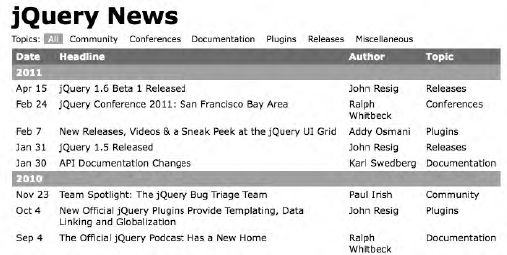
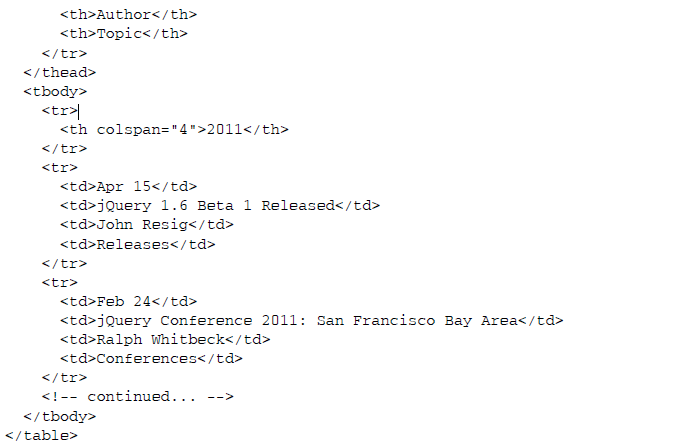
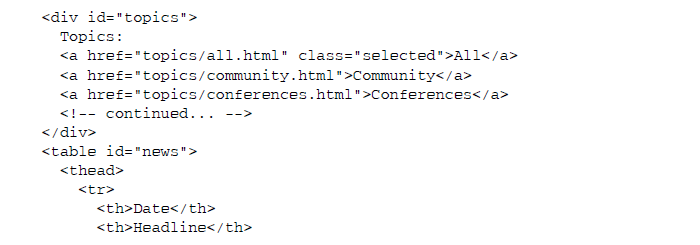
**8高级选择符与遍历**

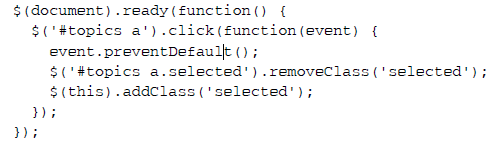
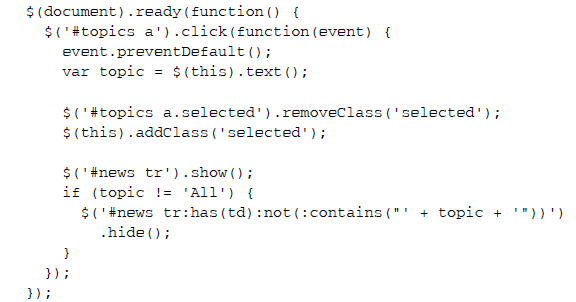
**主要内容**

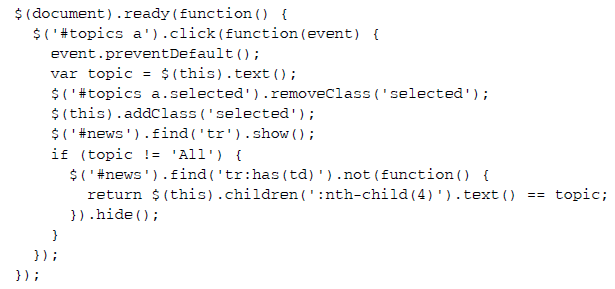
* 以不同的方式使用选择符查找和筛选数据；
* 编写插件以添加新选择符和DOM遍历方法；
* 优化选择符表达式，提高执行速度；
* 理解Sizzle引擎的某些内部工作原理。

**深入选择与遍历**

* 为了讨论有关高级选择符与遍历的内容，我们需要先编写一些脚本，以便有一些可供进一步讨论更高级选择与遍历的实例依据。这个例子是一个HTML页面，其中包含一组新闻列表项。所有新闻列表项都放在一个表格中，可供我们试验选择行与列的各种不同方法。以下是这个页面的代码片段：



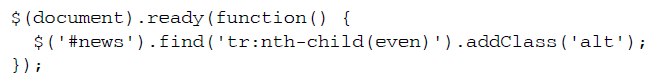
* 通过这一小段代码，大致能够看出整个文档的结构。这个表格包含4列，分别表示日期（Data）、标题（Headline）、作者（Author）和主题（Topic）。此外，表格中的某些行又包含年度“子标题”，而非前述这4项
* 为了通过主题链接来筛选表格内容，需要先阻止每个链接的默认行为。同时，还应该向用户反馈当前选择了哪个链接，如代码清单9-1所示。
* 接下来，就要执行实际的筛选操作，隐藏所有不包含相关主题的表格行，如代码清单9-2所示。
* 以上代码基本上可以使用，但是新闻标题中不能包含主题文本。为了排除这种情况，需要针对每一行多做一些检测如代码清单9-3所示。



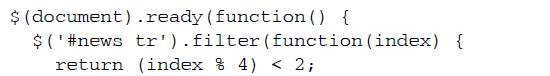
* 这些新代码用DOM遍历方法代替了某些复杂的选择符表达式。首先，.find()方法所起到的作用正如之前代码中#news与tr之间的空格。但是，.not()方法在这里完成的任务则是之前的:not()没有做到的。.not()可以接收一个回调函数，该函数将在检测每个元素的时候调用。如果这个函数返回true，那么被检测的元素就会被排除在结果集之外。

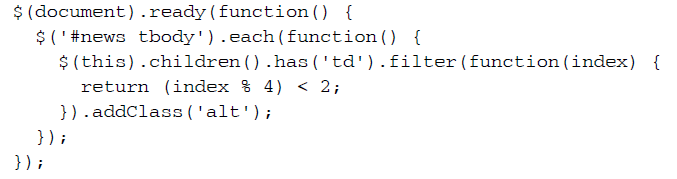
**为表格行添加条纹效果**

* 。我们知道，使用:even和:odd自定义选择符可以迅速实现这一效果，而使用CSS原生的:nth-child()伪类也可以实现同样的效果，如代码清单9-4所示。



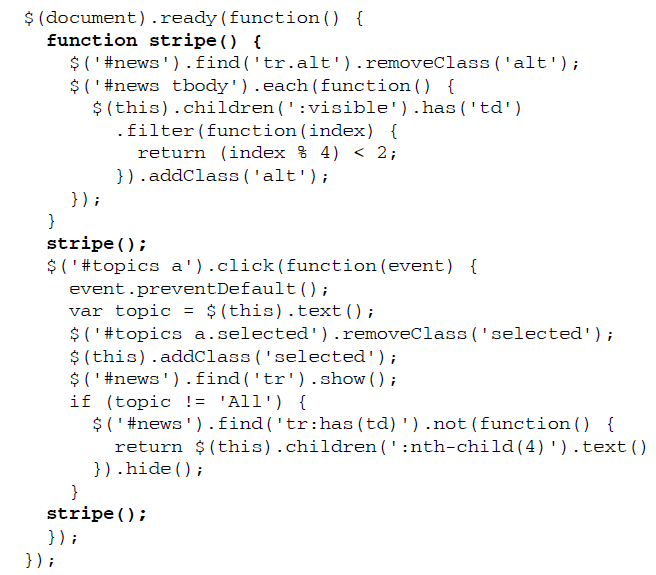
* 如果想实现更复杂一些的行条纹，可以尝试每两行一组地应用alt类。换句话说，给前两行添加这个类，后两行不加，依此类推。为此，需要再用到筛选函数，参见代码清单9-5。

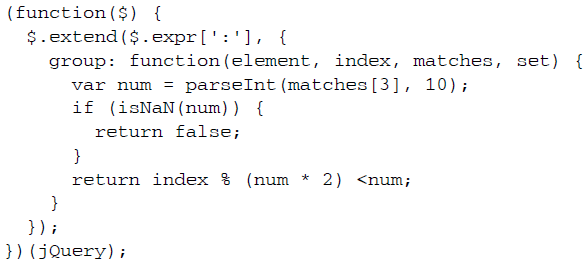


* 因为没有再使用:nth-child()伪类，所以交替效果不会在每个<tbody>元素中分别开始。同时，为了保证外观的一致，还要跳过表格中的标题行。这些问题只要修改代码中的两个地方就可以解决，如代码清单9-6所示。

**组合筛选与条纹**

* 高级的表格条纹效果还不错，然而只要一按照主题来筛选，奇怪的现象就出来了。为了让这两个功能可以完美地共存，必须在每次筛选之后重新应用条纹效果。此外，在应用alt类的时候，还要考虑某些行当前是否隐藏



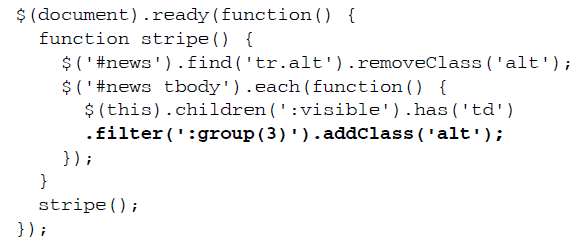
* 9.2定制与优化选择符
* 将我们看到的那么多技术放到一起，就像是配备了一个工具箱。工具箱中的工具能够帮我们在页面中查找想要操作的任何元素。
* 但我们要了解的还不止这些。怎么才能更加有效地查找元素也是必须关心的一个问题。
* 所谓有效，可能会反映在几个方面。比如代码容易读、容易写，再比如代码在浏览器中运行的速度更快。
* 9.2.1编写定制的选择符插件
* 提高代码可读性的一种方式是把代码片段封装为可以重用的组件。我们之所以创建函数就是这个目的。第8章刚刚讨论了这种思想的延伸，那就是以创建jQuery插件的方式为jQuery对象添加方法。不过，插件并不局限于为jQuery对象添加方法，还可以让我们自定义选择符表达式，例如第7章介绍的Cycle插件中的:paused选择符。
* 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()。我们会构建一个名为:group()的伪类。这个新选择符将用于封装代码清单9-6中的那些查找表格行并为它们添加条纹效果的代码。
* 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。
* 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。

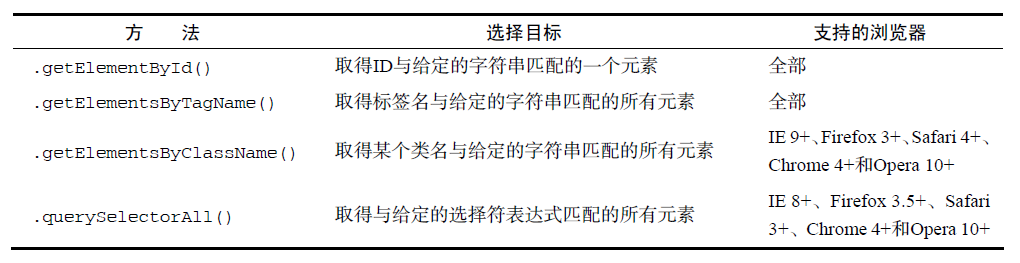
* 对于代码清单9-7的例子，也可以让表格中的每一部分行为保持一致，只要在调用.filter()函数的时候传入:group()表达式即可。甚至，只要向其圆括号中传入不同的数值，就可以改变每一组的行数，如代码清单9-9所示。

**9.2.2 选择符的性能问题**

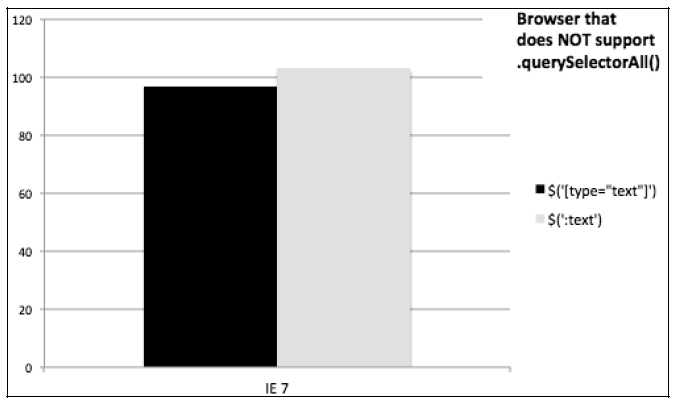
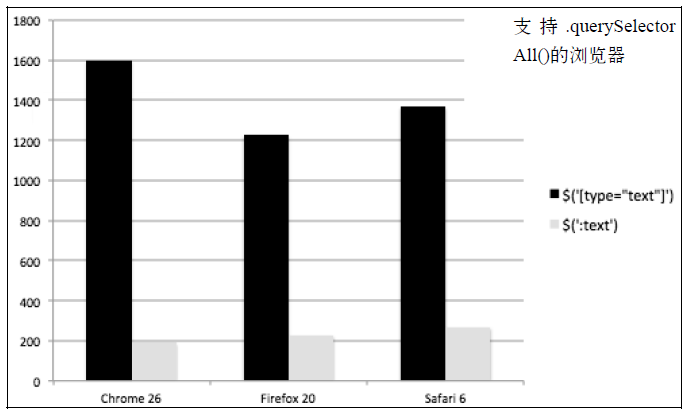
* 在规划任何Web项目的时候，都需要考虑项目周期、维护代码的难易程度和效率，以及用户使用网站过程中的性能等问题。通常，前两个问题比第三个问题更重要。特别是对客户端脚本编程来说，开发人员经常落入“**过早优化**”和“**微观优化**”的陷阱之中。无数个小时的时间投入进去，换来的往往只有JavaScript代码执行过程中毫秒级别的提升，这种提升也很难被用户的眼睛觉察到。
* 针对选择符和遍历速度所作的任何决定，都有可能伴随着更新更快的浏览器发布，或者jQuery新版本加入巧妙的速度优化而变得毫无价值。为了真正提升性能，最好反复思考自己假定的条件，然后在使用jsPerf（http://jsperf.com/）等工具实际测量之后，再动手编写优化代码。

**1. Sizzle的选择符实现**

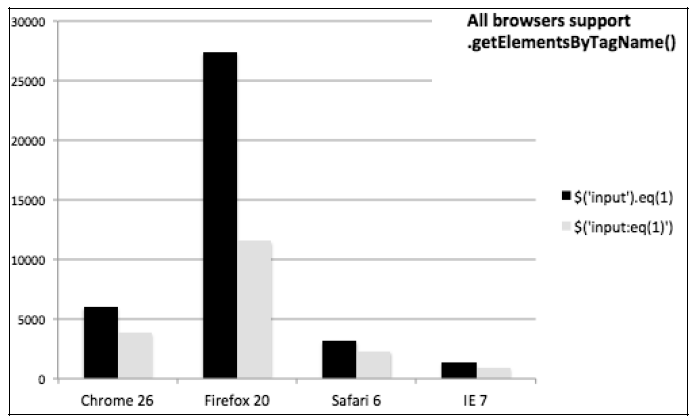
* 在把一个选择符表达式传递给$()函数时，jQuery的Sizzle引擎会解析这个表达式，并确定如何收集该表达式所表示的元素。在最本质的层次上，Sizzle会应用浏览器支持的最高效的原生DOM方法取得nodeList。这个节点列表是一个包含DOM元素的类似数组的对象，jQuery最终会将这个对象转换成真正的数组，并将其添加到jQuery对象中。下面就是jQuery内部使用的几个DOM方法，同时给出了支持它们的浏览器及版本。



* 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。
* 2. 测试选择符的速度
* 为了让大家对使用.querySelectorAll()方法和使用循环加测试的方法的性能差异有个直观的了解，我们假设想要找到一个文档中的所有<input type="text">元素。为此，可以使用两种选择符表达式，一种是CSS属性选择符$('input[type="text"]')，另一种jQuery自定义选择符$('input:text')。为测试选择符中我们关注的部分，我们要去掉其中的input，只比较$('[type="text"]')和$(':text')的速度。使用这两种表达式在JavaScript基准测试网站http://jsperf.com/中进行测试，结果相当有戏剧性。
* 在jsPerf测试中，每个测试用例都会被反复循环，从而得到指定的时间内能够完成多少次循环。因此，结果数字越大，说明速度越快



* 测试$('input:eq(1)') 和$('input').eq(1)这两个选择符的结果，可以看到支持原生方法和不支持原生方法的浏览器性能差异更大。



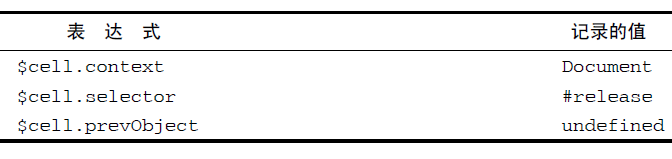
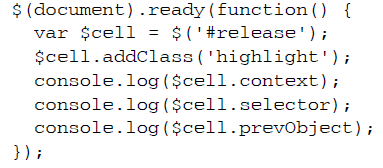
* 一条通用的经验法则：要尽可能使用CSS规范中规定的选择符，除非没有可使用jQuery的自定义选择符。同样，在修改选择符之前，也要记住只在确实有必要提升性能的情况下再去提升。

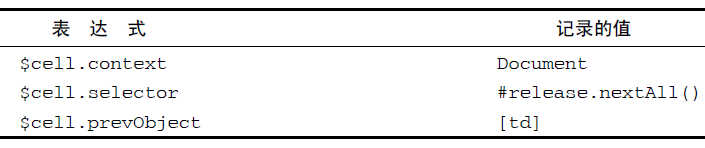
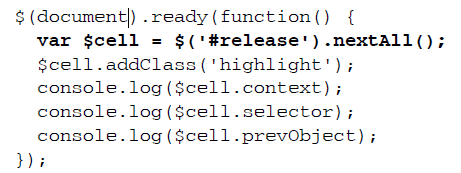
**9.3 DOM 遍历背后的秘密**

* 每当我们从一个（组）DOM元素转移到另一个（组）DOM元素时，jQuery都会留意我们移动的路线并留下“面包屑”，以便我们在必要时能够找到“回家”的路。第2章曾简要提到过的两个方法.end()和.addBack()就利用了这个记录。为了最大限度地利用这些方法，同时写出一般意义上的高效的jQuery代码，我们必须深入理解DOM遍历方法的运作机制。

**9.3.1 jQuery对象属性**

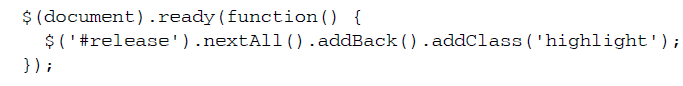
* 众所周知，要得到一个jQuery对象的实例，需要向$()函数传入一个选择符表达式。而得到的对象是一个数组结构，其中包含着与该选择符匹配的每个DOM元素的引用。可是我们并不知道的是，这个对象中还隐藏着其他一些属性。比如.**context**属性中包含着一个DOM节点（通常是document）的引用，搜索就是从这个节点开始的；比如.**selector**属性中保存着创建最终对象的选择符表达式。.**prevObject**这个属性中保存着调用遍历方法的那个jQuery对象。
* 代码清单9-10



* 我们再向代码中添加一个遍历方法，然后结果就会有所变化

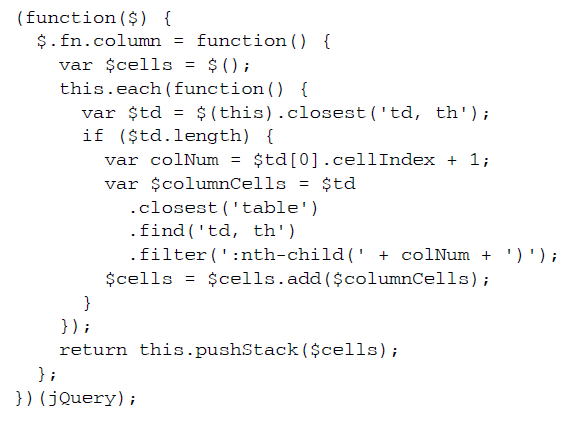
**9.3.2 DOM元素栈**

* 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。
* 参见代码清单9-12。

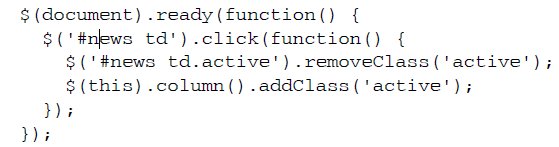


**9.3.3 编写DOM遍历方法插件**

* 与其他的jQuery对象方法类似，也可以通过为$.fn添加属性的方式来向jQuery中添加遍历方法。第8章曾介绍过，新添加的jQuery方法应该在操作匹配的元素集之后返回jQuery对象，以便用户可以再连缀其他方法。在创建DOM遍历方法时，这个过程也是类似的，但是返回的jQuery对象必须要指向一个新匹配的元素集。
* 举个例子会更清楚。比如我们要编写一个插件，用来找到与给定单元格在同一列中的所有单元格，那么可以用代码清单9-13中的代码实现这个插件。

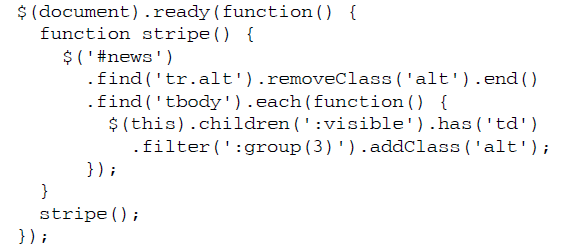


* 下面来看一看这个插件的用法。代码清单9-14



**9.3.4 DOM遍历的性能问题**

* 虽然应该避免过早优化，但最低限度地重复选择符和遍历方法则始终是值得提倡的。因为这些操作都可能会耗费较多时间，用得越少越好。而要避免重复，有两个策略值得讨论，那就是连缀和缓存对象。
* 1. 使用连缀来改进性能
* 我们多次使用了连缀，它能够让代码更简洁。事实上，使用连缀还有可能带来性能上的提升，而关键则是利用它来减少重复。代码清单9-15



* 2. 使用缓存来改进性能
* 我们可以重写stripe()函数，这次使用缓存而不是连缀，同样也可以避免重复使用选择符，如代码清单9-16所示。

