## 全栈教学案例

## day-1 JQ1

### 1.1 jQuery入门

#### 1.1.1什么是 jQuery

jQuery是一个JavaScript库，它通过封装原生JavaScript函数得到一整套定义好的方法。

它是John Resig 于2006年创建的一个开源项目，随着越来越多开发者的加入，

jQuery已经集成了JavaScript、CSS、DOM、和Ajax于一体的强大功能，

它可以用最少的代码，完成更多复杂而困难的功能，从而得到了开发者的青睐。

主指：以更少的代码、实现更多的功能。（write less,do more)

#### 1.1.2学习 jQuery 的条件

Html css javascript

#### 1.1.3 jQuery 的版本

版本的版本号升级主要有三种：

第一种是大版本升级，比如1.x.x升级到2.x.x，这种升级规模是最大的，改动的地方是最多的，周期也是最长的，jQuery从1.x.x到2.x.x用了7年。

第二种是小版本更新，比如1.7升级到1.8，改动适中，增加或减少了一些功能，一般周期半年到一年左右。

第三种是微版本更新，比如1.8.1升级到1.8.2，修复一些bug或错误之类。

版本的内容升级主要也有三种：

第一种是核心库的升级，比如优化选择符、优化DOM或者AJAX等；这种升级不影响开发者的使用。

第二种是功能性的升级，比如剔除一些过时的方法、新增或增强一些方法等等；这种升级需要了解和学习。

第三种就是BUG修复之类的升级，对开发者使用没有影响。

学习者有一种担忧，比如学了1.3版本的jQuery，那么以后升级新版本是不是还需要重学？

没必要，因为并不是每次升级一个版本都会增加或剔除功能的，一半左右都是内部优化，

升级到新版本并不需要任何学习成本。就算在新的版本增加了一些功能，

只需要几分钟了解一下即可使用，无需清零之前的知识，只需后续累加。

当然，在早期的jQuery版本都创建了最常用的功能，而新版本中增加的功能，

也不是最常用的，无需立即学习，立马用起。

jQuery.fn中包含

jQuery.fn["jquery"],

jQuery.fn.jquery



#### 1.1.4 jQuery 的功能和优势

jQuery最大的优势，就是特别的方便。比如模仿CSS获取DOM，比原生的JavaScript要方便太多。并且在多个CSS设置上的集中处理非常舒服，而最常用的CSS功能又封装到单独的方法，感觉非常有心。

最重要的是jQuery的代码兼容性非常好，你不需要总是头疼着考虑不同浏览器的兼容问题。

#### 1.1.5下载及运行 jQuery

jQuery官网：http://www.jQuery.com

jQuery官网下载：http://jquery.com/download/

获取jQuery最新版本

进入jQuery官网，下载最新jQuery库文件。

1.x代表的是兼容ie6-8

2.x代表的是不兼容ie6-8

在页面中引入Jquery，格式如下：

<script type="text/javascript" src="../script/jquery.js></script>;

### 1.2基础核心

#### 1.2.1代码风格

在jQuery程序中，不管是页面元素的选择、内置的功能函数，

都是美元符号“$”来起始的。而这个“$”就是jQuery当中最重要且独有的对象：

jQuery对象，所以我们在页面元素选择或执行功能函数的时候可以这么写：

$(function(){});//执行一个匿名函数

$(‘#box’);//选择ID元素

$(‘#box’).css(‘color’,‘red’);//改变ID元素的样式

由于$本身就是jQuery对象的缩写形式，那么也就是说上面的三段代码也可以写成如下形式：

jQuery(function(){});

jQuery(‘#box’);

jQuery(‘#box’).css(‘color’,‘red’);

jQuery的代码模式是采用的连缀方式，可以不停的连续调用功能函数。

$('#box').css('color','red').css('font-size','50px');//连缀

jQuery中代码注释和JavaScript是保持一致的，

有两种最常用的注释：单行使用“//...”，多行使用“/\*...\*/”

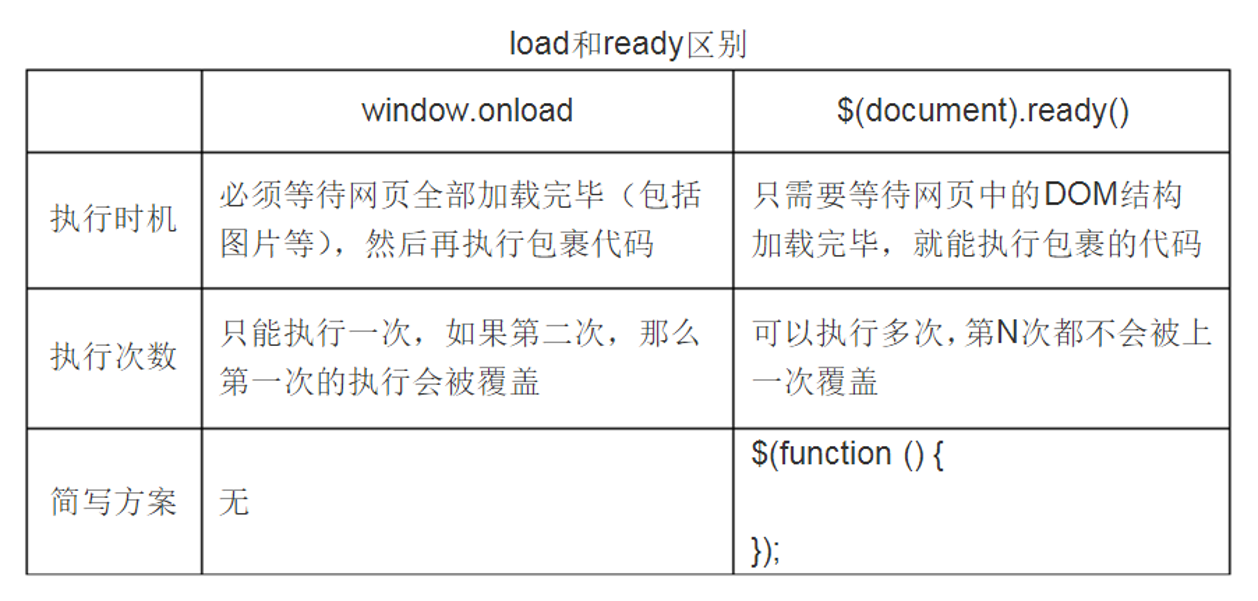
//$('#box').css('color','red');

#### 1.2.2加载模式

等待所有的DOM元素加载后，延迟DOM操作，方法如下：

window.onload=function(){};//JavaScript等待加载

$(document).ready(function(){});//jQuery等待加载



#### 1.2.3对象互换

//Js==>Jq

var jsDom1Jq=$(jsDom1);

console.log("jsDom1Jq:",jsDom1Jq);

//Jq==>Js

var jqDomJs1=jqDom[0];

var jqDomJs2=jqDom.get(0);

console.log("jqDomJs1:",jqDomJs1);

console.log("jqDomJs2:",jqDomJs2);

fgx("size和length");

var n=jsDom1Jq.size();//size()在1.8以后废弃

var le=jsDom1Jq.length;





### 1.3常规选择器

#### 1.3.1简单选择器

#### 1.3.2进阶选择器

#### 1.3.3高级选择器

$(this) 选取当前HTML元素

$("p.info") 选取class为info的p元素

$("p:first") 选取第一个p元素

$("[href]") 选取带有href属性的元素

$("a[target='\_blank']") 选择所有带target属性的值等于"\_blank"的a元素

$(":button") 选取所有type="button"的input元素和button元素

$("tr:even") 选取偶数位置的tr元素

$("tr:odd") 选取奇数位置的tr元素

### 1.4过滤选择器

#### 1.4.1基本过滤器

#### 1.4.2内容过滤器

#### 1.4.3可见性过滤器

#### 1.4.4子元素过滤器

http://www.w3cplus.com/css3/basic-selectors

http://www.w3cplus.com/css3/attribute-selectors

http://www.w3cplus.com/css3/pseudo-class-selector

### 1.5表单选择器

#### 1.5.1常规选择器

#### 1.5.2表单选择器

#### 1.5.3表单过滤器

1选择器

$(this) 选取当前HTML元素

$("p.info") 选取class为info的p元素

$("p:first") 选取第一个p元素

$("[href]") 选取带有href属性的元素

$("a[target='\_blank']") 选择所有带target属性的值等于"\_blank"的a元素

$(":button") 选取所有type="button"的input元素和button元素

$("tr:even") 选取偶数位置的tr元素

$("tr:odd") 选取奇数位置的tr元素

fgx("1.$(this)");

$("#d1")[0].onclick=function(){//会覆盖添加在行内的js

console.log("0",this);

}

$("#d1").click(function(){

console.log("2",$(this));

});

$(selector).find() find() 方法返回被选元素的后代元素，一路向下直到最后一个后代。\*所有

$(selector).find("p") find("p") 方法返回被选元素的后代P元素，一路向下直到最后一个后代。

//

fgx("2.find();find('p')");

var findR1=$(".box1").find("\*");

console.log("findR1:",findR1);

var findR2=$(".box1").find("p");

console.log("findR2:",findR2);

$(selector).children() children() 方法返回被选元素的所有直接子元素。

$(selector).children("p.c") 面的例子返回类名为 "c" 的所有&lt;p&gt; 元素，并且它们是 &lt;div&gt; 的直接子元素：

fgx("2.children();children('p.p2')");

var childrenR1=$(".box2").children();

console.log("childrenR1:",childrenR1);

var childrenR2=$(".box2").children("p.p2");

console.log("childrenR2:",childrenR2);

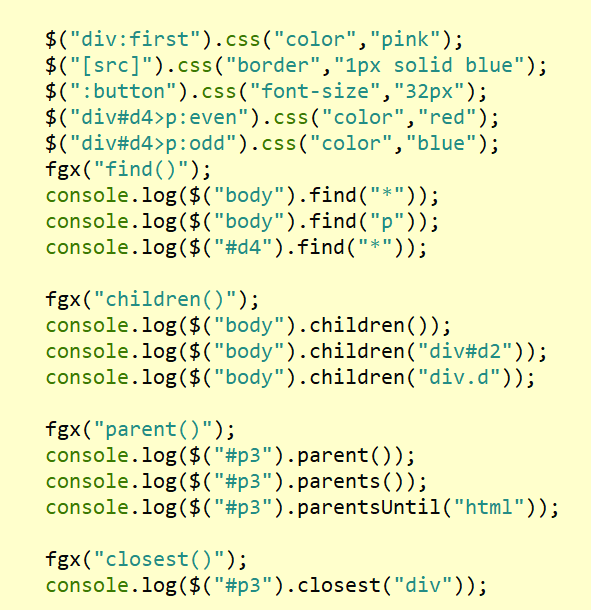
$(selector).parent() parent() 方法返回被选元素的直接父元素。

$(selector).parents() parents() 方法返回被选元素的所有祖先元素。

祖先是父、祖父、曾祖父，依此类推。

$(selector).parentsUntil() parentsUntil()方法返回介于 selector 与 stop 之间的所有祖先元素。

祖先是父、祖父、曾祖父，依此类推。



fgx("3.parent();parents();parentsUntil()");

var parentR1=$(".box2>.p3").parent();

console.log("parentR1:",parentR1);

var parentsR2=$(".box2>.p3").parents();

console.log("parentsR2:",parentsR2);

var parentsR3=$(".box2>.p3").parentsUntil(".box1");

console.log("parentsR3:",parentsR3);

$(selector).closest() 方法返回被选元素的第一个祖先元素。祖先是父、祖父、曾祖父，依此类推。必需。

规定缩小搜索祖先元素范围的选择器表达式、元素或 jQuery 对象。

fgx("4.closest()");//closest() 方法返回被选元素的第一个祖先元素。必需。

//规定缩小搜索祖先元素范围的选择器表达式、元素或 jQuery 对象。

var myCloseSt=$(".p1").closest(".box3");

console.log("myCloseSt:",myCloseSt);

$(selector).siblings() 方法返回被选元素的所有同胞元素

$(selector).siblings("p") 方法返回被选元素的所有同胞p元素

fgx("5.siblings() siblings('p')");

var sliblingsR1=$("p.p2").siblings();

console.log("sliblingsR1:",sliblingsR1);

var sliblingsR2=$("p.p2").siblings("p.p3");

console.log("sliblingsR2:",sliblingsR2);

$(selector).next(); next() 方法返回被选元素的下一个同胞元素。该方法只返回一个元素。

$(selector).next("p"); next() 方法返回被选元素的下一个同胞p元素。该方法只返回一个元素。

$(selector).nextAll(); nextAll() 方法返回被选元素的所有跟随的同胞元素。

$(selector).nextAll("p"); nextAll() 方法返回被选元素的所有跟随的同胞p元素。

$(selector).nextUntil() 方法返回介于两个给定参数之间的所有跟随的同胞元素。

$(selector).prev(); next() 方法返回被选元素的上一个同胞元素。该方法只返回一个元素。

$(selector).prev("p"); next() 方法返回被选元素的上一个同胞p元素。该方法只返回一个元素。

$(selector).prevAll(); nextAll() 方法返回被选元素的所有上面的同胞元素。

$(selector).prevAll("p"); nextAll() 方法返回被选元素的所有上面的同胞p元素。

$(selector).prevUntil() 方法返回介于两个给定参数之间的所有上面的同胞元素。

fgx("6.next() next('p') nextAll() nextUntil() nextUntil('.p3')");

var nextR1=$("p.p2").next();

console.log("nextR1:",nextR1);

var nextR2=$("p.p2").next("div");

console.log("nextR2:",nextR2);

var nextAllR3=$("p.p2").nextAll();

console.log("nextAllR3:",nextAllR3);

var nextAllR4=$("p.p2").nextAll("p.p3");

console.log("nextAllR4:",nextAllR4);

var nextUntilR5=$("p.p2").nextUntil();

console.log("nextUntilR5:",nextUntilR5);

var nextUntilR6=$("p.p2").nextUntil(".p3");

console.log("nextUntilR6:",nextUntilR6);



fgx("7.prev() prev('p') prevAll() prevUntil() prevUntil('.p2')");

$(selector).has() //选择子元素包含有selector的元素 //元素集合

var dHas=$("#box>div:has(.ipt)");

fgx("8.has()");

var eHas=$(".box2").has(".box4");

console.log("eHas:",eHas);

$(selector).is()

fgx("9.is()");

var isR1=$(".box2>p").is(".p3");//true

console.log("isR1:",isR1);

var isR2=$(".box2>p").is(".p4");

console.log("isR2:",isR2);

$(selector).hasClass(class)

fgx("10.hasClass(class)<==>is('.'+class)");//

var hasClassR1=$(".box2>p").hasClass("p3");//true

console.log("hasClassR1:",hasClassR1);

$(selector).slice(start,end)

///

fgx("11.slice(start,end)");//

var sliceR1=$(".box2>\*").slice(0,2);

console.log("sliceR1:",sliceR1);

var sliceR2=$(".box2>\*").slice(2,3);

console.log("sliceR2:",sliceR2);

var sliceR3=$(".box2>\*").slice(0,-1);

console.log("sliceR3:",sliceR3);

////

2.find();next();nextAll();children()

在 find()、next()、nextAll()和 children()这四个方法中，如果不传递参数，

就相当于传递 了“\*”，层次选择器对节点的层次都是有要求的，比如子选择器，

只有子节点才可以被选择到，孙子节点和重孙子节点都无法选择到。

next和nextAll选择器，必须是同一个层次的后一个和后N个，不在同一个层次就无法选取到了。

理论上来讲，jQuery 提供的方法 find()、next()、nextAll()和 children()运行速度要快于使用高级选择器。

因为他们实现的算法有所不同，高级选择器是通过解析字符串来获取节点对象，

而 jQuery 提供的方法一般都是单个选择器，是可以直接获取的。但这种快慢的差异，

对于客户端脚本来说没有太大的实用性，并且速度的差异还要取决了浏览器和选择的元素内容。

3.jQuery 还提供了更加丰富的方法来选择元素：

$('#box').prev('p').css('color','red'); //同级上一个元素

$('#box').prevAll('p').css('color', 'red'); //同级所有上面的元素

nextUntil()和 prevUnitl()方法是选定同级的下面或上面的所有节点，选定非指定的所有 元素，一旦遇到指定的元素就停止选定。

$('#box').prevUntil('p').css('color','red'); //同级上非指定元素选定，遇到则停止

$('#box').nextUntil('p').css('color','red'); //同级下非指定元素选定，遇到则停止

siblings()方法正好集成了 prevAll()和 nextAll()两个功能的效果，及上下相邻的所有元素 进行选定：

$('#box').siblings('p').css('color','red'); //同级上下所有元素选定

//等价于下面：

$('#box').prevAll('p').css('color','red'); //同级上所有元素选定

$('#box').nextAll('p').css('color','red'); //同级下所有元素选定

## day-2 JQ2

### 2.1基础 DOM 和 CSS 操作

#### 2.1.1 DOM 简介

#### 2.1.2设置元素及内容

#### 2.1.3元属性操作

#### 2.1.4元素样式操作

#### 2.1.5 CSS方法

1.基础DOM 和 CSS操作

1.DOM

文档对象模型（Document Object Model，简称DOM），

是W3C组织推荐的处理可扩展标志语言的标准编程接口。

在网页上，组织页面（或文档）的对象被组织在一个树形结构中，

用来表示文档中对象的标准模型就称为DOM。

2.设置元素及内容

html() 获取元素中HTML内容

html(value) 设置元素中HTML内容

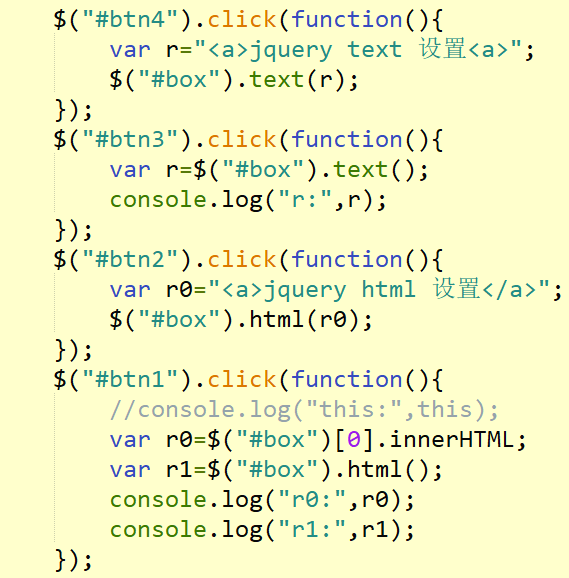
text() 获取元素中文本内容

text(value) 设置元素中文本内容

val() 获取表单中的文本内容

val(value) 设置表单中的文本内容





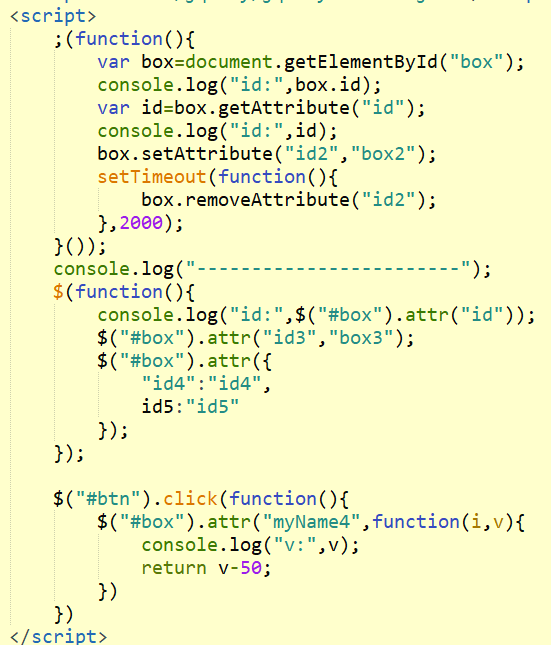
3.元素属性操作

attr(key) 获取某个元素key属性的属性值

attr(key,value) 设置某个元素key属性的属性值

attr({key1:value2,key2:value2}) 设置某个元素多个key属性的属性值

removeAttr("title") //删除指定的属性



4.元素样式操作

css(name) 获取某个元素CSS样式属性

css([name,name2,name3]) 获取某个元素多个CSS样式

css(name,value) 设置某个元素的CSS样式

css({name1:value1,name2:value2}) 设置某个元素多个样式





addClass(class) 给某个元素添加一个CSS类

addClass(class1 class2 class3) 给某个元素添加多个CSS类

removeClass(class) 删除某个元素的一个CSS类

removeClass(class1 class2 class3) 删除某个元素的多个CSS类

toggleClass(class) 来回切换默认样式和指定样式

toggleClass(class1 class2 class3) 来回切换默认样式和指定样式

toggleClass(class,switch) 来回切换样式的时候,switch可选

布尔值，规定是否仅仅添加（true）或移除（false）类。







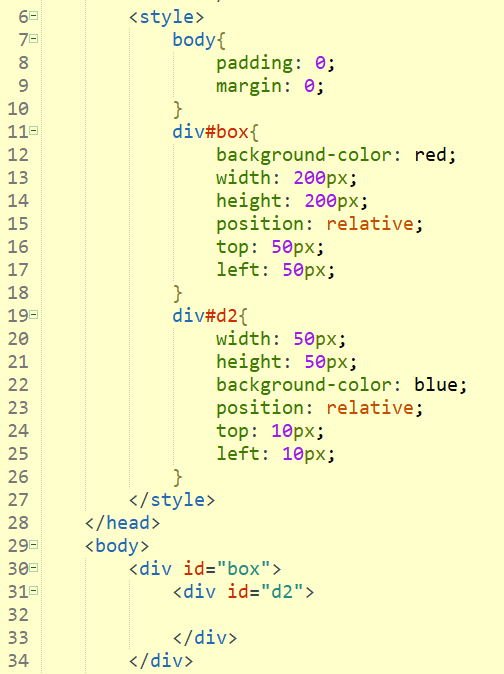
$(selector).offset() 方法设置或返回被选元素相对于<stong>文档</stong>的偏移坐标。

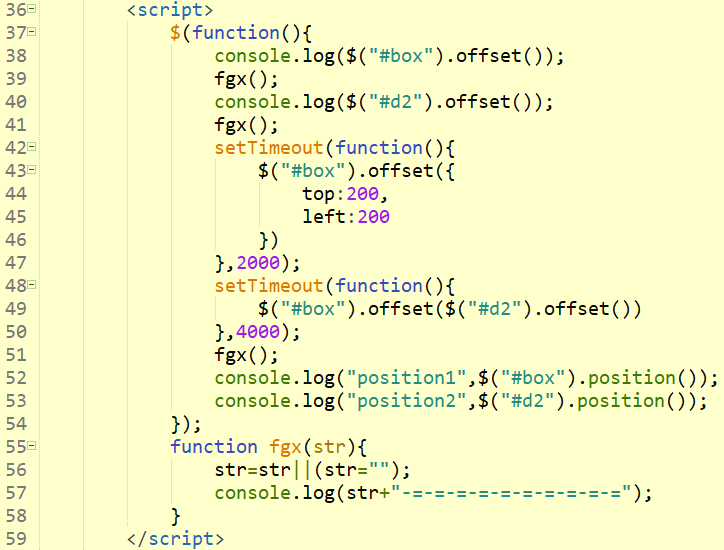
$(selector).offset({top:value,left:value})

$(selector).offset($(selector2).offset());

$(selector).position position() 方法返回第一个匹配元素的位置（相对于它的<strong>父元素</strong>）。

该方法返回一个带有两个属性（以像素为单位的 top 和 left 位置）的对象。





$(selector).scrollTop scrollTop() 方法设置或返回被选元素的垂直滚动条位置。

$(selector).scrollTop(value)

$(selector).scrollLeft() scrollLeft() 方法设置或返回被选元素的水平滚动条位置。

$(selector).scrollLeft(value)



1.attr()

attr() 方法设置或返回被选元素的属性和值。

当该方法用于返回属性值，则返回第一个匹配元素的值。

当该方法用于设置属性值，则为匹配元素设置一个或多个属性/值对。

返回属性的值：

$(selector).attr(attribute)

设置属性和值：

$(selector).attr(attribute,value)

设置多个属性和值：

$(selector).attr({attribute:value, attribute:value,...})

attribute 规定属性的名称。

value 规定属性的值。

2.removeAttr() 方法从被选元素移除一个或多个属性

$(selector).removeAttr(attribute)

attribute 必需。规定要移除的一个或多个属性。如需移除若干个属性，请使用空格分隔属性名称。

3.prop

prop() 方法设置或返回被选元素的属性和值。

当该方法用于返回属性值时，则返回第一个匹配元素的值。

当该方法用于设置属性值时，则为匹配元素集合设置一个或多个属性/值对。

注意：prop() 方法应该用于检索属性值，

返回属性的值：

$(selector).prop(property)

设置属性和值：

$(selector).prop(property,value)

设置多个属性和值：

$(selector).prop({property:value, property:value,...})

property 规定属性的名称。

value 规定属性的值。

4.removeProp() 方法移除由 prop() 方法设置的属性。

注意：不要使用该方法来移除诸如 style、id 或 checked 之类的 HTML 属性。请使用 removeAttr() 方法代替。

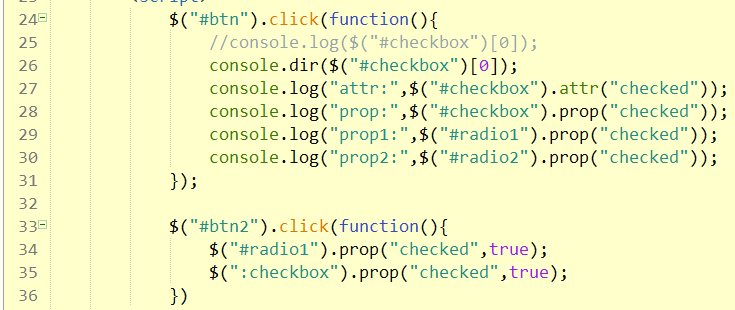
$(selector).removeProp(property)

property 规定要移除的属性的名称。

5.prop() 和 attr() 之间的不同

prop() 和 attr() 可能返回不同的值。本实例演示了当用于返回复选框的 "checked" 状态时的不同。

<b>注意:</b>确认或取消选中该复选框,然后单击按钮刷新内容。



each() 方法为每个匹配元素规定要运行的函数。

提示：返回 false 可用于及早停止循环。

语法

$(selector).each(function(index,element))

参数 描述

function(index,element) 必需。为每个匹配元素规定运行的函数。

index - 选择器的 index 位置。

element - 当前的元素（也可使用 "this" 选择器）。

//each可以遍历对象;数组;元素

$(function(){

var obj={

name:"kky",

age:20

};

//遍历对象

$.each(obj,function(k,v){

console.log("k:",k,"v:",v);

});

console.log("-----------------");

for(var k in obj){

console.log("k:",k,"v:",obj[k]);

}

console.log("-----------------");

var arr=["a","b","c","d"];

//数组

$.each(arr,function(k,v){

console.log("k:",k,typeof k,"v:",v);

});

console.log("-----------------");

for(var k in arr){

console.log("k:",k,typeof k,"v:",arr[k]);

}

console.log("-----------------");

arr.forEach(function(i,k,a){

console.log("i:",i,"k:",k,"a:",a);

});

console.log("-----------------");

//jQuery元素对象

$("div").each(function(k,v){

console.log("k:",k,"v:",v);

})

})



### 2.2 DOM 节点操作

#### 2.2.1创建节点

#### 2.2.2插入节点

#### 2.2.3包裹节点

#### 2.2.4节点操作

1.创建节点

append() 方法在被选元素的结尾插入指定内容。

$(selector).append(content,function(index,html))

content 必需。规定要插入的内容（可包含 HTML 标签）。

可能的值：

HTML 元素

jQuery 对象

DOM 元素

function(index,html) 可选。规定返回待插入内容的函数。

index - 返回集合中元素的 index 位置。

html - 返回被选元素的当前 HTML

////

//append

var d1Btn1=$("#d1>button:nth-child(2)");

var d1Btn2=$("#d1>button:nth-child(3)");

console.log(d1Btn1,d1Btn2);

var d1Div=$("#d1>div");

d1Btn1.click(function(){

var txt1="append追加文本1";//字符串

var txt2="&lt;div>append追加文本2&lt;/div>";//HTML标签文本

var txt3=$("&lt;div>&lt;/div>").text("追加文本3");//使用jQuery

var txt4=document.createElement("p");

txt4.innerHTML="&lt;div>追加文本4&lt;/div>";//使用DOM

d1Div.append(txt1,txt2,txt3,txt4);

});

d1Btn2.click(function(){

d1Div.append(function(i,n){

console.log("i:",i,"n:",n);

return "&lt;span>append(function(){})&lt;/span>"+i+" "+n;

});

});

////

prepend() 方法在被选元素的开头插入指定内容。

$(selector).prepend(content,function(index,html))

content 必需。规定要插入的内容（可包含 HTML 标签）。

可能的值：

HTML 元素

jQuery 对象

DOM 元素

function(index,html) 可选。规定返回待插入内容的函数。

index - 返回集合中元素的 index 位置。

html - 返回被选元素的当前 HTML。

2.插入节点

appendTo() 方法在被选元素的结尾插入 HTML 元素

$(content).appendTo(selector)

content 必需。规定要插入的内容（必须包含 HTML 标签）。

注意：如果 content 是已存在的元素，它将从当前位置被移除，并在被选元素的结尾被插入。

selector 必需。规定把内容追加到哪个元素上。

////

//appendTo

var d2=$("#d2");

var d2Btn=$("#d2>button:nth-child(2)");

d2Btn.click(function(){

$("&lt;div>appendTo&lt;/div>").appendTo("#d2>div:nth-child(1)");

});

////



prependTo() 方法在被选元素的开头插入 HTML 元素。

$(content).prependTo(selector)

content 必需。规定要插入的内容（必须包含 HTML 标签）。

注意：如果 content 是已存在的元素，它将从当前位置被移除，并在被选元素的开头被插入。

selector 必需。规定把内容预加到哪个元素上。

after() 方法在被选元素后插入指定的内容

$(selector).after(content,function(index))

content 必需。规定要插入的内容（可包含 HTML 标签）。

可能的值：

HTML 元素

jQuery 对象

DOM 元素

function(index) 规定返回待插入内容的函数。

index - 返回集合中元素的 index 位置。

////

//after

var d3Div=$("#d3>div:nth-child(1)");

var d3Btn=$("#d3>button:nth-child(2)");

var d3Btn2=$("#d3>button:nth-child(3)");

d3Btn.click(function(){

var txt1="after文本1";

var txt2="<div>after文本2</div>";

var txt3=$("<div></div>").text("after文本3");

var txt4=document.createElement("p");

txt4.innerHTML="after文本4";

d3Div.after(txt1,txt2,txt3,txt4);

});

d3Btn2.click(function(){

d3Div.after(function(i,n){

console.log("i:",i,"n:",n);

return "<div>after文本>/div>"+i+" "+ n;

});

});

////

before() 方法在被选元素之前插入指定的内容。

$(selector).before(content,function(index))

content 规定要插入的内容（可包含 HTML 标签）。

可能的值：

HTML 元素

jQuery 对象

DOM 元素

function(index) 可选。规定返回待插入内容的函数。

index - 返回集合中元素的 index 位置。

insertAfter() 方法在被选元素后插入 HTML 元素。

$(content).insertAfter(selector)

content 必需。规定要插入的内容（必须包含 HTML 标签）。

注意：如果 content 是已存在的元素，它将从它的当前位置被移除，并被插入在被选元素之后。

selector 必需。规定在何处插入内容。

////

//insertAfter

var d4Div=$("#d4>div:nth-child(1)");

var d4Btn=$("#d4>button:nth-child(2)");

var d4Div2=$("#d4>div:nth-child(3)");

d4Btn.click(function(){

$("<div>insertAfter文本</div>").insertAfter(d4Div);

$(d4Div2).insertAfter(d4Div);//类似与剪切

})

////

insertBefore() 方法在被选元素前插入 HTML 元素。

$(content).insertBefore(selector)

content 必需。规定要插入的内容（必须包含 HTML 标签）。

注意：如果 content 是已存在的元素，它将从它的当前位置被移除，并被插入在被选元素之前。

selector 必需。规定在何处插入内容。

3.包裹节点

4.节点操作

4.节点操作

clone() 方法生成被选元素的副本，包含子节点、文本和属性。

$(selector).clone(true|false)

true 规定需复制事件处理程序。

false 默认。规定不复制事件处理程序。

//1.clone();

var d1ClsD1=$("#d1>.d1");

var d1ClsD2=$("#d1>.d2");

var d1btn=$("#d1btn");

d1ClsD1.click(function(){

$(this).css("font-size",function(i,v){

return parseInt(v)+1;

})

})

var flag=false;

d1btn.click(function(){

var r=d1ClsD1.clone(flag);

$(this).html("clone("+!flag+")");

d1ClsD2.append(r);

if(flag){

flag=false;

}else{

flag=true;

}

})

remove() 方法移除被选元素，包括所有的文本和子节点。

该方法也会移除被选元素的数据和事件。

$(selector).remove();

//2.remove();

var d2ClsD2=$("#d2>.d1");

console.log("d2ClsD2:",d2ClsD2);

var d2btn=$("#d2btn");

d2btn.click(function(){

var r=d2ClsD2.remove();

console.log("r:",r);

});

detach() 方法移除被选元素，包括所有的文本和子节点。然后它会保留数据和事件。

该方法会保留移除元素的副本，允许它们在以后被重新插入。

$(selector).detach();

//2.detach();

var d2ClsD1=$("#d2>.d1");

console.log("d2ClsD2:",d2ClsD2);

var d2btn2=$("#d2btn2");

var d2btn3=$("#d2btn3");

var r;

d2btn2.click(function(){

r=d2ClsD2.detach();

console.log("r:",r);

});

d2btn3.click(function(){

$("#d2").prepend(r);

});

//2.detach() remove()

var d2ClsD2=$("#d2>.d1>.d2");

var d2ClsD1=$("#d2>.d1");

d2ClsD2.click(function(){

$(this).css("font-size",function(i,v){

return parseInt(v)+1;

})

})

var d2btn4=$("#d2btn4");

var d2btn5=$("#d2btn5");

d2btn4.click(function(){

var tmp=d2ClsD1.detach();

$("#d2").append(tmp);

});

d2btn5.click(function(){

var tmp=d2ClsD1.remove();

$("#d2").append(tmp);

});

empty() 方法从被选元素所有子节点和内容。

注意：该方法不会移除元素本身，或它的属性。

$(selector).empty()

//2.empty

var d2btn6=$("#d2btn6");

var d2ClsD1=$("#d2>.d1");

d2btn6.click(function(){

d2ClsD1.empty();

});



replaceWith() 方法把被选元素替换为新的内容。

$(selector).replaceWith(content,function(index))

content 必需。规定要插入的内容（可包含 HTML 标签）。

可能的值：

HTML 元素

jQuery 对象

DOM 元素

function(index) 可选。规定返回替换内容的函数。

index - 返回集合中元素的 index 位置。



### 2.3基础事件

#### 2.3.1绑定事件

#### 2.3.2简写事件

#### 2.3.3复合事件

<pre>

1.绑定事件

JavaScript 有一个非常重要的功能，就是事件驱动。当页面完全加载后，

用户通过鼠标 或键盘触发页面中绑定事件的元素即可触发。

jQuery 为开发者更有效率的编写事件行为，封 装了大量有益的事件方法供我们使用。

在 JavaScript 课程的学习中，我们掌握了很多使用的事件，

常用的事件有：click、dblclick、 mousedown、mouseup、mousemove、

mouseover、mouseout、change、select、submit、keydown、

keypress、keyup、blur、focus、load、resize、scroll。

那么，还有更多的事件可以参考 手册中的事件部分。

jQuery 通过on(),bind(),live(),delegate()方法来为元素绑定这些事件

自 jQuery 版本 1.7 起，on() 方法是 bind()、live() 和 delegate() 方法的新的替代品。

该方法给 API 带来很多便利，我们推荐使用该方法，它简化了 jQuery 代码库。

主要讲解on()

on() 方法在被选元素及子元素上添加一个或多个事件处理程序。

自 jQuery 版本 1.7 起，on() 方法是 bind()、live() 和 delegate() 方法的新的替代品。

该方法给 API 带来很多便利，我们推荐使用该方法，它简化了 jQuery 代码库。

注意：使用 on() 方法添加的事件处理程序适用于当前及未来的元素（比如由脚本创建的新元素）。

提示：如需移除事件处理程序，请使用 off() 方法。

提示：如需添加只运行一次的事件然后移除，请使用 one() 方法。

$(selector).on(event,childSelector,data,function)

参数 描述

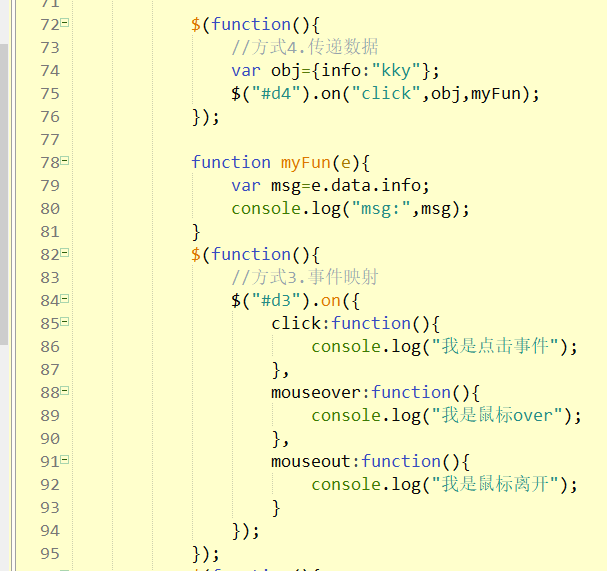
event 必需。规定要从被选元素移除的一个或多个事件或命名空间。

由空格分隔多个事件值，也可以是数组。必须是有效的事件。

childSelector 可选。规定只能添加到指定的子元素上的事件处理程序（且不是选择器本身，比如已废弃的 delegate() 方法）。

data 可选。规定传递到函数的额外数据。

function 可选。规定当事件发生时运行的函数。



<pre>

1.绑定事件

jQuery off()

off() 方法通常用于移除通过 on() 方法添加的事件处理程序。

自 jQuery 版本 1.7 起，off() 方法是 unbind()、die() 和 undelegate() 方法的新的替代品。

该方法给 API 带来很多便利，我们推荐使用该方法，它简化了 jQuery 代码库。

注意：如需移除指定的事件处理程序，当事件处理程序被添加时，

选择器字符串必须匹配 on() 方法传递的参数。

提示：如需添加只运行一次的事件然后移除，请使用 one() 方法。

$(selector).off(event,selector,function(eventObj),map)

参数 描述

event 必需。规定要从被选元素移除的一个或多个事件或命名空间。

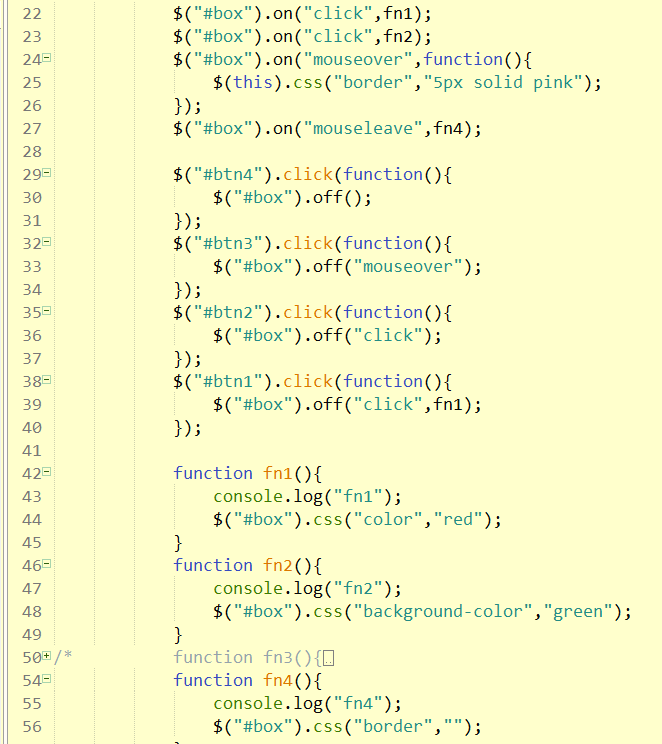
由空格分隔多个事件值。必须是有效的事件。

selector 可选。规定添加事件处理程序时最初传递给 on() 方法的选择器。

function(eventObj) 可选。规定当事件发生时运行的函数。

map 规定事件映射 ({event:function, event:function, ...})，

包含要添加到元素的一个或多个事件，以及当事件发生时运行的函数。



### 2.4事件对象

#### 2.4.1事件对象

#### 2.4.2冒泡和默认行为



### 2.5高级事件

#### 2.5.1模拟操作

#### 2.5.2命名空间

#### 2.5.3事件委托

#### 2.5.4 on、off 和 one

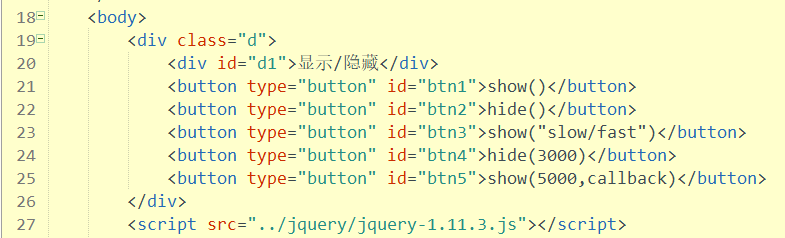
### 2.6案例：

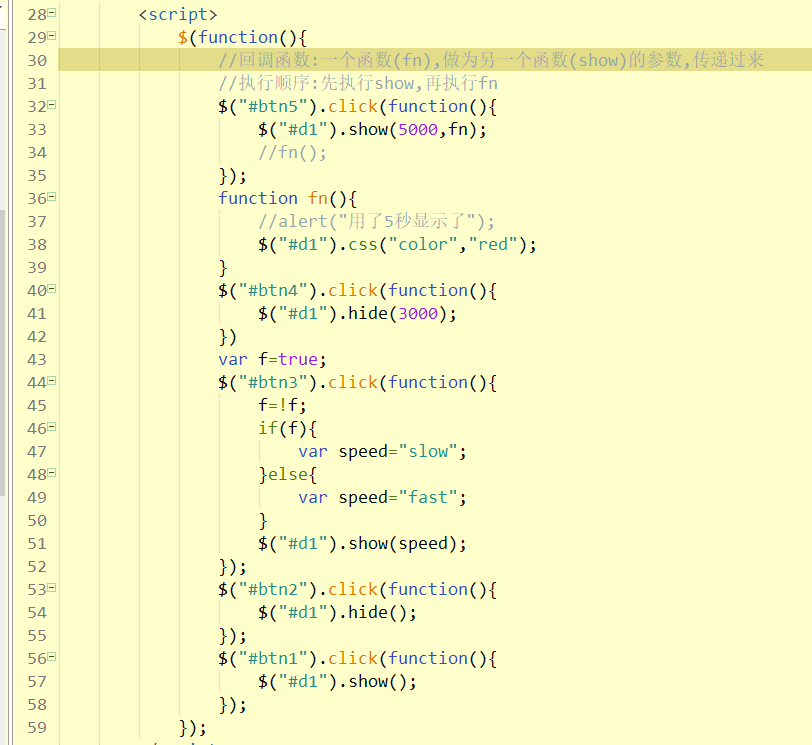
#### 2.6.1数据操作

## day-3 JQ3

### 3.1动画效果

#### 3.1.1显示、隐藏





#### 3.1.2滑动、卷动



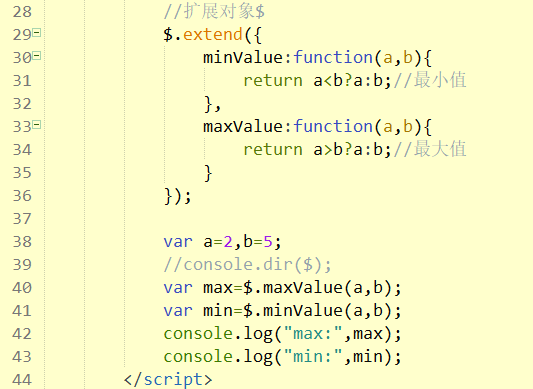


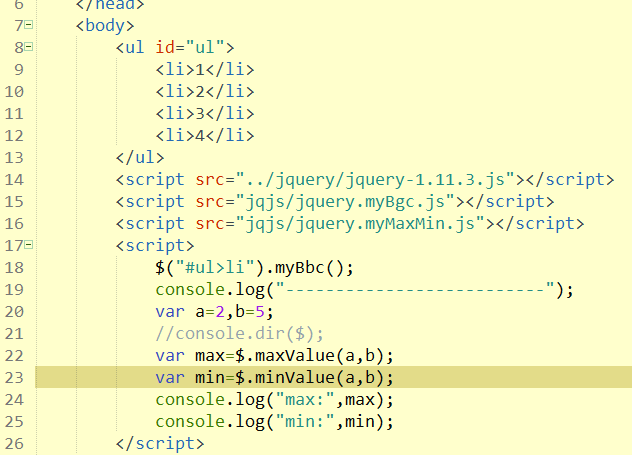
#### 3.1.3淡入、淡出

### 3.2插件开发

#### 3.2.1插件概述

#### 3.2.2自定义插件





## day-4 JQ4 项目1

### 4.1项目练习1

## day-5 JQ5 项目2

### 5.1项目练习2

## day-6 Node.js的简介

### 6.1 Node.js简介

#### 6.1.1 什么是Node

Node.js是一个基于 Chrome V8 引擎的 JavaScript 运行时。

Node.js 使用高效、轻量级的事件驱动、非阻塞 I/O 模型。

它的包生态系统，npm，是目前世界上最大的开源库生态系统。

node比chrome浏览器少了BOM,DOM,window

简单的说 Node.js 就是运行在服务端的 JavaScript。

Node.js 是一个基于Chrome JavaScript 运行时建立的一个平台。

Node.js是一个事件驱动I/O服务端JavaScript环境，基于Google的V8引擎，

#### 6.1.2 Node的用途

Node.js适合用来开发什么样的应用程序呢？

当应用程序需要处理大量并发的I/O，而在向客户端发出响应之前，

应用程序内部并不需要进行非常复杂的处理的时候，Node.js非常适合。

Node.js也非常适合与web socket配合，开发长连接的实时交互应用程序。

比如：

● 用户表单收集

● 考试系统

● 聊天室

#### 6.1.3 Node的特点

单线程;非阻塞I/O;事件驱动

#### 6.1.4 Node的历史

#### 6.1.5 Node的核心模块

### 6.2 Node的安装

#### 6.2.1官网<http://nodejs.org>

#### 6.2.2 Node的安装

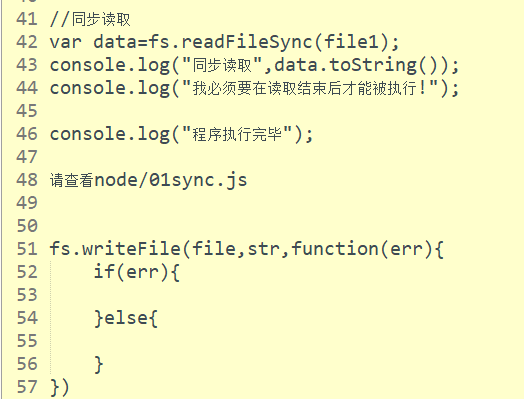
#### 6.2.3 Node的基本命令

### 6.3 Node的fs模块

#### 6.3.1读取文件(同步读取|异步读取)

#### 6.3.2 Node向文件写内容(同步写入|异步写入



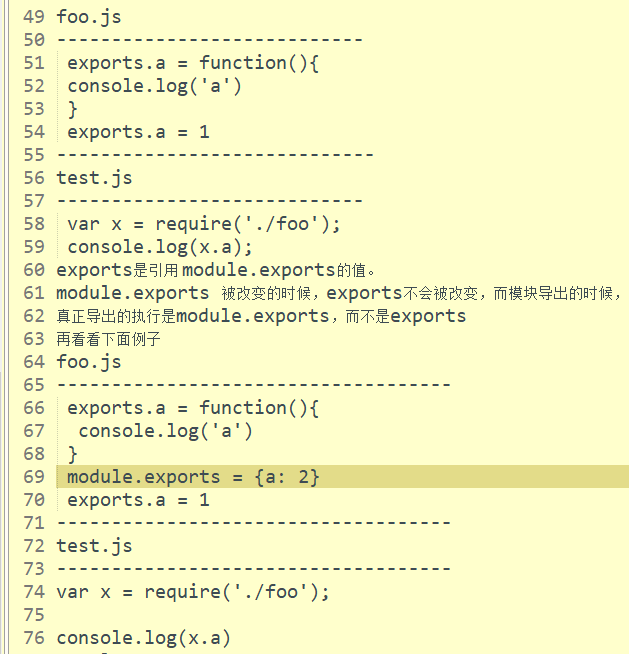


## Day-07 Node.js

### 7.1Node.js 模块

#### 7.1.1 nodejs中的模块化

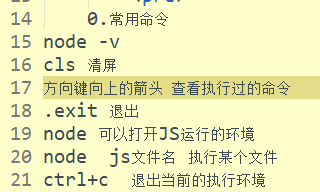




### 7.2 Node.js 包和NPM

#### 7.2.1 nodejs包的概念

#### 7.2.2 npm常用命令



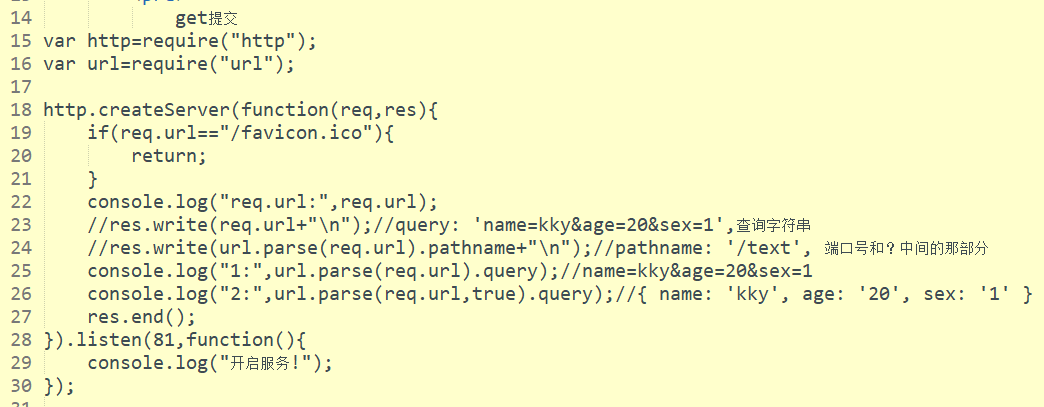
### 7.3 Node.js 常用模块

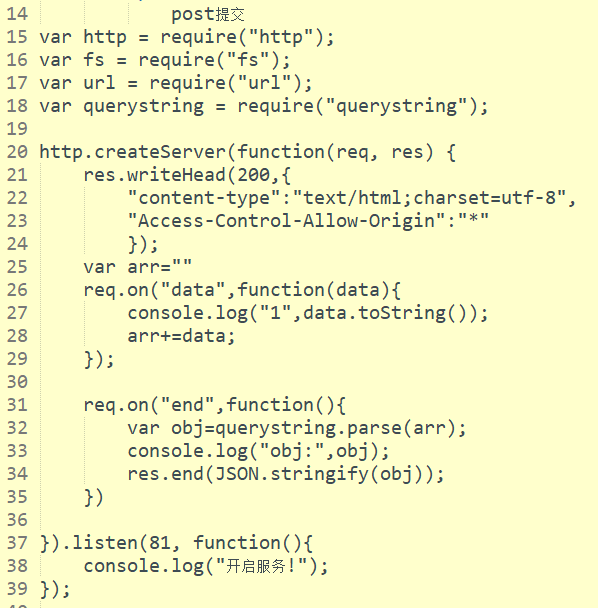
#### 7.3.1 http模块



#### 7.3.2 url模块

#### 7.3.3 querystring模块





404与自定义服务支持js,css

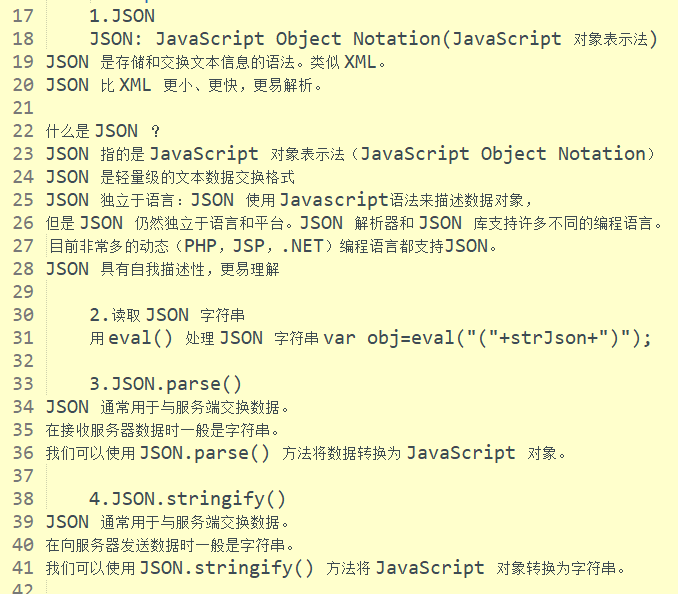


## Day-08 原生Ajax

### 8.1 Json

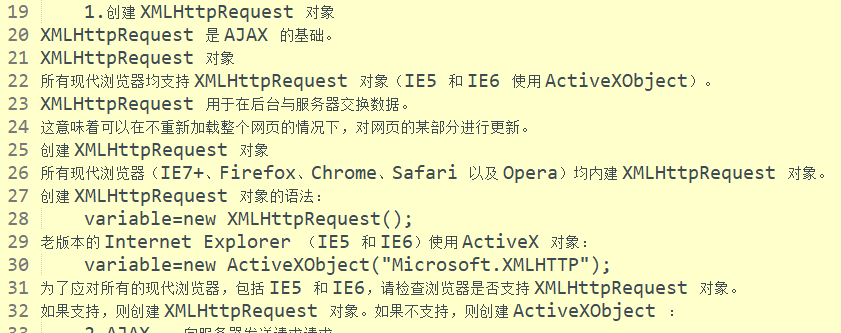
#### 8.1.1 JSON 语法

#### 8.1.2 解析和序列化



### 8.2 Ajax概述

#### 8.2.1 什么是ajax

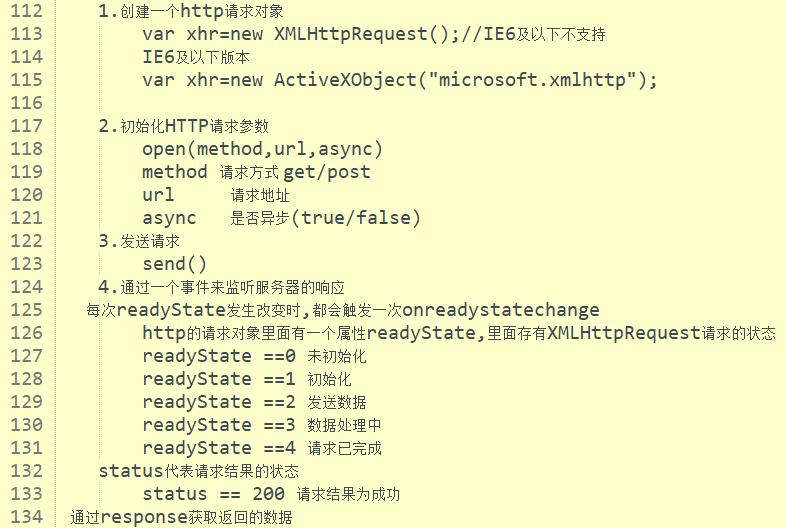


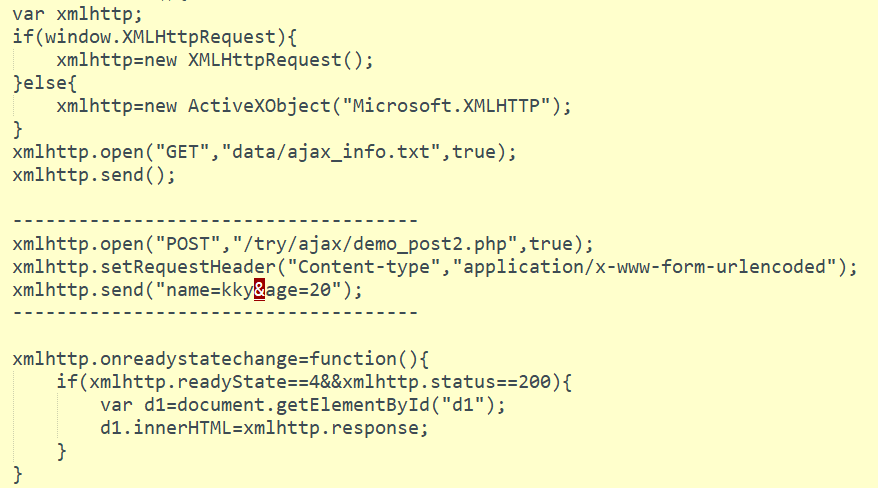
#### 8.2.2 ajax优点

### 8.3 XMLHttpRequest

#### 8.3.1 创建xhr对象

#### 8.3.2 xhr对象创建的兼容问题





### 8.4 GET与POST

#### 8.4.1 http协议

1.http和https

https的SSL加密是在传输层实现的。

(1)http和https的基本概念

http: 超文本传输协议，是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），

用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

https: 是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

https协议的主要作用是：建立一个信息安全通道，来确保数组的传输，确保网站的真实性。

(2)http和https的区别？

http传输的数据都是未加密的，也就是明文的，网景公司设置了SSL协议来对http协议传输的数据进行加密处理，

简单来说https协议是由http和ssl协议构建的可进行加密传输和身份认证的网络协议，比http协议的安全性更高。

主要的区别如下：

Https协议需要ca证书，费用较高。

http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

使用不同的链接方式，端口也不同，一般而言，http协议的端口为80，https的端口为443

http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

(3)https协议的工作原理

客户端在使用HTTPS方式与Web服务器通信时有以下几个步骤，如图所示。

客户使用https url访问服务器，则要求web 服务器建立ssl链接。

web服务器接收到客户端的请求之后，会将网站的证书（证书中包含了公钥），返回或者说传输给客户端。

客户端和web服务器端开始协商SSL链接的安全等级，也就是加密等级。

客户端浏览器通过双方协商一致的安全等级，建立会话密钥，然后通过网站的公钥来加密会话密钥，并传送给网站。

web服务器通过自己的私钥解密出会话密钥。

web服务器通过会话密钥加密与客户端之间的通信。

(4)https协议的优点

使用HTTPS协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；

HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全，

可防止数据在传输过程中不被窃取、改变，确保数据的完整性。

HTTPS是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

谷歌曾在2014年8月份调整搜索引擎算法，并称“比起同等HTTP网站，采用HTTPS加密的网站在搜索结果中的排名将会更高”。

(5)https协议的缺点

https握手阶段比较费时，会使页面加载时间延长50%，增加10%~20%的耗电。

https缓存不如http高效，会增加数据开销。

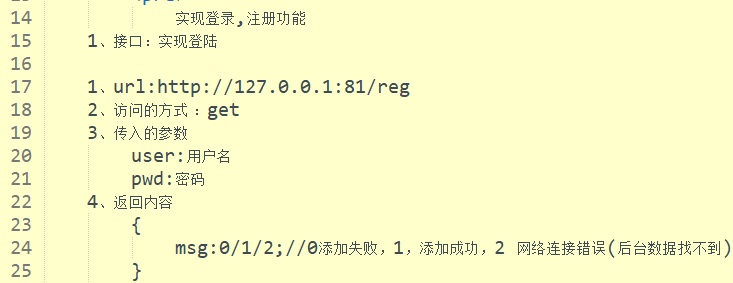
SSL证书也需要钱，功能越强大的证书费用越高。

SSL证书需要绑定IP，不能再同一个ip上绑定多个域名，ipv4资源支持不了这种消耗。

#### 8.4.2 get方式请求接口

#### 8.4.3 post请求接口



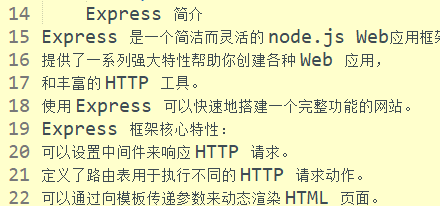


### 8.5 封装Ajax

## Day-09 Express

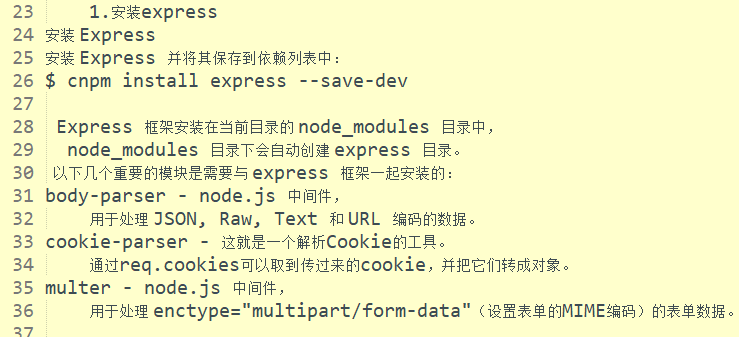
### 9.1 Express基础

#### 9.1.1 什么是Express

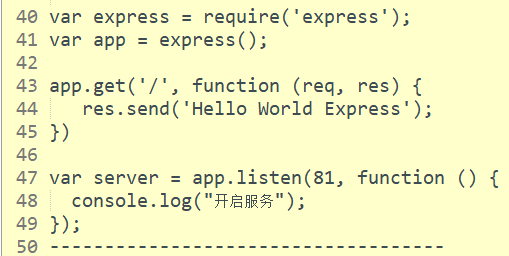


#### 9.1.2 官网http://www.expressjs.com.cn

#### 9.1.3 安装express



#### 9.1.4 express的Helloworld



#### 9.1.7 express的脚手架

#### 9.1.6 express的基本路由

路由

通过HTTP 请求的基本应用，知道了路由决定了由谁(指定脚本)去响应客户端请求。

在HTTP请求中，我们可以通过路由提取出请求的URL以及GET/POST参数。

路由是指如何定义应用的端点（URIs）以及如何响应客户端的请求

-------------------------------------------------------

var express=require("express");

var app=express();

//主页输出这是主页

app.get("/",function(req,res){

console.log("这是主页的GET请求!");

res.send("index Get");

});

// 网站首页接受 POST 请求

app.post("/",function(req,res){

console.log("这是主页的POST请求!");

res.send("index Post");

});

app.get("/del",function(req,res){

console.log("这是del删除界面");

res.send("del");

});

app.get("/user",function(req,res){

console.log("这是用户界面");

res.send("user");

});

//正则表达 abcd ,abxcd,ab123cd,等响应GET请求

app.get("/ab\*cd",function(req,res){

console.log("正则表达式");

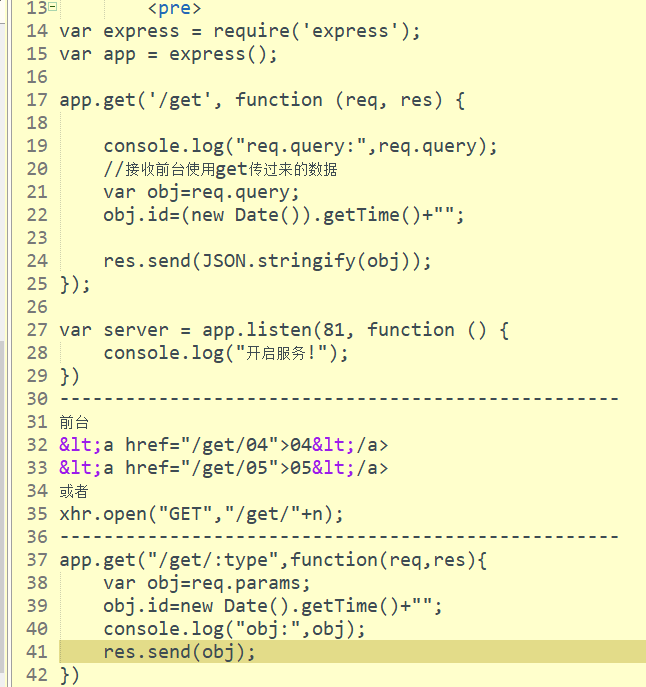
res.send("Reg");

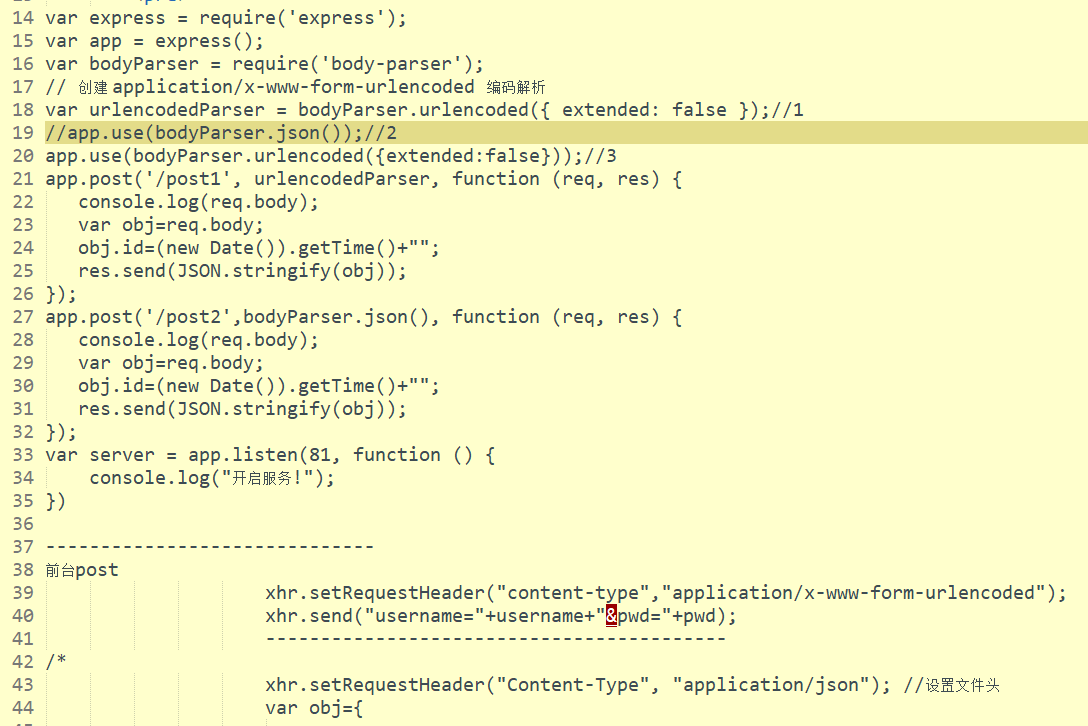
});

var server=app.listen(81,function(){

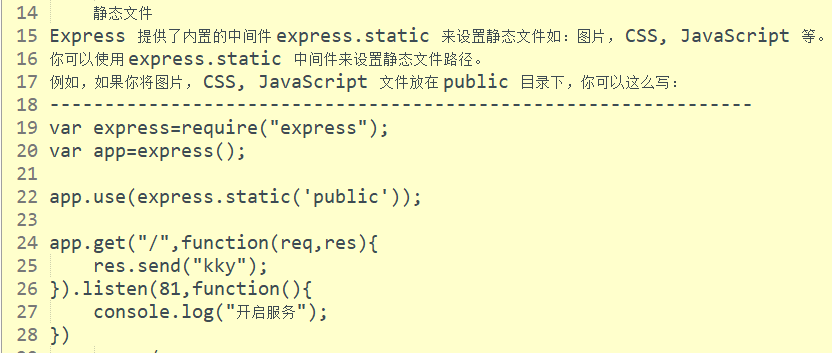
console.log("开启服务");

})





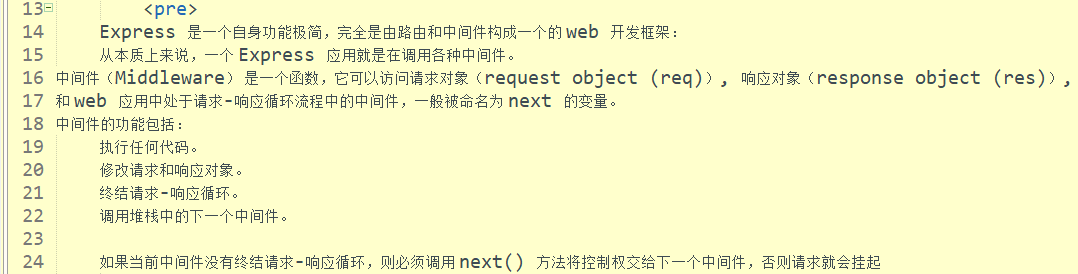
#### 9.1.7静态文件



### 9.2 Express核心

#### 9.2.1路由

#### 9.2.2中间件

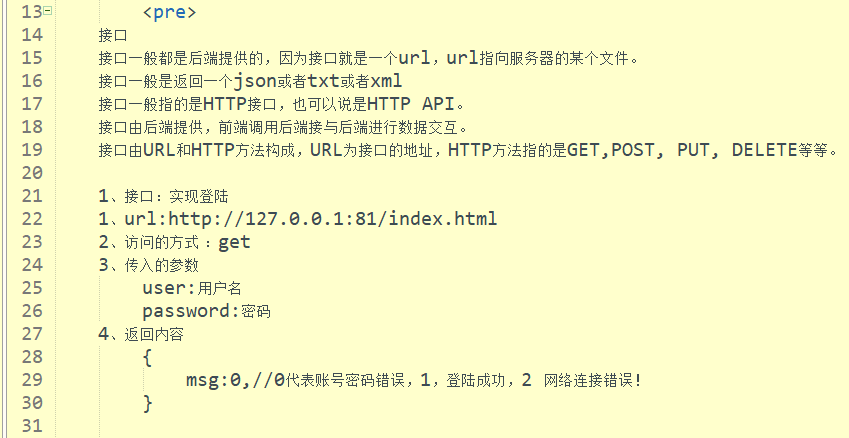


#### 9.2.3模板引擎

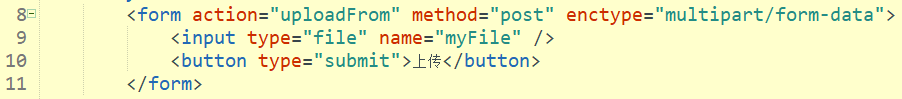
#### 9.2.4错误处理

### 9.4 案例

### 9.4.1定义接口服务



### 9.4.2表单上传

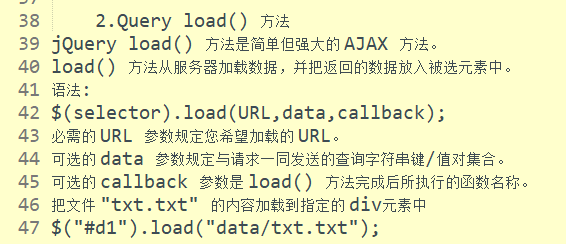




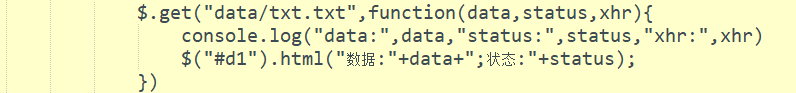
## Day-10 jQuery中的Ajax和跨域

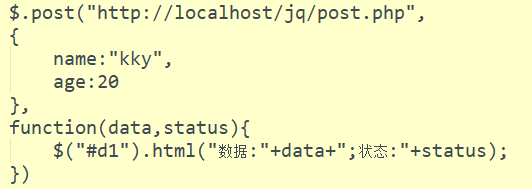
### 10.1 JQ中的AJAX

#### 10.1.1 $.load()方法

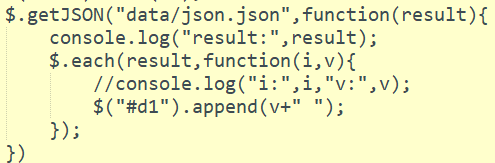


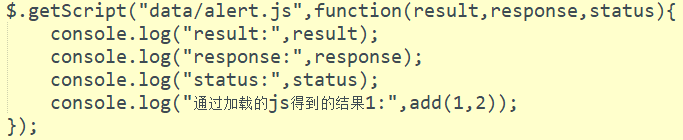
#### 10.1.2 $.get()和$.post()





#### 10.1.3 $.getScript()和$.getJSON()





#### 10.1.4 $.ajax()方法

ajax() 方法用于执行 AJAX（异步 HTTP）请求。

所有的 jQuery AJAX 方法都使用 ajax() 方法。该方法通常用于其他方法不能完成的请求。

语法

$.ajax({name:value, name:value, ... })

该参数规定 AJAX 请求的一个或多个名称/值对。

下面的表格中列出了可能的名称/值：

名称 值/描述

async 布尔值，表示请求是否异步处理。默认是 true。

beforeSend(xhr) 发送请求前运行的函数。

cache 布尔值，表示浏览器是否缓存被请求页面。默认是 true。

complete(xhr,status) 请求完成时运行的函数（在请求成功或失败之后均调用，即在 success 和 error 函数之后）。

contentType 发送数据到服务器时所使用的内容类型。默认是："application/x-www-form-urlencoded"。

context 为所有 AJAX 相关的回调函数规定 "this" 值。

data 规定要发送到服务器的数据。

dataFilter(data,type) 用于处理 XMLHttpRequest 原始响应数据的函数。

dataType 预期的服务器响应的数据类型。

error(xhr,status,error) 如果请求失败要运行的函数。

global 布尔值，规定是否为请求触发全局 AJAX 事件处理程序。默认是 true。

ifModified 布尔值，规定是否仅在最后一次请求以来响应发生改变时才请求成功。默认是 false。

jsonp 在一个 jsonp 中重写回调函数的字符串。

jsonpCallback 在一个 jsonp 中规定回调函数的名称。

password 规定在 HTTP 访问认证请求中使用的密码。

processData 布尔值，规定通过请求发送的数据是否转换为查询字符串。默认是 true。

scriptCharset 规定请求的字符集。

success(result,status,xhr) 当请求成功时运行的函数。

timeout 设置本地的请求超时时间（以毫秒计）。默认值为0,表示永不超时

traditional 布尔值，规定是否使用参数序列化的传统样式。

type 规定请求的类型（GET 或 POST）。

url 规定发送请求的 URL。默认是当前页面。

username 规定在 HTTP 访问认证请求中使用的用户名。

xhr 用于创建 XMLHttpRequest 对象的函数。

&lt;script>

$("#btn").click(function(){

var obj;

var n=parseInt(Math.random()\*2);

//var url=n>0?"data/getJson.json":"空";

if(n!=0){

url="data/getJson.json";

}else{

url="getJson.json";

console.log("0\_1",this);

obj={

"name":"kky"

}

}

console.log("0\_2此时n:",n,"url:",url,"obj:",obj);

$.ajax({

//url:"data/getJson.json",

url:url,//当url为空时获取的是的当前页面

async:"true",//布尔值，表示请求是否异步处理。默认是 true。

cache:true,//布尔值，表示浏览器是否缓存被请求页面。默认是 true。

complete:function(xhr,status){

console.log("1:请求完成时运行的函数（在请求成功或失败之后均调用，即在 success 和 error 函数之后）。","\nxhr:",xhr,"\nstatus:",status);

console.log("1\_1:this",this);

},

context:obj,//为所有 AJAX 相关的回调函数规定 "this" 值。

//data: "para="+para, //此处data可以为 a=1&b=2类型的字符串 或 json数据。

/\*

dataFilter给Ajax返回的原始数据的进行预处理的函数。

它的调用在success之前。

提供data和type两个参数：data是Ajax返回的原始数据，

type是调用jQuery.ajax时提供的dataType参数。

函数返回的值将由jQuery进一步处理。

\* \*/

//dataFilter:function(data,type){//此处不用空时,success:function(){}将不再运行.

//console.log("2:用于处理 XMLHttpRequest 原始响应数据的函数。","\ndata",data,"\ntype:",type);

//},

/\*

可选。规定预期的服务器响应的数据类型。

默认地，jQuery 会智能判断。

可能的类型：

"xml" - 一个 XML 文档

"html" - HTML 作为纯文本

"text" - 纯文本字符串

"script" - 以 JavaScript 运行响应，并以纯文本返回

"json" - 以 JSON 运行响应，并以 JavaScript 对象返回

"jsonp" - 使用 JSONP 加载一个 JSON 块，将添加一个 "?callback=?" 到 URL 来规定回调

\* \*/

dataType:"json",//预期的服务器响应的数据类型。

beforeSend:function(xhr){

console.log("3:发送请求前运行的函数。","\nxhr:",xhr);

},

success:function(data){

console.log("4:获取数据成功!\ndata:",data);

},

error:function(xhr,status,error){

console.log("5:如果请求失败要运行的函数,\nxhr:",xhr,"\nstatus:",status,"\nerror:",error);

},

timeout:0,//设置本地的请求超时时间（以毫秒计）。

/\*\*

\*1.网络不通畅。

2.后台运行比较慢（服务器第一次运行时，容易出现）

超时结果：JQ中 timeout设置请求超时时间。

如果服务器响应时间超过了 设置的时间，则进入 ERROR （错误处理）

超时解决方案：

1.默认的timeout为0，代表永不超时

2.尽量把timeout设置的大一点。坏处：使数据的加载变慢（貌似。呵呵）。

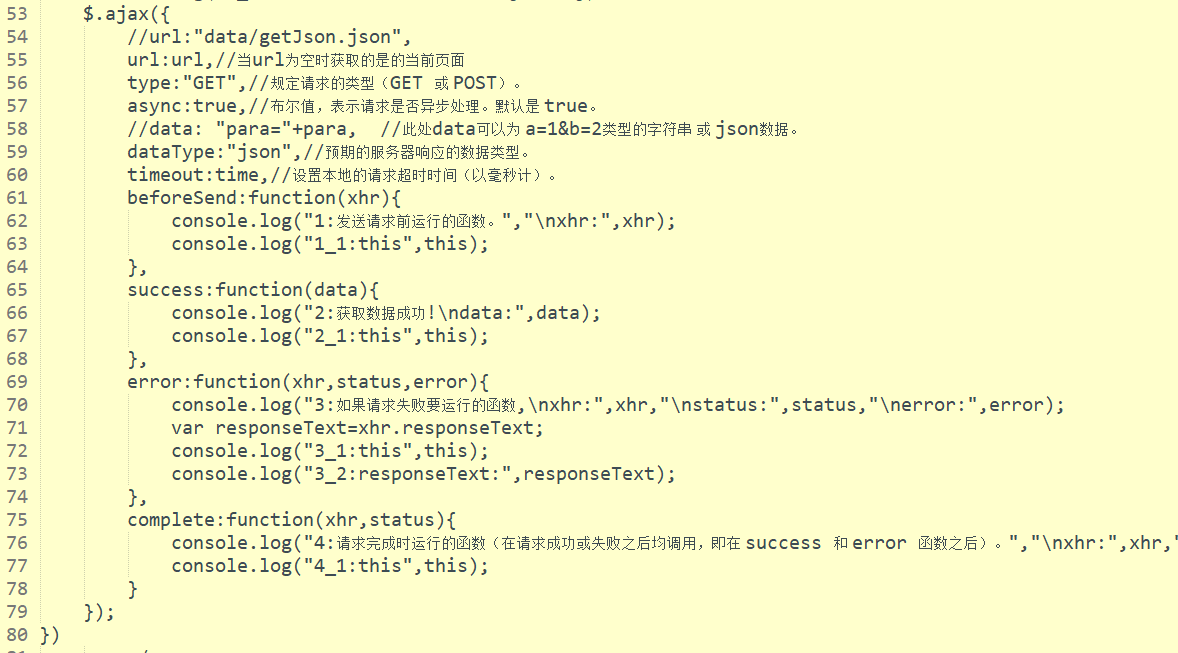
3.在ERROR回调函数中写 有关于 超时 处理 的函数:例如，可以在超时的情况下再次调用取数据函数。

\*/

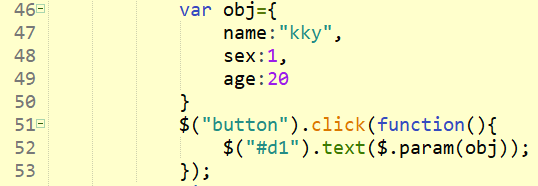
type:"GET",//规定请求的类型（GET 或 POST）。

});

})



#### 10.1.5 表单序列化



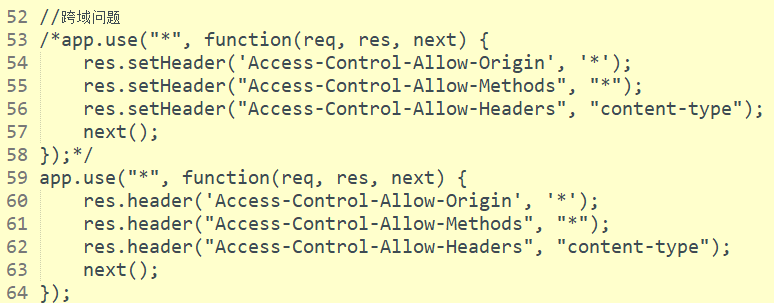
### 10.2跨域访问

#### 10.2.1 同源策略



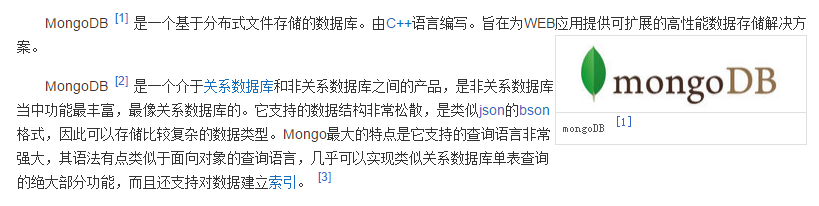
#### 10.2.2 jsonp跨域

#### 10.2.3 CORS跨域



## Day-11 MongoDB

### 11.1 MongoDB概念解析



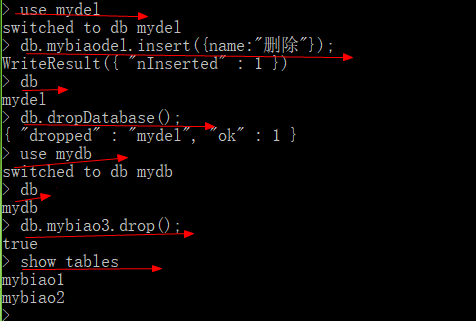
### 11.2 数据库基本操作

#### 11.2.1 连接数据库

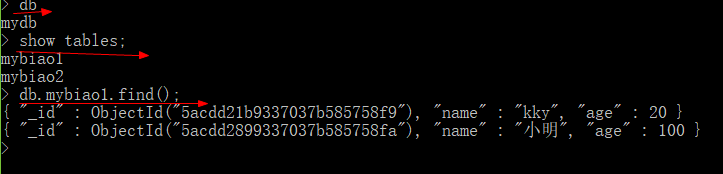
#### 11.2.2 创建数据库

#### 11.2.3 创建数据库

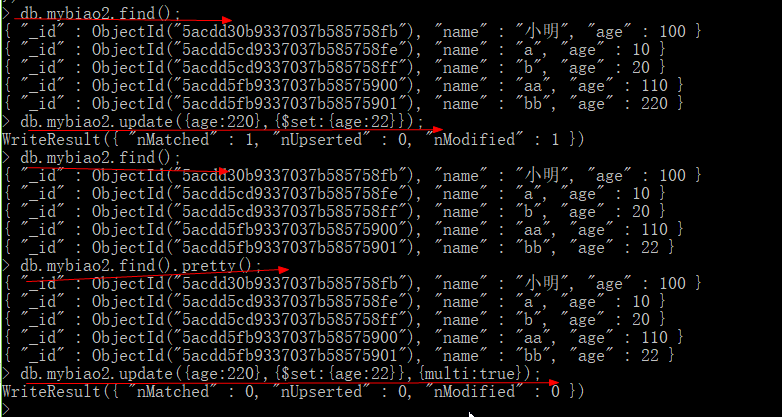
#### 11.2.4 删除数据库



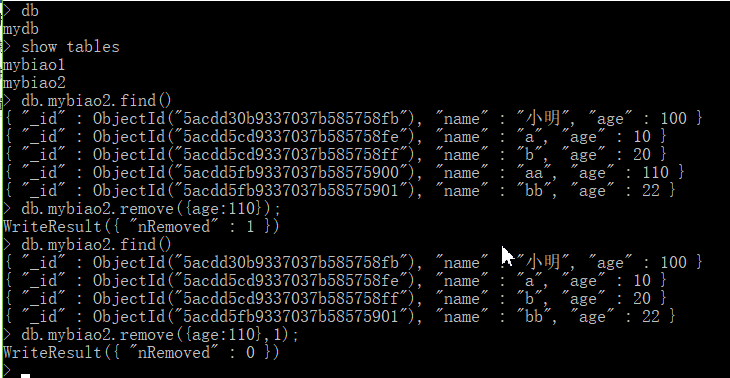
#### 11.2.5 插入文档



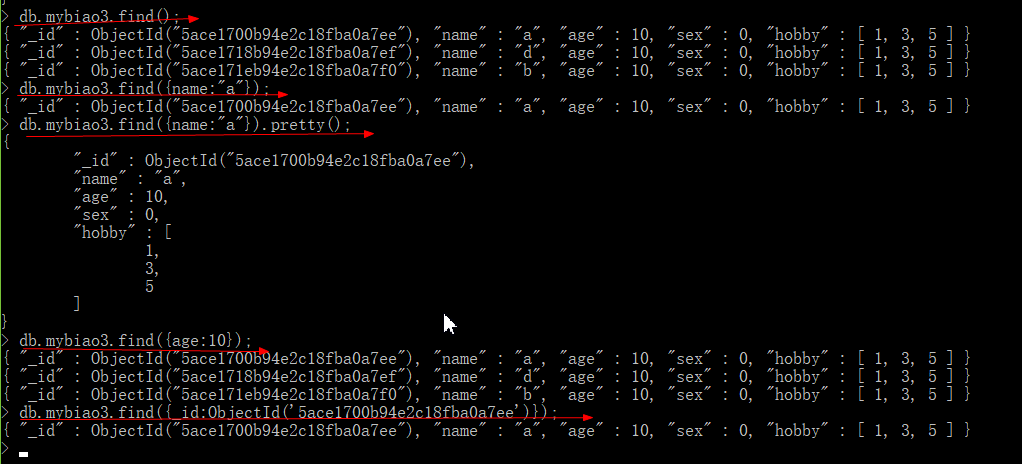
#### 11.2.6 更新文档

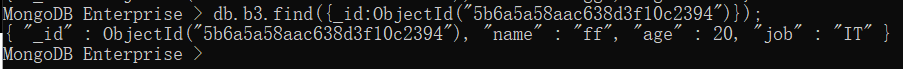


#### 11.2.6 删除文档



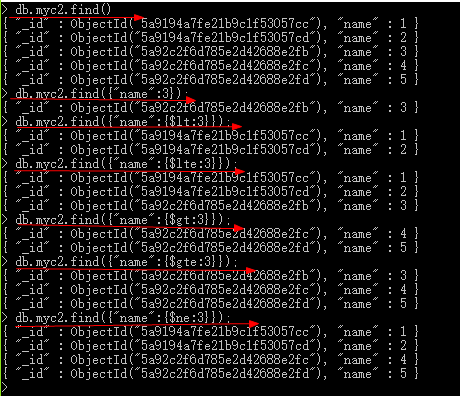
#### 11.2.7 查询文档

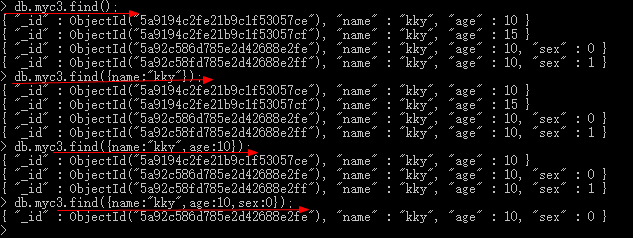


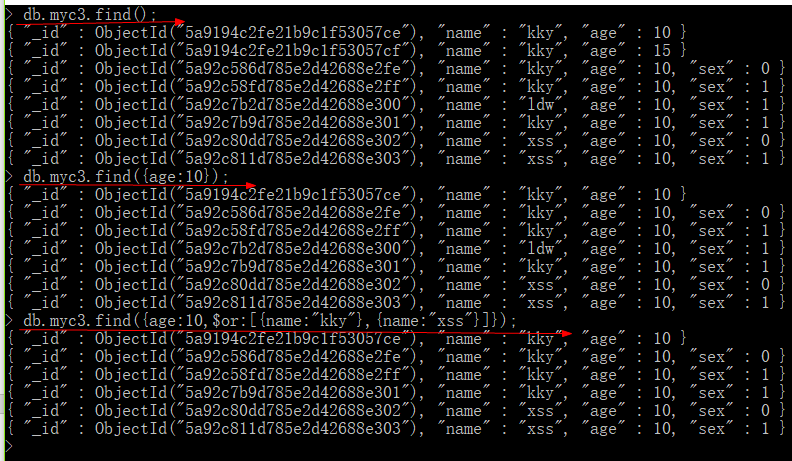


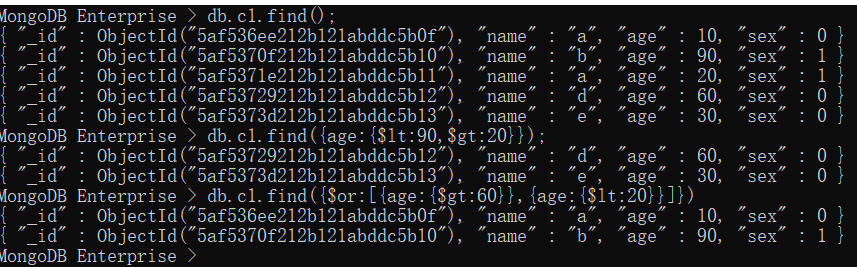
#### 11.2.8 条件操作符





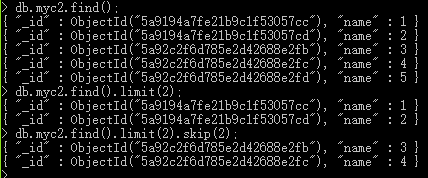




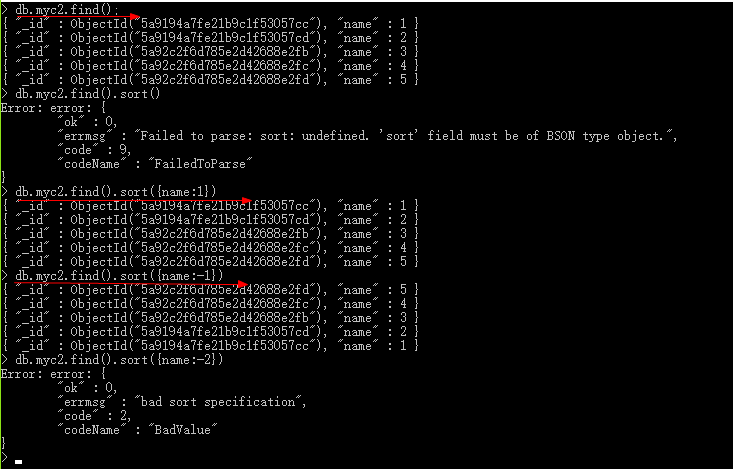


#### 11.2.9 limit和skip

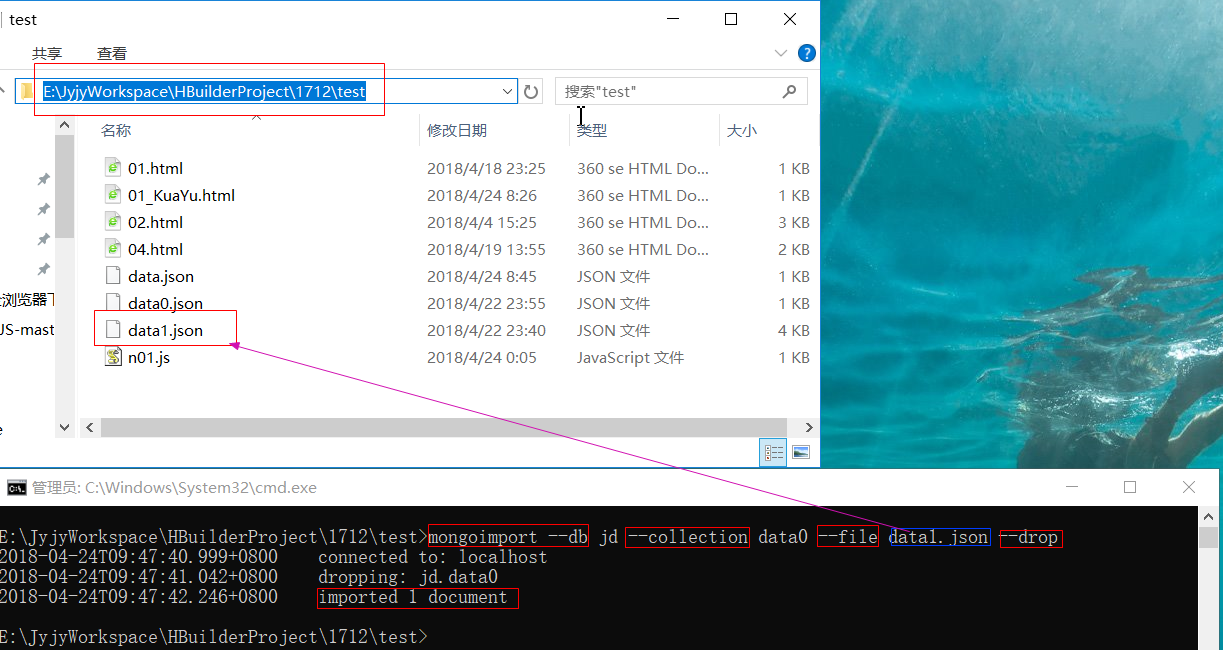




#### 11.2.10 排序



#### 11.2.11 导入json文件



### 11.3 Nodejs连接数据库

#### 11.3.1 nodejs连接数据库

#### 11.3.2 操作数据库

var mongodb = require("mongodb"); //引入mongodb

var MongoClient = mongodb.MongoClient;

var dbUrl = "mongodb://localhost:27017";

var myDb = "mydb02";

//{useNewUrlParser:true}

/\*

//版本1

MongoClient.connect(dbUrl,function(err, db) {

if(err) {

console.log("连接数据库失败"); //网络中断/数据库地址错误/端口号错误/数据服务没有启动

res.send({

msg: 0

});

} else {

console.log("数据库连接成功");

var dbase = db.db(myDb); //数据库名

obj.id = getTime.getTime();

dbase.collection(cName).insert(obj, function(err, result) {

if(err) {

console.log("数据添加到数据库失败");

res.send({

msg: 1

});

} else {

console.log("数据添加到数据库成功", result);

db.close();

res.send({

msg: 2

});

}

})

}

})

\*/

//第2个版本

module.exports.insertOne1 = function(cName,obj,res,cb) {

MongoClient.connect(dbUrl, function(err, db) {

if(err) {

console.log("连接数据库失败"); //网络中断/数据库地址错误/端口号错误/数据服务没有启动

res.send({

msg: 0

});

} else {

console.log("数据库连接成功");

var dbase = db.db(myDb); //数据库名

dbase.collection(cName).insert(obj, function(err, result) {

cb(err,result,db);

})

}

})

}

//封装版本

//连接数据库

function connectMGDB(cb0,res){

MongoClient.connect(dbUrl,function(err,db){

if(err){

console.log("连接数据库失败");

res.send({

msg:0

});

db.close();

}else{

var dbase=db.db(myDb);

cb0(dbase,db);

}

})

}

//插入数据

module.exports.insert = function(cName,obj,res,cb) {

connectMGDB(function(dbase,db){

dbase.collection(cName).insert(obj,function(err,result){

cb(err,result,db);

})

},res);

}

//查询符合条件的所有数据

module.exports.find = function(cName,whereObj,res,cb){

whereObj.find?whereObj.find:whereObj.find={};

whereObj.sort?whereObj.sort:whereObj.sort={};

whereObj.limit?whereObj.limit:whereObj.limit=0;

whereObj.skip?whereObj.skip:whereObj.skip=0;

connectMGDB(function(dbase,db){

dbase.collection(cName).find(whereObj.find).sort(whereObj.sort).skip(whereObj.skip).limit(whereObj.limit).toArray(function(err,result){

cb(err,result,db);

},res);

});

}

//查询符合条件的一条数据

/\*module.exports.findOne=function(cName,whereObj,res,cb){

connectMGDB(function(dbase,db){

dbase.collection(cName).findOne(whereObj,function(err,result){

cb(err,result,db);

})

})

}\*/

//根据ID查找某一条数据

module.exports.findOneById=function(cName,obj,res,cb){

//console.log("mongodb.ObjectId(obj.id):",mongodb.ObjectId(obj.id));

var whereObj={};

try{

whereObj.\_id=mongodb.ObjectId(obj.id);

}catch(err){

console.log("你输入的ID不正确的!");

whereObj.\_id="";

}

connectMGDB(function(dbase,db){

dbase.collection(cName).findOne(whereObj,function(err,result){

cb(err,result,db);

})

})

}

//删除符合条件的所有数据

module.exports.deleteMany=function(cName,obj,res,cb){

connectMGDB(function(dbase,db){

dbase.collection(cName).deleteMany(obj,function(err,result){

cb(err,result,db);

})

})

}

//根据ID删除某一条数据

module.exports.deleteOneById=function(cName,obj,res,cb){

//console.log("mongodb.ObjectId(obj.id):",mongodb.ObjectId(obj.id));

var whereObj={};

try{

whereObj.\_id=mongodb.ObjectId(obj.id);

}catch(err){

console.log("你输入的ID不正确的!");

whereObj.\_id="";

}

connectMGDB(function(dbase,db){

dbase.collection(cName).deleteOne(whereObj,function(err,result){

cb(err,result,db);

})

})

}

//修改符合条件的所有数据

module.exports.updateMany=function(cName,whereObj,updateObj,res,cb){

connectMGDB(function(dbase,db){

dbase.collection(cName).updateMany(whereObj,updateObj,function(err,result){

cb(err,result,db);

})

})

}

//根据ID修改某一条数据

module.exports.updateOneById=function(cName,obj,updateObj,res,cb){

//console.log("mongodb.ObjectId(obj.id):",mongodb.ObjectId(obj.id));

var whereObj={};

try{

whereObj.\_id=mongodb.ObjectId(obj.id);

}catch(err){

console.log("你输入的ID不正确的!");

whereObj.\_id="";

}

connectMGDB(function(dbase,db){

dbase.collection(cName).updateOne(whereObj,updateObj,function(err,result){

cb(err,result,db);

})

})

}

//查询符合条件的所有数据条数

module.exports.count = function(cName,whereObj,res,cb){

connectMGDB(function(dbase,db){

dbase.collection(cName).count(whereObj).then(function(count){

cb(count,db);

},res);

});

}

## Day-012 MongoDB 案例练习

### 12.1 使用Express和MongoDB 实现登陆注册

### 12.2 实现用户添加和删除功能

## Day-13 Express项目

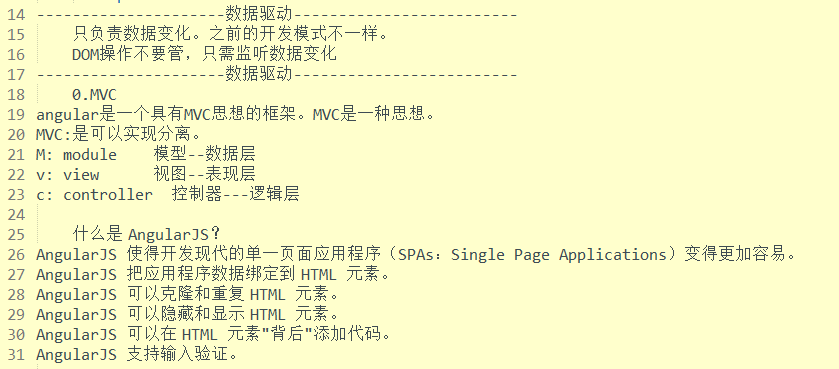
### 13.1 项目

## Day-14 Angular -01

### 14.1 AngularJS的入门

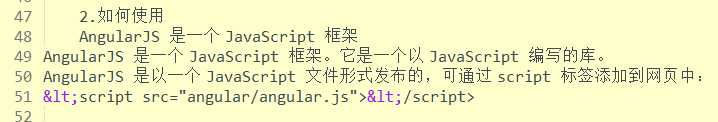
#### 14.1.1 AngularJS简介

#### 14.1.2 AngularJS的简介



#### 14.1.3 AngularJS的历史

#### 14.1.4 如何使用AngularJS



#### 14.1.5 AngularJS的优点

#### 14.1.6 MVC设计模式

#### 14.1.7 MVC程序

### 14.2模块

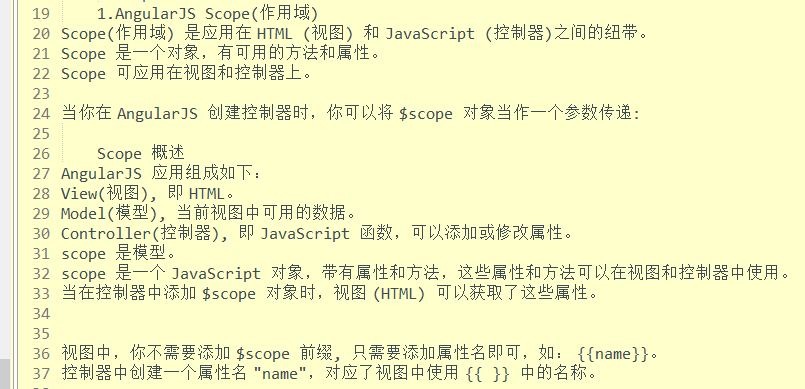
#### 14.2.1 什么是模块

#### 14.2.2 Angular中的模块



### 14.3控制器与作用域

#### 14.3.1 什么是控制器



#### 14.3.2 创建自己的控制器

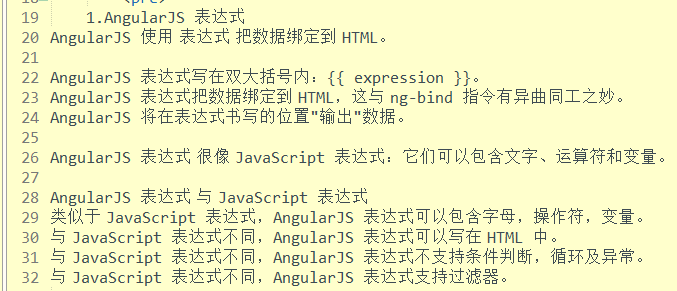
#### 14.3.3 实例化控制器对象ng-controller

#### 14.3.4 作用域与控制器的关系

### 14.4表达式

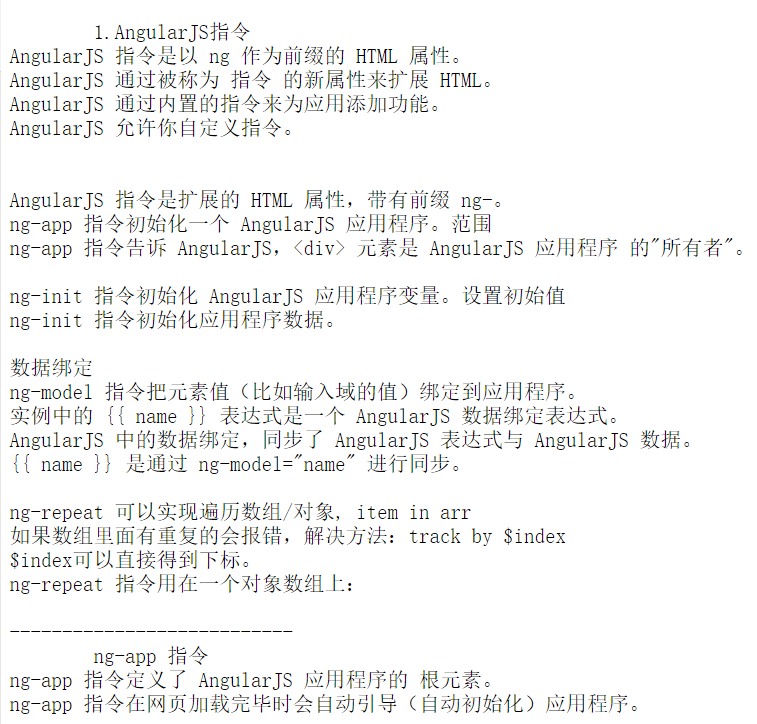
#### 14.4.1书写表达式

#### 14.4.2 常用的表达式

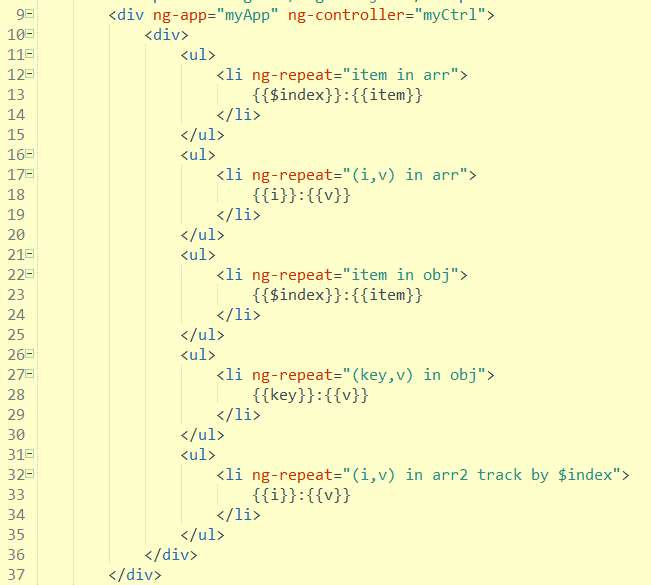


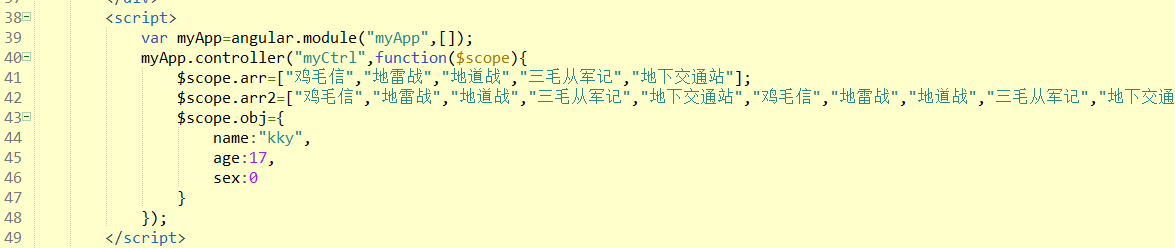
### 14.5基本指令

#### 14.1.1常用的指令

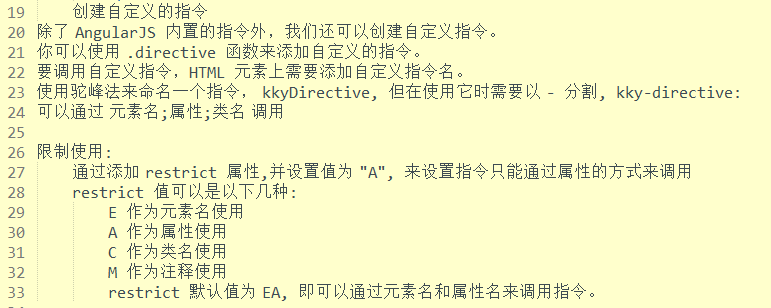








#### 14.1.2自定义指令





## Day-15 Angular -02

### 15.1数据双向绑定的原理

#### 15.1.1 $watch、



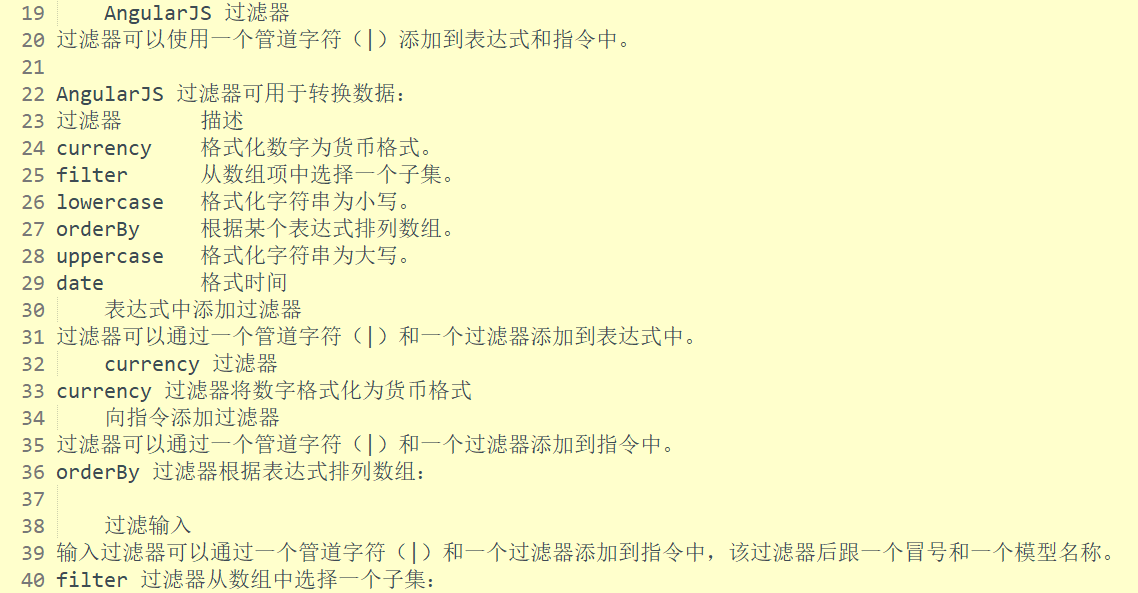
#### 15.1.2 $apply

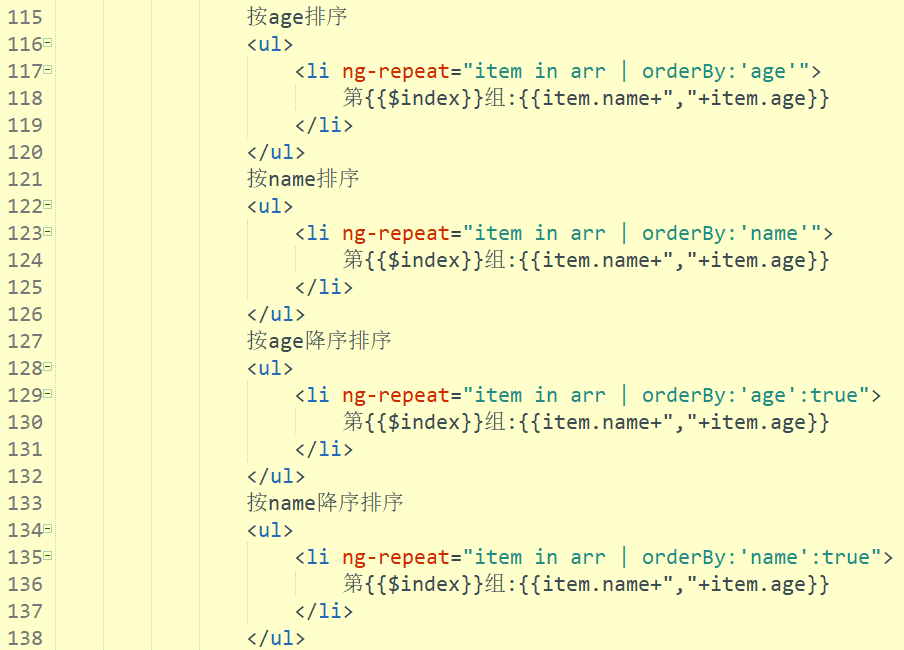


### 15.2过滤器

#### 15.2.1 什么是过滤器

#### 15.2.2 内置过滤器



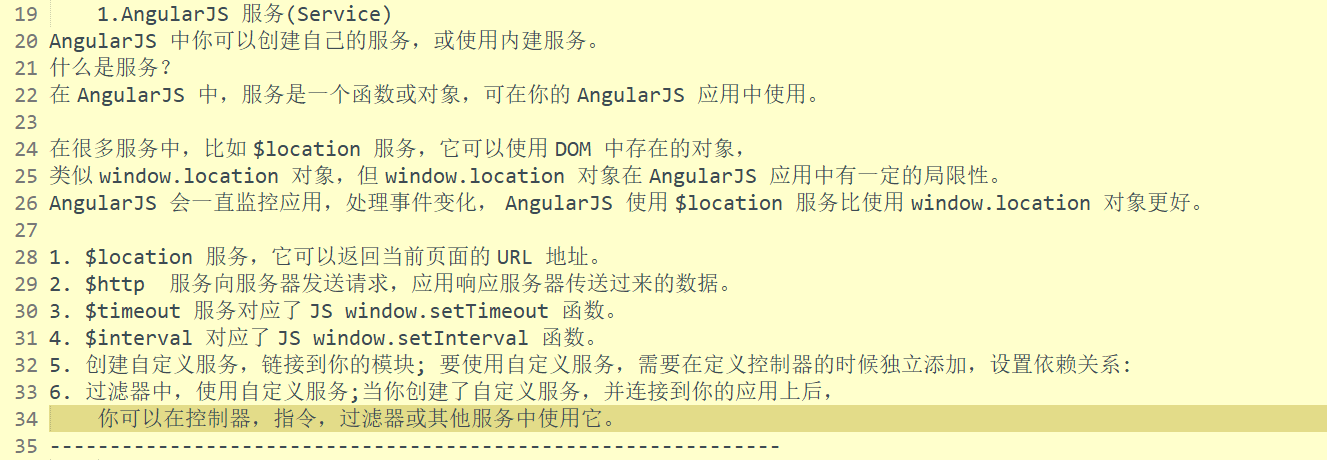


#### 15.2.3 自定义过滤器



### 15.3服务

15.3.1 什么是服务



15.3.2 常见的内置服务

#### 15.3.2.1 $http服务

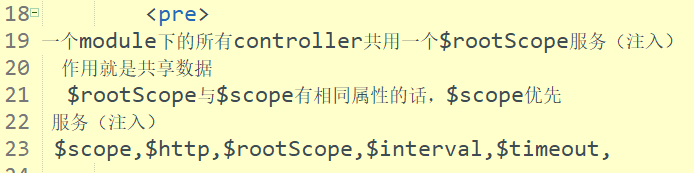
#### 15.3.2.2 $location服务

#### 15.3.2.3 $interval服务

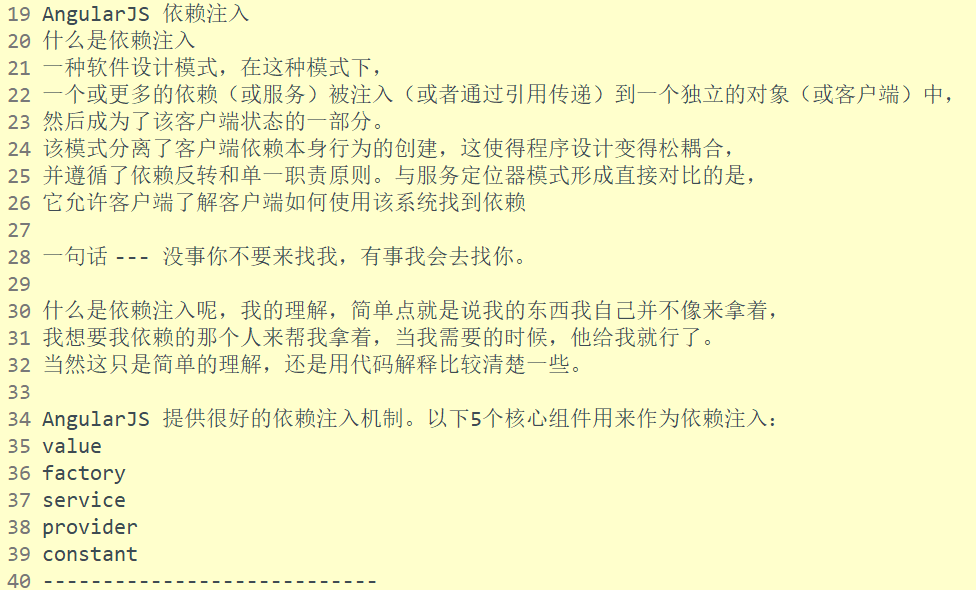
#### 15.3.2.4 $timeout服务

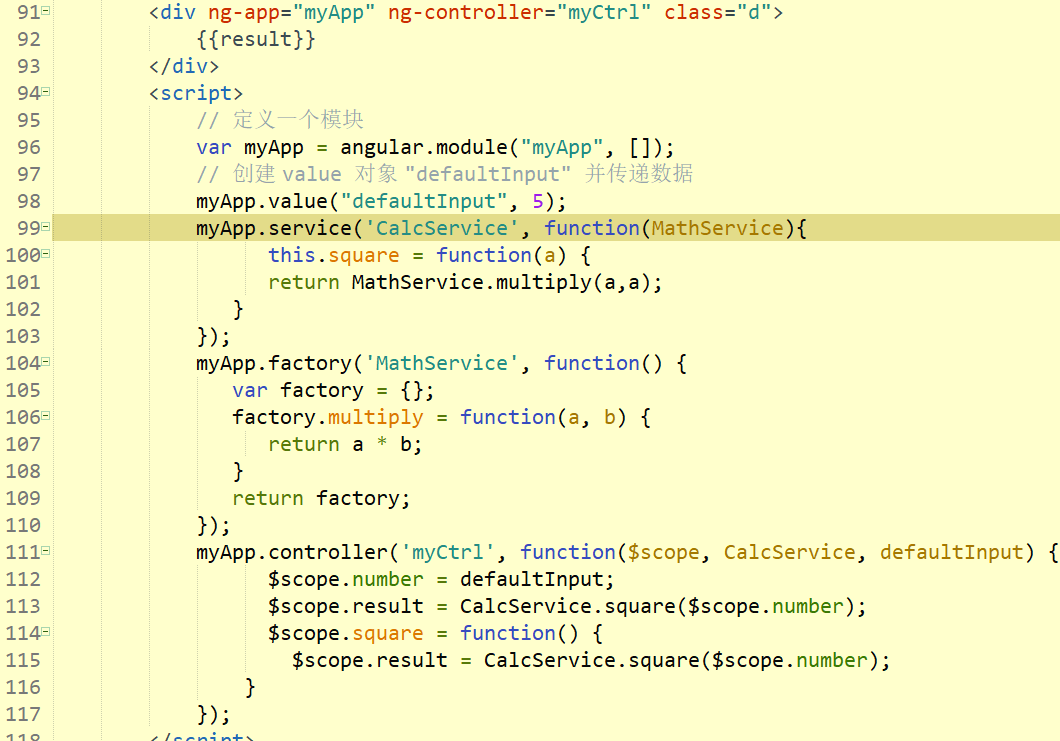


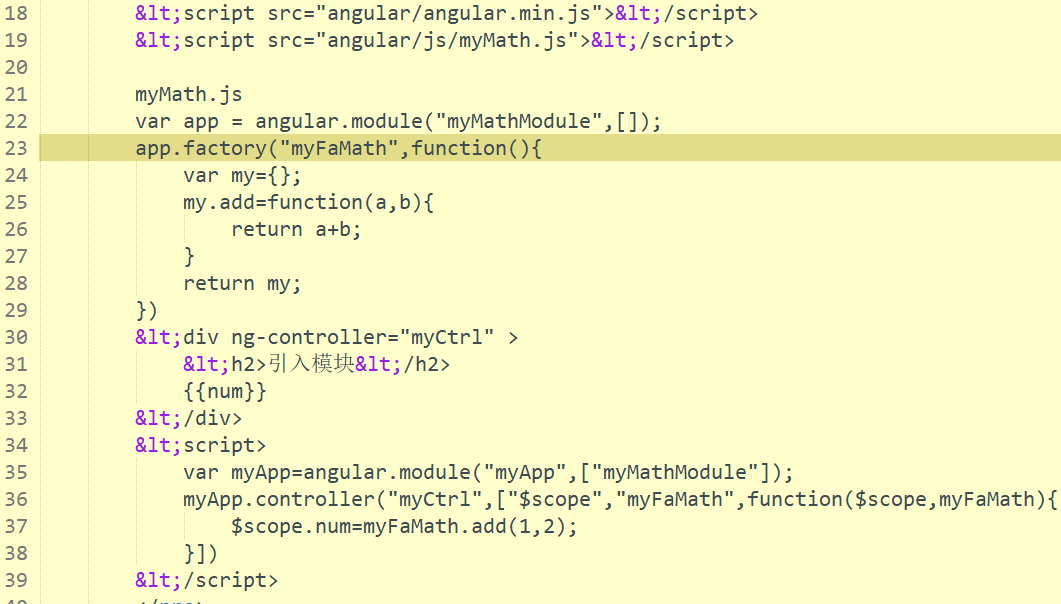




### 15.4 依赖注入



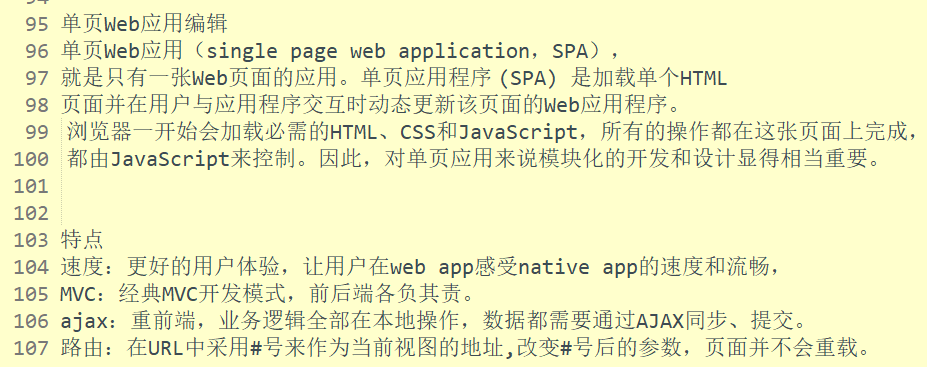




## Day-16 Angular -03

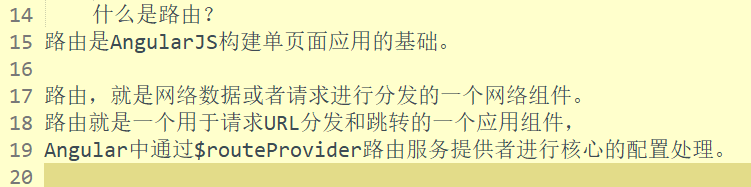
### 16.1SPA

16.1.1 什么是SPA

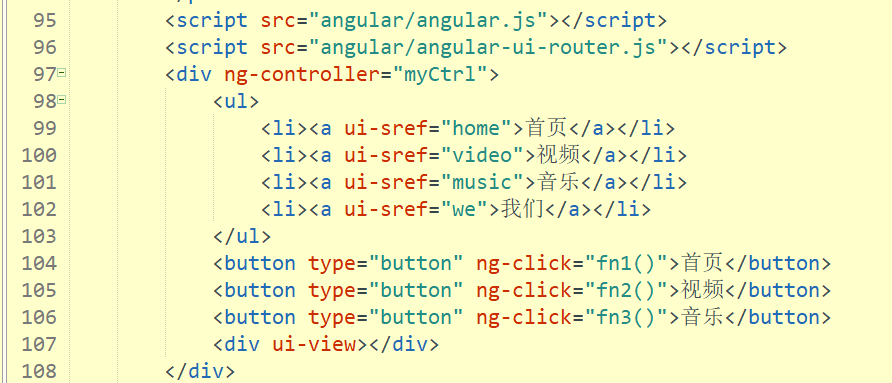


### 16.2 ui.route路由

16.2.1 如何使用UI-Router



16.2.2 UI-Router案例











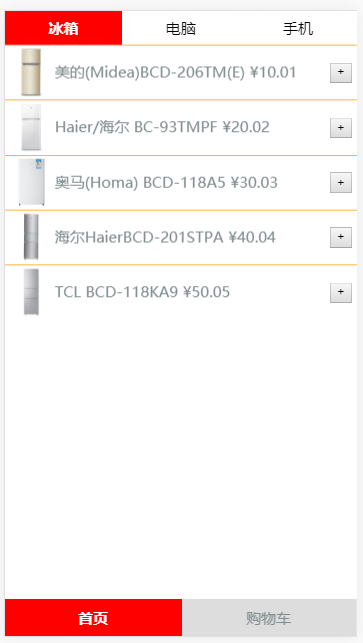
### 16.4项目框架搭建

## Day-17 Angular 04 项目1

### 17.1 首页

17.1.1 完成首页面开发

17.1.2 使用路由跳转详情页



### 17.2 详情页

17.2.1 完成详情页编写



17.2.1 实现添加购物车功能

## Day-18 Angular -05 项目2

### 18.1 购物车模块

18.1.1 实现购物车页面编写

18.1.2 实现购物车添加删除功能

