замыкание становится функцией, оно больше не наследует параметры или локальные переменные от родительской функции, поэтому нам не нужно вообще волноваться по поводу циклических ссылок, вызываемых данным замыканием. Так что бóльшая часть проблем может быть решена просто путем устранения всех замыканий, которые реально не нужны.

Постраничные утечки

Утечки, которые зависят от порядка добавления элементов в DOM-дереве, всегда вызваны тем, что создаются промежуточные объекты, которые затем не удаляются должным образом. Это происходит и в случае создания динамических элементов, которые затем присоединяются к DOM. Базовый шаблон заключается во временном соединении двух только что созданных элементов, из-за чего возникает область видимости, в которой определен дочерний элемент. Затем, при включении этого дерева из двух элементов в основное, они оба наследуют контекст всего документа, и происходит утечка во временном объекте (чей контекст не был закрыт).

На следующей диаграмме показаны два метода присоединения динамически созданных элементов к общему дереву. В первой модели мы присоединяем дочерние элементы к их родителям и, в конце концов, полученное дерево — к DOM. Этот метод может вызвать утечки памяти при неправильном создании временных объектов. Во втором случае мы присоединяем элементы сразу к первичному дереву, начиная динамическое создание узлов с самого верха до последнего дочернего элемента. В силу того, что каждое новое присоединение оказывается в области видимости глобального объекта, мы никогда не создаем временных контекстов. Этот метод значительно лучше, потому что позволяет избежать потенциальных утечек памяти.

Рис. 37. Утечки, связанные с порядком добавления DOM-элементов

Далее стоит проанализировать характерный пример, который обычно упускают из рассмотрения большинство алгоритмов по обнаружению утечек. Поскольку он не затрагивает публично доступных элементов и объекты, которые вызывают утечки, весьма невелики, то вполне возможно, что эту проблему никто и не заметит. Чтобы заставить наш пример работать, нам нужно снабдить динамически создаваемые элементы указателем на какую-либо внутреннюю функцию. Таким образом, при каждом таком вызове будет происходить утечка памяти на создание временного внутреннего объекта (например, обработчика событий), как только мы будет прикреплять создаваемый объект к общему дереву. Поскольку утечка весьма мала, нам придется запустить сотни циклов. Фактически, она составляет всего несколько байтов.

Запуская пример и возвращаясь к пустой странице, можно замерить разницу в объеме памяти между этими двумя случаями. При использовании первой DOM-модели для прикрепления дочернего узла к родительскому, а затем родительского — к общему дереву, нагрузка на память немного возрастает. Данная утечка использования перекрестных ссылок характерна для Internet Explorer. При этом память не высвобождается, если мы перезапустим IE-процесс. Если протестировать пример, используя вторую DOM-модель для тех же самых действий, то никакого изменения в размере памяти не последует. Таким образом, можно исправить утечки такого рода.

```
<script type=”text/javascript”>

        function LeakMemory()
        {
                var hostElement = document.getElementById("hostElement");
```

```
        // Давайте посмотрим, что происходит с памятью в Диспетчере Задач

                for (i = 0; i < 5000; i++)
                {
                        var parentDiv = document.
                                createElement("<div onClick='foo()'>");
                        var childDiv = document.
                                createElement("<div onClick='foo()'>");

                // Здесь вызывается утечка на временном объекте
                        parentDiv.appendChild(childDiv);
                        hostElement.appendChild(parentDiv);
                        hostElement.removeChild(parentDiv);
                        parentDiv.removeChild(childDiv);
                        parentDiv = null;
                        childDiv = null;
                }
                hostElement = null;
        }

        function CleanMemory()
        {
                var hostElement = document.getElementById("hostElement");

        // Опять смотрим в Диспетчере Задач на использование памяти

                for(i = 0; i < 5000; i++)
                {
                        var parentDiv = document.
                                createElement("<div onClick='foo()'>");
                        var childDiv = document.
                                createElement( "<div onClick='foo()'>");

        // Изменение порядка имеет значение. Теперь утечек нет
                        hostElement.appendChild(parentDiv);
                        parentDiv.appendChild(childDiv);
                        hostElement.removeChild(parentDiv);
                        parentDiv.removeChild(childDiv);
                        parentDiv = null;
                        childDiv = null;
                }
                hostElement = null;
        }

</script>

<button onclick="LeakMemory()">Вставить с утечками памяти</button>
<button onclick="CleanMemory()">Вставить без утечек</button>

<div id="hostElement"></div>
```

Стоит немного прокомментировать приведенный пример, потому что он противоречит некоторым практическим советам, которые дают относительно написания скриптов для IE. Ключевым моментом в данном случае для осознания причины утечки является то, что DOM-элементы создаются с прикрепленными к ним обработчиками событий. Это является критичным для утечки, потому что в случае обычных DOM-элементов, которые не содержат никаких скриптов, их можно присоединять друг к другу в обычном режиме, не опасаясь проблем, связанных с утечками.

Это позволяет предложить второй метод решения поставленной проблемы, который может быть даже лучше для больших поддеревьев (в нашем примере мы работали только с двумя элементами, но работа с деревом без использования первичного DOM-объекта может быть весьма далека от оптимальной производительности). Итак, второе решение заключается в том, что мы можем создать вначале все элементы без прикрепленных к ним скриптов, чтобы безопасно собрать все поддерево. После этого можно прикрепить полученную структуру к первичному DOM-дереву, перебрать все необходимые узлы и навесить на них требуемые обработчики событий. Помните об опасностях использования циклических ссылок и замыканий, чтобы не вызвать возможные дополнительные утечки памяти, связанные с обработчиками событий.

Но стоит специально заострить внимание на этом пункте, ибо он демонстрирует, что не все утечки памяти так легко можно обнаружить. Возможно, для того, чтобы проблема стала явной, потребуются тысячи итераций. И заключаться она может в сущей мелочи, например, в порядке вставке элементов в DOM-дерево, но результат будет тот же самый: потеря производительности. Если мы стремимся в своей программе опираться только на широко распространенные методы и практические советы ведущих разработчиков, то стоит учесть, что ошибаться могут все, и даже самые хорошие советы могут быть в каких-то аспектах неверными. В данном случае наше решение заключается в улучшении текущего метода или даже создании нового, который бы решал выявленную проблему.

Псевдо-утечки

Очень часто действительное и ожидаемое поведение некоторых API может привести к тому, что ошибочно называется утечками памяти. Псевдо-утечки практически всегда появляются на самой странице при динамических операциях и очень редко бывают замечены вне страницы, на которой происходит выделение памяти, относительно пустой страницы. Можно подумать, что перед нами характерная постраничная утечка, и начать искать ее первопричины — где же происходит перерасход памяти. Как пример такой псевдо-утечки можно привести скрипт для перезаписи текста.

Так же как и ситуация, связанная с добавлением элементов в DOM-дерево, эта проблема опирается на создание временных объектов и приводит к «съеданию» памяти. Переписывая текстовый узел внутри скриптового элемента раз за разом, можно наблюдать, как количество доступной памяти мало-помалу уменьшается из-за различных объектов внутреннего движка, которые были привязаны к предыдущему содержанию. В частности, позади остаются объекты, отвечающие за отладку скриптов, поскольку они полностью принадлежат предыдущему куску кода.

```
<script type=”text/javascript”>

        function LeakMemory()
        {
        // Посмотрим, что происходит с памятью в Диспетчере Задач

                for(i = 0; i < 5000; i++)
                {
                        hostElement.text = "function foo() { }";
                }
        }

</script>

<button onclick="LeakMemory()">Вставить с утечками памяти</button>
```

```
<script id="hostElement">function foo() { }</script>
```

Если мы запустим приведенный код и посмотрим в Диспетчере Задач, что происходит при переходе с «текущей» страницы на чистую, то не увидим никаких утечек. Скрипт расходует память только внутри текущей страницы, и при перемещении на новую вся задействованная память разом освобождается. Ошибка заключается в неверном ожидании определенного поведения. Казалось бы, что переписывание некоторого скрипта приведет к тому, что предыдущий кусок будет бесследно исчезать, оставляя только дополнительные циклические ссылки или замыкания, однако фактически он не исчезает. Очевидно, это псевдо-утечка. В данном случае размер выделенной памяти выглядит устрашающе, но для этого имеется совершенно законная причина.

Проектируем утечки

Каждый веб-разработчик составляет персональный список примеров кода, для которого известно, что он «течет», и пытается найти для каждого случая достойное решение, когда обнаруживает источник проблемы. Это весьма полезно, и именно по этой причине сейчас веб-страницы относительно свободны от утечек памяти. Размышляя о проблемах выделения памяти в терминах шаблонов, а не индивидуальных кусков кода, можно начать внедрять гораздо более продуктивные и более осмысленные решения.

Идея заключается в том, чтобы уже на этапе проектирования приложения мы имели представление о том, какие утечки возможны и как с ними будет лучше работать. Используйте «оборонительную» тактику при разработке и предполагайте, что вся задействованная приложением память должна быть освобождена. Хотя это и преувеличение действительной проблемы, потому что только в очень редких случаях действительно требуется освободить всю память, однако это становится существенным при наличии у переменных и расширяемых свойств потенциальной склонности к утечкам.

7.5. Оптимизируем «тяжелые» JavaScript-вычисления

Наиболее существенным препятствием для выполнения в веб-браузере «тяжелых» вычислений является тот факт, что весь интерфейс пользователя в браузере останавливается и ждет окончания исполнения JavaScript-кода. Это означает, что ни при каких условиях нельзя допускать того, чтобы для завершения работы скрипта требовалось более 300 мс (а лучше, если гораздо меньше). Нарушение этого правила неминуемо ведет к плохому восприятию ресурса пользователем.

К тому же в веб-браузерах у JavaScript-процесса имеется ограниченное время для завершения своего выполнения (это может быть фиксированное число в случае браузеров на движке Mozilla или какое-либо другое ограничение, например, максимальное число элементарных операций в случае Internet Explorer). Если скрипт выполняется слишком долго, то пользователю выводится диалоговое окно, в котором запрашивается, нужно ли прервать скрипт.

Оптимизируем вычисления

Google Gears (http://gears.google.com/) обеспечивает выполнение напряженных вычислений без двух вышеоговоренных ограничений. Однако в общем случае нельзя полагаться на наличие Gears (в будущем было бы замечательно, чтобы решение по типу Gears WorkerPool API стало частью стандартного API браузеров).

К счастью, у глобального объекта есть метод `setTimeout`, который позволяет выполнять определенный код с задержкой, давая тем самым браузеру возможность обработать события и обновить интерфейс пользователя. Это сработает даже в том случае, если задержка для `setTimeout` выставлена в 0, что позволяет разбить долгоиграющий процесс на множество небольших частей. Общий шаблон для обеспечения такой функциональности можно представить в следующем виде:

```
function doSomething (callbackFn [, additional arguments]) {
        // Выполняем инициализацию
        (function () {
                // Делаем вычисления...
                if (конечное условие) {
                        // мы закончили
                        callbackFn();
                } else {
                        // Обрабатываем следующий кусок
                        setTimeout(arguments.callee, 0);
                }
        })();
}
```

Улучшаем шаблон

Этот шаблон можно немного видоизменить, чтобы он обрабатывался не по завершению процесса, а в ходе его исполнения. Это нам очень поможет при использовании индикатора состояния:

```
function doSomething (progressFn [, дополнительные аргументы]) {
        // Выполняем инициализацию
        (function () {
        // Делаем вычисления...
                if (условие для продолжения) {
                // Уведомляем приложение о текущем прогрессе
                        progressFn(значение, всего);
                // Обрабатываем следующий кусок
                        setTimeout(arguments.callee, 0);
                }
        })();
}
```

Советы и замечания

1. Этот шаблон влечет много накладных расходов (на смену контекста исполнения на интерфейс веб-браузера и обратно), поэтому общее время выполнения задачи может быть сильно больше, чем если запустить ее вычисление в обычном режиме.
2. Чем короче каждый цикл, тем больше накладные расходы, тем более интерактивен интерфейс пользователя (он лучше реагирует на действия пользователя), но тем больше общее время выполнения скрипта.
3. Если есть уверенность, что каждая итерация алгоритма занимает совсем немного времени (скажем, 10 мс), тогда можно сгруппировать несколько итераций в одну группу, чтобы уменьшить издержки. Решение, начинать ли новый цикл (прерывать текущий) или сделать еще одну итерацию, должно приниматься на основе того, как долго выполняется весь цикл.
4. **Никогда** не передавайте строку в `setTimeout`! Если передать строку, то браузер будет **каждый раз** выполнять дополнительный `eval` при ее запуске, что, в общем

счете, довольно сильно увеличит суммарное время выполнения скрипта за счет ненужных вычислений.

5. При использовании глобальных переменных в вычислениях перед выходом из очередного цикла убедитесь, что все необходимые данные синхронизированы, чтобы любой другой JavaScript-поток, который может быть запущен между двумя циклами, мог их свободно изменить.

Заключение

Мы можем, в конце концов, выполнять все вычисления такого рода на сервере (хотя в этом случае придется иметь дело с преобразованием данных из одной формы в другую и сетевыми задержками, особенно, если объем данных достаточно велик). Запуск «тяжелых» вычислений на клиенте, **скорее всего**, является признаком глубоких, серьезных архитектурных проблем в нашем приложении.

Также в качестве альтернативного варианта можно рассмотреть отправку каких-либо данных на сервер с помощью XHR-запроса, их обработку и отображение на клиенте. Поскольку JavaScript — интерпретируемый язык в браузере, то он выполняется на несколько порядков дольше серверных аналогов.

7.6. Быстрый DOM

Работа с DOM-деревом в JavaScript является самым проблематичным местом. Его можно сравнить только разве что с базой данных для серверных приложений. Если JavaScript выполняется очень долго, скорее всего, дело именно в DOM-методах. Ниже рассмотрено несколько прикладных моментов, то есть способов максимально ускорить этот «затор».

DOM `DocumentFragment`: быстрее быстрого

`DocumentFragment` является облегченным контейнером для DOM-узлов. Он описан в спецификации DOM1 и поддерживается во всех современных браузерах (был добавлен в Internet Explorer в 6 версии).

В спецификации говорится, что различные операции — например, добавление узлов как дочерних для другого `Node` — могут принимать в качестве аргумента объекты `DocumentFragment`; в результате этого все дочерние узлы данного `DocumentFragment` перемещаются в список дочерних узлов текущего узла.

Это означает, что если у нас есть группа DOM-узлов, которые мы добавляем к фрагменту документа, то после этого можно этот фрагмент просто добавить к самому документу (результат будет таким же, если добавить каждый узел к документу в индивидуальном порядке). Тут можно заподозрить возможный выигрыш в производительности. Оказалось, что `DocumentFragment` также поддерживает метод `cloneNode`. Это обеспечивает нас полной функциональностью для экстремальной оптимизации процесса добавления узла в DOM-дерево.

Давайте рассмотрим ситуацию, когда у нас есть группа узлов, которую нужно добавить к DOM-дереву документа (в тестовой версии это 12 узлов — 8 на верхнем уровне — против целой кучи `div`).

```
var elems = [
        document.createElement("hr"),
        text( document.createElement("b"), "Links:" ),
        document.createTextNode(" "),
        text( document.createElement("a"), "Link A" ),
        document.createTextNode(" | "),
        text( document.createElement("a"), "Link B" ),
        document.createTextNode(" | "),
        text( document.createElement("a"), "Link C" )
];

function text(node, txt){
        node.appendChild( document.createTextNode(txt) );
        return node;
}
```

Нормальное добавление

Если мы собираемся добавить все эти узлы в документ, мы, скорее всего, будем делать это следующим, традиционным, способом: пройдемся по всем узлам и отклонируем их в индивидуальном порядке (таким образом, мы сможем продолжить их добавление по всему документу).

```
var div = document.getElementsByTagName("div");

for ( var i = 0; i < div.length; i++ ) {
        for ( var e = 0; e < elems.length; e++ ) {
                div[i].appendChild( elems[e].cloneNode(true) );
        }
}
```

Добавление при помощи `DocumentFragment`

Однако если мы будем использовать `DocumentFragment` для совершения тех же самых операций, то ситуация изменится. Для начала мы добавим все наши узлы к самому фрагменту (используя имеющийся метод `createDocumentFragment`).

Самое интересное начинается тогда, когда мы собираемся добавить сами узлы в документ: нам нужно вызвать по одному разу `appendChild` и `cloneNode` для всех узлов!

```
var div = document.getElementsByTagName("div");
var fragment = document.createDocumentFragment();

for ( var e = 0; e < elems.length; e++ ) {
        fragment.appendChild( elems[e] );
}

for ( var i = 0; i < div.length; i++ ) {
        div[i].appendChild( fragment.cloneNode(true) );
}
```

При проведении замеров времени можно увидеть следующую картину:

Браузер	Нормальный	Fragment
Firefox 3.0.1	90	47
Safari 3.1.2	156	44

Браузер	Нормальный	Fragment
Opera 9.51	208	95
IE 6	401	140
IE 7	230	61
IE 8b1	120	40

Таблица 13. Сравнение методов работы с DOM-деревом, результаты в миллисекундах

А если еще быстрее?

Давайте подумаем еще немного. Зачем нам каждый раз создавать фрагмент документа, если мы для этой цели можем использовать обычный его узел (фактически, создавать кэш нашего узла, который мы собираемся везде менять)? Так можно прийти к следующему фрагменту кода:

```
var div = document.getElementsByTagName("div");
var child = document.createElement("div");
var parent = div[0].parentNode;

for ( var e = 0; e < elems.length; e++ ) {
        child.appendChild( elems[e].cloneNode(true) );
}

for ( var i = 0; i < div.length; i++ ) {
// для IE
        if (IE) {
                parent.replaceChild(child.cloneNode(true),div[i]);
// для других браузеров
        } else {
                div[i] = child.cloneNode(true);
        }
}
```

В нем соответствующие узлы документа заменяются на клонированный вариант кэшированной версии (без создания `DocumentFragemnt`). Это работает еще быстрее (везде, кроме IE — примерно на порядок, в IE — в полтора–два раза).

`innerHTML` нам поможет

Чтобы уменьшить отрисовку отдельных частей документа в процессе добавления какого-либо большого фрагмента, мы можем сохранять HTML-код в виде текста и лишь на финальном этапе вставлять его в DOM-дерево. Давайте рассмотрим следующий пример:

```
var i, j, el, table, tbody, row, cell;
el = document.createElement("div");
document.body.appendChild(el);
table = document.createElement("table");
el.appendChild(table);
tbody = document.createElement("tbody");
table.appendChild(tbody);
for (i = 0; i < 1000; i++) {
        row = document.createElement("tr");
        for (j = 0; j < 5; j++) {
                cell = document.createElement("td");
                row.appendChild(cell);
```

```
        }
        tbody.appendChild(row);
}
```

Его можно значительно ускорить, если добавлять узлы не последовательно один за другим, а сначала создав HTML-строку со всем необходимым кодом, которая будет вставлена через `innerHTML` в конце всех операций.

В данном примере кроме уже указанного ускорения еще используется первоначальное создание массива элементов, которые можно объединить через свойство `join` в строку. Для больших строк это работает быстрее, чем последовательная конкатенация отдельных частей.

```
var i, j, el, idx, html;
idx = 0;
html = [];
html[idx++] = "<table>";
for (i = 0; i < 1000; i++) {
        html[idx++] = "<tr>";
        for (j = 0; j < 5; j++) {
                html[idx++] = "<td></td>";
        }
        html[idx++] = "</tr>";
}
html[idx++] = "</table>";
el = document.createElement("div");
document.body.appendChild(el);
el.innerHTML = html.join("");
```

7.7. Кэширование в JavaScript

Очень часто в JavaScript используют глобальные объекты и переменные для чтения каких-либо параметров (или вызова определенных методов). Почти всегда этого можно избежать, если кэшировать объект из глобальной области видимости в локальную — все обращения к закэшированному объекту буду выполняться намного быстрее.

Итерации и локальное кэширование

При DOM-операциях перебор массива объектов является довольно типичной задачей. Давайте предположим, что вы разрабатываете HTML-приложение, которое индексирует содержание страниц. Нашей задачей является сбор всех элементов `h1` на текущей странице, чтобы затем использовать их в проиндексированном массиве.

Ниже приведен пример, как это можно осуществить:

```
function Iterate(aEntries) {
        for (var i=0; i < document.getElementsByTagName('h1').length; i++) {
                aEntries[aEntries.length] =
                        document.getElementsByTagName('h1')[i].innerText;
        }
}
```

Что плохого в приведенном примере? Он содержит два обращения к массиву `document.getElementsByTagName('h1')` на каждой итерации. Внутри цикла наш скрипт будет:

- вычислять размер массива;
- получать значение свойства `innerText` для текущего элемента в массиве.

Это совершенно не эффективно. Дополнительные вычисления, связанные с многочисленными ненужными обращениями к DOM-дереву для получения информации, которую мы и так знаем, являются совершенно лишними. Давайте рассмотрим следующую модификацию этого скрипта:

```
function Iterate2(aEntries) {
        var oH1 = document.getElementsByTagName('h1');
        var iLength = oH1.length;
        for (var i=0; i < iLength; i++) {
                aEntries[aEntries.length] = oH1(i).innerText;
        }
}
```

Таким образом, мы кэшируем DOM-массив в локальную переменную, и затем все действия над ней производятся гораздо быстрее. N обращений к DOM-дереву превращается всего в одно-единственное в результате использования кэширования.

Кэширование ресурсоемких вызовов

Как показывает практика, лучше всего будет использовать переменные максимально близко к области их объявления и избегать использования глобальных переменных любой ценой. Глобальная область видимости обычно содержит десятки, если не сотни объектов, которые добавлены браузером согласно спецификации и для собственных нужд, поэтому обращения к глобальным переменным всегда ресурсоемки.

Также стоит с осторожностью использовать ключевое слово `with`, так как оно не дает компилятору генерировать код для быстрого доступа к локальным переменным (ему приходится сначала пробежаться по цепочке прототипа объекта, затем по цепочке вышестоящей области видимости и т.д.).

Если у вас есть примерно такой участок кода

```
var arr = ...;
var globalVar = 0;
(function () {
        var i;
        for (i = 0; i < arr.length; i++) {
                globalVar++;
        }
})();
```

То его можно оптимизировать следующим образом:

```
var arr = ...;
var globalVar = 0;
(function () {
        var i, l, localVar;
        l = arr.length;
        localVar = globalVar;
        for (i = 0; i < l; i++) {
                localVar++;
        }
```

```
        globalVar = localVar;
})();
```

В этом примере мы уменьшили число обращений к глобальной переменной и устранили расчет размера массива на каждой итерации цикла.

Кэшируем цепочки вызовов

Распознавание (разрешение) ссылки на объект или метод выполняется каждый раз, когда происходит обращение к этому объекту или методу. Переменные разрешаются всегда в обратном порядке: от более частной области видимости к более общей. Поэтому, если у нас есть примерно следующий код

```
for (i=0; i < 10000; i++) a.b.c.d(v);
```

то он будет выполняться несколько медленнее, чем

```
var f=a.b.c.d;
for (i=0; i < 10000; i++) f(v);
```

или

```
var f=a.b.c;
for (i=0; i < 10000; i++) f.d(v);
```

На данный момент браузеры хорошо справляются с кэшированием вызовов функций, и особого прироста производительности при такой оптимизации в IE и Firefox не наблюдается. При проектировании приложений для других браузеров стоит учитывать и этот аспект, однако, с большой вероятностью кэширование вызовов объектов уже добавлено (или будет добавлено в самое ближайшее время) в разрабатываемые JavaScript-движки, ибо это одно из наиболее узких мест в производительности веб-приложений.

Наконец, при использовании кэширования (особенно, частей DOM-дерева) стоит помнить, что оно уменьшает использование CPU, но увеличивает расходование памяти (и может привести к псевдо-утечкам), поэтому в каждом конкретном случае нужно выбирать меньшее из двух зол.

7.8. Быстрые итераторы, регулярные выражения и другие вкусности

В этом разделе собраны некоторые практические советы по производительности отдельных конструкций в JavaScript-движках в браузере.

Итераторы

Давайте рассмотрим, какой способ перебора элементов будет максимально быстрым в JavaScript. У нас есть несколько возможностей, ниже приведен полный вариант кода для тестирования.

```
<!-- набор элементов для результатов тестирования -->
<p id="test1"></p>
```

```
<p id="test2"></p>
<p id="test3"></p>
<p id="test4"></p>
<p id="test5"></p>
<p id="test6"></p>

<script type="text/javascript">
// выбираем все элементы из DOM-дерева
var items = document.getElementsByTagName("*");
// кэшируем текущий размер DOM-дерева
var length = items.length;
// запоминаем текущий момент времени
var time = new Date().getTime();

// запускаем первый тест, обычный перебор элементов массива,
// запускается 10000 раз
for (var j=0; j<10000; j++) {
        for (var i=0; i<items.length; i++) {
                var item = items[i];
        }
}
// выводим результат в подготовленный выше контейнер
document.getElementById('test1').innerHTML =
        "Простой цикл: " + (new Date().getTime() - time);

time = new Date().getTime();

// кэшируем размер массива
for (var j=0; j<10000; j++) {

        for (var i=0; i<length; i++) {

                var item = items[i];

        }

}
document.getElementById('test2').innerHTML =
        "Простой цикл (с кэшированием): " + (new Date().getTime() - time);

time = new Date().getTime();

// встроенный for-in итератор для объекта массива
for (var j=0; j<10000; j++) {
        for (var i in items) {
                var item = items[i];
        }
}
document.getElementById('test3').innerHTML =
        "Простой через for-in: " + (new Date().getTime() - time);

time = new Date().getTime();

// обратный перебор элементов массива
for (var j=0; j<10000; j++) {
        for (var i = length - 1; i >= 0; i--) {
                var item = items[i];
        }
}
document.getElementById('test4').innerHTML =
        "Обратный: " + (new Date().getTime() - time);

time = new Date().getTime();
```

```
// итератор do-while
for (var j=0; j<10000; j++) {
        var i = 0;
        do {
                var item = items[i];
                i++;
        } while (i < length)
}
document.getElementById('test5').innerHTML =
        "do-while: " + (new Date().getTime() - time);


time = new Date().getTime();

// обратный while (самый быстрый)
for (var j=0; j<10000; j++) {
        var i = length - 1;
        while (--i) {
                var item = items[i];
        }
}
document.getElementById('test6').innerHTML =
        "Обратный while: " + (new Date().getTime() - time);
</script>
```

В результате мы получим примерно следующую таблицу.

Браузер	Обычный	С кэшем	for-in	Обратный	do-while	Обратный while
Firefox 3.0.3	714	657	835	280	297	217
Safari 3.1.2	141	140	157	125	125	93
Opera 9.61	188	125	765	94	94	78
IE 6	1281	1219	1094	468	500	360
IE 7	1391	1297	1250	515	532	406
IE 8b2	954	906	922	406	422	328
Chrome 0.2	288	246	332	117	114	95

Таблица 14. Различные варианты перебора массива, результаты в миллисекундах

В общем случае применение обратного `while` для перебора цикла в 2–3 раза быстрее всех остальных вариантов. Если веб-приложение оперирует массивами порядка 1000 элементов, то в результате примнения оптимизированных приемов будет заметен значительный прирост производительности.

Регулярные выражения

В JavaScript есть несколько способов проверить, удовлетворяет ли строка заданному шаблону:

```
// 1. Объявляем объект в виде регулярного выражения
var RegExp = '/script/gi';
// и ищем в элементе массива совпадение с заданным шаблоном
items[i].nodeName.search(RegExp);

// 2. можно просто проверять соответствие строке,
```

```
// а не искать индекс подстроки
items[i].nodeName.match(RegExp);

// 3. Можно обойтись без объявления самого регулярного выражения
items[i].nodeName.match(/script/gi);

// 4. Можно задавать регулярное выражение без глобального модификатора,
// ведь мы ищем любое (=первое) совпадение шаблона
items[i].nodeName.match(/script/i);

// 5. С тем же успехом мы можем выполнить шаблон
/script/i.exec(items[i].nodeName);

// 6. Наконец, можно протестировать сам шаблон на нахождение в строке
/script/i.test(items[i].nodeName);
```

Давайте рассмотрим, что из них работает быстрее всего. Для этого запустим немного модифицированный набор тестов из раздела выше (опять по 10000 раз для всего DOM-дерева). Получим следующие результаты:

Браузер	search	match	«На лету»	Локальный	exec	test
Firefox 3.0.3	2120	2041	1295	1273	1225	1348
Safari 3.1.2	453	469	344	359	360	359
Opera 9.61	2141	2063	406	344	312	313
IE 6	2594	2516	1875	1859	1953	1906
IE 7	2562	2469	1859	1844	2000	1860
IE 8b2	2140	2032	1453	1453	1547	1469
Chrome 0.2	856	870	416	397	385	392

Таблица 15. Различные варианты выполнения регулярного выражения, результаты в миллисекундах

Как мы видим, в данном случае создание нового регулярного выражения — весьма ресурсоемкий процесс, поэтому в большинстве случаев лучше обходиться без него. В остальном все браузеры ведут себя достаточно ровно при сопоставлении `match`, `exec` и `test`.

Глава 8. Приложение

8.1. Обзор аналитических инструментов

Давайте рассмотрим ряд приемов, позволяющих самостоятельно провести анализ производительности сайта. К ним можно отнести как проверку времени создания страницы на сервере, так и измерение фактического времени загрузки страницы у пользователя.

Измеряем эффективную ширину канала пользователей

Можно легко измерить реальную пропускную способность канала для пользователей исследуемого сайта. И в том случае, если пользователи загружают страницу существенно медленнее, чем могли бы (учитывая физические ограничения на их канал), возможно, стоит применить меры к исправлению этой ситуации.

Прежде чем давать браузеру любые ссылки на внешние объекты (`<img src="...">`, `<link rel="stylesheet" href="...">`, `<script src="...">` и т.д.), мы можем записать текущее время. После загрузки всей страницы можно будет его вычесть и получить, таким образом, полное время загрузки страницы (за исключением получения HTML-файла и задержек, связанных с первым соединением с сервером). Полученное время можно затем добавить к вызову любого URL (например, картинки), расположенной на вашем сервере.

JavaScript-код для этого будет выглядеть примерно следующим образом:

```
<html>
<head>
<title>...</title>
<script type="text/javascript">
<!--
var began_loading = new Date().getTime();

window.onload = function(){
  new Image().src = '/timer.gif?u=' + self.location + '&t=' +
  ((new Date().getTime() - began_loading) / 1000);
};
// -->
</script>
<!--
Здесь будут размещаться ссылки на любые внешние JS- или CSS-файлы,
главное, чтобы они шли ниже верхнего блока
// -->
</head>
<body>
<!--
Здесь идет обычное содержание страницы
// -->
</body>
</html>
```

Эта конструкция произведет примерно следующую запись в лог-файл:

```
127.0.0.1 - - [28/Oct/2006:13:47:45 -0700]
  "GET /timer.gif?u=http://example.com/page.html&t=0.971 HTTP/1.1" 200 49 ...
```

В этом случае, как можно понять из записи, загрузка оставшейся части страницы `http://example.com/page.html` заняла у пользователя 0,971 секунды. Если предположить, что всего на странице было загружено файлов общего размера в 57842 байтов, 57842 байтов * 8 битов в байте / 0,971 секунды = 476556 битов в секунду (4б5 Кбит). Такова эффективная пропускная способность канала при загрузке этой страницы. Если у пользователя физический входящий канал 1,5 Мб, значит, есть большой простор для увеличения скорости загрузки.

После того, как будет собрана некоторая статистика по времени загрузки страницы и эффективной ширине канала для реальных пользователей, можно поэкспериментировать с изменениями, которые способны изменить эти показатели к лучшему. В случае значительных улучшений этого показателя стоит закрепить внесенные изменения.

Apache Benchmark и JMeter

Утилита `ab` из пакета для Apache (устанавливается под Linux обычно вместе с самим Apache) позволяет устроить простой тест для производительности сервера. Стоит заметить, что это будет чисто серверная производительность, а не клиентская.

Мы можем запустить тест с помощью команды

```
ab -c10 -n1000 http://www.website.ru/
```

Вышеуказанным способом мы запускаем стресс-тест для сайта `www.website.ru`. При проведении тестирования главная страница сайта будет скачана 1000 раз (модификатор `-n1000`) с использованием 10 параллельных соединений (модификатор `-c10`). Если запустить такой же тест для статических файлов, то можно диагностировать медленное обслуживание обычных запросов (обычно статические файлы отдаются со скоростью порядка 1000 в секунду). Если же сервер отвечает дольше, чем 5–10 миллисекунд при генерации страницы, значит, стоит хорошо разобраться, на что уходит процессорное время.

Основная характеристика здесь — это время, ушедшее на установление соединения. Например, следующее значение будет говорить о хорошей реакции сервера на пользовательские запросы:

```
Time per request:    40.599 [ms]

Connection Times (ms)
       min mean[+/-sd] median  max
Connect:    18    40    12.5  38    61
Processing: 0     0     0.3   0     1
Waiting:    0     0     0.3   0     1
Total:      18    41    12.4  38    61
```

В данном случае на один запрос, в среднем, ушло 41 мс, из них почти все время было затрачено на установление соединения и загрузку данных. Ответа от сервера, практически, ждать не приходилось.

К такому нагрузочному тестированию стоит подходить крайне осмотрительно: не запускать его на том же самом сервере, где располагается исследуемый сайт (ибо `ab` сам по себе достаточно «тяжелый» и будет делить физические ресурсы машины с Apache).

Если в результате таких тестов задержки оказались весьма высокими и процесс веб-сервера (или CGI) «отъедал» слишком много CPU, то причиной этого зачастую может оказаться необходимость в компиляции скриптов в процессе выполнения при каждом запросе.

Для более мощного и точного тестирования стоит посмотреть в сторону другой программы — Apache JMeter, которая предназначенная для нагрузочного тестирования различных сетевых сервисов и приложений, не ограничиваясь только HTTP. Это, по сути, универсальная и более продвинутая замена `ab`, только более требовательная к пользователю, так как содержит ряд мощных инструментов, которые может быть под силу правильно применить только профессионалу. Кстати, JMeter позволяет не только тестировать веб-приложения, но и любые другие сетевые сервисы — SMTP, POP и даже базы данных через JDBC. Если в планах стоит постоянное использование тестирования, то стоит обратить внимание именно на это приложения: из бесплатных и открытых — это, пожалуй, самое гибкое и мощное решение.

Кэширование на сервере

Такое программное обеспечение, как eAccelerator/xCache/ZendOptimizer для PHP, `mod_perl` для perl, `mod_python` для python и др., могут кэшировать серверные скрипты в скомпилированном состоянии, существенно ускоряя загрузку нашего сайта. Кроме этого, стоит найти профилирующий инструмент для используемого языка программирования, чтобы установить, на что же тратятся ресурсы CPU. Если удастся устранить причину больших нагрузок на процессор, то страницы будут отдаваться быстрее и появится возможность выдавать больше трафика при меньшем числе машин.

Если на сайте при создании страницы выполняется много запросов к базе данных или какие-либо другие тяжелые вычисления, стоит рассмотреть добавление кэширования на стороне сервера для медленных операций. Большинство людей начинают с записи кэша в локальную память или на локальный диск, однако эта логика перестает работать, если система расширяется до кластера веб-серверов (каждый со своим локальным диском и локальной памятью).

Можно также рассмотреть использование memcached для кэширования на стороне сервера. Memcached создает очень быстрый общий кэш, который объединяет свободную оперативную память всех имеющихся машин. Клиенты к нему перенесены на большинство распространенных языков.

Web Developer Toolbar для Firefox

Данный инструмент позволяет проанализировать размеры всех загруженных на странице файлов (чтобы его открыть, нужно кликнуть или выбрать через правый клик панель инструментов `Web Developer -> Information -> View Document Size`). Можно посмотреть на список файлов и оценить, что отнимает большую часть времени при загрузке страницы:

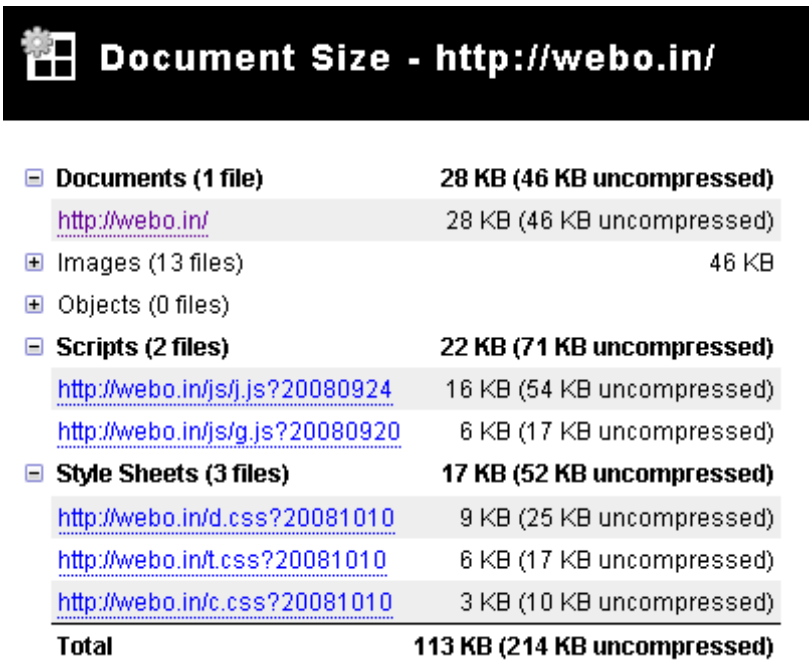


Рис. 38. Результаты анализа загрузки сайта в Web Developer Toolbar

К сожалению, здесь размер встроенных изображений отображается неверно. И нет данных о том, в какую стадию загрузки попал той или иной файл. Работает только под Firefox.

Firebug NET Panel для Firefox

Другим, более популярным инструментом для анализа загрузки сайта в Firefox является Firebug (со встроенной NET Panel). Он отслеживает все пакеты, которые передает или запрашивает Firefox, позволяя тем самым построить вполне точную диаграмму загрузки страницы. Естественно, позволяет увидеть и все HTTP-заголовки (как запроса, так и ответа) для полученных файлов. К сожалению, на данный момент Firebug не учитывает время, затраченное на DNS-запросы, редиректы и отображение страницы.

Однако ситуация обещает исправиться, да еще и кардинальным образом: уже сейчас доступна альфа-версия Firebug 1.4a1, в которой для каждого загружаемого объекта страницы теперь выводится полная статистика затрачиваемого времени. Конечно, есть еще куда стремиться: можно добавить и общую диаграмму загрузки, и затраты времени на все компоненты вместе, а не по отдельности. Но и этот шаг будет весьма значительным.

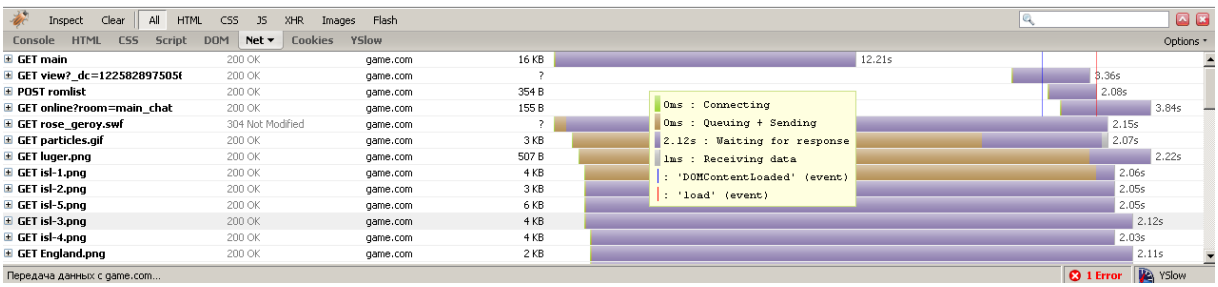


Рис. 39. Результаты анализа загрузки сайта в Firebug Net Panel 1.4a1

На этой диаграмме теперь можно увидеть, когда же наша страница считается загруженной как частично, так и полностью (первая и третья стадия загрузки, соответственно): две вертикальные линии синего и красного цвета показывают моменты событий `DOMContentLoaded` и самого `load`. Очевидно, что после события `load` могут происходить дополнительные загрузки компонентов, которые находятся в четвертой стадии, что, естественно, не всегда совпадает с фактической загрузкой и отображением компонентов страницы.

Но давайте рассмотрим загрузку страницы в текущей версии инструмента:

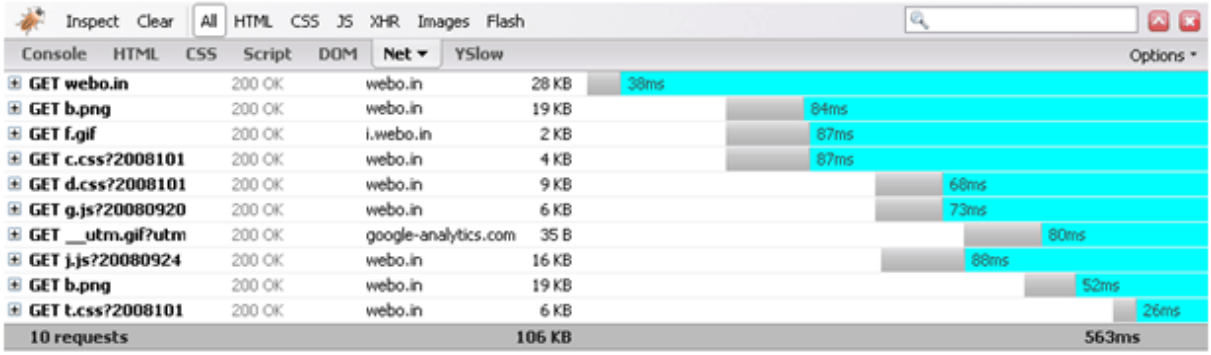


Рис. 40. Результаты анализа загрузки сайта в Firebug Net Panel

На диаграмме загрузки для webo.in хорошо отслеживается стадия предзагрузки (заканчивающаяся с получением файла `c.css?20081010`). На этой же стадии у пользователя оказываются загруженными 2 изображения (оба запрошены через `new Image().src`, одно из них — «на будущее). После того, как страница появилась в браузере пользователя (на это ушло порядка 200 мс с момента запроса страницы), сработало событие готовности документа к дальнейшим действиям. По этому событию Firefox запросил 3 файла: `d.css?20081010`, `g.js?20080920` и `j.js?20080924`.

`g.js` (являясь сжатым скриптом Google Analytics) отправил данные о посещении на сервер статистики с помощью файла `__utm.gif` . Стоит заметить, что все вызовы внешних ресурсов из HTML-файла осуществлялись при использовании DOM-методов добавления элементов, и это позволило максимально их распараллелить. Далее Firefox (так как кэш был отключен) запросил файл `b.png` повторно (основываясь уже на данных из файла `d.css`, содержащего информацию о стилях для фона элементов). При наличии в кэше файл просто отобразился бы на странице, и запроса не произошло.

После получения всех файлов на странице сработало событие `window.onload`, по которому загрузился последний файл (скрытая таблица стилей, используемая на других страницах сайта). Для пользователя это произошло абсолютно прозрачно: страница уже была полностью готова к взаимодействию, не требовалось никакого дополнительного времени ожидания.

В данном случае размер страницы составляет порядка 100 Кб в сжатом виде и около 200 Кб в несжатом. Однако это не помешало ей загрузиться (если не брать в расчет некоторые файлы «на будущее» и отключенное кэширование) менее чем за 500 мс.

Yslow для Firebug для Firefox

В качестве полезного дополнения к Firebug (для Firefox) стоит рассмотреть и YSlow. На данный момент этот инструмент, пожалуй, является наиболее адекватным для анализа скорости загрузки страницы.

Все аспекты производительности разбиты на разделы. В случаем невыполнения каких-либо советов дается ссылка на полный его вариант на английском языке. Естественно, что советы не идеальны, и в некоторых случаях допустимо не следовать им буквально. Однако при прочих равных условиях более высокая оценка (максимум 100) соответствует более оптимизированному сайту.

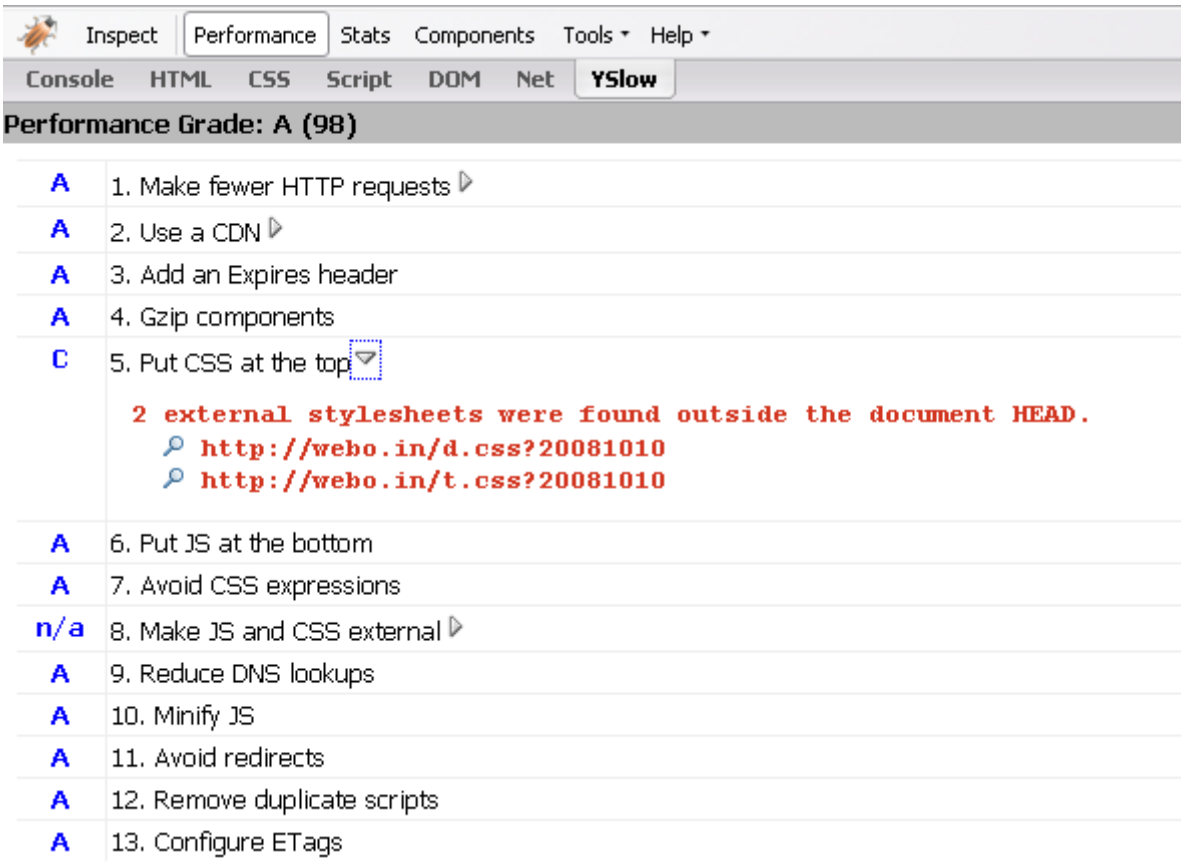


Рис. 41. Результаты анализа загрузки сайта в YSlow

Web Inspector для Safari

Аналогично уже рассмотренной Firebug Net Panel, Web Inspector представляет диаграмму загрузки, основываясь на фактических данных. Однако есть и ряд недостатков: в частности, время отображения (выполнения) элемента не отделено от времени его загрузки, что хорошо видно для встроенных изображений.

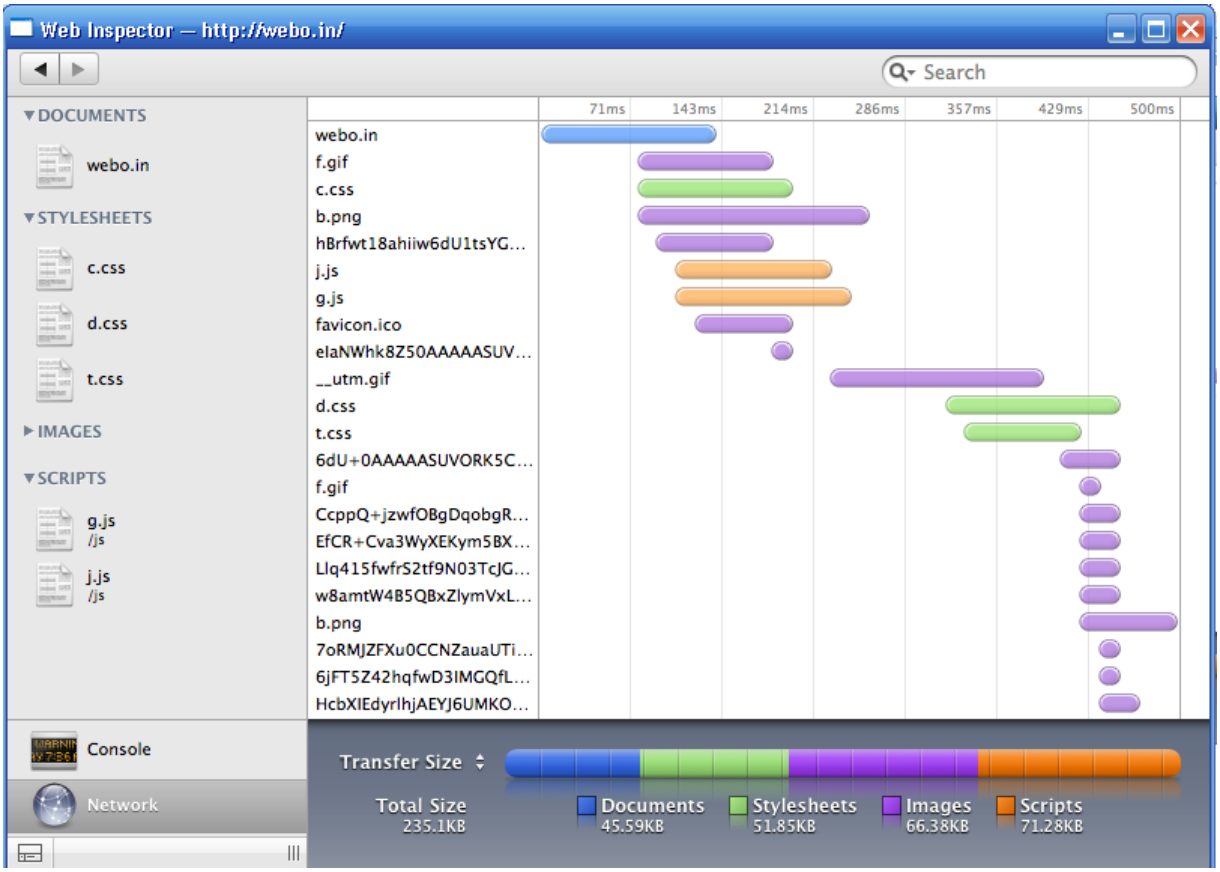


Рис. 42. Результаты анализа загрузки сайта в Web Inspector

HttpWatch

HttpWatch (http://www.httpwatch.com/) может быть установлен как для IE, так и для Firefox. На данный момент кроме самих HTTP-заголовков он предоставляет достаточно подробную диаграмму загрузки сайта, что является хорошим подспорьем при анализе производительности.

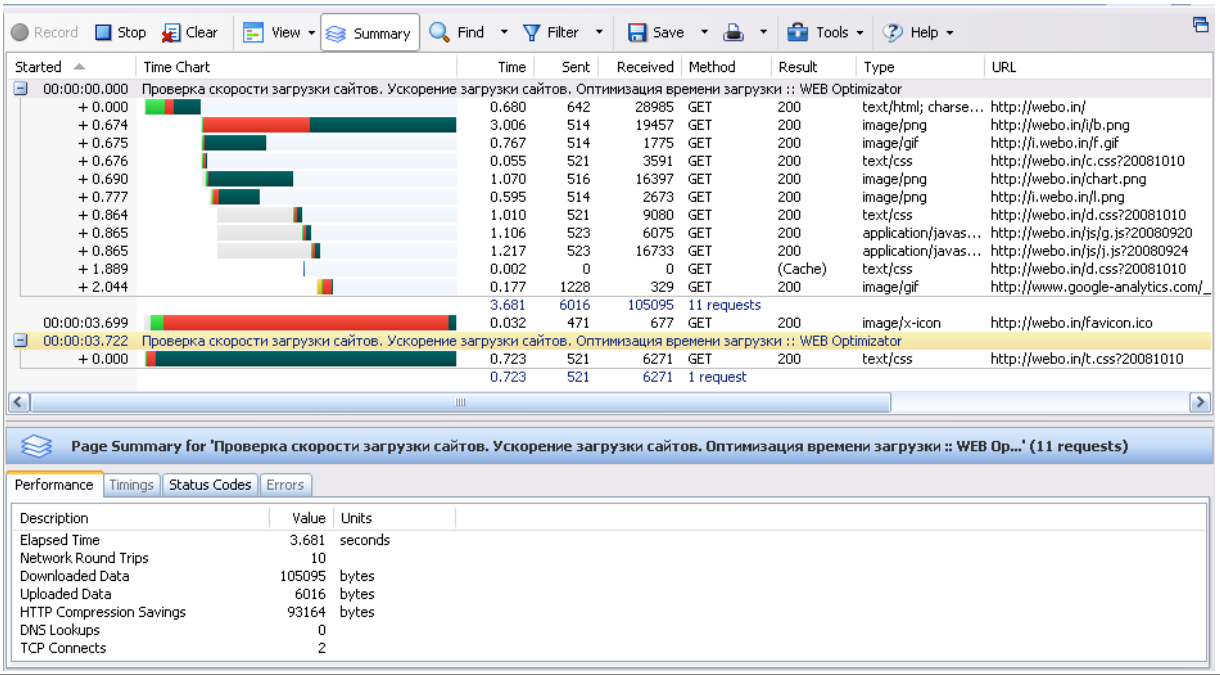


Рис. 43. Результаты анализа загрузки сайта в Http Watch

Полная версия продукта является платной.

Fiddler

Fiddler (http://www.fiddlertool.com/fiddler/) устанавливается как дополнение к IE и позволяет анализировать все загружаемые файлы (заголовки, размер, время загрузки из разных точек земного шара).

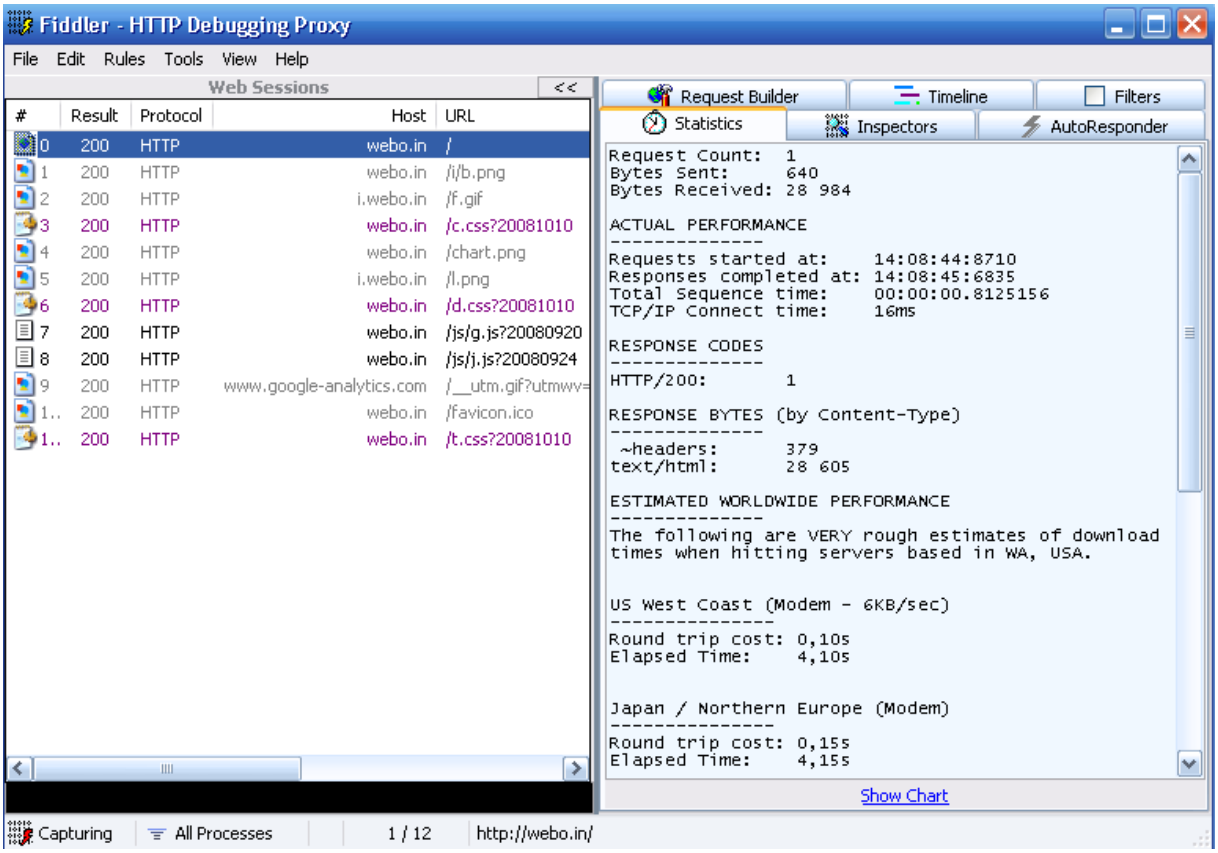


Рис. 44. Результаты анализа загрузки сайта в Fiddler

Live HTTP Headers

Live HTTP Headers (http://livehttpheaders.mozdev.org/) позволяет просматривать HTTP-заголовки для Firefox в режиме реального времени. Может выступать достаточно удобным дополнением в Firefox, если нужно отладить общение браузера с сервером в плане кэширования или сжатия (проверить соответствующие заголовки на «живом» соединении).

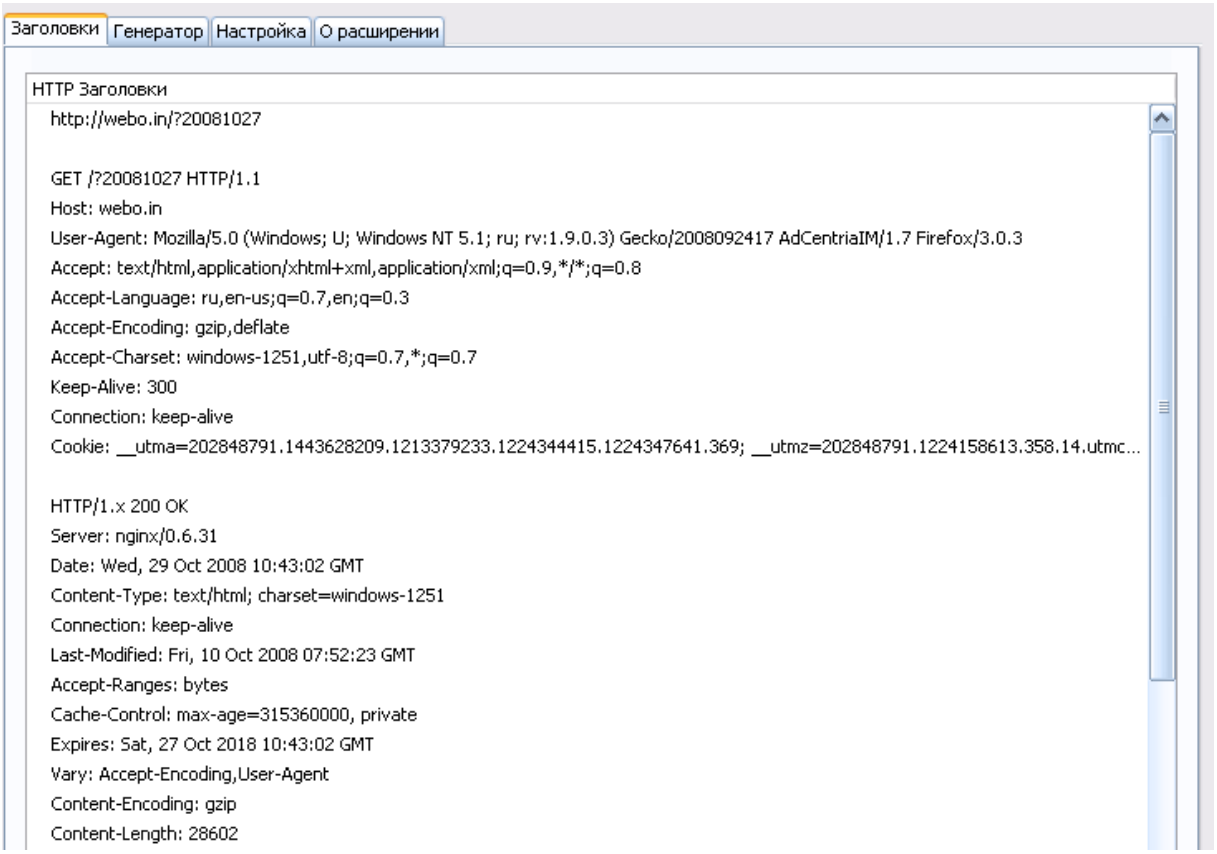


Рис. 45. Заголовки запроса и ответа в Live HTTP Headers для webo.in

Прокси-эмулятор каналов Sloppy

После рассмотрения всех методов ускорения загрузки стоит заметить, что та или иная техника оптимизации направлена, в первую очередь, на пользователей с медленным каналом: именно они в наибольшей мере почувствуют выигрыш от наших кропотливых манипуляций с проектом. Но как же проверить это еще на этапе разработки — ведь в большинстве случаев тестовый сервер располагается на рабочем компьютере, да и интернет-канал у профессиональных веб-разработчиков тоже не самый плохой. Выход есть: эмулировать такие соединения, подключаясь через специальную программу, которая нарочно «затормозит» (но не интернет, а только доступ к тестовому сайту) соединение, имитируя подключение пользователя, скажем, с ADSL-128 Kbps.

Для этого нам подойдет небольшая и очень простая программа Sloppy — прокси-сервер. Он эмулирует доступ к указанному сайту через канал с задаваемой полосой пропускания: от модемного 9,6 Кб/с до выделенного в 512 Кб/с. В том случае, если скорость подключения к интернету 1 Мб или больше, любой проект будет загружаться достаточно быстро, поэтому тестировать его специально не имеет смысла (только в общем порядке). А влияние издержек на установление множества дополнительных соединений можно установить при тестировании на менее мощных каналах.

Из доступных настроек у нас есть: адрес сайта, который будем тестировать, выбор скорости (из набора 9,6, 14.4, 28.8, 56, 128, 256 и 512 Кб), а также порт, по которому мы будем получать страницу. Благодаря «прокси-природе», его можно использовать для

тестирования как локального проекта, так и любого внешнего проекта в сети. Конечно, в этом случае нужен доступ в интернет, тогда как просто для теста локального сервера этого совсем не требуется (после загрузки самого приложения).

Sloppy интересен еще и тем, что распространяется как JNLP-файл, то есть использует Java Web Start для запуска; при этом сам код загружается с сайта проекта, впрочем, можно загрузить исходный код отдельно.

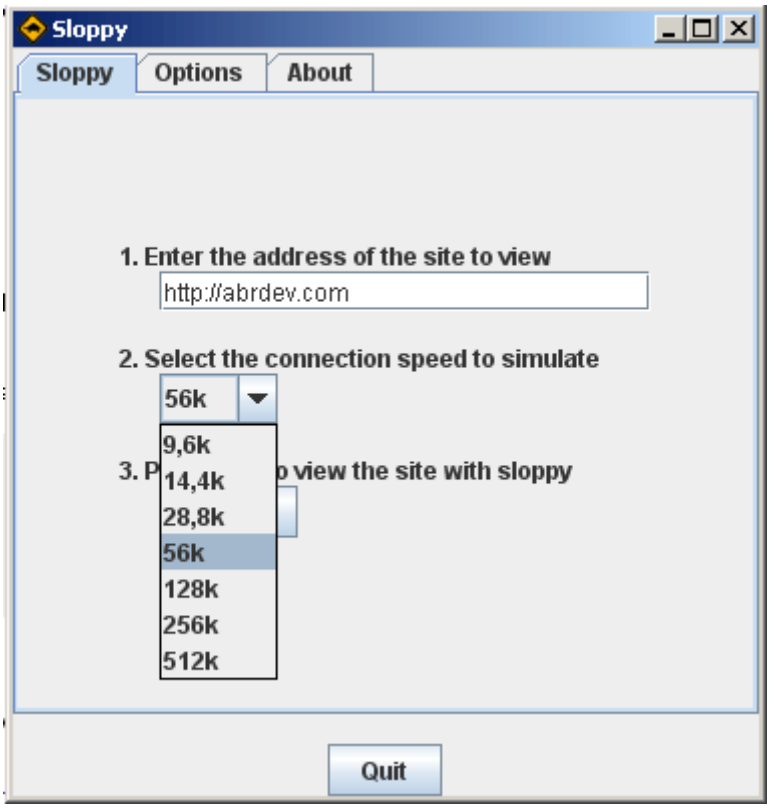


Рис. 46. Настройки Sloopy для тестирования загрузки сайта на медленном канале

Analyze.WebSiteOptimization.com

Пожалуй, самый старый и наиболее известный онлайн-сервис для проверки клиентской оптимизации выбранного сайта. Выдает набор советов (аналогичных рекомендациям Yahoo). Поскольку анализ основан на собственном алгоритме, то не распознает `data:URI` и mhtml-изображения. Также не всегда верно трактует скрипты внутри страницы.

Web Page Speed Report

URL:	http://webo.in/
Title:	Проверка скорости загрузки сайтов. Ускорение загрузки сайтов. Оптимизация времени загрузки :: WEB Optimizator
Date:	Report run on Wed Oct 29 05:23:52EDT2008

Diagnosis

Global Statistics

Total HTTP Requests:	14
Total Size:	68648 bytes

Object Size Totals

Object type	Size (bytes)	Download @ 56K (seconds)	Download @ T1 (seconds)
HTML:	28602	5.90	0.35
HTML Images:	185	0.24	0.20
CSS Images:	20751	6.14	2.11
Total Images:	20936	6.38	2.31
Javascript:	19110	4.21	0.50
CSS:	0	0.00	0.00
Multimedia:	0	0.00	0.00
Other:	0	0.00	0.00

Рис. 47. Результаты анализа загрузки сайта в analyze.websiteoptimization.com

Octagate.com/service/SiteTimer/

С помощью данного инструмента можно построить диаграмму загрузки сайта. К плюсам можно отнести то, что дополнительно показан RSS-поток (при соответствующем объявлении). К несчастью, сервис не распознает `data:URI` и mhtml-изображения; также построение самой диаграммы загрузки оставляет желать лучшего.

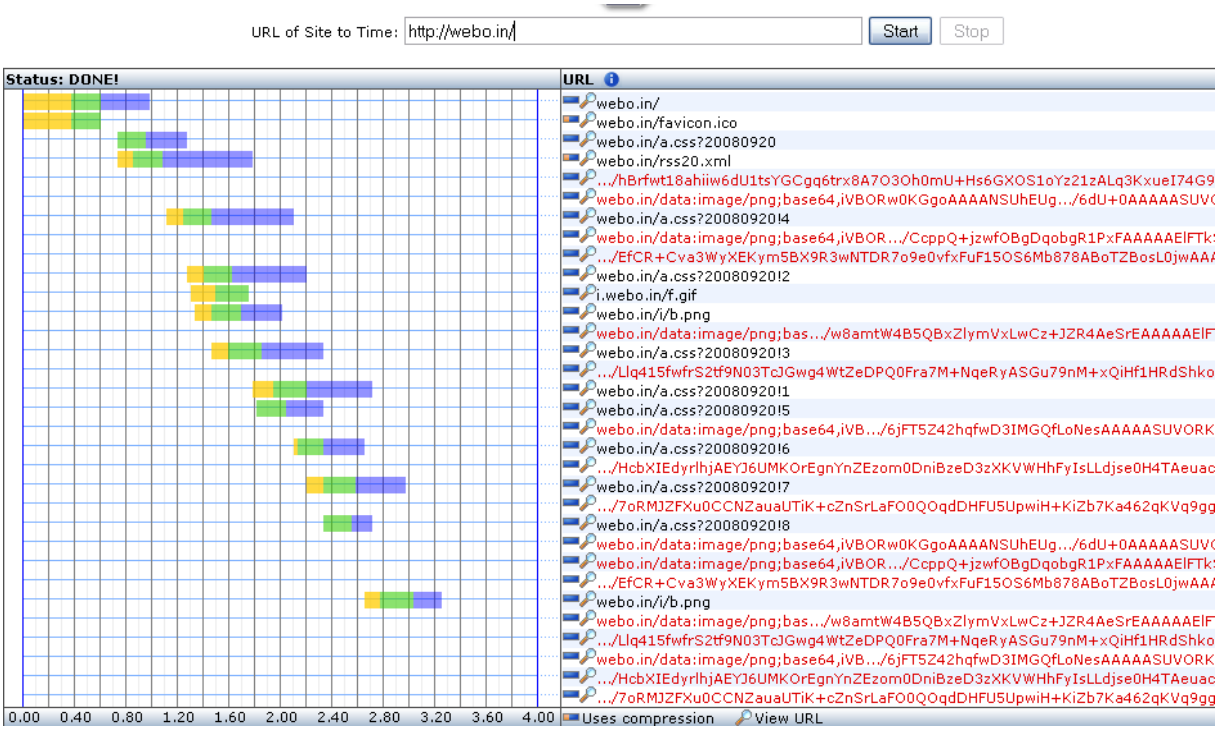


Рис. 48. Результаты анализа загрузки сайта в octagate.com/service/sitetimer/

Tools.Pingdom.com

Сервис позиционирует себя как инструмент для построения диаграммы загрузки, однако, на данный момент не распознает сжатия файлов и выделения фоновых картинок.

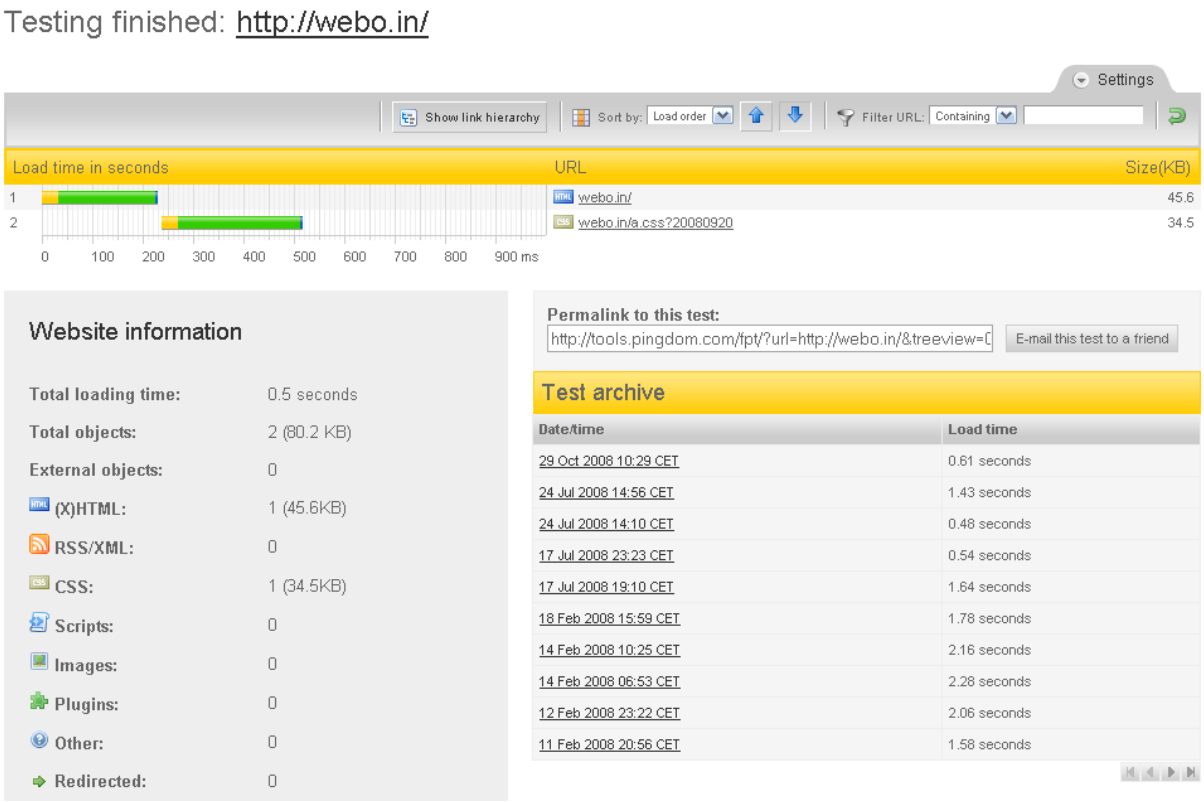
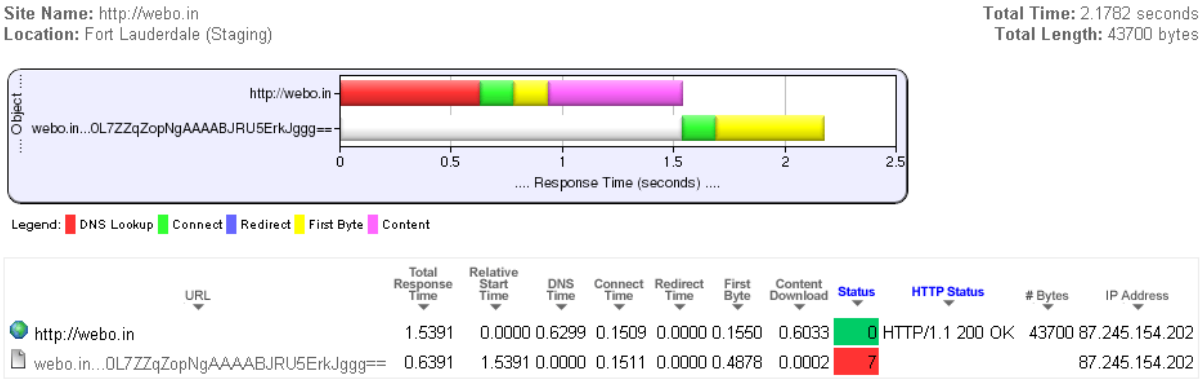


Рис. 49. Результаты анализа загрузки сайта в tools.pingdom.com

AlertSite.com

Сервис позиционирует себя как инструмент для построения диаграммы загрузки, однако, на данный момент не распознает сжатия файлов, `data:URI` и `mhtml`-файлы. Также есть проблемы с распознаванием таблиц стилей. Однако, для всех файлов выводится достаточно большое параметров, характеризующих процесс загрузки.



URL	Total Response Time	Relative Start Time	DNS Time	Connect Time	Redirect Time	First Byte	Content Download	Status	HTTP Status	# Bytes	IP Address
http://webo.in	1.5391	0.0000	0.6299	0.1509	0.0000	0.1550	0.6033	0	HTTP/1.1 200 OK	43700	87.245.154.202
webo.in...0L7ZZqZopNgAAAABJRU5ErkJggg==	0.6391	1.5391	0.0000	0.1511	0.0000	0.4878	0.0002	7			87.245.154.202

Рис. 50. Результаты анализа загрузки сайта в www.alertsite.com/cgi-bin/tsite3.pl

Site-Perf.com

Относительно недавно появившийся сервис, который предлагает комплексный анализ скорости загрузки сайта. Он позволяет оценить как узкие места при загрузке, так и общий «вес» страницы. Возможно моделирование процесса загрузки при использовании нескольких параллельных соединений. Также есть возможность выбрать одну из нескольких тестовых точек.

К минусам можно отнести невозможность распознавания динамических файлов стилей и скриптов, а также условных комментариев.

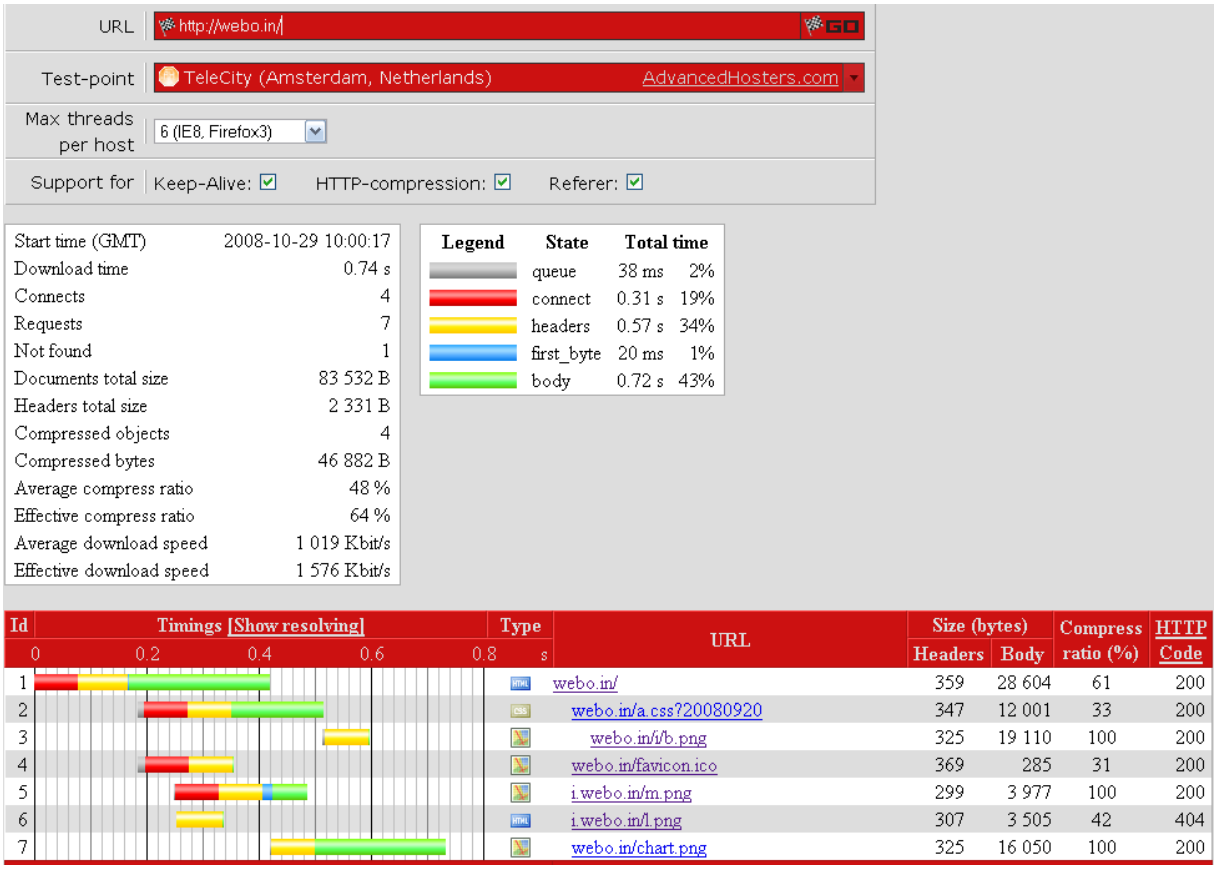


Рис. 51. Результаты анализа загрузки сайта в site-perf.com

GetRPO.com

На данный момент разрабатываются уже и автоматизированные решения для уменьшения времени загрузки страницы. Одним из таких сервисов является Runtime Page Optimizator, который позволяет (в автоматическом режиме для IIS) уменьшить время загрузки страницы. Также есть возможность запросить проверку требуемого сайта и получить вполне детальный анализ, что и как можно сделать для ускорения его загрузки. Поскольку для анализа используется встроенный браузер (MSIE), то результаты проверки наиболее достоверны:

Рис. 52. Результаты анализа загрузки сайта в get-rpo.com

Webo.in

Web Optimizator — единственный на данный момент русскоязычный ресурс, посвященный клиентской оптимизации. При анализе сайта выдается как общая характеристика, так и конкретные советы по возможному ускорению его загрузки:

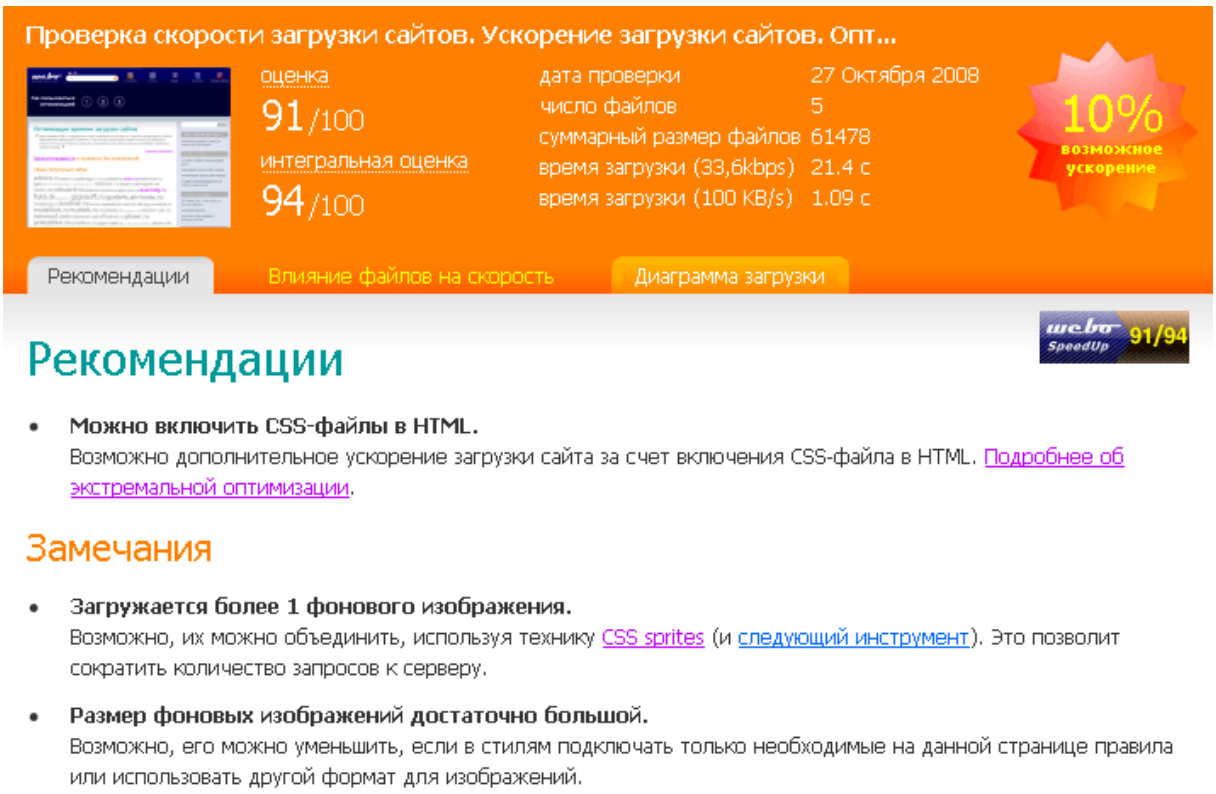


Рис. 53. Результаты анализа загрузки сайта в webo.in

Можно также посмотреть модельную диаграмму загрузки сайта в различных браузерах, учитывая ширину канала и кэширование:

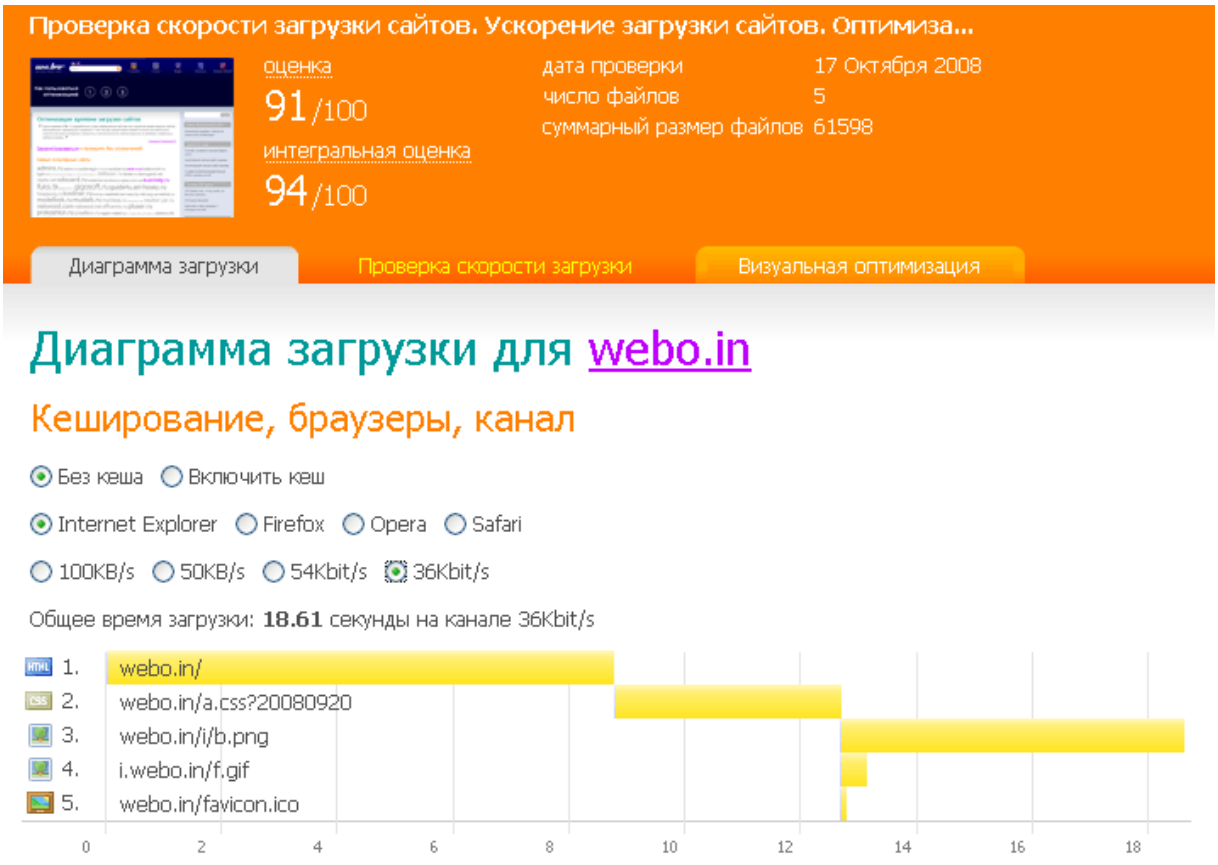


Рис. 54. Результаты анализа загрузки сайта в webo.in (диаграмма загрузки)

Сервис предоставляет возможность отслеживать историю проверок каждого отдельного сайта, а также производить «визуальную оптимизацию», которая проводит все предлагаемые действия в модельном режиме и позволяет понять, что нужно делать в первую очередь и какая будет отдача.

Дополнительно к анализу производительности можно автоматически уменьшить и объединить все тестовые файлы и оптимизировать в размере графику с помощью «Пакетной оптимизации». Этот сервис работает как с уже проверенными сайтами, последовательно загружая и оптимизируя все файлы, так и с архивами.

К недостаткам Web Optimizator стоит отнести отсутствие распознавания динамической загрузки ресурсных файлов (например, через DOM-методы).

Профилирование JavaScript

JSLint (http://www.jslint.com/) позволяет проанализировать код и убедиться в том, что он корректно отформатирован. Очень часто структурные ошибки или неверное форматирование кода пагубно сказываются на дальнейшем его уменьшении, ибо все минимизаторы обладают собственным синтаксическим аппаратом, они не используют браузеры для проверки корректности JavaScript. Поэтому, не проверив код на соответствие стандартам, есть все шансы получить его после уменьшение или обфускации неработающим.

JsUnit (http://www.jsunit.net/) предоставляет мощный фреймворк для тестирования и отладки ваших JavaScript-приложений. С помощью AjaxView (http://research.microsoft.com/projects/ajaxview/) можно проксировать и анализировать AJAX-запросы. Для профессионального профилирования веб-приложений стоит воспользоваться JsLex (http://rockstarapps.com/pmwiki/pmwiki.php?n=JsLex.JsLex), а время выполнения различных операций на странице можно замерить с помощью YUI Profiler (http://developer.yahoo.com/yui/profiler/). Но не стоит забывать, что лучшей проверкой веб-приложений на прочность всегда был и остается пользователь.

8.2. Несколько советов для браузеров

Ускоряем загрузку страниц в Firefox 3

В Firefox можно увеличить скорость загрузки и отображения страниц, значительно повысив комфортность работы в интернете. Что для этого нужно сделать:

1. Открыть страничку настроек, набрав в адресной строке `about:config`.
2. Отредактировать следующие опции:

```
network.http.pipelining — true
network.http.proxy.pipelining — true
network.http.pipelining.maxrequests — 8
nglayout.initialpaint.delay — 0
```

(в последнем случае необходимо щелкнуть правой кнопкой и создать новое **целое** значение с таким именем).

Все, теперь можно наслаждаться возросшей скоростью загрузки и отображения (особенно заметно на широких каналах).

Как это работает?

В протоколе HTTP версии 1.1, внедренном в 1999 году и используемом по сей день во всемирной паутине, появилась возможность в рамках одного TCP-соединения делать несколько запросов веб-серверу.

Учитывая, что страницы современных сайтов обычно содержат большое количество изображений и других объектов, время установления TCP-соединения (для каждого объекта) начинает играть значительную роль. Поэтому грех не использовать возможность гарантированного ускорения загрузки страниц, существующую уже 9 лет.

1. `network.http.version – 1.1`, `network.http.keep-alive – true`

   Эти опции установлены по умолчанию и разрешают Firefox использовать HTTP версии 1.1, включая возможность делать несколько запросов в соединении.

2. `network.http.pipelining – true`, `network.http.proxy.pipelining – true`

   Эти опции предписывают Firefox делать несколько запросов в соединении, не дожидаясь ответа сервера. Фактически, мы начинаем использовать HTTP-конвейер на полную мощность. Но нужно понимать, что не все веб-серверы на текущий момент поддерживают эту технологию и в некоторых случаях возможны проблемы с загрузкой ресурсов.

3. `network.http.pipelining.maxrequests – 8`

   Эта опция задает максимальное количество запросов, которое может быть сделано в соединении: от 1 до 8. Указывать значение более 8 бессмысленно, так как это физически не поддерживается Firefox и эффекта иметь не будет.

4. `nglayout.initialpaint.delay – 0`

   Эта опция уменьшает до нуля задержку перед отображением информации, полученной от веб-сервера, что позволит увидеть запрошенную страницу чуть быстрее.

Ускоряем загрузку страниц в Opera 9

В Opera также имеется ряд настроек, которые пользователь может легко изменить. Для этого нужно зайти на страницу `opera:config`, на которой все необходимые параметры можно легко найти с помощью поиска. Итак, рекомендуется выставить следующие значения:

1. `Max Connections Server = 16`

   Устанавливает число параллельных соединений с одним хостом. Подробнее о влиянии числа используемых соединений на скорость загрузки рассказывается в пятой главе.

2. `Max Connections Total = 32 или 64`

   Устанавливает максимальное число открытых соединений (с различными хостами для одной страницы).

3. `Check local Hostname отключить`

   При вводе адреса Opera пытается его определить, в том числе основываясь на пространстве имен в локальной сети. Если этот браузер не используется для локальных ресурсов, то опцию стоит отключить, что немного ускорит распознавание IP-адреса сайта.

4. `Always load favicon = 0`

   `favicon.ico` (как было описано во второй главе) оказывает существенное влияние на скорость загрузки. Если используется медленное подключение к интернету, то загрузку иконки сайта можно отключить. Это позволит быстрее загружать действительно нужные файлы (например, таблицу стилей).

5. `Documents Expiry = 3600`

   По умолчанию Opera кэширует HTML-файлы на 5 минут (300 секунд). При увеличении этого времени до 1 часа браузер не будет лишний раз запрашивать сервер, а просто возьмет документ из кэша.

6. `Images Expiry = 86400`

   Эта опция распространяется на кэширование картинок, которое, по умолчанию, длится 5 часов. Рекомендуется выставить срок кэширования для изображений на 1 сутки.

7. `Other Expiry = 86400`

   Здесь речь идет о других статических файлах. Для них также можно выставить кэширование на сутки.

8. `Delayed Script Execution включить`

   Отложенное выполнение скриптов позволяет Opera начать отображение страницы сразу по получению HTML-файла (и всех CSS-файлов, если они загрузятся достаточно быстро). При загрузке JavaScript Opera просто перерисует страницу в браузере. Эта функциональность способна существенно ускорить загрузку страницы, однако, при значительных изменениях при загрузке скриптов может ухудшать восприятие сайтов: не всегда приятно, если страница меняется раз в секунду.

После изменения всех настроек нужно их сохранить и перезапустить Opera.

Также можно попробовать уменьшить задержку при перерисовке страницы. Для этого нужно выбрать (англоязычная версия):

```
Tools -> Preferences -> Advanced -> Browsing -> Loading -> Redraw instantly
```

В русскоязычной:

```
Инструменты -> Настройки -> Дополнительно -> Обозреватель ->
Перерисовывать страницу -> Непрерывно
```

Interner Explorer

IE не обладает какой-либо конфигурационной страницей или такой же гибкостью, как два описанных выше браузера. Однако и в нем можно немного ускорить загрузку страниц. Для этого нужно завести в реестре значения, соответствующие числу максимальных соединений для одного хоста.

Мы должны зайти в реестр (например, через `Пуск -> Выполнить -> regedit`) и пройти в следующую ветку:

```
HKEY_CURRENT_USER -> Software -> Microsoft -> Windows -> Current version ->
Internet settings
```

Там нужно создать 2 новых параметра (DWORD) с названиями MaxConnectionsPer1_0Server и MaxConnectionsPer1_0Server. У обоих изменить значение (`Правый клик -> Изменить`) на **10**. Здесь стоит заменить, что, по умолчанию, значение выводится в шестнадцатеричной системе счисления, что соответствует 16 в десятичной. Таким образом, мы выставим число максимальных соединений к одному хосту равным 16.

После этого остается только перезапустить IE и наслаждаться быстрым интернетом.

8.3. Оптимизированные конфигурации

Конфигурация Apache 1.3

```
<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 заголовок Content-Encoding: gzip
        AddEncoding       gzip .gz

# выставляем минимальный размер для сжимаемого файла
        mod_gzip_minimum_file_size  1000

# и максимальный размер файла
        mod_gzip_maximum_file_size  500000

# выставляем максимальный размер файла, сжимаемого прямо в памяти
        mod_gzip_maximum_inmem_size  60000

# устанавливаем версию протокола, с которой будут отдаваться gzip-файлы
```

```
# на клиент
        mod_gzip_min_http       1000

# исключаем известные проблемные случаи
        mod_gzip_item_exclude    reqheader "User-agent: Mozilla/4.0[678]"
        mod_gzip_item_exclude    reqheader "User-agent: Konqueror"

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

# включаем .css / .js файлы, подробнее о них ниже
        mod_gzip_item_include     file    \.js$
        mod_gzip_item_include     file    \.css$

# дополнительно сжимаем другие текстовые файлы
        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 .*\.(js|css|html|txt)$>
                Header set Cache-Control: private
        </FilesMatch>
</IfModule>

<IfModule mod_expires.c>
# включаем кэширование для всех файлов сроком на 10 лет
        ExpiresActive On
        ExpiresDefault "access plus 10 years"

# отключаем его для HTML-файлов
        <FilesMatch .*\.(shtml|html|phtml|php)$>
                ExpiresActive Off
        </FilesMatch>
</IfModule>
```

Конфигурация Apache 2

```
# выставляем заголовок Content-Encoding: gzip
AddEncoding       gzip .gz
# 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 такой мощный,
# то уровень сжатия можно выставить в 1, размер файлов при этом
# увеличивается примерно на 20%.
DeflateCompressionLevel 9
DeflateWindowSize 15

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

# указываем прокси-серверам передавать заголовок User-Agent для
# корректного распознавания сжатия
Header append Vary User-Agent

# 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>

# включаем кэширование для всех файлов сроком на 10 лет
ExpiresActive On
ExpiresDefault "access plus 10 years"

# отключаем его для HTML-файлов
<FilesMatch .*\.(shtml|html|phtml|php)$>
        ExpiresActive Off
</FilesMatch>
```

Конфигурация nginx 0.7+

Пример конфигурационного файла для одного виртуального сервера:

```
server {
#слушаем порт 80
  listen 80;

#перечисляем через пробел имена этого сервера
  server_name core.freewheel.ru:

#путь к корню сервера
  root /my/path/to/core.freewheel.ru;

#пути к логам
  access_log /my/path/to/core-access.log combined;
  error_log /my/path/to/core-access.log info;

#подключаем шаблон настроек сервера, в нем самое интересное
  include _servers_template;

#разрешаем себе посмотреть статус сервера
  location = /nginx_status {
        stub_status on;
        access_log  off;
        allow 127.0.0.1;
        deny all;
```

```
  }

#включаем сжатие для тех браузеров, которые его понимают
  gzip on;

#определяем минимальную версию протокола HTTP, для которой отдаем архивы
  gzip_http_version 1.0;

#устанавливаем максимальный уровень сжатия
  gzip_comp_level 9;

#разрешаем проксировать сжатые файлы
  gzip_proxied any;

#и определяем типы файлов (все, которые хорошо сжимаются)
  gzip_types    text/plain text/css application/x-javascript text/xml
application/xml application/xml+rss text/javascript image/x-icon;
}
```

А теперь собственно основной файл настроек (`_servers_template`), использующийся для всех виртуальных серверов в неизменном виде:

```
index index.php index.html;

location / {

# стили, скрипты и XML-файлы
  location ~* ^.+\.(css|js|xml)$ {

# вот для этого и делались заранее архивированные .gz версии
# css и js файлов. Nginx не будет тратить время и сжимать их каждый раз
# заново, а просто отдаст уже готовые архивы, если браузер клиента может
# их принять
        gzip_static on;
        expires   1y;
  }

# несуществующие файлы html и папки отправляем на бэкенд
  if (!-e $request_filename ) {
        rewrite ^/(.*)$ /index.php ;
  }

# проксируем все запросы к PHP-файлам на FCGI бэкенд
  location ~* \.php$ {
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_index index.php;
        include    _fastcgi_params;
  }

# картинки
  location ~* ^.+\.(bmp|gif|jpg|jpeg|ico|png|swf|tiff)$ {
        expires   1y;
  }

# файлы
  location ~* ^.+\.(bz2|dmg|gz|gzip|rar|tar|zip)$ {
        expires 1y;
  }

# другие статические файлы
  location ~* ^.+\.(pdf|txt)$ {
        expires 1y;
```

```
    }
}
```

Обычно nginx собирается без модуля статического сжатия, поэтому при его сборке надо указать опцию `--with-http_gzip_static_module` — без этого `gzip_static` не заработает, и серверу придется сжимать файлы каждый раз заново. Также надо иметь в виду, что указанная конфигурация приведена для версии 0.7+.

Настройка IIS

В IIS включить сжатие достаточно просто: необходимо в диспетчере служб IIS зайти в свойства элемента «Веб узлы» и перейти во вкладку «Служба».

Плюсы такого включения сжатия:

- простота;
- поддержка сжатия IIS статических файлов;
- поддержка кэширования сжатых файлов;
- не требует написания кода.

Минусы включения поддержки сжатия на сервере IIS:

- сервер решает: вы не будете знать, что, когда и как сжимается;
- глобальное включение: сжатие включается для всей службы разом и будет влиять на все узлы или виртуальные каталоги вашего сервера (по крайне мере, через gui отключить сжатие у конкретного узла нельзя);
- возможны различные проблемы на уровне конфликтов локальных серверных скриптов или проблемы совместимости с некоторыми браузерами. Был зафиксирован случай, когда один из пользователей пожаловался на то, что у него стала отображаться пустая главная страница как раз после включения сжатия через IIS; после выключения все вернулось в норму.

8.4. Разбор полетов

В этом разделе некоторые наиболее посещаемые проекты Рунета будут рассмотрены с точки зрения клиентской оптимизации. Основной целью каждого проводимого ниже анализа является показать на рабочих примерах, каким именно образом можно ускорить загрузку сайта. И показать это на достаточно нагруженных и популярных ресурсах, для которых каждая мелочь имеет значение.

vkontakte.ru

В Контакте (http://vkontakte.ru/) является на данный момент (по числу просмотров страниц) наиболее посещаемым сайтом. Диаграмма загрузки внутренней страницы профиля выглядит примерно следующим образом:

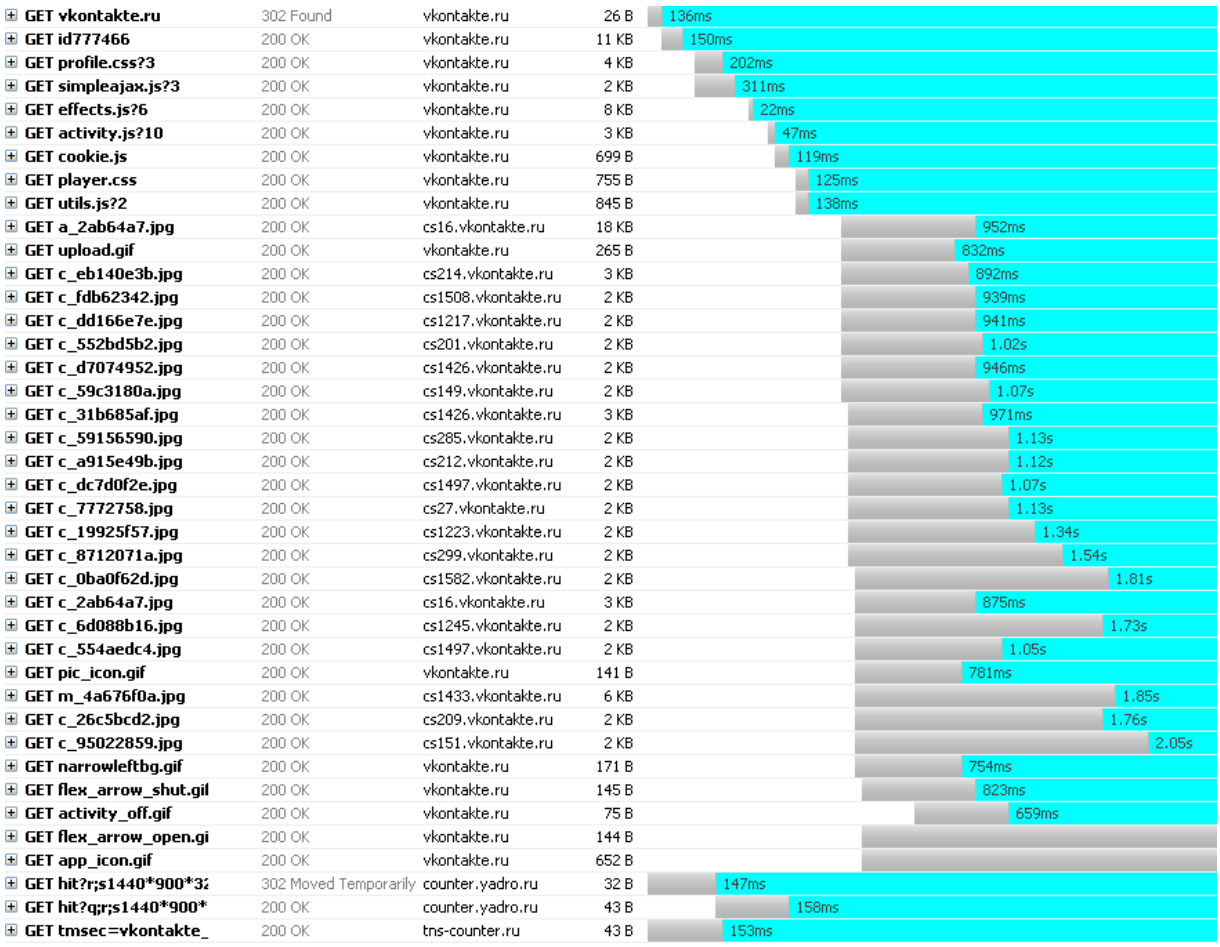


Рис. 55. Результаты анализа загрузки внутренней страницы сайта vkontakte.ru

В самом начале идет редирект с главной страницы на личную страницу участника, — если заходить сразу через страницу профиля, его, естественно, не будет. Однако это повышает удобство для пользователя, которому нужно помнить всего один адрес. Еще один редирект замечен у скрипта счетчика, но он загружается параллельно основному документу и конкретно в данном случае на время загрузки влияет слабо.

Двигаемся дальше: `gzip`. Для всех текстовых файлов (HTML, CSS, JavaScript) он включен. Это не может не радовать. Однако никакой минимизации для них не производилось. С точки зрения производительности это минорная оплошность: при самой лучшей минимизации выиграть больше 5% от уже имеющегося сжатия крайне сложно. Для пользователей же это отразится в 10–50 мс загрузки.

Объединение файлов. После применения сжатия CSS- и JavaScript-файлы стали занимать 2–5 Кб в размере, поэтому загружать их по отдельности особого смысла не имеет. Объединение наиболее используемых стилевых правил в основной файл (пусть даже размером в 15–20 Кб) позволило бы сократить время загрузки на 100 мс (в данном случае). Стоит также отметить, что для пользователей IE загружается дополнительный файл стилей (крайне маленький в размере), который, естественно, можно было бы включить в основной.

Пост-загрузка для JavaScript-компонентов. На странице уже используются данные методы: в частности, рекламные баннеры загружаются как раз через динамическое создание изображений в заранее определенных блоках шаблона страницы. Однако

вынесение всей клиентской логики в загрузку по комбинированному событию `window.onload` позволило бы отрисовать страницу на экране на 300 мс быстрее (в данном случае это 40% от стадии предзагрузки). Хотя, возможно, это потребовало бы достаточно существенной переработки текущего функционирования отдельных частей портала.

Кэширование. На данный момент с этим все замечательно: выставляется как `max-age` для статических файлов, так и `Last-Modified`. Предполагается, что большинство пользователей заходят на vkontakte.ru постоянно, поэтому большая часть файлов берется браузером сразу из кэша. Именно поэтому наличие стилей и скриптов внутри HTML-файла сведено к минимуму: это позволяет уменьшить объем последнего при наличии в кэше всех необходимых файлов. Также стоит отметить, что в качестве внешнего сервера используется nginx.

Заголовки ответа

Имя	Значение
RESPONSE	HTTP/1.1 200 OK
Server	nginx/0.6.31
Date	Fri, 31 Oct 2008 10:53:04 GMT
Content-Type	text/css
Last-Modified	Fri, 29 Aug 2008 12:39:25 GMT
Expires	Tue, 04 Nov 2008 10:53:04 GMT
Cache-Control	max-age=345600
Content-Encoding	gzip

Рис. 56. Заголовки ответа для статического файла с vkontakte.ru

На странице присутствует некоторое количество (5–10) небольших фоновых картинок, которые могли бы быть успешно объединены в CSS Sprites или даже добавлены в соответствующий CSS-файл. Однако данное действие не сильно спасло бы ситуацию: основная нагрузка приходится на пользовательские картинки. А почти все они расположены на различных хостах (на диаграмме загрузки присутствует несколько десятков хостов). Это хорошо для уменьшения времени ожидания ответа, но плохо с точки зрения DNS Lookup. В качестве дополнительного минуса можно назвать то, что средний размер картинки — 2 Кб.

Поскольку мы имеем дело не с любительским сайтом, а с высоконагруженным порталом, то тривиального решения для данной проблемы не существует (точнее, оно уже реализовано: прозрачное распределение картинок по кластерам, в силу чего каждая небольшая картинка прикреплена к единственному хосту). Также при обновлении страницы часть картинок (относящиеся к друзьям пользователя и друзьям онлайн) генерируется случайным образом.

В качестве возможного решения данной проблемы стоит рассматривать создание Image Map (или CSS Sprites) с наиболее часто используемыми изображениями для данной страницы. Для страницы пользователя — это список написавших на стене: он меняется относительно редко, а объединение иконок пользователей в группы по 5–7 (размер итогового файла 10–15 Кб) позволит несколько сократить время загрузки страницы. В общем случае (рассматривая параллельные загрузки) это будет DNS Lookup + время соединения. Хотя vkontakte.ru уже и так использует своего рода CDN (время соединения сведено к минимуму), но выигрыш все равно составит порядка 200–300 мс даже для широкополосного доступа.

В качестве альтернативного решения можно рассмотреть кластеризацию иконок пользователей по друзьям — т.е. создание локального кластера, содержащего все необходимые изображения для отдельного пользователя, например, все иконки его друзей. Тогда при просмотре страницы этого пользователя будет использоваться не такое большое количество хостов, что существенно уменьшит время на DNS Lookup.

odnoklassniki.ru

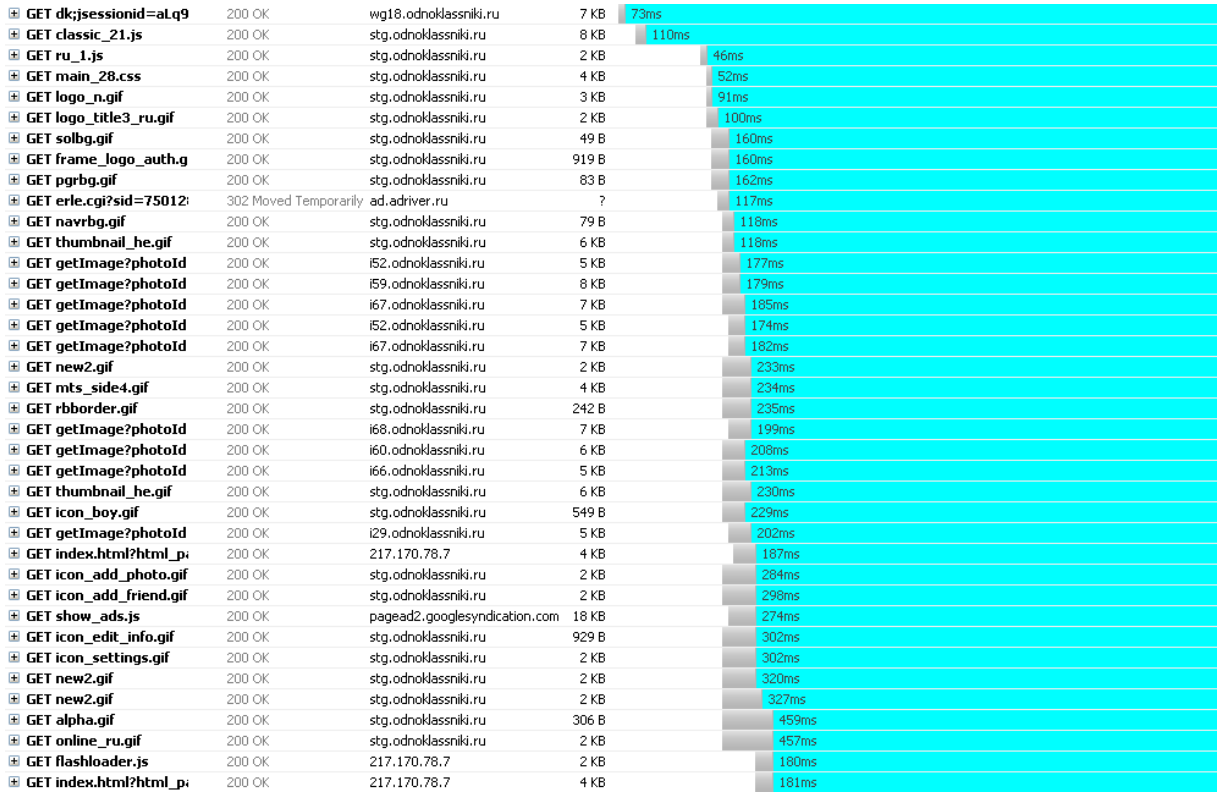


Рис. 57. Результаты анализа загрузки внутренней страницы сайта odnoklassniki.ru

Давайте рассмотрим прямого конкурента vkontakte.ru — odnoklassniki.ru. Для этого сайта ситуация, очевидно, еще хуже. Во-первых, значительная часть декоративных картинок не внесена в число фоновых изображений, что заставляет IE старых версий опрашивать их в обычном порядке. Во-вторых, рекламные баннеры не вынесены в пост-загрузку и сильно мешаются в общем процессе отображения страницы (часть страницы с баннером «зависает» в ожидании ответа рекламного сервера). Обилие счетчиков статистики только усугубляет ситуацию.

Как с этим можно бороться? Естественно, что все файлы скриптов можно и нужно объединить в один, и его вынести в пост-загрузку. Фоновые картинки достаточно объединить в 2–3 файла, что позволит загружать иконки других пользователей (а именно они составляют наиболее значительную часть) быстрее.

Заголовки ответа

Имя	Значение
RESPONSE	HTTP/1.1 200 OK
Server	Resin/2.1.16
Last-Modified	Wed, 29 Oct 2008 14:48:48 GMT
Cache-Control	max-age=100000000
Expires	Tue, 28 Jul 2009 15:13:10 GMT
Vary	Accept-Encoding
Etag	"AAAAR1JFASA"
Content-Encoding	gzip
Content-Type	text/css
Connection	close
Transfer-Encoding	chunked
Date	Fri, 31 Oct 2008 15:13:09 GMT

Рис. 58. Результаты ответа для статического файла с odnoklassniki.ru

В остальном почти все меры уже приняты: текстовые файлы отдаются сжатыми, для статики устанавливается срок кэширования в далекое будущее. Однако, как видно из рис. 58, с кэшированием произошел небольшой перебор: наличие и `ETag`, и `Last-Modified` заголовка является избыточным. Для корректной проверки файла на существование новой версии достаточно только одного из них.

Естественно, что для odnoklassniki.ru проблема распределения различных изображений по нескольким хостам так же актуальна, как и для vkontakte.ru. И на данный момент она решается точно так же. Поскольку статические блоки с большим количеством изображений на странице практически отсутствуют, то более корректное решение указанной проблемы может и не существовать.

yandex.ru

Давайте вслед за самыми популярными социальными сетями рассмотрим наиболее посещаемые поисковые и почтовые порталы Рунета. Начнем с Яндекса.

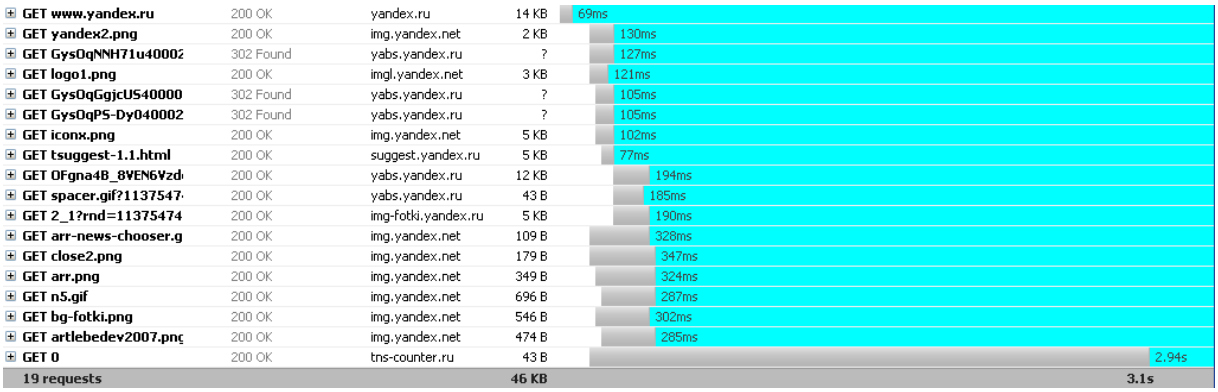
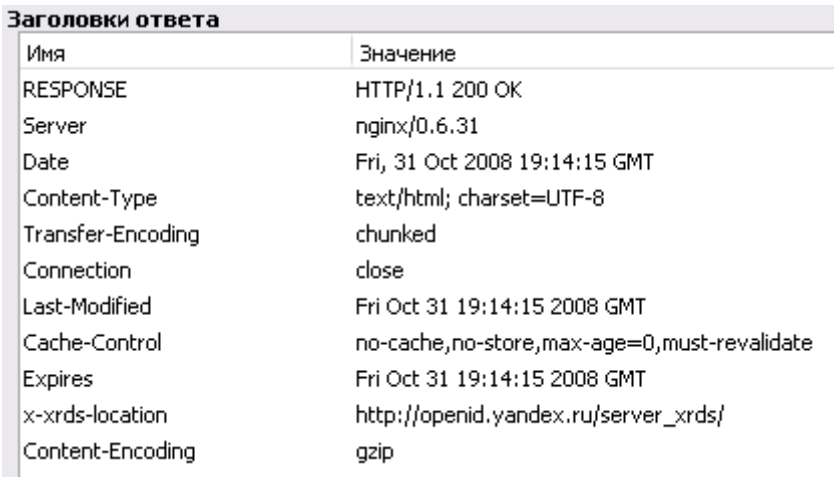


Рис. 59. Результаты анализа загрузки главной страницы www.yandex.ru

Как можно видеть, эта страница уже сильно оптимизирована. При всем объеме информации и внутренней логики используется всего 19 запросов к серверу, общий объем передаваемых данных 49 Кб. В качестве характерного шага оптимизации часто посещаемых главных страниц таких порталов можно назвать то, что CSS-файл внесен

внутрь HTML и вся JavaScript-логика располагается там же (не учитывая, конечно, какие-то непостоянные явления, вроде блока авторизации или сезонной рекламы).

Естественно, что для текстовых файлов применяется сжатие, причем даже сам HTML-код минимизирован почти по максимуму: убраны переводы строк и лишние пробелы. В качестве спорного момента можно отметить отсутствие кэширования для главной страницы. Раньше его включали на 5 минут, чтобы уменьшить число повторных запросов. На данный момент (наверное, в связи с блоком почтовой авторизации) всякое кэширование отключено. Однако, корректная настройка `Last-Modified` (в зависимости от переменных окружения, связанных с конкретным пользователем) могла бы уменьшить число передаваемых данных (сервер мог бы ответить статус-кодом 304 и не передавать всех данных).



Заголовки ответа

Имя	Значение
RESPONSE	HTTP/1.1 200 OK
Server	nginx/0.6.31
Date	Fri, 31 Oct 2008 19:14:15 GMT
Content-Type	text/html; charset=UTF-8
Transfer-Encoding	chunked
Connection	close
Last-Modified	Fri Oct 31 19:14:15 2008 GMT
Cache-Control	no-cache,no-store,max-age=0,must-revalidate
Expires	Fri Oct 31 19:14:15 2008 GMT
x-xrds-location	http://openid.yandex.ru/server_xrds/
Content-Encoding	gzip

Рис. 60. Результаты ответа для HTML-файла с yandex.ru

Во всем остальном соблюдены почти все рекомендации: наиболее часто используемые картинки объединены в CSS Sprites, при загрузке статических файлов задействуется несколько хостов (в том числе те, на которые не передаются cookie). Однако, как видно из диаграммы загрузки, на странице еще есть некоторое количество небольших изображений, которые также можно объединить в одно или же даже внести в сам документ (для всех браузеров, кроме IE) в виде `data:URI`.

rambler.ru

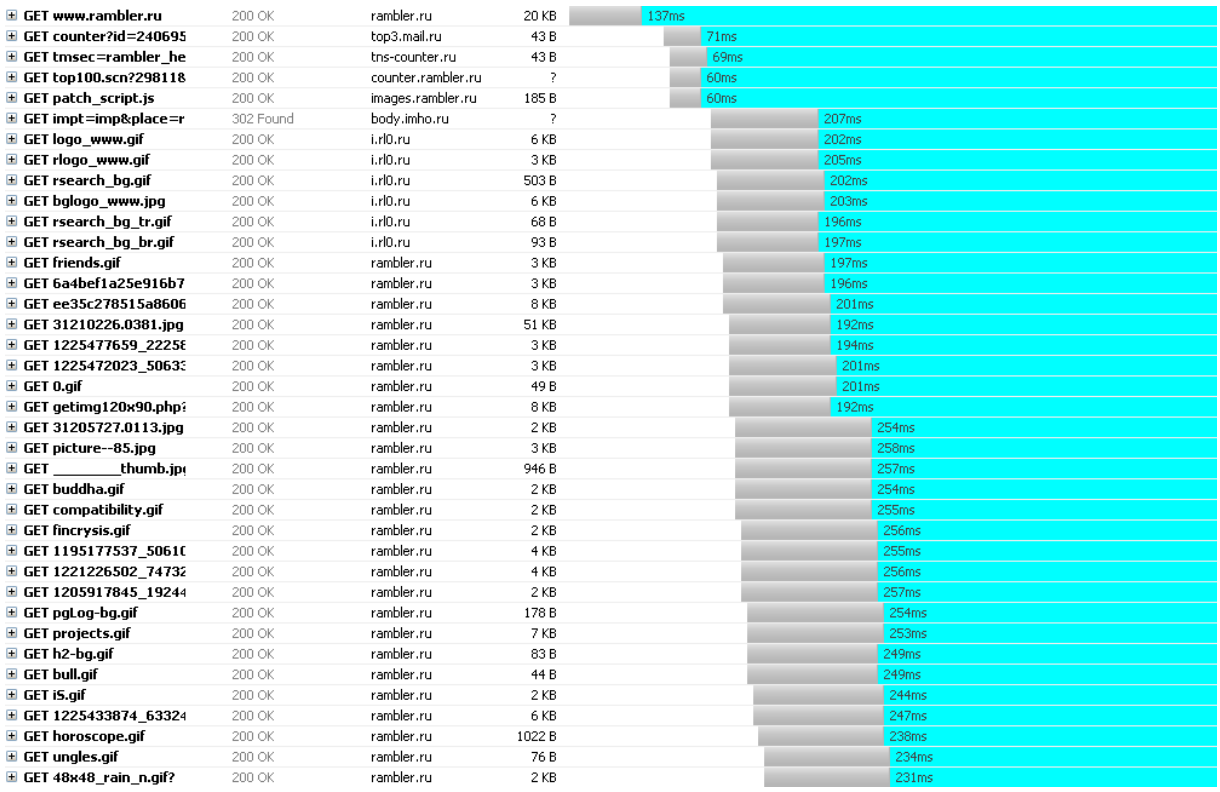


Рис. 61. Результаты анализа загрузки главной страницы www.rambler.ru

В случае в Рамблером ситуация несколько другая: на главной странице портала представлено достаточно большое количество информации, в связи с чем число внешних объектов значительно больше — уже 46.

Естественно, для текстовых файлов уже включено сжатие. Но минимизации конечный HTML-файл не подвергнут: видимо, разработчики Рамблера заботятся о трафике своих пользователей не так тщательно.

CSS Sprites довольно активно используются на странице, но и тут не обошлось без очевидных промахов. Например, иконки для состояний погоды можно было с легкостью объединить в один файл, однако, это не было сделано. Использование `.png` формата вместо `.gif` для фоновых изображений также способно уменьшить размер конечного файла. Применение же скрипта `patch_script.js` размером в 185 байтов (по сравнению с HTML в 20 Кб) крайне неосмотрительно.

Для загрузки большинства изображений (как хорошо видно из диаграммы загрузки) используется всего 2 хоста (`rambler.ru` и `i.rl0.ru`). Увеличение их числа до 4 позволило бы существенно ускорить процесс загрузки многочисленных картинок. Поскольку картинки занимают около 60% от общего времени, то добавление двух хостов для них позволило бы ускорить загрузку страницы на треть. Дополнительно: большинство jpeg изображений сгенерированы не оптимальным образом, и их размер можно уменьшить на 20–30%.

Резюмируя все вышесказанное: для главной страницы Рамблера есть еще куда стремиться в плане обеспечения удобства для своих пользователей.

mail.ru

Mail.ru является, пожалуй, наиболее крупным почтовым порталом в Рунете. Все остальные сервисы на нем появились именно благодаря почте и базе держателей почтовых адресов.

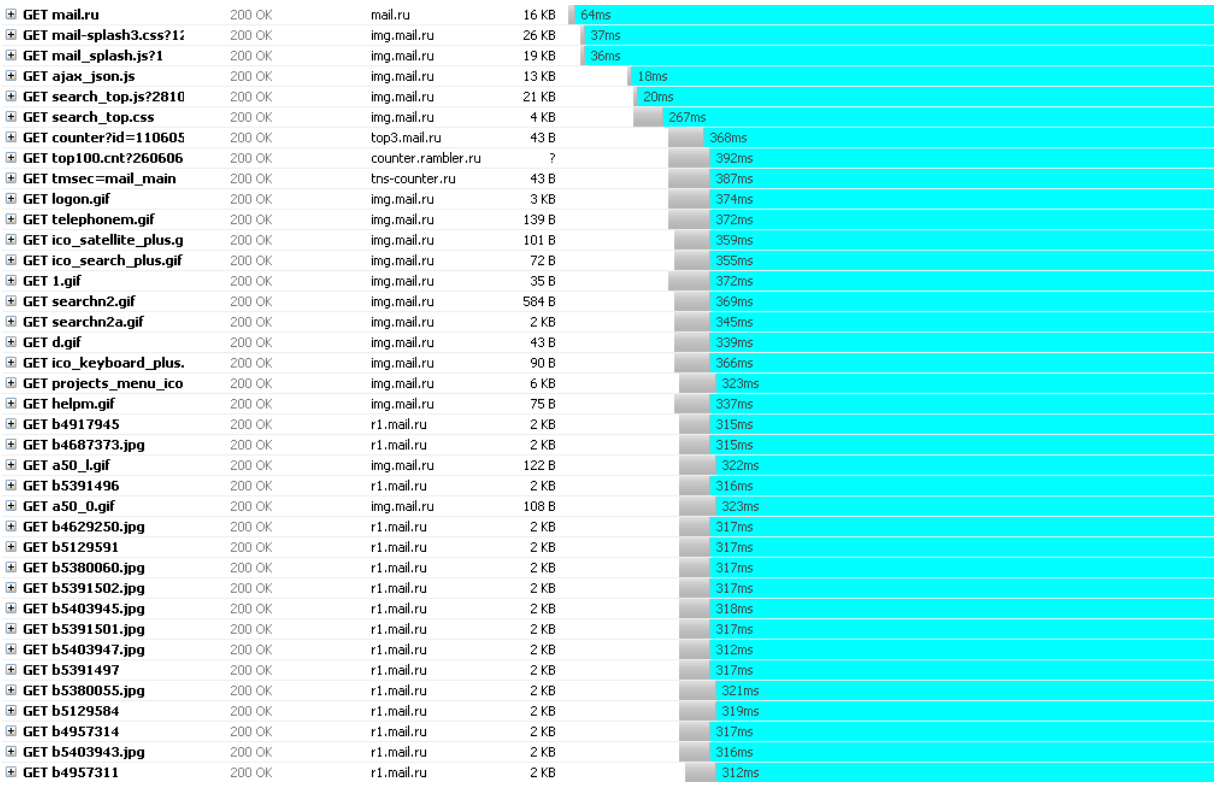


Рис. 62. Результаты анализа загрузки главной страницы www.mail.ru

Сам HTML-документ отдается через gzip, но для CSS- и JavaScript-файлов сжатие не применяется (что, естественно, плохо отражается на времени загрузки). На странице запрашивается 4 JavaScript-файла и 3 CSS. При этом JavaScript подключается, в основном, в `head` страницы. Это все крайне негативно влияет на скорость отображения страницы. Однако, как видно из рис. 63, кэширование для файлов стилей и скриптов включено при помощи `Expires` и `Last-Modified`. Это не может не радовать.

Заголовки ответа

Имя	Значение
RESPONSE	HTTP/1.1 200 OK
Date	Fri, 31 Oct 2008 19:26:17 GMT
Content-Type	text/css
Content-Length	26230
Accept-Ranges	bytes
Last-Modified	Tue, 21 Oct 2008 15:44:22 GMT
Expires	Fri, 07 Nov 2008 19:24:08 GMT

Рис. 63. Результаты ответа для статического файла с mail.ru

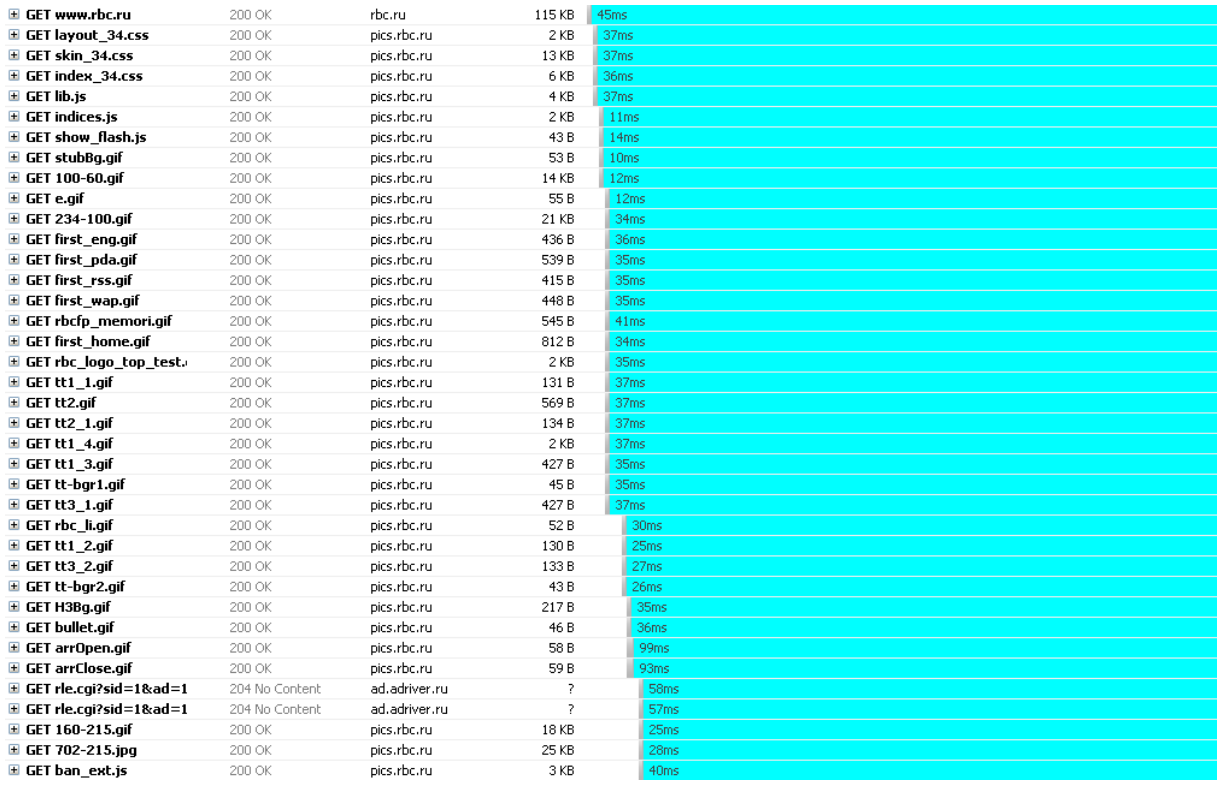
Фоновые картинки уже объединены в спрайты (хотя, есть еще возможности для объединения), однако, дополнительное уменьшение размера возможно за счет использование `.png`. Также большинство `.jpg` картинок можно существенно уменьшить в

размере. Можно рассмотреть кэширование и объединение нескольких картинок (из блоков «Фото» или «Видео») в Image Map, чтобы уменьшить число запросов к серверу.

Для статических файлов используется всего 2 хоста (`r1.mail.ru` и `img.mail.ru`). Добавление еще двух (вместе с уже описанными методами уменьшения размера графики) способно ускорить загрузку примерно в два раза. В целом же, у разработчиков mail.ru есть еще большой простор для творчества в области клиентской оптимизации.

rbc.ru

Давайте от рассмотрения крупных развлекательных порталов переместимся к конкретно новостным. Первым в этой среде можно рассмотреть сайт РИА «РосБизнесКонсалтинг».



GET www.rbc.ru	200 OK	rbc.ru	115 KB	45ms
GET layout_34.css	200 OK	pics.rbc.ru	2 KB	37ms
GET skin_34.css	200 OK	pics.rbc.ru	13 KB	37ms
GET index_34.css	200 OK	pics.rbc.ru	6 KB	36ms
GET lib.js	200 OK	pics.rbc.ru	4 KB	37ms
GET indices.js	200 OK	pics.rbc.ru	2 KB	11ms
GET show_flash.js	200 OK	pics.rbc.ru	43 B	14ms
GET stubBg.gif	200 OK	pics.rbc.ru	53 B	10ms
GET 100-60.gif	200 OK	pics.rbc.ru	14 KB	12ms
GET e.gif	200 OK	pics.rbc.ru	55 B	12ms
GET 234-100.gif	200 OK	pics.rbc.ru	21 KB	34ms
GET first_eng.gif	200 OK	pics.rbc.ru	436 B	36ms
GET first_pda.gif	200 OK	pics.rbc.ru	539 B	35ms
GET first_rss.gif	200 OK	pics.rbc.ru	415 B	35ms
GET first_wap.gif	200 OK	pics.rbc.ru	448 B	35ms
GET rbcfp_memori.gif	200 OK	pics.rbc.ru	545 B	41ms
GET first_home.gif	200 OK	pics.rbc.ru	812 B	34ms
GET rbc_logo_top_test.…	200 OK	pics.rbc.ru	2 KB	35ms
GET tt1_1.gif	200 OK	pics.rbc.ru	131 B	37ms
GET tt2.gif	200 OK	pics.rbc.ru	569 B	37ms
GET tt2_1.gif	200 OK	pics.rbc.ru	134 B	37ms
GET tt1_4.gif	200 OK	pics.rbc.ru	2 KB	37ms
GET tt1_3.gif	200 OK	pics.rbc.ru	427 B	35ms
GET tt-bgr1.gif	200 OK	pics.rbc.ru	45 B	35ms
GET tt3_1.gif	200 OK	pics.rbc.ru	427 B	37ms
GET rbc_li.gif	200 OK	pics.rbc.ru	52 B	30ms
GET tt1_2.gif	200 OK	pics.rbc.ru	130 B	25ms
GET tt3_2.gif	200 OK	pics.rbc.ru	133 B	27ms
GET tt-bgr2.gif	200 OK	pics.rbc.ru	43 B	26ms
GET H3Bg.gif	200 OK	pics.rbc.ru	217 B	35ms
GET bullet.gif	200 OK	pics.rbc.ru	46 B	36ms
GET arrOpen.gif	200 OK	pics.rbc.ru	58 B	99ms
GET arrClose.gif	200 OK	pics.rbc.ru	59 B	93ms
GET rle.cgi?sid=1&ad=1	204 No Content	ad.adriver.ru	?	58ms
GET rle.cgi?sid=1&ad=1	204 No Content	ad.adriver.ru	?	57ms
GET 160-215.gif	200 OK	pics.rbc.ru	18 KB	25ms
GET 702-215.jpg	200 OK	pics.rbc.ru	25 KB	28ms
GET ban_ext.js	200 OK	pics.rbc.ru	3 KB	40ms

Рис. 64. Результаты анализа загрузки главной страницы www.rbc.ru

Сразу бросается в глаза, что на странице используется 3 CSS-файла и 3 JavaScript-файла, которые можно объединить в один и сэкономить приличное время, не утомляя пользователя белым экраном. Также для текстовых файлов не включено никакое сжатие, что существенно увеличивает время их передачи (115 Кб HTML-кода ведь не мгновенно поступают).

Преобразование `.gif` в `.png` и устранение комментариев из `.jpg` файлов способно сократить объем передаваемых данных еще примерно на 20% от 514 Кб, которые загружаются при открытии данной страницы (фактически, объем самого HTML-файла). CSS Sprites в данном случае могут сократить общее число запросов, как минимум, на треть (хотя на этом сайте иконки для погоды уже «склеены» в один файл).

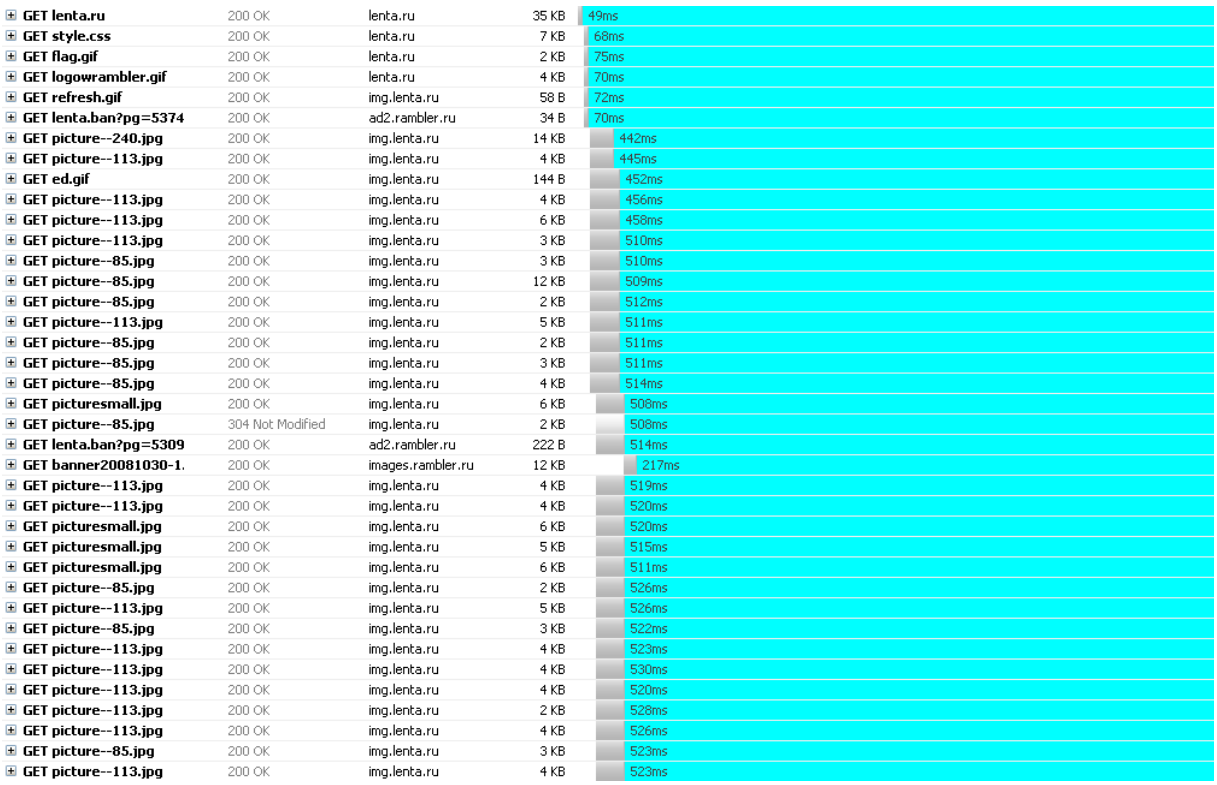
Естественно, добавление еще 3 хостов (к `pics.rbc.ru`) для выдачи картинок только повысит скорость загрузки (еще на 40–50%). С кэшированием, правда, здесь все отлично:

кэш устанавливается на 3 часа, и есть заголовок `Last-Modified`, позволяющий отвечать 304 в том случае, если файл не изменился с момента последнего посещения.

В целом, за исключением явных недосмотров, сайт являет пример грамотного подхода к клиентской оптимизации.

lenta.ru

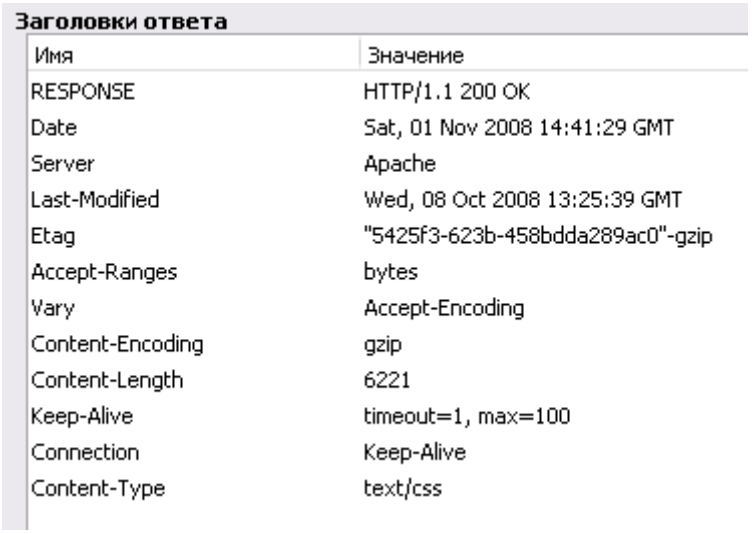
Следующим по курсу будет другое крупное информационное агентство — Lenta.ru.



GET lenta.ru	200 OK	lenta.ru	35 KB	49ms
GET style.css	200 OK	lenta.ru	7 KB	68ms
GET flag.gif	200 OK	lenta.ru	2 KB	75ms
GET logowrambler.gif	200 OK	lenta.ru	4 KB	70ms
GET refresh.gif	200 OK	img.lenta.ru	58 B	72ms
GET lenta.ban?pg=5374	200 OK	ad2.rambler.ru	34 B	70ms
GET picture--240.jpg	200 OK	img.lenta.ru	14 KB	442ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	445ms
GET ed.gif	200 OK	img.lenta.ru	144 B	452ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	456ms
GET picture--113.jpg	200 OK	img.lenta.ru	6 KB	458ms
GET picture--113.jpg	200 OK	img.lenta.ru	3 KB	510ms
GET picture--85.jpg	200 OK	img.lenta.ru	3 KB	510ms
GET picture--85.jpg	200 OK	img.lenta.ru	12 KB	509ms
GET picture--85.jpg	200 OK	img.lenta.ru	2 KB	512ms
GET picture--113.jpg	200 OK	img.lenta.ru	5 KB	511ms
GET picture--85.jpg	200 OK	img.lenta.ru	2 KB	511ms
GET picture--85.jpg	200 OK	img.lenta.ru	3 KB	511ms
GET picture--85.jpg	200 OK	img.lenta.ru	4 KB	514ms
GET picturesmall.jpg	200 OK	img.lenta.ru	6 KB	508ms
GET picture--85.jpg	304 Not Modified	img.lenta.ru	2 KB	508ms
GET lenta.ban?pg=5309	200 OK	ad2.rambler.ru	222 B	514ms
GET banner20081030-1.	200 OK	images.rambler.ru	12 KB	217ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	519ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	520ms
GET picturesmall.jpg	200 OK	img.lenta.ru	6 KB	520ms
GET picturesmall.jpg	200 OK	img.lenta.ru	5 KB	515ms
GET picturesmall.jpg	200 OK	img.lenta.ru	6 KB	511ms
GET picture--85.jpg	200 OK	img.lenta.ru	2 KB	526ms
GET picture--113.jpg	200 OK	img.lenta.ru	5 KB	526ms
GET picture--85.jpg	200 OK	img.lenta.ru	3 KB	522ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	523ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	530ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	520ms
GET picture--113.jpg	200 OK	img.lenta.ru	2 KB	528ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	526ms
GET picture--85.jpg	200 OK	img.lenta.ru	3 KB	523ms
GET picture--113.jpg	200 OK	img.lenta.ru	4 KB	523ms

Рис. 65. Результаты анализа загрузки главной страницы lenta.ru

Здесь заметна забота о пользователях: все текстовые файлы отдаются сжатыми, картинки в новостях минимизированы, используется только один файл стилей и из скриптов — только внешние счетчики. Теперь о грустном:



Заголовки ответа

Имя	Значение
RESPONSE	HTTP/1.1 200 OK
Date	Sat, 01 Nov 2008 14:41:29 GMT
Server	Apache
Last-Modified	Wed, 08 Oct 2008 13:25:39 GMT
Etag	"5425f3-623b-458bdda289ac0"-gzip
Accept-Ranges	bytes
Vary	Accept-Encoding
Content-Encoding	gzip
Content-Length	6221
Keep-Alive	timeout=1, max=100
Connection	Keep-Alive
Content-Type	text/css

Рис. 66. Результаты ответа для статического файла с lenta.ru

Файл стилей отдается без указаний для браузера сохранять его в кэше (`Cache-Control` и(ли) `Expires`), однако, `ETag` дублирует функциональность `Last-Modified`. Это не есть хорошо. Для всех остальных статических файлов (изображений) заголовки настроены абсолютно корректно (хотя время кэша можно было увеличить до недели, например).

Поскольку для всех картинок используется только один хост (`img.lenta.ru`), то введение еще одного-двух хостов способно существенно (до 50%) ускорить загрузку главной страницы (в силу большого количества вызываемых изображений). Из дополнительных мер, которые хорошо было бы применить, но которые не дадут ощутимого прироста, стоит назвать CSS Sprites и добавление рекламы через «ненавязчивый» JavaScript.

Команду разработчиков можно только похвалить за достаточно ответственный подход к своему делу.

kommersant.ru

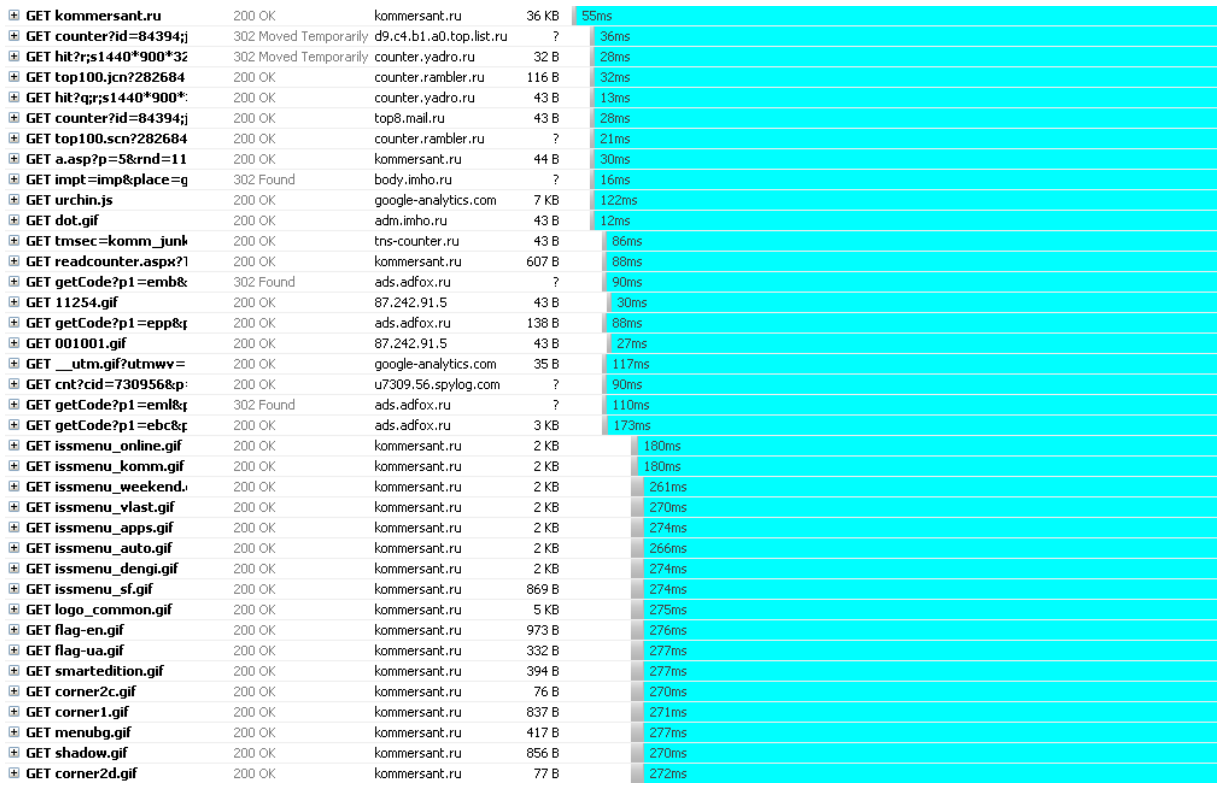


Рис. 67. Результаты анализа загрузки главной страницы kommersant.ru

Для сайта Издательского Дома «КоммерсантЪ» все не так радужно. Как видно из диаграммы загрузки все счетчики загружаются в самую первую очередь, что существенно оттягивает появление содержания страницы на экране. Кроме этого почти все рекламные баннеры загружаются перед основными изображениями. В бочке дегтя есть и ложка меда: включено сжатие для HTML- и CSS-файлов. Но про JavaScript почему-то забыли:

Заголовки ответа

Имя	Значение
RESPONSE	HTTP/1.1 200 OK
Content-Length	8330
Date	Sat, 01 Nov 2008 13:29:46 GMT
Content-Type	application/x-javascript
Etag	"18614e7dee35c91:5353"
Cache-Control	max-age=21600
Last-Modified	Fri, 24 Oct 2008 15:37:57 GMT
Accept-Ranges	bytes
X-Powered-By	ASP.NET

Рис. 68. Результаты ответа для JavaScript-файла с kommersant.ru

Едем дальше. Как видно из рис. 68, кэширующие заголовки в норме, но есть избыточность в виде пары `ETag` / `Last-Modified` (которую можно устранить и сэкономить немного трафика и пользовательского времени). Минимизация для текстовых файлов не применяется, но в случае наличия сжатия это не настолько критично.

Изображения же существенно «раздуты» в размере. Перевод `.gif` в `.png` способен сэкономить до 50 Кб (при общем размере всех файлов более 800 Кб). CSS Sprites и

добавление еще 3 хостов для изображений в статьях могут повлиять гораздо сильнее: ведь именно на картинки падает основная тяжесть. Здесь возможен выигрыш до 60% от текущего времени загрузки. Если добавить к этому отображение баннеров на стадии пост-загрузки, то есть возможность просто феноменально ускорить отображение сайта.

Подводя итоги: для данного сайта выполнено далеко не все, что можно было бы сделать. И эффективность работы сайта может быть значительно улучшена за счет довольно простых действий.

marketgid.ru

Давайте после развлекательных и новостных порталов перейдем ближе к «реальному» бизнесу и рассмотрим загрузку товарного каталога на примере MarketGid.

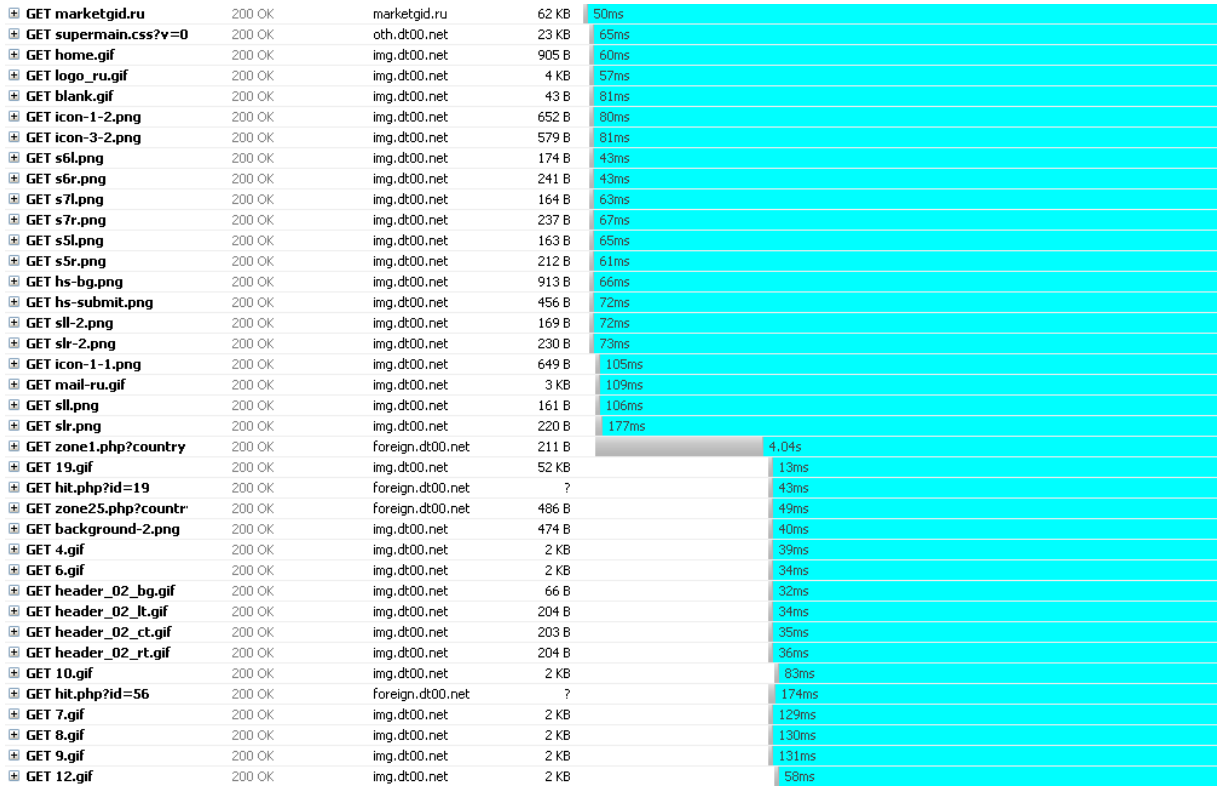


Рис. 69. Результаты анализа загрузки главной страницы marketgid.ru

На первый взгляд, тут все совсем запущено: более 300 HTTP-запросов передают в браузер пользователя около 700 Кб данных. На второй взгляд, тоже все не очень хорошо: сжатие включено только для HTML-файлов, стили и скрипты отдаются как есть. Всего 1 CSS-файл для «нормальных» браузеров компенсируется тремя для любой версии IE (а пользователей с таким браузером сейчас 60–70%). Хотя и вызывается всего 2 внешних JavaScript-файла.

Поскольку даже в сжатом виде HTML занимает около 70 КБ, то стоило бы обратить больше внимание на его структуру. Например, убрав лишние пробелы и переводы строк и включив на сервере максимальные уровень сжатия, можно уменьшить его размер на четверть. Если рассмотреть использование спецификации HTML 4.01 для формирования документа, то, скорее всего, файл можно «порезать» еще сильнее.

Хотя некоторые `.gif` изображения можно перевести в `.png` формат и объединить в CSS Sprites, это не очень поможет делу: у нас на странице несколько сотен изображений размером 2–4 Кб, они формата `.jpg` и уже достаточно хорошо оптимизированы. Поскольку какая-то часть страницы все равно кэшируется для снижения нагрузки на сервер, то логично было бы объединить изображения в Image Map 15–20 Кб, ускорив их появление у пользователя в несколько раз.

Для загрузки статической информации на странице уже применяется несколько хостов, поэтому, скорее всего, увеличение их количества ни к чему не приведет.

Если окинуть взглядом общую картину, то для ускорения загрузки сайта можно предпринять некоторые шаги, но почти все они сложны в технологическом плане и не смогут значительно повлиять на скорость загрузки.

habrahabr.ru

Хабрахабр, являясь одной из наиболее популярных социальных сетей в ИТ-сфере, позиционирует себя, в том числе, как достаточно «продвинутый» ресурс. Давайте посмотрим, насколько же он таковым является.

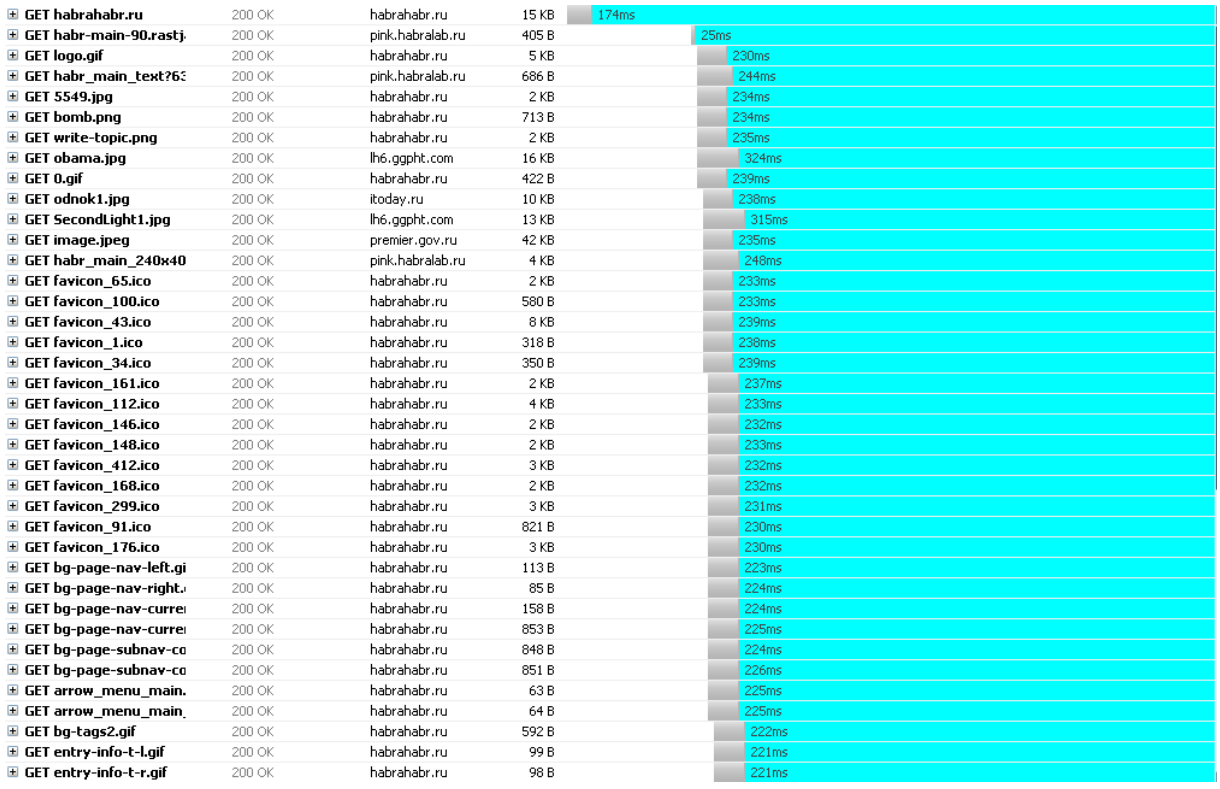


Рис. 70. Результаты анализа загрузки главной страницы habrahabr.ru

Использование 6 CSS- и 15 JavaScript-файлов (подключаемых в `head` страницы) сразу портит все впечатление. Загрузка сайта в самый первый раз (или при отключенном кэше) на небыстром соединении может быть похожа на настоящий кошмар: белый экран в браузере даже не пытается пропасть. Хорошей новостью будет то, что все текстовое содержимое архивируется. Замечательно, но файлы все же нужно объединять.

Далее, смотрим на загрузку картинок, в них большой простыней выделяется favicon-группа. Казалось бы, хорошая идея — выкачивать иконки компаний к себе на сервер и отдавать их пользователю намного быстрее. Однако, в силу того, что получается очень много небольших объектов, разумнее было бы их объединять в Image Map — например, все, что выводятся на главной странице (кэшировать на 10 минут).

Кэширование на сайте уже настроено как должно: для социальной сети количество повторных запросов для просмотра внутренних страниц должно быть минимальным. Однако есть большое количество декоративных изображений, которые можно и нужно объединить в CSS Sprites. После объединения изображений стоит также добавить для них один и два дополнительных хоста: это ускорит их загрузку.

В качестве альтернативных методов есть возможность подключить JavaScript через «отложенную» загрузку и загружать рекламные баннеры только после отображения сайт-страницы.

В качестве послесловия

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 сложилось целостное представление о мире, находящемся между страницей в браузере пользователя и серверными мощностями. О том, что происходит, когда в адресной строке браузера вводятся адрес или имя сайта, какие стадии проходит загрузка страницы, прежде чем полностью завершиться. И самое главное — как этим процессом можно управлять.

Весь представленный выше материал является результатом практического опыта и анализа теоретических основ многих и многих веб-разработчиков по всему миру. Лучшие инженеры искали, находили и внедряли практические решения для того, чтобы их сайты загружались хотя бы немного быстрее. Если объединить все эти знания в один фундаментальный подход, то результат может быть весьма впечатляющим (как уже было продемонстрировано в предыдущей главе).

Клиентская оптимизация лежит на стыке таких областей, как клиентское программирование, серверное программирование, удобство использования, веб-дизайн и многих других. И может быть применимо в любой из указанных сфер деятельности: веб-дизайнер, дорабатывая очень сложный и красивый макет, должен понимать, что время загрузки нарисованной страницы может сильно увеличиться из-за больших размеров графики. Веб-разработчик, отвечающий за серверную часть, должен позаботиться о включении сжатия для HTML-страниц, если на них выводится слишком много данных в результате какой-либо выборки из базы данных.

И так для каждой отрасли и профессии: клиентская оптимизация нужна всем.

А в завершение хочется пожелать только одного — чтобы быстрых сайтов становилось больше и больше.

Контакты для обратной связи с автором приведены по следующему адресу:

http://webo.in/contacts/