### 项目介绍，使用了哪些技术，收获了什么

### 介绍一下backbone

Backbone.js：前端框架（MVC）

**Model：**

1. 组织数据和业务逻辑；
2. 和数据库交互；
3. 数据变化时触发事件

Model可复用，可以在app中传递。

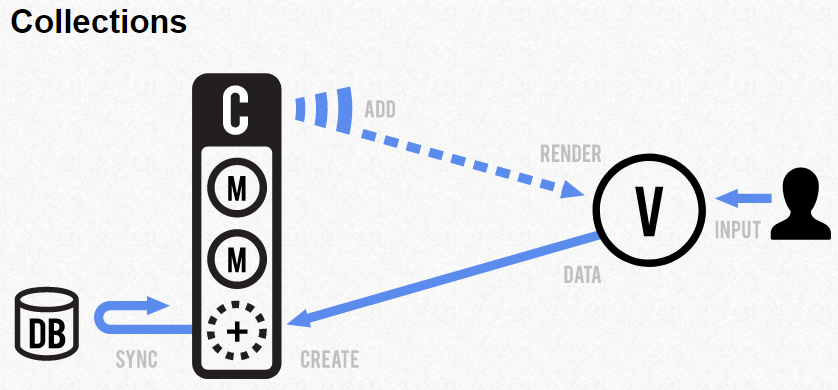
**View：**

1. 监听页面变化和重新渲染页面；
2. 处理用户的输入和交互；
3. 发送捕捉的输入到Model

View是一块独立的UI，从一个或多个models中渲染数据，也可以不需要数据。Models应该不感应views的变化，相反，views监听model的变化，重新渲染页面或者作出适当的反应。

**Collection：**

Collection帮助处理一组相关的models，加载和保存新的models到服务器，提供models集成和计算的方法。除了models自己的事件之外，collections可以集中代理models的所有事件。



### 介绍一下SEO

SEO是搜索引擎优化的缩写（Search Engine Optimization）。SEO是指在了解搜索引擎自然排名机制的基础之上，通过对网站进行站内优化和修复(网站Web结构调整、网站内容建设、网站代码优化和编码等)和站外优化，从而提高网站的网站关键词排名以及公司产品的曝光度。

**搜索引擎不优化的网站的特征：**

1. 网页中大量采用图片或者Flash等富媒体（Rich Media）形式，没有可以检索的文本信息，而SEO最基本的就是文章SEO和图片SEO
2. 网页没有标题，或者标题中没有包含有效的关键词
3. 网页正文中有效关键词比较少
4. 网站导航系统让搜索引擎“看不懂”
5. 大量动态网页影响搜索引擎检索
6. 没有其他被搜索引擎已经收录的网站提供的链接
7. 网站中缺少原创的内容，完全照搬硬抄别人的内容等

**内部优化：**

1. META标签优化：例如：TITLE，KEYWORDS，DESCRIPTION等的优化
2. 内部链接的优化，包括相关性链接（Tag标签），锚文本链接，各导航链接，及图片链接
3. 网站内容更新：每天保持站内的更新(主要是文章的更新等)
4. 网站主页唯一性，网站内页链向主页，301,404等改进
5. 建立网站地图sitemap，同时把网站地图的链接放在首页上
6. 每个网页最多距离首页三次点击就能到达
7. 网站的导航系统最好使用文字链接

**外部优化：**

1. 尽量保持外部链接的多样性
2. 每天添加一定数量的外部链接，使关键词排名稳定提升
3. 与一些和你网站相关性比较高,整体质量比较好的网站交换友情链接，巩固稳定关键词排名

### 低版本浏览器如何兼容HTML5标签

HTML5的语义化标签以及属性，可以让开发者非常方便地实现清晰的web页面布局，加上CSS3的效果渲染，快速建立丰富灵活的web页面显得非常简单。

<header>定义页面或区段的头部；

<footer>定义页面或区段的尾部；

<nav>定义页面或区段的导航区域；

<section>页面的逻辑区域或内容组合；

<article>定义正文或一篇完整的内容；

<aside>定义补充或相关内容；

使用他们能让代码语义化更直观,而且更方便SEO优化。但是此HTML5新标签在IE6/IE7/IE8上并不能识别，需要进行JavaScript处理。以下就介绍几种方式。

方法一：手动创建HTML5标签

<!--[if lt IE 9]>

<script>

(function() {

var e = "abbr, article, aside, audio, canvas, datalist, details, dialog, eventsource, figure, footer, header, hgroup, mark, menu, meter, nav, output, progress, section, time, video".split(', ');

var i= e.length;

while (i--){

document.createElement(e[i])

}

})()

</script>

<![endif]-->

CSS：

/\*html5\*/

article,aside,dialog,footer,header,section,footer,nav,figure,menu{display:block}

方法二：使用Google的html5shiv包（推荐）

<!--[if lt IE 9]>

<script src="bower\_components/html5shiv/dist/html5shiv.js"></script>

<![endif]-->

### setInterval和setTimeout

* Methods setInterval(func, delay, ...args) and setTimeout(func, delay, ...args) allow to run the funcregularly/once after delay milliseconds.
* To cancel the execution, we should call clearInterval/clearTimeout with the value returned by setInterval/setTimeout.
* Nested setTimeout calls is a more flexible alternative to setInterval. Also they can guarantee the minimal time *between* the executions.
* Zero-timeout scheduling setTimeout(...,0) is used to schedule the call “as soon as possible, but after the current code is complete”.

Some use cases of setTimeout(...,0):

* To split CPU-hungry tasks into pieces, so that the script doesn’t “hang”
* To let the browser do something else while the process is going on (paint the progress bar).

**Garbage collection**

When a function is passed in setInterval/setTimeout, an internal reference is created to it and saved in the scheduler. It prevents the function from being garbage collected, even if there are no other references to it.

// the function stays in memory until the scheduler calls it

setTimeout(function() {...}, 100);

For setInterval the function stays in memory until cancelInterval is called.

There’s a side-effect. A function references the outer lexical environment, so, while it lives, outer variables live too. They may take much more memory than the function itself. So when we don’t need the scheduled function any more, it’s better to cancel it, even if it’s very small.

### 内存溢出的情况以及避免办法

**定义：**

应用不需要的内存没有及时被回收。

**JavaScript中的内存管理：**

JavaScript有垃圾回收机制，通过周期性地检查之前分配的内存是否还能被应用访问来确定是否回收。

**JavaScript中的内存溢出：**

JavaScript中的内存溢出主要是因为意外的引用。意外的引用指的是某个不用的变量存在于GC标记的存活变量树中。

1. 意外的全局变量

函数内未用var声明的变量；函数内用this创建的变量；

function foo(arg) {

window.bar = "this is an explicit global variable";

}

function foo() {

this.variable = "potential accidental global";

}

foo();

这些全局变量不会被回收，除非手动赋值为null或者重新赋值。特别是，如果这些全局变量用于存储和处理大量的数据，那么一定要记得手动回收，否则缓存会耗费大量内存。

2. 遗忘的定时函数和回调函数

setInterval如果没有停止，那么定时函数中定义的变量以及函数依赖的变量不会被回收。

var someResource = getData();

setInterval(function() {

var node = document.getElementById('Node');

if(node) {

// Do stuff with node and someResource.

node.innerHTML = JSON.stringify(someResource));

}

}, 1000);

上述例子中：如果Node节点被移除，定时函数就是无效的，但是someResource（如果存储了大量数据）不会被回收。

再比如，元素的监听事件在元素被移除之后，浏览器会自动回收事件的引用。但最好是手动移除监听事件。

var element = document.getElementById('button');

function onClick(event) {

element.innerHtml = 'text';

}

element.addEventListener('click', onClick);

// Do stuff

element.removeEventListener('click', onClick);

element.parentNode.removeChild(element);

// Now when element goes out of scope,

// both element and onClick will be collected even in old browsers that don't

// handle cycles well.

3. 多余的DOM节点引用

在JavaScript中手动创建的节点引用不会在节点被移除时被回收。

var elements = {

button: document.getElementById('button'),

image: document.getElementById('image'),

text: document.getElementById('text')

};

function doStuff() {

image.src = 'http://some.url/image';

button.click();

console.log(text.innerHTML);

// Much more logic

}

function removeButton() {

// The button is a direct child of body.

document.body.removeChild(document.getElementById('button'));

// At this point, we still have a reference to #button in the global

// elements dictionary. In other words, the button element is still in

// memory and cannot be collected by the GC.

}

值得注意的是：如果在JavaScript中引用的某个节点的父节点被移除了，但父节点依旧在内存中。

4. 闭包

闭包引用的包裹函数中的变量常驻在内存中，使用不当容易造成内存溢出。

var theThing = null;

var replaceThing = function () {

var originalThing = theThing;

var unused = function () {

if (originalThing)

console.log("hi");

};

theThing = {

longStr: new Array(1000000).join('\*'),

someMethod: function () {

console.log(someMessage);

}

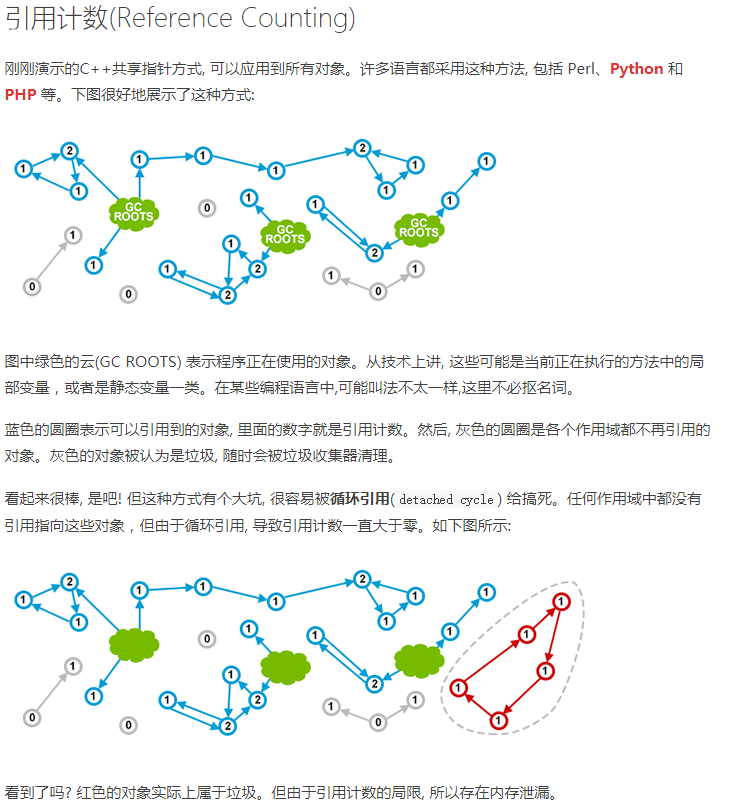
};

};

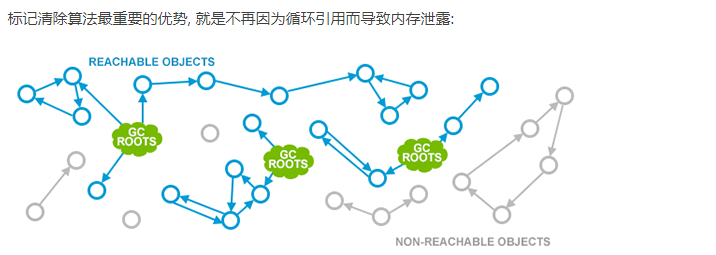
setInterval(replaceThing, 1000);

unused函数没有被使用过，但是由于它引用的变量originalThing导致unused函数不会被回收。

**补充知识点：**







### document创建新节点的方法和查找节点的方法

**创建：**

createAttribute(): creates a new attribute node, and returns it.

createCDATASection(): creates a new CDATA section node, and returns it.

createComment(): creates a new comment node, and returns it.

createDocumentFragment(): Creates a new empty DocumentFragment.

createElement(): creates the HTML element specified by tagName, or an HTMLUnknownElement if tagName isn't recognized.

createElementNS(): creates an element with the specified namespace URI and qualified name.

createTextNode(): creates a new Text node.

~~createNode()~~

**查找：**

getElementById()

getElementsByClassName()

getElementsByName()

getElementsByTagName()

getElementsByTagNameNS(): Returns a list of elements with the given tag name belonging to the given namespace.

### 元素消失的几种方法

**CSS：**

1. visibility: hidden;
2. opacity: 0;
3. position: absolute; left: -1000px;
4. display: none;
5. ~~width: 0; height: 0;~~
6. ~~z-index: -100;~~

### 点击相关事件触发顺序

**HTML：**

<button id="aa">a</button>

**JavaScript：**

var a = document.getElementById("aa")

a.onmousedown = function() {

console.log("down")

}

a.onmouseup = function() {

console.log("up")

}

a.onclick = function(){

console.log("click")

}

a.onfocus = function() {

console.log("focus")

}

**Console：**

down focus up click

### DOM reflow

A reflow computes the layout of the page. A reflow on an element recomputes the dimensions and position of the element, and it also triggers further reflows on that element’s children, ancestors and elements that appear after it in the DOM. Then it calls a final repaint. Reflowing is very expensive, but unfortunately it can be triggered easily.Reflow occurs when you:

1. insert, remove or update an element in the DOM
2. modify content on the page, e.g. the text in an input box
3. move a DOM element
4. animate a DOM element
5. take measurements of an element such as offsetHeight or getComputedStyle
6. change a CSS style
7. change the className of an element
8. add or remove a stylesheet
9. resize the window
10. scroll

### 画出盒子模型，要使谷歌浏览器的盒子模型显示得跟IE浏览器一致（让谷歌跟ie一致，不是ie跟谷歌一致），该怎么做

### 实现一个布局：左边固定宽度为200，右边自适应，而且滚动条要自动选择只出现最高的那个

### 用js使低版本浏览器支持HTML5标签底层是怎么实现的

### 前端工程化技术，自动化流程

### 如何在某个函数中循环调用自己固定次数（100次）后停止

setTimeout嵌套

setTimeout

全局变量 + 函数递归调用

### 网站加载缓慢，可以从哪几个方面入手优化性能

CDN加速

Lazy load

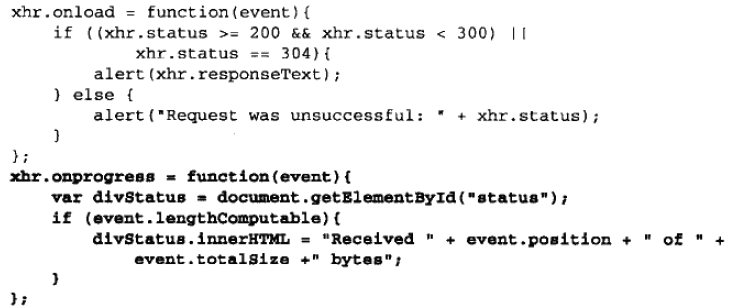
缓存

HTTP2.0

### 移动端呈现网页如何适配不同尺寸大小的设备

### 如何实现Ajax请求

var xhr = new XMLHttpRequest() // 创建xhr实例



xhr.onerror = function() { } // 请求错误时触发

xhr.open(“get”, url, false) // 请求方法、请求地址、是否异步

xhr.timeout = 1000 // 超时时间

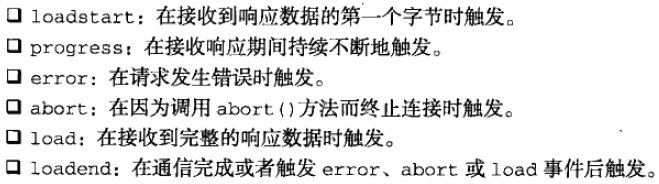
xhr.ontimeout = function() { } // 超时触发的事件

xhr.setRequestHeader(key, value) // 设置自定义请求头

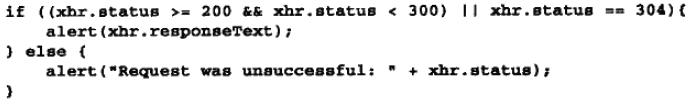
xhr.send(data) // 发送请求

xhr = null // xhr用完之后解引用，不建议重用。

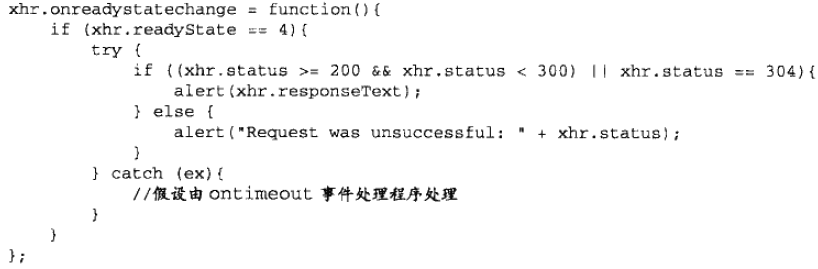
6个事件：



// 以下是发送同步请求时接受响应的代码，放在send()之后。



// 以下是发送异步请求时接受响应的代码，必须在调用open()之前指定事件处理程序。



### 从输入URL到网页完全加载都经历了哪些过程

1. 浏览器搜索自身的DNS缓存
2. 如果没有找到或者缓存已经失效，则搜索操作系统的DNS缓存
3. 读取本地host文件
4. 浏览器发起一个DNS的系统调用
5. 向宽带运营商服务器发起域名解析请求
6. 宽带运营商服务器查看本身缓存
7. 运营商的DNS服务器发起一个迭代的DNS解析请求，获取到IP地址之后返回给操作系统内核并缓存
8. 操作系统系统内核将结果返回给浏览器
9. 浏览器获得域名对应的IP地址之后发起HTTP三次握手
10. 浏览器通过端口向服务器发起TCP连接请求（途径多个路由，到达服务器端，经过网卡和防火墙，进入内核的TCP/IP协议栈，最后到达web服务端），最终建立起TCP连接
11. TCP/IP协议建立之后，浏览器就可以向服务器发送HTTP请求（请求网页，发送GET请求）
12. 服务器接受请求之后，根据请求类型和请求参数经过后端的处理返回数据（请求网页的话，返回完整的HTML代码）
13. 浏览器获得网页数据之后开始解析和渲染界面，对于CSS、JS和图片等静态资源的HTTP请求，重复上述步骤
14. 浏览器根据拿到的静态资源对页面进行渲染，最终把一个完整的页面呈现给用户

### JavaScript中的深复制和浅复制

**浅复制：**

shallow copy是什么？通俗来讲就是值复制，把原始对象中所有属性的值都复制了一份，如果是基本数据类型，如果属性是对象的引用，那么仅仅是引用地址被复制，也就是说源对象和拷贝对象的该属性都指向同一个对象。

方法一：（不推荐，要求很严苛）

Object.assign(target, ...sources)

注意：

1目标对象属性会被源对象同名属性覆盖，源对象从右往左，属性也会覆盖。

2只复制可枚举的自身属性，会触发源对象的getter方法。

3 String和Symbol类型的属性会被复制，属性值为null和undefined的不会被复制。

4 如果目标对象某属性是不可写的，源对象中具有同名属性，复制过程抛出TypeError异常，异常前的属性被复制。

方法二：

var newObj = Object.create(

Object.getPrototypeOf(obj),

Object.getOwnPropertyDescriptors(obj)

)

相较于方法一，可以复制getter方法而不是触发。

方法三：（结果同方法二）

Object.defineProperties(

target,

Object.getOwnPropertyDescriptors(source)

)

方法四：

var newObj = jQuery.extend({}, oldObj)

**深复制：**

deep copy是什么？在浅复制的基础上，对于属性是对象引用的，复制时不是复制引用，而是新建一个相等的对象并引用。

方法一：JSON trick

var newObj = JSON.parse(JSON.stringify(obj))

不能包含function，适用于属性是object、array、string、boolean和number类型

方法二：jQuery

var newObj = jQuery.extend(true, {}, oldObj)

方法三：pure JavaScript

// 递归遍历属性并判断

function keepCloning(obj) {

if (obj === null || typeof obj !== 'object') return obj

// Handle Date

if (obj instanceof Date) {

copy = new Date();

copy.setTime(obj.getTime());

return copy;

}

// Handle Array

if (obj instanceof Array) {

copy = [];

for (var i = 0, len = obj.length; i < len; i++) {

copy[i] = keepCloning (obj[i]);

}

return copy;

}

var tmp = obj.constructor()

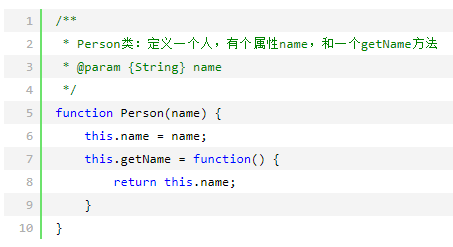
for (var key in obj) tmp[key] = keepCloning(obj[key])

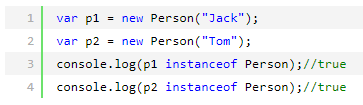
return tmp

}

### JS中如何实现类

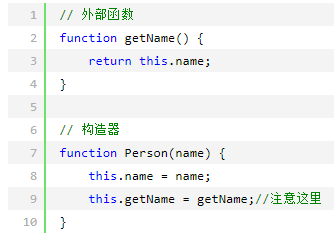
1. **构造函数方式**





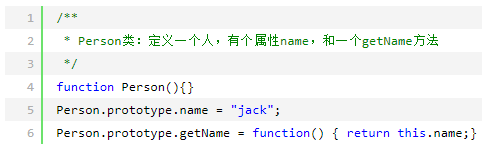
缺点：实例不共享方法，造成内存浪费。

改进：在全局作用域中定义声明方法。

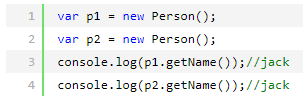


仍然存在的问题：在全局作用域中定义对象的方法seems odd，而且方法有很多，都在全局作用域中定义，封装性差。

1. **原型方式**



把属性和方法都放在原型中。

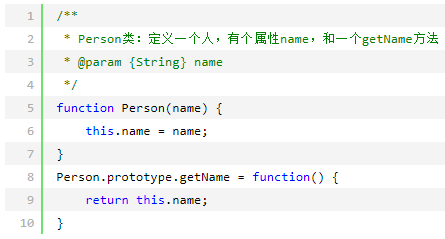


优点：所有实例共享方法，没有内存浪费

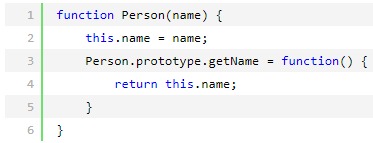
缺点：不能通过参数构建不同属性值的实例

1. **组合构造函数和原型的方式**

在构造函数中定义属性；在原型中定义共享的方法。



为了使代码更加紧凑：

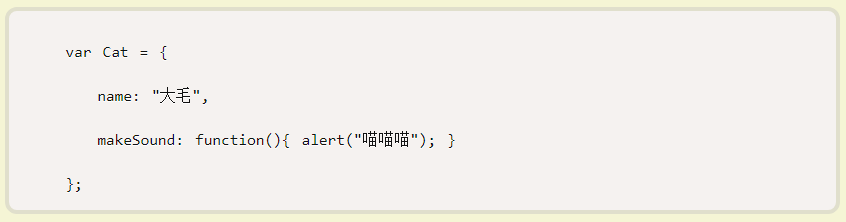


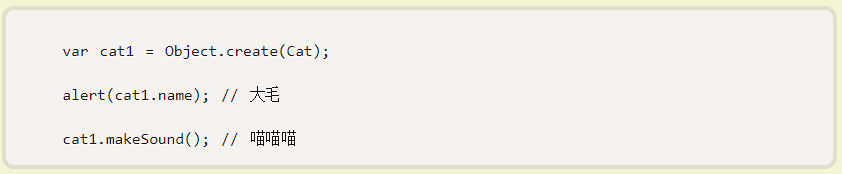
// 缺点就是getName只需要定义一次，

// 而这里每次生成实例时都会执行。

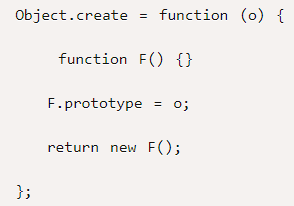
1. **Object.create()**

ECMAScript5提出Object.create()方法，直接利用对象生成实例，不需要new关键字。





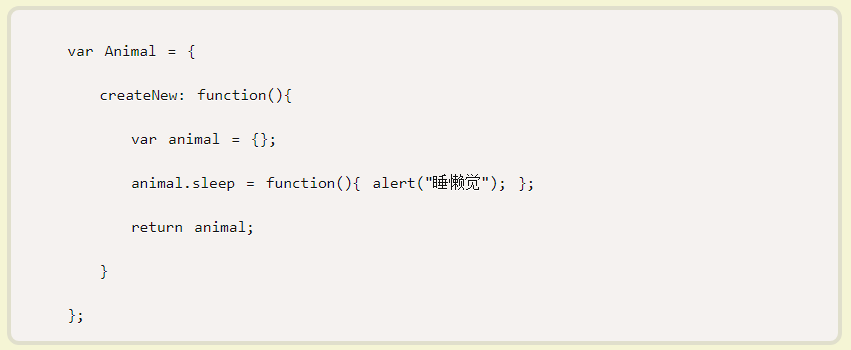
内部的实现原理：

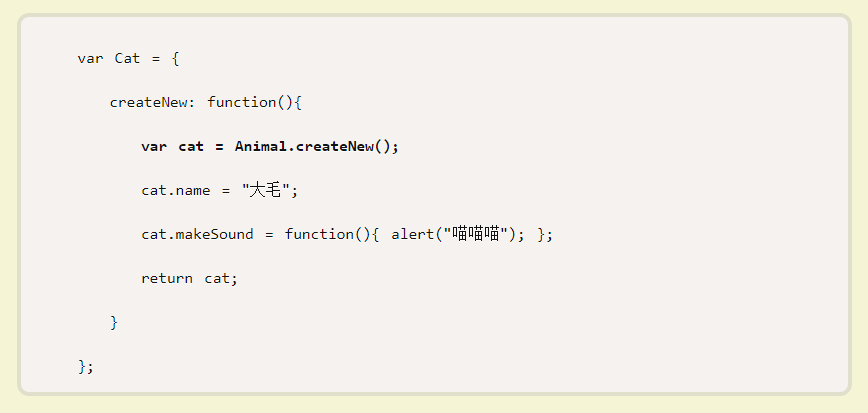
其实就是对象的浅复制。

缺点：不能使用instanceof判断，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。（感觉不对啊）

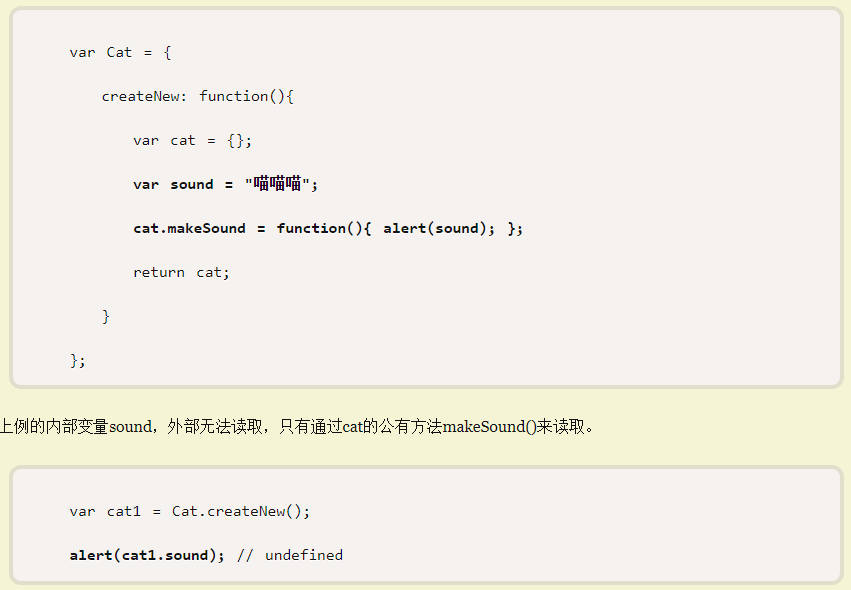
1. **极简主义**

继承：

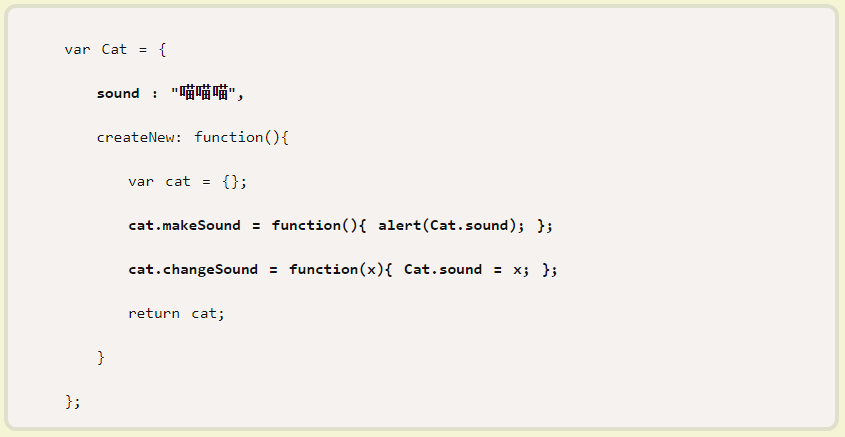


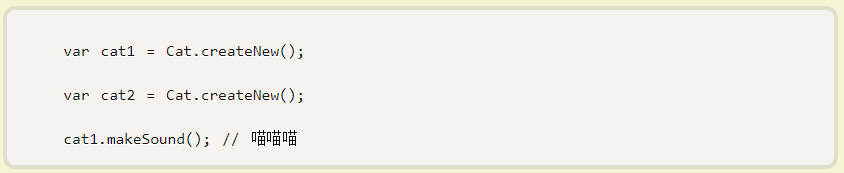


私有属性和方法：



数据共享（静态变量）：







优点：没有用到this和prototype，使用简单，操作方便。

缺点：不能用instanceof判断。

1. class关键字

ECMAScript6中定义了class、extends、super关键字，简化了JavaScript中类的实现。

定义：



继承：



### JS中对象的继承方式都有哪些，分别介绍

function Person(name) {

if(name !== undefined) {

this.name = name;

} else {

this.name = "StrayBugs";

}

this.age = 22;

}

Person.prototype.sayName = function() {

alert(this.name);

};

1. **原型链**

function Man() {

}

Man.prototype = new Person();

var man = new Man();

继承就是子类获得父类的属性和方法。原型链的思路是利用原型共享的特点，让父类的一个实例充当子类的原型。父类的实例必然包括了父类的属性与方法，那么子类的所有实例都可以通过原型链一层层找到父类的属性与方法了。

缺点：不适合继承属性，共享属性互相影响。

1. **借用构造函数**

function Man(name) {

//每个实例都可以有自己的名字

Person.call(this, name);

//子类增加的属性

this.job = 'student';

}

var man1 = new Man();

var man2 = new Man('Jesse');

alert(man1.name); //"StrayBugs"

alert(man2.name); //"Jesse"

alert(man1.sayName); //undefined

alert(man2.sayName); //undefined

借用构造函数技术就是在子类的构造函数中通过父类.apply(this)或者父类.call(this)来借用父类的构造函数，这时每个实例都有单独的副本。

缺点：没有形成原型链，所以无法共享方法。

1. **组合继承**

function Man(name) {

//每个实例都可以有自己的名字

Person.call(this, name);

//子类增加的属性

this.job = 'student';

}

//继承方法

Man.prototype = new Person();

Man.prototype.constructor = Man;

//子类增加的方法

Man.prototype.sayAge = function() {

alert(this.age);

};

var man1 = new Man();

var man2 = new Man('Jesse');

man1.sayName(); //"StrayBugs"

man2.sayName(); //"Jesse"

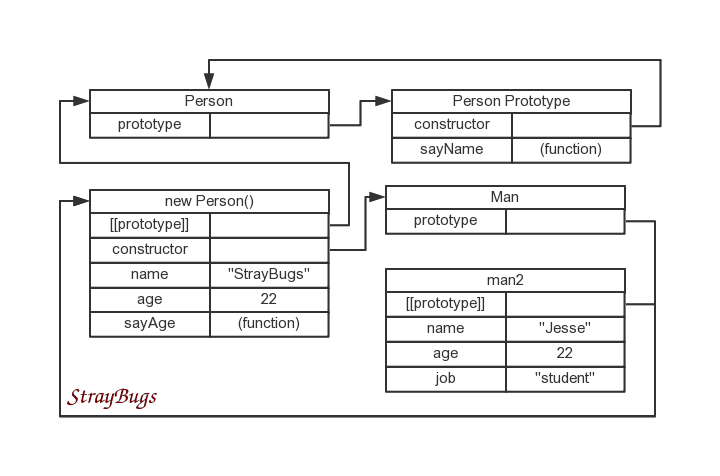
man2.sayAge(); //22

alert(man1 instanceof Man); //true

alert(man1 instanceof Person); //true

alert(man1.constructor.prototype.isPrototypeOf(man2)); //true

组合构造函数与原型链来实现继承，叫做组合继承（Combination Inheritance）。是 JavaScript 中最常用的继承模式。



缺点：子类的原型上还保留了一份无用的共用属性。

1. **原型式继承**

原型式继承（Prototypal Inheritance）很特别，它希望利用现有的对象去继承该对象的类。说白了，就是前文的原型链继承那里，将 new Person() 换成一个现有的对象（比如 Person的一个现有的实例）。封装起来就是这个样子：

function object(o) {

function F() {}

F.prototype = o;

return new F();

}

原理其实和原型链是一样的，只需简单地浅复制一个对象。ECMAScript5 新增 Object.create() 方法规范化了原型式继承。

问题：使用时注意哪些属性是原型的可以共享使用，哪些属性是实例自己定义的。

1. **寄生式继承**

如果需要在原型式继承的基础上添加方法，就需要使用寄生式继承。可以看做是原型式继承的增强版。

function createMan(person) {

//这里不一定是 object()，可以是任意能返回新对象的函数

var man = object(person);

man.sayAge = function() {

alert(this.age);

};

return man;

}

var person = new Person();

var man = createMan(person);

man.sayAge(); //22

缺点：

1. **寄生组合式继承**

组合式继承的缺点是子类的原型上会保留了一份无用的属性。这是因为调用了两次父类的构造函数。第一次调用是为子类添加原型 new Person() 的时候，第二次是子类构造函数内部调用 Person.call(this, name) 。由于第一次调用构造函数的目的只是为了得到父类的原型，利用原型式继承的浅复制方法直接复制原型亦可达到目的。

function inheritPrototype(Man, Person) {

//只复制原型

var p = object(Person.prototype);

p.constructor = Man;

Man.prototype = p;

}

function Man(name) {

Person.call(this, name);

this.job = 'student';

}

inheritPrototype(Man, Person);

Man.prototype.sayAge = function() {

alert(this.age);

};

其实就是原型链继承中将 new Person() 换成 object(Person.prototype)。instanceof与isPrototypeOf 对 Person 依然有效。

使用：

var man = new Man('Jesse');

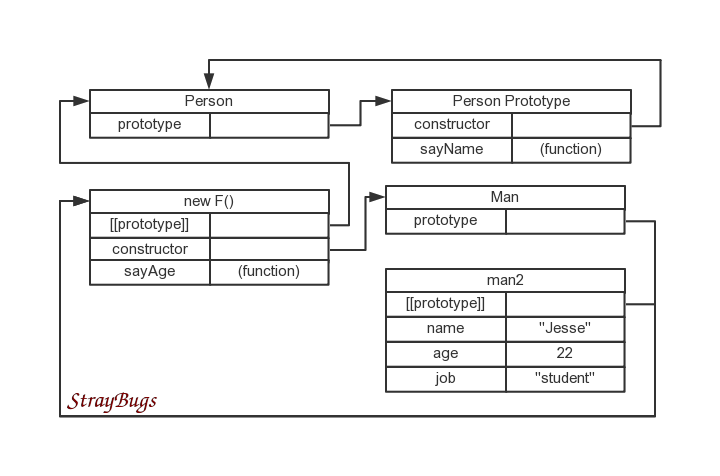
man.sayAge(); //22

alert(man instanceof Man); //true

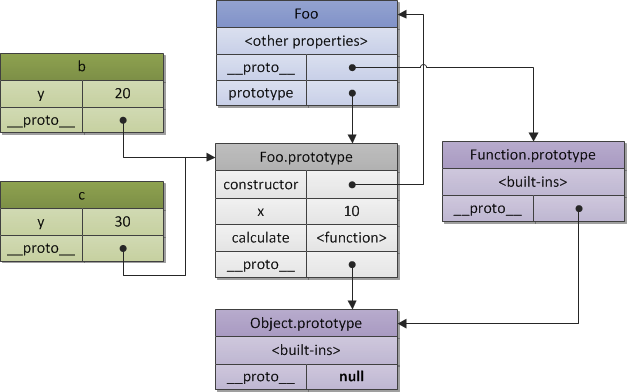
alert(man instanceof Person); //true

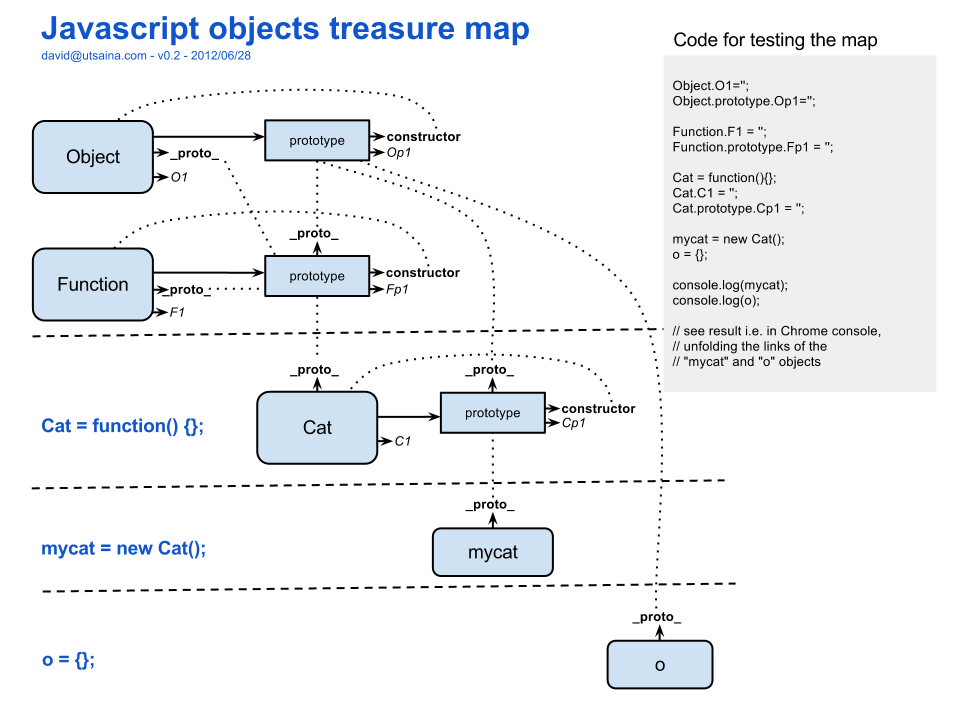
alert(Man.prototype.isPrototypeOf(man)); //true

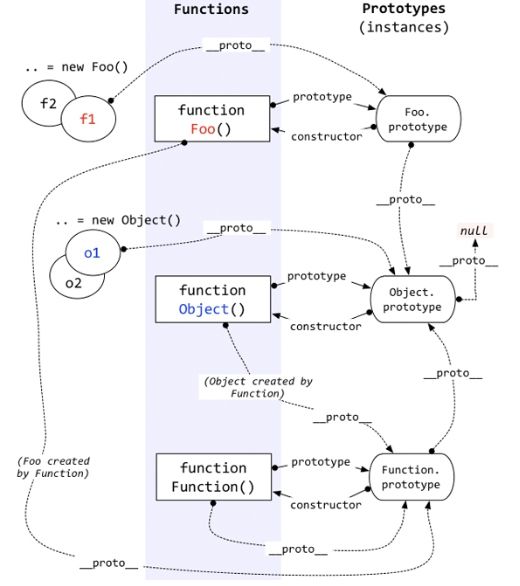
alert(Person.prototype.isPrototypeOf(man)); //true



### \_proto\_和prototype的区别







1 所有的对象都有一个内置属性\_\_proto\_\_（隐式原型）或者说是 [[prototype]]，在ES5之前没有标准的方法访问这个内置属性，但是大多数浏览器都支持通过\_\_proto\_\_来访问。ES5中有了对于这个内置属性标准的Get方法Object.getPrototypeOf()

2 所有的函数都有prototype属性，并且只有函数有

3 在声明函数的时候，会自动创建一个对象(原型)，将引用赋给了函数的prototype属性，并且原型的constructor属性指向该函数，如果修改了函数的prototype属性，那么原型的constructor属性也会跟着改变，于是constructor属性和原来的构造函数也就切断了联系。

4 用该函数创建实例时，所有实例的\_\_proto\_\_属性都指向刚创建的原型，constructor属性都指向构造函数

5 通过Function.prototype.bind方法构造出来的函数是个例外，它没有prototype属性

6 原型链是基于\_\_proto\_\_属性链接起来的，所有对象都继承于Object，所以原型链的最顶端是Object.prototype

7 instanceof 操作符的内部实现机制和隐式原型、显式原型有直接的关系。instanceof的左值一般是一个对象，右值一般是一个构造函数，用来判断左值是否是右值的实例，其原理就是沿着\_\_proto\_\_一直查找到原型链的末端

Function instanceof Object // true

Object instanceof Function // true

Function instanceof Function //true

Object instanceof Object // true

Number instanceof Number //false

### 原型链的原理，new一个对象的过程中都发生了什么

1 创建新对象

2 将新对象的\_\_proto\_\_属性指向构造函数的prototype

3 将构造函数的作用域赋给新对象，即this指向新对象

4 执行构造函数中的代码，即为新对象的属性赋值

5 返回新对象

### 实现跨域请求的几种方法

同源策略（协议、域、端口）

|  |  |  |
| --- | --- | --- |
| URL | 说明 | 是否允许通信 |
| http://www.a.com/a.js http://www.a.com/b.js | 同一域名下 | 允许 |
| http://www.a.com/lab/a.js http://www.a.com/script/b.js | 同一域名下不同文件夹 | 允许 |
| http://www.a.com:8000/a.js http://www.a.com/b.js | 同一域名，不同端口 | 不允许 |
| http://www.a.com/a.js https://www.a.com/b.js | 同一域名，不同协议 | 不允许 |
| http://www.a.com/a.js http://70.32.92.74/b.js | 域名和域名对应ip | 不允许 |
| http://www.a.com/a.js http://script.a.com/b.js | 主域相同，子域不同 | 不允许 |
| http://www.a.com/a.js http://a.com/b.js | 同一域名，不同二级域名（同上） | 不允许（cookie这种情况下也不允许访问） |
| http://www.cnblogs.com/a.js http://www.a.com/b.js | 不同域名 | 不允许 |

特别注意两点：  
第一，如果是协议和端口造成的跨域问题“前台”是无能为力的，  
第二：在跨域问题上，域仅仅是通过“URL的首部”来识别而不会去尝试判断相同的ip地址对应着两个域或两个域是否在同一个ip上。

1. CORS

默认情况下，XHR对象只能访问与包含它的页面位于同一域中的资源。CORS（跨域资源共享）定义了在必须访问跨域资源时，浏览器和服务器应该如何沟通。

基本思想就是使用自定义的HTTP头部Origin，包含请求页面的源信息（协议、域名和端口），以便服务器根据头部信息决定是否响应。如果请求被接受，就在响应头部Access-Control-Allow-Origin中回发相同的源信息。浏览器检验响应头部信息，如果头部不存在或者源信息不一致，浏览器驳回请求。注：请求和响应中都不包含cookie信息。

IE对CORS的实现

IE8中引入了XDR实现安全可靠地跨域通信。所有的XDR请求都是异步执行的。响应有效触发load事件，失败触发error事件。只能获得响应的原始文本，不能确定响应的状态代码。

XDR的特点：

1. 只能设置请求头部的Content-Type字段（在发送POST请求时设置数据格式）
2. 不能访问响应头部
3. 只能发送POST和GET请求
4. 请求和响应都不包含cookie

其他浏览器对CORS的实现

其他浏览器的XHR对象实现了对CORS的原生支持。只需在open（）方法中传入绝对URL即可。

跨域XHR对象的特点：

1. 可以访问status和statusText
2. 支持同步请求
3. 不能使用setRequestHeader设置自定义头部
4. 不能发送和接受cookie

附：CORS可以通过Preflighted Requests的透明服务器验证机制支持使用自定义头部、GET和POST以外的方法以及不同类型的主体内容；

CORS通过将withCredentials属性设置为true可以指定某请求应该发送凭据。如果服务器接受带凭据的请求，就在响应头部加上Access-Control-Allow-Credentials：true。

1. 图像ping

<img>标签的src属性。

通过动态创建图像并监听load和error事件实现与服务器进行简单的单向通信。

应用示例：跟踪用户点击页面或动态广告曝光次数

缺点：只能GET请求；无法访问服务器的响应文本。

1. JSONP

JSON with padding的简写，是被包含在函数调用中的JSON。形式如：

callback（{ “name”： “xxx” }）；

JSONP由两部分组成：回调函数和数据。回调函数是响应到来时调用的函数，函数名字在请求中指定。通过动态创建<script>元素，设置src为跨域的URL。请求完成之后，JSONP作为有效的JavaScript代码加载到页面中立即执行。

优点：简单易用、双向通信、兼容老浏览器

缺点：可能会不安全，其他域可能会夹带恶意代码；很难确定JSONP请求是否失败；只支持GET请求

1. document.domain + iframe 可以解决主域相同，子域不同的跨域通信

在当前页面和iframe中都设置document.domain = 主域

缺点：一个站点受到攻击后，另一个站点会因此引起安全漏洞；主域必须相同

1. 动态创建script
2. location.hash + iframe

在不同域需要通信的页面中创建隐藏的iframe，利用location.hash互相传值。

优点：可以解决域名完全不同的跨域请求；并且是双向通信；

缺点：数据量受到url大小的限制，而且传递的数据类型有限；数据直接暴露在url中，存在安全问题；需要通过轮询得知hash的变化；有的浏览器会在hash变化是增加一条历史记录，这样会影响用户体验。

1. window.name + iframe

该属性用于获取/设置窗口的名称。其特征在于：一个窗口的生命周期内，窗口载入的所有页面共享该值，且都具有对该属性的读写权限。这意味着如果不修改该值，那么在不同页面加载之后该值也不会变，且其支持长达 2MB 的存储量。

步骤：

1. 在 a.github.io/a.html 中创建 iframe 指向 b.github.io/b.html (页面会将自身的 window.name 附在 iframe 上)
2. 给 a.github.io/a.html 添加监听 iframe 的 onload 事件，在该事件中将 iframe 的 src 设置为本地域的代理文件（代理文件和a.html处于同一域下，可以相互通信），同时可以传出 iframe 的 name 值
3. 获取数据后销毁 iframe，释放内存，同时也保证了安全

优点：巧妙地绕过了浏览器的跨域访问限制，但同时它又是安全操作；可传输数据量大

1. postMessage（HTML5中的XMLHttpRequest Level 2中的API）
2. a.com/index.html中的代码：



1. b.com/index.html中的代码：



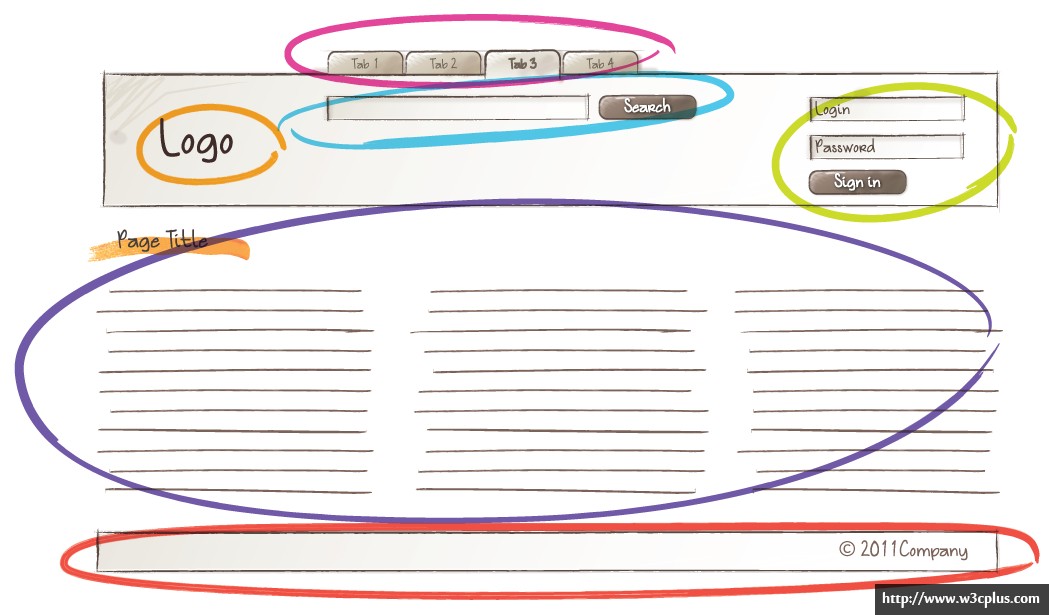
1. web sockets

web sockets是一种浏览器的API，它的目标是在一个单独的持久连接上提供全双工、双向通信。(同源策略对web sockets不适用)

只要服务器支持web sockets，就可以通过它进行跨域通信。

### BEM

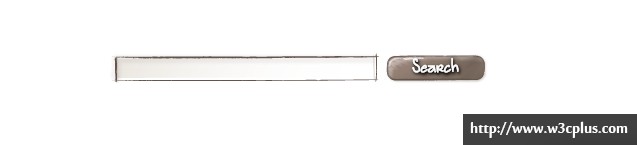
BEM：Block，Element，Modifier。类似于OOP，BEM是一种用代码和一系列模式来描述现实情况的方法，只考虑程序实体和程序语言无关。利用BEM规则指导网站的构建。



遵循统一的原则和规范可以有助于团队协作和沟通。

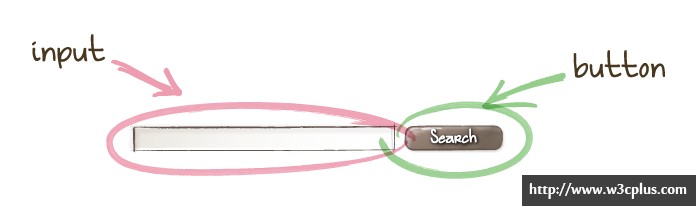
1. 块

一个块是一个独立的实体，如积木一般。块可以复合。下图是搜索块



1. 元素

元素是块的一部分，具有一定的功能，依赖上下文。下图是搜索块中的输入域和按钮



块和元素构成了页面内容，并且遵循着一定的顺序。

为了描述页面布局，需要为每一个块和元素定义可识别的唯一关键字（一般是名字）。

1. 块的独立性

随着项目发展，我们会在页面中添加、移动或者删除某些块，因此每个块必须是独立的，可以放置在页面的任何位置，包括嵌套在其他块中。

1. CSS和BEM

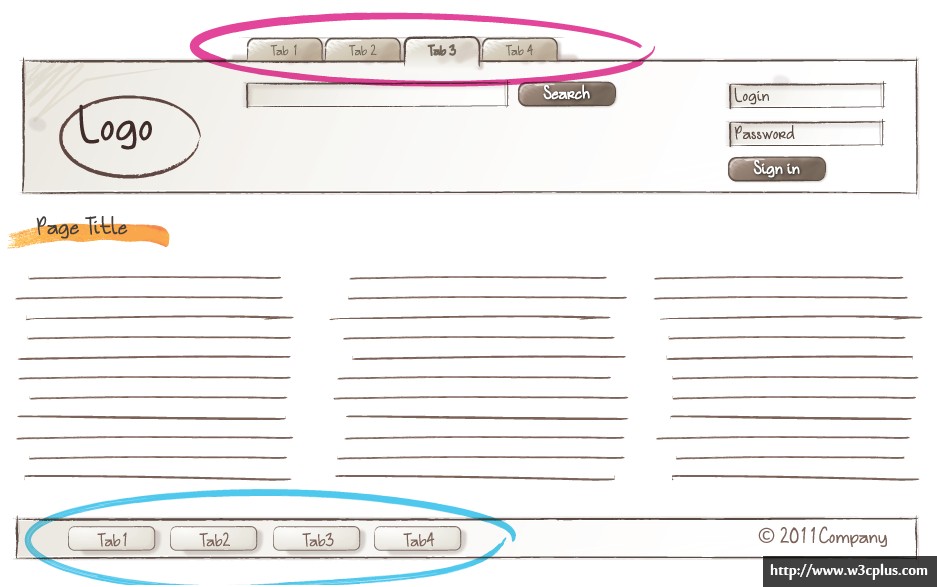
从CSS的角度来看：

1. 每一个块（或者元素）必须有一个唯一的名字（即CSS类名）
2. HTML元素不能作为CSS选择器，因为这样上下文相关，不独立
3. 尽量少用级联（cascade）选择器

CSS命名方案（BEM原则）：

1. 块的CSS类名就是块的名字
2. 一个元素的CSS类名是一个块名和元素名的组合，中间用两个下划线（B\_\_E）
3. 块名和元素名用连字符分割单词（block-name--element-name）
4. 块修饰符

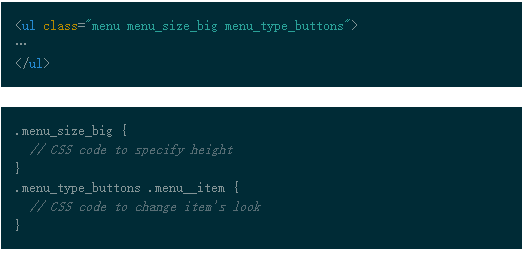
场景：我们经常需要创建一个和已存在的块相似的块，只是外观和行为有些许改变。



为了避免重复开发，引入修饰符（modifier）的概念。修饰符作为块/元素的属性，代表块/元素在外观/行为上的改变。一个修饰符有一个名字和一个值。可以同时使用多个修饰符。

1. 从HTML/CSS角度看修饰符

对于块/元素来说，修饰符就是附加的CSS类。



我们用一个下划线来分隔块名（或元素名）和修饰符名，再用另一个下划线来分隔修饰符名和它对应的值。

1. 块的一致性

复用块不仅要实现相同的CSS，还要复用行为。





### 一道JS编程题



涉及知识点：变量定义提升、函数声明提升、this指针指向、作用域链、运算符优先级、原型与继承、全局变量污染、对象属性和原型属性。

**第二问，考察变量定义提升和函数声明提升**

function Foo() {

    getName = function () { alert (1); };

    return this;

}

var getName;//只提升变量声明

function getName() { alert (5);}//提升函数声明，覆盖var的声明

Foo.getName = function () { alert (2);};

Foo.prototype.getName = function () { alert (3);};

getName = function () { alert (4);};//最终的赋值再次覆盖function getName声明

getName();//最终输出4

**第三问，考察this指向，作用域链，全局变量污染**

Foo()执行之后，覆盖了window的全局变量getName；函数返回的this指向window。

**第五问，考察运算符优先级**



成员访问“.”的优先级高于new（无参数列表）

new Foo.getName(); 等价于new (Foo.getName)();所以实际上将getName函数作为构造函数执行。

**第六问，考察构造函数的返回值、运算符优先级**

由于new（带参数列表）的优先级和成员访问“.”相同，所以从左至右执行

new Foo().getName()；等价于(new Foo()).getName()；

构造函数的返回值问题：

1. 无返回值，返回实例化对象
2. 返回值是非引用类型，也返回实例化对象
3. 返回值是引用类型，返回该值

new Foo()返回this，即当前实例化对象，所以返回的是Foo的实例，然而没有getName属性，所以在原型中寻找getName。

**第七问，考察运算符优先级**

new new Foo().getName();

等价于new (new Foo()).getName();

等价于new ((new Foo()).getName)();

最终其实是将Foo原型的getName函数作为构造函数调用。

### 元素居中

**水平居中：**

inline 元素：text-align: center

block 元素：margin: 0 auto

多个block元素并排：

HTML：

<main class="inline-block-center">  
 <div>  
 I'm an element that is block-like with my siblings and we're centered in a row.  
 </div>  
 <div>  
 I'm an element that is block-like with my siblings and we're centered in a row.  
 </div>  
 <div>  
 I'm an element that is block-like with my siblings and we're centered in a row.  
 </div>  
</main>

CSS：

.inline-block-center {  
 text-align: center;  
}  
.inline-block-center div {  
 display: inline-block;  
 text-align: left;  
}

用flexbox：

.flex-center {  
 display: flex;  
 justify-content: center;  
}

**垂直居中：**

inline元素：

一行：padding-top = padding-bottom或者令line-height = height即可

多行：

HTML：

<div class="center-table">  
 <p>I'm vertically centered multiple lines of text in a CSS-created table layout.</p>  
 </div>

CSS：// 伪装成table的cell，设置vertical-align即可

.center-table {  
 display: table;  
 }  
 .center-table p {  
 display: table-cell;  
  vertical-align: middle;  
 }

用flexbox：

.flex-center-vertically {

display: flex;

justify-content: center;

flex-direction: column;

height: 400px;

}

block元素：

知道需要居中元素的height：

.parent {

position: relative;

}

.child {

position: absolute;

top: 50%;

height: 100px;

margin-top: -50px; /\* account for padding and border if not using box-sizing: border-box; \*/

}

不知道需要居中元素的height：

main {

position: relative;

}

main div {

position: absolute;

top: 50%;

transform: translateY(-50%);

}

用flexbox:

.parent {

display: flex;

flex-direction: column;

justify-content: center;

}

**水平和垂直居中：**

固定宽度和高度的元素：

.parent {

position: relative;

}

.child {

width: 300px;

height: 100px;

padding: 20px;

position: absolute;

top: 50%;

left: 50%;

margin: -70px 0 0 -170px;

}

未知宽度和高度：

.parent {

position: relative;

}

.child {

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%, -50%);

}

使用flexbox：

.parent {

display: flex;

justify-content: center;

align-items: center;

}

### 介绍CSS3中的弹性布局

### 函数的两种创建方式的区别以及NFE和IIFE

从NFE(具名函数表达式)引出的一些函数知识点总结：

1. **函数表达式VS函数声明**

ECMA解释说函数表达式和函数声明的区别主要在于函数声明必须有一个Identifier（或者说是函数名字）而函数表示式可以没有。

函数声明 :

function Identifier ( FormalParameterList opt ){ FunctionBody }

函数表达式：

function Identifier opt ( FormalParameterList opt ){ FunctionBody }

<opt：表示option>

没有函数名字的时候，一定是匿名函数表达式；函数如果有名字，就需要通过所处上下文来判断。如果是传参、赋值或者new运算符，就应该是函数表达式；如果是孤零零地在函数体内或者全局域中，就应该是函数声明。比如：

function foo(){} // declaration, since it's part of a Program

var bar = function foo(){}; // 表达式, 因为是赋值

new function bar(){}; // expression, since it's part of a NewExpression

(function(){

function bar(){} // declaration, since it's part of a FunctionBody

})();// 括号包含的是函数表达式

区别：

1. 函数声明最后没有分号，而函数表达式有
2. 函数声明提前
3. 通过条件语句控制函数声明的行为并未标准化，因此不同环境下可能会得到不同的结果，所以永远都不要依赖条件控制来声明函数，而应该使用函数表达式。

// Never do this!

// Some browsers will declare `foo` as the one returning 'first',

// while others — returning 'second'

if (true) {

function foo() {

return 'first';

}

}

else {

function foo() {

return 'second';

}

}

foo();

// Instead, use function expressions:

var foo;

if (true) {

foo = function() {

return 'first';

};

}

else {

foo = function() {

return 'second';

};

}

foo();

1. **NFE**

具有Identifier（或者说函数名字）的函数表达式被称作具名函数表达式（named function expression）：

var bar = function foo(){};

只能在函数内部访问具名函数表达式的名字foo。

具名函数表达式的作用主要是追踪栈中有函数名，利于调试，其次可以在递归时使用（代替arguments.callee）。

1. **IIFE**

立即执行函数表达式（Immediately Invoked Function Expression）的形式如下：

(function(){

//...

})();

立即执行函数表达式是指程序运行到此时函数立即执行。用法：

1. 传递参数为window，可以更快地访问全局作用域里面的变量，不需要沿着作用域链进行查找
2. 传递参数为函数
3. 利用函数作用域创建块级作用域，防止全局变量污染
4. 模块化编程、测试和部署
5. **函数的特性、模式以及高级用法**

**函数特性：**

1. 函数是一等对象（first-class）

* 可以在程序执行时动态创建函数
* 可以将函数赋值给变量，可以将函数的引用拷贝给另一个变量，可以扩充和删除
* 可以将函数作为参数传递，可以作为返回值返回
* 可以添加属性和方法

“我们首先当它是一个对象，具有可执行的特性”

1. 函数提供作用域支持

在JavaScript中没有块级作用域，只有函数作用域。

**回调模式：**

1. 最简单的回调：函数接受一个函数作为参数并在函数中调用传入的函数。此时，传入的函数就叫做回调函数。
2. 回调和作用域：如果回调函数是匿名函数或者全局函数，在函数中就可以直接调用。如果回调函数是对象的方法并使用了对象的属性，此时就需要注意作用域的问题。除了传入回调函数，还需要传入回调函数所属的对象，并在函数中利用call()和apply()指定回调函数的作用域。
3. 异步事件监听和延时：JavaScript中的事件监听和延时函数都用到了回调函数。
4. 类库中的回调：在类库的设计时经常使用回调模式。设计时着重核心功能的实现，尽可能保持可复用和通用，但同时提供回调的入口作为“钩子（hook）”，定制需要的特性使类库变得可扩展和可定制。

**重定义函数：**

函数可以在运行中动态定义，用新函数覆盖掉旧函数。

当函数中包含一些初始化操作并只需要执行一次时，或者函数里面的控制流每次都是一样时，这种模式非常合适，可以避免执行重复的代码，提高应用的执行效率。这种模式也被称为函数的懒惰定义。

缺陷：原函数的功能丢失；如果这个函数被重定义为不同的名字，被赋值给不同的变量，或者是作为对象的方法使用，那么重定义的部分并不会生效，原来的函数依然会被执行。

**函数属性——记忆模式**

将函数执行结果保存为函数的自定义属性，避免函数下次调用时重复复杂的计算。

var myFunc = function foo () {

var cachekey = JSON.stringify(Array.prototype.slice.call(arguments)),

result;

if (!foo.cache[cachekey]) {

result = {};

// ……复杂的计算……

foo.cache[cachekey] = result;

}

return foo.cache[cachekey];

};

// 缓存

myFunc.cache = {};

**函数柯里化(Currying）:**

解释：让函数理解并处理部分应用的过程叫做柯里化。柯里化是一个变换函数的过程。

通用的柯里化函数：

function schonfinkelize(fn) {

var slice = Array.prototype.slice,

stored\_args = slice.call(arguments, 1);

return function () {

var new\_args = slice.call(arguments),

args = stored\_args.concat(new\_args);

return fn.apply(null, args);

};

}

借用Array.prototype中slice()方法将arguments转换成数组，通过闭包访问存储的旧参数。

用法：

// 普通函数

function add(a, b, c, d, e) {

return a + b + c + d + e;

}

// 参数个数可以随意分割

schonfinkelize(add, 1, 2, 3)(5, 5); // 16

// 两步柯里化

var addOne = schonfinkelize(add, 1);

addOne(10, 10, 10, 10); // 41

var addSix = schonfinkelize(addOne, 2, 3);

addSix(5, 5); // 16

使用场景：

当你发现自己在调用同样的函数并且传入的参数大部分都相同的时候，就是考虑柯里化的理想场景了。你可以通过传入一部分的参数动态地创建一个新的函数。这个新函数会存储那些重复的参数（所以你不需要再每次都传入），然后再在调用原始函数的时候将整个参数列表补全。

### 函数的闭包的解释和用法

闭包（closure）是Javascript语言的一个难点，也是它的特色。

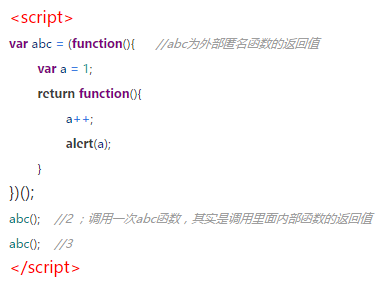
1. 闭包的特性
2. 函数嵌套
3. 函数内部可以引用外部的参数和变量
4. 参数和变量不会被垃圾回收机制回收
5. 闭包的定义和优缺点

闭包是指有权访问另一个函数作用域中的变量的函数。创建闭包的最常见的方式就是在函数内创建函数。

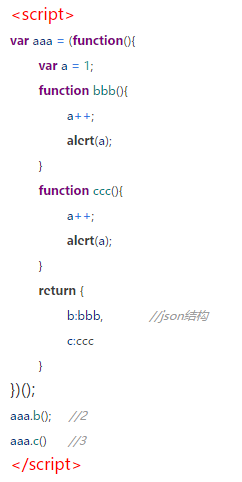
缺点：内存常驻，增加内存使用量，使用不当会造成内存泄漏。

优点：希望某变量长期驻扎在内存中；避免全局变量的污染；创造私有成员

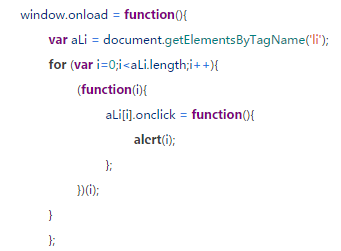
1. 闭包的使用
2. 模块化代码，避免全局变量的污染



1. 创建私有成员



1. 在循环中为DOM元素绑定事件函数



### LocalStorage和SessionStorage区别

### jQuery中的ready（）函数有什么作用

### HTTPS和HTTP区别与联系

TCP/IP协议族分为四层：应用层、传输层、网络层和数据链路层。其中HTTP协议处于应用层，TCP位于传输层和IP位于网络层。HTTP是用于客户端和服务端进行通信的一种无状态协议。由于HTTP协议传输的数据是明文的，存在数据嗅探和篡改的安全问题。于是就有了SSL（Secure Sockets Layer）/TLS（Transport Layer Security）协议，用于对HTTP协议传输的数据进行加密，从而诞生了HTTPS。

HTTPS的原理简要介绍：

在客户端和服务端传输数据之前先进行一次握手，确定双方加密传输数据的密码信息，握手之后，所有的通信数据都采用一致的密码加密传输。在整个过程中，TLS/SSL使用了对称加密、不对称加密以及HASH算法。其中，非对称加密用于加密握手过程中生成的密码，对称加密用于对真正传输的数据进行加密和解密，而HASH算法用于验证数据的完整性。

HTTPS的优势之处：

1. 使用HTTPS协议可以认证用户和服务器，确保数据发送到正确的客户端和服务器上；
2. HTTPS协议是由SSL/TLS+HTTP协议构建的可加密传输、身份认证的网络协议，比HTTP更加安全，可防止数据在传输过程中被窃取和改变，确保数据的完整性；
3. HTTPS是现行架构下最安全的解决方案，虽然不是绝对安全，但至少大幅度增加了中间人攻击的额成本；

HTPPS的缺点：

1. HTTPS会使页面的加载时间延长，连接缓存不如HTTP高效，对服务器端资源占用高，增加数据开销和功耗；
2. HTTPS协议的加密范围有限，对黑客攻击、服务器劫持等方面几乎不起作用，而且SSL证书的信用链体系并不安全；
3. SSL证书需要花钱购买，并且在绑定IP时不能绑定多个域名；

### session和cookie的对比与各自的优缺点以及使用场景