Js主要做了什么

1、客户端页面动态交互功能

2、前端数据的处理

3、前后端数据交互和数据的绑定

首先js基础：作用域、this、闭包、函数面试题，接下来就是提升客户端用户体验js事件的问题和处理客户端数据的实战题，前端的客户端数据处理好要和后端的屌丝们处理数据交互了，就会遇到ajax、和模板引擎的问题。再然后逼格提升，就是使用框架和实现页面组件化（react、angular、vue等）。

前端基础知识面试

javascript的本地对象，内置对象和宿主对象  
本地对象为array obj regexp等可以new实例化  
内置对象为Math 等不可以实例化的  
宿主为浏览器自带的document,window 等

javascript的同源策略  
一段脚本只能读取来自于同一来源的窗口和文档的属性，这里的同一来源指的是主机名、协议和端口号的组合

相关链接：http://www.jb51.net/article/66500.htm

作用域

var a=3 ;

function c(){

alert(a);

}

(function(){

var a=4 ;

c();

})(); //最后输出结果是3

因为：js中变量的作用域链与定义时的环境有关，与执行时无关。执行环境只会改变this、传递的参数、全局变量等

预解释

alert(n); // 弹出undefined  
var n = 10;

下一个预解释的面试题：

alert(n);  
n = 10; // 这里会报错，http://files.jb51.net/file_images/article/201501/201501281529461.png

下一个题

sum(); ————> 正常输出

function sum () {

console.log(‘11111’);

};

thanFn(); ——————> 报错

Var thatFn = function () {

Console.log(‘22222’)

}

解释一下闭包？

function foo(x) {

var tmp = 3;  
    return function (y) {  
        alert(x + y + (++tmp));  
    }  
}  
var bar = foo(2); // bar 现在是一个闭包  
bar(10);

闭包是一种设计原则，它通过分析上下文，来简化用户的调用，让用户在不知晓的情况下，达到他的目的；

或者说任何函数都可以看成是一个闭包，都会保护内部变量不被外界所访问。

闭包的缺点？

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。

解释一下js中的this指向？

this总是指向函数的直接调用对象（而非间接调用者）；

如果有new关键字，this指向new出来的那个对象；  
在事件中，this指向触发这个事件的对象，特殊的是，IE中的attachEvent中的this总是指向全局对象Window；

匿名函数（自执行函数）的this指向永远是window。

字符串

截取字符串abcdace的ace。——>'abcdace'.substring(4)

判断一个字符串中出现次数最多的字符，统计这个次数  
**var *str*** = **'asdfssaaasasasasaa'**;  
**var *json*** = {};  
**for** (**var *i*** = 0; ***i*** < ***str***.**length**; ***i***++) {  
 **if**(!***json***[***str***.charAt(***i***)]){  
 ***json***[***str***.charAt(***i***)] = 1;  
 }**else**{  
 ***json***[***str***.charAt(***i***)]++;  
 }  
};  
**var *iMax*** = 0;  
**var *iIndex*** = **''**;  
**for**(**var *i* in *json***){  
 **if**(***json***[***i***]>***iMax***){  
 ***iMax*** = ***json***[***i***];  
 ***item*** = ***i***;  
 }  
}  
console.log(**'出现次数最多的是:'**+***item***+**'出现'**+***iMax***+**'次'**);

数组方法

给Aarry添加一个删除方法，根据索引删除元素。

Array.**prototype**.remove = **function** (i) {  
 **var** arr = **this**;  
 arr.splice(i,1);  
 **return** arr;  
};

[caller和callee的区别](http://blog.csdn.net/laijieyao/article/details/43404953)

相关链接：<http://blog.csdn.net/laijieyao/article/details/43404953>

**caller**  
caller返回一个函数的引用，这个函数调用了当前的函数。  
使用这个属性要注意:  
1 这个属性只有当函数在执行时才有用  
2 如果在[JavaScript](http://lib.csdn.net/base/javascript)程序中，函数是由顶层调用的，则返回null

functionName.caller: functionName是当前正在执行的函数

**var** *a* = **function**() {  
 alert(*a*.**caller**);  
}  
**var** *b* = **function**() {  
 *a*();  
}  
*b*();

a();

上面的代码中，b调用了a，那么a.caller返回的是b的引用，如果直接调用a(即a在任何函数中被调用，也就是顶层调用),返回null:

**callee**  
callee放回正在执行的函数本身的引用，它是arguments的一个属性  
使用callee时要注意:  
1 这个属性只有在函数执行时才有效  
2 它有一个length属性，可以用来获得形参的个数，因此可以用来比较形参和实参个数是否一致，即比较arguments.length是否等于arguments.callee.length  
3 它可以用来递归匿名函数。

**var** *a* = **function**() {  
 alert(arguments.**callee**);  
}  
**var** *b* = **function**() {  
 *a*();  
}  
*b*();

虽然a在b中被调用，但是它返回了a本身的引用（也就是它函数本身）。

前端模块化相关面试

规避javascript多人开发的问题  
命名空间，封闭空间，js模块化mvc（数据层、表现层、控制层），单例模式对象化

了解一下模块化开发

相关链接：<http://www.ruanyifeng.com/blog/2012/10/javascript_module.html>

**CommonJS,AMD,RequireJS的区别**

RequireJS实现了AMD的API.

CommonJS是使用exports对象来定义模块的一种方法，它定义了模块的内容。简单地实现一个CommonJS的定义就像下面这样：

// someModule.js

exports.doSomething = function() { return "foo"; };

//otherModule.js

var someModule = require('someModule'); // in the vein of node

exports.doSomethingElse = function() { return someModule.doSomething() + "bar"; };

基本上CommonJS明确了你需要有一个require函数来获取依赖，exports变量来输出模块的内容和一些用来获取依赖的模块标识符。CommonJS有多种实现，比如Node.js.

因为CommonJS设计的时候没有考虑[浏览器](http://www.2cto.com/os/liulanqi/)，所以它不适合浏览器环境（我其实对这个不明确，但是这种说法到处都有，比如RequireJS官网）。所以我们得做一些工作来实现异步加载。

相反，RequireJS实现了AMD，它被设计用来适应浏览器环境。表面上看来，AMD开始是CommonJS输出格式的副产品，而且最终进化出了自己的API。在AMD中出现的新东西是define函数，它允许模块在加载依赖之前声明它的依赖。例如定义可能就像下面这样：

define('module/id/string', ['module', 'dependency', 'array'],

function(module, factory function) {

return ModuleContents;

});

因此CommonJS和AMD是[Java](http://www.2cto.com/kf/ware/Java/)script模块定义API的不同的实现，但是他们有相同的根源。AMD更适合浏览器，因为它支持异步加载模块依赖。RequireJS是AMD的一个实现，而且尽量保留了CommonJS的精神（主要是模块标识符上）。更让人混乱的是，RequireJS在实现AMD的同时，还提供了一个CommonJS包裹，这样CommonJS模块可以几乎直接被RequireJS引入。

define(function(require, exports, module) {

var someModule = require('someModule'); // in the vein of node

exports.doSomethingElse = function() { return someModule.doSomething() + "bar"; };

});

**JavaScript模块化编程 - CommonJS, AMD 和 RequireJS之间的关系**

先说说CommonJS

CommonJS - 大家是不是觉得[Java](http://www.2cto.com/kf/ware/Java/)Script仅仅是一个客户端的编译语言，其实[JavaScript](http://www.2cto.com/kf/qianduan/JS/)设计之初不仅仅是针对客户端设计的语言。后来只是由于Web的迅速流行，加之Netscape和微软之间之争过早的将JavaScipt标准化。要了解详细的JS历史请查看：http://zh.wikipedia.org/zh-cn/JavaScript。过早的标准化JS就导致JS的诸多缺陷和标准类库的缺乏，即使这样也不影响JS成为一门优秀的[编程](http://www.2cto.com/kf)语言（比如现在非常流行的Node.js）。目前JS仅仅包括基本的API，如果要作为一个server端的编程语言，像IO, FS, i18n, package等等特性都没有，CommonJS是一个组织，它让JS可以在共同的方向上做努力，来完善JS。尽量Common JS现在还没有一个正式版发布，但是很多方向的草案的实现都已经取得很好的成果。例如现在非常流行的Node.js。

再回到本文的主题来，CommonJS和AMD，RequireJS有什么关系呢？原来CommonJS其中就有一个Modules规范，我们都像JS现在这样所有东西都写在一个文件中来写server端应用是一件非常困难的事情，它就是来解决JS没有模块化管理代码的功能。关键部分就二个函数:

require - 用来引入依赖

export - 用来导出模块，包括标识符(identifier)和模块内容(contents)

CommonJS并没有只是一个规范，就像Java中的Interface一样，并没有注明你应该怎么实现。

问题在于CommonJS的这个Modules规范设计之初是为了server端设计的，它是一个同步的模式。但是这种模式并不适合于[浏览器](http://www.2cto.com/os/liulanqi/)端，大家设想一下如果浏览器同步模式一个一个加载模块，那么打开将会变得非常的慢，所以AMD就是为了这个诞生，它最大的特点就是可以异步的方式加载模块，具体的不同在于AMD有一个define函数，它可以让当前模块运行时先加载当前模块所依赖的模块，例如以下定义的意义就是在运行function时先加载依赖的module, dependency, array模块。

define('module/id/string', ['module', 'dependency', 'array'],

function(module, dependency, array) {

  return ModuleContents;

});

所以说CommonJS Module和AMD都是JS模块化定义的API，出自相同的起源的，就是可以让JS可以模块化加载。

那么RequrieJS其实就是AMD现在用的最广泛，最流行的实现。在RequireJS网站上的介绍其实也有说明RequireJS诞生的原因，只是当时我并没有看懂。

RequireJS is a JavaScript file and module loader. It is optimized for in-browser use, but it can be used in other JavaScript environments, like Rhino and Node. Using a modular script loader like RequireJS will improve the speed and quality of your code.

从以上基本可以看清CommonJS(泛指Modules规范)，AMD和RequireJS之前的关系了。简单的来讲CommonJS Modules和AMD都是为了解决JS模块化的规范API，CommonJS更适合于Server端，而AMD基本是用于浏览器端（不过它也可以用于Server端，比如Node loader的方向的努力：http://requirejs.org/docs/node.html），而RequireJS就是AMD最流行的实现

浏览器机制及优化相关面试

解释一下一个完整的http事物

**一.域名解析**

# 二.发起TCP的3次握手

# 三.建立TCP连接后发起http请求

# 四.服务器端响应http请求，浏览器得到html代码

# 五. 浏览器解析html代码，并请求html代码中的资源

# 六.浏览器对页面进行渲染呈现给用户

浏览器利用自己内部的工作机制，把请求到的静态资源和html代码进行渲染，渲染之后呈现给用户。

解释一下我们从输入一个网址到展现到浏览器是一个什么过程

相关链接：<http://blog.csdn.net/xiaozhuxmen/article/details/52014901>

1、用户访问网页，发送一个http请求到网络服务器。

2、网络服务器（应用服务器）解析请求，发送请求给数据库服务器。

3、数据服务器返回数据给网络服务器，网络服务器解析数据，并生成html文件内容放入http response中，返回给浏览器。

4、浏览器解析http response。

5、浏览器创建DOM树。

6、浏览器下载css生成CSS与DOM合并，构建渲染树（renderingtree），并应用在DOM树上，进行渲染。

7、浏览器下载js，并解析执行js。

重绘和重排何时会发生：

（1）增加或删除DOM节点；

（2）display:none（重排并重绘）；visibility:hidden（重排）；

（3）移动页面中的元素；

（4）增加或修改样式；

（5）用户改变窗口大小，滚动页面等。

如何减少重绘和重排以提升页面性能：

（1）不要一个个修改属性，应通过一个class来修改

（2）clone节点，在副本中修改，然后直接替换当前的节点；

（3）若要频繁获取计算后的样式，请暂存起来；

（4）降低受影响的节点：在页面顶部插入节点将影响后续所有节点。而绝对定位的元素改变会影响较少的元素；

（5）批量添加DOM：多个DOM插入或修改，应组成一个长的字符串后一次性放入DOM。使用innerHTML永远比DOM操作快。（特别注意：innerHTML不会执行字符串中的嵌入脚本，因此不会产生XSS漏洞）。

百度面试题，为什么我们要读写分离呢？

http://blog.csdn.net/lamp\_yang\_3533/article/details/52337061