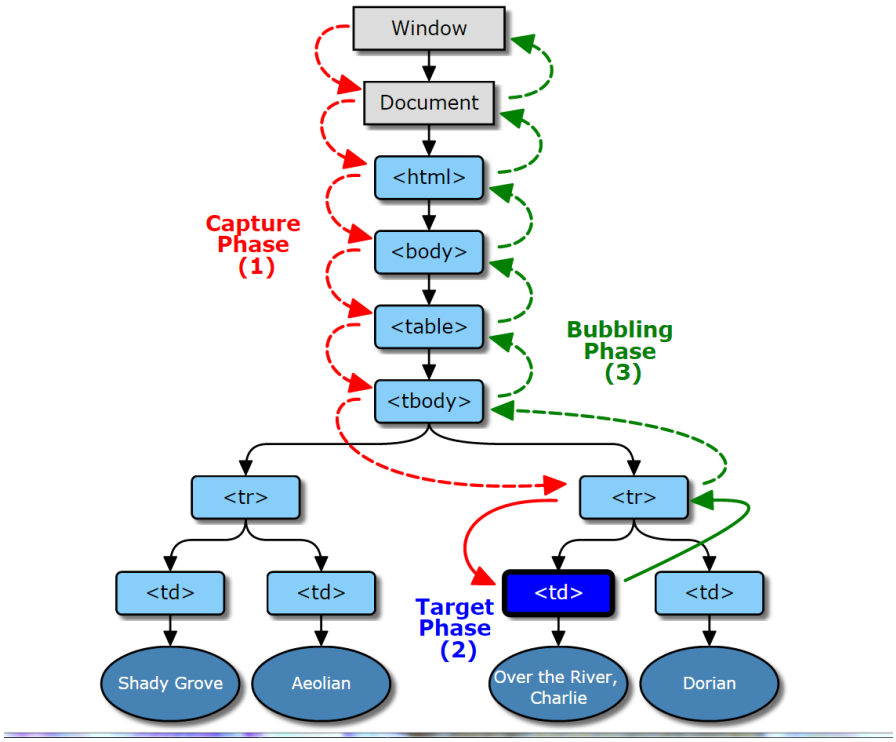
javascript事件

事件触发过程



捕获阶段（Capture Phase）

当我们在 DOM 树的某个节点发生了一些操作（例如单击、鼠标移动上去），就会有一个事件发射过去。这个事件从 Window 发出，不断经过下级节点直到目标节点。在到达目标节点之前的过程，就是捕获阶段（Capture Phase）。

所有经过的节点，都会触发这个事件。捕获阶段的任务就是建立这个事件传递路线，以便后面冒泡阶段顺着这条路线返回 Window。

标准的事件监听函数如下：

```
element.addEventListener(<event-name>, <callback>, <use-capture>);
```

表示在 element 这个对象上面添加一个事件监听器，当监听到有 <event-name> 事件发生的时候，调用 这个回调函数。至于 <use-capture> 这个参数，表示该事件监听是在“捕获”阶段中监听（设置为 true）还是在“冒泡”阶段中监听（设置为 false）。 例：

```
var btn = document.getElementsByTagName('button');
btn[0].addEventListener('click', function() {
    alert('你点击了这个按钮');
}, false);
```

当我们为某个元素绑定了一个事件，每次触发这个事件的时候，都会执行事件绑定的回调函数。如果我们想解除绑定，需要使用 removeEventListener 方法：

```
element.removeEventListener(<event-name>, <callback>, <use-capture>);
```

需要注意的是，绑定事件时的回调函数不能是匿名函数，必须是一个声明的函数，因为解除事件绑定时需要传递这个回调函数的引用，才可以断开绑定。例如：

```
var fun = function() {
    // function logic
};
element.addEventListener('click', fun, false);
element.removeEventListener('click', fun, false);
```

监听某个在捕获阶段触发的事件，需要在事件监听函数传递第三个参数 true。

```
element.addEventListener(<event-name>, <callback>, true);
```

但一般使用时我们往往传递 false，会在后面说明原因。

目标阶段（Target Phase）

当事件跑啊跑，跑到了事件触发目标节点那里，最终在目标节点上触发这个事件，就是目标阶段。

需要注意的是，事件触发的目标总是最底层的节点。比如你点击一段文字，你以为你的事件目标节点在 div 上，但实际上触发在 <p>、 等子节点上。

冒泡阶段（Bubbling Phase）

当事件达到目标节点之后，就会沿着原路返回，由于这个过程类似水泡从底部浮到顶部，所以称作冒泡阶段。

在实际使用中，你并不需要把事件监听函数准确绑定到最底层的节点也可以正常工作。比如你想为这个 绑定单击时的回调函数，你无须为这个 下面的所有子节点全部绑定单击事件，只需要为 这一个节点绑定即可。因为发生在它子节点的单击事件，都会冒泡上去，发生在 上面。

为什么不用第三个参数 true

所有介绍事件的文章都会说，在使用 addEventListener 函数来监听事件时，第三个参数设置为 false，这样监听事件时只会监听冒泡阶段发生的事件。

这是因为 IE 浏览器不支持在捕获阶段监听事件，为了统一而设置的，毕竟 IE 浏览器的份额是不可忽略的。

使用事件代理（Event Delegate）提升性能

因为事件有冒泡机制，所有子节点的事件都会顺着父级节点跑回去，所以我们可以通过监听父级节点来实现监听子节点的功能，这就是事件代理。

使用事件代理主要有两个优势：

1. 减少事件绑定，提升性能。之前你需要绑定一堆子节点，而现在你只需要绑定一个父节点即可。减少了绑定事件监听函数的数量。
2. 动态变化的 DOM 结构，仍然可以监听。当一个 DOM 动态创建之后，不会带有任何事件监听，除非你重新执行事件监听函数，而使用事件监听无须担忧这个问题。

例：

```
//html代码
<ul class="ul1">
    <li><a href="#">监听我</a></li>
    <li><a href="#">监听我</a></li>
</ul>
<ul class="ul2">
    <li><a href="#">监听我</a></li>
    <li><a href="#">监听我</a></li>
</ul>
<a href="#" class="add-more-items">添加更多节点</a>

//script代码
$('.ul1 a').on('click', function(){
    alert('正在监听');
});
$('.ul2').on('click', 'a', function(){
    alert('正在监听');
});
$('.add-more-items').on('click', function(){
    var item = $('<li><a href="#">我是动态新增节点</a></li>');
    $('.ul1,.ul2').append(item);
});
```

上面例子中，为了简便，我使用 jQuery 来实现普通事件绑定和事件代理。我的目标是监听所有 a 链接的单击事件，.ul1 是常规的事件绑定方法，jQuery 会循环每一个 .ul > a 结构并绑定事件监听函数。.ul2 则是事件监听的方法，jQuery 只为 .ul2 结构绑定事件监听函数，因为 .ul2 下面可能会有很多无关节点也会触发click 事件，所以我在 on 函数里传递了第二个参数，表示只监听 a 子节点的事件。

它们都可以正常工作，但是当我动态创建新 DOM 结构的时候，第一个 ul 问题就出现了，新创建结构虽然还是.ul1 > a，但是没有绑定事件，所以无法执行回调函数。而第二个 ul 工作的很好，因为点击新创建的 DOM ，它的事件会冒泡到父级节点进行处理。

如果使用原生的方式实现事件代理，需要注意过滤非目标节点，可以通过 id、class 或者 tagname 等等，例如：

```
element.addEventListener('click', function(event) {
    // 判断是否是 a 节点
    if ( event.target.tagName == 'A' ) {
        // a 的一些交互操作
    }
}, false);
```

停止事件冒泡（stopPropagation）

停止事件冒泡需要使用事件对象的 stopPropagation 方法，具体代码如下：

```
element.addEventListener('click', function(event) {
    event.stopPropagation();
}, false);
```

在事件监听的回调函数里，会传递一个参数，这就是 Event 对象，在这个对象上调用 stopPropagation 方法即可停止事件冒泡。举个停止事件冒泡的应用实例：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8 />
        <title>JS Bin</title>
    </head>
    <body>
        <div class='overlay'>Click outside to close.</div>

        <script>
            var overlay = document.querySelector('.overlay');
            overlay.addEventListener('click', function(event) {
                event.stopPropagation();
            });
            // Remove the overlay when a 'click'
            // is heard on the document (<html>) element
            document.addEventListener('click', function(event) {
                overlay.parentNode.removeChild(overlay);
            });
        </script>
    </body>
</html>
```

事件的 Event 对象

当一个事件被触发的时候，会创建一个事件对象（Event Object），这个对象里面包含了一些有用的属性或者方法。事件对象会作为第一个参数，传递给我们的回调函数。

下面介绍一下比较常用的几个Event对象属性和方法：

- type (string) 事件的名称，比如 “click”。
- target (node) 事件要触发的目标节点。
- bubbles (boolean) 表明该事件是否是在冒泡阶段触发的。
- preventDefault (function) 这个方法可以禁止一切默认的行为，例如点击 a 标签时，会打开一个新页面，如果为 a 标签监听事件click同时调用该方法，则不会打开新页面。
- stopPropagation (function) 停止冒泡。
- stopImmediatePropagation (function) 与 stopPropagation 类似，就是阻止触发其他监听函数。但是与 stopPropagation 不同的是，它更加 “强力”，阻止除了目标之外的事件触发，甚至阻止针对同一个目标节点的相同事件。
- cancelable (boolean) 这个属性表明该事件是否可以通过调用event.preventDefault 方法来禁用默认行为。
- eventPhase (number) 这个属性的数字表示当前事件触发在什么阶段。none：0；捕获：1；目标：2；冒泡：3。
- pageX 和 pageY (number) 这两个属性表示触发事件时，鼠标相对于页面的坐标。

- isTrusted (boolean) 表明该事件是浏览器触发（用户真实操作触发），还是 JavaScript 代码触发的。

IE 下绑定事件

在 IE 下面绑定一个事件监听，在 IE9- 无法使用标准的 addEventListener 函数，而是使用自家的attachEvent，具体用法：

```
element.attachEvent(<event-name>, <callback>);
```

其中 <event-name> 参数需要注意，它需要为事件名称添加 on 前缀，比如有个事件叫 click，标准事件监听函数监听 click，IE 这里需要监听 onclick。

另一个，它没有第三个参数，也就是说它只支持监听在冒泡阶段触发的事件，所以为了统一，在使用标准事件监听函数的时候，第三参数传递 false。

当然，这个方法在 IE9 已经被抛弃，在 IE11 已经被移除了，IE 也在慢慢变好。

IE 中往回调函数中传递的事件对象与标准也有一些差异，你需要使用 window.event 来获取事件对象。所以你通常会写出下面代码来获取事件对象：

```
event = event || window.event
```

此外还有一些事件属性有差别，比如比较常用的 event.target 属性，IE 中没有，而是使用 event.srcElement来代替。如果你的回调函数需要处理触发事件的节点，那么需要写：

```
node = event.srcElement || event.target;
```

事件回调函数的作用域问题

与事件绑定在一起的回调函数作用域会有问题，我们来看个例子：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8 />
        <title>Events in JavaScript: Removing event listeners</title>
    </head>
    <body>
        <button id="element">Click Me</button>

        <script>
            var element = document.getElementById('element');
            var user = {
                firstname: 'Bob',
                greeting: function(){
                    alert('My name is ' + this.firstname);
                }
            };
            // Attach user.greeting as a callback
            element.addEventListener('click', user.greeting);
        </script>
    </body>
</html>
```

回调函数调用的 user.greeting 函数作用域应该是在 user 下的，本期望输出 My name is Bob 结果却输出了My name is undefined。这是因为事件绑定函数时，该函数会以当前元素为作用域执行。

使用匿名函数

我们为回调函数包裹一层匿名函数。例：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8 />
        <title>Events in JavaScript: Removing event listeners</title>
    </head>
    <body>
        <button id="element">Click Me</button>
        <script>
            var element = document.getElementById('element');
            var user = {
                firstname: 'Bob',
                greeting: function(){
                    alert('My name is ' + this.firstname);
                }
            };

            // Call the method with the correct
            // context inside an anonymouse function
            element.addEventListener('click', function() {
                user.greeting();
            });
        </script>
    </body>
</html>
```

包裹之后，虽然匿名函数的作用域被指向事件触发元素，但执行的内容就像直接调用一样，不会影响其作用域。

使用 bind 方法

使用匿名函数是有缺陷的，每次调用都包裹进匿名函数里面，增加了冗余代码等，此外如果想使用removeEventListener 解除绑定，还需要再创建一个函数引用。Function 类型提供了 bind 方法，可以为函数绑定作用域，无论函数在哪里调用，都不会改变它的作用域。通过如下语句绑定作用域：

```
user.greeting = user.greeting.bind(user);
```

这样我们就可以直接使用：

```
element.addEventListener('click', user.greeting);
```

用 JavaScript 模拟触发内置事件

内置的事件也可以被 JavaScript 模拟触发，比如下面函数模拟触发单击事件：

```
//HTML代码
<input type="checkbox" id="checkbox"/><label for="checkbox">Checkbox</label>
<input type="button" onclick="simulateClick();" value="Simulate click"/>
<input type="button" onclick="addHandler();" value="Add a click handler that calls preventDefault"/>
<input type="button" onclick="removeHandler();" value="Remove the click handler that calls preventDefault"/>

//javascript代码
function preventDef(event) {
    event.preventDefault();
}

function addHandler() {
    document.getElementById("checkbox").addEventListener("click", preventDef, false);
}

function removeHandler() {
    document.getElementById("checkbox").removeEventListener("click", preventDef, false);
}

function simulateClick() {
    var evt = document.createEvent("MouseEvents");
    evt.initMouseEvent("click", true, true, window,0, 0, 0, 0, 0, false, false, false, false, 0, null);
    var cb = document.getElementById("checkbox");
    var canceled = !cb.dispatchEvent(evt);
    if(canceled) {
        // A handler called preventDefault
        alert("canceled");
    } else {
        // None of the handlers called preventDefault
        alert("not canceled");
    }
}
```

自定义事件

与自定义事件的函数有 Event、CustomEvent 和 dispatchEvent。

直接自定义事件，使用 Event 构造函数：

```
var event = new Event('build');
// Listen for the event.
elem.addEventListener('build', function (e) { ... }, false);
// Dispatch the event.
elem.dispatchEvent(event);
```

CustomEvent 可以创建一个更高度自定义事件，还可以附带一些数据，具体用法如下：

```
var myEvent = new CustomEvent(eventname, options);
```

其中 options 可以是：

```
{
    detail: {
        ...
    },
    bubbles: true,
    cancelable: false
}
```

其中 detail 可以存放一些初始化的信息，可以在触发的时候调用。其他属性就是定义该事件是否具有冒泡等等功能。

内置的事件会由浏览器根据某些操作进行触发，自定义的事件就需要人工触发。dispatchEvent 函数就是用来触发某个事件：

```
element.dispatchEvent(customEvent);
```

上面代码表示，在 element 上面触发 customEvent 这个事件。结合起来用就是：

```
// add an appropriate event listener
obj.addEventListener("cat", function(e) { process(e.detail) });

// create and dispatch the event
var event = new CustomEvent("cat", {"detail":{"hazcheeseburger":true}});
obj.dispatchEvent(event);
```

使用自定义事件需要注意兼容性问题，而使用 jQuery 就简单多了：

```
// 绑定自定义事件
$(element).on('myCustomEvent', function(){});
// 触发事件
$(element).trigger('myCustomEvent');
```

此外，你还可以在触发自定义事件时传递更多参数信息：

```
$( "p" ).on( "myCustomEvent", function( event, myName ) {
    $( this ).text( myName + ", hi there!" );
});
$( "button" ).click(function () {
    $( "p" ).trigger( "myCustomEvent", [ "John" ] );
});
```

js自定义事件原型链继承实现

```
var EventTarget = function() {
    this._listener = {};
};

EventTarget.prototype = {
    constructor: this,
    addEvent: function(type, fn) {
        if (typeof type === "string" && typeof fn === "function") {
            if (typeof this._listener[type] === "undefined") {
                this._listener[type] = [fn];
            } else {
                this._listener[type].push(fn);
            }
        }
        return this;
    },
    addEvents: function(obj) {
        obj = typeof obj === "object"? obj : {};
        var type;
        for (type in obj) {
            if ( type && typeof obj[type] === "function") {
                this.addEvent(type, obj[type]);
            }
        }
        return this;
    },
    fireEvent: function(type) {
        if (type && this._listener[type]) {
            var events = {
                type: type,
                target: this
            };
            for (var length = this._listener[type].length, start=0; start<length; start+=1) {
                this._listener[type][start].call(this, events);
            }
        }
        return this;
    },
    fireEvents: function(array) {
        if (array instanceof Array) {
            for (var i=0, length = array.length; i<length; i+=1) {
                this.fireEvent(array[i]);
            }
```

```
        }
        return this;
    },
    removeEvent: function(type, key) {
        var listeners = this._listener[type];
        if (listeners instanceof Array) {
            if (typeof key === "function") {
                for (var i=0, length=listeners.length; i<length; i+=1){
                    if (listeners[i] === listener){
                        listeners.splice(i, 1);
                        break;
                    }
                }
            } else if (key instanceof Array) {
                for (var lis=0, lenkey = key.length; lis<lenkey; lis+=1) {
                    this.removeEvent(type, key[lenkey]);
                }
            } else {
                delete this._listener[type];
            }
        }
        return this;
    },
    removeEvents: function(params) {
        if (params instanceof Array) {
            for (var i=0, length = params.length; i<length; i+=1) {
                this.removeEvent(params[i]);
            }
        } else if (typeof params === "object") {
            for (var type in params) {
                this.removeEvent(type, params[type]);
            }
        }
        return this;
    }
};
//------------- 以下为测试代码 -----------
var myEvents = new EventTarget();
myEvents.addEvents({
    "once": function() {
        alert("该弹框只会出现一次！");
        this.removeEvent("once");
    },
    "infinity": function() {
        alert("每次点击页面，该弹框都会出现！");
    }
});
document.onclick = function(e) {
    e = e || window.event;
    var target = e.target || e.srcElement;
    if (!target || !/input|pre/i.test(target.tagName)) {
        myEvents.fireEvents(["once", "infinity"]);
    }
};
```

DOM自定义事件

我们平常所使用的事件基本都是与DOM元素相关的，例如点击按钮，文本输入等，这些为自带浏览器行为事件，而自定义事件与这些行为无关。例如：

```
element.addEventListener("alert", function() {
    alert("弹出！");
});
```

这里的alert就属于自定义事件，后面的function就是自定义事件函数。而这个自定义事件是直接绑定在名为element的DOM元素上的，因此，这个称之为自定义DOM事件。

如何直接在DOM上扩展新的事件处理方法，以及执行自定义的事件呢？
我们可以直接在DOM上进行方法扩展。例如添加个addEvent方法：

```
if (window.HTMLElement) {
    // 使用原型扩展DOM自定义事件
    HTMLElement.prototype.addEvent = function(type, fn, capture) {
        var el = this;
        if (window.addEventListener) {
            el.addEventListener(type, function(e) {
                fn.call(el, e);
            }, capture);
        } else if (window.attachEvent) {
            el.attachEvent("on" + type, function(e) {
                fn.call(el, e);
            });
        }
    };
} else {
    // 如果是不支持HTMLElement扩展的浏览器
    // 通过遍历所有元素扩展DOM事件
    // IE6/7浏览器
    var elAll = document.all, lenAll = elAll.length;
    for (var iAll=0; iAll<lenAll; iAll+=1) {
        elAll[iAll].addEvent = function(type, fn) {
            var el = this;
            el.attachEvent("on" + type, function(e) {
                fn.call(el, e);
            });
        };
    }

}
//--------- 以下是测试代码 -------------
document.getElementById("image").addEvent("click", function() {
    alert("这是：" + this.alt);
});
```

上面代码中的HTMLElement表示HTML元素。以一个<p>标签元素举例，其向上寻找原型对象的过程是这样：`HTMLParagraphElement.prototype → HTMLElement.prototype → Element.prototype → Node.prototype → Object.prototype → null`。这下您应该知道HTMLElement所处的位置了吧，上述代码HTMLElement直接换成Element也是可以的，但是会让其他元素（例如文本元素）也扩展addEvent方法，有些浪费了。

由于IE6, IE7浏览器的DOM水平较低，无法直接进行扩展，因此，原型扩展的方法在这两个浏览器下是行不通的。要想让这两个浏览器也支持addEvent方法，只能是页面载入时候遍历所有DOM，然后每个都直接添加addEvent方法了。

基于DOM扩展缺点有：缺少标准无规律、提高冲突可能性、性能以及浏览器支持。 扩展名字任意命，很有可能就会与未来DOM浏览器本身支持的方法相互冲突；扩展无规律，很有可能出现A和B同名不同功能的扩展而造成冲突；IE6-7浏览器下所有扩展都要通过遍历支持，其性能开销可想而知；另外IE8对DOM扩展的支持并不完整，例如其支持Element.prototype，却没有HTMLElement.prototype

伪DOM自定义事件

“伪DOM自定义事件”是自己定义的一个名词，用来区分DOM自定义事件的。例如jQuery库，其是基于包装器（一个包含DOM元素的中间层）扩展事件的，既与DOM相关，又不直接是DOM，因此，称之为“伪DOM自定义事件”。

自定义事件的触发,对于标准浏览器，其提供了可供元素触发的方法：element.dispatchEvent(). 不过，在使用该方法之前，我们还需要做其他两件事，及创建和初始化。因此，总结说来就是：

```
document.createEvent()
event.initEvent()
element.dispatchEvent()
```

例：

```
$(dom).addEvent("alert", function() {
    alert("弹弹弹，弹走鱼尾纹~~");
});
// 创建
var evt = document.createEvent("HTMLEvents");
// 初始化
evt.initEvent("alert", false, false);
// 触发，即弹出文字
dom.dispatchEvent(evt);
```

createEvent()方法返回新创建的Event对象，支持一个参数，表示事件类型，具体见下表：

参数	事件接口	初始化方法
HTMLEvents	HTMLEvent	initEvent()
MouseEvents	MouseEvent	initMouseEvent()
UIEvents	UIEvent	initUIEvent()

initEvent()方法用于初始化通过DocumentEvent接口创建的Event的值。支持三个参数：initEvent(eventName, canBubble, preventDefault). 分别表示事件名称，是否可以冒泡，是否阻止事件的默认操作。

dispatchEvent()就是触发执行了，dom.dispatchEvent(eventObject), 参数eventObject表示事件对象，是createEvent()方法返回的创建的Event对象。

对于IE浏览器，由于向下很多版本的浏览器都不支持document.createEvent()方法，据说IE有document.createEventObject()和event.fireEvent()方法，但是不支持自定义事件。

综合示例：

```
var $ = function(el) {
    return new _$(el);
};
var _$ = function(el) {
    this.el = (el && el.nodeType == 1)? el: document;
};
_$.prototype = {
    constructor: this,
    addEvent: function(type, fn, capture) {
        var el = this.el;
        if (window.addEventListener) {
            el.addEventListener(type, fn, capture);
            var ev = document.createEvent("HTMLEvents");
            ev.initEvent(type, capture || false, false);
            // 在元素上存储创建的事件，方便自定义触发
            if (!el["ev" + type]) {
                el["ev" + type] = ev;
            }
        } else if (window.attachEvent) {
            el.attachEvent("on" + type, fn);
            if (isNaN(el["cu" + type])) {
                // 自定义属性，触发事件用
                el["cu" + type] = 0;
            }
            var fnEv = function(event) {
                if (event.propertyName == "cu" + type) {
                    fn.call(el);
                }
            };
            el.attachEvent("onpropertychange", fnEv);
            // 在元素上存储绑定的propertychange事件，方便删除
            if (!el["ev" + type]) {
                el["ev" + type] = [fnEv];
            } else {
                el["ev" + type].push(fnEv);
            }
        }
        return this;
    },
    fireEvent: function(type) {
        var el = this.el;
        if (typeof type === "string") {
            if (document.dispatchEvent) {
                if (el["ev" + type]) {
                    el.dispatchEvent(el["ev" + type]);
                }
            } else if (document.attachEvent) {
                // 改变对应自定义属性，触发自定义事件
                el["cu" + type]++;
            }
        }
        return this;
    },
```

```
    removeEvent: function(type, fn, capture) {
        var el = this.el;
        if (window.removeEventListener) {
            el.removeEventListener(type, fn, capture || false);
        } else if (document.attachEvent) {
            el.detachEvent("on" + type, fn);
            var arrEv = el["ev" + type];
            if (arrEv instanceof Array) {
                for (var i=0; i<arrEv.length; i+=1) {
                    // 删除该方法名下所有绑定的propertychange事件
                    el.detachEvent("onpropertychange", arrEv[i]);
                }
            }
        }
        return this;
    }
};
// ------------- 以下为测试用脚本------------
var fnClick = function(e) {
    e = e || window.event;
    var target = e.target || e.srcElement;
    if (target.nodeType === 1) {
        alert("点击类型: " +  e.type);
        $(target).fireEvent("alert");
    }
}, funAlert1 = function() {
    alert("自定义alert事件弹出! ");
}, funAlert2 = function() {
    alert("自定义alert事件再次弹出! ");
};

var elImage = document.getElementById("image");
$(elImage)
    .addEvent("click", fnClick)
    .addEvent("alert", funAlert1)
    .addEvent("alert", funAlert2);
// 删除自定义事件按钮
var elButton = document.getElementById("button");
$(elButton).addEvent("click", function() {
    $(elImage)
        .removeEvent("alert", funAlert1)
        .removeEvent("alert", funAlert2);
    alert("清除成功! ");
});
```

javascript自定义事件监听一个变量的变化

标准浏览器（firefox,chrome,safari,opera等），简单的说来，自定义事件到激发这个事件，需要document.createEvent(), event.initEvent()，element.dispatchEvent()这三部，分别是创建事件对象，初始化事件对象，触发事件。

```
function foo1(){
    console.log("foo1 is execute");
}
function foo2(){
    console.log("foo2 is execute");
}
var ev=document.createEvent('HTMLEvents');
ev.initEvent('fakeEvent',false,false);
document.addEventListener("fakeEvent",foo1,false);
document.addEventListener("fakeEvent",foo2,false);
```

在标准浏览器里的console里执行 document.dispatchEvent(ev); 就可以看到console里显示出来了 foo1 is execute和 foo2 is execute

自定义事件监听一个变量的变化示例:

```
function foo1(){
    addLog("foo1 is excute");
}
function foo2(){
    addLog("the id is "+idChange.getId()+" now!");
}
if(document.createEvent){ //This is for the stand browser.
    var ev=document.createEvent('HTMLEvents');
    ev.initEvent('fakeEvent',false,false);
    document.addEventListener("fakeEvent",foo1,false);
    document.addEventListener("fakeEvent",foo2,false);
}else if(document.attachEvent){ //This is for the damn IE
    document.documentElement.fakeEvents = 0; // an expando property
    document.documentElement.attachEvent("onpropertychange", function(event) {
        if (event.propertyName == "fakeEvents") {
            foo1();
        }
    });
    document.documentElement.attachEvent("onpropertychange",function(event){
        if(event.propertyName == "fakeEvents"){
            foo2();
        }
    });
}
function addLog(log){
    var logDiv=document.getElementById('log');
    var p=document.createElement("p");
    p.appendChild(document.createTextNode(log));
    logDiv.appendChild(p);
}
var idChange=function(){
    var id=1;
    return {getId:function(){return id;},
            setId:function(a){
           id=a;
           if(document.dispatchEvent) document.dispatchEvent(ev);
       else if(document.attachEvent)     document.documentElement.fakeEvents++; //This for IE
                }}
}();
```

jQuery.event自定义事件机制-jQuery.event.special范例

jQuery(elem).bind(type, callbakc)实际上是映射到 jQuery.event.add(elem, types, handler, data)这个方法，每一个类型的事件会初始化一次事件处理器，而传入的回调函数会以数组的方式缓存起来，当事件触发的时候处理器将依次执行这个数组。

jQuery.event.add方法在第一次初始化处理器的时候会检查是否为自定义事件，如果存在则将会把控制权限交给自定义事件的事件初始化函数，同样事件卸载的jQuery.event.remove方法在删除处理器前也会检查此。

如jQuery源码：

初始化处事件处理器

```
// Check for a special event handler
// Only use addEventListener/attachEvent if the special
// events handler returns false
if ( !special.setup || special.setup.call( elem, data, namespaces, eventHandle ) === false ) {
    // Bind the global event handler to the element
    if ( elem.addEventListener ) {
            elem.addEventListener( type, eventHandle, false );
    } else if ( elem.attachEvent ) {
            elem.attachEvent( "on" + type, eventHandle );
    }
}
```

卸载处理器：

```
if ( !special.teardown || special.teardown.call( elem, namespaces ) === false ) {
    jQuery.removeEvent( elem, type, elemData.handle );
}
```

入口

```
jQuery.event.special[youEvent] = {
    /**
     * 初始化事件处理器 - this指向元素
     * @param 附加的数据
     * @param 事件类型命名空间
     * @param 回调函数
     */
    setup: function (data, namespaces, eventHandle) {
    },
    /**
     * 卸载事件处理器 - this指向元素
     * @param 事件类型命名空间
     */
    teardown: function (namespaces) {
    }
};
```

接下来我们做一个最简单的自定义插件，给jQuery提供input跨浏览器事件支持。input事件不同于keydown与keyup，它不依赖键盘响应，只要值改变都会触发input事件，比如粘贴文字、使用在线软键盘等。

范例

```
(function ($) {
    // IE6\7\8不支持input事件，但支持propertychange事件
    if ('onpropertychange' in document) {
        // 检查是否为可输入元素
        var rinput = /^INPUT|TEXTAREA$/,
            isInput = function (elem) {
                return rinput.test(elem.nodeName);
            };
        $.event.special.input = {
            setup: function () {
                var elem = this;
                if (!isInput(elem)) return false;
                $.data(elem, '@oldValue', elem.value);
                $.event.add(elem, 'propertychange', function (event) {
                    // 元素属性任何变化都会触发propertychange事件
                    // 需要屏蔽掉非value的改变，以便接近标准的onput事件
                    if ($.data(this, '@oldValue') !== this.value) {
                        $.event.trigger('input', null, this);
                    };
                    $.data(this, '@oldValue', this.value);
                });
            },
            teardown: function () {
                var elem = this;
                if (!isInput(elem)) return false;
                $.event.remove(elem, 'propertychange');
                $.removeData(elem, '@oldValue');
            }
        };
    };

    // 声明快捷方式：
    $(elem).input(function () {});
    $.fn.input = function (callback) {
        return this.bind('input', callback);
    };
})(jQuery);
```

调用：

```
jQuery(elem).bind('input', function () {});
```

JQuery通过bind绑定一个自定义事件，然后再通过trigger来触发这个事件。例如给element绑定一个hello事件，再通过trigger来触发这个事件：

```
//给element绑定hello事件
element.bind("hello",function(){
    alert("hello world!");
});
//触发hello事件
element.trigger("hello");
```

参考文章:
JavaScript 和事件
http://yujiangshui.com/javascript-event/
http://www.zhangxinxu.com/study/201203/js-custom-events-prototypal.html
http://www.zhangxinxu.com/wordpress/2012/04/js-dom%E8%87%AA%E5%AE%9A%E4%B9%89%E4%BA%8B%E4%BB%B6/
http://blog.allenm.me/2010/02/javascript%E8%87%AA%E5%AE%9A%E4%B9%89%E4%BA%8B%E4%BB%B6event/
http://dean.edwards.name/weblog/2009/03/callbacks-vs-events/
http://www.cnblogs.com/binyong/articles/1750263.html http://bbs.phpchina.com/thread-221206-1-1.html

---

setTimeout及Javascript运行机制Event Loop

setTimeout

setTimeout() 方法用于在指定的毫秒数后调用函数或计算表达式。 语法:

```
setTimeout(fn,millisec)
```

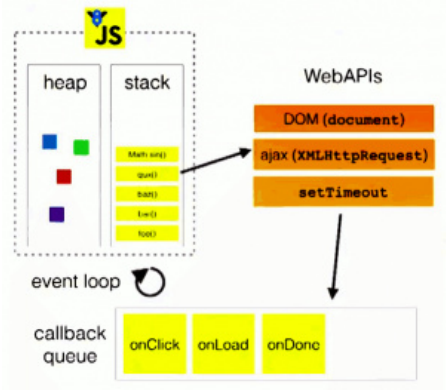
fn表示要执行的代码，可以是一个包含javascript代码的字符串，也可以是一个函数。
millisec是以毫秒表示的时间，表示fn需推迟多长时间执行。
调用setTimeout()方法之后，该方法返回一个数字，这个数字是计划执行代码的唯一标识符，可以通过它来取消超时调用。

```
var start = new Date;
setTimeout(function(){
    var end = new Date;
    console.log('Time elapsed:', end - start, 'ms');
}, 500);
while (new Date - start < 1000) {};
```

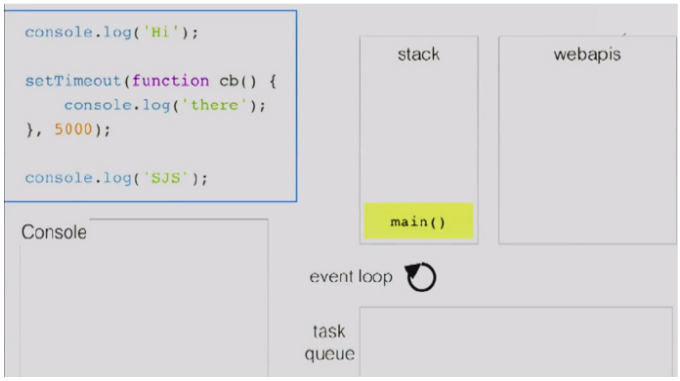
在我最初对setTimeout()的认识中，延时设置为500ms，所以输出应该为Time elapsed: 500 ms。可是实际上，上述代码运行多次后，输出至少是延迟了1000ms。
通过阅读代码不难看出，setTimeout()方法执行在while()循环之前，它声明了“希望”在500ms之后执行一次匿名函数，这一声明，也即对匿名函数的注册，在setTimeout()方法执行后立即生效。代码最后一行的while循环会持续运行1000ms，通过setTimeout()方法注册的匿名函数输出的延迟时间总是大于1000ms，说明对这一匿名函数的实际调用被while()循环阻塞了，实际的调用在while()循环阻塞结束后才真正执行。
在现有浏览器环境中，Javascript执行引擎是单线程的，主线程的语句和方法，会阻塞定时任务的运行，执行引擎只有在执行完主线程的语句后，定时任务才会实际执行，这期间的时间，可能大于注册任务时设置的延时时间。
执行引擎先将setTimeout()方法入栈被执行，执行时将延时方法交给内核相应模块处理。引擎继续处理后面代码，while语句将引擎阻塞了1秒，而在这过程中，内核timer模块在0.5秒时已将延时方法添加到任务队列，在引擎执行栈清空后，引擎将延时方法入栈并处理，最终输出的时间超过预期设置的时间。

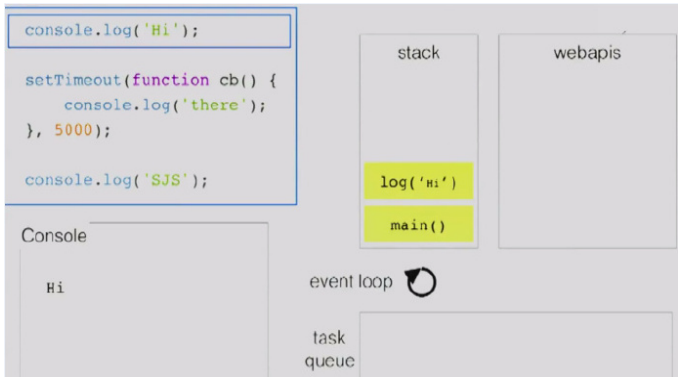
事件循环模型

在单线程的Javascript引擎中，setTimeout()是如何运行的呢，这里就要提到浏览器内核中的事件循环模型了。简单的讲，在Javascript执行引擎之外，有一个任务队列，当在代码中调用setTimeout()方法时，注册的延时方法会交由浏览器内核其他模块（以webkit为例，是webcore模块）处理，当延时方法到达触发条件，即到达设置的延时时间时，这一延时方法被添加至任务队列里。这一过程由浏览器内核其他模块处理，与执行引擎主线程独立，执行引擎在主线程方法执行完毕，到达空闲状态时，会从任务队列中顺序获取任务来执行，这一过程是一个不断循环的过程，称为事件循环模型。

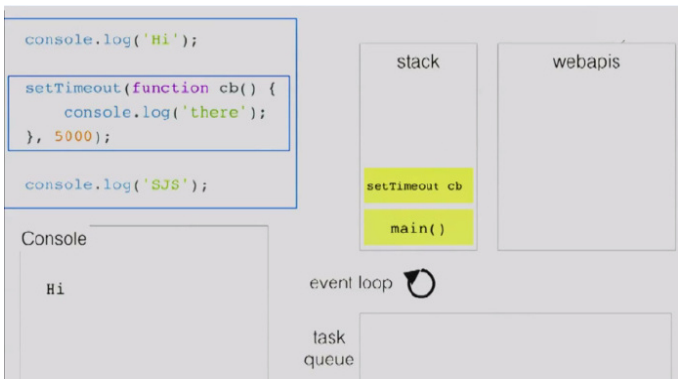


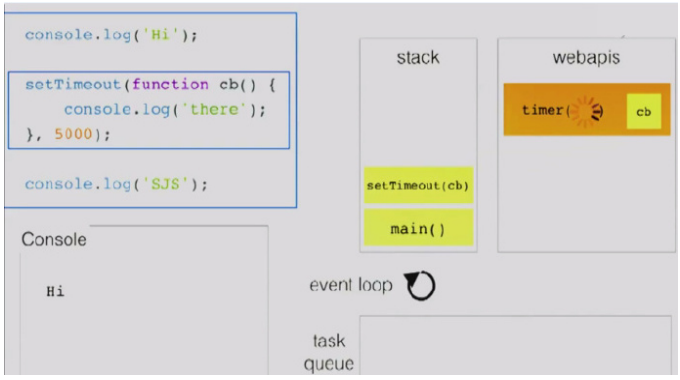
javascript执行引擎的主线程运行的时候，产生堆（heap）和栈（stack）。程序中代码依次进入栈中等待执行，当调用setTimeout()方法时，即图中右侧WebAPIs方法时，浏览器内核相应模块开始延时方法的处理，当延时方法到达触发条件时，方法被添加到用于回调的任务队列，只要执行引擎栈中的代码执行完毕，主线程就会去读取任务队列，依次执行那些满足触发条件的回调函数。



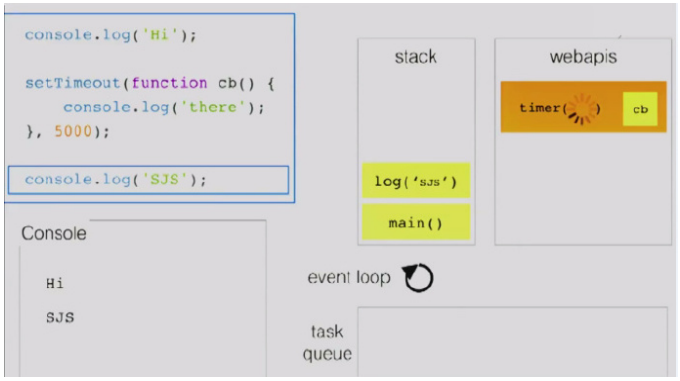


执行引擎开始执行上述代码时，相当于先讲一个main()方法加入执行栈。继续往下开始console.log('Hi')时，log('Hi')方法入栈，console.log方法是一个webkit内核支持的普通方法，而不是前面图中WebAPIs涉及的方法，所以这里log('Hi')方法立即出栈被引擎执行.

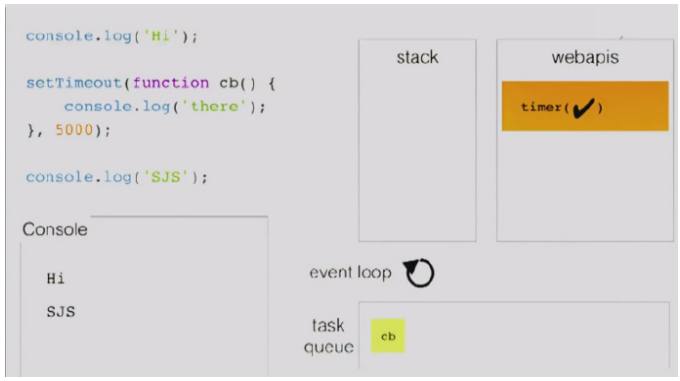


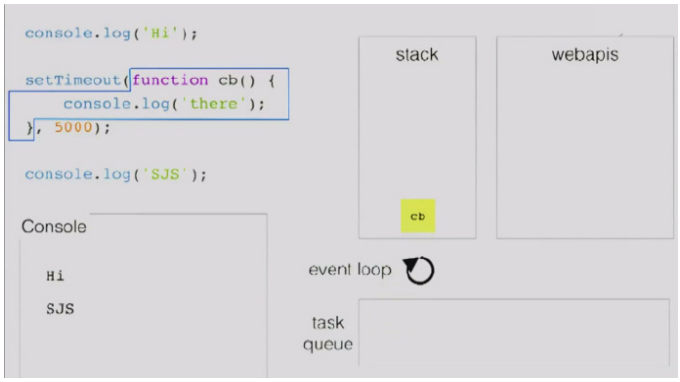


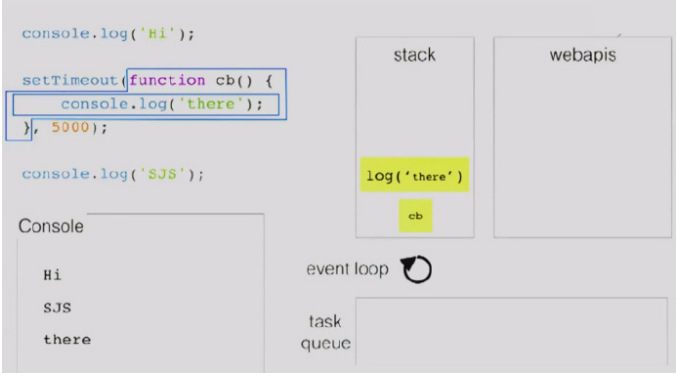
console.log('Hi')语句执行完成后，log()方法出栈执行，输出了Hi。引擎继续往下，将setTimeout(callback,5000)添加到执行栈。setTimeout()方法属于事件循环模型中WebAPIs中的方法，引擎在将setTimeout()方法出栈执行时，将延时执行的函数交给了相应模块，即图右方的timer模块来处理。



执行引擎将setTimeout出栈执行时，将延时处理方法交由了webkit timer模块处理，然后立即继续往下处理后面代码，于是将log('SJS')加入执行栈，接下来log('SJS')出栈执行，输出SJS。而执行引擎在执行万console.log('SJS')后，程序处理完毕，main()方法也出栈。



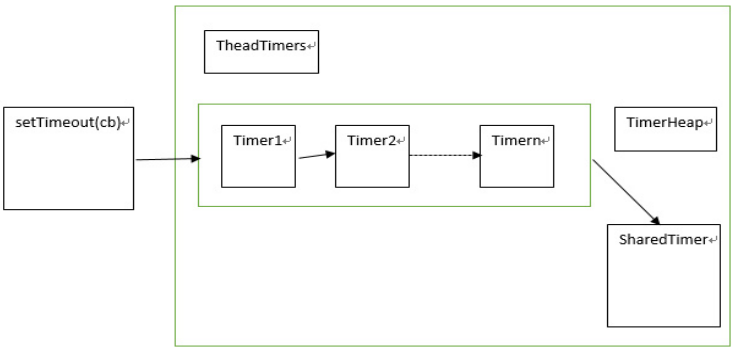




这时在在setTimeout方法执行5秒后，timer模块检测到延时处理方法到达触发条件，于是将延时处理方法加入任务队列。而此时执行引擎的执行栈为空，所以引擎开始轮询检查任务队列是否有任务需要被执行，就检查到已经到达执行条件的延时方法，于是将延时方法加入执行栈。引擎发现延时方法调用了log()方法，于是又将log()方法入栈。然后对执行栈依次出栈执行，输出there，清空执行栈。
清空执行栈后，执行引擎会继续去轮询任务队列，检查是否还有任务可执行。

webkit中timer的实现

事件循环模型图中提到的WebAPIs部分，提到了DOM事件，AJAX调用和setTimeout方法，图中简单的把它们总结为WebAPIs，而且他们同样都把回调函数添加到任务队列等待引擎执行。这是一个简化的描述，实际上浏览器内核对DOM事件、AJAX调用和setTimeout方法都有相应的模块来处理，webkit内核在Javasctipt执行引擎之外，有一个重要的模块是webcore模块，html的解析，css样式的计算等都由webcore实现。对于图中WebAPIs提到的三种API，webcore分别提供了DOM Binding、network、timer模块来处理底层实现。 Timer类是webkit 内核的一个必需的基础组件，通过阅读源码可以全面理解其原理，本文对其简化，分析其执行流程。



通过setTimeout()方法注册的延时方法，被传递给webcore组件timer模块处理。timer中关键类为TheadTimers类，其包含两个重要成员，TimerHeap任务队列和SharedTimer方法调度类。延时方法被封装为timer对象，存储在TimerHeap中。和Java.util.Timer任务队列一样，TimerHeap同样采用最小堆的数据结构，以nextFireTime作为关键字排序。SharedTimer作为TimerHeap调度类，在timer对象到达触发条件时，通过浏览器平台相关的接口，将延时方法添加到事件循环模型中提到的任务队列中。
TimerHeap采用最小堆的数据结构，预期延时时间最小的任务最先被执行，同时，预期延时时间相同的两个任务，其执行顺序是按照注册的先后顺序执行。

```
var start = new Date;

setTimeout(function(){
    console.log('fn1');
}, 20);

setTimeout(function(){
    console.log('fn2');
}, 30);

setTimeout(function(){
    console.log('another fn2');
}, 30);

setTimeout(function(){
    console.log('fn3');
}, 10);

console.log('start while');
while (new Date - start < 1000) {};
console.log('end while');

/*
执行结果
start while
end while
fn3
fn1
fn2
another fn2
*/
```

JavaScript运行机制

任务队列

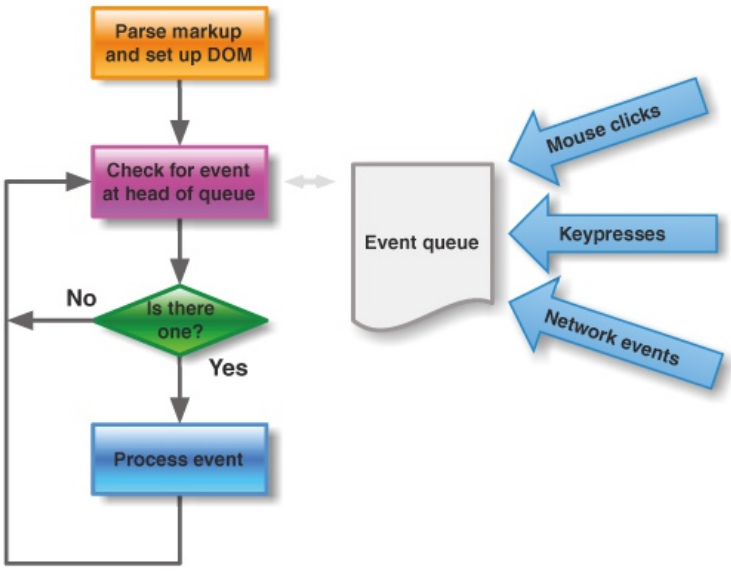
单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。

任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

异步执行的运行机制如下：

1. 所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。
2. 主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。
3. 一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
4. 主线程不断重复上面的第三步。

下图就是主线程和任务队列的示意图



只要主线程空了，就会去读取"任务队列"，这就是JavaScript的运行机制。这个过程会不断重复。

事件和回调函数

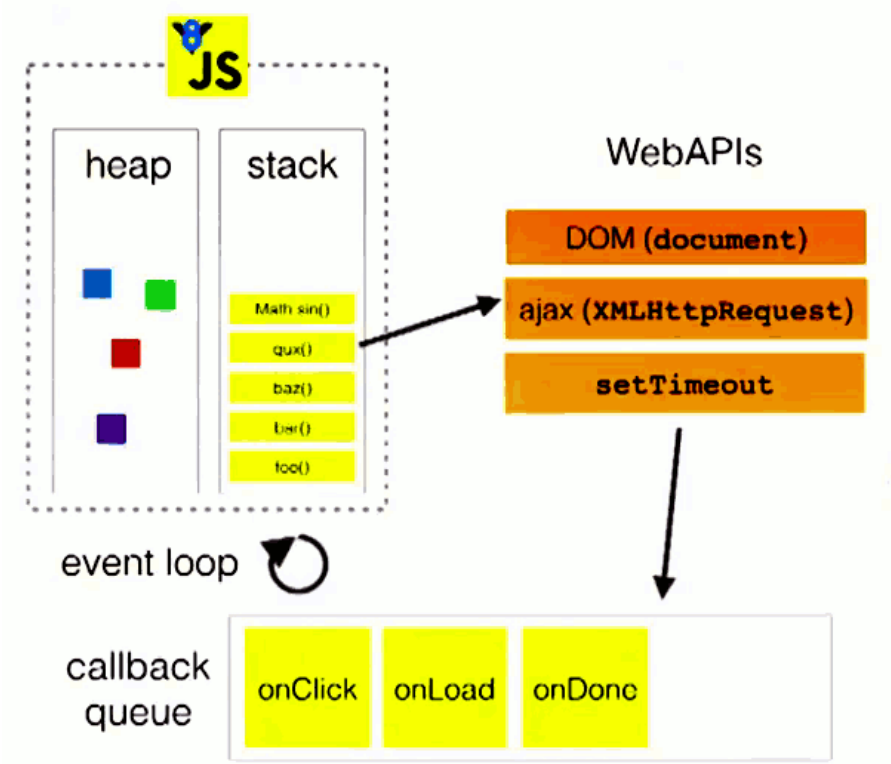
"任务队列"是一个事件的队列（也可以理解成消息的队列），IO设备完成一项任务，就在"任务队列"中添加一个事件，表示相关的异步任务可以进入"执行栈"了。主线程读取"任务队列"，就是读取里面有哪些事件。

"任务队列"中的事件，除了IO设备的事件以外，还包括一些用户产生的事件（比如鼠标点击、页面滚动等等）。只要指定过回调函数，这些事件发生时就会进入"任务队列"，等待主线程读取。

所谓"回调函数"（callback），就是那些会被主线程挂起来的代码。异步任务必须指定回调函数，当主线程开始执行异步任务，就是执行对应的回调函数。

"任务队列"是一个先进先出的数据结构，排在前面的事件，优先被主线程读取。主线程的读取过程基本上是自动的，只要执行栈一清空，"任务队列"上第一位的事件就自动进入主线程。但是，由于存在后文提到的"定时器"功能，主线程首先要检查一下执行时间，某些事件只有到了规定的时间，才能返回主线程。

Event Loop



上图中，主线程运行的时候，产生堆（heap）和栈（stack），栈中的代码调用各种外部API，它们在"任务队列"中加入各种事件（click，load，done）。只要栈中的代码执行完毕，主线程就会去读取"任务队列"，依次执行那些事件所对应的回调函数。 执行栈中的代码（同步任务），总是在读取"任务队列"（异步任务）之前执行。请看下面这个例子。

```
var req = new XMLHttpRequest();
req.open('GET', url);
req.onload = function (){};
req.onerror = function (){};
req.send();
```

上面代码中的req.send方法是Ajax操作向服务器发送数据，它是一个异步任务，意味着只有当前脚本的所有代码执行完，系统才会去读取"任务队列"。所以，它与下面的写法等价。

```
var req = new XMLHttpRequest();
req.open('GET', url);
req.send();
req.onload = function (){};
req.onerror = function (){};
```

也就是说，指定回调函数的部分（onload和onerror），在send()方法的前面或后面无关紧要，因为它们属于执行栈的一部分，系统总是执行完它们，才会去读取"任务队列"。

定时器

除了放置异步任务的事件，"任务队列"还可以放置定时事件，即指定某些代码在多少时间之后执行。这叫做"定时器"（timer）功能，也就是定时执行的代码。

定时器功能主要由setTimeout()和setInterval()这两个函数来完成，它们的内部运行机制完全一样，区别在于前者指定的代码是一次性执行，后者则为反复执行。以下主要讨论setTimeout()。
setTimeout()接受两个参数，第一个是回调函数，第二个是推迟执行的毫秒数。

```
console.log(1);
setTimeout(function(){console.log(2);},1000);
console.log(3);
```

上面代码的执行结果是1，3，2，因为setTimeout()将第二行推迟到1000毫秒之后执行。

如果将setTimeout()的第二个参数设为0，就表示当前代码执行完（执行栈清空）以后，立即执行（0毫秒间隔）指定的回调函数。

```
setTimeout(function(){console.log(1);}, 0);
console.log(2);
```

上面代码的执行结果总是2，1，因为只有在执行完第二行以后，系统才会去执行"任务队列"中的回调函数。

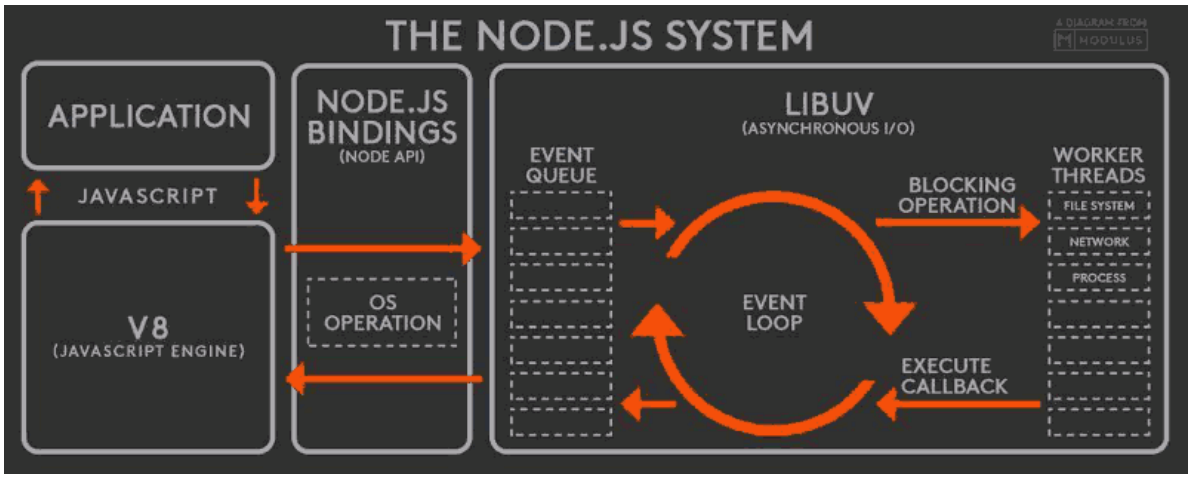
总之，setTimeout(fn,0)的含义是，指定某个任务在主线程最早可得的空闲时间执行，也就是说，尽可能早得执行。它在"任务队列"的尾部添加一个事件，因此要等到同步任务和"任务队列"现有的事件都处理完，才会得到执行。

HTML5标准规定了setTimeout()的第二个参数的最小值（最短间隔），不得低于4毫秒，如果低于这个值，就会自动增加。在此之前，老版本的浏览器都将最短间隔设为10毫秒。另外，对于那些DOM的变动（尤其是涉及页面重新渲染的部分），通常不会立即执行，而是每16毫秒执行一次。这时使用requestAnimationFrame()的效果要好于setTimeout()。

需要注意的是，setTimeout()只是将事件插入了"任务队列"，必须等到当前代码（执行栈）执行完，主线程才会去执行它指定的回调函数。要是当前代码耗时很长，有可能要等很久，所以并没有办法保证，回调函数一定会在setTimeout()指定的时间执行。

Node.js的Event Loop

Node.js也是单线程的Event Loop，但是它的运行机制不同于浏览器环境。



根据上图，Node.js的运行机制如下:

1. V8引擎解析JavaScript脚本。
2. 解析后的代码，调用Node API。
3. libuv库负责Node API的执行。它将不同的任务分配给不同的线程，形成一个Event Loop（事件循环），以异步的方式将任务的执行结果返回给V8引擎。
4. V8引擎再将结果返回给用户。

除了setTimeout和setInterval这两个方法，Node.js还提供了另外两个与"任务队列"有关的方法：process.nextTick和setImmediate。它们可以帮助我们加深对"任务队列"的理解。

process.nextTick方法可以在当前"执行栈"的尾部----下一次Event Loop（主线程读取"任务队列"）之前----触发回调函数。也就是说，它指定的任务总是发生在所有异步任务之前。setImmediate方法则是在当前"任务队列"的尾部添加事件，也就是说，它指定的任务总是在下一次Event Loop时执行，这与setTimeout(fn, 0)很像。请看下面的例子。

```
process.nextTick(function A() {
    console.log(1);
    process.nextTick(function B(){console.log(2);});
});

setTimeout(function timeout() {
    console.log('TIMEOUT FIRED');
}, 0)

//运行结果
// 1
// 2
// TIMEOUT FIRED
```

上面代码中，由于process.nextTick方法指定的回调函数，总是在当前"执行栈"的尾部触发，所以不仅函数A比setTimeout指定的回调函数timeout先执行，而且函数B也比timeout先执行。这说明，如果有多个process.nextTick语句（不管它们是否嵌套），将全部在当前"执行栈"执行。 现在，再看setImmediate。

```
setImmediate(function A() {
    console.log(1);
    setImmediate(function B(){console.log(2);});
    });

setTimeout(function timeout() {
    console.log('TIMEOUT FIRED');
}, 0);
```

上面代码中，setImmediate与setTimeout(fn,0)各自添加了一个回调函数A和timeout，都是在下一次Event Loop触发。那么，哪个回调函数先执行呢？答案是不确定。**运行结果可能是1--TIMEOUT FIRED--2，也可能是TIMEOUT FIRED--1--2**。

令人困惑的是，Node.js文档中称，setImmediate指定的回调函数，总是排在setTimeout前面。实际上，这种情况只发生在递归调用的时候。

```
setImmediate(function (){
    setImmediate(function A() {
        console.log(1);
        setImmediate(function B(){console.log(2);});
    });

    setTimeout(function timeout() {
        console.log('TIMEOUT FIRED');
    }, 0);
});

//运行结果
// 1
// TIMEOUT FIRED
// 2
```

上面代码中，setImmediate和setTimeout被封装在一个setImmediate里面，它的运行结果总是1--TIMEOUT FIRED--2，这时函数A一定在timeout前面触发。至于2排在TIMEOUT FIRED的后面（即函数B在timeout后面触发），是因为setImmediate总是将事件注册到下一轮Event Loop，所以函数A和timeout是在同一轮Loop执行，而函数B在下一轮Loop执行。

我们由此得到了process.nextTick和setImmediate的一个重要区别：`多个process.nextTick语句总是在当前"执行栈"一次执行完，多个setImmediate可能则需要多次loop才能执行完。`事实上，这正是Node.js 10.0版添加setImmediate方法的原因，否则像下面这样的递归调用process.nextTick，将会没完没了，主线程根本不会去读取"事件队列"！

```
process.nextTick(function foo() {
    process.nextTick(foo);
});
```

事实上，现在要是你写出递归的process.nextTick，Node.js会抛出一个警告，要求你改成setImmediate。

另外，由于process.nextTick指定的回调函数是在本次"事件循环"触发，而setImmediate指定的是在下次"事件循环"触发，所以很显然，前者总是比后者发生得早，而且执行效率也高（因为不用检查"任务队列"）。

参考文章:
从setTimeout说事件循环模型
http://web.jobbole.com/83883/#comment-95871
JavaScript 运行机制详解：再谈Event Loop
http://www.ruanyifeng.com/blog/2014/10/event-loop.html

JavaScript异步编程及Promise模式

Javascript异步编程

Javascript语言将任务的执行模式分成两种：同步（Synchronous）和异步（Asynchronous）。

"同步模式"就是上一段的模式，后一个任务等待前一个任务结束，然后再执行，程序的执行顺序与任务的排列顺序是一致的、同步的；"异步模式"则完全不同，每一个任务有一个或多个回调函数（callback），前一个任务结束后，不是执行后一个任务，而是执行回调函

数，后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。

"异步模式"非常重要。在浏览器端，耗时很长的操作都应该异步执行，避免浏览器失去响应，最好的例子就是Ajax操作。

"异步模式"编程的4种方法

一、回调函数

这是异步编程最基本的方法。
假定有两个函数f1和f2，后者等待前者的执行结果。

```
f1();
f2();
```

如果f1是一个很耗时的任务，可以考虑改写f1，把f2写成f1的回调函数。

```
function f1(callback){
    setTimeout(function () {
        // f1的任务代码
        callback();
    }, 1000);
}
```

执行代码就变成下面这样：

```
f1(f2);
```

采用这种方式，我们把同步操作变成了异步操作，f1不会堵塞程序运行，相当于先执行程序的主要逻辑，将耗时的操作推迟执行。

回调函数的优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度耦合（Coupling），流程会很混乱，而且每个任务只能指定一个回调函数。

二、事件监听

另一种思路是采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

还是以f1和f2为例。首先，为f1绑定一个事件（这里采用的jQuery的写法）。

```
f1.on('done', f2);
```

上面这行代码的意思是，当f1发生done事件，就执行f2。然后，对f1进行改写：

```
function f1(){
    setTimeout(function () {
        // f1的任务代码
        f1.trigger('done');
    }, 1000);
}
```

f1.trigger('done')表示，执行完成后，立即触发done事件，从而开始执行f2。

这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以"去耦合"（Decoupling），有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。

三、发布/订阅

我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做"发布/订阅模式"（publish-subscribe pattern），又称"观察者模式"（observer pattern）。

这个模式有多种实现，下面采用的是Ben Alman的Tiny Pub/Sub，这是jQuery的一个插件。

首先，f2向"信号中心"jQuery订阅"done"信号。

```
jQuery.subscribe("done", f2);
```

然后，f1进行如下改写：

```
function f1(){
    setTimeout(function () {
        // f1的任务代码
        jQuery.publish("done");
    }, 1000);
}
```

jQuery.publish("done")的意思是，f1执行完成后，向"信号中心"jQuery发布"done"信号，从而引发f2的执行。

此外，f2完成执行后，也可以取消订阅（unsubscribe）。

```
jQuery.unsubscribe("done", f2);
```

这种方法的性质与"事件监听"类似，但是明显优于后者。因为我们可以通过查看"消息中心"，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

Promises对象

Promises对象是CommonJS工作组提出的一种规范，目的是为异步编程提供统一接口。

简单说，它的思想是，每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数。比如，f1的回调函数f2,可以写成：

```
f1().then(f2);
```

f1要进行如下改写（这里使用的是jQuery的实现）：

```
function f1(){
    var dfd = $.Deferred();
    setTimeout(function () {
        // f1的任务代码
        dfd.resolve();
    }, 500);
    return dfd.promise;
}
```

这样写的优点在于，回调函数变成了链式写法，程序的流程可以看得很清楚，而且有一整套的配套方法，可以实现许多强大的功能。

比如，指定多个回调函数：

```
f1().then(f2).then(f3);
```

再比如，指定发生错误时的回调函数：

```
f1().then(f2).fail(f3);
```

而且，它还有一个前面三种方法都没有的好处：如果一个任务已经完成，再添加回调函数，该回调函数会立即执行。所以，你不用担心是否错过了某个事件或信号。这种方法的缺点就是编写和理解，都相对比较难。

Jquery中的异步编程模式--deferred对象

jQuery 从 1.5 版本引入了 deferred 对象，这是一个基于 CommonJS Promises/A 的设计。

一、什么是deferred对象？

开发网站的过程中，我们经常遇到某些耗时很长的javascript操作。其中，既有异步的操作（比如ajax读取服务器数据），也有同步的操作（比如遍历一个大型数组），它们都不是立即能得到结果的。 通常的做法是，为它们指定回调函数（callback）。即事先规定，一旦它们运行结束，应该调用哪些函数。 但是，在回调函数方面，jQuery的功能非常弱。为了改变这一点，jQuery开发团队就设计了deferred对象。 简单说，deferred对象就是jQuery的回调函数解决方案。在英语中，defer的意思是"延迟"，所以deferred对象的含义就是"延迟"到未来某个点再执行。

二、ajax操作的链式写法

```
$.ajax({
    url: "test.html",
    success: function(){
        alert("哈哈，成功了！");
    },
    error:function(){
        alert("出错啦！");
    }
});
```

在上面的代码中，$.ajax()接受一个对象参数，这个对象包含两个方法：success方法指定操作成功后的回调函数，error方法指定操作失败后的回调函数。

$.ajax()操作完成后，如果使用的是低于1.5.0版本的jQuery，返回的是XHR对象，你没法进行链式操作；如果高于1.5.0版本，返回的是deferred对象，可以进行链式操作。

现在，新的写法是这样的：

```
$.ajax("test.html")
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

可以看到，done()相当于success方法，fail()相当于error方法。采用链式写法以后，代码的可读性大大提高。

三、指定同一操作的多个回调函数

deferred对象的一大好处，就是它允许你自由添加多个回调函数。
还是以上面的代码为例，如果ajax操作成功后，除了原来的回调函数，我还想再运行一个回调函数，怎么办？
很简单，直接把它加在后面就行了。

```
$.ajax("test.html")
.done(function(){ alert("哈哈，成功了！");} )
.fail(function(){ alert("出错啦！"); } )
.done(function(){ alert("第二个回调函数！");} );
```

回调函数可以添加任意多个，它们按照添加顺序执行。

四、为多个操作指定回调函数

deferred对象的另一大好处，就是它允许你为多个事件指定一个回调函数，这是传统写法做不到的。
请看下面的代码，它用到了一个新的方法$.when()：

```
$.when($.ajax("test1.html"), $.ajax("test2.html"))
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

这段代码的意思是，先执行两个操作$.ajax("test1.html")和$.ajax("test2.html")，如果都成功了，就运行done()指定的回调函数；如果有一个失败或都失败了，就执行fail()指定的回调函数。

五、普通操作的回调函数接口（上）

deferred对象的最大优点，就是它把这一套回调函数接口，从ajax操作扩展到了所有操作。也就是说，任何一个操作----不管是ajax操作还是本地操作，也不管是异步操作还是同步操作----都可以使用deferred对象的各种方法，指定回调函数。

我们来看一个具体的例子。假定有一个很耗时的操作wait：

```
var wait = function(){
    var tasks = function(){
        alert("执行完毕！");
    };
    setTimeout(tasks,5000);
};
```

我们为它指定回调函数，应该怎么做呢？
很自然的，你会想到，可以使用$.when()：

```
$.when(wait())
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

但是，这样写的话，done()方法会立即执行，起不到回调函数的作用。原因在于$.when()的参数只能是deferred对象，所以必须对wait()进行改写：

```
var dtd = $.Deferred(); // 新建一个deferred对象
var wait = function(dtd){
    var tasks = function(){
        alert("执行完毕！");
        dtd.resolve(); // 改变deferred对象的执行状态
    };
    setTimeout(tasks,5000);
    return dtd;
};
```

现在，wait()函数返回的是deferred对象，这就可以加上链式操作了。

```
$.when(wait(dtd))
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

wait()函数运行完，就会自动运行done()方法指定的回调函数。

六、deferred.resolve()方法和deferred.reject()方法

jQuery规定，deferred对象有三种执行状态----未完成，已完成和已失败。如果执行状态是"已完成"（resolved）,deferred对象立刻调用done()方法指定的回调函数；如果执行状态是"已失败"，调用fail()方法指定的回调函数；如果执行状态是"未完成"，则继续等待，或者调用progress()方法指定的回调函数（jQuery1.7版本添加）。

前面部分的ajax操作时，deferred对象会根据返回结果，自动改变自身的执行状态；但是，在wait()函数中，这个执行状态必须由程序员手动指定。dtd.resolve()的意思是，将dtd对象的执行状态从"未完成"改为"已完成"，从而触发done()方法。

类似的，还存在一个deferred.reject()方法，作用是将dtd对象的执行状态从"未完成"改为"已失败"，从而触发fail()方法。

```
var dtd = $.Deferred(); // 新建一个Deferred对象
var wait = function(dtd){
    var tasks = function(){
        alert("执行完毕！");
        dtd.reject(); // 改变Deferred对象的执行状态
    };
    setTimeout(tasks,5000);
    return dtd;
};
$.when(wait(dtd))
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

七、deferred.promise()方法

上面这种写法，还是有问题。那就是dtd是一个全局对象，所以它的执行状态可以从外部改变。
请看下面的代码：

```
var dtd = $.Deferred(); // 新建一个Deferred对象
var wait = function(dtd){
    var tasks = function(){
        alert("执行完毕！");
        dtd.resolve(); // 改变Deferred对象的执行状态
    };
    setTimeout(tasks,5000);
    return dtd;
};
$.when(wait(dtd))
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
dtd.resolve();
```

我在代码的尾部加了一行dtd.resolve()，这就改变了dtd对象的执行状态，因此导致done()方法立刻执行，跳出"哈哈，成功了！"的提示框，等5秒之后再跳出"执行完毕！"的提示框。

为了避免这种情况，jQuery提供了deferred.promise()方法。它的作用是，在原来的deferred对象上返回另一个deferred对象，后者只开放与改变执行状态无关的方法（比如done()方法和fail()方法），屏蔽与改变执行状态有关的方法（比如resolve()方法和reject()方法），从而使得执行状态不能被改变。

请看下面的代码:

```
var dtd = $.Deferred(); // 新建一个Deferred对象
var wait = function(dtd){
    var tasks = function(){
        alert("执行完毕！");
        dtd.resolve(); // 改变Deferred对象的执行状态
    };
    setTimeout(tasks,5000);
    return dtd.promise(); // 返回promise对象
};
var d = wait(dtd); // 新建一个d对象，改为对这个对象进行操作
$.when(d)
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
d.resolve(); // 此时，这个语句是无效的
```

在上面的这段代码中，wait()函数返回的是promise对象。然后，我们把回调函数绑定在这个对象上面，而不是原来的deferred对象上面。这样的好处是，无法改变这个对象的执行状态，要想改变执行状态，只能操作原来的deferred对象。

这里有两个地方需要注意。
首先，wait最后一行不能直接返回dtd，必须返回dtd.promise()。原因是jQuery规定，任意一个deferred对象有三种执行状态—-未完成，已完成和已失败。如果直接返回dtd，$.when()的默认执行状态为”已完成”，立即触发后面的done()方法，这就失去回调函数的作用了。dtd.promise()的目的，就是保证目前的执行状态—-也就是”未完成”—-不变，从而确保只有操作完成后，才会触发回调函数。 也就是说，deferred.promise() 只是阻止其他代码来改变这个 deferred 对象的状态。可以理解成，通过 deferred.promise() 方法返回的 deferred promise 对象，是没有 resolve ,reject, progress , resolveWith, rejectWith , progressWith 这些可以改变状态的方法，你只能使用 done, then ,fail 等方法添加 handler 或者判断状态。
deferred.promise() 改变不了 deferred 对象的状态，作用也不是保证目前的状态不变，它只是保证你不能通过 deferred.promise() 返回的 deferred promise 对象改变 deferred 对象的状态。如果我们这个地方直接返回 dtd，也是可以工作的，.done 的处理函数还是会等到 dtd.resolve() 之后才会执行.
我们把代码改成如下的形式:

```
var dtd = $.Deferred(); // 新建一个deferred对象
var wait = function(dtd){
    var tasks = function(){
        alert("执行完毕！");
        dtd.resolve(); // 改变deferred对象的执行状态
    };
    setTimeout(tasks,5000);
    return dtd;
};
$.when(wait(dtd))
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

这样的执行结果和先前返回 dtd.promise 的结果是一样的。
差别在什么地方呢？ 如果我们把 $.when 的这块的代码改成这样的:

```
var d = wait(dtd);
$.when(d)
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
d.resolve();
```

不过，更好的写法是allenm所指出的，将dtd对象变成wait() 函数的内部对象。

```
var wait = function(dtd){
    var dtd = $.Deferred(); //在函数内部，新建一个Deferred对象
    var tasks = function(){
        alert("执行完毕！");
        dtd.resolve(); // 改变Deferred对象的执行状态
    };
    setTimeout(tasks,5000);
    return dtd.promise(); // 返回promise对象
};
$.when(wait())
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

我们会发现 alert(“哈哈，成功了！”) 会立即执行，“执行完毕”却需要5秒后才弹出来。
但是如果我们 wait 函数最后是 return dtd.promise() 这里 d.resolve() 就会报错了，因为对象 d 不存在 resolve() 方法。
同样如果我们把代码改成:

```
var dtd = $.Deferred(); // 新建一个deferred对象
var wait = function(dtd){
    var tasks = function(){
        alert("执行完毕！");
        dtd.resolve(); // 改变deferred对象的执行状态
    };
    setTimeout(tasks,5000);
    return dtd.promise();
};
dtd.resolve();
$.when( wait(dtd))
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

我们也可以发现 alert(“哈哈，成功了！”) 会立即执行，因为 dtd 这个 deferred 对象在被传入 wait 之前，已经被 resolve() 了，而 deferred 对象一旦被 resolve 或者 reject 之后，状态是不会改变的。
然后我们再把 $.wait 这块的代码改成:

```
$.when( wait(dtd))
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
dtd.resolve();
```

我们也会发现 alert(“哈哈，成功了！”); 被立即执行，虽然 wait(dtd) 执行的时候， dtd 还没有被 resolve，而且 wait 方法返回的是 dtd.promise(), 但是 dtd 这个原始的 deferred 对象是暴露在外面的，我们还是可以从外面改变它的状态。
于是，如果我们真的不想让其他代码能改变 wait 方法内部的 deferred 对象的状态，那我们应该写成这样:

```
var wait = function(){
    var dtd = $.Deferred(); // 新建一个deferred对象
    var tasks = function(){
        alert("执行完毕！");
        dtd.resolve(); // 改变deferred对象的执行状态
    };
    setTimeout(tasks,5000);
    return dtd.promise();
};
$.when( wait())
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

也就是不要把 deferred 直接暴露出来，最后返回 deferred.promise() ，让其他地方的代码只能添加 handler 。

八、普通操作的回调函数接口（中）

另一种防止执行状态被外部改变的方法，是使用deferred对象的建构函数$.Deferred()。
这时，wait函数还是保持不变，我们直接把它传入$.Deferred()：

```
$.Deferred(wait)
.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
```

jQuery规定，$.Deferred()可以接受一个函数名（注意，是函数名）作为参数，$.Deferred()所生成的deferred对象将作为这个函数的默认参数。

九、普通操作的回调函数接口（下）

除了上面两种方法以外，我们还可以直接在wait对象上部署deferred接口。

```
var dtd = $.Deferred(); // 生成Deferred对象
var wait = function(dtd){
    var tasks = function(){
        alert("执行完毕！");
        dtd.resolve(); // 改变Deferred对象的执行状态
    };
    setTimeout(tasks,5000);
};
dtd.promise(wait);
wait.done(function(){ alert("哈哈，成功了！"); })
.fail(function(){ alert("出错啦！"); });
wait(dtd);
```

这里的关键是dtd.promise(wait)这一行，它的作用就是在wait对象上部署Deferred接口。正是因为有了这一行，后面才能直接在wait上面调用done()和fail()。

十、小结：deferred对象的方法

前面已经讲到了deferred对象的多种方法，下面做一个总结：

1. $.Deferred() 生成一个deferred对象。
2. deferred.done() 指定操作成功时的回调函数
3. deferred.fail() 指定操作失败时的回调函数
4. deferred.promise() 没有参数时，返回一个新的deferred对象，该对象的运行状态无法被改变；接受参数时，作用为在参数对象上部署deferred接口。
5. deferred.resolve() 手动改变deferred对象的运行状态为"已完成"，从而立即触发done()方法。
6. deferred.reject() 这个方法与deferred.resolve()正好相反，调用后将deferred对象的运行状态变为"已失败"，从而立即触发fail()方法。
7. $.when() 为多个操作指定回调函数。
   除了这些方法以外，deferred对象还有二个重要方法，上面的教程中没有涉及到。
8. deferred.then() 有时为了省事，可以把done()和fail()合在一起写，这就是then()方法。

   ```
   $.when($.ajax( "/main.php" ))
   .then(successFunc, failureFunc );
   ```

   如果then()有两个参数，那么第一个参数是done()方法的回调函数，第二个参数是fail()方法的回调方法。如果then()只有一个参数，那么等同于done()。
9. deferred.always() 这个方法也是用来指定回调函数的，它的作用是，不管调用的是deferred.resolve()还是deferred.reject()，最后总是执行。

   ```
   $.ajax( "test.html" )
   .always( function() { alert("已执行！");} );
   ```

promise模式

promise模式，它代表了一种可能会长时间运行而且不一定必须完整的操作的结果。这种模式不会阻塞和等待长时间的操作完成，而是返

回一个代表了承诺的（promised）结果的对象。

promise模式通常会实现一种称为then的方法，用来注册状态变化时对应的回调函数。

promise模式在任何时刻都处于以下三种状态之一：未完成（unfulfilled）、已完成（resolved）和拒绝（rejected）。

从零开始构建一个promise模式的框架

首先需要一些对象来存储promise

```
var Promise = function () {
    /* initialize promise */
};
```

接下来，定义then方法，接受两个参数用于处理完成和拒绝状态。

```
Promise.prototype.then = function (onResolved, onRejected) {
    /* invoke handlers based upon state transition */
};
```

同时还需要两个方法来执行从未完成到已完成和从未完成到拒绝的状态转变。

```
Promise.prototype.resolve = function (value) {
    /* move from unfulfilled to resolved */
};

Promise.prototype.reject = function (error) {
    /* move from unfulfilled to rejected */
};
```

创建一个方法来发送Ajax请求并将其封装在promise中。这个promise对象分别在xhr.onload和xhr.onerror中指定了完成和拒绝状态的转变过程，请注意searchTwitter函数返回的正是promise对象。然后，在loadTweets中，使用then方法设置完成和拒绝状态对应的回调函数。

```
function searchTwitter(term) {

    var url, xhr, results, promise;
    url = 'http://search.twitter.com/search.json?rpp=100&q=' + term;
    promise = new Promise();
    xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);

    xhr.onload = function (e) {
        if (this.status === 200) {
            results = JSON.parse(this.responseText);
            promise.resolve(results);
        }
    };

    xhr.onerror = function (e) {
        promise.reject(e);
    };

    xhr.send();
    return promise;
}

function handleError(error) {
    /* handle the error */
}

function concatResults() {
    /* order tweets by date */
}

function loadTweets() {
    var container = document.getElementById('container');
    searchTwitter('#IE10').then(function (data) {
        data.results.forEach(function (tweet) {
            var el = document.createElement('li');
            el.innerText = tweet.text;
            container.appendChild(el);
        });
    }, handleError);
}
```

到目前为止，我们可以把promise模式应用于单个Ajax请求，似乎还体现不出promise的优势来。下面来看看多个Ajax请求的并发协作。此时，我们需要另一个方法when来存储准备调用的promise对象。一旦某个promise从未完成状态转化为完成或者拒绝状态，then方法里对应的处理函数就会被调用。when方法在需要等待所有操作都完成的时候至关重要。

```
Promise.when = function () {
    /* handle promises arguments and queue each */
};
```

以获取IE10和IE9两块内容的场景为例，我们可以这样来写代码：

```
var container, promise1, promise2;
container = document.getElementById('container');
promise1 = searchTwitter('#IE10');
promise2 = searchTwitter('#IE9');
Promise.when(promise1, promise2).then(function (data1, data2) {
    /* Reshuffle due to date */
    var totalResults = concatResults(data1.results, data2.results);
    totalResults.forEach(function (tweet) {
        var el = document.createElement('li');
        el.innerText = tweet.text;
        container.appendChild(el);
    });
}, handleError);
```

ES6 javascript promise

例：男神向女神求婚，需要经过女神的爸爸、大伯以及大姑的意思，他们全部都认了，女神再考虑考虑。

男神求婚历程的传统JS实现

```
// 男神的各项参数
var NanShen = {
    "身高": 180,
    "体重": 80,
    "年薪": "200K",
    request: function(obj) {
        // 成功与否随机决定
        // 执行成功的概率为80%
        if (Math.random() > 0.2) {
            obj.success();
        } else {
            obj.error();
        }
    }
};

var Request = function(names, success) {
    var index = 0, first = 0;
    var request = function() {
        if (names[index]) {
            NanShen.request({
                name: names[index],
                success: function() {
                    first = 0;
                    console.log("成功拿下" + names[index]);
                    index++;
                    request();
                },
                error: function() {
                    if (first == 1) {
                        console.log("依旧没能拿下" + names[index] + "，求婚失败");
                        return;
                    } else {
                        console.log("没能拿下" + names[index] + "，再试一次");
                    }
                    first = 1;
                    request();
                }
            });
        } else {
            success();
        }
    };

    request();
};

Request(["岳父", "大伯", "大姑"], function() {
    NanShen.request({
        name: "女神",
        success: function() {
            console.log("女神同意，求婚成功！");
        },
        error: function() {
            console.log("女神不同意，求婚失败！");
        }
    });
});
```

男神求婚历程的Promise实现

```
// 男神的各项参数
var NanShen = {
    "身高": 180,
    "体重": 80,
    "年薪": "200K",
    request: function(obj) {
        // 成功与否随机决定
        // 执行成功的概率为80%
        if (Math.random() > 0.2) {
            obj.success();
        } else {
            obj.error();
        }
    }
};

var Request = function(name) {
    return new Promise(function(resolve, reject) {
        var failed = 0, request = function() {
            NanShen.request({
                name: name,
                success: function() {
                    console.log(name + "攻略成功！");
                    failed = 0;
                    resolve();
                },
                error: function() {
                    if (failed == 0) {
                        console.log("第一次攻略" + name + "失败，重试一次！");
                        failed = 1;
                        // 重新攻略一次
                        request();
                    } else {
                        console.log("依然没有拿下" + name + "，求婚失败！");
                        reject();
                    }
                }
            });
        };

        request();
    });
};

Request("岳父")                                 // 搞定岳父，然后...
.then(function() { return Request("大伯"); })  // 搞定大伯，然后...
.then(function() { return Request("大姑"); })  // 搞定大姑，然后...
.then(function() {                             // 长辈们全部KO后，攻略女神
    NanShen.request({
        name: "女神",
        success: function() {
            console.log("女神同意，求婚成功！");
        },
        error: function() {
            console.log("女神不同意，求婚失败！");
        }
    });
});
```


参考文档:
JavaScript异步编程的Promise模式
http://www.infoq.com/cn/news/2011/09/js-promise/
NodeJS的异步编程风格
http://www.infoq.com/cn/news/2011/09/nodejs-async-code
Javascript异步编程的4种方法
http://www.ruanyifeng.com/blog/2012/12/asynchronous%EF%BC%BFjavascript.html
jQuery的deferred对象详解
http://www.ruanyifeng.com/blog/2011/08/a_detailed_explanation_of_jquery_deferred_object.html jQuery deferred 对象的 promise 方法
http://blog.allenm.me/2012/01/jquery_deferred_promise_method/
ES6 JavaScript Promise的感性认知
http://www.zhangxinxu.com/wordpress/2014/02/es6-javascript-promise-%E6%84%9F%E6%80%A7%E8%AE%A4%E7%9F%A5/

The Evolution of Asynchronous JavaScript
https://blog.risingstack.com/asynchronous-javascript/
Understanding the Publish/Subscribe Pattern for Greater JavaScript Scalability
https://msdn.microsoft.com/en-us/magazine/hh201955.aspx
http://api.jquery.com/category/deferred-object/

---

javascript: throttle和debounce

throttle

throttle就是函数节流的意思。再说的通俗一点就是函数调用的频度控制器，是连续执行时间间隔控制。
throttle形像的比喻是水龙头或机枪，你可以控制它的流量或频率。
`throttle 的关注点是连续的执行间隔时间。`
主要应用的场景比如：

- 鼠标移动，mousemove 事件
- DOM 元素动态定位，window对象的 resize 和 scroll 事件

函数接口

```
/**
* 频率控制 返回函数连续调用时，action 执行频率限定为 次 / delay
* @param delay  {number}    延迟时间，单位毫秒
* @param action {function}  请求关联函数，实际应用需要调用的函数
* @param tail?  {bool}      是否在尾部用定时器补齐调用
* @return {function}    返回客户调用函数
*/
throttle(delay,action,tail?)
```

样例代码

```
// ajaxQuery 将在停止输入 250 毫秒后执行
$('#autocomplete').addEventListener('keyup',debounce(250,function() {
    ajaxQuery(this.value,renderUI);
},true))
// 当窗口大小改变时，以 50 毫秒一次的频率为单位执行定位函数 position
window.addEventListener('resize',throttle(50,position,true) );
```

debounce

debounce是空闲时间必须大于或等于 一定值的时候，才会执行调用方法。
形像的比喻是橡皮球。如果手指按住橡皮球不放，它就一直受力，不能反弹起来，直到松手。
`debounce 的关注点是空闲的间隔时间。`
debounce主要应用的场景比如：

- 文本输入keydown 事件，keyup 事件，例如做autocomplete

函数接口

```
/**
* 空闲控制 返回函数连续调用时，空闲时间必须大于或等于 idle，action 才会执行
* @param idle   {number}    空闲时间，单位毫秒
* @param action {function}  请求关联函数，实际应用需要调用的函数
* @param tail?  {bool}      是否在尾部执行
* @return {function}    返回客户调用函数
*/
debounce(idle,action,tail?)
```

样例代码

```
// ajaxQuery 将在停止输入 250 毫秒后执行
$('#autocomplete').addEventListener('keyup',debounce(250,function() {
    ajaxQuery(this.value,renderUI);
},true))
```

throttle和debounce控制函数

```
/*
* 频率控制 返回函数连续调用时，fn 执行频率限定为每多少时间执行一次
* @param fn {function}  需要调用的函数
* @param delay  {number}    延迟时间，单位毫秒
* @param immediate  {bool} 给 immediate参数传递false 绑定的函数先执行，而不是delay后后执行。
* @return {function}实际调用函数
*/
var throttle = function (fn,delay, immediate, debounce) {
   var curr = +new Date(),//当前事件
       last_call = 0,
       last_exec = 0,
       timer = null,
       diff, //时间差
       context,//上下文
       args,
       exec = function () {
           last_exec = curr;
           fn.apply(context, args);
       };
   return function () {
       curr= +new Date();
       context = this,
       args = arguments,
       diff = curr - (debounce ? last_call : last_exec) - delay;
       clearTimeout(timer);
       if (debounce) {
           if (immediate) {
               timer = setTimeout(exec, delay);
           } else if (diff >= 0) {
               exec();
           }
       } else {
           if (diff >= 0) {
               exec();
           } else if (immediate) {
               timer = setTimeout(exec, -diff);
           }
       }
       last_call = curr;
   }
};

/*
* 空闲控制 返回函数连续调用时，空闲时间必须大于或等于 delay，fn 才会执行
* @param fn {function}  要调用的函数
* @param delay   {number}    空闲时间
* @param immediate  {bool} 给 immediate参数传递false 绑定的函数先执行，而不是delay后后执行。
* @return {function}实际调用函数
*/

var debounce = function (fn, delay, immediate) {
   return throttle(fn, delay, immediate, true);
};
```

jQuery throttle

Window resize

```
$(function(){
    var counter_1 = 0,
        counter_2 = 0
        last_time_1 = +new Date(),
        last_time_2 = +new Date();

    // This function is not throttled, but instead bound directly to the event.
    function resize_1() {
        var now = +new Date(),
            html = 'resize handler executed: ' + counter_1++ + ' times'+ ' (' + ( now - last_time_1 ) + 'ms since
previous execution)'+ '<br/>window dimensions: ' + $(window).width() + 'x' + $(window).height();

        last_time_1 = now;

        $('#text-resize-1').html( html );
    };

    // This function is throttled, and the new, throttled, function is bound to
    // the event. Note that in jQuery 1.4+ a reference to either the original or
    // throttled function can be passed to .unbind to unbind the function.
    function resize_2() {
        var now = +new Date(),
            html = 'throttled resize handler executed: ' + counter_2++ + ' times' + ' (' + ( now - last_time_2 ) +
'ms since previous execution)' + '<br/>window dimensions: ' + $(window).width() + 'x' + $(window).height();

        last_time_2 = now;

        $('#text-resize-2').html( html );
    };

    // Bind the not-at-all throttled handler to the resize event.
    $(window).resize( resize_1 );

    // Bind the throttled handler to the resize event.
    $(window).resize( $.throttle( 250, resize_2 ) ); // This is the line you want!
});
```

Window scroll

```
$(function(){
    var counter_1 = 0,
        counter_2 = 0
        last_time_1 = +new Date(),
        last_time_2 = +new Date();

    // This function is not throttled, but instead bound directly to the event.
    function scroll_1() {
        var now = +new Date(),
            html = 'scroll handler executed: ' + counter_1++ + ' times' + ' (' + ( now - last_time_1 ) + 'ms since
previous execution)' + '<br/>window scrollLeft: ' + $(window).scrollLeft() + ', scrollTop: ' +
(window).scrollTop();

        last_time_1 = now;
        $('#text-scroll-1').html( html );
    };

    // This function is throttled, and the new, throttled, function is bound to
    // the event. Note that in jQuery 1.4+ a reference to either the original or
    // throttled function can be passed to .unbind to unbind the function.
    function scroll_2() {
        var now = +new Date(),
            html = 'throttled scroll handler executed: ' + counter_2++ + ' times' + ' (' + ( now - last_time_2 ) +
'ms since previous execution)' + '<br/>window scrollLeft: ' + $(window).scrollLeft() + ', scrollTop: ' +
$(window).scrollTop();

        last_time_2 = now;
        $('#text-scroll-2').html( html );
    };

    // Bind the not-at-all throttled handler to the scroll event.
    $(window).scroll( scroll_1 );

    // Bind the throttled handler to the scroll event.
    $(window).scroll( $.throttle( 250, scroll_2 ) ); // This is the line you want!
});
```

jQuery debounce

autocomplete

```
$(function(){
    var default_text = $('#text-type').text(),
        text_counter_1 = 0,
        text_counter_2 = 0;
    // This function is not debounced, but instead bound directly to the event.
    function text_1() {
        var val = $(this).val(),
            html = 'Not-debounced AJAX request executed: ' + text_counter_1++ + ' times.' + ( val ? ' Text: ' +
val : '' );

        $('#text-type-1').html( html );
    };

    // This function is debounced, and the new, debounced, function is bound to
    // the event. Note that in jQuery 1.4+ a reference to either the original or
    // debounced function can be passed to .unbind to unbind the function.
    function text_2() {
        var val = $(this).val(),
            html = 'Debounced AJAX request executed: ' + text_counter_2++ + ' times.' + ( val ? ' Text: ' + val :
'' );

        $('#text-type-2').html( html );
    };

    // Bind the not-at-all debounced handler to the keyup event.
    $('input.text').keyup( text_1 );

    // Bind the debounced handler to the keyup event.
    $('input.text').keyup( $.debounce( 250, text_2 ) ); // This is the line you want!

    // Trigger the callbacks once to show some initial (zero) values.
    text_1();
    text_2();
});
```

参考文档:
http://benalman.com/code/projects/jquery-throttle-debounce/docs/files/jquery-ba-throttle-debounce-js.html
http://www.css88.com/archives/4648
http://benalman.com/code/projects/jquery-throttle-debounce/examples/throttle/
http://benalman.com/code/projects/jquery-throttle-debounce/examples/debounce/

跨域

JSONP(JSON with Padding,填充式JSON)

JSONP的格式是把标准JSON文件稍加包装在一对圆括号中，圆括号又前置一个任意字符串。这个字符串即所谓的P(Padding,填充)，由请求数据的客户端来决定。而且由于有一对圆括号，因此返回的数据在客户端可能会导致一次函数调用，或者是为某个变量赋值，取决于客户端请求中发送的填充字符串。

jsonp原理：

使用<script>标签从远程获取Javascript文件的思路，可以通过从其他服务器取得json文件，不过这需要对服务器上的json文件修改。jsonp提供的url（即动态生成的script标签的src），无论看上去是什么形式，最终生成返回的都是一段js代码。
在js中，我们直接用XMLHttpRequest请求不同域上的数据时，是不可以的。但是，在页面上引入不同域上的js脚本文件却是可以的，jsonp正是利用这个特性来实现的。

```
<script>
    function doSomething(){
        //处理获取的json数据
    }
</script>
<script src='http://localhost:8080/ky?callback=doSomething'></script>
```

或

```
<script type="text/javascript">
    function jsonpcallback(json) {
        console.log(json);
    }
    var s = document.createElement('script')
    s.src = 'http://localhost:8080/ky?callback=jsonpcallback';
    document.body.appendChild(s);
</script>
```

注：上面代码所在页面的访问地址为：http://localhost:63342/JSONP/index.html

通过script标签引入一个js文件，这个js文件载入成功后会执行我们在url参数中指定的函数，并且会把我们需要的json数据作为参数传入。所以jsonp是需要服务器端的页面进行相应的配合的。

JQuery对JSONP的封装

1. ajax方法：
   客户端

   ```
   $.ajax({
       url: 'http://localhost:8080/ky',
       dataType: 'jsonp',
       jsonpCallback: 'callBack',
       success: function (data){
           console.log(data);
       }
   });
   ```

   服务器端

   ```
   callBack({
       "name": "jsonp学习",
       "desc": "服务器端返回数据格式"
   });
   ```

   说明：该方法中写死了callback函数名(即callBack)。

2. getJSON方法：

   ```
   $.getJSON('http://localhost:8080/ky?callback=?', function(data){
       console.log(data);
   });
   ```

   说明：
   - 该方法url上必须带上callback=?
   - jquery会自动生成一个全局函数来替换callback=?中的问号，之后获取到数据后又会自动销毁，实际上就是起一个临时代理函数的作用。$.getJSON方法会自动判断是否跨域，不跨域的话，就调用普通的ajax方法；跨域的话，则会以异步加载js文件的形式来调用jsonp的回调函数。
   - 因为jQuery对该方法进行封装的时候并没有默认回调函数变量名为callback。对于jQuery中的jsonp来说，callback参数是自动添加的。默认情况下，jQuery生成的jsonp请求中callback参数是形如callback=jQuery20002355973557569086 6_1434954892929这种看似随机的名字，对应的就是success那个处理函数，所以一般不用特意处理。

3. get方法：

   ```
   $.get('http://localhost:8080/ky', function(data){
       console.log(data);
   }, 'jsonp');
   ```

   说明：该方法不需要指定callback函数名。getJSON方式实际上就是调用的get方法。

4. jQuery JSONP相关源码片段

```
getJSON: function( url, data, callback ) {
    return jQuery.get( url, data, callback, "json" );
},
getScript: function( url, callback ) {
    return jQuery.get( url, undefined, callback, "script" );
}
jQuery.each( [ "get", "post" ], function( i, method ) {
    jQuery[ method ] = function( url, data, callback, type ) {
        // shift arguments if data argument was omitted
        if ( jQuery.isFunction( data ) ) {
            type = type || callback;
            callback = data;
            data = undefined;
        }
        return jQuery.ajax({
            url: url,
            type: method,
            dataType: type,
            data: data,
            success: callback
        });
    };
});
```

跨域

只要协议、域名、端口有任何一个不同，都被当作是不同的域
常用的跨域方法：

1. **jsonp(如上所示)**
2. **通过修改document.domain来跨子域**

   场景：
   有一个页面，它的地址是http://www.example.com/a.html，在这个页面里面有一个iframe，它的src是http://example.com/b.html，很显然，这个页面与它里面的iframe框架是不同域的，所以我们是无法通过在页面中书写js代码来获取iframe中的东西的。

   页面http://www.example.com/a.html中的代码如下：

```
<script>
    function onLoad(){
        var iframe=document.getElementById('iframe');
        //这里能获取到iframe里的window对象，但该对象的属性和方法几乎不可用。
        var win=iframe.contentWindow;
        //获取不到iframe里的document对象
        var doc=win.document;
        //获取不到window对象的name属性
        var name=win.name;
    }
</script>
<iframe id="iframe" src="http://example.com/b.html" onload="onLoad()"></iframe>
```

   解决方案：
   - 只要把http://www.example.com/a.html和http://example.com/b.html这两个页面的document.domain都设成相同的域名就可以了。
   - 但要注意的是，document.domain的设置是有限制的，我们只能把document.domain设置成自身或更高一级的父域，且主域必须相同。
   - 例如：a.b.example.com 中某个文档的document.domain可以设成a.b.example.com、b.example.com 、example.com中的任意一个，但是不可以设成 c.a.b.example.com,因为这是当前域的子域，也不可以设成baidu.com,因为主域已经不相同了。

   页面http://www.example.com/a.html中的代码片段如下：

```
<script>
    //☆设置成主域
    document.domain='example.com';
    function onLoad(){
        var iframe=document.getElementById('iframe');
        var win=iframe.contentWindow;
        var doc=win.document;
        var name=win.name;
    }
</script>
<iframe id="iframe" src="http://example.com/b.html" onload="onLoad()"></iframe>
```

页面http://example.com/b.html中的代码片段如下：

```
<script>
    //☆在iframe载入的这个页面也设置document.domain，使之与主页面的document.domain相同
    document.domain='example.com';
</script>
```

说明：
不过如果你想在http://www.example.com/a.html 页面中通过ajax直接请求http://example.com/b.html 页面，即使你设置了相同的document.domain也还是不行的，所以修改document.domain的方法只适用于不同子域的框架间的交互。如果你想通过ajax的方法去与不同子域的页面交互，除了使用jsonp的方法外，还可以用一个隐藏的iframe来做一个代理。原理就是让这个iframe载入一个与你想要通过ajax获取数据的目标页面处在相同的域的页面，所以这个iframe中的页面是可以正常使用ajax去获取你要的数据的，然后就是通过我们刚刚讲得修改document.domain的方法，让我们能通过js完全控制这个iframe，这样我们就可以让iframe去发送ajax请求，然后收到的数据我们也可以获得了。

3. **使用window.name来进行跨域**

window对象有个name属性，该属性有个特征：即在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个window.name的，每个页面对window.name都有读写的权限，window.name是持久存在一个窗口载入过的所有页面中的，并不会因新页面的载入而进行重置。

例如：
页面a.html中的代码片段如下

```
<script>
    //☆设置window.name的值
    window.name="我是页面a设置的值";
    //3秒后把一个新页面b.html载入到当前的window
    setTimeout(function(){
        window.location='b.html';
    },3000);
</script>
```

页面b.html中的代码片段如下

```
<script>
    //☆读取window.name的值
    alert(window.name)
</script>
```

页面效果：
a.html页面载入后3秒，跳转到了b.html页面，弹出`我是页面a设置的值`。
说明：

- b.html页面上成功获取到了它的上一个页面a.html给window.name设置的值。
- 如果在之后所有载入的页面都没对window.name进行修改的话，那么所有这些页面获取到的window.name的值都是a.html页面设置的那个值。
- 当然，如果有需要，其中的任何一个页面都可以对window.name的值进行修改。
- 注意，window.name的值只能是字符串的形式，这个字符串的大小最大能允许2M左右甚至更大的一个容量，具体取决于不同的浏览器，但一般是够用了。

场景：
有一个www.example.com/a.html页面,需要通过a.html页面里的js来获取另一个位于不同域上的页面www.exampleother.com/b.html里的数据。
页面b.html的代码片段如下：

```
<script>
    window.name="我就是a.html想要的数据，所有可以转化成字符串来传递的数据都可以在这里使用，如果json数据";
</script>
```

页面a.html的代码片段如下：

```
<script>
    //iframe载入b.html页面后会执行的函数
    function getBData(){
        var iframe=document.getElementById("proxy");
        //这个时候a.html与iframe已经是出于同一源了，可以相互访问
        iframe.onload=function(){
            //获取iframe里的window.name,也就是b.html页面给它设置的数据
            var bData=iframe.contentWindow.name;
            alert(bData);
        }
        //这里的c.html为随便一个页面，只要与a.html同一源就行了，目的是让a.html能访问到iframe里的东西，设置成
about:blank也行。
        iframe.src="c.html";
    }
</script>
<iframe id="proxy" src="www.exampleother.com/b.html" style="display:none" onload="getBData()"></iframe>
```

原理：

- 在a.html页面中使用一个隐藏的iframe来充当一个中间人角色，由iframe去获取b.html的数据，然后a.html再去得到iframe获取到的数据。
- 充当中间人的iframe想要获取到b.html的通过window.name设置的数据，只需要把这个iframe的src设为www.exampleother.com/b.html就行了。
- 然后a.html想要得到iframe所获取到的数据，也就是想要得到iframe的window.name的值，还必须把这个iframe的src设成跟a.html页面同一个域才行，不然根据前面讲的同源策略，a.html是不能访问到iframe里的window.name属性的。这就是整个跨域过程。

4. **使用HTML5中新引进的window.postMessage方法来跨域传送数据**

   - window.postMessage(message,targetOrigin) 方法是html5新引进的特性，可以使用它来向其它的window对象发送消息，无论这个window对象是属于同源或不同源。
   - 调用postMessage方法的window对象是指要发送消息的那一个window对象，该方法的第一个参数message为要发送的消息，类型只能为字符串；第二个参数targetOrigin用来限定接收消息的那个window对象所在的域，如果不想限定域，可以使用通配符*
   - 需要接收消息的window对象，可是通过监听自身的message事件来获取传过来的消息，消息内容储存在该事件对象的data属性中。
     页面a.html的代码片段如下：

     ```
     <script>
         function onLoad(){
             var iframe=document.getElementById("iframe");
             var win=iframe.contentWindow;//获取window对象
             //☆向不同域的www.example.com/b.html页面发送消息
             win.postMessage("我是来自a页面的消息");
         }
     </script>
     <iframe id="iframe" src="www.example.com/b.html" onload="onLoad()"></iframe>
     ```

     页面b.html的代码片段如下：

     ```
     <script>
         //☆注册message事件用来接收消息
         window.onmessage=function(e){
             e=e || event;//获取事件对象
             alert(e.data);//☆通过data属性得到传递的消息
         }
     </script>
     ```

     运行a页面后的效果：页面弹出`我是来自a页面的消息`

---

jQuery

jQuery设计思想

一、选择网页元素

jQuery的基本设计思想和主要用法，就是"选择某个网页元素，然后对其进行某种操作"。这是它区别于其他Javascript库的根本特点。

使用jQuery的第一步，往往就是将一个选择表达式，放进构造函数jQuery()（简写为$），然后得到被选中的元素。

选择表达式可以是CSS选择器：

```
$(document) //选择整个文档对象
$('#myId') //选择ID为myId的网页元素
$('div.myClass') // 选择class为myClass的div元素
$('input[name=first]') // 选择name属性等于first的input元素
```

也可以是jQuery特有的表达式：

```
$('a:first') //选择网页中第一个a元素
$('tr:odd') //选择表格的奇数行
$('#myForm :input') // 选择表单中的input元素
$('div:visible') //选择可见的div元素
$('div:gt(2)') // 选择所有的div元素，除了前三个
$('div:animated') // 选择当前处于动画状态的div元素
```

二、改变结果集

jQuery设计思想之二，就是提供各种强大的过滤器，对结果集进行筛选，缩小选择结果。

```
$('div').has('p'); // 选择包含p元素的div元素
$('div').not('.myClass'); //选择class不等于myClass的div元素
$('div').filter('.myClass'); //选择class等于myClass的div元素
$('div').first(); //选择第1个div元素
$('div').eq(5); //选择第6个div元素
```

有时候，我们需要从结果集出发，移动到附近的相关元素，jQuery也提供了在DOM树上的移动方法：

```
$('div').next('p'); //选择div元素后面的第一个p元素
$('div').parent(); //选择div元素的父元素
$('div').closest('form'); //选择离div最近的那个form父元素
$('div').children(); //选择div的所有子元素
$('div').siblings(); //选择div的同级元素
```

三、链式操作

jQuery设计思想之三，就是最终选中网页元素以后，可以对它进行一系列操作，并且所有操作可以连接在一起，以链条的形式写出来，比如：

```
$('div').find('h3').eq(2).html('Hello');
```

分解开来，就是下面这样：

```
$('div') //找到div元素
.find('h3') //选择其中的h3元素
.eq(2) //选择第3个h3元素
.html('Hello'); //将它的内容改为Hello
```

这是jQuery最令人称道、最方便的特点。它的原理在于每一步的jQuery操作，返回的都是一个jQuery对象，所以不同操作可以连在一起。

jQuery还提供了.end()方法，使得结果集可以后退一步：

```
$('div')
.find('h3')
.eq(2)
.html('Hello')
.end() //退回到选中所有的h3元素的那一步
.eq(0) //选中第一个h3元素
.html('World'); //将它的内容改为World
```

四、元素的操作：取值和赋值

操作网页元素，最常见的需求是取得它们的值，或者对它们进行赋值。

jQuery设计思想之四，就是使用同一个函数，来完成取值（getter）和赋值（setter），即"取值器"与"赋值器"合一。到底是取值还是赋值，由函数的参数决定。

```
$('h1').html(); //html()没有参数，表示取出h1的值
$('h1').html('Hello'); //html()有参数Hello，表示对h1进行赋值
```

常见的取值和赋值函数如下：

```
.html() 取出或设置html内容
.text() 取出或设置text内容
.attr() 取出或设置某个属性的值
.width() 取出或设置某个元素的宽度
.height() 取出或设置某个元素的高度
.val() 取出某个表单元素的值
```

需要注意的是，如果结果集包含多个元素，那么赋值的时候，将对其中所有的元素赋值；取值的时候，则是只取出第一个元素的值.

五、元素的操作：移动

jQuery设计思想之五，就是提供两组方法，来操作元素在网页中的位置移动。一组方法是直接移动该元素，另一组方法是移动其他元素，使得目标元素达到我们想要的位置。

假定我们选中了一个div元素，需要把它移动到p元素后面。

第一种方法是使用.insertAfter()，把div元素移动p元素后面：

```
$('div').insertAfter($('p'));
```

第二种方法是使用.after()，把p元素加到div元素前面：

```
$('p').after($('div'));
```

表面上看，这两种方法的效果是一样的，唯一的不同似乎只是操作视角的不同。但是实际上，它们有一个重大差别，那就是返回的元素不一样。第一种方法返回div元素，第二种方法返回p元素。你可以根据需要，选择到底使用哪一种方法。

使用这种模式的操作方法，一共有四对：

```
.insertAfter()和.after()：在现存元素的外部，从后面插入元素
.insertBefore()和.before()：在现存元素的外部，从前面插入元素
.appendTo()和.append()：在现存元素的内部，从后面插入元素
.prependTo()和.prepend()：在现存元素的内部，从前面插入元素
```

六、元素的操作：复制、删除和创建

除了元素的位置移动之外，jQuery还提供其他几种操作元素的重要方法。
复制元素使用.clone()。
删除元素使用.remove()和.detach()。两者的区别在于，前者不保留被删除元素的事件，后者保留，有利于重新插入文档时使用。
清空元素内容（但是不删除该元素）使用.empty()。
创建新元素的方法非常简单，只要把新元素直接传入jQuery的构造函数就行了：

```
$('<p>Hello</p>');
$('<li class="new">new list item</li>');
$('ul').append('<li>list item</li>');
```

七、工具方法

jQuery设计思想之六：除了对选中的元素进行操作以外，还提供一些与元素无关的工具方法（utility）。不必选中元素，就可以直接使用这些方法。

如果你懂得Javascript语言的继承原理，那么就能理解工具方法的实质。它是定义在jQuery构造函数上的方法，即jQuery.method()，所以可以直接使用。而那些操作元素的方法，是定义在构造函数的prototype对象上的方法，即jQuery.prototype.method()，所以必须生成实例（即选中元素）后使用。如果不理解这种区别，问题也不大，只要把工具方法理解成，是像javascript原生函数那样，可以直接使用的方法就行了。

常用的工具方法有以下几种：

```
$.trim() 去除字符串两端的空格。
$.each() 遍历一个数组或对象。
$.inArray() 返回一个值在数组中的索引位置。如果该值不在数组中，则返回-1。
$.grep() 返回数组中符合某种标准的元素。
$.extend() 将多个对象，合并到第一个对象。
$.makeArray() 将对象转化为数组。
$.type() 判断对象的类别（函数对象、日期对象、数组对象、正则对象等等）。
$.isArray() 判断某个参数是否为数组。
$.isEmptyObject() 判断某个对象是否为空（不含有任何属性）。
$.isFunction() 判断某个参数是否为函数。
$.isPlainObject() 判断某个参数是否为用"{}"或"new Object"建立的对象。
$.support() 判断浏览器是否支持某个特性。
```

八、事件操作

jQuery设计思想之七，就是把事件直接绑定在网页元素之上。

```
$('p').click(function(){
    alert('Hello');
});
```

目前，jQuery主要支持以下事件：

```
.blur() 表单元素失去焦点。
.change() 表单元素的值发生变化
.click() 鼠标单击
.dblclick() 鼠标双击
.focus() 表单元素获得焦点
.focusin() 子元素获得焦点
.focusout() 子元素失去焦点
.hover() 同时为mouseenter和mouseleave事件指定处理函数
.keydown() 按下键盘（长时间按键，只返回一个事件）
.keypress() 按下键盘（长时间按键，将返回多个事件）
.keyup() 松开键盘
.load() 元素加载完毕
.mousedown() 按下鼠标
.mouseenter() 鼠标进入（进入子元素不触发）
.mouseleave() 鼠标离开（离开子元素不触发）
.mousemove() 鼠标在元素内部移动
.mouseout() 鼠标离开（离开子元素也触发）
.mouseover() 鼠标进入（进入子元素也触发）
.mouseup() 松开鼠标
.ready() DOM加载完成
.resize() 浏览器窗口的大小发生改变
.scroll() 滚动条的位置发生变化
.select() 用户选中文本框中的内容
.submit() 用户递交表单
.toggle() 根据鼠标点击的次数，依次运行多个函数
.unload() 用户离开页面
```

以上这些事件在jQuery内部，都是.bind()的便捷方式。使用.bind()可以更灵活地控制事件，比如为多个事件绑定同一个函数：

```
$('input').bind(
    'click change', //同时绑定click和change事件
    function() {
        alert('Hello');
    }
);
```

有时，你只想让事件运行一次，这时可以使用.one()方法。

```
$("p").one("click", function() {
    alert("Hello"); //只运行一次，以后的点击不会运行
});
```

.unbind()用来解除事件绑定。

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

所有的事件处理函数，都可以接受一个事件对象（event object）作为参数，比如下面例子中的e：

```
$("p").click(function(e) {
    alert(e.type); // "click"
});
```

这个事件对象有一些很有用的属性和方法：

```
event.pageX 事件发生时，鼠标距离网页左上角的水平距离
event.pageY 事件发生时，鼠标距离网页左上角的垂直距离
event.type 事件的类型（比如click）
event.which 按下了哪一个键
event.data 在事件对象上绑定数据，然后传入事件处理函数
event.target 事件针对的网页元素
event.preventDefault() 阻止事件的默认行为（比如点击链接，会自动打开新页面）
event.stopPropagation() 停止事件向上层元素冒泡
```

在事件处理函数中，可以用this关键字，返回事件针对的DOM元素：

```
$('a').click(function(e) {
    if ($(this).attr('href').match('evil')) { //如果确认为有害链接
        e.preventDefault(); //阻止打开
        $(this).addClass('evil'); //加上表示有害的class
    }
});
```

有两种方法，可以自动触发一个事件。一种是直接使用事件函数，另一种是使用.trigger()或.triggerHandler()。

```
$('a').click();
$('a').trigger('click');
```

九、特殊效果

最后，jQuery允许对象呈现某些特殊效果。

```
$('h1').show(); //展现一个h1标题
```

常用的特殊效果如下：

```
.fadeIn() 淡入
.fadeOut() 淡出
.fadeTo() 调整透明度
.hide() 隐藏元素
.show() 显示元素
.slideDown() 向下展开
.slideUp() 向上卷起
.slideToggle() 依次展开或卷起某个元素
.toggle() 依次展示或隐藏某个元素
```

除了.show()和.hide()，所有其他特效的默认执行时间都是400ms（毫秒），但是你可以改变这个设置。

```
$('h1').fadeIn(300); // 300毫秒内淡入
$('h1').fadeOut('slow'); // 缓慢地淡出
```

在特效结束后，可以指定执行某个函数。

```
$('p').fadeOut(300, function() { $(this).remove(); });
```

更复杂的特效，可以用.animate()自定义。

```
$('div').animate(
    {
        left : "+=50", //不断右移
        opacity : 0.25 //指定透明度

    },
    300, // 持续时间
    function() { alert('done!'); } //回调函数
);
```

.stop()和.delay()用来停止或延缓特效的执行。
$.fx.off如果设置为true，则关闭所有网页特效。

jQuery最佳实践

1. 使用最新版本的jQuery

新版本会改进性能，还有很多新功能。

2. 用对选择器

在jQuery中，你可以用多种选择器，选择同一个网页元素。每种选择器的性能是不一样的，你应该了解它们的性能差异。

1. 最快的选择器：id选择器和元素标签选择器

   举例来说，下面的语句性能最佳：

   ```
   $('#id')
   $('form')
   $('input')
   ```

   遇到这些选择器的时候，jQuery内部会自动调用浏览器的原生方法（比如getElementById()），所以它们的执行速度快。

2. 较慢的选择器：class选择器

   $('.className')的性能，取决于不同的浏览器。

   Firefox、Safari、Chrome、Opera浏览器，都有原生方法getElementByClassName()，所以速度并不慢。但是，IE5-IE8都没有部署这个方法，所以这个选择器在IE中会相当慢。

3. 最慢的选择器：伪类选择器和属性选择器

   先来看例子。找出网页中所有的隐藏元素，就要用到伪类选择器：

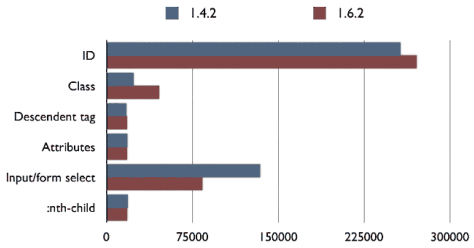
```
$(':hidden')
```

属性选择器的例子则是：

```
$('[attribute=value]')
```

这两种语句是最慢的，因为浏览器没有针对它们的原生方法。但是，一些浏览器的新版本，增加了querySelector()和querySelectorAll()方法，因此会使这类选择器的性能有大幅提高。

最后是不同选择器的性能比较图。



可以看到，ID选择器遥遥领先，然后是标签选择器，第三是Class选择器，其他选择器都非常慢。

3. 理解子元素和父元素的关系

下面六个选择器，都是从父元素中选择子元素。你知道哪个速度最快，哪个速度最慢吗？

```
$('.child', $parent)
$parent.find('.child')
$parent.children('.child')
$('#parent > .child')
$('#parent .child')
$('.child', $('#parent'))
```

我们一句句来看。

1. $('.child', $parent)

   这条语句的意思是，给定一个DOM对象，然后从中选择一个子元素。jQuery会自动把这条语句转成$.parent.find('child')，这会导致一定的性能损失。它比最快的形式慢了5%-10%。

2. $parent.find('.child')

   这条是最快的语句。.find()方法会调用浏览器的原生方法（getElementById，getElementByName，getElementByTagName等等），所以速度较快。

3. $parent.children('.child')

   这条语句在jQuery内部，会使用$.sibling()和javascript的nextSibling()方法，一个个遍历节点。它比最快的形式大约慢50%。

4. $('#parent > .child')

   jQuery内部使用Sizzle引擎，处理各种选择器。Sizzle引擎的选择顺序是从右到左，所以这条语句是先选.child，然后再一个个过滤出父元素#parent，这导致它比最快的形式大约慢70%。

5. $('#parent .child')

   这条语句与上一条是同样的情况。但是，上一条只选择直接的子元素，这一条可以于选择多级子元素，所以它的速度更慢，大概比最快的形式慢了77%。

6. $('.child', $('#parent'))

   jQuery内部会将这条语句转成$('#parent').find('.child')，比最快的形式慢了23%。

所以，最佳选择是$parent.find('.child')。而且，由于$parent往往在前面的操作已经生成，jQuery会进行缓存，所以进一步加快了执行速度。

4. 不要过度使用jQuery

jQuery速度再快，也无法与原生的javascript方法相比。所以有原生方法可以使用的场合，尽量避免使用jQuery。

以最简单的选择器为例，document.getElementById("foo")要比$("#foo")快10多倍。

再来看一个例子，为a元素绑定一个处理点击事件的函数：

```
$('a').click(function(){
    alert($(this).attr('id'));
});
```

这段代码的意思是，点击a元素后，弹出该元素的id属性。为了获取这个属性，必须连续两次调用jQuery，第一次是$(this)，第二次是attr('id')。

事实上，这种处理完全不必要。更正确的写法是，直接采用javascript原生方法，调用this.id：

```
$('a').click(function(){
    alert(this.id);
});
```

根据测试，this.id的速度比$(this).attr('id')快了20多倍。

5. 做好缓存

选中某一个网页元素，是开销很大的步骤。所以，使用选择器的次数应该越少越好，并且尽可能缓存选中的结果，便于以后反复使用。

比如，下面这样的写法就是糟糕的写法：

```
jQuery('#top').find('p.classA');
jQuery('#top').find('p.classB');
```

更好的写法是：

```
var cached = jQuery('#top');
cached.find('p.classA');
cached.find('p.classB');
```

根据测试，缓存比不缓存，快了2-3倍。

6. 使用链式写法

jQuery的一大特点，就是允许使用链式写法。

```
$('div').find('h3').eq(2).html('Hello');
```

采用链式写法时，jQuery自动缓存每一步的结果，因此比非链式写法要快。根据测试，链式写法比（不使用缓存的）非链式写法，大约快了25%。

7. 事件的委托处理（Event Delegation）

javascript的事件模型，采用"冒泡"模式，也就是说，子元素的事件会逐级向上"冒泡"，成为父元素的事件。

利用这一点，可以大大简化事件的绑定。比如，有一个表格（table元素），里面有100个格子（td元素），现在要求在每个格子上面绑定一个点击事件（click），请问是否需要将下面的命令执行100次？

```
$("td").on("click", function(){
    $(this).toggleClass("click");
});
```

回答是不需要，我们只要把这个事件绑定在table元素上面就可以了，因为td元素发生点击事件之后，这个事件会"冒泡"到父元素table上面，从而被监听到。

因此，这个事件只需要在父元素绑定1次即可，而不需要在子元素上绑定100次，从而大大提高性能。这就叫事件的"委托处理"，也就是子元素"委托"父元素处理这个事件。

```
$("table").on("click", "td", function(){
    $(this).toggleClass("click");
});
```

更好的写法，则是把事件绑定在document对象上面。

```
$(document).on("click", "td", function(){
    $(this).toggleClass("click");
});
```

如果要取消事件的绑定，就使用off()方法。

```
$(document).off("click", "td");
```

8. 少改动DOM结构

1. 改动DOM结构开销很大，因此不要频繁使用.append()、.insertBefore()和.insetAfter()这样的方法。

   如果要插入多个元素，就先把它们合并，然后再一次性插入。根据测试，合并插入比不合并插入，快了将近10倍。

2. 如果你要对一个DOM元素进行大量处理，应该先用.detach()方法，把这个元素从DOM中取出来，处理完毕以后，再重新插回文档。根据测试，使用.detach()方法比不使用时，快了60%。

3. 如果你要在DOM元素上储存数据，不要写成下面这样：

   ```
   var elem = $('#elem');
   elem.data(key,value);
   ```

   而要写成

   ```
   var elem = $('#elem');
   $.data(elem[0],key,value);
   ```

   根据测试，后一种写法要比前一种写法，快了将近10倍。因为elem.data()方法是定义在jQuery函数的prototype对象上面的，而$.data()方法是定义jQuery函数上面的，调用的时候不从复杂的jQuery对象上调用，所以速度快得多。

4. 插入html代码的时候，浏览器原生的innterHTML()方法比jQuery对象的html()更快。

9. 正确处理循环

循环总是一种比较耗时的操作，如果可以使用复杂的选择器直接选中元素，就不要使用循环，去一个个辨认元素。

javascript原生循环方法for和while，要比jQuery的.each()方法快，应该优先使用原生方法。

10. 尽量少生成jQuery对象

每当你使用一次选择器（比如$('#id')），就会生成一个jQuery对象。jQuery对象是一个很庞大的对象，带有很多属性和方法，会占用不少资源。所以，尽量少生成jQuery对象。

举例来说，许多jQuery方法都有两个版本，一个是供jQuery对象使用的版本，另一个是供jQuery函数使用的版本。下面两个例子，都是取出一个元素的文本，使用的都是text()方法。

你既可以使用针对jQuery对象的版本：

```
var $text = $("#text");
var $ts = $text.text();
```

也可以使用针对jQuery函数的版本：

```
var $text = $("#text");
var $ts = $.text($text);
```

由于后一种针对jQuery函数的版本不通过jQuery对象操作，所以相对开销较小，速度比较快。

11. 选择作用域链最短的方法

严格地说，这一条原则对所有Javascript编程都适用，而不仅仅针对jQuery。

我们知道，Javascript的变量采用链式作用域。读取变量的时候，先在当前作用域寻找该变量，如果找不到，就前往上一层的作用域寻找该变量。这样的设计，使得读取局部变量比读取全局变量快得多。

请看下面两段代码，第一段代码是读取全局变量：

```
var a = 0;
function x(){
    a += 1;
}
```

第二段代码是读取局部变量：

```
function y(){
    var a = 0;
    a += 1;
}
```

第二段代码读取变量a的时候，不用前往上一层作用域，所以要比第一段代码快五六倍。

同理，在调用对象方法的时候，closure模式要比prototype模式更快。

prototype模式：

```
var X = function(name){ this.name = name; }
X.prototype.get_name = function() { return this.name; };
```

closure模式：

```
var Y = function(name) {
    var y = { name: name };
    return { 'get_name': function() { return y.name; } };
};
```

同样是get_name()方法，closure模式更快。

12. 使用Pub/Sub模式管理事件

当发生某个事件后，如果要连续执行多个操作，最好不要写成下面这样：

```
function doSomthing{
    doSomethingElse();
    doOneMoreThing();
}
```

而要改用事件触发的形式：

```
function doSomething{
    $.trigger("DO_SOMETHING_DONE");
}
$(document).on("DO_SOMETHING_DONE", function(){
    doSomethingElse();
});
```

还可以考虑使用deferred对象。

```
function doSomething(){
    var dfd = new $.Deferred();
    //Do something async, then...
    //dfd.resolve();
    return dfd.promise();
}
function doSomethingElse(){
    $.when(doSomething()).then(//The next thing);
}
```

CSS选择器

一、基本选择器

序号	选择器	含义
1.	*	通用元素选择器，匹配任何元素
2.	E	标签选择器，匹配所有使用E标签的元素
3.	.info	class选择器，匹配所有class属性中包含info的元素
4.	#footer	id选择器，匹配所有id属性等于footer的元素

实例：

```
* { margin:0; padding:0; }
p { font-size:2em; }
.info { background:#ff0; }
p.info { background:#ff0; }
p.info.error { color:#900; font-weight:bold; }
#info { background:#ff0; }
p#info { background:#ff0; }
```

二、多元素的组合选择器

序号	选择器	含义
5.	E,F	多元素选择器，同时匹配所有E元素或F元素，E和F之间用逗号分隔
6.	E F	后代元素选择器，匹配所有属于E元素后代的F元素，E和F之间用空格分隔
7.	E > F	子元素选择器，匹配所有E元素的子元素F
8.	E + F	毗邻元素选择器，匹配所有紧随E元素之后的同级元素F

实例：

```
div p { color:#f00; }
#nav li { display:inline; }
#nav a { font-weight:bold; }
div > strong { color:#f00; }
p + p { color:#f00; }
```

三、CSS 2.1 属性选择器

序号	选择器	含义
9.	E[att]	匹配所有具有att属性的E元素，不考虑它的值。（注意：E在此处可以省略，比如"[cheacked]"。以下同。）
10.	E[att=val]	匹配所有att属性等于"val"的E元素
11.	E[att~=val]	匹配所有att属性具有多个空格分隔的值、其中一个值等于"val"的E元素
12.	E[att|=val]	匹配所有att属性具有多个连字号分隔（hyphen-separated）的值、其中一个值以"val"开头的E元素，主要用于lang属性，比如"en"、"en-us"、"en-gb"等等

实例：

```
p[title] { color:#f00; }
div[class=error] { color:#f00; }
td[headers~=col1] { color:#f00; }
p[lang|=en] { color:#f00; }
blockquote[class=quote][cite] { color:#f00; }
```

四、CSS 2.1中的伪类

序号	选择器	含义
13.	E:first-child	匹配父元素的第一个子元素
14.	E:link	匹配所有未被点击的链接
15.	E:visited	匹配所有已被点击的链接
16.	E:active	匹配鼠标已经其上按下、还没有释放的E元素
17.	E:hover	匹配鼠标悬停其上的E元素
18.	E:focus	匹配获得当前焦点的E元素
19.	E:lang(c)	匹配lang属性等于c的E元素

实例：

```
p:first-child { font-style:italic; }
input[type=text]:focus { color:#000; background:#ffe; }
input[type=text]:focus:hover { background:#fff; }
q:lang(sv) { quotes: "\201D" "\201D" "\2019" "\2019"; }
```

五、 CSS 2.1中的伪元素

序号	选择器	含义
20.	E:first-line	匹配E元素的第一行
21.	E:first-letter	匹配E元素的第一个字母
22.	E:before	在E元素之前插入生成的内容
23.	E:after	在E元素之后插入生成的内容

实例：

```
p:first-line { font-weight:bold; color;#600; }
.preamble:first-letter { font-size:1.5em; font-weight:bold; }
.cbb:before { content:""; display:block; height:17px; width:18px;   background:url(top.png) no-repeat 0 0;
margin:0 0 0 -18px; }
a:link:after { content: " (" attr(href) ") "; }
```

六、CSS 3的同级元素通用选择器

序号	选择器	含义
24.	E ~ F	匹配任何在E元素之后的同级F元素

实例:

```
p ~ ul { background:#ff0; }
```

七、CSS 3 属性选择器

序号	选择器	含义
25.	E[att^="val"]	属性att的值以"val"开头的元素
26.	E[att$="val"]	属性att的值以"val"结尾的元素
27.	E[att*="val"]	属性att的值包含"val"字符串的元素

实例:

```
div[id^="nav"] { background:#ff0; }
```

八、CSS 3中与用户界面有关的伪类

序号	选择器	含义
28.	E:enabled	匹配表单中激活的元素
29.	E:disabled	匹配表单中禁用的元素
30.	E:checked	匹配表单中被选中的radio（单选框）或checkbox（复选框）元素
31.	E::selection	匹配用户当前选中的元素

实例:

```
input[type="text"]:disabled { background:#ddd; }
```

九、CSS 3中的结构性伪类

序号	选择器	含义
32.	E:root	匹配文档的根元素，对于HTML文档，就是HTML元素
33.	E:nth-child(n)	匹配其父元素的第n个子元素，第一个编号为1
34.	E:nth-last-child(n)	匹配其父元素的倒数第n个子元素，第一个编号为1
35.	E:nth-of-type(n)	与:nth-child()作用类似，但是仅匹配使用同种标签的元素
36.	E:nth-last-of-type(n)	与:nth-last-child() 作用类似，但是仅匹配使用同种标签的元素
37.	E:last-child	匹配父元素的最后一个子元素，等同于:nth-last-child(1)
38.	E:first-of-type	匹配父元素下使用同种标签的第一个子元素，等同于:nth-of-type(1)
39.	E:last-of-type	匹配父元素下使用同种标签的最后一个子元素，等同于:nth-last-of-type(1)
40.	E:only-child	匹配父元素下仅有的一个子元素，等同于:first-child:last-child或 :nth-child(1):nth-last-child(1)
41.	E:only-of-type	匹配父元素下使用同种标签的唯一一个子元素，等同于:first-of-type:last-of-type或 :nth-of-type(1):nth-last-of-type(1)
42.	E:empty	匹配一个不包含任何子元素的元素，注意，文本节点也被看作子元素

实例：

```
p:nth-child(3) { color:#f00; }
p:nth-child(odd) { color:#f00; }
p:nth-child(even) { color:#f00; }
p:nth-child(3n+0) { color:#f00; }
p:nth-child(3n) { color:#f00; }
tr:nth-child(2n+11) { background:#ff0; }
tr:nth-last-child(2) { background:#ff0; }
p:last-child { background:#ff0; }
p:only-child { background:#ff0; }
p:empty { background:#ff0; }
```

十、CSS 3的反选伪类

序号	选择器	含义
43.	E:not(s)	匹配不符合当前选择器的任何元素

实例：

```
:not(p) { border:1px solid #ccc; }
```

十一、CSS 3中的 :target 伪类

序号	选择器	含义
44.	E:target	匹配文档中特定"id"点击后的效果

请参看HTML DOG上关于该选择器的详细解释和实例。

参考文档：
jQuery设计思想
http://www.ruanyifeng.com/blog/2011/07/jquery_fundamentals.html
jQuery最佳实践：
http://www.ruanyifeng.com/blog/2011/08/jquery_best_practices.html
CSS选择器笔记
http://www.ruanyifeng.com/blog/2009/03/css_selectors.html
Javascript继承机制的设计思想
http://www.ruanyifeng.com/blog/2011/06/designing_ideas_of_inheritance_mechanism_in_javascript.html
http://htmldog.com/articles/suckerfish/target/

HTTPS: SSL/TLS协议的运行机制

作用

不使用SSL/TLS的HTTP通信，就是不加密的通信。所有信息明文传播，带来了三大风险。

1. 窃听风险（eavesdropping）：第三方可以获知通信内容。
2. 篡改风险（tampering）：第三方可以修改通信内容。
3. 冒充风险（pretending）：第三方可以冒充他人身份参与通信。

SSL/TLS协议是为了解决这三大风险而设计的，希望达到：

1. 所有信息都是加密传播，第三方无法窃听。
2. 具有校验机制，一旦被篡改，通信双方会立刻发现。
3. 配备身份证书，防止身份被冒充。

基本的运行过程

SSL/TLS协议的基本思路是采用公钥加密法，也就是说，客户端先向服务器端索要公钥，然后用公钥加密信息，服务器收到密文后，用自己的私钥解密。

但是，这里有两个问题。

1. 如何保证公钥不被篡改？

   解决方法：将公钥放在数字证书中。只要证书是可信的，公钥就是可信的。

2. 公钥加密计算量太大，如何减少耗用的时间？

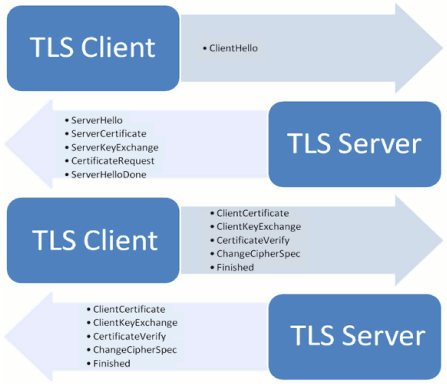
   解决方法：每一次对话（session），客户端和服务器端都生成一个"对话密钥"（session key），用它来加密信息。由于"对话密钥"是对称加密，所以运算速度非常快，而服务器公钥只用于加密"对话密钥"本身，这样就减少了加密运算的消耗时间。

SSL/TLS协议的基本过程是这样的：

1. 客户端向服务器端索要并验证公钥。
2. 双方协商生成"对话密钥"。
3. 双方采用"对话密钥"进行加密通信。

上面过程的前两步，又称为"握手阶段"（handshake）。

握手阶段的详细过程



"握手阶段"涉及四次通信，"握手阶段"的所有通信都是明文的。

1. 客户端发出请求（ClientHello）

   首先，客户端（通常是浏览器）先向服务器发出加密通信的请求，这被叫做ClientHello请求。
   在这一步，客户端主要向服务器提供以下信息。

   1. 支持的协议版本，比如TLS 1.0版。
   2. 一个客户端生成的随机数，稍后用于生成"对话密钥"。
   3. 支持的加密方法，比如RSA公钥加密。
   4. 支持的压缩方法。

   这里需要注意的是，客户端发送的信息之中不包括服务器的域名。也就是说，理论上服务器只能包含一个网站，否则会分不清应该向客户端提供哪一个网站的数字证书。这就是为什么通常一台服务器只能有一张数字证书的原因。

   对于虚拟主机的用户来说，这当然很不方便。2006年，TLS协议加入了一个Server Name Indication扩展，允许客户端向服务器提供它所请求的域名。

2. 服务器回应（SeverHello）

   服务器收到客户端请求后，向客户端发出回应，这叫做SeverHello。服务器的回应包含以下内容。

   1. 确认使用的加密通信协议版本，比如TLS 1.0版本。如果浏览器与服务器支持的版本不一致，服务器关闭加密通信。
   2. 一个服务器生成的随机数，稍后用于生成"对话密钥"。
   3. 确认使用的加密方法，比如RSA公钥加密。
   4. 服务器证书。

   除了上面这些信息，如果服务器需要确认客户端的身份，就会再包含一项请求，要求客户端提供"客户端证书"。比如，金融机构往往只允许认证客户连入自己的网络，就会向正式客户提供USB密钥，里面就包含了一张客户端证书。

3. 客户端回应

   客户端收到服务器回应以后，首先验证服务器证书。如果证书不是可信机构颁布、或者证书中的域名与实际域名不一致、或者证书已经过期，就会向访问者显示一个警告，由其选择是否还要继续通信。
   如果证书没有问题，客户端就会从证书中取出服务器的公钥。然后，向服务器发送下面三项信息。

   1. 一个随机数。该随机数用服务器公钥加密，防止被窃听。
   2. 编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送。
   3. 客户端握手结束通知，表示客户端的握手阶段已经结束。这一项同时也是前面发送的所有内容的hash值，用来供服务器校验。

   上面第一项的随机数，是整个握手阶段出现的第三个随机数，又称"pre-master key"。有了它以后，客户端和服务器就同时有了三个随机数，接着双方就用事先商定的加密方法，各自生成本次会话所用的同一把"会话密钥"。

   至于为什么一定要用三个随机数，来生成"会话密钥"，dog250解释得很好：

   > *不管是客户端还是服务器，都需要随机数，这样生成的密钥才不会每次都一样。由于SSL协议中证书是静态的，因此十分有必要引入一种随机因素来保证协商出来的密钥的随机性。 对于RSA密钥交换算法来说，pre-master-key本身就是一个随机数，再加上hello消息中的随机，三个随机数通过一个密钥导出器最终导出一个对称密钥。 pre master的存在在于SSL协议不信任每个主机都能产生完全随机的随机数，如果随机数不随机，那么pre master secret就有可能被猜出来，那么仅适用pre master secret作为密钥就不合适了，因此必须引入新的随机因素，那么客户端和服务器加上pre master secret三个随机数一同生成的密钥就不容易被猜出了，一个伪随机可能完全不随机，可是是三个伪随机就十分接近随机了，每增加一个自由度，随机性增加的可不是一。"*

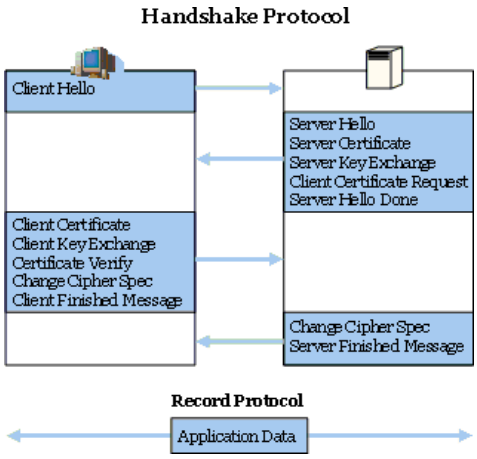
   此外，如果前一步，服务器要求客户端证书，客户端会在这一步发送证书及相关信息。

4. 服务器的最后回应

   服务器收到客户端的第三个随机数pre-master key之后，计算生成本次会话所用的"会话密钥"。然后，向客户端最后发送下面信息。

   1. 编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送。
   2. 服务器握手结束通知，表示服务器的握手阶段已经结束。这一项同时也是前面发送的所有内容的hash值，用来供客户端校验。

至此，整个握手阶段全部结束。接下来，客户端与服务器进入加密通信，就完全是使用普通的HTTP协议，只不过用"会话密钥"加密内容。



参考文档:
SSL/TLS协议运行机制的概述
http://www.ruanyifeng.com/blog/2014/02/ssl_tls.html

---

this关键字、函数的执行环境、作用域和作用域链、闭包、数据类型、prototype原型和原型链、Javascript面向对象编程

this关键字

this是Javascript语言的一个关键字。它代表函数运行时，自动生成的一个内部对象，只能在函数内部使用。 随着函数使用场合的不同，this的值会发生变化。但是有一个总的原则，那就是this指的是，调用函数的那个对象。

JavaScript 语言中的 this,由于其运行期绑定的特性，JavaScript 中的 this 含义要丰富得多，它可以是全局对象、当前对象或者任意对象，这完全取决于函数的调用方式。JavaScript 中函数的调用有以下几种方式：作为函数调用，作为对象方法调用，作为构造函数调用，和使用 apply 或 call 调用。

一、纯粹的函数调用

这是函数的最通常用法，属于全局性调用，因此this就代表全局对象Global。在浏览器中，window 就是该全局对象。

请看下面这段代码，它的运行结果是1。

```
function test(){
    this.x = 1;
    alert(this.x);
}
test(); // 1
```

为了证明this就是全局对象，我对代码做一些改变：

```
var x = 1;
function test(){
    alert(this.x);
}
test(); // 1
```

运行结果还是1。再变一下：

```
var x = 1;
function test(){
    this.x = 0;
}
test();
alert(x); //0
```

二、作为对象方法的调用

函数还可以作为某个对象的方法调用，这时this就指这个上级对象。

```
function test(){
    alert(this.x);
}
var o = {};
o.x = 1;
o.m = test;
o.m(); // 1
```

在 JavaScript 中，函数也是对象，因此函数可以作为一个对象的属性，此时该函数被称为该对象的方法，在使用这种调用方式时，this 被自然绑定到该对象。

```
var point = {
    x : 0,
    y : 0,
    moveTo : function(x, y) {
        this.x = this.x + x;
        this.y = this.y + y;
    }
};
point.moveTo(1, 1)//this 绑定到当前对象，即 point 对象
```

对于内部函数，即声明在另外一个函数体内的函数，这种绑定到全局对象的方式会产生另外一个问题。我们仍然以前面提到的 point 对象为例，这次我们希望在 moveTo 方法内定义两个函数，分别将 x，y 坐标进行平移。结果可能出乎大家意料，不仅 point 对象没有移动，反而多出两个全局变量 x，y。

```
var point = {
    x : 0,
    y : 0,
    moveTo : function(x, y) {
        // 内部函数
        var moveX = function(x) {
            this.x = x;//this 绑定到了哪里?
        };
        // 内部函数
        var moveY = function(y) {
            this.y = y;//this 绑定到了哪里?
        };
        moveX(x);
        moveY(y);
    }
};
point.moveTo(1, 1);
point.x; //==>0
point.y; //==>0
x; //==>1
y; //==>1
```

这属于 JavaScript 的设计缺陷，正确的设计方式是内部函数的 this 应该绑定到其外层函数对应的对象上，为了规避这一设计缺陷，聪明的 JavaScript 程序员想出了变量替代的方法，约定俗成，该变量一般被命名为 that。

```
var point = {
    x : 0,
    y : 0,
    moveTo : function(x, y) {
        var that = this;
        // 内部函数
        var moveX = function(x) {
            that.x = x;
        };
        // 内部函数
        var moveY = function(y) {
            that.y = y;
        }
        moveX(x);
        moveY(y);
    }
};
point.moveTo(1, 1);
point.x; //==>1
point.y; //==>1
```

三、作为构造函数调用

所谓构造函数，就是通过这个函数生成一个新对象（object）。这时，this就指这个新对象。

JavaScript 支持面向对象式编程，与主流的面向对象式编程语言不同，`JavaScript 并没有类（class）的概念，而是使用基于原型（prototype）的继承方式。相应的，JavaScript 中的构造函数也很特殊，如果不使用 new 调用，则和普通函数一样。`作为又一项约定俗成的准则，构造函数以大写字母开头，提醒调用者使用正确的方式调用。如果调用正确，this 绑定到新创建的对象上。

```
function Test(){
    this.x = 1;
}
var o = new Test();
alert(o.x); // 1
```

运行结果为1。为了表明这时this不是全局对象，我对代码做一些改变：

```
var x = 2;
function test(){
    this.x = 1;
}
var o = new test();
alert(x); //2
```

运行结果为2，表明全局变量x的值根本没变。

四、apply调用

`apply()是函数对象的一个方法，它的作用是改变函数的调用对象，它的第一个参数就表示改变后的调用这个函数的对象。因此，this指的就是这第一个参数。`

```
var x = 0;
function test(){
    alert(this.x);
}
var o={};
o.x = 1;
o.m = test;
o.m.apply(); //0
```

`apply()的参数为空时，默认调用全局对象。`因此，这时的运行结果为0，证明this指的是全局对象。

如果把最后一行代码修改为

```
o.m.apply(o); //1
```

运行结果就变成了1，证明了这时this代表的是对象o。

在 JavaScript 中函数也是对象，对象则有方法，apply 和 call 就是函数对象的方法。这两个方法异常强大，他们允许切换函数执行的上下文环境（context），即 this 绑定的对象。

```
function Point(x, y){
    this.x = x;
    this.y = y;
    this.moveTo = function(x, y){
        this.x = x;
        this.y = y;
        console.log("Point moveTo x:"+this.x+" y:"+ this.y);
    }
    console.log("Point x:"+this.x+" y:"+ this.y);
}

var p1 = new Point(0, 0);
console.log("x:"+this.x+" y:"+ this.y);
var p2 = {x: 2, y: 2};
p1.moveTo(1, 1);
p1.moveTo.apply(p2);
p1.moveTo.apply(p2, [10, 10]);
//输入结果
//Point x:0 y:0
//x:undefined y:undefined
//Point moveTo x:1 y:1
//Point moveTo x:undefined y:undefined
//Point moveTo x:10 y:10
```

在上面的例子中，我们使用构造函数生成了一个对象 p1，该对象同时具有 moveTo 方法；使用对象字面量创建了另一个对象 p2，我们看到使用 apply 可以将 p1 的方法应用到 p2 上，这时候 this 也被绑定到对象 p2 上。另一个方法 call 也具备同样功能，不同的是最后的参数不是作为一个数组统一传入，而是分开传入的。

call

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}
hello.call("Yehuda", "world") //=> Yehuda says hello world
```

Simple Function Invocation

```
function hello(thing) {
  console.log("Hello " + thing);
}
// this:
hello("world")
// desugars to:
hello.call(window, "world");
// ECMAScript 5 desugars to:
hello.call(undefined, "world");
```

a function invocation like fn(...args) is the same as fn.call(window [ES5-strict: undefined], ...args).

Member Functions

```
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this + " says hello " + thing);
  }
}
// this:
person.hello("world")//[object Object] says hello world
// desugars to this:
person.hello.call(person, "world"); //[object Object] says hello world
```

defined hello as a standalone function

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}
person = { name: "Brendan Eich" }
person.hello = hello;
// still desugars to person.hello.call(person, "world")
person.hello("world");  //输出：[object Object] says hello world
// "[object DOMWindow]world"
hello("world");//输出：[object Window] says hello world
```

Using Function.prototype.bind

```
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this.name + " says hello " + thing);
  }
}
var boundHello = function(thing) { return person.hello.call(person, thing); }
boundHello("world");
```

抽取 bind

```
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this.name + " says hello " + thing);
  }
}
var bind = function(func, thisValue) {
  return function() {
    return func.apply(thisValue, arguments);
  }
}
var boundHello = bind(person.hello, person);
boundHello("world") // "Brendan Eich says hello world"
```

ES5 introduced a new method bind on all Function objects that implements this behavior:

```
var boundHello = person.hello.bind(person);
boundHello("world") // "Brendan Eich says hello world"
```

函数的执行环境

> *JavaScript 中的函数既可以被当作普通函数执行，也可以作为对象的方法执行，这是导致 this 含义如此丰富的主要原因。一个函数被执行时，会创建一个执行环境（ExecutionContext），函数的所有的行为均发生在此执行环境中，构建该执行环境时，JavaScript 首先会创建 arguments变量，其中包含调用函数时传入的参数。接下来创建作用域链。然后初始化变量，首先初始化函数的形参表，值为 arguments变量中对应的值，如果 arguments变量中没有对应值，则该形参初始化为 undefined。如果该函数中含有内部函数，则初始化这些内部函数。如果没有，继续初始化该函数内定义的局部变量，需要注意的是此时这些变量初始化为 undefined，其赋值操作在执行环境（ExecutionContext）创建成功后，函数执行时才会执行，这点对于我们理解 JavaScript 中的变量作用域非常重要。最后为 this变量赋值，会根据函数调用方式的不同，赋给 this 全局对象，当前对象等。至此函数的执行环境（ExecutionContext）创建成功，函数开始逐行执行，所需变量均从之前构建好的执行环境（ExecutionContext）中读取。*

作用域

作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期。在JavaScript中，变量的作用域有全局作用域和局部作用域两种。

全局作用域（Global Scope）

1. 最外层函数和在最外层函数外面定义的变量拥有全局作用域，例如：

```
var authorName="山边小溪";
function doSomething(){
    var blogName="梦想天空";
    function innerSay(){
        alert(blogName);
    }
    innerSay();
}
alert(authorName); //山边小溪
alert(blogName); //脚本错误
doSomething(); //梦想天空
innerSay() //脚本错误
```

2. 所有未定义直接赋值的变量自动声明为拥有全局作用域，例如：

```
function doSomething(){
    var authorName="山边小溪";
    blogName="梦想天空";
    alert(authorName);
}
doSomething(); //山边小溪
alert(blogName); //梦想天空
alert(authorName); //脚本错误
```

3. 所有window对象的属性拥有全局作用域

一般情况下，window对象的内置属性都拥有全局作用域，例如window.name、window.location、window.top等等。

局部作用域（Local Scope）

和全局作用域相反，局部作用域一般只在固定的代码片段内可访问到。

作用域链（Scope Chain）

当一个函数创建后，它的作用域链会被创建此函数的作用域中可访问的数据对象填充。

```
function add(num1,num2) {
    var sum = num1 + num2;
    return sum;
}
```

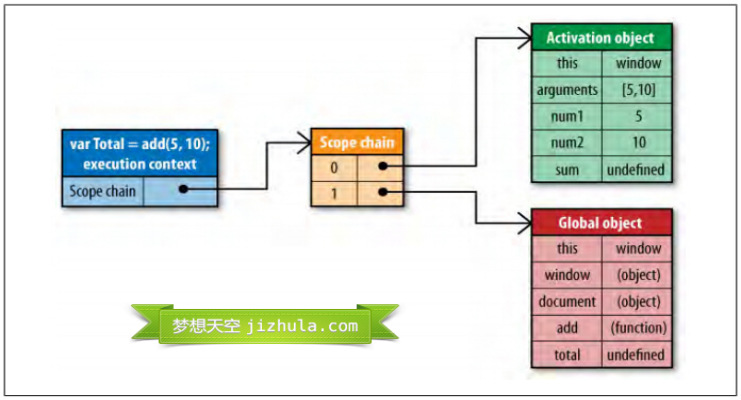
在函数add创建时，它的作用域链中会填入一个全局对象，该全局对象包含了所有全局变量。

函数add的作用域将会在执行时用到。例如执行如下代码：

```
var total = add(5,10);
```

执行此函数时会创建一个称为“运行期上下文(execution context)”的内部对象，运行期上下文定义了函数执行时的环境。每个运行期上下文都有自己的作用域链，用于标识符解析，当运行期上下文被创建时，而它的作用域链初始化为当前运行函数的[[Scope]]所包含的对象。

这些值按照它们出现在函数中的顺序被复制到运行期上下文的作用域链中。它们共同组成了一个新的对象，叫“活动对象(activation object)”，该对象包含了函数的所有局部变量、命名参数、参数集合以及this，然后此对象会被推入作用域链的前端，当运行期上下文被销毁，活动对象也随之销毁。新的作用域链如下图所示：



在函数执行过程中，每遇到一个变量，都会经历一次标识符解析过程以决定从哪里获取和存储数据。该过程从作用域链头部，也就是从活动对象开始搜索，查找同名的标识符，如果找到了就使用这个标识符对应的变量，如果没找到继续搜索作用域链中的下一个对象，如果搜索完所有对象都未找到，则认为该标识符未定义。函数执行过程中，每个标识符都要经历这样的搜索过程。

改变作用域链

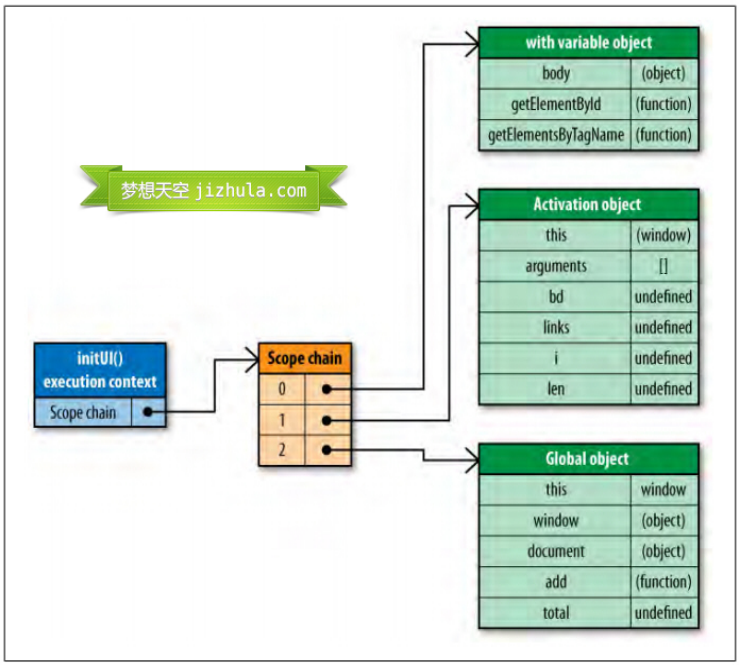
函数每次执行时对应的运行期上下文都是独一无二的，所以多次调用同一个函数就会导致创建多个运行期上下文，当函数执行完毕，执行上下文会被销毁。每一个运行期上下文都和一个作用域链关联。一般情况下，在运行期上下文运行的过程中，其作用域链只会被 with 语句和 catch 语句影响。

with语句是对象的快捷应用方式，用来避免书写重复代码。例如：

```
function initUI(){
    with(document){
        var bd=body,
            links=getElementsByTagName("a"),
            i=0,
            len=links.length;
        while(i < len){
            update(links[i++]);
        }
        getElementById("btnInit").onclick=function(){
            doSomething();
        };
    }
}
```

这里使用width语句来避免多次书写document，看上去更高效，实际上产生了性能问题。

当代码运行到with语句时，运行期上下文的作用域链临时被改变了。一个新的可变对象被创建，它包含了参数指定的对象的所有属性。这个对象将被推入作用域链的头部，这意味着函数的所有局部变量现在处于第二个作用域链对象中，因此访问代价更高了。如下图所示：



因此在程序中应避免使用with语句，在这个例子中，只要简单的把document存储在一个局部变量中就可以提升性能。

另外一个会改变作用域链的是try-catch语句中的catch语句。当try代码块中发生错误时，执行过程会跳转到catch语句，然后把异常对象推入一个可变对象并置于作用域的头部。在catch代码块内部，函数的所有局部变量将会被放在第二个作用域链对象中。示例代码：

```
try{
    doSomething();
}catch(ex){
    alert(ex.message); //作用域链在此处改变
}
```

请注意，一旦catch语句执行完毕，作用域链机会返回到之前的状态。try-catch语句在代码调试和异常处理中非常有用，因此不建议完全避免。你可以通过优化代码来减少catch语句对性能的影响。一个很好的模式是将错误委托给一个函数处理，例如：

```
try{
    doSomething();
}catch(ex){
    handleError(ex); //委托给处理器方法
}
```

优化后的代码，handleError方法是catch子句中唯一执行的代码。该函数接收异常对象作为参数，这样你可以更加灵活和统一的处理错误。由于只执行一条语句，且没有局部变量的访问，作用域链的临时改变就不会影响代码性能了。

Javascript闭包（Closure）

如何从外部读取局部变量?

出于种种原因，我们有时候需要得到函数内的局部变量。但是，前面已经说过了，正常情况下，这是办不到的，只有通过变通方法才能实现。

那就是在函数的内部，再定义一个函数。

```
function f1(){
    var n=999;
    function f2(){
        alert(n); // 999
    }
}
```

在上面的代码中，函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，f2内部的局部变

量，对f1就是不可见的。这就是Javascript语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1中的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗！

```
function f1(){
    var n=999;
    function f2(){
        alert(n);
    }
    return f2;
}
var result=f1();
result(); // 999
```

闭包的概念

闭包就是能够读取其他函数内部变量的函数。

由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成"定义在一个函数内部的函数"。

所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

闭包的用途

- 一个是可以读取函数内部的变量，
- 另一个就是让这些变量的值始终保持在内存中。

例：

```
function f1(){
    var n=999;
    nAdd=function(){n+=1}
    function f2(){
        alert(n);
    }
    return f2;
}
var result=f1();
result(); // 999
nAdd();
result(); // 1000
```

在这段代码中，result实际上就是闭包f2函数。它一共运行了两次，第一次的值是999，第二次的值是1000。这证明了，函数f1中的局部变量n一直保存在内存中，并没有在f1调用后被自动清除。

为什么会这样呢？原因就在于f1是f2的父函数，而f2被赋给了一个全局变量，这导致f2始终在内存中，而f2的存在依赖于f1，因此f1也始终在内存中，不会在调用结束后，被垃圾回收机制（garbage collection）回收。

这段代码中另一个值得注意的地方，就是"nAdd=function(){n+=1}"这一行，首先在nAdd前面没有使用var关键字，因此nAdd是一个全局变量，而不是局部变量。其次，nAdd的值是一个匿名函数（anonymous function），而这个匿名函数本身也是一个闭包，所以nAdd相当于是一个setter，可以在函数外部对函数内部的局部变量进行操作。

使用闭包的注意点

1. 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。
2. 闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

代码片段一。

```
var name = "The Window";
var object = {
    name : "My Object",
    getNameFunc : function(){
        return function(){
            return this.name;
        };
    }
};
alert(object.getNameFunc()());//The Window
```

代码片段二。

```
var name = "The Window";
var object = {
    name : "My Object",
    getNameFunc : function(){
        var that = this;
        return function(){
            return that.name;
        };
    }
};
alert(object.getNameFunc()());//My Object
```

Javascript 面向对象编程

在Javascript语言中，new命令后面跟的不是类，而是构造函数。

new运算符的缺点

用构造函数生成实例对象，有一个缺点，那就是无法共享属性和方法。

比如，在DOG对象的构造函数中，设置一个实例对象的共有属性species。

```
function DOG(name){
    this.name = name;
    this.species = '犬科';
}
```

然后，生成两个实例对象：

```
var dogA = new DOG('大毛');
var dogB = new DOG('二毛');
```

这两个对象的species属性是独立的，修改其中一个，不会影响到另一个。

```
dogA.species = '猫科';
alert(dogB.species); // 显示"犬科"，不受dogA的影响
```

每一个实例对象，都有自己的属性和方法的副本。这不仅无法做到数据共享，也是极大的资源浪费。

prototype属性的引入

prototype属性包含一个对象（以下简称"prototype对象"），所有实例对象需要共享的属性和方法，都放在这个对象里面；那些不需要共享的属性和方法，就放在构造函数里面。

`实例对象一旦创建，将自动引用prototype对象的属性和方法。`也就是说，实例对象的属性和方法，分成两种，一种是本地的，另一种是引用的。

还是以DOG构造函数为例，现在用prototype属性进行改写：

```
function DOG(name){
    this.name = name;
}
DOG.prototype = { species : '犬科' };
var dogA = new DOG('大毛');
var dogB = new DOG('二毛');
alert(dogA.species); // 犬科
alert(dogB.species); // 犬科
```

现在，species属性放在prototype对象里，是两个实例对象共享的。只要修改了prototype对象，就会同时影响到两个实例对象。

```
DOG.prototype.species = '猫科';
alert(dogA.species); // 猫科
alert(dogB.species); // 猫科
```

封装（Encapsulation）

Javascript是一种基于对象（object-based）的语言，你遇到的所有东西几乎都是对象。但是，它又不是一种真正的面向对象编程（OOP）语言，因为它的语法中没有class（类）。

那么，如果我们要把"属性"（property）和"方法"（method），封装成一个对象，甚至要从原型对象生成一个实例对象，我们应该怎么做呢？

一、生成对象的原始模式

假定我们把猫看成一个对象，它有"名字"和"颜色"两个属性。

```
var Cat = {
    name : '',
    color : ''
}
```

现在，我们需要根据这个原型对象的规格（schema），生成两个实例对象。

```
var cat1 = {}; // 创建一个空对象
cat1.name = "大毛"; // 按照原型对象的属性赋值
cat1.color = "黄色";

var cat2 = {};
cat2.name = "二毛";
cat2.color = "黑色";
```

好了，这就是最简单的封装了，把两个属性封装在一个对象里面。但是，这样的写法有两个缺点，一是如果多生成几个实例，写起来就非常麻烦；二是实例与原型之间，没有任何办法，可以看出有什么联系。

二、原始模式的改进

我们可以写一个函数，解决代码重复的问题。

```
function Cat(name,color){
    return {
        name:name,
        color:color
    }
}
```

然后生成实例对象，就等于是在调用函数：

```
var cat1 = Cat("大毛","黄色");
var cat2 = Cat("二毛","黑色");
```

这种方法的问题依然是，cat1和cat2之间没有内在的联系，不能反映出它们是同一个原型对象的实例。

三、构造函数模式

为了解决从原型对象生成实例的问题，Javascript提供了一个构造函数（Constructor）模式。

所谓"构造函数"，其实就是一个普通函数，但是内部使用了this变量。对构造函数使用new运算符，就能生成实例，并且this变量会绑定在实例对象上。

比如，猫的原型对象现在可以这样写，

```
function Cat(name,color){
    this.name=name;
    this.color=color;
}
```

我们现在就可以生成实例对象了。

```
var cat1 = new Cat("大毛","黄色");
var cat2 = new Cat("二毛","黑色");
alert(cat1.name); // 大毛
alert(cat1.color); // 黄色
```

这时cat1和cat2会自动含有一个constructor属性，指向它们的构造函数。

```
alert(cat1.constructor == Cat); //true
alert(cat2.constructor == Cat); //true
```

Javascript还提供了一个instanceof运算符，验证原型对象与实例对象之间的关系。

```
alert(cat1 instanceof Cat); //true
alert(cat2 instanceof Cat); //true
```

四、构造函数模式的问题

构造函数方法很好用，但是存在一个浪费内存的问题。

请看，我们现在为Cat对象添加一个不变的属性"type"（种类），再添加一个方法eat（吃老鼠）。那么，原型对象Cat就变成了下面这样：

```
function Cat(name,color){
    this.name = name;
    this.color = color;
    this.type = "猫科动物";
    this.eat = function(){alert("吃老鼠");};
}
```

还是采用同样的方法，生成实例：

```
var cat1 = new Cat("大毛","黄色");
var cat2 = new Cat ("二毛","黑色");
alert(cat1.type); // 猫科动物
cat1.eat(); // 吃老鼠
```

表面上好像没什么问题，但是实际上这样做，有一个很大的弊端。那就是对于每一个实例对象，type属性和eat()方法都是一模一样的内容，每一次生成一个实例，都必须为重复的内容，多占用一些内存。这样既不环保，也缺乏效率。

```
alert(cat1.eat == cat2.eat); //false
```

能不能让type属性和eat()方法在内存中只生成一次，然后所有实例都指向那个内存地址呢？回答是可以的。

五、Prototype模式

Javascript规定，每一个构造函数都有一个prototype属性，指向另一个对象。这个对象的所有属性和方法，都会被构造函数的实例继承。

这意味着，我们可以把那些不变的属性和方法，直接定义在prototype对象上。

```
function Cat(name,color){
    this.name = name;
    this.color = color;
}
Cat.prototype.type = "猫科动物";
Cat.prototype.eat = function(){alert("吃老鼠")};
```

然后，生成实例。

```
var cat1 = new Cat("大毛","黄色");
var cat2 = new Cat("二毛","黑色");
alert(cat1.type); // 猫科动物
cat1.eat(); // 吃老鼠
```

这时所有实例的type属性和eat()方法，其实都是同一个内存地址，指向prototype对象，因此就提高了运行效率。

```
alert(cat1.eat == cat2.eat); //true
```

六、Prototype模式的验证方法

为了配合prototype属性，Javascript定义了一些辅助方法，帮助我们使用它。

1. isPrototypeOf()

   这个方法用来判断，某个proptotype对象和某个实例之间的关系。

   ```
   alert(Cat.prototype.isPrototypeOf(cat1)); //true
   alert(Cat.prototype.isPrototypeOf(cat2)); //true
   ```

2. hasOwnProperty()

   每个实例对象都有一个hasOwnProperty()方法，用来判断某一个属性到底是本地属性，还是继承自prototype对象的属性。

   ```
   alert(cat1.hasOwnProperty("name")); // true
   alert(cat1.hasOwnProperty("type")); // false
   ```

3. in运算符

   in运算符可以用来判断，某个实例是否含有某个属性，不管是不是本地属性。

   ```
   alert("name" in cat1); // true
   alert("type" in cat1); // true
   ```

   in运算符还可以用来遍历某个对象的所有属性。

   ```
   for(var prop in cat1) { alert("cat1["+prop+"]="+cat1[prop]); }
   ```

构造函数的继承

对象之间的"继承"的五种方法。

比如，现在有一个"动物"对象的构造函数。

```
function Animal(){
    this.species = "动物";
}
```

还有一个"猫"对象的构造函数。

```
function Cat(name,color){
    this.name = name;
    this.color = color;
}
```

怎样才能使"猫"继承"动物"呢?

一、构造函数绑定

第一种方法也是最简单的方法，使用call或apply方法，将父对象的构造函数绑定在子对象上，即在子对象构造函数中加一行:

```
function Cat(name,color){
    Animal.apply(this, arguments);
    this.name = name;
    this.color = color;
}
var cat1 = new Cat("大毛","黄色");
alert(cat1.species); // 动物
```

二、prototype模式

第二种方法更常见，使用prototype属性。

如果"猫"的prototype对象，指向一个Animal的实例，那么所有"猫"的实例，就能继承Animal了。

```
Cat.prototype = new Animal();
Cat.prototype.constructor = Cat;
var cat1 = new Cat("大毛","黄色");
alert(cat1.species); // 动物
```

代码的第一行，我们将Cat的prototype对象指向一个Animal的实例。

```
Cat.prototype = new Animal();
```

它相当于完全删除了prototype 对象原先的值，然后赋予一个新值。但是，第二行又是什么意思呢?

```
Cat.prototype.constructor = Cat;
```

原来，任何一个prototype对象都有一个constructor属性，指向它的构造函数。如果没有"Cat.prototype = new Animal();"这一行，Cat.prototype.constructor是指向Cat的；加了这一行以后，Cat.prototype.constructor指向Animal。

```
alert(Cat.prototype.constructor == Animal); //true
```

更重要的是，每一个实例也有一个constructor属性，默认调用prototype对象的constructor属性。

```
alert(cat1.constructor == Cat.prototype.constructor); // true
```

因此，在运行"Cat.prototype = new Animal();"这一行之后，cat1.constructor也指向Animal!

```
alert(cat1.constructor == Animal); // true
```

这显然会导致继承链的紊乱（cat1明明是用构造函数Cat生成的），因此我们必须手动纠正，将Cat.prototype对象的constructor值改为Cat。这就是第二行的意思。

这是很重要的一点，编程时务必要遵守。下文都遵循这一点，即如果替换了prototype对象，

```
o.prototype = {};
```

那么，下一步必然是为新的prototype对象加上constructor属性，并将这个属性指回原来的构造函数。

```
o.prototype.constructor = o;
```

三、直接继承prototype

第三种方法是对第二种方法的改进。由于Animal对象中，不变的属性都可以直接写入Animal.prototype。所以，我们也可以让Cat()跳过Animal()，直接继承Animal.prototype。

现在，我们先将Animal对象改写：

```
function Animal(){ }
Animal.prototype.species = "动物";
```

然后，将Cat的prototype对象指向Animal的prototype对象，这样就完成了继承。

```
Cat.prototype = Animal.prototype;
Cat.prototype.constructor = Cat;
var cat1 = new Cat("大毛","黄色");
alert(cat1.species); // 动物
```

与前一种方法相比，这样做的优点是效率比较高（不用执行和建立Animal的实例了），比较省内存。缺点是 Cat.prototype和Animal.prototype现在指向了同一个对象，那么任何对Cat.prototype的修改，都会反映到Animal.prototype。

所以，上面这一段代码其实是有问题的。请看第二行

```
Cat.prototype.constructor = Cat;
```

这一句实际上把Animal.prototype对象的constructor属性也改掉了！

```
alert(Animal.prototype.constructor); // Cat
```

四、利用空对象作为中介

由于"直接继承prototype"存在上述的缺点，所以就有第四种方法，利用一个空对象作为中介。

```
var F = function(){};
F.prototype = Animal.prototype;
Cat.prototype = new F();
Cat.prototype.constructor = Cat;
```

F是空对象，所以几乎不占内存。这时，修改Cat的prototype对象，就不会影响到Animal的prototype对象。

```
alert(Animal.prototype.constructor); // Animal
```

我们将上面的方法，封装成一个函数，便于使用。

```
function extend(Child, Parent) {
    var F = function(){};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.prototype.constructor = Child;
    Child.uber = Parent.prototype;
}
```

使用的时候，方法如下

```
extend(Cat,Animal);
var cat1 = new Cat("大毛","黄色");
alert(cat1.species); // 动物
```

这个extend函数，就是YUI库如何实现继承的方法。

另外，说明一点，函数体最后一行

```
Child.uber = Parent.prototype;
```

意思是为子对象设一个uber属性，这个属性直接指向父对象的prototype属性。（uber是一个德语词，意思是"向上"、"上一层"。）这等于在子对象上打开一条通道，可以直接调用父对象的方法。这一行放在这里，只是为了实现继承的完备性，纯属备用性质。

五、拷贝继承

上面是采用prototype对象，实现继承。我们也可以换一种思路，纯粹采用"拷贝"方法实现继承。简单说，如果把父对象的所有属性和方法，拷贝进子对象，不也能够实现继承吗？这样我们就有了第五种方法。

首先，还是把Animal的所有不变属性，都放到它的prototype对象上。

```
function Animal(){}
Animal.prototype.species = "动物";
```

然后，再写一个函数，实现属性拷贝的目的。

```
function extend2(Child, Parent) {
    var p = Parent.prototype;
    var c = Child.prototype;
    for (var i in p) {
        c[i] = p[i];
    }
    c.uber = p;
}
```

这个函数的作用，就是将父对象的prototype对象中的属性，一一拷贝给Child对象的prototype对象。

使用的时候，这样写：

```
extend2(Cat, Animal);
var cat1 = new Cat("大毛","黄色");
alert(cat1.species); // 动物
```

非构造函数的继承

一、什么是"非构造函数"的继承？

比如，现在有一个对象，叫做"中国人"。

```
var Chinese = {
    nation:'中国'
};
```

还有一个对象，叫做"医生"。

```
var Doctor ={
    career:'医生'
}
```

请问怎样才能让"医生"去继承"中国人"，也就是说，我怎样才能生成一个"中国医生"的对象？

这里要注意，这两个对象都是普通对象，不是构造函数，无法使用构造函数方法实现"继承"。

二、object()方法

json格式的发明人Douglas Crockford，提出了一个object()函数，可以做到这一点。

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

这个object()函数，其实只做一件事，就是把子对象的prototype属性，指向父对象，从而使得子对象与父对象连在一起。

使用的时候，第一步先在父对象的基础上，生成子对象：

```
var Doctor = object(Chinese);
```

然后，再加上子对象本身的属性：

```
Doctor.career = '医生';
```

这时，子对象已经继承了父对象的属性了。

```
alert(Doctor.nation); //中国
```

三、浅拷贝

除了使用"prototype链"以外，还有另一种思路：把父对象的属性，全部拷贝给子对象，也能实现继承。

下面这个函数，就是在做拷贝：

```
function extendCopy(p) {
    var c = {};
    for (var i in p) {
        c[i] = p[i];
    }
    c.uber = p;
    return c;
}
```

使用的时候，这样写：

```
var Doctor = extendCopy(Chinese);
Doctor.career = '医生';
alert(Doctor.nation); // 中国
```

但是，这样的拷贝有一个问题。那就是，如果父对象的属性等于数组或另一个对象，那么实际上，子对象获得的只是一个内存地址，而不是真正拷贝，因此存在父对象被篡改的可能。

请看，现在给Chinese添加一个"出生地"属性，它的值是一个数组。

```
Chinese.birthPlaces = ['北京','上海','香港'];
```

通过extendCopy()函数，Doctor继承了Chinese。

```
var Doctor = extendCopy(Chinese);
```

然后，我们为Doctor的"出生地"添加一个城市：

```
Doctor.birthPlaces.push('厦门');
```

发生了什么事？Chinese的"出生地"也被改掉了！

```
alert(Doctor.birthPlaces); //北京，上海，香港，厦门
alert(Chinese.birthPlaces); //北京，上海，香港，厦门
```

所以，extendCopy()只是拷贝基本类型的数据，我们把这种拷贝叫做"浅拷贝"。这是早期jQuery实现继承的方式。

四、深拷贝

所谓"深拷贝"，就是能够实现真正意义上的数组和对象的拷贝。它的实现并不难，只要递归调用"浅拷贝"就行了。

```
function deepCopy(p, c) {
    var c = c || {};
    for (var i in p) {
        if (typeof p[i] === 'object') {
            c[i] = (p[i].constructor === Array) ? [] : {};
            deepCopy(p[i], c[i]);
        } else {
            c[i] = p[i];
        }
    }
    return c;
}
```

使用的时候这样写：

```
var Doctor = deepCopy(Chinese);
```

现在，给父对象加一个属性，值为数组。然后，在子对象上修改这个属性：

```
Chinese.birthPlaces = ['北京','上海','香港'];
Doctor.birthPlaces.push('厦门');
```

这时，父对象就不会受到影响了。

```
alert(Doctor.birthPlaces); //北京，上海，香港，厦门
alert(Chinese.birthPlaces); //北京，上海，香港
```

目前，jQuery库使用的就是这种继承方法。

Javascript定义类（class）的三种方法

在面向对象编程中，类（class）是对象（object）的模板，定义了同一组对象（又称"实例"）共有的属性和方法。Javascript语言不支持"类"，但是可以用一些变通的方法，模拟出"类"。

一、构造函数法

这是经典方法，也是教科书必教的方法。它用构造函数模拟"类"，在其内部用this关键字指代实例对象。

```
function Cat() {
    this.name = "大毛";
}
```

生成实例的时候，使用new关键字。

```
var cat1 = new Cat();
alert(cat1.name); // 大毛
```

类的属性和方法，还可以定义在构造函数的prototype对象之上。

```
Cat.prototype.makeSound = function(){
    alert("喵喵喵");
}
```

它的主要缺点是，比较复杂，用到了this和prototype，编写和阅读都很费力。

二、Object.create()法

为了解决"构造函数法"的缺点，更方便地生成对象，Javascript的国际标准ECMAScript第五版（目前通行的是第三版），提出了一个新的方法Object.create()。

用这个方法，"类"就是一个对象，不是函数。

```
var Cat = {
    name: "大毛",
    makeSound: function(){ alert("喵喵喵"); }
};
```

然后，直接用Object.create()生成实例，不需要用到new。

```
var cat1 = Object.create(Cat);
alert(cat1.name); // 大毛
cat1.makeSound(); // 喵喵喵
```

目前，各大浏览器的最新版本（包括IE9）都部署了这个方法。如果遇到老式浏览器，可以用下面的代码自行部署。

```
if (!Object.create) {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

这种方法比"构造函数法"简单，但是不能实现私有属性和私有方法，实例对象之间也不能共享数据，对"类"的模拟不够全面。

三、极简主义法

荷兰程序员Gabor de Mooij提出了一种比Object.create()更好的新方法，他称这种方法为"极简主义法"（minimalist approach）。这也是我推荐的方法。

3.1 封装

这种方法不使用this和prototype，代码部署起来非常简单，这大概也是它被叫做"极简主义法"的原因。

首先，它也是用一个对象模拟"类"。在这个类里面，定义一个构造函数createNew()，用来生成实例。

```
var Cat = {
    createNew: function(){
        // some code here
    }
};
```

然后，在createNew()里面，定义一个实例对象，把这个实例对象作为返回值。

```
var Cat = {
    createNew: function(){
        var cat = {};
        cat.name = "大毛";
        cat.makeSound = function(){ alert("喵喵喵"); };
        return cat;
    }
};
```

使用的时候，调用createNew()方法，就可以得到实例对象。

```
var cat1 = Cat.createNew();
cat1.makeSound(); // 喵喵喵
```

这种方法的好处是，容易理解，结构清晰优雅，符合传统的"面向对象编程"的构造，因此可以方便地部署下面的特性。

3.2 继承

让一个类继承另一个类，实现起来很方便。只要在前者的createNew()方法中，调用后者的createNew()方法即可。

先定义一个Animal类。

```
var Animal = {
    createNew: function(){
        var animal = {};
        animal.sleep = function(){ alert("睡懒觉"); };
        return animal;
    }
};
```

然后，在Cat的createNew()方法中，调用Animal的createNew()方法。

```
var Cat = {
    createNew: function(){
        var cat = Animal.createNew();
        cat.name = "大毛";
        cat.makeSound = function(){ alert("喵喵喵"); };
        return cat;
    }
};
```

这样得到的Cat实例，就会同时继承Cat类和Animal类。

```
var cat1 = Cat.createNew();
cat1.sleep(); // 睡懒觉
```

3.3 私有属性和私有方法

在createNew()方法中，只要不是定义在cat对象上的方法和属性，都是私有的。

```
var Cat = {
    createNew: function(){
        var cat = {};
        var sound = "喵喵喵";
        cat.makeSound = function(){ alert(sound); };
        return cat;
    }
};
```

上例的内部变量sound，外部无法读取，只有通过cat的公有方法makeSound()来读取。

```
var cat1 = Cat.createNew();
alert(cat1.sound); // undefined
```

3.4 数据共享

有时候，我们需要所有实例对象，能够读写同一项内部数据。这个时候，只要把这个内部数据，封装在类对象的里面、createNew()方法的外面即可。

```
var Cat = {
    sound : "喵喵喵",
    createNew: function(){
        var cat = {};
        cat.makeSound = function(){ alert(Cat.sound); };
        cat.changeSound = function(x){ Cat.sound = x; };
        return cat;
    }
};
```

然后，生成两个实例对象：

```
var cat1 = Cat.createNew();
var cat2 = Cat.createNew();
cat1.makeSound(); // 喵喵喵
```

这时，如果有一个实例对象，修改了共享的数据，另一个实例对象也会受到影响。

```
cat2.changeSound("啦啦啦");
cat1.makeSound(); // 啦啦啦
```

数据类型

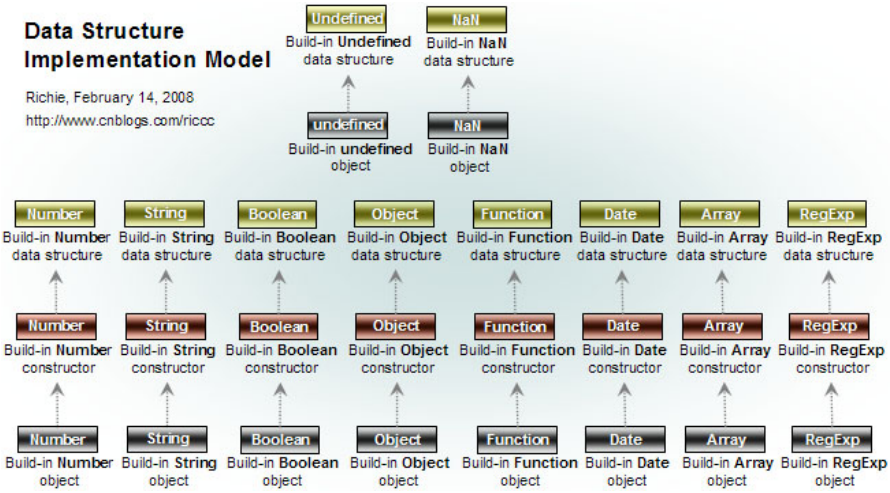
基本数据类型

基本数据类型是JS语言最底层的实现。
简单数值类型: 有Undefined, Null, Boolean, Number和String。注意，描述中的英文单词在这里仅指数据类型的名称，并不特指JS的全局对象NaN, Boolean, Number, String等，它们在概念上的区别是比较大的。
对象: 一个无序属性的集合，这些属性的值为简单数值类型、对象或者函数。同上，这里的对象并不特指全局对象Object。
函数: 函数是对象的一种，实现上内部属性[[Class]]值为"Function"，表明它是函数类型，除了对象的内部属性方法外，还有[[Construct]]、[[Call]]、[[Scope]]等内部属性。函数作为函数调用与构造器(使用new关键字创建实例对象)的处理机制不一样(Function对象除外)，内部方法[[Construct]]用于实现作为构造器的逻辑，方法[[Call]]实现作为函数调用的逻辑。同上，这里的函数并不特指全局对象Function。
函数在JS这个Prototype语言中可以看作是面向对象语言的类，可以用它来构造对象实例。既然函数可以看作是类，所以每一个函数可以看作是一种扩展数据类型。

内置数据类型(内置对象)

Function: 函数类型的用户接口。
Object: 对象类型的用户接口。
Boolean, Number, String: 分别为这三种简单数值类型的对象包装器。
Date, Array, RegExp: 可以把它们看作是几种内置的扩展数据类型。

首先，Function, Object, Boolean, Number, String, Date, Array, RegExp等都是JavaScript语言的内置对象，它们都可以看作是函数的派生类型，例如Number instanceof Function为true，Number instanceof Object为true。在这个意义上，可以将它们跟用户定义的函数等同看待。 其次，它们各自可以代表一种数据类型，由JS引擎用native code或内置的JS代码实现，是暴露给开发者对这些内置数据类型进行操作的接口。在这个意义上，它们都是一种抽象的概念，后面隐藏了具体的实现机制。 在每一个提到Number, Function等单词的地方，应该迅速的在思维中将它们实例化为上面的两种情况之一。
数据类型实现模型描述



Build-in ***data structure: 指JS内部用于实现***类型的数据结构，这些结构我们基本上无法直接操作。
Build-in ***object: 指JS内置的Number, String, Boolean等这些对象，这是JS将内部实现的数据类型暴露给开发者使用的接口。***
Build-in constructor: 指JS内置的一些构造器，用来构造相应类型的对象实例。它们被包装成函数对象暴露出来，例如我们可以使用下面的方法访问到这些函数对象:

```
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
//access the build-in number constructor
var number = new Number(123);
var numConstructor1 = number.constructor; //or
var numConstructor2 = new Object(123).constructor;
//both numConstructor1 and numConstructor2 are the build-in Number constructor
numConstructor1 == numConstructor2 //result: true
//access the build-in object constructor
var objConstructor1 = {}.constructor; //or
var objConstructor2 = new Object().constructor;
//both objConstructor1 and objConstructor2 are the build-in Object constructor
objConstructor1==objConstructor2 //result: true
```

具体实现上，上图中横向之间可能也存在关联，例如对于build-in data structure和constructor，Function、 Date、 Array、 RegExp等都可以继承Object的结构而实现，但这是具体实现相关的事情了。

关于简单数值类型的对象化

这是一个细微的地方，下面描述对于Boolean, String和Number这三种简单数值类型都适用，以Number为例说明。
JS规范要求: 使用var num1=123;这样的代码，直接返回基本数据类型，就是说返回的对象不是派生自Number和Object类型，用num1 instanceof Object测试为false；使用new关键字创建则返回Number类型，例如var num2=new Number(123); num2 instanceof Number为true。
将Number当作函数调用，返回结果会转换成简单数值类型。下面是测试代码:

```
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
var num1 = new Number(123); //num1 derived from Number & Object
num1 instanceof Number //result: true
num1 instanceof Object //result: true
//convert the num1 from Number type to primitive type, so it's no longer an instance of Number or Object
num1 = Number(num1);
num1 instanceof Number //result: false
num1 instanceof Object //result: false
var num2 = 123; //num2 is a primitive type
num2 instanceof Number //result: false
num2 instanceof Object //result: false
```

虽然我们得到了一个简单数值类型，但它看起来仍然是一个JS Object对象，具有Object以及相应类型的所有属性和方法，使用上基本没有差别，唯一不同之处是instanceof的测试结果。

Prototype

每个对象都有一个[[Prototype]]的内部属性，它的值为null或者另外一个对象。函数对象都有一个显示的prototype属性，它并不是内部[[Prototype]]属性。不同的JS引擎实现者可以将内部[[Prototype]]属性命名为任何名字，并且设置它的可见性，只在JS引擎内部使用。虽然

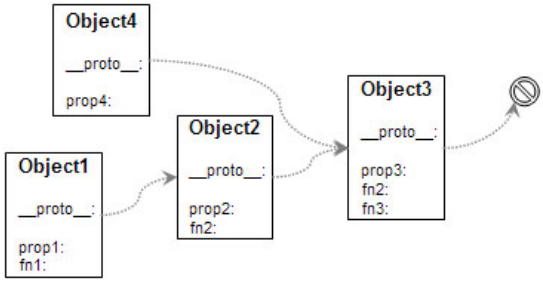
无法在JS代码中访问到内部[[Prototype]](FireFox中可以，名字为*_proto_*因为Mozilla将它公开了)，但可以使用对象的isPrototypeOf()方法进行测试，注意这个方法会在整个Prototype链上进行判断。

使用obj.propName访问一个对象的属性时，按照下面的步骤进行处理(假设obj的内部[[Prototype]]属性名为*_proto_*):

1. 如果obj存在propName属性，返回属性的值，否则
2. 如果obj.*_proto_*为null，返回undefined，否则
3. 返回obj.*_proto_*.propName

调用对象的方法跟访问属性搜索过程一样，因为方法的函数对象就是对象的一个属性值。
提示: 上面步骤中隐含了一个递归过程，步骤3中obj.*_proto_*是另外一个对象，同样将采用1, 2, 3这样的步骤来搜索propName属性。

例如下图所示，object1将具备属性prop1, prop2, prop3以及方法fn1, fn2, fn3。图中虚线箭头表示prototype链。



这就是基于Prototype的继承和共享。其中object1的方法fn2来自object2，概念上即object2重写了object3的方法fn2。
JavaScript对象应当都通过prototype链关联起来，最顶层是Object，即对象都派生自Object类型。

对象创建过程

JS中只有函数对象具备类的概念，因此要创建一个对象，必须使用函数对象。函数对象内部有[[Construct]]方法和[[Call]]方法，[[Construct]]用于构造对象，[[Call]]用于函数调用，`只有使用new操作符时才触发[[Construct]]逻辑`。

var obj=new Object(); 是使用内置的Object这个函数对象创建实例化对象obj。var obj={};和var obj=[];这种代码将由JS引擎触发Object和Array的构造过程。function fn(){}; var myObj=new fn();是使用用户定义的类型创建实例化对象。

new Fn(args)的创建过程如下(即函数对象的[[Construct]]方法处理逻辑，对象的创建过程)。另外函数对象本身的创建过程(指定义函数或者用Function创建一个函数对象等方式)虽然也使用了下面的处理逻辑，但有特殊的地方，后面再描述。

1. 创建一个build-in object对象obj并初始化
2. 如果Fn.prototype是Object类型，则将obj的内部[[Prototype]]设置为Fn.prototype，否则obj的[[Prototype]]将为其初始化值(即Object.prototype)
3. 将obj作为this，使用args参数调用Fn的内部[[Call]]方法
    1. 内部[[Call]]方法创建当前执行上下文
    2. 调用F的函数体
    3. 销毁当前的执行上下文
    4. 返回F函数体的返回值，如果F的函数体没有返回值则返回undefined
4. 如果[[Call]]的返回值是Object类型，则返回这个值，否则返回obj

注意步骤2中， prototype指对象显示的prototype属性，而[[Prototype]]则代表对象内部Prototype属性(隐式的)。

构成对象Prototype链的是内部隐式的[[Prototype]]，而并非对象显示的prototype属性。显示的prototype只有在函数对象上才有意义，从上面的创建过程可以看到，函数的prototype被赋给派生对象隐式[[Prototype]]属性，这样根据Prototype规则，派生对象和函数的prototype对象之间才存在属性、方法的继承/共享关系。

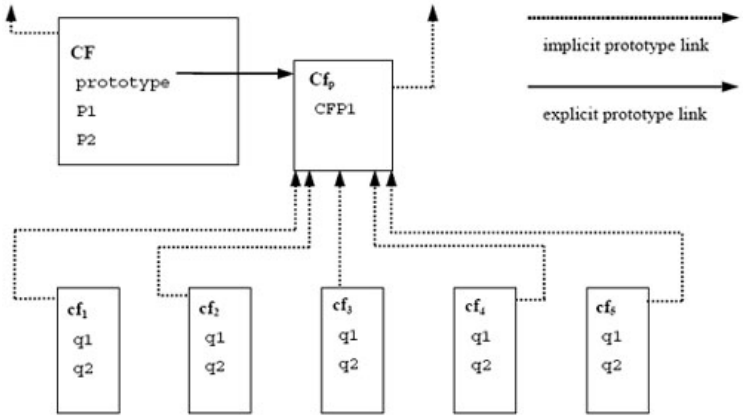
用代码来做一些验证:

```
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
function fn(){}
//the value of implicit [[Prototype]] property of those objects derived from fn will be assigned to fn.prototype
fn.prototype={ attr1:"aaa", attr2:"bbb"};
var obj=new fn();
document.write(obj.attr1 + "<br />"); //result: aaa
document.write(obj.attr2 + "<br />"); //result: bbb
document.write(obj instanceof fn); //result: true
document.write("<br />");
//I change the prototype of fn here, so by the algorithm of Prototype the obj is no longer the instance of fn,
//but this won't affect the obj and its [[Prototype]] property, and the obj still has attr1 and attr2 properties
fn.prototype={};
document.write(obj.attr1 + "<br />"); //result: aaa
document.write(obj.attr2 + "<br />"); //result: bbb
document.write(obj instanceof fn); //result: false
```

关于创建过程返回值的验证:

```
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
function fn(){
    //according to step 4 described above,
    //the new fn() operation will return the object { attr1: 111, attr2: 222 }, it's not an instance of fn!
    return { attr1: 111, attr2: 222 };
}
fn.prototype={ attr1:"aaa", attr2:"bbb"};
var obj=new fn();
document.write(obj.attr1 + "<br />"); //result: 111
document.write(obj.attr2 + "<br />"); //result: 222
document.write(obj instanceof fn); //result: false
```

经过上面的理解应，请写出下面这幅图的实现代码。图中CF是一个函数，Cfp是CF的prototype对象，cf1, cf2, cf3, cf4, cf5都是CF的实例对象。虚线箭头表示隐式Prototype关系，实线箭头表示显示prototype关系.



供参考的实现方案:

```
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
function CF(q1, q2){
    this.q1=q1;
    this.q2=q2;
}
CF.P1="P1 in CF";
CF.P2="P2 in CF";
function Cfp(){
    this.CFP1="CFP1 in Cfp";
}
CF.prototype=new Cfp();
var cf1=new CF("aaa", "bbb");
document.write(cf1.CFP1 + "<br />"); //result: CFP1 in Cfp
document.write(cf1.q1 + "<br />"); //result: aaa
document.write(cf1.q2 + "<br />"); //result: bbb
```

本地属性与继承属性

对象通过隐式Prototype链能够实现属性和方法的继承，但prototype也是一个普通对象，就是说它是一个普通的实例化的对象，而不是纯粹抽象的数据结构描述。所以就有了这个本地属性与继承属性的问题。

首先看一下设置对象属性时的处理过程。JS定义了一组attribute，用来描述对象的属性property，以表明属性property是否可以在JavaScript代码中设值、被for in枚举等。

obj.propName=value的赋值语句处理步骤如下:

1. 如果propName的attribute设置为不能设值，则返回
2. 如果obj.propName不存在，则为obj创建一个属性，名称为propName
3. 将obj.propName的值设为value

可以看到，设值过程并不会考虑Prototype链，道理很明显，`obj的内部[[Prototype]]是一个实例化的对象，它不仅仅向obj共享属性，还可能向其它对象共享属性，修改它可能影响其它对象`。

用上面CF, Cfp的示例来说明，实例对象cf1具有本地属性q1, q2以及继承属性CFP1，如果执行cf1.CFP1=""，那么cf1就具有本地属性CFP1了，测试结果如下:

```
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
var cf1=new CF("aaa", "bbb");
var cf2=new CF(111, 222);
document.write(cf1.CFP1 + "<br />"); //result: CFP1 in Cfp
document.write(cf2.CFP1 + "<br />"); //result: CFP1 in Cfp
//it will result in a local property in cf1
cf1.CFP1="new value for cf1";
//changes on CF.prototype.CFP1 will affect cf2 but not cf1, because there's already a local property with
//the name CFP1 in cf1, but no such one in cf2
CF.prototype.CFP1="new value for Cfp";
document.write(cf1.CFP1 + "<br />"); //result: new value for cf1
document.write(cf2.CFP1 + "<br />"); //result: new value for Cfp
```

语义上的混乱?
还是使用上面CF, Cfp示例的场景。
根据Prototype的机制，我们可以说对象cf1, cf2等都继承了对象Cfp的属性和方法，所以应该说他们之间存在继承关系。属性的继承/共享是沿着隐式Prototype链作用的，所以继承关系也应当理解为沿着这个链。
我们再看instanceOf操作，只有cf1 instanceOf CF才成立，我们说cf1是CF的实例对象，CF充当了类的角色，而不会说cf1是Cfp的实例对象，这样我们应当说cf1继承自CF? 但CF充当的只是一个第三方工厂的角色，它跟cf1之间并没有属性继承这个关系。
把CF, Cfp看作一个整体来理解也同样牵强。

Prototype就是Prototype，没有必要强把JavaScript与面向对象概念结合起来， JavaScript只具备有限的面向对象能力，从另外的角度我们可以把它看成函数语言、动态语言，所以它是吸收了多种语言特性的精简版。

对象模型

Where are we?

1. 了解了JavaScript的数据类型，清楚了象Number这样的系统内置对象具有多重身份:
   a. 它们本身是一个函数对象，只是由引擎内部实现而已，
   b. 它们代表一种数据类型，我们可以用它们定义、操作相应类型的数据，
   c. 在它们背后隐藏了引擎的内部实现机制，例如内部的数据结构、各种被包装成了JavaScript对象的构造器等。
2. 了解了Prototype机制，知道对象是如何通过它们继承属性和方法，知道了在创建对象过程中JS引擎内部是如何设置Prototype关系的。

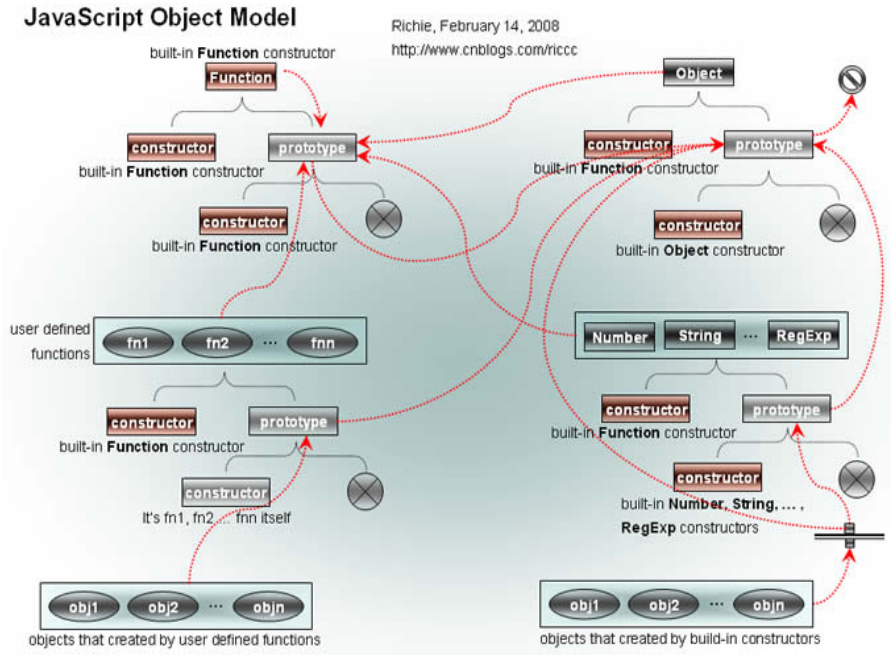
接下来对用户自定义函数对象本身的创建过程进行了解之后，我们就可以对JavaScript的对象模型来一个整体性的overview了。

函数对象创建过程
JavaScript代码中定义函数，或者调用Function创建函数时，最终都会以类似这样的形式调用Function函数:var newFun=Function(funArgs, funBody); 。创建函数对象的主要步骤如下:

1. 创建一个build-in object对象fn
2. 将fn的内部[[Prototype]]设为Function.prototype
3. 设置内部的[[Call]]属性，它是内部实现的一个方法，处理逻辑参考对象创建过程的步骤3
4. 设置内部的[[Construct]]属性，它是内部实现的一个方法，处理逻辑参考对象创建过程的步骤1,2,3,4
5. 设置fn.length为funArgs.length，如果函数没有参数，则将fn.length设置为0
6. 使用new Object()同样的逻辑创建一个Object对象fnProto
7. 将fnProto.constructor设为fn
8. 将fn.prototype设为fnProto
9. 返回fn

步骤1跟步骤6的区别为，步骤1只是创建内部用来实现Object对象的数据结构(build-in object structure)，并完成内部必要的初始化工作，但它的[[Prototype]]、[[Call]]、[[Construct]]等属性应当为null或者内部初始化值，即我们可以理解为不指向任何对象(对[[Prototype]]这样的属性而言)，或者不包含任何处理(对[[Call]]、[[Construct]]这样的方法而言)。步骤6则将按照前面描述的对象创建过程创建一个新的对象，它的[[Prototype]]等被设置了。

从上面的处理步骤可以了解，`任何时候我们定义一个函数，它的prototype是一个Object实例，这样默认情况下我们创建自定义函数的实例对象时，它们的Prototype链将指向Object.prototype`。
另外，Function一个特殊的地方，是它的[[Call]]和[[Construct]]处理逻辑一样。
JavaScript对象模型



红色虚线表示隐式Prototype链。

这张对象模型图中包含了太多东西，不少地方需要仔细体会，可以写些测试代码进行验证。彻底理解了这张图，对JavaScript语言的了解也就差不多了。下面是一些补充说明：

1. 图中有好几个地方提到build-in Function constructor，这是同一个对象，可以测试验证:

```
 //Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
 Function==Function.constructor //result: true
 Function==Function.prototype.constructor //result: true
 Function==Object.constructor //result: true
 //Function also equals to Number.constructor, String.constructor, Array.constructor, RegExp.constructor,
etc.
 function fn(){}
 Function==fn.constructor //result: true
```

   这说明了几个问题: Function指向系统内置的函数构造器(build-in Function constructor)；Function具有自举性；系统中所有函数都是由Function构造。

2. 左下角的obj1, obj2...objn范指用类似这样的代码创建的对象: function fn1(){}; var obj1=new fn1();

   这些对象没有本地constructor方法，但它们将从Prototype链上得到一个继承的constructor方法，即fn.prototype.constructor，从函数对象的构造过程可以知道，它就是fn本身了。

   右下角的obj1, obj2...objn范指用类似这样的代码创建的对象: var obj1=new Object();或var obj1={};或var obj1=new Number(123);或obj1=/\w+/;等等。所以这些对象Prototype链的指向、从Prototype链继承而来的constructor的值(指它们的constructor是build-in Number constructor还是build-in Object constructor等)等依赖于具体的对象类型。另外注意的是，var obj=new Object(123);这样创建的对象，它的类型仍然是Number，即同样需要根据参数值的类型来确定。

   同样它们也没有本地constructor，而是从Prototype链上获得继承的constructor方法，即build-in *** constructor，具体是哪一个由数据类型确定。

3. 关于图中Prototype链的补充说明:

   - Object.prototype是整个链的终结点，它的内部[[Prototype]]为null。

- 所有函数的Prototype链都指向Function.prototype。
- `Function的Prototype链指向Function.prototype，这是规范要求的，因为设计者将Function设计为具有自举性。Function的Prototype链这样设计之后，Function.constructor==Function，Function instanceOf Function都为true。另外Function已经是最顶层的构造器，但Function本身也是一个函数对象，它必然是由某个东西创建出来的，这样自举在语义上合情合理。`
- `Function.prototype的Prototype链指向Object.prototype，这也是规范强制要求的`。首先Function.prototype是Function的一个实例对象(typeof Function.prototype可以知道它是一个Function，instanceOf无法通过测试，因为Prototype链在内部被额外设置了），所以按照Prototype的规则，Function.prototype的内部[[Prototype]]值应当为Function.prototype这个对象，即它的Prototype链指向自己本身。这样一方面在Prototype链上造成一个死循环，另一方面它本身成为了一个终结点，结果就是所有函数对象将不是派生自Object了。加上这个强制要求之后，Prototype链只有唯一的一个终结点。

4. 因为Function.prototype是一个函数对象，所以它应当具有显示的prototype属性，即Function.prototype.prototype，但只有FireFox中可以访问到，IE、Opera、Safari都无法访问。所以图中用了个表示不存在的符号。
5. 用户自定义函数(user defined functions)默认情况下[[Prototype]]值是Object.prototype，即它的隐式Prototype链指向Object.prototype，所以图中就这样表示了，但并不代表总是这样，当用户设置了自定义函数的prototype属性之后，情况就不同了。

执行模型

执行上下文(Execution Context)简介

JavaScript代码运行的地方都存在执行上下文，它是一个概念，一种机制，用来完成JavaScript运行时作用域、生存期等方面的处理。执行上下文包括Variable Object、Variable Instatiation、Scope/Scope Chain等概念，在不同的场景/执行环境下，处理上存在一些差异，下面先对这些场景进行说明。

函数对象分为用户自定义函数对象和系统内置函数对象，对于用户自定义函数对象将按照下面描述的机制进行处理，但内置函数对象与具体实现相关，ECMA规范对它们执行上下文的处理没有要求，即它们基本不适合本节描述的内容。

执行的JavaScript代码分三种类型，后面会对这三种类型处理上不同的地方进行说明:

1. Global Code，即全局的、不在任何函数里面的代码，例如一个js文件、嵌入在HTML页面中的js代码等。
2. Eval Code，即使用eval()函数动态执行的JS代码。
3. Function Code，即用户自定义函数中的函数体JS代码。

基本原理

在用户自定义函数中，可以传入参数、在函数中定义局部变量，函数体代码可以使用这些入参、局部变量。背后的机制是什么样呢?

`当JS执行流进入函数时，JavaScript引擎在内部创建一个对象，叫做Variable Object。对应函数的每一个参数，在Variable Object上添加一个属性，属性的名字、值与参数的名字、值相同。函数中每声明一个变量，也会在Variable Object上添加一个属性，名字就是变量名，因此为变量赋值就是给Variable Object对应的属性赋值。在函数中访问参数或者局部变量时，就是在variable Object上搜索相应的属性，返回其值。`

一般情况下Variable Object是一个内部对象，JS代码中无法直接访问。规范中对其实现方式也不做要求，因此它可能只是引擎内部的一种数据结构。

大致处理方式就这样，但作用域的概念不只这么简单，例如函数体中可以使用全局变量、函数嵌套定义时情况更复杂点。这些情况下怎样处理? JavaScript引擎将不同执行位置上的Variable Object按照规则构建一个链表，在访问一个变量时，先在链表的第一个Variable Object上查找，如果没有找到则继续在第二个Variable Object上查找，直到搜索结束。这就是Scope/Scope Chain的大致概念。

下面是各个方面详细的处理。

- Global Object

  JavaScript的运行环境都必须存在一个唯一的全局对象-Global Object，例如HTML中的window对象。Global Object是一个宿主对象，除了作为JavaScript运行时的全局容器应具备的职责外，ECMA规范对它没有额外要求。它包Math、String、Date、parseInt等JavaScript中内置的全局对象、函数(都作为Global Object的属性)，还可以包含其它宿主环境需要的一些属性。

- Variable Object

  上面简述了Variable Object的基本概念。创建Variable Object，将参数、局部变量设置为Variable Object属性的处理过程叫做Variable Instatiation-变量实例化，后面结合Scope Chain再进行详细说明。

- Global Code

  Variable Object就是Global Object，这是Variable Object唯一特殊的地方(指它是内部的无法访问的对象而言)。

  ```
  var globalVariable = "WWW";
  document.write(window.globalVariable); //result: WWW
  ```

  上面代码在Global Code方式下运行，根据对Variable Object的处理，定义变量globalVariable时就会在Global Object(即window)对象上添加这个属性，所以输出是WWW这个值。

- Function Code

  Variable Object也叫做Activation Object(因为有一些差异存在，所以规范中重新取一个名字以示区别，Global Code/Eval Code中叫Variable Object，Function Code中就叫做Activation Object)。 每次进入函数执行都会创建一个新的`Activation Object`对象，然后创建一个`arguments`对象并设置为`Activation Object`的属性，再进行`Variable Instantiation`处理。 在退出函数时，`Activation Object`会被丢弃(并不是内存释放，只是可以被垃圾回收了)。

  附arguments对象的属性:

  - length: 为实际传入参数的个数。注意，参考函数对象创建过程，函数对象上的length为函数定义时要求的参数个数;
  - callee: 为执行的函数对象本身。目的是使函数对象能够引用自己，例如需要递归调用的地方。
    function fnName(...) { ... }这样定义函数，它的递归调用可以在函数体内使用fnName完成。var fn=function(...) { ... }这样定义匿名函数，在函数体内无法使用名字引用自己，通过arguments.callee就可以引用自己而实现递归调用。
  - 参数列表: 调用者实际传入的参数列表。这个参数列表提供一个使用索引访问实际参数的方法。Variable Instantiation处理时会在Activation Object对象上添加属性，前提是函数声明时有指定参数列表。如果函数声明中不给出参数列表，或者实际调用参数个数与声明时的不一样，可以通过arguments访问各个参数。

    arguments中的参数列表与Activation Object上的参数属性引用的是相同的参数对象(如果修改，在两处都会反映出来)。规范并不要求arguments是一个数组对象，下面是一个测试:

    ```
    //Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
    var argumentsLike = { 0: "aaa", 1: 222, 2: "WWW", length: 3, callee: function() { } };
    document.write(argumentsLike[2] + "<br />"); //result: WWW
    document.write(argumentsLike[1] + "<br />"); //result: 222
    //convert the argumentsLike to an Array object, just as we can do this for the arguments property
    var array = [].slice.apply(argumentsLike);
    document.write(array instanceof Array); //result: true
    document.write("<br />");
    document.write(array.reverse().join("|")); //result: WWW|222|aaa
    ```

- Eval Code

  Variable Object就是调用eval时当前执行上下文中的Variable Object。在Global Code中调用eval函数，它的Variable Object就是Global Object；在函数中调用eval，它的Variable Object就是函数的Activation Object。

  ```
  //Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
  function fn(arg){
      var innerVar = "variable in function";
      eval(' \
          var evalVar = "variable in eval"; \
          document.write(arg + "<br />"); \
          document.write(innerVar + "<br />"); \
      ');
      document.write(evalVar);
  }
  fn("arguments for function");
  输出结果是:
  arguments for function
  variable in function
  variable in eval
  ```

  说明: eval调用中可以访问函数fn的参数、局部变量；在eval中定义的局部变量在函数fn中也可以访问，因为它们的Varible Object是同一个对象。

- Scope/Scope Chain

  首先Scope Chain是一个类似链表/堆栈的结构，里面每个元素基本都是Variable Object/Activation Object。
  其次存在执行上下文的地方都有当前Scope Chain，可以理解为Scope Chain就是执行上下文的具体表现形式。

- Global Code

  Scope Chain只包含一个对象，即Global Object。在开始JavaScript代码的执行之前，引擎会创建好这个Scope Chain结构。

- Function Code

  函数对象在内部都有一个[[Scope]]属性，用来记录该函数所处位置的Scope Chain。

  创建函数对象时，引擎会将当前执行环境的Scope Chain传给Function的[[Construct]]方法。[[Construct]]会创建一个新的Scope Chain，内容与传入的Scope Chain完全一样，并赋给被创建函数的内部[[Scope]]属性。在前面函数对象创建过程一节中，这个处理位于步骤4和5之间。

  进入函数调用时，也会创建一个新的Scope Chain，包括同一个函数的递归调用，退出函数时这个Scope Chain被丢弃。新建的Scope Chain第一个对象是Activation Object，接下来的内容与内部[[Scope]]上存储的Scope Chain内容完全一样。

- Eval Code

  进入Eval Code执行时会创建一个新的Scope Chain，内容与当前执行上下文的Scope Chain完全一样。

  实例说明
  Scope Chain的原理就上面这些，必须结合JS代码的执行、Variable Instantiation的细节处理，才能理解上面这些如何产生作用，下面用一个简单的场景来综合说明。假设下面是一段JavaScript的Global Code:

```
var outerVar1="variable in global code";
function fn1(arg1, arg2){
    var innerVar1="variable in function code";
    function fn2() { return outerVar1+" - "+innerVar1+" - "+" - "+(arg1 + arg2); }
    return fn2();
}
var outerVar2=fn1(10, 20);
```

  执行处理过程大致如下:

  1. 初始化Global Object即window对象，Variable Object为window对象本身。创建Scope Chain对象，假设为scope_1，其中只包含window对象。
  2. 扫描JS源代码(读入源代码、可能有词法语法分析过程)，从结果中可以得到定义的变量名、函数对象。按照扫描顺序:
     a. 发现变量outerVar1，在window对象上添加outerVar1属性，值为undefined；
     b. 发现函数fn1的定义，使用这个定义创建函数对象，传给创建过程的Scope Chain为scope_1。将结果添加到window的属性中，名字为fn1，值为返回的函数对象。注意fn1的内部[[Scope]]就是scope_1。另外注意，创建过程并不会对函数体中的JS代码做特殊处理，可以理解为只是将函数体JS代码的扫描结果保存在函数对象的内部属性上，在函数执行时再做进一步处理。这对理解Function Code，尤其是嵌套函数定义中的Variable Instantiation很关键；
     c. 发现变量outerVar2，在window对象上添加outerVar2属性，值为undefined；
  3. 执行outerVar1赋值语句，赋值为"variable in global code"。
  4. 执行函数fn1，得到返回值:
     a. 创建Activation Object，假设为activation_1；创建一个新的Scope Chain，假设为scope_2，scope_2中第一个对象为activation_1，第二个对象为window对象(取自fn1的[[Scope]]，即scope_1中的内容)；
     b. 处理参数列表。在activation_1上设置属性arg1、arg2，值分别为10、20。创建arguments对象并进行设置，将arguments设置为activation_1的属性；
     c. 对fn1的函数体执行类似步骤2的处理过程:
        - 发现变量innerVar1，在activation_1对象上添加innerVar1属性，值为undefine；
        - 发现函数fn2的定义，使用这个定义创建函数对象，传给创建过程的Scope Chain为scope_2(函数fn1的Scope Chain为当前执行上下文的内容)。将结果添加到activation_1的属性中，名字为fn2，值为返回的函数对象。注意fn2的内部[[Scope]]就是scope_2；
     d. 执行innerVar1赋值语句，赋值为"variable in function code"。
     e. 执行fn2:
        - 创建Activation Object，假设为activation_2；创建一个新的Scope Chain，假设为scope_3，scope_3中第一个对象为activation_2，接下来的对象依次为activation_1、window对象(取自fn2的[[Scope]]，即scope_2)；
        - 处理参数列表。因为fn2没有参数，所以只用创建arguments对象并设置为activation_2的属性。
        - 对fn2的函数体执行类似步骤2的处理过程，没有发现变量定义和函数声明。
        - 执行函数体。对任何一个变量引用，从scope_3上进行搜索，这个示例中，outerVar1将在window上找到；innerVar1、arg1、arg2将在activation_1上找到。
        - 丢弃scope_3、activation_2(指它们可以被垃圾回收了)。
        - 返回fn2的返回值。
     f. 丢弃activation_1、scope_2。
     g. 返回结果。
  5. 将结果赋值给outerVar2。

  其它情况下Scope Chain、Variable Instantiation处理类似上面的过程进行分析就行了。

  根据上面的实例说明，就可以解释下面这个测试代码的结果:

```
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
function fn(obj){
    return {
        //test whether exists a local variable "outerVar" on obj
        exists: Object.prototype.hasOwnProperty.call(obj, "outerVar"),
        //test the value of the variable "outerVar"
        value: obj.outerVar
    };
}
var result1 = fn(window);
var outerVar = "WWW";
var result2 = fn(window);

document.write(result1.exists + " " + result1.value); //result: true undefined
document.write("<br />");
document.write(result2.exists + " " + result2.value); //result: true WWW
```

result1调用的地方，outerVar声明和赋值的语句还没有被执行，但是测试结果window对象已经拥有一个本地属性outerVar，其值为undefined。result2的地方outerVar已经赋值，所以window.outerVar的值已经有了。实际使用中不要出现这种先使用，后定义的情况，否则某些情况下会有问题，因为会涉及到一些规范中没有提及，不同厂商实现方式上不一致的地方。

一些特殊处理

1. with(obj) { ... }这个语法的实现方式，是在当前的Scope Chain最前面位置插入obj这个对象，这样就会先在obj上搜索是否有相应名字的属性存在。其它类似的还有catch语句。
2. 前面对arguments对象的详细说明中，提到了对函数递归调用的支持问题，了解到了匿名函数使用arguments.callee来实现引用自己，而命名函数可以在函数体内引用自己，根据上面Scope Chain的工作原理我们还无法解释这个现象，因为这里有个特殊处理。任何时候创建一个命名函数对象时，JavaScript引擎会在当前执行上下文Scope Chain的最前面插入一个对象，这个对象使用new Object()方式创建，并将这个Scope Chain传给Function的构造函数[[Construct]]，最终创建出来的函数对象内部[[Scope]]上将包含这个object对象。创建过程返回之后，JavaScript引擎在object上添加一个属性，名字为函数名，值为返回的函数对象，然后从当前执行上下文的Scope Chain中移除它。这样函数对象的Scope Chain中第一个对象就是对自己的引用，而移除操作则确保了对函数对象创建处Scope Chain的恢复。

this关键字处理

执行上下文包含的另一个概念是this关键字。

Global Code中this关键字为Global Object；函数调用时this关键字为调用者，例如obj1.fn1()，在fn1中this对象为obj1；Eval Code中this关键字为当前执行上下文的Variable Object。

在函数调用时，JavaScript提供一个让用户自己指定this关键字值的机会，即每个函数都有的call、apply方法。例如:fn1.call(obj1, arg1, arg2, ...)或者fn1.apply(obj1, argArray)，都是将obj1作为this关键字，调用执行fn1函数，后面的参数都作为函数fn1的参数。如果obj1为null或undefined，则Global Object将作为this关键字的值；如果obj1不是Object类型，则转化为Object类型。它们之间的唯一区别在于，apply允许以数组的方式提供各个参数，而call方法必须一个一个参数的给。

前面的测试示例代码中有多处运用到了这个方法。例如window对象并没有hasOwnProperty方法，使用Object.prototype.hasOwnProperty.call(window, "propertyName")也可以测试它是否拥有某个本地属性。

JavaScript中的闭包Closures

示例：

```
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4
function outer(){
    var a="aaa";
    var b="bbb";
    return function(){ return a + " " + b; };
}
var inner=outer();
document.write(inner());
```

outer返回的是一个内嵌函数，内嵌函数使用了outer的局部变量a和b。照理outer的局部变量在返回时就超出了作用域因此inner()调用无法使用才对。这就是闭包Closure，即函数调用返回了一个内嵌函数，而内嵌函数引用了外部函数的局部变量、参数等这些应当被关闭(Close)了的资源。

根据前面Scope Chain的理解可以解释，返回的内嵌函数已经持有了构造它时的Scope Chain，虽然outer返回导致这些对象超出了作用域、生存期范围，但JavaScript使用自动垃圾回收来释放对象内存: 按照规则定期检查，对象没有任何引用才被释放。因此上面的代码能够正确运行。

关于使用Closure时的内存泄漏、效率等问题，参考http://www.jibbering.com/faq/faq_notes/closures.html

参考文档：
深入浅出 JavaScript 中的 this
http://www.ibm.com/developerworks/cn/web/1207_wangqf_jsthis/
http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this/
http://www.cnblogs.com/lhb25/archive/2011/09/06/javascript-scope-chain.html
http://www.ruanyifeng.com/blog/2009/08/learning_javascript_closures.html
http://www.cnblogs.com/RicCC/archive/2008/02/15/JavaScript-Object-Model-Execution-Model.html
Javascript面向对象编程
http://www.ruanyifeng.com/blog/2011/06/designing_ideas_of_inheritance_mechanism_in_javascript.html
http://www.ruanyifeng.com/blog/2010/05/object-oriented_javascript_encapsulation.html
http://www.ruanyifeng.com/blog/2010/05/object-oriented_javascript_inheritance.html
http://www.ruanyifeng.com/blog/2010/05/object-oriented_javascript_inheritance_continued.html
Javascript定义类（class）的三种方法
http://www.ruanyifeng.com/blog/2012/07/three_ways_to_define_a_javascript_class.html

---

javascript之旅

闭包

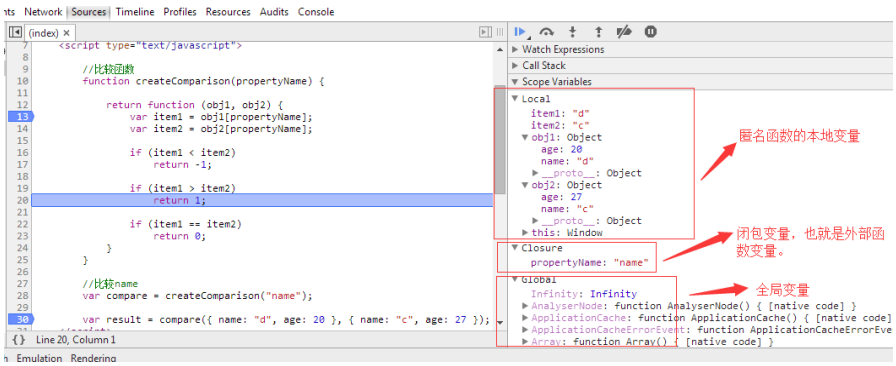
闭包：就是有权访问另一个函数(包含函数)作用域的变量的函数。

闭包经典示例：

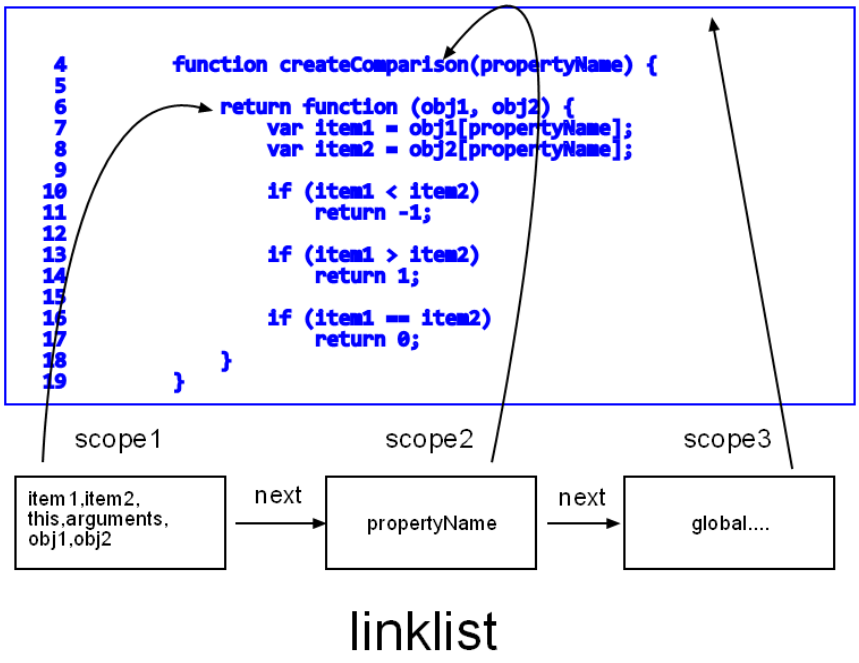
```
//比较函数
function createComparison(propertyName) {

    return function (obj1, obj2) {
        var item1 = obj1[propertyName];
        var item2 = obj2[propertyName];

        if (item1 < item2)
            return -1;
        if (item1 > item2)
            return 1;
        if (item1 == item2)
            return 0;
    }
}
var compare = createComparison("name");
var result = compare({ name: "d", age: 20 }, { name: "c", age: 27 });
```



其实在每个function里面都有一个scope属性，当然这个属性被引擎屏蔽了，你是看不见也摸不着的，它里面就保存着当前函数的 local variables，如果应用到上面demo的话，就是全局函数中有一个scope，createComparison有一个scope，匿名的compare有一个scope，而且这三个scope还是通过链表链接的。



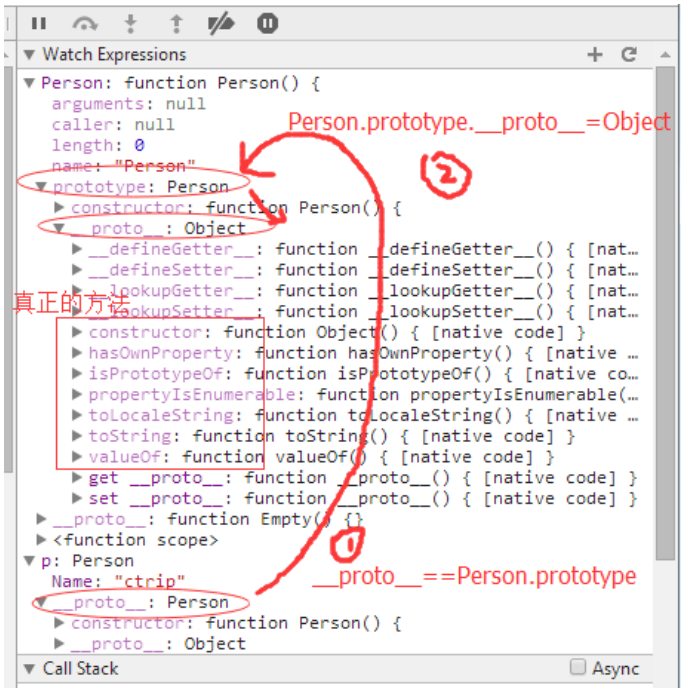
linklist

原型

默认继承Object

```
function Person() {
    this.Name="ctrip";
}
var p=new Person();
```

用chorme的watch expressions看一看



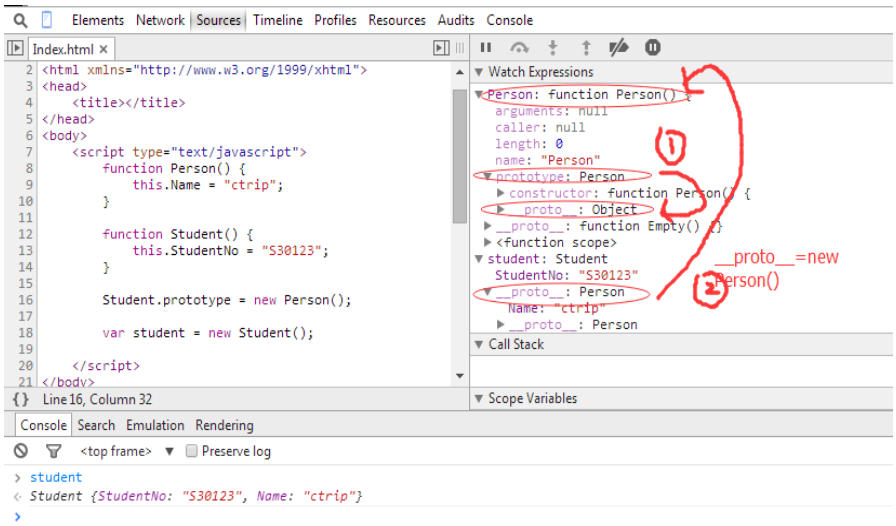
从上面的图中不难看到,其实有这么个原型链的关系:

p.__proto__ =Person.prototype

Person.prototype.__proto__ -> new Object()

自定义继承

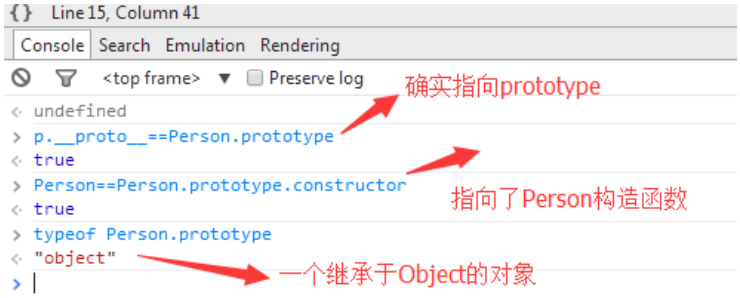
```
function Person() {
    this.Name = "ctrip";
}
function Student() {
    this.StudentNo = "2816";
}
Student.prototype = new Person();
var student = new Student();
```



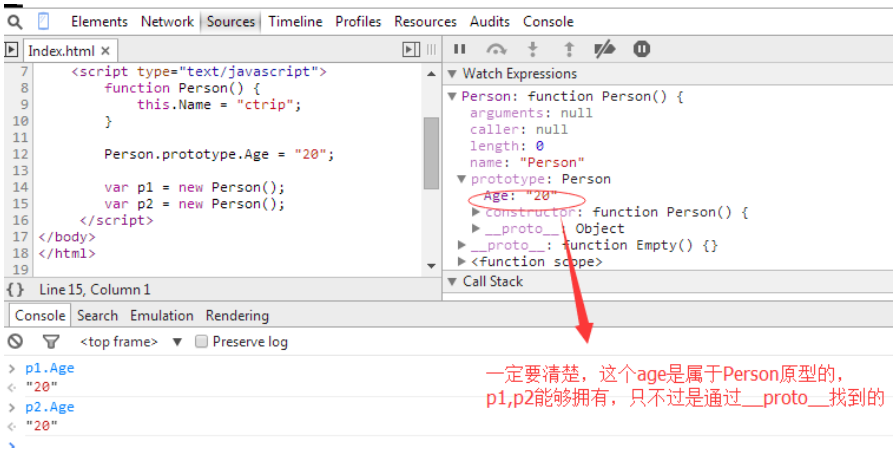
详解prototype

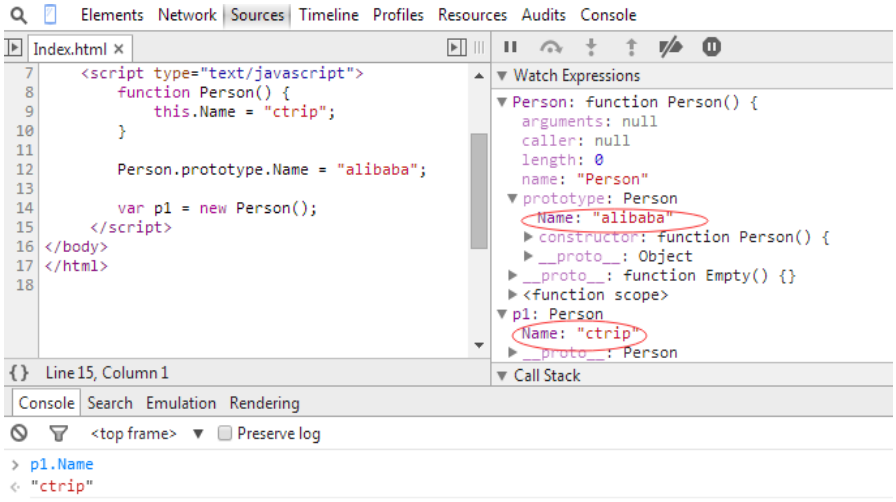
其实prototype只不过是一个包含constructor属性的Object对象，其中constructor属性是指向当前function的一个指针.

```
function Person() {
    this.Name="ctrip";
}
Person.prototype={
    constructor:Person //指向Person的constructor
}
var p=new Person();
```



prototype属性共享





prototype属性冲突

instanceof

主页面给其中的iframe页面传参（以数组为例），进行参数检测，此时不能用instanceof进行判断，因为传递的参数是两个引用，并且Array是挂在window下的一个属性，window属性也就是一个窗口的实例，那就说明主页面是一个window实例，iframe页面也是一个window实例。

解决办法：
每一个function中都会有call方法和prototype属性，而js在Object.prototype中的tostring函数上做了一个封装，就是调用tostring.call后，会返回[object constructorName]的字符串格式，这里的constructorName就是call参数的函数名，比如我们把arr传进去，就会返回“[object Array]”字符串格式，这个方法也可以让我们巧妙的判断是否是Array。

```
var arr=["a","b","c"];
var result=Object.prototype.toString.call(arr);
console.log(result==="[object Array]");//true
```

[Writable]特性

属性的只读操作。在js中你只要使用defineProperty方法就可以了。

使用了defineProperty方法将person.name变成了只读字段.
configurable告诉js引擎是否可以delete,update属性，当把configurable设为false的时候，你就不可以delete p.Name了，因为这会是一个无效操作。

包装类型

String

```
var s="hello";
console.log(typeof s);//string
var r=s.substring(3);
console.log(r);//lo
```

string的值是直接保存在栈上面的，那它怎么会有substring呢？按照官网的解释是这样的：这时候会使用String类型把s包装成引用类型。然后使用String类型的内部实现，正好String内部定义了substring方法，所以其实上面的代码在js的内部应该是这样实现的。

```
var s=new String("hello")
var r=s.substring(3)
s="hello"
```

可以看到，`其实包装类型只是在执行代码的一瞬间，将s包装成了String引用类型`，然后再调用String引用类型下面的substring方法，继而重新将“hello”值赋给s，最后的效果就是s="hello", r="lo"

Boolean

一个引用类型，除非它是null或者undefined，否则它永远都是true，而这个Boolean类型正是做了这个box操作.

参考文档:
http://www.cnblogs.com/huangxincheng/category/633317.html
