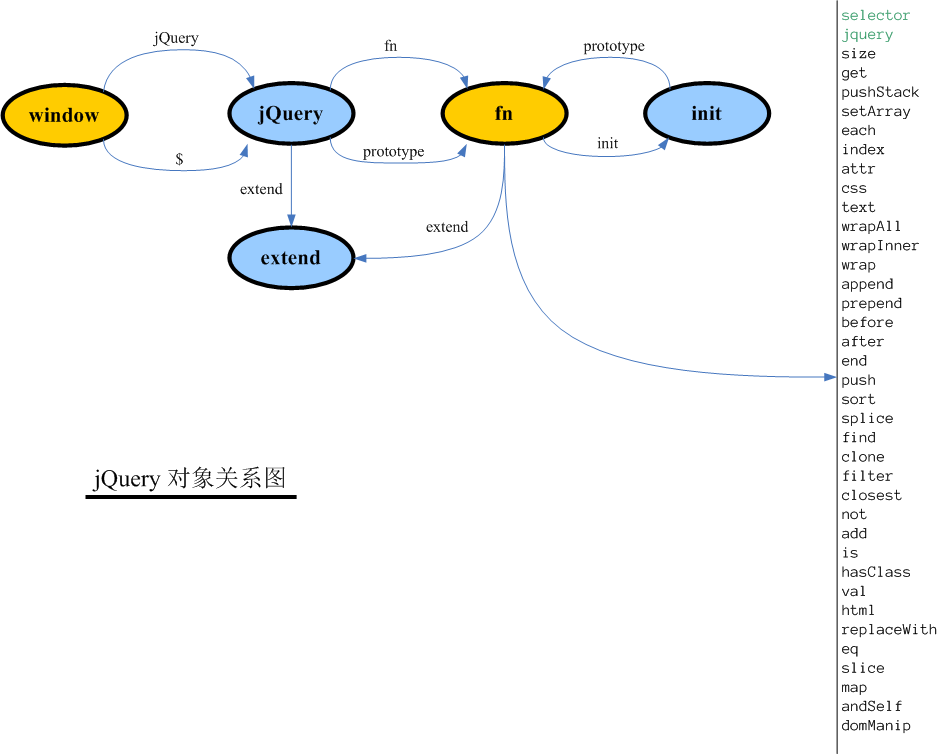
# ****jQuery源码分析****



## 源码结构

(function (global, factory) {

factory(global);

}(typeof window !== "undefined" ? window : this, function (window, noGlobal) {

var jQuery = function (selector, context) {

return new jQuery.fn.init(selector, context);

};

// jQuery对象原型

jQuery.fn = jQuery.prototype = {

constructor: jQuery,

init: function (selector, context, rootjQuery) {

// selector有以下7种分支情况：

// DOM元素

// body（优化）

// 字符串：HTML标签、HTML字符串、#id、选择器表达式

// 函数（作为ready回调函数）

// 最后返回伪数组

}

};

// Give the init function the jQuery prototype for later instantiation

jQuery.fn.init.prototype = jQuery.fn;

// 合并内容到第一个参数中，后续大部分功能都通过该函数扩展

// 通过jQuery.fn.extend扩展的函数，大部分都会调用通过jQuery.extend扩展的同名函数

jQuery.extend = jQuery.fn.extend = function () { };

// 在jQuery上扩展静态方法

jQuery.extend({

// ready bindReady

// isPlainObject isEmptyObject

// parseJSON parseXML

// globalEval

// each makeArray inArray merge grep map

// proxy

// access

// uaMatch

// sub

// browser

});

// 核心方法

// 回调系统

// 异步队列

// 数据缓存

// 队列操作

// 选择器引

// 属性操作

// 节点遍历

// 文档处理

// 样式操作

// 属性操作

// 事件体系

// AJAX交互

// 动画引擎

var

// Map over jQuery in case of overwrite

\_jQuery = window.jQuery,

// Map over the $ in case of overwrite

\_$ = window.$;

jQuery.noConflict = function (deep) {

if (window.$ === jQuery) {

window.$ = \_$;

}

if (deep && window.jQuery === jQuery) {

window.jQuery = \_jQuery;

}

return jQuery;

};

window.jQuery = window.$ = jQuery;

return jQuery;

}));

jQuery一共13个模块，从2.1版开始jQuery支持通过AMD模块划分，jQuery在最开始发布的1.0版本是很简单的，只有CSS选择符、事件处理和AJAX交互3大块。其发展过程中，有几次重要的变革：

☑ 1.2.3 版发布，引入数据缓存，解决循环引用与大数据保存的问题

☑ 1.3 版发布，它使用了全新的选择符引擎Sizzle，在各个浏览器下全面超越其他同类型JavaScript框架的查询速度，程序库的性能也因此有了极大提升

☑ 1.5 版发布，新增延缓对像(Deferred Objects)，并用deferred重写了Ajax模块

☑ 1.7 版发布，抽象出回调对象，提供了强大的的方式来管理回调函数列表。

每一次大的改进都引入了一些新的机制、新的特性，通过这些新的机制就造就了如今jQuery库，一共13个模块，模块不是单一的，比如jQuery动画，都会依赖异步队列、动画队列、回调队列与数据缓存模块等。

jQuery抽出了所有可复用的特性，分离出单一模块，通过组合的用法，不管在设计思路与实现手法上jQuery都是非常高明的。

**五大块：**  
jQuery按我的理解分为五大块，选择器、DOM操作、事件、AJAX与动画，那么为什么有13个模块？因为jQuery的设计中最喜欢的做的一件事，就是抽出共同的特性使之“模块化”，当然也是更贴近S.O.L.I.D五大原则的“单一职责SRP”了，遵守单一职责的好处是可以让我们很容易地来维护这个对象，比如，当一个对象封装了很多职责的时候，一旦一个职责需要修改，势必会影响该对象的其它职责代码。通过解耦可以让每个职责更加有弹性地变化。  
我们来看看jQuery文档针对业务层的Ajax的处理提供了一系列的门面接口：

.ajaxComplete()

.ajaxError()

.ajaxSend()

.ajaxStart()

.ajaxStop()

.ajaxSuccess()

底层接口：

jQuery.ajax()

jQuery.ajaxSetup()

快捷方法：

jQuery.get()

jQuery.getJSON()

jQuery.getScript()

jQuery.post()

**jQuery接口的设计原理**

业务逻辑是复杂多变的，jQuery的高层API数量非常多，而且也非常的细致，这样做可以更友好的便于开发者的操作，不需要必须在一个接口上重载太多的动作。我们在深入内部看看Ajax的高层方法其实都是统一调用了一个静态的jQuery.ajax方法。

jQuery.each(["get", "post"], function (i, method) {

jQuery[method] = function (url, data, callback, type) {

// Shift arguments if data argument was omitted

if (jQuery.isFunction(data)) {

type = type || callback;

callback = data;

data = undefined;

}

return jQuery.ajax({

url: url,

type: method,

dataType: type,

data: data,

success: callback

});

};

});

在jQuery.ajax的内部实现是非常复杂的，首先ajax要考虑异步的处理与回调的统一性，所以就引入了异步队列模块（Deferred）与回调模块（Callbacks）, 所以要把这些模块方法在ajax方法内部再次封装成、构建出一个新的jQXHR对象，针对参数的默认处理，数据传输的格式化等等。

## 立即调用表达式

任何库与框架设计的第一个要点就是解决命名空间与变量污染的问题。jQuery就是利用了JavaScript函数作用域的特性，采用立即调用表达式包裹了自身的方法来解决这个问题。

打开jQuery源码，首先你会看到这样的代码结构：

|  |
| --- |
| (function (global, factory) {  factory(global);  }(typeof window !== "undefined" ? window : this, function (window, noGlobal) {  window.jQuery = window.$ = jQuery;  return jQuery;  })); |

1.     这是一个*自调用匿名函数*。什么东东呢？在第一个括号内，创建一个匿名函数；第二个括号，立即执行；

2.     为什么要创建这样一个“自调用匿名函数”呢？

通过定义一个匿名函数，创建了一个“私有”的命名空间，该命名空间的变量和方法，不会破坏全局的命名空间。这点非常有用也是一个JS框架必须支持的功能，jQuery被应用在成千上万的JavaScript程序中，必须确保jQuery创建的变量不能和导入他的程序所使用的变量发生冲突。

3.     匿名函数从语法上叫函数直接量，JavaScript语法需要包围匿名函数的括号，事实上自调用匿名函数有两种写法（注意标红了的右括号）：

|  |  |
| --- | --- |
| (function() {      console.info( this );      console.info( arguments );  }( window ) ); | (function() {      console.info( this );      console.info( arguments );  })( window ); |

4.     为什么要传入window呢？

通过传入window变量，使得window由全局变量变为局部变量，当在jQuery代码块中访问window时，不需要将作用域链回退到顶层作用域，这样可以更快的访问window；这还不是关键所在，更重要的是，将window作为参数传入，可以在压缩代码时进行优化，看看jquery.js：

|  |
| --- |
| (**function**(a,b){})(window); // window 被优化为 a |

5.     注意到源码最后的分号了吗？

分号是可选的，但省略分号并不是一个好的编程习惯；为了更好的兼容性和健壮性，请在每行代码后加上分号并养成习惯。

总结：全局变量是魔鬼, 匿名函数可以有效的保证在页面上写入JavaScript，而不会造成全局变量的污染，通过小括号，让其加载的时候立即初始化，这样就形成了一个单例模式的效果从而只会执行一次。

## ****jQuery对象的无new构建****

jQuery对象不是通过 **new** jQuery 创建的，而是通过 **new** jQuery.fn.init 创建的；

|  |
| --- |
| **var** jQuery = **function**( selector, context ) {  **return** **new** jQuery.fn.init( selector, context, rootjQuery );  } |

jQuery对象就是jQuery.fn.init对象；

如果执行new jQuery(),生成的jQuery对象会被抛弃，最后返回 jQuery.fn.init对象；因此可以直接调用jQuery( selector, context )，没有必要使用new关键字；

先执行 jQuery.fn = jQuery.prototype，再执行 jQuery.fn.init.prototype = jQuery.fn，合并后的代码如下：

|  |
| --- |
| jQuery.fn.init.prototype = jQuery.fn = jQuery.prototype |

所有挂载到jQuery.fn的方法，相当于挂载到了jQuery.prototype，即挂载到了jQuery 函数上（一开始的 jQuery = **function**( selector, context ) ），但是最后都相当于挂载到了jQuery.fn.init.prototype，即相当于挂载到了一开始的jQuery 函数返回的对象上，即挂载到了我们最终使用的jQuery对象上。

这个过程非常的绕，金玉其外“败絮”其中啊！

下面分析为什么采用无new构建方式。

var aQuery = function(selector, context) { //构造函数}aQuery.prototype = { //原型 name:function(){}, age:function(){}}var a = new aQuery();a.name();

这是常规的使用方法，显而易见jQuery不是这样玩的

jQuery没有使用new运行符将jQuery显示的实例化，还是直接调用其函数

按照jQuery的抒写方式

$().ready() $().noConflict()

要实现这样,那么jQuery就要看成一个类，那么$()应该是返回类的实例才对

所以把代码改一下：

var aQuery = function(selector, context) { return new aQuery();}aQuery.prototype = { name:function(){}, age:function(){}}

通过new aQuery()，虽然返回的是一个实例，但是也能看出很明显的问题，死循环了！

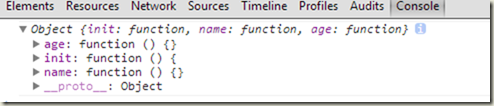
**那么如何返回一个正确的实例？**

在javascript中实例this只跟原型有关系

那么可以把jQuery类当作一个工厂方法来创建实例，把这个方法放到jQuery.prototye原型中

var aQuery = function(selector, context) { return aQuery.prototype.init();}aQuery.prototype = { init:function(){ return this; } name:function(){}, age:function(){}}

当执行aQuery() 返回的实例：

[](http://images.cnitblog.com/blog/329084/201308/23205113-a69b74e9e95b4c05b15493835a855c03.png)

很明显aQuery()返回的是aQuery类的实例，那么在init中的this其实也是指向的aQuery类的实例

问题来了init的this指向的是aQuery类，如果把init函数也当作一个构造器，那么内部的this要如何处理？

var aQuery = function(selector, context) { return aQuery.prototype.init();}aQuery.prototype = { init: function() { this.age = 18 return this; }, name: function() {}, age: 20}aQuery().age //18

这样的情况下就出错了，因为this只是指向aQuery类的，所以需要设计出独立的作用域才行

**jQuery框架分隔作用域的处理**

jQuery = function( selector, context ) { // The jQuery object is actually just the init constructor 'enhanced' return new jQuery.fn.init( selector, context, rootjQuery ); },

很明显通过实例init函数，每次都构建新的init实例对象，来分隔this,避免交互混淆

那么既然都不是同一个对象那么肯定又出现一个新的问题

例如：

var aQuery = function(selector, context) { return new aQuery.prototype.init();}aQuery.prototype = { init: function() { this.age = 18 return this; }, name: function() {}, age: 20}//Uncaught TypeError: Object [object Object] has no method 'name' console.log(aQuery().name())

抛出错误，无法找到这个方法，所以很明显new的init跟jquery类的this分离了

**怎么访问jQuery类原型上的属性与方法？**

     做到既能隔离作用域还能使用jQuery原型对象的作用域呢，还能在返回实例中访问jQuery的原型对象?

**实现的关键点**

// Give the init function the jQuery prototype for later instantiationjQuery.fn.init.prototype = jQuery.fn;

通过原型传递解决问题，把jQuery的原型传递给jQuery.prototype.init.prototype

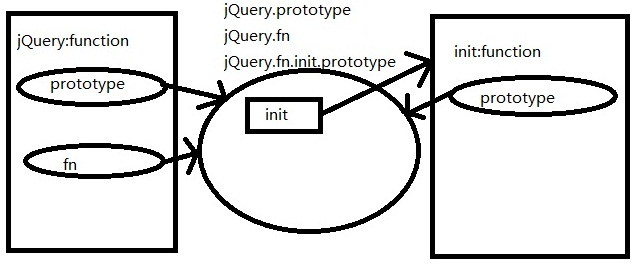
换句话说jQuery的原型对象覆盖了init构造器的原型对象

因为是引用传递所以不需要担心这个循环引用的性能问题

var aQuery = function(selector, context) { return new aQuery.prototype.init();}aQuery.prototype = { init: function() { return this; }, name: function() { return this.age }, age: 20}aQuery.prototype.init.prototype = aQuery.prototype;console.log(aQuery().name()) //20

百度借网友的一张图，方便直接理解：

fn解释下，其实这个fn没有什么特殊意思，只是jQuery.prototype的引用



## ****插件接口的设计****

jQuery插件的开发分为两种：

☑ 一种是挂在jQuery命名空间下的全局函数，也可称为静态方法；

☑ 另一种是jQuery对象级别的方法，即挂在jQuery原型下的方法，这样通过选择器获取的jQuery对象实例也能共享该方法。

        jQuery支持自己扩展属性，这个对外提供了一个接口，jQuery.fn.extend()来对对象增加方法，从jQuery的源码中可以看到，jQuery.extend和jQuery.fn.extend其实是同指向同一方法的不同引用。

        这里有一个设计的重点，通过调用的上下文，我们来确定这个方法是作为静态还是实例处理，在javascript的世界中一共有四种上下文调用方式：方法调用模式、函数调用模式、构造器调用模式、apply调用模式：

☑ jQuery.extend调用的时候上下文指向的是jQuery构造器

☑ jQuery.fn.extend调用的时候上下文指向的是jQuery构造器的实例对象了

        通过extend()函数可以方便快速的扩展功能，不会破坏jQuery的原型结构，jQuery.extend = jQuery.fn.extend = function(){...}; 这个是连等，也就是2个指向同一个函数，怎么会实现不同的功能呢？这就是this力量了！

        fn与jQuery其实是2个不同的对象，在之前有讲解：jQuery.extend 调用的时候，this是指向jQuery对象的(jQuery是函数，也是对象！)，所以这里扩展在jQuery上。而jQuery.fn.extend 调用的时候，this指向fn对象，jQuery.fn 和jQuery.prototype指向同一对象，扩展fn就是扩展jQuery.prototype原型对象。这里增加的是原型方法，也就是对象方法了。所以jQuery的API中提供了以上2个扩展函数。

API 如下：

jQuery.extend( [ deep ], target, [ object1 ], [ objectN ] )

描述：合并两个或更多的对象的内容汇集成到第一个对象。

* deep：如果是true，合并成为递归（又叫做深拷贝）。
* target：一个对象，如果附加的对象被传递给这个方法将那么它将接收新的属性，如果它是唯一的参数将扩展 jQuery 的命名空间。
* object1：一个对象，它包含额外的属性合并到第一个参数
* objectN：包含额外的属性合并到第一个参数

      Jquery的扩展方法extend是我们在写插件的过程中常用的方法，该方法有一些重载原型，在此，我们一起去了解了解。  
     **一、Jquery的扩展方法原型是:**

extend(dest,src1,src2,src3...);

      它的含义是将src1,src2,src3...合并到dest中,返回值为合并后的dest,由此可以看出该方法合并后，是修改了dest的结构的。如果想要得到合并的结果却又不想修改dest的结构，可以如下使用：

var newSrc=$.extend({},src1,src2,src3...)//也就是将"{}"作为dest参数。

      这样就可以将src1,src2,src3...进行合并，然后将合并结果返回给newSrc了。如下例：

var result=$.extend({},{name:"Tom",age:21},{name:"Jerry",sex:"Boy"})

       那么合并后的结果

result={name:"Jerry",age:21,sex:"Boy"}

      也就是说后面的参数如果和前面的参数存在相同的名称，那么后面的会覆盖前面的参数值。

**二、省略dest参数**  
      上述的extend方法原型中的dest参数是可以省略的，如果省略了，则该方法就只能有一个src参数，而且是将该src合并到调用extend方法的对象中去，如：  
 　　**1、$.extend(src)** 　　该方法就是将src合并到jquery的全局对象中去，如：

$.extend({  
 hello:function(){alert('hello');}  
 });

 　　就是将hello方法合并到jquery的全局对象中。  
 　　**2、$.fn.extend(src)**  
 　　该方法将src合并到jquery的实例对象中去，如:

$.fn.extend({  
 hello:function(){alert('hello');}  
 });

　　 就是将hello方法合并到jquery的实例对象中。

 　　下面例举几个常用的扩展实例：

$.extend({net:{}});

　　 这是在jquery全局对象中扩展一个net命名空间。

$.extend($.net,{  
 hello:function(){alert('hello');}  
 })

  　　这是将hello方法扩展到之前扩展的Jquery的net命名空间中去。

**三、Jquery的extend方法还有一个重载原型：**

extend(boolean,dest,src1,src2,src3...)

      第一个参数boolean代表是否进行深度拷贝，其余参数和前面介绍的一致，什么叫深层拷贝，我们看一个例子：

var result=$.extend( true, {},   
 { name: "John", location: {city: "Boston",county:"USA"} },   
 { last: "Resig", location: {state: "MA",county:"China"} } );

      我们可以看出src1中嵌套子对象location:{city:"Boston"},src2中也嵌套子对象location:{state:"MA"},第一个深度拷贝参数为true，那么合并后的结果就是：

result={name:"John",last:"Resig",  
 location:{city:"Boston",state:"MA",county:"China"}}

       也就是说它会将src中的嵌套子对象也进行合并，而如果第一个参数boolean为false，我们看看合并的结果是什么，如下：

var result=$.extend( false, {},   
{ name: "John", location:{city: "Boston",county:"USA"} },   
{ last: "Resig", location: {state: "MA",county:"China"} }   
 );

     那么合并后的结果就是:

result={name:"John",last:"Resig",location:{state:"MA",county:"China"}}

// 从object原型继承的属性会被拷贝

// undefined值不会被拷贝

// 因为性能原因，JavaScript自带类型的属性不会合并

//两种用法：

// jQuery.extend( target, [ object1 ], [ objectN ] )

// jQuery.extend( [ deep ], target, object1, [ objectN ] )

jQuery.extend = jQuery.fn.extend = function () {

var options, name, src, copy, copyIsArray, clone,

target = arguments[0] || {},

i = 1,

length = arguments.length,

deep = false;

// Handle a deep copy situation

// 如果第一个参数是boolean型，可能是深度拷贝

if (typeof target === "boolean") {

deep = target;

// Skip the boolean and the target

target = arguments[i] || {};

i++;// 跳过boolean和target，从第3个开始，此时i=2

}

// Handle case when target is a string or something (possible in deep copy)

// 如果target不是对象也不是函数，则强制设置为空对象

if (typeof target !== "object" && !jQuery.isFunction(target)) {

target = {};

}

// Extend jQuery itself if only one argument is passed

// 如果只传入一个参数，则认为是对jQuery扩展

if (i === length) {

target = this;

i--;//此时i=1

}

for (; i < length; i++) {

// Only deal with non-null/undefined values

//只处理非空参数

if ((options = arguments[i]) != null) {

// Extend the base object

for (name in options) {

src = target[name];

copy = options[name];

// Prevent never-ending loop

// 避免循环引用

if (target === copy) {

continue;

}

// Recurse if we're merging plain objects or arrays

// 深度拷贝且值是纯对象或数组，则递归

if (deep && copy && (jQuery.isPlainObject(copy) || (copyIsArray = jQuery.isArray(copy)))) {

if (copyIsArray) {// 如果copy的是数组

copyIsArray = false;

clone = src && jQuery.isArray(src) ? src : [];// clone为src的修正值

} else {//如果copy的是纯对象

clone = src && jQuery.isPlainObject(src) ? src : {};// clone为src的修正值

}

// Never move original objects, clone them

// 递归调用jQuery.extend

target[name] = jQuery.extend(deep, clone, copy);

// Don't bring in undefined values

// 不能拷贝空值

} else if (copy !== undefined) {

target[name] = copy;

}

}

}

}

// Return the modified object

//返回更改后的对象

return target;

};

我来讲解一下上面的代码：因为extend的核心功能就是通过扩展收集功能（类似于mix混入），所以就会存在收集对象（target）与被收集的数据，因为jQuery.extend并没有明确实参，而且是通过arguments来判断的，所以这样处理起来很灵活。arguments通过判断传递参数的数量可以实现函数重载。其中最重要的一段target = this，通过调用的方式我们就能确实当前的this的指向，所以这时候就能确定target了。最后就很简单了，通过for循环遍历把数据附加到这个target上了。当然在这个附加的过程中我们还可以做数据过滤、深拷贝等一系列的操作了。

**总结：**

* 通过new jQuery.fn.init() 构建一个新的对象，拥有init构造器的prototype原型对象的方法;
* 通过改变prorotype指针的指向，让这个新的对象也指向了jQuery类的原型prototype;
* 所以这样构建出来的对象就继续了jQuery.fn原型定义的所有方法了;

## ****回溯处理的设计****

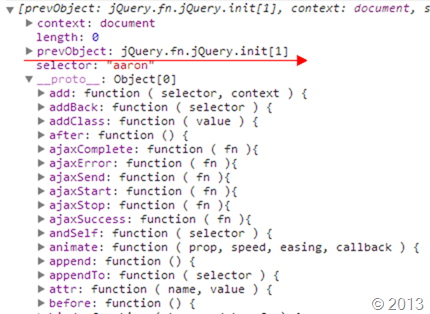
        在这一小节我将会带领你们了解jQuery对DOM进行遍历背后的工作机制，这样可以在编写代码时有意识地避免一些不必要的重复操作，从而提升代码的性能。

**关于jQuery对象的包装**

var $aaron = $("aaron");

通过对sizzle的分析，我们可以得知Query选择器最终都是通过DOM接口实现取值的, 但是通过jQuery处理后返回的不仅仅只有DOM对象，而是一个包装容器，返回jQuery对象：$aaron。

我们来看一下代码：



**在jQuery对象中有个prevObject对象，这个是干嘛用的呢？**

如果你想知道prevObject是做什么的，咱们首先得先来了解一下jQuery对象栈，jQuery内部维护着一个jQuery对象栈。每个遍历方法都会找到一组新元素（一个jQuery对象），然后jQuery会把这组元素推入到栈中。

而每个jQuery对象都有三个属性：context、selector和prevObject，其中的prevObject属性就指向这个对象栈中的前一个对象，而通过这个属性可以回溯到最初的DOM元素集中。

为了方便理解，我们做几个简单的测试：

下面有一个父元素ul,嵌套了一个li节点：

<ul id="aaron">

    parent

    <li>child</li>

</ul>

我们现给li绑定一个事件，这个很简单，找到ul下面的li，绑定即可：

var aaron = $("#aaron");

    aaron.find('li').click(function(){

        alert(1);     //1

    })

此时我又想给父元素绑定一个事件，我们是不是又要在aaron上绑定一次事件呢？是的，上面代码通过find处理后，此时的上下文是指向每一个li了,所以必须要重新引用aaron元素（li的父元素），然后再绑定click事件：

aaron.click(function(){

      alert(2);     //1

})

这样会不会很麻烦，所以jQuery引入一个简单的内部寻址的机制，可以回溯到之前的Dom元素集合，通过end()方法可以实现：

aaron.find('li').click(function() {

        alert(1);

}).end().click(function() {

        alert(2);

})

jQuery为我们操作这个内部对象栈提供个非常有用的2个方法

.end()

.addBack()

这里需要指出来可能有些API上是andSelf，因为就Query的api是这样写的，andSelf现在是.addBack()的一个别名。在jQuery1.8和更高版本中应使用.addBack()

源码其实也是这样的

jQuery.fn.andSelf = jQuery.fn.addBack;

调用第一个方法只是简单地弹出一个对象（结果就是回到前一个jQuery对象）。第二个方法更有意思，调用它会在栈中回溯一个位置，然后把两个位置上的元素集组合起来，并把这个新的、组合之后的元素集推入栈的上方。

利用这个DOM元素栈可以减少重复的查询和遍历的操作，而减少重复操作也正是优化jQuery代码性能的关键所在。

## ****选择器****

jQuery.fn.init的功能是对传进来的selector参数进行分析，进行各种不同的处理，然后生成jQuery对象。

|  |  |  |
| --- | --- | --- |
| **类型（selector）** | **处理方式** |  |
| 空 | 传过来如果为假,返回jQuery对象 | $(""), $(null), $(undefined), $(false) |
| DOM元素 | 包装成jQuery对象，直接返回 | $(document) |
| body（优化） | 从document.body读取 |  |
| 单独的HTML标签 | document.createElement | $(‘div’)、$(‘<div>’) |
| HTML字符串 | document.createDocumentFragment |  |
| #id | document.getElementById | $(‘#test’) |
| 选择器表达式 | $(…).find |  |
| 函数 | 注册到dom ready的回调函数 | $(function(){}) |

**工欲善其事，必先利其器，所以先从正则入手**

我们来分解一个表达式

// A simple way to check for HTML strings

// Prioritize #id over <tag> to avoid XSS via location.hash (#9521)

// Strict HTML recognition (#11290: must start with <)

rquickExpr = /^(?:\s\*(<[\w\W]+>)[^>]\*|#([\w-]\*))$/,

作者的解释呢很简单，一个简单的检测HTML字符串的表达式

分解：

1. **通过选择|分割二义,匹配^开头或者$结尾**

* ^(?:\s\*(<[\w\W]+>)[^>]\*
* #([\w-]\*))$

2. **^(?:\s\*(<[\w\W]+>)[^>]\***

* (?:*pattern*) : 匹配 *pattern* 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用
* \s\* : 匹配任何空白字符，包括空格、制表符、换页符等等 零次或多次 等价于{0,}
* (*pattern*) : 匹配*pattern* 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，使用 **$0**…**$9** 属性
* [\w\W]+ : 匹配于'[A-Za-z0-9\_]'或[^A-Za-z0-9\_]' 一次或多次， 等价{1,}
* (<[wW]+>) :这个表示字符串里要包含用<>包含的字符，例如<p>,<div>等等都是符合要求的
* [^>]\* : 负值字符集合,字符串尾部是除了>的任意字符或者没有字符,零次或多次等价于{0,},

3. **#([\w-]\*))$**

* 匹配结尾带上#号的任意字符，包括下划线与-

4. 还要穿插一下**exec**方法

* 如果执行exec方法的正则表达式没有分组（没有括号括起来的内容），那么如果有匹配，他将返回一个只有一个元素的数组，这个数组唯一的元素就是该正则表达式匹配的第一个串;如果没有匹配则返回null。
* exec如果找到了匹配，而且包含分组的话，返回的数组将包含多个元素，第一个元素是找到的匹配，之后的元素依次为该匹配中的第一、第二...个分组（反向引用）

所以综合起来呢大概的意思就是：**匹配HTML标记和ID表达式（<前面可以匹配任何空白字符，包括空格、制表符、换页符等等）**

简单测试下：

var str = ' <div id=top></div>';

var match = rquickExpr.exec(str);

console.log(match)

//[" <div id=top></div>", "<div id=top></div>", undefined, index: 0, input: " <div id=top></div>"]

var str = '[?\f\n\r\t\v]<div id=top></div>';

var str = '#test';

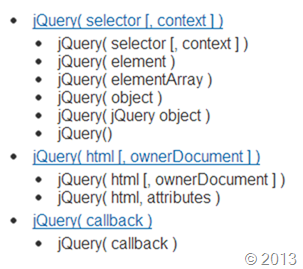
var match = rquickExpr.exec(str);

console.log(match)

//["#test", undefined, "test", index: 0, input: "#test"]

**jQuery选择器接口**

API

[](http://images.cnitblog.com/blog/329084/201308/26115936-6eaad157bdbf4942bd9a49f385edee20.png)

jQuery是总入口，选择器支持9种方式的处理

[复制代码](javascript:void(0);)

1.$(document)

2.$(‘<div>’)

3.$(‘div’)

4.$(‘#test’)

5.$(function(){})

6.$("input:radio", document.forms[0]);

7.$(‘input’, $(‘div’))

8.$()

9.$("<div>", {

"class": "test",

text: "Click me!",

click: function(){ $(this).toggleClass("test"); }

}).appendTo("body");

10$($(‘.test’))

[复制代码](javascript:void(0);)

jQuery这个选择器重构了几次后，现在逻辑结构相当的清晰了，一看大概就知道

不能不得说jQuery的反模式，非职责单一深受开发者喜欢，一个接口承载的职责越多内部处理就越复杂了

**jQuery查询的的对象是dom元素，查询到目标元素后，如何存储？**

* 查询的到结果储存到jQuery对象内部，由于查询的dom可能是单一元素，也可能是合集
* jQuery内部应该要定义一个合集数组，用于存在选择后的dom元素，
* 当然啦，根据API，jQuery构建的不仅仅只是DOM元素，还有HTML字符串,Object,[] 等等…

本质上讲jQuery.fn.init构建的出来的对象，就是jQuery对象

[复制代码](javascript:void(0);)

init: function( selector, context, rootjQuery ) {

var match, elem;

// HANDLE: $(""), $(null), $(undefined), $(false)

if ( !selector ) {

return this;

}

// Handle HTML strings

if ( typeof selector === "string" ) {

// HANDLE: $(DOMElement)

} else if ( selector.nodeType ) {

// HANDLE: $(function)

// Shortcut for document ready

} else if ( jQuery.isFunction( selector ) ) {

return rootjQuery.ready( selector );

}

if ( selector.selector !== undefined ) {

this.selector = selector.selector;

this.context = selector.context;

}

return jQuery.makeArray( selector, this );

},

[复制代码](javascript:void(0);)

源码缩进后的结构：

* 处理"",null,undefined,false,返回this ，增加程序的健壮性
* 处理字符串
* 处理DOMElement,返回修改过后的this,给this添加了
* 处理$(function(){})

**匹配模式一：$("#id")**

1. 进入字符串处理

if ( typeof selector === "string" ) {

2. 发现不是 "<"开始，">"结尾 $('<p id="test">My <em>new</em> text</p>')这种的情况

   如果selector是html标签组成的话，直接match = [ null, selector, null ];

   而不用正则检查

if (selector.charAt(0) === "<" && selector.charAt(selector.length - 1) === ">" && selector.length >= 3) {

3. 否则的话需要match = rquickExpr.exec( selector )

match = rquickExpr.exec( selector );

4. 匹配的html或确保没有上下文指定为# id

if ( match && (match[1] || !context) ) {

5. match[1]存在，处理$(html) -> $(array),,也就是处理的是html方式

if ( match[1] ) {

6. 处理ID

[复制代码](javascript:void(0);)

elem = document.getElementById( match[2] );

// Check parentNode to catch when Blackberry 4.6 returns

// nodes that are no longer in the document #6963

if ( elem && elem.parentNode ) {

// Inject the element directly into the jQuery object

this.length = 1;

this[0] = elem;

}

this.context = document;

this.selector = selector;

return this;

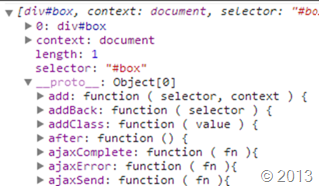
[复制代码](javascript:void(0);)

至此本次检索完毕!

可以看到

this就是jQuery工厂化后返回的对象

* this.length
* this[0] = elem
* this.context = document;
* this.selector = selector;

[](http://images.cnitblog.com/blog/329084/201308/26140234-81ce34c7aa324333ba3971b3431ad873.png)

**匹配模式二：<htmltag>**

重复的地方跳过直接看处理接口

[复制代码](javascript:void(0);)

if ( match && (match[1] || !context) ) {

// HANDLE: $(html) -> $(array)

if ( match[1] ) {

context = context instanceof jQuery ? context[0] : context;

// scripts is true for back-compat

jQuery.merge( this, jQuery.parseHTML(

match[1],

context && context.nodeType ? context.ownerDocument || context : document,

true

) );

// HANDLE: $(html, props)

if ( rsingleTag.test( match[1] ) && jQuery.isPlainObject( context ) ) {

for ( match in context ) {

// Properties of context are called as methods if possible

if ( jQuery.isFunction( this[ match ] ) ) {

this[ match ]( context[ match ] );

// ...and otherwise set as attributes

} else {

this.attr( match, context[ match ] );

}

}

}

return this;

// HANDLE: $(#id)

} else {

[复制代码](javascript:void(0);)

传入上下文：

context && context.nodeType ? context.ownerDocument || context : document

**ownerDocument和 documentElement的区别**

* ownerDocument是Node对象的一个属性，返回的是某个元素的根节点文档对象：即document对象
* documentElement是Document对象的属性，返回的是文档根节点
* 对于HTML文档来说，documentElement是<html>标签对应的Element对象，ownerDocument是document对象

具体请看API手册

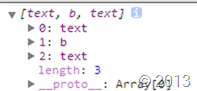
jQuery.merge( first, second ) 合并两个数组内容到第一个数组。

**jQuery.parseHTML**

使用原生的DOM元素的创建函数将字符串转换为一组DOM元素，然后，可以插入到文档中。

str = "hello, <b>my name is</b> jQuery.",

html = $.parseHTML( str ),

[](http://images.cnitblog.com/blog/329084/201308/26190029-eff6b92d5ea346229133b836ccd88800.png)

源码：

[复制代码](javascript:void(0);)

parseHTML: function( data, context, keepScripts ) {

if ( !data || typeof data !== "string" ) {

return null;

}

if ( typeof context === "boolean" ) {

keepScripts = context;

context = false;

}

context = context || document;

var parsed = rsingleTag.exec( data ),

scripts = !keepScripts && [];

// Single tag

if ( parsed ) {

return [ context.createElement( parsed[1] ) ];

}

parsed = jQuery.buildFragment( [ data ], context, scripts );

if ( scripts ) {

jQuery( scripts ).remove();

}

return jQuery.merge( [], parsed.childNodes );

},

[复制代码](javascript:void(0);)

匹配一个独立的标签

rsingleTag = /^<(\w+)\s\*\/?>(?:<\/\1>|)$/,

* ^<(\w+)\s\*\/?>  : 以<开头，至少跟着一个字符和任意个空白字符，之后出现0或1次/>
* (?:<\/\1>|)$        : 可以匹配<、一个/或者空白并以之为结尾

      这样如果没有任何属性和子节点的字符串（比如'<html></html>'或者'<div></div>'这样）会通过正则的匹配，当通过正则的匹配后则会通过传入的上下文直接创建一个节点：

只是单一的标签：

if ( parsed ) {

return [ context.createElement( parsed[1] ) ];

}

而未通过节点的字符串，则通过创建一个div节点，将字符串置入div的innerHTML：

parsed = jQuery.buildFragment( [ data ], context, scripts );

它会把传入的复杂的html转为文档碎片并且存储在jQuery.fragments这个对象里。这里要提一下，document.createDocumentFragment()是相当好用的，可以减少对dom的操作.

创建一个文档碎片DocumentFragment

* 如果要插入多个DOM元素，可以先将这些DOM元素插入一个文档碎片，然后将文档碎片插入文档中，这时插入的不是文档碎片，而是它的子孙节点；相比于挨个插入DOM元素，使用文档碎片可以获得2-3倍的性能提升；
* 如果将重复的HTML代码转换为DOM元素，可以将转换后的DOM元素缓存起来，下次（实际是第3次）转换同样的HTML代码时，可以直接缓存的DOM元素克隆返

当一个HTML比一个没有属性的简单标签复杂的时候，实际上，创建元素的处理是利用了浏览器的innerHTML 机制。

[复制代码](javascript:void(0);)

1 tmp = tmp || fragment.appendChild( context.createElement("div") );

2

3 // Deserialize a standard representation

4 tag = ( rtagName.exec( elem ) || ["", ""] )[ 1 ].toLowerCase();

5 wrap = wrapMap[ tag ] || wrapMap.\_default;

6 tmp.innerHTML = wrap[ 1 ] + elem.replace( rxhtmlTag, "<$1></$2>" ) + wrap[ 2 ];

[复制代码](javascript:void(0);)

* 特别说明，jQuery创建一个新的<div>元素，并且设置innerHTML属性为传入的HTML代码片段。当参数是一个单标签，就像 $('<img />') or $('<a></a>')，jQuery将使用javasrcipt原生的 createElement()函数创建这个元素。
* 当传入一个复杂的html，一些浏览器可能不会产生一个完全复制HTML源代码所提供的DOM。正如前面提到的，jQuery使用的浏览器.innerHTML属性来解析传递的HTML并将其插入到当前文档中。在此过程中，一些浏览器过滤掉某些元素，如<html>, <title>, 或 <head>的元素。其结果是，被插入元素可能不是传入的原始的字符串。
* 不过，这些被过滤掉的标签有限的。有些浏览器可能不完全复制所提供的HTML源代码生成DOM。例如，Internet Explorer的版本8之前转换所有链接的href属性为绝对URL路径，和Internet Explorer第9版之前，不增加一个单独的[兼容层](http://code.google.com/p/html5shiv/)的情况下，将无法正确处理HTML5元素。
* 为了确保跨平台的兼容性，代码片段必须是良好的。标签可以包含其他元素，但需要搭配的结束标记

 如果第一个参数(HTML字符串)为一个空的单标签，且第二个参数context为一个非空纯对象

|  |
| --- |
| var jqHTML = $('<div></div>', { class: 'css-class', data-name: 'data-val' });    console.log(jqHTML.attr['class']); //css-class  console.log(jqHTML.attr['data-name']); //data-val |

**匹配模式三：$(.className)**

如果第一个参数是一个.className，jQuery对象中拥有class名为className的标签元素，并增加一个属性值为参数字符串、document的selector、context属性

|  |
| --- |
| return jQuery(document).find(className); |

**匹配模式四：$(.className, context)**

如果第一个参数是.className，第二个参数是一个上下文对象(可以是.className(等同于处理$(.className .className))，jQuery对象或dom节点)，    
jQuery对象包含第二个参数上下文对象中拥有class名为className的后代节点元素，并增加一个context和selector属性

return jQuery(context).find(className);

**匹配模式五：$(jQuery对象)**

如果第一个参数是jQuery对象，上面已经分析过如果在查询dom时，参数是一个#加元素id，返回的jQuery对象会增加一个属性值为参数字符串、document的selector、context属性

var jq = $('#container');

console.log(jq.selector); // #container

console.log(jq.context); // document

那么当出现$($('#container'))该如何处理呢？同样的，返回的jQuery对象同情况

var jq2 = $($('#container'));

console.log(jq2.selector); // #container

console.log(jq2.context); // document

等等..................

**jQuery 构造器**

     由此可见，从本质上来说，构建的jQuery对象，其实不仅仅只是dom，还有很多附加的元素，用数组的方式存储，当然各种组合有不一样，但是存储的方式是一样的

总的来说分2大类：

* 单个DOM元素，如$(ID),直接把DOM元素作数组传递给this对象
* 多个DOM元素，集合形式，可以通过CSS选择器匹配是有的DOM元素，过滤操作,构建数据结构

CSS选择器是通过jQuery.find(selector)函数完成的，通过它可以分析选择器字符串，并在DOM文档树中查找符合语法的元素集合

选择器这章有点乱，东西太多了，不能一一陈列 , 后期在慢慢整理

## ****jQuery的类数组对象结构****

为什么是类数组对象呢？

很多人迷惑的jQuery为什么能像数组一样操作，通过对象get方法或者直接通过下标0索引就能转成DOM对象。

首先我们看jQuery的入口都是统一的$, 通过传递参数的不同，实现了9种方法的重载：

**1. jQuery([selector,[context]])**

**2. jQuery(element)**

**3. jQuery(elementArray)**

**4. jQuery(object)**

**5. jQuery(jQuery object)**

**6. jQuery(html,[ownerDocument])**

**7. jQuery(html,[attributes])**

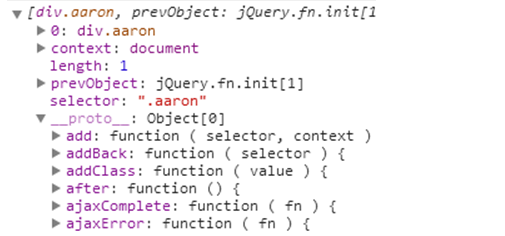
**8. jQuery()**

**9. jQuery(callback)**

9种用法整体来说可以分三大块：选择器、dom的处理、dom加载。  
换句话说jQuery就是为了获取DOM、操作DOM而存在的！所以为了更方便这些操作，让节点与实例对象通过一个桥梁给关联起来，jQuery内部就采用了一种叫“类数组对象”的方式作为存储结构，所以我们即可以像对象一样处理jQuery操作，也能像数组一样可以使用push、pop、shift、unshift、sort、each、map等类数组的方法操作jQuery对象了。

**jQuery对象可用数组下标索引是什么原理？**

通过$(".Class")构建的对象结构如下所示：

[](http://img.mukewang.com/53fad4240001c7b805050236.jpg)  
                     
整个结构很明了，通过对象键值对的关系保存着属性，原型保存着方法。

## ****jQuery中ready与load事件****

jQuery有3种针对文档加载的方法

$(document).ready(function() {

// ...代码...

})

//document ready 简写

$(function() {

// ...代码...

})

$(document).load(function() {

// ...代码...

})

**一个是ready一个是load，这两个到底有什么区别呢？**

ready与load谁先执行：  
大家在面试的过程中，经常会被问到一个问题：ready与load那一个先执行，那一个后执行？答案是ready先执行，load后执行。

**DOM文档加载的步骤：**  
要想理解为什么ready先执行，load后执行就要先了解下DOM文档加载的步骤：

(1) 解析HTML结构。

(2) 加载外部脚本和样式表文件。

(3) 解析并执行脚本代码。

(4) 构造HTML DOM模型。//ready

(5) 加载图片等外部文件。

(6) 页面加载完毕。//load

从上面的描述中大家应该已经理解了吧，ready在第（4）步完成之后就执行了，但是load要在第（6）步完成之后才执行。

**结论：**

ready与load的区别就在于资源文件的加载，ready构建了基本的DOM结构，所以对于代码来说应该越快加载越好。在一个高速浏览的时代，没人愿意等待答案。假如一个网站页面加载超过4秒，不好意思，你1/4的用户将面临着流失，所以对于框架来说用户体验是至关重要的，我们应该越早处理DOM越好，我们不需要等到图片资源都加载后才去处理框架的加载，图片资源过多load事件就会迟迟不会触发。

我们看看jQuery是如何处理文档加载时机的问题：

jQuery.ready.promise = function( obj ) {

if ( !readyList ) {

readyList = jQuery.Deferred();

if ( document.readyState === "complete" ) {

// Handle it asynchronously to allow scripts the opportunity to delay ready

setTimeout( jQuery.ready );

} else {

document.addEventListener( "DOMContentLoaded", completed, false );

window.addEventListener( "load", completed, false );

}

}

return readyList.promise( obj );

};

jQuery的ready是通过promise给包装过的，这也是jQuery擅长的手法，统一了回调体系，以后我们会重点谈到。  
可见jQuery兼容的具体策略**：针对高级的浏览器，我们当前很乐意用DOMContentLoaded事件了，省时省力。**

**那么旧的IE如何处理呢？**

继续看jQuery的方案：

// Ensure firing before onload, maybe late but safe also for iframes

document.attachEvent( "onreadystatechange", completed );

// A fallback to window.onload, that will always work

window.attachEvent( "onload", completed );

// If IE and not a frame

// continually check to see if the document is ready

var top = false;

try {

top = window.frameElement == null && document.documentElement;

} catch(e) {}

if ( top && top.doScroll ) {

(function doScrollCheck() {

if ( !jQuery.isReady ) {

try {

// Use the trick by Diego Perini

// http://javascript.nwbox.com/IEContentLoaded/

top.doScroll("left");

} catch(e) {

return setTimeout( doScrollCheck, 50 );

}

// detach all dom ready events

detach();

// and execute any waiting functions

jQuery.ready();

}

})();

}

    如果浏览器存在 document.onreadystatechange 事件，当该事件触发时，如果 document.readyState=complete 的时候，可视为 DOM 树已经载入。不过，这个事件不太可靠，比如当页面中存在图片的时候，可能反而在 onload 事件之后才能触发，换言之，它只能正确地执行于页面不包含二进制资源或非常少或者被缓存时作为一个备选吧。

**针对IE的加载检测**

Diego Perini 在 2007 年的时候，报告了一种检测 IE 是否加载完成的方式，使用 doScroll 方法调用，详情可见http://javascript.nwbox.com/IEContentLoaded/。  
原理就是对于 IE 在非 iframe 内时，只有不断地通过能否执行 doScroll 判断 DOM 是否加载完毕。在上述中间隔 50 毫秒尝试去执行 doScroll，注意，由于页面没有加载完成的时候，调用 doScroll 会导致异常，所以使用了 try -catch 来捕获异常。  
结论：所以总的来说当页面 DOM 未加载完成时，调用 doScroll 方法时，会产生异常。那么我们反过来用，如果不异常，那么就是页面DOM加载完毕了。

这都是我们在第一时间内处理ready加载的问题，如果ready在页面加载完毕后呢？

jQuery就必须针对这样的情况跳过绑定了：

if ( document.readyState === "complete" ) {

// Handle it asynchronously to allow scripts the opportunity to delay ready

setTimeout( jQuery.ready );

}

直接通过查看readyState的状态来确定页面的加载是否完成了。这里会给一个定时器的最小时间后去执行，主要保证执行的正确。

## ****jQuery多库共存处理****

多库共存换句话说可以叫无冲突处理。

总的来说会有2种情况会遇到：

   1、$太火热，jQuery采用$作为命名空间，不免会与别的库框架或者插件相冲突。

   2、jQuery版本更新太快，插件跟不上，导致不同版本对插件的支持度不一样。

出于以上的原因，jQuery给出了解决方案–– noConflict函数。

    引入jQuery运行这个noConflict函数将变量$的控制权让给第一个实现它的那个库，确保jQuery不会与其他库的$对象发生冲突。  
在运行这个函数后，就只能使用jQuery变量访问jQuery对象。例如，在要用到$("aaron")的地方，就必须换成jQuery("aaron")，因为$的控制权已经让出去了。

使用DEMO：

jQuery.noConflict();

// 使用 jQuery

jQuery("aaron").show();

// 使用其他库的 $()

$("aaron").style.display = ‘block’;

   这个函数必须在你导入jQuery文件之后，并且在导入另一个导致冲突的库之前使用。当然也应当在其他冲突的库被使用之前，除非jQuery是最后一个导入的。

由于比较简单，我们直接上代码解说：

Var \_jQuery = window.jQuery,

\_$ = window.$;

jQuery.noConflict = function( deep ) {

if ( window.$ === jQuery ) {

window.$ = \_$;

}

if ( deep && window.jQuery === jQuery ) {

window.jQuery = \_jQuery;

}

return jQuery;

};

    如果我们需要同时使用jQuery和其他JavaScript库，我们可以使用 $.noConflict()把$的控制权交给其他库。旧引用的$ 被保存在jQuery的初始化; noConflict() 简单的恢复它们。  
    通过类似swap交换的概念，先把之前的存在的命名空间给缓存起来，通过对比当前的命名空间达到交换的目的，首先，我们先判断下当前的的$空间是不是被jQuery接管了，如果是则让出控制权给之前的\_$引用的库，如果传入deep为true的话等于是把jQuery的控制权也让出去了。  
    如果不通过noConflict处理的话其后果可想而知，香喷喷的$大家都“觊觎已久”。

## ****工具方法****

以下方法是在jQuery的core定义的工具方法（可以去github的[jQuery项目](https://github.com/jquery/jquery)），core是整个jQuery最核心的组成部分，所以从这部分先剖析：

$.trim() 去除字符串两端的空格。（内部调用7次）

$.each() 遍历数组或对象，这个方法在jQuery内部中被使用很多次，有几个不错的用法，之后剖析再举例吧。（内部调用59次）

$.inArray() 返回一个值在数组中的索引位置。如果该值不在数组中，则返回-1。（内部调用9次）

$.grep() 返回数组中符合某种标准的元素。（内部调用6次）

$.merge() 合并两个数组。（内部调用11次）

$.map() 将一个数组中的元素转换到另一个数组中。（内部调用12次）

$.makeArray() 将对象转化为数组。（内部调用6次）

$.globalEval() 在全局作用域下执行一段js脚本。（内部调用2次）

$.proxy() 接受一个函数，然后返回一个新函数，并且这个新函数始终保持了特定的上下文(context)语境。（内部调用0次）

$.nodeName() 返回DOM节点的节点名字，或者判断DOM节点名是否为某某名字。（内部调用51次）

$.extend() 将多个对象，合并到第一个对象。（内部调用42次）

以下均是对类型的判断，本文只是针对$.type做一下讨论，isXXX的方法基本都是调用$.type来实现，不对它们做细节探讨。

$.type() 判断对象的类别（函数对象、日期对象、数组对象、正则对象等等）。这个方法的实现就是用$.each辅助的。（内部调用65次）

$.isArray() 判断某个参数是否为数组。（内部调用12次）

$.isEmptyObject() 判断某个对象是否为空（不含有任何属性）。（内部调用4次）

$.isFunction() 判断某个参数是否为函数。（内部调用32次）

$.isPlainObject() 判断某个参数是否为用”{}”或”new Object”建立的对象。（内部调用4次）

$.isWindow() 判断是否为window对象。（内部调用6次）

以下三个函数比较简单，没必要在文章剖析。

$.noop() 一个空函数，个人觉得是用来作为一个默认的回调函数，无需每次去定义一个空的function消耗资源。（内部调用2次）

$.now() 获取当前时间戳，代码很简单：return (new Date()).getTime();。（内部调用4次）

$.error() 报错，对外抛出一个异常，代码很简单：throw new Error(msg);。（内部调用2次）

以下三个是jQuery主要用来在ajax处理返回数据时使用，其中parseJSON这个接口在实际工程中被用得最多，经常用来把一段文本解析成json格式

$.parseHTML() 解析HTML，之后再单独一节写。（内部调用2次）

$.parseJSON() 解析JSON，之后再单独一节写。（内部调用2次）

$.parseXML() 解析XML，之后再单独一节写。（内部调用1次）

其中我认为是内部辅助函数如下：

$.access() 这个函数我更认为是jQuery内部的辅助函数，没必要暴漏出来，在内部用于去一些对象的属性值等，在之后剖析到DOM操作等再细细探讨一下。（内部调用9次）

$.camelCase() 转化成骆驼峰命名。（内部调用12次）

### isFunction ()

首先就告诉你自从1.3版本就有bug ,一些dom方法和函数例如alert在ie里面会返回false,看了下这个bug，因为toString方法和valueOf方法都会被重写所以有人就提出了用instanceof方法检测但是在ie6还是有问题。目前为止这个bug还没有关闭具体大家可以从参考[官网bug页](http://bugs.jquery.com/ticket/2968)

### isPlainObject()

**Description:***Check to see if an object is a plain object (created using "{}" or "new Object").*

测试对象是否是纯粹的对象（通过 "{}" 或者 "new Object" 创建的）

该方法在 jQuery 社区存在不少[质疑](http://api.jquery.com/jQuery.isPlainObject/#comment-31062800)。代码实现上，不断有各种[改进方案](http://forum.jquery.com/topic/improvement-to-isplainobject)涌现。甚至，从无 bug 的角度讲，该问题是[无解](http://www.cnblogs.com/fool/archive/2010/10/17/1853813.html)的。

isPrototypeOf只会存在于Object.prototype上，也就是说如果obj的原型中存在isPrototypeOf则obj是Object的直接实例。当然除非人为的将obj的构造函数的prototype.constructor置为Object，或者将构造函数的prototype置为{}。

// 漏网之鱼

jQuery.isPlainObject(location); // true in ie8-

// 自杀绝技

function Kill() {}

Kill.prototype = Object;

jQuery.isPlainObject(new Kill()); // true

// 万箭齐发

Object.prototype.ninja = 1;

jQuery.isPlainObject({}); // false

**场景挖掘**

发现这篇文章很难结尾，赶紧回归，来段模拟问答：

Q: 在什么情况下使用 isPlainObject?   
A: 在判断一个对象是不是对象字面量时使用。

Q: 什么情况下需要判断一个对象是对象字面量？   
A: 要对该对象进行一些操作，如果该对象不是对象字面量，后续操作可能会出错。

Q: 能否详细说明在哪些情况下会出错？   
A: 当该对象是 null/undefined/string 等情况时，会导致 merge config 出错。

Q: 还有哪些应用场景？   
A: 有时会传入 this 当 config, this 有可能为 window 或 dom 节点，要排除掉。

Q: 还有哪些场景吗？   
A: 想不起来了。

通过上面这种问答，弄清楚实际使用场景后，很容易写出“完美”代码：

// 版本 B

function isPlainObject(o) {

return o &&

// 排除 boolean/string/number/function 等

// 标准浏览器下，排除 window 等非 JS 对象

// 注：ie8- 下，toString.call(window 等对象) 返回 '[object Object]'

Object.prototype.toString.call(o) === '[object Object]' &&

// ie8- 下，排除 window 等非 JS 对象

('isPrototypeOf' in o);

}

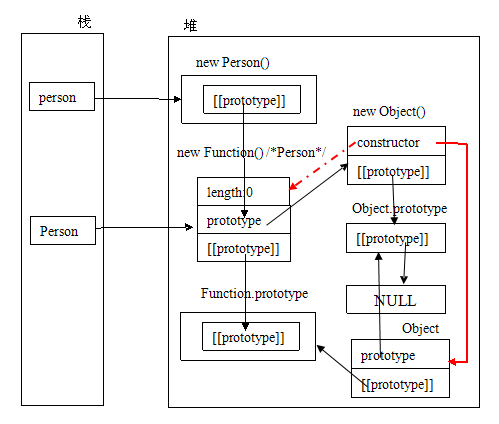
这样，从使用场景出发，isPlainObject 的功用也发生了变化： 检测一个对象是不是 JavaScript 语言内部的纯对象（{} 或 new Object()）或函数的实例对象。

**解答无解**

本以为这个问题很好解决，结果深入后，发现这是一个无解的问题。原因如下：

|  |  |
| --- | --- |
| 1  2  3 | function Person(){};  Person.prototype.constructor=Object;  var person=new Person; |

让我们来看一下person现在的状态：



person和其构造函数Person唯一的联系就是其prototype链中的constructor属性。而在我们判断是否为'纯粹的对象'主要是依据对象实例的constructor进行的。如果我们将其指向Object，正如图中看到的那样，那么person和Person在代码上就没有关系了。也正是因为这一点，让类型的判断出现了问题。

### type()

jQuery.type方法是检测数据类型的工具方法，在分析其用法之前先总结下js给我们提供了那些监测数据类型的方法；

**一、typeof 操作符**

下面是测试代码

var data=[],a='123',b=0,c=true,d={1:23},e=[123],f=function(){},g=null,h=undefined,i=Math,j=/$.+^/,k= new Date();

data.push(a,b,c,d,e,f,g,h,i,j,k);

for(var key=0;key<data.length;key++){

console.log(data[key]+'的数据类型是'+typeof data[key]);

}

在上面的代码中我尽量列举了js不同的数据类型和对象，执行结果如下：

[复制代码](javascript:void(0);)

//123的数据类型是string

// 0的数据类型是number

// true的数据类型是boolean

//[object Object]的数据类型是object

//123的数据类型是object

// function (){}的数据类型是function

//undefined的数据类型是undefined

// [object Math]的数据类型是object

// /$.+^/的数据类型是object

// Wed Jul 22 2015 15:47:25 GMT+0800 (中国标准时间)的数据类型是object

[复制代码](javascript:void(0);)

我们可以看到typeof能检测到js的6大基本类型中的5个，即String,Boolean,Number,Undefined,Object  其中null被归为了Object把Function单独拿了出来，基本上还是能完成任务的，但那是对于复合类型而言就无法进一步区分了，比如到底是数组还是对象呢？这个时候就可以利用另外一个操作符instanceOf了

**二、instanceOf操作符**

同样的写一段测试代码

[复制代码](javascript:void(0);)

var data=[],a='123',b=0,c=true,d={1:23},e=[123],f=function(){},g=null,h=undefined,i=Math,j=/$.+^/,k= new Date();

data.push(a,b,c,d,e,f,g,h,i,j,k);

console.log(a instanceof String);

console.log(b instanceof Number);

console.log(c instanceof Boolean);

console.log(d instanceof Object);

console.log(e instanceof Array);

console.log(f instanceof Function);

console.log(j instanceof RegExp);

console.log(k instanceof Date);

[复制代码](javascript:void(0);)

在浏览器中的运行结果如下:

[复制代码](javascript:void(0);)

// false

// false

// true

// true

// true

// true

// true

[复制代码](javascript:void(0);)

可以看到只有复合类型的结果为真，而且必须保证类型是一一对应的，显然这个方法只能做个一检验方法存在，并不能在我们不知道具体数据类型的时候去做判断，可以作为typeof的一个辅助测试手段

**三、constructor属性**

同样的先写下测试代码

[复制代码](javascript:void(0);)

var data=[],a='123',b=0,c=true,d={1:23},e=[123],f=function(){},g=null,h=undefined,i=Math,j=/$.+^/,k= new Date();

data.push(a,b,c,d,e,f,g,h,i,j,k);

for(var key=0;key<data.length;key++){

try{

console.log(data[key]+'的检测结果是' +data[key].constructor);

}catch(e){

}

}

[复制代码](javascript:void(0);)

运行结果如下：

[复制代码](javascript:void(0);)

//123的检测结果是function String() { [native code] }

//0的检测结果是function Number() { [native code] }

//true的检测结果是function Boolean() { [native code] }

//[object Object]的检测结果是function Object() { [native code] }

//123的检测结果是function Array() { [native code] }

//function (){}的检测结果是function Function() { [native code] }

//[object Math]的检测结果是function Object() { [native code] }

///$.+^/的检测结果是function RegExp() { [native code] }

//Wed Jul 22 2015 16:23:41 GMT+0800 (中国标准时间)的检测结果是function Date() { [native code] }

[复制代码](javascript:void(0);)

其中null调用是会报错所以加了try语句，相对而言此方法能够很方便的获取其构造函数，这样就能判断了，遗憾的是该属性并非是只读属性是可以被修改的，一旦被修改或者涉及到对象继承等问题时会导致不准而且在遇到某些值得时候会报错导致程序无法运行比如null，还有没有其他方法呢？

**四、Object.prototype.toString方法**

该方法通过调用待测试数据的toString方法来获得其构造函数的字符串表示,测试代码如下：

var data=[],a='123',b=0,c=true,d={1:23},e=[123],f=function(){},g=null,h=undefined,i=Math,j=/$.+^/,k= new Date();

data.push(a,b,c,d,e,f,g,h,i,j,k);

for(var key=0;key<data.length;key++){

console.log(data[key]+'的检测结果是' +Object.prototype.toString.call(data[key]));

}

运行结果如下：

[复制代码](javascript:void(0);)

//123的检测结果是[object String]

//0的检测结果是[object Number]

//true的检测结果是[object Boolean]

//[object Object]的检测结果是[object Object]

//123的检测结果是[object Array]

//function (){}的检测结果是[object Function]

//null的检测结果是[object Null]

//undefined的检测结果是[object Undefined]

//[object Math]的检测结果是[object Math]

///$.+^/的检测结果是[object RegExp]

//Wed Jul 22 2015 16:33:05 GMT+0800 (中国标准时间)的检/测结果是[object Date]

[复制代码](javascript:void(0);)

看到结果是不是感觉很爽！不仅可以检测到所有数据类型，而且把Object子类型也现实了出来，也不用担心报错，之所以能实现是因为所有的对象都是基于Object而来的，其实jQery也是采取的这个方法，只不过是做了进一步处理让我们看着更爽而已！

看看使用jQuery.type的结果

var data=[],a='123',b=0,c=true,d={1:23},e=[123],f=function(){},g=null,h=undefined,i=Math,j=/$.+^/,k= new Date();

data.push(a,b,c,d,e,f,g,h,i,j,k);

for(var key=0;key<data.length;key++){

console.log(data[key]+'的检测结果是' +$.type(data[key]));

}

运行结果:

[复制代码](javascript:void(0);)

//123的检测结果是string

//0的检测结果是number

//的检测结果是boolean

//[object Object]的检测结果是object

//123的检测结果是array

//function (){}的检测结果是function

//null的检测结果是null

//undefined的检测结果是undefined

//[object Math]的检测结果是object

///$.+^/的检测结果是regexp

//Wed Jul 22 2015 16:44:25 GMT+0800 (中国标准时间)的检测结果是date

[复制代码](javascript:void(0);)

ok，结果无可挑剔了，下面附上源码：

type: function( obj ) {

return obj == null ?

String( obj ) :

class2type[ toString.call(obj) ] || "object";

},

如果是undefined或者是null他们的数据累类型就是自己，直接返回字符串形式，如果是其他数据就执行toString方法，该方法在之前有介绍

toString = Object.prototype.toString,

返回的结果就像之前测试过的结果类似  [object Date]这样的 如果不能取到就返回object，结果作class2type的键，下面来看下class2type的定义：

// Populate the class2type map

jQuery.each("Boolean Number String Function Array Date RegExp Object".split(" "), function(i, name) {

class2type[ "[object " + name + "]" ] = name.toLowerCase();

});

好了type方法分析完毕。

### globalEval()

此方法的表现不同于正常使用的JavaScript eval()，因为它是在全局上下文下执行（这对加载外部动态脚本很重要）。

示例：

var name = "全局变量";

function test(){

var name = "局部变量";

alert(name); // 局部变量

eval( "alert(name);" ); // 局部变量

$.globalEval( "alert(name);" ); // 全局变量

}

test();

/\*\*  
 \* eval与jQuery.globalEval的不同之处: 作用域  
 \*   
 \* @author <http://www.douban.com/people/ufologist/>  
 \* @see <http://weblogs.java.net/blog/driscoll/archive/2009/09/08/eval-javascript-global-context>  
 \*/  
(function() {  
    var foo = "foo";  
    // eval的作用域是当前作用域(在这里就是local也就是function作用域),  
    // 因此既能访问local也能访问global作用域  
    eval("alert('eval: ' + foo)");  
})();  
  
(function() {  
    var foo = "foo";  
    // jQuery.globalEval的作用域是global, 因此只能访问global作用域的变量  
    // 这里的foo是local变量, 因此在global作用域下找不到  
    // ReferenceError: foo is not defined  
    $.globalEval("alert('globalEval: ' + foo)");  
})();

### access()

access() 是个很巧妙的方法，它被放置于 core.js 中，作为静态方法存在。

jQuery 是以 DOM 节点为核心，当你使用 $ 获取到对应的 DOM 节点集合后，一切的操作基本上都会围绕这些节点展开。

我粗略的将 jQuery 针对 DOM 节点的操作方法分为两种，一种是 get，获取节点信息，一种是 set，设置节点信息。

对于 get，通常只需要获取节点集合中的第一个元素，例如你要获取元素的高度、宽度、颜色等等，你通常只会得到一个值。

而对于 set，则会对整个集合中的节点生效，例如：

$('div').height(100);

这便将所有 div 的高度设置为 100 像素。

既然一切围绕着节点的集合来操作，那么循环集合便是不可避免的，但如果在每个方法里面都写个 for 循环，是在是有些难堪。这些循环的内容从形式上基本相同，放到每个方法里面纯属冗余，还不如单独提取出来，在一个新的方法里面对集合做操作。这就是 access() 方法的作用。

先来看看 access() 的参数列表：

access: function( elems, fn, key, value, chainable, emptyGet, raw )

elems 就是要循环的节点集合。

fn 是需要对节点进行操作的函数。

key 是属性名，例如 $('#test').height(); 这里的 height(字符串)。

value 是值，例如 $('#test').height(100); 中的 100。

chainable 表示是否链式执行，对于 get 类方法，我们会获得一个返回值，例如字符串、数字等等，这时候是不需要链式执行的，而对于 set 类方法，通常需要如此，例如：

$('#test').height(100).width(100).css('color', 'red');

emptyGet 用于节点集合中没有元素时返回的默认值。

raw 为 true，表明 value 是个函数，你经常会在 jQuery 的 API 中看到参数可以为函数，举个我都快举烂的例子，height() 方法，点击<http://api.jquery.com/height/> 查看 height() 的文档，你会看到，

.height( function(index, height) )

raw 就是用于区分这种参数的。

## ****精妙设计—存储Array的concat方法****

/\*

需要调用concat时可以通过以下方法调用，关于call跟apply的用法自行理解，:)

var arr = [];

方式一：arr.concat();

方式二：core\_concat.call(arr);

方式三：core\_concat.apply(arr);

思考下边2个问题：

jQuery为什么要先把这些方法存储起来？

jQuery为什么要采用方式二或者三，而不直接使用方式一的做法？

在不查阅资料的前提下，唯一让我觉得作者这么做的原因是因为效率问题。

以下是我的理解：

调用实例arr的方法concat时，首先需要辨别当前实例arr的类型是Array，在内存空间中寻找

Array的concat内存入口，把当前对象arr的指针和其他参数压入栈，跳转到concat地址开始执行。

当保存了concat方法的入口core\_concat时，完全就可以省去前面两个步骤，从而提升一些性能。

nodejser在评论中也给出了另一种答案：

var obj = {};

此时调用obj.concat是非法的，但是如果jQuery采用上边方式二或者三的话，能够解决这个问题。

也即是让类数组也能用到数组的方法（这就是call跟apply带来的另一种用法），尤其在jQuery里

边引用一些DOM对象时，也能完美的用这个方法去解决，妙！

\*/

var arr = [];

var slice = arr.slice;

var concat = arr.concat;

var push = arr.push;

var indexOf = arr.indexOf;

var class2type = {};

var toString = class2type.toString;

var hasOwn = class2type.hasOwnProperty;

## **Callbacks**

## **Deferred**

## E**vent事件**

**jQuery事件处理工具函数**

// 事件管理的工具函数

jQuery**.**event **=** **{**

global**:** **{},**

// 绑定事件函数

add**:** ***function*()** **{},**

// 移除事件函数

remove**:** ***function*()** **{},**

// 主动触发事件函数

trigger**:** ***function*()** **{},**

// 事件分发函数

dispatch**:** ***function*()** **{},**

// 修复jQuery.Event对象的函数

fix**:** ***function*()** **{},**

// 包含一些事件常用属性，让KeyEvent和MouseEvent共享

// 新建JQ事件对象的时候，这些属性就会被扩展进去

props**:** "altKey bubbles cancelable ctrlKey currentTarget eventPhase metaKey relatedTarget shiftKey target timeStamp view which"**.**split**(**" "**),**

// fix的勾子，应该是提供自定义勾子使用的，如果不存在的话，fix方法里面会使用keyHooks跟mouseHooks

fixHooks**:** **{},**

// 键盘事件勾子

keyHooks**:** **{},**

// 鼠标事件勾子

mouseHooks**:** **{},**

// 特殊事件类型的一些属性跟方法，这些属性跟方法均在前面的jQuery.event中被使用到，应该可以看做是跟勾子类似的对象

special**:** **{},**

// simulate这个方法暂时也是不理解其用途

// 貌似是模仿一些具体类型的事件，然后触发

simulate**:** ***function*()** **{}**

**};**

// 删除事件监听，在jQuery.event.remove里会用到

jQuery**.**removeEvent **=** ***function*()** **{};**

**jQuery事件对象**

// 构造函数

jQuery**.**Event **=** ***function*(** src**,** props **)** **{};**

// jQuery.Event是基于ECMAScript标准的DOM3级事件开发的

jQuery**.**Event**.*prototype*** **=** **{**

// 定义三个函数属性，值全是returnFalse

isDefaultPrevented**:** returnFalse**,**

isPropagationStopped**:** returnFalse**,**

isImmediatePropagationStopped**:** returnFalse**,**

// jQuery.Event的preventDefault方法

preventDefault**:** ***function*()** **{},**

// jQuery.Event的stopPropagation方法

stopPropagation**:** ***function*()** **{},**

// stopImmediatePropagation的作用是使阻止事件冒泡的同时，阻止该元素绑定的该事件类型的其他回调函数的执行

stopImmediatePropagation**:** ***function*()** **{}**

**};**

**jQuery事件处理实例函数**

// 实例对象的事件处理方法

jQuery**.**fn**.**extend**({**

// 绑定事件方法on（参数one仅限是内部使用，JQ实例对象需要的话可以直接调用后面的one方法）

on**:** ***function*(** **)** **{},**

// one方法，表明事件回调只处理一次

one**:** ***function*(** types**,** selector**,** data**,** fn **)** **{},**

// 解除事件绑定off

off**:** ***function*()** **{},**

// 触发事件方法trigger

trigger**:** ***function*()** **{},**

// 会影响到前面event.isTrigger的值，用途暂时不明

triggerHandler**:** ***function*()** **{}**

**});**

**流程简介**

工具函数无疑是服务于实例函数的，每个JQ实例对象都可以调用on，off，trigger等方法去执行相应的事件处理操作，实例函数最终调用工具函数实现功能，而jQuery.Event则是jQuery工具函数处理事件时用到的模型，关系可简化如下：

<pre name=**"code"** class=**"javascript"**>**jQuery.fn.on|off|trigger ->** <span style=**"font-family: Arial, Helvetica, sans-serif;"**>**jQuery.event.add|remove|triiger(jQuery.Event)**</span>

**引入jQuery的Special Event机制**

什么时候要用到自定义函数？有些浏览器并不兼容某类型的事件，如IE6～8不支持hashchange事件，你无法通过jQuery(window).bind('hashchange', callback)来绑定这个事件，这个时候你就可以通过jQuery自定义事件接口来模拟这个事件，做到跨浏览器兼容。

**原理**

jQuery(elem).bind(type, callbakc)实际上是映射到 jQuery.event.add(elem, types, handler, data)这个方法，每一个类型的事件会初始化一次事件处理器，而传入的回调函数会以数组的方式缓存起来，当事件触发的时候处理器将依次执行这个数组。  
jQuery.event.add方法在第一次初始化处理器的时候会检查是否为自定义事件，如果存在则将会把控制权限交给自定义事件的事件初始化函数，同样事件卸载的jQuery.event.remove方法在删除处理器前也会检查此。

!special.setup || special.setup.call( elem, data, namespaces, eventHandle ) === false

jQuery.removeEvent( elem, type, elemData.handle );

jQuery.event.special对象中，保存着为适配特定事件所需的变量和方法，

具体有:  
delegateType / bindType （用于事件类型的调整）  
setup （在某一种事件第一次绑定时调用）  
add （在事件绑定时调用）  
remove （在解除事件绑定时调用）  
teardown （在所有事件绑定都被解除时调用）  
trigger （在内部trigger事件的时候调用）  
noBubble  
\_default  
handle （在实际触发事件时调用）  
preDispatch （在实际触发事件前调用）  
postDispatch （在实际触发事件后调用）

在适配工作完成时，会产生一个handleObj对象，这个对象包含了所有在事件实际被触发是所需的所有参数

**采用自定义事件或者浏览器接口绑定事件**

if ( !special.setup || special.setup.call( elem, data, namespaces, eventHandle ) === false ) {

if ( elem.addEventListener ) {

elem.addEventListener( type, eventHandle, false );

}

}

**冒泡标记**

  handlers.splice( handlers.delegateCount++, 0, handleObj );