jQuery源码分析系列

作者: nuysoft http://nuysoft.iteye.com

目 录

1. JavaScript

1.1 [原创] jQuery源码分析系列目录（持续更新）

发表时间: 2011-10-12 关键字: javascript, jquery, 源码分析

作者：nuysoft/高云　QQ：47214707　EMail：nuysoft@gmail.com

[原创] jQuery源码分析（版本1.6.1）

附录：

TODO：

14 CSS操作 CSS

1.2 [原创] jQuery源码分析-00前言开光

发表时间: 2011-09-21 关键字: javascript, jquery, 源码

jQuery源码分析 - 前言

jQuery凭借简洁的语法和跨平台的兼容性，极大地简化了JavaScript开发开发人员遍历HTML文档、操作DOM、处理事件、执行动画和开发Ajax的操作。其独特而又优雅的代码风格改变了JavaScript程序员的设计思路和编写程序的方式。---摘自《锋利的jQuery》

通过分析jQuery的源码，我们能（这也是本文的写作目的）：

l 学习先进的设计理念

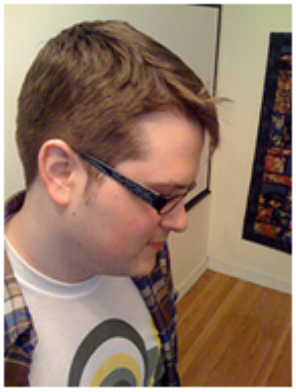
l 学习各种实现技巧

l 巩固JavaScript基础

l 无限的接近这些牛人↓↓↓（记住他们吧，记住他们改变了JavaScript）

jQuery团队核心人物

John Resig	Brandon Aaron	Jorn Zaefferer
http://ejohn.org/	http://brandonaaron.net/	http://bassistance.de/



1.3 [原创] jQuery源码分析-01总体架构

发表时间: 2011-09-21 关键字: web, javascript, jquery, 源码, 总体架构

1. 总体架构

1.1 自调用匿名函数 self-invoking anonymous function

打开jQuery源码，首先你会看到这样的代码结构：

```
(function( window, undefined ) {
    // jquery code
})(window);
```

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-1.6.1.min.js：

```
(function(a,b){})(window); // window 被优化为 a
```

5. 为什么要在在参数列表中增加undefined呢？

在 自调用匿名函数 的作用域内，确保undefined是真的未定义。因为undefined能够被重写，赋予新的值。

```
undefined = "now it's defined";
alert( undefined );
```

浏览器测试结果：

浏览器	测试结果	结论
ie	**now it's defined**	可以改变
firefox	undefined	不能改变
chrome	now it's defined	可以改变
opera	now it's defined	可以改变

6. 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。

1.2 总体架构

接下来看看在 自调用匿名函数 中都实现了什么功能，按照代码顺序排列：

```
(function( window, undefined ) {
    // 构造jQuery对象
```

```
    var jQuery = function( selector, context ) {
     return new jQuery.fn.init( selector, context, rootjQuery );
    }
    // 工具函数 Utilities
    // 异步队列 Deferred
    // 浏览器测试 Support
    // 数据缓存 Data
    // 队列 queue
    // 属性操作 Attribute
    // 事件处理 Event
    // 选择器 Sizzle
    // DOM遍历
    // DOM操作
    // CSS操作
    // 异步请求 Ajax
    // 动画 FX
    // 坐标和大小
        window.jQuery = window.$ = jQuery;
})(window);
```

从上边的注释看，jQuery的源码结构相当清晰、条理，不像代码那般晦涩和让人纠结。

后边的章节基本将以这个顺序展开。

1.3 下节预告

如果你看过jQuery源码，很快就会发现这里到处充斥着正则表达式，而很多JavaScript开发人员又疏于正则基础知识，为了扫清这个障碍，下一章将先温习JavaScript正则表达式的基础知识，再详细剖析jQuery中的正则表达式。

在正式开始分析源码之前，还有没有要准备的基础知识呢？

当然有。比如JavaScript API中的类和对象，如果你不熟练的话，至少手头要有一本参考手册。

除了正则，其他的知识点会在分析过程中穿插讲解，不计划辟出新的章节。

1.4 [原创] jQuery源码分析-03构造jQuery对象-源码结构和核心函数

发表时间: 2011-09-28 关键字: web, javascript, jquery, 构造jquery对象, 源码结构和核心函数

作者：nuysoft/高云 QQ：47214707 EMail：nuysoft@gmail.com

毕竟是边读边写，不对的地方请告诉我，多多交流共同进步。本章还未写完，完了会提交PDF。

前记：

想系统的好好写写，但是会先从感兴趣的部分开始。

近期有读者把PDF传到了百度文库上，首先感谢转载和传播，但是据为已有并设置了挺高的财富值才能下载就不好了，以后我整理好了会传到文库上。请体谅一下。

3. 构造jQuery对象

3.1 源码结构

先看看总体结构，再做分解：

```
(function( window, undefined ) {


    var jQuery = (function() {
      // 构建jQuery对象
      var jQuery = function( selector, context ) {
          return new jQuery.fn.init( selector, context, rootjQuery );
      }


      // 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
    });

    // 到这里，jQuery对象构造完成，后边的代码都是对jQuery或jQuery对象的扩展
    return jQuery;

})();

window.jQuery = window.$ = jQuery;
})(window);
```

l 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 );
    }
```

n jQuery对象就是jQuery.fn.init对象

n 如果执行new jQeury(),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

l 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！

3.2 jQuery.fn.init

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**
DOM元素	包装成jQuery对象，直接返回
body（优化）	从document.body读取
单独的HTML标签	document.createElement
HTML字符串	document.createDocumentFragment
#id	document.getElementById
选择器表达式	$(...).find

函数 注册到dom ready的回调函数

3.3 jQuery.extend = jQuery.fn.extend

```
// 合并两个或更多对象的属性到第一个对象中，jQuery后续的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

// 如果传入两个或多个对象，所有对象的属性会被添加到第一个对象target

// 如果只传入一个对象，则将对象的属性添加到jQuery对象中。
// 用这种方式，我们可以为jQuery命名空间增加新的方法。可以用于编写jQuery插件。
// 如果不想改变传入的对象，可以传入一个空对象：$.extend({}, object1, object2);
// 默认合并操作是不迭代的，即便target的某个属性是对象或属性，也会被完全覆盖而不是合并
// 第一个参数是true，则会迭代合并
// 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

// 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;
    target = arguments[1] || {};
    // skip the boolean and the target
    // 跳过boolean和target，从第3个开始
    i = 2;
}

// Handle case when target is a string or something (possible in deep copy)
// 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 ( length === i ) {
    target = this;
```

```
    --i;
}

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)) ) ) {
                // 如果copy是数组
                if ( copyIsArray ) {
```

```
                copyIsArray = false;
                // clone为src的修正值
                clone = src && jQuery.isArray(src) ? src : [];
            // 如果copy的是对象
            } else {
                // clone为src的修正值
                clone = src && jQuery.isPlainObject(src) ? src : {};
            }

            // 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;
};
```

未完待续

1.5 [原创] jQuery源码分析-03构造jQuery对象-工具函数

发表时间: 2011-09-29 关键字: javascript, jquery, 源码分析, 构造jQuery对象, 工具函数

作者：nuysoft/高云 QQ：47214707 EMail：nuysoft@gmail.com

读读写写，不对的地方请告诉我，多多交流共同进步，本章的的PDF等本章写完了发布。

jQuery源码分析系列的目录请查看 http://nuysoft.iteye.com/blog/1177451，想系统的好好写写，目前还是从我感兴趣的部分开始，如果大家有对哪个模块感兴趣的，建议优先分析的，可以告诉我，一起学习。

3.4 其他静态工具函数

```
// 扩展工具函数
jQuery.extend({
    // http://www.w3school.com.cn/jquery/core_noconflict.asp
    // 释放$的 jQuery 控制权
    // 许多 JavaScript 库使用 $ 作为函数或变量名，jQuery 也一样。
    // 在 jQuery 中，$ 仅仅是 jQuery 的别名，因此即使不使用 $ 也能保证所有功能性。
    // 假如我们需要使用 jQuery 之外的另一 JavaScript 库，我们可以通过调用 $.noConflict() 向该库返回控制权。

    // 通过向该方法传递参数 true，我们可以将 $ 和 jQuery 的控制权都交还给另一JavaScript库。
    noConflict: function( deep ) {
      // 交出$的控制权
      if ( window.$ === jQuery ) {
```

```
        window.$ = _$;
    }
    // 交出jQuery的控制权
    if ( deep && window.jQuery === jQuery ) {
        window.jQuery = _jQuery;
    }

    return jQuery;
},

// Is the DOM ready to be used? Set to true once it occurs.
isReady: false,

// A counter to track how many items to wait for before
// the ready event fires. See #6781
// 一个计数器，用于跟踪在ready事件出发前的等待次数
readyWait: 1,

// Hold (or release) the ready event
// 继续等待或触发
holdReady: function( hold ) {
    if ( hold ) {
        jQuery.readyWait++;
```

```
    } else {
        jQuery.ready( true );
    }
},

// Handle when the DOM is ready
// 文档加载完毕句柄
// http://www.cnblogs.com/fjzhou/archive/2011/05/31/jquery-source-4.html
ready: function( wait ) {
    // Either a released hold or an DOMready/load event and not yet ready
    //
    if ( (wait === true && !--jQuery.readyWait) || (wait !== true && !jQuery.isReady) ) {
        // Make sure body exists, at least, in case IE gets a little overzealous (ticket #5443).
        // 确保document.body存在
        if ( !document.body ) {
            return setTimeout( jQuery.ready, 1 );
        }

        // Remember that the DOM is ready
        jQuery.isReady = true;

        // If a normal DOM Ready event fired, decrement, and wait if need be
        if ( wait !== true && --jQuery.readyWait > 0 ) {
```

```
            return;
        }

        // If there are functions bound, to execute
        readyList.resolveWith( document, [ jQuery ] );

        // Trigger any bound ready events
        if ( jQuery.fn.trigger ) {
            jQuery( document ).trigger( "ready" ).unbind( "ready" );
        }
    }
},

// 初始化readyList事件处理函数队列
// 兼容不同浏览对绑定事件的区别
bindReady: function() {
    if ( readyList ) {
        return;
    }

    readyList = jQuery._Deferred();

    // Catch cases where $(document).ready() is called after the
```

```
// browser event has already occurred.
if ( document.readyState === "complete" ) {
    // Handle it asynchronously to allow scripts the opportunity to delay ready
    return setTimeout( jQuery.ready, 1 );
}

// Mozilla, Opera and webkit nightlies currently support this event
// 兼容事件，通过检测浏览器的功能特性，而非嗅探浏览器
if ( document.addEventListener ) {
    // Use the handy event callback
    // 使用较快的加载完毕事件
    document.addEventListener( "DOMContentLoaded", DOMContentLoaded, false );

    // A fallback to window.onload, that will always work
    // 注册window.onload回调函数
    window.addEventListener( "load", jQuery.ready, false );

// If IE event model is used
} else if ( document.attachEvent ) {
    // ensure firing before onload,
    // maybe late but safe also for iframes
    // 确保在onload之前触发onreadystatechange，可能慢一些但是对iframes更安全
    document.attachEvent( "onreadystatechange", DOMContentLoaded );
```

```
        // A fallback to window.onload, that will always work
        // 注册window.onload回调函数
        window.attachEvent( "onload", jQuery.ready );

        // If IE and not a frame
        // continually check to see if the document is ready
        var toplevel = false;

        try {
          toplevel = window.frameElement == null;
        } catch(e) {}

        if ( document.documentElement.doScroll && toplevel ) {
          doScrollCheck();
        }
      }
    },

    // See test/unit/core.js for details concerning isFunction.
    // Since version 1.3, DOM methods and functions like alert
    // aren't supported. They return false on IE (#2968).
    // 是否函数
```

```
isFunction: function( obj ) {
    return jQuery.type(obj) === "function";
},

// 是否数组
// 如果浏览器有内置的 Array.isArray 实现，就使用浏览器自身的实现方式，
// 否则将对象转为String，看是否为"[object Array]"。
isArray: Array.isArray || function( obj ) {
    return jQuery.type(obj) === "array";
},

// A crude way of determining if an object is a window
// 简单的判断（判断setInterval属性）是否window对象
isWindow: function( obj ) {
    return obj && typeof obj === "object" && "setInterval" in obj;
},

// 是否是保留字NaN
isNaN: function( obj ) {
    // 等于null 或 不是数字 或调用window.isNaN判断
    return obj == null || !rdigit.test( obj ) || isNaN( obj );
},
// 获取对象的类型
```

```
type: function( obj ) {
  // 通过核心API创建一个对象，不需要new关键字
  // 普通函数不行
  // 调用Object.prototype.toString方法，生成 "[object Xxx]"格式的字符串
  // class2type[ "[object " + name + "]" ] = name.toLowerCase();
  return obj == null ?
    String( obj ) :
    class2type[ toString.call(obj) ] || "object";
},

// 检查obj是否是一个纯粹的对象（通过"{}" 或 "new Object"创建的对象）
// console.info( $.isPlainObject( {} ) ); // true
// console.info( $.isPlainObject( '' ) ); // false
// console.info( $.isPlainObject( document.location ) ); // true
// console.info( $.isPlainObject( document ) ); // false
// console.info( $.isPlainObject( new Date() ) ); // false
// console.info( $.isPlainObject( ) ); // false

// isPlainObject分析与重构 http://www.jb51.net/article/25047.htm
// 对jQuery.isPlainObject()的理解 http://www.cnblogs.com/phpmix/articles/1733599.html
isPlainObject: function( obj ) {
  // Must be an Object.
  // Because of IE, we also have to check the presence of the constructor property.
```

```
// Make sure that DOM nodes and window objects don't pass through, as well
// 必须是一个对象
// 因为在IE8中会抛出非法指针异常，必须检查constructor属性
// DOM节点和window对象，返回false

// obj不存在 或 非object类型 或 DOM节点 或 widnow对象，直接返回false
// 测试以下三中可能的情况：
// jQuery.type(obj) !== "object" 类型不是object，忽略
// obj.nodeType 认为DOM节点不是纯对象
// jQuery.isWindow( obj ) 认为window不是纯对象
if ( !obj || jQuery.type(obj) !== "object" || obj.nodeType || jQuery.isWindow( obj ) ) {
    return false;
}

// Not own constructor property must be Object
// 测试constructor属性
// 具有构造函数constructor，却不是自身的属性（即通过prototype继承的），
if ( obj.constructor &&
    !hasOwn.call(obj, "constructor") &&
    !hasOwn.call(obj.constructor.prototype, "isPrototypeOf") ) {
    return false;
}
```

```
    // Own properties are enumerated firstly, so to speed up,
    // if last one is own, then all properties are own.

    var key;
    for ( key in obj ) {}
    // key === undefined及不存在任何属性，认为是简单的纯对象
    // hasOwn.call( obj, key ) 属性key不为空，且属性key的对象自身的（即不是通过prototype继承的）
    return key === undefined || hasOwn.call( obj, key );
},
// 是否空对象
isEmptyObject: function( obj ) {
    for ( var name in obj ) {
        return false;
    }
    return true;
},
// 抛出一个异常
error: function( msg ) {
    throw msg;
},
// 解析JSON
// parseJSON把一个字符串变成JSON对象。
// 我们一般使用的是eval。parseJSON封装了这个操作，但是eval被当作了最后手段。
```

```
// 因为最新JavaScript标准中加入了JSON序列化和反序列化的API。
// 如果浏览器支持这个标准，则这两个API是在JS引擎中用Native Code实现的，效率肯定比eval高很多。
// 目前来看，Chrome和Firefox4都支持这个API。
parseJSON: function( data ) {
  if ( typeof data !== "string" || !data ) {
    return null;
  }

  // Make sure leading/trailing whitespace is removed (IE can't handle it)
  data = jQuery.trim( data );

  // Attempt to parse using the native JSON parser first
  // 原生JSON API。反序列化是JSON.stringify(object)
  if ( window.JSON && window.JSON.parse ) {
    return window.JSON.parse( data );
  }

  // Make sure the incoming data is actual JSON
  // Logic borrowed from http://json.org/json2.js
  // ... 大致地检查一下字符串合法性
  if ( rvalidchars.test( data.replace( rvalidescape, "@" )
    .replace( rvalidtokens, "]" )
    .replace( rvalidbraces, "")) ) {
```

```
            return (new Function( "return " + data ))();

        }
        jQuery.error( "Invalid JSON: " + data );
    },

    // Cross-browser xml parsing
    // (xml & tmp used internally)
    // 解析XML 跨浏览器
    // parseXML函数也主要是标准API和IE的封装。
    // 标准API是DOMParser对象。
    // 而IE使用的是Microsoft.XMLDOM的 ActiveXObject对象。
    parseXML: function( data , xml , tmp ) {

        if ( window.DOMParser ) { // Standard 标准XML解析器
            tmp = new DOMParser();
            xml = tmp.parseFromString( data , "text/xml" );
        } else { // IE IE的XML解析器
            xml = new ActiveXObject( "Microsoft.XMLDOM" );
            xml.async = "false";
            xml.loadXML( data );
        }
```

```
        tmp = xml.documentElement;

        if ( ! tmp || ! tmp.nodeName || tmp.nodeName === "parsererror" ) {
            jQuery.error( "Invalid XML: " + data );
        }

        return xml;
    },
    // 无操作函数
    noop: function() {},

    // Evaluates a script in a global context
    // Workarounds based on findings by Jim Driscoll
    // http://weblogs.java.net/blog/driscoll/archive/2009/09/08/eval-javascript-global-context
    // globalEval函数把一段脚本加载到全局context（window）中。
    // IE中可以使用window.execScript。
    // 其他浏览器 需要使用eval。
    // 因为整个jQuery代码都是一整个匿名函数，所以当前context是jQuery，如果要将上下文设置为window则需使用globalEval。
    globalEval: function( data ) {
        // data非空
        if ( data && rnotwhite.test( data ) ) {
```

```
        // We use execScript on Internet Explorer
        // We use an anonymous function so that context is window
        // rather than jQuery in Firefox
        ( window.execScript || function( data ) {
          window[ "eval" ].call( window, data );
        } )( data );
      }
    },
    // 判断节点名称是否相同
    nodeName: function( elem, name ) {
      // 忽略大小写
      return elem.nodeName && elem.nodeName.toUpperCase() === name.toUpperCase();
    },

    // args is for internal usage only
    // 遍历对象或数组
    each: function( object, callback, args ) {
      var name, i = 0,
        length = object.length,
        isObj = length === undefined || jQuery.isFunction( object );
      // 如果有参数args，调用apply，上下文设置为当前遍历到的对象，参数使用args
      if ( args ) {
        if ( isObj ) {
```

```
            for ( name in object ) {
                if ( callback.apply( object[ name ], args ) === false ) {
                    break;
                }
            }
        } else {
            for ( ; i < length; ) {
                if ( callback.apply( object[ i++ ], args ) === false ) {
                    break;
                }
            }
        }

    // A special, fast, case for the most common use of each
    // 没有参数args则调用，则调用call，上下文设置为当前遍历到的对象，参数设置为key/index和value
    } else {
        if ( isObj ) {
            for ( name in object ) {
                if ( callback.call( object[ name ], name, object[ name ] ) === false ) {
                    break;
                }
            }
        } else {
```

```
            for ( ; i < length; ) {
                if ( callback.call( object[ i ], i, object[ i++ ] ) === false ) {
                    break;
                }
            }
        }
    }

    return object;
},

// Use native String.trim function wherever possible
// 尽可能的使用本地String.trim方法，否则先过滤开头的空格，再过滤结尾的空格
trim: trim ?
    function( text ) {
        return text == null ?
            "" :
            trim.call( text );
    } :

    // Otherwise use our own trimming functionality
    function( text ) {
        return text == null ?
```

```
            "" :
            text.toString().replace( trimLeft, "" ).replace( trimRight, "" );
    },

// results is for internal usage only
// 将伪数组转换为数组
makeArray: function( array, results ) {
    var ret = results || [];

    if ( array != null ) {
        // The window, strings (and functions) also have 'length'
        // The extra typeof function check is to prevent crashes
        // in Safari 2 (See: #3039)
        // Tweaked logic slightly to handle Blackberry 4.7 RegExp issues #6930
        // 一大堆浏览器兼容性测试，真实蛋疼
        var type = jQuery.type( array );
        // 测试：有没有length属性、字符串、函数、正则
        // 不是数组，连伪数组都不是
        if ( array.length == null
                || type === "string"
                || type === "function"
                || type === "regexp"
                || jQuery.isWindow( array ) ) {
```

```
            push.call( ret, array );
        } else {
            // $.type( $('div') ) // object
            jQuery.merge( ret, array );
        }
    }

    return ret;
},
//
inArray: function( elem, array ) {
    // 是否有本地化的Array.prototype.indexOf
    if ( indexOf ) {
        // 直接调用Array.prototype.indexOf
        return indexOf.call( array, elem );
    }
    // 遍历数组，查找是否有完全相等的元素，并返回下标
    // 循环的小技巧：把array.length存放到length变量中，可以减少一次作用域查找
    for ( var i = 0, length = array.length; i < length; i++ ) {
        if ( array[ i ] === elem ) {
            return i;
        }
    }
```

```
    // 如果返回-1，则表示不在数组中
    return -1;
  },
  // 将数组second合并到数组first中
  merge: function( first, second ) {
    var i = first.length, //
      j = 0;

    // 如果second的length属性是Number类型，则把second当数组处理
    if ( typeof second.length === "number" ) {
      for ( var l = second.length; j < l; j++ ) {
        first[ i++ ] = second[ j ];
      }

    } else {
      // 遍历second，将非undefined的值添加到first中
      while ( second[j] !== undefined ) {
        first[ i++ ] = second[ j++ ];
      }
    }
    // 修正first的length属性，因为first可能不是真正的数组
     first.length = i;
```

```
    return first;
},
// 过滤数组，返回新数组；callback返回true时保留；如果inv为true，callback返回false才会保留
grep: function( elems, callback, inv ) {
    var ret = [], retVal;
    inv = !!inv;

    // Go through the array, only saving the items
    // that pass the validator function
    // 遍历数组，只保留通过验证函数callback的元素
    for ( var i = 0, length = elems.length; i < length; i++ ) {
        // 这里callback的参数列表为：value, index，与each的习惯一致
        retVal = !!callback( elems[ i ], i );
        // 是否反向选择
        if ( inv !== retVal ) {
            ret.push( elems[ i ] );
        }
    }

    return ret;
},

// arg is for internal usage only
```

```
    // 将数组或对象elems的元素/属性，转化成新的数组
    map: function( elems, callback, arg ) {
        var value, key, ret = [],
            i = 0,
            length = elems.length,
            // jquery objects are treated as arrays
            // 检测elems是否是（伪）数组
            // 1. 将jQuery对象也当成数组处理
            // 2. 检测length属性是否存在，length等于0，或第一个和最后一个元素是否存在，或jQuery.isArray返回true
            isArray = elems instanceof jQuery
              || length !== undefined && typeof length === "number"
                 && ( ( length > 0 && elems[ 0 ] && elems[ length -1 ] ) || length === 0 || jQuery.isArray( elems ) ) ;

        // 是数组或对象的差别，仅仅是遍历的方式不同，没有其他的区别

        // Go through the array, translating each of the items to their
        // 遍历数组，对每一个元素调用callback，将返回值不为null的值，存入ret
        if ( isArray ) {
            for ( ; i < length; i++ ) {
                // 执行callback，参数依次为value, index, arg
                value = callback( elems[ i ], i, arg );
```

```
            // 如果返回null，则忽略（无返回值的function会返回undefined）
            if ( value != null ) {
                ret[ ret.length ] = value;
            }
        }

    // Go through every key on the object,
    // 遍历对象，对每一个属性调用callback，将返回值不为null的值，存入ret
    } else {
        for ( key in elems ) {
            // 执行callback，参数依次为value, key, arg
            value = callback( elems[ key ], key, arg );
            // 同上
            if ( value != null ) {
                ret[ ret.length ] = value;
            }
        }
    }

    // Flatten any nested arrays
    // 使嵌套数组变平
    // concat：
    // 如果某一项为数组，那么添加其内容到末尾。
```

```
    // 如果该项目不是数组，就将其作为单个的数组元素添加到数组的末尾。
    return ret.concat.apply( [], ret );
},

// A global GUID counter for objects
guid: 1,

// Bind a function to a context, optionally partially applying any
// arguments.
// 代理方法：为fn指定上下文（即this）
// jQuery.proxy( function, context )
// jQuery.proxy( context, name )
proxy: function( fn, context ) {
    // 如果context是字符串，设置上下文为fn，fn为fn[ context ]
    // 即设置fn的context方法的上下文为fn（默认不是这样吗？？？TODO）
    if ( typeof context === "string" ) {
        var tmp = fn[ context ];
        context = fn;
        fn = tmp;
    }

    // Quick check to determine if target is callable, in the spec
    // this throws a TypeError, but we will just return undefined.
```

```
    // 快速测试fn是否是可调用的（即函数），在文档说明中，会抛出一个TypeError，
    // 但是这里仅返回undefined
    if ( !jQuery.isFunction( fn ) ) {
        return undefined;
    }

    // Simulated bind
    var args = slice.call( arguments, 2 ), // 从参数列表中去掉fn,context
        proxy = function() {
            // 设置上下文为context和参数
            return fn.apply( context, args.concat( slice.call( arguments ) ) );
        };

    // Set the guid of unique handler to the same of original handler, so it can be removed
    // 统一guid，使得proxy能够被移除
    proxy.guid = fn.guid = fn.guid || proxy.guid || jQuery.guid++;

    return proxy;
},

// Mutifunctional method to get and set values to a collection
// The value/s can be optionally by executed if its a function
// 多功能函数，读取或设置集合的属性值；值为函数时会被执行
```

```
// fn : jQuery.fn.css, jQuery.fn.attr, jQuery.fn.prop
access: function( elems, key, value, exec, fn, pass ) {
  var length = elems.length;

  // Setting many attributes
  // 如果有多个属性，则迭代
  if ( typeof key === "object" ) {
    for ( var k in key ) {
      jQuery.access( elems, k, key[k], exec, fn, value );
    }
    return elems;
  }

  // Setting one attribute
  // 只设置一个属性
  if ( value !== undefined ) {
    // Optionally, function values get executed if exec is true
    exec = !pass && exec && jQuery.isFunction(value);

    for ( var i = 0; i < length; i++ ) {
      fn( elems[i], key, exec ? value.call( elems[i], i, fn( elems[i], key ) ) : value, pass );
    }
```

```
        return elems;
    }

    // Getting an attribute
    // 读取属性
    return length ? fn( elems[0], key ) : undefined;
},
// 获取当前时间的便捷函数
now: function() {
    return (new Date()).getTime();
},

// Use of jQuery.browser is frowned upon.
// More details: http://docs.jquery.com/Utilities/jQuery.browser
// 不赞成使用jQuery.browser，推荐使用jQuery.support
// Navigator 正在使用的浏览器的信息
// Navigator.userAgent 一个只读的字符串，声明了浏览器用于HTPP请求的用户代理头的值
uaMatch: function( ua ) {
    ua = ua.toLowerCase();
    // 依次匹配各浏览器
    var match = rwebkit.exec( ua ) ||
        ropera.exec( ua ) ||
        rmsie.exec( ua ) ||
```

```
        ua.indexOf("compatible") < 0 && rmozilla.exec( ua ) ||
        [];
    // match[1] || ""
    // match[1]为false（空字符串、null、undefined、0等）时，默认为""
    // match[2] || "0"
    // match[2]为false（空字符串、null、undefined、0等）时，默认为"0"
    return { browser: match[1] || "", version: match[2] || "0" };
},
// 创建一个新的jQuery副本，副本的属性和方法可以被改变，但是不会影响原始的jQuery对象
// 有两种用法：
// 1. 覆盖jQuery的方法，而不破坏原始的方法
// 2.封装，避免命名空间冲突，可以用来开发jQuery插件
// 值得注意的是，jQuery.sub()函数并不提供真正的隔离，所有的属性、方法依然指向原始的jQuery
// 如果使用这个方法来开发插件，建议优先考虑jQuery UI widget工程
sub: function() {
    function jQuerySub( selector, context ) {
        return new jQuerySub.fn.init( selector, context );
    }
    jQuery.extend( true, jQuerySub, this ); // 深度拷贝，将jQuery的所有属性和方法拷贝到jQuerySub
    jQuerySub.superclass = this;
    jQuerySub.fn = jQuerySub.prototype = this(); //
    jQuerySub.fn.constructor = jQuerySub;
    jQuerySub.sub = this.sub;
```

```
        jQuerySub.fn.init = function init( selector, context ) {
            if ( context && context instanceof jQuery && !(context instanceof jQuerySub) ) {
                context = jQuerySub( context );
            }

            return jQuery.fn.init.call( this, selector, context, rootjQuerySub );
        };
        jQuerySub.fn.init.prototype = jQuerySub.fn;
        var rootjQuerySub = jQuerySub(document);
        return jQuerySub;
    },
    // 浏览器类型和版本：
    // $.browser.msie/mozilla/webkit/opera
    // $.browser.version
    // 不推荐嗅探浏览器类型jQuery.browser，而是检查浏览器的功能特性jQuery.support
    // 未来jQuery.browser可能会移到一个插件中
    browser: {}
});
```

1.6 [原创] jQuery源码分析-05异步队列 Deferred

发表时间: 2011-09-19 关键字: web, javascript, jquery, deferred, ajax

5. 异步队列 Deferred

5.1 概述

异步队列是一个链式对象，增强对回调函数的管理和调用，用于处理异步任务。

异步队列有三种状态：初始化（unresolved），成功（resolved），失败（rejected）。

执行哪些回调函数依赖于状态。

状态变为成功（resolved）或失败（rejected）后，将保持不变。

回调函数的绑定可以是同步，也可以是异步的，即可以在任何时候绑定。

（本节中的 绑定 注册 增加 具有相同的含义）

5.2 关键方法

先看看jQuery. Deferred()中的关键方法

分类	方法	说明
增加	deferred.done()	增加成功回调函数 状态为成功（resolved）时立即调用
增加	deferred.fail()	增加失败回调函数 状态为失败（rejected）时立即调用
增加	deferred.then()	增加成功回调函数和失败回调函数到各自的队列中 便捷方法，两个参数可以是数组或null

	状态为成功（resolved）时立即调用成功回调函数 状态为失败（rejected）时立即调用失败回调函数	
	deferred.always()	增加回调函数，同时增加到成功队列和失败队列 状态已确定（无论成功或失败）时立即调用回调函数
执行	deferred.resolve()	调用成功回调函数队列 通过调用deferred.resolveWith()实现
deferred.resolveWith()	使用指定的上下文和参数执行成功回调函数	
deferred.reject()	调用失败回调函数队列 通过调用deferred.rejectWith()实现	
deferred.rejectWith()	使用指定的上下文和参数执行失败回调函数队列	
其他	deferred.isRejected()	判断状态是否为成功（resolved）
deferred.isResolved()	判断状态是否为失败（rejected）	
deferred.pipe()	每次调用回调函数之前先调用传入的成功过滤函数或失败过滤函数，并将过滤函数的返回值作为回调函数的参数 最终返回一个只读视图（调用promise实现）	
deferred.promise()	返回deferred的只读视图	

接下来将会jQuery._Deferred和jQuery.Deferred的源码详细剖析。

5.3 jQuery._Deferred

局部变量

```
// 参考资料：
// 官网文档 http://api.jquery.com/category/deferred-object/
// Deferred机制 http://www.cnblogs.com/fjzhou/archive/2011/05/30/jquery-source-3.html
// 在jQuery 1.5中使用deferred对象 http://developer.51cto.com/art/201103/248638.htm
// 拿着放大镜看Promise http://www.cnblogs.com/sanshi/archive/2011/03/11/1981789.html
// Promises/A http://wiki.commonjs.org/wiki/Promises/A

var // Promise methods
    // 注意，没有以下方法：resolveWith resolve rejectWith reject pipe when cancel
    // 即不允许调用resolve reject cancel等
    promiseMethods = "done fail isResolved isRejected promise then always pipe".split( " " ),
    // Static reference to slice
    // 静态引用slice方法，借鸡生蛋
    sliceDeferred = [].slice;
```

_Deferred：

```
_Deferred: function() {
    var // callbacks list
        // 回调函数数组（这里不翻译为队列，避免概念上的混淆）
        callbacks = [],
        // stored [ context , args ]
        // 存储上下文、参数，同时还可以标识是否执行完成（fired非空即表示已完成）
        // 这里的“完成”指回调函数数组中“已有”的函数都已执行完成；
        // 但是可以再次调用done添加回调函数，添加时fired会被重置为0
        fired,
        // to avoid firing when already doing so
        // 如果已经触发正在执行，避免再次触发
        firing,
        // flag to know if the deferred has been cancelled
        // 标识异步队列是否已被取消，取消后将忽略对done resolve resolveWith的调用
        cancelled,
        // 异步队列定义（这才是正主，上边的局部变量通过闭包引用）
        // the deferred itself
        deferred  = {
            // done( f1, f2, ...)
            // 增加成功回调函数，状态为成功（resolved）时立即调用
            done: function() {
                // 如果已取消，则忽略本次调用
                if ( !cancelled ) {
```

```
// 将后边代码用到的局部变量定义在代码块开始处的好处：
// 1.声明变量，增加代码可读性；
// 2.共享变量，提高性能
// 注：多年写Java的经验，养成了全局变量在开头、临时变量随用随定义的习惯，看来JavaScript有些不同

var args = arguments, // 回调函数数组
  i, // 遍历变量
  length, // 回调函数数组长度
  elem, // 单个回调函数
  type, // elem类型
  _fired; // 用于临时备份fired（fired中存储了上下文和参数）

// 如果已执行完成（即fired中保留了上下文和参数）
// 则备份上下文和参数到_fired，同时将fired置为0
if ( fired ) {
  _fired = fired;
  fired = 0;
}
// 添加arguments中的函数到回调函数数组
for ( i = 0, length = args.length; i < length; i++ ) {
  elem = args[ i ];
  type = jQuery.type( elem );
```

```
                // 如果是数组，则递归调用
                if ( type === "array" ) {
                    // 强制指定上下文为deferred，个人认为这里没必要指定上下文，因为默认的上下文即为deferred
                    deferred.done.apply( deferred, elem );
                } else if ( type === "function" ) {
                    callbacks.push( elem );
                }
            }
            // 如果已执行（_fired表示Deferred的状态是确定的），则立即执行新添加的函数
            // 使用之前指定的上下文context和参数args
            if ( _fired ) {
                deferred.resolveWith( _fired[ 0 ], _fired[ 1 ] );
            }
        }
        return this;
    },

    // resolve with given context and args
    // 执行，使用指定的上下文和参数
    resolveWith: function( context, args ) {
        // 满足以下全部条件，才会执行：没有取消  没有正在执行 没有执行完成
        // 如果已取消 或 已执行完成 或 正在执行，则忽略本次调用
```

```
    if ( !cancelled && !fired && !firing ) {
        // make sure args are available (#8421)
        // 确保args可用，一个避免null、undefined造成ReferenceError的常见技巧
        args = args || [];
        // 执行过程中将firing改为1
        firing = 1;
        try {
          // 遍历动态数组的技巧
          while( callbacks[ 0 ] ) {
              // 注意这里使用指定的context，而不是this
              callbacks.shift().apply( context, args );
          }
        }
        // JavaScript支持try/catch/finally
        finally {
          fired = [ context, args ];
          firing = 0;
        }
    }
    return this;
},

// resolve with this as context and given arguments
```

```
// 把状态设置为Resolved
// 设置的理解不准确，因为是否Resolved，是调用isResolved判断firing、fired的状态得到的。
// 可以理解为执行
resolve: function() {
   deferred.resolveWith( this, arguments );
   return this;
},

// Has this deferred been resolved?
// 是否已执行（或解决）？
// 在执行或已执行完毕，都认为已执行/解决
// “已”可能不准确，因为执行过程中也认为是已执行
isResolved: function() {
   // 正在运行中
   // 或
   // 已运行完（即fired不为空/0）
   return !!( firing || fired );
},

// Cancel
// 取消异步队列
// 设置标记位，清空函数队列
cancel: function() {
```

```
            cancelled = 1;
            callbacks = [];
            return this;
        }
    };

    return deferred;
}
```

5.4 jQuery.Deferred

```
// Full fledged deferred (two callbacks list)
// 创建一个完整的异步队列（包含两个回调函数数组）
// 异步队列有三种状态：初始化（unresolved），成功（resolved），失败（rejected）。
// 执行哪些回调函数依赖于状态。
// 状态变为成功（resolved）或失败（rejected）后，将保持不变。
Deferred: function( func ) {
    // _Deferred本无成功状态或失败状态，有四种状态：初始化、执行中、执行完毕、已取消
    // 为了代码复用， 内部先实现了一个_Deferred
    // failDeferred通过闭包引用
    var deferred = jQuery._Deferred(),
        failDeferred = jQuery._Deferred(),
        promise;
    // Add errorDeferred methods, then and promise
```

```
jQuery.extend( deferred, {
    // 增加成功回调函数和失败回调函数到各自的队列中
    // 便捷方法，两个参数可以是数组或null
    // 状态为成功（resolved）时立即调用成功回调函数
    // 状态为失败（rejected）时立即调用失败回调函数
    then: function( doneCallbacks, failCallbacks ) {
        // 上下文在这里有切换：虽然done返回的是deferred，但是fail指向failDeferred.done，执行fail是上下文变为failDeferred
        // 简单点说就是：
        // 调用done时向deferred添加回调函数doneCallbacks
        // 调用fail时向failDeferred添加回调函数failCallbacks

        // 因此这行表达式执行完后，返回的是failDeferred
        deferred.done( doneCallbacks ).fail( failCallbacks );
        // 强制返回deferred
        return this;
    },
    // 注册一个callback函数，无论是resolved或者rejected都会被 调用。
    // 其实，是把传入的函数（数组），同时添加到deferred和failDeferred
    // 并没有像我想象的那样，存到单独的函数数组中
    always: function() {
        // done的上下文设置为deferred，fail的上下文设置为this
        // done和fail的上下文不一致吗？一致！在这里this等于deferred
```

```
        // 但是这里如此设置上下文应该该如何解释呢？与then的实现有什么不一样呢？

        // fail指向fail指向failDeferred.done，默认上下文是failDeferred，failDeferred的回调函数数组callbacks是通过闭包引用的，
        // 这里虽然将failDeferred.done方法的上下文设置为deferred，但是不影响failDeferred.done的执行，
        // 在failDeferred.done的最后将this替换为deferred，实现链式调用，
        // 即调用过程中没有丢失上下文this，可以继续链式调用其他的方法而不会导致this混乱

        // 从语法上，always要达到的效果与then要达到的效果一致
        // 因此，这行代码可以改写为两行（类似then的实现方式）,效果是等价的：
        // deferred.done( arguments ).fail( arguments );
        // returnr this;
        return deferred.done.apply( deferred, arguments ).fail.apply( this, arguments );
    },
    // 增加失败回调函数
    // 状态为失败（rejected）时立即调用
    fail: failDeferred.done,
    // 使用指定的上下文和参数执行失败回调函数队列
    // 通过调用failDeferred.rejectWith()实现
    rejectWith: failDeferred.resolveWith,
    // 调用失败回调函数队列
```

```
// 通过调用failDeferred.resolve()实现
reject: failDeferred.resolve,
// 判断状态是否为成功（resolved）
isRejected: failDeferred.isResolved,

// 每次调用回调函数之前先调用传入的成功过滤函数或失败过滤函数，并将过滤函数的返回值作为回调函数的参数
// 最终返回一个只读视图（调用promise实现）
// fnDone在状态是否为成功（resolved）时被调用
// fnFail在状态是否为失败（rejected）时被调用

// 关于其他的解释：
// 1. 有的文章翻译为“管道机制”，从字面无法理解要表达什么含义，因此至少是不准确
// 2. 错误理解：所谓的pipe，只是把传入的fnDone和fnFail放到了成功队列和失败队列的数组头部
pipe: function( fnDone, fnFail ) {
    return jQuery.Deferred(function( newDefer ) {
      jQuery.each( {
        done: [ fnDone, "resolve" ], // done在后文中会指向deferred.done
        fail: [ fnFail, "reject" ]
      }, function( handler, data ) {
        var fn = data[ 0 ],
          action = data[ 1 ],
          returned;
```

```
            if ( jQuery.isFunction( fn ) ) {
                deferred[ handler ](function() {
                    returned = fn.apply( this, arguments );
                    if ( returned && jQuery.isFunction( returned.promise ) ) {
                        returned.promise().then( newDefer.resolve, newDefer.reject );
                    } else {
                        newDefer[ action ]( returned );
                    }
                });
            } else {
                deferred[ handler ]( newDefer[ action ] );
            }
        });
    }).promise();
},
// Get a promise for this deferred
// If obj is provided, the promise aspect is added to the object

// 返回的是一个不完整的Deferred的接口，没有resolve和reject，即不能 修改Deferred对象的状态，
// 这是为了不让外部函数提早触发回调函数，可以看作是一种只读视图。
//
// 比如$.ajax在1.5版本后不再返回XMLHttpRequest，而是返回一个封装了
XMLHttpRequest和Deferred对象接口的object。
```

```
    // 其中Deferred部分就是promise()得到 的，这样不让外部函数调用resolve和reject，防止在ajax完成前触发回调函数。
    // 把这两个函数的调用权限保留给ajax内部。
    promise: function( obj ) {
        if ( obj == null ) {
            // 实际只会执行一次promise，第一次执行的结果被存储在promise变量中
            if ( promise ) {
                return promise;
            }
            promise = obj = {};
        }
        var i = promiseMethods.length;
        // 又一种循环遍历方式
        // 我习惯用:
        // for( i = 0; i < len; i++ ) 或 for( i = len-1; i >=0; i-- ) 或 for( i = len; i--; )
        // jQuery真是遍地是宝！
        while( i-- ) {
            obj[ promiseMethods[i] ] = deferred[ promiseMethods[i] ];
        }
        return obj;
    }
});
// Make sure only one callback list will be used
```

```
    // 成功队列执行完成后，会执行失败带列的取消方法
    // 失败队列执行完成后，会执行成功队列的取消方法
    // 确保只有一个函数队列会被执行，即要么执行成功队列，要么执行失败队列；
    // 即状态只能是或成功、或失败，无交叉调用
    // deferred和failDeferred的canceled属性，只能通过闭包引用，因此不用担心状态、上下文的混乱
    deferred.done( failDeferred.cancel ).fail( deferred.cancel );
    // Unexpose cancel
    // 隐藏cancel接口，即无法从外部取消成功函数队列
    delete deferred.cancel;
    // Call given func if any
    // 执行传入的func函数
    if ( func ) {
        func.call( deferred, deferred );
    }
    return deferred;
}
```

5.5 jQuery.when

```
// Deferred helper
// 异步队列工具函数
// firstParam：一个或多个Deferred对象或JavaScript普通对象
when: function( firstParam ) {
    var args = arguments,
```

```
        i = 0,
        length = args.length,
        count = length,
        // 如果arguments.length等于1,并且firstParam是Deferred，则deferred=firstParam
        // 否则创建一个新的Deferred对象（如果arguments.length等于0或大于1，则创建一个新的Deferred对象）
        // 通过jQuery.isFunction( firstParam.promise )简单的判断是否是Deferred对象
        deferred = length <= 1 && firstParam && jQuery.isFunction( firstParam.promise ) ?
            firstParam :
            jQuery.Deferred();
    // 构造成功（resolve）回调函数
    function resolveFunc( i ) {
        return function( value ) {
            // 如果传入的参数大于一个，则将传入的参数转换为真正的数组（arguments没有slice方法，借鸡生蛋）
            args[ i ] = arguments.length > 1 ? sliceDeferred.call( arguments, 0 ) : value;
            if ( !( --count ) ) {
                // Strange bug in FF4:
                // Values changed onto the arguments object sometimes end up as undefined values
                // outside the $.when method. Cloning the object into a fresh array solves the issue
                // 执行成功回调函数队列，上下文强制为传入的第一个Deferred对象
                deferred.resolveWith( deferred, sliceDeferred.call( args, 0 ) );
            }
```

```
    };
}
// 如果参数多于一个
if ( length > 1 ) {
    for( ; i < length; i++ ) {
        // 简单的判断是否是Deferred对象，是则调用.promise().then()，否则忽略
        if ( args[ i ] && jQuery.isFunction( args[ i ].promise ) ) {
            // 增加成功回调函数和失败回调函数到各自的队列中
            args[ i ].promise().then( resolveFunc(i), deferred.reject );
        } else {
            // 计数器，表示发现不是Deferred对象，而是普通JavaScript对象
            --count;
        }
    }
    // 计数器为0时，表示传入的参数都不是Deferred对象
    // 执行成功回调函数队列，上下文强制为传入的第一个Deferred对象
    if ( !count ) {
        deferred.resolveWith( deferred, args );
    }
// deferred !== firstParam，即deferred为新创建的Deferred对象
// 即length == 0
} else if ( deferred !== firstParam ) {
    // 执行成功回调函数队列，上下文强制为新创建的Deferred对象
```

```
        deferred.resolveWith( deferred, length ? [ firstParam ] : [] );
    }
    // 返回传入的第一个Deferred或新创建的Deferred对象的只读视图
    return deferred.promise();
}
```

5.6 Deferred应用

l jQuery.ajax()

n TODO

5.7 可以学习的技巧

l 闭包

```
function a(){
    var guid = 1;
    return function(){
        return guid++;
    }
}
```

```
var defer = a();

console.info( defer() ); // 1
console.info( defer() ); // 2
console.info( defer() ); // 3
console.info( defer() ); // 4
```

l 避免null、undefined造成ReferenceError的常见技巧

```
args = args || [];
```

l 遍历动态数组的技巧

```
while( callbacks[ 0 ] ) {
    callbacks.shift().apply( context, args );
}
```

l try/catch/finally 实现错误处理

语法		说明

try {
// tryStatements
} catch(exception) {
// catchStatements
} finally {
	tryStatements	必选项。
		可能发生错误的语句。

```
    // finallyStatements
}
```

exception	必选项。任何变量名。 exception 的初始化值是扔出的错误的值。
catchStatements	可选项。 处理在相关联的 tryStatement 中发生的错误的语句。
finallyStatements	可选项。 在所有其他过程发生之后无条件执行的语句。

l 链式对象：通过返回this实现链式调用

方法	返回值
done	this（即deferred）
resolveWith	this（即deferred）
resolve	this（即deferred）
cancel	this（即deferred）

l 代码复用 $.each

```
jQuery.each( {
    done: [ fnDone, "resolve" ], // done在后文中会指向deferred.done
    fail: [ fnFail, "reject" ]
```

```
}, function( handler, data ) {

    // 公共代码复用

});
```

5.8 后续

- Deferred在jQuery中的应用
- Deferred的自定义应用

1.7 [原创] jQuery源码分析-08队列 Queue

发表时间: 2011-10-10 关键字: javascript, jquery, 源码分析, 队列, queue

作者：nuysoft/高云 QQ：47214707 EMail：nuysoft@gmail.com

读读写写，不对的地方请告诉我，多多交流共同进步，本章的的PDF下载在最后。

前记：
国庆给自己放了个安静的长假，日游杭州大小景点，夜宿西湖边上，于大街小巷中遍尝美味小吃，没有电脑没有网络，这样的日子真是是好日子啊；回京开始工作了，编程是我的兴趣，虽然变成了工作，但是享受的心态要继续保持下去。
白天工作，不管忙不忙，jQuery源码分析系列只能放在晚上写，经常看的朋友兴许也注意到更新时间一般是凌晨，经常觉的挺累的，想今天算了吧上床睡觉明天再说吧，但还是坚持下来了，我会尽量要求自己1-2天发布一篇。刚开始写的时候，有些担心写出来的东西会幼稚肤浅或讲不清楚，有些地方也确实是这样，这个假期让我想明白了很多事，以后的文章欢迎各位道友拍各种砖和石头，用力点，不要停。

8. 队列 Queue

8.1 概述

队列是一种特殊的线性表，只允许在表的前端（队头）进行删除操作（出队），在表的后端（队尾）进行插入操作（入队）。队列的特点是先进先出（FIFO-first in first out），即最先插入的元素最先被删除。

jQuery提供了jQuery.queue/dequeue和jQuery.fn.queue/dequeue，实现对队列的入队、出队操作。不同于队列定义的是，jQuery.queue和jQuery.fn.queue不仅执行出队操作，返回队头元素，还会自动执行返回的队头元素。

8.2 用途

在jQuery源码中，仅用于动画模块，这里入队的函数的功能是：

l 遍历要动画的属性，修正要执行的动画动作，修正/备份属性

l 遍历要动画的属性，为每一个属性创建jQuery.fx对象，计算起始值和结束值，调用fx对象的custom开始动画

看看源码：

```
/**
 * .animate( properties, [duration], [easing], [complete] )
 *
 * .animate( properties, options )
 *
 * animate做了三件事：
 * 1. 调用jQuery.speed修正传入的参数（时间、算法、回调函数）
 * 2. 遍历要动画的属性，修正要执行的动画动作，修正/备份属性
 * 3. 遍历要动画的属性，为每一个属性创建jQuery.fx对象，计算起始值和结束值，调用fx对象的custom开始动画
 */
animate: function( prop, speed, easing, callback ) {
    var optall = jQuery.speed(speed, easing, callback); // 修正参数

    // 如果是空对象，则直接运行callback
    if ( jQuery.isEmptyObject( prop ) ) {
        return this.each( optall.complete, [ false ] );
    }

    // Do not change referenced properties as per-property easing will be lost
    // 复制一份prop，不改变原有的属性
    prop = jQuery.extend( {}, prop );
    // queue为false，动画立即开始，否则则放入动画队列
```

```
return this[ optall.queue === false ? "each" : "queue" ](function() {
  // XXX 'this' does not always have a nodeName when running the
  // test suite

  if ( optall.queue === false ) {
    jQuery._mark( this );
  }

  var opt = jQuery.extend( {}, optall ), // 复制一份
    isElement = this.nodeType === 1,
    hidden = isElement && jQuery(this).is(":hidden"), // 是否隐藏
    name, val, p,
    display, e,
    parts, start, end, unit;

  // will store per property easing and be used to determine when an animation is complete
  // 已完成的属性
  opt.animatedProperties = {};

  // 遍历每一个属性，本次遍历仅仅是修正和记录属性值，为后边的动画做准备
  for ( p in prop ) {
    // property name normalization
    // property格式化，转换为驼峰式，因为style的属性名是驼峰式
```

```
        name = jQuery.camelCase( p );
        if ( p !== name ) {
          prop[ name ] = prop[ p ];
          delete prop[ p ];
        }

        val = prop[ name ];

        // easing resolution: per property > opt.specialEasing > opt.easing > 'swing' (default)
        // 样式值允许以数组的方式，数组第一个表示动画是否完成，第二个表示动画的值，这样做有两点好处：
        // 1. 允许在某些属性还未完成时就执行回调函数、执行下一个动画
        // 2. 允许为默写属性指定算法
        if ( jQuery.isArray( val ) ) {
          opt.animatedProperties[ name ] = val[ 1 ]; //
          val = prop[ name ] = val[ 0 ];
        } else {
          // 对指定属性设置动画算法
          opt.animatedProperties[ name ] = opt.specialEasing && opt.specialEasing[ name ] || opt.easing || 'swing';
        }

        // 如果属性值是hide，并且已经隐藏，则直接调用回调函数；show同理
```

```
if ( val === "hide" && hidden || val === "show" && !hidden ) {
    return opt.complete.call( this );
}

// 如果是width/height，修正它的：
// overflow
// display
if ( isElement && ( name === "height" || name === "width" ) ) {
    // Make sure that nothing sneaks out
    // Record all 3 overflow attributes because IE does not
    // change the overflow attribute when overflowX and
    // overflowY are set to the same value
    opt.overflow = [ this.style.overflow, this.style.overflowX, this.style.overflowY ];

    // Set display property to inline-block for height/width
    // animations on inline elements that are having width/height
    // animated
    if ( jQuery.css( this, "display" ) === "inline" &&
            jQuery.css( this, "float" ) === "none" ) {
        if ( !jQuery.support.inlineBlockNeedsLayout ) {
            this.style.display = "inline-block";

        } else {
```

```
            display = defaultDisplay( this.nodeName );

            // inline-level elements accept inline-block;
            // block-level elements need to be inline with layout
            if ( display === "inline" ) {
                this.style.display = "inline-block";

            } else {
                this.style.display = "inline";
                this.style.zoom = 1;
            }
        }
    }
}

// 如果是width/height，先设置为超出部分隐藏
if ( opt.overflow != null ) {
    this.style.overflow = "hidden";
}

// 遍历每一个属性，
for ( p in prop ) {
```

```
e = new jQuery.fx( this, opt, p ); // 构造动画fx对象
val = prop[ p ];

// 如果是toggle/show/hide，
if ( rfxtypes.test(val) ) {
  // 如果是toggle，则判断当前是否hidden，如果hidden则show，否则hide
  // 如果不是toggle，说明val是hide/show之一
  e[ val === "toggle" ? hidden ? "show" : "hide" : val ]();

} else {
  parts = rfxnum.exec( val ); // 数值型，1+=/-=，2数值，3单位
  start = e.cur(); // 取出当前值

  // 如果是数值型
  if ( parts ) {
    end = parseFloat( parts[2] ); // 包括了正负号
    unit = parts[3] || ( jQuery.cssNumber[ p ] ? "" : "px" ); // jQuery.cssNumber中包含的是无单位的数值型属性，比如zIndex/zoom

    // We need to compute starting value
    // 计算开始值
    if ( unit !== "px" ) {
      jQuery.style( this, p, (end || 1) + unit);
```

```
            start = ((end || 1) / e.cur()) * start;
            jQuery.style( this, p, start + unit);
          }

          // If a +=/-= token was provided, we're doing a relative animation
          // 如果以+=/-=开头，则做相对动画
          if ( parts[1] ) {
            // 计算出相对值
            end = ( (parts[ 1 ] === "-=" ? -1 : 1) * end ) + start;
          }
          // 开始执行动画
          e.custom( start, end, unit );
        // 不是数值型
        } else {
          // 开始执行动画
          e.custom( start, val, "" );
        }
      }
    }

    // For JS strict compliance
    // 严格遵守？
    return true;
```

```
    });
}
```

8.3 实现思路

jQuery队列的实现依赖于jQuery.data，用数组实现，作为私有数据存储在jQuery的全局变量jQuery.cache中。

l 调用jQuery.queue入队时，如果不传入队列名，则默认为fx（标准动画）

n 队列用数组实现，入队直接调用数组对象的方法push

n 入队的元素必须是函数，或由函数构成的数组

n 所有队列名会自动加上queue后缀，表示这是一个队列

n 如果传入的是数组，则覆盖现有的队列

n 如果不是数组，则直接入队

l 调用jQuery. dequeue出队时，会先调用jQuery.queue取得整个队列，因为队列用数组实现，可以调用数组的shift方法取出第一个元素并执行

n 执行第一个元素时采用function.call(context, args)，由此可以看出jQuery队列只支持函数（这么说不完全准确，fx动画是个特例，会在队列头端插入哨兵inprogress，类型为字符串）

n 出队的元素会自动执行，无论这个元素是不是函数，如果不是函数此时就会抛出异常（这个异常并没有处理）

n 如果队列变成空队列，则用关键delete删除jQuery.cache中type对应的属性

8.4 验证（firefox+firebug）

我们验证一下上面的思路：

1. 先入队3个弹窗函数，分别弹出1、2、3

```
$('body').queue( 'test', function(){ alert(1); } )
```

```
$('body').queue( 'test', function(){ alert(2); } )

$('body').queue( 'test', function(){ alert(3); } )
```

2. 查看jQuery.data为body分配的唯一id（为什么要查看body的唯一id，请参考数据缓存的解析）

```
>>> $.expando

"jQuery161017518149125935123"

command: >>> $('body')[0][$.expando]

5

>>> $('body')[0]["jQuery161017518149125935123"]

5
```

$.expando有三部分构成：字符串"jQuery" + 版本号jQuery.fn.jquery + 随机数Math.random()，因此每次加载页面后都不相同。

3. 查看jQuery.cache对属性5对应的数据，格式化如下：

```
{

  "1" : { ... },

  "2" : { ... },

  "3" : { ... },

  "4" : { ... },

   "5" : {

     "jQuery161017518149125935123" : {

       "testqueue" : [

         (function () {alert(1);}),

         (function () {alert(2);}),
```

```
        (function () {alert(3);})
      ]
    }
  }
}
```

内部数据存储在$.expando属性（"jQuery161017518149125935123"）中，这点区别于普通数据

4. 外事具备，我们出队试试，连续3次调用出队$('body').dequeue('test')，每次调用dequeue后用$('body').queue('test').length检查队列长度

控制台命令	$('body').dequeue('test')	$('body').dequeue('test')	$('body').dequeue('test')
浏览器截图	见附件PDF	见附件PDF	见附件PDF
队列长度	2	1	0

果不其然，调用出队函数dequeue后，入队的函数按照先进先出的顺序，依次被执行

5. 最后看看全部出队后，jQuery.cache中的状态

```
>>> $.cache[5][$.expando]['testqueue']
undefined
```

可以看到，testqueue属性已经从body的缓存中移除

8.5 源码分析

```
jQuery.extend({
    // 计数器，用在动画animate中
    _mark: function( elem, type ) {
```

```
    if ( elem ) {
        type = (type || "fx") + "mark";
        // 取出数据加1，存储在内部对象上
        jQuery.data( elem, type, (jQuery.data(elem,type,undefined,true) || 0) + 1, true );
    }
},
// 用在动画animate中
_unmark: function( force, elem, type ) {
    if ( force !== true ) {
        type = elem;
        elem = force;
        force = false;
    }
    if ( elem ) {
        type = type || "fx";
        var key = type + "mark",
            // 减1
            count = force ? 0 : ( (jQuery.data( elem, key, undefined, true) || 1 ) - 1 );
        if ( count ) {
            jQuery.data( elem, key, count, true );
        } else {
            jQuery.removeData( elem, key, true );
            handleQueueMarkDefer( elem, type, "mark" );
```

```
        }
    }
},
// jQuery.queue( element, [queueName] )
// 返回在指定的元素element上将要执行的函数队列

// jQuery.queue( element, queueName, newQueue or callback )
// 修改在指定的元素element上将要执行的函数队列
// 使用jQuery.queue添加函数后，最后要调用jQuery.dequeue()，使得下一个函数能线性执行
//
// 调用jQuery.data，存储为内部数据（pvt为true）
queue: function( elem, type, data ) {
    // elem必须存在，否则没有意义
    if ( elem ) {
        type = (type || "fx") + "queue"; // 改名，每个都要加上queue
        // 取出队列
        var q = jQuery.data( elem, type, undefined, true );
        // Speed up dequeue by getting out quickly if this is just a lookup
        // 如果data存在，才会进行后边转换数组、入队等操作，可以加速取出整个队列
        if ( data ) {
            // 如果队列不存在，或者data是数组，则调用makeArray转换为数组，并覆盖队列（入队）
            if ( !q || jQuery.isArray(data) ) {
                // 用数组实现队列
```

```
            q = jQuery.data( elem, type, jQuery.makeArray(data), true );
        } else {
            // 队列存在的话，且data不是数组，直接入队
            // 这里并没有判断data的类型，不管data是不是函数
            q.push( data );
        }
    }
    // 返回队列（即入队的同时，返回整个队列）
    // 简洁实用的避免空引用的技巧
    return q || [];
  }
},
// 出队并执行
// 调用jQuery.queue取得整个队列，在调用shift取出第一个元素
dequeue: function( elem, type ) {
  type = type || "fx"; // 默认fx，但是入队时不是被改为fxqueue了么，别着急！

  var queue = jQuery.queue( elem, type ), // 取出队列，调用queue时type变成了type+quque
    fn = queue.shift(), // 取出第一个
    defer;

  // If the fx queue is dequeued, always remove the progress sentinel
  // 如果取出的fn是一个正在执行中标准动画fx，抛弃执行哨兵（inprogress），再取一个
```

```
    if ( fn === "inprogress" ) {
        fn = queue.shift();
    }

    if ( fn ) {
        // Add a progress sentinel to prevent the fx queue from being
        // automatically dequeued
        // 如果是标准动画，则在队列头部增加处理中哨兵属性，阻止fx自动处理
        if ( type === "fx" ) {
            // 在队列头部增加哨兵inprogress
            queue.unshift("inprogress");
        }
        // 执行取出的fn，并传入回调函数jQuery.dequeue
        // 可以看到fn必须是函数，否则会出错
        fn.call(elem, function() {
            // 但是这个回调函数不会自动执行
            jQuery.dequeue(elem, type);
        });
    }
    // 如果执行完毕，则调用jQuery.removeData移除type指定的队列
    // 此时的队列成为空队列，实质是一个空数组，jQuery.removeData内部使用delete关键字删除type对应的空数组
    if ( !queue.length ) {
```

```
            jQuery.removeData( elem, type + "queue", true );
            handleQueueMarkDefer( elem, type, "queue" );
        }
    }
});

jQuery.fn.extend({
    // queue( [ queueName ] )
    // 返回在指定的元素element上将要执行的函数队列
    //
    // queue( [ queueName ], newQueue or callback )
    // 修改在指定的元素element上将要执行的函数队列
    queue: function( type, data ) {
        // 修正参数：只传了一个非字符串的参数，则默认为动画fx
        if ( typeof type !== "string" ) {
            data = type;
            type = "fx";
        }
        // 如果data等于undefined，则认为是取队列
        if ( data === undefined ) {
            return jQuery.queue( this[0], type );
        }
        // 如果传入了data参数，则在每一个匹配的元素上执行入队操作
```

```
    // 在jQuery中，使用each遍历匹配的元素，是一种安全的惯例做法
    return this.each(function() {
        var queue = jQuery.queue( this, type, data );
        // 如果动画执行完毕（即不是inprogress），则从队列头部移除
        if ( type === "fx" && queue[0] !== "inprogress" ) {
            jQuery.dequeue( this, type );
        }
    });
},
// 调用jQuery.dequeue出队
dequeue: function( type ) {
    return this.each(function() {
        jQuery.dequeue( this, type );
    });
},
// Based off of the plugin by Clint Helfers, with permission.
// http://blindsignals.com/index.php/2009/07/jquery-delay/
// 延迟执行队列中未执行的函数，通过在队列中插入一个延时 出队的函数来实现
delay: function( time, type ) {
    time = jQuery.fx ? jQuery.fx.speeds[time] || time : time;
    type = type || "fx";

    return this.queue( type, function() {
```

```
        var elem = this;
        setTimeout(function() {
          jQuery.dequeue( elem, type );
        }, time );
      });
    },
    // 清空队列，通过将第二参数设置为空数组[]
    clearQueue: function( type ) {
      return this.queue( type || "fx", [] );
    },
    // Get a promise resolved when queues of a certain type
    // are emptied (fx is the type by default)
    // 返回一个只读视图，当队列中指定类型的函数执行完毕后
    promise: function( type, object ) {
      if ( typeof type !== "string" ) {
        object = type;
        type = undefined;
      }
      type = type || "fx";
      var defer = jQuery.Deferred(),
        elements = this,
        i = elements.length,
        count = 1,
```

```
        deferDataKey = type + "defer",
        queueDataKey = type + "queue",
        markDataKey = type + "mark",
        tmp;
    function resolve() {
        if ( !( --count ) ) {
            defer.resolveWith( elements, [ elements ] );
        }
    }
    while( i-- ) {
        if (( tmp = jQuery.data( elements[ i ], deferDataKey, undefined, true ) ||
                ( jQuery.data( elements[ i ], queueDataKey, undefined, true ) ||
                    jQuery.data( elements[ i ], markDataKey, undefined, true ) ) &&
                jQuery.data( elements[ i ], deferDataKey, jQuery._Deferred(), true ) )) {
            count++;
            tmp.done( resolve );
        }
    }
    resolve();
    return defer.promise();
  }
});
```

8.6 总结

jQuery队列的实现并不复杂，它的核心思路是：

在传统队列的实现上增加了出队自动执行，执行完成后再次自动出队。

详见小节：8.3 实现思路

附件下载:

- _原创__jQuery源码分析-08队列-Queue.pdf (396.5 KB)
- dl.iteye.com/topics/download/10fbd9a9-e69a-3332-80e1-5e297c779fbd

1.8 [原创] jQuery源码分析-09属性操作

发表时间: 2011-09-14 关键字: javascript, jqeury, attr, prop, web

属性操作主要介绍prop、attr、val三个接口的实现，相对于其他的接口，这三个的源码实现复杂，更容易让人混淆，一不小心就回使用错误的接口或返回错误的值，因此重点分析。

9.1 .prop() vs .attr()

9.1.1 概述

1.6.1相对1.5.x最大的改进，莫过于对属性.attr()的重写了。在1.6.1中，将.attr()一分为二： .attr()、.prop()，这是一个令人困惑的变更，也是一个破坏性的升级，会直接影响到无数的网站和项目升级到1.6。

简单的说，.attr()是通过setAttribute、getAttribute实现，.prop()则通过Element[name]实现：

jQuery.attr	setAttribute, getAttribute
jQuery.removeAttr	removeAttribute, removeAttributeNode(getAttributeNode)
jQuery.prop	Element[name]
jQuery.removeProp	delete Element[name]

事实上.attr()和.prop()的不同，是HTML属性（HTML attributes）和DOM属性（DOM properties）的不同。HTML属性解析的是HTML代码中的存在的属性，返回的总是字符串，而DOM属性解析的是DOM对象的属性，可能是字符串，也可能是一个对象，可能与HTML属性相同，也可能不同。

9.1.2 测试

看个例子，让我们对HTML属性和DOM属性的区别有个直观的概念：

HTML代码：

```
<a href="abc.html" class="csstest" style="font-size: 30px;">link</a>
<input type="text" value="123">
<input type="checkbox" checked="checked">
```

JavaScript代码：

```
console.info( $('#a').attr('href') ); // abc.html
console.info( $('#a').prop('href') ); // file:///H:/open/ws-nuysoft/com.jquery/jquery/abc.html

console.info( $('#a').attr('class') ); // csstest
console.info( $('#a').prop('class') ); // csstest

console.info( document.getElementById('a').getAttribute('class') ); // csstest
console.info( document.getElementById('a').className ); // csstest
```

```
console.info( $('#a').attr('style') ); // font-size: 30px;
console.info( $('#a').prop('style') ); // CSSStyleDeclaration { 0="font-size", fontSize="30px", ...}
console.info( document.getElementById('a').getAttribute('style') ); // font-size: 30px;
console.info( document.getElementById('a').style ); // CSSStyleDeclaration { 0="font-size", fontSize="30px", ...}

console.info( $('#text').attr('value') ); // 123
console.info( $('#text').prop('value') ); // 123

console.info( $('#checkbox').attr('checked') ); // checked
console.info( $('#checkbox').prop('checked') ); // true
```

可以看到HTML属性和DOM属性在属性名、属性值上都诸多不同。

9.1.3 区别

不同之处总结如下：

? 属性名可能不同，尽管大部分的属性名还是相似或一致的

? HTML属性值总是返回字符串，DOM属性值则可能是整型、字符串、对象，可以获取更多的内容

? DOM属性总是返回当前的状态（值），而HTML属性（在大多数浏览）返回的初始化时的状态（值）

? DOM属性只能返回固定属性名的值，而HTML属性则可以返回在HTML代码中自定义的属性名的值

? 相对于HTML属性的浏览器兼容问题，DOM属性名和属性值在浏览器之间的差异更小，并且DOM属性也有标准可依

9.1.4 建议

下边让我们回到.attr()和.prop()，经过以上测试和分析，可以得出对.attr()和.prop()的使用建议如下

? 优先使用.prop()，因为.prop()总是返回最新的状态（值）

? 只有涉及到自定义HTML属性时使用.attr()，或者可以说，忘掉.attr()吧

9.1.5 源码

? jQuery.attr

```
// 设置或获取HTML属性
// http://stackoverflow.com/questions/5874652/prop-vs-attr
attr: function( elem, name, value, pass ) {
    var nType = elem.nodeType;

    // don't get/set attributes on text, comment and attribute nodes
    // 忽略文本、注释、属性节点
    if ( !elem || nType === 3 || nType === 8 || nType === 2 ) {
      return undefined;
    }
    // 遇到与方法同名的属性，则执行方法
```

```
// 如果遇到的是扩展或需要修正的属性，则执行相应的方法
if ( pass && name in jQuery.attrFn ) {
  return jQuery( elem )[ name ]( value );
}

// Fallback to prop when attributes are not supported
// 如果不支持getAttribute，则调用jQuery.prop
// 求助于prop？prop是从attr中分化出来的，居然要求助于prop，看来attr要被放弃了
if ( !("getAttribute" in elem) ) {
  return jQuery.prop( elem, name, value );
}

var ret, hooks,
  notxml = nType !== 1 || !jQuery.isXMLDoc( elem );

// Normalize the name if needed
// 格式化name（修正tabindex > tabIndex）
name = notxml && jQuery.attrFix[ name ] || name;
// 属性钩子：type tabIndex
hooks = jQuery.attrHooks[ name ];

// 如果没有name对应的钩子
```

```
    if ( !hooks ) {
      // Use boolHook for boolean attributes
      // 使用boolean钩子处理boolean属性
      if ( rboolean.test( name ) &&
        (typeof value === "boolean" || value === undefined || value.toLowerCase() === name.toLowerCase()) )
{
        // 使用 布尔钩子（静态方法对象）：set get
        hooks = boolHook;

      // Use formHook for forms and if the name contains certain characters
      // 使用表单钩子
      } else if ( formHook && (jQuery.nodeName( elem, "form" ) || rinvalidChar.test( name )) ) {
        // 使用 表单钩子（静态方法对象）：set get
        hooks = formHook;
      }
    }

    // 如果value已定义，则设置或移除
    // 设置
    if ( value !== undefined ) {
      // typeof null === 'object' // true
      // typeof undefined === 'undefined' // true
      // null == undefined true
```

```
    // null === undefined false
    // value为null，则移除name属性
    // 注意这里用的都是恒等号
    if ( value === null ) {
        jQuery.removeAttr( elem, name );
        return undefined;

    }
    // 属性钩子、布尔钩子、表单钩子，如果有对应的钩子，则调用钩子的set方法
    else if ( hooks && "set" in hooks && notxml && (ret = hooks.set( elem, value, name )) !== undefined ) {
        return ret;

    } else {
        // 最后的最后，还是调用setAttribute，前边的各种钩子，都是修正属性
        // 强制将value转换为字符串
        elem.setAttribute( name, "" + value );
        return value;
    }
// 如果value是undefined，说明是取属性值，如果对应的钩子的有get方法，则调用钩子的get方法
} else if ( hooks && "get" in hooks && notxml ) {
    return hooks.get( elem, name );
```

```
    } else {
        // 最后的最后：getAttribute
        ret = elem.getAttribute( name );

        // Non-existent attributes return null, we normalize to undefined
        // 不存在的属性返回null，格式化为undefined
        return ret === null ?
            undefined :
            ret;
    }
}
```

? jQuery.prop

```
// 设置或获取DOM属性
prop: function( elem, name, value ) {
    var nType = elem.nodeType;

    // don't get/set properties on text, comment and attribute nodes
    // 忽略文本、注释、属性节点
    if ( !elem || nType === 3 || nType === 8 || nType === 2 ) {
        return undefined;
```

```
}

var ret, hooks,
  notxml = nType !== 1 || !jQuery.isXMLDoc( elem );

// Try to normalize/fix the name
// 属性名name修正
name = notxml && jQuery.propFix[ name ] || name;

hooks = jQuery.propHooks[ name ];
// 设置
// 看着prop的实现是不是很眼熟？嗯，和attr的思路类似！
if ( value !== undefined ) {
  // 如果钩子存在set方法，则调用钩子的set方法
  if ( hooks && "set" in hooks && (ret = hooks.set( elem, value, name )) !== undefined ) {
    return ret;

  } else {
    return (elem[ name ] = value);
  }
// 读取
} else {
```

```
        if ( hooks && "get" in hooks && (ret = hooks.get( elem, name )) !== undefined ) {
            return ret;

        } else {
            return elem[ name ];
        }
    }
}
```

? jQuery.fn.attr、jQuery.fn.prop

内部通过jQuery.access调用jQuery.attr、jQuery.prop实现

```
attr: function( name, value ) {
    return jQuery.access( this, name, value, true, jQuery.attr );
},

prop: function( name, value ) {
    return jQuery.access( this, name, value, true, jQuery.prop );
}
```

? jQuery.access

```
// Mutifunctional method to get and set values to a collection
// The value/s can be optionally by executed if its a function
// 多功能函数，读取或设置集合的属性值；值为函数时会被执行
// fn : jQuery.fn.css, jQuery.fn.attr, jQuery.fn.prop
access: function( elems, key, value, exec, fn, pass ) {
    var length = elems.length;

    // Setting many attributes
    // 如果有多个属性，则迭代
    if ( typeof key === "object" ) {
      for ( var k in key ) {
        jQuery.access( elems, k, key[k], exec, fn, value );
      }
      return elems;
    }

    // Setting one attribute
    // 只设置一个属性
    if ( value !== undefined ) {
      // Optionally, function values get executed if exec is true
      exec = !pass && exec && jQuery.isFunction(value);
      // 调用fn
```

```
        for ( var i = 0; i < length; i++ ) {
            fn( elems[i], key, exec ? value.call( elems[i], i, fn( elems[i], key ) ) : value, pass );
        }

        return elems;
    }

    // Getting an attribute
    // 读取属性
    return length ? fn( elems[0], key ) : undefined;
}
```

1.9 [原创] jQuery源码分析-10事件处理-Event-概述和基础知识

发表时间: 2011-10-12 关键字: javascript, jquery, 源码分析, 事件event, 事件模型

作者：nuysoft/高云 QQ：47214707 EMail：nuysoft@gmail.com

读读写写，不对的地方请告诉我，多多交流共同进步，本章的的PDF下载在最后。

前记：

关于缺少示例，首先感谢这位朋友的反馈。在读源码的过程中，必然要写很多例子来跟踪代码的调用过程，经常还需要单步调试，遇到难度的地方，比如动画、AJAX要反复尝试，尽量覆盖各个分支，这个过程如果用文字记录下来的话会很长，而且未必能讲清楚，因为不像DEMO看效果就可以了，最好的办法是录成视频，这个有点遥远看时间和学的情况吧。

关于入门的示例，有的朋友建议多做一些入门教程，我的建议是，去看官网，官网有很全的DEMO，详细的解释，其他的入门教程也是抄官网的，官网上的很多评论很有价值值得仔细看看

关于造轮子，有的人在分析的过程中，会尝试用原生js去实现一个类似的功能，这是个好习惯；本系列的目的是深入的去分析jQuery的思想和技巧，如果理解透彻了模仿一个类似的自然水到渠成，我会在分析的过程将思路、用到的原生js都分解出来。重复造轮子是好事，而且必须要重复造，不重复造怎能明白原创的精髓，不重复造怎么能创新，关键的关键在于：多造少用，造出来的轮子暂时让它留在实验室里，就像我的篮球老师（同事）告诉我的，在平时练球时，要多用你不熟悉的那只手，但是比赛时要用你擅长的那只手，就是多练少用。直到你的双手都娴熟了，直到你的轮子达到原创的水平了，才能把它应用到项目中。

本章后续小节预告：

封装事件对象 接口应用 接口调用链 源码解析 DOMReady专题

10. 事件处理

10.1 概述

做为JavaScript库，首先要解决浏览器事件兼容问题，正确的管理事件，并提供便捷的接口方便开发；jQuery的事件模型以简单优雅的方式实现了这些需求：

n 自定义jQuery.Event对象，模拟实现W3C标准的DOM 3级别事件模型，统一了事件的属性和方法

n 可以在一个事件类型上添加多个事件处理函数，可以一次添加多个事件类型的事件处理函数

n 提供了常用事件的便捷方法，例如 $(selector).click(function ...)；同时支持通过代码触发事件，例如 $(selector).click( )

n 支持自定义事件

n 扩展了组合事件：toggle（click时顺序执行绑定的事件）、hover（鼠标移入、移除）

n 扩展了one（只执行一次）、live-die（延迟绑定）、delegate-undelegate（类似live）

n 提供了统一的事件封装、绑定、执行、销毁机制

n 优化DOM ready事件

事件的管理不外乎绑定、触发、销毁（详见下一节基础知识），jQuery则在浏览器原生的支持上进行了封装和完善。

jQuery并没有将事件处理函数直接绑定到DOM元素上，而是通过$.data存储在缓存$.cahce上；首先为DOM元素分配一个唯一ID，绑定的事件存储在$.cahce[唯一ID][$.expand]['events']上，而events是个键-值映射对象，键就是事件类型，对应的值就是由事件处理函数组成的数组；最后在DOM元素上绑定（addEventListener/ attachEvent）一个事件处理函数eventHandle，这个过程由 jQuery.event.add 实现。

当事件触发时eventHandle被执行，eventHandle再去$.cache中寻找曾经绑定的事件处理函数并执行，这个过程由 jQuery.event. trigger 和 jQuery.event.handle实现。

事件的销毁则由jQuery.event.remove 实现，remove对缓存$.cahce中存储的事件数组进行销毁，当缓存中的事件全部销毁时，调用removeEventListener/ detachEvent销毁绑定在DOM元素上的事件处理函数eventHandle。

10.2 基础知识

在我们对jQuery的事件模型进行介绍和分析之前，如果读者不熟悉浏览器的事件模型，建议先读一读《JavaScript权威指南》第17章 事件和事件处理。以下是关键内容的笔记：

l 0级事件模型（所有浏览器都支持的实际标准API）

n 方式一：作为JavaScript属性的事件句柄（将函数直接绑定到事件属性上）

```
elment.onclick = function(){
    // ...
}
```

n 方式二：作为HTML属性的事件句柄（将函数句柄以字符串的方式直接赋值给DOM元素的HTML属性）

```
<input type="button" value="0级事件模型" onclick="alert(this.nodeName);" />
```

l 2级事件模型（除Internet Explorer以外的所有现在浏览器都支持2级DOM事件模型）

n 事件传播分三个阶段进行：

u 捕捉阶段，事件从Document对象沿着文档树向下传播给目标节点。如果目标的任何一个祖先（不是目标自身）专门注册了捕捉事件句柄，那么在事件传递过程中，就会运行这些句柄。

u 目标阶段，直接注册在目标上的适合的事件句柄将运行。这与0级事件模型提供的事件处理方法相似

u 起泡阶段，事件将从目标元素向上传播回或起泡回Document对象的文档层次。

虽然所有事件都受事件传播的捕捉阶段的支配，但并非所有类型的事件都起泡。一般来说，原始输入事件起泡（大部分事件），而高级语义事件不起泡（blue focus load unload）

n 注册一个事件句柄 Element.addEventListener(String type, Function listener, boolean useCapture)

u type 事件监听器调用所针对的时间类型。例如，load click mousedown

u listener 当指定类型的事件分派给这个元素的时候，事件监听器函数被调用。调用的时候，这个监听器函数被传递给一个Event对象，然后作为在该元素上注册的一个方法来调用

u useCapture 如果为true，那么只有在事件传播的捕捉阶段，指定的listener才会被调用。更常用的值是false，意味着在捕捉阶段不会调用listener，而当该节点是实际的事件目标或事件从它的原始目标起泡到该节点时，调用listener。

该方法将指定的事件监听器函数添加到当前节点的监听器函数结合中，以处理指定类型type的事件。如果useCapture为true，则监听器被注册为捕捉阶段监听器。如果useCapture为false，它就注册普通事件监听器

addEventListener()可能被调用多次，在同一个节点上为同一种类型的事件注册多个事件句柄。但要注意，DOM不能确定多个事件句柄被调用的顺序。

如果一个事件监听器函数在同一个节点上用相同的type和useCapture参数注册了两次，那么第二次注册将被忽略。如果正在处理一个节点上的事件时，在这个节点上注册了一个新的事件监听器，则不会为那个事件调用新的事件监听器。

当用Node.cloneNode()方法或Document.importNode()方法复制一个Document节点时，不会复制为原始节点注册的事件监听器。

n 删除一个事件句柄 Element.removeEventListener(String type, Function listener, boolean useCapture)

u type 要删除事件监听器的事件类型

u listener 要删除的事件监听器函数

u useCapture 如果要删除是捕捉事件监听器，则为true；如果要删除的是普通事件监听器，则为false。

该方法将删除指定的事件监听器函数。参数type和useCapture必须与调用addEventListener()方法的相应参数一致。如果没有找到与指定的参数匹配的事件监听器，该方法什么都不做。

如果一个事件监听器函数被该方法删除，那么当节点发生指定类型的事件时，就不再调用它。即使一个事件监听器被同一个节点上同类型事件注册的另一个事件监听器删除，它也不再被调用。

n 停止事件传播 Event.stopPropagation()

n 阻止默认动作 Event.preventDefault()

l IE事件模型

n 注册一个事件句柄 Element.attachEvent(String type, Function listener)

u type 事件监听器调用所针对的事件的类型，带有一个“on”前缀。例如，onload onclick onmousedown

u listener 当指定类型的事件分派给这个元素的时候，事件监听器函数被调用。不会为这个函数传递任何参数，但是它可以从Window对象的event属性获得Event对象。

这个方法是一个特定与IE的事件注册方法。它和标准的addEventListener()方法（IE不支持它）具有相同的作用，但是，它和该函数也有一些重要的差别：

u 由于IE事件模型不支持事件捕获，所以attachEvent()和detachEvent()只需要两个参数：事件类型和句柄函数

u 传递给IE方法的事件句柄名应该包含on前缀。例如，和attachEvent()一起使用的是onlick，而不是和addEventListener()一起使用的click

u 使用attachEvent()注册的函数调用的时候没有Event对象参数。相反，它们必须读取Window对象的event属性

u 使用attachEvent()注册的函数作为全局函数调用，而不是作为事件发生其上的文档元素的方法调用。也就是说，当使用attachEvent()注册的一个事件句柄执行的时候，this关键字引用的是Window对象，而不是事件目标元素。

u attachEvent()允许同一个事件句柄函数注册多次。当制定类型的一个事件发生的时候，注册的哈苏调用次数与它注册的次数相同

n 删除一个事件监听器 Event.detachEvent(String type, Function listener)

u type 要删除的事件监听器所针对的事件的类型，带有一个on前缀。例如 onclick

u listener 要删除事件监听器函数

这个方法解除掉由attachEvent()方法所执行的事件句柄函数注册。它是removeEventListener()方法的特定与IE的替代。要为一个元素删除一个事件函数句柄，只需要使用你最初传递attachEvent()的相同参数来调用detachEvent()

n 停止事件传播 window.event.cancelBubule = true;

n 阻止默认动作 window.event.returnValue = false

1.10 [原创] jQuery源码分析-15AJAX-前置过滤器和请求分发器

发表时间: 2011-09-23 关键字: web, javascript, ajax, 前置过滤器, 请求分发器

边读边写，不正确的地方，还请各位告诉我，多多交流共同学习。

15.4 AJAX中的前置过滤器和请求分发器

自jQuery1.5以后，AJAX模块提供了三个新的方法用于管理、扩展AJAX请求，分别是：

l 前置过滤器 jQuery. ajaxPrefilter

l 请求分发器 jQuery. ajaxTransport，

l 类型转换器 ajaxConvert

这里先分析前置过滤器和请求分发器，类型转换器下一节再讲。

15.4.1 前置过滤器和请求分发器的初始化

前置过滤器和请求分发器在执行时，分别遍历内部变量prefilters和transports，这两个变量在jQuery加载完毕后立即初始化，初始化的过程很有意思。

首先，prefilters和transports被置为空对象：

```
prefilters = {}, // 过滤器

transports = {}, // 分发器
```

然后，创建jQuery.ajaxPrefilter和jQuery.ajaxTransport，这两个方法都调用了内部函数addToPrefiltersOrTransports，addToPrefiltersOrTransports返回一个匿名闭包函数，这个匿名闭包函数负责将单一前置过滤和单一请求分发器分别放入prefilters和transports。我们知道闭包会保持对它所在环境变量的引用，而jQuery.ajaxPrefilter和jQuery.ajaxTransport的实现又完全一样,都是对Map结构的对象进行赋值操作,因此这里利用闭包的特性巧妙的将两个方法的实现合二为一。函数addToPrefiltersOrTransports可视为模板模式的一种实现。

```
ajaxPrefilter: addToPrefiltersOrTransports( prefilters ), // 通过闭包保持对prefilters的引用，将前置过滤器添加到prefilters
```

```
ajaxTransport: addToPrefiltersOrTransports( transports ), // 通过闭包保持对transports的引用，将请求分发器添加到transports


// 添加全局前置过滤器或请求分发器，过滤器的在发送之前调用，分发器用来区分ajax请求和script标签请求
function addToPrefiltersOrTransports( structure ) {
    // 通过闭包访问structure
    // 之所以能同时支持Prefilters和Transports，关键在于structure引用的时哪个对象
    // dataTypeExpression is optional and defaults to "*"
    // dataTypeExpression是可选参数，默认为*
    return function( dataTypeExpression, func ) {
        // 修正参数
        if ( typeof dataTypeExpression !== "string" ) {
            func = dataTypeExpression;
            dataTypeExpression = "*";
        }

        if ( jQuery.isFunction( func ) ) {
            var dataTypes = dataTypeExpression.toLowerCase().split( rspacesAjax ), // 用空格分割数据类型表达式dataTypeExpression
                i = 0,
                length = dataTypes.length,
                dataType,
                list,
```

```
            placeBefore;


        // For each dataType in the dataTypeExpression
        for(; i < length; i++ ) {
            dataType = dataTypes[ i ];
            // We control if we're asked to add before
            // any existing element
            // 如果以+开头，过滤+
            placeBefore = /^\+/.test( dataType );
            if ( placeBefore ) {
                dataType = dataType.substr( 1 ) || "*";
            }
            list = structure[ dataType ] = structure[ dataType ] || [];
            // then we add to the structure accordingly
            // 如果以+开头，则插入开始位置，否则添加到末尾
            // 实际上操作的是structure
            list[ placeBefore ? "unshift" : "push" ]( func );
        }
    }
};
}
```

最后，分别调用jQuery.ajaxPrefilter和jQuery.ajaxTransport填充prefilters和transports.

填充prefilters:

```
// Detect, normalize options and install callbacks for jsonp requests
// 向前置过滤器对象中添加特定类型的过滤器
// 添加的过滤器将格式化参数，并且为jsonp请求增加callbacks
// MARK：AJAX模块初始化
jQuery.ajaxPrefilter( "json jsonp", function( s, originalSettings, jqXHR ) {

    var inspectData = s.contentType === "application/x-www-form-urlencoded" &&
        ( typeof s.data === "string" ); // 如果是表单提交，则需要检查数据

    // 这个方法只处理jsonp，如果json的url或data有jsonp的特征，会被当成jsonp处理
    // 触发jsonp的3种方式：
    if ( s.dataTypes[ 0 ] === "jsonp" || // 如果是jsonp
        s.jsonp !== false && ( jsre.test( s.url ) || // 未禁止jsonp，s.url中包含=?& =?$ ??
                inspectData && jsre.test( s.data ) ) ) { // s.data中包含=?& =?$ ??

        var responseContainer,
            jsonpCallback = s.jsonpCallback =
                jQuery.isFunction( s.jsonpCallback ) ? s.jsonpCallback() : s.jsonpCallback, // s.jsonpCallback时函数，则执行函数用返回值做为回调函数名
            previous = window[ jsonpCallback ],
            url = s.url,
            data = s.data,
            // jsre = /(\=)\?(&|$)|\?\?/i; // =?& =?$ ??
```

```
        replace = "$1" + jsonpCallback + "$2"; // $1 =, $2 &|$


    if ( s.jsonp !== false ) {
        url = url.replace( jsre, replace ); // 将回调函数名插入url
        if ( s.url === url ) { // 如果url没有变化，则尝试修改data
            if ( inspectData ) {
                data = data.replace( jsre, replace ); // 将回调函数名插入data
            }
            if ( s.data === data ) { // 如果data也没有变化
                // Add callback manually
                url += (/\?/.test( url ) ? "&" : "?") + s.jsonp + "=" + jsonpCallback; // 自动再url后附加回调函
数名
            }
        }
    }

    // 存储可能改变过的url、data
    s.url = url;
    s.data = data;

    // Install callback
    window[ jsonpCallback ] = function( response ) { // 在window上注册回调函数
        responseContainer = [ response ];
```

```
};

// Clean-up function
jqXHR.always(function() {
    // Set callback back to previous value
    // 将备份的previous函数恢复
    window[ jsonpCallback ] = previous;
    // Call if it was a function and we have a response
    // 响应完成时调用jsonp回调函数，问题是这个函数不是自动执行的么？
    if ( responseContainer && jQuery.isFunction( previous ) ) {
        window[ jsonpCallback ]( responseContainer[ 0 ] ); // 为什么要再次执行previous呢？
    }
});

// Use data converter to retrieve json after script execution
s.converters["script json"] = function() {
    if ( !responseContainer ) { // 如果
        jQuery.error( jsonpCallback + " was not called" );
    }
    return responseContainer[ 0 ]; // 因为是作为方法的参数传入，本身就是一个json对象，不需要再做转换
};
```

```
        // force json dataType
        s.dataTypes[ 0 ] = "json"; // 强制为json

        // Delegate to script
        return "script"; // jsonp > json
    }
});
// Handle cache's special case and global
// 设置script的前置过滤器，script并不一定意思着跨域
// MARK：AJAX模块初始化
jQuery.ajaxPrefilter( "script", function( s ) {
    if ( s.cache === undefined ) { // 如果缓存未设置，则设置false
        s.cache = false;
    }
    if ( s.crossDomain ) { // 跨域未被禁用，强制类型为GET，不触发全局时间
        s.type = "GET";
        s.global = false;
    }
});
```

填充transports：

```
// Bind script tag hack transport
// 绑定script分发器,通过在header中创建script标签异步载入js,实现过程很简介
```

```
// MARK：AJAX模块初始化

jQuery.ajaxTransport( "script", function(s) {


    // This transport only deals with cross domain requests

    if ( s.crossDomain ) { // script可能时json或jsonp，jsonp需要跨域，ajax模块大约有1/3的代码时跨域的

        // 如果在本域中设置了跨域会怎么处理呢？


        var script,

            head = document.head || document.getElementsByTagName( "head" )[0] ||
document.documentElement; // 充分利用布尔表达式的计算顺序


        return {


            send: function( _, callback ) { // 提供与同域请求一致的接口


                script = document.createElement( "script" ); // 通过创script标签来实现


                script.async = "async";


                if ( s.scriptCharset ) {

                    script.charset = s.scriptCharset; // 字符集

                }
```

```
            script.src = s.url; // 动态载入

            // Attach handlers for all browsers
            script.onload = script.onreadystatechange = function( _, isAbort ) {

                if ( isAbort || !script.readyState || /loaded|complete/.test( script.readyState ) ) {

                    // Handle memory leak in IE
                    script.onload = script.onreadystatechange = null; // onload事件触发后，销毁事件句柄，因为IE内存泄漏？

                    // Remove the script
                    if ( head && script.parentNode ) {
                        head.removeChild( script ); // onloda后，删除script节点
                    }

                    // Dereference the script
                    script = undefined; // 注销script变量

                    // Callback if not abort
                    if ( !isAbort ) {
                        callback( 200, "success" ); // 执行回调函数，200为HTTP状态码
                    }
```

```
                }
            };
            // Use insertBefore instead of appendChild  to circumvent an IE6 bug.
            // This arises when a base node is used (#2709 and #4378).
            // 用insertBefore代替appendChild，如果IE6的bug
            head.insertBefore( script, head.firstChild );
        },

        abort: function() {
            if ( script ) {
                script.onload( 0, 1 ); // 手动触发onload事件,jqXHR状态码为0,HTTP状态码为1xx
            }
        }
    };
}
});

// Create transport if the browser can provide an xhr
if ( jQuery.support.ajax ) {
    // MARK：AJAX模块初始化
    // 普通AJAX请求分发器，dataType默认为*
    jQuery.ajaxTransport(function( s ) { // *
        // Cross domain only allowed if supported through XMLHttpRequest
```

```
// 如果不是跨域请求，或支持身份验证
if ( !s.crossDomain || jQuery.support.cors ) {

    var callback;

    return {
      send: function( headers, complete ) {

        // Get a new xhr
        // 创建一个XHR
        var xhr = s.xhr(),
          handle,
          i;

        // Open the socket
        // Passing null username, generates a login popup on Opera (#2865)
        // 调用XHR的open方法
        if ( s.username ) {
          xhr.open( s.type, s.url, s.async, s.username, s.password ); // 如果需要身份验证
        } else {
          xhr.open( s.type, s.url, s.async );
        }
```

```
// Apply custom fields if provided
// 在XHR上绑定自定义属性
if ( s.xhrFields ) {
  for ( i in s.xhrFields ) {
    xhr[ i ] = s.xhrFields[ i ];
  }
}

// Override mime type if needed
// 如果有必要的话覆盖mineType,overrideMimeType并不是一个标准接口,因此需要做特性检测
if ( s.mimeType && xhr.overrideMimeType ) {
  xhr.overrideMimeType( s.mimeType );
}

// X-Requested-With header
// For cross-domain requests, seeing as conditions for a preflight are
// akin to a jigsaw puzzle, we simply never set it to be sure.
// (it can always be set on a per-request basis or even using ajaxSetup)
// For same-domain requests, won't change header if already provided.
// X-Requested-With同样不是一个标注HTTP头,主要用于标识Ajax请求.大部分JavaScript框架将这个头设置为XMLHttpRequest
if ( !s.crossDomain && !headers["X-Requested-With"] ) {
  headers[ "X-Requested-With" ] = "XMLHttpRequest";
```

```
}

// Need an extra try/catch for cross domain requests in Firefox 3
// 设置请求头
try {
  for ( i in headers ) {
      xhr.setRequestHeader( i, headers[ i ] );
  }
} catch( _ ) {}

// Do send the request
// This may raise an exception which is actually
// handled in jQuery.ajax (so no try/catch here)
// 调用XHR的send方法
xhr.send( ( s.hasContent && s.data ) || null );

// Listener
// 封装回调函数
callback = function( _, isAbort ) {

  var status,
      statusText,
      responseHeaders,
```

```
                responses, // 响应内容,格式为text:text, xml:xml
                xml;

            // Firefox throws exceptions when accessing properties
            // of an xhr when a network error occured
            // http://helpful.knobs-dials.com/index.php/
Component_returned_failure_code:_0x80040111_(NS_ERROR_NOT_AVAILABLE)
            // 在FF下当网络异常时,访问XHR的属性会抛出异常
            try {

                // Was never called and is aborted or complete
                if ( callback && ( isAbort || xhr.readyState === 4 ) ) { // 4表示响应完成

                    // Only called once
                    callback = undefined; // callback只调用一次,注销callback

                    // Do not keep as active anymore
                    if ( handle ) {
                        xhr.onreadystatechange = jQuery.noop; // 将onreadystatechange句柄重置为空函数
                        if ( xhrOnUnloadAbort ) { // 如果是界面退出导致本次请求取消
                            delete xhrCallbacks[ handle ]; // 注销句柄
                        }
                    }
```

```
// If it's an abort
if ( isAbort ) { // 如果是取消本次请求
    // Abort it manually if needed
    if ( xhr.readyState !== 4 ) {
        xhr.abort(); // 调用xhr原生的abort方法
    }
} else {
    status = xhr.status;
    responseHeaders = xhr.getAllResponseHeaders();
    responses = {};
    xml = xhr.responseXML;

    // Construct response list
    if ( xml && xml.documentElement /* #4958 */ ) {
        responses.xml = xml; // 提取xml
    }
    responses.text = xhr.responseText; // 提取text

    // Firefox throws an exception when accessing
    // statusText for faulty cross-domain requests
    // FF在跨域请求中访问statusText会抛出异常
    try {
```

```
                statusText = xhr.statusText;
            } catch( e ) {
                // We normalize with Webkit giving an empty statusText
                statusText = ""; // 像WebKit一样将statusText置为空字符串
            }

            // Filter status for non standard behaviors

            // If the request is local and we have data: assume a success
            // (success with no data won't get notified, that's the best we
            // can do given current implementations)
            // 过滤不标准的服务器状态码
            if ( !status && s.isLocal && !s.crossDomain ) {
                status = responses.text ? 200 : 404; //
            // IE - #1450: sometimes returns 1223 when it should be 204
            // 204 No Content
            } else if ( status === 1223 ) {
                status = 204;
            }
        }
    }
} catch( firefoxAccessException ) {
    if ( !isAbort ) {
```

```
                complete( -1, firefoxAccessException ); // 手动调用回调函数
            }
        }

        // Call complete if needed
        // 在回调函数的最后,如果请求完成,立即调用回调函数
        if ( responses ) {
            complete( status, statusText, responses, responseHeaders );
        }
    };

    // if we're in sync mode or it's in cache
    // and has been retrieved directly (IE6 & IE7)
    // we need to manually fire the callback
    // 同步模式下:同步导致阻塞一致到服务器响应完成,所以这里可以立即调用callback
    if ( !s.async || xhr.readyState === 4 ) {
        callback();
    } else {
        handle = ++xhrId; // 请求计数
        // 如果时页面退出导致本次请求取消,修正在IE下不断开连接的bug
        if ( xhrOnUnloadAbort ) {
            // Create the active xhrs callbacks list if needed
            // and attach the unload handler
```

```
                if ( !xhrCallbacks ) {
                    xhrCallbacks = {};
                    jQuery( window ).unload( xhrOnUnloadAbort ); // 手动触发页面销毁事件
                }
                // Add to list of active xhrs callbacks
                // 将回调函数存储在全局变量中,以便在响应完成或页面退出时能注销回调函数
                xhrCallbacks[ handle ] = callback;
            }
            xhr.onreadystatechange = callback; // 绑定句柄,这里和传统的ajax写法没什么区别
        }
    },

    abort: function() {
        if ( callback ) {
            callback(0,1); // 1表示调用callback时,isAbort为true,在callback执行过程中能区分出是响应完成还是取消导致的调用
        }
    }
  };
 }
});
}
```

15.4.2 前置过滤器和请求分发器的执行过程

prefilters中的前置过滤器在请求发送之前、设置请求参数的过程中被调用，调用prefilters的是函数inspectPrefiltersOrTransports；巧妙的时，transports中的请求分发器在大部分参数设置完成后，也通过函数inspectPrefiltersOrTransports取到与请求类型匹配的请求分发器：

```
// some code...

// Apply prefilters
// 应用前置过滤器，参数说明：
inspectPrefiltersOrTransports( prefilters, s, options, jqXHR );

// If request was aborted inside a prefiler, stop there
// 如果请求已经结束，直接返回
if ( state === 2 ) {
    return false;
}

// some code...
// 注意：从这里开始要发送了

// Get transport
// 请求分发器
transport = inspectPrefiltersOrTransports( transports, s, options, jqXHR );
```

```
// some code...
```

函数inspectPrefiltersOrTransports从prefilters或transports中取到与数据类型匹配的函数数组,然后遍历执行，看看它的实现：

```
// Base inspection function for prefilters and transports
// 执行前置过滤器或获取请求分发器
function inspectPrefiltersOrTransports( structure, options, originalOptions, jqXHR,
    dataType /* internal */, inspected /* internal */ ) {

  dataType = dataType || options.dataTypes[ 0 ];
  inspected = inspected || {};

  inspected[ dataType ] = true;

  var list = structure[ dataType ],
    i = 0,
    length = list ? list.length : 0,
    executeOnly = ( structure === prefilters ),
    selection;

  for(; i < length && ( executeOnly || !selection ); i++ ) {
    selection = list[ i ]( options, originalOptions, jqXHR ); // 遍历执行
    // If we got redirected to another dataType
```

```
        // we try there if executing only and not done already
        if ( typeof selection === "string" ) {
            if ( !executeOnly || inspected[ selection ] ) {
                selection = undefined;
            } else {
                options.dataTypes.unshift( selection );
                selection = inspectPrefiltersOrTransports(
                        structure, options, originalOptions, jqXHR, selection, inspected );
            }
        }
    }
    // If we're only executing or nothing was selected
    // we try the catchall dataType if not done already
    if ( ( executeOnly || !selection ) && !inspected[ "*" ] ) {
        selection = inspectPrefiltersOrTransports(
                structure, options, originalOptions, jqXHR, "*", inspected );
    }
    // unnecessary when only executing (prefilters)
    // but it'll be ignored by the caller in that case
    return selection;
}
```

15.4.3 总结

通过前面的源码解析，可以将前置过滤器和请求分发器总结如下：

前置过滤器 jQuery.ajaxPrefilter，prefilters

属性	值	功能
*	undefined	不做任何处理，事实上也没有*属性
json	[function]	被当作*处理
jsonp	[function]	修正url或data，增加回调函数名 在window上注册回调函数 注册script>json数据转换器 （被当作script处理）
script	[function]	设置设置以下参数： 是否缓存 cache、（如果跨域）请求类型 、（如果跨域）是否触发AJAX全局事件

请求分发器 jQuery.ajaxTransport，transports

属性	值	功能
*	[function]	返回xhr分发器，分发器带有send、abort方法 send方法依次调用XMLHTTPRequest的open、send方法，向服务端发送请求，并绑定onreadystatechange事件句柄
script	[function]	返回script分发器，分发器带有send、abort方法 send方法通过在header中创建script标签异步载入js，并在script元素上绑定onload、script.onreadystatechange事件句柄

附件下载:

- _原创__jQuery源码分析-15AJAX-前置过滤器和请求分发器.pdf (411.5 KB)
- dl.iteye.com/topics/download/09891435-cddd-3e12-a871-c225ae9ff268

1.11 [原创] jQuery源码分析-15AJAX-类型转换器

发表时间: 2011-09-29 关键字: javascript, jquery, 源码分析, ajax, 类型转换器

作者：nuysoft/高云　QQ：47214707　EMail：nuysoft@gmail.com

边读边写，不对的地方请告诉我，多多交流共同进步，PDF下载在最后

jQuery源码分析系列的目录请查看 http://nuysoft.iteye.com/blog/1177451，想系统的好好写写，目前还是从我感兴趣的部分开始，如果大家有对哪个模块感兴趣的，建议优先分析的，可以告诉我，一起学习。

15.5 AJAX中的类型转换器

前置过滤器、 请求分发器、类型转换器是读懂jQuery AJAX实现的关键，可能最难读的又是类型转换器。除此之外的源码虽然同样的让人纠结，但相较而言并不算复杂。

类型转换器将服务端响应的responseText或responseXML，转换为请求时指定的数据类型dataType，如果没有指定类型就依据响应头Content-Type自动猜测一个。在分析转换过程之前，很有必要先看看类型转换器的初始化过程，看看支持哪些类型之间的转换。

15.5.1 类型转换器的初始化

类型转换器ajaxConvert在服务端响应成功后，对定义在jQuery. ajaxSettings中的converters进行遍历，找到与数据类型相匹配的转换函数，并执行。我们先看看converters的初始化过程，对类型类型转换器的功能有个初步的认识。jQuery. ajaxSettings定义了所有AJAX请求的默认参数，我们暂时先忽略其他属性、方法的定义和实现：

```
jQuery.extend({

    // some code ...
```

```
// ajax请求的默认参数
ajaxSettings: {
  // some code ...

  // List of data converters
  // 1) key format is "source_type destination_type" (a single space in-between)
  // 2) the catchall symbol "*" can be used for source_type
  // 类型转换映射,key格式为单个空格分割的字符串：源格式 目标格式
  converters: {

    // Convert anything to text、
    // 任意内容转换为字符串
    // window.String 将会在min文件中被压缩为 a.String
    "* text": window.String,

    // Text to html (true = no transformation)
    // 文本转换为HTML（true表示不需要转换，直接返回）
    "text html": true,

    // Evaluate text as a json expression
    // 文本转换为JSON
    "text json": jQuery.parseJSON,
```

```
            // Parse text as xml
            // 文本转换为XML
            "text xml": jQuery.parseXML
        }
    }
    // some code ...
});
```

然后在jQuery初始化过程中，对jQuery. ajaxSettings.converters做了扩展，增加了text>script的转换：

```
// Install script dataType
// 初始化script对应的数据类型
// MARK：AJAX模块初始化
jQuery.ajaxSetup({
    accepts: {
        script: "text/javascript, application/javascript, application/ecmascript, application/x-ecmascript"
    },
    contents: {
        script: /javascript|ecmascript/
    },
    // 初始化类型转换器,这个为什么不写在jQuery.ajaxSettings中而要用扩展的方式添加呢?
    // 这个转换器是用来出特殊处理JSONP请求的，显然,jQuery的作者John Resig,时时刻刻都认为JSONP和跨域要特殊处理!
    converters: {
```

```
        "text script": function( text ) {
            jQuery.globalEval( text );
            return text;
        }
    }
});
```

初始化过程到这里就结束了，很简单，就是填充jQuery. ajaxSettings.converters。

当一个AJAX请求完成后，会调用闭包函数done，在done中判断本次请求是否成功，如果成功就调用ajaxConvert对响应的数据进行类型转换（闭包函数done在讲到jQuery.ajax()时一并分析）：

```
// 服务器响应完毕之后的回调函数，done将复杂的善后事宜封装了起来，执行的动作包括：
// 清除本次请求用到的变量、解析状态码&状态描述、执行异步回调函数队列、执行complete队列、触发全局Ajax事件
// status: -1 没有找到请求分发器
function done( status, statusText, responses, headers ) {
    // 省略代码...
    // If successful, handle type chaining
    // 如果成功的话，处理类型
    if ( status >= 200 && status < 300 || status === 304 ) {
        // 如果没有修改，修改状态数据，设置成功
        if ( status === 304 ) { 省略代码... }
        else {

            try {
```

```
            // 获取相应的数据
            // 在ajaxConvert这个函数中，将Server返回的的数据进行相应的转换（js、json等等）
            success = ajaxConvert( s, response ); // 注意:这里的succes变为转换后的数据对象!
            statusText = "success";
            isSuccess = true;
        } catch(e) {
            // We have a parsererror
            // 数据类型转换器解析时出现错误
            statusText = "parsererror";
            error = e;
        }
    }
// 非200~300，非304
} else {
    // We extract error from statusText
    // then normalize statusText and status for non-aborts
    // 其他的异常状态，格式化statusText、status，不采用HTTP标准状态码和状态描述
    error = statusText;
    if( !statusText || status ) {
        statusText = "error";
        if ( status < 0 ) {
            status = 0;
        }
```

```
        }
    }
    // 省略代码...
}
```

15.5.2 类型转换器的执行过程

类型转换器ajaxConvert根据请求时设置的数据类型，从jQuery. ajaxSettings.converters寻找对应的转换函数，寻找的过程非常绕。假设有类型A数据和类型B数据，A要转换为B（A > B），首先在converters中查找能A > B 对应的转换函数，如果没有找到，则采用曲线救国的路线，寻找类型C，使得类型A数据可以转换为类型C数据，类型C数据再转换为类型B数据，最终实现 A > B。类型转换器的原理并不复杂，复杂的是它的实现：

```
// Chain conversions given the request and the original response
// 转换器,内部函数,之所以不添加到jQuery中,可能是因为这个函数不需要客户端显示调用吧
function ajaxConvert( s, response ) {

    // Apply the dataFilter if provided
    // dataFilter也是一个过滤器,在调用时的参数options中设置,在类型类型转换器执行之前调用
    if ( s.dataFilter ) {
        response = s.dataFilter( response, s.dataType );
    }

    var dataTypes = s.dataTypes, // 取出来,减少作用域查找,缩短后边的拼写
        converters = {},
        i,
        key,
```

```
    length = dataTypes.length,
    tmp,
    // Current and previous dataTypes
    current = dataTypes[ 0 ], // 取出第一个作为起始转换类型
    prev, // 每次记录前一个类型,以便数组中相邻两个类型能形成链条
    // Conversion expression
    conversion, // 类型转换表达式 被转换类型>目标了类型
    // Conversion function
    conv, // 从jQuery.ajaxSetting.converts中取到特定类型之间转换的函数
    // Conversion functions (transitive conversion)
    conv1, // 两个临时表达式
    conv2;

// For each dataType in the chain
// 从第2个元素开始顺序向后遍历,挨着的两个元素组成一个转换表达式,形成一个转换链条
for( i = 1; i < length; i++ ) {

    // Create converters map
    // with lowercased keys
    // 将s.converters复制到converters,为什么要遍历转换呢?直接拿过来用不可以么?
    if ( i === 1 ) {
        for( key in s.converters ) {
            if( typeof key === "string" ) {
```

```
                converters[ key.toLowerCase() ] = s.converters[ key ];
            }
        }
    }

    // Get the dataTypes
    prev = current; // 取出前一个类型,每次遍历都会把上一次循环时的类型记录下来
    current = dataTypes[ i ]; // 取出当前的类型

    // If current is auto dataType, update it to prev
    if( current === "*" ) { // 如果碰到了*号,即一个任意类型,而转换为任意类型*没有意义
        current = prev; // 所以回到前一个,跳过任意类型,继续遍历s.dataTypes
    // If no auto and dataTypes are actually different
    // 这里开始才是函数ajaxConvert的重点
    // 前一个不是任意类型*,并且找到了一个不同的类型(注意这里隐藏的逻辑:如果第1个元素是*,跳过,再加上中间遇到的*都被跳过了,所以结论就是s.dataTypes中的*都会被忽略!)
    } else if ( prev !== "*" && prev !== current ) {

        // Get the converter 找到类型转换表达式对应的转化函数
        conversion = prev + " " + current; // 合体,组成类型转换表达式
        conv = converters[ conversion ] || converters[ "* " + current ]; // 如果没有对应的,就默认被转换类型为*
```

```
        // If there is no direct converter, search transitively
        // 如果没有找到转变表达式,则向后查找
        // 因为:jsonp是有浏览器执行的呢,还是要调用globalEval呢?
        if ( !conv ) { // 如果没有对应的转换函数,则寻找中间路线
          conv2 = undefined; //
          for( conv1 in converters ) { // 其实是遍历s.converts
            tmp = conv1.split( " " ); // 将s.convsert中的类型转换表达式拆分,tmp[0] 源类型 tmp[1] 目标类型
            // 如果tmp[0]与前一个类型相等,或者tmp[0]是*(*完全是死马当作活马医,没办法的办法)
            if ( tmp[ 0 ] === prev || tmp[ 0 ] === "*" ) { // 这里的*号与conv=这条语句对应
              // 形成新的类型转换表达式, 看到这里,我有个设想,简单点说就是:
              // A>B行不通,如果A>C行得通,C>B也行得通,那么A>B也行的通:A > C > B
              // 将这个过程与代码结合起来,转函数用fun(?>?)表示:
              // A == tmp[0] == prev
              // C == tmp[1]
              // B == current
              // conv1 == A>C
              conv2 = converters[ tmp[1] + " " + current ]; // 看看C>B转换函数有木有,conv==fun(C>B)
              if ( conv2 ) { // 如果fun(C>B)有,Great!看来有戏,因为发现了新的类型转换表达式A>C>B
                conv1 = converters[ conv1 ]; // conv1是A>C,将A>C转换函数取出来,conv1由A>C变成fun(A>C)
                if ( conv1 === true ) { // true是什么东西呢?参看jQuery.ajaxSettings知道 "text html": true,意思是不需要转换,直接那来用
                  conv = conv2; // conv2==fun(C>B),赋给conv,conv由func(A>B)变成fun(C>B)
```

```
                    // 详细分析一下:
                    // 这里A==text,C==html,B是未知,发现A>B行不通,A>C和C>B都行得通,
                    // 但是因为A>C即text>html不需要额外的转换可以直接使用,可以理解为A==C,所以可以忽略A>C,将A>C>B链条简化为C>B
                    // 结果就变成这样: A>C>B链条中的A>C被省略,表达式简化为C>B
                // 如果conv1不是text>html,即A!=C,那么就麻烦了,但是,又发现conv2即fun(C>B)是text>html,C==B,那么A>C>B链条简化为A>C
                } else if ( conv2 === true ) { // conv2==func(C>B)
                    conv = conv1; // A>C>B链条简化为A>C
                }
                /**
                 * 将上边与代码紧密结合的注释再精炼:
                 * 目标是A>B但是行不通,但是A>C可以,C>B可以,表达式变成A>C>B
                 * 如果A=C,表达式变成C>B,上边的conv2
                 * 如果C=B,表达式变成A>C,上边的conv1
                 */
                /**
                 * 但是要注意,到这里还没完,如果既不是A==C,也不是C==B,表达式A>C>B就不能简化了!
                 * 怎么办?虽然到这里conv依然是undefined,但是知道了一条A通往B的路,剩下的工作在函数ajaxConvert的最后完成!
                 */
                break; // 找到一条路就赶紧跑,别再转悠了

            }
```

```
            }
        }
    }
    // If we found no converter, dispatch an error
    // 如果A>B行不通,A>?>B也行不通,?表示中间路线,看来此路是真心不通啊,抛出异常
    if ( !( conv || conv2 ) ) { // 如果conv,conv2都为空
        jQuery.error( "No conversion from " + conversion.replace(" "," to ") );
    }
    // If found converter is not an equivalence
    // 如果找到的conv是一个函数或者是undefined
    if ( conv !== true ) {
        // Convert with 1 or 2 converters accordingly
        // 分析下边的代码之前,我们先描述一下这行代码的运行环境,看看这些变量分别代表什么含义:
        // 1. conv可能是函数表示A>B可以
        // 2. conv可能是undefined表示A>B不可以,但是A>C>B可以
        // 3. conv1是fun(A>C),表示A>C可以
        // 4. conv2是fun(C>B),表示C>B可以

        // 那么这行代码的含义就是:
        // 如果conv是函数,执行conv( response )
        // 如果conv是undefined,那么先执行conv1(response),即A>C,再执行conv2( C ),即C>B,最终完成A>C>B的转换
        response = conv ? conv( response ) : conv2( conv1(response) );
```

```
            }
        }
    }
    return response;
}
```

ajaxConvert的源码分析让我破费一番脑筋，因为它的很多逻辑没有显示的体现在代码上，是隐式的。我不敢对这样的编程方式妄下定论，也许这样的代码不易维护，也许仅仅是我的水平还不够。

15.5.3 总结

通过前面的源码解析，可以将类型转换器总结如下：

属性	值	功能
* text	window.String	任意内容转换为字符串
text html	true	文本转换为HTML（true表示不需要转换，直接返回）
text json	jQuery.parseJSON	文本转换为JSON

text script	function(text) {
	jQuery.globalEval(text);
	return text;
	}
		用eval执行text

text xml	jQuery.parseXML	文本转换为XML

如果在上表示中没有找到对应的转换函数（类型A > 类型B），就再次遍历上表，寻找新的类型C，使得可以 A > C > B。

后记：

到此，最关键的前置过滤器、请求分发器、类型转换器已经分析完毕，但是AJAX实现的复杂度还是远远超过了我最初的认识，还有很多地方值得深入学习和分析，比如：jQuery.ajax的实现、数据的解析、异步队列在AJAX中的应用、多个内部回调函数调用链、jqXHR的状态码与HTTP状态码的关系、数据类型的修正、跨域、对缓存的修正、对HTTP状态码的修正，等等等等，每一部分都可以单独成文，因此对AJAX的分析还会继续。提高自己的同时，希望对读者有所启发和帮助。

附件下载:

- _原创__jQuery源码分析-15AJAX-类型转换器.pdf (385.6 KB)
- dl.iteye.com/topics/download/e219a5dd-3620-3895-9b90-c32beb967d00

1.12 [原创] jQuery源码分析-16动画分析和扩展 Effects

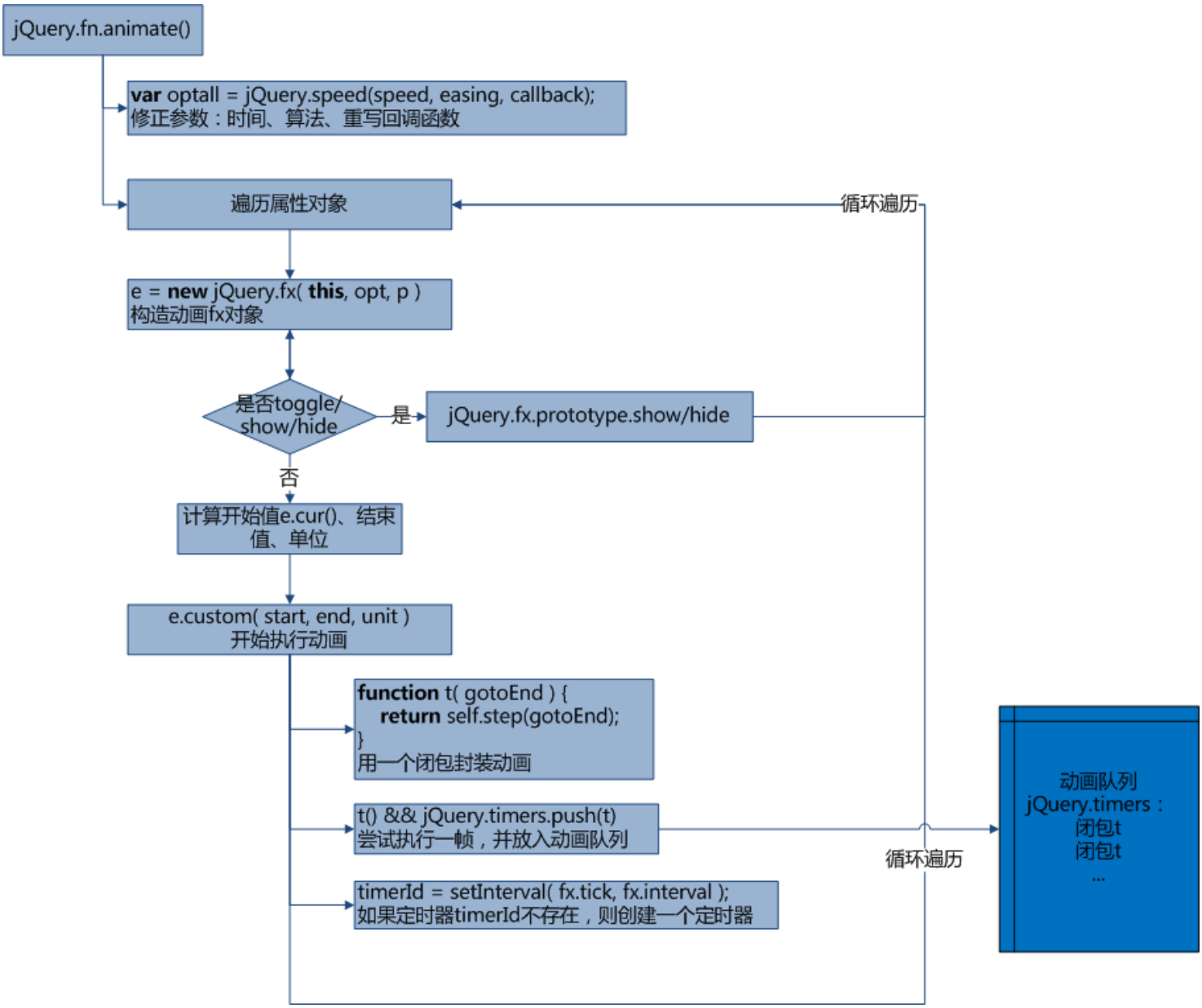
发表时间: 2011-09-14 关键字: jquery, animate, easing, step

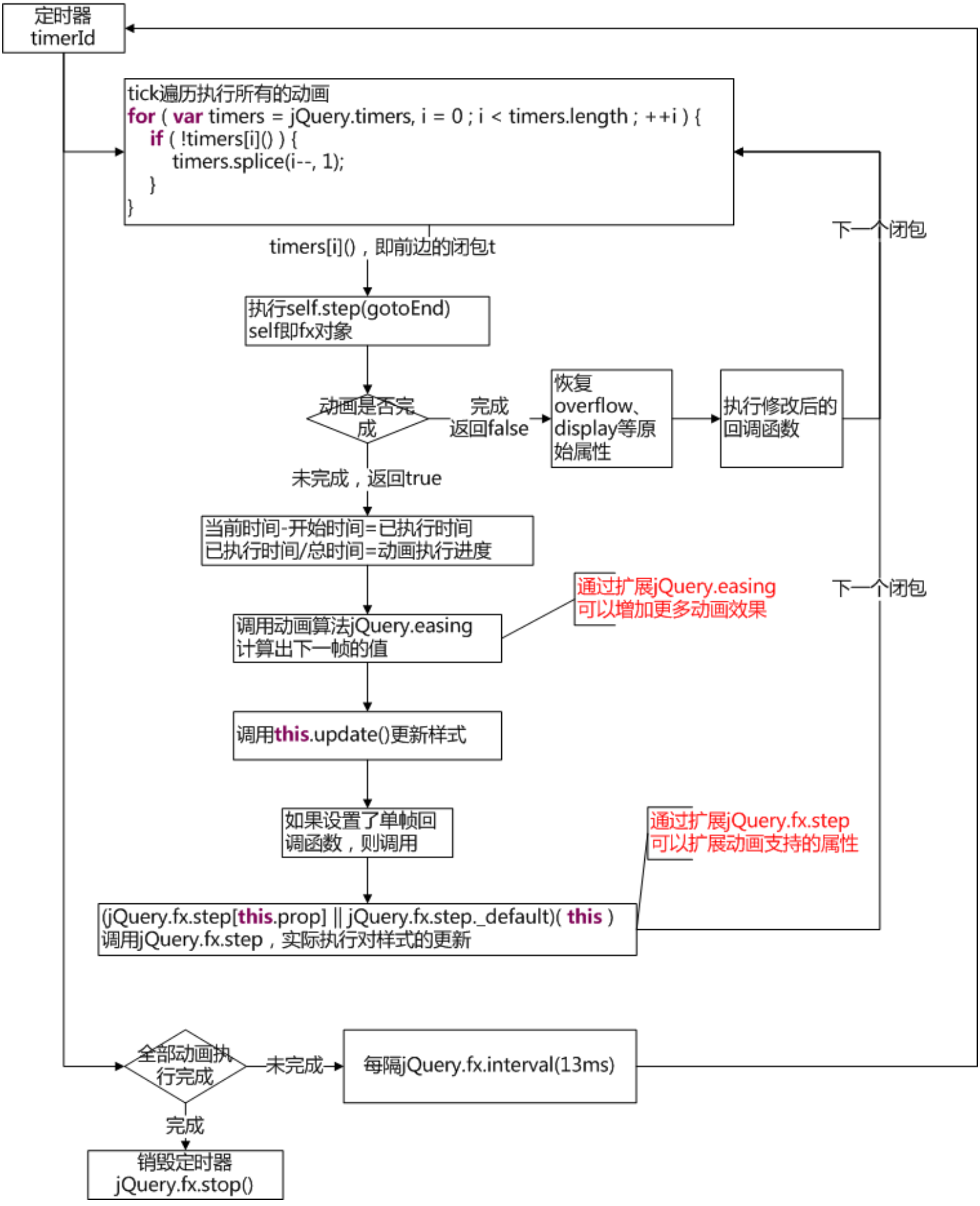
16. 动画

16.1 源码解析

jquery1.43源码分析之动画部分 http://www.iteye.com/topic/786984

上边这篇文章分析的很透彻，由浅入深，我就不再重复了，下面附两张jQuery 1.6.1的程序执行流程图：


jQuery.fn.animate()
var optall = jQuery.speed(speed, easing, callback);
修正参数：时间、算法、重写回调函数
遍历属性对象
循环遍历
e = new jQuery.fx(this, opt, p)
构造动画fx对象
是否toggle/show/hide
是
jQuery.fx.prototype.show/hide
否
计算开始值e.cur()、结束值、单位
e.custom(start, end, unit)
开始执行动画
function t(gotoEnd) {
return self.step(gotoEnd);
}
用一个闭包封装动画
t() && jQuery.timers.push(t)
尝试执行一帧，并放入动画队列
timerId = setInterval(fx.tick, fx.interval);
如果定时器timerId不存在，则创建一个定时器
循环遍历
动画队列
jQuery.timers：
闭包t
闭包t
...


定时器
timerId
tick遍历执行所有的动画
for (var timers = jQuery.timers, i = 0 ; i < timers.length ; ++i) {
if (!timers[i]()) {
timers.splice(i--, 1);
}
}
下一个闭包
timers[i]()，即前边的闭包t
执行self.step(gotoEnd)
self即fx对象
动画是否完成
完成
返回false
恢复overflow、display等原始属性
执行修改后的回调函数
未完成，返回true
当前时间-开始时间=已执行时间
已执行时间/总时间=动画执行进度
通过扩展jQuery.easing
可以增加更多动画效果
下一个闭包
调用动画算法jQuery.easing
计算出下一帧的值
调用this.update()更新样式
如果设置了单帧回调函数，则调用
通过扩展jQuery.fx.step
可以扩展动画支持的属性
(jQuery.fx.step[this.prop] || jQuery.fx.step._default)(this)
调用jQuery.fx.step，实际执行对样式的更新
全部动画执行完成
未完成
每隔jQuery.fx.interval(13ms)
完成
销毁定时器
jQuery.fx.stop()

16.2 动画支持的属性

jQuery仅支持数值型的属性和特殊标记show/hide/toggle，非数值型的属性需要插件支持。

16.3 动画算法 jQuery.easing

jQuery自带了线性动画linear、余弦动画swing，下边是源码分析：

```
/**
* 如果要扩展动画效果，则扩展这个属性jQuery.easing
*/
easing: {
    /**
     * 线性
     * p 完成时间百分比
     * n 已执行时间
     * firstNum 0 起始偏移百分比
     * diff 1 结束偏移百分比
     * 通常情况下，等于直接返回p
     * 动画效果：匀速运动
     */
    linear: function( p, n, firstNum, diff ) {
        return firstNum + diff * p;
```

```
    },
    /**
     * http://baike.baidu.com/view/303443.htm
     * 余弦
     * -Math.cos(p*Math.PI) -1~1
     * ( ( -Math.cos(p*Math.PI)/2 ) + 0.5 ) 0~1
     * 根据余弦图，最终的动画效果是：慢>快>慢
     */
    swing: function( p, n, firstNum, diff ) {
      return ( ( -Math.cos(p*Math.PI)/2 ) + 0.5 ) * diff + firstNum;
    }
}
```

16.4 扩展

从上面的图（定时器timerId）可以看到，动画的执行进度用执行时间衡量，再由jQuery.easing转换为带特效的进度，最后实际执行样式更新的是jQuery.fx.step，jQuery.easing并不涉及具体的样式值，jQuery的这种低耦合的架构使得无论是扩展属性还是扩展动画，都变得非常方便。

1. 扩展支持动画的属性 jQuery.fx.step，jQuery自带opacity _default；

? 颜色 color

颜色的表示方法有rgb(num, num, num)、#RRGGBB、#RGB等，例如rgb(12, 23, 45)、#123456、#123，要产生颜色动画的关键在于，将颜色值转换为可变化的数值型，作为起始值和结束值。

? rgb(num, num, num) 用正则解析其中3个num，放进一个数组中；这样开始值和结束值都变为包含了三个整型元素的数组，step调用时分别计算3个整型元素的当前值，再拼装成颜色值，更新样值就可以实现颜色动画。

? # RRGGBB 是24位色，由6个十六进制数组成，可以分为三部分，第一部分代表红色，第二部分绿色，最后是蓝色，将三部分分别解析成整型，其余部分同rgb(num, num, num)

? #RGB 是# RRGGBB的简写，解析时将每一位重复一次，其余部分同# RRGGBB

? 旋转 rotate

旋转的效果很酷，配合精美的图片可以做出很玄的网页。旋转的实现关键是兼容IE和非IE，有两种实现方式：使用浏览器自定义接口和canvas标记，因为canvas标记在IE9以下的版本中不支持，需要额外的插件支持，单独再讲，这里先介绍下浏览器自定义接口实现：

? IE: matirx滤镜

? webkit: elemstyle.webkitTransform = 'rotate(45deg)'

? Opera: Otransform = 'rotate(45deg)'

? firefox: MozTransform = 'rotate(45deg)'

2. 扩展动画算法 jQuery.easing，jQuery自带linear swing

从上边的jQuery.easing 源码分析可以看到，最关键的参数是p，表示已执行时间的百分比，jQueryUI扩展的动画算法以p为输入，经过不同的公式，输出一个具有动画效果的新的进度，这个进度乘以结束值减去开始值的差值，就是当前值。

jQueryUI扩展的动画算法 http://jqueryui.com/demos/effect/easing.html

1.13 [原创] jQuery源码分析-17尺寸和大小 Dimensions & Offset

发表时间: 2011-09-25 关键字: web, javascript, jquery, 坐标Offset, 尺寸Dimensions

边读边写，不正确的地方，还请各位告诉我，多多交流共同学习，PDF下载地址在最后。

17. 坐标和尺寸 Offset & Dimensions

初学者经常会迷惑于jQuery的提供的获取/设置坐标和尺寸接口的差异，不知道在什么情况下该使用什么接口，现将接口和差异整理如下：

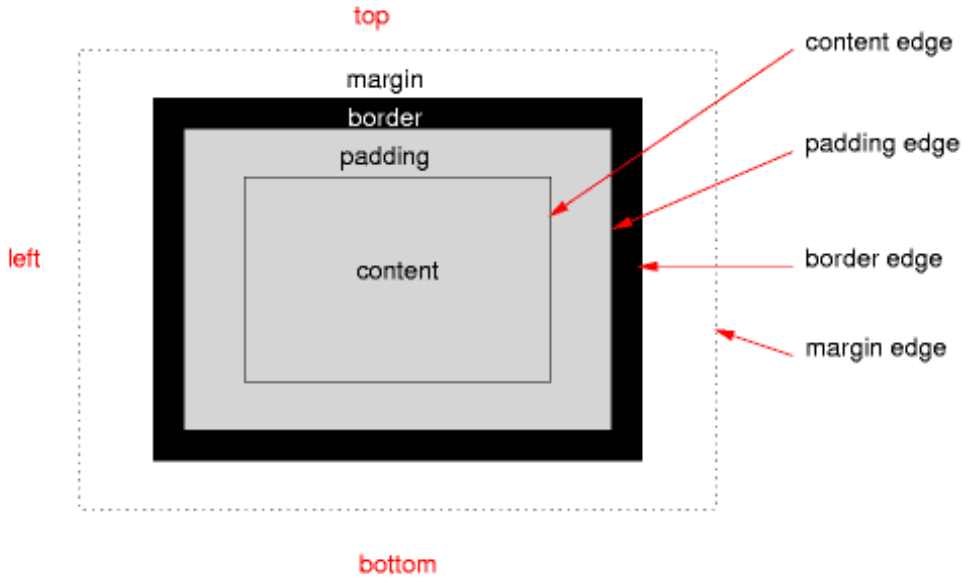
| 坐标 Offset

接口	公式	说明
.offset()	相对于文档document的坐标	返回或设置匹配元素相对于文档的偏移（位置），返回的对象包含两个整形属性：top和left，以像素计。此方法只对可见元素有效。 设置时可以接受带有left和top属性的对象，或函数，使用函数来设置所有匹配元素的偏移坐标。 隐藏元素、window、document无效
.offsetParent()	取到最近的父节点	不是坐标接口
.position()	相对于父元素parent的坐标	返回匹配元素相对于父元素的位置（偏移）。该方法返回的对象包含两个整型属性：top 和 left，以像素计。此方法只对可见元素有效。 .position()把元素当绝对定位来处理，获取的是该元素相当于最近的一个拥有绝对absolute或者相对定位relative的父元素的偏移位置。使用.position()方法时如果其所有的父元素都为默认定位（static）方式，则其处理方式和offset()一样，是当前窗口的相对偏移 只能获取，没有设置接口

		对隐藏元素、window、document无效
.scrollLeft() .scrollTop()	滚动条的水平和垂直位置	获取或设置滚动条的水平和垂直位置。 对可见或隐藏元素都有效，对window、document有效 对非容器型元素无效

l 尺寸 Demensions（结合后边的图一起理解）

接口	公式	说明
.width(), .height()	content	获取或设置匹配元素的高度、宽度，如果不为该方法设置参数，则返回第一个匹配元素的高度、宽度（单位是像素，整型值，不带单位） 对可见和隐藏元素都有效
.innerWidth() .innerHeight()	content+padding	只能获取，没有设置接口（单位是像素，整型值，不带单位） 对window、document无效，用.width()/.height()代替
.outerWidth() .outerHeight()	content+padding+border （+可选的margin）	只能获取，没有设置接口（单位是像素，整型值，不带单位） 对window、document无效，用.width()/.heigth()代替



附件下载:

- _原创__jQuery源码分析-17坐标和尺寸-Offset_Dimensions.pdf (171.6 KB)
- dl.iteye.com/topics/download/3a15b610-d8d0-3a07-b347-a2ee280b1b80

1.14 [原创] jQuery源码分析-如何做jQuery源码分析

发表时间: 2011-09-27 关键字: javascript, jquery, 源码分析方法, 幸福, 勇气

近期在ITEYE陆续写了几篇jQuery源码分析，乐在其中的同时愈发佩服jQuery的神乎其技，为我打开了一扇软件以用为本的窗户，以至于写出来Java代码也有了jQuery的味道。

jQuery的源码有些晦涩难懂，本文分享一些我看源码的方法，每一个模块我尽量按照这样的顺序去学习：

1. 读官方文档，官方有非常详细的文档说明

2. 试验官方的示例，需要的话搭建自己的服务器

3. 读源码，加注释，把自己思考的过程和结果记录下来

4. 大量阅读相关的网文和书籍，比如相同主题的分析文档，网上常问的问题等

5. 写一篇应用教程（可看的应用教程已经很多了，有时间也会写写，但不多）

6. 写一篇源码分析，记录自己的心得，加深理解

希望对大家能有所启发。

最后附一个DEMO，不复杂，但是很有爱：

http://nuysoft.com/love.html

长大以后，支撑我们一路走过来的，不只是兴趣和技能，更多的是勇气，每一个单身的人心里都住着一个不可能的人，工程师也要勇敢些，去努力寻找自己的幸福，祝福你，也祝福我，国庆快乐。

2011.9.26