Construa interfaces mais rápidas para aplicações web

JavaScript

de Alto Desempenho

O'REILLY® novatec | YAHOO! PRESS

Nicholas C. Zakas

Nicholas C. Zakas


O'REILLY®
Novatec

Tradução autorizada da edição em inglês da obra High Performance JavaScript ISBN 978-0-596-80279-0 © 2010, Nicholas C. Zakas. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

Editor: Rubens Prates
Tradução: Rafael Zanolli
Revisão gramatical: Lia Gabriele Regius
Editoração eletrônica: Camila Kuwabata / Carolina Kuwabata

ISBN do ebook: 978-85-7522-775-6

ISBN do impresso: 978-85-7522-241-6

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Dedico este livro a minha família,minha mãe, meu pai e Greg,
que com seu amor e apoio sempre me ajudaram por todos esses anos.

Sumário

Capítulo 10 ■ Ferramentas

Prefácio

Quando o JavaScript foi introduzido pela primeira vez como parte do Netscape Navigator, em 1996, seu desempenho não era tão importante. A Internet dava ainda seus primeiros passos e era, para todos os efeitos, lenta. De conexões discadas a computadores pessoais precários, navegar na web costumava ser mais uma lição de paciência do que qualquer outra coisa. Usuários já esperavam ter de aguardar para que as páginas fossem carregadas, e isso, quando finalmente acontecia, era motivo de grande celebração.

O objetivo original do JavaScript era o de melhorar a experiência do usuário nas páginas web. Em vez de retornar ao servidor para tarefas simples como a validação de formulários, o JavaScript permitia a incorporação dessas funcionalidades diretamente na página. Ao fazê-lo, economizava uma longa viagem de volta ao servidor. Imagine a frustração de preencher um extenso formulário, enviá-lo e depois aguardar de 30 a 60 segundos apenas para receber uma mensagem dizendo que você preencheu um único campo de modo incorreto. O JavaScript merece o crédito de ter poupado muito do tempo dos usuários iniciais da Internet.

A Internet evolui

Na década que se seguiu, os computadores e a Internet continuaram a evoluir. Para começar, ambos tornaram-se muito mais rápidos. A aceleração dos microprocessadores, a disponibilidade de memória a preços acessíveis e o surgimento de conexões em fibra óptica deram origem a uma nova era na Internet. Com conexões de alta velocidade em abundância, as páginas começaram a se tornar mais pesadas, incluindo mais informações e arquivos multimídia. A rede havia se transformado: de uma paisagem relativamente insípida, formada por documentos

interligados, em um ambiente recheado de designs e interfaces diferentes. Tudo havia mudado. Tudo menos o JavaScript.

O que previamente era utilizado apenas para economizar viagens de retorno ao servidor começava agora a se tornar mais presente. Onde antes havia dúzias de linhas de código em JavaScript, agora havia centenas e eventualmente milhares. A introdução do Internet Explorer 4 e do dynamic HTML (a capacidade de alterar aspectos da página sem a necessidade de recarregá-la) garantiu que a quantidade de JavaScript nas páginas apenas aumentasse com a passagem do tempo.

O último passo importante na evolução dos navegadores foi a introdução do Document Object Model (Modelo de Objeto de Documentos, DOM), uma abordagem unificada ao dynamic HTML que foi adotada pelo Internet Explorer 5, pelo Netscape 6 e pelo Opera. Isso rapidamente foi acompanhado pela padronização do JavaScript no ECMA-262, terceira edição. Com todos os navegadores oferecendo suporte ao DOM e (mais ou menos) à mesma versão do JavaScript, nascia uma plataforma de aplicação web. Apesar desse grande passo adiante, representado pela adoção de uma API comum sobre a qual escrever JavaScript, as engines JavaScript responsáveis pela execução do código permaneciam praticamente idênticas.

Por que a otimização é necessária

As mesmas engines JavaScript que lidavam com páginas contendo apenas algumas dúzias de linhas de JavaScript em 1996 rodam aplicações web com milhares de linhas de JavaScript atualmente. De muitas maneiras, os navegadores ficaram para trás no que se refere ao gerenciamento da linguagem e à preparação das fundações para que o JavaScript pudesse atingir seu sucesso em uma larga escala. Isso se tornou evidente com o Internet Explorer 6, anunciado com elogios por sua estabilidade e velocidade quando foi lançado, mas posteriormente apontado como uma péssima

plataforma devido a seus bugs e lentidão.

Na verdade, o IE6 não havia se tornado mais lento; ele apenas estava tendo de realizar mais do que antes. Os tipos de aplicações web iniciais criados quando o IE6 foi introduzido em 2001 eram muito mais leves e usavam muito menos JavaScript do que os criados em 2005. A diferença na quantidade de código em JavaScript tornava-se evidente conforme a engine de JavaScript do IE6 se esforçava para manter-se atualizada devido à sua rotina estática de coleta de lixo. A engine procurava um número fixo de objetos na memória para determinar quando deveria coletar o lixo. Os desenvolvedores de aplicações web anteriores não costumavam atingir esse limiar, mas com o advento de mais código JavaScript vieram também mais objetos, e aplicações web mais complexas começaram a atingir esse limite com grande frequência. O problema tornou-se claro: os desenvolvedores de JavaScript e de aplicações web tinham evoluído, enquanto as engines JavaScript não.

Ainda que outros navegadores tivessem rotinas de coleta de lixo mais lógicas e um desempenho de tempo de execução melhor, a maioria deles ainda utilizava um interpretador JavaScript para executar o código. A interpretação de código é inerentemente mais lenta do que a compilação, uma vez que há um processo de tradução entre o código e as instruções do computador que devem ser executadas. Não importa quão inteligentes e otimizados os interpretadores se tornem: sempre haverá uma penalização sobre o desempenho.

Compiladores são recheados de todo tipo de otimizações que permitem aos desenvolvedores escrever código da forma que quiserem sem ter de se preocupar com o modo mais eficiente. O compilador pode determinar, com base em uma análise léxica, o que o código está tentando fazer e, então, otimizá-lo produzindo o código mais rápido para a execução da tarefa. Interpretadores têm poucas otimizações desse tipo, o que normalmente significa que o código será executado exatamente da forma em que foi escrito.

Na verdade, o JavaScript obriga o desenvolvedor a realizar as otimizações que um compilador normalmente realizaria em outras linguagens.

Engines JavaScript de nova geração

Em 2008, as engines JavaScript receberam seu primeiro grande incremento de desempenho. O Google introduziu seu novo navegador chamado Chrome. O Chrome foi o primeiro navegador lançado com uma engine otimizadora de JavaScript, chamada V8. A JavaScript V8 é uma engine de compilação just-in-time (JIT) para JavaScript, capaz de produzir código de máquina a partir de código JavaScript e executá-lo. A experiência resultante é uma execução JavaScript incrivelmente rápida.

Outros navegadores logo acompanharam a novidade com suas próprias engines otimizadoras de JavaScript. O Safari 4 apresenta o JIT JavaScript chamado Squirrel Fish Extreme (também chamado de Nitro), e o Firefox 3.5 inclui o TraceMonkey, que otimiza caminhos de código executados com frequência.

Com essas engines JavaScript novas, as otimizações são efetuadas no nível do compilador, onde deveriam ser feitas. Talvez algum dia os desenvolvedores não tenham mais de preocupar-se com otimizações de desempenho em seus códigos. Esse dia, entretanto, ainda não chegou.

O desempenho ainda é uma preocupação

Apesar dos avanços no tempo de execução do núcleo (core) do JavaScript, ainda há aspectos com os quais essas novas engines não são capazes de lidar. Atrasos provocados por latência da rede e operações que afetam a aparência da página são áreas que ainda tem de ser otimizadas de modo adequado pelos navegadores. Enquanto otimizações simples como a distribuição alinhada de funções, code folding[1] e algoritmos de concatenação de linhas são facilmente otimizados pelos compiladores, a estrutura dinâmica e

multifacetada das aplicações web significa que essas otimizações resolvem apenas parte do problema do desempenho.

Ainda que engines JavaScript mais recentes tenham nos dado um vislumbre do que está por vir, oferecendo uma Internet muito mais rápida, as lições de desempenho que aprendemos hoje continuarão sendo relevantes e importantes para o futuro.

As técnicas e abordagens ensinadas neste livro tratam de muitos aspectos diferentes do JavaScript, como tempo de execução, downloads, interação com o DOM, ciclo de vida da página e muito mais. Desses tópicos, apenas alguns, os relacionados ao desempenho do núcleo (core) (ECMAScript), podem tornar-se irrelevantes por avanços futuros nas engines JavaScript, mas isso ainda está para acontecer.

Os outros tópicos tratam de assuntos onde engines JavaScript mais velozes não ajudarão: interação com o DOM, latência de rede, bloqueio e download paralelo de JavaScript e muito mais. Esses tópicos não apenas continuarão a ser relevantes como também devem se tornar áreas de maior foco e pesquisa, conforme o tempo de execução de JavaScript de baixo nível continua melhorando.

Como este livro é organizado

Os capítulos deste livro são organizados com base em um ciclo de vida de desenvolvimento JavaScript normal. Começa, no capítulo 1, com as melhores formas de carregar JavaScript na página. Do capítulo 2 ao 8 tratamos de técnicas de programação específicas capazes de ajudar seu código JavaScript a ser executado o mais rapidamente possível. O capítulo 9 discute as melhores formas de se construir e disponibilizar (deploy) seus arquivos JavaScript em um ambiente de produção, e o capítulo 10 cobre as ferramentas de desempenho que podem ajudá-lo a identificar problemas adicionais uma vez que o código tenha sido disponibilizado. Cinco dos capítulos foram escritos por autores convidados:

- Capítulo 3, *Criação de scripts DOM*, por Stoyan Stefanov

- Capítulo 5, *Strings e expressões regulares*, por Steven Levithan
- Capítulo 7, *Ajax*, por Ross Harmes
- Capítulo 9, *Criação e disponibilização de aplicações JavaScript de alto desempenho*, por Julien Lecomte
- Capítulo 10, *Ferramentas*, por Matt Sweeney

Cada um desses autores é um desenvolvedor web de sucesso, responsável por importantes contribuições à comunidade de desenvolvimento como um todo. Seus nomes aparecem na página inicial de seus respectivos capítulos, identificando facilmente seus trabalhos.

Carregamento do JavaScript

O Capítulo 1, *Carregamento e execução*, tem início tratando do básico em JavaScript: a colocação do código na página. As considerações quanto ao desempenho devem começar desde a colocação do código na página do modo mais eficiente possível. O capítulo fala dos problemas de desempenho associados ao carregamento do código JavaScript e apresenta várias maneiras de mitigar seus efeitos.

Técnica de codificação

Uma grande fonte de problemas no JavaScript diz respeito à existência de código pobremente escrito, usando algoritmos ou utilitários ineficientes. Os sete capítulos seguintes tratam da identificação de códigos problemáticos e da apresentação de alternativas mais rápidas que realizem a mesma tarefa.

O capítulo 2, *Acesso aos dados*, concentra-se em como o JavaScript armazena e acessa dados dentro de um script. O local de armazenamento de seus dados é algo tão importante quanto o que será armazenado. Este capítulo servirá para explicar de que modo conceitos como a cadeia de escopos (scope chain) e a cadeia de protótipos (prototype chain) podem afetar o desempenho geral de seu script.

Stoyan Stefanov, versado no funcionamento interno de um navegador web, escreveu o capítulo 3, *Criação de scripts DOM*. Stoyan explica que a interação DOM é mais lenta do que outras partes do JavaScript devido ao modo como é implementada. Ele cobre todos os aspectos do DOM, incluindo uma descrição de como o repaint e o reflow podem desacelerar seu código.

O capítulo 4, *Algoritmos e controle de fluxo*, explica de que modo paradigmas comuns de programação, como loops e recursão, podem prejudicá-lo quando se trata do desempenho do tempo de execução. Técnicas de otimização como a memoização[2] são discutidas, assim como limitações do tempo de execução do JavaScript nos navegadores.

Muitas aplicações web realizam complexas operações com strings em JavaScript. É por isso que um especialista neles, Steven Levithan, cobre o tópico no capítulo 5: *Strings e expressões regulares*. Desenvolvedores web lutam há anos contra problemas de desempenho nos navegadores causados pela manipulação inadequada de strings, e Steven explicará por que algumas operações são lentas e como podemos contorná-las.

O capítulo 6, *Interfaces responsivas*, dirige nossa atenção à experiência do usuário. O JavaScript pode travar o navegador durante sua execução, deixando usuários extremamente frustrados. Este capítulo discute várias técnicas para garantir que a interface do usuário permaneça sempre responsiva.

No capítulo 7, *Ajax*, Ross Harmes discute as melhores formas de atingir uma comunicação servidor-cliente rápida em JavaScript. Ross fala sobre como formatos de dados diferentes podem afetar o desempenho do Ajax e por que o XMLHttpRequest não é sempre a melhor opção.

O capítulo 8, *Práticas de programação*, é um conjunto de boas práticas específicas da programação JavaScript.

Disponibilização (deployment)

Uma vez que o código tenha sido escrito e testado, é hora de tornarmos as mudanças disponíveis para todos. Entretanto, você não deve simplesmente forçar seus arquivos-fonte em estado bruto (raw source files) para uso na produção. Julien Lecomte mostra como é possível melhorar o desempenho de seu código JavaScript durante a disponibilização no capítulo 9, *Criação e disponibilização de aplicações JavaScript de alto desempenho*. Julien discute a utilização de um sistema integrado para comprimir automaticamente seus arquivos e da compressão HTTP para entregá-los ao navegador.

Testes

Quando todo seu código JavaScript tiver sido disponibilizado, o próximo passo será o início dos testes de desempenho. Matt Sweeney cobre a metodologia e as ferramentas para testes no capítulo 10, *Ferramentas*. Ele discute como podemos utilizar o JavaScript para avaliar o desempenho, bem como descreve ferramentas comuns para avaliação do desempenho do tempo de execução JavaScript e detecção de problemas de desempenho por meio da análise do HTTP sniffing[3].

Para quem se destina este livro?

Este livro é dirigido a desenvolvedores web com uma compreensão intermediária a avançada de JavaScript, que estejam buscando formas de melhorar o desempenho de interfaces de aplicações web.

Convenções adotadas neste livro

As seguintes convenções tipográficas são utilizadas neste livro:

Itálico
: Indica novos termos, URLs, endereços de e-mail, nomes de arquivos e suas extensões.

Largura constante
: Utilizada na listagem de programas, assim como dentro de

parágrafos para fazer referência a elementos de programa como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, declarações e palavras-chave.

Largura constante com negrito

Mostra comandos ou outros textos que devem ser digitados literalmente pelo usuário.

Largura constante com itálico

Mostra texto que deve ser substituído por valores fornecidos pelo usuário ou por valores determinados pelo contexto.

 Significa uma dica, sugestão ou observação geral.

 Indica uma advertência ou um alerta.

Uso dos exemplos de código de acordo com a política da O’Reilly

Este livro está aqui para ajudá-lo a fazer seu trabalho. Em geral, você pode usar o código do livro em seus programas e em sua documentação. Não é preciso entrar em contato conosco para obter permissão, a não ser que você esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes do código presentes neste livro ão requer qualquer tipo de permissão. No entanto, para venda ou distribuição de um CD-ROM com exemplos extraídos dos livros da O’Reilly e necessário obter permissão. Responder a uma pergunta citando este livro e o código de exemplo não requer permissão. Incorporar uma quantidade significativa de código de amostra deste livro na documentação de seu produto requer permissão.

Agradecemos, mas não exigimos, a atribuição de créditos. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo, “*High Performance JavaScript*, por Nicholas C. Zakas. Copyright 2010 Yahoo!, Inc., 978-0-596-80279-0”.

Se acreditar que seu uso de exemplos de código ultrapassa o uso permitido ou a permissão concedida aqui, sinta-se à vontade para

entrar em contato direto com a O'Reilly pelo e-mail *permissions@oreilly.com* ou com a Novatec (*novatec@novatec.com.br*).

Como entrar em contato conosco

Envie comentários e dúvidas sobre este livro para: *novatec@novatec.com.br*.

Temos uma página da web para este livro, onde incluímos a lista de erratas, exemplos e qualquer outra informação adicional.

- Página da edição em português

  http://www.novatec.com.br/livros/javascriptdesemp
- Página da edição original, em inglês

  http://www.oreilly.com/catalog/9780596802790

Para obter mais informações sobre livros da Novatec, acesse nosso site em:

http://www.novatec.com.br

Agradecimentos

Primeiro e acima de tudo, gostaria de agradecer a todos os autores convidados: Matt Sweeney, Stoyan Stefanov, Stephen Levithan, Ross Harmes e Julien Lecomte. Poder contar com sua experiência e seu conhecimento combinados como parte deste livro tornou o processo ainda mais estimulante e o resultado final ainda mais completo.

Agradeço também a todos os gurus de desempenho que tive a oportunidade de conhecer e com quem pude interagir, especialmente Steve Souders, Tenni Theurer e Nicole Sullivan. Vocês três ajudaram a expandir meu horizonte no que se refere a desempenho na rede, e sou incrivelmente grato por isso.

Um grande obrigado a todos que revisaram o livro antes da publicação, incluindo Ryan Grove, Oliver Hunt, Matthew Russell, Ted Roden, Remy Sharp e Venkateswaran Udayasankar. O feedback

que me forneceram foi imprescindível na preparação do livro.

Por fim, um enorme agradecimento a todos da O'Reilly e do Yahoo! que tornaram este livro possível. Desde que me juntei à empresa, em 2006, sempre quis escrever um livro para o Yahoo!, e a Yahoo! Press foi uma ótima forma de tornar isso realidade.

---

1 N.T.: Code folding é um recurso de alguns editores de código-fonte e IDEs, que permite ao usuário ocultar e exibir seletivamente seções de um arquivo como parte de operações de edição rotineiras. Isso permite que o usuário gerencie grandes quantidades de texto enquanto visualiza apenas as subseções que são especificamente relevantes no momento. (Fonte: Wikipédia)

2 N.T.: Memoização é uma técnica de otimização utilizada principalmente para acelerar a execução de programas de computadores fazendo com que as chamadas feitas às funções evitem repetir o cálculo de resultados para inputs já processados. (Fonte: Wikipédia)

3 N.T.: Sniffing, em rede de computadores, é o procedimento realizado por uma ferramenta conhecida como Sniffer (também conhecido como Packet Sniffer, Analisador de Rede, Analisador de Protocolo, Ethernet Sniffer em redes do padrão Ethernet ou ainda Wireless Sniffer em redes weireless). Essa ferramenta, constituída de um software ou hardware, é capaz de interceptar e registrar o tráfego de dados em uma rede de computadores. Conforme o fluxo de dados trafega na rede, o sniffer captura cada pacote e eventualmente decodifica e analisa o seu conteúdo de acordo com o protocolo definido em um RFC ou outra especificação. (Fonte: Wikipédia)

CAPÍTULO 1

Carregamento e execução

Pode-se dizer que o desempenho do JavaScript no navegador é a questão de usabilidade mais importante que os desenvolvedores enfrentam. Trata-se de um problema complexo devido à natureza bloqueadora do JavaScript, que significa que nada mais pode ocorrer enquanto o código em JavaScript é executado. Na verdade, a maioria dos navegadores utiliza um único processo tanto para atualizações à interface do usuário (user interface, ou UI) quanto para a execução do JavaScript, de modo que apenas uma dessas tarefas pode ocorrer em um dado momento. Quando mais tempo demorar a execução de um código em JavaScript, mais tempo levará para que o navegador possa responder à interação do usuário.

Em um nível básico, isso significa que a simples presença da tag `<script>` já é suficiente para fazer com que a página aguarde o parsing do script e sua execução. É irrelevante se o código JavaScript em si está embutido na tag ou incluso em um arquivo externo: o download e a exibição da página têm de ser interrompidos, aguardando até que o script termine antes de prosseguir. Essa é uma parte necessária do ciclo de vida da página, uma vez que o script pode provocar mudanças na renderização conforme é executado. O exemplo típico é a utilização do `document.write()` no meio da página (muitas vezes usado na publicidade). Por exemplo:

```
<html>
<head>
  <title>Script Example</title>
</head>
<body>
  <p>
  <script type="text/javascript">
```

```
    document.write("The date is " + (new Date()).toDateString());
  </script>
  </p>
</body>
</html>
```

Quando o navegador encontra uma tag <script>, assim como nessa página HTML, não há como saber se o JavaScript irá inserir conteúdo no <p>, introduzir elementos adicionais ou talvez até fechar a tag. Assim, o navegador interrompe o processamento da página, executa o código JavaScript e então continua o parsing da página. O mesmo ocorre para código JavaScript carregado pelo atributo src; o navegador deve primeiro fazer o download do código a partir do arquivo externo e, somente então, fazer o parsing e a execução do código. A renderização da página e a interação do usuário são completamente bloqueadas durante esse intervalo.

As duas fontes principais de informações sobre o efeito do JavaScript no desempenho do download de páginas são a equipe do Yahoo! Exceptional Performance (*http://developer.yahoo.com/performance/*) e Steve Souders, autor de *High Performance Web Sites* (O'Reilly) e *Even Faster Web Sites* (O'Reilly). Este capítulo foi bastante influenciado pela pesquisa combinada dessas duas fontes.

Posicionamento do script

A especificação HTML 4 indica que uma tag <script> deve ser posicionada dentro de uma tag <head> ou <body> em um documento HTML e que pode aparecer um número qualquer de vezes em cada uma delas. Tradicionalmente, tags <script> utilizadas para carregar arquivos JavaScript externos aparecem em <head>, junto de tags <link> para carregar arquivos CSS externos e outras metainformações sobre a página. A teoria era a de que é melhor manter o máximo de dependências de estilo e comportamento juntas, carregando-as primeiro para que a página surja e se comporte corretamente. Por exemplo:

```
<html>
<head>
  <title>Script Example</title>
  <-- Exemplo de posicionamento JavaScript ineficiente -->
```

```
  <script type="text/javascript" src="file1.js"></script>
  <script type="text/javascript" src="file2.js"></script>
  <script type="text/javascript" src="file3.js"></script>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>
</body>
</html>
```

Ainda que esse código possa parecer inócuo, ele apresenta um problema de desempenho grave: há três arquivos JavaScript sendo carregados no <head>. Uma vez que cada tag <script> impede que a página continue a ser renderizada até que o código JavaScript carregue e execute totalmente, o desempenho aparente dessa página será penalizado. Tenha em mente que navegadores não começam a renderizar nada na página até que a tag <body> de abertura seja encontrada. Colocar scripts no topo da página desse modo geralmente leva a uma demora perceptível, muitas vezes na forma de uma página vazia em branco, antes que o usuário possa começar a ler e interagir com o documento. Para compreender melhor como isso ocorre, é interessante observar um diagrama de cascata mostrando quando cada recurso será baixado. A figura 1.1 mostra quando ocorrerá o download de cada script e do arquivo da folha de estilo conforme a página carrega.

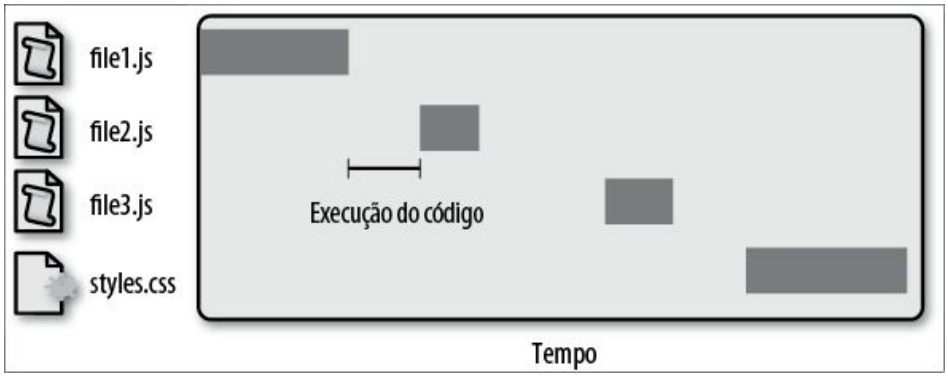


Figura 1.1 – A execução de código JavaScript bloqueia o download de outros arquivos.

A figura 1.1 mostra um padrão interessante. O primeiro arquivo JavaScript começa a ser baixado e bloqueia o download de qualquer outro arquivo nesse intervalo. Além disso, há um atraso

entre o momento no qual o arquivo *file1.js* termina de ser copiado e o momento no qual *file2.js* começa a ser baixado. Esse espaço é o período necessário para que o código contido em *file1.js* seja completamente executado. Cada arquivo deve esperar a conclusão do download e a execução do anterior para poder ser baixado. Nesse intervalo, o usuário encontra uma página em branco, enquanto os arquivos vão sendo baixados um de cada vez. Esse é o comportamento da maioria dos navegadores atuais.

O Internet Explorer 8, o Firefox 3.5, o Safari 4 e o Chrome 2 permitem downloads paralelos de arquivos JavaScript. Isso é algo positivo, visto que não se permite que tags <script> impeçam outras tags <script> de efetuar os downloads de recursos externos. Infelizmente, downloads de JavaScript ainda bloqueiam o download de outros recursos, como imagens. Mesmo que o download de um script não bloqueie o carregamento de outros scripts, a página ainda deve esperar para que o código JavaScript seja baixado e executado antes de continuar. Assim, mesmo que os mais recentes navegadores tenham passado por melhorias de desempenho permitindo downloads paralelos, esse problema ainda não foi completamente solucionado. O bloqueio de scripts ainda continua a ser um incômodo.

Uma vez que os scripts bloqueiam o download de todos os tipos de recursos da página, é recomendado que todas as tags <script> sejam posicionadas o mais próximo possível do fim da tag <body> para que não afetem o download da página. Por exemplo:

```
<html>
<head>
  <title>Script Example</title>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>
    <-- Exemplo do posicionamento recomendado do script -->
    <script type="text/javascript" src="file1.js"></script>
    <script type="text/javascript" src="file2.js"></script>
    <script type="text/javascript" src="file3.js"></script>
```

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

Esse código representa o posicionamento adequado das tags <script> em um arquivo HTML. Mesmo que os downloads dos scripts continuem a bloquear uns aos outros, o restante da página já terá sido baixado e já estará sendo exibido para o usuário, sem dar a impressão de se tratar de uma página inteiramente lenta. Essa é a primeira regra da equipe Yahoo! Exceptional Performance: coloque os scripts no final.

Agrupamento de scripts

Uma vez que cada tag <script> bloqueia a renderização da página durante o download inicial, é interessante limitarmos o número total de tags <script> contidas na página. Isso diz respeito tanto a scripts embutidos quanto àqueles encontrados em arquivos externos. Toda vez que uma tag <script> for encontrada durante o parsing de uma página HTML, haverá um atraso enquanto o código é executado. Minimizar esses atrasos melhora o desempenho geral da página.

Steve Souders também constatou que um script embutido posicionado depois de uma tag <link> que faz referência a uma folha de estilo externa faz com que os navegadores fiquem bloqueados enquanto aguardam o download da folha de estilo. Isso objetiva garantir que o script embutido apresente a informação de estilo mais correta para sua execução. Por essa razão, Souders recomenda que nunca se coloque um script embutido depois de uma tag <link>.

O problema é um tanto diferente quando se trata de arquivos JavaScript externos. Cada solicitação HTTP envolve um custo adicional de desempenho. Assim, fazer o download de um arquivo de 100 KB é mais rápido que fazer o download de quatro arquivos de 25 KB. Portanto, é interessante limitar o número de arquivos de script externos aos quais sua página faz referência.

Normalmente um grande website, ou aplicação web, contará com vários arquivos JavaScript necessários. É possível minimizar seu impacto sobre o desempenho concatenando esses arquivos juntos em um único arquivo e chamando-o com apenas uma tag <script>. A concatenação pode ocorrer off-line com uma ferramenta integrada

(discutida no capítulo 9) ou em tempo real com uma ferramenta como o Yahoo! combo handler.

O Yahoo! criou o combo handler para utilização na distribuição de arquivos da biblioteca da Yahoo! User Interface (YUI) por meio de sua Content Delivery Network (Rede de entrega de conteúdo, ou CDN[1]). Qualquer website pode acessar vários arquivos da YUI usando uma URL gerenciada pelo combo handler e especificando quais arquivos devem ser incluídos. Por exemplo, esta URL inclui dois arquivos: *http://yui.yahooapis.com/combo ? 2.7.0/build/yahoo/yahoo-min.js&2.7.0/build/event/event-min.js*.

Essa URL carrega as versões 2.7.0 dos arquivos *yahoo-min.js* e *event-min.js*. Esses arquivos existem separadamente em cada servidor, mas são combinados quando a URL é solicitada. Em vez de utilizar duas tags <script> (uma para carregar cada arquivo), uma única tag <script> pode ser usada para carregar os dois:

```
<html>
<head>
  <title>Script Example</title>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>
    <-- Exemplo do posicionamento recomendado do script -->
    <script type="text/javascript" src="
http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&
2.7.0/build/event/event-min.js "></script>
</body>
</html>
```

Esse código apresenta uma única tag <script> ao final da página, que atua carregando múltiplos arquivos JavaScript, exemplificando a boa prática ideal para a inclusão de JavaScript externo em uma página HTML.

Scripts não bloqueadores

A tendência do JavaScript em bloquear processos do navegador, tanto solicitações HTTP quanto atualizações da UI, é o problema de

desempenho mais notável que se apresenta aos desenvolvedores. Manter seus arquivos JavaScript pequenos e limitar o número de solicitações HTTP são apenas os primeiros passos para a criação de uma aplicação web mais responsiva. Quanto mais rica for a funcionalidade exigida pela aplicação, mais código JavaScript será necessário. Dessa forma, manter o volume de código-fonte pequeno nem sempre será uma opção. Limitar seu site ao download de um único grande arquivo JavaScript servirá apenas para bloquear o navegador por um longo período, mesmo que se trate de uma única solicitação HTTP. Para contornar essa situação, você deve adicionar JavaScript à página de uma forma que não bloqueie o navegador.

O segredo para scripts não bloqueadores é carregar o código-fonte JavaScript após o término do carregamento da página. Em termos técnicos, isso significa realizar o download do código depois do disparo do evento `load` de `window`. Há algumas técnicas que conduzem a esse resultado.

Scripts adiados

O HTML 4 define um atributo adicional para a tag `<script>` chamado `defer`. O atributo `defer` indica que o script contido no elemento não deverá modificar o DOM, fazendo com que sua execução seja adiada até um momento futuro. O atributo `defer` é aceito apenas no Internet Explorer 4+ e no Firefox 3.5+, o que faz com que sua utilização não seja ideal caso se pretenda uma solução compatível com múltiplos navegadores. Em outros navegadores, o atributo `defer` é simplesmente ignorado, fazendo com que a tag `<script>` seja tratada em seu modo padrão (bloqueadora). Ainda assim, essa solução é útil caso inclua os navegadores de seu público-alvo. A seguir temos um exemplo de sua utilização:

```
<script type="text/javascript" src="file1.js" defer></script>
```

Uma tag `<script>` com `defer` pode ser posicionada em qualquer ponto do documento. O download do arquivo JavaScript começará no momento em que for feito o parse da tag `<script>`, mas o código não será executado até que o DOM tenha sido completamente

carregado (antes que o manipulador de evento onload seja chamado). Quando um arquivo JavaScript com defer é baixado, ele não bloqueia os outros processos do navegador, o que significa que o download desses arquivos pode ser feito junto ao de outros arquivos da página.

Qualquer elemento <script> marcado com o defer não será executado até que o DOM tenha carregado completamente: isso vale para scripts embutidos e para arquivos de script externos. A página que a seguir demonstra como o atributo defer altera o comportamento dos scripts:

```
<html>
<head>
  <title>Script Defer Example</title>
</head>
<body>
  <script defer>
    alert("defer");
  </script>
  <script>
    alert("script");
  </script>
  <script>
    window.onload = function() {
      alert("load");
    };
  </script>
</body>
</html>
```

Esse código exibe três alertas conforme a página é processada. Em navegadores que não aceitam o atributo defer, a ordem dos alertas é "*defer*", "*script*" e "*load*". Em navegadores que o aceitam, a ordem dos alertas é "*script*", "*defer*" e "*load*". Observe que o elemento <script> deferido não será executado antes do fim do segundo, mas será executado antes que o manipulador de evento onload seja chamado.

Caso seus navegadores almejados incluam apenas o Internet Explorer e o Firefox 3.5, essa forma de aplicação do defer pode ser

útil. Se, por outro lado, você tiver de oferecer compatibilidade a um número maior de navegadores, há outras soluções que funcionam de modo mais consistente.[2]

Elementos de script dinâmicos

O Document Object Model (DOM) permite que você crie de modo dinâmico praticamente qualquer parte de um documento HTML utilizando JavaScript. Em seu cerne, o elemento <script> não é diferente de qualquer outro elemento de uma página: referências podem ser recuperadas pelo DOM, podendo ser movidas, removidas do documento e até mesmo criadas. Um novo elemento <script> pode ser muito facilmente criado por meio de métodos DOM padronizados:

```
var script = document.createElement("script");
script.type = "text/javascript";
script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

Esse novo elemento <script> carrega o arquivo-fonte *file1.js*. Seu download começa assim que o elemento é adicionado à página. O ponto a ser destacado nessa técnica é o de que o download e a execução do arquivo não bloquearão outros processos da página, independentemente do local onde esse download seja iniciado. Você pode até mesmo posicionar esse código no <head> de um documento sem afetar o restante da página (com exceção da conexão HTTP que será utilizada para o download do arquivo).[3]


Geralmente é mais seguro adicionar novos nodos <script> ao elemento <head> em vez do elemento <body>, especialmente se o código estiver em execução durante o carregamento da página. O Internet Explorer pode apresentar um erro de “operação abortada” caso todo o conteúdo de <body> ainda não tenha sido carregado.

Quando um arquivo é carregado por meio de um nodo de script dinâmico, o código recuperado em geral é executado imediatamente (exceto no Firefox e no Opera, que aguardarão até que qualquer nodo de script dinâmico prévio tenha sido executado). Isso funciona bem se o script for autoexecutável, mas pode ser problemático caso

o código contenha apenas interfaces a serem utilizadas por outros scripts na página. Nesse caso, você terá de monitorar o carregamento total do código até que esteja pronto para uso. Isso pode ser realizado por meio de eventos disparados pelo nodo <script> dinâmico.

Tanto o Firefox quanto o Opera, o Chrome e o Safari 3+ disparam um evento load quando o src de um elemento <script> for recuperado. Dessa forma, você pode ser notificado da prontidão do script por um alerta deste evento:

```
var script = document.createElement("script")
script.type = "text/javascript";
// Firefox, Opera, Chrome, Safari 3+
script.onload = function() {
  alert("Script loaded!");
};
script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

O Internet Explorer aceita uma implementação alternativa que dispara um evento readystatechange. Há uma propriedade readyState no elemento <script> que é alterada inúmeras vezes durante o download de um arquivo externo. Há cinco valores possíveis para readyState:

"uninitialized"
: O estado padrão.

"loading"
: O download teve início.

"loaded"
: O download está completo.

"interactive"
: Os dados já foram baixados, mas ainda não estão inteiramente disponíveis.

"complete"
: Todos os dados estão prontos para uso.

A documentação da Microsoft para o readyState e para cada um de

seus possíveis valores parece indicar que nem todos os estados serão utilizados durante a vida do elemento <script>, mas não há indicação de quais deles estarão sempre presentes. Na prática, os dois estados de maior interesse para nós são "loaded" e "complete". O Internet Explorer é inconsistente quanto a qual desses dois valores indica o estado final, já que algumas vezes o elemento <script> atingirá o estado "loaded" sem nunca atingir "complete", enquanto em outras ocasiões "complete" será atingido sem que "loaded" chegue a ser utilizado. O modo mais seguro de usar o evento readystatechange é conferindo a presença desses dois estados e removendo o manipulador do evento quando qualquer um deles ocorrer (para garantir que o evento não seja manipulado duas vezes):

```
var script = document.createElement("script")
script.type = "text/javascript";

// Internet Explorer
script.onreadystatechange = function() {
  if (script.readyState == "loaded" || script.readyState == "complete") {
    script.onreadystatechange = null;
    alert("Script loaded.");
  }
};

script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

Na maioria dos casos, você desejará utilizar uma única abordagem para carregar dinamicamente arquivos JavaScript. A função seguinte encapsula tanto a funcionalidade padrão quanto a específica ao IE:

```
function loadScript(url, callback) {
  var script = document.createElement("script")
  script.type = "text/javascript";

  if (script.readyState) { // IE
    script.onreadystatechange = function() {
      if (script.readyState == "loaded" || script.readyState == "complete") {
        script.onreadystatechange = null;
        callback();
      }
    };
    } else { // Outros
```

```
        script.onload = function() {
          callback();
        };
      }
      script.src = url;
      document.getElementsByTagName("head")[0].appendChild(script);
    }
```

Essa função aceita dois argumentos: a URL do arquivo JavaScript a ser acessado e uma função de callback a ser executada quando o JavaScript tiver sido completamente carregado. A detecção de atributos é utilizada para determinar qual manipulador de evento deve monitorar o progresso do script. O último passo deve ser a designação da propriedade `src` e a adição do elemento `<script>` à página. A função `loadScript()` é usada da seguinte forma:

```
loadScript("file1.js", function() {
  alert("File is loaded!");
});
```

Você pode carregar dinamicamente quantos arquivos JavaScript forem necessários em uma página, mas lembre-se de levar em consideração a ordem em que devem ser carregados. De todos os navegadores principais, apenas o Firefox e o Opera garantem que a ordem da execução dos scripts ocorrerá de acordo com sua especificação. Outros navegadores efetuarão o download e a execução de vários códigos na ordem na qual eles retornarem do servidor. Você pode garantir a ordem efetuando um encadeamento dos downloads, da seguinte maneira:

```
loadScript("file1.js", function() {
  loadScript("file2.js", function() {
    loadScript("file3.js", function() {
      alert("All files are loaded!");
    });
  });
});
```

Esse código apenas começa a carregar *file2.js* depois que *file1.js* estiver disponível, e também posterga o download de *file3.js* até que *file2.js* esteja disponível. Ainda que seja possível, a administração dessa abordagem pode se tornar um pouco complicada caso

existam vários arquivos que devam ser baixados e executados.

Caso a ordem dos vários arquivos seja importante, a abordagem preferível é a concatenação dos arquivos em um só, onde cada parte ocupa a ordem correta. Esse arquivo único poderá então ser baixado para que se acesse todo o código de uma única vez (uma vez que isso ocorrerá de modo assíncrono, não haverá penalidade pelo download de um arquivo maior).

O carregamento dinâmico de scripts é o padrão mais comum para o desbloqueio de downloads JavaScript, devido à sua compatibilidade com vários navegadores e à facilidade de uso.

Injeção de script com XMLHttpRequest

Outra abordagem para o desbloqueio de scripts é o acesso ao código JavaScript por meio de um objeto XMLHttpRequest (XHR), seguido pela injeção do script na página. Essa técnica envolve a criação de um objeto XHR, o download do arquivo JavaScript e a injeção do código JavaScript na página valendo-se de um elemento <script> dinâmico. Aqui está um exemplo simples:

```
var xhr = new XMLHttpRequest();
xhr.open("get", "file1.js", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    if (xhr.status >= 200 && xhr.status < 300 || xhr.status == 304) {
      var script = document.createElement("script");
      script.type = "text/javascript";
      script.text = xhr.responseText;
      document.body.appendChild(script);
    }
  }
};
xhr.send(null);
```

Esse código envia uma solicitação GET pelo arquivo *file1.js*. O manipulador de evento onreadystatechange busca um readyState de valor 4 e então verifica se o código de status HTTP é válido (qualquer valor dentro de 200 é uma resposta válida, e 304 significa uma resposta armazenada em cache). Caso uma resposta válida seja recebida,

um novo elemento <script> será criado, atribuindo-se à sua propriedade text o valor de responseText recebido do servidor. Ao fazê-lo, essencialmente criamos um elemento <script> com código embutido. Assim que o novo elemento <script> for adicionado ao documento, o código será executado e estará pronto para uso.

A vantagem principal dessa abordagem é a de que você poderá efetuar o download do código JavaScript sem ter de executá-lo imediatamente. Uma vez que o código será retornado fora de uma tag <script>, ele não será automaticamente executado, permitindo que você prorrogue sua aplicação até o momento desejado. Outra vantagem é a de que esse mesmo código funciona em todos os navegadores modernos sem casos de exceção.

A principal limitação da abordagem é que o arquivo JavaScript tem de estar localizado no mesmo domínio da página que o solicita, o que torna impossível o download a partir de CDNs. Por essa razão, a injeção de script XHR não é normalmente utilizada em aplicações de grande escala.

Padrão de desbloqueio recomendado

Para o carregamento de uma quantidade significativa de JavaScript em uma página, a abordagem recomendada é um processo de dois passos: primeiro, inclua o código necessário para carregar dinamicamente o JavaScript e depois carregue o restante do código necessário para a inicialização da página. Como a primeira parte do código deve ter o menor tamanho possível, potencialmente contendo apenas a função loadScript(), seu download e sua execução ocorrerão com rapidez, não causando grande interferência na página. Com o código inicial posicionado, utilize-o para carregar o JavaScript restante. Por exemplo:

```
<script type="text/javascript" src="loader.js"></script>
<script type="text/javascript">
  loadScript("the-rest.js", function() {
    Application.init();
  });
</script>
```

Posicione esse código de carregamento logo acima da tag </body> de encerramento. Ao fazê-lo você terá vários benefícios. Primeiro, como discutimos anteriormente, isso garante que a execução do JavaScript não impeça a exibição do restante da página. Em segundo lugar, quando terminar o download do segundo arquivo JavaScript, todo o DOM necessário para a aplicação já terá sido criado e estará pronto para interação, o que evita a necessidade da busca por outro evento (como o window.onload) para que se saiba se a página está pronta para inicialização.

Outra opção é a incorporação da função loadScript() diretamente na página, evitando outra solicitação HTTP. Por exemplo:

```
<script type="text/javascript">
  function loadScript(url, callback) {
    var script = document.createElement("script")
    script.type = "text/javascript";
    if (script.readyState) { // IE
      script.onreadystatechange = function() {
        if (script.readyState == "loaded" ||
            script.readyState == "complete") {
          script.onreadystatechange = null;
          callback();
        }
      };
    } else { // Outros
      script.onload = function() {
        callback();
      };
    }
    script.src = url;
    document.getElementsByTagName("head")[0].appendChild(script);
  }
  loadScript("the-rest.js", function() {
    Application.init();
  });
</script>
```

Caso decida utilizar essa última abordagem, é recomendado que minifique [4] (minify) o script inicial com uma ferramenta como o YUI Compressor (consulte o capítulo 9) para obter o menor impacto

sobre sua página.

Uma vez que o download do código para a inicialização da página esteja completo, você estará livre para continuar utilizando o loadScript() para carregar funcionalidades adicionais em sua página sempre que necessário.

A abordagem YUI 3

O conceito de um volume inicial pequeno de código na página, seguido pelo download de funcionalidades adicionais é a ideia central do design com a biblioteca YUI 3. Para utilizar a YUI 3 em sua página, comece pela inclusão do arquivo básico (seed) da YUI:

```
<script type="text/javascript"
src="http://yui.yahooapis.com/combo?3.0.0/build/yui/yui-min.js"></script>
```

O arquivo básico tem em torno de 10 KB (6 KB quando se utiliza o GZip) e inclui funcionalidades suficientes para o download de quaisquer outros componentes YUI a partir do Yahoo! CDN. Por exemplo, caso deseje utilizar a função DOM, você pode especificar seu nome ("dom") com o método use() da YUI e depois fornecer uma callback que será executada quando o código estiver pronto:

```
YUI().use("dom", function(Y) {
  Y.DOM.addClass(document.body, "loaded");
});
```

Esse exemplo cria uma nova instância do objeto YUI e, em seguida, chama o método use(). O arquivo básico contém toda a informação referente aos nomes de arquivos e dependências, assim, ao especificar "dom", você constrói uma URL combo-handler, com todos os arquivos de dependências corretos, criando um elemento de script dinâmico e executando esses arquivos. Quando todo o código estiver disponível, o método callback será chamado e a instância YUI será passada como argumento, permitindo que você utilize imediatamente a funcionalidade baixada.

A biblioteca LazyLoad

Para uma ferramenta mais genérica, Ryan Grove do Yahoo! Search

criou a biblioteca LazyLoad (disponível em *http://github.com/rgrove/lazyload/*). A LazyLoad é uma versão mais poderosa da função loadScript(). Ao ser minificada por uma ferramenta como o YUI Compressor, o arquivo LazyLoad terá em torno de 1,5 KB (minificado e não comprimido pelo GZip). Temos aqui um exemplo de sua utilização:

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
  LazyLoad.js("the-rest.js", function() {
    Application.init();
  });
</script>
```

O LazyLoad também é capaz de efetuar o download de vários arquivos JavaScript e garantir que sejam executados na ordem correta em todos os navegadores. Para carregar diversos arquivos JavaScript, simplesmente passe um array de URLs ao método LazyLoad.js():

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
  LazyLoad.js(["first-file.js", "the-rest.js"], function() {
    Application.init();
  });
</script>
```

Ainda que o download dos arquivos seja efetuado de modo que não bloqueie outras operações, pela utilização do carregamento dinâmico de scripts, é recomendado que se tenha o menor número possível de arquivos. Cada download ainda será uma solicitação HTTP separada, e a função callback não será efetuada até que o download e a execução de todos os arquivos tenham sido concluídos.


O LazyLoad também é capaz de carregar arquivos CSS dinamicamente. Isso representa uma questão de menor importância, uma vez que o download de arquivos CSS sempre é feito em paralelo, sem bloquear outras atividades das páginas.

A biblioteca LABjs

Outra abordagem para utilização de JavaScript não bloqueador é a

LABjs (*http://labjs.com/*), uma biblioteca de código aberto escrita por Kyle Simpson com a colaboração de Steve Souders. Essa biblioteca oferece um controle minucioso sobre o processo de carregamento e tenta efetuar o download em paralelo do maior volume possível de código. A LABjs também é relativamente pequena, 4,5 KB (minificada, não comprimida com o GZip), e por isso tem um custo mínimo sobre o desempenho da página. A seguir, temos um exemplo de sua utilização:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
  $LAB.script("the-rest.js")
    .wait(function() {
      Application.init();
    });
</script>
```

O método `$LAB.script()` é utilizado para definir o download de um arquivo JavaScript, enquanto `$LAB.wait()` é usado para indicar que a execução deve aguardar o download e a execução do arquivo antes da execução de uma dada função. A LABjs estimula o encadeamento, de modo que todos os métodos retornam uma referência ao objeto `$LAB`. Para efetuar o download de vários arquivos JavaScript, basta encadear outra chamada `$LAB.script()`:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
  $LAB.script("first-file.js")
    .script("the-rest.js")
    .wait(function() {
      Application.init();
    });
</script>
```

O que distingue a LABjs das outras bibliotecas é sua capacidade de gerenciar dependências. A inclusão normal, com tags `<script>`, significa que cada arquivo será baixado (seja em sequência ou em paralelo, como mencionado anteriormente) e, então, executado de modo sequencial. Em alguns casos isso é verdadeiramente necessário, mas em outros não.

A LABjs permite especificar quais arquivos devem aguardar por

outros utilizando a função wait(). No exemplo prévio, não podemos garantir que o código em *first-file.js* será executado antes do código *the-rest.js*. Para ter essa certeza, você deve adicionar uma chamada wait() depois do primeiro script():

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
  $LAB.script("first-file.js").wait()
    .script("the-rest.js")
    .wait(function() {
      Application.init();
  });
</script>
```

Agora, temos certeza de que o código em *first-file.js* será executado antes do código em *the-rest.js*, ainda que o conteúdo dos arquivos seja baixado em paralelo.

Resumo

Administrar a utilização de JavaScript no navegador é algo delicado, já que a execução do código bloqueia outros processos, como a renderização da interface do usuário. Toda vez que uma tag <script> for encontrada, a página terá de parar e aguardar o download do código (caso seja externo) e sua execução antes que seu restante possa ser processado. Há, entretanto, várias maneiras de minimizar o impacto ao desempenho causado pelo JavaScript:

- Coloque todas as tags <script> ao final da página, dentro da tag </body> de encerramento. Isso garante que a página esteja praticamente renderizada antes que a execução do script tenha início.
- Agrupe seus scripts. Quanto menos tags <script> existirem na página, mais rapidamente ela será carregada e se tornará interativa. Isso é verdadeiro tanto para as tags <script> que carregam arquivos JavaScript externos, quanto para aquelas com código embutido.
- Há várias formas de efetuar o download de JavaScript de modo não bloqueador:

- Utilize o atributo defer da tag <script> (apenas no Internet Explorer e no Firefox 3.5+).
- Crie dinamicamente elementos <script> para efetuar e executar o código.
- Faça o download do código JavaScript utilizando um objeto XHR e, então, injete o código na página.

Ao utilizar essas estratégias você será capaz de melhorar muito o desempenho de uma aplicação web que demanda grande quantidade de código JavaScript.

---

1 N.T.: Uma content delivery network ou content distribution network (CDN) é uma rede de computadores que contém cópias de dados, posicionados em diversos pontos da rede de modo a maximizar a largura de banda para o acesso a dados por clientes. Um cliente acessa uma cópia dos dados próxima à ele, em lugar de todos os clientes terem de acessar o mesmo servidor central, evitando, dessa forma, gargalos próximos a esse servidor. (Fonte: Wikipédia)

2 Uma alteração na definição do atributo defer no HTML5 foi implementada no Internet Explorer 8 (modo standards) e no Firefox 3.6. Em a forma, o exemplo anterior não funcionará corretamente nesses navegadores. Quando executado em modo de compatibilidade, o Internet Explorer ainda apresenta o antigo comportamento.

3 Consulte "*The dreaded operation aborted error*" em *http://www.nczonline.net/blog/2008/03/17/the -dreaded-operation-aborted-error/* para uma análise mais profunda desse problema.

4 N.T.: A minificação, em linguagens de programação e, especialmente em JavaScript, é o processo de remoção de todos os caracteres desnecessários do código-fonte, sem alterar sua funcionalidade. Esses caracteres desnecessários geralmente incluem caracteres de espaços em branco, caracteres nova linha, comentários e às vezes delimitadores de bloco; que são utilizados para adicionar legibilidade ao código, mas que não são necessários à sua execução. (Fonte: Wikipédia)

CAPÍTULO 2

Acesso aos dados

Um dos problemas clássicos da ciência da computação é a determinação do local onde os dados devem ser armazenados para uma leitura e gravação ideais. O local em que os dados são armazenados está diretamente relacionado à rapidez com que poderão ser acessados durante a execução do código. Quando se trata de JavaScript, esse problema é relativamente simplificado, já que há apenas um número pequeno de opções para o armazenamento de dados. Entretanto, assim como no que se refere a outras linguagens, a localização pode afetar consideravelmente a velocidade em que os dados serão acessados no futuro. Há quatro locais básicos onde dados podem ser acessados em JavaScript:

Valores literais

Qualquer valor que representa apenas a si mesmo e que não se encontra armazenado em um local particular. A linguagem JavaScript pode representar sequências de caracteres (strings), números, valores booleanos, objetos, arrays, funções, expressões regulares e os valores especiais `null` e `undefined` como literais.

Variáveis

Qualquer localização definida pelo desenvolvedor para armazenamento de dados criados pela utilização da palavra-chave `var`.

Itens de array

Uma localização numericamente indexada dentro de um objeto `Array` do JavaScript.

Membros de objetos

Uma localização indexada por strings dentro de um objeto JavaScript.

Cada um desses locais para armazenamento vem acompanhado de um custo particular associado a operações de leitura e escrita envolvendo os dados. Na maioria dos casos, a diferença de desempenho entre o acesso de informações a partir de um valor literal comparada a uma variável local é mínima. Já o acesso a informações contidas em itens de arrays e membros de objetos é mais custoso. Para determinar qual é o mais dispendioso, temos que analisar qual o navegador utilizado. A figura 2.1 mostra a velocidade relativa do acesso de 200 mil valores a partir de cada um desses locais em vários navegadores.

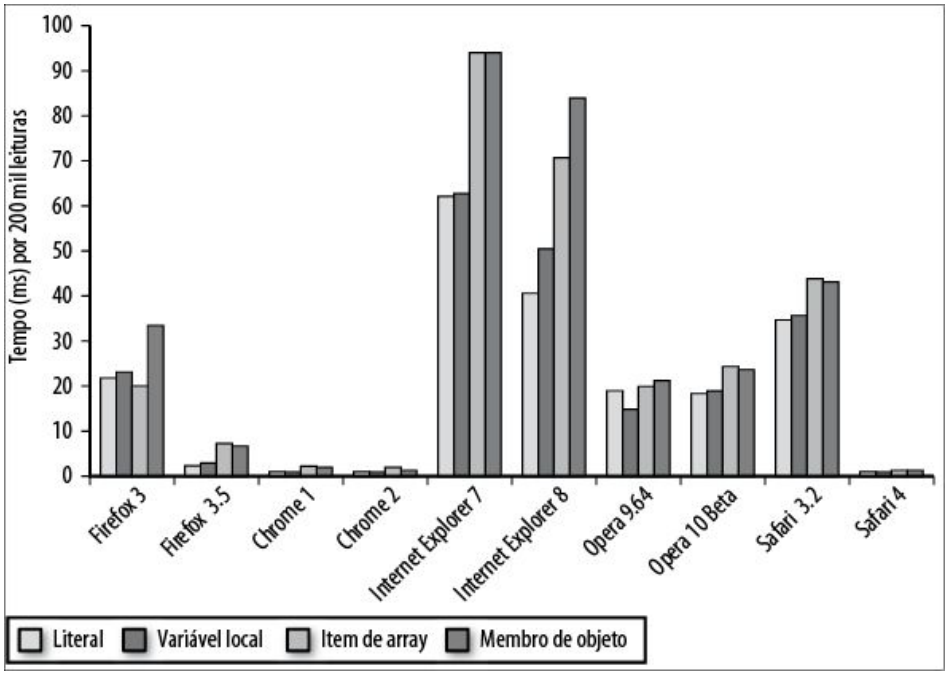


Figura 2.1 – Tempo por 200 mil leituras a partir de várias localizações de dados.

Navegadores mais antigos, que utilizavam engines JavaScript mais tradicionais, como o Firefox 3, o Internet Explorer e o Safari 3.2, necessitam de um intervalo de tempo muito maior para poder acessar valores se comparados a navegadores que usam engines JavaScript otimizadoras. A tendência geral, entretanto, permanece a mesma em todos eles: o acesso a valores literais e variáveis locais tende a ser mais rápido do que a itens de array e a membros de

objetos. A única exceção é o Firefox 3, que otimizava o acesso a itens de array, tornando-se muito mais rápido. Ainda assim, a recomendação geral é utilizar valores literais e variáveis locais sempre que possível e limitar o uso de itens de array e membros de objeto sempre que a velocidade da execução for importante. Para essa finalidade, existem vários padrões que podem ser almejados, evitados e otimizados em seu código.

Gerenciamento de escopo

O conceito de escopo é essencial para o entendimento da linguagem JavaScript, não apenas da perspectiva do desempenho, mas também a partir de uma perspectiva funcional. Ele produz efeitos que vão desde a determinação de quais variáveis uma função poderá acessar à designação do valor de `this`. Há também considerações importantes quanto ao desempenho quando falamos de escopos em JavaScript. Contudo, para entender qual a relação entre velocidade e escopo, é necessário saber exatamente de que forma o escopo funciona.

Cadeias de escopo e resolução do identificador

Cada função em JavaScript é representada como um objeto – mais especificamente como uma instância de função. Objetos de função têm propriedades da mesma maneira que qualquer outro objeto, e estas incluem tanto as propriedades que podem ser acessadas de modo programático quanto uma série de propriedades internas usadas pela engine do JavaScript, mas que não são acessíveis por meio do código. Uma dessas propriedades é `[[Scope]]`, conforme definida pela ECMA-262, Terceira Edição.

A propriedade interna `[[Scope]]` contém um acervo de objetos que representam o escopo dentro do qual essa função foi criada. Esse acervo é chamado de cadeia de escopo da função e determina os dados que uma função pode acessar. Cada objeto na cadeia de

escopo da função é chamado de objeto variável e contém entradas para variáveis na forma de pares chave-valor. Quando uma função é criada, sua cadeia de escopo é preenchida com objetos que representam os dados acessíveis no escopo dentro do qual a função foi criada. Por exemplo, considere a seguinte função global:

```
function add(num1, num2) {
  var sum = num1 + num2;
  return sum;
}
```

Quando a função add() é criada, sua cadeia de escopo é preenchida com um único objeto variável: o objeto global que representa todas as variáveis definidas globalmente. Esse objeto contém entradas para window, navigator e document, apenas para citar algumas. A figura 2.2 mostra esse relacionamento (observe que o objeto global da figura exibe apenas algumas variáveis globais como exemplo: existem muitas outras).

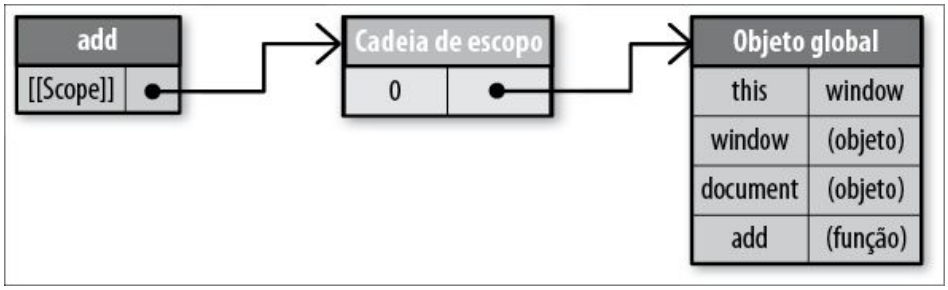


Figura 2.2 – Cadeia de escopo para a função add().

A cadeia de escopo para a função add é posteriormente utilizada quando a função é executada. Suponha que o seguinte código seja executado:

```
var total = add(5, 10);
```

A execução da função add dispara a criação de um objeto interno chamado de *contexto de execução*. Um contexto de execução define o ambiente no qual uma função está sendo executada. Cada contexto de execução é único a uma execução particular da função, de modo que chamadas múltiplas a uma mesma função resultam na criação de vários contextos de execução. O contexto de execução é destruído uma vez que a função tenha sido completamente

executada.

Um contexto de execução tem sua própria cadeia de escopo, que é utilizada para a resolução do identificador. Quando o contexto de execução é criado, sua cadeia de escopo é inicializada com os objetos contidos na propriedade [[Scope]] da função executada. Esses valores são copiados para a cadeia de escopo do contexto de execução na ordem em que aparecem na função. Uma vez que isso esteja feito, um novo objeto chamado *objeto de ativação* é criado para o contexto de execução. O objeto de ativação atua como o objeto variável para essa execução e contém entradas para variáveis locais, argumentos nomeados, a coleção arguments e this. O objeto é, a seguir, empurrado para frente da cadeia de escopo. Quando o contexto de execução é destruído, também é destruído o objeto de ativação. A figura 2.3 mostra o contexto de execução e sua cadeia de escopo para o código do exemplo prévio.

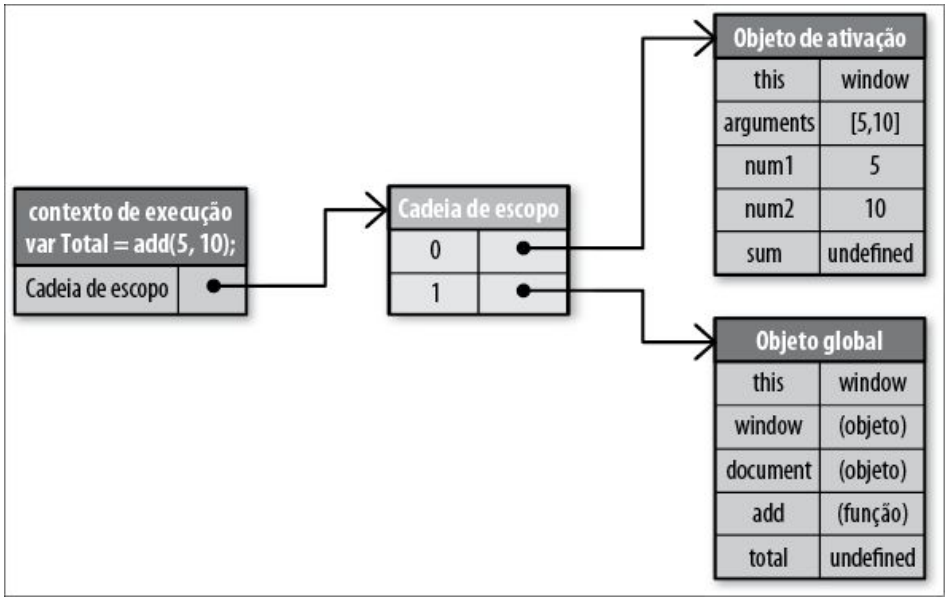


Figura 2.3 – Cadeia de escopo durante a execução de add().

Toda vez que uma variável for encontrada quando a função é executada, o processo de resolução do identificador ocorre para determinar onde devem ser acessados e armazenados os dados. Durante esse processo, a cadeia de escopo do contexto de execução é pesquisada em busca de um identificador com o mesmo

nome. A busca começa na frente da cadeia de escopo, no objeto de ativação da função de execução. Caso seja encontrada, a variável com o identificador especificado será utilizada; caso não seja, a busca continua para o próximo objeto da cadeia de escopo. Esse processo continua até que o identificador seja encontrado ou até que não existam mais objetos variáveis a pesquisar, caso em que o identificador será considerado indefinido. A mesma abordagem é utilizada para cada identificador encontrado durante a execução da função. Assim, no exemplo prévio, isso ocorreria para `sum`, `num1` e `num2`. É esse processo de busca que afeta o desempenho.


Observe que duas variáveis com o mesmo nome podem existir em partes diferentes da cadeia de escopo. Nesse caso, o identificador se atém à variável que encontra primeiro em sua travessia pela cadeia de escopo, e é comum dizermos dizer que a primeira variável encobre a segunda.

Desempenho da resolução do identificador

A resolução do identificador não é gratuita, uma vez que, na verdade, não existe operação computacional que ocorra sem algum tipo de consumo do desempenho. Quanto mais fundo na cadeia de escopo do contexto de execução estiver um identificador, mais lento será seu acesso tanto para leitura quanto para escrita. Dessa forma, variáveis locais serão sempre acessadas com maior velocidade dentro de uma função, enquanto variáveis globais serão geralmente as mais lentas (engines otimizadoras de JavaScript são capazes de contornar essa característica em algumas situações). Não se esqueça de que variáveis globais sempre existem no último objeto variável da cadeia de escopo do contexto de execução, assim, são sempre as mais distantes a serem resolvidas. As figuras 2.4 e 2.5 mostram a velocidade de resolução do identificador de acordo com sua profundidade na cadeia de escopo. Uma profundidade 1 indica uma variável local.

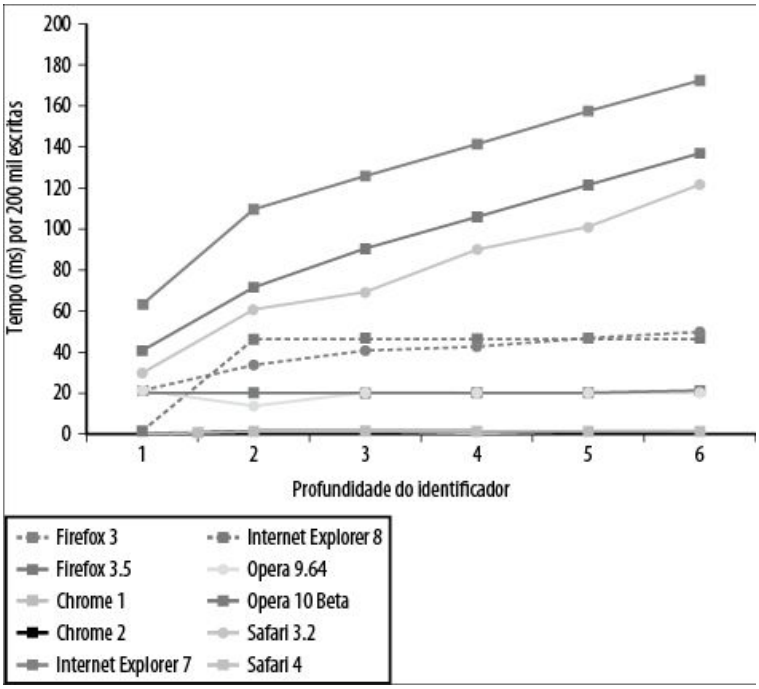


Figura 2.4 – Resolução do identificador para operações de escrita.

A tendência comum em todos os navegadores é a de que quanto mais fundo em uma cadeia de escopo existir um identificador, mais lenta será sua leitura ou escrita. Navegadores com engines otimizadoras de JavaScript, como o Chrome e o Safari 4, não sofrem essa espécie de penalidade de desempenho por acessarem identificadores fora do escopo, enquanto o Internet Explorer, o Safari 3.2 e outros mostram um efeito mais drástico. Vale a pena lembrar que, caso seus dados tivessem sido incluídos, navegadores mais antigos como Internet Explorer 6 e o Firefox 2 apresentariam elevações tão íngremes que nem apareceriam dentro dos limites máximos de nossos gráficos.

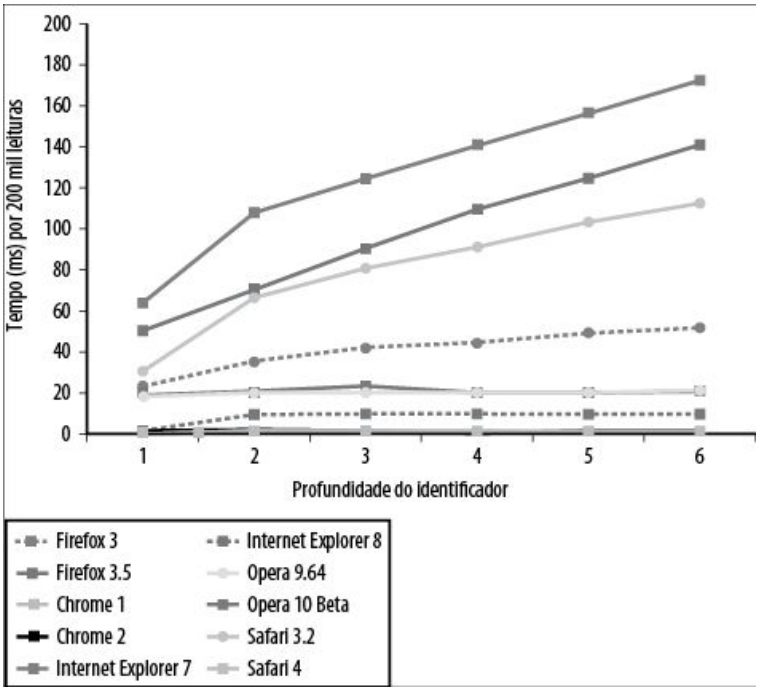


Figura 2.5 – Resolução do identificador para operações de leitura.

Dada essa informação, aconselhamos o uso de variáveis locais sempre que possível para alcançar um melhor desempenho em navegadores que não apresentam engines otimizadoras de JavaScript. Uma boa regra prática é a de sempre armazenar valores fora do escopo em variáveis locais, caso sejam utilizados mais do que uma vez dentro de uma função. Considere o seguinte exemplo:

```
function initUI() {
  var bd = document.body,
    links = document.getElementsByTagName("a"),
    i= 0,
    len = links.length;

  while(i < len) {
    update(links[i++]);
  }
  document.getElementById("go-btn").onclick = function() {
    start();
  };
  bd.className = "active";
}
```

Essa função apresenta três referências a document, um objeto global. A busca por essa variável deverá percorrer toda a cadeia de escopo para finalmente ser resolvida no objeto variável global. Podemos

mitigar o impacto sobre o desempenho que esses acessos repetidos à variável global teriam, armazenando a referência em uma variável local e utilizando justamente essa nova variável no lugar da global. Como exemplo, o código anterior pode ser reescrito da seguinte maneira:

```
function initUI() {
  var doc = document,
    bd = doc.body,
    links = doc.getElementsByTagName("a"),
    i= 0,
    len = links.length;

  while(i < len) {
    update(links[i++]);
  }

  doc.getElementById("go-btn").onclick = function() {
    start();
  };

  bd.className = "active";
}
```

A versão atualizada de initUI() primeiramente armazena uma referência a document na variável local doc. Em vez de acessar a variável global três vezes, esse número é reduzido a apenas um acesso. Acessar doc em lugar de document é mais rápido, pois se trata de uma variável local. Certamente essa função simplificada não produzirá um enorme ganho de desempenho, já que não está realizando muito, mas imagine o caso de funções maiores, com dúzias de variáveis globais sendo acessadas repetidamente. É nesses casos que as mais impressionantes melhoras de desempenho serão encontradas.

Acréscimo na cadeia de escopo

Em termos gerais, a cadeia de escopo de um contexto de execução não se altera. Há, entretanto, duas instruções que temporariamente acrescem à cadeia de escopo do contexto de execução enquanto ela é executada. A primeira delas é with.

A instrução with é utilizada na criação de variáveis para todas as

propriedades de um objeto. Isso se compara a outras linguagens com atributos similares e é normalmente visto como uma conveniência que busca evitar a escrita repetitiva do mesmo código. A função initUI() pode ser escrita da seguinte maneira:

```
function initUI() {
  with (document) { // evite!
    var bd = body,
      links = getElementsByTagName("a"),
      i= 0,
      len = links.length;
    while(i < len) {
      update(links[i++]);
  }
    getElementById("go-btn").onclick = function() {
      start();
  };
    bd.className = "active";
  }
}
```

Essa versão reescrita do initUI() utiliza uma instrução para que não tenhamos de repetir document em outros locais. Ainda que possa parecer mais eficiente, isso na verdade cria um problema de desempenho.

Quando a execução do código flui dentro de uma instrução with, a cadeia de escopo do contexto da execução é temporariamente acrescida. Um novo objeto variável é criado contendo todas as propriedades do objeto especificado. Esse objeto, então, é empurrado para frente da cadeia de escopo, o que significa que todas as variáveis locais da função estão agora no segundo objeto da cadeia de escopo, representando um acesso mais custoso (veja a figura 2.6).

Ao passar o objeto document dentro da instrução with, um novo objeto variável contendo todas as propriedades de objeto de document é empurrado para frente da cadeia de escopo. Isso torna muito rápido o acesso às propriedades de document, mas desacelera o acesso a variáveis locais, como bd. Por essa razão, é preferível evitar a

utilização da instrução with. Como já mostramos, é igualmente simples armazenar document em uma variável local e obter a melhora de desempenho dessa forma.

A instrução with não é a única parte do JavaScript que provoca um acréscimo atrificial à cadeia de escopo do contexto de execução: a cláusula catch da instrução try-catch tem também o mesmo efeito. Quando um erro ocorre no bloco try, a execução automaticamente flui para catch e o objeto de exceção é empurrado para dentro de um objeto variável que é, por sua vez, colocado na frente da cadeia de escopo. Dentro de cada bloco catch, todas as variáveis locais à função estão agora em segundo lugar na cadeia de escopo. Por exemplo:

```
try {
  methodThatMightCauseAnError();
} catch (ex) {
  alert(ex.message); // a cadeia de escopo é acrescida aqui
}
```

Observe que assim que a cláusula catch concluir sua execução, a cadeia de escopo retornará para seu estado prévio.

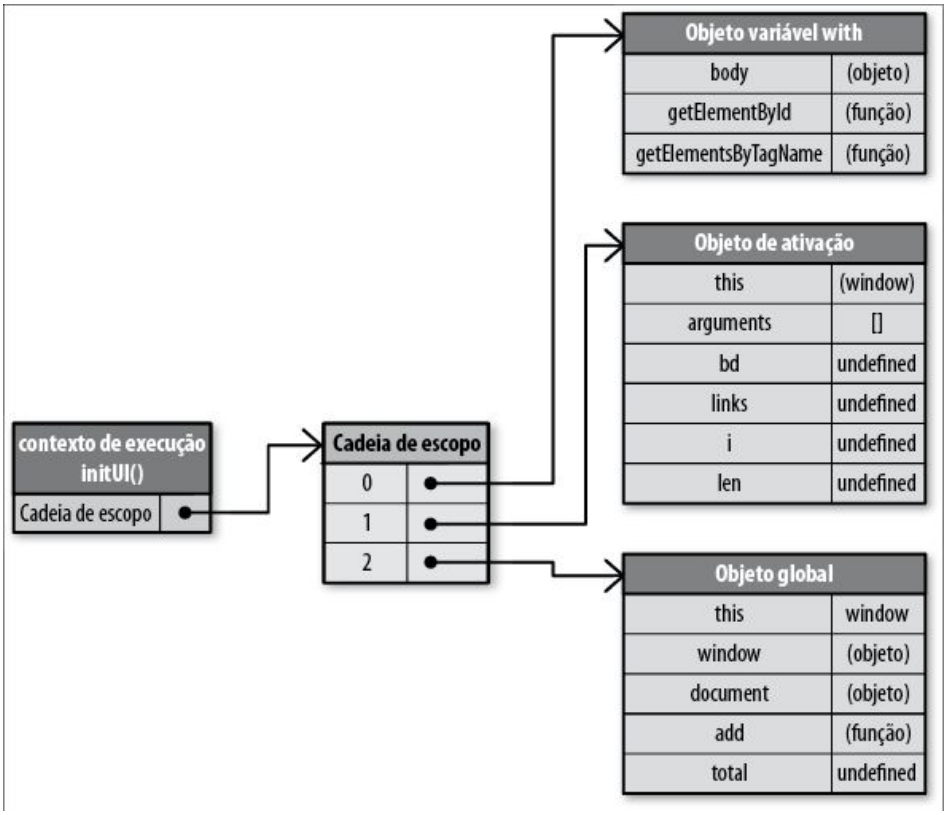


Figura 2.6 – Cadeia de escopo acrescida com uma instrução with.

A instrução try-catch é muito útil quando aplicada apropriadamente, e por isso não faz sentido recomendar que seja sempre evitada. Caso pretenda usar uma try-catch, certifique-se de compreender a probabilidade de erro. Uma try-catch nunca deve ser utilizada como a solução para um erro JavaScript. Caso você saiba que um erro vai ocorrer com frequência, há um problema com o código em si e ele deverá ser corrigido.

É possível minimizar o impacto sobre o desempenho causado pela cláusula catch executando apenas o mínimo de código necessário dentro dela. Um bom padrão é a adoção de um método para o tratamento (handling) de erros delegados pela cláusula catch. Isso pode ser feito desta maneira:

```
try {
  methodThatMightCauseAnError();
} catch (ex) {
  handleError(ex); // delega ao método manipulador
}
```

Aqui um método handleError() é o único código executado na cláusula catch. Esse método está livre para tratar o erro do modo apropriado e recebe o objeto de exceção gerado a partir do erro. Uma vez que há apenas uma única instrução em execução e nenhuma variável local está sendo acessada, o acréscimo temporário à cadeia de escopo não afeta o desempenho do código.

Escopos dinâmicos

Tanto a instrução with quanto a cláusula catch de uma instrução try-catch, assim como uma função que contenha eval(), são consideradas escopos dinâmicos. Um escopo dinâmico existe apenas pela execução do código e, portanto, não pode ser determinado simplesmente por análise estática (observando a estrutura do código). Por exemplo:

```
function execute(code) {
  eval(code);
  function subroutine() {
    return window;
  }
  var w = subroutine(); // qual o valor de w?
};
```

A função execute() representa um escopo dinâmico devido ao uso de eval(). O valor de w pode ser alterado com base no valor do código. Na maioria dos casos, w será igual ao objeto window global, mas considere o seguinte:

```
execute("var window = {};")
```

Nesse caso, eval() cria uma variável window local em execute(), assim w acaba sendo igual ao window local em vez do global. Não há como saber se esse é o caso até que o código seja executado, o que significa que o valor do identificador window não pode ser predeterminado.

Engines otimizadoras de JavaScript como a Nitro, do Safari, tentam acelerar a resolução do identificador analisando o código para determinar quais variáveis devem estar acessíveis em um dado

momento. Essas engines buscam evitar a busca tradicional pela cadeia de escopo indexando identificadores para uma resolução mais rápida. Entretanto, quando um escopo dinâmico está envolvido, essa otimização já não é mais válida. As engines têm de recorrer a uma abordagem com base em hash para a resolução de identificadores que mais se equipara a uma consulta tradicional à cadeia de escopo.

Por esse motivo, é recomendado utilizar escopos dinâmicos apenas quando absolutamente necessário.

Closures, escopo e memória

Closures[1] representam um dos aspectos mais poderosos do JavaScript, permitindo que uma função acesse dados externos a seu escopo local. A utilização de closures foi popularizada pelos textos de Douglas Crockford e encontra-se presente na maioria das aplicações web complexas. Há, entretanto, um impacto sobre o desempenho associado à utilização de closures.

Para compreender as questões de desempenho envolvidas nas closures, considere o seguinte:

```
function assignEvents() {
  var id = "xdi9592";
  document.getElementById("save-btn").onclick = function(event) {
    saveDocument(id);
  };
}
```

A função `assignEvents()` designa um manipulador de evento a um único elemento DOM. Esse manipulador é uma closure, uma vez que é criado quando `assignEvents()` é executada e pode acessar a variável `id` do escopo que a contém. Para que essa closure acesse `id`, uma cadeia de escopo específica deve ser criada.

Quando `assignEvents()` é executada, cria-se um objeto de ativação que contém, entre outras coisas, a variável `id`. Esse se torna o primeiro objeto na cadeia de escopo do contexto de execução, com o objeto global vindo em segundo lugar. Quando a closure é criada, sua

propriedade [[Scope]] é inicializada com ambos os objetos (veja a figura 2.7).

Uma vez que a propriedade [[Scope]] da closure contém referências aos mesmos objetos que a cadeia de escopo do contexto de execução, há um efeito colateral. Geralmente, o objeto de ativação de uma função é destruído quando o contexto de execução o é. Entretanto, quando há uma closure envolvida, o objeto de ativação não é destruído, já que uma referência ainda existe na propriedade [[Scope]] da closure. Isso significa que closures representam um custo maior de memória para o script. Em aplicações web maiores, isso pode se tornar um problema, especialmente quando se trata do Internet Explorer. O IE implementa objetos DOM como objetos JavaScript não-nativos e, dessa forma, closures podem provocar memory leaks[2] (o capítulo 3 contém mais informações).

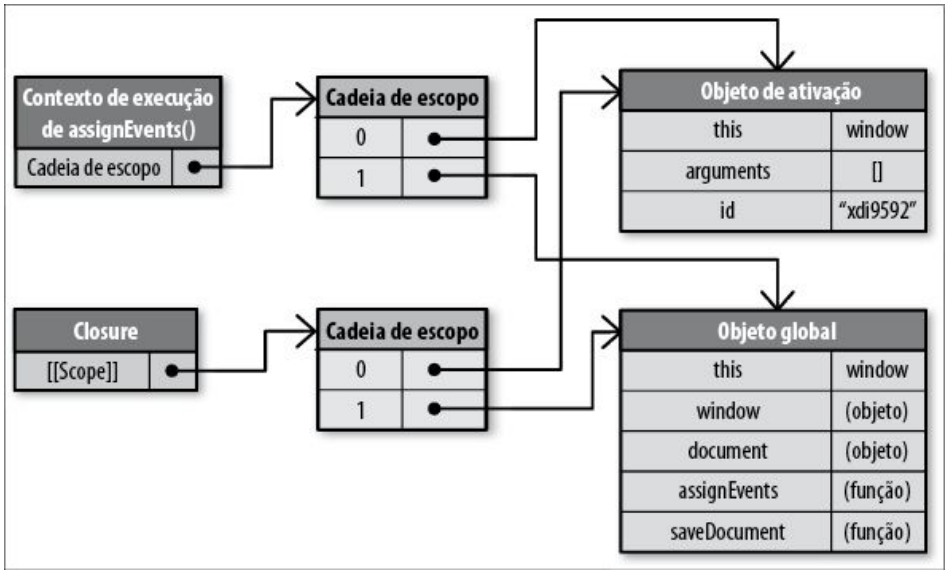


Figura 2.7 – Cadeias de escopo do contexto de execução e da closure de assignEvents().

Quando a closure é executada, um contexto de execução é criado, cuja cadeia de escopo é inicializada com os mesmos dois objetos da cadeia de escopo referidos em [[Scope]], seguido pela criação de um novo objeto de ativação para a closure em si (veja a figura 2.8).

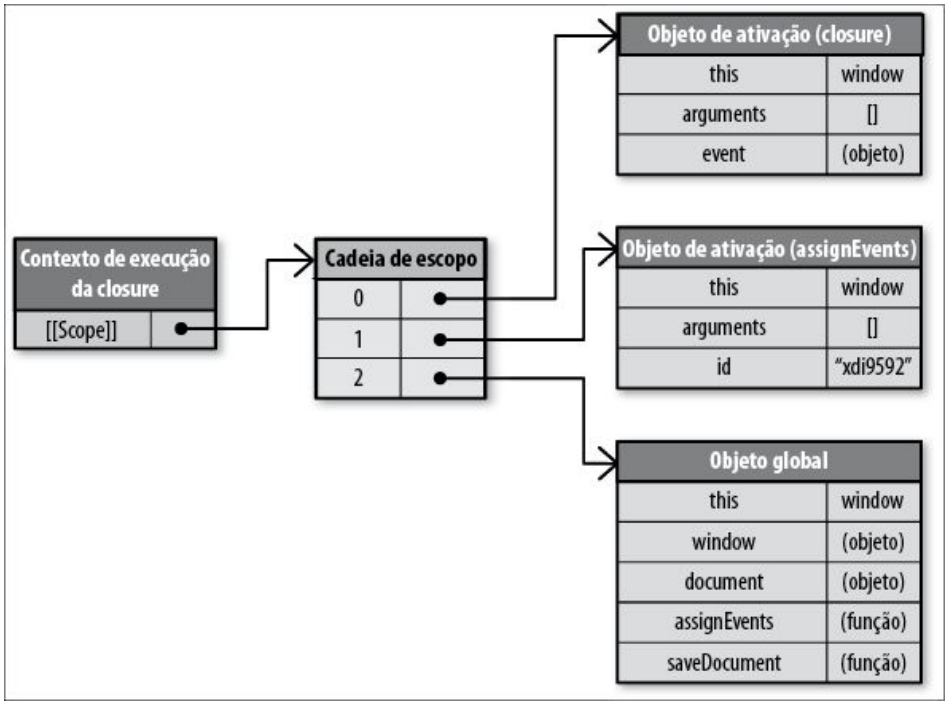


Figura 2.8 – Execução da closure.

Observe que os dois identificadores utilizados na closure, id e saveDocument, existem depois do primeiro objeto na cadeia de escopo. Essa é a principal preocupação quanto ao desempenho no que se refere às closures: você muitas vezes estará acessando muitos identificadores fora do escopo e, justamente por isso, causará penalidades sobre o desempenho a cada um desses acessos.

Por trazerem considerações quanto à memória, bem como quanto à velocidade, é aconselhável ter cautela na utilização de closures em seus scripts. Ainda assim, você pode mitigar o impacto sobre a velocidade da execução seguindo o conselho dado anteriormente neste capítulo quanto a variáveis fora do escopo: armazene em variáveis locais todas as variáveis desse tipo utilizadas com frequência e, então, acesse-as diretamente.

Membros de objeto

A maior parte do código JavaScript é escrita por orientação a objetos, seja pela criação de objetos personalizados ou pelo uso de objetos inclusos como os do Document Object Model (DOM) e do Browser Object Model (BOM). Assim, é comum que haja muito

acesso a membros de objeto.

Membros de objeto são tanto propriedades quanto métodos, e há pouca diferença entre ambos em JavaScript. Um membro nomeado de um objeto pode conter qualquer tipo de dados. Uma vez que as funções são representadas como objetos, um membro pode conter uma função além de tipos de dados mais tradicionais. Quando um membro nomeado faz referência a uma função, ele é considerado um método, enquanto um membro que faça referência a um tipo de dado que não seja uma função é considerado uma propriedade.

Conforme discutido anteriormente neste capítulo, o acesso a membros de objetos tende a ser mais lento do que o acesso a dados em literais ou variáveis e, em alguns navegadores, mais lento também do que o acesso a itens de array. Para compreender a razão disso, é necessário entender a natureza dos objetos em JavaScript.

Protótipos

Objetos em JavaScript tem como base *protótipos*. Um protótipo é um objeto que serve como base para outro, definindo e implementando membros que um novo objeto deve ter. Esse é um conceito completamente diferente do conceito tradicional de classes da programação orientada a objetos, que define o processo para a criação de um novo objeto. Objetos protótipos são compartilhados por todas as instâncias de um dado tipo de objeto, de modo que todas elas também compartilhem seus membros.

Um objeto é preso a seu protótipo por uma propriedade interna. O Firefox, o Safari e o Chrome expõem essa propriedade aos desenvolvedores como `__proto__`: outros navegadores não permitem o acesso por scripts a essa propriedade. Sempre que criar uma nova instância de tipo incluso, como `Object` ou `Array`, essas instâncias automaticamente terão uma instância de `Object` como seu protótipo.

Consequentemente, objetos podem ter dois tipos de membros: membros de instância (também chamados de membros “próprios”) e

membros de protótipo. Membros de instância existem diretamente na instância do objeto em si, enquanto membros de protótipo são herdados do protótipo do objeto. Considere o seguinte exemplo:

```
var book = {
  title: "High Performance JavaScript",
  publisher: "Yahoo! Press"
};
alert(book.toString()); // "[object Object]"
```

Nesse código, o objeto book tem dois membros de instância: título (title) e editora (publisher). Observe que não há definição para o método toString(), mas que o método é chamado e se comporta apropriadamente sem devolver um erro. O método toString() é um membro protótipo que o objeto book está herdando. A figura 2.9 mostra esse relacionamento.

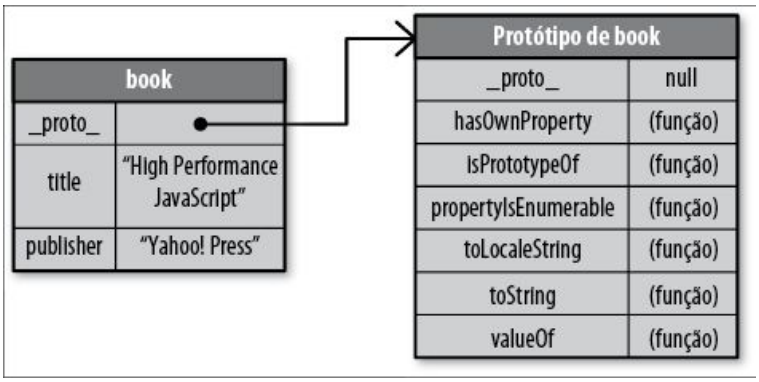


Figura 2.9 – Relacionamento entre uma instância e um protótipo.

O processo da resolução de um membro de objeto é muito semelhante à resolução de uma variável. Quando book.toString() é chamado, a busca por um membro chamado "toString" tem início na instância do objeto. Uma vez que book não tem um membro chamado toString, a busca flui para o objeto protótipo, onde o método toString() é encontrado e executado. Dessa maneira, book tem acesso a todas as propriedades ou métodos de seu protótipo.

Você pode determinar se um objeto tem um membro de instância com um dado nome utilizando o método hasOwnProperty() e passando a ele o nome do membro. Para determinar se um objeto tem acesso a uma propriedade com um dado nome, você pode utilizar o operador

in. Por exemplo:

```
var book = {
  title: "High Performance JavaScript",
  publisher: "Yahoo! Press"
};
alert(book.hasOwnProperty("title")); // verdadeiro
alert(book.hasOwnProperty("toString")); // falso
alert("title" in book); // verdadeiro
alert("toString" in book); // verdadeiro
```

Nesse código, hasOwnProperty() retorna verdadeiro quando “title” é passado, já que title é uma instância do objeto. O método retorna false quando “toString” é passado, já que não existe na instância. Quando cada nome de propriedade é utilizado com o operador in, o resultado é verdadeiro nas duas ocasiões, pois a busca é feita na instância e no protótipo.

Cadeias de protótipos

O protótipo de um objeto determina o tipo ou os tipos dos quais ele é uma instância. Por padrão, todos os objetos são instância de Object e herdam todos os métodos básicos, como toString(). Você pode criar um protótipo de outro tipo definindo e utilizando um construtor. Por exemplo:

```
function Book(title, publisher) {
  this.title = title;
  this.publisher = publisher;
}
Book.prototype.sayTitle = function() {
  alert(this.title);
};
var book1 = new Book("High Performance JavaScript", "Yahoo! Press");
var book2 = new Book("JavaScript: The Good Parts", "Yahoo! Press");
alert(book1 instanceof Book); // verdadeiro
alert(book1 instanceof Object); // verdadeiro
book1.sayTitle(); // "High Performance JavaScript"
alert(book1.toString()); // "[object Object]"
```

O construtor Book é utilizado para criar uma nova instância de Book. O protótipo da instância book1 (__proto__) é Book.prototype, e o protótipo de

Book.prototype é Object. Isso cria uma cadeia de protótipo da qual tanto book1 quanto book2 herdam seus membros. A figura 2.10 mostra esse relacionamento.

Observe que ambas as instâncias de Book compartilham a mesma cadeia de protótipos. Cada instância tem suas próprias propriedades title e publisher, mas todo o resto é herdado por meio de protótipos.

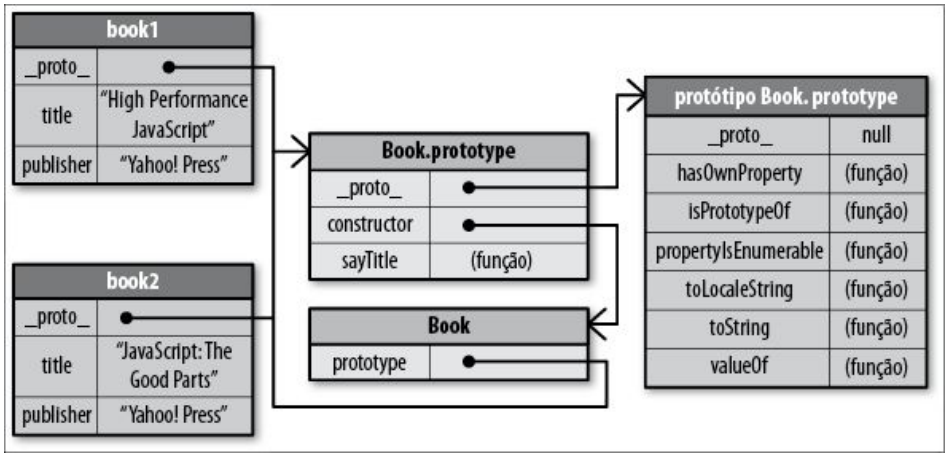


Figura 2.10 – Cadeias de protótipos.

Agora quando book1.toString() é chamada, a busca deve ir mais fundo na cadeia de protótipos para resolver o objeto membro “toString”. Como é de se esperar, quanto mais fundo na cadeia de protótipos um membro existir, mais lenta será sua recuperação. A figura 2.11 mostra o relacionamento entre a profundidade do membro no protótipo e o tempo que leva o acesso a ele.

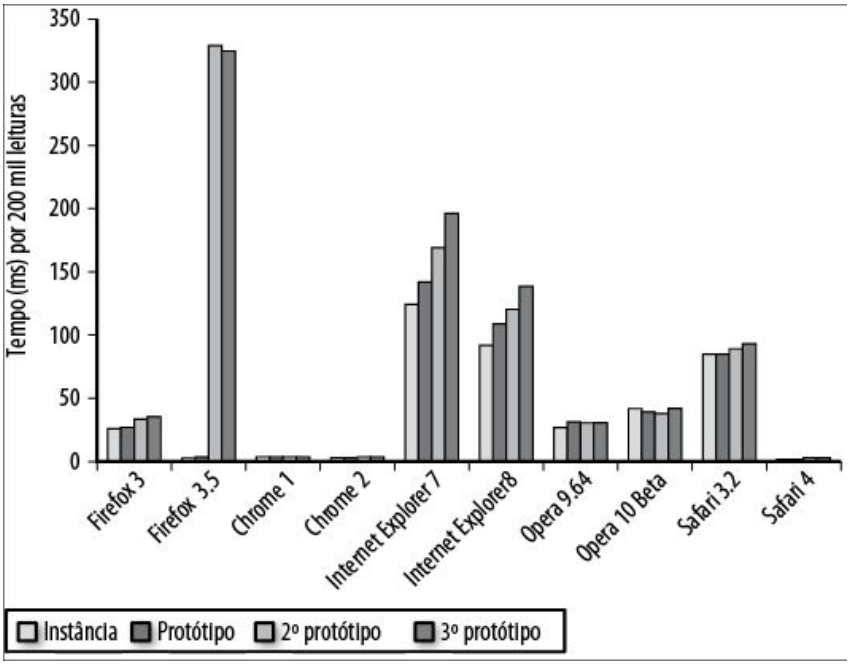


Figura 2.11 – Acesso a dados indo mais fundo na cadeia de protótipos.

Ainda que navegadores mais modernos, com engines otimizadoras de JavaScript, realizem bem essa tarefa, navegadores mais antigos – especialmente o Internet Explorer e o Firefox 3.5 – incorrem em penalidades sobre o desempenho a cada passo adicional dado na cadeia de protótipos. Tenha em mente que o processo de procura por um membro de instância ainda é mais dispendioso do que o acesso aos dados de uma variável literal ou local, de modo que sua adição à cadeia de protótipos apenas amplifica esse efeito.

Membros aninhados

Uma vez que membros de objetos podem conter outros membros, não é raro observar padrões como window.location.href em códigos JavaScript. Esses membros aninhados fazem com que a engine JavaScript tenha de passar pelo processo de resolução do membro objeto sempre que um ponto for encontrado. A figura 2.12 mostra o relacionamento entre a profundidade do membro do objeto e o tempo de acesso.

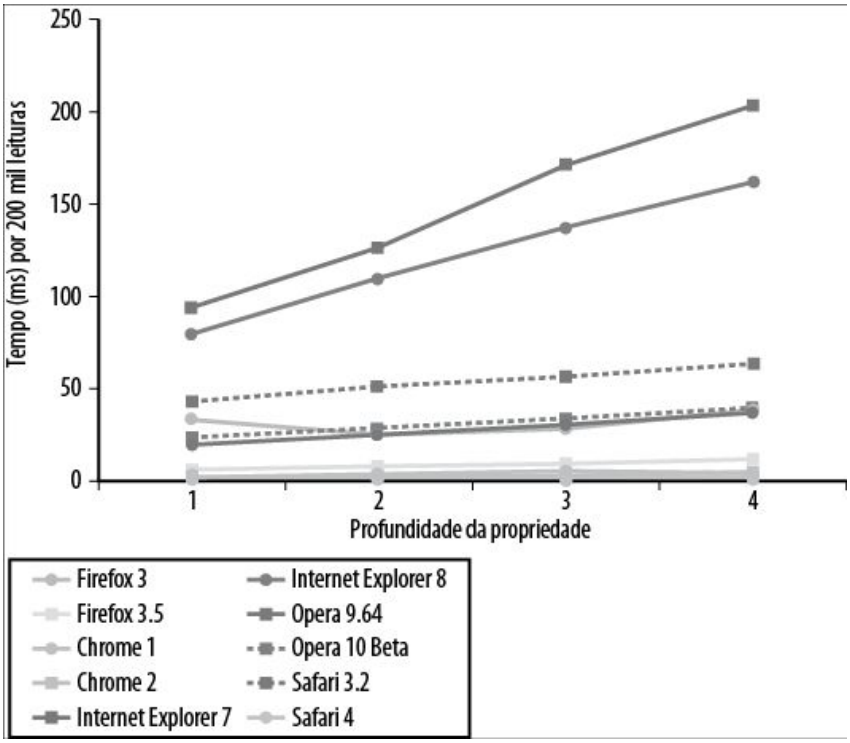


Figura 2.12 – Tempo de acesso em relação à profundidade da propriedade.

Não deve ser nenhuma surpresa que quando mais fundo estiver o membro aninhado mais lento será o acesso aos dados. A avaliação de location.href é sempre mais rápida que a de window.location.href, que, por sua vez, é mais rápida que a de window.location.href.toString(). Caso essas propriedades não estejam nas instâncias do objeto, a resolução dos membros irá demorar mais conforme a cadeia de protótipos é buscada a cada ponto.

Na maioria dos navegadores, não há diferença discernível entre o acesso a um membro de objeto utilizando a notação com pontos (objeto.nome) versus a notação com colchetes (objeto["nome"]). O Safari é o único navegador em que a notação com pontos é sempre mais rápida, mas não o bastante para justificar que não se utilize a notação com colchetes.

Armazenamento em cache de valores de membros de objetos

Com todos os problemas de desempenho que se relacionam a membros de objetos, é fácil acreditar que eles devem ser evitados sempre que possível. Para sermos mais precisos, é recomendado

utilizar membros de objetos apenas quando necessário. Por exemplo, não há razão para lermos o valor de um membro de objeto mais do que uma vez em uma única função:

```
function hasEitherClass(element, className1, className2) {
  return element.className == className1 || element.className == className2;
}
```

Nesse código, `element.className` é acessado duas vezes. Fica claro que esse valor não sofrerá alterações durante o curso da função: ainda assim, consultas por membros do objeto são realizadas. Podemos eliminar uma delas armazenando o valor em uma variável local e utilizando essa variável em lugar da consulta:

```
function hasEitherClass(element, className1, className2) {
  var currentClassName = element.className;
  return currentClassName == className1 || currentClassName == className2;
}
```

Essa versão reescrita da função limita a uma única vez o número de consulta aos membros. Uma vez que ambas as consultas estavam lendo o valor da propriedade, faz sentido armazenarmos esse dado em uma variável local. Essa variável local, por sua vez, será acessada muito mais rapidamente.

Em termos gerais, caso você tenha de ler a propriedade de um objeto mais de uma vez em uma função, será melhor armazenar seu valor em uma variável local. Essa variável local poderá então ser utilizada no lugar da propriedade para evitar prejuízos ao desempenho advindos de novas consultas. Isso é muito importante, especialmente quando se trata de membros de objetos aninhados que têm um efeito mais drástico sobre a velocidade de execução.

A criação de namespaces em JavaScript, como a técnica utilizada na biblioteca YUI, é uma fonte de propriedades aninhadas acessadas com frequência. Por exemplo:

```
function toggle(element) {
  if (YAHOO.util.Dom.hasClass(element, "selected")) {
    YAHOO.util.Dom.removeClass(element, "selected");
    return false;
  } else {
    YAHOO.util.Dom.addClass(element, "selected");
```

```
    return true;
  }
}
```

Esse código repete YAHOO.util.Dom três vezes para acessar três métodos diferentes. Para cada método, há três consultas de membros, em um total de nove, tornando essa implementação muito ineficiente. Uma melhor abordagem seria armazenar YAHOO.util.Dom em uma variável local para depois acessá-la:

```
function toggle(element) {
  var Dom = YAHOO.util.Dom;
  if (Dom.hasClass(element, "selected")) {
    Dom.removeClass(element, "selected");
    return false;
  } else {
    Dom.addClass(element, "selected");
    return true;
  }
}
```

O número total de consultas por membros nesse código foi reduzido de nove para cinco. Nunca se deve buscar um membro de objeto mais de uma vez em uma única função, a menos que seu valor tenha sido alterado.

Entretanto, tenha cuidado: não é recomendável usar essa técnica para métodos de objetos. Muitos métodos de objetos utilizam this para determinar o contexto no qual estão sendo chamados, fazendo com que o armazenamento de um método em uma variável local prenda this a window. A alteração do valor de this leva a erros programáticos, uma vez que a engine JavaScript não será capaz de resolver os membros de objeto apropriados dos quais ela pode depender.

Resumo

O local em que você armazena e acessa dados em JavaScript pode ter um impacto mensurável sobre o desempenho geral de seu código. Há quatro locais a partir dos quais dados podem ser acessados: valores literais, variáveis, itens de array e membros de objetos. Todos apresentam considerações de desempenho diferentes.

- Valores literais e variáveis locais podem ser acessados muito

rapidamente, enquanto itens de array e membros de objetos demoram mais.

- Variáveis locais são mais rápidas para serem acessadas do que variáveis fora do escopo, uma vez que existem no primeiro objeto variável da cadeia de escopo. Quanto mais distante na cadeia de escopo uma variável se encontrar, mais tempo levará seu acesso. Variáveis globais são sempre as de mais lento acesso, já que se encontram sempre na última cadeia de escopo.
- Evite a instrução with, pois ela acresce à cadeia de escopo do contexto de execução. Da mesma forma, tenha cuidado com a cláusula catch de uma instrução try-catch, visto que tem o mesmo efeito.
- Membros de objeto aninhados representam um impacto significativo sobre o desempenho e devem, por isso, ser minimizados.
- Quando mais fundo na cadeia de protótipo uma propriedade ou método existir, mais lento será seu acesso.
- Em termos gerais, é possível melhorar o desempenho de seu código JavaScript armazenando em variáveis locais os membros de objeto, itens de array e variáveis fora de escopo utilizados com frequência.

Ao adotar essas estratégias, você pode melhorar muito o desempenho observado de uma aplicação web que demanda grande quantidade de código JavaScript.

---

1 N.T.: Uma função que referencia variáveis livres no contexto léxico. Uma closure ocorre normalmente quando uma função é declarada dentro do corpo de outra, e a função interior referencia variáveis locais da função exterior.(Fonte: Wikipédia)

2 N.T.: Memory leak é um tipo específico de consumo de memória em que o programa é incapaz de liberar a memória que adquire. (Fonte: Wikipédia)

CAPÍTULO 3

Criação de scripts DOM

Stoyan Stefanov

A criação de scripts DOM é uma tarefa dispendiosa que representa um gargalo de desempenho comum em aplicações web ricas. Este capítulo discute as áreas da criação de scripts DOM que podem ter um efeito negativo sobre a responsividade[1] (responsiveness) de uma aplicação e oferece recomendações acerca de como podemos melhorar o tempo de resposta. As três categorias dos problemas discutidos no capítulo incluem:

- Acesso e modificação dos elementos DOM.
- Modificação dos estilos de elementos DOM e a execução de repaints e reflows.
- Manipulação da interação do usuário por meio de eventos DOM.

Antes de mais nada – o que é o DOM e por que ele é lento?

DOM no mundo dos navegadores

O Document Object Model (DOM) é uma interface de aplicação (API) independente de linguagem para trabalho com documentos XML e HTML. No navegador, você trabalha principalmente com documentos HTML, ainda que não seja raro que aplicações web recuperem documentos XML e utilizem as APIs DOM para acessar dados desses documentos.

Ainda que o DOM seja uma API independente de linguagem, sua interface, no navegador, é implementada em JavaScript. Visto que a maior parte do trabalho na criação de scripts no lado cliente está relacionada ao documento subjacente, o DOM é uma parte importante da codificação JavaScript diária.

É comum nos navegadores que as implementações DOM e JavaScript sejam mantidas independentes. No Internet Explorer, por exemplo, a implementação JavaScript é chamada JScript e está localizada em um arquivo de biblioteca chamado *jscript.dll*, enquanto a implementação DOM encontra-se em outra biblioteca, a *mshtml.dll* (internamente chamada de Trident). Essa separação permite que outras tecnologias e linguagens, como VBScript, se beneficiem do DOM e da funcionalidade de renderização que a Trident oferece. O Safari utiliza o WebCore do WebKit para DOM e renderização e apresenta uma engine JavaScriptCore separada (chamada SquirrelFish em suas versões mais recentes). O Google Chrome também usa bibliotecas WebCore do WebKit para renderização de páginas, mas implementa sua própria engine JavaScript chamada V8. No Firefox, o SpiderMonkey (TraceMonkey na última versão) é a implementação JavaScript, uma parte separada da engine Gecko de renderização.

Inerentemente lento

O que isso significa em termos de desempenho? O simples fato de haver dois pedaços separados de funcionalidade interagindo entre si já traz consigo um custo. Uma analogia excelente é visualizarmos o DOM como uma área de terra e o JavaScript (significando ECMAScript) como outra, ambas conectadas por uma ponte com um pedágio (consulte John Hrvatin, Microsoft, MIX09, *http://videos.visitmix.com/MIX09/T53F*). Toda vez que seu ECMAScript tem de acessar o DOM, é preciso cruzar essa ponte e pagar a taxa de pedágio sobre o desempenho. Quando mais você trabalhar com o DOM, mais terá de pagar. Assim, a recomendação geral é a de que você cruze a ponte o mínimo necessário, buscando sempre permanecer na terra do ECMAScript. O restante do capítulo enfoca o que isso significa exatamente e o que podemos fazer para tornar as interações do usuário mais rápidas.

Acesso e modificação do DOM

O simples acesso a um elemento DOM envolve um preço – o "pedágio" discutido anteriormente. Por sua vez, a modificação de elementos é ainda mais cara, já que muitas vezes significa que o navegador terá de recalcular alterações à geometria da página.

Naturalmente, o pior caso de acesso e modificação de elementos ocorre quando isso é feito em loops, especialmente em loops sobre coleções HTML.

Apenas para dar uma ideia da escala dos problemas relacionados à criação de scripts em DOM, considere este simples exemplo:

```
function innerHTMLLoop() {
  for (var count = 0; count < 15000; count++) {
    document.getElementById('here').innerHTML += 'a';
  }
}
```

Essa é uma função que atualiza o conteúdo de um elemento da página em um loop. O problema com esse código é que, para cada iteração do loop, o elemento é acessado duas vezes: uma para leitura do valor da propriedade `innerHTML` e outra para sua escrita.

Uma versão mais eficiente dessa função utilizaria uma variável local para armazenar o conteúdo atualizado, escrevendo o valor apenas uma vez no final do loop:

```
function innerHTMLLoop2() {
  var content = '';
  for (var count = 0; count < 15000; count++) {
    content += 'a';
  }
  document.getElementById('here').innerHTML += content;
}
```

Essa nova versão da função rodará muito mais rápido em todos os navegadores. A figura 3.1 mostra os resultados da medição da melhora de desempenho nos diferentes navegadores. O eixo y da figura (assim como em todas as figuras deste livro) mostra a melhora no tempo de execução, por exemplo, quão mais rápida é a utilização de uma abordagem em relação à outra. Nesse caso, por exemplo, o uso de `innerHTMLLoop2()` é 155 vezes mais rápido do que o

de innerHTMLLoop() no IE6.

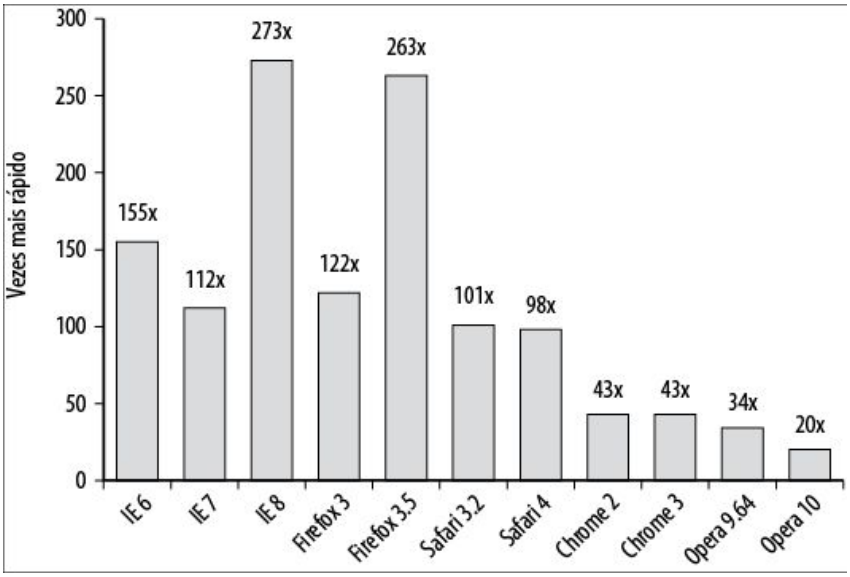


Figura 3.1 – Um benefício em se ater ao ECMAScript: innerHTMLLoop2() é centenas de vezes mais rápida do que innerHTMLLoop().

Como esses resultados claramente evidenciam, quanto mais vezes o DOM for acessado, mais lenta será a execução de seu código. Dessa forma, a regra prática geral é esta: utilize o DOM apenas raramente e permaneça dentro do ECMASCript sempre que possível.

innerHTML versus métodos DOM

Com o passar dos anos houve muitas discussões na comunidade de desenvolvimento web sobre esta questão: é melhor usar a propriedade innerHTML, não padronizada mas bem aceita, para atualizar uma seção da página, ou é preferível utilizar apenas os métodos DOM puros, como o document.createElement()? Deixando de lado a questão dos padrões da web, isso afeta de alguma maneira o desempenho? A resposta é: cada vez menos, mas, mesmo assim, o innerHTML ainda é mais rápido em todos os navegadores com exceção dos que têm como fundamento as últimas versões do WebKit (caso do Chrome e do Safari).

Vamos examinar a simples tarefa da criação de uma tabela de mil linhas de dois modos diferentes:

- Concatenando uma string HTML e atualizando o DOM com o innerHTML.
- Utilizando apenas métodos DOM padronizados, como o document.createElement() e o document.createTextNode().

Nossa tabela de exemplo tem um conteúdo semelhante ao que viria de um Content Management System (CMS). O resultado final é mostrado na figura 3.2.

id	yes?	name	url	action
1	And the answer is... yes	my name is #1	http://example.org/1.html	• edit • delete
2	And the answer is... no	my name is #2	http://example.org/2.html	• edit • delete

Figura 3.2 – Resultado final da geração de uma tabela HTML com mil linhas e cinco colunas.

O código para geração da tabela com innerHTML é o seguinte:

```
function tableInnerHTML() {
  var i, h = ['<table border="1" width="100%">'];
  h.push('<thead>');
h.push('<tr><th>id<\/th><th>yes?<\/th><th>name<\/th><th>url<\/th><th>action<\/th>
<\/tr>');
  h.push('<\/thead>');
  h.push('<tbody>');
  for (i = 1; i <= 1000; i++) {
    h.push('<tr><td>');
    h.push(i);
    h.push('<\/td><td>');
    h.push('And the answer is... ' + (i % 2 ? 'yes' : 'no'));
    h.push('<\/td><td>');
    h.push('my name is #' + i);
    h.push('<\/td><td>');
    h.push('<a href="http://example.org/' + i + '.html">http://example.org/'+ i + '.html<\/a>');
    h.push('<\/td><td>');
    h.push('<ul>');
    h.push(' <li><a href="edit.php?id=' + i + '">edit<\/a><\/li>');
    h.push(' <li><a href="delete.php?id=' + i + '">delete<\/a><\/li>');
    h.push('<\/ul>');
    h.push('<\/td>');
```

```
    h.push('<\/tr>');
  }
  h.push('<\/tbody>');
  h.push('<\/table>');
  document.getElementById('here').innerHTML = h.join('');
};
```

Para gerar a mesma tabela apenas com elementos DOM, o código seria um pouco maior:

```
function tableDOM() {
  var i, table, thead, tbody, tr, th, td, a, ul, li;
  tbody = document.createElement('tbody');
  for (i = 1; i <= 1000; i++) {
    tr = document.createElement('tr');
    td = document.createElement('td');
    td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
    tr.appendChild(td);
    td = document.createElement('td');
    td.appendChild(document.createTextNode(i));
    tr.appendChild(td);
    td = document.createElement('td');
    td.appendChild(document.createTextNode('my name is #' + i));
    tr.appendChild(td);
    a = document.createElement('a');
    a.setAttribute('href', 'http://example.org/' + i + '.html');
    a.appendChild(document.createTextNode('http://example.org/' + i + '.html'));
    td = document.createElement('td');
    td.appendChild(a);
    tr.appendChild(td);
    ul = document.createElement('ul');
    a = document.createElement('a');
    a.setAttribute('href', 'edit.php?id=' + i);
    a.appendChild(document.createTextNode('edit'));
    li = document.createElement('li');
    li.appendChild(a);
    ul.appendChild(li);
    a = document.createElement('a');
    a.setAttribute('href', 'delete.php?id=' + i);
    a.appendChild(document.createTextNode('delete'));
    li = document.createElement('li');
    li.appendChild(a);
    ul.appendChild(li);
    td = document.createElement('td');
```

```
    td.appendChild(ul);
    tr.appendChild(td);
    tbody.appendChild(tr);
  }
  tr = document.createElement('tr');
  th = document.createElement('th');
  th.appendChild(document.createTextNode('yes?'));
  tr.appendChild(th);
  th = document.createElement('th');
  th.appendChild(document.createTextNode('id'));
  tr.appendChild(th);
  th = document.createElement('th');
  th.appendChild(document.createTextNode('name'));
  tr.appendChild(th);
  th = document.createElement('th');
  th.appendChild(document.createTextNode('url'));
  tr.appendChild(th);
  th = document.createElement('th');
  th.appendChild(document.createTextNode('action'));
  tr.appendChild(th);
  thead = document.createElement('thead');
  thead.appendChild(tr);
  table = document.createElement('table');
  table.setAttribute('border', 1);
  table.setAttribute('width', '100%');
  table.appendChild(thead);
  table.appendChild(tbody);
  document.getElementById('here').appendChild(table);
};
```

Os resultados da geração da tabela HTML utilizando innerHTML quando comparados com a utilização de métodos DOM puros são mostrados na figura 3.3. Os benefícios do innerHTML são mais evidentes em versões mais antigas de navegadores (o innerHTML é 3,6 vezes mais rápido no IE6), mas são menos pronunciados em versões mais novas. Quando se trata de navegadores mais novos com base no WebKit ocorre o oposto: a utilização de métodos DOM é levemente mais rápida. Assim, a decisão quanto a qual abordagem deve ser escolhida dependerá do tipo de navegador que seus usuários estão acostumados a utilizar, assim como de suas preferências quanto à codificação.

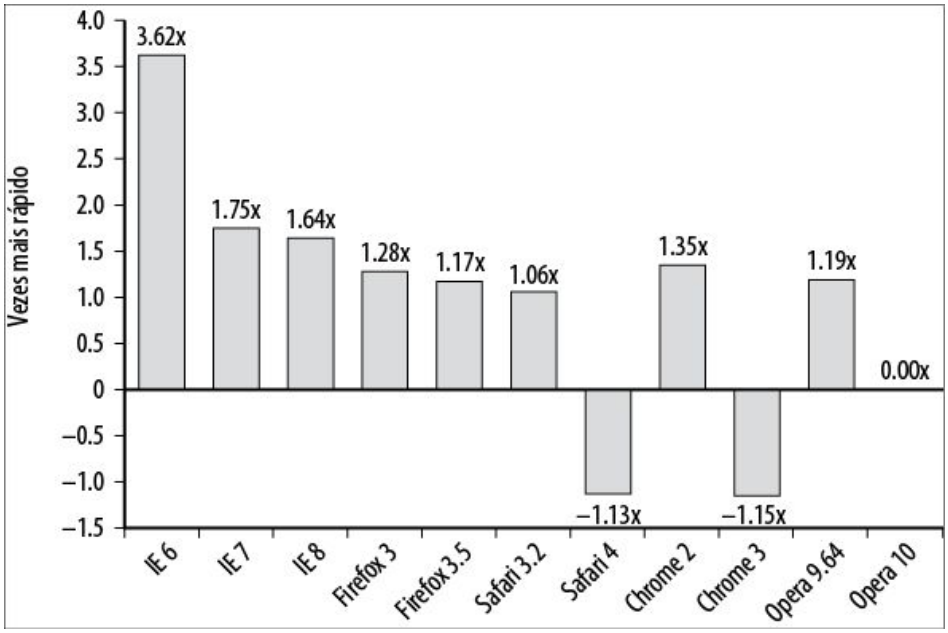


Figura 3.3 – O benefício da utilização do innerHTML em comparação ao DOM para criação de uma tabela com mil linhas: o innerHTML é mais de três vezes mais rápido no IE6 e um pouco mais lento que o DOM nos navegadores que utilizam WebKit mais recentes.

Como lembrete, não se esqueça de que esse exemplo utilizou a concatenação de strings, que não é otimizada em versões mais antigas do IE (veja o capítulo 5). A utilização de um array para concatenar strings extensas fará com que o innerHTML seja ainda mais rápido nesses navegadores.

A utilização do innerHTML resultará em uma execução mais rápida na maioria dos navegadores em caso de operações essenciais ao desempenho que demandam a atualização de uma grande parte da página HTML. Entretanto, nos casos mais corriqueiros não haverá uma diferença tão notável, por isso é interessante que você sempre considere a facilidade de leitura, a manutenção, as preferências de sua equipe e as convenções de codificação quando chegar o momento de decidir sua abordagem preferida.

Clonagem de nodos

Outro modo de atualizar o conteúdo de uma página utilizando métodos DOM é clonar elementos DOM existentes, em vez da criação de novos – em outras palavras, utilizar element.cloneNode() (onde element representa um nodo existente) em lugar de document.createElement().

A clonagem de nodos é mais eficiente na maioria dos navegadores, mas não de maneira muito significativa. Repetindo a tabela do exemplo prévio, podemos criar os elementos apenas uma vez e então copiá-los, o que resulta em um tempo de execução mais rápido:

- 2% no IE8, mas nenhuma alteração no IE6 e no IE7.
- Até 5,5% no Firefox 3.5 e no Safari 4.
- 6% no Opera (mas sem diferença no Opera 10).
- 10% no Chrome 2 e 3% no Chrome3.

Como ilustração, veja uma listagem de código parcial para criação da tabela utilizando elemento.cloneNode():

```
function tableClonedDOM() {
  var i, table, thead, tbody, tr, th, td, a, ul, li,
    oth = document.createElement('th'),
    otd = document.createElement('td'),
    otr = document.createElement('tr'),
    oa = document.createElement('a'),
    oli = document.createElement('li'),
    oul = document.createElement('ul');
  tbody = document.createElement('tbody');
  for (i = 1; i <= 1000; i++) {
    tr = otr.cloneNode(false);
    td = otd.cloneNode(false);
    td.appendChild(document.createTextNode((i % 2) ? 'yes' : 'no'));
    tr.appendChild(td);
    td = otd.cloneNode(false);
    td.appendChild(document.createTextNode(i));
    tr.appendChild(td);
    td = otd.cloneNode(false);
    td.appendChild(document.createTextNode('my name is #' + i));
    tr.appendChild(td);
    // ... o restante do loop ...
  }
  // ... o restante da geração da tabela ...
}
```

Coleções HTML

Coleções HTML são objetos do tipo array que contêm referências a

nodos DOM. Exemplos de coleções são os valores retornados pelos seguintes métodos:

- document.getElementsByName()
- document.getElementsByClassName()
- document.getElementsByTagName()

As seguintes propriedades também retornam coleções HTML:

document.images
: Todos os elementos img da página.

document.links
: Todos os elementos a.

document.forms
: Todos os formulários.

document.forms[0].elements
: Todos os campos do primeiro formulário da página.

Esses métodos e propriedades retornam objetos HTMLCollection, na forma de listas semelhantes a arrays. Eles não são arrays (pois não têm métodos como push() ou slice()), mas fornecem uma propriedade length, assim como os arrays, e permitem o acesso indexado aos elementos da lista. Por exemplo, document.images[1] retorna o segundo elemento da coleção. Conforme definido no padrão DOM, coleções HTML são "presumidas como ativas, o que significa que são automaticamente atualizadas quando o documento subjacente é atualizado" (veja *http://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-75708506*).

As coleções HTML são, na verdade, consultas sobre um documento, e essas consultas são reexecutadas sempre que forem necessárias informações atualizadas, como o número de elementos na coleção (p. ex., o length da coleção). Isso pode ser uma fonte de ineficiências.

Coleções dispendiosas

Para demonstrar que as coleções estão ativas, considere o seguinte fragmento de código:

```
// um loop acidentalmente infinito
var alldivs = document.getElementsByTagName('div');
for (var i = 0; i < alldivs.length; i++) {
  document.body.appendChild(document.createElement('div'))
}
```

Esse código parece apenas duplicar o número de elementos div da página. Ele faz um loop pelos divs existentes e cria, a cada vez, um novo div, acrescentando-o a body. Na verdade, trata-se de um loop infinito, já que a condição de saída do loop, alldivs.length, aumenta em um a cada iteração, refletindo o estado atual do documento subjacente.

Realizar um loop por coleções HTML como essa pode levar a erros lógicos. Além disso, é também mais lento, visto que a consulta deverá ser executada em cada iteração (veja a figura 3.4).

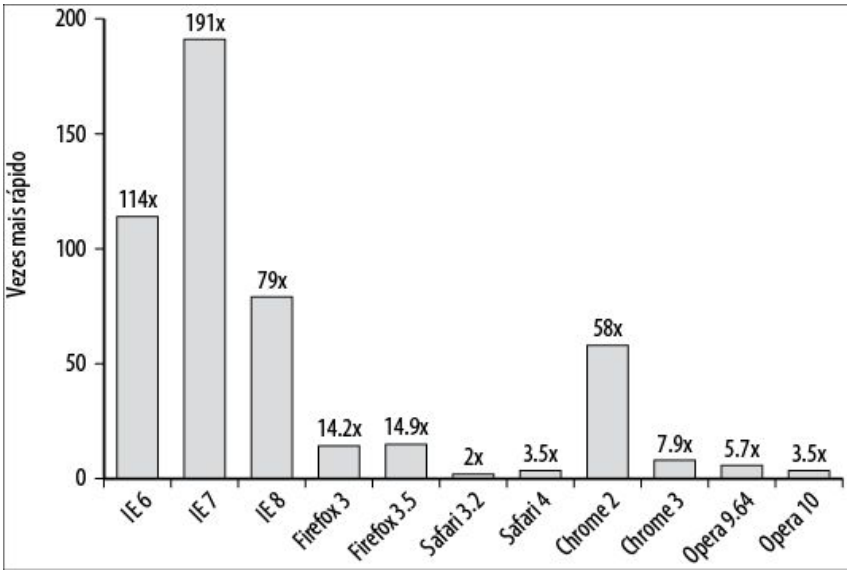


Figura 3.4 – A realização de um loop por um array é significativamente mais rápida do que por uma coleção HTML de mesmo tamanho e conteúdo.

Conforme discutido no capítulo 4, o acesso à propriedade length de um array em condições de controle de loop não é recomendado. O acesso ao length de uma coleção é ainda mais lento que ao length de um array regular por implicar a reexecução da consulta a cada vez. Isso é demonstrado pelo seguinte exemplo, que toma uma coleção col, faz sua cópia para um array arr e depois compara quanto tempo leva uma iteração em cada um deles.

Considere uma função que copia uma coleção HTML em um array regular:

```
function toArray(coll) {
  for (var i = 0, a = [], len = coll.length; i < len; i++) {
    a[i] = coll[i];
  }
  return a;
}
```

E a preparação de uma coleção e uma cópia sua dentro de um array:

```
var coll = document.getElementsByTagName('div');
var ar = toArray(coll);
```

As duas funções a serem comparadas seriam:

```
// mais lenta
function loopCollection() {
  for (var count = 0; count < coll.length; count++) {
    /* não faça nada */
  }
}
// mais rápida
function loopCopiedArray() {
  for (var count = 0; count < arr.length; count++) {
    /* não faça nada */
  }
}
```

Quando o `length` da coleção é acessado em cada iteração, ele faz com que a coleção seja atualizada, prejudicando significativamente o desempenho em todos os navegadores. Podemos otimizar essa situação armazenando o `length` da coleção em cache, dentro de uma variável, seguido pela utilização dessa variável para a comparação com a condição de saída do loop:

```
function loopCacheLengthCollection() {
  var coll = document.getElementsByTagName('div'),
    len = coll.length;
  for (var count = 0; count < len; count++) {
    /* não faça nada */
  }
}
```

Essa solução rodará quase tão rapidamente quanto `loopCopiedArray()`.

Na maioria dos casos que demandam um único loop sobre uma coleção relativamente pequena, o simples armazenamento em cache de length já será suficiente. Contudo, a realização de loops em um array é mais rápido do que loops em uma coleção, de modo que se primeiro copiarmos os elementos da coleção para um array, o acesso a suas propriedades será mais rápido. Tenha em mente que isso envolverá um passo extra e mais um loop pela coleção, portanto, é importante decidir se a utilização de uma cópia do array será benéfica em seu caso específico.

Consulte a função toArray() mostrada anteriormente para ver um exemplo de uma função genérica do tipo coleção-para-array.

Variáveis locais durante o acesso a elementos de uma coleção

O exemplo prévio utilizou apenas um loop vazio, mas o que acontece quando os elementos da coleção são acessados dentro do loop?

Em geral, para qualquer tipo de acesso DOM, é preferível utilizar uma variável local quando a mesma propriedade ou o mesmo método DOM é acessado mais de uma vez. Ao realizar um loop em uma coleção, a primeira otimização que podemos fazer é armazenar a coleção em uma variável local e armazenar length fora do loop, seguido pelo uso de uma variável local para elementos que são acessados mais de uma vez.

No próximo exemplo, três propriedades de cada elemento são acessadas dentro do loop. A versão mais lenta acessa o document global a cada iteração, uma versão otimizada armazena em cache uma referência à coleção e a versão mais rápida guarda o elemento atual da coleção em uma variável. As três versões armazenam em cache o length da coleção.

```
// versão lenta
function collectionGlobal() {
  var coll = document.getElementsByTagName('div'),
    len = coll.length,
```

```
    name = '';
  for (var count = 0; count < len; count++) {
    name = document.getElementsByTagName('div')[count].nodeName;
    name = document.getElementsByTagName('div')[count].nodeType;
    name = document.getElementsByTagName('div')[count].tagName;
  }
  return name;
};
// versão rápida
function collectionLocal() {
  var coll = document.getElementsByTagName('div'),
    len = coll.length,
    name = '';
  for (var count = 0; count < len; count++) {
    name = coll[count].nodeName;
    name = coll[count].nodeType;
    name = coll[count].tagName;
  }
  return name;
};
// versão mais rápida
function collectionNodesLocal() {
  var coll = document.getElementsByTagName('div'),
    len = coll.length,
    name = '',
    el = null;
  for (var count = 0; count < len; count++) {
    el = coll[count];
    name = el.nodeName;
    name = el.nodeType;
    name = el.tagName;
  }
  return name;
};
```

A figura 3.5 mostra os benefícios da otimização feita por meio de loops de coleção. A primeira barra exibe quantas vezes mais rápido é o acesso da coleção utilizando uma referência local, enquanto a segunda barra mostra que há benefício adicional no armazenamento em cache de itens de coleção quando são acessados várias vezes.

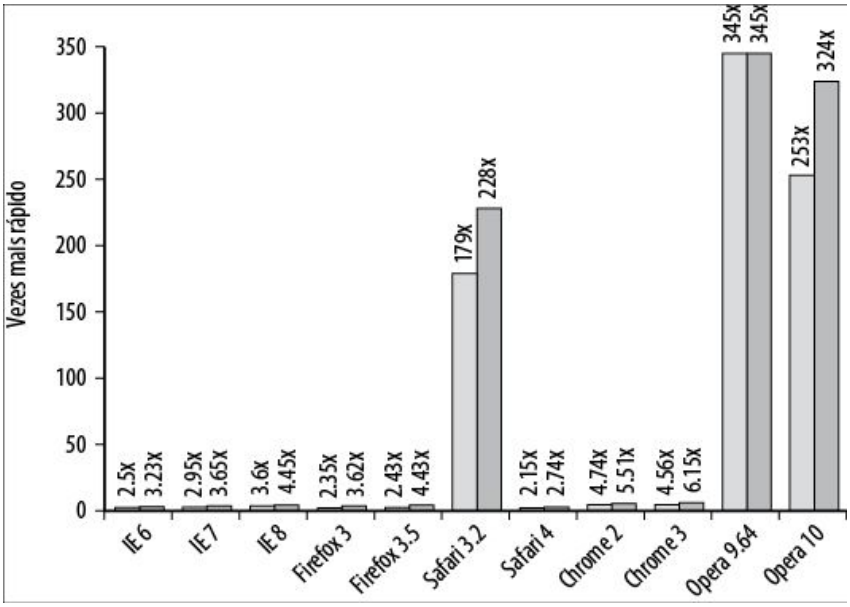


Figura 3.5 – Benefício da utilização de variáveis locais para armazenar referências a uma coleção e a seus elementos durante loops.

Caminhando pelo DOM

A API do DOM oferece várias formas pelas quais partes específicas da estrutura geral de um documento podem ser acessadas. Em casos onde podemos escolher entre as abordagens disponíveis, é benéfico usar a API mais eficiente para o trabalho específico.

Rastreamento (crawling) do DOM

Muitas vezes você tem de partir de um elemento DOM e lidar com os elementos que o cercam, talvez iterando de modo recursivo por todos os filhos. Isso pode ser feito utilizando a coleção childNodes ou pela obtenção do irmão de cada elemento por meio de nextSibling.

Considere estas duas abordagens equivalentes mostrando uma visita não recursiva aos filhos de um elemento:

```
function testNextSibling() {
  var el = document.getElementById('mydiv'),
    ch = el.firstChild,
    name = '';
  do {
    name = ch.nodeName;
  } while (ch = ch.nextSibling);
  return name;
```

```
};
function testChildNodes() {
  var el = document.getElementById('mydiv'),
    ch = el.childNodes,
    len = ch.length,
    name = '';
  for (var count = 0; count < len; count++) {
    name = ch[count].nodeName;
  }
  return name;
};
```

Não se esqueça de que childNodes é uma coleção e, por isso, deve ser abordada com cautela, cacheando length em loops para que não seja atualizado em cada iteração.

As duas abordagens são praticamente idênticas em termos de tempo de execução nos vários navegadores. Entretanto, no IE, nextSibling é executada muito mais rápido do que childNodes. No IE6, nextSibling é 16 vezes mais rápida e, no IE7, 105 vezes mais rápida. Dados esses resultados, a utilização de nextSibling surge como o método preferido para o rastreamento do DOM em versões mais antigas do IE, nos casos em que o desempenho for essencial. Em todos os outros casos, trata-se em grande parte de uma questão de preferência pessoal ou de sua equipe.

Nodos de elementos

Propriedades DOM como childNodes, firstChild e nextSibling não distinguem entre nodos de elementos e outros tipos de nodos, como comentários e nodos de texto (muitas vezes apenas espaços entre duas tags). Na maioria dos casos, apenas os nodos de elementos tem de ser acessados, assim, em um loop, é mais provável que o código tenha de conferir o tipo de nodo retornado e filtrar nodos que não sejam de elementos. Esse tipo de verificação e filtragem representa trabalho desnecessário no DOM.

Muitos navegadores modernos oferecem APIs que retornam apenas nodos de elementos. É melhor usar essas APIs quando estiverem disponíveis, uma vez que serão mais rápidas do que se você

realizar, por conta própria, a filtragem no JavaScript. A tabela 3.1 lista essas convenientes propriedades DOM.

Todas as propriedades listadas na tabela 3.1 são aceitas desde o Firefox 3.5, o Safari 4, o Chrome 2 e o Opera 9.62. Dessas propriedades, as versões 6, 7 e 8 do IE aceitam apenas children.

Tabela 3.1 – Propriedades DOM que distinguem nodos de elementos (tags HTML) de todos os nodos

Propriedade	Utilize como substituto de
children	childNodes
childElementCount	childNodes.length
firstElementChild	firstChild
lastElementChild	lastChild
nextElementSibling	nextSibling
previousElementSibling	previousSibling

A realização de um loop em children, em vez de childNodes é mais rápida porque, em geral, existem menos itens pelos quais passar. Espaços em branco (whitespace) no código-fonte HTML são na verdade nodos de texto e não estão incluídos na coleção children. children é mais rápido do que childNodes em todos os navegadores, ainda que normalmente não por uma grande margem – de uma vez e meia a três vezes mais rápido. Uma exceção notável é o IE, onde a iteração pela coleção children é significativamente mais rápida do que a iteração por childNodes – 24 vezes mais rápida no IE6 e 127 vezes mais rápida no IE7.

A API de seletores

Ao identificar os elementos do DOM com os quais realizarão seu trabalho, os desenvolvedores muitas vezes necessitam de um controle mais minucioso do que o oferecido por métodos como getElementById() e getElementsByTagName(). Às vezes você combinará essas chamadas e fará uma iteração pelos nodos retornados para alcançar uma lista de elementos desejada, mas esse processo de refinamento pode se tornar ineficiente.

Por outro lado, a utilização de seletores CSS pode ser um modo

conveniente de identificar nodos, uma vez que os desenvolvedores já estão familiarizados com CSS. Muitas bibliotecas JavaScript fornecem APIs para esse propósito, sendo que, atualmente, versões recentes de navegadores fornecem o querySelectorAll() como um método DOM nativo. Naturalmente essa abordagem é mais rápida do que a utilização de JavaScript e DOM para iterar e examinar uma lista de elementos.

Considere o seguinte:

```
var elements = document.querySelectorAll('#menu a');
```

O valor de elements conterá uma lista de referências a todos os elementos a encontrados em um elemento com id="menu". O método querySelectorAll() toma uma string CSS seletora como argumento e retorna uma NodeList – um objeto de tipo array contendo todos os nodos correspondentes. O método não retorna uma coleção HTML, de modo que os nodos retornados não representam a estrutura ativa do documento. Isso evita os problemas de desempenho (e potencialmente lógicos) advindos de uma coleção HTML, que discutimos anteriormente neste capítulo.

Para alcançar o mesmo objetivo do código anterior sem usar querySelectorAll(), é necessário utilizar um código maior:

```
var elements = document.getElementById('menu').getElementsByTagName('a');
```

Nesse caso, elements será uma coleção HTML, assim, você terá de copiá-la em um array caso queira o mesmo tipo de lista estática como a retornada por querySelectorAll().

A utilização de querySelectorAll() é ainda mais conveniente quando temos de trabalhar com um agrupamento de várias consultas. Por exemplo, se a página apresentar alguns elementos div com um nome de classe “warning” e outros com o nome “notice”, para obter uma lista de todos eles você pode utilizar querySelectorAll():

```
var errs = document.querySelectorAll('div.warning, div.notice');
```

Produzir a mesma lista sem utilizar o querySelectorAll() é consideravelmente mais trabalhoso. Uma forma de fazê-lo é pela seleção de todos os elementos div, seguida por iterações para

filtragem daqueles que não são necessários.

```
var errs = [],
  divs = document.getElementsByTagName('div'),
  classname = '';
for (var i = 0, len = divs.length; i < len; i++) {
  classname = divs[i].className;
  if (classname === 'notice' || classname === 'warning') {
    errs.push(divs[i]);
  }
}
```

A comparação dos dois códigos mostra que a utilização da API de seletores é duas a seis vezes mais rápida em todos os navegadores (Figura 3.6).

A API de seletores é aceita de modo nativo em navegadores desde as versões Internet Explorer 8, Firefox 3.5, Safari 3.1, Chrome 1 e Opera 10.

Como mostram os resultados da figura, é uma boa ideia checar se existe suporte a `document.querySelectorAll()` e utilizar essa opção quando ela estiver disponível. Além disso, caso esteja utilizando uma API de seletores fornecida por uma biblioteca JavaScript, certifique-se de que a biblioteca usa uma API nativa. Se esse não for o caso, talvez seja necessário atualizar a versão da biblioteca.

Há também outro método que podemos aproveitar: `querySelector()`. Trata-se de um método conveniente que retorna apenas o primeiro nodo que corresponde à consulta.

Esses dois métodos são propriedades de nodo DOM, de modo que você pode usar `document.querySelector('.myclass')` para consultar nodos em todo o documento ou consultar apenas uma subárvore utilizando `elref.querySelector('.myclass')`, onde `elref` é uma referência a um elemento DOM.

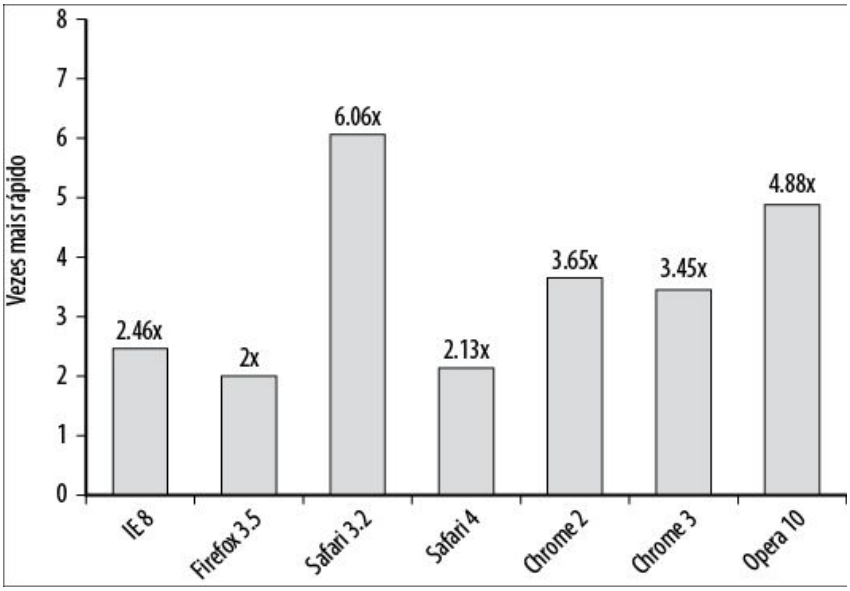


Figura 3.6 – O benefício da utilização da API de seletores quando comparado à iteração pelos resultados de getElementsByTagName().

Repaints e reflows

Uma vez que o navegador tenha executado o download de todos os componentes de uma página – a marcação HTML, o JavaScript, o CSS, as imagens – ele fará o parsing de todos os arquivos e criará duas estruturas de dados internas:

Uma árvore DOM

Representação da estrutura da página.

Uma árvore de renderização

Uma representação de como os nodos DOM serão exibidos.

A árvore de renderização tem ao menos um nodo para cada nodo da árvore DOM que deve ser exibido (elementos DOM ocultos não apresentam um nodo correspondente na árvore de renderização). Nodos da árvore de renderização são chamados de *frames* ou *boxes* de acordo com o modelo CSS, que trata os elementos da página como boxes com espaçamento, margens, bordas e posicionamento. Assim que as árvores DOM e de renderização tiverem sido construídas, o navegador poderá exibir (pintar, ou "paint") os elementos da página.

Quando uma alteração no DOM afeta a geometria de um elemento (sua largura ou altura) – como uma alteração na grossura da borda ou a adição de mais texto a um parágrafo resultando em uma nova linha – o navegador tem de recalcular a geometria do elemento, bem como a geometria e o posicionamento de outros elementos que possam ter sido afetados pela alteração. A parte da árvore de renderização que foi afetada pela mudança é invalidada, reconstruindo a árvore de renderização. Esse processo é conhecido por reflow. Uma vez que um reflow esteja completo, o navegador redesenha as partes afetadas da tela em um processo chamado repaint.

Nem todas as alterações ao DOM afetam a geometria. Por exemplo, a alteração da cor de fundo de um elemento não altera sua largura ou altura. Nesse caso, há apenas um repaint (e nenhum reflow), já que o layout da página não foi alterado.

Repaints e reflows são operações dispendiosas que podem fazer com que a interface de usuário da aplicação web se torne menos responsiva. Dessa forma, é interessante que eles ocorram o mínimo de vezes possível.

Quando ocorre um reflow?

Como mencionado anteriormente, um reflow é necessário sempre que houver uma mudança de layout ou geometria. Isso ocorre quando:

- Elementos DOM visíveis são adicionados ou removidos.
- Elementos mudam de posição.
- Elementos mudam de tamanho (devido a uma mudança na margem, no espaçamento, na largura da borda, na largura, na altura etc.).
- Há uma mudança de conteúdo, por exemplo, alterações de texto ou substituição de uma imagem por uma de tamanho diferente.
- Ocorre a renderização inicial da página.
- A janela do navegador é redimensionada.

Dependendo da natureza da mudança, uma parte maior ou menor da árvore de renderização terá de ser recalculada. Algumas alterações podem provocar um reflow da página inteira: por exemplo, quando uma barra de rolagem aparece.

Enfileiramento e esvaziamento das alterações à árvore de renderização

Devido aos custos computacionais associados a cada reflow, a maioria dos navegadores otimiza esse processo enfileirando alterações e executando-as em lotes. Entretanto, você pode (muitas vezes involuntariamente) forçar o esvaziamento dessa fila e demandar que todas as alterações agendadas sejam aplicadas imediatamente. O esvaziamento da fila ocorre quando você deseja acessar informações de layout, o que significa usar qualquer um destes comandos:

- offsetTop, offsetLeft, offsetWidth, offsetHeight
- scrollTop, scrollLeft, scrollWidth, scrollHeight
- clientTop, clientLeft, clientWidth, clientHeight
- getComputedStyle() (currentStyle no IE)

A informação de layout retornada por essas propriedades e métodos tem de estar atualizada, o que faz com que o navegador tenha de executar quaisquer alterações pendentes no enfileiramento e reflow da renderização para que possa retornar os valores corretos.

Durante o processo da alteração de estilos, é preferível não utilizar nenhuma das propriedades mostradas na lista precedente. Todas elas provocarão o esvaziamento da fila, mesmo em casos em que você esteja recuperando informações de layout que não foram alteradas recentemente ou nem mesmo sejam relevantes às últimas alterações.

Considere o seguinte exemplo, onde temos a alteração da mesma propriedade de estilo três vezes seguidas (isso provavelmente não é algo que você veria em um código real, mas é uma ilustração

isolada de um tópico importante):

```
// definição e recuperação dos estilos em sucessão
var computed,
  tmp = '',
  bodystyle = document.body.style;
if (document.body.currentStyle) { // IE, Opera
  computed = document.body.currentStyle;
} else { // W3C
  computed = document.defaultView.getComputedStyle(document.body, '');
}
// modo ineficiente de modificação da mesma propriedade
// e da recuperação da informação de estilo logo em seguida
bodystyle.color = 'red';
tmp = computed.backgroundColor;
bodystyle.color = 'white';
tmp = computed.backgroundImage;
bodystyle.color = 'green';
tmp = computed.backgroundAttachment;
```

Nesse exemplo, a cor do primeiro plano do elemento `body` sofre três alterações, sendo que cada uma delas é seguida pelo acesso à propriedade de estilo computada. As propriedades acessadas – `backgroundColor`, `backgroundImage` e `backgroundAttachment` – não têm relação com a cor sendo alterada. Ainda assim, o navegador tem de esvaziar a fila de renderização, já que uma propriedade de estilo computada foi solicitada.

Uma abordagem preferível a esse exemplo ineficiente é nunca solicitar informações de layout enquanto o layout sofre alterações. Caso a recuperação do estilo computado fosse movida para o final, o código teria a seguinte aparência:

```
bodystyle.color = 'red';
bodystyle.color = 'white';
bodystyle.color = 'green';
tmp = computed.backgroundColor;
tmp = computed.backgroundImage;
tmp = computed.backgroundAttachment;
```

O segundo exemplo terá um melhor desempenho em todos os navegadores, como mostra a figura 3.7.

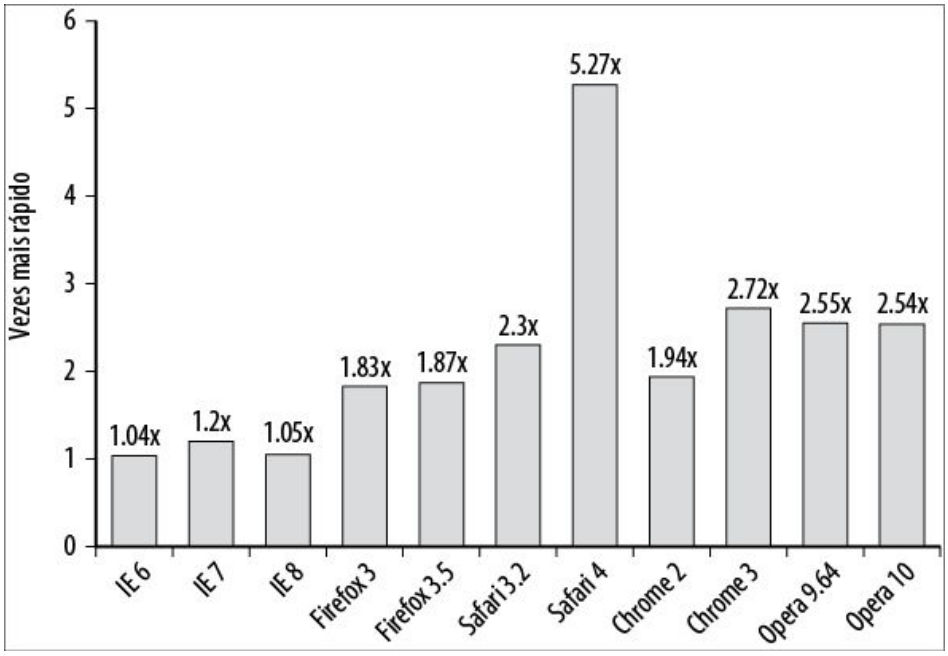


Figura 3.7 – Benefício da prevenção de reflows pelo adiamento do acesso às informações de layout.

Minimização de repaints e reflows

Reflows e repaints podem ser custosos. Assim, se quisermos aplicações responsivas devemos reduzir a ocorrência dessas operações. Para minimizar a frequência desses eventos, podemos combinar várias alterações de estilo ao DOM em um único lote e aplicá-las de uma vez só.

Alterações de estilo

Considere o seguinte exemplo:

```
var el = document.getElementById('mydiv');
el.style.borderLeft = '1px';
el.style.borderRight = '2px';
el.style.padding = '5px';
```

Aqui temos três propriedades de estilo sendo alteradas, cada uma delas afetando a geometria do elemento. Na pior das hipóteses, isso fará com que o navegador execute três vezes o processo de reflow. A maioria dos navegadores modernos contém otimizações para esses casos, efetuando um único reflow, mas ainda podemos encontrar ineficiências em navegadores mais antigos, ou caso haja

um processo assíncrono separado ocorrendo ao mesmo tempo (p. ex., o uso de um timer). Se outro código solicitar informações de layout enquanto esse código estiver em execução podem ocorrer até três refluxos. Da mesma forma, o código está acionando o DOM quatro vezes e pode ser otimizado.

Um modo mais eficiente de se atingir o mesmo resultado é combinar todas as alterações e depois aplicá-las de uma só vez, modificando apenas uma vez o DOM. Podemos fazê-lo utilizando a propriedade cssText:

```
var el = document.getElementById('mydiv');
el.style.cssText = 'border-left: 1px; border-right: 2px; padding: 5px;';
```

A modificação da propriedade cssText, como mostra o exemplo, substitui informações de estilo existentes. Desse modo, se desejar manter os estilos existentes, você pode anexar o seguinte fragmento à string cssText:

```
el.style.cssText += '; border-left: 1px;';
```

Outra forma de aplicar alterações de estilo apenas uma vez é pela alteração do nome da classe CSS em vez da mudança dos estilos embutidos. Essa abordagem é aplicável em casos onde os estilos não dependem de lógica ou de cálculos do tempo de execução. A alteração do nome da classe CSS é mais limpa e de manutenção mais fácil; também ajuda a manter seus scripts livres de código de apresentação, ainda que possa envolver um leve impacto no desempenho devido ao fato da cascata ter de ser conferida quando as classes são alteradas.

```
var el = document.getElementById('mydiv');
el.className = 'active';
```

Realização em lotes de alterações ao DOM

Quando há um grande volume de mudanças a serem feitas ao elemento DOM, você pode reduzir a quantidade de repaints e reflows seguindo os seguintes passos:

1. Remova o elemento do fluxo do documento.
2. Aplique as várias alterações.

3. Traga o elemento de volta ao documento.

Esse processo causa dois reflows – um no passo número 1, outro no número 3. Caso omita passos desse processo, toda mudança efetuada no passo 2 poderá provocar seus próprios reflows.

Há três modos básicos de modificar o DOM fora do documento:

- Oculte o elemento, aplique as alterações e exiba-o novamente.
- Utilize um fragmento de documento para criar uma subárvore externa ao DOM e copie-a ao documento.
- Copie o elemento original em um nodo externo ao documento, faça as modificações necessárias e substitua o elemento original quando terminar.

Para ilustrar as manipulações externas ao documento, considere uma lista de links que deve ser atualizada com mais informações:

```
<ul id="mylist">
  <li><a href="http://phpied.com">Stoyan</a></li>
  <li><a href="http://julienlecomte.com">Julien</a></li>
</ul>
```

Suponha que dados adicionais, já contidos em um objeto, devam ser inseridos nessa lista. Os dados são definidos da seguinte forma:

```
var data = [
  {
    "name": "Nicholas",
    "url": "http://nczonline.net"
  },
  {
    "name": "Ross",
    "url": "http://techfoolery.com"
  }
];
```

A seguir, temos uma função genérica para atualização de um nodo com os novos dados:

```
function appendDataToElement(appendToElement, data) {
  var a, li;
  for (var i = 0, max = data.length; i < max; i++) {
    a = document.createElement('a');
    a.href = data[i].url;
    a.appendChild(document.createTextNode(data[i].name));
```

```
    li = document.createElement('li');
    li.appendChild(a);
    appendToElement.appendChild(li);
  }
};
```

O modo mais óbvio de efetuarmos a atualização da lista com os dados, sem nos preocuparmos com reflows, seria o seguinte:

```
var ul = document.getElementById('mylist');
appendDataToElement(ul, data);
```

Entretanto, com essa abordagem, cada nova entrada a partir do array de dados será anexada à árvore DOM ativa e causará um reflow. Como discutimos anteriormente, um modo de reduzirmos os reflows é a remoção temporária do elemento <ul> do fluxo do documento alterando a propriedade display, seguida por sua reversão:

```
var ul = document.getElementById('mylist');
ul.style.display = 'none';
appendDataToElement(ul, data);
ul.style.display = 'block';
```

Também podemos minimizar o número de reflows criando e atualizando um fragmento de documento fora do documento e, depois, anexando-o à lista original. Um fragmento de documento é uma versão mais leve do objeto document, sendo indicado para ajudar exatamente nesse tipo de tarefa – atualização e movimentação de nodos. Um atributo sintaticamente conveniente dos fragmentos de documento é o de que, quando anexamos um fragmento a um nodo, os filhos do fragmento são anexados, e não o fragmento em si. A solução a seguir utiliza uma linha a menos de código, causa apenas um reflow e toca o DOM ativo apenas uma vez:

```
var fragment = document.createDocumentFragment();
appendDataToElement(fragment, data);
document.getElementById('mylist').appendChild(fragment);
```

Uma terceira solução seria a criação de uma cópia do nodo que você deseja atualizar, o trabalho sobre ela e, então, quando tiver terminado, a substituição do nodo anterior pela nova cópia atualizada:

```
var old = document.getElementById('mylist');
var clone = old.cloneNode(true);
appendDataToElement(clone, data);
old.parentNode.replaceChild(clone, old);
```

Recomenda-se que você utilize fragmentos de documento (a segunda solução) sempre que possível, uma vez que envolvem o menor número de reflows e de manipulações ao DOM. A única desvantagem potencial é que essa prática é atualmente pouco comum, e alguns membros de sua equipe podem não estar familiarizados com ela.

Armazenamento em cache de informações de layout

Como já mencionamos, navegadores tentam minimizar o número de reflows enfileirando alterações e executando-as em lotes. Todavia, quando você solicita informações de layout, como offsets, valores de scroll ou de estilo computados, o navegador esvazia a fila e aplica todas as alterações para que possa retornar o valor atualizado. É sempre interessante minimizar o número de solicitações por informações de layout, e quando elas tiverem de ser feitas, que sejam designadas a variáveis locais, para que o trabalho seja feito justamente sobre esses valores locais.

Considere como exemplo a movimentação diagonal de um elemento `myElement`, um pixel por vez, partindo da posição 100 x 100 px e terminando em 500 x 500 px. No corpo de um loop de tempo limite você pode utilizar:

```
// ineficiente
myElement.style.left = 1 + myElement.offsetLeft + 'px';
myElement.style.top = 1 + myElement.offsetTop + 'px';
if (myElement.offsetLeft >= 500) {
  stopAnimation();
}
```

Toda vez que o elemento se mover, o código solicitará os valores de offset, fazendo com que o navegador esvazie a fila de renderização e não se beneficie de suas otimizações. Um modo mais eficiente de

fazer a mesma operação seria tomar o valor da posição inicial e designar esse valor a uma variável como `var current = myElement.offsetLeft;`. A seguir, dentro do loop de animação, trabalharíamos com a variável atual, sem solicitar offsets:

```
current++
myElement.style.left = current + 'px';
myElement.style.top = current + 'px';
if (current >= 500) {
  stopAnimation();
}
```

Remova elementos do fluxo para animações

Mostrar e ocultar partes de uma página de modo expandir/ocultar é um padrão de interação comum. Muitas vezes inclui uma animação de geometria da área sendo expandida, empurrando para baixo o resto do conteúdo da página.

Reflows às vezes afetam apenas uma pequena parte da árvore de renderização, mas há situações em que podem afetar uma grande porção ou até mesmo a árvore inteira. Quanto menos reflows o navegador tiver de efetuar, mais responsivo será sua aplicação. Assim, sempre que uma animação no topo da página empurrar para baixo praticamente todo o conteúdo, teremos um grande reflow que será consideravelmente dispendioso e que fornecerá ao usuário uma aparência visual “picada”. Quanto mais nodos na árvore de renderização demandarem recálculos, pior será sua situação.

Uma técnica preferível para evitar o reflow de uma grande parte da página envolve os seguintes passos:

1. Utilize posicionamento absoluto para o elemento que você deseja animar na página, removendo-o do fluxo de layout.
2. Anime o elemento. Quando ele for expandido, cobrirá temporariamente parte da página. Isso é um repaint, mas afeta apenas uma pequena parte da página em lugar do reflow e repaint de um trecho substancial.
3. Quando a animação for concluída, restaure o posicionamento

empurrando para baixo o restante do documento uma única vez.

O IE e o :hover

Desde a versão 7, o IE pode aplicar o pseudosseletor CSS :hover sobre qualquer elemento (de modo estrito). Entretanto, caso você tenha um número significativo de elementos com :hover, sua responsividade será impactada. O problema é ainda mais visível no IE8.

Vejamos um exemplo: caso crie uma tabela com 500 a 1.000 linhas e cinco colunas, utilizando tr:hover para alterar a cor de fundo e destacar a linha selecionada, seu desempenho sofrerá muito conforme o usuário se movimentar pela tabela. A aplicação do efeito de destaque é lenta, e a utilização da CPU aumenta em 80 a 90%. Por isso, evite esse efeito quando trabalhar com um grande número de elementos, como grandes tabelas ou longas listas de itens.

Delegação de eventos

Onde há um grande número de elementos em uma página, cada um apresentando um ou mais manipuladores de eventos associados (como onclick), poderemos notar um prejuízo sobre o desempenho. Anexar cada manipulador traz consigo um custo – tornando as páginas mais pesadas (mais marcação ou código JavaScript) ou afetando o tempo de execução. Quanto mais nodos DOM você tiver de usar e modificar, mais lenta será sua aplicação, especialmente porque a fase de associação de eventos geralmente ocorre no evento onload (ou DOMContentReady), um período bem disputado em todas as páginas web que apresentam grande interatividade. A associação de eventos consome tempo de processamento e, além disso, faz com que o navegador tenha de monitorar cada manipulador, consumindo mais memória. No final das contas, um grande número desses manipuladores de eventos pode nunca ser utilizado (uma vez que o usuário clica em um botão ou link, não em 100 deles, por exemplo), fazendo com que muito do trabalho seja desnecessário.

Uma técnica simples e elegante para a manipulação de eventos DOM é a delegação de eventos. Ela tem como base o fato de que eventos vêm à tona e podem ser manipulados por um elemento-pai. Com a delegação de eventos, apenas um manipulador é anexado ao elemento envoltório (wrapper). Esse manipulador lidará com todos os eventos que ocorram com os descendentes do pai.

Segundo o padrão DOM, cada evento tem três fases:

- Captura
- No alvo
- Borbulhamento

A fase de captura não é aceita pelo IE, mas seu “borbulhamento” é suficiente para o propósito da delegação. Considere uma página com a estrutura mostrada na figura 3.8.

Figura 3.8 – Um exemplo da árvore DOM.

Quando o usuário clica no link “menu #1”, o evento do clique é primeiro recebido pelo elemento <a>. Em seguida, ele “borbulha” mais para cima na árvore DOM sendo recebido pelo elemento <li>, depois por <ul>, e assim por diante, até o topo do documento, podendo atingir até window. Isso permite que você anexe apenas um manipulador de evento para um elemento-pai e receba notificações para todos os eventos que ocorram aos filhos.

Suponha que você deseje oferecer uma experiência Ajax progressivamente aprimorada para o documento mostrado na figura. Caso o usuário esteja com JavaScript desabilitado, os links do menu

funcionarão normalmente e recarregarão a página. No entanto, caso o JavaScript esteja ligado e o usuário agente tenha capacidade, você deseja interceptar todos os cliques, impedir o comportamento normal (que seria o acompanhamento do link), enviar uma solicitação Ajax para obtenção do conteúdo e atualizar uma porção da página recarregá-la por inteiro. Para fazê-lo utilizando a delegação de eventos, você pode anexar um detector de cliques ao elemento “menu” de UL que envolve todos os links e inspeciona todos os cliques para ver se eles vêm de um link.

```
document.getElementById('menu').onclick = function(e) {

  // target do navegador-x
  e = e || window.event;
  var target = e.target || e.srcElement;

  var pageid, hrefparts;

  // estamos interessados em hrefs
  // saia da função em cliques que não venham de links
  if (target.nodeName !== 'A') {
    return;
  }

  // identifique o ID da página a partir do link
  hrefparts = target.href.split('/');
  pageid = hrefparts[hrefparts.length - 1];
  pageid = pageid.replace('.html', '');

  // atualize a página
  ajaxRequest('xhr.php?page=' + id, updatePageContents);

  // navegador-x impede a ação normal e cancela o borbulhamento
  if (typeof e.preventDefault === 'function') {
    e.preventDefault();
    e.stopPropagation();
  } else {
    e.returnValue = false;
    e.cancelBubble = true;
  }
};
```

Como podemos ver, a técnica de delegação de eventos não é complicada; você tem apenas de inspecionar eventos para ver se são originados de elementos nos quais você está interessado. É preciso um pouco mais de código para que haja compatibilidade

com vários navegadores, mas se mover essa parte a uma biblioteca reutilizável, seu código se tornará bem limpo. As partes necessárias para compatibilidade com vários navegadores são:

- Acesso ao objeto do evento e identificação da fonte (alvo) do evento.
- Cancelamento do "borbulhamento" pela árvore do documento (opcional).
- Prevenção do comportamento normal (opcional, mas necessário em nosso exemplo porque a tarefa buscava capturar os links, e não segui-los).

Resumo

O acesso e a manipulação do DOM são partes importantes das aplicações web modernas. Ainda assim, todas as vezes que cruzar a ponte entre os reinos do ECMAScript e do DOM, haverá um preço a ser pago. Para reduzir os custos de desempenho relacionados à criação de scripts que utilizem DOM, lembre-se destes pontos:

- Minimize o acesso ao DOM e tente fazer o máximo de suas tarefas apenas com JavaScript.
- Utilize variáveis locais para armazenar referências ao DOM que serão acessadas várias vezes.
- Tenha cuidado quando lidar com coleções HTML, uma vez que elas representam o documento subjacente ativo. Armazene em cache o `length` da coleção e utilize-o para iterações. Faça uma cópia da coleção em um array para realização de trabalhos pesados.
- Utilize APIs mais rápidas sempre que possível, como `querySelectorAll()` e `firstElementChild`.
- Tenha em mente os repaints e reflows; realize alterações de estilo em lotes, manipule a árvore DOM apenas em modo "off-line", minimize o acesso a informações de layout e armazene-as em cache.
- Utilize posicionamento absoluto durante animações, valendo-se

de proxies do tipo arrastar e soltar.

- Aproveite a delegação de eventos para minimizar o número de manipuladores de eventos.

---

1 N.T.: A responsividade de um sistema interativo descreve com que rapidez ele responde ao input de um usuário (p. ex. o grau de comunicação com o sistema). (Fonte: Wikipédia)

CAPÍTULO 4

Algoritmos e controle de fluxo

A estrutura geral de seu código é um dos principais determinantes da velocidade de execução que ele terá. Ter pouco código não necessariamente significa que ele será executado com rapidez, assim como ter muito código não necessariamente significa que sua execução será lenta. Muito do impacto sobre o desempenho está diretamente relacionado à forma como o código foi organizado e ao modo como se pretende resolver um dado problema.

As técnicas deste capítulo não são necessariamente exclusivas à linguagem JavaScript e são comumente ensinadas como otimizações de desempenho inclusive para outras linguagens. Entretanto, há diferenças quanto a alguns conselhos comuns, já que existem muitas engines JavaScript com as quais temos que lidar e suas peculiaridades têm de ser consideradas. Ainda assim, todas as técnicas são baseadas em conhecimentos notórios da ciência da computação.

Loops

Na maioria das linguagens de programação, grande parte do tempo de execução do código é gasta dentro de loops. A realização de loops por uma série de valores é um dos padrões de programação mais frequentemente utilizados. Justamente por isso ela é também uma área onde os esforços para melhoria do desempenho devem ser concentrados. A compreensão do impacto sobre o desempenho causado pelos loops é particularmente importante, já que loops infinitos ou de longa duração afetam de modo intenso a experiência geral do usuário.

Tipos de loops

A ECMA-262, terceira edição, especificação que define a sintaxe e o comportamento básico do JavaScript, define quatro tipos de loops. O primeiro é o loop padrão, que compartilha sua sintaxe com outras linguagens que têm C como base:

```
for (var i=0; i < 10; i++) {
  // corpo do loop
}
```

O loop for tende a ser a forma de loop JavaScript mais utilizada. Existem quatro partes do loop for: inicialização, condição pré-teste, pós-execução e corpo do loop. Quando um loop é encontrado, o código de inicialização é executado primeiro, seguido pela condição pré-teste. Caso essa condição seja avaliada como verdadeira (true), o corpo do loop é executado. Depois da execução do corpo, o código pós-execução é executado. O encapsulamento do loop for acaba por torná-lo um favorito entre desenvolvedores.

 Observe que o posicionamento de uma instrução var na parte de inicialização de um loop for cria uma variável no nível da função, e não no nível apenas do loop. O JavaScript apresenta apenas o escopo do nível da função, de modo que a definição de uma variável dentro de um loop for tem o mesmo efeito que a definição de uma nova variável fora dele.

O segundo tipo de loop é o loop while. Um loop while é um simples loop pré-teste composto de uma condição pré-teste e de um corpo:

```
var i = 0;
while(i < 10) {
  // corpo do loop
  i++;
}
```

Antes que o corpo do loop seja executado, a condição pré-teste é avaliada. Caso a condição seja avaliada como verdadeira, o corpo do loop será executado; caso contrário, pula-se o corpo do loop. Todo loop for pode ser escrito como um loop while e vice-versa.

O terceiro tipo de loop é o do-while. Um loop do-while é o único loop pós-teste disponível em JavaScript e é formado de duas partes, o corpo do loop e a condição pós-teste:

```
var i = 0;
do {
  // corpo do loop
} while (i++ < 10);
```

Em um loop do-while o corpo será sempre executado ao menos uma vez, sendo que a condição pós-teste servirá para determinar se o loop será executado novamente.

O quarto e último tipo de loop é o loop for-in. Ele tem um propósito muito especial: enumerar as propriedades nomeadas de qualquer objeto. O formato básico é o seguinte:

```
for (var prop in object) {
  // corpo do loop
}
```

Cada vez que o loop é executado, a variável prop é preenchida com o nome de outra propriedade (uma string) que existe no objeto até que todas as propriedades tenham sido retornadas. As propriedades retornadas são tanto as que existem na instância do objeto quanto as herdadas pela cadeia de protótipos.

Desempenho do loop

Uma fonte constante de debate acerca do desempenho dos loops é sobre qual deles deve ser preferido. Dos quatro oferecidos pelo JavaScript, apenas um é significativamente mais lento: o loop for-in.

Uma vez que cada iteração resulta em uma verificação de propriedade na instância ou em um protótipo, o loop for-in representa um custo de desempenho bem mais considerável, sendo mais lento do que os outros tipos de loops. Por esse motivo, é recomendado evitar o loop for-in a não ser que a intenção seja iterar sobre um número desconhecido de propriedades do objeto. Caso tenha um número conhecido e finito de propriedades sobre as quais iterar, prefira a utilização de outros tipos de loops, valendo-se de padrões como o seguinte:

```
var props = ["prop1", "prop2"],
  i = 0;

while (i < props.length) {
```

```
  process(object[props[i++]])
}
```

Esse código produz um array cujos membros são nomes de propriedades. O loop while é utilizado para iterar sobre esse pequeno número de propriedades e processar o membro apropriado em object. Em lugar de verificar cada propriedade em object, o código enfoca apenas as propriedades de interesse, poupando processamento e tempo.

 Nunca utilize o for-in para iterar sobre membros de um array.

À parte do for-in, todos os outros loops apresentam características de desempenho equivalentes, de modo que não é necessário determinar qual é o mais rápido. A escolha de um tipo de loop deve ter como base suas necessidades, e não preocupações de desempenho.

Se o tipo do loop não influencia seu desempenho, o que influencia? Há, na verdade, dois fatores:

- O trabalho feito a cada iteração.
- O número de iterações.

Ao reduzir qualquer um desses fatores, você pode afetar de modo positivo o desempenho do loop.

Diminuição do trabalho por iteração

Faz sentido pensarmos que se uma única passagem por um loop toma tempo, passagens múltiplas tomarão ainda mais tempo. Limitar o número de operações dispendiosas feitas no corpo do loop é uma boa maneira de acelerar o loop como um todo.

Um loop típico de processamento de arrays pode ser criado utilizando qualquer um dos três mais rápidos tipos de loops. O código será mais comumente escrito desta forma:

```
// loops originais
for (var i=0; i < items.length; i++) {
  process(items[i]);
}
```

```
var j=0;
while (j < items.length) {
  process(items[j++]]);
}
var k=0;
do {
  process(items[k++]);
} while (k < items.length);
```

Em cada um desses loops, há várias operações ocorrendo sempre que o corpo do loop é executado:

1. Uma verificação de propriedade (items.length) na condição de controle.
2. Uma comparação (i < items.length) na condição de controle.
3. Uma comparação verificando se a condição de controle é verdadeira (i < items.length == true).
4. Uma operação de incremento (i++).
5. Uma verificação ao array (items[i]).
6. Uma chamada à função (process(items[i])).

Há muita coisa acontecendo em cada iteração desses simples loops, ainda que não haja tanto código. A velocidade em que o código será executado é em grande parte determinada pelo que process() faz a cada item, mas, ainda assim, uma redução no número total de operações por iteração pode melhorar muito o desempenho geral do loop.

O primeiro passo para a otimização da quantidade de trabalho em um loop é a minimização do número de verificações a membros de objeto e itens de array. Como discutido no capítulo 2, essas verificações demoram mais para serem acessadas na maioria dos navegadores quando comparadas a variáveis locais ou valores literais. Os exemplos prévios realizam uma consulta (look up) de propriedade para items.length a cada vez que o loop é executado. Trata-se de um desperdício, visto que esse valor não sofrerá alteração durante a execução do loop, representando, justamente

por isso, um impacto desnecessário no desempenho. Podemos facilmente melhorar o desempenho do loop efetuando a verificação de propriedade apenas uma vez, armazenando seu valor em uma variável local e utilizando-a na condição de controle:

```
// minimizando consultas de propriedade
for (var i=0, len=items.length; i < len; i++) {
  process(items[i]);
}
var j=0,
  count = items.length;
while (j < count) {
  process(items[j++]]);
}
var k=0,
  num = items.length;
do {
  process(items[k++]);
} while (k < num);
```

Cada um desses loops reescritos faz apenas uma consulta de propriedade para o comprimento do array antes da execução do loop. Isso permite que a condição de controle seja composta apenas de variáveis locais e que, justamente por isso, execute muito mais rapidamente. Dependendo do comprimento do array, podemos poupar algo em torno de 25% do tempo de execução total do loop na maioria dos navegadores (e até 50% no Internet Explorer).

Também podemos melhorar o desempenho dos loops revertendo sua ordem. Com frequência, a ordem em que os itens do array são executados é irrelevante para a tarefa. Assim, partir do último item e processar em direção ao primeiro é uma alternativa aceitável. A reversão da ordem do loop é uma otimização de desempenho comum em linguagens de programação, mas geralmente não é bem compreendida. Em JavaScript, a reversão do loop resulta em um pequeno ganho de desempenho, desde que você elimine operações extras como resultado:

```
// minimizando as consultas de propriedade e realizando a reversão
for (var i=items.length; i--; ) {
  process(items[i]);
```

```
}
var j = items.length;
while (j--) {
  process(items[j]]);
}
var k = items.length-1;
do {
  process(items[k]);
} while (k--);
```

Os loops nesse exemplo são revertidos e combinam a condição de controle com a operação de incremento. Cada condição de controle representa agora uma simples comparação a zero. Condições de controle são comparadas ao valor true. Qualquer número diferente de zero é considerado true (verdadeiro), enquanto zero é considerado false (falso). Na realidade, a condição de controle foi alterada, passando de duas comparações (o iterador é menor que o total e igual a true?) para apenas uma comparação (o valor é verdadeiro?). A redução de duas comparações por iteração para apenas uma acelera o loop ainda mais. Ao reverter os loops e minimizar as consultas de propriedades, podemos atingir tempos de execução 50 a 60% mais rápidos que o original.

Como uma comparação ao original, veja a seguir as operações realizadas a cada iteração para esses loops:

1. Uma comparação (i == true) na condição de controle.
2. Uma operação de decremento (i--).
3. Uma verificação ao array (items[i]).
4. Uma chamada à função (process(items[i])).

O novo código do loop apresenta duas operações a menos por cada iteração, o que pode levar a melhorias de desempenho à medida que aumentam o número de iterações.

A diminuição do trabalho feito por cada iteração é mais eficiente quando o loop apresenta uma complexidade de tipo *O(n)*. Quando ele é mais complexo que *O(n)*, é aconselhável dar mais atenção à diminuição do número de iterações.

Diminuição do número de iterações

Até mesmo o mais rápido código no corpo de um loop representará um problema quando iterado milhares de vezes. Além disso, há um pequeno custo de desempenho associado à execução do corpo do loop, o que apenas acresce ao tempo de execução geral. A diminuição do número de iterações pelo loop pode conduzir a ganhos de desempenho. A abordagem mais comum à limitação de iterações é conhecida como *Dispositivo de Duff* (Duff's Device).

O Dispositivo de Duff é a técnica pela qual fazemos com que cada iteração faça o trabalho de várias. Jeff Greenberg é creditado com a primeira utilização publicada da técnica em JavaScript, partindo de sua implementação original em C. Uma implementação típica tem a seguinte aparência:

```
// crédito: Jeff Greenberg
var iterations = Math.floor(items.length / 8),
  startAt = items.length % 8,
  i = 0;

do {
  switch(startAt) {
    case 0: process(items[i++]);
    case 7: process(items[i++]);
    case 6: process(items[i++]);
    case 5: process(items[i++]);
    case 4: process(items[i++]);
    case 3: process(items[i++]);
    case 2: process(items[i++]);
    case 1: process(items[i++]);
  }
  startAt = 0;
} while(iterations--);
```

A ideia básica por trás da implementação do Dispositivo de Duff é a de que cada viagem pelo loop deve conter um máximo de oito chamadas a `process()`. O número de iterações pelo loop será determinado pela divisão do número total de itens por oito. Uma vez que nem todos os números podem ser divididos naturalmente por oito, a variável `startAt` abriga o resto dessa divisão e indica quantas chamadas a `process()` ocorrerão na primeira viagem pelo loop. Caso existam 12 itens, a primeira passagem pelo loop chamaria `process()`

quatro vezes, enquanto a segunda chamaria oito vezes. Teríamos um total de duas passagens pelo loop, em lugar de 12. Uma versão um pouco mais rápida desse algoritmo remove a instrução switch e separa o processamento do resto da divisão do processamento principal:

```
// crédito: Jeff Greenberg
var i = items.length % 8;
while(i) {
  process(items[i--]);
}
i = Math.floor(items.length / 8);
while(i) {
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
    process(items[i--]);
}
```

Ainda que essa implementação contenha agora dois loops em vez de um, ela executa mais rapidamente do que a original, devido à remoção da instrução switch do corpo do loop.

Decidir pela utilização ou não do Dispositivo de Duff, seja em sua versão original ou na modificada, dependerá muito do número de iterações que estivermos realizando. Em casos onde o número de iterações for menor que mil, provavelmente teremos um ganho de desempenho praticamente insignificante quando comparado à realização de um loop normal. Entretanto, à medida que aumenta o número de iterações, aumenta também a eficácia da técnica. Com 500 mil iterações, por exemplo, o tempo de execução chega a ser 70% mais rápido do que a execução por um loop regular.

Iteração com base em função

A quinta edição do ECMA-262 introduziu um novo método no objeto de array nativo chamado forEach(). Esse método itera sobre os

membros de um array e executa uma função em cada um deles. A função a ser executada em cada item é passada no forEach() como um argumento e recebe três argumentos quando chamada: o valor do item do array, o índice do item do array e o array em si. A seguir, temos um exemplo de sua utilização.

```
items.forEach(function(value, index, array) {
  process(value);
});
```

O método forEach() é implementado de modo nativo no Firefox, no Chrome e no Safari. Além disso, a maior parte das bibliotecas JavaScript apresenta seu equivalente lógico:

```
// YUI 3
Y.Array.each(items, function(value, index, array) {
  process(value);
});
// jQuery
jQuery.each(items, function(index, value) {
  process(value);
});
// Dojo
dojo.forEach(items, function(value, index, array) {
  process(value);
});
// Prototype
items.each(function(value, index) {
  process(value);
});
// MooTools
$each(items, function(value, index) {
  process(value);
});
```

Ainda que a iteração com base em função represente um método mais conveniente para realização de iterações, ela também é um pouco mais lenta do que a iteração com base em loop. Esse fato pode ser atribuído ao custo associado ao método extra que é chamado em cada item de array. De todas as formas, a iteração com base em função chega a ser até oito vezes mais longa do que uma iteração com base em loop e, justamente por isso, não é uma

abordagem conveniente quando o tempo de execução é uma consideração importante.

Condicionais

Semelhante em natureza aos loops, condicionais determinam como a execução flui pelo JavaScript. O argumento tradicional, presente na ciência da computação, quanto à escolha entre a utilização de instruções if-else ou switch, também se aplica em JavaScript. Uma vez que diferentes navegadores implementaram otimizações de controle de fluxo distintas, não é sempre evidente qual técnica deve ser preferida.

if-else versus switch

A teoria prevalente quanto à utilização do if-else *versus* o switch tem como base o número de condições que será testado: quanto maior o número de condições, mais indicada será a escolha do switch em lugar do if-else. Isso tipicamente se resume a qual código terá uma leitura mais fácil. O argumento é o de que instruções if-else são mais fáceis de serem lidas quando existem menos condições, enquanto instruções switch são mais fáceis de serem lidas quando há um grande número de condições. Considere o seguinte:

```
if (found) {
  // faça algo
} else {
  // faça algo diferente
}
switch(found) {
  case true:
    // faça algo
    break;
  default:
    // faça algo diferente
}
```

Ainda que esses dois pedaços de código executem a mesma tarefa, muitos argumentariam que a instrução if-else é de leitura muito mais fácil do que o switch. Entretanto, ao aumentar o número de

condições, nós normalmente invertemos essa opinião:

```
if (color == "red") {
  // faça algo
} else if (color == "blue") {
  // faça algo
} else if (color == "brown") {
  // faça algo
} else if (color == "black") {
  // faça algo
} else {
  // faça algo
}
```

```
switch (color) {
  case "red":
    // faça algo
    break;
  case "blue":
    // faça algo
    break;
  case "brown":
    // faça algo
    break;
  case "black":
    // faça algo
    break;
  default:
    // faça algo
}
```

A maioria das pessoas consideraria que a instrução switch desse código é de leitura mais fácil do que a instrução if-else.

Na realidade, a instrução switch é mais rápida na maioria dos casos quando comparada a if-else, sendo significativamente mais rápida quando é grande o número de condições. A principal diferença quanto ao desempenho das duas é a de que o custo incremental de uma condição adicional é maior em if-else do que em switch. Assim, nossa inclinação natural pela utilização de if-else quando houver um pequeno número de condições, e de switch para um grande número de condições, é também o conselho correto no que diz respeito ao desempenho.

Em termos gerais, if-else é preferível quando existem dois valores distintos para poucos intervalos de valores a serem testados. Por outro lado, quando existem mais do que dois valores distintos a serem testados, a instrução switch é a melhor escolha.

Otimização do if-else

Sempre que quisermos otimizar a instrução if-else, devemos minimizar o número de condições a serem avaliadas antes da escolha do caminho correto. Portanto, a otimização mais fácil é sempre a garantia de que as condições mais comuns venham primeiro. Considere o seguinte:

```
if (value < 5) {
  // faça algo
} else if (value > 5 && value < 10) {
  // faça algo
} else {
  // faça algo
}
```

Esse código é ideal apenas se value frequentemente tiver um valor menor que cinco. Se value é geralmente maior ou igual a dez, duas condições terão de ser avaliadas antes que o caminho correto seja tomado, aumentando o tempo médio gasto nessa instrução. As condições em uma instrução if-else devem sempre ser ordenadas a partir da mais provável em direção à menos provável, para garantir que o tempo de execução seja o mais rápido possível.

Outra abordagem para a minimização das avaliações de condição é a organização do if-else em uma série de instruções if-else aninhadas. A utilização de um único e grande if-else geralmente conduz a um tempo de execução mais lento, conforme cada condição adicional é avaliada. Por exemplo:

```
if (value == 0) {
  return result0;
} else if (value == 1) {
  return result1;
} else if (value == 2) {
  return result2;
```

```
} else if (value == 3) {
  return result3;
} else if (value == 4) {
  return result4;
} else if (value == 5) {
  return result5;
} else if (value == 6) {
  return result6;
} else if (value == 7) {
  return result7;
} else if (value == 8) {
  return result8;
} else if (value == 9) {
  return result9;
} else {
  return result10;
}
```

Com essa instrução if-else, o número máximo de condições a serem avaliadas é dez. Caso os valores possíveis estejam igualmente distribuídos entre zero e dez, o tempo médio de execução será desacelerado. Para minimizar o número de condições que deverão ser avaliadas, este código pode ser reescrito em uma série de instruções if-else aninhadas, da seguinte maneira:

```
if (value < 6) {
  if (value < 3) {
    if (value == 0) {
      return result0;
    } else if (value == 1) {
      return result1;
    } else {
      return result2;
    }
  } else {
    if (value == 3) {
      return result3;
    } else if (value == 4) {
      return result4;
    } else {
      return result5;
    }
  }
} else {
```

```
    if (value < 8) {
      if (value == 6) {
        return result6;
      } else {
        return result7;
      }
    } else {
      if (value == 8) {
        return result8;
      } else if (value == 9) {
        return result9;
      } else {
        return result10;
      }
    }
  }
```

Essa instrução if-else reescrita apresenta um número máximo de quatro avaliações de condição por execução. Isso é realizado pela aplicação de uma abordagem de tipo pesquisa-binária, dividindo os valores possíveis em uma série de intervalos a serem conferidos, seguidos pelo exame de cada uma dessas seções. Quando os valores estão distribuídos por igual entre zero e dez, o tempo médio gasto para a execução desse código é aproximadamente metade do necessário para a execução da instrução if-else prévia. Essa abordagem é indicada para situações em que existem intervalos de valores que devem ser testados (não quando existem valores distintos, caso em que uma instrução switch será mais apropriada).

Tabelas de consulta

Por vezes, a melhor abordagem às instruções condicionais é simplesmente evitarmos por completo sua utilização. Quando há um grande número de valores distintos que devem ser testados, tanto if-else quanto switch são significativamente mais lentos do que uma tabela de verificação. Tabelas de consulta (lookup tables) podem ser criadas utilizando arrays ou objetos regulares em JavaScript. O acesso a seus dados é muito mais rápido do que a utilização de if-else ou switch, especialmente se o número de condições for grande.

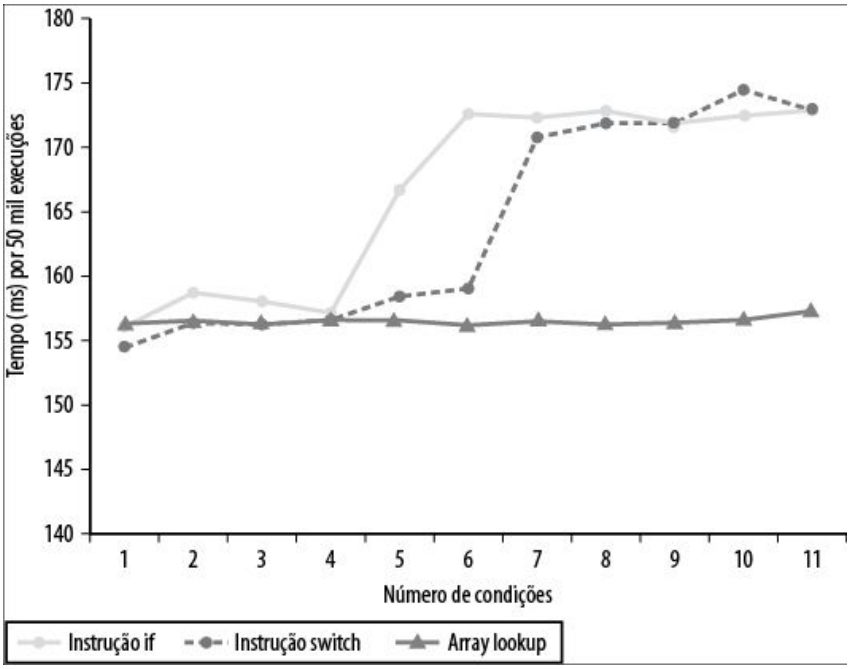


Figura 4.1 – Consulta de item de array versus a utilização de if-else ou switch no Internet Explorer 7.

Tabelas de consulta não apenas são mais rápidas quando comparadas a instruções if-else ou switch como também facilitam a leitura de seu código quando há um grande número de valores distintos que devem ser testados. Por exemplo, instruções switch costumam se tornar de difícil administração quando apresentam um tamanho extenso, como podemos ver a seguir:

```
switch(value) {
  case 0:
    return result0;
  case 1:
    return result1;
  case 2:
    return result2;
  case 3:
    return result3;
  case 4:
    return result4;
  case 5:
    return result5;
  case 6:
    return result6;
  case 7:
    return result7;
```

```
    case 8:
      return result8;
    case 9:
      return result9;
    default:
      return result10;
  }
```

A quantidade de espaço que essa instrução switch ocupa no código provavelmente não é proporcional à sua importância. Toda a estrutura pode ser substituída pelo uso de um array como tabela de consulta:

```
// defina o array de resultados
var results = [result0, result1, result2, result3, result4, result5, result6,
        result7, result8, result9, result10]

// retorne o valor correto
return results[value];
```

Ao utilizar uma tabela de consulta, todas as avaliações de condição são eliminadas por completo. A operação se torna uma consulta por item de array ou uma consulta por membro de objeto. Essa é uma grande vantagem das tabelas de consulta: uma vez que não existem condições que devem ser avaliadas, há pouco ou nenhum prejuízo adicional sobre o desempenho conforme aumenta o número de valores possíveis.

Tabelas de consulta são especialmente úteis quando há um mapeamento lógico entre uma única chave e um único valor (como no exemplo prévio). Uma instrução switch é mais apropriada quando cada chave demanda a execução de uma ação ou série de ações específicas.

Recursão

Algoritmos complexos são normalmente facilitados pelo uso de recursão. Na verdade, há até alguns algoritmos tradicionais que presumem a recursão como implementação, como uma função para retorno de fatoriais:

```
function factorial(n) {
  if (n == 0) {
```

```
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

O problema com função recursivas é que se houver uma condição de término mal definida ou ausente, teremos um tempo de execução mais longo que poderá até travar a interface do usuário. Além disso, funções recursivas têm maior probabilidade de se chocarem com os limites de tamanho para as pilhas de chamadas dos navegadores.

Limites da pilha de chamadas

A quantidade de recursão aceita por engines JavaScript varia, estando diretamente relacionada ao tamanho da pilha de chamadas do JavaScript. Com exceção do Internet Explorer, onde a pilha de chamadas está relacionada à memória disponível no sistema, todos os outros navegadores apresentam limites estáticos da pilha de chamadas. O tamanho da pilha de chamadas para os navegadores mais recentes é relativamente grande quando comparado aos navegadores mais antigos (o Safari 2, por exemplo, apresentava um limite de pilha de chamadas de 100). A figura 4.2 mostra os tamanhos das pilhas de chamadas nos principais navegadores.

Quando o tamanho máximo da pilha de chamadas é ultrapassado, introduzindo recursão excessiva, o navegador reportará um erro com alguma das seguintes mensagens:

- Internet Explorer: “Estouro de pilha na linha x”
- Firefox: “Too much recursion”
- Safari: “Maximum call stack size exceeded”
- Opera: “Abort (control stack overflow)”

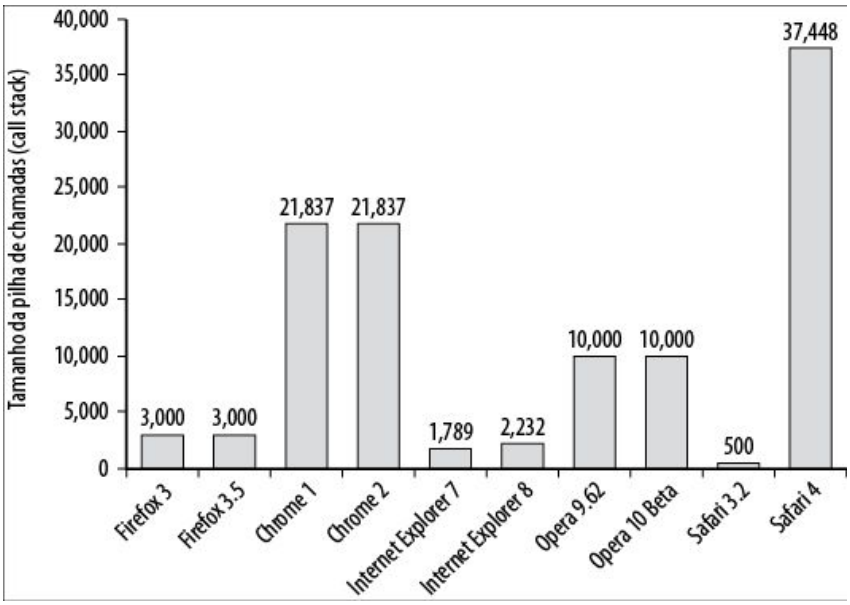


Figura 4.2 – Tamanho da pilha de chamadas nos navegadores.

O Chrome é o único navegador que não exibe uma mensagem ao usuário quando o tamanho-limite da pilha de chamadas é excedido.

Talvez a parte mais interessante dos erros de estouro de pilha é o fato de que, em alguns navegadores, eles são verdadeiramente erros de JavaScript, podendo, portanto, serem capturados pela utilização de uma instrução try-catch. O tipo de exceção varia de acordo com o navegador utilizado. No Firefox é um InternalError; no Safari e no Chrome, trata-se de um RangeError; no Internet Explorer, um tipo genérico de Error. (O Opera não retorna um erro; ele simplesmente interrompe a engine JavaScript). Isso faz com que possamos tratar desses erros diretamente a partir do JavaScript:

```
try {
  recurse();
} catch (ex) {
  alert("Too much recursion!");
}
```

Se não forem tratados, esses erros “borbulham” do mesmo modo que fariam outros tipos de erros (no Firefox, eles acabam nos consoles de erro e do Firebug; no Safari e no Chrome, eles surgem no console JavaScript), exceto no Internet Explorer. O IE não apenas exibe uma mensagem de erro como também uma caixa de diálogo, com a mesma aparência de um alerta, contendo a

mensagem de estouro da pilha.

 Ainda que seja possível capturar esses erros no JavaScript, isso não é recomendado. Nenhum script deve ser disponibilizado se apresentar o potencial de exceder o limite máximo da pilha de chamadas.

Padrões de recursão

Quando se atinge um limite de pilha de chamadas (call stack), sua primeira ação deve ser a identificação das instâncias de recursão do código. Para isso, há dois padrões recursivos para os quais devemos estar atentos. O primeiro é o padrão recursivo direto representado na função `factorial()` que mostramos anteriormente, quando uma função chama a si própria. O padrão geral é o seguinte:

```
function recurse() {
  recurse();
}
recurse();
```

Esse padrão é tipicamente mais fácil de identificar quando ocorrem erros. Um segundo padrão, mais sutil, envolve duas funções:

```
function first() {
  second();
}
function second() {
  first();
}
first();
```

Nesse padrão de recursão, duas funções chamam a si mesmas, formando um loop infinito. Esse é o padrão mais problemático e muito mais difícil de identificar em grandes bases de código.

A maior parte dos erros de pilha de chamadas está relacionada a um desses dois padrões de recursão. Uma causa frequente de estouro de pilha é uma condição terminal incorreta, de modo que o primeiro passo na identificação do padrão deve ser a validação da condição. Caso a condição terminal esteja correta, podemos concluir que o algoritmo contém recursão demais para que possa

ser executado com segurança no navegador. Nesses casos, devemos tentar o uso de iteração, de memoização ou de ambas estas técnicas.

Iteração

Qualquer algoritmo que possa ser implementado com recursão também pode ser implementado por iteração. Algoritmos iterativos tipicamente consistem em vários loops diferentes que realizam aspectos diversos do processo, introduzindo suas próprias considerações quanto ao desempenho. Ainda assim, o uso de loops otimizados em vez de funções recursivas de longa duração pode resultar em ganhos de desempenho advindos do menor custo que representam os loops quando comparados à execução de uma função.

Como exemplo, o algoritmo merge sort é frequentemente implementado por meio de recursão. Uma simples implementação em JavaScript do merge sort pode ser vista a seguir:

```
function merge(left, right) {
  var result = [];
  while (left.length > 0 && right.length > 0) {
    if (left[0] < right[0]) {
      result.push(left.shift());
    } else {
      result.push(right.shift());
    }
  }
  return result.concat(left).concat(right);
}
function mergeSort(items) {
  if (items.length == 1) {
    return items;
  }
  var middle = Math.floor(items.length / 2),
    left = items.slice(0, middle),
    right = items.slice(middle);
  return merge(mergeSort(left), mergeSort(right));
}
```

O código para esse merge sort é relativamente simples e direto, mas a função mergeSort() em si acaba sendo chamada com muita frequência. Um array de *n* itens chama mergeSort() *2 * n -1* vezes, o que significa que um array com mais de 1.500 itens causará um erro de estouro de pilha (stack overflow) no Firefox.

O fato de encontrar o erro de estouro de pilha não implica que todo o algoritmo tenha de ser alterado: significa simplesmente que a recursão não é a melhor implementação nesse caso. O algoritmo merge sort também pode ser implementado por iteração, da seguinte maneira:

```
// utiliza a mesma função mergeSort() do exemplo prévio
function mergeSort(items) {
  if (items.length == 1) {
    return items;
  }
  var work = [];
  for (var i=0, len=items.length; i < len; i++) {
    work.push([items[i]]);
  }
  work.push([]); // no caso de um número ímpar de itens
  for (var lim=len; lim > 1; lim = (lim+1)/2) {
    for (var j=0,k=0; k < lim; j++, k+=2) {
      work[j] = merge(work[k], work[k+1]);
    }
    work[j] = []; // no caso de um número ímpar de itens
  }
  return work[0];
}
```

Essa implementação de mergeSort() realiza o mesmo trabalho da função anterior sem usar a recursão. Ainda que a versão iterativa do merge sort possa ser um pouco mais lenta do que a opção recursiva, ela não representa o mesmo impacto sobre a pilha de chamadas. A alteração de algoritmos recursivos para iterativos é apenas uma das opções disponíveis para evitar erros de estouro de pilha.

Memoização

Evitar a realização excessiva de trabalho é a melhor técnica para otimizar o desempenho. Quanto menos trabalho seu código tiver de realizar, mais rápido ele será executado. Dentro desse raciocínio, também faz sentido evitar a realização de trabalhos repetitivos. A efetuação das mesmas tarefas repetidas vezes é sempre um desperdício de tempo de execução. Para prevenir a execução de trabalho repetitivo, a memoização (memoization) é uma das abordagens disponíveis. Ela atua valendo-se do armazenamento em cache de cálculos prévios para uso futuro, o que a torna uma técnica especialmente útil em algoritmos recursivos.

Quando funções recursivas são chamadas várias vezes durante a execução do código, é comum que haja muita duplicação de ações. A função factorial(), introduzida anteriormente em “Recursão”, é um ótimo exemplo de como ações podem ser repetidas várias vezes em funções recursivas. Considere o seguinte código:

```
var fact6 = factorial(6);
var fact5 = factorial(5);
var fact4 = factorial(4);
```

Esse código produz três fatoriais e faz com que a função factorial() seja chamada 18 vezes. Sua desvantagem é o fato de que todo o trabalho necessário é realizado logo na primeira linha. Uma vez que o fatorial de 6 é igual a 6 multiplicado pelo fatorial de 5, o fatorial de 5 está sendo calculado duas vezes. Para piorar, o fatorial de 4 está sendo calculado três vezes. Faz muito mais sentido salvar e reutilizar esses cálculos em vez de começar do zero sempre que a função for chamada.

Utilizando a memoização, podemos reescrever a função factorial() da seguinte maneira:

```
function memfactorial(n) {
  if (!memfactorial.cache) {
    memfactorial.cache = {
      "0": 1,
      "1": 1
    };
  }
```

```
  if (!memfactorial.cache.hasOwnProperty(n)) {
    memfactorial.cache[n] = n * memfactorial(n-1);
  }
  return memfactorial.cache[n];
}
```

A chave para essa versão memoizada da função fatorial é a criação de um objeto cache. Esse objeto é armazenado na função em si, sendo preenchido previamente com os dois fatoriais mais simples: 0 e 1. Antes de calcular um fatorial, esse cache é conferido para verificar se o cálculo já foi realizado. A inexistência de um valor no cache significa que o cálculo deve ser feito pela primeira vez, sendo seu resultado armazenado para uso futuro. Essa função é utilizada da mesma maneira que a `factorial()` original:

```
var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

Esse código retorna três fatoriais diferentes, mas realiza um total de oito chamadas a `memfactorial()`. Uma vez que todos os cálculos necessários são efetuados na primeira linha, as duas seguintes não têm de efetuar nenhuma recursão, já que valores armazenados em cache é que serão retornados.

O processo de memoização pode ser levemente diferente para cada função recursiva, mas em geral sempre se aplica o mesmo padrão. Para tornar mais fácil a memoização de uma função, podemos definir uma função `memoize()` que encapsula a funcionalidade básica que buscamos. Por exemplo:

```
function memoize(fundamental, cache) {
  cache = cache || {};
  var shell = function(arg) {
    if (!cache.hasOwnProperty(arg)) {
      cache[arg] = fundamental(arg);
    }
    return cache[arg];
  };
  return shell;
}
```

Essa função `memoize()` aceita dois argumentos: uma função a ser

memoizada e um objeto cache opcional. O objeto cache pode ser passado caso você queira preencher alguns valores antecipadamente: caso contrário, um novo objeto cache será criado. Uma função shell é então criada, envolvendo a original (fundamental) e garantindo que um novo resultado seja calculado apenas se nunca tiver sido calculado antes. Essa função shell é retornada para que possa ser chamada diretamente, da seguinte maneira:

```
// memoize a função factorial
var memfactorial = memoize(factorial, { "0": 1, "1": 1 });

// chame a nova função
var fact6 = memfactorial(6);
var fact5 = memfactorial(5);
var fact4 = memfactorial(4);
```

Uma memoização genérica desse tipo é menos adequada do que a atualização manual do algoritmo para uma dada função. Isso ocorre porque a função memoize() armazena em cache o resultado de uma chamada à função com argumentos específicos. Chamadas recursivas, portanto, são economizadas apenas quando a função shell é chamada várias vezes com esses mesmos argumentos. Por esse motivo, naquelas funções que apresentam problemas significativos de desempenho, é sempre preferível implementar manualmente a memoização em vez de aplicar uma solução genérica.

Resumo

Assim como em outras linguagens de programação, o modo como você fatora seu código e o algoritmo que utiliza são capazes de afetar o tempo de execução do JavaScript. Diferentemente de outras linguagens, o JavaScript tem um número restrito de recursos que podem ser utilizados, fazendo com que as técnicas de otimização sejam ainda mais importantes.

- Os loops for, while e do-while apresentam características de desempenho semelhantes. Isso faz com que nenhum deles seja significativamente mais rápido ou mais lento que os outros.

- Evite os loops for-in, a menos que precise iterar sobre propriedades de objetos desconhecidas.
- As melhores maneiras de melhorar o desempenho dos loops são a diminuição da quantidade de trabalho efetuado em cada iteração e a diminuição do número de iterações contidas nos loops.
- Em geral, o switch é sempre mais rápido que o if-else. Ainda assim, ele nem sempre é a melhor solução.
- Tabelas de consulta são uma alternativa mais veloz que o uso de if-else ou switch para a execução de várias avaliações de condição.
- O tamanho da pilha de chamadas do navegador limita o volume de recursão que o JavaScript pode realizar; erros de estouro de pilha impedem a execução do restante do código.
- Caso encontre um erro de estouro de pilha, altere o método para um algoritmo iterativo ou utilize a memoização para evitar a ocorrência de ações repetitivas.

Quanto mais código estiver sendo executado, maior será o ganho de desempenho advindo da adoção dessas estratégias.

CAPÍTULO 5

Strings e expressões regulares

Steven Levithan

Praticamente todos os programas em JavaScript estão fortemente ligados às strings. Como exemplo, muitas aplicações usam Ajax para recuperar strings de um servidor, convertê-las em objetos JavaScript mais facilmente utilizáveis e gerar strings de HTML a partir dos dados. Um programa típico lida com inúmeras tarefas como essas. Elas normalmente demandam a junção (merge), repartição (split), reorganização, busca, iteração ou a simples manipulação das strings; da mesma maneira, conforme as aplicações web se tornam mais complexas, uma parte cada vez maior desse processamento é feita no navegador.

Em JavaScript, expressões regulares são essenciais para qualquer coisa que esteja além do processamento trivial de strings. Grande parte deste capítulo será dedicada a auxiliar você a compreender como engines[1] de expressões regulares processam internamente suas strings, bem como a mostrar como é possível escrever expressões regulares que se beneficiam desse conhecimento.

Uma vez que o termo expressão regular é um pouco incômodo, a expressão regex (do inglês regular expression) é preferida como abreviação, sendo que regexes denota seu plural.

Também neste capítulo você aprenderá os mais rápidos métodos, compatíveis com vários navegadores, para concatenação e aparo de suas strings, descobrirá como incrementar o desempenho de suas expressões regulares pela redução do backtracking[2], e terá acesso a muitas outras dicas e truques para o processamento eficiente de suas strings e expressões regulares.

Concatenação de strings

A concatenação de strings pode influenciar de modo surpreendentemente intenso o desempenho de seu código. É prática comum construir uma string com adições contínuas sendo feitas no seu final em um loop (p. ex., quando criamos uma tabela HTML ou um documento XML), mas esse tipo de processamento é conhecido por seu pobre desempenho em alguns navegadores.

Então, como podemos otimizar esse tipo de tarefa? Para começar, há mais de uma maneira de fazer o merge de strings (veja a tabela 5.1).

Tabela 5.1 – Métodos de concatenação de strings

Método	Exemplo
Operador +	str = “a” + “b” + “c”;
Operador +=	str = “a”;
	str += “b”;
	str += “c”;
array.join()	str = [“a”, “b”, “c”].join(“”);
string.concat()	str = “a”;
	str = str.concat(“b”, “c”);

Todos esses métodos são rápidos na concatenação de poucas strings, sendo ideais para utilização casual. Basta escolher qual mais se identifica com suas necessidades. Entretanto, conforme aumenta o comprimento e o número das strings que devem ser fundidas (merged), alguns métodos começam a se destacar.

Operadores de adição (+) e de atribuição por adição (+=)

Esses operadores oferecem o método mais simples para a concatenação de strings e, na verdade, todos os navegadores modernos, com exceção do IE7 e de suas versões anteriores, são capazes de otimizá-los com a eficiência necessária para que você não tenha de examinar outras opções. Mesmo assim existem várias técnicas que são capazes de maximizar sua eficiência.

Primeiro, um exemplo. Veja a seguir um modo comum de designarmos uma string concatenada:

```
str += "one" + "two";
```

Ao avaliar esse código, quatro passos serão dados:

1. Uma string temporária será criada na memória.
2. O valor concatenado “onetwo” será designado à string temporária.
3. A string temporária será concatenada com o valor atual de `str`.
4. O resultado será designado a `str`.

Essa é apenas uma aproximação da forma como os navegadores implementam essa tarefa, mas é próxima o bastante para dar uma ideia do que ocorre.

O código seguinte evita a string temporária (passos 1 e 2 da lista) anexando diretamente à str por meio de duas instruções distintas. Isso acaba sendo executado com uma velocidade que é 10 a 40% mais rápida, na maioria dos navegadores:

```
str += "one";
str += "two";
```

Você pode receber a mesma melhora de desempenho utilizando apenas uma instrução, da seguinte maneira:

```
str = str + "one" + "two";
// equivalente a str = ((str + "one") + "two")
```

Uma vez que a expressão de atribuição parte de str como base e anexa uma string por vez, evita-se a string temporária. Cada concatenação intermediária é executada da esquerda para a direita. Se a concatenação for executada em uma ordem diferente (p. ex., str = "one" + str + "two"), você perderá essa otimização. Isso ocorre pela forma como os navegadores alocam a memória quando fazem a junção de strings. À parte do IE, os navegadores buscam expandir a alocação de memória para a string à esquerda de uma expressão e simplesmente copiar a segunda string ao seu final (veja a figura 5.1). Em um loop, ao colocar a string base na posição mais distante à esquerda, você pode evitar a cópia repetida de uma string base

progressivamente maior.

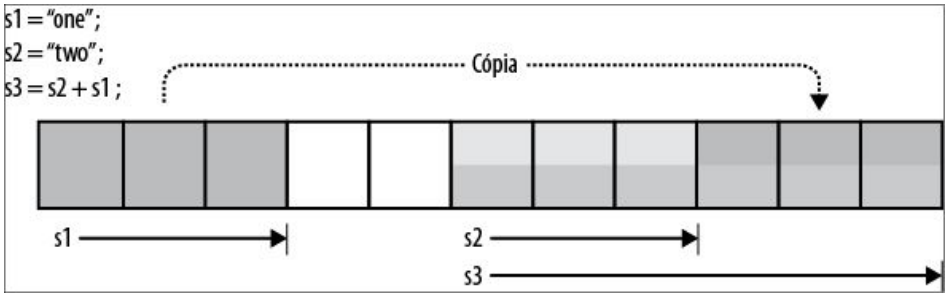


Figura 5.1 – Exemplo da utilização da memória quando se realiza a concatenação de strings: s1 é copiada ao final de s2 para criar s3; a string de base s2 não é copiada.

Essas técnicas não se aplicam ao IE. Elas têm pouco, ou nenhum, efeito no IE8 e podem até tornar as coisas mais lentas no IE7 ou em versões anteriores. Isso se deve ao fato de que o IE oculta a execução da concatenação. Na implementação do IE8, a concatenação de strings simplesmente armazena referências a partes existentes de strings que compõem a nova. No último momento possível (quando você realmente utiliza a string concatenada), as partes da string são copiadas em uma nova string "real" que, por sua vez, substitui as referências previamente armazenadas. Graças a isso, essa montagem não tem de ser efetuada sempre que a string for utilizada.


A implementação do IE8 pode enganar marcadores de desempenho (benchmarks) sintéticos – fazendo com que a concatenação pareça mais rápida do que realmente é – a menos que você obrigue a concatenação a ocorrer depois do término da construção de sua string teste. Por exemplo, poderíamos fazê-lo por meio de uma chamada ao método toString() em sua string final, pela verificação de sua propriedade length ou pela inserção dela dentro do DOM.

O IE7 e as versões anteriores utilizam uma implementação inferior da concatenação na qual cada par de strings concatenadas deve sempre ser copiado a um novo local da memória. Você presenciará o impacto potencialmente drástico disso na seção que veremos adiante, chamada "Junção de arrays". Com a implementação pré-IE8, o conselho desta seção pode tornar as coisas mais lentas, visto que é mais rápido primeiro concatenar strings mais curtas, antes de fundi-las em uma grande string base (evitando assim a necessidade da cópia repetitiva de uma string maior). Com largeStr = largeStr + s1 + s2,

por exemplo, o IE7 e as versões anteriores têm de copiar a string grande (largeStr) duas vezes, primeiro para fundi-la com s1 e depois para fundi-la com s2. De modo oposto, largeStr += s1 + s2 primeiro funde as duas strings menores e depois concatena o resultado com a string maior. A criação da string intermediária, s1 + s2, tem um impacto muito mais leve sobre o desempenho do que a cópia por duas vezes da string maior.

O Firefox e a fusão no tempo de compilação

Sempre que todas as strings concatenadas em uma expressão de atribuição forem constantes do tempo de compilação, o Firefox será capaz de fundi-las automaticamente no momento da compilação. Aqui está um modo de ver isso em ação:

```
function foldingDemo() {
  var str = "compile" + "time" + "folding";
  str += "this" + "works" + "too";
  str = str + "but" + "not" + "this";
}
alert(foldingDemo.toString());
/* No Firefox, você verá isso:
function foldingDemo() {
  var str = "compiletimefolding";
  str += "thisworkstoo";
  str = str + "but" + "not" + "this";
} */
```

Quando strings são unidas dessa forma, não há strings intermediárias no tempo de execução, fazendo com que o tempo e a memória que seriam gastos em sua concatenação sejam reduzidos a zero. É ótimo quando isso ocorre, mas não é produtivo na maioria dos casos, já que é muito mais comum construirmos strings a partir de dados no tempo de execução do que de constantes no tempo de compilação.

 O YUI Compressor realiza essa otimização no momento da construção. Consulte “Minificação do JavaScript” para obter mais informações sobre essa ferramenta.

Junção de arrays

O método Array.prototype.join funde todos os elementos de um array em uma string, aceitando uma string separadora que pode ser inserida entre cada elemento. Ao passar uma string vazia como separadora, você pode realizar facilmente a concatenação de todos os elementos de um array.

A junção de arrays é, na maioria dos navegadores, mais lenta do que os outros métodos de concatenação, mas isso é mais do que compensado pelo fato de que se trata do único método eficiente para concatenação de um grande número de strings no IE7 e em suas versões anteriores.

O exemplo seguinte de código demonstra o tipo de problema de desempenho que a junção de arrays pode solucionar:

```
var str = "I'm a thirty-five character string.",
  newStr = "",
  appends = 5000;
while (appends--) {
  newStr += str;
}
```

Esse código concatena 5 mil strings de 35 caracteres. A figura 5.2[3] mostra quanto tempo leva para que esse teste seja realizado no IE7, começando com 5 mil concatenações e aumentando gradualmente esse número.

O ingênuo algoritmo de concatenação apresentado pelo IE7 exige que o navegador repetidamente copie e aloque memória a strings progressivamente maiores cada vez que o loop é efetuado. O resultado é um crescimento exponencial do tempo de execução e do consumo de memória.

A boa notícia é que todos os outros navegadores modernos (incluindo o IE8) têm um desempenho bem melhor nesse teste, sem exibir a complexidade exponencial que representa o verdadeiro problema. Mesmo assim, fica evidente o impacto que uma concatenação aparentemente simples de strings pode ter; até mesmo o gasto de 226 milissegundos para a realização de 5 mil concatenações representa uma perda significativa de desempenho. Por sua vez, travar o navegador por mais de 32 segundos para

realizar a concatenação de 20 mil strings curtas é inaceitável em praticamente todas as aplicações.

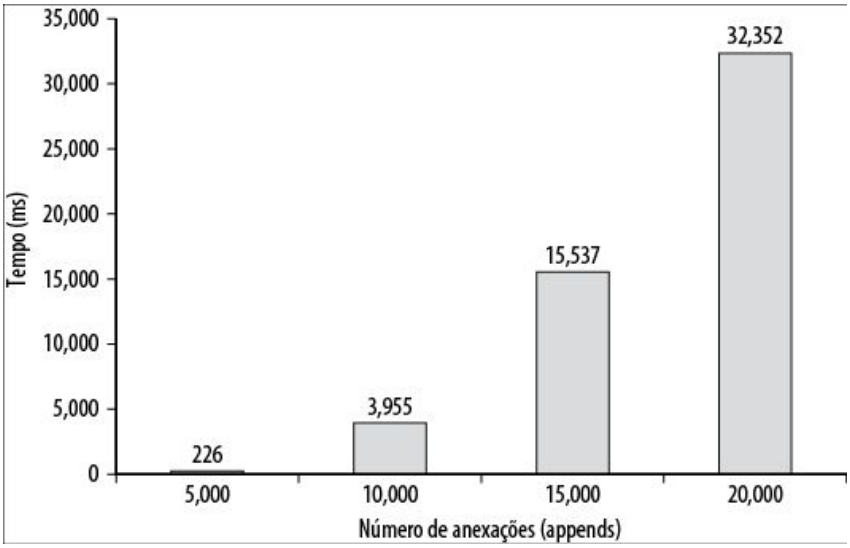


Figura 5.2 – Tempo gasto na concatenação de strings utilizando += no IE7.

Agora, considere o teste a seguir. A mesma string será gerada utilizando a junção de arrays:

```
var str = "I'm a thirty-five character string.",
  strs = [],
  newStr,
  appends = 5000;
while (appends--) {
  strs[strs.length] = str;
}
newStr = strs.join("");
```

A figura 5.3 mostra o tempo de execução desse teste no IE7.

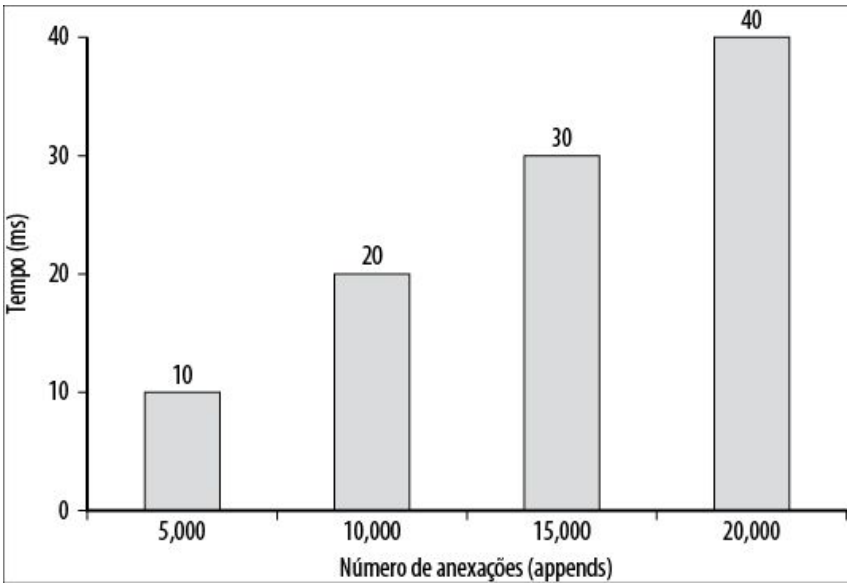


Figura 5.3 – Tempo gasto na concatenação de strings utilizando a junção de arrays no IE7.

Ao evitarmos a alocação repetida de memória e a cópia de strings cada vez maiores, alcançamos ganhos de desempenho drásticos. Quando se faz a junção de um array, o navegador aloca memória suficiente para abrigar a string completa, nunca copiando a mesma parte da string final mais que uma única vez.

String.prototype.concat

O método nativo para strings concat aceita vários argumentos e anexa cada um deles à string sobre a qual é chamado. Essa é a forma mais flexível de concatenar strings, uma vez que pode ser utilizada para anexar uma única string, várias delas ou até um array inteiro de strings.

```
// anexa uma string
str = str.concat(s1);

// anexa três strings
str = str.concat(s1, s2, s3);

// anexa cada string em um array utilizando o array
// como a lista de argumentos
str = String.prototype.concat.apply(str, array);
```

Infelizmente, o concat é, na maioria dos casos, um pouco mais lento do que os operadores simples + e +=, podendo ser substancialmente mais lento no IE, no Opera e no Chrome. Além

disso, ainda que sua utilização para junção (merge) de todas as strings possa parecer semelhante à abordagem de junção discutida anteriormente, ela é geralmente mais lenta (exceto no Opera), sofrendo os mesmos problemas de desempenho potencialmente catastróficos advindos do uso do `+` e do `+=` para criação de strings maiores no IE7 e em suas versões anteriores.

Otimização de expressões regulares

Expressões regulares criadas com desatenção podem representar um grande gargalo de desempenho (a seção que veremos mais à frente, chamada “Backtracking desenfreado”, contém diversos exemplos mostrando a gravidade que isso pode ter), mas muito pode ser feito para melhorar a eficiência de suas regexes. O simples fato de duas expressões regulares produzirem o mesmo texto não significa que elas o farão com a mesma velocidade.

Muitos fatores afetam a eficiência de uma expressão regular. Para começar, o texto ao qual ela se aplica tem grande influência sobre seu desempenho, já que expressões regulares gastam mais tempo em correspondências parciais do que em expressões de não correspondência óbvia. A engine de regex presente em cada navegador também traz consigo otimizações internas diferentes[4].

A otimização de expressões regulares é um tópico consideravelmente grande e diversificado. Há um limite quanto ao que pode ser discutido nesta seção, mas tudo que incluímos poderá ajudá-lo a compreender melhor o tipo de problema que afeta o desempenho de suas expressões regulares, assim como capacitá-lo a dominar a arte da criação de expressões regulares eficientes.

Observe que essa seção presume que você já tenha alguma experiência com expressões regulares e que esteja mais interessado em como torná-las mais rápidas. Caso o tópico seja novidade para você ou seja necessário rever os fundamentos do assunto, há inúmeras fontes disponíveis tanto na web quanto em material impresso. O livro *Regular Expression Cookbook* (Editora O’Reilly), de Jan Goyvaerts e Steven Levithan (eu mesmo!), foi

escrito para pessoas que gostam de aprender fazendo. O livro cobre igualmente JavaScript e outras linguagens de programação.

Como funcionam as expressões regulares

Para utilizar expressões regulares de modo eficiente, é importante compreender como elas funcionam. A seguir temos uma síntese dos passos básicos realizados por uma regex:

Passo 1: Compilação

Quando você cria um objeto regex (utilizando uma expressão regular literal ou o construtor `RegExp`), o navegador confere seu padrão em busca de erros, convertendo-o em uma rotina nativa que pode ser utilizada para realização das correspondências. Caso atribuia sua expressão regular a uma variável, você evita a necessidade de realizar esse passo mais de uma vez para um dado padrão.

Passo 2: Definição do ponto de partida

Quando uma expressão regular é utilizada, o primeiro passo é a determinação da posição na string-alvo onde a busca deverá ter início. Normalmente essa posição é no início da string ou no local determinado pela propriedade regex `lastIndex`[5], mas quando se retorna do passo 4 (devido a uma tentativa de correspondência sem sucesso), a posição será determinada sempre um caractere depois de onde teve início a última tentativa.

Otimizações incluídas pelos criadores dos navegadores em suas engines de expressões regulares podem ajudar a evitar muito trabalho desnecessário nesse estágio, decidindo com antecipação que certas ações poderão ser evitadas. Por exemplo, caso uma regex comece com `^`, o IE e o Chrome são capazes de determinar que uma correspondência não poderá ser encontrada depois do início da string, evitando a busca desnecessária em posições subsequentes. Outro exemplo ocorre quando todas as correspondências possíveis apresentam `x` como terceiro caractere. Uma implementação inteligente pode ser

capaz de determinar esse fato, buscar rapidamente pelo próximo `x` e definir o ponto de partida dois caracteres para trás da posição encontrada (p. ex., versões recentes do Chrome incluem essa otimização).

Passo 3: Correspondência de cada símbolo da regex

Uma vez que saiba por onde começar, a expressão regular percorre o texto e o padrão determinado. Quando não ocorre a correspondência de um símbolo em particular, a regex tenta retornar a um ponto anterior e seguir outros caminhos possíveis.

Passo 4: Sucesso ou fracasso

Caso uma correspondência completa seja encontrada na posição atual da string, a expressão regular declara um sucesso. Se todos os caminhos possíveis já tiverem sido tentados, mas uma correspondência não tiver sido encontrada, a engine retornará para o passo 2 e tentará novamente partindo do próximo caractere da string. Apenas depois do término desse ciclo para cada caractere da string (assim como para a posição depois do último caractere) e depois que nenhuma correspondência tiver sido encontrada é que a regex declarará um fracasso geral.

Ter sempre em mente esse processo poderá ajudá-lo a tomar decisões inteligentes sobre os tipos de problemas que afetam o desempenho das expressões regulares. Seguindo em frente, examinaremos com maior cuidado um atributo essencial do processo de correspondência do passo 3: *backtracking.*

Compreensão do backtracking

Na maioria das implementações modernas de expressões regulares (incluindo as necessárias em JavaScript), o backtracking é um componente fundamental do processo de correspondência. Também é parte importante do que torna as expressões regulares tão expressivas e poderosas. Ainda assim, é uma prática dispendiosa em termos computacionais e pode facilmente fugir de controle caso você não seja cuidadoso. Ainda que o backtracking seja apenas

parte da equação geral do desempenho, sua compreensão, assim como o entendimento de como podemos minimizar sua utilização, é talvez o ponto mais importante para a criação de expressões regulares eficientes. As seções seguintes cobrem detalhadamente esse tópico.

Conforme uma regex percorre uma string-alvo, ela testa a existência de correspondências em cada posição passando pelos componentes da expressão da esquerda para a direita. Para cada quantificador e alternação[6], uma decisão tem de ser tomada sobre como ela deve proceder. Na presença de quantificadores (como `*`, `+?` ou `{2,}`), a expressão regular tem de decidir quando deve tentar corresponder caracteres adicionais. Na presença de alternações (pelo operador `|`), ela deve tentar uma das opções disponíveis.

Cada vez que a regex toma tal decisão, ela se recorda das outras opções para as quais pode retornar depois, se necessário for. Caso a opção escolhida tenha sucesso, a regex percorre o padrão. Se também houver concordância com o restante da expressão, a correspondência estará completa. Caso a opção escolhida não encontre uma correspondência ou qualquer outra operação na expressão regular falhe, ela recua ao último ponto de decisão onde existem opções ainda não testadas e escolhe uma delas. Isso se repete até que uma correspondência seja encontrada ou até que todas as permutações possíveis de opções de quantificadores e alternações tenham fracassado. Nesse ponto, ela desiste e reinicia o processo para o próximo caractere da string.

Alternação e backtracking

A seguir há um exemplo que demonstra como esse processo é executado com alternação.

```
/h(ello|appy) hippo/.test("hello there, happy hippo");
```

Essa expressão regular realiza testes de correspondência para “hello hippo” ou “happy hippo”. Ela começa o teste buscando um `h`, encontrado imediatamente no primeiro caractere da string-alvo. A seguir, a subexpressão (`ello|appy`) oferece dois caminhos a serem

seguidos. A regex escolhe a opção à esquerda (alternações sempre atuam da esquerda para a direita) e checa se ello corresponde aos caracteres seguintes da string. De fato correspondem, e a regex também é capaz de identificar o caractere de espaço que vem em sequência. Nesse ponto, entretanto, ela atinge um beco sem saída, já que o h em hippo não corresponde ao t que vem a seguir na string. A expressão regular não desiste, pois ainda não tentou todas as opções disponíveis. Ela recua para o último ponto de decisão (logo após a correspondência do h inicial) e tenta casar a segunda opção de alternação. Isso também não dá certo e, uma vez que não existem mais opções, a regex determina que uma correspondência não pôde ser encontrada partindo do primeiro caractere da string. Seu próximo passo é avançar para uma tentativa a partir do segundo caractere. Lá, ela não encontra um h, o que faz com que continue sua busca até que alcance o 14º caractere, onde confere o h em "happy". Repetem-se as alternativas. Dessa vez, ello não combina com o texto da string, mas depois de recuar e tentar a segunda alternativa, ela é capaz de continuar até que corresponda a string completa "happy hippo" (veja a figura 5.4). Sucesso.

Figura 5.4 – Exemplo de backtracking com alternação.

Repetição e backtracking

O próximo exemplo mostra como funciona o backtracking com quantificadores de repetição.

```
var str = "<p>Para 1.</p>" +
  "<img src='smiley.jpg'>" +
  "<p>Para 2.</p>" +
  "<div>Div.</div>";
/<p>.*<\/p>/i.test(str);
```

Aqui, a regex começa comparando os três caracteres literais `<p>` no início da string. Em seguida vem `.*`. O ponto corresponde a qualquer caractere excetuando-se quebras de linhas, e o quantificador "guloso" asterisco realiza a repetição por zero ou mais vezes – o maior número de vezes possível. Uma vez que não existem quebras de linhas na string, abrange-se a string inteira. Entretanto, ainda há mais para ser conferido no padrão da expressão regular. Por isso, a regex busca por correspondências a `<`. Ela não obtém sucesso até o final da string, o que faz com que recue um caractere por vez, continuamente buscando correspondências até retornar ao caractere `<`, presente no início da tag `</div>`. Segue-se uma busca pela correspondência de `\` (uma barra invertida escapada), com sucesso, e de `p`, com fracasso. Mais uma vez a regex recua, repetindo o processo até que finalmente encontra o `</p>` ao final do segundo parágrafo. A correspondência retorna com sucesso, desde o início do primeiro parágrafo, até o término do último. Esse provavelmente não era o resultado desejado. Podemos alterar a regex fazendo com que ela confira parágrafos isolados, substituindo o quantificador guloso `*` pelo "preguiçoso" (não-guloso) `*?`. O recuo em quantificadores desse tipo funciona de modo oposto. Quando a expressão regular `/<p>.*?<\/p>/` alcança `.*?`, ela primeiro tenta simplesmente pulá-lo e avança para a correspondência de `<\/p>`. Isso ocorre porque `*?` repete seu elemento precedente por zero ou mais vezes, sempre o mínimo de vezes possível. O menor número possível sempre é zero. Entretanto, quando a opção de seguir `<` falha nesse ponto da string, a regex recua e tenta corresponder o menor número seguinte de caracteres: um. Ela continua a recuar dessa forma até que o `<\/p>` que acompanha o quantificador encontra

uma correspondência completa ao final do primeiro parágrafo.

Podemos ver que mesmo que haja apenas um parágrafo na string-alvo, as versões .* e .*? dessa expressão regular realizarão seu trabalho de modo diferente (veja a figura 5.5).

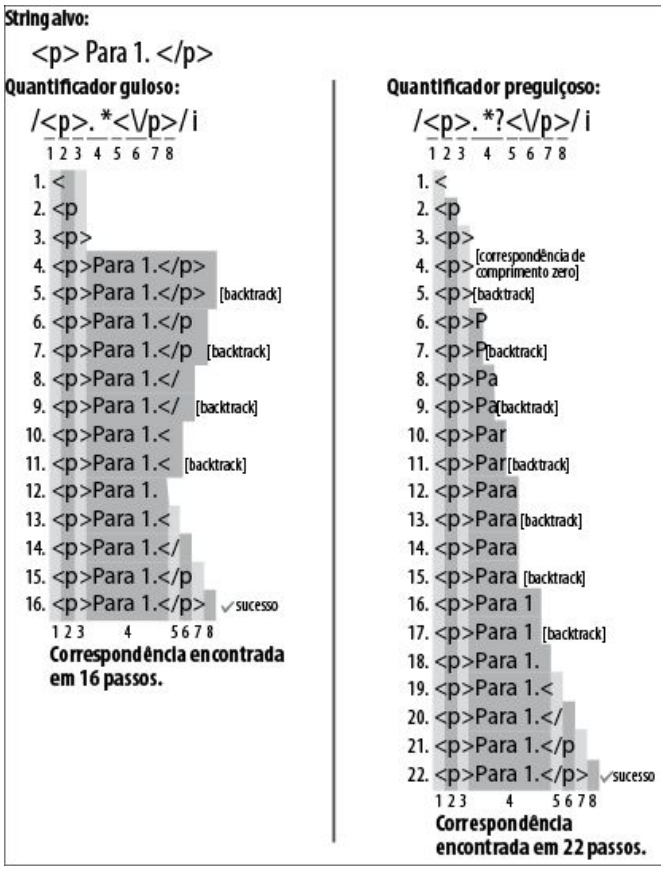


Figura 5.5 – Exemplo de recuo com quantificadores . e .*?.*

Backtracking desenfreado

Quando uma expressão regular paralisa seu navegador por segundos ou minutos, o problema geralmente se resume a um caso de backtracking desenfreado. Para demonstrar uma dessas situações, considere a seguinte regex, criada para buscar a correspondência de um arquivo HTML inteiro. Foi quebrada em duas linhas para adequação à página. Diferentemente da maior parte das outras regexes, o JavaScript não apresenta uma opção que corresponde pontos a quaisquer caracteres, incluindo quebras de linha. Dessa forma, esse exemplo utiliza [\s\S] para significar a

correspondência a qualquer caractere.

```
/<html>[\s\S]*?<head>[\s\S]*?<title>[\s\S]*?<\/title>[\s\S]*?<\/head>
[\s\S]*?<body>[\s\S]*?<\/body>[\s\S]*?<\/html>/
```

Essa expressão regular funciona bem quando se busca a correspondência de uma string HTML adequada, mas pode resultar em sérios problemas quando faltam uma ou mais tags necessárias na string. Caso a tag `</html>` esteja ausente, o último `[\s\S]*?` expandirá até o final da string, onde também não encontrará nenhuma tag `</html>`. Nesse ponto, em vez de desistir, a regex vê que as sequências prévias de `[\s\S]*?` recordam posições de backtracking que permitem uma expansão ainda maior. Ela tenta então expandir o penúltimo `[\s\S]*?` – utilizando-o para corresponder à tag `<\body>`, anteriormente casada com o padrão literal `<\/body>` na regex – e continua a fazê-lo em busca de uma segunda tag desse tipo até que o final da string seja novamente atingido. Quando nada disso dá certo, será a vez do antepenúltimo `[\s\S]*?` expandir até o fim, e assim por diante.

A solução: seja específico

Uma forma de evitar um problema como esse é ser bem mais específico quando definir quais caracteres podem ser comparados entre os delimitadores necessários. Tome como exemplo o padrão `".*?"`. Ele serve para buscar a correspondência de uma string delimitada por aspas duplas. Ao substituirmos o operador `.*?`, que é exageradamente permissivo, por um mais específico, como o `[^"\r\n]*`, eliminamos a possibilidade de que recuos forcem o ponto a casar com as aspas duplas e a expandir para além dos limites desejados.

Já no exemplo com HTML, não podemos contornar com tanta facilidade essa situação. Não se pode utilizar uma classe de caractere negado como `[^<]` no lugar de `[\s\S]`, já que podem existir outras tags entre as que estamos buscando. Ainda assim, podemos reproduzir o efeito repetindo um grupo de não captura que contenha um lookahead negativo (bloqueando a próxima tag necessária) e a metassequência `[\s\S]` (para qualquer caractere). Isso garante que as

tags que estamos buscando falhem em todas as posições intermediárias. Além disso, e o que é ainda mais importante, essa prática também impede que os padrões [\s\S] se expandam para além das tags bloqueadas. Confira a aparência da adoção dessa abordagem:

```
/<html>(?:(?!<head>)[\s\S])*<head>(?:(?!<title>)[\s\S])*<title>
(?:(?!<\/title>)[\s\S])*<\/title>(?:(?!<\/head>)[\s\S])*<\/head>
(?:(?!<body>)[\s\S])*<body>(?:(?!<\/body>)[\s\S])*<\/body>
(?:(?!<\/html>)[\s\S])*<\/html>/
```

Ainda que essa prática remova o potencial de backtracking desenfreado e permita que a *regex* falhe em casar strings HTML incompletas em tempo linear, ela não representa nenhuma espécie de grande aumento de eficiência. A repetição de um lookahead para cada caractere casado dessa forma é considerada ineficiente e reduz de modo significativo a realização de correspondências de sucesso. Essa abordagem funciona bem quando na busca de correspondência de pequenas strings, mas uma vez que nesse caso os lookaheads podem ter de ser testados milhares de vezes para que haja uma correspondência com um arquivo HTML, outra solução funciona melhor. Ela depende de um pequeno truque mostrado a seguir.

Emulação de grupos atômicos com lookaheads e referências anteriores

Algumas espécies de expressões regulares, incluindo as do tipo .NET, Java, Oniguruma, PCRE e Perl, aceitam um atributo chamado *agrupamento atômico*. Grupos atômicos – escritos como (?>...), onde a elipse representa qualquer padrão de regex – são grupos de não captura com algo especial. Assim que a expressão regular sai de um grupo atômico, todas as posições de backtracking dentro do grupo são eliminadas. Isso oferece uma solução muito melhor ao problema de backtracking das expressões regulares em HTML: caso você venha a posicionar cada sequência [\s\S]*? e a tag HTML seguinte juntas dentro de um grupo atômico, toda vez que uma das tags HTML necessárias for encontrada a correspondência

alcançada será aprisionada. Se uma parte futura da expressão regular não encontrar correspondência, nenhuma posição de backtracking será lembrada para os quantificadores dentro dos grupos atômicos, fazendo com que as sequências `[\s\S]*?` não possam expandir para além do que já conseguiram corresponder.

Isso é ótimo, mas o JavaScript não aceita grupos atômicos nem oferece qualquer outro tipo de função capaz de eliminar backtracking desnecessário. Ainda assim, somos capazes de emular grupos atômicos tirando proveito de um comportamento pouco conhecido dos *lookaheads*: o fato de que eles mesmos são grupos atômicos[7]. A diferença é que *lookaheads* não consomem caracteres como parte da correspondência geral: simplesmente conferem se o padrão que contêm pode ser casado àquela posição. Entretanto, podemos contornar esse fato envolvendo o padrão de um *lookahead* dentro de um grupo de captura e adicionando a ele uma referência anterior, fora do *lookahead*. Esta seria a aparência do resultado:

```
(?=(pattern to make atomic))\1
```

Essa construção pode ser reutilizada em qualquer padrão no qual você deseje utilizar um grupo atômico. Apenas não se esqueça de que é preciso utilizar o número apropriado da referência anterior caso sua expressão regular contenha mais do que um grupo de captura.

Confira a aparência que isso teria quando aplicado à expressão regular HTML:

```
/<html>(?=([\s\S]*?<head>))\1(?=([\s\S]*?<title>))\2(?=([\s\S]*?
<\/title>))\3(?=([\s\S]*?<\/head>))\4(?=([\s\S]*?<body>))\5
(?=([\s\S]*?<\/body>))\6[\s\S]*?<\/html>/
```

Agora, caso não haja uma tag `</html>` delimitadora e o último `[\s\S]*?` puder expandir até o fim da string, a expressão regular falhará imediatamente. Isso ocorre porque não existem pontos de backtracking aos quais ela pode retornar. Cada vez que a regex encontra uma tag intermediária e sai de um lookahead, ela elimina todas as posições de backtracking de dentro do lookahead. A

referência anterior seguinte simplesmente faz uma nova correspondência dos caracteres literais encontrados dentro do lookahead, tornando-os parte da correspondência real.

Quantificadores aninhados e backtracking desenfreado

Os chamados quantificadores aninhados sempre merecem uma atenção e um cuidado adicionais se não quisermos correr o risco de criar backtracking desenfreado. Um quantificador é aninhado quando ocorre dentro de um agrupamento que é, por sua vez, repetido por outro quantificador (p. ex. `(x+)*`).

A utilização de quantificadores aninhados não representa por si só um perigo ao desempenho. Entretanto, caso você não tenha cuidado, ela pode facilmente criar um número enorme de modos de divisão do texto entre os quantificadores internos e externos enquanto busca casar uma string.

Como exemplo, digamos que você queira buscar a correspondência de tags HTML e o resultado seja a seguinte expressão regular:

```
/<(?:[^>'"]|"[^"]*"|'[^']*')*>/
```

Talvez isso seja simplista demais, já que não lida corretamente com todos os casos de marcação válida e inválida, mas pode dar certo se for utilizado apenas para processar fragmentos válidos de HTML. Sua vantagem em comparação a soluções ainda mais ingênuas, como `/<[^>]*>/`, é a de que essa implementação leva em conta os caracteres `>` que ocorrem dentro dos valores de atributos. Ela o faz utilizando a segunda e a terceira alternativas no grupo de não captura, que casam valores inteiros de atributos com aspas simples ou duplas em passos individuais, permitindo que todos os caracteres, com exceção de seu tipo específico de aspas, ocorram dentro deles.

Até aqui, não há risco da ocorrência de backtracking desenfreado, apesar dos quantificadores `*` aninhados. A segunda e a terceira opção da alternação correspondem a exatamente uma sequência de string por repetição do grupo, o que faz com que o número potencial

de pontos de backtracking aumente de modo linear junto do comprimento da string-alvo.

Entretanto, observe a primeira alternativa do grupo de não captura: `[^>'"]`. Ela pode corresponder apenas a um caractere por vez, o que parece um pouco ineficiente. Talvez você esteja pensando que seria melhor adicionarmos um quantificador `+` ao final dessa classe de caracteres para que mais de um caractere adequado possa ser correspondido a cada repetição do grupo – em posições dentro da string-alvo onde a expressão regular encontra correspondências – e nisso você está certo. Ao casar mais de um caractere por vez, você permite que a expressão regular pule muitos passos desnecessários em seu caminho a uma correspondência de sucesso.

O que talvez não seja tão aparente são as consequências negativas que tal mudança trará consigo. Caso a expressão regular case com um caractere `<` de abertura, mas não encontre o caractere `>` na sequência, cenário em que a tentativa de correspondência se completaria com sucesso, certamente ocorrerá um backtracking desenfreado devido ao enorme número de formas pelas quais o novo quantificador interno pode ser combinado com o quantificador externo (que segue o grupo de não captura) para casar com o texto que segue `<`. A regex deve testar todas essas permutações antes de desistir da tentativa de correspondência. Tenha cuidado!

De mal a pior. Para um exemplo ainda mais extremo da aplicação de quantificadores aninhados tendo como resultado backtracking desenfreado, aplique a expressão regular `/(A+A+)+B/` a uma string contendo apenas `A`s. Ainda que essa expressão regular seja melhor do que `/AA+B/`, apenas para a discussão imagine que os dois `A`s representam padrões diferentes capazes de corresponder a algumas das mesmas strings.

Aplicada a uma string composta de dez `A`s (`"AAAAAAAAAA"`), a expressão regular começa utilizando o primeiro `A+` para casar com todos os dez caracteres. Em seguida, ela recua um caractere, permitindo que o segundo `A+` case com o último. Então, o

agrupamento tenta se repetir, mas como agora não existem mais As e o quantificador + do grupo já cumpriu seu objetivo de casar ao menos uma vez, a expressão regular buscará por B. Dessa vez ela não encontra, mas ainda não pode desistir, pois existem mais caminhos dentro da expressão regular que ainda não foram experimentados. Por exemplo, podemos pensar que o primeiro A+ corresponde a oito caracteres, enquanto o segundo, a dois. Ou que o primeiro corresponde a três caracteres, o segundo a dois, e o grupo se repete duas vezes. Ou até que durante a primeira repetição do grupo o primeiro A+ corresponde a dois caracteres e o segundo a três; já na segunda, o primeiro corresponde a um e o segundo a quatro. Ainda que para nós pareça tolo pensar que qualquer tipo de backtracking possa produzir o B faltante, a expressão regular checará obstinadamente cada uma dessas infrutíferas opções, e ainda outras mais. A pior complexidade possível dessa expressão regular é uma de magnitude O(2n), ou dois elevado à nona potência, onde n é o comprimento da string. Com os dez As que utilizamos, a expressão regular necessita de 1.024 passos de backtracking para que a correspondência falhe. Com 20 As, o número explode para mais de um milhão. Trinta e cinco As devem ser suficientes para que o Chrome, o IE, o Firefox e o Opera fiquem paralisados por pelo menos dez minutos (se não permanentemente) enquanto processam mais de 34 bilhões de passos de backtracking necessários para invalidar todas as permutações da regex. A exceção se refere às versões recentes do Safari, capazes de detectar os movimentos em círculo da expressão regular e rapidamente cancelar a correspondência (o Safari também impõem um limite ao número máximo de passos de backtracking permitidos, abortando tentativas caso esse limite seja excedido).

A chave para evitar esse tipo de problema passa pela certificação de que duas partes de uma expressão regular não possam corresponder à mesma parte de uma string. Para essa regex, a solução seria reescrevê-la na forma de /AA+B/. O problema, todavia, pode ser mais difícil de evitar em regexes complexas. A adição de

um grupo atômico emulado muitas vezes funciona bem como um último recurso, ainda que outras soluções, quando disponíveis, sejam capazes de manter suas expressões regulares mais compreensíveis. Por exemplo, reescrevendo nossa expressão regular da seguinte maneira, `/((?=(A+A+))\2)+B/`, eliminamos completamente o problema do backtracking.

Um lembrete sobre benchmarking

Uma vez que o desempenho das expressões regulares pode ser consideravelmente diferente dependendo do texto sobre o qual elas são aplicadas, não existem meios diretos para realização de avaliações de desempenho comparativas entre regexes. Para um resultado ideal, você deve avaliar o desempenho de suas expressões regulares sobre strings de teste, com diferentes extensões que apresentem correspondências totais e parciais, inclusive incluindo casos em que não ocorra casamento algum.

Há uma razão por trás da prolongada cobertura que damos ao backtracking nesse capítulo. Sem um firme entendimento sobre esse tópico, você não será capaz de se antecipar e identificar problemas dessa ordem. Para ajudá-lo a capturar rapidamente backtracking desenfreado, sempre teste suas expressões regulares com longas strings que contenham correspondências parciais. Pense nos tipos de strings com as quais suas expressões terão correspondências desse tipo e acrescente-as a seus testes.

Outras formas de melhorar a eficiência das expressões regulares

A seguir há uma série de técnicas adicionais para melhorar a eficiência de suas regexes. Vários dos pontos mencionados aqui já foram lembrados durante nossa discussão sobre backtracking.

Busque fracassos rápidos

O processamento lento de expressões regulares é normalmente causado por uma lentidão nos fracassos, e não nas

correspondências. Lembre-se de que caso esteja utilizando uma expressão regular para casar com pequenas partes de uma string mais longa, a regex falhará em muito mais posições do que obterá sucesso. Uma alteração que faça com que a expressão regular realize suas correspondências mais rapidamente, mas demore mais em seus fracassos (p. ex., um aumento no número dos passos de backtracking necessários para que sejam testadas todas as permutações da expressão) normalmente é uma má ideia.

Comece suas expressões regulares com sinais simples e necessários

É preferível que o sinal inicial de uma regex seja testado com rapidez, eliminando o maior número possível de posições não correspondentes. Bons sinais iniciais para esse propósito são âncoras (`^` ou `$`), caracteres específicos (p. ex., `x` ou `\u263A`), classes de caracteres (p. ex., `[a-z]` ou abreviações como `\d`) e limites de palavras (`\b`). Se possível, evite começar suas regexes com agrupamentos ou sinais opcionais e com alternações de topo como `/one|two/`, visto que esses casos forçam a regex a considerar vários sinais iniciais. O Firefox é sensível à utilização de qualquer quantificador nos sinais iniciais, sendo mais capacitado para realizar otimizações nesses cenários, por exemplo, `\s\S*` no lugar de `\s+` ou de `\s{1,}`. Outros navegadores eliminam tais diferenças por otimização.

Faça com que padrões quantificados e seus sinais seguintes sejam mutuamente excludentes

Quando há uma sobreposição nos caracteres que sinais (tokens) adjacentes e subexpressões são capazes de casar, o número de formas em que uma expressão regular tentará dividir o texto entre eles aumenta. Para evitar tal cenário, torne seus padrões bem específicos. Não use `".*?"` (que depende de backtracking) quando o que você realmente quer dizer é `"[^"\r\n]*"`.

Reduza a quantidade e o alcance das alternações

A alternação utilizando a barra vertical, |, pode exigir que todas as opções sejam testadas em cada posição de uma string. Podemos frequentemente reduzir a necessidade de alternações utilizando classes de caracteres e componentes opcionais ou empurrando a alternação mais para dentro da expressão regular (permitindo que mais tentativas de correspondência falhem antes que a alternação seja alcançada). A tabela seguinte mostra exemplos dessas técnicas.

Em lugar de	Use
cat|bat	[cb]at
red|read	rea?d
red|raw	r(?:ed|aw)
(.|\r|\n)	[\s\S]

 Classes de caracteres que casam com qualquer caractere (como [\s\S], [\d\D], [\w\W] ou [\0-\uFFFF) são, na verdade, equivalentes a (?:.|\r|\n| \u2028|\u2029). Essa opção inclui os quatro caracteres que não são casados pelo ponto: o carriage return, o line feed (ou nova linha), o separador de linha e o separador de parágrafo.

Classes de caracteres são mais rápidas do que a alternação, uma vez que são implementadas pela utilização de vetores de bit (ou de outras implementações mais rápidas) em lugar do backtracking. Quando a alternação for necessária, coloque primeiro as alternativas mais frequentes, desde que isso não afete o que casa com a regex. Opções de alternação são testadas da esquerda para a direita, de modo que quanto maior é o número de vezes que se espera o casamento de uma expressão, mais rapidamente ela deve ser considerada.

Observe que o Chrome e o Firefox realizam algumas dessas otimizações automaticamente, sendo menos afetados pelas técnicas de alternação com ajustes manuais.

Utilize grupos de não captura

Grupos de captura consomem tempo e memória para memorizar referências anteriores e mantê-las atualizadas. Caso não precise de uma referência anterior, evite esse custo adicional utilizando um grupo de não captura, por exemplo, (?:...) em vez de (...).

Algumas pessoas gostam de envolver suas expressões regulares em um grupo de captura quando precisam de uma referência anterior ligada à correspondência inteira. Isso é desnecessário. Podemos fazer referências a correspondências inteiras com, por exemplo, o elemento zero em arrays retornados por `regex.exec()` ou por `$&` em strings de substituição.

A substituição dos grupos de captura por seus parentes de não captura tem impacto mínimo no Firefox, mas pode representar uma grande diferença em outros navegadores quando lidamos com strings mais longas.

Capture texto interessante para reduzir o pós-processamento

Como uma ressalva à última dica, caso tenha de fazer referências a partes de uma correspondência, capture essas partes e utilize as referências anteriores produzidas. Por exemplo, caso esteja redigindo um código para processar o conteúdo das strings entre aspas, casadas por uma expressão regular, use `/"([^"]*)"/` e trabalhe com a referência anterior número um, em vez de utilizar `/"[^"]*"/` e ter de remover manualmente as aspas do resultado. Quando utilizada em um loop, esse forma de economia de trabalho pode resultar em um ganho significativo de tempo.

Exponha os sinais necessários

Para auxiliar as engines de regex a tomarem decisões inteligentes sobre como otimizar uma rotina de busca, tente facilitar a determinação de quais sinais são necessários. Quando sinais são utilizados dentro de subexpressões ou de alternações, é mais difícil para as engines determinarem quais são necessários. Algumas delas nem mesmo tentarão realizar essa tarefa. Por exemplo, a expressão regular `/^(ab|cd)/` expõe sua âncora de começo de string. Tanto o IE quanto o Chrome veem isso e impedem previamente que a expressão regular tente encontrar outras correspondências depois do começo da string, tornando essa busca praticamente instantânea, não importando o

tamanho da string. Entretanto, uma vez que a regex equivalente `/(âb|^cd)/` não expõe sua âncora ^, o IE não aplica o mesmo tipo de otimização, acabando por buscar, desnecessariamente, correspondências em cada posição da string.

Utilize quantificadores apropriados

Conforme descrito na seção “Repetição e backtracking”, quantificadores do tipo gulosos e preguiçosos realizam a busca por correspondências de modos diferentes, mesmo quando casam as mesmas strings. A utilização do tipo de quantificador apropriado (de acordo com a quantidade prevista de backtracking), nos casos adequados, pode melhorar sensivelmente o desempenho de seu código, especialmente em strings mais longas.

Quantificadores do tipo `*?` são particularmente lentos no Opera 9.x e nas versões anteriores. O Opera 10 elimina essa desvantagem.

Reutilize regexes guardando-as em variáveis

A atribuição de suas expressões regulares a variáveis permite evitar sua compilação repetitiva. Algumas pessoas exageram, adotando estratégias para armazenamento em cache de regexes que buscam evitar a compilação de um dado padrão e assinalam a combinação mais de uma vez. Isso não é necessário; a compilação de expressões regulares é rápida, e tais estratégias provavelmente apenas adicionam um custo maior ao desempenho, em vez de ajudar. O importante mesmo é evitar a recompilação repetitiva de expressões regulares dentro de um loop. Em outras palavras, evite o seguinte:

```
while (/regex1/.test(str1)) {
  /regex2/.exec(str2);
  ...
}
```

Prefira esta opção:

```
var regex1 = /regex1/,
  regex2 = /regex2/;
```

```
while (regex1.test(str1)) {
  regex2.exec(str2);
  ...
}
```

Quebre expressões regulares complexas em partes menores

Tente evitar um exagero de funções dentro de uma única expressão regular. Complicados problemas de busca que demandam lógica condicional são mais fáceis de serem solucionados e normalmente mais eficientes quando quebrados em duas ou mais expressões regulares, cada uma delas buscando dentro das correspondências da anterior. Expressões regulares gigantescas que fazem tudo em um único padrão são de difícil manutenção, além de tenderem a apresentar mais problemas relacionados a backtracking.

Quando não devemos utilizar expressões regulares

Quando aplicadas com critério, expressões regulares são muito rápidas. Entretanto, sua utilização é desnecessária quando estamos simplesmente buscando strings literais. Esse é o caso quando sabemos com antecedência qual parte de uma string desejamos testar. Por exemplo, caso deseje conferir se uma string termina em um ponto-e-vírgula, você poderia utilizar algo do tipo:

```
endsWithSemicolon = /;$/.test(str);
```

Talvez você se surpreenda ao saber que nenhum dos grandes navegadores é atualmente inteligente o bastante para perceber com antecedência que essa expressão regular deve casar apenas ao final da string. O que eles acabam fazendo é avaliar a string por inteiro. Cada vez que um ponto-e-vírgula é encontrado, a regex avança para o sinal seguinte (`$`), que confere se a correspondência se encontra no fim da string. Em caso negativo, a expressão continua sua busca por uma correspondência até que finalmente alcança a ponta final da string. Quanto maior for sua string (e mais pontos-e-vírgulas ela utilizar), mais o processo demorará.

Nesse caso, uma melhor abordagem é pular todos os passos intermediários exigidos por uma expressão regular e simplesmente conferir se o último caractere é realmente um ponto-e-vírgula:

```
endsWithSemicolon = str.charAt(str.length - 1) == ";";
```

Quando as strings-alvo forem pequenas, essa abordagem será apenas um pouco mais rápida do que o teste com base em expressões regulares. Entretanto, o importante é que nesse caso o comprimento da string já não afetará mais o tempo exigido para a realização do teste.

Esse exemplo utilizou o método charAt para ler o caractere em uma posição específica. Os métodos slice, substr e substring funcionam bem quando você deseja extrair e conferir o valor de mais de um caractere em uma posição específica. Além disso, os métodos indexOf e lastIndexOf são ótimos para encontrar a posição de strings literais ou para conferir se elas estão presentes. Todos esses métodos são rápidos e podem ajudá-lo a evitar o custo adicional oriundo das expressões regulares. Utilize-os quando buscar strings literais que não demandem expressões regulares com atributos sofisticados.

Aparo de suas strings

A remoção de espaços em branco no início ou ao final de uma string é uma tarefa simples, mas comum. Ainda que o ECMAScript 5 tenha adicionado um método nativo trim (o que faz com que ele deva começar a ser visto nos navegadores futuros), tradicionalmente ele não era incluído no JavaScript. Assim, para a atual leva de navegadores, ainda é necessário implementarmos nós mesmos um método trim (do inglês *to trim*, aparar) ou utilizarmos uma biblioteca que o inclua.

O aparo de strings (trimming) não é um gargalo de desempenho comum, mas serve como um ótimo estudo de caso para a otimização de expressões regulares, já que existem várias formas pelas quais ele pode ser implementado.

Aparo com expressões regulares

Utilizando expressões regulares, podemos implementar um método trim com um volume relativamente pequeno de código. Isso será especialmente importante em bibliotecas JavaScript voltadas ao tamanho dos arquivos. Talvez a melhor solução disponível seja a utilização de duas substituições – uma para remoção do espaço em branco inicial e outra para o final. Dessa forma, mantemos as coisas simples e rápidas, especialmente quando estivermos lidando com strings mais longas.

```
if (!String.prototype.trim) {
  String.prototype.trim = function() {
    return this.replace(/^\s+/, "").replace(/\s+$/, "");
  }
}
// teste o novo método...
// caracteres tab (\t) e line feed (\n)
// estão inclusos no espaço em branco inicial
var str = " \t\n test string ".trim();
alert(str == "test string"); // alerta "true"
```

O bloco if que cerca esse código evita a sobreposição (override) do método trim caso ele já exista, uma vez que métodos nativos são otimizados e geralmente muito mais rápidos do que qualquer coisa que você possa implementar sozinho com uma função JavaScript. Implementações subsequentes desse exemplo presumem que esse condicional existe, ainda que ele não seja copiado sempre.

Podemos melhorar em torno de 35% o desempenho no Firefox[8] (um valor menor ou maior depende do comprimento e do conteúdo da string-alvo), substituindo /\s+$/ (a segunda expressão regular) por /\s\s*$/. Ainda que essas duas expressões regulares sejam funcionalmente idênticas, o Firefox oferece uma otimização adicional para regexes que começam com um sinal não-quantificado. Em outros navegadores, a diferença será menos significativa, ou mesmo eliminada totalmente pela otimização Por outro lado, alterar a regex, que casa no início das strings, para /^\s\s*/ não produz uma diferença significativa. Isso ocorre porque a âncora

inicial ^ toma a precaução de invalidar posições não correspondentes (impossibilitando que uma pequena diferença de desempenho se avolume em milhares de tentativas de correspondência dentro de uma longa string).

A seguir veremos várias outras implementações de trimming utilizando expressões regulares. Elas representam algumas das alternativas mais comuns que você poderá encontrar. Também podemos ver os índices de desempenho dos vários navegadores, para todas as implementações de aparo descritas aqui, na tabela 5.2 ao final da seção. Há, na verdade, muitas formas além das listadas que podem ser usadas para escrever uma expressão regular capaz de ajudá-lo a aparar strings. Todavia, elas são invariavelmente mais lentas (ou de desempenho menos convincente quando aplicadas em vários navegadores) do que o uso de duas substituições simples quando trabalhamos com strings mais longas.

```
// aparo 2
String.prototype.trim = function() {
  return this.replace(/^\s+|\s+$/g, "");
}
```

Essa é provavelmente a solução mais comum. Ela combina as duas regexes simples por alternação e utiliza o sinalizador global (/g) para substituir todas as correspondências, em vez de apenas a primeira (ela casará duas vezes quando seu alvo apresentar espaços em branco tanto no início quanto no fim). Não se trata de uma abordagem terrível, mas é mais lenta do que a simples utilização de duas substituições quando estivermos trabalhando com strings longas, já que as duas opções de alternação têm de ser testadas a cada posição de caractere.

```
// aparo 3
String.prototype.trim = function() {
  return this.replace(/^\s*([\s\S]*?)\s*$/, "$1");
}
```

Essa regex atua comparando toda a string e capturando a sequência entre o primeiro e o último caractere de espaço em branco (se houver) como referência anterior número um. Ao

substituir toda a string pela referência anterior um, temos como resultado uma versão aparada da string.

Essa abordagem é conceitualmente simples, mas a presença do quantificador `*?` no grupo de captura força a expressão regular a realizar muito trabalho extra (p. ex., backtracking). Justamente por isso, essa opção costuma ser lenta quando aplicada sobre strings-alvo mais longas. Depois que a regex entra no grupo de captura, o quantificador `*?` da classe `[\s\S]` exige que ela seja repetida o mínimo possível de vezes. Assim, a regex casa um caractere por vez, parando depois de cada um para tentar casar o padrão `\s*$` remanescente. Se isso falha devido à presença em algum ponto de caracteres que sejam espaços em branco depois da posição atual na string, a regex casa mais um caractere, atualizando a referência anterior e, depois, testando uma vez mais o restante do padrão.

A repetição por `*?` é particularmente lenta nas versões 9.x do Opera e nas versões anteriores. Como exemplo, o aparo de strings mais longas com esse método no Opera 9.64 apresenta um desempenho que é 10 a 100 vezes mais lento do que nos outros grandes navegadores. O Opera 10 corrige esse persistente problema, alinhando seu desempenho ao apresentado por outros navegadores.

```
// aparo 4
String.prototype.trim = function() {
  return this.replace(/^\s*([\s\S]*\S)?\s*$/, "$1");
}
```

Esse exemplo é semelhante à última expressão regular, mas nele substitui-se o quantificador preguiçoso por um guloso, por considerações de desempenho. Para garantir que o grupo de captura case apenas até o último caractere que não seja um espaço em branco, utilizamos um `\S` final. Entretanto, como a regex deve ser capaz de casar strings formadas apenas por espaços em branco, todo o grupo de captura é tornado opcional pela adição de um quantificador ponto de interrogação final.

Aqui, o asterisco guloso em `[\s\S]*` repete seu padrão de qualquer

caractere até o fim da string. Nesse ponto, a regex recua um caractere por vez até que possa casar o \S seguinte ou até que recue ao primeiro caractere casado dentro do grupo (depois disso, pula o grupo).

A menos que haja mais espaço em branco final do que outro tipo de texto, isso geralmente resulta em uma experiência mais rápida do que a solução prévia, que utilizava um quantificador *?. Na verdade, ela é tão mais rápida no IE, no Safari, no Chrome e no Opera 10 que é melhor até que a utilização das duas substituições. Isso porque esses navegadores contêm uma otimização especial para a repetição gulosa de classes de caracteres que casam com qualquer caractere. A engine de regex pula para o término da string sem avaliar caracteres intermediários (ainda que as posições de recuo continuem a ser lembradas), e então recua do modo apropriado. Infelizmente, esse método é consideravelmente mais lento no Firefox e no Opera 9. Dessa forma, pelo menos por enquanto, a utilização de duas substituições ainda é preferível se quisermos uma solução própria para o maior número de navegadores.

```
// aparo 5
String.prototype.trim = function() {
  return this.replace(/^\s*(\S*(\s+\S+)*)\s*$/, "$1");
}
```

Trata-se de uma abordagem relativamente comum, mas não há razão que a justifique, uma vez que é consistentemente uma das mais lentas opções mostradas aqui em todos os navegadores. Tem como semelhança às duas últimas regexes o fato de que casa a string inteira e depois a substitui com a parte que desejamos manter. Entretanto, uma vez que o grupo interno casa apenas uma palavra por vez, há muitos passos distintos que devem ser dados pela expressão. O dano ao desempenho poderá até ser imperceptível quando estivermos lidando com strings curtas, mas com strings longas, que contenham muitas palavras, essa regex se tornará um sério problema.

A alteração do grupo interno para um grupo de não captura – por

exemplo, a mudança de (\s+\S+) para (?:\s+\S+) – ajuda um pouco, eliminando em torno de 20 a 45% do tempo necessário para execução no Opera, no IE e no Chrome, além de apresentar melhorias mais sutis no Safari e no Firefox. Ainda assim, um grupo de não captura não é capaz de redimir essa implementação. Lembre-se de que o grupo externo não pode ser convertido em um grupo de não captura, já que é referenciado na string substituta.

Aparo sem a utilização de expressões regulares

Ainda que expressões regulares sejam rápidas, vale a pena considerar o desempenho de aparos efetuados sem sua ajuda. Veja uma forma de realizar essa tarefa:

```
// aparo 6
String.prototype.trim = function() {
  var start = 0,
    end = this.length - 1,
    ws = " \n\r\t\f\x0b\xa0\u1680\u180e\u2000\u2001\u2002\u2003
\u2004\u2005\u2006\u2007\u2008\u2009\u200a\u200b\u2028\u2029\u202f
\u205f\u3000\ufeff";
  while (ws.indexOf(this.charAt(start)) > -1) {
    start++;
  }
  while (end > start && ws.indexOf(this.charAt(end)) > -1) {
    end--;
  }
  return this.slice(start, end + 1);
}
```

A variável ws desse código inclui todos os caracteres de espaço em branco conforme definidos pelo ECMAScript 5. Por motivo de eficiência, evitamos a cópia de qualquer parte da string até que as posições de início e fim sejam conhecidas.

O resultado é que, quando houver apenas um pouco de espaço em branco nas pontas da string, essa abordagem deixa para trás o desempenho que conseguimos com as regexes. Isso porque ainda que as expressões regulares sejam adequadas para a remoção de

espaço em branco do início de uma string elas não são tão rápidas no aparo a partir do final de strings mais longas. Como vimos na seção “Quando não devemos utilizar expressões regulares”, uma regex não pode pular para o final de uma string sem considerar os caracteres pelo caminho. Entretanto, essa implementação faz exatamente isso, com o segundo loop trabalhando de trás para frente a partir do fim da string até que encontre um caractere que não seja espaço em branco.

Ainda que essa versão não seja afetada pela extensão geral da string, ela também apresenta sua própria fraqueza: a presença de longo espaço em branco inicial ou final. Isso porque a realização de um loop sobre caracteres verificando se eles são espaços em branco não pode se comparar à eficiência do código de procura otimizado de uma expressão regular.

Uma solução híbrida

A abordagem final que veremos nesta seção, é a combinação da eficiência universal de uma regex no aparo dos espaços em branco iniciais combinada à velocidade característica do método sem expressão regular no aparo dos espaços em branco finais.

```
// aparo 7
String.prototype.trim = function() {
  var str = this.replace(/^\s+/, ""),
    end = str.length - 1,
    ws = /\s/;
  while (ws.test(str.charAt(end))) {
    end--;
  }
  return str.slice(0, end + 1);
}
```

Esse método híbrido é incrivelmente rápido no aparo de uma quantidade pequena de espaços em branco, mas também elimina o risco ao desempenho associado à presença de strings com muitos espaços em branco iniciais ou até de strings formadas apenas por espaços em branco (ainda que continue presente a fraqueza

relacionada a strings com muitos espaços em branco em seu final). Observe que essa solução utiliza uma regex no loop para conferir se os caracteres ao final da string correspondem a espaços em branco. Ainda que o uso de uma expressão regular para isso acrescente um pouco de perda de desempenho, ele também permite que você defira a lista de caracteres de espaço em branco ao navegador no intuito de se beneficiar em termos de brevidade e compatibilidade.

A tendência geral para todos os métodos trim mostrados aqui é de que, nas soluções com base em expressões regulares, o comprimento geral da string tenha um maior impacto sobre o desempenho do que o número de caracteres a serem aparados. Por outro lado, as soluções que não utilizam regexes e que trabalham a partir do fim da string não são afetadas pela extensão geral da string, mas sim pela quantidade de espaço em branco a ser aparado. A simplicidade do uso de duas substituições com expressões regulares oferece um desempenho consistentemente respeitável nos vários navegadores, mesmo quando há variações no comprimento e no tamanho das strings, sendo por isso a melhor solução geral. A solução híbrida é excepcionalmente rápida no tratamento de strings longas, ao custo de um código um pouco maior e de uma fraqueza presente quando há um longo espaço em branco final. Consulte a tabela 5.2 para conferir todos os detalhes.

Tabela 5.2 – Desempenho nos vários navegadores das várias implementações de aparo

Navegador	Tempo (ms)[a]						
	Aparo 1[b]	Aparo 2	Aparo 3	Aparo 4	Aparo 5[c]	Aparo 6	Aparo 7
IE 7	80/80	315/312	547/539	36/42	218/224	14/1015	18/409
IE 8	70/70	252/256	512/425	26/30	216/222	4/334	12/205
Firefox 3	136/147	164/174	650/600	1098/1525	1.416/1.488	21/151	20/144
Firefox 3.5	130/147	157/172	500/510	1.004/1.437	1.344/1.394	21/332	18/50
Safari 3.2.3	253/253	424/425	351/359	27/29	541/554	2/140	5/80
Safari 4	37/37	33/31	69/68	32/33	510/514	<0,5/29	4/18
Opera 9.64	494/517	731/748	9.066/9.601	901/955	1.953/2.016	<0,5/210	20/241
Opera 10	75/75	94/100	360/370	46/46	514/514	2/186	12/198
Chrome 2	78/78	66/68	100/101	59/59	140/142	1/37	24/55

[a] O tempo reportado foi gerado pelo aparo de uma longa string (40 KB) 100 vezes, primeiro com dez e depois com mil espaços adicionados a cada ponta..
[b] Testado sem a otimização /\s\s*$/.
[c] Testado sem a otimização do grupo de não captura.

Resumo

Operações intensas com strings e regexes criadas sem cautela podem representar grandes obstruções ao bom desempenho de seu código, mas os conselhos dados neste capítulo devem ajudá-lo a evitar as armadilhas mais comuns.

- Ao concatenar várias strings ou strings grandes, a junção de arrays é o único método com um desempenho respeitável no IE7 e nas versões anteriores.
- Caso não se preocupe com o IE7 ou com suas versões anteriores, lembre-se de que a junção de arrays é um dos métodos mais lentos para a concatenação de strings. Prefira o uso dos operadores `+` e `+=` em seu lugar, evitando strings intermediárias desnecessárias.
- O backtracking é, ao mesmo tempo, um componente fundamental da correspondência de expressões regulares e uma fonte frequente de ineficiência nessas expressões.
- Backtracking desenfreado pode fazer com que uma regex que normalmente encontra suas correspondências com rapidez rode com lentidão ou até trave seu navegador quando aplicada a strings com correspondência parcial. Técnicas para prevenção desse problema incluem a criação de sinais adjacentes que sejam mutuamente excludentes, a prevenção de quantificadores aninhados que permitam a correspondência de uma mesma parte da string mais de uma vez e a eliminação de backtracking desnecessário pela adaptação da natureza atômica de um lookahead.
- Existem várias abordagens capazes de melhorar a eficiência de suas expressões regulares ajudando-as a encontrar mais rapidamente suas correspondências e a consumir menos tempo

considerando posições que não casam (consulte “Outras formas de melhorar a eficiência das expressões regulares”).

- *Regexes* nem sempre são a melhor ferramenta para o trabalho, especialmente quando estamos apenas buscando strings literais.
- Ainda que existam muitas formas de aparar uma string, o uso de duas simples expressões regulares (uma para remoção do espaço em branco inicial e outra para o final) oferece uma boa combinação de brevidade e eficiência compatível com muitos navegadores, mesmo que haja variação no conteúdo e no comprimento das strings. A realização de um loop a partir do fim da string em busca do primeiro caractere que não seja um espaço em branco ou a combinação dessa técnica com regexes em uma abordagem híbrida é capaz de oferecer uma boa alternativa que é menos influenciada pelo comprimento geral das strings-alvo.

---

1 O engine é apenas o software que faz com que suas expressões regulares funcionem. Cada navegador conta com sua própria engine regex (ou, se preferir, sua implementação), cada uma com suas próprias vantagens quanto ao desempenho.

2 N.T.: Backtracking é um algoritmo geral para encontrar todas (ou algumas) soluções para um problema computacional, que cria (builds) incrementalmente propostas para a solução e abandona (faz o backtrack) de cada proposta parcial assim que determina que ela não pode ser completada como uma solução válida. (Fonte: Wikipédia)

3 Os números das figuras 5.2 e 5.3 foram gerados pela média dos resultados da realização de cada teste dez vezes no IE7 em uma máquina virtual Windows XP com especificações modestas (Core 2 Duo de 2GHz e 1 GB de RAM dedicada).

4 Uma consequência disso é que alterações aparentemente insignificantes podem tornar suas expressões regulares mais rápidas em alguns navegadores e mais lentas em outros.

5 O valor da propriedade lastIndex de uma expressão regular é utilizado como a posição de partida da busca apenas pelos métodos exec e test, e somente se a regex tiver sido construída com o sinalizador /g (global). Regexes não globais e qualquer uma que seja passada aos métodos match, replace, search e split sempre iniciam sua busca do início da string-alvo.

6 Ainda que classes de caracteres como [a-z] e classes abreviadas, como \s ou ponto (.) permitam variações, elas não são implementadas utilizando recuos e, desse modo, não enfrentam os mesmos problemas de desempenho.

7 Podemos confiar nesse comportamento dos lookaheads, já que ele é consistente em todas as principais formas de expressões regulares, sendo ainda explicitamente exigido pelos padrões ECMAScript.

8 Testado no Firefox, nas versões 2, 3 e 3.5.

CAPÍTULO 6

Interfaces responsivas

Não há nada mais frustrante do que clicar sobre algo em uma página web e nada acontecer como resposta. Esse problema remonta à origem das aplicações web transacionais e resulta na agora comum mensagem “por favor, clique apenas uma vez” que acompanha a maioria das páginas de envio de formulários. A inclinação natural de um usuário é sempre repetir qualquer ação que não resulta em uma mudança óbvia. Justamente por isso, assegurar a responsividade de suas aplicações deve ser sempre uma preocupação importante.

O capítulo 1 introduziu o conceito de *thread* na interface do usuário, ou UI (de user interface), do navegador. Como recapitulação, a maioria dos navegadores tem um único processo que é compartilhado entre a execução do JavaScript e as atualizações da interface do usuário. Apenas uma dessas operações pode ser realizada de cada vez, o que significa que a interface do usuário não poderá responder a comandos enquanto o código JavaScript estiver sendo executado, e vice-versa. A interface do usuário efetivamente “trava” durante a execução do JavaScript. Como afeta muito o desempenho aparente de uma aplicação web, a administração do tempo que seu código leva para ser executado é incrivelmente importante.

Thread da interface do usuário do navegador

O processo compartilhado pelo JavaScript e pelas atualizações da interface do usuário é frequentemente chamado de thread da UI do navegador (ainda que o termo “thread” não seja necessariamente correto para todos os navegadores). O processo de thread da UI

trabalha com um simples sistema de fila onde as tarefas são mantidas em espera até que ele esteja ocioso. Uma vez ocioso, a próxima tarefa colocada na fila é recuperada e executada. Essas tarefas consistem em código JavaScript a ser executado ou em atualizações que devem ser realizadas à UI, incluindo re-desenhos e reflows (discutidos no capítulo 3). Talvez a parte mais interessante desse processo seja o fato de que cada ação do usuário pode resultar na adição de uma ou mais tarefas à fila.

Considere a simples interface em que um clique sobre um botão resulta na exibição de uma mensagem na tela:

```
<html>
<head>
  <title>Browser UI Thread Example</title>
</head>
<body>
  <button onclick="handleClick()">Click Me</button>
  <script type="text/javascript">
    function handleClick() {
      var div = document.createElement("div");
      div.innerHTML = "Clicked!";
      document.body.appendChild(div);
    }
  </script>
</body>
</html>
```

Quando o botão desse exemplo é clicado, ele faz com que o thread da UI crie e adicione duas tarefas à fila. A primeira tarefa é uma atualização para o botão, que deve mudar de aparência para indicar que foi clicado. A segunda é uma tarefa de execução JavaScript que contém o código para `handleClick()`, fazendo com que o único código executado seja esse método e qualquer coisa que ele chamar. Presumindo que o thread da UI esteja ocioso, a primeira tarefa será recuperada e executada para atualizar a aparência do botão, seguida pela recuperação e execução da tarefa JavaScript. Durante o curso da execução, `handleClick()` cria um novo elemento `<div>`, anexando-o ao elemento `<body>`. Esse procedimento provoca outra alteração ao thread da UI. Isso significa que durante a execução do

JavaScript uma nova tarefa de atualização da UI será adicionada à fila, devendo ocorrer depois que o código JavaScript tiver sido executado. Veja a figura 6.1.

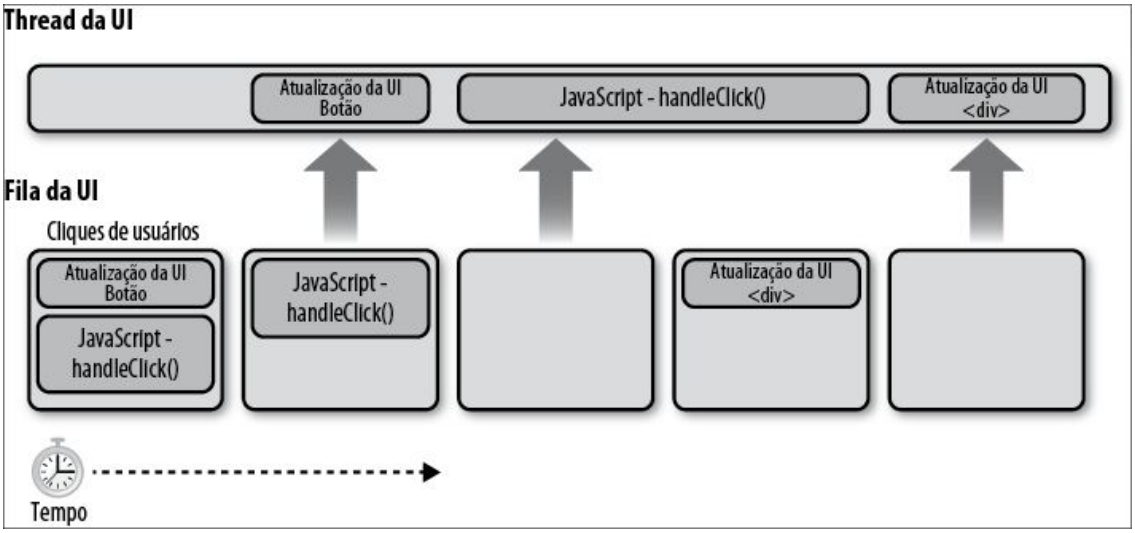


Figura 6.1 – Tarefas são adicionadas ao processo de thread da UI conforme o usuário interage com a página.

Quando todas as tarefas de thread da UI já tiverem sido executadas, o processo se tornará mais uma vez ocioso e aguardará a adição de mais tarefas à fila. O estado ocioso é considerado ideal, pois nele todas as ações do usuário resultam em uma atualização imediata da UI. Caso o usuário tente interagir com a página durante a execução de uma tarefa, ele não só não receberá uma resposta pronta, como também uma nova tarefa de atualização da UI muitas vezes nem será criada ou colocada na fila. Na verdade, a maior parte dos navegadores deixa de adicionar tarefas à fila enquanto códigos JavaScript estão sendo executados, o que torna imperativo que as tarefas JavaScript sejam executadas o mais rápido possível para que não afetem de modo adverso a experiência do usuário.

Limites dos navegadores

Os navegadores impõem limites ao tempo disponível para a execução de códigos JavaScript. Essa limitação é necessária para garantir que codificadores mal-intencionados não possam travar um navegador ou computador realizando operações intensas que nunca

terão fim. Esses limites vêm de duas formas: o limite de tamanho da pilha de chamadas (discutido no capítulo 4) e o limite para script de longa duração. O limite para script de longa duração é muitas vezes chamado de tempo limite de script de longa duração ou tempo limite de script máximo, mas a ideia básica é de que o navegador deve ser capaz de monitorar a duração da execução de um script e de interrompê-la quando se alcança um determinado tempo limite. Nesse ponto, uma caixa de diálogo é exibida ao usuário, como a da figura 6.2.

Figura 6.2 – O aviso de script de longa duração do Internet Explorer é exibido quando mais de 5 milhões de instruções tiverem sido executadas.

Podemos medir quanto tempo já foi gasto na execução de um script de duas formas. Primeiro, monitorando quantas instruções foram executadas desde que o script começou. Essa abordagem significa que o script pode ser executado com velocidades diferentes em máquinas diferentes, já que a memória disponível e a velocidade da CPU podem afetar quanto tempo é gasto na execução de uma única instrução. A segunda forma é pelo monitoramento do tempo total pelo qual um script está sendo executado. O volume de script que pode ser processado dentro de um dado limite de tempo também varia de acordo com a capacidade da máquina do usuário, mas o script é sempre interrompido quando se alcança determinado limite. Previsivelmente, cada navegador apresenta uma abordagem diferente para a detecção de scripts de longa duração:

- O Internet Explorer, desde a versão 4, define um limite padrão de 5 milhões de instruções; esse limite é armazenado em uma

configuração do registro do Windows chamada *HKEY_CURRENT_USER\Software\Microsoft\InternetExplorer\Styles\MaxScriptStatements*.

- O Firefox apresenta um limite de dez segundos; esse limite é armazenado nas configurações do navegador (acessíveis pela digitação de about:config na caixa de endereço) na chave *dom.max_script_run_time*.
- O Safari define um tempo limite de cinco segundos; essa configuração não pode ser alterada, mas você pode desabilitar o timer habilitando o menu *Develop* e selecionando a opção *Disable Runaway JavaScript Timer*.
- O Chrome não tem um limite para scripts de longa duração. Nele, é o sistema geral de erros que lida com tais instâncias.
- O Opera não tem um limite para scripts de longa duração e continuará a execução do código JavaScript até seu término. Mesmo assim, devido à sua arquitetura, isso não provocará instabilidade no sistema até que se complete a execução.

Quando o tempo limite para scripts de longa duração do navegador tiver sido atingido, uma caixa de diálogo será exibida para o usuário, independentemente da presença de qualquer outro código para manipulação do erro na página. Trata-se de um problema de usabilidade significativo, já que a maioria dos usuários da Internet não é composta de peritos em termos técnicos. Essas pessoas podem ficar confusas quanto ao significado da mensagem de erro e não saber qual opção é mais apropriada (suspender o script ou permitir que continue a ser executado).

Caso seu script dispare esse diálogo em algum navegador, isso significa que ele simplesmente está demorando tempo demais para completar sua tarefa. Também serve para indicar que o navegador do usuário se tornou não responsivo à suas ações enquanto o código JavaScript continuava a ser executado. Do ponto de vista dos desenvolvedores, não há forma de se recuperar depois do surgimento de uma caixa de diálogo avisando sobre o tempo limite

de um script; ela não pode ser detectada e, justamente por isso, não podemos nos preparar para os problemas que podem surgir como resultado. Fica claro que a melhor forma de lidarmos com o tempo limite de scripts de longa duração é simplesmente evitando-os totalmente.

Qual a duração ideal?

Só porque o navegador permite que um script continue a ser executado por um número determinado de segundos não significa que você deve permitir que isso ocorra. Na verdade, o tempo de execução de seu código JavaScript deve ser sempre bem menor do que os limites impostos pelos navegadores para que se crie uma boa experiência ao usuário. Atribui-se a Brendan Eich, criador do JavaScript, a seguinte frase “[JavaScript] que demora segundos para ser executado provavelmente está fazendo algo errado ...”

Se segundos inteiros são um tempo exagerado para a execução, qual é a melhor quantidade? Na verdade, uma demora de até mesmo um segundo já é considerada excessiva para a execução de um script. O tempo total que uma única operação JavaScript deve demorar (no máximo) é de 100 milissegundos. Esse número foi retirado de uma pesquisa conduzida por Robert Miller em 1968[1]. Desperta interesse o comentário[2] que fez Jakob Nielsen, no livro Usability Engineering (Editora Morgan Kaufmann, 1994). Ele diz que esse número não sofreu alterações com o passar do tempo, tendo sido, na verdade, reafirmado em 1991 por pesquisas realizadas no Xerox-PARC[3].

Nielsen ainda afirma que caso a interface responda à ação do usuário dentro de 100 milissegundos, o usuário terá a sensação de estar “manipulando diretamente os objetos na interface”. Qualquer intervalo de tempo maior do que 100 milissegundos significa uma sensação de desconexão. Como resultado, seu usuário não se sentirá no controle caso a execução demore mais do que esse limite.

Acrescenta-se uma nova complicação quando vemos que alguns navegadores preferem nem mesmo enfileirar atualizações à UI durante a execução do JavaScript. Por exemplo, caso você venha a clicar em um botão durante a execução de um código JavaScript, o navegador pode não registrar a necessidade da atualização da UI com a redefinição do botão nem iniciar nenhum JavaScript pela ação do usuário. O resultado será uma UI não responsiva que parece “travar” ou “congelar”.

Os vários navegadores se comportam mais ou menos da mesma forma. Enquanto um script está sendo executado, a UI não será atualizada pela interação do usuário. Tarefas JavaScript criadas como resultado de interações durante esse intervalo serão enfileiradas e somente executadas, ordenadamente, quando a tarefa JavaScript original tiver terminado. Atualizações à UI causadas pela interação do usuário serão automaticamente ignoradas durante esse tempo, já que é dada prioridade aos aspectos dinâmicos da página. Assim, um botão clicado enquanto um script está sendo executado nunca terá essa aparência, mesmo que seu manipulador `onclick` venha a ser executado.

O Internet Explorer restringe o número de tarefas JavaScript disparadas pela interação do usuário reconhecendo apenas duas ações repetidas em sequência. Por exemplo, ao clicar, durante a execução de um script, sobre um botão quatro vezes, você terá como resultado uma chamada apenas dupla ao manipulador de eventos onclick.

Ainda que os navegadores procurem fazer algo lógico nesses casos, todos esses comportamentos conduzem a uma experiência incômoda. Portanto, a melhor abordagem é a prevenção. Faça com que suas tarefas JavaScript não excedam a duração de 100 milissegundos. Essa medição deve ser feita dentro do mais lento navegador que você pretende atingir (para ferramentas que medem o desempenho de seu código JavaScript, consulte o capítulo 10).

Utilização de temporizadores

Apesar de seus esforços, haverá momentos em que uma tarefa

JavaScript simplesmente não poderá ser executada em menos de 100 milissegundos, devido a sua complexidade. Nesses casos, é preferível que você ceda o controle sobre o processo de thread da UI permitindo a realização de atualizações à interface. Ceder o controle significa interromper a execução de códigos JavaScript e dar à UI a chance de se atualizar antes que prossiga a execução de seu código. É aqui que entram em cena os temporizadores (timers) para JavaScript.

Informações básicas sobre temporizadores

Temporizadores são criados em JavaScript utilizando `setTimeout()` ou `setInterval()`, sendo que ambos aceitam os mesmos argumentos: uma função a ser executada e o tempo que deverá ser aguardado (em milissegundos) para sua execução. A função `setTimeout()` cria um temporizador que executa apenas uma vez, enquanto a função `setInterval()` cria um temporizador que se repete periodicamente.

A forma pela qual temporizadores interagem com o processo de thread da UI é útil para quebra de scripts de longa duração em segmentos menores. Chamadas a `setTimeout()` ou `setInterval()` dizem à engine que espere certo tempo e, somente então, adicione uma tarefa JavaScript à fila da UI. Por exemplo:

```
function greeting() {
  alert("Hello world!");
}
setTimeout(greeting, 250);
```

Esse código insere uma tarefa JavaScript que deverá executar a função `greeting()` dentro da fila da UI após a passagem de 250 milissegundos. Antes desse ponto, todas as outras atualizações à UI e tarefas JavaScript já deverão ter sido executadas. Tenha em mente que o segundo argumento indica apenas quando a tarefa deverá ser adicionada à fila da UI, o que não necessariamente corresponde ao momento em que será executada; primeiro, ela terá de aguardar a execução de todas as outras tarefas já enfileiradas. Considere o seguinte:

```
var button = document.getElementById("my-button");
button.onclick = function() {
  oneMethod();
  setTimeout(function() {
    document.getElementById("notice").style.color = "red";
  }, 250);
};
```

Quando o botão desse exemplo é clicado ele chama um método e define um temporizador. O código para alterar a cor do elemento `notice` está contido em um temporizador que será enfileirado em 250 milissegundos. Esses 250 milissegundos começam a contar a partir do momento em que `setTimeout()` é chamada, e não ao término da execução da função geral. Assim, se `setTimeout()` é chamada no ponto *n*, a tarefa JavaScript para execução do código será adicionada à fila da UI em *n* + 250. A figura 6.3 mostra o relacionamento quando o botão do exemplo é clicado.

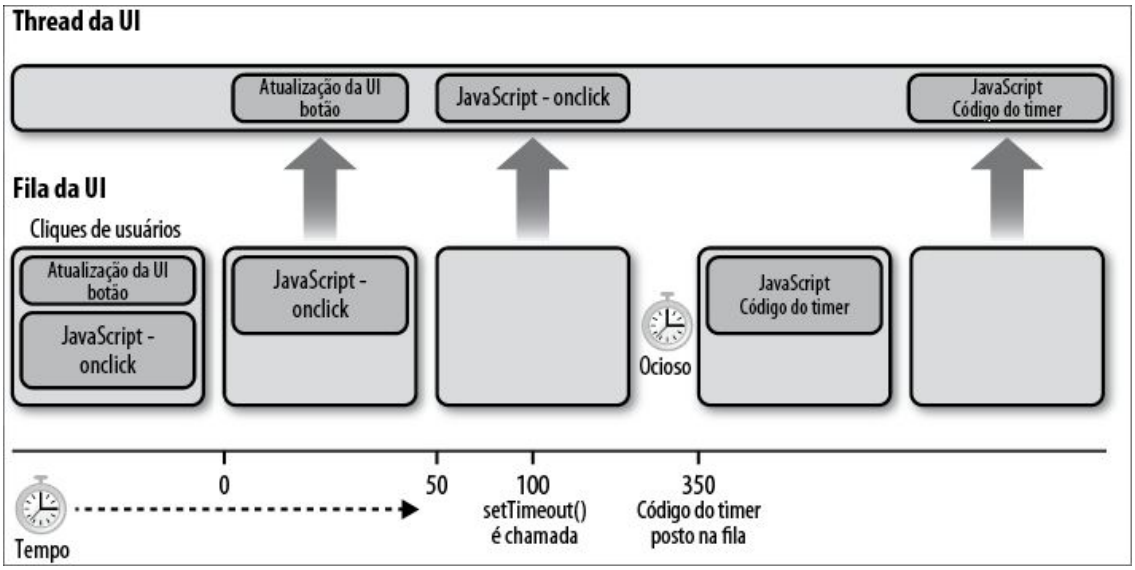


Figura 6.3 – O segundo argumento de setTimeout() indica quando a nova tarefa JavaScript deverá ser inserida na fila da UI.

Lembre-se que o código do temporizador nunca pode ser executado antes que termine a execução da função na qual ele foi criado. Por exemplo, caso façamos uma alteração ao código prévio de modo que a espera do temporizador seja menor e que haja outra chamada à função depois da criação do temporizador, é possível que o código da função temporizada seja posto na fila antes que o manipulador

de evento onclick tenha terminado sua execução:

```
var button = document.getElementById("my-button");
button.onclick = function() {
  oneMethod();
  setTimeout(function() {
    document.getElementById("notice").style.color = "red";
  }, 50);
  anotherMethod();
};
```

Se a execução de anotherMethod() demorar mais do que 50 milissegundos, o código do temporizador será adicionado à fila antes do término do manipulador onclick. O efeito será a execução praticamente imediata do código, assim que o manipulador tiver executado completamente, sem espera perceptível. A figura 6.4 ilustra essa situação.

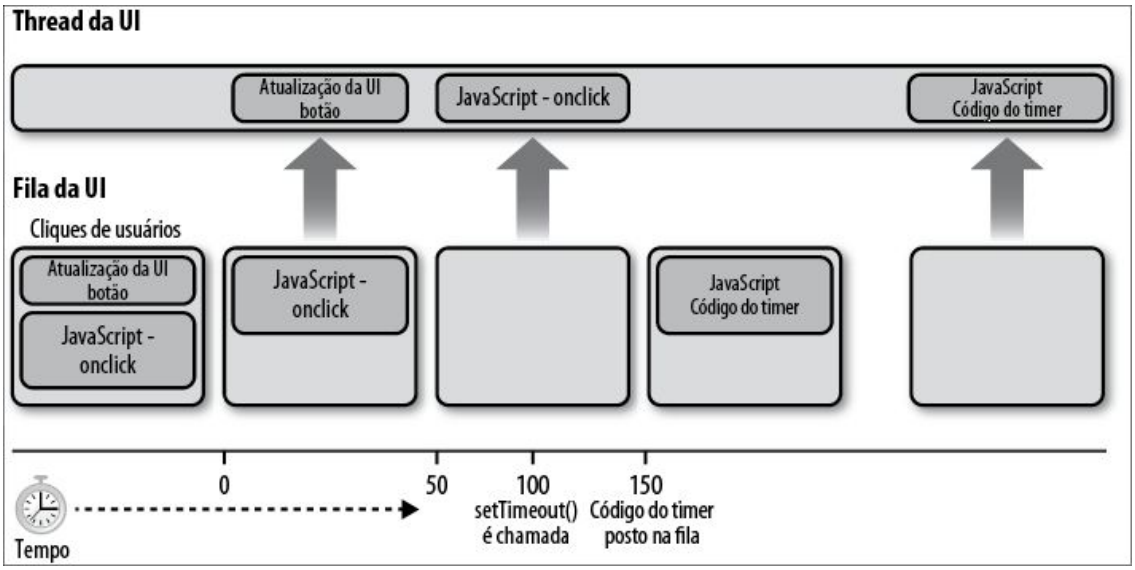


Figura 6.4 – Pode não haver espera perceptível na execução do código do temporizador caso a função em que setTimeout() é chamado demore mais para ser executada do que a espera determinada pelo temporizador.

De qualquer maneira, a criação de um temporizador cria uma pausa no processo de thread da UI conforme alterna de uma tarefa para a próxima. Consequentemente, o código do temporizador zera todos os limites relevantes do navegador, incluindo o tempo limite dos scripts de longa duração. Além disso, nesse código, a pilha de chamadas também é zerada. Essas características fazem dos temporizadores a melhor alternativa, compatível com vários

navegadores, no que diz respeito à criação de uma solução para código JavaScript de longa execução.

A função setInterval() é praticamente idêntica a setTimeout(), exceto pelo fato de a primeira adicionar repetidamente tarefas JavaScript à fila da UI. Sua principal diferença é que ela não fará adições se uma tarefa criada pela mesma chamada a setInterval() já estiver presente na fila.

A precisão dos temporizadores

Atrasos programados, criados por temporizadores em JavaScript, são muitas vezes imprecisos, podendo apresentar lapsos de milissegundos. Simplesmente por especificar uma duração de 250 milissegundos, você não pode ter certeza de que a tarefa será enfileirada exatamente nesse intervalo depois da chamada de setTimeout(). Os navegadores sempre tentam ter o máximo de precisão, mas muitas vezes ocorrem deslizes de alguns milissegundos, tanto para mais quanto para menos. Por esse motivo, não devemos confiar absolutamente nos temporizadores para medição do tempo real transcorriA resolução do temporizador nos sistemas Windows é de 15 milissegundos, o que significa que um temporizador de atraso definido como 15 será interpretado como 0 ou 15, dependendo de quando o tempo do sistema foi atualizado pela última vez. A definição de temporizadores de atrasos menores que 15 pode fazer, no Internet Explorer, com que o navegador trave. Justamente por isso, o menor atraso recomendado é de 25 milissegundos (o que, na verdade, acabará sendo um atraso de 15 ou 30) para que se possa garantir um atraso de ao menos 15 milissegundos.

A utilização de um atraso mínimo em seus temporizadores também ajuda a evitar problemas de resolução em outros navegadores e sistemas. A maioria dos navegadores exibe alguma variação nos atrasos quando tem de lidar com atrasos iguais a ou menores que 10 milissegundos.

Processamento de arrays com

temporizadores

Uma causa comum para scripts de longa duração são loops que tomam tempo demais em sua execução. Caso você já tenha experimentado as técnicas para otimização de loops apresentadas no capítulo 4, mas ainda não tenha conseguido reduzir suficientemente seu tempo de execução, os temporizadores podem ser o próximo passo a ser dado. A abordagem básica é a quebra do trabalho do loop em uma série de temporizadores.

Loops típicos seguem um padrão simples, da seguinte maneira:

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

Loops com essa estrutura podem demorar tempo demais para serem executados devido à complexidade de `process()`, ao tamanho de `items` ou à presença desses dois fatores combinados. No livro *Professional JavaScript for Web Developers*, segunda edição (Editora Wrox, 2009), exponho os dois fatores determinantes para a realização assíncrona de um loop que utilize temporizadores na forma de dois questionamentos:

- O processamento tem de ser feito de modo sincronizado?
- Os dados têm de ser processados de modo sequencial?

Caso sua resposta para essas duas perguntas seja “não”, seu código representará um bom candidato para a utilização de temporizadores na divisão do trabalho. Um padrão básico para a execução assíncrona de código é o seguinte:

```
var todo = items.concat(); // crie um clone do original

setTimeout(function() {

  // pegue e processe o próximo item do array
  process(todo.shift());

  // se houver mais itens para processar, crie outro temporizador
  if(todo.length > 0) {
    setTimeout(arguments.callee, 25);
  } else {
    callback(items);
  }
```

```
}, 25);
```

A ideia básica desse padrão é a criação de um clone do array original e de sua utilização como fila dos itens a serem processados. A primeira chamada a setTimeout() cria um temporizador para processamento do primeiro item do array. Chamar todo.shift() retorna o primeiro item removendo-o do array. Esse valor é passado para process(). Depois de processar o item, confere-se a presença de outros itens a serem processados. Caso ainda haja itens no array todo, outro temporizador será criado. Uma vez que o temporizador deve executar o mesmo código do original, arguments.callee é passada como primeiro argumento. Esse valor aponta para a função anônima dentro da qual o código está sendo executado. Caso não haja mais itens a serem processados, uma função callback() é chamada.


O valor correto para cada temporizador depende muito do caso em questão. Em termos gerais, é melhor utilizar ao menos 25 milissegundos, visto que atrasos menores podem deixar muito pouco tempo para a realização da maioria das atualizações da UI.

Como esse padrão demanda código demais para um loop regular, é preferível encapsular essa funcionalidade. Podemos fazê-lo desta forma:

```
function processArray(items, process, callback) {
  var todo = items.concat(); // crie um clone do original

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

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

A função processArray() implementa o padrão anterior de modo reutilizável aceitando três argumentos: o array a ser processado, a função a ser chamada em cada item e um callback a ser executado quando o processamento tiver terminado. Essa função pode ser utilizada da seguinte maneira:

```
var items = [123, 789, 323, 778, 232, 654, 219, 543, 321, 160];
function outputValue(value) {
  console.log(value);
}
processArray(items, outputValue, function() {
  console.log("Done!");
});
```

Esse código utiliza o método processArray() para entregar valores do array ao console e depois exibir uma mensagem quando todo o processamento tiver terminado. Ao encapsular o código do temporizador em uma função, podemos reutilizá-la em vários locais sem que sejam necessárias implementações diferentes.

 Um efeito colateral adverso da utilização de temporizadores para o processamento de arrays é o aumento do tempo total necessário para que o array seja processado. Isso porque há um intervalo obrigatório entre o processamento de cada item. Mesmo assim, trata-se de um mal necessário se quisermos evitar prejuízos à experiência do usuário, como o travamento do navegador.

Divisão de tarefas

O processo que normalmente imaginamos como uma só tarefa pode muitas vezes ser dividido em uma série de tarefas menores. Caso uma única função esteja demorando demais para ser executada, verifique se ela não pode ser quebrada em uma série de funções menores que sejam completadas mais rapidamente. Isso costuma ser tão simples quanto considerar uma única linha do código como uma tarefa atômica, mesmo que várias linhas também possam ser agrupadas em uma única tarefa. Algumas funções, por natureza, são facilmente divisíveis com base nas outras funções que chamam. Por exemplo:

```
function saveDocument(id) {
  // salve o documento
  openDocument(id)
  writeText(id);
  closeDocument(id);
  // atualize a UI indicando sucesso
  updateUI(id);
}
```

Caso essa função esteja demorando demais, ela pode facilmente ser dividida em uma série de passos menores pela divisão dos métodos individuais em temporizadores separados. Isso pode ser feito adicionando cada função a um array e, depois, utilizando um padrão semelhante ao processamento de array da seção anterior:

```
function saveDocument(id) {
  var tasks = [openDocument, writeText, closeDocument, updateUI];
  setTimeout(function() {
    // execute a próxima tarefa
    var task = tasks.shift();
    task(id);
    // determine se existem outras
    if (tasks.length > 0) {
      setTimeout(arguments.callee, 25);
    }
  }, 25);
}
```

Essa versão da função posiciona cada método no array `tasks`, executando apenas um método por temporizador. Fundamentalmente, esse exemplo se transforma em um padrão de processamento de array, tendo como única diferença o fato de que o processamento de um item envolverá a execução da função contida nele. Como discutimos na seção anterior, esse padrão também pode ser encapsulado, facilitando seu uso futuro:

```
function multistep(steps, args, callback) {
  var tasks = steps.concat(); // clone o array
  setTimeout(function() {
    // execute a tarefa seguinte
    var task = tasks.shift();
    task.apply(null, args || []);
    // determine se existem outras
    if (tasks.length > 0) {
      setTimeout(arguments.callee, 25);
    } else {
      callback();
    }
  }, 25);
}
```

A função multistep() aceita três argumentos: um array de funções a ser executado, um array de argumentos a ser passado a cada função durante sua execução e uma função de callback a ser chamada quando o processo estiver completo. A função pode ser utilizada da seguinte maneira:

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

Código cronometrado

Muitas vezes a execução de apenas uma tarefa por vez não é suficiente. Considere o processamento de um array de mil itens, onde cada item leva um milésimo de segundo. Se um item for processado em cada temporizador e houver um intervalo de 25 milissegundos entre cada processamento, isso implicará em um tempo total necessário para processamento do array de (25 + 1) x 1.000 = 26.000 milésimos de segundos, ou 26 segundos. O que aconteceria se processássemos os itens em lotes de 50 com um intervalo de 25 milissegundos entre cada lote? O tempo total de processamento se tornaria (1.000 / 50) x 25 + 1.000 = 1.500 milissegundos, ou um 1,5 segundo, com o benefício adicional de que o usuário não teria sua interface bloqueada uma vez que o maior intervalo de tempo pelo qual o script executou continuamente foi de apenas 50 milissegundos. O processamento de itens em lotes costuma ser muito mais rápido.

Caso você não se esqueça do limite de 100 milissegundos, como intervalo máximo pelo qual códigos JavaScript devem ser executados continuamente em suas aplicações, poderá otimizar os padrões apresentados anteriormente. Minha recomendação é dividir esse valor pela metade, nunca permitindo que seu código JavaScript execute continuamente por mais de 50 milissegundos. Desta forma, você tem a garantia de que o código não afetará de

modo adverso a experiência do usuário.

Cronometre quanto dura a execução de um trecho de seu código utilizando o objeto nativo Date. É dessa forma que a maior parte dos perfis JavaScript é traçada:

```
var start = +new Date(),
  stop;
someLongProcess();
stop = +new Date();
if(stop-start < 50) {
  alert("Just about right.");
} else {
  alert("Taking too long.");
}
```

Já que cada novo objeto Date é inicializado com o tempo atual do sistema, criando uma série periódica de objetos Date e comparando seus valores, podemos cronometrar o código. O operador de adição (+) converte o objeto Date em uma representação numérica, fazendo com que cálculos futuros não envolvam conversões. Essa mesma técnica básica pode ser utilizada para otimização dos padrões prévios de temporização.

Podemos acrescer o método processArray() dando a ele a capacidade de processar vários itens por cada temporizador, incluindo uma verificação do tempo:

```
function timedProcessArray(items, process, callback) {
  var todo = items.concat(); // crie um clone do original
  setTimeout(function() {
    var start = +new Date();
    do {
      process(todo.shift());
    } while (todo.length > 0 && (+new Date() - start < 50));
    if (todo.length > 0) {
      setTimeout(arguments.callee, 25);
    } else {
      callback(items);
    }
  }, 25);
}
```

A adição do loop do-while a essa função permite registrar o tempo depois que cada item for processado. O array sempre incluirá ao menos um item quando a função temporizada executar, de modo que um loop pós-teste faz mais sentido do que um pré-teste. Quando executada no Firefox 3, essa função processa um array de 1.000 itens, onde process() é uma função vazia, em um intervalo de 38 a 43 milissegundos; a função original processArray() processa o mesmo array em mais ou menos 25.000 milissegundos. Esse é um retrato claro do ganho que obtemos ao cronometrar tarefas antes de quebrá-las em pedaços menores.

Temporizadores e desempenho

Temporizadores podem fazer uma enorme diferença no desempenho geral de seu código JavaScript, mas sua utilização em excesso costuma ter efeitos negativos. Os códigos nesta seção têm utilizado apenas temporizadores sequenciados. Dessa forma, a cada momento existe somente um único temporizador em ação. A criação de novos deve sempre aguardar até que o último tenha terminado. Utilizar temporizadores dessa maneira não acarretará prejuízos a seu código.

Problemas de desempenho começam a aparecer quando vários temporizadores repetitivos são criados ao mesmo tempo. Uma vez que existe apenas um processo de thread da UI, todos eles competem pelo tempo de execução. Neil Thomas, do Google Mobile, pesquisou esse tópico como uma forma de medir o desempenho do aplicativo móvel do Gmail para iPhone e Android[4].

Thomas descobriu que temporizadores repetitivos de baixa frequência – aqueles que ocorrem em intervalos de um segundo ou mais – têm pouco ou nenhum efeito sobre a responsividade geral da aplicação web. Os intervalos do temporizador nesse caso são grandes demais para que se criem gargalos no processo de thread da UI, tornando segura sua utilização repetitiva. Entretanto, quando vários temporizadores repetitivos são utilizados com frequência

maior (entre 100 e 200 milissegundos), Thomas descobriu que o aplicativo móvel do Gmail se torna notavelmente mais lento e menos responsivo.

Podemos concluir que o número de temporizadores repetitivos de alta frequência deve ser sempre limitado em nossas aplicações. Como alternativa, Thomas sugere a criação de um único temporizador repetitivo que realize várias operações a cada execução.

Web workers

Desde a introdução do JavaScript, não existe uma forma de executarmos código fora do thread da UI do navegador. A API web workers altera essa característica introduzindo uma interface pela qual o código pode ser executado sem afetar o tempo de execução do thread da UI do navegador. Originalmente parte do HTML 5, a API web workers ganhou sua própria especificação (*http://www.w3.org/TR/workers/*), tendo sido implementada nativamente no Firefox 3.5, no Chrome 3 e no Safari 4.

Essa API representa o potencial de uma enorme melhoria de desempenho para as aplicações web. Nela, cada novo worker dá origem ao seu próprio thread onde deve ser executado o JavaScript. Isso significa que não apenas o código que está sendo executado em um worker não afetará a UI do navegador, como também não afetará o código sendo executado em outros workers.

Ambiente workers

Já que os web workers não estão presos ao processo de thread da UI, eles também não podem acessar muitos dos recursos do navegador. Parte da justificativa para que o JavaScript e a UI compartilhem do mesmo processo é o fato de que um frequentemente afeta o outro, fazendo com que a execução dessas tarefas fora de ordem possa resultar em uma experiência desagradável para o usuário. Os web workers poderiam introduzir

erros na interface ao realizarem alterações no DOM a partir de um thread externo, mas cada web worker tem seu próprio ambiente global, que apresenta apenas um subgrupo de atributos JavaScript disponíveis. O ambiente dos workers é composto da seguinte forma:

- Um objeto navigator que contém apenas quatro propriedades: appName, appVersion, userAgent e plataform.
- Um objeto location (semelhante a window, exceto pelo fato de que todas as suas propriedades são apenas para leitura).
- Um objeto self que aponta para o objeto worker global.
- Um método importScripts(), utilizado no carregamento de JavaScript externo para utilização no worker.
- Todos os objetos ECMAScript, como Object, Array, Date etc.
- O construtor XMLHttpRequest.
- Os métodos setTimeout() e setInterval().
- Um método close() que interrompe imediatamente o worker.

Uma vez que os web workers têm um ambiente global diferente, você não pode criar um a partir de qualquer código JavaScript. Na verdade, para criarmos um worker, temos antes de criar um arquivo JavaScript inteiramente separado que contenha apenas o código que ele deverá executar. Para criar um web worker, você deve passar a URL do arquivo JavaScript:

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

Uma vez que isso tenha sido executado, um novo thread com um novo ambiente será criado para o arquivo específico. Esse arquivo é baixado de modo assíncrono, e o worker não começará seu trabalho até que o arquivo tenha sido completamente baixado e executado.

Comunicação worker

A comunicação entre um worker e o código da página web é estabelecida por uma interface de evento. O código da página web pode passar dados ao worker utilizando o método postMessage(), que aceita um único argumento indicando os dados que devem ser

passados. Há também um manipulador de evento onmessage, que é utilizado para receber informações do worker. Por exemplo:

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

O worker recebe esses dados pelo disparo de um evento message. Um manipulador onmessage é definido, sendo que o objeto event tem uma propriedade data que contém todos os dados que foram passados. O worker poderá, então, passar informações de volta à página web utilizando seu próprio método postMessage():

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

A string final acaba no manipulador de evento onmessage para o worker. Esse sistema de mensagens é a única forma pela qual o worker e a página web podem se comunicar.

Apenas determinados tipos de dados podem ser passados utilizando o postMessage(). Você pode passar valores primitivos (strings, números, booleanos, null e undefined) assim como instâncias de Object e Array; não é possível a transmissão de qualquer outro tipo de dados. Dados válidos são dispostos em série, transmitidos de ou para o worker e então desserializados. Ainda que pareça que os objetos estão sendo passados diretamente, as instâncias são representações completamente separadas dos mesmos dados. A tentativa da transmissão de um tipo de dado não aceito resulta em um erro JavaScript.

 A implementação dos workers no Safari 4 permite a passagem apenas de strings utilizando postMessage(). Desde então ela foi atualizada para permitir a transmissão de dados serializados. Esse também é o modo como workers são implementados no Firefox 3.5.

Carregamento de arquivos externos

O carregamento de arquivos JavaScript externos em um worker é realizado pelo método importScripts(), que aceita uma ou mais URLs para carregamento de arquivos JavaScript. A chamada a importScripts() tem efeito bloqueador dentro do worker, fazendo com que o script não continue até que todos os arquivos tenham sido baixados e executados. Uma vez que o worker está sendo executado fora do thread da UI, não há preocupações quanto à responsividade quando ocorre esse tipo de bloqueio. Por exemplo:

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

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

A primeira linha desse código inclui dois arquivos JavaScript que serão disponibilizados no contexto do worker.

Utilização na prática

Web workers são indicados para quaisquer scripts de longa duração que funcionem com dados puros e que não tenham ligações à UI do navegador. Pode não parecer muito, mas embutidas em aplicações web, temos muitas abordagens que se beneficiariam do uso de workers no lugar de temporizadores.

Considere, por exemplo, o parsing de uma longa string JSON (o parsing em JSON é discutido mais detalhadamente no capítulo 7). Suponha que os dados sejam grandes o bastante para que o processo leve 500 milissegundos: tempo demais para permitirmos a execução do JavaScript no cliente, já que isso interferiria na experiência do usuário. Não é tão simples quebrarmos essa tarefa específica em pedaços menores para que possamos utilizar os temporizadores, assim, um worker é a solução ideal. O código seguinte ilustra a utilização em uma página web:

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

// quando os dados estão disponíveis, esse manipulador de eventos é chamado
worker.onmessage = function(event) {

  // a estrutura JSON é passada de volta
```

```
  var jsonData = event.data;
  // a estrutura JSON é utilizada
  evaluateData(jsonData);
};
// passe a longa string JSON a ser analisada
worker.postMessage(jsonText);
```

O código para o worker responsável pelo parsing JSON é o seguinte:

```
// dentro de jsonparser.js
// esse manipulador de eventos é chamado quando os dados JSON estão disponíveis
self.onmessage = function(event) {
  // a string JSON vem como event.data
  var jsonText = event.data;
  // faça o parsing da estrutura
  var jsonData = JSON.parse(jsonText);
  // envie de volta os resultados
  self.postMessage(jsonData);
};
```

Observe que mesmo que JSON.parse() venha a levar por volta de 500 milissegundos para ser executada, não há necessidade de redigirmos código adicional para dividir seu processamento. Essa execução ocorrerá em um thread separado, por isso, podemos deixar que leve o tempo necessário sem que venha a interferir na experiência do usuário.

A página repassa uma string JSON ao worker utilizando postMessage(). O worker recebe a string como event.data em seu manipulador de evento onmessage, começando em seguida a realizar o parsing. Quando termina, o objeto JSON resultante é repassado de volta à página pelo método postMessage() do worker. Esse objeto torna-se disponível como event.data no manipulador de evento onmessage da página. Lembre-se que isso atualmente somente funciona no Firefox 3.5 e nas versões posteriores, já que as implementações do Safari 4 e do Chrome 3 permitem a transmissão de strings apenas entre a página e o worker.

O parsing de uma longa string é apenas uma das muitas tarefas que podem ser realizadas pelos web workers. Algumas outras

possibilidades incluem:

- A codificação e decodificação de uma grande string.
- A realização de cálculos matemáticos complexos (incluindo o processamento de imagens ou vídeos).
- A ordenação de um grande array.

Sempre que um processo demorar mais que 100 milissegundos para completar, você deverá considerar se a solução utilizando workers não é mais adequada do que uma com base em temporizadores. Isso, é claro, depende da capacidade de seus navegadores-alvo.

Resumo

O JavaScript e as atualizações da interface do usuário operam dentro do mesmo processo. Isso faz com que apenas um possa ser feito de cada vez, o que significa que a interface do usuário não será capaz de reagir a suas ações enquanto códigos JavaScript estiverem sendo executados, e vice-versa. Administrar corretamente o processo de thread da UI significa não permitir que elementos JavaScript sejam executados por longos intervalos de tempo, afetando a experiência do usuário. Para tanto, lembre-se do seguinte:

- Nenhuma tarefa JavaScript deve demorar mais do que 100 milissegundos para ser executada. Intervalos maiores causam atrasos perceptíveis na atualização da UI e afetam de modo negativo a experiência geral do usuário.
- Os navegadores se comportam de modo diferente em resposta à interação do usuário durante a execução do JavaScript. Independentemente de seu comportamento, quando elementos JavaScript demoram muito em sua execução, a experiência do usuário se torna confusa e desconjuntada.
- Temporizadores podem ser utilizados para agendar a execução futura de partes de seu código, permitindo a divisão de scripts longos em uma série de tarefas menores.

- Web workers são um atributo de navegadores mais novos que permitem a execução de código JavaScript fora do processo de thread da UI, impedindo que ela trave.

Quanto mais complexo for a aplicação web, mais importante será que você administre o processo de thread da UI de modo proativo. Nenhum código JavaScript é tão importante que deva afetar de maneira adversa a experiência do usuário.

---

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

2 Disponível em: *www.useit.com/papers/responsetime.html.*

3 Card, S. K., G.G. Robertson, e J.D. Mackinlay, "*The information visualizer: An information workspace*", Proc. ACM CHI'91 Conf. (Nova Orleans: 28 Abril–2 Maio), 181–188. Disponível em: *http://portal.acm.org/citation.cfm?id=108874*.

4 Postagem completa disponível em: *http://googlecode.blogspot.com/2009/07/gmail-for-mobile-html5-series -using.html.*

CAPÍTULO 7

Ajax

Ross Harmes

O Ajax é um dos fundamentos do JavaScript de alto desempenho. Por exemplo, ele pode ser utilizado para carregar mais rapidamente uma página, postergando (delaying) o download de grandes recursos. Pode prevenir por completo carregamentos da página permitindo que dados sejam transferidos entre o cliente e o servidor de modo assíncrono. Pode até mesmo ser utilizado para buscar todos os recursos de uma página em uma única solicitação HTTP. Ao escolher a técnica de transmissão correta e o formato de dados mais eficiente, você é capaz de melhorar significativamente a forma como seus usuários interagem com seu site.

Este capítulo examina as mais rápidas técnicas para envio e recebimento de dados entre o cliente e o servidor, assim como os formatos mais eficientes para codificação de seus dados.

Transmissão de dados

O Ajax, em seu nível mais básico, é uma forma de comunicação com o servidor que não implica no descarregamento da página atual; dados podem ser solicitados ou enviados ao servidor. Há muitas formas diferentes pelas quais esse canal de comunicação pode ser estabelecido, cada uma com suas próprias vantagens e restrições. Esta seção examina as abordagens diferentes e discute suas implicações particulares ao desempenho.

Solicitação de dados

Existem cinco técnicas gerais para solicitação de dados de um servidor:

- XMLHttpRequest (XHR)
- Inserção dinâmica de tag script
- iframes
- Comet
- XHR Multiparte

As três mais utilizadas em JavaScript moderno e de alto desempenho são XHR, inserção dinâmica da tag script e XHR multiparte. A utilização do Comet e do iframes (como técnicas de transporte de dados) tende a ser extremamente específica e, por isso, não será examinada nessa seção.

XMLHttpRequest

De longe a técnica mais comum, o XMLHttpRequest (XHR) permite que você envie e receba dados de modo assíncrono. É bem aceito por todos os navegadores modernos e permite um bom grau de controle sobre a solicitação feita e os dados recebidos. Você pode adicionar cabeçalhos e parâmetros (tanto GET quanto POST) arbitrários à solicitação e ler todos os cabeçalhos retornados do servidor, assim como o texto de resposta em si. A seguir temos um exemplo de como ele pode ser utilizado:

```
var url = '/data.php';
var params = [
  'id=934875',
  'limit=20'
];
var req = new XMLHttpRequest();
req.onreadystatechange = function() {
  if (req.readyState === 4) {
    var responseHeaders = req.getAllResponseHeaders(); // pegue os cabecalhos de
resposta
    var data = req.responseText; // pegue os dados
    // processe os dados aqui...
  }
}
req.open('GET', url + '?' + params.join('&'), true);
req.setRequestHeader('X-Requested-With', 'XMLHttpRequest'); // define um cabeçalho
de solicitação
```

```
req.send(null); // envia a solicitação
```

Esse exemplo mostra como solicitar dados de uma URL com parâmetros e como ler o texto e os cabeçalhos de resposta. Um readyState de valor 4 indica que toda a resposta foi recebida e está disponível para manipulação.

É possível interagir com a resposta do servidor conforme ela é transferida escutando por um readyState valor 3. Isso é conhecido como streaming e é uma poderosa ferramenta para melhoria do desempenho de suas solicitações de dados:

```
req.onreadystatechange = function() {
  if (req.readyState === 3) { // alguns, mas não todos, os dados foram recebidos
    var dataSoFar = req.responseText;
    ...
  }
  else if (req.readyState === 4) { // todos os dados foram recebidos
    var data = req.responseText;
    ...
  }
}
```

Devido ao alto grau de controle que oferece o XHR, os navegadores colocam algumas restrições sobre ele. Você não pode utilizar XHR para solicitar dados de um domínio diferente de onde o código está sendo executado, sendo que versões mais antigas do IE não permitem o acesso ao readyState 3, impedindo o streaming. Dados que retornam da solicitação são tratados como uma string ou como um objeto XML; isso significa que grandes quantidades de dados demorarão bastante para serem processadas.

Apesar dessas desvantagens, o método XHR é a técnica mais utilizada para solicitação de dados, continuando a ser a mais produtiva. Justamente por isso, ela deve ser sua primeira opção.

POST versus GET na utilização da técnica XHR. Quando utilizar XHR para solicitar dados, você terá sempre que escolher entre os métodos POST e GET. Para solicitações que não alteram o estado de servidor e que apenas puxam dados de volta (chamadas de ação idempotente), prefira GET. Solicitações GET são armazenadas em

cache, melhorando o desempenho do código caso você busque os mesmos dados por repetidas vezes.

O método POST deve ser utilizado para buscar dados apenas quando o comprimento da URL e os parâmetros estiverem próximos a ou excederem 2.048 caracteres. Explica-se: o Internet Explorer limita as URLs a esse comprimento, e excedê-lo poderá fazer com que sua solicitação seja truncada.

Inserção dinâmica da tag script

Essa técnica supera a principal limitação da XHR: ela permite a solicitação de dados de um servidor em um domínio diferente. Trata-se de um hack[1]; em vez de instanciar um objeto construído com um propósito, você utiliza JavaScript para criar uma nova tag `script` e definir seu atributo `src` como uma URL em um domínio diferente.

```
var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/javascript/lib.js';
document.getElementsByTagName('head')[0].appendChild(scriptElement);
```

Ainda assim, inserções dinâmicas da tag `script` oferecem um controle muito menor do que o oferecido pela técnica XHR. Não se pode enviar cabeçalhos com a solicitação. Parâmetros apenas podem ser passados utilizando GET e não POST. Não é possível definir limites de tempo ou realizar novas tentativas da solicitação; na verdade, você nem necessariamente ficará sabendo se sua solicitação falhar. Terá de esperar até que todos os dados retornem antes que possa ter acesso a eles. Você também não terá acesso aos cabeçalhos de resposta ou à resposta inteira como uma string.

Esse último ponto é especialmente importante. Uma vez que a resposta está sendo utilizada como fonte de uma tag `script`, ela tem de vir como JavaScript executável. Não se pode utilizar XML cru, ou mesmo JSON cru; qualquer dado, independentemente do formato, deverá estar envolto em um função callback.

```
var scriptElement = document.createElement('script');
scriptElement.src = 'http://any-domain.com/javascript/lib.js';
document.getElementsByTagName('head')[0].appendChild(scriptElement);
function jsonCallback(jsonString) {
```

```
    var data = eval('(' + jsonString + ')');
    // processe os dados aqui...
  }
```

Nesse exemplo, o arquivo lib.js envolveria os dados na função jsonCallback:

```
  jsonCallback({ "status": 1, "colors": [ "#fff", "#000", "#ff0000" ] });
```

Apesar dessas limitações, essa técnica pode ser extremamente rápida. Uma vez que a resposta é executada como JavaScript, ela não é tratada como uma string que tem de ser processada. Por isso, este tem o potencial de ser o método mais rápido para o recebimento e o parsing dos dados, transformando-os em algo acessível a partir do lado cliente. Faremos uma comparação do desempenho da inserção dinâmica de tag script com desempenho do objeto XHR na seção sobre JSON, mais adiante no capítulo.

Tenha cuidado ao utilizar essa técnica para solicitar dados de um servidor que você não controla diretamente. A linguagem JavaScript não apresenta o conceito de autorização ou de controle de acesso, o que faz com que qualquer código que você incorpore utilizando a inserção dinâmica da tag script tenha controle completo sobre a página. Isso inclui a habilidade de modificar qualquer conteúdo, de redirecionar usuários para outro site ou até de monitorar suas ações nessa página, enviando os dados para um terceiro. Utilize muita cautela ao obter código de uma fonte externa.

XHR Multiparte

A mais nova das técnicas mencionadas aqui, a XHR multiparte (multipart XHR, ou MXHR) permite repassar diversos recursos do lado servidor para o lado cliente com apenas uma solicitação HTTP. Isso é feito pelo empacotamento dos recursos (sejam arquivos CSS, fragmentos HTML, código JavaScript ou imagens codificadas base64) no lado servidor seguido de seu envio ao cliente como uma longa sequência de caracteres, separados por alguma string previamente conhecida. O código JavaScript processa essa longa string e faz o parsing de cada recurso de acordo com seu tipo MIME

e com qualquer outro “cabeçalho” repassado junto a ela.

Vamos acompanhar esse processo do início ao fim. Primeiro, uma solicitação é feita ao servidor por vários recursos de imagens:

```
var req = new XMLHttpRequest();
req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = function() {
  if (req.readyState == 4) {
    splitImages(req.responseText);
  }
};
req.send(null);
```

Trata-se de uma solicitação bem simples. Você pede dados referentes a `rollup_images.php` e, quando os recebe, envia-os para a função `splitImages`.

Em seguida, no servidor, as imagens são lidas e convertidas em strings:

```
// leia as imagens e converta-as em strings codificadas em base64
$images = array('kitten.jpg', 'sunset.jpg', 'baby.jpg');
foreach ($images as $image) {
  $image_fh = fopen($image, 'r');
  $image_data = fread($image_fh, filesize($image));
  fclose($image_fh);
    $payloads[] = base64_encode($image_data);
  }
}
// agrupe essas strings em uma longa string e apresente-a como resultado
$newline = chr(1); // esse caractere não será exibido naturalmente em nenhuma string base64
echo implode($newline, $payloads);
```

Esse pedaço de código lê três imagens e faz sua conversão em longas strings de base com 64 caracteres. Elas são concatenadas utilizando um único caractere, o Unicode 1, e entregues ao cliente.

Uma vez no lado cliente, os dados são processados pela função `splitImages`:

```
function splitImages(imageString) {
  var imageData = imageString.split("\u0001");
  var imageElement;
```

```
  for (var i = 0, len = imageData.length; i < len; i++) {
    imageElement = document.createElement('img');
    imageElement.src = '' + imageData[i];
    document.getElementById('container').appendChild(imageElement);
  }
}
```

Essa função toma a string concatenada dividindo-a novamente em três partes. Cada parte é então utilizada para criar um elemento de imagem, que é inserido na página. A imagem não é convertida a partir de uma string base64 de volta a dados binários; em vez disso, ela é repassada ao elemento imagem utilizando uma URL `data:` e o tipo MIME `image/jpeg`.

O resultado final faz com que três imagens sejam repassadas ao navegador como uma única solicitação HTTP. Isso poderia ser feito para 20 imagens, ou mesmo para 100; a reposta seria maior, mas continuaríamos precisando apenas de uma única solicitação HTTP. Também podemos expandir essa técnica para outros tipos de recursos. Arquivos JavaScript, arquivos CSS, fragmentos HTML e imagens de vários tipos podem ser combinados em uma única resposta. Qualquer tipo de dados que possa ser manipulado como uma string em JavaScript pode ser enviado. Aqui temos funções que tomarão strings para código JavaScript, estilos CSS e imagens, convertendo-as em recursos que o navegador possa usar:

```
function handleImageData(data, mimeType) {
  var img = document.createElement('img');
  img.src = 'data:' + mimeType + ';base64,' + data;
  return img;
}
function handleCss(data) {
  var style = document.createElement('style');
  style.type = 'text/css';
  var node = document.createTextNode(data);
  style.appendChild(node);
  document.getElementsByTagName('head')[0].appendChild(style);
}
  function handleJavaScript(data) {
    eval(data);
}
```

Conforme as respostas MXHR tornam-se maiores, passa a ser necessário que cada recurso seja processado conforme é recebido, em vez de esperarmos pela resposta inteira. Isso pode ser feito detectando um readyState de valor 3:

```
for readyState 3:
var req = new XMLHttpRequest();
var getLatestPacketInterval, lastLength = 0;
req.open('GET', 'rollup_images.php', true);
req.onreadystatechange = readyStateHandler;
req.send(null);
function readyStateHandler{
  if (req.readyState === 3 && getLatestPacketInterval === null) {
    // comece a checagem
    getLatestPacketInterval = window.setInterval(function() {
      getLatestPacket();
    }, 15);
  }
  if (req.readyState === 4) {
    // interrompa a checagem
    clearInterval(getLatestPacketInterval);
    // pegue o último pacote
    getLatestPacket();
  }
}
function getLatestPacket() {
  var length = req.responseText.length;
  var packet = req.responseText.substring(lastLength, length);
  processPacket(packet);
  lastLength = length;
}
```

Assim que o readyState 3 for disparado pela primeira vez, um temporizador será iniciado. A cada 15 milissegundos a resposta será conferida pela presença de novos dados. Cada pedaço de dados será coletado até que um caractere delimitador seja encontrado. Ao término, tudo será processado como um recurso completo.

Mesmo que o código necessário para utilização de MXHR de modo abrangente seja complexo, vale a pena estudá-lo. A biblioteca

completa pode ser facilmente encontrada em *http://techfoolery.com/mxhr/*.

Há algumas desvantagens em usar essa técnica. A principal é a de que nenhum dos recursos buscados será armazenado em cache pelo navegador. Caso você busque um arquivo CSS utilizando MXHR e depois o carregue normalmente na página, ele não será armazenado em cache. Isso porque os recursos arrolados são transmitidos como uma longa string e então divididos pelo código JavaScript. Uma vez que não há modos de injetar por código um arquivo no cache do navegador, nenhum dos recursos buscados dessa forma será armazenado.

Outro ponto negativo é que versões mais antigas do Internet Explorer não aceitam o `readyState` 3 ou URLs do tipo `data:`. O Internet Explorer 8 aceita ambos, mas gambiarras ainda têm de ser utilizadas para as versões do Internet Explorer 6 e 7.

Mesmo com esses problemas, ainda há situações em que a técnica MXHR melhora de maneira significativa o desempenho geral da página:

- Páginas que contêm muitos recursos não utilizados em outros locais do site (e que, dessa forma, não precisam ser armazenados em cache), especialmente imagens.
- Sites que já estejam utilizando um único arquivo JavaScript ou CSS em cada página para reduzir o número de solicitações HTTP; uma vez que é único a cada página, ele nunca será lido a partir do cache a menos que uma página específica seja recarregada.

Uma vez que solicitações HTTP são um dos gargalos de desempenho mais extremos no Ajax, a redução de seu número necessário tem um grande impacto sobre o desempenho geral da página. Isso é especialmente verdadeiro quando somos capazes de converter 100 solicitações de imagens em uma única solicitação XHR multiparte. A realização de testes *ad hoc* com grande número de imagens pelos navegadores modernos tem mostrado que essa

técnica é de 4 a 10 vezes mais rápida que a realização de solicitações individuais. Execute você mesmo esses testes visitando *http://techfoolery.com/mxhr/*.

Envio de dados

Há momentos em que você não se importa com a recuperação de dados, desejando apenas enviá-los ao servidor. Você pode estar enviando informações não pessoais sobre um usuário para serem analisadas futuramente ou realizando a captura de todos os erros de script que ocorrem e enviando os seus detalhes ao servidor para criação de logs e alertas. Quando os dados têm apenas de ser enviados ao servidor, há duas técnicas amplamente utilizadas: XHR e beacons.

XMLHttpRequAinda que seja mais utilizada para recuperação de dados do servidor, a XHR também pode ser aplicada no envio de dados. Dados podem ser enviados como GET ou POST, assim como com outros cabeçalhos HTTP, o que confere uma grande flexibilidade a essas operações. A técnica XHR é especialmente útil quando a quantidade de dados enviados de volta ao servidor excede o comprimento máximo da URL em um navegador. Nessa situação, você pode enviar os dados de volta como POST:

```
var url = '/data.php';
var params = [
  'id=934875',
  'limit=20'
];
var req = new XMLHttpRequest();
req.onerror = function() {
  // erro
};
req.onreadystatechange = function() {
  if (req.readyState == 4) {
    // sucesso
  }
};
req.open('POST', url, true);
req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

```
  req.setRequestHeader('Content-Length', params.length);
  req.send(params.join('&'));
```

Como podemos ver no exemplo, nada será feito caso a postagem falhe. Não há problema nisso quando a XHR é usada para capturar estatísticas abrangentes, mas se for crucial que os dados cheguem até o servidor, deve-se acrescentar códigos adicionais que provoquem novas tentativas em caso de fracasso:

```
function xhrPost(url, params, callback) {
  var req = new XMLHttpRequest();
  req.onerror = function() {
    setTimeout(function() {
      xhrPost(url, params, callback);
    }, 1000);
  };
  req.onreadystatechange = function() {
    if (req.readyState == 4) {
      if (callback && typeof callback === 'function') {
        callback();
      }
    }
  };
  req.open('POST', url, true);
  req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
  req.setRequestHeader('Content-Length', params.length);
  req.send(params.join('&'));
}
```

Quando utilizar XHR no envio de dados de volta ao servidor, prefira o GET. Para pequenas quantidades de dados, uma solicitação GET será enviada ao servidor em um único pacote. Uma solicitação POST, por outro lado, será enviada em ao menos dois pacotes, um para os cabeçalhos e outro para o corpo do POST. Um POST é mais adequado para envio de grandes quantidades de dados ao servidor, casos em que o pacote extra não será tão significativo. Da mesma forma, o limite de comprimento de URL do Internet Explorer torna longas solicitações GET impossíveis.

Beacons

Essa técnica é muito semelhante à inserção dinâmica da tag `script`.

Nela, JavaScript é utilizado para criar um novo objeto Image, com a propriedade src definida como a URL de um script em seu servidor. Essa URL contém os dados que desejamos enviar de volta no formato GET de pares chave-valor. Observe que nenhum elemento img tem de ser criado ou inserido no DOM.

```
var url = '/status_tracker.php';
var params = [
  'step=2',
  'time=1248027314'
];
(new Image()).src = url + '?' + params.join('&');
```

O servidor toma esses dados e os armazena; ele não tem de enviar nada de volta ao cliente, já que a imagem não é exibida. Esse é o modo mais eficiente para envio de informações de volta ao servidor. Há pouco custo, e os erros que venham a ocorrer no lado servidor não afetam de modo algum o lado cliente.

A simplicidade dos beacons de imagem também indica a presença de restrições quanto ao que pode ser feito. Você não pode enviar dados POST, ficando limitado a um número bem pequeno de caracteres antes que alcance o limite máximo de tamanho da URL. Dados podem ser recebidos de volta, mas apenas por meios limitados. A escuta pelo evento load do objeto Image, que sinalizará o recebimento bem-sucedido dos dados, é possível. Você também pode verificar a largura e altura da imagem que o servidor retorna (caso seja retornada) e utilizar esses números para informar o estado do servidor. Por exemplo, uma largura de 1 pode ser “sucesso”, enquanto um valor 2 pode representar “tente novamente”.

Caso você não tenha de retornar dados em sua resposta, envie um código 204 No Content e nenhum corpo na mensagem. Isso impedirá que o cliente fique esperando por um corpo de mensagem que nunca chegará:

```
var url = '/status_tracker.php';
var params = [
  'step=2',
```

```
  'time=1248027314'
];
var beacon = new Image();
beacon.src = url + '?' + params.join('&');
beacon.onload = function() {
  if (this.width == 1) {
    // sucesso
  }
  else if (this.width == 2) {
    // fracasso; crie um novo beacon e tente novamente
  }
};
beacon.onerror = function() {
  // erro; espere um pouco, crie outro beacon e tente novamente
};
```

Beacons são o meio mais rápido e eficiente de enviar dados de volta ao servidor. O servidor não tem de enviar nenhum corpo (body) em sua resposta, por isso, você não precisa se preocupar em baixar dados no cliente. O único lado negativo é que o número de respostas que podem ser recebidas é limitado. Caso passe grandes quantidades de dados de volta ao cliente, use XHR. Se apenas se importar com o envio de dados ao servidor (possivelmente com uma resposta bem diminuta), utilize os beacons de imagens.

Formatos de dados

Ao considerar as técnicas de transmissão de dados, você deve considerar vários fatores: o grupo de funções disponíveis, a compatibilidade, o desempenho e a direção (de ou para o servidor). Ao considerar formatos de dados, a única escala necessária para comparação é a velocidade.

Não há um formato de dados que seja sempre melhor que os outros. Dependendo de quais dados serão transferidos e de seu uso pretendido na página, um poderá ser mais rápido para baixar, enquanto outro poderá ser mais rápido em termos de parsing. Nesta seção, criaremos um widget[2] para busca entre os usuários e o implementaremos utilizando cada um dos quatro principais formatos de dados. Isso exigirá a formatação de uma lista de usuários no

servidor, sua transmissão de volta ao navegador, o parsing da lista em uma estrutura de dados JavaScript nativa e a pesquisa, nesta, por uma dada string. Cada um dos formatos de dados será comparado com base no tamanho do arquivo da lista, na velocidade de parsing e na facilidade com que ele é formado no servidor.

XML

Quando o Ajax tornou-se popular, o XML era o formato preferido de dados. Havia muito a ser elogiado: extrema interoperabilidade (com suporte excelente tanto no lado servidor quanto no cliente), formatação estrita e fácil validação. O JSON ainda não tinha sido formalizado como um formato de troca e praticamente todas as linguagens utilizadas em servidores tinham uma biblioteca disponível para o trabalho com XML.

Confira um exemplo com a codificação em XML de nossa lista de usuários:

```
<?xml version="1.0" encoding='UTF-8'?>
<users total="4">
  <user id="1">
    <username>alice</username>
    <realname>Alice Smith</realname>
    <email>alice@alicesmith.com</email>
  </user>
  <user id="2">
    <username>bob</username>
    <realname>Bob Jones</realname>
    <email>bob@bobjones.com</email>
  </user>
  <user id="3">
    <username>carol</username>
    <realname>Carol Williams</realname>
    <email>carol@carolwilliams.com</email>
  </user>
  <user id="4">
    <username>dave</username>
    <realname>Dave Johnson</realname>
    <email>dave@davejohnson.com</email>
  </user>
</users>
```

Em comparação aos outros formatos, o XML é extremamente verboso. Cada pedaço de dados demanda muita estrutura, o que torna a proporção dados-estrutura extremamente baixa. O XML também apresenta uma sintaxe levemente ambígua. Ao codificar uma estrutura de dados em XML devemos tornar os parâmetros dos objetos atributos do elemento objeto ou elementos filhos independentes? Devemos utilizar nomes longos e descritivos para as tags ou nomes pequenos que sejam eficientes, mas indecifráveis? O parsing desta sintaxe é igualmente ambíguo, sendo necessário saber com antecedência o layout de uma resposta XML para poder entendê-la.

Em geral, o parsing de textos XML demanda muito esforço da parte do programador JavaScript. Além de saber os detalhes da estrutura antecipadamente, também é preciso saber exatamente como quebrar e depois remontar essa estrutura em algo que se pareça com um objeto JavaScript. Não se trata de um processo automático, diferente de todos os outros três formatos de dados.

Veja um exemplo de como podemos fazer o parsing dessa resposta particular XML em um objeto:

```
function parseXML(responseXML) {
  var users = [];
  var userNodes = responseXML.getElementsByTagName('users');
  var node, usernameNodes, usernameNode, username,
    realnameNodes, realnameNode, realname,
    emailNodes, emailNode, email;
  for (var i = 0, len = userNodes.length; i < len; i++) {
    node = userNodes[i];
    username = realname = email = '';
    usernameNodes = node.getElementsByTagName('username');
    if (usernameNodes && usernameNodes[0]) {
      usernameNode = usernameNodes[0];
      username = (usernameNodes.firstChild) ?
        usernameNodes.firstChild.nodeValue : '';
    }
    realnameNodes = node.getElementsByTagName('realname');
    if (realnameNodes && realnameNodes[0]) {
      realnameNode = realnameNodes[0];
```

```
      realname = (realnameNodes.firstChild) ?
        realnameNodes.firstChild.nodeValue : '';
    }
    emailNodes = node.getElementsByTagName('email');
    if (emailNodes && emailNodes[0]) {
      emailNode = emailNodes[0];
      email = (emailNodes.firstChild) ?
        emailNodes.firstChild.nodeValue : '';
    }
    users[i] = {
      id: node.getAttribute('id'),
      username: username,
      realname: realname,
      email: email
    };
  }
  return users;
}
```

Como podemos ver, temos que conferir a existência de cada tag antes que seu valor seja lido. É uma abordagem muito dependente da estrutura do XML.

Uma abordagem mais eficiente seria a codificação de cada um desses valores como um atributo da tag <user>. Com isso, temos um arquivo de tamanho menor para o mesmo volume de dados. A seguir, podemos ver um exemplo da lista de usuários quando codificamos os valores como atributos:

```
<?xml version="1.0" encoding='UTF-8'?>
<users total="4">
  <user id="1-id001" username="alice" realname="Alice Smith"
    email="alice@alicesmith.com" />
  <user id="2-id001" username="bob" realname="Bob Jones"
    email="bob@bobjones.com" />
  <user id="3-id001" username="carol" realname="Carol Williams"
    email="carol@carolwilliams.com" />
  <user id="4-id001" username="dave" realname="Dave Johnson"
    email="dave@davejohnson.com" />
</users>
```

A análise dessa resposta XML simplificada é consideravelmente mais fácil:

```
function parseXML(responseXML) {
```

```
  var users = [];
  var userNodes = responseXML.getElementsByTagName('users');
  for (var i = 0, len = userNodes.length; i < len; i++) {
    users[i] = {
      id: userNodes[i].getAttribute('id'),
      username: userNodes[i].getAttribute('username'),
      realname: userNodes[i].getAttribute('realname'),
      email: userNodes[i].getAttribute('email')
    };
  }
  return users;
}
```

XPath

Ainda que esteja além do escopo deste capítulo, XPath pode ser muito mais rápida do que o método getElementsByTagName no parsing de um documento XML. Seu ponto negativo é o fato de que não é universalmente aceita, assim você também deve escrever outro código como segunda opção utilizando o estilo mais antigo da travessia DOM. No momento, o XPath Nível 3 do DOM foi implementado pelo Firefox, pelo Safari, pelo Chrome e pelo Opera. O Internet Explorer 8 apresenta uma interface semelhante, mas um pouco menos avançada.

Tamanho de resposta e tempo de parsing

Vamos examinar os números de desempenho para o XML na tabela seguinte.

Formato	Tamanho	Tempo de download	Tempo de parsing	Tempo total de carregamento
XML detalhado	582.960 bytes	999,4 ms	343,1 ms	1342,5 ms
XML simplificado	437.960 bytes	475,1 ms	83,1 ms	558,2 ms

Cada um dos tipos de dados foi testado utilizando listas de usuários com comprimentos de 100, 500, 1.000 e 5.000 itens. Cada lista foi baixada e teve seu parsing feito dez vezes no mesmo navegador, e médias foram tomadas para cada tempo, tempo de parsing e tamanho de arquivo. Resultados completos para todos os formatos de dados e técnicas de transferência, bem como testes que você mesmo pode executar, podem ser encontrados em *http://techfoolery.com/formats/*.

Como podemos ver, ao favorecer atributos no lugar de tags filhas temos como resultado um menor tamanho de arquivo e um tempo de parsing bem mais rápido. Isso porque nesse caso o DOM da estrutura XML não tem de ser percorrido tantas vezes, tendo apenas que ler os atributos.

Deve você considerar a utilização de XML? Dada sua prevalência em APIs públicas, muitas vezes você simplesmente não terá outra escolha. Se os dados estão disponíveis apenas em XML, arregace as mangas e escreva o código para seu parsing. Mas se houver outro formato de dados disponível, prefira essa alternativa. Os números de desempenho que você vê aqui para os XMLs detalhados são extremamente lentos quando comparados às técnicas mais avançadas. Para navegadores que a aceitam, XPath pode melhorar o tempo de parsing, mas ao custo de termos de escrever e manter três caminhos separados de código (um para navegadores que aceitam o XPath Nível3 do DOM, outro para o Internet Explorer 8 e um terceiro para todos os outros navegadores). A comparação com o formato XML simples é favorável, mas ainda consideravelmente mais lenta do que o formato mais veloz. Não há lugar para XML quando falamos em Ajax de alto desempenho.

JSON

Formalizado e popularizado por Douglas Crockford, o JSON é um formato de dados leve e de parsing facilitado que utiliza uma sintaxe literal de objetos e arrays JavaScript. Confira um exemplo da lista de usuários escrita em JSON:

```
[
  { "id": 1, "username": "alice", "realname": "Alice Smith",
    "email": "alice@alicesmith.com" },
  { "id": 2, "username": "bob", "realname": "Bob Jones",
    "email": "bob@bobjones.com" },
  { "id": 3, "username": "carol", "realname": "Carol Williams",
    "email": "carol@carolwilliams.com" },
  { "id": 4, "username": "dave", "realname": "Dave Johnson",
    "email": "dave@davejohnson.com" }
]
```

Os usuários são representados como objetos e a lista de usuários é um array, assim como qualquer outro array ou objeto seria escrito em JavaScript. Isso significa que, quando utilizados junto a um método eval() ou envoltos em uma função callback, dados JSON são código JavaScript executável. Fazer o parsing de uma string JSON em JavaScript é tão simples quanto usar a função eval():

```
function parseJSON(responseText) {
  return eval('(' + responseText + ')');
}
```

Uma nota sobre JSON e eval: o uso do eval em seu código é perigoso, especialmente na avaliação de dados JSON de terceiros (que podem conter código malicioso ou defeituoso). Sempre que possível, prefira o método JSON.parse() para fazer o parsing da string de modo nativo. Esse método detectará erros de sintaxe dentro do JSON e permitirá a passagem de uma função capaz de filtrar ou transformar os resultados. Atualmente ele é implementado no Firefox 3.5, no Internet Explorer 8 e no Safari 4. A maioria das bibliotecas JavaScript contém código para parsing de JSON que chamará a versão nativa, se presente, ou uma versão um pouco menos encorpada não nativa. Uma implementação de referência de uma versão não nativa pode ser encontrada em *http://json.org/json2.js*. Por questões de consistência, utilizaremos o eval no exemplo do código.

Assim como no caso do XML, podemos destilar esse formato em uma versão ainda mais simples. Neste caso, substituímos os nomes dos atributos por versões abreviadas (menos inteligíveis):

```
[
  { "i": 1, "u": "alice", "r": "Alice Smith", "e": "alice@alicesmith.com" },
  { "i": 2, "u": "bob", "r": "Bob Jones", "e": "bob@bobjones.com" },
  { "i": 3, "u": "carol", "r": "Carol Williams",
    "e": "carol@carolwilliams.com" },
  { "i": 4, "u": "dave", "r": "Dave Johnson", "e": "dave@davejohnson.com" }
]
```

Temos os mesmos dados como resultado, com uma estrutura um pouco menor e menos bytes a serem transmitidos para o navegador. Podemos ainda ir um passo adiante, removendo completamente os

nomes dos atributos. A inteligibilidade desse formato será menor se comparada à dos outros dois, mas o tamanho do arquivo será bem mais diminuto: quase metade do tamanho apresentado pelo JSON detalhado.

```
[
  [ 1, "alice", "Alice Smith", "alice@alicesmith.com" ],
  [ 2, "bob", "Bob Jones", "bob@bobjones.com" ],
  [ 3, "carol", "Carol Williams", "carol@carolwilliams.com" ],
  [ 4, "dave", "Dave Johnson", "dave@davejohnson.com" ]
]
```

A realização do parsing com sucesso desses dados exige que a ordem dos termos seja mantida. Dito isso, podemos facilmente convertê-lo em um formato que contenha os mesmos nomes de atributos apresentados no primeiro formato JSON:

```
function parseJSON(responseText) {
  var users = [];
  var usersArray = eval('(' + responseText + ')');
  for (var i = 0, len = usersArray.length; i < len; i++) {
    users[i] = {
      id: usersArray[i][0],
      username: usersArray[i][1],
      realname: usersArray[i][2],
      email: usersArray[i][3]
    };
  }
  return users;
}
```

Nesse exemplo, usamos `eval()` para converter a string em um array JavaScript nativo. Esse array de arrays é, em seguida, convertido em um array de objetos. Essencialmente, estamos trocando um tamanho menor de arquivo e um tempo de `eval()` mais rápido por uma função de parsing mais complicado. A tabela seguinte lista os números de desempenho para os três formatos JSON, quando transferidos por meio de XHR.

Formato	Tamanho	Tempo de download	Tempo de parsing	Tempo total de carregamento
JSON detalhado	487.895 bytes	527,7 ms	26,7 ms	554,4 ms

JSON simples	392.895 bytes	498,7 ms	29,0 ms	527,7 ms
JSON de arrays	292.895 bytes	305,4 ms	18,6 ms	324,0 ms

A opção com JSON utilizando arrays vence em todas as categorias, tendo o menor tamanho de arquivo, o tempo médio de download mais rápido e o melhor tempo médio de parsing. Apesar da função de parsing ter de iterar por todas as 5.000 entradas contidas na lista, ela ainda é 30% mais rápida em termos de parsing.

JSON-P

O fato de o JSON poder ser executado de modo nativo tem várias implicações importantes sobre o desempenho.

Quando XHR é utilizado, dados JSON são retornados como uma string. Essa string é então avaliada usando `eval()` para ser convertida em um objeto nativo. Entretanto, quando é utilizada a inserção dinâmica da tag `script`, dados JSON são tratados como qualquer outro tipo de arquivo JavaScript, sendo executados como código nativo. Para fazê-lo, os dados devem estar envoltos em uma função callback. Isso é conhecido como “JSON com enchimento” (*JSON with padding*) ou JSON-P. A seguir temos a lista de usuários formatada como JSON-P:

```
parseJSON([
  { "id": 1, "username": "alice", "realname": "Alice Smith",
    "email": "alice@alicesmith.com" },
  { "id": 2, "username": "bob", "realname": "Bob Jones",
    "email": "bob@bobjones.com" },
  { "id": 3, "username": "carol", "realname": "Carol Williams",
    "email": "carol@carolwilliams.com" },
  { "id": 4, "username": "dave", "realname": "Dave Johnson",
    "email": "dave@davejohnson.com" }
]);
```

O JSON-P acresce um pouco ao tamanho do arquivo com a utilização do envoltório (wrapper) callback, mas tal acréscimo é insignificante quando comparado ao melhor tempo de parsing. Uma vez que os dados são tratados como JavaScript nativo, seu parsing também é feito com velocidade de JavaScript nativo. Aqui estão os

mesmos formatos JSON quando transmitidos como JSON-P.

Formato	Tamanho	Tempo de download	Tempo de parsing	Tempo total de carregamento
JSON-P detalhado	487.913 bytes	598,2 ms	0,0 ms	598,2 ms
JSON-P simples	392.913 bytes	454,0 ms	3,1 ms	457,1 ms
JSON-P de arrays	292.912 bytes	316,0 ms	3,4 ms	319,4 ms

O tamanho dos arquivos e os tempos de download são praticamente idênticos aos apresentados nos testes XHR, mas os tempos de parsing são quase dez vezes mais rápidos. No caso do JSON-P detalhado este tempo é zero, já que não é necessária nenhuma análise: os dados já se encontram em formato nativo. O mesmo vale para o formato JSON-P simples e JSON-P de arrays, mas cada um ainda tem de ser iterado para que seja convertido ao formato que o JSON-P detalhado produz naturalmente.

O formato JSON mais rápido é o JSON-P utilizando arrays. Ainda que pareça somente um pouco mais rápido do que o JSON transmitido utilizando XHR, sua diferença aumenta conforme cresce o tamanho da lista. Caso esteja trabalhando em um projeto que exija uma lista de 10.000 ou 100.000 elementos, prefira o JSON-P em lugar do JSON.

Existe um motivo para evitar a utilização do JSON-P, que nada tem a ver com o desempenho: uma vez que o JSON-P deve ser formado por JavaScript executável, ele pode ser chamado por qualquer pessoa e incluído em qualquer website usando a inserção dinâmica da tag `script`. O JSON por outro lado, não é JavaScript válido até que tenha sido submetido a `eval`, podendo ser buscado apenas como uma string utilizando XHR. Não codifique dados sensíveis em JSON-P, já que não há como garantir que continuarão privados, mesmo com URLs randômicas e cookies.

Como saber se devo utilizar JSON?

O formato JSON tem muitas vantagens em comparação ao XML.

Primeiro, é um formato muito menor. Sua estrutura também ocupa um volume mais discreto, deixando mais espaço para os dados, especialmente quando contêm arrays em vez de objetos. Também é extremamente interoperável, com bibliotecas de codificação e decodificação disponíveis para a maioria das linguagens do lado servidor. Seu parsing no lado cliente também não é difícil, permitindo que você passe mais tempo escrevendo códigos que realmente atuem sobre os dados. Para os desenvolvedores web é ainda mais importante o fato de se tratar do formato de melhor desempenho, tanto por ser relativamente pequeno quanto por ser processado com maior rapidez. Por tudo isso, o JSON é uma das peças fundamentais da utilização em alto desempenho do Ajax, especialmente quando posto em prática junto à inserção dinâmica da tag script.

HTML

Muitas vezes os dados que você solicita serão transformados em HTML para exibição na página. A conversão de uma grande estrutura de dados em HTML simples ocorre com relativa rapidez em JavaScript, mas pode ser mais rápida no servidor. Uma técnica a ser considerada é a formação de todo o HTML no servidor, seguida por sua transmissão de modo intacto ao cliente. Dessa forma, o JavaScript é capaz de simplesmente encaixar o código utilizando a propriedade innerHTML. Veja um exemplo da lista de usuários codificada como HTML:

```
<ul class="users">
  <li class="user" id="1-id002">
    <a href="http://www.site.com/alice/" class="username">alice</a>
    <span class="realname">Alice Smith</span>
    <a href="mailto:alice@alicesmith.com"
      class="email">alice@alicesmith.com</a>
  </li>
  <li class="user" id="2-id002">
    <a href="http://www.site.com/bob/" class="username">bob</a>
    <span class="realname">Bob Jones</span>
    <a href="mailto:bob@bobjones.com" class="email">bob@bobjones.com</a>
  </li>
```

```
    <li class="user" id="3-id002">
      <a href="http://www.site.com/carol/" class="username">carol</a>
      <span class="realname">Carol Williams</span>
      <a href="mailto:carol@carolwilliams.com"
        class="email">carol@carolwilliams.com</a>
    </li>
    <li class="user" id="4-id002">
      <a href="http://www.site.com/dave/" class="username">dave</a>
      <span class="realname">Dave Johnson</span>
      <a href="mailto:dave@davejohnson.com"
        class="email">dave@davejohnson.com</a>
    </li>
  </ul>
```

O problema dessa técnica é que o HTML é um formato de dados muito verboso, até mais que o XML. Imagine que sobre os dados que vimos você poderia ainda ter aninhado tags HTML, cada uma com suas IDs, classes e outros atributos. É possível que a formatação HTML chegue a ocupar mais espaço do que os dados em si, ainda que isso possa ser mitigado pelo uso do mínimo possível de tags e atributos. Justamente por isso, adote essa técnica apenas quando a CPU do lado cliente for mais limitada que a largura de banda.

Em um extremo temos o JSON, o formato que exige a menor quantidade de estrutura para o parsing de dados no lado cliente. Seu download no lado cliente é extremamente rápido; entretanto, ele consome muito tempo da CPU para ser convertido em HTML e exibido na página. Muitas operações com strings são necessárias, e estas consistem em uma das coisas mais demoradas que podem ser feitas em JavaScript.

No outro extremo, temos o HTML criado no servidor. Um formato de tamanho maior e tempo de download muito mais custoso, mas que uma vez baixado, exige apenas uma única operação para ser exibido na página:

```
document.getElementById('data-container').innerHTML = req.responseText;
```

A tabela seguinte mostra os números do desempenho para a lista de usuário quando ela é codificada com HTML. Lembre-se da

diferença principal entre esse formato e os demais: o “parsing” nesse caso se refere à simples ação de inserir o HTML no DOM. Da mesma forma, o HTML também não pode ser iterado com tanta facilidade e rapidez, ao contrário, por exemplo, de um array JavaScript nativo.

Formato	Tamanho	Tempo de download	Tempo de parsing	Tempo total de carregamento
HTML	1.063.416 bytes	273,1 ms	121,4 ms	394,5 ms

Como podemos ver, o HTML é significativamente maior e também leva um bom tempo para seu parsing. A simples operação de inseri-lo no DOM é ilusoriamente simples; apesar de se tratar de uma única linha de código, ainda é preciso uma quantidade significativa de tempo para que todos esses dados sejam carregados na página. Não podemos comparar facilmente esses números aos outros, já que o resultado final não é um array de dados, mas elementos HTML exibidos em uma página. Mesmo assim, eles são capazes de ilustrar o fato de que HTML, como formato de dados, é lento e inchado.

Formatação personalizada

O formato ideal deve incluir o mínimo de estrutura possível, mas ainda ser capaz de permitir que você separe os campos individuais entre si. Tal formato pode ser criado com facilidade simplesmente concatenando seus dados com um caractere separador:

```
Jacob;Michael;Joshua;Matthew;Andrew;Christopher;Joseph;Daniel;Nicholas;
Ethan;William;Anthony;Ryan;David;Tyler;John
```

Tais separadores criam essencialmente um array de dados, semelhante a uma lista separada por vírgulas. Utilizando separadores diferentes, você é capaz de criar arrays multidimensionais. Veja nossa lista de usuários quando codificada em um formato personalizado delimitado por caracteres:

```
1:alice:Alice Smith:alice@alicesmith.com;
2:bob:Bob Jones:bob@bobjones.com;
3:carol:Carol Williams:carol@carolwilliams.com;
```

```
4:dave:Dave Johnson:dave@davejohnson.com
```

Esse tipo de formato é extremamente conciso e oferece uma razão dados-estrutura bem elevada (consideravelmente maior que qualquer outro formato, com exceção de texto puro). Formatos personalizados são mais rápidos de baixar e mais rápidos e fáceis de analisar e processar: basta chamar `split()` sobre a string, utilizando o separador como argumento. Formatos personalizados mais complexos, com vários separadores, exigem loops para separar todos os dados (mas lembre-se de que esses loops são extremamente rápidos em JavaScript). O `split()` é uma das operações sobre strings mais ágeis, sendo capaz de lidar com listas delimitadas por separadores formadas com mais de 10.000 elementos em questão de milissegundos. Confira um exemplo do parsing do formato precedente:

```
function parseCustomFormat(responseText) {
  var users = [];
  var usersEncoded = responseText.split(';');
  var userArray;
  for (var i = 0, len = usersEncoded.length; i < len; i++) {
    userArray = usersEncoded[i].split(':');
    users[i] = {
      id: userArray[0],
      username: userArray[1],
      realname: userArray[2],
      email: userArray[3]
    };
  }
  return users;
}
```

Ao criar seu próprio formato personalizado, uma das decisões mais importantes que devem ser tomadas é a escolha do separador. Idealmente, cada separador deve ser um único caractere, não encontrado naturalmente em seus dados. Caracteres ASCII de número baixo funcionam bem e são facilmente representados na maioria das linguagens do lado servidor. Por exemplo, veja como você utilizaria caracteres ASCII em PHP:

```
function build_format_custom($users) {
```

```
  $row_delimiter = chr(1); // \u0001 em JavaScript
  $field_delimiter = chr(2); // \u0002 em JavaScript
  $output = array();
  foreach ($users as $user) {
    $fields = array($user['id'], $user['username'], $user['realname'], $user['email']);
    $output[] = implode($field_delimiter, $fields);
  }
  return implode($row_delimiter, $output);
}
```

Esses caracteres de controle são representados em JavaScript utilizando a notação Unicode (p. ex., \u0001). A função split() pode receber tanto uma string quanto uma expressão regular como argumentos. Caso seja provável que existam campos em branco em seus dados, utilize uma string; no IE, quando o delimitador é passado como uma expressão regular, o split() ignora o segundo delimitador se dois existirem lado a lado. Os dois tipos de argumentos são equivalentes nos outros navegadores.

```
// delimitador de expressão regular
var rows = req.responseText.split(/\u0001/);
// delimitador de string (mais seguro)
var rows = req.responseText.split("\u0001");
```

Os valores de desempenho para um formato personalizado delimitado com caracteres, utilizando tanto XHR quanto a inserção dinâmica da tag script, são listados a seguir.

Formato	Tamanho	Tempo de download	Tempo de parsing	Tempo total de carregamento
Formato personalizado (XHR)	222.892 bytes	63,1 ms	14,5 ms	77,6 ms
Formato personalizado (inserção de script)	222.912 bytes	66,3 ms	11,7 ms	78,0 ms

Tanto a técnica XHR quanto a inserção dinâmica da tag script podem ser usadas nesse formato. Uma vez que o parsing da resposta é feito como uma string em ambos os casos, não há diferença real no desempenho. Para séries de dados muito grandes, este é o melhor formato, batendo até o JSON executado nativamente em velocidade de parsing e no tempo geral de carregamento. Nesse formato, o envio de enormes quantidades de dados ao lado cliente em um

curto intervalo de tempo torna-se possível.

Conclusões quanto aos formatos de dados

Prefira formatos leves para seus dados; os melhores são o JSON e um formato personalizado delimitado por caracteres. Caso o grupo de dados seja grande, fazendo com que o tempo de parsing se torne um problema, utilize uma destas duas técnicas:

- Dados JSON-P, adquiridos utilizando a inserção dinâmica da tag script. Dessa forma, os dados são tratados como JavaScript executável, não como strings, permitindo um parsing mais rápido. Essa abordagem pode ser adotada em domínios diferentes, mas não deve ser aplicada a dados privados.
- Um formato personalizado de dados, delimitado por caracteres, que seja buscado utilizando XHR ou a inserção dinâmica da tag script e que tenha seu parsing feito com `split()`. Essa técnica é capaz de realizar o parsing de conjuntos de dados extremamente extensos, levemente mais rápido do que a técnica JSON-P. Além disso, ela costuma apresentar um tamanho de arquivo menor.

A tabela seguinte e a figura 7.1 mostram novamente todos os números de desempenho (na ordem do mais lento para o mais rápido), permitindo comparar todos os formatos em um único local. O formato HTML foi excluído por não poder ser comparado diretamente com os outros formatos.

Tenha em mente que esses números dizem respeito a um único teste realizado em um único navegador. Os resultados devem ser utilizados apenas como indicadores gerais de desempenho, e não como números absolutos. Execute você mesmo seus testes em *http://techfoolery.com/formats/*.

Formato	Tamanho	Tempo de download	Tempo de parsing	Tempo total de carregamento
XML detalhado	582.960 bytes	999,4 ms	343,1 ms	1.342,5 ms
JSON-P detalhado	487.913 bytes	598,2 ms	0,0 ms	598,2 ms
XML simplificado	437.960	475,1 ms	83,1 ms	558,2 ms

	bytes			
JSON detalhado	487.895 bytes	527,7 ms	26,7 ms	554,4 ms
JSON simplificado	392.895 bytes	498,7 ms	29,0 ms	527,7 ms
JSON-P simplificado	392.913 bytes	454,0 ms	3,1 ms	457,1 ms
JSON de arrays	292.895 bytes	305,4 ms	18,6 ms	324,0 ms
JSON-P de arrays	292.912 bytes	316,0 ms	3,4 ms	319,4 ms
Formato personalizado (inserção de script)	222.912 bytes	66,3 ms	11,7 ms	78,0 ms
Formato personalizado (XHR)	222.892 bytes	63,1 ms	14,5 ms	77,6 ms

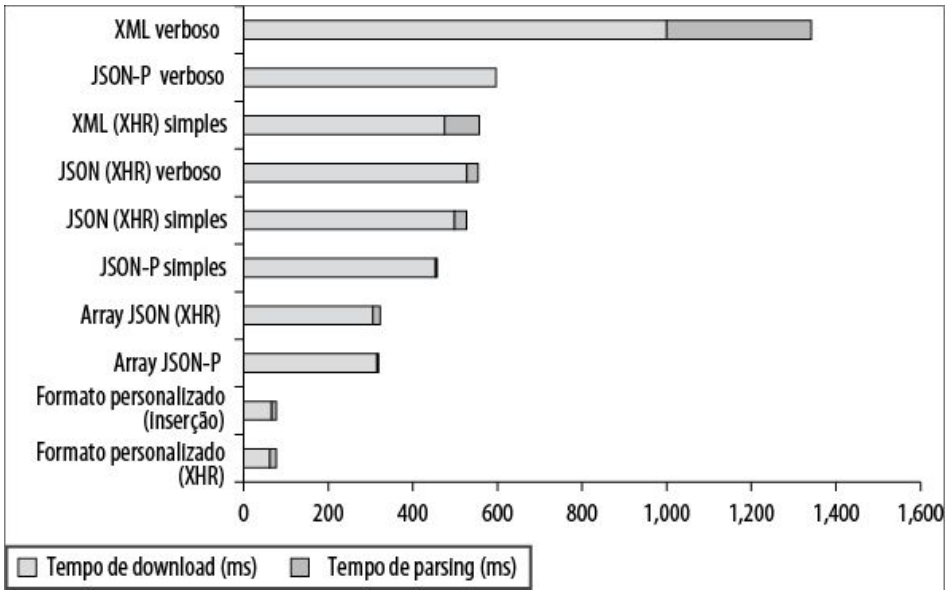


Figura 7.1 – Uma comparação dos tempos de download e de parsing dos diferentes formatos de dados.

Diretrizes de desempenho do Ajax

Uma vez que tenha selecionado a técnica mais apropriada para transmissão de seus dados, você deve começar a considerar outras técnicas de otimização. Como elas costumam depender muito de casos individuais, assegure-se de que sua aplicação se encaixa nos perfis descritos antes de considerá-las seriamente.

Dados em cache

A mais rápida solicitação Ajax é uma que não tem de ser feita.

Existem dois modos de prevenir a ocorrência de uma solicitação desnecessária:

- No lado servidor, criando cabeçalhos HTTP que garantam o armazenamento de sua resposta no cache do navegador.
- No lado cliente, armazenando localmente os dados buscados para que não tenham de ser solicitados novamente.

A primeira técnica é mais fácil de ser configurada e mantida, enquanto a segunda é capaz de fornecer o mais alto grau de controle.

Definição de cabeçalhos HTTP

Caso deseje que suas respostas Ajax sejam armazenadas em cache pelo navegador, você terá de utilizar o GET para fazer a solicitação. Todavia, isso não basta; você também terá de enviar os cabeçalhos HTTP corretos junto à resposta. O cabeçalho Expires informa ao navegador até quando uma resposta deve ser armazenada. Seu valor deve ser uma data; depois que esta expirar, qualquer solicitação por essa URL deixará de ser entregue a partir do cache, sendo passada ao servidor. A seguir, temos a aparência comum de um cabeçalho Expires:

```
Expires: Mon, 28 Jul 2014 23:30:00 GMT
```

Esse cabeçalho específico informa ao navegador que essa resposta deve ser armazenada em cache até julho de 2014. Chama-se isso de cabeçalho Expires de futuro distante, sendo adequado para o armazenamento de conteúdo que nunca sofrerá alteração, como imagens e conjuntos estáticos de dados.

A data de um cabeçalho Expires é uma data GMT. Ela pode ser configurada em PHP com este código:

```
$lifetime = 7 * 24 * 60 * 60; // 7 dias, em segundos
header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

Isso solicitará a seu navegador que armazene o arquivo por 7 dias. Para criar um cabeçalho Expires de futuro distante, defina a duração como algo maior; o próximo exemplo solicita que o navegador

armazene o arquivo em cache por 10 anos:

```
$lifetime = 10 * 365 * 24 * 60 * 60; // 10 anos, em segundos
header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $lifetime) . ' GMT');
```

Um cabeçalho `Expires` é a forma mais fácil de garantir o armazenamento em cache de nossas respostas Ajax pelo navegador. Para isso, não é necessária nenhuma alteração no código do lado cliente, além de ser possível continuar fazendo solicitações Ajax normalmente, sabendo que o navegador enviará a solicitação ao servidor apenas se o arquivo não estiver dentro do cache. Também é muito fácil fazer sua implementação no lado servidor, já que todas as linguagens permitem a definição de alguma forma dos cabeçalhos. Essa é a abordagem mais simples para garantir que os dados sejam armazenados em cache.

Armazenamento local de dados

Em vez de depender do navegador para cuidar do armazenamento em cache, também é possível fazê-lo de modo manual, armazenando as respostas recebidas a partir do servidor. Isso pode ser feito colocando o texto da resposta dentro de um objeto, indicado pela URL utilizada para adquiri-lo. Veja um exemplo de um envoltório (wrapper) XHR que primeiro confere se uma URL já foi adquirida:

```
var localCache = {};
function xhrRequest(url, callback) {
  // confere o cache local em busca desta URL
  if (localCache[url]) {
    callback.success(localCache[url]);
    return;
  }
  // caso esta URL não esteja no cache, uma solicitação é feita
var req = createXhrObject();
  req.onerror = function() {
    callback.error();
  };
  req.onreadystatechange = function() {
    if (req.readyState == 4) {
      if (req.responseText === '' || req.status == '404') {
```

```
        callback.error();
        return;
      }
      // armazena a resposta no cache local
      localCache[url] = req.responseText;
      callback.success(req.responseText);
      }
    };
    req.open("GET", url, true);
    req.send(null);
  }
```

Em geral, a definição de um cabeçalho Expires é a solução preferível. Além de mais fácil, ela também armazena respostas por vários carregamentos da página e por várias sessões. Todavia, um cache manual também pode ser útil quando você desejar expirá-lo programaticamente e adquirir dados novos. Imagine uma situação na qual você deseje utilizar dados armazenados para cada solicitação, exceto quando o usuário tomar uma ação que faça com que uma ou mais das respostas armazenadas se tornem inválidas. Nesse caso, a remoção das respostas do cache é bem simples:

```
delete localCache['/user/friendlist/'];
delete localCache['/user/contactlist/'];
```

Um cache local também funciona bem com usuários que navegam a partir de dispositivos móveis. A maioria dos navegadores presentes nesses dispositivos tem caches pequenos ou inexistentes, tornando os caches manuais sua melhor opção para prevenção de solicitações desnecessárias.

Conheça as limitações de sua biblioteca Ajax

Todas as bibliotecas JavaScript oferecem acesso a um objeto Ajax. Isso normaliza as diferenças entre os navegadores fornecendo uma interface constante e permitindo que você se foque em seu projeto, sem ter de se preocupar com os detalhes do funcionamento do XHR em algum navegador obscuro. Entretanto, ao oferecer uma interface unificada, essas bibliotecas também simplificam a interface, já que

nem todos os navegadores implementam todos os recursos. Por isso, muitas vezes você não terá acesso a toda a funcionalidade do XMLHttpRequest.

Algumas das técnicas mostradas neste capítulo podem ser implementadas apenas pelo acesso direto ao objeto XHR. Entre essas, a mais notável é a função de streaming da técnica XHR multiparte. Ao escutar pelo readyState 3, podemos começar a dividir uma resposta grande demais antes mesmo que ela seja completamente recebida. Com isso, somos capazes de trabalhar com os trechos da resposta em tempo real, o que torna essa técnica um dos motivos pelos quais a MXHR melhora tanto o desempenho de seu código. Todavia, a maioria das bibliotecas JavaScript não permite acesso direto ao evento readystatechange. Nesses casos, você terá de esperar até que toda a resposta tenha sido recebida (o que pode significar bastante tempo) antes que possa começar a utilizá-la.

A utilização direta do objeto XMLHttpRequest não é tão assustadora quanto parece. Pondo de lado algumas de suas peculiaridades, todas as versões mais recentes dos grandes navegadores aceitam o objeto XMLHttpRequest da mesma forma, oferecendo acesso aos diferentes readyStates. Com apenas algumas linhas extras de código, você também é capaz de oferecer suporte a versões mais antigas do IE. A seguir, temos um exemplo de uma função que retorna um objeto XHR, com o qual você pode interagir diretamente (uma versão modificada da utilizada pelo YUI 2 Connection Manager):

```
function createXhrObject() {
  var msxml_progid = [
    'MSXML2.XMLHTTP.6.0',
    'MSXML3.XMLHTTP',
    'Microsoft.XMLHTTP', // não oferece suporte ao readyState 3
    'MSXML2.XMLHTTP.3.0', // não oferece suporte ao readyState 3
  ];
  var req;
  try {
    req = new XMLHttpRequest(); // tenta primeiro o modo padrão
  }
```

```
  catch(e) {
    for (var i = 0, len = msxml_progid.length; i < len; ++i) {
      try {
        req = new ActiveXObject(msxml_progid[i]);
        break;
      }
      catch(e2) { }
    }
  }
  finally {
    return req;
  }
}
```

Com esse código, você tenta primeiro as versões do XMLHttp que aceitam o readyState 3, partindo em seguida para as que não aceitam caso as primeiras não estejam disponíveis.

A interação direta com o objeto XHR também reduz os custos representados pelas funções, melhorando seu desempenho. Apenas lembre-se de que ao abdicar do uso de uma biblioteca Ajax você poderá encontrar alguns problemas com navegadores mais antigos e obscuros.

Resumo

O uso de Ajax de alto desempenho exige que você saiba as necessidades específicas de sua situação e que selecione o formato de dados correto, junto à técnica de transmissão adequada.

Como formatos de dados, o texto puro e o HTML dependem muito de cada situação, mas são capazes de poupar ciclos da CPU no lado cliente. A linguagem XML está bem disponível, sendo aceita praticamente em todos os lugares, mas é extremamente verbosa e lenta em termos de parsing. O JSON é leve e de parsing rápido (quando tratado como código nativo e não como string), sendo quase tão interoperável quanto o XML. Formatos personalizados delimitados por caracteres são extremamente leves, apresentando também o menor tempo de parsing para grandes conjuntos de dados, mas podem demandar um trabalho de programação maior

no lado servidor e um tempo de parsing maior no lado cliente.

Na solicitação de dados, quando acessa os dados a partir do domínio da página, o XHR oferece o maior controle e flexibilidade. Por outro lado, trata todos os dados que chegam como uma string, potencialmente prejudicando o tempo de parsing. A inserção dinâmica da tag `script`, por sua vez, permite solicitações a domínios externos e execução nativa de JavaScript e JSON, ainda que ofereça uma interface menos encorpada e que não possa ler cabeçalhos ou códigos de resposta. A técnica XHR multiparte pode ser utilizada para reduzir o número de solicitações, sendo capaz de manipular tipos diferentes de arquivos em uma única resposta, ainda que não armazene em cache os recursos recebidos. No envio de dados, beacons de imagens representam uma abordagem simples e eficiente. A técnica XHR também pode ser utilizada para envio de grandes quantidades de dados em um método POST.

Além desses formatos e técnicas de transmissão existem várias diretrizes capazes de fazer com que seu Ajax apareça mais rapidamente:

- Reduza o número de solicitações feitas, pela concatenação de arquivos JavaScript e CSS ou pela utilização de MXHR.
- Melhore o tempo perceptível de carregamento de sua página utilizando Ajax para buscar arquivos menos importantes depois que o restante da página estiver carregado.
- Certifique-se de que seu código é capaz de lidar com problemas no lado servidor, além de prepará-lo para falhar sem prejudicar a experiência do usuário.
- Saiba quando deve ser utilizada uma biblioteca Ajax robusta e quando você deve escrever seu próprio código Ajax de baixo nível.

O Ajax oferece uma das maiores áreas para aumento potencial do desempenho de seu site, tanto pelo número de sites que utilizam solicitações assíncronas pesadamente quanto por oferecer soluções para problemas que não têm relação com ele, como a existência de

um número excessivo de recursos a ser carregado. A utilização criativa do XHR pode representar a diferença entre uma página lenta e não convidativa e uma que responda com rapidez e eficiência; pode ser a diferença entre um site com o qual os usuários odeiam interagir e um em que adoram navegar.

---

1 N.T.: Hack refere-se a uma solução engenhosa ou rápida a um problema de um programa de computador, ou a uma solução para um problema que pode ser vista como desajeitada ou deselegante (mas que, em geral, é relativamente rápida).(Fonte: Wikipédia)

2 N.T.: Widget é um bloco de código que pode ser instalado e executado dentro de qualquer página web separada, com base em HTML, por um usuário final sem que seja necessária qualquer compilação adicional. São derivados da ideia da reutilização de código (code reuse). Widgets normalmente tomam a forma de ferramentas on-screen (como relógios, contagens regressivas para eventos, tickers de leilões, tickers de bolsas de valores, informações sobre chegadas de voos, meteorologia diária etc.). (Fonte: Wikipédia)

CAPÍTULO 8

Práticas de programação

Toda linguagem de programação apresenta pontos problemáticos e padrões ineficientes que se desenvolvem com o passar do tempo. O aparecimento desses traços ocorre conforme as pessoas migram para a linguagem e começam a forçar seus limites. Desde 2005 quando o termo “Ajax” se tornou popular, os desenvolvedores têm estendido a aplicação do JavaScript para além do que antes parecia possível. Como resultado, alguns padrões muito específicos surgiram tanto como boas práticas quanto como práticas indesejáveis. Tal evolução é resultado da própria natureza do JavaScript na web.

Evite a avaliação dupla

O JavaScript, assim como muitas outras linguagens de codificação, permite que você tome uma string contendo código e execute-a dentro do próprio código. Há quatro formas padrão de fazê-lo: pelo uso de `eval()`, do construtor `Function()`, de `setTimeout()` e de `setInterval()`. Cada uma dessas funções permite que você passe e execute uma string de código JavaScript. Alguns exemplos:

```
var num1 = 5,
  num2 = 6,

// eval() avaliando uma string de código
result = eval("num1 + num2"),

// Function() avaliando uma string de código
sum = new Function("arg1", "arg2", "return arg1 + arg2");

// setTimeout() avaliando uma string de código
setTimeout("sum = num1 + num2", 100);

// setInterval() avaliando uma string de código
setInterval("sum = num1 + num2", 100);
```

Sempre que estiver avaliando seu código JavaScript de dentro do

próprio código, você incorrerá em uma penalidade por avaliação dupla. Esse código será primeiro avaliado normalmente e então avaliado uma vez mais para execução do código da string. A avaliação dupla é uma operação custosa e toma muito mais tempo do que a inclusão nativa do mesmo código.

Como uma comparação, o tempo gasto no acesso a um item de array costuma variar de navegador para navegador, mas varia de modo muito mais pronunciado quando o item de array é acessado utilizando eval(). Por exemplo:

```
// mais rápido
var item = array[0];

// mais lento
var item = eval("array[0]");
```

A diferença em todos os navegadores torna-se ainda mais drástica se 10.000 itens de array são lidos utilizando eval() em vez de código nativo. A tabela 8.1 mostra os intervalos de tempo gastos nessa operação.

Tabela 8.1 – Comparação da velocidade do código nativo versus eval() para acesso a 10.000 itens de array

Navegador	Código nativo (ms)	Código eval() (ms)
Firefox 3	10,57	822,62
Firefox 3.5	0,72	141,54
Chrome 1	5,7	106,41
Chrome 2	5,17	54,55
Internet Explorer 7	31,25	5086,13
Internet Explorer 8	40,06	420,55
Opera 9.64	2,01	402,82
Opera 10 Beta	10,52	315,16
Safari 3.2	30,37	360,6
Safari 4	22,16	54,47

Tamanha diferença no tempo de acesso aos itens de array ocorre devido à necessidade de criarmos uma nova instância interpretador/compilador sempre que eval() for chamada. O mesmo processo ocorre para Function(), setTimeout() e setInterval(), tornando a execução do código automaticamente mais lenta.

Na maior parte do tempo, não é preciso utilizar eval() ou Function(). Por isso, você deve evitá-las sempre que possível. Já para as outras duas funções, setTimeout() e setInterval(), procure sempre passar uma função como primeiro argumento e não uma string. Por exemplo:

```
setTimeout(function() {
  sum = num1 + num2;
}, 100);
setInterval(function() {
  sum = num1 + num2;
}, 100);
```

Evitar a avaliação dupla é algo vital para que se possa alcançar o melhor desempenho no tempo de execução de seu JavaScript.

 Engines otimizadoras de JavaScript podem muitas vezes armazenar em cache o resultado de avaliações de código repetidas feitas com eval(). Caso esteja avaliando repetidamente a mesma string, você verá grandes melhorias de desempenho no Safari 4 e em todas as versões do Chrome.

Utilize literais objeto/array

Há várias formas de criar objetos e arrays em JavaScript, mas nada é mais rápido do que a criação de literais objetos e arrays. Quando não utilizamos literais, a criação e designação típicas dos objetos têm a seguinte aparência:

```
// crie um objeto
var myObject = new Object();
myObject.name = "Nicholas";
myObject.count = 50;
myObject.flag = true;
myObject.pointer = null;
// crie um array
var myArray = new Array();
myArray[0] = "Nicholas";
myArray[1] = 50;
myArray[2] = true;
myArray[3] = null;
```

Ainda que tecnicamente não haja nada de errado com essa abordagem, literais são sempre avaliados mais rapidamente. Como bônus, ocupam um espaço menor no código, fazendo com que o

tamanho geral do arquivo também seja menor. O código precedente pode ser reescrito com literais da seguinte maneira:

```
// crie um objeto
var myObject = {
  name: "Nicholas",
  count: 50,
  flag: true,
  pointer: null
};
// crie um array
var myArray = ["Nicholas", 50, true, null];
```

O resultado final desse código é o mesmo da versão anterior, mas ele será executado mais rapidamente em todos os navegadores (com exceção do Firefox 3.5, que quase não apresenta diferença). Conforme aumenta o número de propriedades do objeto e de itens do array, também aumenta o benefício do uso de literais.

Não repita seu trabalho

Uma das principais técnicas para otimização do desempenho na ciência da computação em geral é a prevenção do trabalho. O conceito de prevenção do trabalho envolve, na verdade, dois mandamentos: não faça trabalho desnecessário e não repita trabalho que já foi executado. A primeira parte é normalmente mais fácil de ser identificada conforme o código é refatorado. Já a segunda – a não repetição do trabalho – é normalmente mais difícil de identificar, uma vez que a repetição pode ocorrer em inúmeros locais diferentes e por muitas razões.

Talvez o tipo mais comum de trabalho repetido seja a detecção do navegador. Muitos códigos apresentam bifurcações com base nos recursos dos navegadores. Considere a adição e remoção de manipuladores como um exemplo. Um código típico com essa finalidade poderia ter a seguinte aparência:

```
function addHandler(target, eventType, handler) {
  if (target.addEventListener) { // eventos DOM2
    target.addEventListener(eventType, handler, false);
  } else { // IE
```

```
    target.attachEvent("on" + eventType, handler);
  }
}
function removeHandler(target, eventType, handler) {
  if (target.removeEventListener) { // eventos DOM2
    target.removeEventListener(eventType, handler, false);
  } else { // IE
    target.detachEvent("on" + eventType, handler);
  }
}
```

O código verifica o suporte a Eventos DOM Nível 2 testando a presença dos métodos `addEventListener()` e `removeEventListener()`, aceitos por todos os navegadores modernos, com exceção do Internet Explorer. Caso esses métodos não existam no alvo, presume-se o uso do IE e passam a ser utilizados métodos específicos desse navegador.

À primeira vista, essas funções parecem bem otimizadas para seus propósitos. Entretanto, elas escondem um problema de desempenho no trabalho repetido que será realizado cada vez que uma das funções for chamada. Todas as vezes, a mesma verificação será feita, em busca da presença de determinado método. Caso você presuma que os únicos valores disponíveis para `target` são, na verdade, objetos DOM e que o usuário não pode alterar magicamente seu navegador enquanto carrega a página, fica fácil perceber que essa avaliação será repetitiva. Se `addEventListener()` estava presente na primeira chamada a `addHandler()`, ele também estará presente em cada chamada subsequente. A repetição do mesmo trabalho em cada chamada à função será apenas um desperdício. Felizmente existem formas de evitá-lo.

Carregamento tardio

A primeira forma de eliminarmos a repetição de trabalhos é utilizando o carregamento tardio. Carregamento tardio significa que nenhum trabalho será feito até que sua informação seja necessária. No caso do exemplo prévio, não há necessidade de determinarmos onde devemos adicionar ou remover manipuladores até que alguém

faça uma chamada à função. Uma versão da função prévia, agora com carregamento tardio, tem a seguinte aparência:

```
function addHandler(target, eventType, handler) {
  // sobrescreve a função existente
  if (target.addEventListener) { // eventos DOM2
    addHandler = function(target, eventType, handler) {
      target.addEventListener(eventType, handler, false);
    };
  } else { //IE
    addHandler = function(target, eventType, handler) {
      target.attachEvent("on" + eventType, handler);
    };
  }
  // chama a nova função
  addHandler(target, eventType, handler);
}
function removeHandler(target, eventType, handler) {
  // sobrescreve a função existente
  if (target.removeEventListener) { //DOM2 Events
    removeHandler = function(target, eventType, handler) {
      target.addEventListener(eventType, handler, false);
    };
  } else { //IE
    removeHandler = function(target, eventType, handler) {
      target.detachEvent("on" + eventType, handler);
    };
  }
  // chama a nova função
  removeHandler(target, eventType, handler);
}
```

Essas duas funções implementam um padrão de carregamento tardio. Na primeira vez que cada método for chamado, uma verificação será feita para determinar o modo apropriado de anexar ou desanexar o manipulador de evento. Então, a função original será sobrescrita com uma nova função que apresentará o curso apropriado de ação. O último passo durante a primeira chamada será a execução da nova função com os argumentos originais. Cada chamada subsequente a `addHandler()` ou a `removeHandler()` não implica em uma nova operação de detecção, uma vez que o código foi sobrescrito pela nova função.

Lembre-se de que a chamada a uma função de carregamento tardio sempre demora mais da primeira vez, devido à execução da detecção e à chamada de outra função para realização da tarefa. Chamadas subsequentes, por outro lado, ocorrem muito mais rapidamente, uma vez que não incluem lógica de detecção. Recomenda-se o carregamento tardio quando não se pretende utilizar a função imediatamente na página.

Carregamento prévio condicional

Uma alternativa às funções de carregamento tardio é o carregamento prévio condicional. Nesse caso, realiza-se a detecção antecipadamente, enquanto o script carrega, em vez de se aguardar por uma chamada à função. O processo de detecção continua sendo feito apenas uma vez, mas agora ele ocorre antecipadamente. Por exemplo:

```
var addHandler = document.body.addEventListener ?
  function(target, eventType, handler) {
    target.addEventListener(eventType, handler, false);
  }:
  function(target, eventType, handler) {
    target.attachEvent("on" + eventType, handler);
  };
var removeHandler = document.body.removeEventListener ?
  function(target, eventType, handler) {
    target.removeEventListener(eventType, handler, false);
  }:
  function(target, eventType, handler) {
    target.detachEvent("on" + eventType, handler);
  };
```

Esse exemplo verifica se `addEventListener()` e `removeEventListener()` estão presentes, utilizando essa informação para designar a função mais apropriada. O operador ternário retorna a função de DOM Nível 2 caso esses métodos estejam presentes. Do contrário, ele retorna a função específica do IE. O resultado é que todas as chamadas a `addHandler()` e `removeHandler()` têm a mesma rapidez, já que a detecção foi feita com antecedência.

O carregamento prévio condicional faz com que todas as chamadas à função demorem o mesmo intervalo de tempo. O lado negativo é que a detecção ocorre durante o carregamento do script, em vez de acontecer mais adiante. Recomenda-se o carregamento prévio condicional nos casos em que se pretende utilizar a função imediatamente e por reiteradas vezes durante a duração da vida da página.

Utilize as partes rápidas

Ainda que o JavaScript seja acusado muitas vezes de lentidão, há partes da linguagem que são incrivelmente rápidas. Isso não deve ser surpresa alguma, já que as engines JavaScript são construídas em linguagens de código de baixo nível, sendo também compiladas dessa forma. Mesmo que seja fácil atribuir à engine a culpa pela lentidão de seu JavaScript, lembre-se de que ela é a parte mais rápida do processo; o problema provavelmente está em seu código. Mostraremos partes da engine que são muito mais rápidas que outras justamente por permitirem acesso a caminhos que evitam as partes lentas.

Operadores bit a bit

Operadores bit a bit são um dos aspectos mais mal-compreendidos da linguagem JavaScript. O consenso geral é de que os desenvolvedores simplesmente não compreendem como devem utilizá-los, confundindo-os com seus equivalentes booleanos. Como resultado, operadores bit a bit são usados apenas esporadicamente no desenvolvimento do código, apesar das vantagens que apresentam.

Números em JavaScript são todos armazenados no formato 64 bits IEEE-754. Entretanto, para operações bit a bit, todo número tem de ser convertido a uma representação 32 bits assinalada. Cada operador atua diretamente sobre sua representação para alcançar um resultado. Apesar da conversão, esse processo é incrivelmente rápido quando comparado a outras operações matemáticas e

booleanas em JavaScript.

Caso você não esteja acostumado com a representação binária dos números, o JavaScript pode ajudá-lo convertendo um número em uma string composta por seu equivalente binário, utilizando o método toString() e passando nele o número 2. Veja um exemplo:

```
var num1 = 25,
  num2 = 3;
alert(num1.toString(2)); // "11001"
alert(num2.toString(2)); // "11"
```

Observe que essa representação omite os zeros iniciais contidos em um número.

Há quatro operadores lógicos bit a bit em JavaScript:

E bit a bit

Retorna um número, apresentando 1 em cada bit onde ambos os operandos têm um 1.

OU bit a bit

Retorna um número, apresentando 1 em cada bit onde um dos operandos tem um 1.

OU EXCLUSIVO bit a bit

Retorna um número, apresentando 1 em cada bit onde exatamente um operando tem um 1.

NÃO bit a bit

Retorna 1 em cada posição onde o operando tem 0 e vice-versa.

Esses operadores são utilizados da seguinte forma:

```
// E bit-a-bit
var result1 = 25 & 3; // 1
alert(result.toString(2)); // "1"
// OU bit-a-bit
var result2 = 25 | 3; // 27
alert(resul2.toString(2)); // "11011"
// OU EXCLUSIVO bit-a-bit
var result3 = 25 ^ 3; // 26
alert(resul3.toString(2)); // "11000"
// NÃO bit-a-bit
```

```
var result = ~25; // -26
alert(resul2.toString(2)); // "-11010"
```

Há algumas formas de utilizar operadores bit a bit para acelerar o código. A primeira é sua utilização no lugar das operações matemáticas puras. Por exemplo, é comum alternarmos cores das linhas de uma tabela calculando o módulo de 2 para um dado número, da seguinte maneira:

```
for (var i=0, len=rows.length; i < len; i++) {
  if (i % 2) {
    className = "even";
  } else {
    className = "odd";
  }
  // aplica classe
}
```

O cálculo do módulo de 2 exige que o número seja dividido por 2 para determinar o resto. Se pudesse olhar a representação 32 bits dos números, você veria que um número é par quando seu primeiro bit é zero e ímpar, quando seu primeiro bit é 1. Isso pode facilmente ser determinado utilizando uma operação bit a bit “E” sobre um dado número e o número 1. Quando o número for par, o resultado de “E” 1 será 0; quando ímpar, o resultado de “E” 1 será 1. Podemos escrever o código anterior da seguinte maneira:

```
for (var i=0, len=rows.length; i < len; i++) {
  if (i & 1) {
    className = "odd";
  } else {
    className = "even";
  }
// aplica classe
}
```

Ainda que o código continue a ser pequeno, a versão bit a bit do “E” é até 50% mais rápida do que a original (dependendo do navegador).

A segunda forma de utilizarmos operadores bit a bit é conhecida como *bitmask* (máscara de bit). O *bitmasking* é uma técnica popular na ciência da computação quando existem várias opções booleanas

que podem estar presentes ao mesmo tempo. A ideia aqui é utilizarmos cada bit de um número individual para indicar se uma opção está ou não presente, transformando o número, para todos os efeitos, em um array de sinalizadores booleanos. Cada opção recebe um valor equivalente a uma potência de 2 para que a máscara funcione. Por exemplo:

```
var OPTION_A = 1;
var OPTION_B = 2;
var OPTION_C = 4;
var OPTION_D = 8;
var OPTION_E = 16;
```

Depois de definir as opções, você pode criar um único número que contenha várias configurações utilizando o operador “OU” bit a bit:

```
var options = OPTION_A | OPTION_C | OPTION_D;
```

Podemos conferir se uma dada opção está disponível utilizando o operador bit a bit “E”. A operação retornará 0 caso a opção não esteja presente e 1 se estiver:

```
// a opção A está na lista?
if (options & OPTION_A) {
  // faça algo
// a opção B está na lista?
if (options & OPTION_B) {
  // faça algo
}
```

Operações com bitmasking são consideravelmente rápidas, já que, como mencionado anteriormente, o trabalho ocorre em um nível mais baixo do sistema. Se existirem várias opções que estão sendo salvas juntas e verificadas com frequência, as bitmasks poderão ajudá-lo a acelerar o desempenho geral de seu código.

 A linguagem JavaScript também aceita os operadores bit-a-bit de deslocamento para esquerda (<<), de deslocamento para direita (>>) e de deslocamento para direita com preenchimento de zeros (>>>).

Métodos nativos

Não importa quão otimizado seja seu código JavaScript, ele nunca será mais rápido do que os métodos nativos oferecidos pela engine JavaScript. O motivo é simples: as partes nativas do JavaScript – aquelas já presentes no navegador antes mesmo que você escreva sequer uma linha de código – são todas escritas em uma linguagem de nível mais baixo, como C++. Isso significa que esses métodos são compilados em código de máquina como parte do navegador, não tendo as mesmas limitações de seu código JavaScript.

Um erro comum cometido por desenvolvedores inexperientes é a realização, em código, de complexas operações matemáticas, quando existem versões de melhor desempenho disponíveis no objeto Math embutido. O objeto Math contém propriedades e métodos projetados para realizar operações matemáticas com facilidade. Veja as várias constantes matemáticas disponíveis:

Constante	Significado
Math.E	O valor de E, a base do logaritmo natural
Math.LN10	O logaritmo natural de 10
Math.LN2	O logaritmo natural de 2
Math.LOG2E	O logaritmo na base 2 de E
Math.LOG10E	O logaritmo na base 10 de E
Math.PI	O valor de π
Math.SQRT1_2	A raiz quadrada de ½
Math.SQRT2	A raiz quadrada de 2

Cada um desses valores é pré-calculado, não sendo preciso calculá-los manualmente. Também existem métodos para lidar com cálculos matemáticos:

Método	Significado
Math.abs(num)	O valor absoluto de num
Math.exp(num)	$Math.E^{num}$
Math.log(num)	O logaritmo de num
Math.pow(num,potência)	$Num^{potência}$
Math.sqrt(num)	A raiz quadrada de num
Math.acos(x)	O arco cosseno de x
Math.asin(x)	O arco seno de x
Math.atan(x)	O arco tangente de x
Math.atan2(y,x)	O arco tangente de y/x

Math.cos(x)	O cosseno de x
Math.sin(x)	O seno de x
Math.tan(x)	A tangente de x

A utilização desses métodos é mais rápida do que a recriação da mesma funcionalidade em código JavaScript. Sempre que você tiver de realizar complexos cálculos matemáticos, lembre-se de conferir primeiro o objeto Math.

Outro exemplo é a API de seletores, que permite a realização de consultas a um documento DOM utilizando seletores CSS. Consultas CSS são implementadas de modo nativo em JavaScript e ganharam muita popularidade com a biblioteca jQuery. A engine jQuery é amplamente considerada a engine mais rápida para consultas CSS, mas, ainda assim, veremos que ela é muito mais lenta do que os métodos nativos. Os métodos nativos querySelector() e querySelectorAll() completam suas tarefas gastando, em média, 10% do tempo necessário para realização de consultas CSS com base em JavaScript[1]. A maioria das bibliotecas JavaScript já adota o uso da funcionalidade nativa, quando disponível, para acelerar seu desempenho geral.

Sempre utilize métodos nativos quando estes estiverem disponíveis, especialmente para cálculos matemáticos e operações com o DOM. Quanto mais trabalho for feito em código compilado, mais rápido se tornará seu código.


O Chrome chega a implementar muito de sua funcionalidade JavaScript nativa em JavaScript. Por utilizar um compilador JavaScript just-in-time (no ato), tanto para funcionalidades nativas quanto para seu código, há pouca diferença de desempenho entre as duas em algumas circunstâncias.

Resumo

O JavaScript apresenta alguns desafios particulares de desempenho relacionados à forma como você organiza seu código. Conforme as aplicações web se tornam mais avançadas, contendo cada vez mais linhas de JavaScript em seu funcionamento, padrões favoráveis e desfavoráveis começam a surgir. Vale a pena ter em

mente algumas práticas de programação:

- Evite a penalidade por avaliação dupla, não utilizando `eval()` e o construtor `Function()`. Da mesma forma, em `setTimeout()` e `setInterval()`, passe funções, e não strings.
- Utilize literais de objetos e arrays quando criar novos objetos e arrays. Eles são criados e inicializados mais rapidamente do que suas formas não-literais.
- Evite a realização repetitiva do mesmo trabalho. Aplique o carregamento tardio ou o carregamento prévio condicional quando for necessária lógica de detecção de navegador.
- Ao realizar operações matemáticas, considere a utilização de operadores bit a bit, que trabalham diretamente sobre a representação subjacente do número.
- A execução de métodos nativos é sempre mais rápida do que qualquer coisa que você possa escrever em JavaScript. Utilize métodos nativos sempre que possível.

Assim como em muitas das outras técnicas e abordagens tratadas neste livro, você perceberá os maiores ganhos de desempenho quando essas otimizações forem aplicadas a códigos executados com frequência.

---

1 Segundo testes realizados pelo pacote de testes SlickSpeed, disponível em: *http://www2.webkit.org/perf/slickspeed/.*

CAPÍTULO 9

Criação e disponibilização de aplicações JavaScript de alto desempenho

Julien Lecomte

Segundo um estudo de 2007 feito pela equipe Exceptional Performance do Yahoo!, em torno de 40% a 60% dos usuários do Yahoo! navegam com cache vazio, e cerca de 20% de todas as visualizações de páginas são feitas também da mesma forma, tendo de ser inteiramente (ou em grande parte) buscadas no servidor (*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*). Além disso, outro estudo mais recente, feito pela equipe Yahoo! Search (e confirmado independentemente por Steve Souders, do Google), indica que 15% de todo o conteúdo entregue pelos grandes sites nos Estados Unidos é transmitido sem compressão.

Esses fatos enfatizam a necessidade de garantir a entrega mais eficiente possível das aplicações web com base em JavaScript. Enquanto parte do trabalho é feita durante os ciclos de design e desenvolvimento, a fase de criação e disponibilização também é essencial, ainda que muitas vezes seja negligenciada. Caso são se tenha cuidado durante essa fase crucial, o desempenho de sua aplicação sofrerá independentemente do esforço investido em torná-la mais rápido.

O propósito deste capítulo é fornecer a você o conhecimento necessário para formação e disponibilização eficientes de uma aplicação com base em JavaScript. Muitos conceitos são ilustrados utilizando o Apache Ant, uma ferramenta de criação em Java que

rapidamente se tornou o padrão em todo o mercado para criação de aplicações web. Próximo ao fim do capítulo, será apresentada uma ferramenta de build personalizada escrita em PHP5 como exemplo.

Apache Ant

O Apache Ant (*http://ant.apache.org/*) é uma ferramenta para automatização de processos de criação (build) de software. É semelhante ao make, mas é implementado em Java e utiliza XML para descrever o processo de build, enquanto o make utiliza seu próprio formato Makefile. O Ant é um projeto da Apache Software Foundation (*http://www.apache.org/licenses/*).

O principal benefício apresentado pelo Ant, quando comparado ao make e a outras ferramentas, é sua portabilidade. O Ant em si encontra-se disponível em muitas plataformas diferentes, e o formato de seus arquivos de build é independente de plataformas.

Um arquivo de build do Ant é escrito em XML e chamado build.xml por padrão. Cada arquivo contém exatamente um projeto e ao menos um target (alvo). Um alvo do Ant pode depender de outros alvos.

Alvos contêm elementos de tarefas: ações que são executadas atomicamente. O Ant vem com muitas tarefas embutidas, sendo que tarefas adicionais podem ser adicionadas de acordo com a necessidade do usuário. Da mesma forma, tarefas personalizadas podem ser desenvolvidas em Java para serem utilizadas em um arquivo de build do Ant.

Um projeto pode ter uma série de propriedades, ou variáveis. Uma propriedade tem um nome e um valor. Ela pode ser definida de duas formas: dentro do arquivo de build utilizando a tarefa property, ou fora do Ant. Podemos avaliar uma propriedade posicionando seu nome entre ${ e }.

A seguir temos um exemplo de um arquivo de build. A execução do alvo (target) padrão (dist) compila o código Java contido no diretório de origem, empacotando-o em um arquivo JAR.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project name="MyProject" default="dist" basedir=".">
<!-- define as propriedades globais para esse build -->
<property name="src" location="src"/>
<property name="build" location="build"/>
<property name="dist" location="dist"/>
<target name="init">
  <!-- Cria o time stamp -->
  <tstamp/>
  <!-- Cria a estrutura do diretório build utilizada pelo compile -->
  <mkdir dir="${build}"/>
</target>
<target name="compile" depends="init" description="compile the source">
  <!-- Compila o código java de ${src} em ${build} -->
  <javac srcdir="${src}" destdir="${build}"/>
</target>
<target name="dist" depends="compile" description="generate the distribution">
  <!-- Cria a propriedade de distribuição -->
  <mkdir dir="${dist}/lib"/>
  <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
</target>
<target name="clean" description="clean up">
<!-- Deleta as árvores dos diretórios ${build} e ${dist} -->
  <delete dir="${build}"/>
  <delete dir="${dist}"/>
</target>
</project>
```

Ainda que o Apache Ant seja utilizado para ilustrar os conceitos centrais deste capítulo, muitas outras ferramentas estão disponíveis para criação de aplicações web. Uma delas, o Rake (*http://rake.rubyforge.org/*), vem ganhando popularidade nos últimos anos. Trata-se de um programa de build com base em Ruby que apresenta capacidades semelhantes ao make. Os Rakefiles (as versões do Rake dos Makefiles) são escritos com sintaxe Ruby padrão sendo, portanto, independentes de plataforma.

Combinação de arquivos JavaScript

Segunda a equipe Exceptional Performance do Yahoo!, a primeira e possivelmente mais importante diretriz para aceleração de seu website, especialmente no que diz respeito à primeira visita de seus

usuários, é a redução do número de solicitações HTTP necessárias para renderização da página (*http://yuiblog.com/blog/2006/11/28/performance-research-part-1/*). É nesse ponto que você deve começar a buscar otimizações, já que em geral a combinação dos recursos demanda uma quantidade relativamente pequena de trabalho apresentando o maior benefício potencial para seus usuários.

A maioria dos websites modernos utiliza vários arquivos JavaScript: normalmente uma pequena biblioteca, que contém uma série de utilitários e controles para simplificar o desenvolvimento de aplicações web interativas em vários navegadores e alguns códigos específicos do site, divididos em diversas unidades lógicas para não enlouquecer o desenvolvedor. A CNN (*http://www.cnn.com/*), por exemplo, utiliza as bibliotecas Prototype e Script.aculo.us. Sua front page exibe um total de 12 scripts externos e mais de 20 blocos de script embutidos. Uma simples otimização seria o agrupamento parcial, se não total, desse código em um único arquivo JavaScript externo, diminuindo drasticamente o número de solicitações HTTP necessárias para renderização da página.

O Apache Ant apresenta a habilidade de combinar vários arquivos por meio da tarefa concat. Entretanto, é importante lembrarmos que arquivos JavaScript normalmente têm de ser concatenados em uma ordem específica para que as dependências sejam respeitadas. Uma vez que elas tenham sido estabelecidas, podemos preservar a ordem dos elementos utilizando um filelist ou uma combinação de elementos fileset. Confira a aparência do alvo do Ant:

```
<target name="js.concatenate">
  <concat destfile="${build.dir}/concatenated.js">
    <filelist dir="${src.dir}"
      files="a.js, b.js"/>
    <fileset dir="${src.dir}"
      includes="*.js"
      excludes="a.js, b.js"/>
  </concat>
</target>
```

Esse alvo cria o arquivo *concatenated.js* no diretório build, como resultado da concatenação de *a.js*, seguida por *b.js*, seguida por todos os outros arquivos sob o diretório de origem em ordem alfabética.

Observe que se qualquer um dos arquivos de origem (exceto talvez o último) não terminar com um ponto-e-vírgula ou com um fim de linha, o arquivo concatenado resultante poderá não conter código JavaScript válido. Isso pode ser corrigido instruindo o Ant a verificar se cada arquivo de origem concatenado termina com um caractere de fim de linha, utilizando o atributo fixlastline:

```
<concat destfile="${build.dir}/concatenated.js" fixlastline="yes">
...
</concat>
```

Pré-processamento de arquivos JavaScript

> Na ciência da computação, um pré-processador é um programa que processa seu input para produzir um resultado que seja utilizado como entrada em outro programa. Diz-se que o resultado é uma forma pré-processada dos dados de input, sendo também utilizado frequentemente por programas subsequentes como compiladores. A quantidade e o tipo do processamento feito dependem da natureza do pré-processador; alguns pré-processadores são capazes apenas de realizar substituições textuais relativamente simples e macroexpansões, enquanto outros têm o poder de linguagens de programação completas.
>
> – http://en.wikipedia.org/wiki/Preprocessor

Ao pré-processar seus arquivos-fonte JavaScript , você não apenas torna sua aplicação mais rápida como também permite, entre outras coisas, a instrumentação condicional do código para que possa medir o desempenho de sua aplicação.

Uma vez que nenhum pré-processador é especificamente projetado para trabalhar com JavaScript, é interessante que você utilize um processador flexível o bastante para que as regras de sua análise

léxica possam ser personalizadas, ou que utilize um que seja projetado para trabalhar com uma linguagem cuja gramática léxica seja próxima à apresentada pelo JavaScript. Uma vez que a sintaxe da linguagem de programação C é próxima à apresentada pelo JavaScript, o pré-processador C (C preprocessor, ou cpp) é uma boa escolha. Confira a aparência do alvo do Ant:

```
<target name="js.preprocess" depends="js.concatenate">
  <apply executable="cpp" dest="${build.dir}">
    <fileset dir="${build.dir}"
      includes="concatenated.js"/>
    <arg line="-P -C -DDEBUG"/>
    <srcfile/>
    <targetfile/>
    <mapper type="glob"
      from="concatenated.js"
      to="preprocessed.js"/>
  </apply>
</target>
```

Esse alvo, que depende, por sua vez, do alvo `js.concatenate`, cria o arquivo *preprocessed.js* no diretório build como resultado da execução do `cpp` no arquivo concatenado anteriormente. Observe que o `cpp` é executado utilizando as opções padrões `–P` (que inibe a geração de marcadores de linha) e `–C` (que não descarta os comentários). Nesse exemplo, o macro `DEBUG` também é definido.

Com esse alvo, você pode agora utilizar a definição macro (`#define`, `#undef`) e as diretrizes de compilação condicionais (`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`) diretamente dentro dos arquivos JavaScript, permitindo, por exemplo, embutir condicionalmente (ou remover) o código de definição de perfil:

```
#ifdef DEBUG
(new YAHOO.util.YUILoader({
  require: ['profiler'],
  onSuccess: function(o) {
  YAHOO.tool.Profiler.registerFunction('foo', window);
  }
})).insert();
#endif
```

Caso você pretenda utilizar macros multilinhas, não se esqueça de utilizar o caractere Unix de fim de linha (LF). Você pode utilizar a tarefa Ant fixcrlf para fazer automaticamente essa correção.

Outro exemplo, não estritamente relacionado ao desempenho, mas que demonstra quão poderoso pode ser o processamento JavaScript, é a utilização dos “macros variadic” (macros que aceitam um número variável de argumentos) e a inclusão de arquivos para implementar afirmações JavaScript. Considere o seguinte arquivo, chamado *include.js*:

```
#ifndef _INCLUDE_JS_
#define _INCLUDE_JS_
#ifdef DEBUG
function assert(condition, message) {
  // Manipule a afirmação exibindo uma mensagem de alerta
  // possivelmente contendo um rastro de pilha como exemplo.
}
#define ASSERT(x, ...) assert(x, ## __VA_ARGS__)
#else
#define ASSERT(x, ...)
#endif /* DEBUG */
#endif /* _INCLUDE_JS_ */
```

Agora você pode escrever código JavaScript com a seguinte aparência:

```
#include "include.js"
function myFunction(arg) {
ASSERT(YAHOO.lang.isString(argvar), "arg should be a string");
...
#ifdef DEBUG
YAHOO.log("Log this in debug mode only");
#endif
...
}
```

A afirmação e o código extra de logging aparecem apenas quando o macro DEBUG é definido durante o desenvolvimento. Essas instruções desaparecem no build de produção final.

Minificação do JavaScript

A “minificação” do JavaScript é o processo pelo qual um arquivo

JavaScript é destituído de tudo que não contribui para sua execução. Isso inclui comentários e espaços em branco desnecessários. O processo normalmente reduz o tamanho do arquivo pela metade, resultando em downloads mais rápidos e encorajando programadores a escrever documentação embutida de melhor qualidade e mais extensiva.

O JSMin (*http://www.crockford.com/javascript/jsmin.html*), desenvolvido por Douglas Crockford, permaneceu por muito tempo como padrão para a minificação de JavaScript. Entretanto, à medida que as aplicações web continuaram crescendo em tamanho e complexidade, muitos sentiram que era hora de levar a minificação JavaScript um passo adiante. Essa é a principal justificativa por trás do desenvolvimento do YUI Compressor (*http://developer.yahoo.com/yui/compressor/*), uma ferramenta que realiza todo tipo de operações inteligentes para oferecer, de modo completamente seguro, um grau maior de compactação do que o oferecido por outras ferramentas. Além de eliminar comentários e espaços em branco desnecessários, o YUI Compressor oferece as seguintes funções:

- Substituição dos nomes de variáveis locais por nomes menores (com um, dois ou três caracteres), escolhidos para otimizar o download pelo usuário da compressão gzip.
- Substituição das anotações com colchetes por anotações com pontos sempre que possível (p. ex., `foo["bar"]` se torna `foo.bar`).
- Substituição de nomes de propriedades literais citadas sempre que possível (p. ex., `{"foo":"bar"}` se torna `{foo:"bar"}`).
- Substituição de aspas escapadas em strings (p. ex., `'aaa\'bbb'` se torna `"aaa'bbb"`).
- União de constantes (p. ex., `"foo"+"bar"` se torna `"foobar"`).

A execução de seu código JavaScript no YUI Compressor resulta em uma economia tremenda quando comparada ao JSMin, sem que seja necessária nenhuma ação adicional. Considere os seguintes números dos arquivos centrais da biblioteca YUI (versão 2.7.0,

disponível em *http://developer.yahoo .com/yui/*):

yahoo.js, dom.js e event.js crus 192.164 bytes

yahoo.js, dom.js e event.js + JSMin 47.316 bytes

yahoo.js, dom.js e event.js + YUI Compressor 35.896 bytes

Nesse exemplo, o YUI Compressor oferece uma economia de 24% sobre o JSMin. Entretanto, há coisas que podem ser feitas para aumentar ainda mais a economia de bytes. Armazenar referências locais a objetos/valores, envolver o código em uma closure, usar constantes para valores repetidos, evitar o `eval` (e seus parentes, o construtor `Function`, e as funções `setTimeout` e `setInterval` quando utilizadas com um literal tipo string como primeiro argumento), evitar a palavra-chave `with` e evitar comentários condicionais JScript, tudo contribui para tornar ainda menor o tamanho minificado. Considere a seguinte função, projetada para "ativar/desativar" (toggle) a classe `selected` no elemento especificado do DOM (220 bytes):

```
function toggle (element) {
  if (YAHOO.util.Dom.hasClass(element, "selected")){
    YAHOO.util.Dom.removeClass(element, "selected");
  } else {
    YAHOO.util.Dom.addClass(element, "selected");
  }
}
```

O YUI Compressor transformará esse código no seguinte (de 147 bytes):

```
function toggle(a){if(YAHOO.util.Dom.hasClass(a,"selected")){
YAHOO.util.Dom.removeClass(a,"selected")}else{YAHOO.util.Dom.
addClass(a,"selected")}};
```

Caso você refatore a versão original armazenando uma referência local a `YAHOO.util.Dom` e utilizando uma constante para o valor `"selected"`, o código terá a seguinte aparência (de 232 bytes):

```
function toggle (element) {
  var YUD = YAHOO.util.Dom, className = "selected";
  if (YUD.hasClass(element, className)){
    YUD.removeClass(element, className);
  } else {
    YUD.addClass(element, className);
  }
```

```
}
```

Essa versão mostra uma economia ainda maior depois de ter sido minificada pelo YUI Compressor (resultando em 115 bytes):

```
function toggle(a){var c=YAHOO.util.Dom,b="selected";if(c.hasClass(a,b)){
c.removeClass(a,b)}else{c.addClass(a,b)}};
```

A razão de compactação avançou de 33% para 48%, um resultado espantoso, dada a pequena quantidade de trabalho necessário. Entretanto, é importante notar que a compressão em gzip, que ocorre no downstream, pode produzir resultados conflitantes; em outras palavras, o menor arquivo minificado pode nem sempre resultar no menor arquivo depois de comprimido pelo gzip. Esse resultado estranho é uma consequência direta da diminuição da redundância no arquivo original. Além disso, esse tipo de micro-otimização apresenta um pequeno custo sobre o tempo de execução, já que agora variáveis serão utilizadas no lugar dos valores literais, exigindo verificações adicionais. Dessa forma, recomendo que você não abuse dessas técnicas, mesmo que valha à pena levá-las em consideração quando conteúdo estiver sendo entregue a usuários que não aceitam (ou não tornam pública sua aceitação) a compressão gzip.

Em novembro de 2009, o Google lançou uma ferramenta de minificação ainda mais avançada chamada Closure Compiler (*http://code.google.com/closure/compiler/*). Essa nova ferramenta vai ainda mais longe do que o YUI Compressor quando utilizamos sua opção de otimizações avançadas. Nesse modo, o Closure Compiler é extremamente agressivo na forma que transforma o código e renomeia os símbolos. Ainda que resulte em resultados incríveis, é preciso que o desenvolvedor seja muito cuidadoso para garantir que o código de resultado funcione da mesma forma que o código de input. Além disso, ele dificulta o processo de depuração, já que praticamente todos os símbolos são renomeados. A biblioteca Closure vem com uma extensão para o Firebug, chamada Closure Inspector (*http://code.google.com/closure/compiler/docs/inspector.html*), que

oferece um mapeamento dos símbolos recriados em relação a suas versões originais. Mesmo assim, essa extensão não está disponível em navegadores que não sejam o Firefox, o que pode ser um problema quando se deseja fazer o debug de caminhos de código específicos a determinados navegadores. A depuração continuará sendo mais difícil do que a realizada com outras ferramentas de minificação menos agressivas.

Processos no tempo de build versus processos no tempo de execução

A concatenação, o pré-processamento e a minificação são passos que podem ser tomados no tempo de build (*buildtime*) ou no de execução (*runtime*). Processos de build no tempo de execução (runtime) são muito úteis durante o desenvolvimento, mas geralmente não são recomendados em um ambiente de produção devido aos problemas de escalonamento que podem apresentar. Como regra geral para criação de aplicações de alto desempenho, tudo que puder ser feito durante o tempo de build não deve ser feito no de execução.

Enquanto o Apache Ant é um programa de criação definitivamente off-line, a ágil ferramenta de build que apresentaremos próximo ao fim do capítulo representa um meio termo, podendo ser utilizada tanto durante o desenvolvimento quanto na criação dos atributos finais que serão usados em um ambiente de produção.

Compressão JavaScript

Quando um navegador solicita um recurso, ele normalmente envia um cabeçalho HTTP Accept-Encoding (começando com HTTP/1.1) para permitir que o servidor web saiba que formas de transformação de codificação são aceitas. Essa informação é utilizada principalmente para permitir que um documento seja comprimido, permitindo downloads mais rápidos e, dessa forma, uma melhor experiência ao usuário. Valores possíveis para os sinais de valor Accept-Encoding

incluem: gzip, compress, deflate e identify (valores registrados pela Internet Assigned Numbers Authority, ou IANA).

Caso um servidor web veja esse cabeçalho na solicitação, ele escolherá o método de codificação mais apropriado e notificará ao navegador sua decisão por meio do cabeçalho HTTP Content-Encoding.

O gzip é de longe o método de codificação mais popular. Geralmente reduz o tamanho do objeto em 70%, tornando-se a melhor escolha para melhorias de desempenho em uma aplicação web. Observe que a compressão com gzip deve ser utilizada principalmente em respostas de texto, incluindo arquivos JavaScript. Outros tipos de arquivos, como imagens ou arquivos PDF, não devem ser comprimidos com gzip. Esses arquivos já estão comprimidos, fazendo com que novas tentativas de redução representem apenas um desperdício dos recursos do navegador.

Caso você esteja utilizando o servidor web Apache (certamente o mais popular), a habilitação da compressão gzip requer a instalação e configuração do módulo mod_gzip (para o Apache 1.3, disponível em *http://www.schroepl.net/projekte/mod_gzip/*) ou do módulo mod_deflate (para o Apache 2).

Estudos recentes realizados independentemente pelo Yahoo! Search e pelo Google mostram que cerca de 15% do conteúdo entregue pelos grandes websites nos Estados Unidos não está comprimido. Isso ocorre principalmente pela falta de um cabeçalho HTTP Accept-Encoding na solicitação, removido por certos proxies corporativos, firewalls ou até alguns softwares de segurança. Ainda que a compressão gzip seja uma ferramenta fantástica para os desenvolvedores web, você não deve se esquecer desses fatos e sempre criar seu código da forma mais concisa possível. Outra técnica disponível é a entrega de conteúdo JavaScript diferente a usuários que não se beneficiarão da compressão gzip, mas que podem se beneficiar de uma experiência mais leve (ainda que deva ser dada aos usuários a escolha de retornar à versão completa).

Nesse sentido, vale a pena mencionar o Packer

(*http://dean.edwards.name/packer/*), um minificador JavaScript desenvolvido por Dean Edwards. O Packer é capaz de encolher arquivos JavaScript em um grau maior do que o oferecido pelo YUI Compressor. Considere os seguintes resultados na biblioteca jQuery (versão 1.3.2, disponível em *http://www.jquery.com/*):

jQuery	120.180 bytes
jQuery + YUI Compressor	56.814 bytes
jQuery + Packer	**39.351 bytes**
jQuery cru + gzip	34.987 bytes
jQuery + YUI Compressor + gzip	**19.457 bytes**
jQuery + Packer + gzip	***19.228 bytes***

Depois do gzip, ao passarmos a biblioteca jQuery pelo Packer ou pelo YUI Compressor, produzimos resultados muito semelhantes. Entretanto, arquivos comprimidos utilizando o Packer implicam um custo fixo ao tempo de execução (cerca de 200 a 300 milissegundos em meu laptop moderno). Portanto, utilizando o YUI Compressor junto do gzip sempre conseguimos os melhores resultados. Ainda assim, o Packer pode ser utilizado satisfatoriamente no caso de usuários com conexões lentas que não aceitam a compressão gzip, para os quais o custo da descompressão será irrisório quando comparado ao download de grandes quantidades de código. O único ponto negativo da entrega de conteúdo JavaScript distinto para usuários diferentes é o custo acrescido de controle de qualidade.

Armazenamento em cache de arquivos JavaScript

Tornar componentes HTTP armazenáveis em cache servirá para melhorar sensivelmente a experiência de visitantes em retorno a seu website. Como exemplo concreto, o carregamento da home page do Yahoo! (*http:// www.yahoo.com/*) na presença de um cache cheio exige um número 90% menor de solicitações HTTP e 83% menor de bytes a serem baixados do que quando sua visualização é feita com cache vazio. O tempo de ida e volta (gasto entre o momento em que

a página é solicitada e o disparo do evento onload) varia de 2,4 segundos para 0,9 segundo (*http://yuiblog.com/blog/2007/01/04/performance-research-part-2/*). Ainda que o armazenamento em cache seja mais utilizado para imagens, ele pode e deve ser usado para todos os componentes estáticos, incluindo arquivos JavaScript.

Servidores web utilizam o cabeçalho HTTP de resposta Expires para fazer com que seus clientes saibam por quanto tempo um recurso deve ser armazenado em cache. O formato é um registro de tempo absoluto no formato RFC 1123. Um exemplo de sua utilização é: Expires: Thu, 01 Dec 1994 16:00:00 GMT. Para marcar uma resposta como "never expires" (que nunca expira), um servidor web envia uma data de Expires situada aproximadamente um ano no futuro da data na qual a resposta é enviada. Servidores web nunca devem posicionar datas de Expires a um intervalo de tempo maior do que um ano no futuro, de acordo com o HTTP 1.1 RFC (RFC 2616, seção 14.21).

Caso utilize o servidor web Apache, a diretriz ExpiresDefault permite a definição de uma data expiratória em relação à data atual. O exemplo a seguir aplica essa diretriz a imagens, arquivos JavaScript e folhas de estilo CSS:

```
<FilesMatch "\.(jpg|jpeg|png|gif|js|css|htm|html)$">
  ExpiresActive on
  ExpiresDefault "access plus 1 year"
</FilesMatch>
```

Alguns navegadores, especialmente quando executados em dispositivos móveis, podem ter capacidades de cache limitadas. Por exemplo, o navegador Safari do iPhone não armazena um componente em cache caso seu tamanho seja maior que 25 KB quando não comprimido (veja *http://yuiblog.com/blog/2008/02/06/iphone-cacheability/*) ou 15 KB para o OS do iPhone 3.0. Nesses casos, é relevante considerar uma concessão entre o número de componentes HTTP e a capacidade de seu armazenamento em cache, dividindo-os em pedaços menores.

Você também pode pensar na utilização de mecanismos de

armazenamento no lado servidor, se estes estiverem disponíveis. Nesses casos, o código JavaScript deve lidar, ele mesmo, com a expiração.

Por fim, outra técnica é a utilização do cache de aplicações off-line do HTML5, implementado no Firefox 3.5, no Safari 4.0 e no iPhone a partir do OS 2.1. Essa tecnologia depende da presença da listagem dos recursos a serem armazenados em um arquivo manifesto. Esse arquivo deve ser declarado pela adição de um atributo manifest à tag <html> (observe a utilização de DOCTYPE HTML 5):

```
<!DOCTYPE html>
<html manifest="demo.manifest">
```

O arquivo manifesto utiliza uma sintaxe específica para listar recursos off-line e deve ser servido utilizando o tipo MIME text/cache-manifest. Mais informações sobre o armazenamento off-line de aplicações em cache podem ser encontradas no website do W3C, em *http://www.w3.org/TR/html5/offline.html*.

Como lidar com problemas de caching

Um controle de cache adequado pode realmente melhorar a experiência do usuário, mas tem também um lado negativo: quando estiver incrementando sua aplicação, não se esqueça de fornecer a seus usuários a versão mais atualizada do conteúdo estático. Isso pode ser feito renomeando os recursos estáticos sempre que eles sofrerem alterações.

É comum que desenvolvedores adicionem um número de versão, ou de build, aos nomes de seus arquivos. Outros gostam de anexar um valor de checksum. Pessoalmente, prefiro o uso de um registro de tempo, ou timestamp. Essa tarefa pode ser automatizada no Ant. O alvo a seguir renomeia os arquivos JavaScript anexando um timestamp na forma de aaaammddhhmm:

```
target name="js.copy">
  <!-- Crie o timestamp -->
  <tstamp/>
```

```
    <!-- Renomeie os arquivos JavaScript anexando um timestamp -->
    <copy todir="${build.dir}">
      <fileset dir="${src.dir}" includes="*.js"/>
      <globmapper from="*.js" to="*-${DSTAMP}${TSTAMP}.js"/>
    </copy>
  </target>
```

Utilização de uma rede de entrega de conteúdo

Uma rede de entrega de conteúdo (*Content Delivery Network*, ou CDN) é uma rede de computadores distribuída geograficamente pela Internet, responsável pela entrega de conteúdo aos usuários finais. As principais justificativas para o uso de uma CDN são sua confiabilidade, capacidade de escalonamento e, acima de tudo, seu desempenho. Na verdade, ao servir conteúdo a partir de um local mais próximo ao usuário, as CDNs são capazes de diminuir drasticamente a latência de rede.

Algumas grandes empresas mantêm suas próprias CDNs, mas normalmente vale a pena utilizar um fornecedor terceirizado, como a Akamai Technologies (*http://www.akamai.com/*) ou a Limelight Networks (*http://www.limelightnetworks.com/*).

A adoção de uma CDN é realizada por uma alteração relativamente simples em seu código, mas tem o potencial de melhorar drasticamente os tempos de resposta de seus usuários.

Vale a pena lembrar que as bibliotecas JavaScript mais comuns são todas acessíveis por CDNs. Por exemplo, a biblioteca YUI é servida diretamente pela rede Yahoo! (cujo nome de servidor é *yui.yahooapis.com*: maiores detalhes em *http://developer.yahoo.com/yui/articles/hosting/*), enquanto as bibliotecas jQuery, Dojo, Prototype, Script.aculo.us, MooTools e YUI, bem como ainda outras mais, estão disponíveis diretamente via a CDN do Google (cujo nome de servidor é *ajax.googleapis.com*: maiores detalhes em *http://code.google.com/apis/ajaxlibs/*).

Disponibilização de recursos JavaScript

A disponibilização (deployment) de recursos JavaScript normalmente se resume à cópia de arquivos a um ou mais hosts remotos e, às vezes, à execução de uma série de comandos shell nesses hosts, especialmente quando se utiliza uma CDN na distribuição dos novos arquivos.

O Apache Ant oferece várias opções para copiar arquivos para servidores remotos. Você pode usar a tarefa copy para copiar arquivos para um sistema de arquivos localmente montado, ou pode utilizar as tarefas opcionais FTP e SCP. Minha preferência pessoal é pela utilização do utilitário scp, disponível em todas as principais plataformas. A seguir temos um exemplo bem simples dessa prática em demonstração:

```
<apply executable="scp" failonerror="true" parallel="true">
  <fileset dir="${build.dir}" includes="*.js"/>
  <srcfile/>
  <arg line="${live.server}:/var/www/html/"/>
</apply>
```

Por fim, para executar os comandos shell em um host remoto que roda o daemon SSH, utilize a tarefa opcional SSHEXEC ou simplesmente invoque o utilitário ssh diretamente, como demonstra o exemplo seguinte, utilizado para reiniciar o servidor web Apache em um host Unix:

```
<exec executable="ssh" failonerror="true">
  <arg line="${live.server}"/>
  <arg line="sudo service httpd restart"/>
</exec>
```

Processos JavaScript de build

Ferramentas tradicionais de build são ótimas, mas a maior parte dos desenvolvedores costuma achá-las incômodas por exigirem a compilação manual da solução após cada etapa individual que altere o código. Em vez disso, é preferível que você apenas tenha de recarregar a janela do navegador e pular por completo o passo de compilação. Como consequência, somente alguns desenvolvedores web utilizam as técnicas destacadas neste

capítulo, o que resulta em aplicações e sites de desempenho baixo. Por sorte, a criação de uma ferramenta que combina todas essas técnicas avançadas não é complicada, permitindo que os desenvolvedores trabalhem com eficiência, ao mesmo tempo em que garantem o melhor desempenho possível para suas aplicações.

O smasher é um aplicativo PHP5 que tem como base uma ferramenta interna utilizada pelo Yahoo! Search. Ele combina vários arquivos JavaScript, faz seu pré-processamento e opcionalmente minifica seu conteúdo. Pode ser executado a partir da linha de comando ou durante o desenvolvimento para lidar com solicitações web e automaticamente combinar recursos no ato. Seu código-fonte pode ser encontrado em *http://github.com/jlecomte/smasher*, contendo os seguintes arquivos:

smasher.php

Arquivo central.

smasher.xml

Arquivo de configuração.

smasher

Envoltório (wrapper) da linha de comando.

smasher_web.php

Ponto de entrada do servidor web.

O smasher exige um arquivo de configuração XML que contenha a definição dos grupos de arquivos que ele combinará, assim como algumas informações adicionais sobre o sistema. A seguir temos um exemplo da aparência desse arquivo:

```
<?xml version="1.0" encoding="utf-8"?>
<smasher>
  <temp_dir>/tmp/</temp_dir>
  <root_dir>/home/jlecomte/smasher/files/</root_dir>
  <java_bin>/usr/bin/java</java_bin>
  <yuicompressor>/home/jlecomte/smasher/yuicompressor-2-4-2.jar</yuicompressor>
  <group id="yui-core">
    <file type="css" src="reset.css" />
```

```
    file type="css" src="fonts.css" />
    <file type="js" src="yahoo.js" />
    <file type="js" src="dom.js" />
    <file type="js" src="event.js" />
  </group>
  <group id="another-group">
    <file type="js" src="foo.js" />
    <file type="js" src="bar.js" />
    <macro name="DEBUG" value="1" />
  </group>
  ...
</smasher>
```

Cada elemento de grupo contém uma série de arquivos JavaScript e/ou CSS. O elemento de topo root_dir contém o caminho ao diretório onde esses arquivos podem ser encontrados. Opcionalmente, elementos de grupos podem também conter uma lista de definições macro de pré-processamento.

Uma vez que esse arquivo de configuração tenha sido salvo, você poderá executar o smasher a partir da linha de comando. Caso venha a executá-lo sem nenhum dos parâmetros necessários, ele exibirá apenas suas informações de uso e fechará. O exemplo seguinte mostra como podermos combinar, pré-processar e minificar os arquivos Javascript do núcleo YUI:

```
$ ./smasher -c smasher.xml -g yui-core -t js
```

Se tudo der certo, o arquivo de resultado poderá ser encontrado no diretório de trabalho, sendo nomeado de acordo com a denominação do grupo (yui-core, neste exemplo) seguido por um timestamp e a extensão apropriada (p. ex., *yui-core-200907191539.js*).

De modo semelhante, você também pode utilizar o smasher para lidar com solicitações web durante o desenvolvimento posicionando o arquivo *smasher_web.php* em algum ponto dentro da raiz do documento de seu servidor web e usando uma URL como esta:

```
http://<host>/smasher_web.php?conf=smasher.xml&group=yui-core&type=css&nominify
```

Ao utilizar URLs diferentes para seus recursos JavaScript e CSS durante o desenvolvimento e na produção você pode trabalhar de

modo eficiente, além de obter o melhor desempenho possível em seu processo de build.

Resumo

O processo de build e disponibilização pode ter um impacto tremendo sobre o desempenho de uma aplicação com base em JavaScript. Os passos mais importantes desse processo são:

- A combinação dos arquivos JavaScript para reduzir o número de solicitações HTTP necessárias.
- A minificação dos arquivos JavaScript utilizando o YUI Compressor.
- A entrega dos arquivos JavaScript depois que tiverem sido comprimidos (por codificação gzip).
- O armazenamento em cache dos arquivos definindo os cabeçalhos HTTP apropriados e uma solução aos problemas de caching anexando um timestamp aos nomes dos arquivos.
- A utilização de uma rede de entrega de conteúdo, ou CDN, para entrega de seus arquivos JavaScript. Uma CDN não só melhorará seu desempenho como também cuidará da compressão e do armazenamento em cache.

Todos esses passos devem ser automatizados utilizando ferramentas de build disponíveis publicamente como o Apache Ant ou ferramentas personalizadas ideais para suas necessidades. Caso faça com que o processo de build trabalhe a seu favor, você poderá melhorar muito o desempenho de suas aplicações web e de websites que demandam grande quantidade de código JavaScript.

CAPÍTULO 10

Ferramentas

Matt Sweeney

Utilizar o software correto é essencial para que você possa identificar os gargalos prejudiciais ao desempenho tanto no carregamento quanto na execução de seus scripts. Vários fornecedores de navegadores e websites de larga escala têm técnicas compartilhadas e ferramentas que podem ajudá-lo a tornar a web mais rápida e eficiente. Este capítulo irá tratar de algumas das ferramentas gratuitas disponíveis para:

Criação do perfil (profiling)
: A cronometragem de várias funções e operações durante a execução do script, identificando áreas onde podem ser feitas otimizações.

Análise da rede
: Um exame do carregamento das imagens, folhas de estilo e scripts e seus efeitos sobre o carregamento e a renderização geral da página.

Quando um script ou aplicação em particular estiver sendo executado abaixo do desempenho pretendido, um dispositivo para criação de perfis (profiler) pode ajudar a priorizar as áreas onde cabem otimizações. Tal tarefa pode se tornar complicada devido à variedade de navegadores aceitos, mas muitos fornecedores já oferecem um profiler junto a suas ferramentas de depuração. Em alguns casos, problemas de desempenho podem ser específicos a um navegador em particular; em outras situações, os sintomas podem ocorrer em vários navegadores. Lembre-se de que as otimizações aplicadas tendo em mente um navegador podem ser

benéficas também a outros, mas que podem, da mesma forma, ter o efeito oposto. Ao utilizar profilers, em vez de simplesmente presumir quais funções ou operações estão lentas, você é capaz de garantir que o tempo de otimização será gasto nas áreas mais problemáticas do sistema e que afetam a maioria dos navegadores.

Ainda que a maior parte deste capítulo trate das ferramentas de profiling, os analisadores de rede também podem ser altamente eficientes, garantindo que os scripts e as páginas sejam carregados e executados com a maior velocidade possível. Antes de mergulhar nas alterações em seu código, certifique-se de que todos os scripts e outros recursos estão sendo carregados da melhor forma. O carregamento de imagens e folhas de estilo pode afetar também o carregamento dos scripts dependendo de quantas solicitações simultâneas são permitidas pelo navegador e de quantos recursos estão sendo carregados.

Algumas dessas ferramentas fornecem dicas que podem ajudá-lo a melhorar o desempenho das páginas web. Lembre-se de que a melhor forma de interpretar as informações que essas ferramentas oferecem é aprendendo o raciocínio por trás de suas regras. Assim como na maioria dos casos nos quais encontramos regras, existem também exceções. Uma melhor compreensão das regras permite que você saiba quando deve quebrá-las.

Criação de perfis com JavaScript

A ferramenta que acompanha todas as implementações JavaScript é a linguagem em si. Utilizando o objeto `Date`, podemos fazer uma medição do script a qualquer momento. Antes que outras ferramentas existissem, essa era uma forma comum de cronometrar a execução do script, continuando a ser útil em alguns casos. Por padrão, o objeto `Date` retorna o tempo atual, fazendo com que ao subtrairmos uma instância de `Date` de outra tenhamos acesso ao tempo transcorrido em milissegundos. Considere o seguinte exemplo, que compara a criação de elementos do zero com a

clonagem de um elemento existente (veja o capítulo 3, Criação de scripts DOM):

```
var start = new Date(),
  count = 10000,
  i, element, time;
for (i = 0; i < count; i++) {
  element = document.createElement('div');
}
time = new Date() - start;
alert('created ' + count + ' in ' + time + 'ms');
start = new Date();
for (i = 0, i < count; i++) {
  element = element.cloneNode(false);
}
time = new Date() - start;
alert('created ' + count + ' in ' + time + 'ms');
```

Como demanda a instrumentação manual de seu próprio código de cronometragem, esse tipo de criação de perfis é excessivamente trabalhoso. A inclusão de um objeto Timer para lidar com os cálculos de tempo e armazenar os dados seria um bom próximo passo.

```
var Timer = {
  _data: {},
start: function(key) {
  Timer._data[key] = new Date();
},
stop: function(key) {
  var time = Timer._data[key];
  if (time) {
    Timer._data[key] = new Date() - time;
  }
},
getTime: function(key) {
  return Timer._data[key];
  }
};
Timer.start('createElement');
for (i = 0; i < count; i++) {
  element = document.createElement('div');
}
Timer.stop('createElement');
```

```
alert('created ' + count + ' in ' + Timer.getTime('createElement');
```

Como podemos ver, a instrumentação manual ainda é necessária, mas já temos um padrão para construção de um criador de perfis em JavaScript puro. Ao estender o conceito do objeto Timer, um profiler pode ser construído para registrar as funções e instrumentá-las com código de cronometragem.

YUI Profiler

O YUI Profiler (*http://developer.yahoo.com/yui/profiler/*), uma contribuição de Nicholas Zakas, é um profiler JavaScript escrito em JavaScript. Além de atuar como um cronômetro, ele oferece interfaces para criação de perfis de funções, objetos e construtores, além de relatórios detalhados dos dados analisados. Outros de seus recursos incluem a criação de perfis para vários navegadores, a exportação dos dados para análises adicionais e a criação de relatórios mais encorpados.

O YUI Profiler oferece um temporizador genérico que coleta os dados de desempenho, além de apresentar métodos estáticos para dar início e interromper os timings nomeados e recuperar os dados de perfil.

```
var count = 10000, i, element;
  Y.Profiler.start('createElement');
  for (i = 0; i < count; i++) {
    element = document.createElement('div');
  }
  Y.Profiler.stop('createElement');
alert('created ' + count + ' in ' +
  Y.Profiler.getAverage('createElement') + 'ms');
```

Fica evidente que isso representa uma melhora quando comparado à abordagem com Date e Timer embutidos, além de oferecer dados adicionais em relação ao número de chamadas, assim como dados como o tempo médio, mínimo e máximo. Tais dados são coletados e podem ser analisados ao lado de outros resultados apresentados.

Também podem ser traçados perfis de funções. A função registrada

é instrumentada com código que coleta dados de desempenho. Por exemplo, para criação do perfil do método global initUI, mostrado no capítulo 2, basta o nome:

```
Y.Profiler.registerFunction("initUI");
```

Muitas funções estão ligadas a objetos para impedir a desorganização do namespace global. Métodos de objetos podem ser registrados passando-os como o segundo argumento de registerFunction. Por exemplo, presuma um objeto chamado uiTest que implementa duas abordagens de initUI como uiTest.test1 e uiTest.test2. Cada uma pode ser registrada individualmente:

```
Y.Profiler.registerFunction("test1", uiTest);
Y.Profiler.registerFunction("test2", uiTest);
```

Apesar de funcionar corretamente, essa abordagem não é adequada quando cresce o número de funções que desejamos analisar ou quando desejamos traçar o perfil de uma aplicação inteira. O método registerObject é capaz de registrar automaticamente todos os métodos ligados ao objeto:

```
Y.Profiler.registerObject("uiTest", uiTest);
```

O primeiro argumento é o nome do objeto (para criação do relatório), e o segundo é o objeto em si. Essa linha permitirá a criação de perfis para todos os métodos de uiTest.

Objetos que dependem da herança de protótipos necessitam de um tratamento especial. O YUI Profiler permite o registro de uma função construtora que serve para instrumentar todos os métodos em todas as instâncias do objeto:

```
Y.Profiler.registerConstructor("MyWidget", myNameSpace);
```

Dessa forma, toda função em cada instância de myNameSpace.MyWidget será analisada. Um relatório individual pode ser recuperado como um objeto:

```
var initUIReport = Y.Profiler.getReport("initUI");
```

Assim, temos um objeto que contém todos os dados de perfil, incluindo um array de pontos. Esses pontos são cronometragens de cada chamada, na ordem em que foram chamadas, e podem ser

organizados e analisados de outras formas interessantes para exame de variações no tempo. Esse objeto tem os seguintes campos:

```
{
  min: 100,
  max: 250,
  calls: 5,
  avg: 120,
  points: [100, 200, 250, 110, 100]
};
```

Às vezes, você pode estar interessado apenas no valor de um campo específico. Métodos estáticos Profiler oferecem dados separados por função ou método:

```
var uiTest1Report = {
  calls: Y.Profiler.getCalls("uiTest.test1"),
  avg: Y.Profiler.getAvg("uiTest.test1")
};
```

Para analisar corretamente o desempenho de um script, desejamos uma visualização que destaque as áreas mais lentas do código. Também está disponível um relatório de todas as funções registradas chamadas no objeto ou construtor:

```
var uiTestReport = Y.Profiler.getReport("uiTest");
```

Isso retorna um objeto com os seguintes dados:

```
{
  test1: {
    min: 100,
    max: 250,
    calls: 10,
    avg: 120
  },
  test2:
    min: 80,
    max: 210,
    calls: 10,
    avg: 90
  }
};
```

Dessa forma, temos a oportunidade de organizar e visualizar os

dados de modo mais significativo, permitindo que as áreas mais lentas do código sejam analisadas minuciosamente. Um relatório completo com todos os dados atuais de perfil também pode ser gerado. Entretanto, tal relatório pode conter informações inúteis, como funções que não foram chamadas nenhuma vez ou que já preenchem às expectativas que temos quanto ao desempenho. Para minimizar a presença desses dados desnecessários, uma função opcional pode ser passada como filtro:

```
var fullReport = Y.Profiler.getFullReport(function(data) {
  return (data.calls > 0 && data.avg > 5);
};
```

O valor booleano retornado indicará se a função deve ser incluída no relatório, permitindo que dados menos interessantes sejam omitidos.

Ao terminar o perfil, dados quanto às funções, objetos e construtores podem ser desarquivados individualmente, limpando os dados do perfil:

```
Y.Profiler.unregisterFunction("initUI");
Y.Profiler.unregisterObject("uiTests");
Y.Profiler.unregisterConstructor("MyWidget");
```

O método `clear()` mantém o registro do perfil atual, mas limpa todos os dados associados.

Essa função pode ser chamada individualmente por função ou por tempo:

```
Y.Profiler.clear("initUI");
```

Ou podemos limpar todos os dados de uma só vez, omitindo o argumento de nome:

```
Y.Profiler.clear();
```

Como está no formato JSON, os dados do relatório de perfil podem ser visualizados de muitas formas. A mais simples é por meio de uma página web, produzindo o resultado dos dados em HTML. Também podem ser enviados a um servidor, onde serão armazenados em um banco de dados para criação de relatórios mais minuciosos. Isso é especialmente útil quando comparamos

várias técnicas de otimização em todos os navegadores.

Vale a pena mencionar que funções anônimas são particularmente problemáticas para esse tipo de ferramenta de perfil, já que não há um nome a ser utilizado. O YUI Profiler oferece um mecanismo para instrumentação de funções anônimas, permitindo sua análise. O registro de uma função anônima retorna uma função substituta que pode ser chamada no lugar da anônima:

```
var instrumentedFunction =
  Y.Profiler.instrument("anonymous1", function(num1, num2) {
    return num1 + num2;
  });
instrumentedFunction(3, 5);
```

Dessa forma, adicionamos dados da função anônima à série de resultados do Profiler, permitindo que sejam acessados da mesma forma que outros dados recuperados:

```
var report = Y.Profiler.getReport("anonymous1");
```

Funções anônimas

Dependendo do mecanismo utilizado para criação do perfil, alguns dados podem ser omitidos pela presença de funções anônimas ou de tarefas de funções. Como se trata de um padrão comum em JavaScript, muitas das funções avaliadas podem ser anônimas, o que dificulta, ou até impossibilita, sua medição e análise. A melhor forma de habilitar a criação de perfis para funções anônimas é renomeando-as. A utilização de ponteiros a métodos de objetos, no lugar de closures, permitirá a mais ampla cobertura na criação de perfis.

Compare a utilização de uma função embutida:

```
myNode.onclick = function() {
  myApp.loadData();
};
```

com uma chamada de método:

```
myApp._onClick = function() {
  myApp.loadData();
};
```

```
myNode.onclick = myApp._onClick;
```

A utilização da chamada de método permite que qualquer uma das ferramentas de perfis mencionadas automaticamente instrumente o manipulador onclick. Como demanda uma refatoração significativa para criação dos perfis, isso nem sempre é prático.

Para profilers que automaticamente instrumentam funções anônimas, a adição de um nome interno torna os relatórios mais legíveis:

```
myNode.onclick = function myNodeClickHandler() {
  myApp.loadData();
};
```

Isso também funciona em funções que são declaradas como variáveis. Alguns criadores de perfis encontram dificuldades ao tentar atribuir nomes a esse tipo de funções:

```
var onClick = function myNodeClickHandler() {
  myApp.loadData();
};
```

Agora, a função anônima está nomeada, fornecendo à maioria dos profilers algo que possam exibir junto aos resultados. A implementação desses nomes não é difícil de ser feita, podendo até ser realizada automaticamente como parte da depuração de um processo de build.

Sempre utilize versões não comprimidas de seus scripts para realização de depurações e criação de perfis. Isso garantirá que suas funções sejam de fácil identificação.

Firebug

O Firefox é um navegador muito popular entre os desenvolvedores. Isso se deve parcialmente à presença do addon Firebug (disponível em *http://www.getfirebug.com/*). O Firebug foi desenvolvido inicialmente por Joe Hewitt e é atualmente mantido pela Mozilla Foundation. Essa ferramenta aumentou a produtividade dos desenvolvedores web em todo o mundo oferecendo visualizações do código que antes eram impossíveis.

O Firebug oferece um console para registro do resultado do código, uma árvore DOM que pode ser atravessada, informações de estilo e a habilidade de analisar por dentro objetos DOM e JavaScript, além de muitas outras funções. Também inclui um criador de perfis e um analisador de rede, que serão o foco desta seção, além de ser altamente expansível, permitindo que painéis personalizados sejam facilmente adicionados.

Profiler do painel Console

O Firebug Profiler está disponível como parte do painel Console (Figura 10.1). Ele faz medições e produz relatórios sobre a execução do JavaScript na página. O relatório detalha cada função chamada enquanto o profiler está em execução, oferecendo dados de desempenho altamente precisos e insights valiosos quanto ao que pode estar causando a lentidão dos scripts.

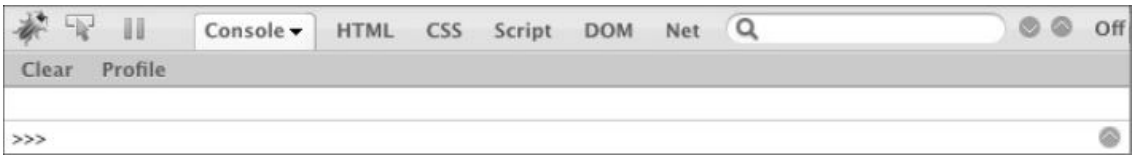
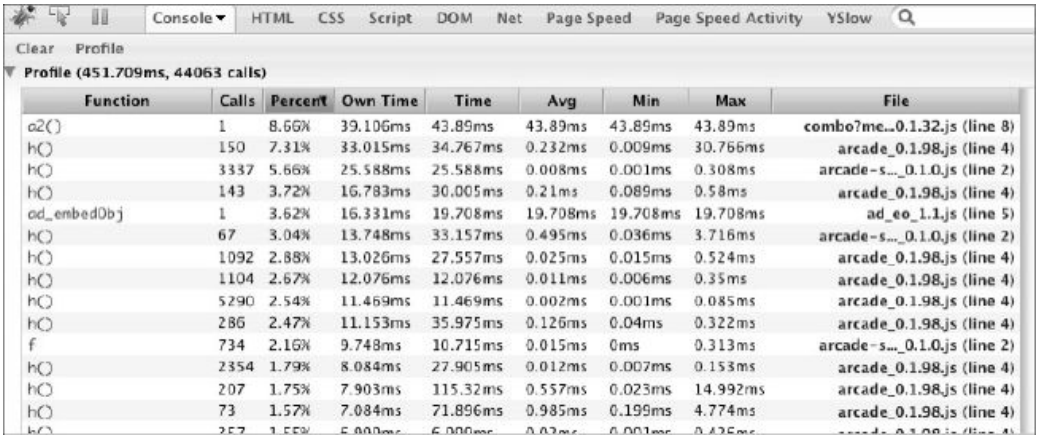


Figura 10.1 – Painel Console do FireBug.

Podemos criar um perfil clicando no botão Profile, disparando o script e clicando novamente no botão para interromper a análise. A figura 10.2 mostra um relatório típico de dados analisados. Estes incluem *Calls*, o número de vezes que a função foi chamada; *Own Time*, o tempo gasto na função em si; e *Time*, o tempo geral gasto em uma função e em qualquer função que ela possa ter chamado. A criação do perfil é instrumentada em nível baixo, fazendo com que haja pouca perda de desempenho quando se criam perfis a partir do painel Console.



Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
o2()	1	8.66%	39.106ms	43.89ms	43.89ms	43.89ms	43.89ms	combo?me...0.1.32.js (line 8)
h()	150	7.31%	33.015ms	34.767ms	0.232ms	0.009ms	30.766ms	arcade_0.1.98.js (line 4)
h()	3337	5.66%	25.588ms	25.588ms	0.008ms	0.001ms	0.308ms	arcade-s..._0.1.0.js (line 2)
h()	143	3.72%	16.783ms	30.005ms	0.21ms	0.089ms	0.58ms	arcade_0.1.98.js (line 4)
od_embedObj	1	3.62%	16.331ms	19.708ms	19.708ms	19.708ms	19.708ms	ad_eo_1.1.js (line 5)
h()	67	3.04%	13.748ms	33.157ms	0.495ms	0.036ms	3.716ms	arcade-s..._0.1.0.js (line 2)
h()	1092	2.88%	13.026ms	27.557ms	0.025ms	0.015ms	0.524ms	arcade_0.1.98.js (line 4)
h()	1104	2.67%	12.076ms	12.076ms	0.011ms	0.006ms	0.35ms	arcade_0.1.98.js (line 4)
h()	5290	2.54%	11.469ms	11.469ms	0.002ms	0.001ms	0.085ms	arcade_0.1.98.js (line 4)
h()	286	2.47%	11.153ms	35.975ms	0.126ms	0.04ms	0.322ms	arcade_0.1.98.js (line 4)
f	734	2.16%	9.748ms	10.715ms	0.015ms	0ms	0.313ms	arcade-s..._0.1.0.js (line 2)
h()	2354	1.79%	8.084ms	27.905ms	0.012ms	0.007ms	0.153ms	arcade_0.1.98.js (line 4)
h()	207	1.75%	7.903ms	115.32ms	0.557ms	0.023ms	14.992ms	arcade_0.1.98.js (line 4)
h()	73	1.57%	7.084ms	71.896ms	0.985ms	0.199ms	4.774ms	arcade_0.1.98.js (line 4)

Figura 10.2 – Painel Profile do Firebug.

API do Console

O Firebug também oferece uma interface JavaScript para iniciar e interromper o profiler, permitindo controle mais preciso sobre quais partes do código devem ser mensuradas. Por essa interface também podemos dar um nome ao relatório, o que é muito útil quando comparamos várias técnicas de otimização.

```
console.profile("regexTest");
regexTest('foobar', 'foo');
console.profileEnd();
console.profile("indexOfTest");
indexOfTest('foobar', 'foo');
console.profileEnd();
```

Ao iniciar e interromper o profiler nos momentos mais interessantes, você minimiza os efeitos colaterais e a confusão oriunda de outros scripts que possam estar sendo executados. Lembre-se de que quando acionar o profiler dessa maneira você estará aumentando o custo de desempenho do script. Isso ocorre principalmente graças ao tempo gasto na criação do relatório depois da chamada a profileEnd(), que atua bloqueando execuções subsequentes até que o relatório tenha sido gerado.


A interface JavaScript também está disponível por meio da linha de comando do Console Firebug.

Depois de terminar um perfil, um novo relatório será gerado, mostrando quanto tempo cada função utilizou, o número de vezes

que foi chamada, a porcentagem do overhead total e outros dados interessantes. Essas informações oferecem perspectivas valiosas sobre onde você deve gastar seu tempo na otimização de suas funções e na diminuição do número de chamadas.

Da mesma forma que o YUI Profiler, a função console.time() do Firebug pode ajudar a medir loops e outras operações que o profiler não monitora. Por exemplo, a execução seguinte cronometra uma pequena seção de código que contém um loop:

```
console.time("cache node");
for (var box = document.getElementById("box"),
    i = 0;
  i < 100; i++) {
  value = parseFloat(box.style.left) + 10;
  box.style.left = value + "px";
}
console.timeEnd("cache node");
```

Depois do término do timer, o tempo é enviado ao console. Isso pode ser útil quando buscamos comparar várias abordagens de otimização. Cronometragens adicionais podem ser capturadas e registradas no console, tornando fácil a análise lado a lado de seus resultados. Por exemplo, para comparar o armazenamento em cache da referência ao nodo com o armazenamento em cache da referência ao estilo do nodo, tudo que precisamos fazer é escrever a implementação e incluir o código de cronometragem:

```
console.time("cache style");
for (var style = document.getElementById("box").style,
    i = 0;
    i < 100; i++) {
  value = parseFloat(style.left) + 10;
  style.left = value + "px";
}
console.timeEnd("cache style");
```

A API do console oferece aos programadores a flexibilidade necessária para instrumentar a criação de perfis em várias camadas e consolidar seus resultados em relatórios que podem ser analisados de muitas formas interessantes.

 Ao clicar sobre uma função, você a exibe no contexto de arquivo-fonte. Isso ajuda muito quando estamos lidando com funções anônimas ou de nomes obscuros.

Painel Net

Muitas vezes, quando encontramos problemas de desempenho, é bom darmos um passo para trás e analisarmos o código como um todo. O Firebug oferece uma visualização dos recursos de rede no painel Net (Figura 10.3). Esse painel apresenta uma representação das pausas que ocorrem entre os scripts e outros recursos, fornecendo uma análise mais profunda do efeito que o script está tendo no carregamento de outros arquivos e sobre a página em geral.

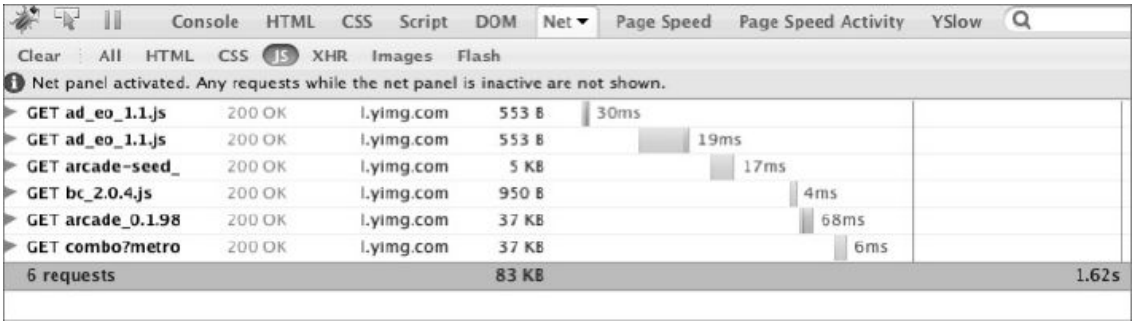


Figura 10.3 – Painel Net do Firebug.

As barras coloridas presentes ao lado de cada recurso dividem o ciclo de vida do carregamento da página em fases de componentes (consulta de DNS, espera por resposta etc.). A primeira linha vertical (exibida em azul) indica quando o evento DOMContentLoaded da página foi disparado. Esse evento sinaliza que o parsing da página já foi feito e que ela está pronta para ser utilizada. A segunda linha vertical (vermelha) indica quando o evento load da janela foi disparado, indicando que o DOM está pronto e que todos os recursos externos já foram carregados. Dessa forma, temos uma noção de quanto tempo foi gasto no parsing e na execução em comparação à renderização da página.

Como podemos ver na figura, existem vários scripts sendo baixados. Com base na linha temporal, cada script parece estar aguardando o script anterior antes de iniciar a solicitação seguinte.

A otimização mais simples para melhorar o desempenho do carregamento é a redução do número de solicitações, especialmente solicitações de scripts e folhas de estilo, que podem bloquear outros recursos e a renderização da página. Quando possível, combine todos os scripts em um único arquivo para minimizar o número total de solicitações. Isso também se aplica às folhas de estilo e imagens.

Ferramentas para desenvolvimento do Internet Explorer

A partir da versão 8, o Internet Explorer oferece um kit de ferramentas que inclui um profiler. Esse kit é embutido no IE 8, fazendo com que nenhum download ou instalação adicional seja necessário. Assim como o Firebug, o profiler do IE inclui a criação de perfis para funções, oferecendo um relatório detalhado que apresenta o número de chamadas, o tempo gasto e outros pontos de dados. Ainda adiciona o recurso de visualização do relatório como uma árvore de chamadas, de criação de perfis para funções nativas e de exportação dos dados produzidos. Mesmo que falte um analisador de rede, o profiler pode ser acrescido de uma ferramenta genérica como o Fiddler, detalhado mais adiante neste capítulo. Consulte *http://msdn.microsoft .com/en-us/library/dd565628(VS.85).aspx* para obter mais detalhes.

O Profiler do IE8 pode ser encontrado em Developer Tools (Tools → Developer Tools). Depois de pressionar o botão Start Profiling, toda atividade JavaScript subsequente será monitorada e registrada. Ao clicar sobre o botão Stop Profiling (na verdade, o mesmo botão anterior, mas com novo nome), você interrompe o processo e gera um novo relatório. Por padrão, a tecla F5 também inicia o Profiler, e Shift-F5 interrompe sua execução.

O relatório oferece tanto visualização do tempo e da duração de cada chamada como visualização em árvore que exibe a pilha de chamadas da função. A visualização em árvore permite caminhar

pela pilha de chamadas e identificar caminhos lentos de código (veja a Figura 10.4). O profiler do IE utilizará o nome da variável quando nenhum nome estiver disponível para a função.

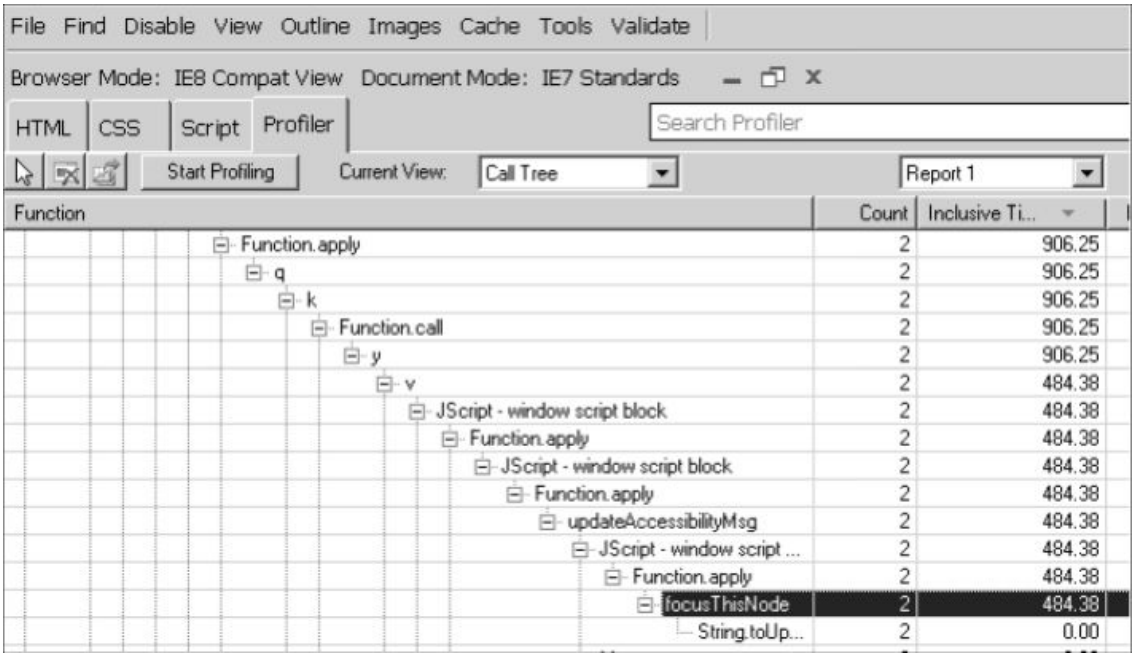


Figura 10.4 – Árvore de chamadas do IE 8 Profiler.

 Clique com o botão direito sobre a tabela de resultado para adicionar ou remover colunas.

O IE Profiler também oferece visualização dos métodos de objetos JavaScript nativos, permitindo que você trace seu perfil e possibilitando a realização de tarefas como a comparação de String::indexOf com RegExp::test para determinar se a propriedade className de um elemento HTML começa com um certo valor:

```
var count = 10000,
  element = document.createElement('div'),
  result, i, time;

element.className = 'foobar';

for (i = 0; i < count; i++) {
  result = /^foo/.test(element.className);
}

for (i = 0; i < count; i++) {
  result = element.className.search(/^foo/);
}

for (i = 0; i < count; i++) {
```

```
  result = (element.className.indexOf('foo') === 0);
}
```

Como podemos ver na figura 10.5, parece haver uma grande variação de tempo entre essas várias abordagens. Lembre-se de que o tempo médio para cada chamada é zero. Métodos nativos são geralmente o último local onde devemos tentar realizar otimizações, mas podem ser um experimento interessante na comparação de abordagens. Não se esqueça também de que com números tão pequenos, os resultados poderão ser inconclusivos devido a erros de arredondamento e a flutuações na memória do sistema.

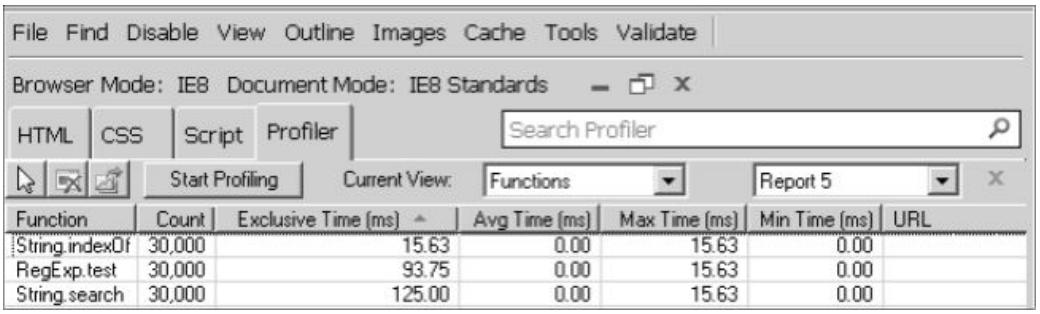


Figura 10.5 – Resultados de perfis para métodos nativos.

Ainda que o IE Profiler atualmente não ofereça uma API JavaScript, ele apresenta uma API de console com recursos de logging. Ela pode ser utilizada na adaptação das funções `console.time()` e `console.timeEnd()` do Firebug, permitindo que os mesmos testes sejam efetuados no IE.

```
if (console && !console.time) {
  console._timers = {};
  console.time = function(name) {
    console._timers[name] = new Date();
  };
  console.timeEnd = function(name) {
    var time = new Date() - console._timers[name];
    console.info(name + ': ' + time + 'ms');
  };
}
```


Os resultados de perfis do IE 8 podem ser exportados em formato .csv pelo botão Export Data.

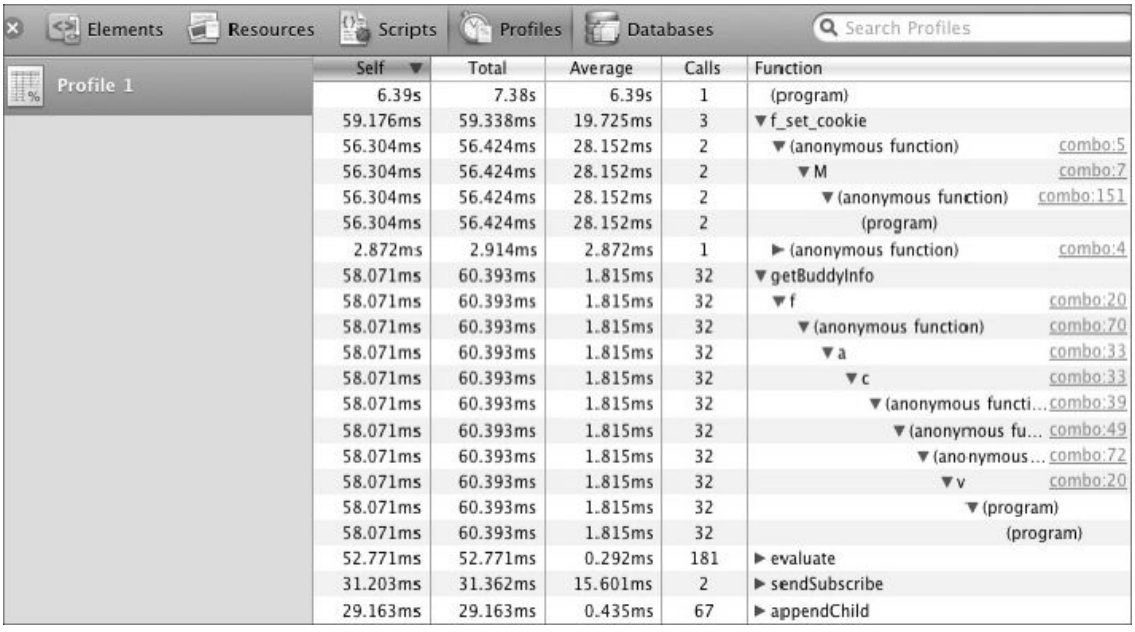
Web Inspector do Safari

O Safari, a partir da versão 4, também oferece um profiler, além de outras ferramentas, incluindo um analisador de rede, como parte de seu pacote Web Inspector. Assim como as ferramentas para desenvolvedores do Internet Explorer, o Web Inspector realiza o perfil de funções nativas e oferece uma árvore de chamadas expansível. Também inclui uma API de console semelhante à do Firebug com funções para criação de perfis e um painel Resource para análise de rede.

Para acessar o Web Inspector, primeiro certifique-se de que o menu Develop está disponível. O menu Develop pode ser habilitado abrindo Preferences → Advanced e clicando na caixa “Show Develop menu in menu bar”. Depois disso, o Web Inspector estará disponível em Develop → Show Web Inspector (ou pelo atalho do teclado Option-Command-I).

Painel de perfis

Ao clicar no botão Profile você produz o painel Profile (Figura 10.6). O botão Enable Profiling habilita o painel Profiles. Para começar a traçar seus perfis, clique no botão Start Profiling (o círculo escuro no canto inferior direito). Clique em Stop Profiling (mesmo botão, agora vermelho) para interromper o perfil e exibir o relatório.



Self	Total	Average	Calls	Function
6.39s	7.38s	6.39s	1	(program)
59.176ms	59.338ms	19.725ms	3	▼f_set_cookie
56.304ms	56.424ms	28.152ms	2	▼(anonymous function) combo:5
56.304ms	56.424ms	28.152ms	2	▼M combo:7
56.304ms	56.424ms	28.152ms	2	▼(anonymous function) combo:151
56.304ms	56.424ms	28.152ms	2	(program)
2.872ms	2.914ms	2.872ms	1	▶(anonymous function) combo:4
58.071ms	60.393ms	1.815ms	32	▼getBuddyInfo
58.071ms	60.393ms	1.815ms	32	▼f combo:20
58.071ms	60.393ms	1.815ms	32	▼(anonymous function) combo:70
58.071ms	60.393ms	1.815ms	32	▼a combo:33
58.071ms	60.393ms	1.815ms	32	▼c combo:33
58.071ms	60.393ms	1.815ms	32	▼(anonymous functi... combo:39
58.071ms	60.393ms	1.815ms	32	▼(anonymous fu... combo:49
58.071ms	60.393ms	1.815ms	32	▼(anonymous... combo:72
58.071ms	60.393ms	1.815ms	32	▼v combo:20
58.071ms	60.393ms	1.815ms	32	▼(program)
58.071ms	60.393ms	1.815ms	32	(program)
52.771ms	52.771ms	0.292ms	181	▶evaluate
31.203ms	31.362ms	15.601ms	2	▶sendSubscribe
29.163ms	29.163ms	0.435ms	67	▶appendChild

Figura 10.6 – Painel Profile do Safari Web Inspector.


Você também podem digitar Option-Shift-Command-P para iniciar/interromper a criação do perfil.

O Safari emulou a API JavaScript do Firebug (console.profile(), console.time() etc.), permitindo o início e a interrupção da criação de perfis de modo programático. A funcionalidade é a mesma do Firebug, permitindo nomear relatórios e cronometragens para facilitar o gerenciamento de seus perfis.


Um nome também pode ser passado a console.profileEnd(). Isso interrompe um perfil específico caso vários perfis estejam sendo executados simultaneamente.

O Safari ainda oferece tanto uma visualização Heavy (de baixo para cima, ou *bottom-up*) das funções analisadas quanto uma visualização Tree (de cima para baixo, ou *top-down*) da pilha de chamadas. A visualização padrão Heavy ordena primeiro as funções mais lentas e permite que você "suba" pela pilha de chamadas, enquanto a visualização Tree permite "mergulhar" a partir do topo até a base do caminho de execução do código partindo da chamada mais externa. A análise da árvore de chamadas pode ajudá-lo a descobrir problemas de desempenho mais sutis relacionados ao modo como uma função pode estar chamando outra.

O Safari também adiciona o suporte a uma propriedade chamada displayName para criação de perfis. Dessa forma, você tem como adicionar nomes a funções anônimas que serão utilizados no relatório de resultado. Considere a seguinte função, atribuída à variável foo:

```
var foo = function() {
  return 'foo!';
};
console.profile('Anonymous Function');
foo();
console.profileEnd();
```

Como mostra a figura 10.7, o relatório de perfil resultante é de difícil compreensão devido à falta de nomes das funções. Ao clicar sobre a URL à direita da função, você exibe a função no contexto do

código-fonte.

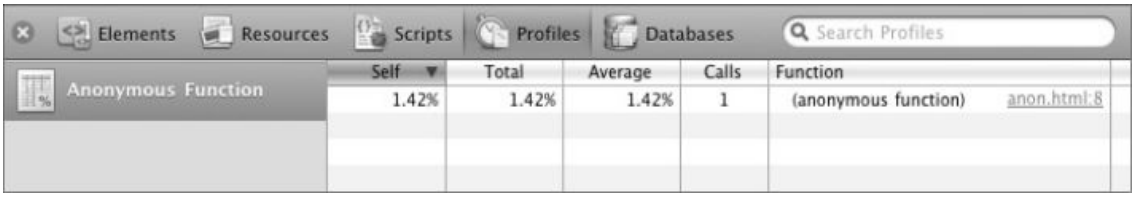


Figura 10.7 – Painel Profile do Web Inspector mostrando uma função anônima.

Ao adicionar um displayName você torna o relatório mais legível. Isso também permite a utilização de nomes mais descritivos, que não sejam limitados a nomes válidos para suas funções:

```
var foo = function() {
  return 'foo!';
};
foo.displayName = 'I am foo';
```

Como mostra a figura 10.8, o displayName agora substitui a função anônima. Entretanto, essa propriedade está disponível apenas em navegadores com base no Webkit. Também exige a refatoração de funções verdadeiramente anônimas, o que não é recomendado. Conforme discutimos, a adição embutida do nome é a forma mais simples de nomear funções anônimas e, além disso, esta abordagem também funciona para outros profilers:

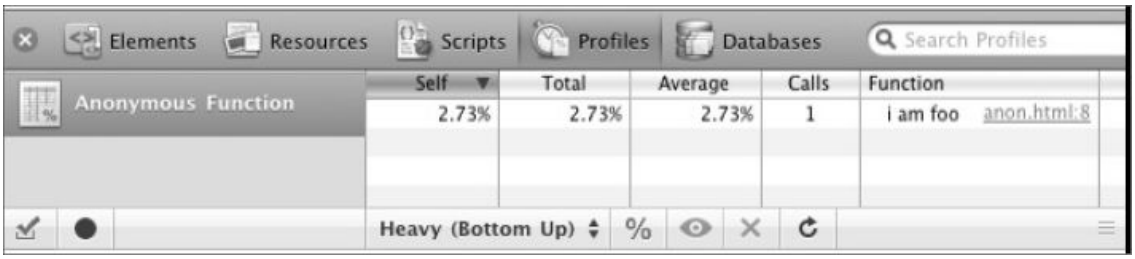


Figura 10.8 – Painel Profile do Web Inspector exibindo displayName.

```
var foo = function foo() {
  return 'foo!';
};
```

Painel Resources

O painel Resources ajuda você a compreender melhor como o Safari está carregando e efetuando o parsing dos scripts e de outros recursos externos. Assim como o painel Net do Firebug, ele oferece

uma visualização dos recursos, mostrando quando uma solicitação foi iniciada e quanto tempo ela levou. Recursos são convenientemente codificados em cores para melhorar sua legibilidade. O painel Resources do Web Inspector separa a indicação do tamanho da indicação do tempo, minimizando a confusão visual (veja a figura 10.9).

Observe que, diferentemente de alguns navegadores, o Safari 4 carrega scripts em paralelo e de modo não bloqueador. Elimina-se a necessidade do bloqueio, garantindo que os scripts sejam executados na ordem correta. Lembre-se de que isso apenas se aplica a scripts inicialmente embutidos no HTML durante o carregamento; scripts adicionados dinamicamente não bloqueiam o carregamento nem a execução (veja o capítulo 1).

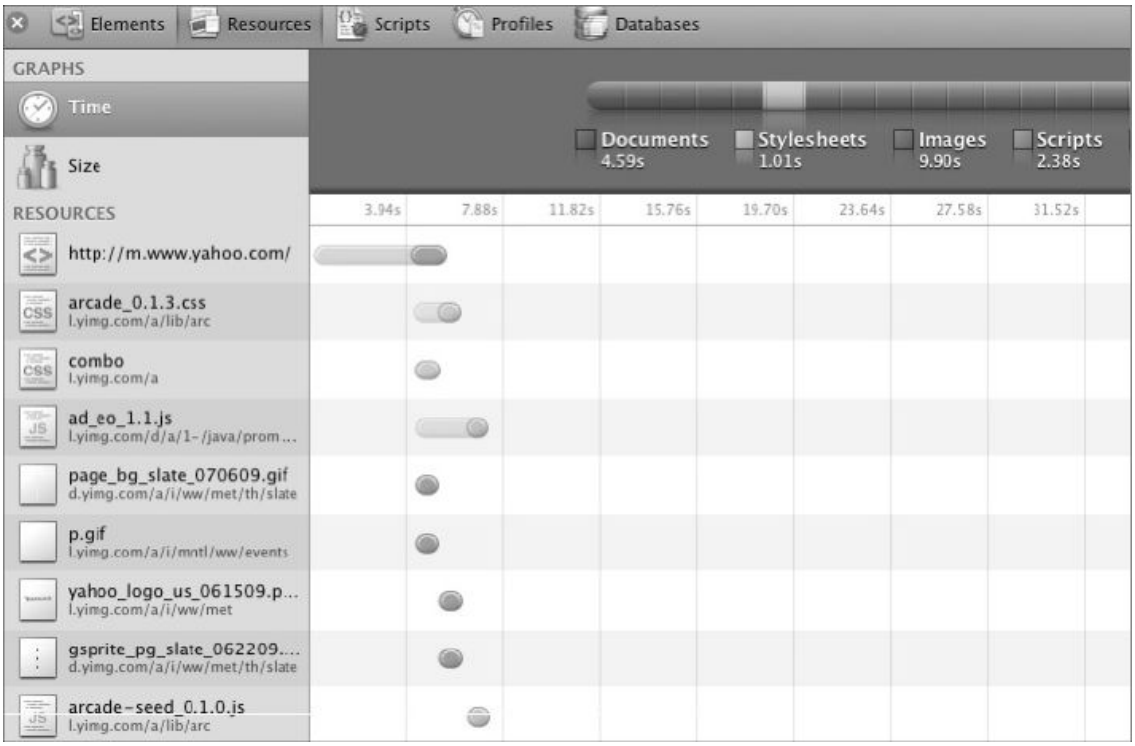


Figura 10.9 – Painel Resources do Safari.

Ferramentas para desenvolvedores do Chrome

O Google também oferece um conjunto de ferramentas de

desenvolvimento para seu navegador Chrome, algumas das quais têm como base o Web Inspector do WebKit/Safari. Além do painel Resources para monitoramento do tráfego da rede, o Chrome adiciona uma visualização em Timeline (linha do tempo) de todos os eventos da página e da rede. Ainda inclui o painel Profiles do Web Inspector e adiciona o recurso de tirar instantâneos da pilha da memória atual. Assim como no caso do Safari, o Chrome é capaz de traçar perfis de funções nativas e implementar a API de console do Firebug, incluindo o console.profile e o console.time.

Como mostra a figura 10.10, o painel Timeline oferece uma visão geral de todas as atividades, categorizadas como “Loading” (carregando), “Scripting” (executando scripts) ou “Rendering” (renderizando). Isso permite que os desenvolvedores rapidamente voltem sua atenção aos aspectos mais lentos do sistema. Alguns eventos contêm uma subárvore de outras linhas de eventos, que pode ser expandida ou ocultada apresentando mais ou menos detalhes na visualização Records.

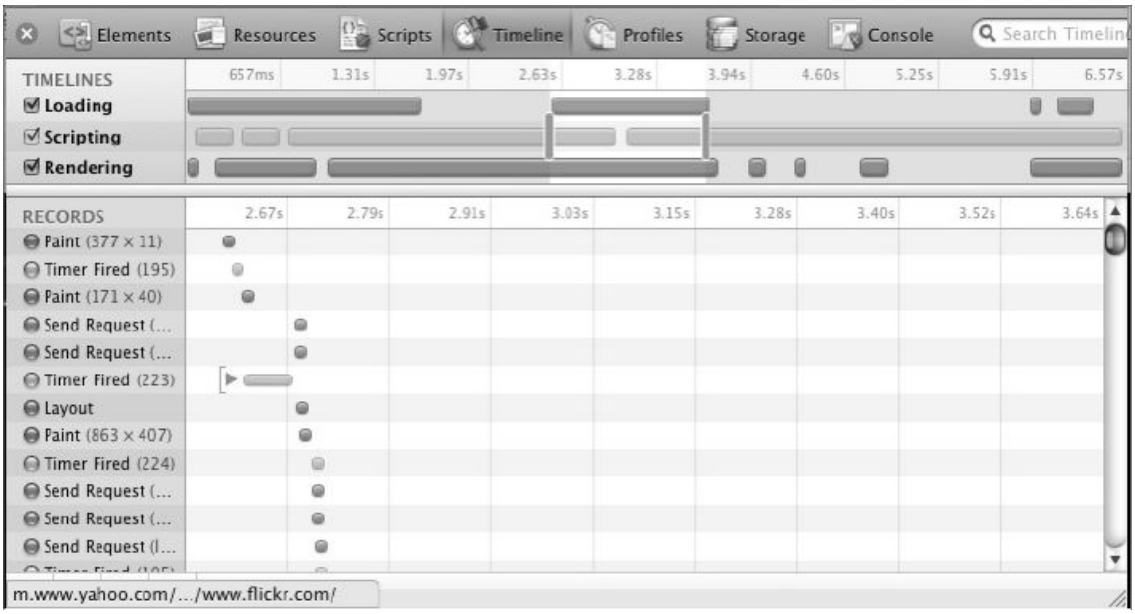


Figura 10.10 – Painel Timeline do Chrome Developer Tools.

Ao clicar sobre o ícone de olho no painel Profiles do Chrome você tira um “instantâneo” (snapshot) da atual pilha da memória. Os resultados são agrupados por construtor e podem ser expandidos,

exibindo cada instância. Esses “retratos” podem ser comparados utilizando a opção “Compared to Snapshot” na base do painel Profiles. As colunas +/- de Count e Size mostram as diferenças entre os itens comparados.

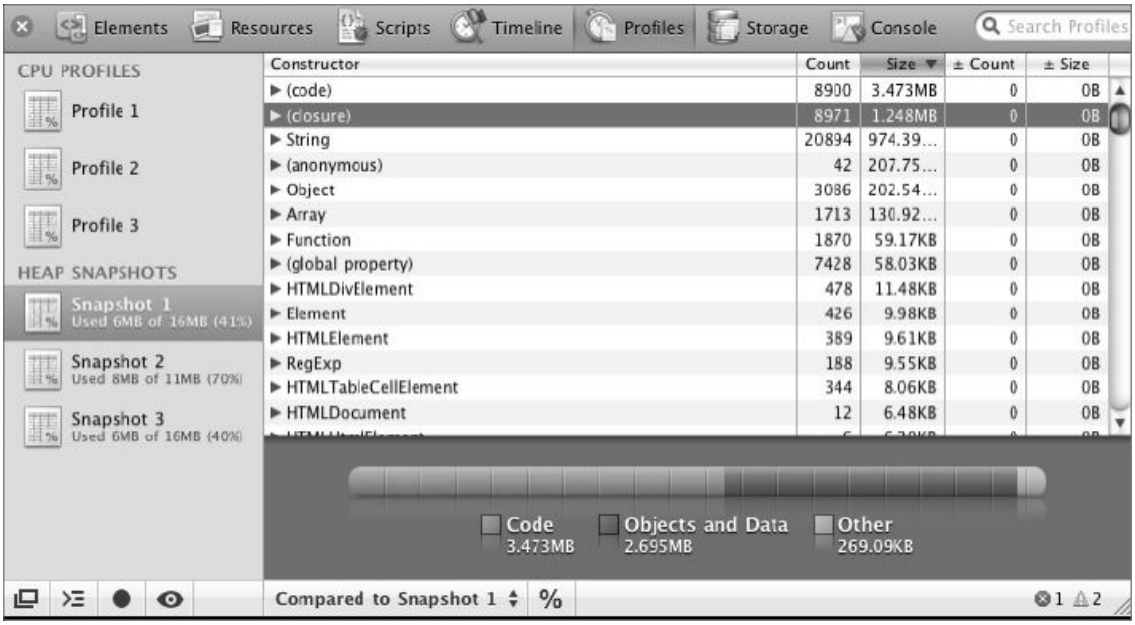


Figura 10.11 – Retrato da pilha do JavaScript no Chrome Developer Tools.

Bloqueio de scripts

Tradicionalmente os navegadores permitem a ocorrência de apenas uma solicitação por vez. Isso é feito para administrar as dependências entre os arquivos. Desde que um arquivo que depende de outro venha depois do primeiro no código-fonte, teremos a garantia de que suas dependências estarão prontas antes de sua execução. Os intervalos que ocorrem entre os scripts podem indicar bloqueios. Navegadores mais recentes, como o Safari 4, o IE 8, o Firefox 3.5 e o Chrome trataram dessa questão permitindo downloads paralelos de scripts, mas bloqueando sua execução, para garantir que as dependências estejam disponíveis. Ainda que isso permita que os recursos sejam baixados mais rapidamente, a renderização da página continuará sendo bloqueada até que todos os scripts tenham executado.

O bloqueio de scripts pode ser agravado pela inicialização mais

lenta de um ou mais arquivos, o que pode indicar a necessidade da realização de perfis e, potencialmente, de otimizações e refatoração. O carregamento dos scripts pode ser lento ou interromper totalmente a renderização da página, deixando o usuário na espera. Ferramentas para análise de rede podem ajudar a identificar e otimizar os intervalos no carregamento de recursos. A visualização dessas falhas na entrega dos scripts fornece uma ideia de quais scripts apresentam lentidão de execução. Pode valer à pena adiar tais scripts até que a página tenha sido renderizada ou possivelmente otimizar e refatorar o código para reduzir o tempo de execução.

Page Speed

O Page Speed é uma ferramenta inicialmente desenvolvida para utilização interna pelo Google e depois lançada como um addon do Firebug que, assim como o painel Net, oferece informações sobre os recursos sendo carregados em uma página web. Todavia, além do tempo de carregamento e do status HTTP, o Page Speed também mostra o intervalo de tempo gasto no parsing e na execução do JavaScript, identificando scripts que podem ser adiados e relatando funções que não estão sendo utilizadas. São informações valiosas que podem ajudar a identificar as áreas para onde devemos voltar nossa atenção, realizando talvez otimizações e refatoração. Visite *http://code.google.com/speed/page-speed/* para obter mais instruções sobre a instalação do Page Speed e outros detalhes do produto.

A opção Profile Deferrable JavaScript, disponível no painel do Page Speed, identifica arquivos que podem ser adiados ou quebrados para entrega de uma carga inicial menor. Muitas vezes, apenas alguns dos scripts sendo executados em uma página são necessários para sua renderização inicial. Na figura 10.12 você pode ver que a maior parte do código sendo carregado não é utilizada antes do disparo do evento load da janela. Ao adiar scripts que não estão sendo utilizados imediatamente, você permite que a

página inicial carregue muito mais rapidamente. Scripts e outros recursos podem ser carregados posteriormente de modo seletivo, conforme se tornam necessários.

Figura 10.12 – Resumo do JavaScript que pode ser adiado apresentado no Page Speed.

O Page Speed ainda adiciona um painel Page Speed Activity ao Firebug. O painel é semelhante ao próprio painel Net do Firebug, com a exceção de que oferece dados mais minuciosos sobre cada solicitação. Os dados incluem um resumo do ciclo de vida de cada script, incluindo as fases de parsing e execução, e fornecem um relato detalhado dos intervalos existentes entre os scripts. Essas características ajudam a identificar em quais áreas devemos concentrar a criação de perfis e possivelmente alertam quanto à necessidade da refatoração de determinados arquivos. Como visto na legenda, a figura 10.13 mostra o tempo gasto no parsing do script com a cor vermelha e o tempo gasto em sua execução em azul. Um longo tempo de execução pode significar a necessidade de examinar determinada seção mais detalhadamente.

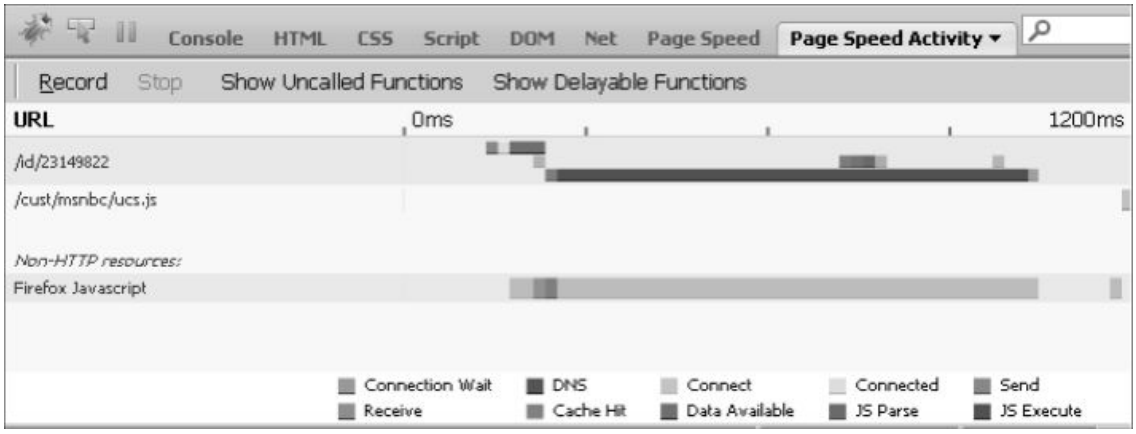


Figura 10.13 – Tempos de parsing e execução no Page Speed.

Devemos estar atentos, pois podemos gastar um tempo considerável efetuando o parsing e o carregamento de scripts que não serão utilizados até que a página tenha sido renderizada. O painel Activity do Page Speed pode também oferecer um relatório destacando quais funções não chegam a ser chamadas e quais podem ser adiadas, com base no momento em que foi feito seu parsing e no momento em que foram chamadas pela primeira vez (Figura 10.14).

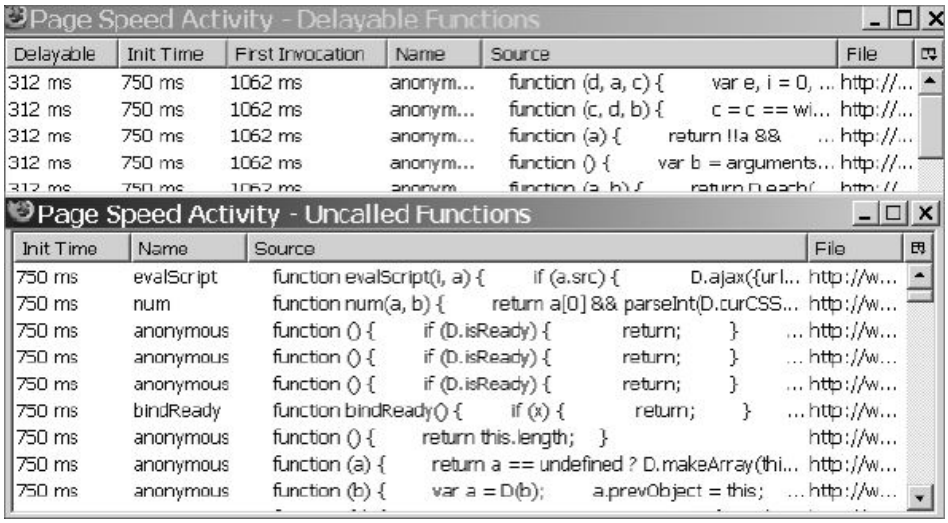


Figura 10.14 – Relatórios de funções que podem ser atrasadas ou que nem chegam a ser chamadas.

Esses relatórios mostram o tempo gasto na inicialização de funções que nunca chegam a ser chamadas ou que poderiam ser chamadas posteriormente. Considere a refatoração do código para remover funções que não são chamadas e para adiar código que não é

necessário durante a fase inicial de renderização e configuração.

Fiddler

O Fiddler é um proxy HTTP de depuração capaz de examinar os recursos que chegam pela conexão de rede e ajudar na identificação da presença de gargalos no carregamento. Criado por Eric Lawrence, o Fiddler é uma ferramenta para análise de rede de uso geral, projetada para o Windows, que oferece relatórios detalhados sobre qualquer navegador ou solicitação web. Visite *http://www.fiddler2.com/fiddler2/* para saber mais sobre sua instalação e outras informações.

Durante a instalação, o Fiddler faz automaticamente a integração com o IE e o Firefox. Um botão é adicionado à barra de ferramentas do IE, enquanto uma entrada é adicionada ao menu Tools do Firefox. O Fiddler também pode ser inicializado manualmente. Qualquer navegador ou aplicação que realize solicitações web pode ser analisado. Durante a execução, todo tráfego Win|NET é roteado por meio do Fiddler, permitindo seu monitoramento e a análise do desempenho dos recursos baixados. Alguns navegadores (p. ex., o Opera e o Safari) não utilizam Win|NET, mas detectarão automaticamente o proxy do Fiddler, desde que ele esteja sendo executado antes da inicialização do navegador. Qualquer programa que permita a configuração de proxies pode ser manualmente apontado para o Fiddler, indicando seu proxy (127.0.0.1, porta: 8888).

Assim como o Firebug, o Web Inspector e o Page Speed, o Fiddler oferece um diagrama em cascata que fornece dados indicando quais recursos demoram mais para carregar e quais podem estar afetando de maneira adversa o carregamento de outros recursos (Figura 10.15).

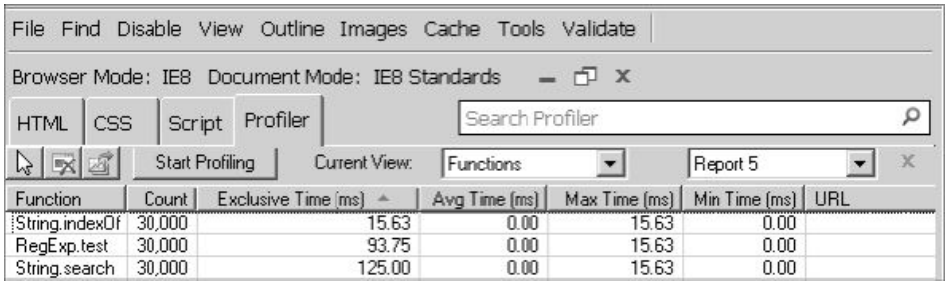


Figura 10.15 – Diagrama em cascata do Fiddler.

A seleção de um ou mais recursos a partir do painel à esquerda carrega sua exibição na janela principal. Clique na aba Timeline para visualizar os recursos que estão sendo transmitidos. Essa visualização propicia a cronometragem de cada recurso em relação aos outros, permitindo estudar os efeitos de diferentes estratégias de carregamento e evidenciando quando algo está bloqueando sua aplicação.

A aba Statistics mostra uma visualização detalhada do desempenho real de todos os recursos selecionados – oferecendo indicações do tempo gasto nas consultas DNS e nas conexões TCP/IP –, assim como uma exibição do tamanho e do tipo dos vários recursos sendo solicitados (Figura 10.16).

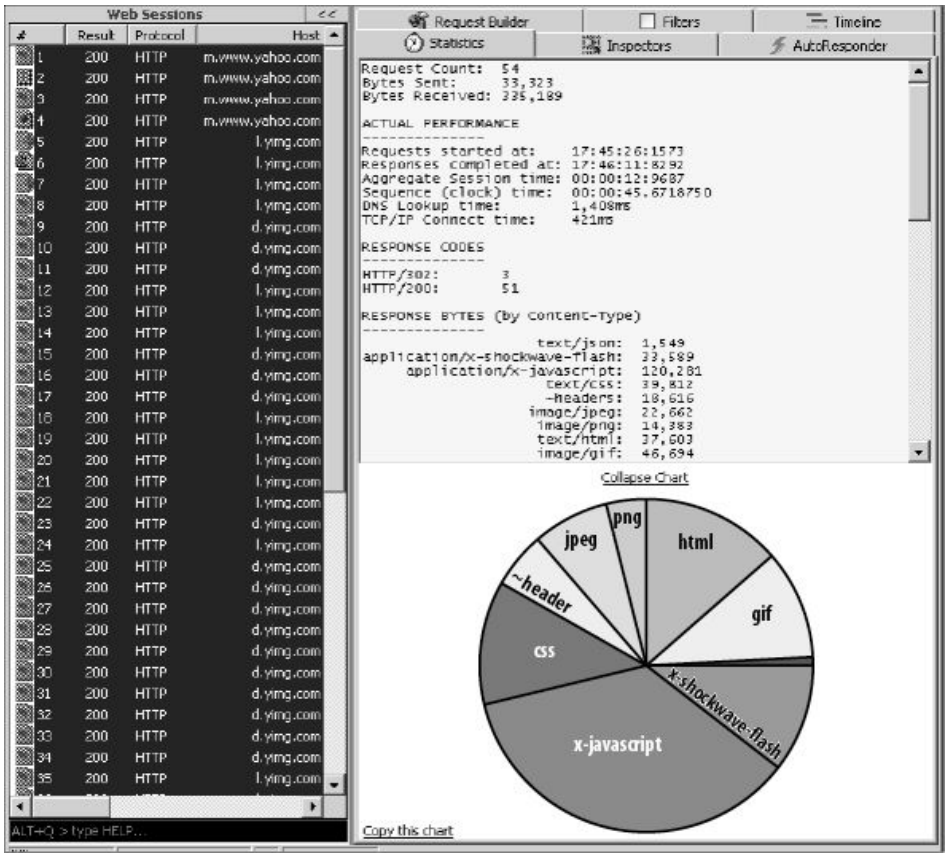


Figura 10.16 – Aba Statistics do Fiddler.

Esses dados podem ajudá-lo a decidir quais áreas merecem um exame mais cuidadoso. Por exemplo, longos intervalos de tempo gastos com as consultas DNS e com as conexões TCP/IP podem indicar problemas na rede. O gráfico torna óbvio quais tipos de recursos são responsáveis pela maior parte do tempo de carregamento da página, identificando candidatos possíveis para carregamento adiado e criação de perfis (no caso dos scripts).

Como o Fiddler está disponível apenas para Windows, vale a pena mencionar um produto shareware chamado Charles Proxy, que funciona tanto no Windows quanto no Mac. Visite *http://www.charlesproxy.com/* para ter acesso a um teste gratuito e documentação detalhada.

YSlow

A ferramenta YSlow fornece informações sobre o carregamento e a execução geral da visualização inicial de uma página. Essa ferramenta foi desenvolvida primeiro internamente pelo Yahoo!, por

Steve Souders, como um addon para o Firefox (via GreaseMonkey). Depois, tornou-se disponível ao público como um addon do Firebug, sendo mantida e atualizada regularmente pelos desenvolvedores do Yahoo!. Visite *http://developer.yahoo.com/yslow/* para obter instruções de instalação e outros detalhes do produto.

O YSlow avalia o carregamento de recursos externos à página, fornece um relatório do desempenho da página e dá dicas oferecendo meios para melhorar a velocidade de carregamento. A avaliação apresentada tem como base pesquisas extensas feitas por peritos em desempenho. Ao aplicar esse feedback e ler mais sobre o que cada avaliação significa você ajuda a garantir a experiência de carregamento mais rápida possível e com o menor número de recursos.

A figura 10.17 mostra a exibição padrão no YSlow de uma página web analisada. Ele fará sugestões para otimização da velocidade de carregamento e renderização da página. Cada uma das avaliações, ou notas, inclui detalhes com informações adicionais e uma explicação do raciocínio por trás da regra.

Figura 10.17 – YSlow: todos os resultados.

Normalmente, a melhoria da avaliação geral da página resultará em um carregamento e execução mais rápidos. A figura 10.18 mostra os resultados filtrados pela opção JAVASCRIPT, junto a alguns conselhos sobre como podemos otimizar a entrega e execução dos scripts.

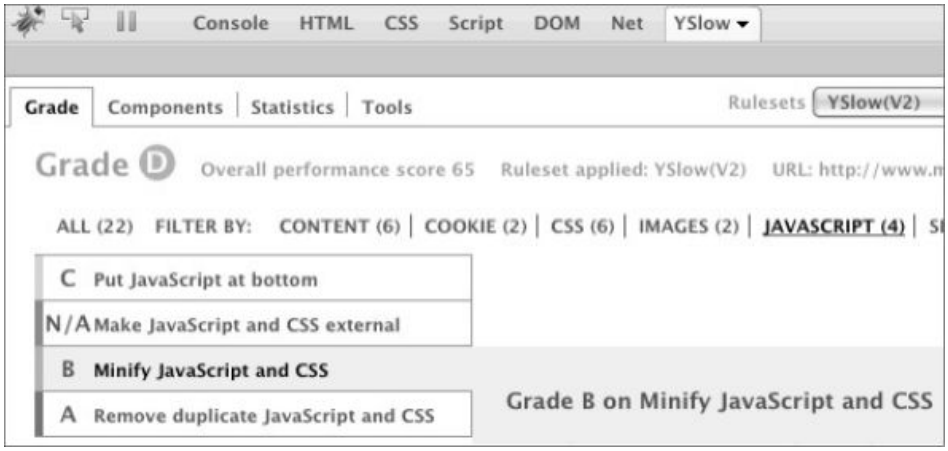


Figura 10.18 – YSlow: resultados JavaScript.

Ao interpretar esses dados, lembre-se de que existem exceções que devem ser consideradas, como a decisão da realização de solicitações múltiplas por scripts *versus* a combinação em um único arquivo e o adiamento de alguns scripts até que a página tenha sido renderizada.

dynaTrace Edição Ajax

Os desenvolvedores do dynaTrace, uma poderosa ferramenta Java/.NET de diagnósticos, lançaram uma “Edição Ajax” capaz de medir o desempenho do Internet Explorer (uma versão para Firefox será lançada em breve). Essa ferramenta gratuita fornece uma análise de desempenho completa, indo da rede e da renderização da página ao tempo de execução de scripts e utilização da CPU. Os relatórios apresentam todos os aspectos juntos, para que você possa facilmente descobrir gargalos que estejam ocorrendo. Além disso, eles também podem ser exportados para maiores análises. Para fazer o download, visite *http://ajax.dynatrace.com/pages/*.

O relatório de Summary mostrado na figura 10.19 fornece uma visão geral, permitindo que você rapidamente determine quais áreas demandam mais atenção. Partindo desse ponto você pode examinar os relatórios mais específicos, obtendo dados mais detalhados no que diz respeito aos aspectos particulares do desempenho de seu site.

A visualização Network mostrada na figura 10.20 oferece um

relatório altamente detalhado do tempo gasto em cada aspecto do ciclo de vida da rede, incluindo as consultas DNS, a conexão e os tempos de respostas do servidor. Esses dados podem conduzi-lo diretamente às áreas que precisam de ajustes. O painel abaixo do relatório mostra os cabeçalhos de solicitação e resposta (à esquerda) e a resposta da solicitação em si (à direita).

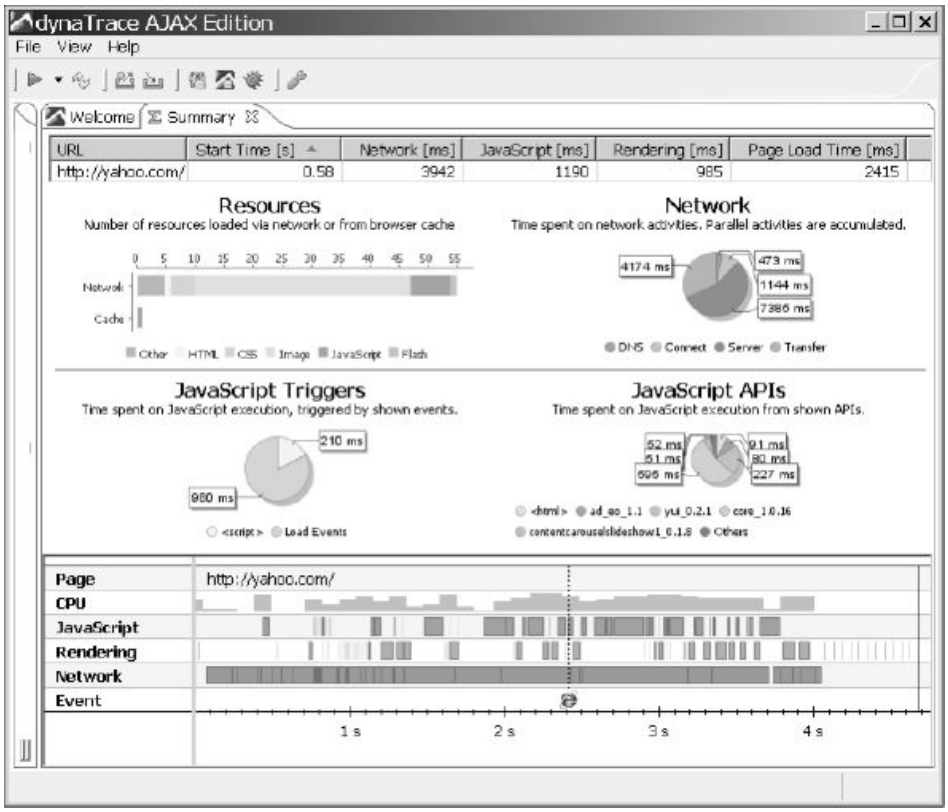


Figura 10.19 – dynaTrace Ajax Edition: relatório Summary.

Ao selecionar a visualização JavaScript Triggers será exibido um relatório detalhado de cada evento que foi disparado durante a análise (veja a figura 10.21). Desse ponto, podemos examinar eventos específicos (“load”, “click”, “mouseover” etc.) para descobrir a raiz dos problemas no desempenho do tempo de execução.

Essa tela inclui qualquer solicitação dinâmica (Ajax) que um evento possa estar disparando e os “callbacks” de script executados como resultado da solicitação. Assim, você é capaz de compreender melhor o desempenho geral que seus usuários experimentam. Essas informações, devido à natureza assíncrona do Ajax, podem

não ser evidentes a partir de um relatório de perfil do script.

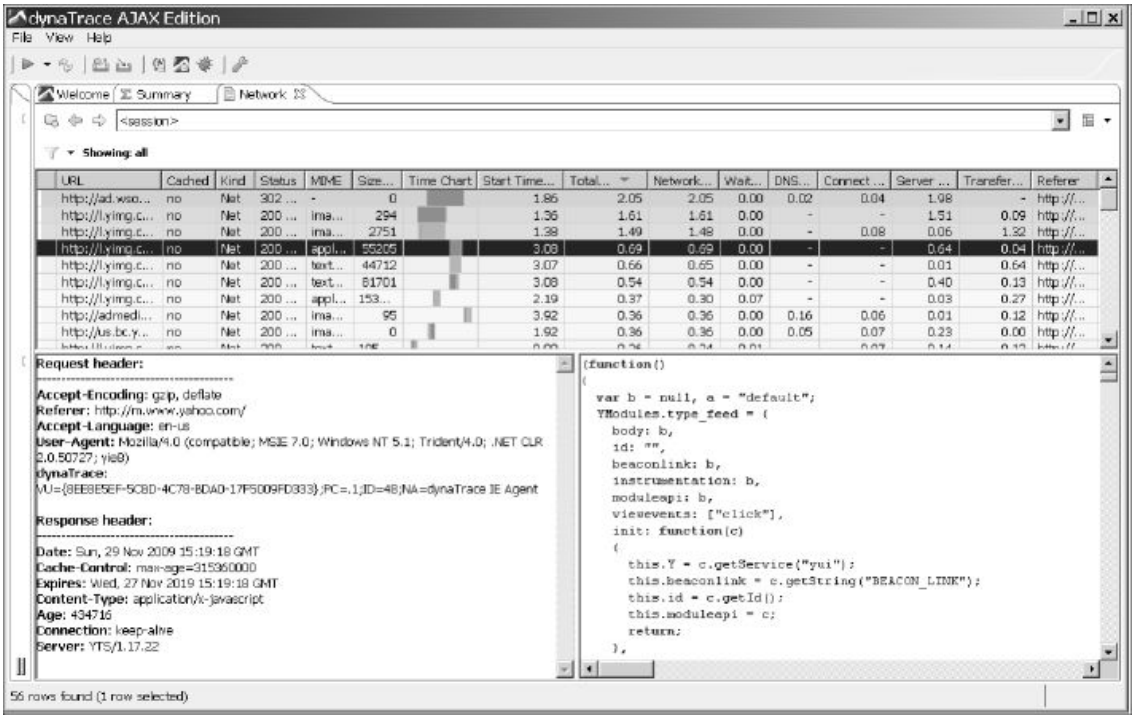


Figura 10.20 – dynaTrace Ajax Edition: relatório Network.

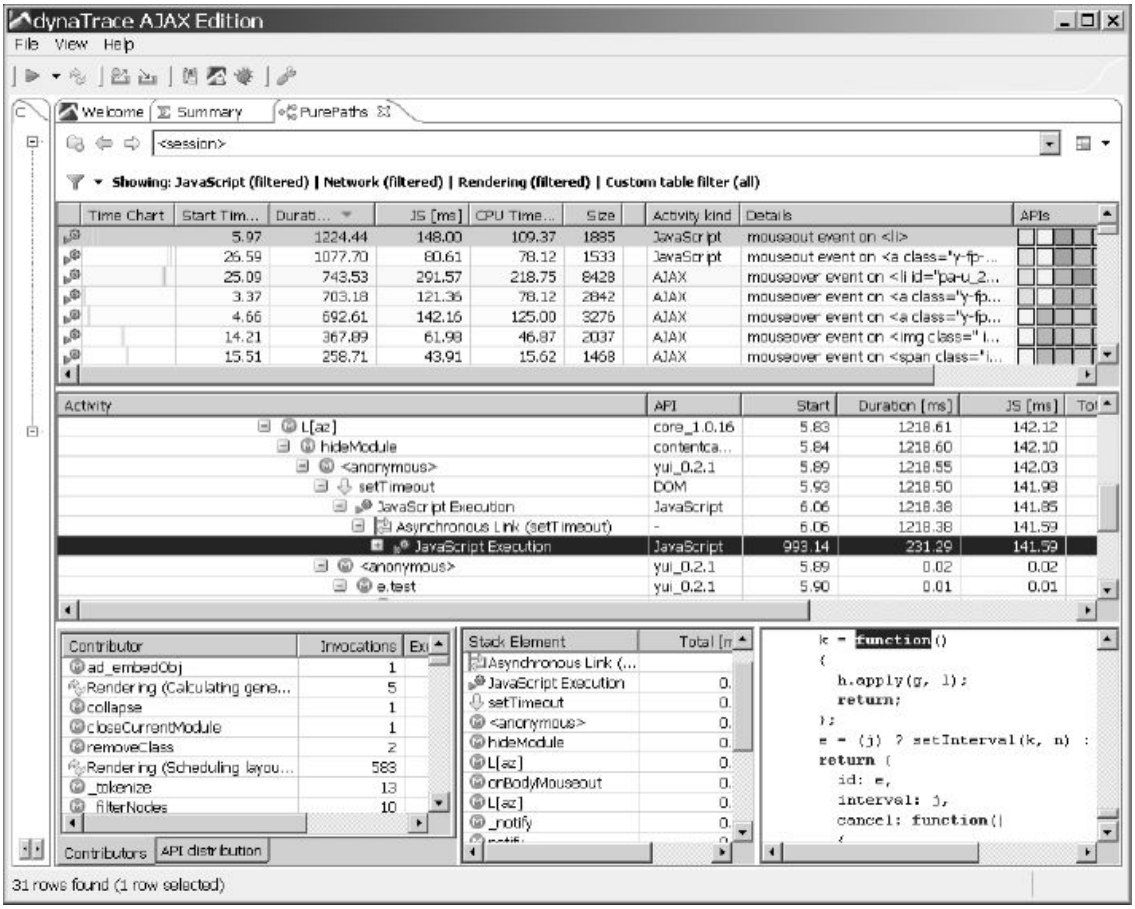


Figura 10.21 – dynaTrace Ajax Edition: painel PurePaths.

Resumo

Quando as páginas web ou as aplicações começam a apresentar lentidão, a análise dos recursos que chegam pela conexão e a criação de perfis enquanto os scripts são executados permitem que você direcione seu trabalho de otimização aos pontos que mais necessitam de ações.

- Utilize uma ferramenta de análise de rede para identificar gargalos no carregamento dos scripts e de outros recursos da página: dessa forma é possível determinar se é necessário o adiamento de scripts ou a criação de perfis.
- Ainda que o bom senso aconselhe a diminuição do número de solicitações HTTP, a utilização sempre que possível de scripts adiados permite que a página carregue mais rapidamente, proporcionando aos usuários uma experiência geral mais

interessante.

- Utilize um profiler, ou criador de perfis, para identificar as áreas lentas na execução de seus scripts. Ao examinar o tempo gasto em cada função, o número de vezes que uma função é chamada e a pilha de chamadas em si você reúne pistas que indicam onde devem ser concentrados seus esforços de otimização.
- Ainda que o tempo gasto e o número de chamadas sejam normalmente os dados mais valiosos, um exame mais profundo da forma como as funções estão sendo chamadas pode apresentar outros candidatos a otimização.

Essas ferramentas ajudam a desmistificar a hostilidade dos ambientes de programação nos quais códigos modernos são executados. Ao utilizá-las antes de iniciar suas otimizações você garante que o tempo de desenvolvimento seja gasto tratando dos problemas certos.

Sobre o autor

Nicholas C. Zakas é um engenheiro de software web especializado em design e implementação de interface de usuários para aplicações web, utilizando JavaScript, Dynamic HTML, CSS, XML e XSLT. Atualmente, é o principal engenheiro de front-end da homepage do Yahoo! e colaborador da biblioteca Yahoo! User Interface (YUI), tendo escrito o Cookie Utility, o Profiler e o YUI Test.

É autor de Professional JavaScript for Web Developers e co-autor de Professional Ajax (ambos pela editora Wrox) e já contribuiu para muitos outros livros. Também já escreveu vários artigos on-line para os sites WebReference, Sitepoint e YUI Blog.

Nicholas regularmente dá palestras sobre desenvolvimento web, JavaScript e boas práticas. Já o fez em empresas como o Yahoo!, LinkedIn, Google e NASA, além de em conferências como a Ajax Experience, a Rich Web Experience e a Velocity.

Por meio de seus textos e suas palestras, Nicholas busca ensinar a outros as lições valiosas que aprendeu em seu trabalho com

algumas das aplicações web mais exigentes e populares do mundo.

Para saber mais sobre Nicholas, acesse *http://www.nczonline.net/about/*.

Colofão

O animal na capa é um mocho-dos-banhados (Asio flammeus). Os tufos em forma de orelha, característicos da ave, são pequenos e aparecem simplesmente como contornos no alto de sua grande cabeça. Tornam-se mais visíveis, entretanto, quando a coruja se sente ameaçada e assume uma postura defensiva. De tamanho médio, essa ave tem olhos amarelos, plumagem que apresenta nuances de marrom, peito claro e rajado e barras escuras em suas amplas asas.

Uma das espécies de ave mais comuns do mundo, esse pássaro migratório pode ser encontrado em todos os continentes, com exceção da Austrália e da Antártica. O mocho-dos-banhados vive em campo aberto, como prados, charcos ou tundra. Ele captura pequenos mamíferos, como ratos silvestres (e pássaros ocasionais), voando baixo, próximo ao solo, ou se posicionando sobre pequenas árvores e mergulhando em direção a suas presas. É mais ativo ao amanhecer, no início da noite e no entardecer.

O voo do mocho-dos-banhados é muitas vezes comparado ao de uma mariposa ou morcego, visto que se move para frente e para trás com batidas de asa lentas e irregulares. Durante sua temporada de acasalamento, os machos realizam exibições aéreas espetaculares de corte, nas quais batem suas asas, elevam-se em círculos até grandes alturas e mergulham rapidamente em direção ao solo. O mocho-dos-banhados é também uma espécie de ator no reino animal – capaz de fingir-se de morto para evitar ser visto ou de aparentar uma asa aleijada para atrair predadores, afastando-os de seu ninho.

A imagem da capa é do livro Natural History da editora Cassell.


PRINCÍPIOS DE
ORIENTAÇÃO A OBJETOS
EM JAVASCRIPT
NICHOLAS C. ZAKAS
novatec
no starch press

Princípios de Orientação a Objetos em JavaScript

Zakas, Nicholas C.

9788575225899

128 páginas

Se você já usou uma linguagem orientada a objetos mais tradicional, como C++ ou Java, o JavaScript provavelmente não parecerá uma linguagem orientada a objetos. Ela não tem conceito de classes, e você nem mesmo precisa definir objetos para começar a programar. Mas não se engane – o JavaScript é uma linguagem orientada a objetos incrivelmente eficiente e expressiva, que coloca muitas decisões de design diretamente em suas mãos. No livro Princípios de Orientação a Objetos em JavaScript, Nicholas C. Zakas explora minuciosamente a natureza orientada a objetos do JavaScript, revelando a implementação única de herança e outras características fundamentais da linguagem. Você irá aprender: A diferença entre valores primitivos e de referência O que faz com que as funções em JavaScript sejam únicas As diversas maneiras de criar objetos Como

definir seus próprios construtores Como entender e trabalhar com protótipos Padrões de herança para tipos e objetos Princípios de Orientação a Objetos em JavaScript proporcionará até mesmo aos desenvolvedores mais experientes um entendimento mais profundo de JavaScript. Descubra os segredos de como os objetos funcionam em JavaScript, para que você escreva um código mais claro, flexível e eficiente.


UMA INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES
COM EXEMPLOS E EXERCÍCIOS PARA INICIANTES
Lógica de Programação e Algoritmos com JavaScript
novatec
Edécio Fernando Iepsen

Lógica de Programação e Algoritmos com JavaScript

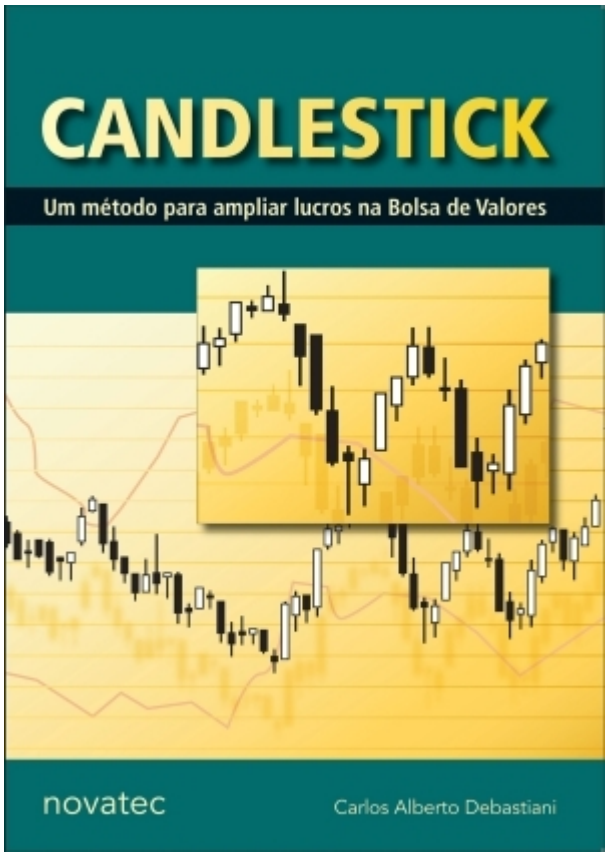
Iepsen, Edécio Fernando

9788575226575

320 páginas

Os conteúdos abordados em Lógica de Programação e Algoritmos são fundamentais a todos aqueles que desejam ingressar no universo da Programação de Computadores. Esses conteúdos, no geral, impõem algumas dificuldades aos iniciantes. Neste livro, o autor utiliza sua experiência de mais de 15 anos em lecionar a disciplina de Algoritmos em cursos de graduação, para trabalhar o assunto passo a passo. Cada capítulo foi cuidadosamente planejado a fim de evitar a sobrecarga de informações ao leitor, com exemplos e exercícios de fixação para cada assunto. Os exemplos e exercícios são desenvolvidos em JavaScript, linguagem amplamente utilizada no desenvolvimento de páginas para a internet. Rodar os programas JavaScript não exige nenhum software adicional; é preciso apenas abrir a página em seu navegador favorito. Como o aprendizado de Algoritmos

ocorre a partir do estudo das técnicas de programação e da prática de exercícios, este livro pretende ser uma importante fonte de conhecimentos para você ingressar nessa fascinante área da programação de computadores. Assuntos abordados no livro: Fundamentos de Lógica de Programação Programas de entrada, processamento e saída Integração do código JavaScript com as páginas HTML Estruturas condicionais e de repetição Depuração de Programas JavaScript Manipulação de listas de dados (vetores) Operações sobre cadeias de caracteres (strings) e datas Eventos JavaScript e funções com passagem de parâmetros Persistência dos dados de um programa com localStorage Inserção de elementos HTML via JavaScript com referência a DOM No capítulo final, um jogo em JavaScript e novos exemplos que exploram os recursos discutidos ao longo do livro são apresentados com comentários e dicas, a fim de incentivar o leitor a prosseguir nos estudos sobre programação.

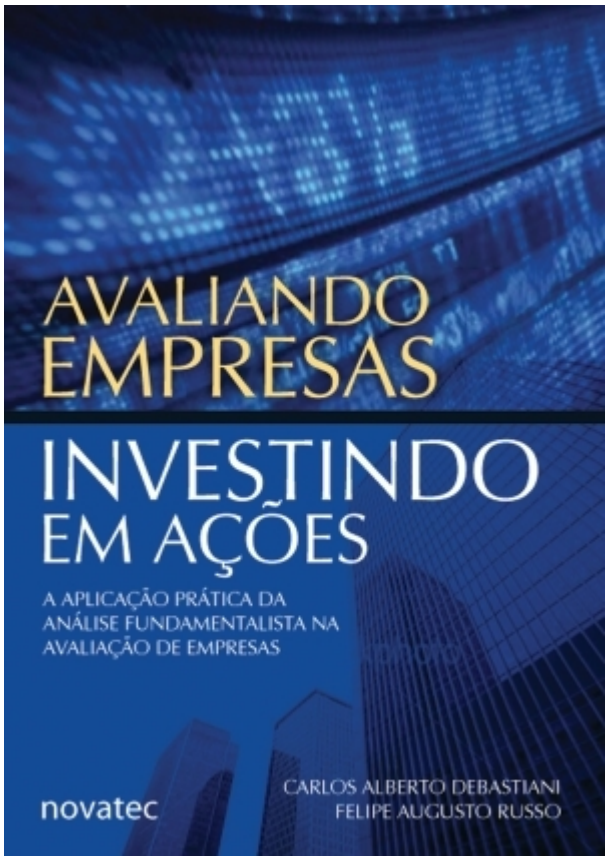

CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto
9788575225943
200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro

possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.


Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por

meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.