**前端面试题精选**

1. **简述栈和队列的区别？你知道的栈方法和队列方法有哪些？**

区别：

（1）规则不同

        队列：先进先出（First In First Out）FIFO

        栈：先进后出（First In Last Out ）FILO

  （2）对插入和删除操作的限定不同

        队列：只能在表的一端进行插入，并在表的另一端进行删除；

栈：只能在表的一端插入和删除。

（3）遍历数据速度不同

        队列：基于地址指针进行遍历，而且可以从头部或者尾部进行遍历，但不能同时遍历，无需开辟空间，因为在遍历的过程中不影响[数据结构](http://lib.csdn.net/base/datastructure)，所以遍历速度要快；

        栈：只能从顶部取数据，也就是说最先进入栈底的，需要遍历整个栈才能取出来，而且在遍历数据的同时需要为数据开辟临时空间，保持数据在遍历前的一致性。

栈方法：push、pop

队列方法：shift、unshift

1. **如何理解JS是单线程这一说法？为什么不会卡住？（扩展：单线程、多线程、回调、异步、http如何请求数据、http协议三次握手）**

JavaScript语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？

Javascript除了一个主线程外,还配有一个代码队列,这个队列用以存放定时器、HTTP请求、事件响应的回调。

1. **什么是浏览器事件模型？请描述js的事件冒泡和捕获(event bubble and capturing), 如何停止冒泡(bubble)?**

在各种浏览器中存在三种事件模型:原始事件模型( original event model),DOM2事件模型,IE事件模型.其中原始的事件模型被所有浏览器所支持,而DOM2中所定义的事件模型目前被除了IE以外的所有主流浏览器支持。

冒泡：从目标元素向上一级一级触发事件

捕获：从上到下一级一级寻找到目标元素

停止冒泡：

在微软的模型中，你必须设置事件的cancelBubble的属性为true：Window.event.cancelBubble=true

在w3c模型中你必须调用事件的stopPropagation()方法：

e.stopPropagation()

**4.Ajax的全称？原理？优点？**

（详细了解：<http://www.cnblogs.com/SanMaoSpace/archive/2013/06/15/3137180.html>）

全称：Asynchronous js and xml

原理：Ajax的原理简单来说通过XmlHttpRequest对象来向服务器发异步请求，从服务器获得数据，然后用javascript来操作DOM而更新页面。

把服务器端看成一个数据接口(**只负责吐数据**)，它返回的是一个纯文本流，当然，这个文本流可以是XML格式，可以是Html，可以是Javascript代码，也可以只是一个字符串。这时候，XMLHttpRequest向服务器端请求这个页面，服务器端将文本的结果写入页面，这和普通的web开发流程是一样的，不同的是，客户端在异步获取这个结果后，不是直接显示在页面，而是先由javascript来处理，然后再显示在页面。

优点：

（1）无刷新更新数据；

（2）异步与服务器通信；

（3）前端和后端负载平衡；

（4）基于标准被广泛支持；

（5）界面与应用分离。

缺点：

（1）AJAX干掉了Back和History功能，即对浏览器机制的破坏。

（2）AJAX的安全问题。

（3）对搜索引擎支持较弱。

（4）破坏程序的异常处理机制。

（5）违背URL和资源定位的初衷。

（6）AJAX不能很好支持移动设备。

（7）客户端过肥，太多客户端代码造成开发上的成本。

**5.说出下面这些表达式的返回结果**

**null == undefined**

**Boolean( “true” )**

***Boolean( 0 )***

***isNaN( 10 )***

***isNaN( “10” )***

***NaN === NaN***

（true true false false false false）

**6.闭包的原理以及优点、缺点？**

原理：闭包就是能够读取其他函数内部变量的函数。

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

缺点：

1）由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

2）闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

**7.列出js中创建对象的方法，并写出一个js中的继承关系的实现？**

四种常见创建对象的方法：

1 用{}，对象字面量**。**

 先创建再添加：

[复制代码](javascript:void(0);)

1 //创建一个简单对象字面量

2 var person = {};

3 // 加入属性和方法

4 person.name = 'ifcode';

5 person.setName = function(theName) {

6 person.name = theName;

7 }

[复制代码](javascript:void(0);)

2  JS good parts中推荐这种写法:

[复制代码](javascript:void(0);)

1 var person = {

2 **name: 'ifcode',**

3 setName: function(theName) {

4 this.name = theName;

5 }

6 }

[复制代码](javascript:void(0);)

1 var clock={

2 hour:12,

3 minute:10,

4 second:10,

5 showTime:function(){

6 alert(this.hour+":"+this.minute+":"+this.second);

7 }

8 }

9 clock.showTime();//调用

10 //var m = new myObj(); //不支持

[复制代码](javascript:void(0);)

上述两种方式只存在于一个实例的对象，也就是单例模式。

下面是可以创建多个实例的方式：

3 构造函数一般都符合factory pattern，根据默认的规则，构造函数应当首字母大写：

[复制代码](javascript:void(0);)

1 Person = function(defaultName) {

2 this.name = defaultName;

3 this.setName = function(theName) {

4 this.name = theName;

5 }

6 }

7 person = new Person('ifcode'); // new是调用构造函数

[复制代码](javascript:void(0);)

利用构造函数就可以方便地创建多个对象实例了。

4 利用 prototype的构造函数：

[复制代码](javascript:void(0);)

1 Person = function(defaultName) {

2 this.name = defaultName;

3 }

4

5

6 Person.prototype.setName = function(theName) {

7 this.name = theName;

8 }

[复制代码](javascript:void(0);)

所有创建在prototype上得属性和方法，都将被所有对象实例分享。

一般在创建单例的时候，用2 ，在创建多个对象的时候，使用4.

JS继承的实现方式

既然要实现继承，那么首先我们得有一个父类，代码如下：

// 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);

};

**1、原型链继承**

**核心：** 将父类的实例作为子类的原型

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

特点：

1. 非常纯粹的继承关系，实例是子类的实例，也是父类的实例
2. 父类新增原型方法/原型属性，子类都能访问到
3. 简单，易于实现

缺点：

1. 要想为子类新增属性和方法，必须要在new Animal()这样的语句之后执行，不能放到构造器中
2. 无法实现多继承
3. 来自原型对象的引用属性是所有实例共享的（详细请看附录代码： [示例1](javascript:void(0);)）
4. 创建子类实例时，无法向父类构造函数传参

**2、构造继承**

**核心：**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：

1. 解决了1中，子类实例共享父类引用属性的问题
2. 创建子类实例时，可以向父类传递参数
3. 可以实现多继承（call多个父类对象）

缺点：

1. 实例并不是父类的实例，只是子类的实例
2. 只能继承父类的实例属性和方法，不能继承原型属性/方法
3. 无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

**3、实例继承**

**核心：**为父类实例添加新特性，作为子类实例返回

function Cat(name){

var instance = new Animal();

instance.name = name || 'Tom';

return instance;

}

// 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); // false

特点：

1. 不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

1. 实例是父类的实例，不是子类的实例
2. 不支持多继承

**4、拷贝继承**

function Cat(name){

var animal = new Animal();

for(var p in animal){

Cat.prototype[p] = animal[p];

}

Cat.prototype.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：

1. 支持多继承

缺点：

1. 效率较低，内存占用高（因为要拷贝父类的属性）
2. 无法获取父类不可枚举的方法（不可枚举方法，不能使用for in 访问到）

**5、组合继承**

**核心：**通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

Cat.prototype = new Animal();

// 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：

1. 弥补了方式2的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法
2. 既是子类的实例，也是父类的实例
3. 不存在引用属性共享问题
4. 可传参
5. 函数可复用

缺点：

调用了两次父类构造函数，生成了两份实例（子类实例将子类原型上的那份屏蔽了）

**6、寄生组合继承**

**核心：**通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点

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

特点：堪称完美

缺点：实现较为复杂

**8.什么是跨域？你用过的跨域方法有哪些？（同源策略）**

跨域，指的是浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对[JavaScript](http://lib.csdn.net/base/javascript)施加的安全限制。

常见的跨域方法：

1. 通过document.domain跨域；
2. 通过location.hash跨域；
3. 通过HTML5的postMessage方法跨域；
4. 通过jsonp跨域；
5. 通过CORS跨域；
6. 通过window.name跨域

**9.说出几种es6的方法？**

1.let、const关键字；

2.箭头函数；

3.字符串模版；

4.数组对象解构；

5.类的支持，引入class关键字；

6. Promises是处理异步操作的一种模式

**10.写一个方法，去掉一个整型数组中重复的值？**

**1.直接使用es6的Set**

|  |  |
| --- | --- |
| 1  2  3 | let unique = function(array){  return [...new Set(array)];  } |

**2.使用Object**

我们知道Object对象的键是唯一的，可以利用这个特性为数组去重。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | let unique = function (array) {  let ro = {};  let ra = [];  array.forEach(item=>{  if(!ro[item]){  ro[item] = item;  ra.push(item);  }  });    return ra;  } |

**11.写一个方法，判断一个字符串是否回文（回文即“abcba”这种形式）？**

**1.split -🡪 reverse ---🡪 join**

* 将字符串拆分成数组  
  将字符串拆分成数组其实也也有多种方法：
  + split()方法

|  |  |
| --- | --- |
| 1  2  3 | let str\_to\_array = function(str){  return str.split('');  } |

* + Array.prototype.map()

|  |  |
| --- | --- |
| 1  2  3 | let str\_to\_array = function (str) {  return Array.prototype.map.call(str,function(x){return x});  } |

* 数组反转 reverse()
* 拼接成字符串 join()

|  |  |
| --- | --- |
| 1  2  3 | let checkPalindrom = function(str){  return str\_to\_array(str).reverse().join('');  } |

**2.数组的reduceRight()方法**

我们可以直接在字符串上调用数组的reduceRight()方法将字符串逆转

|  |  |
| --- | --- |
| 1  2  3 | let rs = Array.prototype.reduceRight.apply('abc',[function(pre,current){  return pre + current;  },'']); |

|  |  |
| --- | --- |
| 1  2  3 | let checkPalindrom = function(str){  return str == rs(str);  } |

**3.栈数据结构**

我们在学习栈这个数据结构的时候，老师讲的最生动的一个例子就是，判断回文有木有。  
先将字符串中字符依次入栈，然后出栈组成新的字符串，即为逆转的字符串，然后做比较。

**12.描述angular中的数据绑定(data-binding)，描述angular依赖注入(dependency injection)？**

ng-bind 单向数据绑定（$scope -> view），用于数据显示，简写形式是 {{}}。

ng-model 是双向数据绑定（$scope -> view and view -> $scope），用于绑定值会变化的表单元素等。

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

在软件工程中，依赖注入是实现控制反转的一种软件设计模式，一个依赖是一个被其他对象(client)调用的对象（服务）,注入则是将被依赖的对象(service)实例传递给依赖对象(client)的行为。将 被依赖的对象传给依赖者，而不需要依赖者自己去创建或查找所需对象是DI的基本原则。 依赖注入允许程序设计遵从依赖倒置原则（简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合） 调用者(client)只需知道服务的接口，具体服务的查找和创建由注入者(injector)负责处理并提供给client，这样就分离了服务和调用者的依赖，符合低耦合的程序设计原则。

#### Angular依赖注入分析

先看看下面这段 hello,world代码 （注意：设置了严格模式或压缩混淆代码后 下面的代码不能正常工作,后面有解释）

angular.module('myApp', [])

.controller('Ctl', function ($scope, $log) {

$scope.name = 'leonwgc';

$log.log('hello,world');

});

上面这段代码就用到了angular的依赖注入，代码首先创建了一个myApp模块，然后在此模块中创建了Ctl控制器，创建控制器函数的第二个参数则是控制器的构造函数， 构造函数声明了对$scope和$log服务的依赖。 当构造函数执行时， 即可获得$scope和$log服务实例，进行操作。 从我们前面对DI的了解，$scope和$log是由注入器injector 提供，知道了injector的存在，我们直接从angular的源码中将其找出，如下：

[复制代码](javascript:void(0);)

function createInternalInjector(cache, factory) {

// 中间一段略去...

// 调用client

function invoke(fn, self, locals, serviceName) {

if (typeof locals === 'string') {

serviceName = locals;

locals = null;

}

var args = [],

// 查询依赖

$inject = createInjector.$$annotate(fn, strictDi, serviceName),

length, i,

key;

// 中间一段略去...

// 遍历$inject数组调用getService获取服务....

//开始执行client , args则是依赖的全部服务,injector都为我们创建好了

return fn.apply(self, args);

}

// 中间一段略去...

// 这里返回公开的injector对象

return {

// 执行DI方法,比如上面的控制器函数

// invoke方法首先就是调用annotate取得依赖

// 然后调用get取得服务

// 如果缓存中没有服务，get内部调用instantiate创建服务并缓存

// 最后利用function.apply传入依赖并执行

invoke: invoke,

// 实例化(创建)服务

instantiate: instantiate,

// 获取服务(如果缓存中有，直接从缓存拿，没有则调用instantiate创建并放入缓存,下次直接从缓存拿)

get: getService,

// 获得依赖服务

annotate: createInjector.$$annotate,

// 检查缓存中是否包含服务

has: function(name) {

return providerCache.hasOwnProperty(name + providerSuffix)

|| cache.hasOwnProperty(name);

}

};

}

[复制代码](javascript:void(0);)

源码中查询依赖的源码如下：

[复制代码](javascript:void(0);)

function annotate(fn, strictDi, name) {

var $inject,

fnText,

argDecl,

last;

if (typeof fn === 'function') {

// 如果我们直接给函数添加了$inject依赖

// 则直接返回依赖，后面不做处理

if (!($inject = fn.$inject)) {

$inject = [];

if (fn.length) {

if (strictDi) {

if (!isString(name) || !name) {

name = fn.name || anonFn(fn);

}

throw $injectorMinErr('strictdi',

'{0} is not using explicit annotation...', name);

}

// 针对直接在构造函数中使用服务的情况

// 使用function.toString() 然后正则匹配出依赖的对象

// 所以上面例子如果混淆了代码就呵呵了

// 最后存入$inject数组

fnText = fn.toString().replace(STRIP\_COMMENTS, '');

argDecl = fnText.match(FN\_ARGS);

forEach(argDecl[1].split(FN\_ARG\_SPLIT), function(arg) {

arg.replace(FN\_ARG, function(all, underscore, name) {

$inject.push(name);

});

});

}

//给构造函数添加$inject属性

fn.$inject = $inject;

}

} else if (isArray(fn)) {

last = fn.length - 1;

assertArgFn(fn[last], 'fn');

// 如果是数组格式，则依赖对象是数组的第一个到倒数第二个对象

// 要调用的函数则是数组的最后一个元素

$inject = fn.slice(0, last);

} else {

assertArgFn(fn, 'fn', true);

}

// 返回依赖数组

return $inject;

}

[复制代码](javascript:void(0);)

看了上面的源码片段和解释，想必大家对angular的依赖注入有了整体的认识。

下面是另外两种推荐的声明依赖的方式

#### 1. 数组注释 (推荐), js压缩混淆不会有影响。

angular.module('myApp', [])

.controller('Ctl', ['$scope', '$log', function ($scope, $log) {

$scope.name = 'leonwgc';

$log.log('hello,world');

}]);

#### 2.$inject 属性 ,js压缩混淆不会有影响

angular.module('myApp', [])

.controller('Ctl', Ctrl);

function Ctrl($scope, $log) {

$scope.name = 'leonwgc';

$log.log('hello,world');

}

// 给构造函数添加$inject属性,

// $inject是一个数组，元素是依赖的服务名.

Ctrl.$inject = ["$scope", "$log"];

**13.描述一下vue父子组件之间是如何通信的，知道非父子组件之间是如何通信的吗？**

### 1.父组件传递数据给子组件

父组件数据如何传递给子组件呢？可以通过props属性来实现

父组件：

<parent>

<child :child-msg="msg"></child>//这里必须要用 - 代替驼峰

</parent>

data(){

return {

msg: [1,2,3]

｝,

子组件通过props来接收数据:   
方式1：

props: ['childMsg']

方式2 :

props: {

childMsg: Array //这样可以指定传入的类型，如果类型不对，会警告

}

方式3：

props: {

childMsg: {

type: Array,

default: [0,0,0] //这样可以指定默认的值

}

}

这样呢，就实现了父组件向子组件传递数据.

### 2.子组件与父组件通信

那么，如果子组件想要改变数据呢？这在vue中是不允许的，因为vue只允许单向数据传递，这时候我们可以通过触发事件来通知父组件改变数据，从而达到改变子组件数据的目的.

子组件:

<template>

<div @click="up"></div>

</template>

methods: {

up() {

this.$emit('upup','hehe'); //主动触发upup方法，'hehe'为向父组件传递的数据

}

}

父组件:

<div>

<child @upup="change" :msg="msg"></child> //监听子组件触发的upup事件,然后调用change方法

</div>

methods: {

change(msg) {

this.msg = msg;

}

}

### 3.非父子组件通信

如果2个组件不是父子组件那么如何通信呢？这时可以通过eventHub来实现通信.   
所谓eventHub就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件.

let Hub = new Vue(); //创建事件中心

组件1触发：

<div @click="eve"></div>

methods: {

eve() {

Hub.$emit('change','hehe'); //Hub触发事件

}

}

组件2接收:

<div></div>

created() {

Hub.$on('change', () => { //Hub接收事件

this.msg = 'hehe';

});

}

这样就实现了非父子组件之间的通信了.原理就是把Hub当作一个中转站！

**14.JSX是什么？描述一下react主组件与分组件之间是如何传递数据的？**

JSX就是Javascript和XML结合的一种格式。React发明了JSX，利用HTML语法来创建虚拟DOM。当遇到<，JSX就当HTML解析，遇到{就当JavaScript解析。

(<http://blog.csdn.net/limm33/article/details/50942863>)

**15.你用过的前端框架都有哪些？挑出两个说说它们的优缺点？**

JQuery/jQuery Mobile

Dojo/[Dojo Mobile](http://dojotoolkit.org/features/mobile)

zepto.js是一个专为mobile WebKit浏览器(如：Safari和Chrome), 语法借鉴并兼容JQuery。

Node.js

angularJS

vueJS

bootstrap 是一套html css和js的框架。

React.js（React）是 Facebook 推出的一个用来构建用户界面的 JavaScript 库.

Slider 滑动条控件。（不算一个框架）

**16.了解nodejs吗？什么是错误优先的回调函数？nodejs中如何避免“回调地狱”？什么是事件循环？**

错误优先的回调函数用于传递错误和数据。第一个参数始终应该是一个错误对象， 用于检查程序是否发生了错误。其余的参数用于传递数据。例如：

fs.readFile(filePath, function(err, data) {

if (err) {

//handle the error

}

// use the data object

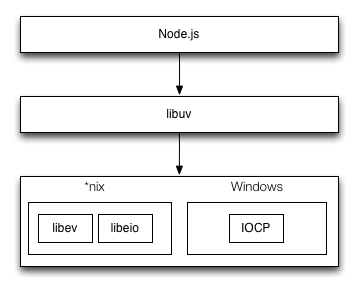
});

避免回调地狱

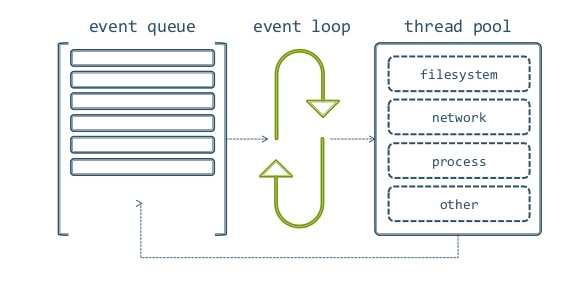
* 模块化：将回调函数分割为独立的函数
* 使用Promises
* 使用yield
* 来计算生成器或Promise

事件循环

Node采用的是单线程的处理机制（所有的I/O请求都采用非阻塞的工作方式），至少从Node.js开发者的角度是这样的。 而在底层，Node.js借助[libuv](https://github.com/libuv/libuv)来作为抽象封装层， 从而屏蔽不同操作系统的差异，Node可以借助livuv来来实现多线程。下图表示了Node和libuv的关系。



Libuv库负责Node API的执行。它将不同的任务分配给不同的线程，形成一个事件循环， 以异步的方式将任务的执行结果返回给V8引擎。可以简单用下面这张图来表示。



每一个I/O都需要一个回调函数——一旦执行完便推到事件循环上用于执行。

**17.你如何对网站的文件和资源进行优化？**

1、文件合并（目的是减少http请求）

2、文件压缩（目的是直接减少文件下载的体积）

3、使用cdn托管资源

4、使用缓存

5、gizp压缩你的[**js**](http://lib.csdn.net/base/javascript)和css文件

6、meta标签优化（title,description,keywords）、heading标签的优化、alt优化

7、反向链接，网站外链接优化