**jQuery笔记**

何清 2016-8-8~

[jQuery 为什么如此出色 2](#_Toc458501851)

[$(document).ready() 2](#_Toc458501852)

[属性选择符 3](#_Toc458501853)

[自定义选择符 3](#_Toc458501854)

[基于上下文内容选择元素 4](#_Toc458501855)

[基于表单的选择符 4](#_Toc458501856)

[DOM 遍历方法 4](#_Toc458501857)

[.next() 和 .nextAll()，.prev()和 .prevAll()，.siblings()，.addBack() 5](#_Toc458501858)

[连缀 5](#_Toc458501859)

[访问 DOM 元素 5](#_Toc458501860)

[事件 6](#_Toc458501861)

[在页面加载后执行任务 6](#_Toc458501862)

[代码执行的时机选择 6](#_Toc458501863)

[基于一个页面执行多个脚本 7](#_Toc458501864)

[.ready() 的简写形式 7](#_Toc458501865)

[向 .ready() 回调函数中传入参数（避免$名称冲突） 8](#_Toc458501866)

[处理事件 8](#_Toc458501867)

[利用事件处理程序的上下文 9](#_Toc458501868)

[简写的事件 9](#_Toc458501869)

[事件捕获 9](#_Toc458501870)

[事件冒泡 10](#_Toc458501871)

[事件目标 10](#_Toc458501872)

[停止事件传播 11](#_Toc458501873)

[阻止默认操作 11](#_Toc458501874)

[事件委托 12](#_Toc458501875)

[模仿用户操作 12](#_Toc458501876)

[响应键盘事件 13](#_Toc458501877)

[样式与动画 13](#_Toc458501878)

[修改内联 CSS 13](#_Toc458501879)

[带厂商前缀的样式属性 14](#_Toc458501880)

[隐藏和显示元素.hide() 和 .show() 方法 15](#_Toc458501881)

[淡入和淡出 15](#_Toc458501882)

[滑上和滑下 15](#_Toc458501883)

## jQuery 为什么如此出色

1. **利用CSS的优势**。 通过将查找页面元素的机制构建于CSS选择符之上， jQuery继承了简明清晰地表达文档结构的方式。由于进行专业Web开发的一个必要条件是掌握CSS语法，因而jQuery成为希望向页面中添加行为的设计者们的切入点。
2. **抽象浏览器不一致性**。Web开发领域中一个令人遗憾的事实是，每种浏览器对颁布的标准都有自己的一套不太一致的实现方案。 任何Web应用程序中都会包含一个用于处理这些平台间特性差异的重要组成部分。虽然不断发展的浏览器前景，使得为某些高级特性提供浏览器中立的完美的基础代码（code base）变得不大可能，但jQuery添加一个抽象层来标准化常见的任务，从而有效地减少了代码量，同时，也极大地简化了这些任务。
3. **总是面向集合**。当我们指示jQuery“找到带有 collapsible 类的全部元素，然后隐藏它们”时，不需要循环遍历每一个返回的元素。相反， .hide() 之类的方法被设计成自动操作对象集合，而不是单独的对象。利用这种称作**隐式迭代（implicit iteration）**的技术，就可以抛弃那些臃肿的循环结构，从而大幅地减少代码量。
4. **将多重操作集于一行**。为了避免过度使用临时变量或不必要的代码重复，jQuery在其多数方法中采用了一种称作连缀（chaining）② 的编程模式。这种模式意味着基于一个对象进行的多数操作的结果，都会返回这个对象自身，以便为该对象应用下一次操作。
5. **支持扩展**。为了避免特性蠕变（feature creep）① ，jQuery将特殊情况下使用的工具归入插件当中。创建新插件的方法很简单，而且拥有完备的文档说明，这促进了大量有创意且有实用价值的模块的开发。甚至在下载的基本jQuery库文件当中，多数特性在内部都是通过插件架构实现的。而且，如有必要，可以移除这些内部插件，从而生成更小的库文件。

**提示：**

1. 由于JavaScript是一种解释型语言，所以不必担心编译和构建。什么时候需要使用jQuery，只要在HTML文档中使用 <script> 元素把它导入进来即可。
2. 为了确保在现代浏览器中速度更快，代码更简洁，jQuery从2.0版开始**不再支持IE6、IE7和IE8。**jQuery开发团队知道，支持这些老版本浏览器也很重要。正因为如此，该团队还会继续维护jQuery 1.x版。
3. 这里要注意的是，引用jQuery库文件的 <script> 标签，必须放在引用自定义脚本文件的 <script> 标签之前。否则，在我们编写的代码中将引用不到jQuery框架。
4. $() 函数会返回一个新的jQuery对象实例。

## 重要链接

1. 这里有所有选择符的介绍： <http://api.jquery.com/category/selectors/>。
2. 这里有遍历方法的介绍： <http://api.jquery.com/category/traversing>。
3. 基准测试工具：<http://jsperf.com/>。

## $(document).ready()

$(document).ready() 方法， jQuery支持我们预定在DOM加载完毕后调用某个函数， **而不必等待页面中的图像加载**。 尽管不使用jQuery， 也可以做到这种预定， 但 $(document).

ready() 为我们提供了很好的跨浏览器解决方案，涉及如下功能：

1. 尽可能使用浏览器原生的DOM就绪实现， 并以 window.onload 事件处理程序作为后备；
2. 可以多次调用 $(document).ready() 并按照调用它们的顺序执行；
3. 即便是在浏览器事件发生之后把函数传给 $(document).ready() ， 这些函数也会执行；
4. 异步处理事件的预定，必要时脚本可以延迟执行；
5. 通过重复检查一个几乎与DOM同时可用的方法，在较早版本的浏览器中模拟DOM就绪事件。

**提示：**

1. 让jQuery与其他JavaScript库协同工作

在jQuery中， 美元符号 $ 只不过标识符 jQuery 的 “别名” 。 由于 $() 在JavaScript

库中很常见， 所以， 如果在一个页面中使用了几个这样的库， 那么就会导致冲突。

在这种情况下，可以在我们自定义的jQuery代码中，通过将每个 $ 的实例替换成

jQuery 来避免这种冲突。

## 属性选择符^,$,\*，!

属性选择符使用一种从正则表达式中借鉴来的通配符语法， 以 ^ 表示值在字符串的开始， 以 $表示值在字符串的结尾。而且，也是用星号 \* 表示要匹配的值可以出现在字符串中的任意位置，用叹号 ! 表示对值取反。

## 自定义选择符

自定义选择符通常跟在一个CSS选择符后面，基于已经选择的元素集的位置来查找元素。自定义选择符的语法与CSS中的伪类选择符语法相同，即选择符以**冒号（ : ）开头**。

$('div.horizontal:eq(1)')

**注意：**

注意， 因为**JavaScript数组采用从0开始的编号方式**， 所以 eq(1) 取得的是集合中的第2个元素。**而CSS则是从1开始的**，因此CSS选择符 **$('div:nth-child(1)')** 取得的是作为其父元素第1个子元素的所有 div 元素。如果记不清哪个从0开始，哪个从1开始，可以参考jQuery API文档：<http://api.jquery.com/category/selectors/>。

**:eq()** 选择符、 **:odd** 和 **:even** 选择符都使用JavaScript内置从0开始的编号方式。

**$('tr:even').addClass('alt');**与**$('tr:nth-child(odd)').addClass('alt');**的**区别：**

前者是根据文档中当前选择的所有元素来统一计算位置，而后者是相对于**元素的父元素**而非当前选择的所有元素来计算位置，它可以接受数值、 odd 或 even 作为参数。值得一提的是， :nth-child() 是jQuery中**唯一从1开始计数的选择符**。

## 基于上下文内容选择元素

我们介绍最后一个自定义选择符。假设出于某种原因，我们希望突出显示提到任何一

种Henry游戏的所有表格单元。为此，我们所要做的就是在样式表中添加一个声明了粗体和斜体文本的类（ .highlight {font-weight:bold; font-style:italic; } ） ，然后向jQuery代码中添加一行代码，其中使用的是 :contains() 选择符。

$('td:contains(Henry)').addClass('highlight');

**必须注意**， :contains() 选择符**区分大小写**。 换句话说， 使用不带大写 “H”的 $('td:contains(henry)') ，不会选择任何单元格。

## 基于表单的选择符

|  |  |
| --- | --- |
| 表单选择符 | 选 择 符 匹 配 |
| :input | 输入字段、文本区、选择列表和按钮元素 |
| :button | 按钮元素或type属性值为button的输入元素 |
| :enabled | 启用的表单元素 |
| :disabled | 禁用的表单元素 |
| :checked | 勾选的单选按钮或复选框 |
| :selected | 选择的选项元素 |

**例如**，使用 $('input[type="radio"]:checked') 可以选择所有选中的单选按钮 （而不是复选框） ， 而使用 $('input[type="password"],input[type="text"]:disabled') 则可以选择所有密码输入字段和禁用的文本输入字段。

## DOM 遍历方法（.filter()函数）

**.filter()函数**

用于添加 alt 类的代码 $('tr:even').addClass('alt'); ，可以通过 .filter() 方法重写成下面这样：$('tr').filter(':even').addClass('alt');

.filter() 的功能也十分强大，因为它可以接受函数参数。通过传入的函数，可以执行复杂的测试，以决定相应元素是否应该保留在匹配的集合中。例如，假设我们要为所有外部链接添加一个类。

**$('a').filter(function() {**

**return this.hostname && this.hostname != location.hostname;**

**}).addClass('external');**

第二行代码可以筛选出符合下面两个条件的 <a> 元素。

1. 必须包含一个带有域名（ this.hostname ）的 href 属性。这个测试可以排除 mailto 及类似链接。
2. 链接指向的域名（还是 this.hostname ）必须不等于（ != ）页面当前所在域的名称（ location.hostname ） 。更准确地说， .filter() 方法会迭代所有匹配的元素， 对每个元素都调用传入的函数并测试函数的返回值。如果函数返回 false ，则从匹配集合中删除相应元素；如果返回 true ，则保留相应元素。

## .next() 和 .nextAll()，.prev()和 .prevAll()，.siblings()，.addBack()

.next() 方法只选择下一个最接近的同辈元素。

.siblings() 能够选择处于相同DOM层次的所有其他元素，无论这些元素处于当前元素之前还是之后。

要在这些单元格中再包含原来的单元格 （即包含Henry的那个单元格） ， 可以添加 .addBack()方法，例如：  
$(document).ready(function() {

$('td:contains(Henry)').nextAll().addBack()

.addClass('highlight');

});

## 连缀

$('td:contains(Henry)') //取得包含Henry的所有单元格

.parent() //取得它的父元素

.find('td:eq(1)') //在父元素中查找第2个单元格

.addClass('highlight') //为该单元格添加hightlight类

.end() //恢复到包含Henry的单元格的父元素

.find('td:eq(2)') //在父元素中查找第3个单元格

.addClass('highlight'); //为该单元格添加hightlight类

不可否认，这个例子中展示的迂回曲折的DOM遍历过程几近荒谬。我们当然不建议读者使用如此复杂的连缀方式，因为还有更简单、更直接的方法。这个例子的用意只是演示一下连缀为我们带来的极大灵活性。

## 访问 DOM 元素

所有选择符表达式和多数jQuery方法都返回一个jQuery对象，而这通常都是我们所希望的，因为jQuery对象能够提供隐式迭代和连缀能力。尽管如此，我们仍然有可能需要在代码中直接访问DOM元素。例如，可能需要为另一个JavaScript库提供一组元素的结果集合。或者可能不得不访问某个元素的标签名——通过DOM元素的属性。对于这些少见但合理的情形，jQuery提供了 .get() 方法。要访问jQuery对象引用的第一个DOM元素，可以使用 .get(0) 。因而，如果想知道带有 id="my-element" 属性的元素的标签名，应该使用如下代码：

var myTag = $('#my-element').get(0).tagName;

为了进一步简化这些代码，jQuery还为 .get() 方法提供了一种简写方式。比如，可以将

$('#my-element').get(0) 简写为：

var myTag = $('#my-element')[0].tagName;

## 事件

### 在页面加载后执行任务

我们已经看到如何让jQuery响应网页的加载事件， $(document).ready() 事件处理程序可以用来触发函数中的代码，但对这个过程还有待深入分析。

### 代码执行的时机选择

我们知道了 $(document).ready() 是jQuery基于页面加载执行任务的一种主要方式。但这并不是唯一的方式，原生的 window.onload 事件也可以实现相同的效果。虽然这两个方法具有类似的效果，但是，它们在触发操作的时间上存在着微妙的差异，这种差异只有在加

载的资源多到一定程度时才会体现出来。当文档完全下载到浏览器中时，会触发 window.onload 事件。这意味着页面上的全部元素对JavaScript而言都是可以操作的， 这种情况对编写功能性的代码非常有利， 因为无需考虑加载的次序。

另一方面， 通过 $(document).ready() 注册的事件处理程序， 则会在DOM完全就绪并可以使用时调用。虽然这也意味着所有元素对脚本而言都是可以访问的，但是，却不意味着所有关联的文件都已经下载完毕。**换句话说，当HTML下载完成并解析为DOM树之后，代码就可以运行。**

**提示：**

加载样式与执行代码

为了保证JavaScript代码执行以前页面已经应用了样式，最好是在 <head>元素中把 <link rel="stylesheet"> 标签和 <style> 标签放在 <script> 标签前面。

**什么是加载完成**

一般来说， 使用 $(document).ready() 要优于使用 onload 事件处理程序，但必须要明确的一点是，因为支持文件可能还没有加载完成，所以类似图像的高度和宽度这样的属性此时则不一定会有效。如果需要访问这些属性，可能就得选择实现一个 onload 事件处理程序（或者是使用jQuery为 load 事件设置处理程序） 。这两种机制能够和平共存。

### 基于一个页面执行多个脚本

通过JavaScript（而不是指直接在HTML中添加处理程序属性）注册事件处理程序的传统机制是，把一个函数指定给DOM元素的对应属性。例如，假设我们已经定义了如下函数：

function doStuff() {

//执行某种任务……

}

那么，我们既可以在HTML标记中指定该函数：

<body onload="doStuff();">

也可以在JavaScript代码中指定该函数：

window.onload = doStuff;

这两种方式都会在页面加载完成后执行这个函数。但第2种方式的优点在于，它能使行为更清晰地从标记中分离出来。

**引用函数与调用函数**

这里在将函数指定为处理程序时，省略了后面的圆括号，只使用了函数名。**如果带着圆括号，函数会被立即调用**；**没有圆括号，函数名就只是函数的标识符或函数引用，可以用于在将来再调用函数。**

在只有一个函数的情况下，这样做没有什么问题。但是，假设我们又定义了第二个函数：

function doOtherStuff() {

//执行另外一种任务……

}

我们也可以将它指定为基于页面的加载来运行：

window.onload = doOtherStuff;

然而，这次指定的函数**会取代刚才指定的第一个函数**。因为 **.onload 属性一次只能保存对一个函数的引用，所以不能在现有的行为基础上再增加新行为。**

通过 $(document).ready() 机制能够很好地解决这个问题。每次调用这个方法都会向内

部的**行为队列中添加一个新函数**，当页面加载完成后，所有函数都会被执行。而且，这些函数会按照注册它们的**顺序依次执行**。

### .ready() 的简写形式

前面提到的 $(document).ready() 结构，实际上是在基于 document 这个DOM元素构建而成的jQuery对象上调用了 .ready() 方法。 $() 函数为我们提供了一种简写方式。当给它传递一个函数作为参数时，jQuery会隐式调用 .ready() 。也就是说，对于：

$(document).ready(function() {

//这里是代码……

});

也可以简写成：

$(function() {

//这里是代码……

});

虽然这种语法更短一些，但作者推荐使用较长的形式，因为较长的形式能够更清楚地表明代码在做什么。

### 向 .ready() 回调函数中传入参数（避免$名称冲突）

在某些情况下，可能有必要在同一个页面中使用多个JavaScript库。由于很多库都使用 $ 标识符（因为它简短方便） ，因此就需要一种方式来避免名称冲突。为解决这个问题，jQuery提供了一个 jQuery.noConflict() 方法，调用该方法可以把对 $标识符的控制权让渡还给其他库。使用 jQuery.noConflict() 方法的一般模式如下：

<script src="prototype.js"></script>

<script src="jquery.js"></script>

<script>

jQuery.noConflict();

</script>

<script src="myscript.js"></script>

首先，包含jQuery之外的库（这里是Prototype） 。然后，包含jQuery库，取得对 $ 的使用权。接着，调用 .noConflict() 方法让出 $ ，以便将控制权交还给最先包含的库（Prototype） 。这样就可以在自定义脚本中使用两个库了——但是， 在需要使用jQuery方法时， **必须记住要用 jQuery而不是 $ 来调用**。

在这种情况下， 还有一个在 .ready() 方法中使用 $ 的技巧。 我们传递给它的回调函数可以接收一个参数——jQuery对象本身。利用这个参数，可以重新命名 jQuery 为 $ ，而不必担心造成冲突，如下面的代码所示：

jQuery(document).ready(function($) {

//在这里，可以正常使用!

});

或者，也可以使用刚刚介绍的简写语法：

jQuery(function($) {

//使用$的代码

});

### 处理事件

我们需要引入 **.on() 方法**。通过这个方法，可以**指定任何DOM事件**，并为该事件添加一种行为。多次调用 .on() 也没有任何问题，即可以按需为同一个事件追加更多的行为。

### 利用事件处理程序的上下文

当触发任何事件处理程序时，**关键字 this 引用的都是携带相应行为的DOM元素**。前面我们谈到过， $() 函数可以将DOM元素作为参数，而 this 关键字是实现这个功能的关键。通过在事件处理程序中使用 $(this) ，可以为相应的元素创建jQuery对象，然后就如同使用CSS选择符找到该元素一样对它进行操作。

**注意：**

上下文关键字**this** 引用的是**DOM元素**，而**不是jQuery对象。**

### 简写的事件

鉴于为某个事件（例如简单的单击事件）绑定处理程序极为常用，jQuery提供了一种简化事件操作的方式——简写事件方法，简写事件方法的原理与对应的 .on() 调用相同，可以减少一定的代码输入量。

例如，不使用 .on() 而使用 .click() 可以将前面的样式转换器程序重写为如下代码。

$(document).ready(function() {

$('#switcher-default').addClass('selected');

$('#switcher button').**click**(function() {

var bodyClass = this.id.split('-')[1];

$('body').removeClass().addClass(bodyClass);

$('#switcher button').removeClass('selected');

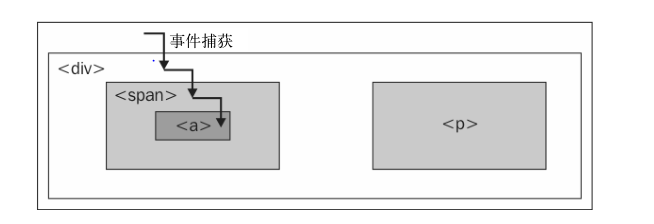
$(this).addClass('selected');

});

});

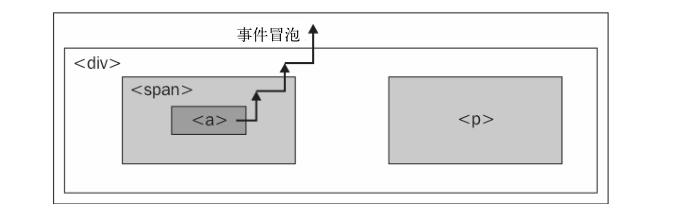
### 事件捕获

允许多个元素响应单击事件的一种策略叫做**事件捕获**。在事件捕获的过程中，事件首先会交给最外层的元素，接着再交给更具体的元素。



### 事件冒泡

另一种相反的策略叫做事件冒泡。即当事件发生时，会首先发送给最具体的元素，在这个元素获得响应机会之后，事件会向上冒泡到更一般的元素。



毫不奇怪， 不同的浏览器开发者最初采用的是不同的事件传播模型。 因而， **最终出台的DOM标准规定应该同时使用这两种策略**：首先，事件要从一般元素到具体元素逐层捕获，然后，事件再通过冒泡返回DOM树的顶层。而事件处理程序可以注册到这个过程中的任何一个阶段。为了确保跨浏览器的一致性，而且也为了让人容易理解，jQuery始终会在模型的冒泡阶段注册事件处理程序。因此，我们总是可以假定最具体的元素会首先获得响应事件的机会。

### 事件目标

现在，事件处理程序中的变量 event 保存着事件对象。而 event.target 属性保存着发生事件的目标元素。这个属性是DOM API中规定的，但是没有在某些旧版本的浏览器中实现。jQuery对这个事件对象进行了必要的扩展，从而在任何浏览器中都能够使用这个属性。通过 .target ，可以确定DOM中首先接收到事件的元素（即实际被单击的元素） 。而且，我们知道 this 引用的是处理事件的DOM元素，所以可以编写出代码清单3-12。

$(document).ready(function() {

$('#switcher').click(function(**event**) {

**if (event.target == this) {**

$('#switcher button').toggleClass('hidden');

}

});

});

### 停止事件传播

事件对象还提供了一个 .**stopPropagation**() 方法，该方法可以完全阻止事件冒泡。与 .target 类似，这个方法也是一种基本的DOM特性，但在IE8及更早版本中则无法安全地使用 。不过，只要我们通过jQuery来注册所有的事件处理程序，就可以放心地使用这个方法。

$(document).ready(function() {

$('#switcher-default').addClass('selected');

$('#switcher button').click(function(event) {

var bodyClass = this.id.split('-')[1];

$('body').removeClass().addClass(bodyClass);

$('#switcher button').removeClass('selected');

$(this).addClass('selected');

**event.stopPropagation();**

});

});

### 阻止默认操作

如果我们把单击事件处理程序注册到锚元素（ <a> ） ，而不是外层的 <div> 上，那么就要面对另外一个问题：当用户单击链接时，浏览器会加载一个新页面。这种行为与我们讨论的事件处理程序不是同一个概念，它是单击锚元素的默认操作。类似地，当用户在编辑完表单后按下回车键时，会触发表单的 submit 事件，在此事件发生后，表单提交才会真正发生。

即便在事件对象上**调用 .stopPropagation() 方法也*不能禁止*这种默认操作**，因为**默认操**

**作*不是*在正常的事件传播流中发生的。在这种情况下**， **.preventDefault()** 方法则可以在触发默认操作之前**终止**事件。

**事件传播和默认操作是相互独立的两套机制，在二者任何一方发生时，都可以终止另一方。如果想要同时停止事件传播和默认操作，可以在事件处理程序中返回 false ，这是对在事件对象上同时调用 .stopPropagation() 和 .preventDefault() 的一种简写方式。**

### 事件委托

事件冒泡并不总是带来问题，也可以利用它为我们带来好处。**事件委托就是利用冒泡的一项高级技术。**通过事件委托，可以借助一个元素上的事件处理程序完成很多工作。

在我前面的例子中，只有3个 <div class="button"> 元素注册了单击处理程序。假如我们想为更多元素注册处理程序怎么办？这种情况比我们想象的更常见。例如，有一个显示信息的大型表格，每一行都有一项需要注册单击处理程序。虽然不难通过隐式迭代来指定所有单击处理程序，但**性能可能会很成问题，因为循环是由jQuery在内部完成的，而且要维护所有处理程序也需要占用很多内存。**

为解决这个问题，可以只在DOM中的一个祖先元素上指定一个单击处理程序。**由于事件会冒泡，未遭拦截的单击事件最终会到达这个祖先元素，而我们可以在此时再作出相应处理。**下面我们就以样式转换器为例（尽管其中的按钮数量还不至于使用这种方法） ，说明如何使用这种技术。从代码清单3-12中可以看到，当发生单击事件时，可以使用 event.target 属性检查鼠标指针下方是什么元素。下面是代码清单3-14。

$(document).ready(function() {

$('#switcher').click(function(event) {

if ($(event.target).is('button')) {

var bodyClass = event.target.id.split('-')[1];

$('body').removeClass().addClass(bodyClass);

$('#switcher button').removeClass('selected');

$(event.target).addClass('selected');

event.stopPropagation();

}

});

});

### 模仿用户操作

通过 .trigger() 方法就可以完成模拟事件的操作。

$(document).ready(function() {

$('#switcher').trigger('click');

});

.trigger() 方法提供了一组与 .on() 方法相同的简写方法。 当使用这些方法而不带参数时，结果将是触发操作而不是绑定行为，如：

$(document).ready(function() {

$('#switcher').click();

});

### 响应键盘事件

键盘事件可以分为两类：直接对键盘按键给出响应的事件（ keyup 和 keydown ）和对文本输入给出响应的事件（ keypress ） 。输入一个字母的事件可能会对应着几个按键，例如输入大写的X要同时按Shift和X键。虽然各种浏览器的具体实现有所不同（毫不奇怪） ，但有一条实践经验还是比较可靠的：如果想知道用户按了哪个键，应该侦听 keyup 或 keydown 事件；如果想知道用户输入的是什么字符，应该侦听 keypress 事件。

## 样式与动画

### 修改内联 CSS

jQuery提供了 **.css()** 方法。这个方法集getter（获取方法）和setter（设置方法）于一身。为取得某个样式属性的值，可以为这个方法传递一个字符串形式的属性名，然后同样得到一个字符串形式的属性值。要取得多个样式属性的值，可以传入属性名的数组，得到的则是属性和值构成的对象。对于backgroundColor 这样由多个单词构成的属性名， **jQuery既可以解释连字符版的CSS表示法 （如background-color ） ，也可以解释驼峰大小写形式（camel-cased）的DOM表示法（如backgroundColor ）** 。

//取得单个属性的值

.css('property')

//返回"value"

//取得多个属性的值

.css(['property1', 'property-2'])

//返回{"property1": "value1", "property-2": "value2"}

在设置样式属性时， .css() 方法能够接受的参数有两种，一种是为它传递一个单独的样式属性和值，另一种是为它传递一个由属性 — 值对构成的对象：

//单个属性及其值

.css('property', 'value')

//属性值对构成的对象

.css({

property1: 'value1',

'property-2': 'value2'

})

这些键值的集合叫**对象字面量**，是在代码中直接创建的JavaScript对象。

**对象字面量**

一般来说，数字值不需要加引号而字符串值需要加引号。由于属性名是字符串，所以属性通常是需要加引号的。但是，如果对象字面量中的属性名是有效的JavaScript标识符，比如使用驼峰大小写形式的DOM表示法时，则可以省略引号。

**注意：**

通过 $('div.speech').css('fontSize') 可以轻而易举地取得当前的字体大小。不过，由于返回的值中包含数字值及其单位（ px ） ，需要去掉单位部分才能执行计算。

例如：

$(document).ready(function() {

var $speech = $('div.speech');

$('#switcher-large').click(function() {

var num = parseFloat($speech.css('fontSize'));

});

});

**parseFloat（）函数用法：**

通过 parseFloat() 函数只取得字体大小属性中的数值部分。parseFloat() 函数会在一个字符串中从左到右地查找一个浮点（十进制）数。例如，它会将字符串 '12' 转换成数字 12 。另外，它还会去掉末尾的非数字字符，因此 '12px' 就变成了 12 。如果字符串本身以一个非数字开头，那么 parseFloat() 会返回 NaN ，即Not a Number（非数字） 。

### 带厂商前缀的样式属性

浏览器厂商在引入试验性的样式属性时，通常会在实现达到CSS规范要求之前，在属性名前面添加一个前缀。等到实现和规范都稳定之后，这些属性的前缀就会被去掉，让开发人员使用标准的名称。因此，我们经常会在样式表里看到一些类似下面这样的CSS声明：

-webkit-property-name: value;

-moz-property-name: value;

-ms-property-name: value;

-o-property-name: value;

property-name: value;

如果想在JavaScript中设置这些属性，需要提前检测它们在DOM中是否存在，从

propertName 到 WebkitPropertyName ，再到 msPropertyName ……都要检测。**但在jQuery中，我们可以直接使用标准的属性名**，比如： .css('propertyName', 'value') 。**如果样式对象**

**中不存在这个属性，jQuery就会依次检测所有带前缀（ Webkit 、 O 、 Moz 、 ms ）的属性，然后使用第一个找到的那个属性。**

### 隐藏和显示元素.hide() 和 .show() 方法

基本的 .hide() 和 .show() 方法不带任何参数。可以把它们想象成类似 .css('display','string') 方法的简写方式，其中 string 是适当的显示值。不错，这两个方法的作用就是立即隐藏或显示匹配的元素集合，不带任何动画效果。

其中， .hide() 方法会将匹配的元素集合的内联 style 属性设置为 display:none 。但它的**聪明之处是**，它能够在把 display 的值变成 none 之前，记住原先的 display 值，通常是 block 、inline 或 inline-block 。恰好相反， .show() 方法会将匹配的元素集合的 display 属性，恢复为应用 display: none 之前的可见属性。

.show() 和 .hide() 的这种特性， 使得它们非常适合隐藏那些默认的 display 属性在样式表中被修改的元素。

当在 .show() 或 .hide() 中指定时长（或更准确地说，一个速度）参数时，就会产生动画效果，即效果会在一个特定的时间段内发生。例如 .hide('duration') 方法，会同时减少元素的高度、宽度和不透明度，直至这3个属性的值都达到0，与此同时会为该元素应用CSS规则display:none 。而 .show(' duration ') 方法则会从上到下增大元素的高度，从左到右增大元素的宽度，同时从0到1增加元素的不透明度，直至其内容完全可见。

对于jQuery提供的任何效果，都可以指定两种预设的速度参数： 'slow' 和 'fast' 。使

用 .show('slow') 会在600毫秒（0.6秒）内完成效果，而 .show('fast') 则是200毫秒（0.2秒） 。如果传入的是其他字符串，jQuery就会在默认的400毫秒内完成效果。要指定更精确的速度，可以使用毫秒数值，例如 **.show(850)** 。注意，与字符串表示的速度参数名称不同，数**值不需要使用引号**。

### 淡入和淡出

如果想在显示整个段落时，只是逐渐地增大其不透明度，那么可以使用 .fadeIn('slow') 方法。.fadeIn() 会在一开始设置段落的尺寸， 以便内容能够逐渐显示出来。类似地，要逐渐减少不透明度，可以使用 .fadeOut() 。

### 滑上和滑下

对于本来就处于文档流之外的元素，比较适合使用淡入和淡出动画。例如，对于那些覆盖在页面之上的“亮盒”元素来说，采用淡入和淡出就显得很自然。不过，假如某个元素本来就处在文档流中，那再调用 .fadeIn() 就会导致文档“跳一下” ，以便为新元素腾出地方来。但这种跳跃感在用户眼里就不总是那么美观了。

此时，使用jQuery的 .slideDown() 和 .slideUp() 方法通常是正确的选择。这两个动画方法仅改变元素的高度。

### 创建自定义动画

除了预置的效果方法外，jQuery还提供了一个强大的 .animate() 方法，用于创建控制更加精细的自定义动画。 .animate() 方法有两种形式：

* 第一种形式接收以下4个参数。

1. 一个包含样式属性及值的对象：与本章前面讨论的 .css() 方法中的参数类似。
2. 可选的时长参数：既可以是预置的字符串，也可以是毫秒数值。
3. 可选的缓动（easing）类型：现在我们先不介绍，这是第11章中将要讨论的一个高级选项。
4. 可选的回调函数：将在本章后面讨论。

把这4个参数放到一起，结果如下所示：

.animate({property1: 'value1', property2: 'value2'},

duration, easing, function() {

alert('The animation is finished.');

}

);

* 第二种形式接受两个参数，一个属性对象和一个选项对象：

.animate({properties}, {options})

实际上，这里的第二个参数是把第一种形式的第2～4个参数封装在了另一个对象中，同时又添加了两个选项。考虑到可读性并调整了换行之后，调用第二种形式的方法的代码如下：

.animate({

property1: 'value1',

property2: 'value2'

}, {

duration: 'value',

easing: 'value',

specialEasing: {

property1: 'easing1',

property2: 'easing2'

},

complete: function() {

alert('The animation is finished.');

},

queue: true,

step: callback

});

### 并发与排队效果

* 越过队列

不过，要是想在这个 <div> 不透明度减退至一半的同时，把它移动到右侧应该怎么办呢如果两个动画以相同速度执行，则可以简单地把它们组合到一个 .animate() 方法中。但这个例子中的 .fadeTo() 使用的速度字符串是 'fast' ， 而向右移动的动画使用的速度字符串是 'slow' 。在这种情况下，第二种形式的 .animate() 方法又可以派上用场了。

$(document).ready(function() {

$('div.label').click(function() {

var paraWidth = $('div.speech p').outerWidth();

var $switcher = $(this).parent();

var switcherWidth = $switcher.outerWidth();

$switcher

.css({position: 'relative'})

.fadeTo('fast', 0.5)

.animate({

left: paraWidth - switcherWidth

}, {

duration: 'slow',

queue: false

})

.fadeTo('slow', 1.0)

.slideUp('slow')

.slideDown('slow');

});

});

第二个参数（即选项对象）包含了 queue 选项，把该选项设置为 false 即可让当前动画与前一个动画同时开始。

* 手工队列

有关为一组元素应用排队效果的最后一个需要注意的问题， 就是**排队不能自动应用到其他的非效果方法**，如 .css() 。下面，假设我们想在 .slideUp() 执行后但在 .slideDown() 执行前，

把 <div id="switcher"> 的背景颜色修改为红色，可以尝试像代码清单4-21这样来做。

//未完成的代码

$(document).ready(function() {

$('div.label').click(function() {

var paraWidth = $('div.speech p').outerWidth();

var $switcher = $(this).parent();

var switcherWidth = $switcher.outerWidth();

$switcher

.css({position: 'relative'})

.fadeTo('fast', 0.5)

.animate({

left: paraWidth - switcherWidth

}, {

duration: 'slow',

queue: false

})

.fadeTo('slow', 1.0)

.slideUp('slow')

.css({backgroundColor: '#f00'})

.slideDown('slow');

});

});

然而， 即使把修改背景颜色的代码放在连缀序列中正确的位置上， 它也会在单击后**立即执行**。把非效果方法添加到队列中的一种方式，**就是使用 .queue() 方法**。代码清单4-22就是在这个例子中使用 .queue() 方法的代码片段。

$(document).ready(function() {

$('div.label').click(function() {

var paraWidth = $('div.speech p').outerWidth();

var $switcher = $(this).parent();

var switcherWidth = $switcher.outerWidth();

$switcher

.css({position: 'relative'})

.fadeTo('fast', 0.5)

.animate({

left: paraWidth - switcherWidth

}, {

duration: 'slow',

queue: false

})

.fadeTo('slow', 1.0)

.slideUp('slow')

**.queue(function(next) {**

**$switcher.css({backgroundColor: '#f00'});**

**next();**

**})**

.slideDown('slow');

});

});

像这样传递一个回调函数， .queue() 方法就可以把该函数添加到相应元素的效果队列中。在这个函数内部，我们把背景颜色设置为红色，然后又调用了 next() 方法，其返回的结果将作为参数传给回调函数。**添加的这个 next () 方法可以让队列在中断的地方再接续起来，然后再与后续的 .slideDown ('slow') 连缀在一起**。如果在此**不使用 next() 方法，动画就会中断。**

### 概括

1. 一组元素上的效果：

* 当在一个.animate()方法中以多个属性的方式应用时，是同时发生的；
* 当以方法连缀的形式应用时，是按顺序发生的（排队效果）——除非queue选项值为false。

(2) 多组元素上的效果：

* 默认情况下是同时发生的；
* 当在另一个效果方法或者在 .queue() 方法的回调函数中应用时，是按顺序发生的（排队效果） 。

**例如：**

$(document).ready(function() {

$('div.label').click(function() {

var paraWidth = $('div.speech p').outerWidth();

var $switcher = $(this).parent();

var switcherWidth = $switcher.outerWidth();

$switcher

.css({

position: 'relative'

})

.fadeTo('fast', 0.5)

.animate({

left: paraWidth - switcherWidth

}, {

duration: 'slow',

**queue: false**

})

.fadeTo('slow', 1.0)

.slideUp('slow')

.queue(function(next) {

$switcher.css({

backgroundColor: '#f00'

});

**next();**

})

.slideDown('slow');

});

});

**可改为：**

$(document).ready(function() {

$('div.label').click(function() {

var paraWidth = $('div.speech p').outerWidth();

var $switcher = $(this).parent();

var switcherWidth = $switcher.outerWidth();

$switcher

.css({

position: 'relative'

})

.fadeTo('fast', 0.5)

.animate({

left: paraWidth - switcherWidth

}, {

duration: 'slow',

**queue: false**

})

.fadeTo('slow', 1.0)

.slideUp('slow', **function() {**

**$switcher.css({**

**backgroundColor: '#f00'**

**});**

})

.slideDown('slow');

});

});

### 值回调

值回调其实就是给参数传入一个函数，而不是传入具体的值。这个函数会针对匹配的元素集中的每个元素都调用一次，调用后的返回值将作为属性的值。例如，可以使用值回调来为每个元素生成唯一的 id 值。例如：

$(document).ready(function() {

$('div.chapter a').attr({

rel: 'external',

title: 'Learn more at Wikipedia',

**id: function(index, oldValue) {**

**return 'wikilink-' + index;**

**}**

});

});

每次触发值回调，都会给它传入两个参数。第一个是一个整数，表示**迭代次数**，我们在此利用这个值为第一个链接生成的 id 是 wikilink-0 ，为第二个链接生成的 id 是 wikilink-1 ，以此类推。代码清单5-3并没有用到第二个参数，这个参数中保存的是**修改之前属性的值。**

## 操作DOM

### 表单控件的值

HTML属性与DOM属性差别最大的地方，恐怕就要数表单控件的值了。比如，文本输入框的value 属性在DOM中的属性叫 defaultValue ，DOM中就没有 value 属性。而选项列表（ select ）元素呢，其选项的值在DOM中通常是通过 selectedIndex 属性，或者通过其选项元素的selected 属性来取得。

由于存在这些差异，在取得和设置表单控件的值时，最好不要使用 .attr() 方法。而对于选项列表呢，最好连 .prop() 方法也不要使用。那使用什么呢，建议使用jQuery提供的 .val()方法：

//取得文本输入框的当前值

var inputValue = $('#my-input').val();

//取得选项列表的当前值

var selectValue = $('#my-select').val();

//设置单选列表的值

$('#my-single-select').val('value3');

//设置多选列表的值

$('#my-multi-select').val(['value1', 'value2']);

与 .attr() 和 .prop() 一样， .val() 方法也可以接受一个函数作为其setter参数。有了这个多用途的 .val() 方法，使用jQuery做Web开发你又会倍感高效。

### .insertBefore()，.prependTo()，.appendTo()，.insertAfter()方法：

1. .insertBefore() 在现有元素外部、之前添加内容；
2. .prependTo() 在现有元素内部、之前添加内容；
3. .appendTo() 在现有元素内部、之后添加内容；
4. .insertAfter() 在现有元素外部、之后添加内容。

### 显示迭代（.each）

.each() 方法就是一个显式迭代器，与最近加入JavaScript语言中的数组迭代器forEach() 非常相似。 如果在使用隐式迭代的情况下， 我们想为每个匹配的元素应用的代码显得太过复杂，就可以转而使用 .each() 。这个方法接受一个回调函数，这个函数会针对匹配的元素集中的每个元素都调用一次.

### 复制元素

在复制元素时，需要使用jQuery的 .clone() 方法，这个方法能够创建任何匹配的元素集合的副本以便将来使用。与本章前面使用 $() 创建元素时一样，在为复制的元素应用一种插入方法之前，这些元素不会出现在文档中。

例如，下面这行代码将创建 <div class="chapter"> 中第一段落的副本：

$('div.chapter p:eq(0)').clone();

但仅创建副本还不足以改变页面的内容。要想让复制的内容显示在网页中，可以使用插入方法将其放到 <div class="chapter"> 前面 。

$('div.chapter p:eq(0)').clone().insertBefore('div.chapter');

### 连同事件一起复制

在默认情况下， .clone() 方法不会复制匹配的元素或其后代元素中绑定的事件。不过，可以为这个方法传递一个布尔值参数，将这个**参数设置为 true** ，就可以连同事件一起复制，即 .clone(true) 。这样一来，就可以避免每次复制之后还要手工重新绑定事件的麻烦

### .html() 与 .text()的区别

与 .html() 方法类似， .text() 也可以取得匹配元素的内容，或者用新字符串替换匹配元素的内容。但是，与 .html() 不同的是， .text() 始终会取得或设置纯文本内容。在使用 .text()取得内容时，所有HTML标签都将被忽略，而所有HTML实体也会被转换成对应的字符。而在通过它设置内容时，诸如 < 这样的特殊字符，都会被转换成等价的HTML实体。

**例如：**

<span class="pull-quote">It is a Law of Nature

<span class="drop">with us</span> that a male child shall have

<strong>one more side</strong> than his father</span>

调用 $(‘span. pull-quote’).html()得到的内容是：（包含了标签）

It is a Law of Nature <span class="drop">with us</span> that a male child shall have <strong>one more side</strong> than his father

调用 $(‘span. pull-quote’).text()得到的内容是：（没有包含了标签）

It is a Law of Nature with us that a male child shall have one more side than his father

### DOM 操作方法的简单归纳

(1) 要在HTML中创建新元素，使用 $() 函数。

(2) 要在每个匹配的元素中插入新元素，使用：

 .append()

 .appendTo()

 .prepend()

 .prependTo()

(3) 要在每个匹配的元素相邻的位置上插入新元素，使用：

 .after()

 .insertAfter()

 .before()

 .insertBefore()

(4) 要在每个匹配的元素外部插入新元素，使用：

 .wrap()

 .wrapAll()

 .wrapInner()

(5) 要用新元素或文本替换每个匹配的元素，使用：

 .html()

 .text()

 .replaceAll()

 .replaceWith()

(6) 要移除每个匹配的元素中的元素，使用：

 .empty()

(7) 要从文档中移除每个匹配的元素及其后代元素，但不实际删除它们，使用：

 .remove()

 .detach()

## 通过Ajax发送数据（Asynchronous JavaScript and XML，异步JavaScript和XML）

### 跨域访问

不过， JavaScript的安全模型会限制数据文件必须与网页保存在相同的服务器上，这样就可以保证数据的可靠性。

但是，从第三方来源中加载数据往往是很有必要的。因而，也有许多方式可以绕过上述安全限制，即能够实现通过Ajax请求取得其他站点的数据。

**方式：**

* 加载外来JavaScript文件的一种流行方法是根据请求注入 <script> 标签。由于jQuery能帮我们插入新的DOM元素，因此向文档中注入 <script> 标签非常简单：

$(document.createElement('script'))

.attr('src', 'http://example.com/example.js')

.appendTo('head');

实际上， $.getScript() 方法在检测到其URL参数中包含远程主机时，就会自动采用这种技术；也就是说，该方法已经替我们想到了这一点。

* 另一种方法是使用 <iframe> 这个HTML标签来加载远程数据。 可以为 <iframe> 元素**指定任何URL作为其获取数据的来源，包括与提供页面的服务器不匹配的URL**。因此，第三方服务器上的数据能够轻易地加载到 <iframe> 中，并在当前页面上显示出来。然而，要操作 <iframe> 中的数据， 仍然存在同使用 <script> 标签时一样的协作需求； 位于 <iframe> 中的脚本需要明确地向父文档中的对象提供数据。

## 开发插件

### 在插件中使用 $ 别名

在编写jQuery插件时，必须假设jQuery库已经加载到了页面中。可是，我们不能假设 $ 别名一定可用。第3章曾经讲过， $.noConflict() 方法就是用于让渡这个快捷方式使用权的。考虑到这一点，我们自定义的插件就应该始终都使用 jQuery 这个名字来调用jQuery方法，或者也可以在内部定义一个 $ 别名。

对于代码比较长的插件来说，很多开发人员都觉得不能使用 $ 别名会导致代码难以理解。为了解决这个问题，我们可以在插件的作用域内定义这个快捷方式，方法就是定义一个函数并马上调用它。这种定义并立即调用函数的语法通常被称为立即调用的函数表达式（IIFE，Immediately Invoked Function Expression） ：

**(function($) {**

**//在这里添加代码**

**})(jQuery);**

这个包装函数只接收一个参数， 我们通过这个参数传入了 jQuery 对象。 这个参数的名字是 $ ，因此在这个函数内部，使用 $ 别名就不会有冲突了。

### 添加新的全局函数

jQuery内置的某些功能是通过全局函数提供的。所谓**全局函数**，实际上就是 jQuery 对象的方法，但从实践的角度上看，它们是位于 **jQuery 命名空间内部**的函数。

要向 jQuery 的命名空间中添加一个函数，只需将这个新函数指定为 jQuery 对象的一个属性即可。

**(function($) {**

**$.sum = function(array) {**

**//在这里添加代码**

**};**

**})(jQuery);**

**于是，我们就可以在使用这个插件的任何代码中，编写如下代码：**

**$.sum();**

### 扩展全局jQuery对象

事实上，利用 $.extend() 函数，还可以通过另外一种语法来定义全局函数。例如：

(function($) {

$.extend({

sum: function(array) {

var total = 0;

$.each(array, function(index, value) {

value = $.trim(value);

value = parseFloat(value) || 0;

total += value;

});

return total;

},

average: function(array) {

if ($.isArray(array)) {

return $.sum(array) / array.length;

}

return '';

}

});

})(jQuery);

这样调用 $.extend() 就可以给全局jQuery对象添加属性（如果原来有相同的属性，就会替换原来的属性） 。

### 使用命名空间隔离函数

我们的插件在jQuery命名空间中创建了两个独立的全局函数。但这样写有可能污染命名空间。换句话说，其他jQuery插件也可能定义相同的函数名。为了避免冲突，最好的办法是把属于一个插件的全局函数都封装到一个对象中，例如：

(function($) {

$.mathUtils = {

sum: function(array) {

var total = 0;

$.each(array, function(index, value) {

value = $.trim(value);

value = parseFloat(value) || 0;

total += value;

});

return total;

},

average: function(array) {

if ($.isArray(array)) {

return $.mathUtils.sum(array) / array.length;

}

return '';

}

};

})(jQuery);

这个模式的本质是为所有的全局函数又创建了一个命名空间，叫做 jQuery.mathUtils 。虽然我们还称它们为全局函数，但实际上它们已经成了 mathUtils 对象的方法了，而mathUtils 对象则保存在 jQuery 对象的属性中。结果，在调用它们时就必须得加上插件的名字了：

$.mathUtils.sum(sum);

$.mathUtils.average(average);

使用这种技术（以及足够独特的命名空间） ，就能够避免全局函数污染命名空间。至此，我们已经掌握了开发插件的基本方法。在把这些函数保存到名为jquery.mathutils.js的文件中之后，就可以将其包含在其他页面中通过其他脚本来使用这些函数了。

### 添加 jQuery 对象方法

jQuery中大多数内置的功能都是通过其对象实例的方法提供的，而且这些方法也是插件之所以诱人的关键。当函数需要操作DOM元素时，就是将函数创建为 jQuery 实例方法的好机会。

前面我们已经看到，添加全局函数需要以新方法来扩展 jQuery 对象。添加实例方法也与此类似，但扩展的却是 jQuery.fn 对象：

jQuery.fn.myMethod = function() {

alert('Nothing happens.');

};

然后，就可以在使用任何选择符表达式之后调用这个新方法了：

$('div').myMethod();

### 对象方法的上下文

在任何插件方法内部，关键字 this 引用的都是当前的jQuery对象。因而，可以在 this 上面调用任何内置的jQuery方法， 或者提取它包含的DOM节点并操作该节点。 为了确定可以怎样利用对象的上下文，下面我们来编写一个小插件，用以操作匹配元素的类。

(function($) {

$.fn.swapClass = function(class1, class2) {

console.log('到这里1');

this.each(function() {

var $element = $(this);

console.log('到这里2');

if ($element.hasClass(class1)) {

$element.removeClass(class1).addClass(class2);

} else if ($element.hasClass(class2)) {

$element.removeClass(class2).addClass(class1);

}

});

};

})(jQuery);

$(document).ready(function() {

$('table').click(function() {

$('tr').swapClass('one', 'two');

});

});

**this 的含义：**

注意！在对象方法体内，关键字 this 引用的是一个jQuery对象，但在每次

调用的 .each() 方法中， this 引用的则是一个DOM元素。

### 隐式迭代

jQuery的选择符表达式可能会匹配零、一或多个元素。因此，在设计插件时必须考虑到所有这些可能的情况。然而，我们在此调用的 .hasClass() 只会检查匹配的第一个元素。换句话说，我们应该独立检查和操作每一个元素。

要在无论匹配多个元素的情况下都保证行为正确， 最简单的方式就是始终在方法的上下文上调用 .each() 方法；这样就会执行隐式迭代，而执行隐式迭代对于维护插件与内置方法的一致性是至关重要的。

在调用的 .each() 方法内部， this 依次引用每个DOM元素，因此可以调整代码依次检测每个匹配的元素，并为它们应用相应的类。

(function($) {

$.fn.swapClass = function(class1, class2) {

console.log('到这里1');

this.each(function() {

var $element = $(this);

console.log('到这里2');

if ($element.hasClass(class1)) {

$element.removeClass(class1).addClass(class2);

} else if ($element.hasClass(class2)) {

$element.removeClass(class2).addClass(class1);

}

});

};

})(jQuery);

$(document).ready(function() {

$('table').click(function() {

$('tr').swapClass('one', 'two');

});

});

**this 的含义：**

注意！在对象方法体内，关键字 this 引用的是一个jQuery对象，但在每次

调用的 .each() 方法中， this 引用的则是一个DOM元素。

### 方法连缀

除了隐式迭代之外，jQuery用户也应该能够正常使用连缀行为。因而，我们必须在所有插件方法中返回一个jQuery对象，除非相应的方法明显用于取得不同的信息。返回的jQuery对象通常就是 this 所引用的对象。如果我们使用 .each() 迭代遍历 this ，那么可以只返回迭代的结果，参见代码清单8-10。

(function($) {

$.fn.swapClass = function(class1, class2) {

**return this.each(function() {**

var $element = $(this);

if ($element.hasClass(class1)) {

$element.removeClass(class1).addClass(class2);

} else if ($element.hasClass(class2)) {

$element.removeClass(class2).addClass(class1);

}

});

};

})(jQuery);

前面在调用了 .swapClass() 之后，如果想对元素再执行其他操作，必须通过一条新语句重新取得元素。而在添加 return 之后，就可以在我们的插件方法上面连缀内置的方法了。

## 高级选择符

### 编写定制的选择符插件

最容易添加的选择符是伪类，也就是以冒号开头的选择符表达式，比如 :checked或 :nth-child() 。为了演示创建选择符表达式的过程，我们会讨论构建一个名为 :group()的伪类。

在使用选择符表达式查找元素的时候，jQuery会在一个内部的对象 expr 中取得JavaScript代码。这个对象中的值与我们传入到 .filter() 或 .not() 中的筛选函数非常相似，当且仅当取得的函数返回 true 的情况下，才会让每个元素包含在结果集中。使用 $.extend() 函数可以为这个对象添加新的表达式，参见代码清单9-8。

(function($) {

$.extend($.expr[':'], {

group: function(element, index, matches, set) {

var num = parseInt(matches[3], 10);

if (isNaN(num)) {

return false;

}

return index % (num \* 2) < num;

}

});

})(jQuery);

以上代码告诉jQuery： group 是一个有效的字符串，可以放在一个冒号的后面构成选择符表达式。而在遇到这个选择符表达式的时候，应该调用给定的函数，用以决定相应的元素是否应该

包含在结果集中。

这个被求值的函数一共接收了4个参数。

**(1) element ：当前考虑的DOM元素。这个参数对于大多数选择符都是必须的，但我们这个**

**选择符则不需要。**

**(2) index ：DOM元素在结果集中的索引。**

**(3) matches ：数组，包含用于解析这个选择符的正则表达式的解析结果。一般来说，**

**matches[3] 是这个数组中唯一有用的值；假设有一个选择符的形式为 :group(b) ，则**

**matches[3] 中包含的值就是 b ，也就是括号中的文本。**

**(4) set ：匹配到当前元素的整个DOM元素集合。这个参数很少用。**

伪类选择符需要使用包含在这4个参数中的信息，决定当前元素是否应该包含在结果集中。

在我们这个例子中，只需要 index 和 matches 这两个参数。

定义了 :group 选择符之后，就有了交替元素分组的更灵活的方式。比如，代码清单9-5中的选择符表达式与 .filter() 函数可以组合为一个选择符表达式： $('#news tr:group(2)') 。而对于代码清单9-7的例子，也可以让表格中的每一部分行为保持一致，只要在调用 .filter()函数的时候传入 :group() 表达式即可。甚至，只要向其圆括号中传入不同的数值，就可以改变每一组的行数，如代码清单9-9所示。

### 选择符的性能问题

在把一个选择符表达式传递给 $() 函数时，jQuery的Sizzle引擎会解析这个表达式，并确定如何收集该表达式所表示的元素。在最本质的层次上，Sizzle会应用浏览器支持的最高效的原生DOM方法取得nodeList。 这个节点列表是一个包含DOM元素的类似数组的对象，jQuery最终会将这个对象转换成真正的数组，并将其添加到 jQuery 对象中。下面就是jQuery内部使用的几个DOM方法，同时给出了支持它们的浏览器及版本。

|  |  |  |
| --- | --- | --- |
| 方 法 | 选择目标 | 支持的浏览器 |
| .getElementById() | 取得ID与给定的字符串匹配的一个元素 | 全部 |
| .getElementsByTagName() | 取得标签名与给定的字符串匹配的所有元素 | 全部 |
| .getElementsByClassName() | 取得某个类名与给定的字符串匹配的所有元素 | IE 9+、 Firefox 3+、 Safari 4+、  Chrome 4+和Opera 10+ |
| .querySelectorAll() | 取得与给定的选择符表达式匹配的所有元素 | IE 8+、Firefox 3.5+、Safari  3+、Chrome 4+和Opera 10+ |

在这些方法都不能处理某个选择符表达式的情况下，Sizzle会退而求其次地循环遍历已经收集到的所有元素，并根据这个表达式来测试每一个元素。具体来说，假如没有现成的DOM方法可以拿来处理这个选择符表达式，**Sizzle就会使用 document.getElementsByTagName('\*') 来取得文档中的全部元素，然后再遍历并测试每个元素。**

与使用任何一个原生DOM方法相比，这种遍历和测试每个元素的方法十分影响性能。好就好在，所有现代浏览器的最新版本都开始原生支持 .querySelectorAll() 方法了，此时Sizzle就会在其他（也许更快的）原生方法不可用的情况下使用这个方法。但也有一个例外：**如果选择符表达式中包含自定义的jQuery选择符（例如 :eq() 、 :odd 或 :even ） ，而这些选择符并没有对应的CSS版本，那Sizzle也别无选择，只能循环加测试了。**

### jQuery对象属性

1. .context 属性中包含着一个DOM节点（通常是 document ）的引用，搜索就是从这个节点开始的；
2. .selector 属性中保存着创建最终对象的选择符表达式。
3. .prevObject 这个属性中保存着调用遍历方法的那个jQuery对象。

$(document).ready(function() {

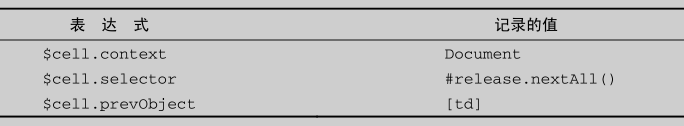
var $cell = $('#release').nextAll();

$cell.addClass('highlight');

console.log($cell.context);

console.log($cell.selector);

console.log($cell.prevObject);

});

### DOM元素栈

每个jQuery对象都有一个 .prevObject 属性指向前一个对象。这样，就有了一个实现了栈的列表结构。每个遍历方法都会找到一组新元素，然后把这组元素压入到栈中。这个栈只有我们需要它的时候才有用，而 .end() 和 .addBack() 方法就是用来操作这个栈的。

先来看 .end() 方法，这个方法只是简单地从栈中弹出一个元素，结果就是栈的最上方保存着与 .prevObject 属性中相同的引用。

.addBack()方法：

$(document).ready(function() {

$('#release').nextAll().addBack().addClass('highlight');

});

调用 .addBack() 时，jQuery会在栈中回溯一个位置，把两个位置上的**元素集组合起来**。具体到这个例子来说，这意味着被突出显示的不仅会包括 .nextAll() 找到的那两个单元格，而且还会包括最初通过ID选择符找到的那个单元格。然后，这个**新的、组合之后的元素集就会被压入栈的上方。**

### 编写DOM遍历方法插件