

Fundamentos de jQuery (jQuery Fundamentals)

Rebecca Murphey [<http://www.rebeccamurphey.com>]

Fundamentos de jQuery (jQuery Fundamentals)

Rebecca Murphey [<http://www.rebeccamurphey.com>]

Herberth Amaral (tradução e revisão), Elvis Luciano Guimarães (tradução e revisão), Diego Guimarães (tradução), Diego Caxito (tradução), Luciana Balieiro (revisão), Washington Botelho (revisão), Giovani Generali (revisão)

Copyright © 2010

Licensed by Rebecca Murphey under the Creative Commons Attribution-Share Alike 3.0 United States license [<http://creativecommons.org/licenses/by-sa/3.0/us/>]. You are free to copy, distribute, transmit, and remix this work, provided you attribute the work to Rebecca Murphey as the original author and reference the GitHub repository for the work [<http://github.com/rmurphey/jqfundamentals>]. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. Any of the above conditions can be waived if you get permission from the copyright holder. For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to the license [<http://creativecommons.org/licenses/by-sa/3.0/us/>].

Índice

1. Bem vindo	1
Obtendo o código	1
Software	1
Adicionando JavaScript à sua página	1
Debugando JavaScript	2
Exercícios	2
Convenções usadas neste livro.	2
Material de referência	3
I. JavaScript 101	4
2. O Básico de JavaScript	5
Introdução	5
O Básico de sintaxe	5
Operadores	5
Operadores Básicos	5
Operações sobre números & Strings	6
Operadores lógicos	6
Operadores de comparação	7
Código condicional	7
Coisas Verdadeiras e Falsas	8
Associação Condicional de Variáveis Com o Operador Ternário	9
Declaração Switch	9
Laços	10
O laço for	10
O laço while	11
O laço do-while	11
Parando e continuando	12
Palavras Reservadas	12
Arrays	14
Objetos	15
Funções	16
Usando funções	16
Funções anônimas autoexecutáveis	17
Funções como argumento	17
Testando Tipo	18
Escopo	18
Closures	20
II. jQuery: Conceitos básicos	22
3. O básico de jQuery	23
\$(document).ready()	23
Selecionando elementos	23
Minha seleção contém algum elemento?	25
Salvando seleções	25
Refinando & Filtrando Seleções	26
Seletores relacionados à formulários	26
Trabalhando com seleções	27
Encadeamento	27
Getters & Setters	28
CSS, Styling, & Dimensões	28
Usando classes do CSS para estilos	28
Dimensões	29
Atributos	29

Travessia	30
Manipulando elementos	30
Obtendo e Definindo informações sobre elementos	31
Movendo, copiando e removendo elementos	31
Criando novos elementos	33
Manipulando atributos	34
Exercícios	34
Selecionando	34
Atravessamento	35
Manipulando	35
4. Núcleo do jQuery	36
\$ vs \$()	36
Métodos Utilitários	36
Verificando tipos	37
Métodos de Dados	38
Detecção de Navegador & Funcionalidades	39
Evitando Conflitos com Outras Bibliotecas	39
5. Eventos	40
Introdução	40
Conectando Eventos a Elementos	40
Conectando Eventos para Executar Apenas uma vez	40
Desconectando Eventos	41
Usando Namespaces com Eventos	41
Por Dentro da Função Manipuladora de Evento	41
Disparando Eventos	42
Aumentando a Performance com Delegação de Evento	42
Desvinculando Eventos Delegados	43
Auxiliares de Eventos	43
\$.fn.hover	43
\$.fn.toggle	44
Exercícios	44
Criando uma dica de Input	44
Adicionando navegação por abas	44
6. Efeitos	46
Visão Geral	46
Efeitos Embutidos	46
Mudando a Duração de Efeitos Embutidos	46
Fazendo algo quando um Efeito tiver Terminado	47
Efeitos customizados com \$.fn.animate	47
Easing	48
Controlando efeitos	48
Exercícios	49
Mostrando texto oculto	49
Criando menus dropdown	49
Crie um slideshow	49
7. Ajax	50
Introdução	50
Conceitos Chave	50
GET vs. POST	50
Tipos de dados	50
A é de assíncrono	51
Regra da mesma origem e JSONP	51
Ajax e o Firebug	51
Métodos do jQuery relacionados ao Ajax	52

\$.ajax	52
Métodos de conveniência	54
\$.fn.load	55
Ajax e formulários	56
Trabalhando com JSONP	56
Eventos do Ajax	57
Exercícios	57
Carregar conteúdo externo	57
Carregando conteúdo usando JSON	58
8. Plugins	59
O que exatamente é um plugin?	59
Como criar um plugin básico	59
Procurando & Avaliando Plugins	61
Escrevendo Plugins	61
Escrevendo plugins com estado utilizando a fábrica de widgets do jQuery UI	64
Adicionando métodos a um Widget	65
Trabalhando com Opções de Widget	67
Adicionando Callbacks	68
Limpando	70
Conclusão	70
Exercícios	71
Fazendo uma tabela ordenável	71
Escreva um plugin para listar tabelas	71
III. Tópicos Avançados	72
Esta seção é um trabalho em andamento	73
9. Melhores práticas para performance	74
Armazene o length em loops	74
Adicione novo conteúdo fora de um loop	74
Mantenha as coisas DRY	74
Cuidado com funções anônimas	75
Otimize seletores	76
Seletores baseados em ID	76
Especificidade	76
Evite o seletor universal	76
Use Delegação de Evento	77
Desanexe elementos para trabalhar com eles	77
Use folhas de estilo para mudar o CSS em vários elementos	77
Use \$.data ao invés de \$.fn.data	78
Não faça nada quando não tiver elementos	78
Definição de Variável	78
Condicionais	79
Não trate o jQuery como uma Caixa Preta	79
10. Organização de código	80
Visão Geral	80
Conceitos Chave	80
Encapsulamento	80
O literal objeto	80
O Module Pattern	84
Gerenciando dependências	87
Obtendo o RequireJS	87
Utilizando o RequireJS com jQuery	87
Criando módulos reusáveis com RequireJS	88
Otimizando seu código: A ferramenta de build do RequireJS	90
Exercícios	91

Crie um módulo Portlet	91
11. Eventos Customizados	92
Introdução à Eventos Customizados	92
Uma aplicação de amostra	94

Lista de Exemplos

1.1. Um exemplo de JavaScript inline	1
1.2. Um exemplo de inclusão de JavaScript externo	1
1.3. Exemplo de um exemplo	3
2.1. Uma simples declaração de variável	5
2.2. Espaços em branco não interferem fora das aspas.	5
2.3. Parênteses indicam precedência	5
2.4. Tabulação melhora a leitura, mas não tem significado especial	5
2.5. Concatenação	5
2.6. Multiplicação e divisão	5
2.7. Incrementando e decrementando	6
2.8. Adição vs. Concatenação	6
2.9. Forçando uma string atuar como um número	6
2.10. Forçando uma string a atuar como um número (usando o operador unário de adição)	6
2.11. Os operadores lógicos AND e OR	6
2.12. Operadores de comparação	7
2.13. Controle de fluxo	8
2.14. Valores que são avaliados como true	8
2.15. Valores que são avaliados como false	8
2.16. O operador ternário	9
2.17. Uma declaração switch	9
2.18. Laços	10
2.19. Um laço for típico	11
2.20. Um laço while típico	11
2.21. Um laço while com uma condição e incremento combinado	11
2.22. Um laço do-while	12
2.23. Parando um laço	12
2.24. Pulando para a próxima iteração de um loop	12
2.25. Um array simples	14
2.26. Acessando os itens do array pelo índice	15
2.27. Testando o tamanho de um array	15
2.28. Mudando o valor de um item do array	15
2.29. Adicionando elementos em um array	15
2.30. Trabalhando com arrays	15
2.31. Criando um "literal objeto"	15
2.32. Declaração de Função	16
2.33. Expressão de Função Nomeada	16
2.34. Uma função simples	16
2.35. Uma função que retorna um valor	16
2.36. Uma função que retorna outra função	17
2.37. Uma função anônima auto-executável	17
2.38. Passando uma função anônima como argumento	17
2.39. Passando uma função nomeada como argumento	17
2.40. Testando o tipo de várias variáveis	18
2.41. Funções tem acesso a variáveis definidas num mesmo escopo	19
2.42. Código fora do escopo em que uma variável foi definida não tem acesso à variável.	19
2.43. Variáveis com o mesmo nome podem existir em escopos diferentes com valores diferentes... ..	19
2.44. Funções podem "ver" mudanças em valores de variáveis depois que a função é definida	20
2.45. Insanidade do Escopo	20
2.46. Como bloquear no valor de i?	21
2.47. Bloqueando no valor de i com uma closure	21
3.1. Um bloco \$(document).ready()	23

3.2. Atalho para <code>\$(document).ready()</code>	23
3.3. Passando uma função nomeada ao invés de uma anônima	23
3.4. Seleccionando elementos por ID	23
3.5. Seleccionando elementos pelo nome da classe	23
3.6. Seleccionando elementos por atributo	24
3.7. Seleccionando elementos através da composição de seletores CSS	24
3.8. Pseudo-seletores	24
3.9. Testando se uma seleção contém elementos.	25
3.10. Armazenando seleções em variáveis	26
3.11. Refinando seleções	26
3.12. Usando pseudo-seletores relacionados à formulários	27
3.13. Encadeamento	27
3.14. Formatando código encadeado	27
3.15. Restaurando sua seleção original usando <code>\$.fn.end</code>	27
3.16. O método <code>\$.fn.html</code> usado como setter	28
3.17. O método <code>html</code> usado como getter	28
3.18. Pegando propriedades CSS	28
3.19. Setando propriedades CSS	28
3.20. Trabalhando com classes	29
3.21. Métodos básicos de dimensões	29
3.22. Definindo atributos	29
3.23. Obtendo atributos	30
3.24. Movendo pelo DOM usando métodos de travessia.	30
3.25. Iterando sobre uma seleção	30
3.26. Mudando o HTML de um elemento.	31
3.27. Movendo elementos usando outras formas	32
3.28. Fazendo uma cópia de um elemento	32
3.29. Criando novos elementos	33
3.30. Criando um novo elemento com um objeto atributo	33
3.31. Inserindo um novo elemento na página	33
3.32. Criando e adicionando um elemento à página ao mesmo tempo	33
3.33. Manipulando um único atributo	34
3.34. Manipulando múltiplos atributos	34
3.35. Usando uma função para determinar um novo valor de atributo	34
4.1. Verificando o tipo de um valor arbitrário	38
4.2. Armazenando e recuperando dados relacionados a um elemento	38
4.3. Armazenando um relacionamento entre elementos usando <code>\$.fn.data</code>	38
4.4. Colocando jQuery em modo no-conflict	39
4.5. Usando o <code>\$</code> dentro de uma função anônima auto-executada	39
5.1. Vinculando Eventos Usando um Método de Conveniência	40
5.2. Vinculando eventos usando o método <code>\$.fn.bind</code>	40
5.3. Vinculando eventos usando o método <code>\$.fn.bind</code> com dados	40
5.4. Trocando manipuladores usando o método <code>\$.fn.one</code>	41
5.5. Desconectando todos os manipuladores de clique em uma seleção	41
5.6. Desconectando um manipulador de clique específico	41
5.7. Usando Namespaces com eventos	41
5.8. Impedindo um link de ser seguindo	42
5.9. Disparando um evento do jeito certo	42
5.10. Delegação de Evento usando <code>\$.fn.delegate</code>	43
5.11. Delegação de Evento usando <code>\$.fn.live</code>	43
5.12. Desvinculando eventos delegados	43
5.13. A função auxiliar <code>hover</code>	44
5.14. A função auxiliar <code>toggle</code>	44
6.1. Um uso básico de um efeito embutido	46

6.2. Configurando a duração de um efeito	46
6.3. Acrescentando definições de velocidade customizadas ao <code>jQuery.fx.speeds</code>	47
6.4. Executando código quando uma animação tiver completado	47
6.5. Executa uma callback mesmo se não houver elementos para animar	47
6.6. Efeitos customizados com <code>\$.fn.animate</code>	48
6.7. Easing por propriedade	48
7.1. Usando o método <code>core \$.ajax</code>	53
7.2. Usando os métodos de conveniência do jQuery	55
7.3. Usando o <code>\$.fn.load</code> para popular um elemento	56
7.4. Usando o <code>\$.fn.load</code> para popular um elemento baseado no seletor	56
7.5. Transformando os dados de um formulário em uma string de requisição	56
7.6. Criando um array de objetos contendo dados do formulário.	56
7.7. Usando YQL e JSONP	57
7.8. Configurando um indicador de carregamento usando eventos do Ajax	57
8.1. Criando um plugin para adicionar e remover uma classe no hover	62
8.2. O Padrão de Desenvolvimento de Plugins do Mike Alsup	63
8.3. Um plugin simples, stateful utilizando a fábrica de widgets do jQuery UI.	64
8.4. Passando opções para um Widget	65
8.5. Setando opções padrão para um Widget	65
8.6. Criando métodos para um Widget	66
8.7. Chamando métodos numa instância do plugin	67
8.8. Respondendo quando a opção é setada	67
8.9. Provendo callbacks para extensão do usuário	68
8.10. Vinculando à eventos do Widget	69
8.11. Adicionando um método <code>destroy</code> à um Widget	70
10.1. Um literal objeto	81
10.2. Usando um literal de objeto numa funcionalidade do jQuery	83
10.3. O module pattern	84
10.4. Usando o module patterns numa funcionalidade do jQuery	86
10.5. Utilizando o RequireJS: Um exemplo simples	87
10.6. Um simples arquivo JavaScript com dependências	88
10.7. Definindo um módulo do RequireJS que não tem dependências	89
10.8. Definindo um módulo do RequireJS com dependências	89
10.9. Definindo um módulo do RequireJS que retorna uma função	89
10.10. Um arquivo de configuração de build do RequireJS	90

Capítulo 1. Bem vindo

jQuery está se tornando rapidamente uma habilidade fundamental para desenvolvedores de interface. O objetivo deste livro é prover uma visão geral do jQuery, uma biblioteca para JavaScript; quando você terminar com este livro, você deverá conseguir completar tarefas básicas usando jQuery, e conseguirá uma base sólida se continuar estudando. Este livro foi feito para servir de material para ser usado em sala de aula, mas pode servir para estudo individual.

Esta é uma aula prática. Nós vamos usar um pouco de tempo para cobrir conceitos, e então você terá a chance de trabalhar num exercício relacionado ao conceito. Alguns dos exercícios podem parecer triviais; outros podem ser assustadores. Em ambos os casos, não há problema; o objetivo é simplesmente fazer você ficar confortável com problemas comuns que você irá resolver no uso do jQuery. As soluções de exemplo para todos os exercícios estão inclusos no código de amostra.

Obtendo o código

O código que iremos usar neste livro está num repositório do Github [<http://github.com/rmurphey/jqfundamentals>]. Você pode baixar um arquivo .zip ou .tar contendo o código e então descompactá-lo para usar no seu servidor. Se você souber usar o git, você será bem vindo para clonar ou 'forkar' o repositório.

Software

Você precisará das seguintes ferramentas para aproveitar as aulas ao máximo:

- O navegador Firefox
- A extensão Firebug para o Firefox
- Um editor de textos
- Para as partes com Ajax: um servidor local (como o MAMP ou WAMP), ou um cliente FTP ou SSH para acessar um servidor remoto.

Adicionando JavaScript à sua página

JavaScript pode ser incluído inline ou incluindo um arquivo externo através de uma tag script. A ordem em que você inclui o JavaScript é muito importante: as dependências precisam ser incluídas antes do script que depende dela.

Para o bem da performance da página, o JavaScript deve ser colocado o mais próximo possível do fim do seu HTML. Múltiplos arquivos JavaScript devem ser combinados para uso em produção.

Exemplo 1.1. Um exemplo de JavaScript inline

```
<script>
console.log('olah');
</script>
```

Exemplo 1.2. Um exemplo de inclusão de JavaScript externo

```
<script src='/js/jquery.js'></script>
```

Debugando JavaScript

Uma ferramenta de debugging é essencial para desenvolvimento em JavaScript. O Firefox provê um debugger através da extensão Firebug; Chrome e Safari possuem consoles embutidos.

Cada console oferece:

- Editores para testar JavaScript.
- Um inspetor para analisar o código gerado da sua página
- Um visualizador de rede ou recursos, para examinar as requisições de rede

Quando você estiver escrevendo código em JavaScript, você pode usar os seguintes métodos para enviar mensagens ao console:

- `console.log()` para enviar mensagens de log
- `console.dir()` para registrar um objeto navegável no log
- `console.warn()` para registrar avisos (warnings) no log
- `console.error()` para registrar mensagens de erro no log

Outros métodos para console estão disponíveis, apesar deles terem uma diferença em cada navegador. Os consoles também tem a habilidade de setar breakpoints e observar expressões no seu código para propósitos de debug.

Exercícios

A maioria dos capítulos no livro são concluídos com um ou mais exercícios. Para alguns exercícios, você poderá trabalhar diretamente no Firebug; para outros, você precisará incluir outros scripts depois da tag de script do jQuery nos exercícios individuais.

Em alguns casos, você precisará consultar a documentação do jQuery para concluir um exercício, pois, por praticidade, não cobriremos todas as informações que são relevantes no livro. O jQuery é uma biblioteca grande, e aprender a encontrar respostas na documentação é uma parte importante no processo.

Aqui vão algumas sugestões para combater estes problemas:

- Primeiro, tenha certeza que você entende completamente o problema que você irá pedir por solução.
- Depois, descubra quais elementos você precisará acessar para conseguir resolver o problema e determine como você irá conseguir estes elementos. Use o Firebug para verificar se você está obtendo os elementos que está procurando.
- Por último, descubra o que você precisa fazer com os elementos para resolver o problema. Pode ser útil escrever comentários explicando o que você irá fazer antes de tentar escrever código.

Não tenha medo de errar! Não tente fazer seu código ficar perfeito de início! Errar e experimentar soluções é parte do aprendizado da biblioteca, e você será um melhor desenvolvedor com isso. Exemplos de soluções para estes exercícios estão localizados no `dir /solutions` no código de exemplo.

Convenções usadas neste livro.

Métodos que podem ser chamados em objetos jQuery serão referenciados como `$.fn.methodName`. Métodos que existem no namespace do jQuery mas não podem ser chamados em objetos jQuery serão

referidos como `$.methodName`. Se não fizer muito sentido para você, não se preocupe - ficará mais claro à medida que você avançar no livro.

Exemplo 1.3. Exemplo de um exemplo

```
// exemplos de código serão parecidos com isto
```

Comentários irão aparecer assim.

Nota

Notas sob um tópico irá aparecer assim.

Material de referência

Há muitos artigos e posts de blog por aí que falam de alguma forma de jQuery. Alguns são fenomenais; outros são certamente errados. Quando você ler um artigo sobre jQuery, tenha certeza que está falando sobre a mesma versão que você está usando e resista à tentação de somente copiar e colar - leve um tempo para entender o código no artigo.

Aqui vão alguns excelentes recursos para usar durante seu aprendizado em jQuery. O mais importante de todos é o código fonte do jQuery: ele contém, em forma de código, a documentação completa da biblioteca. Não é uma caixa preta - seu conhecimento sobre a biblioteca crescerá exponencialmente se você gastar um tempo visitando-o agora e novamente - e eu recomendo fortemente que você favorite-o no seu navegador e visite-o com frequência.

- O código fonte do jQuery [<http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js>]
- A documentação do jQuery [<http://api.jquery.com>]
- O fórum do jQuery [<http://forum.jquery.com/>]
- Favoritos no Delicious [<http://delicious.com/rdmey/jquery-class>]
- Canal IRC #jquery na Freenode [http://docs.jquery.com/Discussion#Chat_.2F_IRC_Channel]

Parte I. JavaScript 101

Capítulo 2. O Básico de JavaScript

Introdução

jQuery é construído sobre JavaScript, uma linguagem rica e expressiva por si só. Esta sessão cobre os conceitos básicos de JavaScript, assim como algumas pegadinhas que aparecem com frequência para pessoas que ainda não usaram JavaScript. Por enquanto, para pessoas sem experiência em programação, até mesmo para aquelas que já programam em outras linguagens, podem se beneficiar ao aprender um pouco sobre algumas particularidades de JavaScript.

Se você está interessado em aprender mais sobre a linguagem JavaScript, eu recomento fortemente o livro *JavaScript: The Good Parts* do Douglas Crockford.

O Básico de sintaxe

Entendendo declarações, nomes de variáveis, espaço, e outras sintaxes básicas do JavaScript.

Exemplo 2.1. Uma simples declaração de variável

```
var foo = 'ola mundo';
```

Exemplo 2.2. Espaços em branco não interferem fora das aspas.

```
var foo =      'olá mundo';
```

Exemplo 2.3. Parênteses indicam precedência

```
2 * 3 + 5;    // retorna 11; multiplicação acontece antes
2 * (3 + 5);  // retorna 16; adição acontece primeiro
```

Exemplo 2.4. Tabulação melhora a leitura, mas não tem significado especial

```
var foo = function() {
    console.log('olá');
};
```

Operadores

Operadores Básicos

Operadores básicos permitem que você manipule valores.

Exemplo 2.5. Concatenação

```
var foo = 'olá';
var bar = 'mundo';

console.log(foo + ' ' + bar); // 'olá mundo'
```

Exemplo 2.6. Multiplicação e divisão

```
2 * 3;
2 / 3;
```

Exemplo 2.7. Incrementando e decrementando

```
var i = 1;

var j = ++i; // pré-incremento: j é igual a 2; i é igual a 2
var k = i++; // pós-incremento: k é igual a 2; i é igual a 3
```

Operações sobre números & Strings

Em JavaScript, números e strings ocasionalmente irão se comportar de formas inesperadas.

Exemplo 2.8. Adição vs. Concatenação

```
var foo = 1;
var bar = '2';

console.log(foo + bar); // 12. uh oh
```

Exemplo 2.9. Forçando uma string atuar como um número

```
var foo = 1;
var bar = '2';

// converte string para numero
console.log(foo + Number(bar));
```

O construtor `Number`, quando chamado como função (como no exemplo acima), terá o efeito de mudar seu argumento para um número. Você pode também usar o operador unário de adição, que faz a mesma coisa:

Exemplo 2.10. Forçando uma string a atuar como um número (usando o operador unário de adição)

```
console.log(foo + +bar);
```

Operadores lógicos

Operadores lógicos permitem que você teste uma série de operadores usando as operações lógicas AND e OR.

Exemplo 2.11. Os operadores lógicos AND e OR

```
var foo = 1;
var bar = 0;
var baz = 2;

foo || bar; // retorna 1, que é verdadeiro
bar || foo; // retorna 1, que é verdadeiro

foo && bar; // retorna 0, que é falso
foo && baz; // retorna 2, que é verdadeiro
baz && foo; // retorna 1, que é verdadeiro
```

Apesar de não estar claro pelo exemplo, o operador `||` retorna o valor do primeiro operador verdadeiro, ou, em casos em que nenhum operador seja verdadeiro, ele irá retornar o último dos operandos. O operador `&&` retorna o valor do primeiro operando falso, ou o valor do último operador se ambos operadores forem verdadeiros.

Tenha certeza de consultar “Coisas Verdadeiras e Falsas” para maiores detalhes sobre quais valores são avaliados como `true` e quais são avaliados como `false`.

Nota

Algumas vezes, você verá alguns desenvolvedores usando estes operadores lógicos para controle de fluxo ao invés de utilizar o `if`. Por exemplo:

```
// faz alguma coisa com foo se foo for verdadeiro
foo && doSomething(foo);

// seta bar para baz se baz é verdadeiro;
// senão for verdadeiro, seta-o para retornar
// o valor de createBar()
var bar = baz || createBar();
```

Este estilo é elegante e agradavelmente curto; por isso, pode ser difícil ler, especialmente por iniciantes. Eu o mostro aqui para que você possa reconhecê-lo no código que você lê, mas eu não recomendo o uso até que você esteja extremamente confortável com o que isto significa e como você espera que se comporte.

Operadores de comparação

Operadores de comparação permite testar quando valores são equivalentes ou quando são idênticos.

Exemplo 2.12. Operadores de comparação

```
var foo = 1;
var bar = 0;
var baz = '1';
var bim = 2;

foo == bar;    // retorna false
foo != bar;    // retorna true
foo == baz;    // retorna true; cuidado!

foo === baz;           // retorna false
foo !== baz;           // retorna true
foo === parseInt(baz); // retorna true

foo > bim;    // retorna false
bim > baz;    // retorna true
foo <= baz;   // retorna true
```

Código condicional

Algumas vezes você só irá querer executar um bloco de código sob certas condições. Controle de fluxo - através dos blocos `if` e `else` - permite que você execute uma parte do código em certas condições.

Exemplo 2.13. Controle de fluxo

```
var foo = true;
var bar = false;

if (bar) {
    // este código nunca será executado
    console.log('hello!');
}

if (bar) {
    // este código não irá executar
} else {
    if (foo) {
        // este código irá executar
    } else {
        // este código poderá ser executado se foo e bar forem ambos falsos.
    }
}
```

Nota

As chaves não são estritamente necessárias quando você tem blocos `if` com uma só linha, porém, se forem usados consistentemente, mesmo que não sejam estritamente requeridos, eles podem fazer seu código ficar mais legível.

Esteja atento para não definir funções com o mesmo nome várias vezes dentro de blocos de código `if/else`, senão pode haver resultados inesperados.

Coisas Verdadeiras e Falsas

De forma a usar o controle de fluxo com sucesso, é importante entender quais tipos de valores são "verdadeiros" e quais tipos de valores são "falsos". Algumas vezes, valores que parecem ser avaliados de uma forma, são avaliados de outra.

Exemplo 2.14. Valores que são avaliados como `true`

```
'0';
'qualquer string';
[]; // um array vazio
{}; // um objeto vazio
1; // qualquer número diferente de zero
```

Exemplo 2.15. Valores que são avaliados como `false`

```
0;
''; // uma string vazia
NaN; // Variável "not-a-number" do JavaScript
null;
undefined; // seja cuidadoso -- undefined pode ser redefinido!
```

Associação Condicional de Variáveis Com o Operador Ternário

Algumas vezes você precisará colocar um valor numa variável dependendo de uma condição. Você poderia usar uma declaração `if/else`, mas em muitos casos, o operador ternário é mais conveniente. [Definition: O *operador ternário* testa uma condição; se a condição for verdadeira, ele retorna um valor correto, senão retorna um valor diferente.]

Exemplo 2.16. O operador ternário

```
// seta foo pra 1 se bar for verdadeiro;
// senão, seta foo para 0
var foo = bar ? 1 : 0;
```

Enquanto o operador ternário pode ser usado sem associação do valor de retorno para uma variável, isto é geralmente desencorajado.

Declaração Switch

Ao invés de usar uma série de blocos `if/else if/else`, algumas vezes pode ser útil usar a declaração `switch`. [Definition: *Declarações Switch* olham o valor da variável ou expressão e executa blocos de código diferentes dependendo do valor.]

Exemplo 2.17. Uma declaração switch

```
switch (foo) {

    case 'bar':
        alert('o valor é bar -- yay!');
        break;

    case 'baz':
        alert('boo baz :(');
        break;

    default:
        alert('todo o resto está ok');
        break;

}
```

As declarações `Switch` de alguma forma favorecem o JavaScript, pois frequentemente o mesmo comportamento pode ser obtido através da criação de um objeto que tem um maior potencial para reuso, teste, etc. Por exemplo:

```
var stuffToDo = {
    'bar' : function() {
        alert('o valor é bar -- yay!');
    },

    'baz' : function() {
        alert('boo baz :(');
    }
};
```

```
    },  
  
    'default' : function() {  
        alert('todo o resto está ok');  
    }  
};  
  
if (stuffToDo[foo]) {  
    stuffToDo[foo]();  
} else {  
    stuffToDo['default']();  
}
```

Nós daremos uma olhada mais profunda em objetos mais a frente neste capítulo.

Laços

Laços permitem que você execute um bloco de código por um certo número de vezes.

Exemplo 2.18. Laços

```
// loga 'tentativa 0', 'tentativa 1', ..., 'tentativa 4'  
for (var i=0; i<5; i++) {  
    console.log('tentativa ' + i);  
}
```

Note que em Exemplo 2.18, “Laços” mesmo que usemos a palavra chave var antes do nome da variável i, o escopo da variável i não é alterado para o bloco do laço. Nós iremos discutir sobre escopo com mais profundidade mais a frente neste capítulo.

O laço for

O laço for é constituído de quatro declarações e possui a seguinte estrutura:

```
for ([inicialização]; [condição]; [incremento])  
    [corpoDoLaço]
```

A declaração *inicialização* é executada somente uma vez, antes do loop iniciar. Ela lhe dá uma oportunidade de preparar ou declarar qualquer variável.

A declaração *condição* é executada antes de cada iteração, e o seu valor de retorno decide se o loop continua ou não. Se a declaração condicional for interpretada como um valor falso, o laço para de ser executado.

A declaração *incremento* é executada no fim de cada iteração e lhe dá a oportunidade de mudar o estado de variáveis importantes. Tipicamente, isso irá envolver o incremento ou decremento de um contador e assim fazer com que o loop sempre chegue ao seu fim.

A declaração *corpoDoLaço* é o que é executado em cada iteração. Ele pode conter o que você quiser. Tipicamente, você terá múltiplas declarações que precisam ser executadas e portanto você irá colocá-las dentro de um bloco de código ({ . . . }).

Este é um uso típico para o laço for:

Exemplo 2.19. Um laço `for` típico

```
for (var i = 0, limit = 100; i < limit; i++) {  
  // Este bloco será executado 100 vezes  
  console.log('Estou em ' + i);  
  // Nota: o último log será: "Estou em 99"  
}
```

O laço `while`

O laço `while` é similar a uma declaração `if`, exceto que seu corpo continuará executando até que a condição seja interpretada como falso.

```
while ([condição]) [corpoDoLaço]
```

Este é um laço `while` típico:

Exemplo 2.20. Um laço `while` típico

```
var i = 0;  
while (i < 100) {  
  
  // Este bloco será executado 100 vezes  
  console.log('Estou em ' + i);  
  
  i++; // incrementa i  
}
```

Você irá perceber que nós temos que incrementar o contador dentro do corpo do laço. É possível combinar o incremento e a condição, desse jeito:

Exemplo 2.21. Um laço `while` com uma condição e incremento combinado

```
var i = -1;  
while (++i < 100) {  
  // Este bloco será executado 100 vezes  
  console.log('Estou em ' + i);  
}
```

Perceba que nós estamos iniciando em `-1` e usando o incremento prefixo (`++i`).

O laço `do-while`

É quase a mesma coisa do laço `while`, exceto pelo fato que o corpo do laço é executado pelo menos uma vez antes da condição ser testada.

```
do [corpoDoLaço] while ([condição])
```

Este é um exemplo de código com `do-while`:

Exemplo 2.22. Um laço do-while

```
do {  
  
    // Mesmo que a condição seja interpretada como false  
    // este corpo do laço ainda será executado uma vez  
  
    alert('Olá!');  
  
} while (false);
```

Estes tipos de laços são relativamente raros, pois poucas situações requerem um laço que execute cegamente um bloco de código ao menos uma vez. Independentemente, é bom ficar sabendo disso.

Parando e continuando

Geralmente, um loop termina porque a declaração condicional não é interpretada como verdadeira, mas é possível parar um laço dentro do corpo do mesmo utilizando a declaração `break`.

Exemplo 2.23. Parando um laço

```
for (var i = 0; i < 10; i++) {  
    if (algumaCoisa) {  
        break;  
    }  
}
```

Você também pode desejar continuar o laço sem executar o resto do seu corpo. Isso é feito usando a declaração `continue`.

Exemplo 2.24. Pulando para a próxima iteração de um loop

```
for (var i = 0; i < 10; i++) {  
  
    if (algumaCoisa) {  
        continue;  
    }  
  
    // A declaração seguinte somente será executada se  
    // a condição 'algumaCoisa' for falsa  
    console.log('Cheguei até aqui');  
  
}
```

Palavras Reservadas

JavaScript possui um número de “palavras reservadas”, ou palavras que têm significado especial na linguagem. Você deve evitar usar estas palavras em seu código, exceto quando você as usar com seu significado desejado.

- `break`
- `case`

- catch
- continue
- default
- delete
- do
- else
- finally
- for
- function
- if
- in
- instanceof
- new
- return
- switch
- this
- throw
- try
- typeof
- var
- void
- while
- with
- abstract
- boolean
- byte
- char
- class
- const
- debugger

- double
- enum
- export
- extends
- final
- float
- goto
- implements
- import
- int
- interface
- long
- native
- package
- private
- protected
- public
- short
- static
- super
- synchronized
- throws
- transient
- volatile

Arrays

Arrays são listas de valores indexadas por zero. Eles são uma forma útil de armazenar um conjunto de itens relacionados do mesmo tipo (como strings), apesar que na realidade, um array pode incluir múltiplos tipos de itens, incluindo outros arrays.

Exemplo 2.25. Um array simples

```
var meuArray = [ 'Olá', 'Mundo' ];
```

Exemplo 2.26. Acessando os itens do array pelo índice

```
var meuArray = [ 'ola', 'mundo', 'foo', 'bar' ];
console.log(meuArray[3]); // loga 'bar'
```

Exemplo 2.27. Testando o tamanho de um array

```
var meuArray = [ 'olá', 'mundo' ];
console.log(meuArray.length); // loga 2
```

Exemplo 2.28. Mudando o valor de um item do array

```
var meuArray = [ 'olá', 'mundo' ];
meuArray[1] = 'mudado';
```

Enquanto é possível mudar o valor de um item do array como mostrado no Exemplo 2.28, “Mudando o valor de um item do array”, geralmente não é recomendado.

Exemplo 2.29. Adicionando elementos em um array

```
var meuArray = [ 'olá', 'mundo' ];
meuArray.push('novo');
```

Exemplo 2.30. Trabalhando com arrays

```
var meuArray = [ 'h', 'e', 'l', 'l', 'o' ];
var minhaString = meuArray.join(''); // 'hello'
var meuSplit = minhaString.split(''); // [ 'h', 'e', 'l', 'l', 'o' ]
```

Objetos

Objetos contêm um ou mais pares chave-valor. A porção chave pode ser qualquer string. A porção valor pode ser qualquer tipo de valor: um número, uma string, um array, uma função ou até um outro objeto.

[Definition: Quando um desses valores é uma função, é chamado de *método* do objeto.] Senão, elas são chamadas de propriedades.

Na prática, quase tudo em JavaScript é um objeto — arrays, funções, números e até strings - e todos eles possuem propriedades e métodos.

Exemplo 2.31. Criando um "literal objeto"

```
var myObject = {
  sayHello : function() {
    console.log('olá');
  },
  myName : 'Rebecca'
};

myObject.sayHello(); // loga 'olá'
console.log(myObject.myName); // loga 'Rebecca'
```


Nota

Ao criar literais objeto, você deve notar que a porção chave de cada par chave-valor pode ser escrito como qualquer identificador válido em JavaScript, uma string (entre aspas) ou um número:

```
var myObject = {  
    validIdentifier: 123,  
    'some string': 456,  
    99999: 789  
};
```

Literais objeto podem ser extremamente úteis para organização de código; para mais informações leia *Using Objects to Organize Your Code* [<http://blog.rebeccamurphey.com/2009/10/15/using-objects-to-organize-your-code/>] de Rebecca Murphey.

Funções

Funções contêm blocos de código que precisam ser executados repetidamente. As funções podem ter zero ou mais argumentos, e podem opcionalmente retornar um valor.

Funções podem ser criadas em uma série de formas:

Exemplo 2.32. Declaração de Função

```
function foo() { /* faz alguma coisa */ }
```

Exemplo 2.33. Expressão de Função Nomeada

```
var foo = function() { /* faz alguma coisa */ }
```

Eu prefiro a expressão de função nomeada para definir o nome da função por algumas questões técnicas [<http://yura.thinkweb2.com/named-function-expressions/>]. Você provavelmente verá ambos os métodos usados em código JavaScript de outros.

Usando funções

Exemplo 2.34. Uma função simples

```
var greet = function(person, greeting) {  
    var text = greeting + ', ' + person;  
    console.log(text);  
};
```

```
greet('Rebecca', 'Olá');
```

Exemplo 2.35. Uma função que retorna um valor

```
var greet = function(person, greeting) {  
    var text = greeting + ', ' + person;  
    return text;  
};
```

```
console.log(greet('Rebecca', 'Olá'));
```

Exemplo 2.36. Uma função que retorna outra função

```
var greet = function(person, greeting) {  
    var text = greeting + ', ' + person;  
    return function() { console.log(text); };  
};  
  
var greeting = greet('Rebecca', 'Olá');  
greeting();
```

Funções anônimas autoexecutáveis

Um padrão comum em JavaScript é a função anônima autoexecutável. Este padrão cria uma função e a executa imediatamente. Este padrão é extremamente útil para casos onde você quer evitar poluir o namespace global com seu código -- nenhuma variável declarada dentro da função é visível fora dele.

Exemplo 2.37. Uma função anônima auto-executável

```
(function(){  
    var foo = 'Olá mundo';  
})();  
  
console.log(foo);    // undefined!
```

Funções como argumento

Em JavaScript, funções são "cidadãos de primeira classe" -- eles podem ser atribuídos a variáveis ou passados para outras funções como argumentos. Passar funções como argumento é um idioma extremamente comum em jQuery.

Exemplo 2.38. Passando uma função anônima como argumento

```
var myFn = function(fn) {  
    var result = fn();  
    console.log(result);  
};  
  
myFn(function() { return 'olá mundo'; });    // loga 'olá mundo'
```

Exemplo 2.39. Passando uma função nomeada como argumento

```
var myFn = function(fn) {  
    var result = fn();  
    console.log(result);  
};  
  
var myOtherFn = function() {  
    return 'olá mundo';  
};  
  
myFn(myOtherFn);    // loga 'olá mundo'
```

Testando Tipo

JavaScript oferece um meio de testar o "tipo" de uma variável. Entretanto, o resultado pode ser confuso -- por exemplo, o tipo de um Array é "objeto".

É uma prática comum usar o operador `typeof` ao tentar determinar o tipo de um valor específico.

Exemplo 2.40. Testando o tipo de várias variáveis

```
var myFunction = function() {
    console.log('olá');
};

var myObject = {
    foo : 'bar'
};

var myArray = [ 'a', 'b', 'c' ];

var myString = 'olá';

var myNumber = 3;

typeof myFunction;    // retorna 'function'
typeof myObject;      // retorna 'object'
typeof myArray;       // retorna 'object' -- cuidado!
typeof myString;      // retorna 'string';
typeof myNumber;      // retorna 'number'

typeof null;          // retorna 'object' -- cuidado!

if (myArray.push && myArray.slice && myArray.join) {
    // provavelmente um array
    // (isso é chamado de "duck typing")
}

if (Object.prototype.toString.call(myArray) === '[object Array]') {
    // Definitivamente um array!
    // Isso é largamente considerado como a forma mais robusta para
    // determinar se um valor específico é um Array.
}
```

O jQuery oferece métodos utilitários para ajudar você determinar o tipo de um valor arbitrário. Estes métodos serão mostrados depois.

Escopo

"Escopo" refere-se a variáveis que estão disponíveis em um pedaço do código em um determinado tempo. A falta de entendimento de escopo pode levar a experiências de debugging frustrantes.

Quando uma variável é declarada dentro de uma função usando a palavra chave `var`, ela é a única disponível para o código dentro daquela função -- código fora desta função não podem acessar a variável. Por outro lado, funções definidas *dentro* daquela função *terá acesso* à variável declarada.

Além disso, variáveis que são declaradas dentro de uma função sem o código `var` não são locais para a função -- JavaScript irá atravessar a cadeia do escopo para indicar onde a variável foi definida anteriormente. Se a variável não foi previamente definida, ela será definida no escopo global, onde podem acontecer consequências extremamente inesperadas;

Exemplo 2.41. Funções tem acesso a variáveis definidas num mesmo escopo

```
var foo = 'olá';

var sayHello = function() {
  console.log(foo);
};

sayHello();           // loga 'olá'
console.log(foo);     // também loga 'olá'
```

Exemplo 2.42. Código fora do escopo em que uma variável foi definida não tem acesso à variável.

```
var sayHello = function() {
  var foo = 'olá';
  console.log(foo);
};

sayHello();           // loga 'olá'
console.log(foo);     // não loga nada
```

Exemplo 2.43. Variáveis com o mesmo nome podem existir em escopos diferentes com valores diferentes.

```
var foo = 'mundo';

var sayHello = function() {
  var foo = 'olá';
  console.log(foo);
};

sayHello();           // loga 'olá'
console.log(foo);     // loga 'mundo'
```

Exemplo 2.44. Funções podem "ver" mudanças em valores de variáveis depois que a função é definida

```
var myFunction = function() {  
    var foo = 'olá';  
  
    var myFn = function() {  
        console.log(foo);  
    };  
  
    foo = 'mundo';  
  
    return myFn;  
};  
  
var f = myFunction();  
f(); // loga 'mundo' -- uh oh
```

Exemplo 2.45. Insanidade do Escopo

```
// uma função anônima auto-executável  
(function() {  
    var baz = 1;  
    var bim = function() { alert(baz); };  
    bar = function() { alert(baz); };  
})();  
  
console.log(baz); // baz não é definido fora da função  
  
bar(); // bar é definida fora da função anônima  
      // porque ela não foi declarada com var; além disso,  
      // porque ela foi definida no mesmo escopo de baz,  
      // ela tem acesso a baz mesmo que outro código fora dela  
      // não tenha  
  
bim(); // bim não está definido fora da função anônima,  
      // então isto irá resultar em um erro
```

Closures

Closures são uma extensão do conceito de escopo — funções têm acesso a variáveis que estão disponíveis no escopo onde a função foi criada. Se isso é confuso, não se preocupe: closures são geralmente entendidas melhor através de um exemplo.

No Exemplo 2.44, “Funções podem “ver” mudanças em valores de variáveis depois que a função é definida” nós vimos como as funções têm acesso para modificar valores de variáveis. O mesmo tipo de comportamento existe com funções definidas dentro de laços -- a função “vê” a mudança no valor da variável mesmo depois que a função foi definida, resultando em todos os cliques alertando 5.

Exemplo 2.46. Como bloquear no valor de i?

```
/* isso não se comportará como nós quisermos; */  
/* todo clique irá alertar 5 */  
for (var i=0; i<5; i++) {  
    $('<p>click me</p>').appendTo('body').click(function() {  
        alert(i);  
    });  
}
```

Exemplo 2.47. Bloqueando no valor de i com uma closure

```
/* conserto: "fecha" o valor de i dentro de createFunction, então ela não irá mudar  
var createFunction = function(i) {  
    return function() { alert(i); };  
};  
  
for (var i=0; i<5; i++) {  
    $('p').appendTo('body').click(createFunction(i));  
}
```

Parte II. jQuery: Conceitos básicos

Capítulo 3. O básico de jQuery

`$(document).ready()`

Você não pode manipular a página com segurança até o documento estar "pronto" (ready). O jQuery detecta o estado de prontidão para você; o código incluído dentro de `$(document).ready()` somente irá rodar depois que a página estiver pronta executar o código JavaScript.

Exemplo 3.1. Um bloco `$(document).ready()`

```
$(document).ready(function() {  
    console.log('pronto!');  
});
```

Há um atalho para `$(document).ready()` que você verá algumas vezes; entretando, eu não recomendo usá-lo se você estiver escrevendo código que pessoas que não têm experiência com jQuery poderá ver.

Exemplo 3.2. Atalho para `$(document).ready()`

```
$(function() {  
    console.log('pronto!');  
});
```

Você ainda pode passar uma função nomeada para `$(document).ready()` ao invés de passar uma função anônima.

Exemplo 3.3. Passando uma função nomeada ao invés de uma anônima

```
function readyFn() {  
    // código para executar quando o documento estiver pronto  
}  
  
$(document).ready(readyFn);
```

Selecionando elementos

O conceito mais básico do jQuery é "selecionar alguns elementos e fazer alguma coisa com eles." O jQuery suporta a maioria dos seletores CSS3, assim como alguns seletores não-padrão. Para uma referência completa de seletores, visite <http://api.jquery.com/category/selectors/>.

A seguir, alguns exemplos de técnicas comuns de seleção.

Exemplo 3.4. Selecionando elementos por ID

```
$('#myId'); // lembre-se que os IDs devem ser únicos por página
```

Exemplo 3.5. Selecionando elementos pelo nome da classe

```
$('.div.myClass'); // há um aumento de performance se você especificar o tipo de el
```


Exemplo 3.6. Selecionando elementos por atributo

```
$('input[name=first_name]'); // cuidado, isto pode ser muito lento
```

Exemplo 3.7. Selecionando elementos através da composição de seletores CSS

```
$('#contents ul.people li');
```

Exemplo 3.8. Pseudo-seletores

```
$('a.external:first');  
$('tr:odd');  
$('#myForm :input'); // selecione todos os elementos input num formulário  
$('div:visible');  
$('div:gt(2)'); // todos exceto as três primeiras divs  
$('div:animated'); // todas as divs com animação
```

Nota

Quando você usa os pseudos-seletores `:visible` e `:hidden`, o jQuery testa a visibilidade atual do elemento, não os atributos `visibility` ou `display` do CSS - ou seja, ele olha se a altura e a largura física do elemento na página são ambas maiores que zero. No entanto, este teste não funciona com elementos `<tr>`; neste caso, o jQuery faz a verificação da propriedade `display` do CSS, e considera um elemento como oculto se sua propriedade `display` for `none`. Elementos que não forem adicionados no DOM serão sempre considerados ocultos, mesmo que o CSS que os afetam torná-los visíveis. (Consulte a seção de Manipulação mais adiante neste capítulo para aprender como criar e adicionar elementos ao DOM).

Para referência, aqui está o código que jQuery usa para determinar se um elemento é visível ou oculto, com os comentários adicionados para maior esclarecimento:

```
jQuery.expr.filters.hidden = function( elem ) {  
    var width = elem.offsetWidth, height = elem.offsetHeight,  
        skip = elem.nodeName.toLowerCase() === "tr";  
  
    // o elemento tem 0 de altura e 0 de largura e não é uma <tr>?  
    return width === 0 && height === 0 && !skip ?  
  
        // então deve estar escondido  
        true :  
  
        // mas se o elemento tiver largura e altura e não for uma <tr>  
        width > 0 && height > 0 && !skip ?  
  
            // então deve estar visível  
            false :  
  
            // se chamamos aqui, o elemento tem largura  
            // e altura, mas também é uma <tr>,  
            // então verifica a propriedade display para  
            // decidir se ele está escondido  
            jQuery.curCSS(elem, "display") === "none";  
};
```

```
jQuery.expr.filters.visible = function( elem ) {  
    return !jQuery.expr.filters.hidden( elem );  
};
```

Escolhendo seletores

Escolher bons seletores é uma forma de melhorar a performance do seu JavaScript. Uma pequena especificidade - por exemplo, incluir um elemento como `div` quando selecionar elementos pelo nome da classe - pode ir por um longo caminho. Geralmente, sempre que você puder dar ao jQuery alguma dica sobre onde ele pode esperar encontrar o que você estiver procurando, você deve dar. Por outro lado, muita especificidade pode não ser muito bom. Um seletor como `#myTable thead tr th.special` é um desperdício se um seletor como `#myTable th.special` lhe dará o que você precisa.

O jQuery oferece muitos seletores baseados em atributo, permitindo que você selecione elementos baseados no conteúdo de atributos arbitrários usando expressões regulares simplificadas.

```
// encontre todos elementos <a>s em que o atributo  
// rel termine com "thinger"  
$("a[rel$='thinger']");
```

Se por um lado estes seletores podem ser bem úteis, eles também podem ser extremamente lerdos - Uma vez eu escrevi um seletor baseado em atributos que travou minha página por múltiplos segundos. Sempre que possível, faça suas seleções usando IDs, nomes de classe e nomes de tags.

Quer saber mais? Paul Irish tem uma excelente apresentação sobre melhorar a performance do JavaScript [<http://paulirish.com/perf>], com vários slides focando especificamente em performance de seletores.

Minha seleção contém algum elemento?

Uma vez que você fez uma seleção, você irá querer saber se há algo para trabalhar com ela. Você talvez se sinta tentado a fazer algo assim:

```
if ($('#div.foo')) { ... }
```

Isso não irá funcionar. Quando você faz uma seleção usando `$()`, um objeto é sempre retornado, e objetos sempre são tratados como `true`. Mesmo se sua seleção não tiver nenhum elemento, o código dentro do `if` vai executar do mesmo jeito.

Ao invés disso, você precisa testar a propriedade `length` da seleção, que diz a você quantos elementos foram selecionados. Se a resposta for 0, a propriedade `length` será interpretada como falso quando usada como um valor booleano.

Exemplo 3.9. Testando se uma seleção contém elementos.

```
if ($('#div.foo').length) { ... }
```

Salvando seleções

Toda vez que você faz uma seleção, um monte de código é executado, e o jQuery não faz caching de seleções para você. Se você fez uma seleção que você talvez precise fazer novamente, você deve salvar a seleção numa variável ao invés de fazer a seleção várias vezes.

Exemplo 3.10. Armazenando seleções em variáveis

```
var $divs = $('div');
```

Nota

No Exemplo 3.10, “Armazenando seleções em variáveis”, o nome da variável começa com um sinal de dólar. Ao invés de outras linguagens, não há nada especial sobre o sinal de dólar em JavaScript -- é apenas outro caracter. Nós o usamos para indicar que a variável contém um objeto jQuery. Esta prática -- um tipo de Notação Húngara [http://en.wikipedia.org/wiki/Hungarian_notation] -- é meramente uma convenção, e não é obrigatória.

Uma vez que você armazenou sua seleção, você pode chamar os métodos do jQuery na variável que você armazenou, da mesma forma que você faria na seleção original.

Nota

Uma seleção somente obtém os elementos que estão na página quando você faz a seleção. Se você adicionar elementos na página depois, você terá que repetir a seleção ou então adicioná-la à seleção armazenada na variável. Seleções armazenadas não atualizam automaticamente quando o DOM muda.

Refinando & Filtrando Seleções

Algumas vezes você tem uma seleção que contém mais do que você quer; neste caso, você talvez queira refinar sua seleção. O jQuery oferece vários métodos para você obter exatamente o que precisa.

Exemplo 3.11. Refinando seleções

```
$('div.foo').has('p');           // o elemento div.foo que contém <p>'s
$('h1').not('.bar');             // elementos h1 que não têm a classe bar
$('ul li').filter('.current');  // itens de listas não-ordenadas com a classe curr
$('ul li').first();             // somente o primeiro item da lista não ordenada
$('ul li').eq(5);               // o sexto item da lista
```

Seletores relacionados à formulários

O jQuery oferece vários pseudo-seletores que lhe ajudam a encontrar elementos nos seus formulários; estes são especialmente úteis porque pode ser difícil distinguir entre elementos form baseados no seu estado ou tipo usando seletores CSS padrão.

:button	Seleciona elementos do tipo <button> e elementos com type="button"
:checkbox	Seleciona inputs com type="checkbox"
:checked	Seleciona inputs selecionados
:disabled	Seleciona elementos de formulário desabilitados
:enabled	Seleciona elementos de formulário habilitados
:file	Seleciona inputs com type="file"
:image	Seleciona inputs com type="image"

<code>:input</code>	Seleciona <code><input></code> , <code><textarea></code> , e elementos <code><select></code>
<code>:password</code>	Seleciona inputs com <code>type="password"</code>
<code>:radio</code>	Seleciona inputs com <code>type="radio"</code>
<code>:reset</code>	Seleciona inputs com <code>type="reset"</code>
<code>:selected</code>	Seleciona inputs que estão selecionados
<code>:submit</code>	Seleciona inputs com <code>type="submit"</code>
<code>:text</code>	Seleciona inputs com <code>type="text"</code>

Exemplo 3.12. Usando pseudo-seletores relacionados à formulários

```
$('#myForm :input'); // obtém todos os elementos que aceitam entrada de dados
```

Trabalhando com seleções

Uma vez que você tem uma seleção, você pode chamar métodos nela. Métodos geralmente vêm em duas formas diferentes: getters e setters. Getters retornam uma propriedade do primeiro elemento selecionado; setters ajustam (setam) uma propriedade em todos os elementos selecionados

Encadeamento

Se você chamar um método numa seleção e este método retornar um objeto jQuery, você pode continuar a chamar métodos do jQuery sem precisar pausar com um ponto-e-vírgula.

Exemplo 3.13. Encadeamento

```
$('#content').find('h3').eq(2).html('o novo texto do terceiro h3!');
```

Se você estiver escrevendo uma cadeia que inclui vários passos, você (e a pessoa que virá depois de você) talvez ache seu código mais legível se você quebrar o código em várias linhas.

Exemplo 3.14. Formatando código encadeado

```
$('#content')
    .find('h3')
    .eq(2)
    .html('novo texto do terceiro h3!');
```

Se você mudar sua seleção no meio de uma cadeia, o jQuery provê o método `$.fn.end` para você voltar para sua seleção original.

Exemplo 3.15. Restaurando sua seleção original usando `$.fn.end`

```
$('#content')
    .find('h3')
    .eq(2)
    .html('novo texto para o terceiro h3!')
    .end() // restaura a seleção para todos h3 em #context
    .eq(0)
    .html('novo texto para o primeiro h3!');
```

Nota

Encadeamento é um recurso extraordinariamente poderoso, e muitas bibliotecas adotaram-no desde que o jQuery o tornou popular. Entretanto, deve ser usado com cuidado. Encadeamentos extensos podem deixar o código extremamente difícil de debugar ou modificar. Não há uma regra que diz o quão grande uma cadeia deve ser -- mas saiba que é fácil fazer bagunça

Getters & Setters

O jQuery “sobrecarrega” seus métodos, então o método usado para setar um valor geralmente tem o mesmo nome do método usado para obter um valor. [Definition: Quando um método é usado para setar um valor, ele é chamado de *setter*]. [Definition: Quando um método é usado para pegar (ou ler) um valor, ele é chamado de *getter*]. Os setters afetam todos os elementos na seleção; getters obtêm o valor requisitado somente do primeiro elemento na seleção.

Exemplo 3.16. O método `$.fn.html` usado como setter

```
$('#h1').html('olá mundo');
```

Exemplo 3.17. O método `html` usado como getter

```
$('#h1').html();
```

Os setters retornam um objeto jQuery, permitindo que você continue chamando métodos jQuery na sua seleção; getters retornam o que eles foram pedidos para retornar, o que significa que você não pode continuar chamando métodos jQuery no valor retornado pelo getter.

CSS, Styling, & Dimensões

O jQuery possui uma forma bem prática para pegar e setar propriedades CSS dos elementos.

Nota

Propriedades CSS que normalmente incluem um hífen, precisam ser acessadas no estilo *camel case* em JavaScript. Por exemplo, a propriedade CSS `font-size` é expressada como `fontSize` em JavaScript.

Exemplo 3.18. Pegando propriedades CSS

```
$('#h1').css('fontSize'); // retorna uma string, como "19px"
```

Exemplo 3.19. Setando propriedades CSS

```
$('#h1').css('fontSize', '100px'); // setando uma propriedade individual
$('#h1').css({ 'fontSize' : '100px', 'color' : 'red' }); // setando múltiplas propr
```

Note o estilo do argumento que usamos na segunda linha -- é um objeto que contém múltiplas propriedades. Este é um jeito comum de passar múltiplos argumentos para uma função, e muitos setters do jQuery aceitam objetos para setar múltiplos valores de uma só vez.

Usando classes do CSS para estilos

Como um getter, o método `$.fn.css` é útil; Entretanto, ele geralmente deve ser evitado como um setter em código de produção, pois você não quer informação de apresentação no seu JavaScript. Ao invés disso,

escreva regras CSS para classes que descrevam os vários estados visuais, e então mude a classe no elemento que você quer afetar.

Exemplo 3.20. Trabalhando com classes

```
var $h1 = $('h1');

$h1.addClass('big');
$h1.removeClass('big');
$h1.toggleClass('big');

if ($h1.hasClass('big')) { ... }
```

Classes também podem ser úteis para armazenar informações de estado de um elemento, como indicar se um elemento está selecionado, por exemplo.

Dimensões

O jQuery oferece uma variedade de métodos para obter e modificar informações sobre dimensões e posições de um elemento.

O código do Exemplo 3.21, “Métodos básicos de dimensões” é somente uma introdução muito curta sobre as funcionalidades de dimensões do jQuery; para detalhes completos sobre os métodos de dimensão do jQuery, visite <http://api.jquery.com/category/dimensions/>.

Exemplo 3.21. Métodos básicos de dimensões

```
$('h1').width('50px'); // seta a largura de todos os elementos h1
$('h1').width();      // obtém a largura do primeiro h1

$('h1').height('50px'); // seta a altura de todos os elementos h1
$('h1').height();      // obtém a altura do primeiro h1

$('h1').position();    // retorna um objeto contendo informações
                      // sobre a posição do primeiro h1 relativo
                      // a seu pai
```

Atributos

Atributos de elementos podem conter informações úteis para sua aplicação, então é importante saber como setá-los e obtê-los.

O método `$.fn.attr` atua como getter e setter. Assim como o método `$.fn.css`, `$.fn.attr` atuando como setter, pode aceitar tanto uma chave e um valor ou um objeto contendo um ou mais pares chave/valor.

Exemplo 3.22. Definindo atributos

```
$('a').attr('href', 'todosMeusHrefsSaoOMesmoAgora.html');
$('a').attr({
  'title' : 'todos os títulos são os mesmos também!',
  'href' : 'algoNovo.html'
});
```

Agora, nós quebramos o objeto em múltiplas linhas. Lembre-se, espaços não importam em JavaScript, então você deve se sentir livre para usá-lo do jeito que quiser para fazer seu código ficar mais legível! Depois, você pode usar uma ferramenta de minificação para remover espaços desnecessários para seu código de produção.

Exemplo 3.23. Obtendo atributos

```
$('#a').attr('href'); // retorna o href do primeiro elemento <a> do documento
```

Travessia

Uma vez que você tem uma seleção do jQuery, você pode encontrar outros elementos usando sua seleção como ponto de início.

Para documentação completa dos métodos de travessia do jQuery, visite <http://api.jquery.com/category/traversing/>.

Nota

Seja cuidadoso com travessias de longas distâncias nos seus documentos -- travessias complexas torna imperativo que a estrutura do seu documento permaneça a mesma, uma dificuldade à estabilidade, mesmo se você for o responsável por criar toda a aplicação desde o servidor até o cliente. Travessias de um ou dois passos são legais, mas geralmente você irá querer evitar travessias que levem você de um container para outro.

Exemplo 3.24. Movendo pelo DOM usando métodos de travessia.

```
$('#h1').next('p');  
$('#div:visible').parent();  
$('#input[name=first_name]').closest('form');  
$('#myList').children();  
$('#li.selected').siblings();
```

Você também pode iterar sobre uma seleção usando `$.fn.each`. Este método itera sobre todos os elementos numa seleção e executa uma função para cada um. A função recebe como argumento um índice com o elemento atual e com o próprio elemento DOM. Dentro da função, o elemento DOM também está disponível como `this` por padrão.

Exemplo 3.25. Iterando sobre uma seleção

```
$('#myList li').each(function(idx, el) {  
    console.log(  
        'O elemento ' + idx +  
        'tem o seguinte html: ' +  
        $(el).html()  
    );  
});
```

Manipulando elementos

Uma vez que você fez uma seleção, a diversão começa. Você pode mudar, mover, remover e clonar elementos. Você ainda pode criar novos elementos através de uma sintaxe simples

Para uma referência completa dos métodos de manipulação do jQuery, visite <http://api.jquery.com/category/manipulation/>.

Obtendo e Definindo informações sobre elementos

Há um certo número de formas que você pode mudar um elemento existente. Dentre as tarefas mais comuns que você irá fazer é mudar o HTML interno ou o atributo de um elemento. O jQuery oferece métodos simples e cross-navegador para estes tipos de manipulação. Você ainda pode obter informações sobre elementos usando muitos dos mesmos métodos nas suas formas de getter. Nós veremos exemplos destes métodos durante esta seção, mas especificamente, aqui vão alguns poucos métodos que você pode usar para obter e definir informação sobre elementos.

Nota

Mudar coisas em elementos é trivial, mas lembre-se que a mudança irá afetar *todos* os elementos na seleção, então se você quiser mudar um elemento, esteja certo de especificá-lo em sua seleção antes de chamar o método setter

Nota

Quando os métodos atuam como getters, eles geralmente só trabalham no primeiro elemento da seleção e eles não retornam um objeto jQuery, portanto você não pode encadear métodos adicionais a eles. Uma exceção notável é `$.fn.text`; como mencionado acima, ele obtém o texto para todos os elementos da seleção

<code>\$.fn.html</code>	Obtém ou seta o conteúdo html.
<code>\$.fn.text</code>	Obtém ou seta os conteúdos de texto; HTML será removido.
<code>\$.fn.attr</code>	Obtém ou seta o valor do atributo fornecido.
<code>\$.fn.width</code>	Obtém ou seta a largura em pixels do primeiro elemento na seleção como um inteiro.
<code>\$.fn.height</code>	Obtém ou seta a altura em pixels do primeiro elemento na seleção.
<code>\$.fn.position</code>	Obtém um objeto com a informação de posição do primeiro elemento na seleção, relativo a seu primeiro ancestral (pai). <i>Este é somente um getter.</i>
<code>\$.fn.val</code>	Obtém ou seta o valor de elementos de formulários.

Exemplo 3.26. Mudando o HTML de um elemento.

```
$('#myDiv p:first')  
  .html('Primeiro parágrafo <strong>novo</strong>!');
```

Movendo, copiando e removendo elementos.

Há uma variedade de formas de mover elementos pelo DOM; geralmente, há duas abordagens:

- Coloque o(s) elemento(s) selecionado(s) relativo à outro elemento
- Coloque um elemento relativo ao(s) elemento(s) selecionado(s)

Por exemplo, o jQuery fornece `$.fn.insertAfter` e `$.fn.after`. O método `$.fn.insertAfter` coloca o(s) elemento(s) selecionado(s) depois do elemento que você passou como argumento; o método `$.fn.after` coloca o elemento passado como argumento depois do elemento

selecionado. Vários outros métodos seguem este padrão: `$.fn.insertBefore` e `$.fn.before`; `$.fn.appendTo` e `$.fn.append`; e `$.fn.prependTo` e `$.fn.prepend`.

O método que faz mais sentido para você dependerá de quais elementos você já selecionou e quais você precisará armazenar uma referência para os elementos que você está adicionando na página. Se você precisar armazenar uma referência, você sempre irá querer fazer pela primeira forma -- colocando os elementos selecionados relativos à outro elemento -- de forma que ele retorne o(s) elemento(s) que você está colocando. Neste caso, os métodos `$.fn.insertAfter`, `$.fn.insertBefore`, `$.fn.appendTo`, e `$.fn.prependTo` serão suas ferramentas para escolha.

Exemplo 3.27. Movendo elementos usando outras formas

```
// faz o primeiro item da lista se tornar o último
var $li = $('#myList li:first').appendTo('#myList');

// outra forma de resolver o mesmo problema
$('#myList').append($('#myList li:first'));

// perceba que não tem como acessar o item
// da lista que movemos, pois ele retorna
// a própria lista
```

Clonando elementos

Quando você usa métodos como `$.fn.appendTo`, você está movendo o elemento; porém, algumas vezes você irá querer fazer uma cópia do elemento. Neste caso, você precisará usar `$.fn.clone` primeiro.

Exemplo 3.28. Fazendo uma cópia de um elemento

```
// copia o primeiro item da lista para o fim
$('#myList li:first').clone().appendTo('#myList');
```

Nota

Se você precisa copiar dados e eventos relacionados, esteja certo de passar `true` como um argumento para `$.fn.clone`.

Removendo elementos

Há duas formas de remover elementos da página: `$.fn.remove` e `$.fn.detach`. Você irá usar `$.fn.remove` quando você quiser remover a seleção permanentemente da página; enquanto o método retorna os elementos removidos, estes elementos não terão seus eventos e dados associados a ele se você retorná-los à página.

Se você precisa que os dados e eventos persistam, você irá usar `$.fn.detach`. Da mesma forma que `$.fn.remove`, ele retorna uma seleção, mas também mantém os dados e os eventos associados com a seleção para que você possa restaurar a seleção para a página no futuro.

Nota

O método `$.fn.detach` é extremamente útil se você estiver fazendo uma manipulação pesada à um elemento. Neste caso, é bom aplicar um `$.fn.detach` no elemento da página, trabalhar no seu próprio código, e então restaurá-lo à página quando você terminar. Isto evita que você faça "toques ao DOM" caros enquanto mantém os dados e eventos do elemento.

Se você quer deixar um elemento na página mas simplesmente quer remover seu conteúdo, você pode usar `$.fn.empty` para retirar o HTML interno do elemento.

Criando novos elementos

O jQuery oferece uma forma elegante e trivial para criar novos elementos usando o mesmo método `$()` que você usava para seleções.

Exemplo 3.29. Criando novos elementos

```
$( '<p>Este é um novo parágrafo</p>' );  
$( '<li class="new">novo item de lista</li>' );
```

Exemplo 3.30. Criando um novo elemento com um objeto atributo

```
$( '<a/>', {  
    html : 'Este é um link <strong>new</strong>',  
    'class' : 'new',  
    href : 'foo.html'  
});
```

Perceba que no objeto de atributos nós incluímos como segundo argumento a propriedade `class` entre aspas, enquanto as propriedades `html` e `href` não. Geralmente, nomes de propriedades não precisam estar entre aspas a não ser que elas sejam palavras reservadas (como a `class` neste caso)

Quando você cria um novo elemento, ele não é adicionado imediatamente à página. Há várias formas de adicionar um elemento à página uma vez que ele esteja criado.

Exemplo 3.31. Inserindo um novo elemento na página

```
var $myNewElement = $( '<p>Novo elemento</p>' );  
$myNewElement.appendTo( '#content' );  
  
$myNewElement.insertAfter( 'ul:last' ); // isto irá remover p de #content!  
$( 'ul' ).last().after( $myNewElement.clone() ); // clona o p, portanto temos 2 agora
```

Estritamente falando, você não precisa armazenar o elemento criado numa variável -- você pode simplesmente chamar o método para adicioná-lo diretamente depois do `$()`. Entretanto, a maior parte do tempo você precisará de uma referência ao elemento que você adicionou para que você não o selecione depois.

Você ainda pode criar um elemento ao mesmo tempo que você o adiciona à página, mas note que neste caso você não obtém a referência do novo objeto criado.

Exemplo 3.32. Criando e adicionando um elemento à página ao mesmo tempo

```
$( 'ul' ).append( '<li>item de lista</li>' );
```

Nota

A sintaxe para adicionar novos elementos à página é tão fácil que é tentador esquecer que há um enorme custo de performance por adicionar ao DOM repetidas vezes. Se você está adicionando muitos elementos ao mesmo container, você irá concatenar todo html numa única string e então adicionar a string ao container ao invés de ir adicionando um elemento de cada vez. Você pode

usar um array para colocar todas os pedaços juntos e então aplicar um `join` nele em uma única string para adicionar ao container.

```
var myItems = [], $myList = $('#myList');

for (var i=0; i<100; i++) {
    myItems.push('<li>item ' + i + '</li>');
}

$myList.append(myItems.join(''));
```

Manipulando atributos

Os recursos de manipulação de atributos do jQuery são muitos. Mudanças básicas são simples, mas o método `$.fn.attr` também permite manipulações mais complexas.

Exemplo 3.33. Manipulando um único atributo

```
$('#myDiv a:first').attr('href', 'novoDestino.html');
```

Exemplo 3.34. Manipulando múltiplos atributos

```
$('#myDiv a:first').attr({
    href : 'novoDestino.html',
    rel  : 'super-special'
});
```

Exemplo 3.35. Usando uma função para determinar um novo valor de atributo

```
$('#myDiv a:first').attr({
    rel : 'super-special',
    href : function() {
        return '/new/' + $(this).attr('href');
    }
});

$('#myDiv a:first').attr('href', function() {
    return '/new/' + $(this).attr('href');
});
```

Exercícios

Selecionando

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/sandbox.js` ou trabalhe no Firebug para fazer o seguinte:

1. Selecione todos os elementos DIV que têm a classe "module".
2. Escreva três seletores que você pode usar para obter o terceiro item na lista não-ordenada `#myList`. Qual é o melhor para se usar? Por quê?
3. Selecione o label para o input de busca usando um seletor de atributo.

4. Encontre quantos elementos na página estão escondidos (dica: `.length`).
5. Encontre quantos elementos na página têm um atributo `alt`.
6. Selecione todas as linhas ímpares no corpo de uma tabela.

Atravessamento

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/sandbox.js` ou trabalhe no Firebug para fazer o seguinte:

1. Selecione todos os elementos de imagem na página; logue o atributo `alt` de cada imagem.
2. Selecione a caixa de texto de busca, então atravesse para o form e adicione uma classe nele.
3. Selecione o item de lista dentro de `#myList` que possui uma classe `"current"` e remova esta classe dele; adicione uma classe `"current"` no próximo item de lista.
4. Selecione o elemento `select` dentro de `#specials`; atravesse para o botão de submit.
5. Selecione o primeiro item de lista no elemento `#slideshow`; adicione a classe `"current"` a ele e então adicione a classe `"disabled"` para seus elementos sibling.

Manipulando

Abra o arquivo `/exercises/index.html` no seu navegador. Utilize o arquivo `/exercises/js/sandbox.js` ou use o Firebug para realizar o seguinte:

1. Adicione cinco novos itens de lista no fim da lista não ordenada `#myList`. Dica:

```
for (var i = 0; i<5; i++) { ... }
```

2. Remova os itens ímpares
3. Adicione outro `h2` e outro parágrafo à última `div.module`
4. Adicione outra opção para o elemento `select`; dê ao option o valor `"Quarta-Feira"`.
5. Adicione uma nova `div.module` à página depois da última; coloque uma cópia de uma das imagens existentes dentro dela.

Capítulo 4. Núcleo do jQuery

\$ vs \$()

Até agora, estivemos lidando inteiramente com métodos que são chamados em um objeto jQuery. Por exemplo:

```
$( 'h1' ).remove( );
```

A maioria dos métodos jQuery é chamada em objetos jQuery como mostrado acima; esses métodos são considerados parte do namespace `$.fn`, ou o “protótipo do jQuery”, e são melhor imaginados como métodos do objeto jQuery.

No entanto, há vários métodos que não atuam em uma seleção; esses métodos são considerados parte do namespace jQuery, e são melhor imaginados como métodos do núcleo do jQuery.

Esta distinção pode ser incrivelmente confusa para os novos usuários do jQuery. Aqui está o que você precisa se lembrar:

- Métodos chamados em seleções do jQuery estão no namespace `$.fn`, e automaticamente recebem e retornam a seleção como está.
- Métodos no namespace `$` geralmente são métodos do tipo utilitário, e não funcionam em seleções; não são passados quaisquer argumentos automaticamente para eles, e seu valor de retorno variará.

Há alguns casos onde métodos de objeto e métodos do núcleo tem os mesmos nomes, tal como `$.each` e `$.fn.each`. Nesses casos, seja extremamente cuidadoso(a) ao ler a documentação para que você explore o método correto.

Métodos Utilitários

jQuery oferece diversos métodos utilitários no namespace `$`. Estes métodos são úteis para realizar tarefas rotineiras de programação. Abaixo estão exemplos de alguns dos métodos utilitários; para uma referência completa sobre métodos utilitários do jQuery, visite <http://api.jquery.com/category/utilities/>.

`$.trim` Remove espaços em branco à esquerda e à direita.

```
$.trim('    muitos espaços em branco extras    ');  
// returns 'muitos espaços em branco extras'
```

`$.each` Itera sobre arrays e objetos.

```
$.each([ 'foo', 'bar', 'baz' ], function(idx, val) {  
    console.log('elemento ' + idx + ' é ' + val);  
});  
  
$.each({ foo : 'bar', baz : 'bim' }, function(k, v) {  
    console.log(k + ' : ' + v);  
});
```

Nota

Há também um método `$.fn.each`, que é usado para iterar sobre uma seleção de elementos.

<code>\$.isArray</code>	<p>Retorna o índice de um valor em um array, ou -1 se o valor não estiver no array.</p> <pre>var meuArray = [1, 2, 3, 5]; if (\$.isArray(4, meuArray) !== -1) { console.log('encontrei!'); }</pre>
<code>\$.extend</code>	<p>Muda as propriedades do primeiro objeto usando as propriedades dos objetos subsequentes.</p> <pre>var primeiroObjeto = { foo : 'bar', a : 'b' }; var segundoObjeto = { foo : 'baz' }; var novoObjeto = \$.extend(primeiroObjeto, segundoObjeto); console.log(primeiroObjeto.foo); // 'bar' console.log(novoObjeto.foo); // 'baz'</pre> <p>Se você não quer mudar nenhum dos objetos que passa para o <code>\$.extend</code>, passe um objeto vazio como primeiro argumento.</p> <pre>var primeiroObjeto = { foo : 'bar', a : 'b' }; var segundoObjeto = { foo : 'baz' }; var novoObjeto = \$.extend({}, primeiroObjeto, segundoObjeto); console.log(primeiroObjeto.foo); // 'bar' console.log(novoObjeto.foo); // 'baz'</pre>
<code>\$.proxy</code>	<p>Retorna uma função que sempre será executada no escopo fornecido — isto é, seta o significado do <code>this</code> dentro da função passada ao segundo argumento.</p> <pre>var minhaFuncao = function() { console.log(this); }; var meuObjeto = { foo : 'bar' }; minhaFuncao(); // loga o objeto window var minhaFuncaoComProxy = \$.proxy(minhaFuncao, meuObjeto); minhaFuncaoComProxy(); // loga o objeto meuObjeto</pre> <p>Se você tiver um objeto com métodos, pode passar o objeto e o nome do método para retornar uma função que sempre será executada no escopo do objeto.</p> <pre>var meuObjeto = { minhaFn : function() { console.log(this); } }; \$('#foo').click(meuObjeto.minhaFn); // loga o elemento DOM #foo \$('#foo').click(\$.proxy(meuObjeto, 'minhaFn')); // loga meuObjeto</pre>

Verificando tipos

Como mencionado na seção "O Básico do jQuery", jQuery oferece alguns métodos utilitários para determinar o tipo de um valor específico.

Exemplo 4.1. Verificando o tipo de um valor arbitrário

```
var meuValor = [1, 2, 3];

// Usando o operador typeof do JavaScript para testar tipos primitivos
typeof meuValor == 'string'; // false
typeof meuValor == 'number'; // false
typeof meuValor == 'undefined'; // false
typeof meuValor == 'boolean'; // false

// Usando o operador de igualdade estrita para verificar null
meuValor === null; // false

// Usando os métodos do jQuery para verificar tipos não primitivos
jQuery.isFunction(meuValor); // false
jQuery.isPlainObject(meuValor); // false
jQuery.isArray(meuValor); // true
```

Métodos de Dados

Assim que seu trabalho com o jQuery avança, você descobrirá com frequência que há dados sobre um elemento que você quer armazenar com o elemento. Em JavaScript puro, você pode fazer isso adicionando uma propriedade ao elemento do DOM, mas você terá que lidar com memory leaks em alguns navegadores. jQuery oferece um jeito simples de armazenar dados relacionados a um elemento, e gerencia os problemas de memória para você.

Exemplo 4.2. Armazenando e recuperando dados relacionados a um elemento

```
$('#meuDiv').data('nomeChave', { foo : 'bar' });
$('#meuDiv').data('nomeChave'); // { foo : 'bar' }
```

Você pode armazenar qualquer tipo de dados em um elemento, e é difícil expressar a importância disto quando você está desenvolvendo uma aplicação complexa. Para os fins deste curso, usaremos `$.fn.data` na maioria das vezes para armazenar referências a outros elementos.

Por exemplo, podemos querer estabelecer um relacionamento entre um item de lista e um div que está dentro dele. Poderíamos estabelecer este relacionamento cada vez que interagimos com o item de lista, mas uma solução melhor seria estabelecer o relacionamento uma vez, e então armazenar um ponteiro para o div no item de lista usando `$.fn.data`:

Exemplo 4.3. Armazenando um relacionamento entre elementos usando `$.fn.data`

```
$('#minhaLista li').each(function() {
    var $li = $(this), $div = $li.find('div.content');
    $li.data('contentDiv', $div);
});

// depois não temos que procurar o div de novo;
// podemos apenas lê-lo dos dados do item de lista
var $primeiroLi = $('#minhaLista li:first');
$primeiroLi.data('contentDiv').html('novo conteúdo');
```

Além de passar um único par chave-valor para `$.fn.data` para armazenar dados, você pode passar um objeto contendo um ou mais pares.

Detecção de Navegador & Funcionalidades

Embora o jQuery elimine a maioria das peculiaridades dos navegadores, há ainda ocasiões quando seu código precisa saber sobre o ambiente do navegador.

jQuery oferece o objeto `$.support`, assim como o objeto obsoleto `$.browser`, para este propósito. Para uma documentação completa sobre esses objetos, visite <http://api.jquery.com/jquery.support/> e <http://api.jquery.com/jquery.browser/>.

O objeto `$.support` é dedicado a determinar quais funcionalidades um navegador suporta; é recomendado como um método mais “à prova de futuro” de customizar seu JavaScript para diferentes ambientes de navegador.

O objeto `$.browser` foi substituído em favor do objeto `$.support`, mas não será removido do jQuery tão cedo. Ele fornece detecção direta de versão e marca do navegador.

Evitando Conflitos com Outras Bibliotecas

Se você está usando outra biblioteca JavaScript que usa a variável `$`, você pode cair em conflitos com o jQuery. Para evitar esses conflitos, você precisa colocar o jQuery em modo no-conflict (sem conflito) imediatamente depois que ele é carregado na página e antes que você tente usar o jQuery em sua página.

Quando você coloca o jQuery em modo no-conflict, tem a opção de atribuir um nome de variável para substituir a `$`.

Exemplo 4.4. Colocando jQuery em modo no-conflict

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>var $j = jQuery.noConflict();</script>
```

Você pode continuar a usar o `$` padrão envolvendo seu código em uma função anônima auto-executada; este é o modelo padrão para criação de plugin, onde o autor não saberá se uma outra biblioteca estará usando o `$`.

Exemplo 4.5. Usando o `$` dentro de uma função anônima auto-executada

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
jQuery.noConflict();

(function($) {
    // seu código aqui, usando o $
})(jQuery);
</script>
```

Capítulo 5. Eventos

Introdução

jQuery fornece métodos simples para anexar manipuladores de eventos (event handlers) em seleções. Quando um evento ocorre, a função fornecida é executada. Dentro da função, `this` refere-se ao elemento que foi clicado.

Para detalhes sobre eventos do jQuery, visite <http://api.jquery.com/category/events/>.

A função manipuladora de evento pode receber um objeto de evento. Este objeto pode ser usado para determinar a natureza do evento, e para impedir o comportamento padrão do evento.

Para detalhes sobre o objeto de evento, visite <http://api.jquery.com/category/events/event-object/>.

Conectando Eventos a Elementos

jQuery oferece métodos de conveniência para a maioria dos eventos comuns, e estes são os métodos que você verá sendo usados com mais frequência. Estes métodos -- incluindo `$.fn.click`, `$.fn.focus`, `$.fn.blur`, `$.fn.change`, etc. -- são atalhos para o método `$.fn.bind` do jQuery. O método `bind` é útil para vincular a mesma função a múltiplos eventos, e é também usado quando você quer fornecer dados para o manipulador de eventos, ou quando você está trabalhando com eventos personalizados.

Exemplo 5.1. Vinculando Eventos Usando um Método de Conveniência

```
$('#p').click(function() {  
    console.log('clique');  
});
```

Exemplo 5.2. Vinculando eventos usando o método `$.fn.bind`

```
$('#p').bind('click', function() {  
    console.log('clique');  
});
```

Exemplo 5.3. Vinculando eventos usando o método `$.fn.bind` com dados

```
$('#input').bind(  
    'click change', // anexa múltiplos eventos  
    { foo : 'bar' }, // passa dados  
  
    function(eventObject) {  
        console.log(eventObject.type, eventObject.data);  
        // loga o tipo de evento, e então { foo : 'bar' }  
    }  
);
```

Conectando Eventos para Executar Apenas uma vez

Algumas vezes você precisa que um handler específico execute apenas uma vez -- depois disso, você pode querer que nenhum handler execute, ou pode querer que um handler diferente execute. jQuery fornece o método `$.fn.one` para este propósito.

Exemplo 5.4. Trocando manipuladores usando o método `$.fn.one`

```
$('#p').one('click', function() {  
    $(this).click(function() { console.log('Você clicou nisto antes!'); });  
});
```

O método `$.fn.one` é especialmente útil se você precisa fazer alguma configuração complicada na primeira vez que um elemento é clicado, mas não nas vezes subsequentes.

Desconectando Eventos

Para desconectar um manipulador de evento, você usa o método `$.fn.unbind` e passa o tipo de evento para desatar. Se você anexou uma função nomeada para o evento, então pode isolar a desconexão para essa função nomeada passando-a como o segundo argumento.

Exemplo 5.5. Desconectando todos os manipuladores de clique em uma seleção

```
$('#p').unbind('click');
```

Exemplo 5.6. Desconectando um manipulador de clique específico

```
var foo = function() { console.log('foo'); };  
var bar = function() { console.log('bar'); };  
  
$('#p').bind('click', foo).bind('click', bar);  
$('#p').unbind('click', bar); // foo ainda está ligado ao evento de clique
```

Usando Namespaces com Eventos

Para aplicações complexas e para plugins que você compartilha com outros, pode ser útil usar um namespace para seus eventos de forma que você não desconecte sem querer eventos sobre os quais você não conhece ou não poderia conhecer.

Exemplo 5.7. Usando Namespaces com eventos

```
$('#p').bind('click.meuNamespace', function() { /* ... */ });  
$('#p').unbind('click.meuNamespace');  
$('#p').unbind('.meuNamespace'); // desconecta todos os eventos no namespace
```

Por Dentro da Função Manipuladora de Evento

Como mencionado na introdução, a função manipuladora de evento recebe um objeto de evento, que contém muitas propriedades e métodos. O objeto de evento é mais comumente usado para impedir a ação padrão do evento através do método `preventDefault`. Contudo, o objeto de evento contém vários outros métodos e propriedades úteis, incluindo:

<code>pageX, pageY</code>	A posição do mouse no momento em que o evento ocorreu, relativa ao canto superior esquerdo da página.
<code>type</code>	O tipo do evento (por ex. "click").
<code>which</code>	O botão ou tecla que foi pressionado(a).
<code>data</code>	Quaisquer dados que foram passados quando o evento foi anexado.

target	O elemento do DOM que iniciou o evento.
preventDefault()	Impede a ação padrão do evento (por ex. seguir um link).
stopPropagation()	Impede o evento de se propagar (bubble up) para os outros elementos.

Além do objeto de evento, a função manipuladora de evento também tem acesso ao elemento do DOM no qual o evento foi anexado através da palavra-chave `this`. Para transformar um elemento do DOM em um objeto jQuery no qual podemos usar métodos do jQuery, simplesmente fazemos `$(this)`, frequentemente seguindo este idioma:

```
var $this = $(this);
```

Exemplo 5.8. Impedindo um link de ser seguido

```
$('#a').click(function(e) {  
    var $this = $(this);  
    if ($this.attr('href').match('mau')) {  
        e.preventDefault();  
        $this.addClass('mau');  
    }  
});
```

Disparando Eventos

jQuery fornece um método de disparar os eventos ligados a um elemento sem qualquer interação do usuário através do método `$.fn.trigger`. Enquanto este método tem seus usos, ele não deve ser usado simplesmente para chamar uma função que foi ligada como um tratador de clique. Ao invés disso, você deve armazenar a função que você quer chamar em uma variável, e passar o nome da variável quando fizer sua ligação. Então você pode chamar a própria função sempre que quiser, sem a necessidade do `$.fn.trigger`.

Exemplo 5.9. Disparando um evento do jeito certo

```
var foo = function(e) {  
    if (e) {  
        console.log(e);  
    } else {  
        console.log('isto não vem de um evento!');  
    }  
};  
  
$('#p').click(foo);  
  
foo(); // ao invés de $('#p').trigger('click')
```

Aumentando a Performance com Delegação de Evento

Você frequentemente usará o jQuery para adicionar novos elementos a página, e quando o faz, pode precisar anexar eventos a esses novos elementos -- eventos que você já anexou a elementos similares que

estavam na página originalmente. Ao invés de repetir a ligação de evento toda vez que adicionar elementos à página, você pode usar delegação de evento. Com delegação de evento você anexa seu evento ao elemento container, e então quando o evento ocorre, você verifica em qual elemento contido ele ocorreu. Se isto soa complicado, por sorte o jQuery facilita com seus métodos `$.fn.live` e `$.fn.delegate`.

Enquanto a maioria das pessoas descobre a delegação de evento ao lidar com elementos adicionados na página depois, isso tem alguns benefícios de performance mesmo se você nunca adiciona mais elementos para a página. O tempo requerido para ligar eventos a centenas de elementos individuais é não-trivial; se você tem um grande conjunto de elementos, deve considerar a delegação de eventos relacionados a um elemento container.

Nota

O método `$.fn.live` foi introduzido no jQuery 1.3, e na época apenas certos tipos de eventos eram suportados. Com o jQuery 1.4.2, o método `$.fn.delegate` está disponível, e é o método preferencial.

Exemplo 5.10. Delegação de Evento usando `$.fn.delegate`

```
$('#minhaListaNaoOrdenada').delegate('li', 'click', function(e) {  
    var $meuItemDeLista = $(this);  
    // ...  
});
```

Exemplo 5.11. Delegação de Evento usando `$.fn.live`

```
$('#minhaListaNaoOrdenada li').live('click', function(e) {  
    var $meuItemDeLista = $(this);  
    // ...  
});
```

Desvinculando Eventos Delegados

Se você precisa remover eventos delegados, não pode simplesmente usar `unbind` neles. Ao invés disso, use `$.fn.undelegate` para eventos conectados com `$.fn.delegate`, e `$.fn.die` para eventos conectados com `$.fn.live`. Assim como com `bind`, você pode opcionalmente passar o nome da função ligada ao evento.

Exemplo 5.12. Desvinculando eventos delegados

```
$('#minhaListaNaoOrdenada').undelegate('li', 'click');  
$('#minhaListaNaoOrdenada li').die('click');
```

Auxiliares de Eventos

jQuery oferece duas funções auxiliares relacionadas a eventos que economizam algumas teclas digitadas.

`$.fn.hover`

O método `$.fn.hover` deixa você passar uma ou duas funções para serem executadas quando os eventos `mouseenter` e `mouseleave` ocorrem em um elemento. Se você passar uma função, ela será executada para ambos os eventos; se passar duas funções, a primeira será executada para `mouseenter`, e a segunda será executada para `mouseleave`.

Nota

Antes do jQuery 1.4, o método `$.fn.hover` exigia duas funções.

Exemplo 5.13. A função auxiliar `hover`

```
$('#menu li').hover(function() {  
    $(this).toggleClass('hover');  
});
```

`$.fn.toggle`

Assim como `$.fn.hover`, o método `$.fn.toggle` recebe duas ou mais funções; cada vez que o evento ocorre, a próxima função na lista é chamada. Geralmente, `$.fn.toggle` é usado apenas com duas funções, mas tecnicamente você pode usar tantas quantas desejar.

Exemplo 5.14. A função auxiliar `toggle`

```
$('#p.expander').toggle(  
    function() {  
        $(this).prev().addClass('open');  
    },  
    function() {  
        $(this).prev().removeClass('open');  
    }  
);
```

Exercícios

Criando uma dica de Input

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/inputHint.js` ou trabalhe no Firebug. Sua tarefa é usar o texto do label para o input da busca para criar um texto de "hint" para o input da busca. Os passos são os seguintes:

1. Sete o valor do input de busca para o texto do elemento label
2. Adicione uma classe "hint" ao input de busca
3. Remova o elemento label
4. Faça um Bind do evento focus no input de busca que remove o texto de hint e a classe "hint".
5. Faça um bind do evento blur para o input de busca que restaura o texto de hint e a classe "hint" se nenhum texto for informado.

Quais outras considerações poderiam haver se você estivesse criando esta funcionalidade para um site real?

Adicionando navegação por abas

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/tabs.js`. Sua tarefa é criar uma navegação por abas para os dois elementos `div.module`. Para fazer isto:

1. Esconda todos os modules.
2. Crie uma lista não-ordenada antes do primeiro módulo.
3. Itere sobre os módulos usando `$. fn . each`. Para cada módulo, use o texto para o elemento h2 como o texto para um item da lista.
4. Crie um bind do evento click no item de lista que:
 - Exiba o módulo relacionado e esconda qualquer outro módulo.
 - Adicione uma classe "current" para o item de lista clicado.
 - Remova a classe "current" do outro item de lista.
5. Por último, mostre a primeira tabela.

Capítulo 6. Efeitos

Visão Geral

jQuery torna trivial adicionar efeitos simples a sua página. Efeitos podem usar configurações embutidas, ou fornecer uma duração customizada. Você pode também criar animações customizadas de propriedades CSS arbitrárias.

Para detalhes completos sobre efeitos do jQuery, visite <http://api.jquery.com/category/effects/>.

Efeitos Embutidos

Efeitos frequentemente usados estão embutidos no jQuery como métodos:

<code>\$.fn.show</code>	Mostra o elemento selecionado.
<code>\$.fn.hide</code>	Esconde o elemento selecionado.
<code>\$.fn.fadeIn</code>	Anima a opacidade dos elementos selecionados para 100%.
<code>\$.fn.fadeOut</code>	Anima a opacidade dos elementos selecionados para 0%.
<code>\$.fn.slideDown</code>	Mostra os elementos selecionados com um deslizamento vertical.
<code>\$.fn.slideUp</code>	Esconde os elementos selecionados com um deslizamento vertical.
<code>\$.fn.slideToggle</code>	Mostra ou esconde os elementos selecionados com um deslizamento vertical, dependendo se os elementos atualmente estão visíveis.

Exemplo 6.1. Um uso básico de um efeito embutido

```
$('h1').show();
```

Mudando a Duração de Efeitos Embutidos

Com exceção de `$.fn.show` e `$.fn.hide`, todos os métodos embutidos são animados ao longo de 400ms por padrão. Mudar a duração de um efeito é simples.

Exemplo 6.2. Configurando a duração de um efeito

```
$('h1').fadeIn(300); // fade in durante 300ms
$('h1').fadeOut('slow'); // usando uma definição de duração nativa
```

jQuery.fx.speeds

jQuery tem um objeto em `jQuery.fx.speeds` que contém a velocidade padrão, assim como as configurações para "slow" e "fast".

```
velocidades: {
```

```
    slow: 600,  
    fast: 200,  
    // Velocidade padrão  
    _default: 400  
  }
```

É possível sobrescrever ou adicionar a este objeto. Por exemplo, você pode querer mudar a duração padrão dos efeitos, ou pode querer criar suas próprias velocidades de efeitos.

Exemplo 6.3. Acrescentando definições de velocidade customizadas ao `jQuery.fx.speeds`

```
jQuery.fx.speeds.blazing = 100;  
jQuery.fx.speeds.turtle = 2000;
```

Fazendo algo quando um Efeito tiver Terminado

Frequentemente, você irá querer executar algum código uma vez que uma animação tenha terminado -- se você executá-lo antes que a animação tenha terminado, ele pode afetar a qualidade da animação, ou pode remover elementos que são parte da animação. [Definition: *Funções de callback* fornecem um jeito de registrar seu interesse em um evento que acontecerá no futuro.] Neste caso, o evento que estaremos respondendo é a conclusão da animação. Dentro da função de callback, a palavra-chave `this` refere-se ao elemento no qual o efeito foi chamado; como se estivéssemos dentro de funções de manipulação de eventos, podemos transformá-lo em um objeto jQuery via `$(this)`.

Exemplo 6.4. Executando código quando uma animação tiver completado

```
$('#div.old').fadeOut(300, function() { $(this).remove(); });
```

Note que se sua seleção não retorna nenhum elemento, seu callback nunca será executado! Você pode resolver este problema testando se sua seleção retornou algum elemento; se não, você pode só executar a callback imediatamente.

Exemplo 6.5. Executa uma callback mesmo se não houver elementos para animar

```
var $thing = $('#naoexistente');  
  
var cb = function() {  
    console.log('pronto!');  
};  
  
if ($thing.length) {  
    $thing.fadeIn(300, cb);  
} else {  
    cb();  
}
```

Efeitos customizados com `$.fn.animate`

jQuery torna possível animar propriedades CSS arbitrárias através do método `$.fn.animate`. O método `$.fn.animate` deixa você animar para um valor definido ou para um valor relativo ao valor atual.

Exemplo 6.6. Efeitos customizados com \$.fn.animate

```
$( 'div.funtimes' ).animate(  
  {  
    left : "+=50",  
    opacity : 0.25  
  },  
  300, // duração  
  function() { console.log('pronto!'); // callback  
});
```

Nota

Propriedades relacionadas a cor não podem ser animadas com \$.fn.animate usando jQuery de forma convencional. Animações de cor podem ser facilmente efetuadas incluindo o plugin de cor [<http://plugins.jquery.com/files/jquery.color.js.txt>]. Discutiremos o uso de plugins depois no livro.

Easing

[Definition: *Easing* descreve a maneira na qual um efeito ocorre -- se a taxa de variação é constante, ou varia com a duração da animação.] jQuery inclui apenas dois métodos de easing: swing and linear. Se você quer transições mais naturais em suas animações, vários plugins de easing estão disponíveis.

A partir do jQuery 1.4, é possível fazer easing por propriedade ao usar o método \$.fn.animate.

Exemplo 6.7. Easing por propriedade

```
$( 'div.funtimes' ).animate(  
  {  
    left : [ "+=50", "swing ],  
    opacity : [ 0.25, "linear" ]  
  },  
  300  
);
```

Para mais detalhes sobre opções de easing, veja <http://api.jquery.com/animate/>.

Controlando efeitos

jQuery fornece várias ferramentas para controlar animações.

\$.fn.stop	Pára animações atualmente sendo executadas nos elementos selecionados.
\$.fn.delay	Espera um número de milissegundos especificado antes de executar a próxima animação. <pre>\$('h1').show(300).delay(1000).hide(300);</pre>
jQuery.fx.off	Se este valor for true, não haverá transição para as animações; os elementos serão imediatamente setados para o estado alvo final em vez disso. Isto pode ser especialmente útil ao lidar com navegadores antigos; você pode querer fornecer a opção para seus usuários.

Exercícios

Mostrando texto oculto

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/blog.js`. Sua tarefa é adicionar alguma interatividade à seção de blog na página. A especificação para a funcionalidade é a seguinte:

- O parágrafo de excerpt deve deslizar para baixo (slideDown) ao clicar na headline dentro da div `#blog`.
- O parágrafo de excerpt deve deslizar para baixo ao clicar na sua headline, e os outros excerpts devem deslizar para cima (slideUp).

Dica: não se esqueça do seletor `:visible`!

Criando menus dropdown

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/navigation.js`. Sua tarefa é adicionar menus dropdown à navegação principal no topo da página.

- Passar o mouse sobre um item no menu principal deve mostrar os itens do submenu, se houver.
- Tirar o mouse de cima do item deve esconder todos os itens do submenu.

Para fazer este exercício, use o método `$.fn.hover` para adicionar e remover uma classe dos itens do submenu para controlar quando ele será visível ou invisível. (O arquivo em `/exercises/css/styles.css` inclui a classe "hover" para este propósito.)

Crie um slideshow

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/slideshow.js`. Sua tarefa é pegar HTML puro e melhorá-lo com JavaScript adicionando um slideshow.

1. Mova o elemento `#slideshow` para o topo do corpo.
2. Escreva o código para navegar em torno dos itens da lista dentro do elemento; Exiba um com `fadeIn`, mostre-o por alguns segundos e então faça o desaparecer com o `fadeOut` e faça o próximo aparecer com o `fadeIn`.
3. Quando você chegar ao final da lista, comece denovo do início.

Para um desafio extra, crie uma área de navegação debaixo do slideshow que mostra quantas imagens existem e quais imagens você está vendo agora. (Dica: `$.fn.prevAll` será útil para isto.)

Capítulo 7. Ajax

Introdução

O método XMLHttpRequest permite que navegadores se comuniquem com o servidor sem precisar recarregar a página. Este método, também conhecido como Ajax (Asynchronous JavaScript and XML - JavaScript Assíncrono e XML), permite ter experiências ricas e interativas nas páginas web.

Requisições Ajax são disparadas por código JavaScript; seu código envia uma requisição a uma URL, e quando recebe uma resposta, uma função de callback pode ser acionada para tratá-la. Pelo motivo da requisição ser assíncrona, o resto do seu código continuará executando enquanto a requisição é processada, sendo assim, é imperativo que um callback seja usado para lidar com a resposta.

O jQuery provê suporte Ajax que abstrai as dolorosas diferenças entre navegadores. Ele oferece métodos completos como o `$.ajax()`, e alguns métodos de conveniência como `$.get()`, `$.getScript()`, `$.getJSON()`, `$.post()` e `$.load()`.

A maior parte das aplicações jQuery não usam XML apesar do nome "Ajax"; ao invés disso, elas transportam HTML puro ou JSON (JavaScript Object Notation - Notação de Objeto do JavaScript).

Em geral, Ajax não funciona através de domínios diferentes. As exceções são serviços que fornecem suporte ao JSONP (JSON com padding), que permite uma limitada funcionalidade entre domínios.

Conceitos Chave

O uso apropriado de métodos relacionados ao Ajax requer o conhecimento de alguns conceitos-chave.

GET vs. POST

Os métodos mais comuns para enviar uma requisição ao servidor são o GET e o POST. É importante entender a aplicação apropriada para cada um.

O método GET deve ser usado para operações não-destrutivas - ou seja, operações que você apenas esteja "pegando" dados do servidor, sem modificar nenhum dado no servidor. Por exemplo, uma consulta para fazer uma busca pode ser uma requisição GET. Requisições GET podem ser cacheadas pelo navegador, que pode levar a comportamentos imprevisíveis se você não tomar cuidado. Requisições GET geralmente enviam todos os seus dados na string de requisição.

O método POST deve ser usado para operações destrutivas - ou seja, operações onde você muda dados no servidor. Por exemplo, um usuário salvando o post de um blog deve ser uma requisição POST. Requisições POST geralmente não são cacheadas pelo navegador; uma string de requisição pode fazer parte da URL, mas os dados tendem a ser enviados separadamente como uma requisição POST.

Tipos de dados

O jQuery geralmente requer alguma instrução a respeito do tipo de dados que você espera obter com uma requisição Ajax; em alguns casos, o tipo de dado é especificado no nome do método e em outros casos é fornecido como parte do objeto de configuração. Há várias opções:

texto Para transportar strings simples

html Para transportar blocos de HTML para serem colocados na página

script	Para adicionar um novo script à página
json	Para transportar dados formatados no estilo JSON, que pode incluir strings, arrays e objetos

Nota

No jQuery 1.4, se os dados JSON enviados pelo seu servidor não estiverem propriamente formatados, a requisição pode falhar silenciosamente. Dê uma olhada em <http://json.org> para detalhes sobre formatação correta do JSON, mas como regra geral, use métodos já prontos da linguagem usada no servidor para gerar JSON sem erros de sintaxe.

jsonp	Para transportar dados JSON de outro domínio.
xml	Para transporte de dados em um XML

Eu sou um forte proponente no uso de JSON na maioria dos casos, de forma que ele provê a maior flexibilidade. É especialmente útil para enviar HTML e dados ao mesmo tempo.

A é de assíncrono

A assincronicidade do Ajax pega muitos novos usuários do jQuery desprevinidos. Pelo fato das chamadas Ajax serem assíncronas por padrão, a resposta não estará disponível imediatamente. Respostas só podem ser manipuladas por um callback. Então, por exemplo, o código seguinte não irá funcionar:

```
$.get('foo.php');  
console.log(response);
```

Ao invés disso, nós temos que passar um função callback para nossa requisição; Este callback será executado quando a requisição for realizada com sucesso, e é neste ponto que poderemos acessar os dados que ela retornou, se houver algum.

```
$.get('foo.php', function(response) { console.log(response); });
```

Regra da mesma origem e JSONP

Em geral, as requisições Ajax são limitadas ao mesmo protocolo (http ou https), a mesma porta, e ao mesmo domínio da página que está fazendo a requisição. Estas limitações não se aplicam a scripts que são carregados pelos métodos Ajax do jQuery.

As outras exceções são requisições direcionadas a um serviço JSONP em outro domínio. No caso do JSONP, o provedor do serviço tem que concordar em responder a sua requisição com um script que pode ser carregado dentro da mesma página usando uma tag `<script>`, assim evitando a limitação da mesma origem; este script terá os dados que você requisitou encapsulado em uma função de callback que você especificou.

Ajax e o Firebug

O Firebug (ou o Webkit Inspector no Chrome ou Safari) é uma ferramenta indispensável para trabalhar com requisições Ajax. Você pode ver as requisições Ajax no mesmo instante que elas acontecem na tab Console do Firebug (e no painel Resources > XHR do Webkit Inspector), e você pode clicar numa requisição para expandí-la e ver os detalhes como os headers de requisição, de resposta, conteúdo e mais. Se alguma coisa não estiver acontecendo como esperado com uma requisição Ajax, este é o primeiro lugar para ver o que está acontecendo de errado.

Métodos do jQuery relacionados ao Ajax

Enquanto o jQuery oferece muitos métodos convenientes relacionados ao Ajax, o coração de todos eles é o método `$.ajax` e entendê-lo é imperativo. Nós vamos revisá-lo primeiro e então dar uma olhada rápida nos métodos de conveniência.

Eu geralmente uso o método `$.ajax` e não uso os métodos de conveniência. Como você poderá ver, ele fornece recursos que os métodos de conveniência não oferecem e sua sintaxe é mais facilmente inteligível, na minha opinião.

`$.ajax`

O método core `$.ajax` é uma forma simples e poderosa de criar requisições Ajax. Ele utiliza um objeto de configuração que contém todas as instruções que o jQuery precisa para completar a requisição. O método `$.ajax` é particularmente útil porque ele oferece a habilidade de especificar callbacks de sucesso e falha. Há também a possibilidade de pegar um objeto de configuração definido separadamente, fazendo que seja fácil escrever código reutilizável. Para documentação completa das opções de configuração, visite <http://api.jquery.com/jquery.ajax/>.

Exemplo 7.1. Usando o método core \$.ajax

```
$.ajax({
    // a URL para requisição
    url : 'post.php',

    // Os dados a serem enviados
    // (serão convertidos em uma string de requisição)
    data : { id : 123 },

    // se esta é uma requisição POST ou GET
    method : 'GET',

    // o tipo de dados que nós esperamos
    dataType : 'json',

    // código a ser executado se a requisição
    // for executada com sucesso; a resposta
    // é passada para a função
    success : function(json) {
        $('<h1/>').text(json.title).appendTo('body');
        $('<div class="content"/>')
            .html(json.html).appendTo('body');
    },

    // código a ser executado se a requisição
    // falhar. A requisição bruta e os códigos
    // de status são passados para função
    error : function(xhr, status) {
        alert('Desculpa, aconteceu um problema!');
    },

    // Código a ser executado independentemente
    // do sucesso ou falha
    complete : function(xhr, status) {
        alert('A requisição está completa!');
    }
});
```

Nota

Uma nota a respeito da configuração `dataType`: se o servidor enviar dados que estão num formato diferente que você especificou, seu código poderá falhar, e a razão nem sempre será clara, por que o código de resposta HTTP não irá mostrar um erro. Quando estiver trabalhando com requisições Ajax, tenha certeza que o servidor está retornando os dados com o formato que você especificou e verifique se o header `Content-type` está de acordo com o tipo de dados. Por exemplo, para dados JSON, o header `Content-type` deve ser `application/json`.

Opções do \$.ajax

Há várias, várias opções para o método `$.ajax`, que é parte do seu poder. Para uma lista completa de opções, visite <http://api.jquery.com/jquery.ajax/>; aqui vão algumas opções que você irá usar frequentemente:

<code>async</code>	Configure para <code>false</code> se a requisição deve ser enviada de forma síncrona. O padrão é <code>true</code> . Perceba que se você configurar esta opção para falso, sua requisição bloqueará a execução de outro código até que a resposta seja recebida.
<code>cache</code>	Usado se a resposta disponível for cacheável. O padrão é <code>true</code> para todos os tipos de dados, exceto "script" e "jsonp". Quando setado para falso, a URL simplesmente terá um parâmetro a mais anexado a ela para evitar o cache.
<code>complete</code>	Uma função callback para executar quando a requisição estiver completa, independentemente de sucesso ou falha. A função recebe o objeto bruto da requisição e o texto de status da mesma.
<code>context</code>	O escopo em que a função callback deve executar (ou seja, o que <code>this</code> irá significar dentro da(s) função(ões) callback). Por padrão, <code>this</code> dentro das funções callback refere ao objeto originalmente passado para <code>\$.ajax</code> .
<code>data</code>	Os dados a serem enviados para o servidor. Isto pode ser tanto um objeto quanto uma string de requisição, como <code>foo=bar&baz=bim</code> .
<code>dataType</code>	O tipo de dado que você espera do servidor. Por padrão, o jQuery irá olhar o MIME type da resposta se nenhum tipo de dados for especificado.
<code>error</code>	Uma função callback para ser executada se a requisição resultar em um erro. A função recebe o objeto bruto de requisição e o texto de status da requisição.
<code>jsonp</code>	O nome da função de callback para enviar na string de requisição quando estiver fazendo uma requisição JSONP. O padrão é "callback".
<code>success</code>	Uma função de callback para ser executada se a requisição tiver êxito. A função recebe os dados de resposta (convertidos para um objeto JavaScript se o tipo de dados for JSON), e o texto de status da requisição e o objeto bruto da requisição.
<code>timeout</code>	O tempo em milissegundos a se esperar antes de considerar que a requisição falhou.
<code>traditional</code>	Configure para <code>true</code> para usar o tipo de serialização <code>param</code> em versões anteriores ao jQuery 1.4. Para detalhes, veja: http://api.jquery.com/jquery.param/ .
<code>type</code>	O tipo da requisição, "POST" ou "GET". O padrão é "GET". Outros tipos de requisições, como "PUT" e "DELETE" podem ser utilizados, mas eles talvez não sejam suportados por todos os navegadores.
<code>url</code>	A URL para requisição.

A opção `url` é a única propriedade obrigatória do objeto de configuração do método `$.ajax`; todas as outras propriedades são opcionais.

Métodos de conveniência

Se você não precisa da configuração extensiva do `$.ajax`, e você não se preocupa sobre manipulação de erros, as funções de conveniência providas pelo jQuery podem ser uma forma útil e lapidada para executar requisições Ajax. Estes métodos são somente "encapsulamentos" em torno do método `$.ajax`, e simplesmente pré setam algumas das opções no método `$.ajax`.

Os métodos de conveniência providos pelo jQuery são:

<code>\$.get</code>	Executa uma requisição GET para a URL informada.
<code>\$.post</code>	Executa uma requisição POST para a URL informada.
<code>\$.getScript</code>	Adiciona um script à página.
<code>\$.getJSON</code>	Executa uma requisição GET com expectativa de retorno de um JSON.

Em cada caso, os métodos pegam os seguintes argumentos, em ordem:

<code>url</code>	A URL para requisição. Obrigatório.
<code>data</code>	Os dados para serem enviados para o servidor. Opcional. Pode ser tanto um objeto quanto uma string de requisição, como <code>foo=bar&baz=bim</code> .

Nota

Esta opção não é válida para `$.getScript`.

<code>callback de sucesso</code>	Uma função de callback para executar se a requisição for executada com sucesso. Opcional. A função recebe os dados de resposta (convertidos para um objeto JavaScript se o tipo de dados for JSON), assim como o texto de status e o objeto bruto da requisição.
<code>tipo de dados</code>	O tipo de dados que você espera de retorno do servidor. Opcional.

Nota

Esta opção só é aplicável para métodos que ainda não especificaram o tipo de dados no seu nome.

Exemplo 7.2. Usando os métodos de conveniência do jQuery

```
// pega um texto puro ou html
$.get('/users.php', { userId : 1234 }, function(resp) {
    console.log(resp);
});

// adiciona um script na página, e então executa uma função
// definida nele
$.getScript('/static/js/myScript.js', function() {
    functionFromMyScript();
});

// pega dados formatados em JSON do servidor
$.getJSON('/details.php', function(resp) {
    $.each(resp, function(k, v) {
        console.log(k + ' : ' + v);
    });
});
```

`$.fn.load`

O método `$.fn.load` é único dentre os métodos de Ajax do jQuery que é chamado em uma seleção. O método `$.fn.load` pega o HTML de uma URL, e usa o HTML retornado para popular o(s) elemento(s)

selecionados. Em adição à URL informada, você ainda pode informar um seletor; o jQuery irá pegar somente o elemento correspondente do HTML retornado.

Exemplo 7.3. Usando o `$.fn.load` para popular um elemento

```
$('#newContent').load('/foo.html');
```

Exemplo 7.4. Usando o `$.fn.load` para popular um elemento baseado no seletor

```
$('#newContent').load('/foo.html #myDiv h1:first', function(html) {  
    alert('Conteúdo atualizado!');  
});
```

Ajax e formulários

As funcionalidades de Ajax do jQuery podem ser especialmente úteis ao lidar com formulários. O jQuery Form Plugin [<http://jquery.malsup.com/form/>] é uma ferramenta bem testada para adicionar funcionalidades de Ajax nos formulários e você deve usá-lo para manipular formulários com Ajax ao invés de usar sua própria solução para alguma coisa mais complexa. Mas ainda há dois métodos do jQuery relacionados à processamento de formulários que você deve conhecer: `$.fn.serialize` e `$.fn.serializeArray`.

Exemplo 7.5. Transformando os dados de um formulário em uma string de requisição

```
$('#myForm').serialize();  
// cria uma estrutura como esta:  
  
name=campo1&value=123
```

Exemplo 7.6. Criando um array de objetos contendo dados do formulário.

```
$('#myForm').serializeArray();  
  
// cria uma estrutura como esta:  
[  
    { name : 'campo1', value : 123 },  
    { name : 'campo2', value : 'olá mundo!' }  
]
```

Trabalhando com JSONP

O advento do JSONP -- essencialmente um hack consensual para cross-site scripting -- abriu a porta para mashups de conteúdo bem poderosos. Muitos sites provêem serviços JSONP, permitindo que você acesse o conteúdo deles através de uma API pré-definida. Uma boa fonte de dados formatados em JSONP é o Yahoo! Query Language [<http://developer.yahoo.com/yql/console/>], que nós iremos usar no próximo exemplo para pegar notícias a respeito de gatos.

Exemplo 7.7. Usando YQL e JSONP

```
$.ajax({
  url : 'http://query.yahooapis.com/v1/public/yql',

  // o nome do parâmetro de callback,
  // como é especificado pelo serviço do YQL
  jsonp : 'callback',

  // fala para o jQuery que estamos esperando JSONP
  dataType : 'jsonp',

  // fala para o YQL o que nós queremos e que queremos em JSON
  data : {
    q : 'select title,abstract,url from search.news where query="gato"',
    format : 'json'
  },

  // trabalha com a resposta
  success : function(response) {
    console.log(response);
  }
});
```

O jQuery manipula todos os aspectos complexos por trás do JSONP -- tudo que nós temos que fazer é falar para o jQuery o nome do callback JSONP especificado pelo YQL ("callback" neste caso), e todo o processo se parece com uma requisição Ajax normal.

Eventos do Ajax

Frequentemente, você irá querer fazer algo depois que uma requisição Ajax inicia ou termina, como exibir ou mostrar um indicador de carregamento. Ao invés de definir este comportamento dentro de cada requisição Ajax, você pode associar eventos do Ajax à elementos da mesma forma que você associa outros eventos. Para uma lista completa de eventos do Ajax, visite http://docs.jquery.com/Ajax_Events.

Exemplo 7.8. Configurando um indicador de carregamento usando eventos do Ajax

```
$('#loading_indicator')
  .ajaxStart(function() { $(this).show(); })
  .ajaxStop(function() { $(this).hide(); });
```

Exercícios

Carregar conteúdo externo

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/load.js`. Sua tarefa é carregar o conteúdo de um item de blog quando um usuário clicar no título do item.

1. Crie uma div alvo depois da headline para cada post do blog e armazene uma referência para ela na headline usando `$.fn.data`.
2. Crie um bind do evento click na headline que irá usar o método `$.fn.load` para carregar o conteúdo apropriado de `/exercises/data/blog.html` dentro da div alvo. Não se esqueça de prevenir a ação padrão do evento click.

Perceba que cada headline do blog em `index.html` inclui um link para o post. Você precisará aproveitar o href daquele link para obter o conteúdo de `blog.html`. Uma vez que você tem o href, esta é uma forma de processá-lo num ID que você pode usar como seletor em `$.fn.load`:

```
var href = 'blog.html#post1';
var tempArray = href.split('#');
var id = '#' + tempArray[1];
```

Lembre-se de fazer uso liberal de `console.log` para ter certeza que você está no caminho correto!

Carregando conteúdo usando JSON

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/specials.js`. Sua tarefa é mostrar os detalhes do usuário para um dado dia quando o usuário seleciona um dia no menu dropdown de seleção.

1. Anexe uma div alvo depois do formulário que está dentro do elemento `#specials`; este será o lugar onde você colocará informações sobre o especial uma vez que você recebeu a resposta do servidor.
2. Faça um bind no evento `change` do elemento `select`; quando o usuário mudar a seleção, envie uma requisição Ajax para `/exercises/data/specials.json`.
3. Quando a requisição retornar um resposta, use o valor que o usuário selecionou no `select` (dica: `$.fn.val`) para procurar informações sobre o especial na resposta JSON.
4. Adicione algum HTML sobre o especial na div alvo que você criou.
5. Por último, pelo fato do form ter Ajax habilitado, retire o botão submit do form.

Perceba que nós estamos carregando o JSON todas as vezes que o usuário muda a seleção. Como podemos mudar o código para que uma requisição só seja feita uma única vez e então usarmos a resposta cacheada quando o usuário modificar a caixa de seleção?

Capítulo 8. Plugins

O que exatamente é um plugin?

Um plugin do jQuery é simplesmente um novo método que nós usamos para estender o protótipo de objeto do jQuery. Através da extensão do protótipo, você permite que todos os objetos do jQuery herdem quaisquer métodos que você adicionar. Como estabelecido, sempre que você chama `jQuery()` você está criando um novo objeto do jQuery, com todos os métodos herdados.

A idéia de um plugin é fazer alguma coisa com uma coleção de elementos. Você pode considerar que cada método que vem com o core do jQuery seja um plugin, como `fadeOut` ou `addClass`.

Você pode fazer seus próprios plugins e usá-los privadamente no seu código ou você pode liberá-lo para comunidade. Há milhares de plugins para o jQuery disponíveis online. A barreira para criar um plugin próprio é tão pequena que você desejará fazer um logo "de cara"!

Como criar um plugin básico

A notação para criar um novo plugin é a seguinte:

```
(function($) {  
    $.fn.myNewPlugin = function() {  
        return this.each(function() {  
            // faz alguma coisa  
        });  
    };  
})(jQuery);
```

Mas não se deixe confundir. O objetivo de um plugin do jQuery é estender o protótipo do objeto do jQuery, e isso é o que está acontecendo nesta linha:

```
$.fn.myNewPlugin = function() { //...
```

Nós encapsulamos esta associação numa função imediatamente invocada:

```
(function($) {  
    //...  
})(jQuery);
```

Isso possui o efeito de criar um escopo "privado" que nos permite estender o jQuery usando o símbolo de dólar sem ter o risco de ter o dólar sobrescrito por outra biblioteca.

Então nosso verdadeiro plugin, até agora, é este:

```
$.fn.myNewPlugin = function() {  
    return this.each(function() {  
        // faz alguma coisa  
    });  
};
```

A palavra chave `this` dentro do novo plugin refere-se ao objeto jQuery em que o plugin está sendo chamado.

```
var somejQueryObject = $('#something');

$.fn.myNewPlugin = function() {
    alert(this === somejQueryObject);
};

somejQueryObject.myNewPlugin(); // alerta 'true'
```

Seu objeto jQuery típico conterá referências para qualquer número de elementos DOM, e esse é o porquê que objetos jQuery são geralmente referenciados à coleções.

Então, para fazer algo com uma coleção, nós precisamos iterar sobre ela, que é mais facilmente feito utilizando o método `each()` do jQuery:

```
$.fn.myNewPlugin = function() {
    return this.each(function(){

    });
};
```

O método `each()` do jQuery, assim como a maioria dos outros métodos, retornam um objeto jQuery, permitindo-nos assim que possamos saber e adorar o 'encadeamento' (`$(...).css().attr()...`). Nós não gostaríamos de quebrar esta convenção, então nós retornamos o objeto `this`. Neste loop, você poder fazer o que você quiser com cada elemento. Este é um exemplo de um pequeno plugin utilizando uma das técnicas que nós discutimos:

```
(function($){
    $.fn.showLinkLocation = function() {
        return this.filter('a').each(function(){
            $(this).append(
                ' (' + $(this).attr('href') + ') '
            );
        });
    };
})(jQuery));

// Exemplo de uso:
$('a').showLinkLocation();
```

Este prático plugin atravessa todas as âncoras na coleção e anexa o atributo `href` entre parênteses.

```
<!-- antes do plugin ser chamado: -->
<a href="page.html">Foo</a>

<!-- Depois que o plugin foi chamado: -->
<a href="page.html">Foo (page.html)</a>
```

Nosso plugin pode ser otimizado:

```
(function($){
    $.fn.showLinkLocation = function() {
        return this.filter('a').append(function(){
            return ' (' + this.href + ') '
        });
    };
};
```

```
})(jQuery));
```

Nós estamos utilizando a capacidade de aceitação de um callback do método `append`, e o valor de retorno deste callback irá determinar o que será aplicado a cada elemento na coleção. Perceba também que nós não estamos usando o método `attr` para obter o atributo `href`, pois a API do DOM nativa nos dá um acesso facilitado através da propriedade `href`.

Este é um outro exemplo de plugin. Este não requer que nós façamos uma iteração sobre todos os elementos com o método `each()`. Ao invés disso, nós simplesmente iremos delegar para outros método do jQuery diretamente:

```
(function($){
    $.fn.fadeInAndAddClass = function(duration, className) {
        return this.fadeIn(duration, function(){
            $(this).addClass(className);
        });
    };
})(jQuery);
```

```
// Exemplo de uso:
$('a').fadeInAndAddClass(400, 'finishedFading');
```

Procurando & Avaliando Plugins

Os plugins estendem funcionalidades básicas do jQuery, e um dos aspectos mais celebrados da biblioteca é seu extensivo ecossistema de plugins. De ordenação de tabelas à validação de formulário e autocompletamento ... se há uma necessidade para algo, há boas chances que alguém já tenha escrito um plugin para isso.

A qualidade dos plugins do jQuery varia muito. Muitos plugins são extensivamente testados e bem mantidos, mas outros são porcammente criados e então ignorados. Mais do que algumas falhas para seguir as melhores práticas.

O Google é seu melhor recurso inicial para localização de plugins, embora o time do jQuery esteja trabalhando em um repositório de plugin melhorado. Uma vez que você identificou algumas opções através de uma busca do Google, você pode querer consultar a lista de emails do jQuery ou o canal de IRC `#jquery` para obter informações de outros.

Quando estiver procurando por um plugin para preencher uma necessidade, faça seu trabalho de casa. Tenha certeza que o plugin é bem documentado, e veja se o autor provê vários exemplos do seu uso. Tenha cuidado com plugins que fazem muito mais do que você precisa; eles podem acabar adicionando um overhead substancial à sua página. Para mais dicas sobre como identificar um plugin ruim, leia *Signs of a poorly written jQuery plugin* [<http://remysharp.com/2010/06/03/signs-of-a-poorly-written-jquery-plugin/>] do Remy Sharp.

Uma vez que você escolhe um plugin, você precisará adicioná-lo à sua página. Baixe o plugin, descompacte-o se necessário, coloque-o no diretório da sua aplicação e então inclua o plugin na sua página usando uma tag script (depois que você incluir o jQuery).

Escrevendo Plugins

Algumas vezes você quer que uma pequena funcionalidade esteja disponível pelo seu código; por exemplo, talvez você queira que um simples método possa ser chamado para executar uma série de operações sobre uma seleção do jQuery. Neste caso, você pode querer escrever um plugin.

A maioria dos plugins do jQuery são simplesmente métodos criados no namespace `$.fn`. O jQuery garante que um método chamado num objeto jQuery possa acessar aquele objeto jQuery como `this` dentro do método. Em retorno, seu plugin precisa garantir que ele retorna o mesmo objeto que ele recebeu, a menos que o contrário seja explicitamente documentado.

Este é um exemplo de um plugin simples:

Exemplo 8.1. Criando um plugin para adicionar e remover uma classe no hover

```
// definindo o plugin
(function($){
    $.fn.hoverClass = function(c) {
        return this.hover(
            function() { $(this).toggleClass(c); }
        );
    };
})(jQuery);

// utilizando o plugin
$('li').hoverClass('hover');
```

Para mais informações sobre desenvolvimento de plugins, leia o post essencial do Mike Alsup's, A Plugin Development Pattern [<http://www.learningjquery.com/2007/10/a-plugin-development-pattern>]. Nele, ele cria um plugin chamado `$.fn.hilight`, que provê suporte para o plugin de metadados se ele estiver presente e provê um método centralizado para configuração central e opções de instância para o plugin.

Exemplo 8.2. O Padrão de Desenvolvimento de Plugins do Mike Alsup

```
//
// cria a closure
//
(function($) {
  //
  // definição do plugin
  //
  $.fn.hilight = function(options) {
    debug(this);
    // constrói as opções principais antes da iteração com elemento
    var opts = $.extend({}, $.fn.hilight.defaults, options);
    // itera e reformata cada elemento encontrado
    return this.each(function() {
      $this = $(this);
      // constrói opções específicas do elemento
      var o = $.meta ? $.extend({}, opts, $this.data()) : opts;
      // atualiza estilos do elemento
      $this.css({
        backgroundColor: o.background,
        color: o.foreground
      });
      var markup = $this.html();
      // chama nossa função de formatação
      markup = $.fn.hilight.format(markup);
      $this.html(markup);
    });
  };
  //
  // função privada para debugging
  //
  function debug($obj) {
    if (window.console && window.console.log)
      window.console.log('hilight selection count: ' + $obj.size());
  };
  //
  // define e expõe nossa função de formatação
  //
  $.fn.hilight.format = function(txt) {
    return '<strong>' + txt + '</strong>';
  };
  //
  // padrões do plugin
  //
  $.fn.hilight.defaults = {
    foreground: 'red',
    background: 'yellow'
  };
  //
  // fim da closure
  //
})(jQuery);
```


Escrevendo plugins com estado utilizando a fábrica de widgets do jQuery UI

Nota

Esta seção é baseada, com permissão, no post de blog Building Stateful jQuery Plugins [<http://blog.nemikor.com/2010/05/15/building-stateful-jquery-plugins/>] de Scott Gonzalez.

Enquanto a maioria dos plugins para jQuery são stateless - isto é, nós os chamamos num elemento e isso é a extensão de nossa interação com o plugin - há um grande conjunto de funcionalidades que não se encaixam no padrão básico de plugins.

Como forma de preencher esta lacuna, o jQuery UI implementa um sistema de plugin mais avançado. O novo sistema gerencia estado, permite múltiplas funções sendo expostas através de um plugin simples, e provê vários pontos de extensão. Este sistema é chamado de fábrica de widgets e é exposto como `jQuery.widget` e faz parte do jQuery UI 1.8; entretanto, pode ser usado independentemente do jQuery UI.

Para demonstrar as capacidades da fábrica de widget, nós iremos fazer um simples plugin para barra de progresso.

Para iniciar, nós iremos criar uma barra de progresso que nos permite especificar o progresso uma só vez. Como podemos ver abaixo, isso pode ser feito através da chamada à `jQuery.widget` com dois parâmetros: o nome do plugin a ser criado e um literal objeto contendo funções para dar suporte ao nosso plugin. Quando nosso plugin for chamado, ele irá criar uma nova instância do plugin e todas as funções serão executadas dentro do contexto desta instância. Isso é diferente de um plugin padrão do jQuery em duas formas importantes. Primeiro, o contexto é um objeto, não um elemento do DOM. Segundo, o contexto é sempre um objeto, nunca uma coleção.

Exemplo 8.3. Um plugin simples, stateful utilizando a fábrica de widgets do jQuery UI.

```
$.widget("nmk.progressbar", {
  _create: function() {
    var progress = this.options.value + "%";
    this.element
      .addClass("progressbar")
      .text(progress);
  }
});
```

O nome do plugin precisa conter um namespace; neste caso nós usamos o namespace `nmk`. Há uma limitação que namespaces têm que ter exatamente um nível de profundidade - isto é, nós não podemos usar um namespace como `nmk.foo`. Nós podemos também ver que a fábrica de widgets nos deu duas propriedades. `this.element` é um objeto jQuery contendo exatamente um elemento. Se nosso plugin é chamado num objeto do jQuery contendo múltiplos elementos, uma instância nova será criada para cada elemento e cada instância terá seu próprio `this.element`. A segunda propriedade, `this.options`, é um hash contendo pares chave/valor para todas as nossas opções do plugin. Estas opções podem ser passadas para nosso plugin como mostrado aqui.

Nota

No nosso exemplo, nós usamos o namespace `nmk`. O namespace `ui` é reservado para plugins oficiais do jQuery UI. Quando fizer seus próprios plugins, você deve criar seu próprio namespace. Isso deixa claro de onde o plugin veio e se ele faz parte de uma coleção maior.

Exemplo 8.4. Passando opções para um Widget

```
$("<div></div>")
  .appendTo( "body" )
  .progressbar({ value: 20 });
```

Quando nós chamamos `jQuery.widget`, ele estende o jQuery adicionando um método em `jQuery.fn` (da mesma forma que nós criamos um plugin padrão). O nome da função que ele adiciona é baseado no nome que você passa para o `jQuery.widget`, sem o namespace; no nosso caso ele irá criar `jQuery.fn.progressbar`. As opções passadas para nosso plugins estão em `this.options`, dentro de nossa instância do plugin. Como mostrado abaixo, nós podemos especificar valores padrão para qualquer uma das nossas opções. Quando estiver projetando sua API, você deve perceber o caso mais comum para seu plugin de modo que você possa setar valores padrão e fazer todas as opções verdadeiramente opcionais.

Exemplo 8.5. Setando opções padrão para um Widget

```
$.widget("nmk.progressbar", {
  // opções padrão
  options: {
    value: 0
  },

  _create: function() {
    var progress = this.options.value + "%";
    this.element
      .addClass( "progressbar" )
      .text( progress );
  }
});
```

Adicionando métodos a um Widget

Agora que nós podemos inicializar nossa barra de progresso, nós iremos adicionar a habilidade de executar ações através da chamada de métodos na instância do nosso plugin. Para definir um método do plugin nós simplesmente incluímos a função num literal objeto que nós passamos para `jQuery.widget`. Nós também podemos definir métodos “privados” se adicionarmos um underscore (“_”) antes do nome da função.

Exemplo 8.6. Criando métodos para um Widget

```
$.widget("nmk.progressbar", {
  options: {
    value: 0
  },

  _create: function() {
    var progress = this.options.value + "%";
    this.element
      .addClass("progressbar")
      .text(progress);
  },

  // cria um método público
  value: function(value) {
    // nenhum valor passado, atuando como um getter
    if (value === undefined) {
      return this.options.value;
    }
    // valor passado, atuando como um setter
    } else {
      this.options.value = this._constrain(value);
      var progress = this.options.value + "%";
      this.element.text(progress);
    }
  },

  // criando um método privado
  _constrain: function(value) {
    if (value > 100) {
      value = 100;
    }
    if (value < 0) {
      value = 0;
    }
    return value;
  }
});
```

Para chamar um método numa instância do plugin, você precisa passar o nome do método para o plugin do jQuery. Se você estiver chamando um método que aceita parâmetros, você simplesmente passa estes parâmetros depois do nome do método.

Exemplo 8.7. Chamando métodos numa instância do plugin

```
var bar = $("<div></div>")
    .appendTo("body")
    .progressbar({ value: 20 });

// pega o valor atual
alert(bar.progressbar("value"));

// atualiza o valor
bar.progressbar("value", 50);

// pega o valor atual denovo
alert(bar.progressbar("value"));
```

Nota

Executar métodos através da passagem do nome do mesmo para a mesma função do jQuery que foi usada para inicializar o plugin pode parecer estranho. Isso é feito para prevenir a poluição do namespace do jQuery enquanto mantém a habilidade de encadear chamadas de métodos.

Trabalhando com Opções de Widget

Um dos métodos que é automaticamente disponível para nosso plugin é o método `option`. O método `option` permite que você obtenha e defina opções depois da inicialização. Este método funciona exatamente como os métodos `css` e `attr` do jQuery: você pode passar somente o nome para usá-lo como um getter, um nome e um valor para usá-lo com um setter único, ou um hash de pares nome/valor para setar múltiplos valores. Quando usado como um getter, o plugin irá retornar o valor atual da opção que corresponde ao nome que foi passado. Quando usado como um setter, o método `_setOption` do plugin será chamado para cada opção que estiver sendo setada. Nós podemos especificar um método `_setOption` no nosso plugin para reagir a mudanças de opção.

Exemplo 8.8. Respondendo quando a opção é setada

```
$.widget("nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
    },

    _update: function() {
        var progress = this.options.value + "%";
        this.element.text(progress);
    }
});
```

Adicionando Callbacks

Uma das formas mais fáceis de fazer seu plugin extensível é adicionar callbacks para que os usuários possam reagir quando o estado do plugin mudar. Nós podemos ver abaixo como adicionar um callback à nossa barra de progresso para enviar um sinal quando a barra de progresso chegar à 100%. O método `_trigger` requer três parâmetros: o nome do callback, um objeto de evento nativo que iniciou o callback, e um hash de dados relevantes para o evento. O nome do callback é o único parâmetro requerido, mas os outros podem ser muito úteis para usuários que querem implementar funcionalidades customizadas no topo do seu plugin. Por exemplo, se nós construíssemos um plugin arrastável, nós poderíamos passar o evento `mouseover` nativo quando dispararmos o callback de arrastar; Isso permitiria que usuários reagissem ao arraste baseado nas coordenadas x/y providas pelo objeto do evento.

Exemplo 8.9. Provendo callbacks para extensão do usuário

```
$.widget("nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
    },

    _update: function() {
        var progress = this.options.value + "%";
        this.element.text(progress);
        if (this.options.value == 100) {
            this._trigger("complete", null, { value: 100 });
        }
    }
});
```

Funções de callback são essencialmente opções adicionais, então você pode obter e setá-las como qualquer outra opção. Quando um callback é executado, um evento correspondente é executado também. O tipo do evento é determinado pela concatenação do nome do plugin e do nome do callback. O callback e o evento recebem os mesmos dois parâmetros: um objeto evento e um hash dos dados relevantes para o evento, como veremos a seguir.

Se seu plugin tem funcionalidade que você deseja permitir que o usuário previna, a melhor forma de suportar isso é criando callbacks canceláveis. Usuários podem cancelar um callback, ou seu evento associado, da mesma forma que eles cancelam qualquer evento nativo: chamando `event.preventDefault()` ou usando `return false`. Se o usuário cancelar o callback, o método `_trigger` irá retornar falso para que você implemente a funcionalidade apropriada dentro do seu plugin.

Exemplo 8.10. Vinculando à eventos do Widget

```
var bar = $("<div></div>")
    .appendTo("body")
    .progressbar({
        complete: function(event, data) {
            alert( "Callbacks são ótimos!" );
        }
    })
    .bind("progressbarcomplete", function(event, data) {
        alert("Eventos borbulham e suportam muitos manipuladores para extrema flex
        alert("O valor da barra de progresso é " + data.value);
    });

bar.progressbar("option", "value", 100);
```

A Fábrica de Widget: Nos Bastidores

Quando você chama `jQuery.widget`, ele cria uma função construtora para seu plugin e seta o literal objeto que você passou como protótipo para suas instâncias do plugin. Todas as funcionalidades que são automaticamente adicionadas a seu plugin vem de um protótipo base de widget, que é definido como `jQuery.Widget.prototype`. Quando uma instância do plugin é criada, ela é armazenada no elemento DOM original utilizando `jQuery.data`, com o plugin tendo o mesmo nome da chave.

Pelo fato da instância do plugin estar diretamente ligada ao elemento DOM, você pode acessar a instância do plugin diretamente ao invés de ir ao método exposto do plugin. Isso permitirá que você chame métodos diretamente na instância do plugin ao invés de passar nomes de métodos como strings e também lhe dará acesso direto às propriedades do plugin.

```
var bar = $("<div></div>")
    .appendTo("body")
    .progressbar()
    .data("progressbar");

// chama um método diretamente na instância do plugin
bar.option("value", 50);

// acessa propriedades na instância do plugin
alert(bar.options.value);
```

Um dos maiores benefícios de ter um construtor e um protótipo para um plugin é a facilidade de estendê-lo. Podemos modificar o comportamento de todas as instâncias do nosso plugin adicionando ou modificando métodos no protótipo do plugin. Por exemplo, se quiséssemos adicionar um método à nossa barra de progresso para resetar o progresso para 0%, nós poderíamos adicionar este método ao protótipo e ele estaria instantaneamente disponível para ser chamado em qualquer instância do plugin.

```
$.nmk.progressbar.prototype.reset = function() {
    this._setOption("value", 0);
};
```

Limpendo

Em alguns casos, faz sentido permitir usuários aplicar seu plugin e desaplicá-lo depois. Você pode fazer isso pelo método `destroy`. Dentro do método `destroy`, você deve desfazer qualquer coisa que seu plugin possa ter feito desde o início. O método `destroy` é automaticamente chamado se o elemento que a instância do seu plugin está amarrado é removido do DOM, para que isso seja usado para coleção de lixo (garbage collection). O método `destroy` padrão remove a ligação entre o elemento DOM e a instância do plugin, então é importante chamar a função `destroy` base dentro do `destroy` do seu plugin.

Exemplo 8.11. Adicionando um método `destroy` à um Widget

```
$.widget( "nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
    },

    _update: function() {
        var progress = this.options.value + "%";
        this.element.text(progress);
        if (this.options.value == 100) {
            this._trigger("complete", null, { value: 100 });
        }
    },

    destroy: function() {
        this.element
            .removeClass("progressbar")
            .text("");

        // chama função destroy base
        $.Widget.prototype.destroy.call(this);
    }
});
```

Conclusão

A fábrica de widget é o único jeito de criar plugins com estado. Há alguns poucos modelos diferentes que podem ser usados e cada um tem suas vantagens e desvantagens. A fábrica de widget resolve muitos dos problemas comuns para você e pode melhorar muito a produtividade e também melhora bastante o reuso de código, fazendo com que funcione bem para o jQuery UI e para outros plugins com estado.

Exercícios

Fazendo uma tabela ordenável

Para este exercício, sua tarefa é identificar, baixar, e implementar um plugin para ordenamento de tabela na página index.html. Quando terminar, todas as colunas da tabela na página devem ser ordenáveis.

Escreva um plugin para listrar tabelas

Abra o arquivo `/exercises/index.html` no seu navegador. Use o arquivo `/exercises/js/stripe.js`. Sua tarefa é escrever um plugin chamado "stripe" que você possa chamar em qualquer elemento table. Quando o plugin for chamado no elemento table, ele deve modificar as cores das linhas ímpares no corpo da tabela para uma cor especificada pelo usuário.

```
$('#myTable').stripe('#cccccc');
```

Não se esqueça de retornar a tabela para que outros métodos possam ser encadeados depois do plugin!

Parte III. Tópicos Avançados

Esta seção é um trabalho em andamento

Por favor, visite <http://github.com/rmurphey/jqfundamentals> para contribuir!

Capítulo 9. Melhores práticas para performance

Este capítulo cobre um número de melhores práticas de jQuery e JavaScript, sem uma ordem particular. Muitas das melhores práticas neste capítulo são baseadas na apresentação Anti-padrões do jQuery para performance [<http://paulirish.com/perf>] por Paul Irish.

Armazene o length em loops

Em um loop, não acesse a propriedade length de um array todo tempo; armazene-o em outra variável antes de manipulá-lo.

```
var myLength = myArray.length;

for (var i = 0; i < myLength; i++) {
    // faz alguma coisa
}
```

Adicione novo conteúdo fora de um loop

Tocar o DOM tem um custo; se você estiver adicionando muitos elementos ao DOM, faça tudo de uma vez, não um de cada vez.

```
// isto é ruim
$.each(myArray, function(i, item) {
    var newListItem = '<li>' + item + '</li>';
    $('#ballers').append(newListItem);
});

// melhor: faça isso
var frag = document.createDocumentFragment();

$.each(myArray, function(i, item) {
    var newListItem = '<li>' + item + '</li>';
    frag.appendChild(newListItem);
});
$('#ballers')[0].appendChild(frag);

// ou isso
var myHtml = '';

$.each(myArray, function(i, item) {
    myHtml += '<li>' + item + '</li>';
});
$('#ballers').html(myHtml);
```

Mantenha as coisas DRY

Don't repeat yourself; se você estiver se repetindo, você está fazendo errado.

```
// RUIM
if ($eventfade.data('currently') != 'showing') {
    $eventfade.stop();
}

if ($eventhover.data('currently') != 'showing') {
    $eventhover.stop();
}

if ($spans.data('currently') != 'showing') {
    $spans.stop();
}

// BOM!!
var $elems = [$eventfade, $eventhover, $spans];
$.each($elems, function(i,elem) {
    if (elem.data('currently') != 'showing') {
        elem.stop();
    }
});
```

Cuidado com funções anônimas

Funções anônimas em todo lugar são dolorosas. Elas são difíceis de debugar, manter, testar ou reusar. Ao invés disso, utilize um literal objeto para organizar e nomear seus manipuladores e callbacks.

```
// RUIM
$(document).ready(function() {
    $('#magic').click(function(e) {
        $('#yayeffects').slideUp(function() {
            // ...
        });
    });

    $('#happiness').load(url + ' #unicorns', function() {
        // ...
    });
});

// MELHOR
var PI = {
    onReady : function() {
        $('#magic').click(PI.candyMtn);
        $('#happiness').load(PI.url + ' #unicorns', PI.unicornCb);
    },

    candyMtn : function(e) {
        $('#yayeffects').slideUp(PI.slideCb);
    },

    slideCb : function() { ... },

    unicornCb : function() { ... }
```

```
};  
  
$(document).ready(PI.onReady);
```

Otimize seletores

Otimização de seletores é menos importante do que costumava ser, pois mais navegadores implementam `document.querySelector()` e a carga de selecionamento muda do jQuery para o navegador. Entretanto, ainda há algumas dicas para se manter em mente.

Seletores baseados em ID

Começar seu seletor com um ID é sempre melhor.

```
// rápido  
$('#container div.robotarm');  
  
// super-rápido  
$('#container').find('div.robotarm');
```

A abordagem `$.fn.find` é mais rápida pois a primeira seleção é manipulada sem passar pelo engine de seleção do Sizzle - seleções só com ID são manipuladas usando `document.getElementById()`, que é extremamente rápido, pois é nativo para o navegador.

Especificidade

Seja específico no lado direito do seu seletor, e menos específico no esquerdo.

```
// não otimizado  
$('div.data .gonzalez');  
  
// otimizado  
$('.data td.gonzalez');
```

Use tag `.class` se possível no seu seletor mais a direita, e somente tag ou `.class` na esquerda.

Evite especificidade excessiva.

```
$('.data table.attendees td.gonzalez');  
  
// melhor: remova o meio se possível  
$('.data td.gonzalez');
```

Um DOM "magro" também ajuda a melhorar a performance do seletor, pois a engine de seleção possui poucas camadas para atravessar quando estiver procurando por um elemento.

Evite o seletor universal

Seleções que especificam ou deixam implícito que uma combinação pode ser encontrada em qualquer lugar pode ser muito lenta.

```
$('.buttons > *'); // extremamente caro  
$('.buttons').children(); // muito melhor
```

```
$('.gender :radio'); // seleção universão implícita
$('.gender *:radio'); // mesma coisa, explícito agora
$('.gender input:radio'); // muito melhor
```

Use Delegação de Evento

Delegação de evento permite que você delegue um manipulador de evento a um elemento container (uma lista não-ordenada, por exemplo) ao invés de múltiplos elementos contidos (uma lista de itens, por exemplo). O jQuery torna isso fácil com `$.fn.live` e `$.fn.delegate`. Sempre que possível, você deve usar `$.fn.delegate` ao invés de `$.fn.live`, de forma que isso elimina a necessidade de uma seleção desnecessária e seu contexto explícito (vs. o contexto de `$.fn.live` do document) reduz a sobrecarga por aproximadamente 80%.

Em adição aos benefícios de performance, delegação de eventos também permite que você adicione novos elementos contidos a sua página sem ter que re-delegar os manipuladores de eventos para eles quando são necessários.

```
// ruim (se houver muitos itens de lista)
$('li.trigger').click(handlerFn);

// melhor: delegação de eventos com $.fn.live
$('li.trigger').live('click', handlerFn);

// melhor ainda: delegação de eventos com $.fn.delegate
// permite que você especifique um contexto facilmente
$('#myList').delegate('li.trigger', 'click', handlerFn);
```

Desanexe elementos para trabalhar com eles

O DOM é lento; você quer evitar a manipulação o tanto quanto possível. O jQuery introduziu o `$.fn.detach` na versão 1.4 para ajudar a solucionar este problema, permitindo que você remova um elemento do DOM enquanto trabalha com o mesmo.

```
var $table = $('#myTable');
var $parent = table.parent();

$table.detach();
// ... adiciona um bocado de linhas à tabela
$parent.append(table);
```

Use folhas de estilo para mudar o CSS em vários elementos

Se você estiver mudando o CSS de mais de 20 elementos utilizando `$.fn.css`, considere adicionar uma tag de estilo à página para uma melhoria de performance de aproximadamente 60%.

```
// tudo certo para até 20 elementos, lento depois disso
$('a.swedberg').css('color', '#asd123');
$('<style type="text/css">a.swedberg { color : #asd123 }</style>')
  .appendTo('head');
```

Use \$.data ao invés de \$.fn.data

Usando \$.data em um elemento DOM ao invés de chamar \$.fn.data numa seleção do jQuery pode ser até 10 vezes mais rápido. Tenha certeza que você entendeu a diferença entre um elemento do DOM e uma seleção do jQuery antes de fazer isso.

```
// regular
$(elem).data(key,value);
```

```
// 10x mais rápido

$.data(elem,key,value);
```

Não faça nada quando não tiver elementos

O jQuery não dirá a você que você está tentando executar um monte de código numa seleção vazia - ele vai proceder como se nada estivesse errado. É sua responsabilidade verificar se sua seleção contém alguns elementos.

```
// RUIM: executa três funções
// antes de perceber que não a nada a se fazer com
// a seleção
$('#nosuchthing').slideUp();
```

```
// MELHOR
var $mySelection = $('#nosuchthing');
if ($mySelection.length) { mySelection.slideUp(); }
```

```
// MELHOR AINDA: adiciona um plugin doOnce
jQuery.fn.doOnce = function(func){
    this.length && func.apply(this);
    return this;
}
```

```
$('#li.cartitems').doOnce(function(){

    // faça com ajax! \o/

});
```

Este guia é especialmente aplicável para UI widgets do jQuery, que possuem muita sobrecarga mesmo quando a seleção não contém elementos.

Definição de Variável

Variáveis podem ser definidas em uma declaração ao invés de várias.

```
// velho & preso
```

```
var test = 1;
var test2 = function() { ... };
var test3 = test2(test);

// nova onda
var test = 1,
    test2 = function() { ... },
    test3 = test2(test);
```

Em funções auto-executáveis, podem ser puladas todas as definições de variáveis.

```
(function(foo, bar) { ... })(1, 2);
```

Condicionais

```
// forma antiga
if (type == 'foo' || type == 'bar') { ... }

// melhor
if (/^(foo|bar)$/.test(type)) { ... }

// procura por literal objeto
if (({ foo : 1, bar : 1 })[type]) { ... }
```

Não trate o jQuery como uma Caixa Preta

Use os fontes como sua documentação — favorite <http://bit.ly/jqsource> e vá a ela frequentemente.

Capítulo 10. Organização de código

Visão Geral

Quando você vai além de adicionar melhorias simples em seu website com jQuery e começa a desenvolver aplicações client-side completas, você precisa considerar como organizar seu código. Neste capítulo, nós iremos dar uma olhada em vários padrões para organização de código que você pode utilizar em sua aplicação com jQuery e explorar o gerenciador de dependências e sistema de build RequireJS.

Conceitos Chave

Antes de irmos para os padrões de organização de código, é importante entender alguns conceitos que são comuns a todos bons padrões de organização de código.

- Seu código deve ser dividido em unidades de funcionalidade - módulos, serviços, etc. Evite a tentação de ter todo seu código em um enorme bloco dentro do `$(document).ready()`.
- Não se repita. Identifique similaridades entre pedaços de funcionalidade e use técnicas de herança para evitar código repetitivo.
- Apesar da natureza centrada no DOM do jQuery, aplicações em JavaScript não são só sobre o DOM. Lembre-se que não são todas as funcionalidades que necessitam - ou devem - ter uma representação no DOM.
- Unidades de funcionalidade devem ser fracamente acopladas [http://en.wikipedia.org/wiki/Loose_coupling] — uma unidade de funcionalidade deve conseguir existir por si só, e a comunicação entre unidades devem ser manipuladas através de um sistema de mensagens, como eventos customizados ou pub/sub. Evite comunicação direta entre unidades de funcionalidade sempre que possível.
- Unidades de funcionalidade devem ser divididas em pequenos métodos que fazem exatamente uma só coisa. Se seus métodos são maiores que algumas linhas, você deve considerar uma refatoração.
- Opções de configuração para suas unidades de funcionalidade - URLs, strings, timeouts, etc. - devem ser informadas em propriedades ou em um objeto de configuração, não espalhadas através do código.

O conceito de baixo acoplamento pode ser especialmente problemático para desenvolvedores fazendo sua primeira investida em aplicações complexas, então preste atenção nisso quando você estiver começando.

Encapsulamento

O primeiro passo para organização de código é separar as partes de sua aplicação em peças distintas; algumas vezes, mesmo este esforço é suficiente para ceder

O literal objeto

Um literal objeto é talvez a forma mais simples de encapsular código relacionado. Ele não oferece nenhuma privacidade para propriedades ou métodos, mas é útil para eliminar funções anônimas do seu código, centralizar opções de configuração e facilitar o caminho para o reuso e refatoração.

Exemplo 10.1. Um literal objeto

```
var myFeature = {
  myProperty : 'olá',

  myMethod : function() {
    console.log(myFeature.myProperty);
  },

  init : function(settings) {
    myFeature.settings = settings;
  },

  readSettings : function() {
    console.log(myFeature.settings);
  }
};

myFeature.myProperty; // 'olá'
myFeature.myMethod(); // loga 'olá'
myFeature.init({ foo : 'bar' });
myFeature.readSettings(); // loga { foo : 'bar' }
```

O literal objeto acima é simplesmente um objeto associado a uma variável. O objeto tem uma propriedade e vários métodos. Todas as propriedades e métodos são públicas, então qualquer parte da sua aplicação pode ver as propriedades e chamar métodos no objeto. Enquanto há um método `init`, não há nada requerendo que ele seja chamado antes do objeto estar funcional.

Como aplicaríamos este padrão no código com jQuery? Vamos dizer que nós temos este código escrito no estilo tradicional do jQuery:

```
// carrega algum conteúdo ao clicar num item da lista
// utilizando o ID do item da lista e esconde o conteúdo
// em itens de lista similares
$(document).ready(function() {
  $('#myFeature li')
    .append('<div/>')
    .click(function() {
      var $this = $(this);
      var $div = $this.find('div');
      $div.load('foo.php?item=' +
        $this.attr('id'),
        function() {
          $div.show();
          $this.siblings()
            .find('div').hide();
        }
      );
    });
});
```

Se isso for estender nossa aplicação, deixar como está pode ser legal. Por outro lado, se esse for um pedaço de uma aplicação maior, nós temos que manter esta funcionalidade separada da funcionalidade não relacionada. Talvez nós ainda queiramos mover a URL para fora do código e dentro de uma área de

configuração. Por último, nós talvez precisaremos quebrar o encadeamento de métodos para ficar mais fácil modificar pedaços de funcionalidade depois.

```
var myFeature = {
  init : function(settings) {
    myFeature.config = {
      $items : $('#myFeature li'),
      $container : $('<div class="container"></div>'),
      urlBase : 'http://www.oreilja.com'
    };
  },

  // permite a sobreposição da configuração padrão
  $.extend(myFeature.config, settings);

  myFeature.setup();
},

setup : function() {
  myFeature.config.$items
    .each(myFeature.createContainer)
    .click(myFeature.showItem);
},

createContainer : function() {
  var $i = $(this),
      $c = myFeature.config.$container.clone()
        .appendTo($i);

  $i.data('container', $c);
},

buildUrl : function() {
  return myFeature.config.urlBase +
    myFeature.$currentItem.attr('id');
},

showItem : function() {
  var myFeature.$currentItem = $(this);
  myFeature.getContent(myFeature.showContent);
},

getContent : function(callback) {
  var url = myFeature.buildUrl();
  myFeature.$currentItem
    .data('container').load(url, callback);
},

showContent : function() {
  myFeature.$currentItem
    .data('container').show();
  myFeature.hideContent();
},

hideContent : function() {
  myFeature.$currentItem.siblings()
    .each(function() {
      $(this).data('container').hide();
    });
}
};

$(document).ready(myFeature.init);
```

Exemplo 10.2. Usando um literal de objeto numa funcionalidade do jQuery

```

var myFeature = {
  init : function(settings) {
    myFeature.config = {
      $items : $('#myFeature li'),
      $container : $('<div class="container"></div>'),
      urlBase : 'http://www.oreilja.com'
    };
  },

  // permite a sobreposição da configuração padrão
  $.extend(myFeature.config, settings);

  myFeature.setup();
},

setup : function() {
  myFeature.config.$items
    .each(myFeature.createContainer)
    .click(myFeature.showItem);
},

createContainer : function() {
  var $i = $(this),
      $c = myFeature.config.$container.clone()
        .appendTo($i);

  $i.data('container', $c);
},

buildUrl : function() {
  return myFeature.config.urlBase +
    myFeature.$currentItem.attr('id');
},

showItem : function() {
  var myFeature.$currentItem = $(this);
  myFeature.getContent(myFeature.showContent);
},

getContent : function(callback) {
  var url = myFeature.buildUrl();
  myFeature.$currentItem
    .data('container').load(url, callback);
},

showContent : function() {
  myFeature.$currentItem
    .data('container').show();
  myFeature.hideContent();
},

hideContent : function() {
  myFeature.$currentItem.siblings()
    .each(function() {
      $(this).data('container').hide();
    });
}
};

$(document).ready(myFeature.init);

```

A primeira coisa que você irá perceber é que esta abordagem é obviamente muito maior que a original - novamente, se isso for estender nossa aplicação, o uso do literal objeto não irá fazer sentido algum. Embora assumindo que não vai estender nossa aplicação, nós ganhamos várias coisas:

- Nós quebramos nossas funcionalidades em métodos pequenos; no futuro, se nós precisarmos mudar como o conteúdo é mostrado, é claro onde nós iremos mudar. No código original, este passo é muito mais difícil para se localizar.
- Nós eliminamos o uso de funções anônimas.
- Nós movemos as opções de configuração para fora do corpo do código e colocamos em uma localização central.
- Nós eliminamos as restrições do encadeamento, fazendo o código mais refatorável, modificável e rearranjável.

Para funcionalidades não triviais, literais de objeto são uma clara melhora sobre o longo pedaço de código dentro do bloco `$(document).ready()`, de forma nos leva a pensar sobre pedaços de funcionalidade. Entretanto, eles não são muito mais avançados do que ter um monte de declarações de funções dentro do bloco `$(document).ready()`.

O Module Pattern

O module pattern supera algumas das limitações do literal objeto, oferecendo privacidade para variáveis e funções enquanto expõe uma API pública se assim for necessário.

Exemplo 10.3. O module pattern

```
var featureCreator = function() {
    var privateThing = 'segredo',
        publicThing = 'não é segredo',

        changePrivateThing = function() {
            privateThing = 'super secreto';
        },

        sayPrivateThing = function() {
            console.log(privateThing);
            changePrivateThing();
        };

    return {
        publicThing : publicThing,
        sayPrivateThing : sayPrivateThing
    }
};

var feature = featureCreator();
feature.publicThing; // 'não é segredo'
feature.sayPrivateThing();
// loga 'segredo' e muda o valor de
// privateThing
```

No exemplo acima, nós criamos uma função `featureCreator` que retorna um objeto. Dentro da função, nós definimos algumas variáveis. Pelo fato das variáveis estarem definidas dentro da função, nós não temos

acesso à ela fora da função, a não ser que nós coloquemos no objeto de retorno. Isto significa que nenhum código externo da função tem acesso à variável `privateThing` ou a função `changePrivateThing`. Entretanto, `sayPrivateThing` tem acesso a `privateThing` e `changePrivateThing`, por causa que ambas foram definidas no mesmo escopo de `sayPrivateThing`.

Este padrão é poderoso pois, da mesma forma que você pode obter os nomes de variáveis, ele pode dar a você variáveis privadas e funções enquanto expõe uma API limitada consistindo das propriedades retornadas dos objetos e métodos.

Abaixo temos uma versão revisada do exemplo anterior, mostrando como poderíamos criar a mesma funcionalidade utilizando o `module pattern` e somente expondo um método público do módulo, `showItemByIndex()`.

Exemplo 10.4. Usando o module patterns numa funcionalidade do jQuery

```

(function() {
    var myFeature = (function() {

        var $items = $('#myFeature li'),
            $container = $('<div class="container"></div>'),
            $currentItem,

            urlBase = '/foo.php?item=',

            createContainer = function() {
                var $i = $(this),
                    $c = $container.clone().appendTo($i);

                $i.data('container', $c);
            },

            buildUrl = function() {
                return urlBase + $currentItem.attr('id');
            },

            showItem = function() {
                var $currentItem = $(this);
                getContent(showContent);
            },

            showItemByIndex = function(idx) {
                $.proxy(showItem, $items.get(idx));
            },

            getContent = function(callback) {
                $currentItem.data('container').load(buildUrl(), callback);
            },

            showContent = function() {
                $currentItem.data('container').show();
                hideContent();
            },

            hideContent = function() {
                $currentItem.siblings()
                    .each(function() {
                        $(this).data('container').hide();
                    });
            };

        config.$items
            .each(createContainer)
            .click(showItem);

        return { showItemByIndex : showItemByIndex };
    })();

    myFeature.showItemByIndex(0);
})();

```

Gerenciando dependências

Nota

Esta seção é fortemente baseada na excelente documentação do RequireJS disponível em <http://requirejs.org/docs/jquery.html>, e usada com permissão do autor do RequireJS, James Burke.

Quando um projeto alcança um determinado tamanho, gerenciar módulos de scripts começa a ficar complicado. Você precisa ter certeza da seqüência correta dos scripts e você começa a pensar seriamente em combinar scripts em um único arquivo, para que somente uma ou um número pequeno de requisições sejam feitas para carregar os scripts. Você pode também carregar código assim que necessário, depois que a página carregar.

O RequireJS é uma ferramenta para gerenciamento de dependências feita pelo James Burke e pode lhe ajudar a gerenciar módulos de scripts, carregá-los na ordem correta e tornar fácil a combinação de scripts mais tarde através de uma ferramenta própria de otimização, sem necessidade de alterar seu código de marcação. Ele também lhe fornece um meio fácil de carregar os scripts depois da página ter carregado, permitindo que você distribua o tamanho do download através do tempo.

O RequireJS tem um sistema de módulos que permite que você defina módulos com escopo bem definido, mas você não precisa seguir o sistema para obter os benefícios do gerenciamento de dependências e otimizações em tempo de build. Com o tempo, se você começar a criar código mais modular que precisa ser reutilizado em algumas outras partes, o formato de módulo do RequireJS torna fácil escrever código encapsulado que pode ser carregado sob demanda. Ele pode crescer com sua aplicação, particularmente se você deseja incorporar strings de internacionalização (i18n), para que seu projeto funcione em línguas diferentes, ou carregar algumas strings HTML e ter certeza que estas strings estão disponíveis antes de executar o código, ou mesmo usar serviços JSONP como dependências.

Obtendo o RequireJS

A forma mais fácil de usar o RequireJS com jQuery é baixando um build do jQuery que já tem o RequireJS embutido [<http://requirejs.org/docs/download.html>]. Este build exclui porções do RequireJS que duplicam funcionalidades do jQuery. Você também pode achar útil baixar um projeto jQuery de amostra que utiliza o RequireJS [<http://requirejs.org/docs/release/0.11.0/jquery-require-sample.zip>].

Utilizando o RequireJS com jQuery

Utilizar o RequireJS na sua página é simples: inclua o jQuery que tenha o RequireJS embutido e depois faça a requisição dos arquivos da sua aplicação. O exemplo a seguir assume que o jQuery e seus outros scripts estejam todos no diretório `scripts/`.

Exemplo 10.5. Utilizando o RequireJS: Um exemplo simples

```
<!DOCTYPE html>
<html>
  <head>
    <title>jQuery+RequireJS Sample Page</title>
    <script src="scripts/require-jquery.js"></script>
    <script>require([ "app" ] );</script>
  </head>
  <body>
    <h1>jQuery+RequireJS Sample Page</h1>
  </body>
</html>
```


A chamada `require(["app"])` fala para o RequireJS carregar o arquivo `scripts/app.js`. O RequireJS irá carregar todas as dependências que é passada para o `require()` sem a extensão `.js` do mesmo diretório do `require-jquery.js`, apesar que isto pode ser configurado para se comportar de forma diferente. Se você se sentir mais confortável ao especificar o caminho completo, você pode fazer o seguinte:

```
<script>require(["scripts/app.js"]);</script>
```

O que há em `app.js`? Outra chamada para o `require.js` carregar todos os scripts que você precisa e todo trabalho inicial que você quer para a página. Neste exemplo, o script `app.js` carrega dois plugins, `jquery.alpha.js` e `jquery.beta.js` (não é o nome real dos plugins, somente um exemplo). Os plugins devem estar no mesmo diretório do `require-jquery.js`:

Exemplo 10.6. Um simples arquivo JavaScript com dependências

```
require(["jquery.alpha", "jquery.beta"], function() {  
    //os plugins jquery.alpha.js e jquery.beta.js foram carregados.  
    $(function() {  
        $('body').alpha().beta();  
    });  
});
```

Criando módulos reusáveis com RequireJS

RequireJS torna fácil a definição de módulos reusáveis via `require.def()`. Um módulo RequireJS pode ter dependências que podem ser usadas para definir um módulo e um módulo RequireJS pode retornar um valor - um objeto, uma função, o que for - que pode ser consumido por outros módulos.

Se seu módulo não tiver nenhuma dependência, então simplesmente especifique o nome do módulo como primeiro argumento do `require.def()`. O segundo argumento é somente um literal objeto que define as propriedades do módulo. Por exemplo:

Exemplo 10.7. Definindo um módulo do RequireJS que não tem dependências

```
require.def("my/simpleshirt",
  {
    color: "black",
    size: "unysize"
  }
);
```

Este exemplo seria melhor armazenado no arquivo my/simpleshirt.js.

Se seu módulo possui dependências, você pode especificá-las como segundo argumento para `require.def()` (como um array) e então passar uma função como terceiro argumento. A função será chamada para definir o módulo quando todas as dependências tiverem sido carregadas. A função recebe os valores retornados pelas dependências como seus argumentos. (na mesma ordem que elas são requeridas no array), e a função deve retornar um objeto que define o módulo.

Exemplo 10.8. Definindo um módulo do RequireJS com dependências

```
require.def("my/shirt",
  ["my/cart", "my/inventory"],
  function(cart, inventory) {
    //retorna um objeto para definir o módulo "my/shirt".
    return {
      color: "blue",
      size: "large"
      addToCart: function() {
        inventory.decrement(this);
        cart.add(this);
      }
    }
  }
);
```

Neste exemplo, um módulo my/shirt é criado. Ele depende de my/cart e my/inventory. No disco, os arquivos são estruturados assim:

```
my/cart.js
my/inventory.js
my/shirt.js
```

A função que define my/shirt não é chamada até que os módulos my/cart e my/inventory sejam carregados, e a função recebe os módulos como argumentos cart e inventory. A ordem dos argumentos da função precisam combinar com a ordem em que as dependências foram requeridas no array de dependências. O objeto retornado pela chamada da função define o módulo my/shirt. Por definir os módulos dessa forma, my/shirt não existe como um objeto global. Módulos que definem globais são explicitamente desencorajados, então múltiplas versões de um módulo podem existir na página ao mesmo tempo.

Módulos não precisam retornar objetos; qualquer retorno válido a partir de uma função é permitido.

Exemplo 10.9. Definindo um módulo do RequireJS que retorna uma função

```
require.def("my/title",
  ["my/dependency1", "my/dependency2"],
  function(dep1, dep2) {
    //retorna uma função para definir "my/title". Ele obtém ou muda
    //o título da janela.
    return function(title) {
      return title ? (window.title = title) : window.title;
    }
  }
);
```

Somente um módulo deve ser requerido por arquivo JavaScript.

Otimizando seu código: A ferramenta de build do RequireJS

Uma vez que você incorporou o RequireJS para gerenciamento de dependência, sua página está configurada para ser otimizada bem facilmente. Baixe o código fonte do RequireJS e coloque onde você quiser, preferencialmente em um lugar fora da sua área de desenvolvimento. Para os propósitos deste exemplo, o fonte do RequireJS é colocado num diretório irmão do webapp, que contém a página HTML e o diretório de scripts com todos os scripts. Estrutura completa do diretório:

```
requirejs/ (utilizado pelas ferramentas de build)
webapp/app.html
webapp/scripts/app.js
webapp/scripts/require-jquery.js
webapp/scripts/jquery.alpha.js
webapp/scripts/jquery.beta.js
```

Então, nos diretórios de scripts que tem o require-jquery.js e app.js, crie um arquivo chamado app.build.js com o seguinte conteúdo:

Exemplo 10.10. Um arquivo de configuração de build do RequireJS

```
{
  appDir: "../",
  baseUrl: "scripts/",
  dir: "../../webapp-build",
  //Comente a linha de otimização (optimize) se você quer
  //o código minificado pelo Compilador Closure Compiler utilizando
  //o modo de otimização "simple"
  optimize: "none",

  modules: [
    {
      name: "app"
    }
  ]
}
```

Para usar a ferramenta de build, você precisa do Java 6 instalado. O Compilador Closure é usado para o passo de minificação do JavaScript (se optimize: "none" estiver comentado), e requer o Java 6.

Para iniciar o build, vá no diretório webapp/scripts, e execute o seguinte comando:

```
# sistemas não-windows
../..requirejs/build/build.sh app.build.js

# sistemas windows
..\..\requirejs\build\build.bat app.build.js
```

Agora, no diretório webapp-build, app.js terá o conteúdo de app.js, jquery.alpha.js e jquery.beta.js numa só linha. Então, se você carregar o arquivo app.html no diretório webapp-build, você não deverá ver nenhuma requisição de rede para jquery.alpha.js e jquery.beta.js.

Exercícios

Crie um módulo Portlet

Abra o arquivo `/exercises/portlets.html` em seu navegador. Use o arquivo `/exercises/js/portlets.js`. Sua tarefa é criar um função criadora de portlet que utilize o module pattern, fazendo que o seguinte código funcione:

```
var myPortlet = Portlet({
  title : 'Curry',
  source : 'data/html/curry.html',
  initialState : 'aberto' // ou 'fechado'
});
```

```
myPortlet.$element.appendTo('body');
```

Cada portlet deve ser uma div com um título, uma área de conteúdo, um botão para abrir/fechar o portlet, um botão para remover o portlet, e um botão para atualizar o portlet. O portlet retornado pela função `Portlet` deve ter a seguinte API pública:

```
myPortlet.open(); // força o estado aberto
myPortlet.close(); // força o estado fechado
myPortlet.toggle(); // inverte o estado aberto/fechado
myPortlet.refresh(); // atualiza o conteúdo
myPortlet.destroy(); // remove o portlet da página
myPortlet.setSource('data/html/onions.html');
// muda a fonte
```

Capítulo 11. Eventos Customizados

Introdução à Eventos Customizados

Nós todos estamos familiares com eventos básicos - click, mouseover, focus, blur, submit, etc. - estes que nós podemos usar para interações entre o usuário e o navegador. Eventos customizados abrem um mundo completamente novo de programação orientada a eventos. Neste capítulo, nós iremos utilizar o sistema de eventos customizados do jQuery para fazer uma aplicação de busca simples no Twitter.

Pode ser difícil à primeira instância entender por que você iria querer utilizar eventos customizados, quando os eventos já prontos parecem satisfazer bem suas necessidades. Acontece que eventos customizados oferecem um jeito completamente novo de pensar sobre JavaScript orientado a eventos. Ao invés de focar no elemento que dispara uma ação, eventos customizados colocam o holofote no elemento em ação. Isso traz muitos benefícios, incluindo:

- Comportamentos do elemento alvo podem ser facilmente disparados por elementos diferentes utilizando o mesmo código.
- Comportamentos podem ser disparados através de elementos-alvo múltiplos e similares de uma vez.
- Comportamentos podem ser mais facilmente associados com o elemento alvo no código, tornando o código mais fácil de ler e manter.

Por que você deveria se importar? Um exemplo é provavelmente a melhor forma de explicar. Suponha que você tenha uma lâmpada num quarto na casa. A lâmpada está ligada e ela é controlada por dois switches de três vias e um clapper:

```
<div class="room" id="kitchen">
  <div class="lightbulb on"></div>
  <div class="switch"></div>
  <div class="switch"></div>
  <div class="clapper"></div>
</div>
```

O estado da lâmpada irá mudar disparando o clapper ou mesmo os switches. Os switches e o clapper não se importam em qual estado a lâmpada está; eles somente querem mudar o estado.

Sem eventos customizados, você talvez poderia escrever algo assim:

```
$('.switch, .clapper').click(function() {
  var $light = $(this).parent().find('.lightbulb');
  if ($light.hasClass('on')) {
    $light.removeClass('on').addClass('off');
  } else {
    $light.removeClass('off').addClass('on');
  }
});
```

Com eventos customizados, seu código pode ficar parecido com este:

```
$('.lightbulb').bind('changeState', function(e) {
  var $light = $(this);
  if ($light.hasClass('on')) {
    $light.removeClass('on').addClass('off');
  }
```

```
    } else {  
        $light.removeClass('off').addClass('on');  
    }  
});  
  
$('.switch, .clapper').click(function() {  
    $(this).parent().find('.lightbulb').trigger('changeState');  
});
```

Este último pedaço de código não é lá essas coisas, mas algo importante aconteceu: nós movemos o comportamento da lâmpada para a lâmpada, e ficamos longe dos switches e do clapper.

Vamos tornar nosso exemplo um pouco mais interessante. Nós vamos adicionar outro quarto à nossa casa, junto com o switch mestre, como mostrado a seguir:

```
<div class="room" id="kitchen">  
    <div class="lightbulb on"></div>  
    <div class="switch"></div>  
    <div class="switch"></div>  
    <div class="clapper"></div>  
</div>  
<div class="room" id="bedroom">  
    <div class="lightbulb on"></div>  
    <div class="switch"></div>  
    <div class="switch"></div>  
    <div class="clapper"></div>  
</div>  
<div id="master_switch"></div>
```

Se há várias luzes na casa, nós queremos que o switch mestre desligue todas as luzes; de outra forma, nós queremos todas as luzes acesas. Para fazer isso, nós iremos adicionar dois ou mais eventos customizados às lâmpadas: `turnOn` e `turnOff`. Nós iremos fazer uso deles no evento customizado `changeState`, e usar alguma lógica para decidir qual que o switch mestre vai disparar:

```
$('.lightbulb')  
    .bind('changeState', function(e) {  
        var $light = $(this);  
        if ($light.hasClass('on')) {  
            $light.trigger('turnOff');  
        } else {  
            $light.trigger('turnOn');  
        }  
    })  
    .bind('turnOn', function(e) {  
        $(this).removeClass('off').addClass('on');  
    })  
    .bind('turnOff', function(e) {  
        $(this).removeClass('off').addClass('on');  
    });  
  
$('.switch, .clapper').click(function() {  
    $(this).parent().find('.lightbulb').trigger('changeState');  
});  
  
$('#master_switch').click(function() {
```

```

    if ($('#lightbulb.on').length) {
        $('#lightbulb').trigger('turnOff');
    } else {
        $('#lightbulb').trigger('turnOn');
    }
});

```

Note como o comportamento do switch mestre está vinculado ao switch mestre; O comportamento de uma lâmpada pertence às lâmpadas.

Nota

Se você está acostumado com programação orientada a objetos, talvez seja útil pensar em eventos como se fossem métodos de objetos. Grosseiramente falando, o objeto que o método pertence é criado pelo seletor do jQuery. Vinculando o evento customizado `changeState` para todos os elementos `$('.light')` é similar a ter uma classe chamada `Light` com um método `changeState`, e então instanciamos novos objetos `Light` para cada elemento com o nome de classe `light`.

Recapitulando: `$.fn.bind` e `$.fn.trigger`

No mundo dos eventos customizados, há dois métodos importantes do jQuery: `$.fn.bind` e `$.fn.trigger`. No capítulo Eventos, nós vimos como usar estes métodos para trabalhar com eventos do usuário; neste capítulo, é importante relemebrar duas coisas:

- O método `$.fn.bind` recebe o tipo do evento e uma função manipuladora do mesmo como argumentos. Opcionalmente, ele pode também receber dados relacionados ao eventos como seu segundo argumento, passando a função manipuladora do evento para o terceiro argumento. Qualquer dado que é passado será disponível para a função manipuladora do evento na propriedade `data` do objeto do evento. A função manipuladora do evento sempre recebe o objeto do evento como seu primeiro argumento.
- O método `$.fn.trigger` recebe um tipo de evento como argumento. Opcionalmente, ele também pode receber um array de valores. Estes valores serão passados para a função manipuladora de eventos como argumentos depois do objeto de evento.

Aqui há um exemplo de uso do `$.fn.bind` e `$.fn.trigger` que usa dados customizados em ambos os casos:

```

$(document).bind('myCustomEvent', { foo : 'bar' }, function(e, arg1, arg2) {
    console.log(e.data.foo); // 'bar'
    console.log(arg1); // 'bim'
    console.log(arg2); // 'baz'
});

$(document).trigger('myCustomEvent', [ 'bim', 'baz' ]);

```

Uma aplicação de amostra

Para demonstrar o poder dos eventos customizados, nós vamos criar uma ferramenta simples para buscar no Twitter. A ferramenta irá oferecer várias formas de um usuário fazer a busca por um termo: digitando numa caixa de texto, múltiplos termos de busca na URL e por requisitar os termos do trending topics.

Os resultados para cada termo serão mostrados em um container de resultados; Estes containers poderão ser expandidos, contraídos, atualizados, e removidos, individualmente ou todos de uma vez.

Quando terminarmos, a aplicação ficará dessa forma:

Twitter Search

[Load Trending Terms](#)[Refresh All Results](#)[Remove All Results](#)[Collapse All Results](#)[Expand All Results](#)[Add Search Term](#)[Refresh](#)

Search Results for #gha

[Nenevieve](#): @milky868(God I didnt go to church cos of the #Gha match)..na Advance hell I go to room...peeping though..enjoy the Victory

Sat, 26 Jun 2010 21:32:01 +0000

[jagsinghb](#): @Blanconer10 what did you think of #usa vs #gha.-- who do reckon will winning #usa stands??

Sat, 26 Jun 2010 21:32:00 +0000

[KaVilIeia](#): RT @marcelotas: Torci muito pros #USA continuar na Copa. Mas é muito legal ver o africano com a categoria e força de #GHA

Sat, 26 Jun 2010 21:32:00 +0000

[Newsrub](#): SpiesList: HuffingtonPost: #USA #GHA In extra time -- LIVE blog, tweets, photos + http://huff.to/c8GXzM; Huff... http://bit.ly/cThHCD

Sat, 26 Jun 2010 21:32:00 +0000

[DamarisPlati](#): Mais uma 'zebra' na copa! Isso aí, camisa não ganha jogo! #gha

Sat, 26 Jun 2010 21:31:59 +0000

[Refresh](#)

Search Results for #usa

[iamMATTF](#): RT @jalenrose: RT @wingoz: (#USA loses to Ghana 2-1)Now I'm depressed....ME

Sat, 26 Jun 2010 21:31:53 +0000

[patrickryan](#): @nod #USA 1st half was sloppy, #GHA took advantage, any team will waste time in 2nd half, part of the game.

Sat, 26 Jun 2010 21:31:53 +0000

[MatthewDiffie](#): I can appreciate the reasons for being happy for Ghana, but maybe wait a few days before jumping ship #worldcup #usa

Sat, 26 Jun 2010 21:31:53 +0000

[digressus](#): I feel like I just broke up with my girlfriend or something. #abouttocry #USA #worldcup

Sat, 26 Jun 2010 21:31:52 +0000

[misssalazar](#): team #usa...so proud of those guys. grateful to them for putting their hearts and souls into the game. #usa #usa #usa

Sat, 26 Jun 2010 21:31:50 +0000

O Setup

Nós iniciaremos com HTML básico:

```
<h1>Twitter Search</h1>
<input type="button" id="get_trends"
  value="Load Trending Terms" />

<form>
  <input type="text" class="input_text"
    id="search_term" />
  <input type="submit" class="input_submit"
    value="Add Search Term" />
</form>

<div id="twitter">
  <div class="template results">
    <h2>Search Results for
    <span class="search_term"></span></h2>
  </div>
</div>
```

Isso nos dá um container (`#twitter`) para nosso widget, um template para nossos resultados (escondido via CSS), e um formulário simples, onde usuários podem entrar com um termo de busca. (Para o bem da simplicidade, nós iremos assumir que a aplicação é somente em JavaScript e que nossos usuários sempre terão CSS.)

Há dois tipos de objetos que nós iremos atuar sobre: os containers de resultados e o container do Twitter.

Os containers de resultados são o coração da aplicação. Nós criaremos um plugin que preparará cada container de resultados assim que eles forem adicionados ao container do Twitter. Entre outras coisas, ele vinculará os eventos customizados para cada container e adicionará os botões de ação no topo a direita de cada container. Cada container de resultado terá os seguintes eventos customizados:

refresh	Coloca o container no estado “atualizando”, e dispara a requisição para obter os dados para o termo de busca.
populate	Recebe os dados retornados em JSON e usa-os para alimentar o container.
remove	Remove o container da página depois que o usuário efetua a requisição para fazê-lo. A verificação pode ser ignorada se <code>true</code> for passado como segundo argumento do manipulador de eventos. O evento remove também o termo associado com o container de resultados do objeto global contendo o termo de busca.
collapse	Adiciona a classe <code>collapsed</code> ao container, que esconderá os resultados via CSS. Ele também irá mudar o botão “Collapse” do container em um botão “Expand”.
expand	Remove a classe <code>collapsed</code> do container. Ele também irá mudar o botão “Expand” em um botão “Collapse”.

O plugin é responsável por adicionar os botões de ação ao container. Ele vincula cada evento de clique à cada item da lista de ações e usa a classe do item de lista para determinar qual evento customizado será disparado no container de resultados correspondente.

```
$.fn.twitterResult = function(settings) {
  return $(this).each(function() {
    var $results = $(this),
```

```
    $actions = $.fn.twitterResult.actions =
        $.fn.twitterResult.actions ||
        $.fn.twitterResult.createActions(),
    $a = $actions.clone().prependTo($results),
    term = settings.term;

    $results.find('span.search_term').text(term);

    $.each(
        ['refresh', 'populate', 'remove', 'collapse', 'expand'],
        function(i, ev) {
            $results.bind(
                ev,
                { term : term },
                $.fn.twitterResult.events[ev]
            );
        }
    );

    // usa a classe de cada ação para determinar qual
    // evento ele irá disparar no painel de resultados.
    $a.find('li').click(function() {
        // passa o li que foi clicado para a função
        // para que o mesmo possa ser manipulado se necessário
        $results.trigger($(this).attr('class'), [ $(this) ]);
    });
});

$.fn.twitterResult.createActions = function() {
    return $('<ul class="actions" />').append(
        '<li class="refresh">Refresh</li>' +
        '<li class="remove">Remove</li>' +
        '<li class="collapse">Collapse</li>'
    );
};

$.fn.twitterResult.events = {
    refresh : function(e) {
        // indica que os resultados estão sendo atualizados
        var $this = $(this).addClass('refreshing');

        $this.find('p.tweet').remove();
        $results.append('<p class="loading">Loading ...</p>');

        // obtém os dados do Twitter via jsonp
        $.getJSON(
            'http://search.twitter.com/search.json?q=' +
                escape(e.data.term) + '&rpp=5&callback=?',
            function(json) {
                $this.trigger('populate', [ json ]);
            }
        );
    },
```

```
populate : function(e, json) {
    var results = json.results;
    var $this = $(this);

    $this.find('p.loading').remove();

    $.each(results, function(i,result) {
        var tweet = '<p class="tweet">' +
            '<a href="http://twitter.com/' +
            result.from_user +
            '">' +
            result.from_user +
            '</a>: ' +
            result.text +
            ' <span class="date">' +
            result.created_at +
            '</span>' +
            '</p>';
        $this.append(tweet);
    });

    // indica que a atualização dos
    // resultados está terminada
    $this.removeClass('refreshing');
},

remove : function(e, force) {
    if (
        !force &&
        !confirm('Remove panel for term ' + e.data.term + '?')
    ) {
        return;
    }
    $(this).remove();

    // indica que nós não temos mais
    // um painel para o termo
    search_terms[e.data.term] = 0;
},

collapse : function(e) {
    $(this).find('li.collapse').removeClass('collapse')
        .addClass('expand').text('Expand');

    $(this).addClass('collapsed');
},

expand : function(e) {
    $(this).find('li.expand').removeClass('expand')
        .addClass('collapse').text('Collapse');

    $(this).removeClass('collapsed');
}
```

```
};
```

O container do Twitter terá somente dois eventos:

<code>getResults</code>	Recebe um termo de busca e checka se há um container de resultados para o termo; Se não, adiciona um container de resultados utilizando o template de resultados, configura o container utilizando o plugin <code>\$.fn.twitterResult</code> discutido acima, e então dispara o evento <code>refresh</code> no container de resultados para carregar de fato os resultados. Finalmente, ele armazena o termo de busca para que a aplicação não obtenha novamente o termo.
<code>getTrends</code>	Consulta o Twitter para obter os top 10 trending topics, depois itera sobre os resultados e dispara o evento <code>getResults</code> para cada um deles, adicionando assim um container de resultados para cada um dos termos.

As vinculações do container do Twitter vão ficar parecidas com isso:

```
$('#twitter')
    .bind('getResults', function(e, term) {
        // certifica que nós já não temos um box para este termo
        if (!search_terms[term]) {
            var $this = $(this);
            var $template = $this.find('div.template');

            // faz uma cópia da div do template
            // e insere-a como primeiro box de resultados
            $results = $template.clone().
                removeClass('template').
                insertBefore($this.find('div:first')).
                twitterResult({
                    'term' : term
                });

            // carrega o conteúdo utilizando o evento customizado
            // "refresh" que nós vinculamos ao container de resultados
            $results.trigger('refresh');
            search_terms[term] = 1;
        }
    })
    .bind('getTrends', function(e) {
        var $this = $(this);
        $.getJSON('http://search.twitter.com/trends.json?callback=?', function(json) {
            var trends = json.trends;
            $.each(trends, function(i, trend) {
                $this.trigger('getResults', [ trend.name ]);
            });
        });
    });
```

Até agora, nós escrevemos muito código que não faz quase nada, mas ok. Ao especificar todos os comportamentos que nós queremos que nossos objetos de base tenham, nós criamos um framework sólido para construir rapidamente a interface.

Vamos começar ligando nossa caixa de texto e o botão “Load Trending Terms”. Para a caixa de texto, nós iremos capturar o termo que foi digitado e iremos passá-lo para o evento `getResults` do container do Twitter. O evento `getTrends` será disparado ao clicarmos no botão “Load Trending Terms”:

```
$('#form').submit(function(e) {  
    e.preventDefault();  
    var term = $('#search_term').val();  
    $('#twitter').trigger('getResults', [ term ]);  
});  
  
$('#get_trends').click(function() {  
    $('#twitter').trigger('getTrends');  
});
```

Nós podemos remover, contrair, expandir e atualizar todos os resultados se adicionarmos alguns botões com IDs apropriados, assim como mostrado abaixo. Perceba como nós estamos passando o valor `true` para o manipulador de evento como seu segundo argumento, dizendo ao mesmo que nós não queremos verificar a remoção de containers individuais.

```
$.each(['refresh', 'expand', 'collapse'], function(i, ev) {  
    $('#'+ ev).click(function(e) { $('#twitter div.results').trigger(ev); });  
});  
  
$('#remove').click(function(e) {  
    if (confirm('Remove all results?')) {  
        $('#twitter div.results').trigger('remove', [ true ]);  
    }  
});
```

Conclusão

Eventos customizados oferecem uma nova forma de pensar sobre seu código: eles colocam ênfase no comportamento, não no elemento que o dispare. Se você gastar um tempo no início para estabelecer as partes da sua aplicação e os comportamentos que essas partes necessitam ter, os eventos customizados podem fornecer um método poderoso para você fazer com que estas partes "conversem", seja uma de cada vez ou uma conversa em massa. Uma vez que os comportamentos de uma parte foram descritos, torna-se trivial disparar estes comportamentos de qualquer parte, permitindo uma criação rápida da lógica da aplicação e que experimentos com opções da interface sejam triviais. Por último, eventos customizados podem melhorar a clareza do código e facilitar sua manutenção, pois eles deixam claro o relacionamento entre um elemento e seus comportamentos.

Você pode ver a aplicação completa em `demos/custom-events.html` e `demos/js/custom-events.js` no código de amostra.