(2) 前端基础 (3) JS(1)1

一 类的继承与创建

1 类的创建（ES5 实例方法【每生成一个实例，就会不断有新的实例方法产生、浪费空间】和 原型方法 的区别，）

（1）类的创建（es5）：new一个function，在这个function的prototype里面增加属性和方法。

下面来创建一个Animal类：

```
// 定义一个动物类

function Animal (name) {

// 属性

this.name = name || 'Animal';

// 实例方法

this.sleep = function(){

console.log(this.name + '正在睡觉！');

}

}

// 原型方法

Animal.prototype.eat = function(food) {

console.log(this.name + '正在吃：' + food);

};
```

这样就生成了一个Animal类，实力化生成对象后，有方法和属性。

2 类的继承 —— 原型链继承【核心点： Cat.prototype = new Animal(); 】。

（2）类的继承——原型链继承

--原型链继承

```
function Cat(){ }
Cat.prototype = new Animal();
Cat.prototype.name = 'cat';
//  Test Code
var cat = new Cat();
console.log(cat.name);
console.log(cat.eat('fish'));
console.log(cat.sleep());
console.log(cat instanceof Animal); //true
console.log(cat instanceof Cat); //true
```

介绍：在这里我们可以看到new了一个空对象,这个空对象指向Animal并且Cat.prototype指向了这个空对象，这种就是基于原型链的继承。

3 构造继承 Animal.call（this）【”有点类似 super(this)的味道”】
注意：使用父类的构造函数来去增强子类实例，
等于复制了父类的 实例属性、方法 给了 子类，
那父类通过 Animal.prototype.getName = function() {} 的原型方法是怎么样的？？】

结论：智能集成父类实例的属性 和 方法，
不能继承原型上的属性和方法【“因为没有 Cat.prototype = new Animal(); 的语句？？！”】。

（3）构造继承：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

```
function Cat(name){
Animal.call(this);
this.name = name || 'Tom';
}
// Test Code
var cat = new Cat();
console.log(cat.name);
console.log(cat.sleep());
console.log(cat instanceof Animal); // false
console.log(cat instanceof Cat); // true
```

特点：可以实现多继承

缺点：只能继承父类实例的属性和方法，不能继承原型上的属性和方法。

4 实例继承 和 拷贝继承【实用性不强，这里不多举例。】

实例继承：为父类实例添加新特性，作为子类实例返回

拷贝继承：拷贝父类元素上的属性和方法

上述两个实用性不强，不一一举例。

5 组合继承（相当于 构造继承【Animal.call(this)】 和 原型继承【Cat.prototype = new Animal(); 但别忘了 Cat.prototype.constructor = Cat】 的组合体）。

结论：既可以继承父类实例的属性和方法，
又可以继承父类的原型属性和方法。
但调用了 2次父类的 构造函数，生成了2份实例？？
所以就有了 寄生组合继承的出现。

```
function Cat(name){
Animal.call(this);
this.name = name || 'Tom';
}
Cat.prototype = new Animal();
Cat.prototype.constructor = Cat;
// Test Code
var cat = new Cat();
console.log(cat.name);
console.log(cat.sleep());
console.log(cat instanceof Animal); // true
console.log(cat instanceof Cat); // true
```

特点：可以继承实例属性/方法，也可以继承原型属性/方法

缺点：调用了两次父类构造函数，生成了两份实例

6 寄生组合继承【关键在于 Super的出现？？！ 砍掉父类的实例属性，这样、在调用2次父类的构造函数时，就不会2次初始化实力方法/属性！！！】

```
function Cat(name){
Animal.call(this);
this.name = name || 'Tom';
}
(function(){
```

// 创建一个没有实例方法的类

```
var Super = function(){};
Super.prototype = Animal.prototype;
```

//将实例作为子类的原型

```
Cat.prototype = new Super();
})();
// Test Code
var cat = new Cat();
console.log(cat.name);
console.log(cat.sleep());
console.log(cat instanceof Animal); // true
console.log(cat instanceof Cat); //true
```

较为推荐

7 如何解决异步回调地狱？？
promise、generator、async/await。

8 前端中的事件流【这里是说的 事件捕获、目标、冒泡 阶段。 不是事件循环 非 Event Loop 事件轮训】
事件流：描述的是从页面接收事件的顺序。
3阶段【捕获、目标、冒泡阶段】。
addEventListener 接收3个参数，如 click, fn, true；
第3个参数useCaptrue 默认为 true，表示在 捕获阶段就调用 事件处理程序（如这里的 fn函数），
false就在 冒泡时，
目标阶段没有对应的参数。

HTML中与javascript交互是通过事件驱动来实现的，例如鼠标点击事件onclick、页面的滚动事件onscroll等等，可以向文档或者文档中的元素添加事件侦听器来预订事件。想要知道这些事件是在什么时候进行调用的，就需要了解一下“事件流”的概念。

什么是事件流：事件流描述的是从页面中接收事件的顺序,DOM2级事件流包括下面几个阶段。

事件捕获阶段

处于目标阶段

事件冒泡阶段

addEventListener：addEventListener 是DOM2 级事件新增的指定事件处理程序的操作，这个方法接收3个参数：要处理的事件名、作为事件处理程序的函数和一个布尔值。最后这个布尔值参数如果是true，表示在捕获阶段调用事件处理程序；如果是false，表示在冒泡阶段调用事件处理程序。

IE只支持事件冒泡。

注意：IE只支持事件冒泡、不支持 捕获？？【这是什么历史原因？？】

9 如何让事件 先冒泡 后 捕获？？【标准中是 先捕获、后冒泡， setTimeout 下一轮机制可以吗？？】
在 DOM标准事件模型中，是先捕获后冒泡。
如果要实现先冒泡后捕获的效果，对同一个事件进行 捕获、冒泡的 监听、并对应相应的处理函数。
监听到捕获函数、先暂停执行，直到冒泡事件被捕获后再执行。

在DOM标准事件模型中，是先捕获后冒泡。但是如果要实现先冒泡后捕获的效果，对于同一个事件，监听捕获和冒泡，分别对应相应的处理函数，监听到捕获事件，先暂缓执行，直到冒泡事件被捕获后再执行捕获之间。

10 说一下事件委托【其实应该也叫事件代理。核心：子target 父currentTarget】。
10.1 定义：不直接在发生的DOM上设置监听函数，而是 在其父元素上 设置监听函数！！
父元素通过 事件冒泡【子：target 父：currentTarget】 可以监听到子元素上事件的触发，通过 e.target 去得知是具体哪个子元素 “被用户动了”。

10.2 例子：ul “统一监听了旗下的“ 所有 li 元素！！

简介：事件委托指的是，不在事件的发生地（直接dom）上设置监听函数，而是在其父元素上设置监听函数，通过事件冒泡，父元素可以监听到子元素上事件的触发，通过判断事件发生元素DOM的类型，来做出不同的响应。

举例：最经典的就是ul和li标签的事件监听，比如我们在添加事件时候，采用事件委托机制，不会在li标签上直接添加，而是在ul父元素上添加。

好处：比较合适动态元素的绑定，新添加的子元素也会有监听函数，也可以有事件触发机制。

11 图片懒加载 和 预加载

预加载：提前加载图片，当用户需要查看时可直接从本地缓存中渲染。

懒加载：懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数。

两种技术的本质：两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。
懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

12 mouseover（mouseout） mouseenter（mouseleave） 联系与区别？？

mouseover：当鼠标移入元素或其子元素都会触发事件，所以有一个重复触发，冒泡的过程。对应的移除事件是mouseout

mouseenter：当鼠标移除元素本身（不包含元素的子元素）会触发事件，也就是不会冒泡，对应的移除事件是mouseleave

1 13 改变函数内部this指针的函数 —— call、apply、bind的区别？？

通过apply和call改变函数的this指向，他们两个函数的第一个参数都是一样的表示要改变指向的那个对象，第二个参数，apply是数组，而call则是arg1,arg2...这种形式。通过bind改变this作用域会返回一个新的函数，这个函数不会马上执行。

14 异步加载JS的 3种方法 —— defer、async、创建script标签并插入DOM中：

defer：只支持IE如果您的脚本不会改变文档的内容，可将 defer 属性加入到<script>标签中，以便加快处理文档的速度。因为浏览器知道它将能够安全地读取文档的剩余部分而不用执行脚本，它将推迟对脚本的解释，直到文档已经显示给用户为止。
async，HTML5属性仅适用于外部脚本，并且如果在IE中，同时存在defer和async，那么defer的优先级比较高，脚本将在页面完成时执行。

创建script标签，插入到DOM中

15 Ajax解决浏览器的缓存问题

在ajax发送请求前加上 anyAjaxObj.setRequestHeader("If-Modified-Since","0")。

在ajax发送请求前加上 anyAjaxObj.setRequestHeader("Cache-Control","no-cache")。

在URL后面加上一个随机数： "fresh=" + Math.random()。

在URL后面加上时间搓："nowtime=" + new Date().getTime()。

如果是使用jQuery，直接这样就可以了 $.ajaxSetup({cache:false})。这样页面的所有ajax都会执行这条语句就是不需要保存缓存记录。

16 JS中的垃圾回收机制

16.1 垃圾回收的2种方法： 标记清除、计数引用！！

必要性：由于字符串、对象和数组没有固定大小，所有当他们的大小已知时，才能对他们进行动态的存储分配。JavaScript程序每次创建字符串、数组或对象时，解释器都必须分配内存来存储那个实体。只要像这样动态地分配了内存，最终都要释放这些内存以便他们能够被再用，否则，JavaScript的解释器将会消耗完系统中所有可用的内存，造成系统崩溃。

这段话解释了为什么需要系统需要垃圾回收，JS不像C/C++，他有自己的一套垃圾回收机制（Garbage Collection）。JavaScript的解释器可以检测到何时程序不再使用一个对象了，当他确定了一个对象是无用的时候，他就知道不再需要这个对象，可以把它所占用的内存释放掉了。例如：

```
var a="hello world";
var b="world";
var a=b;
```

//这时，会释放掉"hello world"，释放内存以便再引用

垃圾回收的方法：标记清除、计数引用。

标记清除

这是最常见的垃圾回收方式，当变量进入环境时，就标记这个变量为"进入环境",从逻辑上讲，永远不能释放进入环境的变量所占的内存，永远不能释放进入环境变量所占用的内存，只要执行流程进入相应的环境，就可能用到他们。当离开环境时，就标记为离开环境。

垃圾回收器在运行的时候会给存储在内存中的变量都加上标记（所有都加），然后去掉环境变量中的变量，以及被环境变量中的变量所引用的变量（条件性去除标记），删除所有被标记的变量，删除的变量无法在环境变量中被访问所以会被删除，最后垃圾回收器，完成了内存的清除工作，并回收他们所占用的内存。

引用计数法

另一种不太常见的方法就是引用计数法，引用计数法的意思就是每个值没引用的次数，当声明了一个变量，并用一个引用类型的值赋值给改变量，则这个值的引用次数为1,；相反的，如果包含了对这个值引用的变量又取得了另外一个值，则原先的引用值引用次数就减1，当这个值的引用次数为0的时候，说明没有办法再访问这个值了，因此就把所占的内存给回收进来，这样垃圾收集器再次运行的时候，就会释放引用次数为0的这些值。

用引用计数法会存在内存泄露，下面来看原因：

```
function problem() {
var objA = new Object();
var objB = new Object();
objA.someOtherObject = objB;
objB.anotherObject = objA;
}
```

在这个例子里面，objA和objB通过各自的属性相互引用，这样的话，两个对象的引用次数都为2，在采用引用计数的策略中，由于函数执行之后，这两个对象都离开了作用域，函数执行完成之后，因为计数不为0，这样的相互引用如果大量存在就会导致内存泄露。

特别是在DOM对象中，也容易存在这种问题：

```
var element=document.getElementById ('') ;
var myObj=new Object();
myObj.element=element;
element.someObject=myObj;
```

这样就不会有垃圾回收的过程。

17 eval是什么？？【将字符串解析成JS并执行，但是 过程多了一层 —— 一次解析成JS、一次去运行，较耗性能】

作用：将 对应的字符串解析成JS并执行，应该避免使用eval —— 因为非常耗性能（2次，一次解析成JS、 一次执行！！）

18 前端模块化的理解？？

前端模块化就是复杂的文件编程一个一个独立的模块，比如js文件等等，分成独立的模块有利于重用（复用性）和维护（版本迭代），这样会引来模块之间相互依赖的问题，所以有了commonJS规范，AMD，CMD规范等等，以及用于js打包（编译等处理）的工具webpack

19 说一下 Common.js、AMD、CMD？？

一个模块是能实现特定功能的文件，有了模块就可以方便的使用别人的代码，想要什么功能就能加载什么模块。

Commonjs：开始于服务器端的模块化，同步定义的模块化，每个模块都是一个单独的作用域，模块输出，modules.exports，模块加载require()引入模块。

AMD：中文名异步模块定义的意思。

requireJS实现了AMD规范，主要用于解决下述两个问题。

1.多个文件有依赖关系，被依赖的文件需要早于依赖它的文件加载到浏览器

2.加载的时候浏览器会停止页面渲染，加载文件越多，页面失去响应的时间越长。

语法：requireJS定义了一个函数define，它是全局变量，用来定义模块。

requireJS的例子：

//定义模块

```
define(['dependency'], function(){
var name = 'Byron';
function printName(){
console.log(name);
}
return {
printName: printName
};
});
```

//加载模块

```
require(['myModule'], function (my){
my.printName();
}
```

requirejs定义了一个函数define,它是全局变量，用来定义模块：

define(id?dependencies?,factory)

在页面上使用模块加载函数：

require([dependencies],factory);

总结AMD规范：require（）函数在加载依赖函数的时候是异步加载的，这样浏览器不会失去响应，它指定的回调函数，只有前面的模块加载成功，才会去执行。因为网页在加载js的时候会停止渲染，因此我们可以通过异步的方式去加载js,而如果需要依赖某些，也是异步去依赖，依赖后再执行某些方法。

20 对象深克隆的是简单实现【arr数组类型可以使用 for...in！！ 有时间再次整理所有相关的方法出来】

```
function deepClone(obj){
var newObj= obj instanceof Array ? []:{};
for(var item in obj){
var temple= typeof obj[item] == 'object' ? deepClone(obj[item]):obj[item];
newObj[item] = temple;
}
return newObj;
}
```

21 实现一个 ones 函数，传入的 函数参数 只执行一次【
优先想到闭包。
好像以下代码不行？！！】

```
function ones(func){
var tag=true;
return function(){
if(tag==true){
func.apply(null,arguments);
tag=false;
}
return undefined
}
}
```

22 将原生的ajax封装成promise【主要留意到 new Promise 构造函数传入的东西就很简单了！！】

```
var  myNewAjax=function(url){
return new Promise(function(resolve,reject){
var xhr = new XMLHttpRequest();
xhr.open('get',url);
xhr.send(data);
xhr.onreadystatechange=function(){
if(xhr.status==200&&readyState==4){
var json=JSON.parse(xhr.responseText);
resolve(json)
}else if(xhr.readyState==4&&xhr.status!=200){
reject('error');
}
}
})
}
```

23 JS监听一个对象属性的改变【2方法： ES 5的Object.defineProperty(vue的双向绑定) + ES 6的 new Proxy...】

假设这里有一个 user对象

23.1 在ES 5中，可以通过 Object.defineProperty 来实现已有属性的监听。

```
Object.defineProperty(user,'name',{
set: function(key,value){
}
})
```

缺点：如果 id【为啥是 id？？ 不是 name？？】不在 user对象中，则不能监听 id 变化？？

23.2 ES 6的 Proxy 。

```
var  user = new Proxy({}, {
set: function(target,key,value,receiver){
}
})
```

即使有属性在 user对象中不存在，通过 user.id 来定义也可以监听这个属性的变化！！

24 实现一个私有变量，用 getName 方法可以访问，不能直接访问【defineProperty || 闭包】

24.1 通过 defineProperty

```
obj={
name:yuxiaoliang,
getName:function(){
return this.name
}
}
object.defineProperty(obj,"name",{
```

//不可枚举不可配置

});

24.2 通过函数的创建形式【形成闭包，里面的函数、才能访问其外层函数里面的闭包！！】

```
function product(){
var name='yuxiaoliang';
this.getName=function(){
return name;
}
}
var obj=new product();
```

25 ==、 ===、 Object.is() 的区别

(1) ==

主要存在：强制转换成number,null==undefined

" "==0 //true

"0"==0 //true

" " !="0" //true

123=="123" //true

null==undefined //true

(2)Object.js

主要的区别就是+0！=-0 而NaN==NaN

(相对比===和==的改进)

26 setTimeout、setInterval 和 requestAnimationFrame 之间的区别。

这里有一篇文章讲的是requestAnimationFrame：http://www.cnblogs.com/xiaohuochai/p/5777186.html
与setTimeout和setInterval不同，requestAnimationFrame不需要设置时间间隔，
大多数电脑显示器的刷新频率是60Hz，大概相当于每秒钟重绘60次。大多数浏览器都会对重绘操作加以限制，不超过显示器的重绘频率，因为即使超过那个频率用户体验也不会有提升。因此，最平滑动画的最佳循环间隔是1000ms/60，约等于16.6ms。

RAF采用的是系统时间间隔，不会因为前面的任务，不会影响RAF，但是如果前面的任务多的话，
会响应setTimeout和setInterval真正运行时的时间间隔。

特点：

（1）requestAnimationFrame会把每一帧中的所有DOM操作集中起来，在一次重绘或回流中就完成，并且重绘或回流的时间间隔紧紧跟随浏览器的刷新频率。

（2）在隐藏或不可见的元素中，requestAnimationFrame将不会进行重绘或回流，这当然就意味着更少的CPU、GPU和内存使用量

（3）requestAnimationFrame是由浏览器专门为动画提供的API，在运行时浏览器会自动优化方法的调用，并且如果页面不是激活状态下的话，动画会自动暂停，有效节省了CPU开销。

27 实现一个2列等高布局【原理？？】。

为了实现两列等高，可以给每列加上 padding-bottom:9999px;

margin-bottom:-9999px;同时父元素设置overflow:hidden;

28 实现一个 bind 函数（需要好好研究、并手撕！！）

原理：通过 aplly 或 call 方法去实现。

28.1 初始版本【对！！】：

```
Function.prototype.bind=function(obj,arg){
var arg=Array.prototype.slice.call(arguments,1);
var context=this;
return function(newArg){
arg=arg.concat(Array.prototype.slice.call(newArg));
return context.apply(obj,arg);
}
}
```

28.2 考虑原型链

为什么要考虑？因为在new 一个bind过生成的新函数的时候，必须的条件是要继承原函数的原型

```
Function.prototype.bind=function(obj,arg){
var arg=Array.prototype.slice.call(arguments,1);
var context=this;
var bound=function(newArg){
arg=arg.concat(Array.prototype.slice.call(newArg));
return context.apply(obj,arg);
}
var F=function(){}
```

//这里需要一个寄生组合继承

```
F.prototype=context.prototype;
bound.prototype=new F();
return bound;
}
```

29 JS怎么控制一次加载一张图片，加载完了在加载下一张

2种方法：new Image() 的 onload || onreadystatechange 方法

```
<script type="text/javascript">
var obj=new Image();
obj.src="http://www.phpernote.com/uploadfiles/editor/201107240502201179.jpg";
obj.onload=function(){
```

alert('图片的宽度为：'+obj.width+'；图片的高度为：'+obj.height);

```
document.getElementById("mypic").innnerHTML="<img src='"+this.src+"' />";
}
</script>
<div id="mypic">onloading……</div>
```

(2)方法2

```
<script type="text/javascript">
var obj=new Image();
obj.src="http://www.phpernote.com/uploadfiles/editor/201107240502201179.jpg";
obj.onreadystatechange=function(){
if(this.readyState=="complete"){
```

alert('图片的宽度为：'+obj.width+'；图片的高度为：'+obj.height);

```
document.getElementById("mypic").innnerHTML="<img src='"+this.src+"' />";
}
}
</script>
<div id="mypic">onloading……</div>
```

30 考察事件轮询【event loop】

```
setTimeout(function(){console.log(1)},0);
new Promise(function(resolve,reject){
console.log(2);
resolve();
}).then(function(){console.log(3)
}).then(function(){console.log(4)});
process.nextTick(function(){console.log(5)});
console.log(6);
```

讲解地址：https://github.com/forthealllight/blog/issues/5

31 实现 sleep 的效果

31.1 while循环【注意：易造成 死循环】

```
function sleep(ms){
var start=Date.now(),expire=start+ms;
while(Date.now()<expire);
console.log('1111');
return;
}
```

31.2 通过 promise 去实现【对，再结合 setTimeout】

```
function sleep(ms){
var temple=new Promise(
(resolve)=>{
console.log(111);setTimeout(resolve,ms)
});
return temple
}
sleep(500).then(function(){
//console.log(222)
})
```

```
//先输出了111，延迟500ms后输出222
```

31.3 通过 generator 来实现

```
function* sleep(ms){
yield new Promise(function(resolve,reject){
console.log(111);
setTimeout(resolve,ms);
})
}
sleep(500).next().value.then(function(){console.log(2222)})
```

31.4 通过 async 封装【比较复杂？！！】

```
function sleep(ms){
return new Promise((resolve)=>setTimeout(resolve,ms));
}
async function test(){
var temple=await sleep(1000);
console.log(1111)
return temple
}
test();
```

//延迟1000ms输出了1111

32 实现简单的 Node 的 Events模块【尽量自己动手做一下？！！！】

简介：观察者模式或者说订阅模式，它定义了对象间的一种一对多的关系，让多个观察者对象同时监听某一个主题对象，当一个对象发生改变时，所有依赖于它的对象都将得到通知。

node中的Events模块就是通过观察者模式来实现的：

```
var events=require('events');
var eventEmitter=new events.EventEmitter();
eventEmitter.on('say',function(name){
console.log('Hello',name);
})
eventEmitter.emit('say','Jony yu');
```

这样，eventEmitter发出say事件，通过On接收，并且输出结果，这就是一个订阅模式的实现，下面我们来简单的实现一个Events模块的EventEmitter。

(1)实现简单的Event模块的emit和on方法

```
function Events(){
this.on=function(eventName,callBack){
if(!this.handles){
this.handles={};
}
if(!this.handles[eventName]){
this.handles[eventName]=[];
}
this.handles[eventName].push(callBack);
}
this.emit=function(eventName,obj){
if(this.handles[eventName]){
for(var i=0;o<this.handles[eventName].length;i++){
this.handles[eventName][i](obj);
}
}
}
return this;
}
```

这样我们就定义了Events，现在我们可以开始来调用：

```
var events=new Events();
events.on('say',function(name){
console.log('Hello',nama)
});
events.emit('say','Jony yu');
```

//结果就是通过emit调用之后，输出了Jony yu

(2)每个对象是独立的

因为是通过new的方式，每次生成的对象都是不相同的，因此：

```
var event1=new Events();
var event2=new Events();
event1.on('say',function(){
console.log('Jony event1');
});
event2.on('say',function(){
console.log('Jony event2');
})
event1.emit('say');
event2.emit('say');
```

//event1、event2之间的事件监听互相不影响

//输出结果为'Jony event1' 'Jony event2'

1 33 箭头函数中 this的指向【箭头函数往外看，它的 this指向就是 第一层包住它的 普通函数的this指向！！】

```
var a=11;
function test2(){
this.a=22;
let b=()=>{console.log(this.a)}
b();
}
var x=new test2();
```

//输出22

定义时绑定。