**AngularJS面试題**

ng-if 跟 ng-show/hide 的区别有哪些？

第一点区别是，ng-if 在后面表达式为 true 的时候才创建这个 dom 节点，ng-show 是初始时就创建了，用 display:block 和 display:none 来控制显示和不显示。

第二点区别是，ng-if 会（隐式地）产生新作用域，ng-switch 、 ng-include 等会动态创建一块界面的也是如此。

这样会导致，在 ng-if 中用基本变量绑定 ng-model，并在外层 div 中把此 model 绑定给另一个显示区域，内层改变时，外层不会同步改变，因为此时已经是两个变量了。

<p>{{name}}</p>

<div ng-if="true">

<input type="text" ng-model="name">

</div>

ng-show 不存在此问题，因为它不自带一级作用域。

避免这类问题出现的办法是，始终将页面中的元素绑定到对象的属性（data.x）而不是直接绑定到基本变量（x）上。

详见 AngularJS 中的作用域

ng-repeat迭代数组的时候，如果数组中有相同值，会有什么问题，如何解决？

会提示 Duplicates in a repeater are not allowed. 加 track by $index 可解决。当然，也可以 trace by 任何一个普通的值，只要能唯一性标识数组中的每一项即可（建立 dom 和数据之间的关联）。

ng-click 中写的表达式，能使用 JS 原生对象上的方法吗？

不止是 ng-click 中的表达式，只要是在页面中，都不能直接调用原生的 JS 方法，因为这些并不存在于与页面对应的 Controller 的 $scope 中。

举个例子：

<p>{{parseInt(55.66)}}<p>

会发现，什么也没有显示。

但如果在 $scope 中添加了这个函数：

$scope.parseInt = function(x){

return parseInt(x);

}

这样自然是没什么问题了。

对于这种需求，使用一个 filter 或许是不错的选择：

<p>{{13.14 | parseIntFilter}}</p>

app.filter('parseIntFilter', function(){

return function(item){

return parseInt(item);

}

})

{{now | 'yyyy-MM-dd'}} 这种表达式里面，竖线和后面的参数通过什么方式可以自定义？

filter，格式化数据，接收一个输入，按某规则处理，返回处理结果。

内置 filter

ng 内置的 filter 有九种：

date（日期）

currency（货币）

limitTo（限制数组或字符串长度）

orderBy（排序）

lowercase（小写）

uppercase（大写）

number（格式化数字，加上千位分隔符，并接收参数限定小数点位数）

filter（处理一个数组，过滤出含有某个子串的元素）

json（格式化 json 对象）

filter 有两种使用方法，一种是直接在页面里：

<p>{{now | date : 'yyyy-MM-dd'}}</p>

另一种是在 js 里面用：

// $filter('过滤器名称')(需要过滤的对象, 参数1, 参数2,...)

$filter('date')(now, 'yyyy-MM-dd hh:mm:ss');

自定义 filter

// 形式

app.filter('过滤器名称',function(){

return function(需要过滤的对象,过滤器参数1,过滤器参数2,...){

//...做一些事情

return 处理后的对象;

}

});

// 例子

app.filter('timesFilter', function(){

return function(item, times){

var result = '';

for(var i = 0; i < times; i++){

result += item;

}

return result;

}

})

factory、service 和 provider 是什么关系？

factory

把 service 的方法和数据放在一个对象里，并返回这个对象

app.factory('FooService', function(){

return {

target: 'factory',

sayHello: function(){

return 'hello ' + this.target;

}

}

});

service

通过构造函数方式创建 service，返回一个实例化对象

app.service('FooService', function(){

var self = this;

this.target = 'service';

this.sayHello = function(){

return 'hello ' + self.target;

}

});

provider

创建一个可通过 config 配置的 service，$get 中返回的，就是用 factory 创建 service 的内容

app.provider('FooService', function(){

this.configData = 'init data';

this.setConfigData = function(data){

if(data){

this.configData = data;

}

}

this.$get = function(){

var self = this;

return {

target: 'provider',

sayHello: function(){

return self.configData + ' hello ' + this.target;

}

}

}

});

// 此处注入的是 FooService 的 provider

app.config(function(FooServiceProvider){

FooServiceProvider.setConfigData('config data');

});

从底层实现上来看，service 调用了 factory，返回其实例；factory 调用了 provider，返回其 $get 中定义的内容。factory 和 service 功能类似，只不过 factory 是普通 function，可以返回任何东西（return 的都可以被访问，所以那些私有变量怎么写，你懂的）；service 是构造器，可以不返回（绑定到 this 的都可以被访问）；provider 是加强版 factory，返回一个可配置的 factory。

详见 AngularJS 之 Factory vs Service vs Provider

angular 的数据绑定采用什么机制？详述原理

脏检查机制。

双向数据绑定是 AngularJS 的核心机制之一。当 view 中有任何数据变化时，会更新到 model ，当 model 中数据有变化时，view 也会同步更新，显然，这需要一个监控。

原理就是，Angular 在 scope 模型上设置了一个 监听队列，用来监听数据变化并更新 view 。每次绑定一个东西到 view 上时 AngularJS 就会往 $watch 队列里插入一条 $watch，用来检测它监视的 model 里是否有变化的东西。当浏览器接收到可以被 angular context 处理的事件时，$digest 循环就会触发，遍历所有的 $watch，最后更新 dom。

举个例子

<button ng-click="val=val+1">increase 1</button>

click 时会产生一次更新的操作（至少触发两次 $digest 循环）

按下按钮

浏览器接收到一个事件，进入到 angular context

$digest 循环开始执行，查询每个 $watch 是否变化

由于监视 $scope.val 的 $watch 报告了变化，因此强制再执行一次 $digest 循环

新的 $digest 循环未检测到变化

浏览器拿回控制器，更新 $scope.val 新值对应的 dom

$digest 循环的上限是 10 次（超过 10次后抛出一个异常，防止无限循环）。

详见 关于 AngularJS 的数据绑定

两个平级界面块 a 和 b，如果 a 中触发一个事件，有哪些方式能让 b 知道？详述原理

这个问题换一种说法就是，如何在平级界面模块间进行通信。有两种方法，一种是共用服务，一种是基于事件。

**共用服务**

在 Angular 中，通过 factory 可以生成一个单例对象，在需要通信的模块 a 和 b 中注入这个对象即可。

**基于事件**

这个又分两种方式

第一种是借助父 controller。在子 controller 中向父 controller 触发（$emit）一个事件，然后在父 controller 中监听（$on）事件，再广播（$broadcast）给子 controller ，这样通过事件携带的参数，实现了数据经过父 controller，在同级 controller 之间传播。

第二种是借助 $rootScope。每个 Angular 应用默认有一个根作用域 $rootScope， 根作用域位于最顶层，从它往下挂着各级作用域。所以，如果子控制器直接使用 $rootScope 广播和接收事件，那么就可实现同级之间的通信。

详见 AngularJS 中 Controller 之间的通信

一个 angular 应用应当如何良好地分层？

目录结构的划分

对于小型项目，可以按照文件类型组织，比如：

css

js

controllers

models

services

filters

templates

但是对于规模较大的项目，最好按业务模块划分，比如：

css

modules

account

controllers

models

services

filters

templates

disk

controllers

models

services

filters

templates

modules 下最好再有一个 common 目录来存放公共的东西。

逻辑代码的拆分

作为一个 MVVM 框架，Angular 应用本身就应该按照 模型，视图模型（控制器），视图来划分。

这里逻辑代码的拆分，主要是指尽量让 controller 这一层很薄。提取共用的逻辑到 service 中 （比如后台数据的请求，数据的共享和缓存，基于事件的模块间通信等），提取共用的界面操作到 directive 中（比如将日期选择、分页等封装成组件等），提取共用的格式化操作到 filter 中等等。

在复杂的应用中，也可以为实体建立对应的构造函数，比如硬盘（Disk）模块，可能有列表、新建、详情这样几个视图，并分别对应的有 controller，那么可以建一个 Disk 构造函数，里面完成数据的增删改查和验证操作，有跟 Disk 相关的 controller，就注入 Disk 构造器并生成一个实例，这个实例就具备了增删改查和验证方法。这样既层次分明，又实现了复用（让 controller 层更薄了）。

参考 AngularJS在苏宁云中心的深入实践

angular 应用常用哪些路由库，各自的区别是什么？

Angular1.x 中常用 ngRoute 和 ui.router，还有一种为 Angular2 设计的 new router（面向组件）。后面那个没在实际项目中用过，就不讲了。

无论是 ngRoute 还是 ui.router，作为框架额外的附加功能，都必须以 模块依赖 的形式被引入。

**区别**

ngRoute 模块是 Angular 自带的路由模块，而 ui.router 模块是基于 ngRoute模块开发的第三方模块。

ui.router 是基于 state （状态）的， ngRoute 是基于 url 的，ui.router模块具有更强大的功能，主要体现在视图的嵌套方面。

使用 ui.router 能够定义有明确父子关系的路由，并通过 ui-view 指令将子路由模版插入到父路由模板的 <div ui-view></div> 中去，从而实现视图嵌套。而在 ngRoute 中不能这样定义，如果同时在父子视图中 使用了 <div ng-view></div> 会陷入死循环。

示例

ngRoute

var app = angular.module('ngRouteApp', ['ngRoute']);

app.config(function($routeProvider){

$routeProvider

.when('/main', {

templateUrl: "main.html",

controller: 'MainCtrl'

})

.otherwise({ redirectTo: '/tabs' });

ui.router

var app = angular.module("uiRouteApp", ["ui.router"]);

app.config(function($urlRouterProvider, $stateProvider){

$urlRouterProvider.otherwise("/index");

$stateProvider

.state("Main", {

url: "/main",

templateUrl: "main.html",

controller: 'MainCtrl'

})

如果通过angular的directive规划一套全组件化体系，可能遇到哪些挑战？

没有自己用 directive 做过一全套组件，讲不出。

能想到的一点是，组件如何与外界进行数据的交互，以及如何通过简单的配置就能使用吧。

分属不同团队进行开发的 angular 应用，如果要做整合，可能会遇到哪些问题，如何解决？

可能会遇到不同模块之间的冲突。

比如一个团队所有的开发在 moduleA 下进行，另一团队开发的代码在 moduleB 下

angular.module('myApp.moduleA', [])

.factory('serviceA', function(){

...

})

angular.module('myApp.moduleB', [])

.factory('serviceA', function(){

...

})

angular.module('myApp', ['myApp.moduleA', 'myApp.moduleB'])

会导致两个 module 下面的 serviceA 发生了覆盖。

貌似在 Angular1.x 中并没有很好的解决办法，所以最好在前期进行统一规划，做好约定，严格按照约定开发，每个开发人员只写特定区块代码。

angular 的缺点有哪些？

强约束

导致学习成本较高，对前端不友好。

但遵守 AngularJS 的约定时，生产力会很高，对 Java 程序员友好。

不利于 SEO

因为所有内容都是动态获取并渲染生成的，搜索引擎没法爬取。

一种解决办法是，对于正常用户的访问，服务器响应 AngularJS 应用的内容；对于搜索引擎的访问，则响应专门针对 SEO 的HTML页面。

性能问题

作为 MVVM 框架，因为实现了数据的双向绑定，对于大数组、复杂对象会存在性能问题。

可以用来 优化 Angular 应用的性能 的办法：

减少监控项（比如对不会变化的数据采用单向绑定）

主动设置索引（指定 track by，简单类型默认用自身当索引，对象默认使用 $$hashKey，比如改为 track by item.id）

降低渲染数据量（比如分页，或者每次取一小部分数据，根据需要再取）

数据扁平化（比如对于树状结构，使用扁平化结构，构建一个 map 和树状数据，对树操作时，由于跟扁平数据同一引用，树状数据变更会同步到原始的扁平数据）

另外，对于Angular1.x ，存在 脏检查 和 模块机制 的问题。

移动端

可尝试 Ionic，但并不完善。

参考 如何看2015年1月Peter-Paul Koch对Angular的看法？

如何看待 angular 1.2 中引入的 controller as 语法？

最根本的好处

在 angular 1.2 以前，在 view 上的任何绑定都是直接绑定在 $scope 上的

function myCtrl($scope){

$scope.a = 'aaa';

$scope.foo = function(){

...

}

}

使用 controllerAs，不需要再注入 $scope，controller 变成了一个很简单的 javascript 对象（POJO），一个更纯粹的 ViewModel。

function myCtrl(){

// 使用 vm 捕获 this 可避免内部的函数在使用 this 时导致上下文改变

var vm = this;

vm.a = 'aaa';

}

原理

从源码实现上来看，controllerAs 语法只是把 controller 这个对象的实例用 as 别名在 $scope 上创建了一个属性。

if (directive.controllerAs) {

locals.$scope[directive.controllerAs] = controllerInstance;

}

但是这样做，除了上面提到的使 controller 更加 POJO 外，还可以避免遇到 AngularJS 作用域相关的一个坑（就是上文中 ng-if 产生一级作用域的坑，其实也是 javascript 原型链继承中值类型继承的坑。因为使用 controllerAs 的话 view 上所有字段都绑定在一个引用的属性上，比如 vm.xx，所以坑不再存在）。

<div ng-controller="TestCtrl as vm">

<p>{{name}}</p>

<div ng-if="vm.name">

<input type="text" ng-model="vm.name">

</div>

</div>

问题

使用 controllerAs 会遇到的一个问题是，因为没有注入 $scope，导致 $emit、 $broadcast、 $on、 $watch 等 $scope 下的方法无法使用。这些跟事件相关的操作可以封装起来统一处理，或者在单个 controller 中引入 $scope，特殊对待。

参考 angular controller as syntax vs scope

详述 angular 的 “依赖注入”

例子

依赖注入是一种软件设计模式，目的是处理代码之间的依赖关系，减少组件间的耦合。

举个栗子，如果没有使用 AngularJS，想从后台查询数据并在前端显示，可能需要这样做：

var animalBox = document.querySelector('.animal-box');

var httpRequest = {

get: function(url, callback){

console.log(url + ' requested');

var animals = ['cat', 'dog', 'rabbit'];

callback(animals);

}

}

var render = function(el, http){

http.get('/api/animals', function(animals){

el.innerHTML = animals;

})

}

render(httpRequest, animalBox);

但是，如果在调用 render 的时候不传参数，像下面这样，会报错，因为找不到 el 和 http（定义的时候依赖了，运行的时候不会自动查找依赖项）

render();

// TypeError: Cannot read property 'get' of undefined

而使用 AngularJS，可以直接这样

function myCtrl = ($scope, $http){

$http.get('/api/animals').success(function(data){

$scope.animals = data;

})

}

也就是说，在 Angular App 运行的时候，调用 myCtrl，自动做了 $scope 和 $http 两个依赖性的注入。

**原理**

AngularJS 是通过构造函数的参数名字来推断依赖服务名称的，通过 toString() 来找到这个定义的 function 对应的字符串，然后用正则解析出其中的参数（依赖项），再去依赖映射中取到对应的依赖，实例化之后传入。

简化一下，大概是这样：

var inject = {

// 存储依赖映射关系

storage: {},

// 注册依赖

register: function(name, resource){

this.storage[name] = resource;

},

// 解析出依赖并调用

resolve: function(target){

var self = this;

var FN\_ARGS = /^function\s\*[^\(]\*\(\s\*([^\)]\*)\)/m;

var STRIP\_COMMENTS = /((\/\/.\*$)|(\/\\*[\s\S]\*?\\*\/))/mg;

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

argDecl = fnText.match(FN\_ARGS)[1].split(/, ?/g);

var args = [];

argDecl.forEach(function(arg){

if(self.storage[arg]){

args.push(self.storage[arg]);

}

})

return function(){

target.apply({}, args);

}

}

}

使用这个 injector，前面那个不用 AngularJS 的栗子这样改造一下就可以调用了

inject.register('el', animalBox);

inject.register('ajax', httpRequest);

render = inject.resolve(render);

render();

问题

因为 AngularJS 的 injector 是假设函数的参数名就是依赖的名字，然后去查找依赖项，那如果按前面栗子中那样注入依赖，代码压缩后（参数被重命名了），就无法查找到依赖项了。

// 压缩前

function myCtrl = ($scope, $http){

...

}

// 压缩后

function myCtrl = (a, b){

...

}

所以，通常会使用下面两种方式注入依赖（对依赖添加的顺序有要求）。

数组注释法

myApp.controller('myCtrl', ['$scope', '$http', function($scope, $http){

...

}])

显式 $inject

myApp.controller('myCtrl', myCtrl);

function myCtrl = ($scope, $http){

...

}

myCtrl.$inject = ['$scope', '$http'];

补充

对于一个 DI 容器，必须具备三个要素：依赖项的注册，依赖关系的声明和对象的获取。

在 AngularJS 中，module 和 $provide 都可以提供依赖项的注册；内置的 injector 可以获取对象（自动完成依赖注入）；依赖关系的声明，就是前面问题中提到的那样。

下面是个例子

// 对于 module，传递参数不止一个，代表新建模块，空数组代表不依赖其他模块

// 只有一个参数（模块名），代表获取模块

// 定义 myApp，添加 myApp.services 为其依赖项

angular.module('myApp', ['myApp.services']);

// 定义一个 services module，将 services 都注册在这个 module 下面

angular.module('myApp.services', [])

// $provider 有 factory, service, provider, value, constant

// 定义一个 HttpService

angular.module('myApp.services').service('HttpService', ['$http', function($http){

...

}])

参考

[AngularJS] 自己实现一个简单的依赖注入

理解angular中的module和injector，即依赖注入

AngularJS中的依赖注入实际应用场景

如何看待angular2

相比 Angular1.x，Angular2的改动很大，几乎算是一个全新的框架。

基于 TypeScript（可以使用 TypeScript 进行开发），在大型项目团队协作时，强语言类型更有利。

组件化，提升开发和维护的效率。

还有 module 支持动态加载，new router，promise的原生支持等等。

迎合未来标准，吸纳其他框架的优点，值得期待，不过同时要学习的东西也更多了（ES next、TS、Rx等）。

**vue1.x面试题**

**1.Vuejs组件**

vuejs构建组件使用

Vue.component('componentName',{ /\*component\*/ })；

这里注意一点，组件要先注册再使用，也就是说：

Vue.component('mine',{

template:'#mineTpl',

props:['name','title','city','content']

});

var v=new Vue({

el:'#vueInstance',

data:{

name:'zhang',

title:'this is title',

city:'Beijing',

content:'these are some desc about Blog'

}

});

如果反过来会报错，因为反过来代表先使用了组件的，但是组件却没注册。

webpack报错后，使用webpack --display-error-details可以排错

**2.指令keep-alive**

在看demo的时候看到在vue-router写着keep-alive，keep-alive的含义：

如果把切换出去的组件保留在内存中，可以保留它的状态或避免重新渲染。为此可以添加一个keep-alive指令

<component :is='curremtView' keep-alive></component>

**3.如何让css只在当前组件中起作用**

在每一个vue组件中都可以定义各自的css，js，如果希望组件内写的css只对当前组件起作用，只需要在style中写入scoped，即：

<style scoped></style>

**4.vuejs循环插入图片**

在写循环的时候，写入如下代码：

<div class="bio-slide" v-for="item in items">

<img src="{{item.image}}">

</div>

此时在控制台会出现警告

[Vue Warn]: src="{{item.image}}": interpolation in "src" attribute will cause a 404 request. Use v-bind:src instead.这里意思是在“src”属性插值将导致404请求。使用v-bind：src代替。

所以替换成如下：

<div class="bio-slide" v-for="item in items">

<img v-bind:src="item.image">

</div>

这里需要主要，v-bind在写的时候不能再用{{}}，根据官方的说法：

<a v-bind:href="url"></a>

这里 href 是参数，它告诉 v-bind 指令将元素的 href 特性跟表达式 url 的值绑定。可能你已注意到可以用特性插值 href="{{url}}" 获得同样的结果：这样没错，并且实际上在内部特性插值会转为 v-bind 绑定。

**5.绑定value到Vue实例的一个动态属性上**

对于单选按钮，勾选框及选择框选项，v-model绑定的value通常是静态字符串（对于勾选框是逻辑值）：

<!-- `toggle` 为 true 或 false -->

<input type="checkbox" v-model="toggle">

但是有时候想绑定value到vue实例的一个动态属性上，这时可以用v-bind实现，并且这个属性的值可以不是字符串。例如绑定Checkbox的value到vue实例的一个动态属性：

<input

type="checkbox"

v-model="toggle"

v-bind:true-value="a"

v-bind:false-value="b">

<p>{{toggle}}</p>

这里绑定后，并不是说就可以点击后由true,false的切换变为a,b的切换，因为这里定义的动态a，b是scope上的a,b，并不能直接显示出来，此时

//当选中时

vm.toggle === vm.a

//当没选中时

vm.toggle === vm.b

所以此时需要在data中定义a,b，即：

new Vue({

el:'...',

data:{

a:'a',

b:'b'

}

});

**6.片段实例**

下面几种情况会让实例变成一个片断实例：

模板包含多个顶级元素。

模板只包含普通文本。

模板只包含其它组件（其它组件可能是一个片段实例）。

模板只包含一个元素指令，如<partial> 或 vue-router 的 <router-view>。

模板根节点有一个流程控制指令，如 v-if 或 v-for。

这些情况让实例有未知数量的顶级元素，它将把它的 DOM 内容当作片断。片断实例仍然会正确地渲染内容。不过，它没有一个根节点，它的$el 指向一个锚节点，即一个空的文本节点（在开发模式下是一个注释节点）。

但是更重要的是，组件元素上的非流程控制指令，非 prop 特性和过渡将被忽略，因为没有根元素供绑定：

<!-- 不可以，因为没有根元素 -->

<example v-show="ok" transition="fade"></example>

<!-- props 可以 -->

<example :prop="someData"></example>

<!-- 流程控制可以，但是不能有过渡 -->

<example v-if="ok"></example>

片段实例也有用处，但是通常情况下组件有一个根节点比较好，它会保证组件元素上的指令和特性能正确的转换，同时性能也稍微好些。

**7.路由嵌套**

路由嵌套会将其他组件渲染到该组件内，而不是进行整个页面跳转router-view本身就是将组件渲染到该位置，想要进行页面跳转，就要将页面渲染到根组件，在起始配置路由时候写到：

var App = Vue.extend({ root });

router.start(App,'#app');

这里首先将根组件注册进来，用于将路由中配置好的各个页面渲染出来，然后将根组件挂载到与#app匹配的元素上。

**8.实现多个根据不同条件显示不同文字的方法**

v-if,v-else可以实现条件选择，但是如果是多个连续的条件选择，则需要用到计算属性computed。例如实现当输入框中什么都没写的时候显示字符串‘empty’,否则显示输入框中的内容，代码如下：

<div id="test">

<input type="text" v-model="inputValue">

<h1>{{changeVaule}}</h1>

</div>

new Vue({

el:'#test',

data:{

changeVaule:'123'

},

computed :{

changeVaule:function(){

if(this.inputValue!==''){

return this.inputValue;

}else{

return 'empty';

}

}

}

});

**9.Vuejs在变化检测问题**

1.检测数组

由于javascript的限制，vuejs不能检测到下面数组的变化：

直接索引设置元素，如vm.item[0]={};

修改数据的长度，如vm.item.length。

为了解决问题1，Vuejs扩展了观察数组，为它添加一个$set()方法：

// 与 `example1.items[0] = ...` 相同，但是能触发视图更新

example1.items.$set(0, { childMsg: 'Changed!'})

问题2，需要一个空数组替换items。

除了$set()，vuejs也为观察数组添加了$remove()方法，用于从目标数组中查找并删除元素，在内部调用了splice()。因此，不必：

var index = this.items.indexOf(item)

if (index !== -1) {

this.items.splice(index, 1)

}

只需：

this.items.$remove(item);

2.检测对象

受ES5的显示，Vuejs不能检测到对象属性的添加或删除。因为Vuejs在初始化时候将属性转化为getter/setter，所以属性必须在data对象才能让Vuejs转换它，才能让它是响应的，例如：

var data = { a: 1 }

var vm = new Vue({

data: data

})

// `vm.a` 和 `data.a` 现在是响应的

vm.b = 2

// `vm.b` 不是响应的

data.b = 2

// `data.b` 不是响应的

不过，有办法在实例创建之后添加属性并且让它是响应的。对于Vue实例，可以使用$set(key,value)实例方法：

vm.$set('b', 2)

// `vm.b` 和 `data.b` 现在是响应的

对于普通数据对象，可以使用全局方法Vue.set(object, key, value):

Vue.set(data, 'c', 3)

// `vm.c` 和 `data.c` 现在是响应的

有时你想向已有对象上添加一些属性，例如使用 Object.assign() 或 \_.extend() 添加属性。但是，添加到对象上的新属性不会触发更新。这时可以创建一个新的对象，包含原对象的属性和新的属性：

// 不使用 `Object.assign(this.someObject, { a: 1, b: 2 })`

this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })

**10.关于vuejs页面闪烁{{message}**}

在vuejs指令中有v-cloak，这个指令保持在元素上直到关联实例结束编译。和CSS规则如[v-cloak]{display:none}一起用时，这个指令可以隐藏未编译的Mustache标签直到实例准备完毕。用法如下：

[v-cloak]{

display:none;

}

<div v-cloak>{{message}}</div>

这样<div>不会显示，直到编译结束

**11.关于在v-for循环时候v-model的使用**

有时候需要循环生成input，用v-model绑定后，利用vuejs操作它，此时我们可以在v-model中写一个数组selected[$index]，这样就可以给不同的input绑定不同的v-model，从而分别操作他们。这个我在demo中的dataBind.vue中用到。

**12.vuejs中过渡动画**

在vuejs中，css定义动画：

.zoom-transition{

width:60%;

height:auto;

position: absolute;

left:50%;

top:50%;

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

-webkit-transition: all .3s ease;

transition: all .3s ease;

}

.zoom-enter, .zoom-leave{

width:150px;

height:auto;

position: absolute;

left:20px;

top:20px;

transform: translate(0,0);

}

其中动画在定的时候要注意上下对应，上面有什么，下面有什么，都要变化的，如果有不变化的，应该抽离出去，作为公共css样式，在上面的css中，如果我只写 transform: translate(-50%,-50%);而不写下面的transform: translate(0,0);则会导致上面的transform: translate(-50%,-50%);被添加到下面，认为这个是不变的。

**13.指令v-el的使用**

有时候我们想就像使用jquery那样去访问一个元素，此时就可以使用v-el指令，去给这个元素注册一个索引，方便通过所属实例的$el访问这个元素。

注意

HTML不区分大小写，所以v-el:someEl将转换为全小写。可以用v-el:some-el然后设置this.$el.someEl。

示例

<span v-el:msg>hello</span>

<span v-el:other-msg>world</span>

this.$els.msg.textContent // -> "hello"

this.$els.otherMsg.textContent // -> "world"

this.$els.msg //-><span>hello</span>

**14.关于vuejs中使用事件**名

在vuejs中，我们经常要绑定一些事件，有时候给DOM元素绑定，有时候给组件绑定。绑定事件在HTML中用v-on:click-"event",这时evet的名字不要出现大写，因为在1.x中不区分大小写，所以如果我们在HTML写v-on:click="myEvent"而在js中写myEvent就出错误，所以在vuejs的1.x绑定事件时候，要尽量避免使用大写字母。在2.0中没有该限制！

**15.v-if与v-show的区别**

v-if直接不渲染这个DOM元素，而v-show是会渲染DOM元素，只是使用display:none隐藏，打开开发者工具可以看到该DOM

**16.关于transition全局钩子如何在组件中使用**

Vue.transition是定义一个全局transition钩子的，如果想针对组件定义，则需要如下写法：

export default{

transition:{

'fade':{

enter() {},

leave() {}

}

}

}

这样fade这个过度钩子只会作用于组件内，如果同时有同名的全局钩子，则会优先使用组建定义的

**17.利用vue-router如何实现组件在渲染出来前执行某个事件**

export default{

data(){

return{

selected:0,

currentView:'view\_0'

}

},

methods:{

choose(index) {

this.selected=index;

this.currentView='view\_'+index;

}

},

route:{

data() {

/\*每次切换路由，在渲染出页面前都会执行\*/

}

}

}

**VueJs2.x面试题**

**1.独立构建vs运行时构建**

在按照vue1.0的配置配置好webpack后，会出现Failed to mount component: template or render function not defined. (found in root instance) 的错误，这里涉及到vue2.0与vue1.0的第一个不同的地方。具体区别独立构建 vs 运行时构建。解决方法为在webpack配置文件中添加如下配置项：

resolve: {

alias: {

'vue$': 'vue/dist/vue.common.js'

}

}

**2.挂载点的选择**

在原来的vue1.0的项目中我使用body元素作为挂载点，但是在vue2.0中，如果使用body或者html作为挂载点，则会报以下警告：Do not mount Vue to <html> or <body> - mount to normal elements instead.

<font color='red'>在vue1.0中允许开发者以body或者html作为根实体的挂载点，但是到了2.0后，只能通过独立的节点挂载，例如：div等，否则报警告</font>

**3.动态组件渲染（跟1.x类似）**

多个组件可以使用同一个挂载点，然后动态地在它们之间切换。使用保留的 <component> 元素，动态地绑定到它的 is 特性：

<!-- 动态组件由 vm 实例的属性值 `componentId` 控制 -->

<component :is="componentId"></component>

<!-- 也能够渲染注册过的组件或 prop 传入的组件 -->

<component :is="$options.components.child"></component>

keep-alive

如果把切换出去的组件保留在内存中，可以保留它的状态或避免重新渲染。为此可以添加一个 keep-alive 指令参数：

<keep-alive>

<component :is="currentView">

<!-- 非活动组件将被缓存！ -->

</component>

</keep-alive>

**4.ref**

有时候需要直接在父组件中访问子组件实例，或者直接操作DOM元素，此时需要使用ref。

ref被用来给元素或子元素注册引用信息。引用信息会根据父组件的$refs对象进行注册。如果在普通的DOM元素上使用，引用信息就是元素，如果用在子组件上，引用信息就是组件实例。

<!-- vm.$refs.p will be the DOM node -->

<p ref="p">hello</p>

<!-- vm.$refs.child will be the child comp instance -->

<child-comp ref="child"></child-comp>

当 v-for 用于元素或组件的时候，引用信息将是包含DOM节点或组件实例数组。

关于ref注册时间的重要说明: 因为ref本身是作为渲染结果被创建的，在初始渲染的时候你不能访问它们 - 它们还不存在！$refs 也不是响应式的，因此你不应该试图用它在模版中做数据绑定。

**5.自定义事件**

在vue自定义事件使用$on与$emit，前者用于触发监听，后者用于触发，监听可以有两种方式

<!--使用v-on在html中监听-->

<my-component v-on:test="callbackFun"></my-component>

<script>

//直接用$on监听

vm.$on('text',function(){})

</script>

**6.组件命名的约定**

当注册组件（或者 props）时，可以使用 kebab-case ，camelCase ，或 TitleCase

// 在组件定义中

components: {

// 使用 kebab-case 形式注册

'kebab-cased-component': { /\* ... \*/ },

// register using camelCase

'camelCasedComponent': { /\* ... \*/ },

// register using TitleCase

'TitleCasedComponent': { /\* ... \*/ }

}

在 HTML 模版中，只能使用 kebab-case 形式：

<!-- 在HTML模版中始终使用 kebab-case -->

<kebab-cased-component></kebab-cased-component>

<camel-cased-component></camel-cased-component>

<title-cased-component></title-cased-component>

当使用字符串模式时可以使用 camelCase 、 TitleCase 或者 kebab-case 来引用：

<!-- 在字符串模版中可以用任何你喜欢的方式! -->

<my-component></my-component>

<myComponent></myComponent>

<MyComponent></MyComponent>

**7.子组件中使用this**

有时候子组件简单，可以在父组件中直接注册，此时在子组件内使用this就是子组件实例并不是父组件，例如：

export default{

data(){

return{

parentMsg:'hello!'

}

},

components:{

child:{

props:['inputMessage'],

template:'<span>{{inputMessage}}</span>'

},

'child-secound':{

props:['inputMessage'],

template:'<span>{{upperCase}}</span>',

computed:{

upperCase(){

return this.inputMessage.toUpperCase();

}

}

}

}

}

**8.key的使用**

一般情况下，vue在渲染完成后，如果数据发生变化，只会重新渲染数据，不会重新渲染整个元素，但是有时候我们需要元素被重新渲染，此时就需要使用key关键字，使用v-bind绑定key关键字，可以实现在数据发生变化时候重新渲染整个元素。注：同一父级元素下所有子元素如果都要在数据变化后重新渲染元素，则需要被绑定的key

**9.v-move的使用**

在使用<transition-group>时候，不仅可以定义进入离开动画，还可以使用新增的v-move特性，与过渡一样，默认为v-move，可以用name进行自定义前缀，也可以用move-class属性手动设定。用了这个之后就可以实现移动过程中的动画。

**10.跳过css检测**

对于只使用js过度的元素使用v-bind:css="false"跳过vue对css的检测。

render函数的使用

createElement接受三个参数：

{String | Object | Function}即一个HTML标签 | 组件选项 | 一个函数，必须返回上述其中一个

{Object}一个对应HTML标签属性的数据对象（可选）

{String | Array}子节点（VNode）（可选）

⚠️ 关于第三个参数的说明

createElement第三个参数，如果是String，则类似于innerHTML，如果是Array，则可以写入一个执行函数，这个函数用于创建另一个DOM结构（而且这里如果想写入一个执行函数，必须是数组！！）

每个createElement只能创建一个元素，所以如果是创建多个元素相互嵌套，需要多个createElement函数相互嵌套，最后再render，这个跟原生js创建DOM元素类似

如果需要同时渲染多个元素，则需要在第三个参数的数组中，分别写入需要渲染的元素，此时Vue会按照数组中顺序进行渲染

完整数据对象：

{

// 和`v-bind:class`一样的 API

'class': {

foo: true,

bar: false

},

// 和`v-bind:style`一样的 API

style: {

color: 'red',

fontSize: '14px'

},

// 正常的 HTML 特性

attrs: {

id: 'foo'

},

// 组件 props

props: {

myProp: 'bar'

},

// DOM 属性

domProps: {

innerHTML: 'baz'

},

// 事件监听器基于 "on"

// 所以不再支持如 v-on:keyup.enter 修饰器

// 需要手动匹配 keyCode。

on: {

click: this.clickHandler

},

// 仅对于组件，用于监听原生事件，而不是组件使用 vm.$emit 触发的事件。

nativeOn: {

click: this.nativeClickHandler

},

// 自定义指令. 注意事项：不能对绑定的旧值设值

// Vue 会为您持续追踨

directives: [

{

name: 'my-custom-directive',

value: '2'

expression: '1 + 1',

arg: 'foo',

modifiers: {

bar: true

}

}

],

// Scoped slots in the form of

// { name: props => VNode | Array<VNode> }

scopedSlots: {

default: props => h('span', props.text)

},

// 如果子组件有定义 slot 的名称

slot: 'name-of-slot'

// 其他特殊顶层属性

key: 'myKey',

ref: 'myRef'

}

**vue-router2.x**

**1.router-view**

在vue-router2中<router-view>是最顶层的出口，渲染最高级路由匹配到组件。同样地，一个被渲染组件同样可以包含自己的嵌套<router-view>。

**2.挂载**

在router1.0中，挂载节点的方式为router.start()而在router2.0中使用vue自己的$mount手动挂载

**3.给link添加事件**

在vue-router1中使用v-link写入路由，但是在vue-router2中要使用router-link写入路由，在浏览器渲染的时候会把router-link渲染成a。

有时候需要为router-link注册事件，对于一般的html元素，直接使用@click="eventFun"即可，但是对于router-link，像普通html元素那样注册事件后并不管用，需要添加.native才会成功注册。

事实上给组件绑定原生事件就需要.native修饰v-on，否则无法注册成功。

<my-component v-on:click.native="doTheThing"></my-component>

**4.利用vue-router做导航**

在利用vue-router做导航的时候，需要用到redirect关键字的重定向功能，具体写法如下：

const router = new VueRouter({

routes : [

{path:'/',redirect:'/ZY'},

{path:'/ZY',component:ZY}

]

});

**5.路由嵌套**

vue-router的路由嵌套指的是子组件会在父组件中渲染出来，必须是子组件的父组件，祖先不可以实现，例如：

/user/foo/profile /user/foo/posts

+------------------+ +-----------------+

| User | | User |

| +--------------+ | | +-------------+ |

| | Profile | | +------------> | | Posts | |

| | | | | | | |

| +--------------+ | | +-------------+ |

+------------------+ +-----------------+

**WebPack2.x版本**

webpack1.x升级2.x

**1.module.loaders改成了module.rules**

旧的loaders被新的rules取代，后者允许配置loader以及其他更多项。

module: {

- loaders: [

+ rules: [

{

test: /\.css$/,

- loaders: [

+ use: [

{

loader: "style-loader"

},

{

loader: "css-loader",

- query: {

+ options: {

modules: true

}

]

},

{

test: /\.jsx$/,

loader: "babel-loader", // Do not use "use" here

options: {

// ...

}

}

]

}

以上写法中，Rule.loader是Rule.use: [ { loader } ]的简写。

**2.链式调用loaders**

在webpack1.x中loaders可以链式调用，在2.x中依旧有该特性，使用rule.use配置项，use中设置一个loaders的数组而在1.x中使用!连接各个loader，旧版写法只有在使用旧的module.loaders时有效。

module: {

- loaders: {

+ rules: {

test: /\.less$/,

- loader: "style-loader!css-loader!less-loader"

+ use: [

+ "style-loader",

+ "css-loader",

+ "less-loader"

+ ]

}

}

**3.取消在模块中自动添加-loader后缀**

module: {

rules: [

{

use: [

- "style",

+ "style-loader",

- "css",

+ "css-loader",

- "less",

+ "less-loader",

]

}

]

}

根据官方说法，做出这样更改的原因是省略-loader会对新手造成误解，所以去掉这个功能，如果想打开这个旧的功能，可以配置resolveLoader.moduleExtensions，但是并不推荐这么做，具体参阅issues#2986

+ resolveLoader: {

+ moduleExtensions: ["-loader"]

+ }

**4.使用options配置loader**

在webpack1.x中可以通过webpack.config.js的自定义属性来配置loader，这在webpack2.x中无法执行

module.exports = {

...

module: {

use: [{

test: /\.tsx?$/,

loader: 'ts-loader'

}]

},

// does not work with webpack 2

ts: { transpileOnly: false }

}

//webpack2 use options

module.exports = {

...

module: {

use: [{

test: /\.tsx?$/,

loader: 'ts-loader'

options: { transpileOnly: false }

}]

}

}

webpack插件的使用

webpack-dev-server1.x升级2.x

1.在CLI使用的时候，--inline默认开启，无需在输入命令时添加

**2.删除contentBase用proxy代替**

**3.减少控制台无用输出，在1.x中，当我们停掉服务器后，控制台会一直输出错误信息**，但是在2.x中只会输出[WDS] Disconnected!

extract-text-webpack-plugin

在使用webpack将vue\_spa打包后，并不会出现css，因为css被打包入build.js，如果从vue组件中抽离出css，**需要安装插件extract-text-webpack-plugin，在使用的时候需要配合v2版本的才可以使用**（如果使用了webpack2，则对应的插件版本都需要用v2版本）。具体配置如下：

module:{

rules:[

{

test: /\.vue$/,

loader: 'vue',

options: {

loaders:{

css: extractTextPlugin.extract({

loader: 'css-loader',

fallbackLoader: 'vue-style-loader'

})

}

}

}

]

},

plugins: [

new webpack.HotModuleReplacementPlugin(),

new extractTextPlugin({

filename:'/style.css',

allChunks:true

})

],

在options中：

options.loader: string | object | loader[] (必填项) 这里的 loader(s) 用于将资源转换为css导出模块

options.fallbackLoader: string | object | loader[] 当css没有被导出的时候这里的 loader(s) 会被使用 （即当在plugins模块中设置allChunks:false的时候）

在plugins中：

filename：可以设置被导出的css文件的路径以及名字

allChunks：从所有附加块中提取（默认情况下，它只从初始块中提取）

html-webpack-plugin

plugins: [

new htmlWwebpackPlugin({

filename: 'assets/admin.html'

}),

]

title：用于生成文档的document

filename：要注入的html文件，默认为index.html。可以自定义（例如：assets/admin.html）

inject：

true | ‘head’ | ‘body’ | false

将资源注入所给的template或templateContent，当设置为 true 或者 ‘body’，所有的资源会被注入到body底部。而head则会将js放到headelement

favicon: 在输出的html中添加favicon

hash: true | false 如果为true，则将一个唯一的webpack编译散列附加到所有包含的脚本和CSS文件。这对缓存清除很有用。

cache: true | false 如果为true（默认），尝试仅在更改后才发出文件。

是否要更换preset？

webpack2.x默认支持es6的模块，所以在编译时候没有必要将它们先转换为CommonJS模块再处理，所以在github中出现了babel-preset-es2015-webpack，但是这个模块我在使用的时候出现了Cannot remove 'babel-plugin-transform-es2015-modules-commonjs' from the plugin list.的问题，根据babel-preset-es2015-webpack中的叙述以及issues#14可知，babel-preset-es2015已经支持不转换模块中的import和export，只需要设置.babelrc

{

"presets": [

[

"es2015",

{

"modules": false

}

]

]

}

**React面试题**

# 调用 setState 之后发生了什么？

在代码中调用setState函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个UI界面。在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

# React 中 Element 与 Component 的区别是？

简单而言，React Element 是描述屏幕上所见内容的数据结构，是对于 UI 的对象表述。典型的 React Element 就是利用 JSX 构建的声明式代码片然后被转化为createElement的调用组合。而 React Component 则是可以接收参数输入并且返回某个 React Element 的函数或者类。更多介绍可以参考[React Elements vs React Components](https://link.zhihu.com/?target=https://tylermcginnis.com/react-elements-vs-react-components/" \t "_blank)。

# 在什么情况下你会优先选择使用 Class Component 而不是 Functional Component？

在组件需要包含内部状态或者使用到生命周期函数的时候使用 Class Component ，否则使用函数式组件。

# React 中 refs 的作用是什么？

注意，根据[React最新文档](https://link.zhihu.com/?target=https://facebook.github.io/react/docs/refs-and-the-dom.html" \t "_blank)，下面这种用法已经被弃用了，统一改为回调函数模式

this.refs.textInput

Refs 是 React 提供给我们的安全访问 DOM 元素或者某个组件实例的句柄。我们可以为元素添加ref属性然后在回调函数中接受该元素在 DOM 树中的句柄，该值会作为回调函数的第一个参数返回：

class CustomForm extends Component {

handleSubmit = () => {

console.log("Input Value: ", this.input.value)

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={(input) => this.input = input} />

<button type='submit'>Submit</button>

</form>

)

}

}

上述代码中的input域包含了一个ref属性，该属性声明的回调函数会接收input对应的 DOM 元素，我们将其绑定到this指针以便在其他的类函数中使用。另外值得一提的是，refs 并不是类组件的专属，函数式组件同样能够利用闭包暂存其值：

function CustomForm ({handleSubmit}) {

let inputElement

return (

<form onSubmit={() => handleSubmit(inputElement.value)}>

<input

type='text'

ref={(input) => inputElement = input} />

<button type='submit'>Submit</button>

</form>

)

}

# React 中 keys 的作用是什么？

Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识。

render () {

return (

<ul>

{this.state.todoItems.map(({task, uid}) => {

return <li key={uid}>{task}</li>

})}

</ul>

)

}

在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染。此外，React 还需要借助 Key 值来判断元素与本地状态的关联关系，因此我们绝不可忽视转换函数中 Key 的重要性。

# 如果你创建了类似于下面的Twitter元素，那么它相关的类定义是啥样子的？

<Twitter username='tylermcginnis33'>

{(user) => user === null

? <Loading />

: <Badge info={user} />}

</Twitter>

import React, { Component, PropTypes } from 'react'

import fetchUser from 'twitter'

// fetchUser take in a username returns a promise

// which will resolve with that username's data.

class Twitter extends Component {

// finish this

}

如果你还不熟悉回调渲染模式（Render Callback Pattern），这个代码可能看起来有点怪。这种模式中，组件会接收某个函数作为其子组件，然后在渲染函数中以props.children进行调用：

import React, { Component, PropTypes } from 'react'

import fetchUser from 'twitter'

class Twitter extends Component {

state = {

user: null,

}

static propTypes = {

username: PropTypes.string.isRequired,

}

componentDidMount () {

fetchUser(this.props.username)

.then((user) => this.setState({user}))

}

render () {

return this.props.children(this.state.user)

}

}

这种模式的优势在于将父组件与子组件解耦和，父组件可以直接访问子组件的内部状态而不需要再通过Props传递，这样父组件能够更为方便地控制子组件展示的UI界面。譬如产品经理让我们将原本展示的Badge替换为Profile，我们可以轻易地修改下回调函数即可：

<Twitter username='tylermcginnis33'>

{(user) => user === null

? <Loading />

: <Profile info={user} />}

</Twitter>

# Controlled Component 与 Uncontrolled Component 之间的区别是什么？

React 的核心组成之一就是能够维持内部状态的自治组件，不过当我们引入原生的HTML表单元素时（input,select,textarea 等），我们是否应该将所有的数据托管到 React 组件中还是将其仍然保留在 DOM 元素中呢？这个问题的答案就是受控组件与非受控组件的定义分割。受控组件（Controlled Component）代指那些交由 React 控制并且所有的表单数据统一存放的组件。譬如下面这段代码中username变量值并没有存放到DOM元素中，而是存放在组件状态数据中。任何时候我们需要改变username变量值时，我们应当调用setState函数进行修改。

class ControlledForm extends Component {

state = {

username: ''

}

updateUsername = (e) => {

this.setState({

username: e.target.value,

})

}

handleSubmit = () => {}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

value={this.state.username}

onChange={this.updateUsername} />

<button type='submit'>Submit</button>

</form>

)

}

}

而非受控组件（Uncontrolled Component）则是由DOM存放表单数据，并非存放在 React 组件中。我们可以使用 refs 来操控DOM元素：

class UnControlledForm extends Component {

handleSubmit = () => {

console.log("Input Value: ", this.input.value)

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={(input) => this.input = input} />

<button type='submit'>Submit</button>

</form>

)

}

}

竟然非受控组件看上去更好实现，我们可以直接从 DOM 中抓取数据，而不需要添加额外的代码。不过实际开发中我们并不提倡使用非受控组件，因为实际情况下我们需要更多的考虑表单验证、选择性的开启或者关闭按钮点击、强制输入格式等功能支持，而此时我们将数据托管到 React 中有助于我们更好地以声明式的方式完成这些功能。引入 React 或者其他 MVVM 框架最初的原因就是为了将我们从繁重的直接操作 DOM 中解放出来。

# 在生命周期中的哪一步你应该发起 AJAX 请求？

我们应当将AJAX 请求放到 componentDidMount 函数中执行，主要原因有下：

React 下一代调和算法 Fiber 会通过开始或停止渲染的方式优化应用性能，其会影响到 componentWillMount 的触发次数。对于 componentWillMount 这个生命周期函数的调用次数会变得不确定，React 可能会多次频繁调用 componentWillMount。如果我们将 AJAX 请求放到 componentWillMount 函数中，那么显而易见其会被触发多次，自然也就不是好的选择。

如果我们将 AJAX 请求放置在生命周期的其他函数中，我们并不能保证请求仅在组件挂载完毕后才会要求响应。如果我们的数据请求在组件挂载之前就完成，并且调用了setState函数将数据添加到组件状态中，对于未挂载的组件则会报错。而在 componentDidMount 函数中进行 AJAX 请求则能有效避免这个问题。

# shouldComponentUpdate 的作用是啥以及为何它这么重要？

shouldComponentUpdate 允许我们手动地判断是否要进行组件更新，根据组件的应用场景设置函数的合理返回值能够帮我们避免不必要的更新。

# 如何告诉 React 它应该编译生产环境版本？

通常情况下我们会使用 Webpack 的 DefinePlugin 方法来将 NODE\_ENV 变量值设置为 production。编译版本中 React 会忽略 propType 验证以及其他的告警信息，同时还会降低代码库的大小，React 使用了 Uglify 插件来移除生产环境下不必要的注释等信息。

# 为什么我们需要使用 React 提供的 Children API 而不是 JavaScript 的 map？

props.children并不一定是数组类型，譬如下面这个元素：

<Parent>

<h1>Welcome.</h1>

</Parent>

如果我们使用props.children.map函数来遍历时会受到异常提示，因为在这种情况下props.children是对象（object）而不是数组（array）。React 当且仅当超过一个子元素的情况下会将props.children设置为数组，就像下面这个代码片：

<Parent>

<h1>Welcome.</h1>

<h2>props.children will now be an array</h2>

</Parent>

这也就是我们优先选择使用React.Children.map函数的原因，其已经将props.children不同类型的情况考虑在内了。

# 概述下 React 中的事件处理逻辑

为了解决跨浏览器兼容性问题，React 会将浏览器原生事件（Browser Native Event）封装为合成事件（SyntheticEvent）传入设置的事件处理器中。这里的合成事件提供了与原生事件相同的接口，不过它们屏蔽了底层浏览器的细节差异，保证了行为的一致性。另外有意思的是，React 并没有直接将事件附着到子元素上，而是以单一事件监听器的方式将所有的事件发送到顶层进行处理。这样 React 在更新 DOM 的时候就不需要考虑如何去处理附着在 DOM 上的事件监听器，最终达到优化性能的目的。

# createElement 与 cloneElement 的区别是什么？

createElement 函数是 JSX 编译之后使用的创建 React Element 的函数，而 cloneElement 则是用于复制某个元素并传入新的 Props。

# 传入 setState 函数的第二个参数的作用是什么？

该函数会在setState函数调用完成并且组件开始重渲染的时候被调用，我们可以用该函数来监听渲染是否完成：

this.setState(

{ username: 'tylermcginnis33' },

() => console.log('setState has finished and the component has re-rendered.')

)

# 下述代码有错吗？

this.setState((prevState, props) => {

return {

streak: prevState.streak + props.count

}

})

这段代码没啥问题，不过只是不太常用罢了

**NodeJs（含JS）相关面试题**

**1. ES6有哪些新特性？**

参考答案：类的支持，模块化，箭头操作符，let/const块作用域，字符串模板，解构，参数默认值/不定参数/拓展参数,for-of遍历,generato r器, Map/Set, Promise

**2. 你对ES6的个人看法？**

参考答案：ES6必火！从软件工程角度来看，以前真的很弱，不适合做大型应用，很容易导致烂尾工程。ES6就相当于当年的Java5,是历史性的发展，从此我们可以用js做大型项目了。事实上，各大主流浏览器现在已经支持大部分新特性了，后端的Node.js更是可以直接使用ES6的绝大多数语法。

推荐ES6入门好书: 阮一峰 ECMAScript 6入门

javascript高级话题(面向对象，作用域，闭包，设计模式等)

**1. 常用js类定义的方法有哪些？**

参考答案：主要有构造函数原型和对象创建两种方法。原型法是通用老方法，对象创建是ES5推荐使用的方法.目前来看，原型法更普遍.

代码演示

1) 构造函数方法定义类

function Person(){

this.name = 'michaelqin';

}

Person.prototype.sayName = function(){

alert(this.name);

}

var person = new Person();

person.sayName();

2) 对象创建方法定义类

var Person = {

name: 'michaelqin',

sayName: function(){ alert(this.name); }

};

var person = Object.create(Person);

person.sayName();

**2. js类继承的方法有哪些**

参考答案：原型链法，属性复制法和构造器应用法. 另外，由于每个对象可以是一个类，这些方法也可以用于对象类的继承．

代码演示

1) 原型链法

function Animal() {

this.name = 'animal';

}

Animal.prototype.sayName = function(){

alert(this.name);

};

function Person() {}

Person.prototype = Animal.prototype; // 人继承自动物

Person.prototype.constructor = 'Person'; // 更新构造函数为人

2) 属性复制法

function Animal() {

this.name = 'animal';

}

Animal.prototype.sayName = function() {

alert(this.name);

};

function Person() {}

for(prop in Animal.prototype) {

Person.prototype[prop] = Animal.prototype[prop];

} // 复制动物的所有属性到人量边

Person.prototype.constructor = 'Person'; // 更新构造函数为人

3) 构造器应用法

function Animal() {

this.name = 'animal';

}

Animal.prototype.sayName = function() {

alert(this.name);

};

function Person() {

Animal.call(this); // apply, call, bind方法都可以．细微区别，后面会提到．

}

**3. js类多重继承的实现方法是怎么样的?**

参考答案：就是类继承里边的属性复制法来实现．因为当所有父类的prototype属性被复制后，子类自然拥有类似行为和属性．

**4. js里的作用域是什么样子的？**

参考答案：大多数语言里边都是块作作用域，以{}进行限定，js里边不是．js里边叫函数作用域，就是一个变量在全函数里有效．比如有个变量p1在函数最后一行定义，第一行也有效，但是值是undefined.

代码演示

var globalVar = 'global var';

function test() {

alert(globalVar); // undefined, 因为globalVar在本函数内被重定义了，导致全局失效，这里使用函数内的变量值，可是此时还没定义

var globalVar = 'overrided var'; //　globalVar在本函数内被重定义

alert(globalVar);　// overrided var

}

alert(globalVar); // global var，使用全局变量

**5. js里边的this指的是什么?**

参考答案: this指的是对象本身，而不是构造函数．

代码演示

function Person() {

}

Person.prototype.sayName() { alert(this.name); }

var person1 = new Person();

person1.name = 'michaelqin';

person1.sayName(); // michaelqin

**6. apply, call和bind有什么区别?**

参考答案：三者都可以把一个函数应用到其他对象上，注意不是自身对象．apply,call是直接执行函数调用，bind是绑定，执行需要再次调用．apply和call的区别是apply接受数组作为参数，而call是接受逗号分隔的无限多个参数列表，

代码演示

function Person() {

}

Person.prototype.sayName() { alert(this.name); }

var obj = {name: 'michaelqin'}; // 注意这是一个普通对象，它不是Person的实例

1) apply

Person.prototype.sayName.apply(obj, [param1, param2, param3]);

2) call

Person.prototype.sayName.call(obj, param1, param2, param3);

3) bind

var sn = Person.prototype.sayName.bind(obj);

sn([param1, param2, param3]); // bind需要先绑定，再执行

sn(param1, param2, param3); // bind需要先绑定，再执行

**7. caller, callee和arguments分别是什么?**

参考答案: caller,callee之间的关系就像是employer和employee之间的关系，就是调用与被调用的关系，二者返回的都是函数对象引用．arguments是函数的所有参数列表，它是一个类数组的变量．

代码演示

function parent(param1, param2, param3) {

child(param1, param2, param3);

}

function child() {

console.log(arguments); // { '0': 'mqin1', '1': 'mqin2', '2': 'mqin3' }

console.log(arguments.callee); // [Function: child]

console.log(child.caller); // [Function: parent]

}

parent('mqin1', 'mqin2', 'mqin3');

**8. 什么是闭包，闭包有哪些用处?**

参考答案: 闭包这个术语，无论中文翻译还是英文解释都太２Ｂ了，我必须骂人，因为它什么其实都不是．非要讲它是什么的话，两个字函数，更多字嵌套函数的父子自我引用关系．所有函数都是闭包．通俗的说，闭包就是作用域范围，因为js是函数作用域，所以函数就是闭包．全局函数的作用域范围就是全局，所以无须讨论．更多的应用其实是在内嵌函数，这就会涉及到内嵌作用域，或者叫作用域链．说到内嵌，其实就是父子引用关系(父函数包含子函数，子函数因为函数作用域又引用父函数，这它妈不是死结吗？所以叫闭包），这就会带来另外一个问题，什么时候引用结束？如果不结束，就会一直占用内存，引起内存泄漏．好吧，不用的时候就引用设为空，死结就解开了．

**9. defineProperty, hasOwnProperty, propertyIsEnumerable都是做什么用**的？

参考答案：Object.defineProperty(obj, prop, descriptor)用来给对象定义属性,有value,writable,configurable,enumerable,set/get等.hasOwnProerty用于检查某一属性是不是存在于对象本身，继承来的父亲的属性不算．propertyIsEnumerable用来检测某一属性是否可遍历，也就是能不能用for..in循环来取到.

**10. js常用设计模式的实现思路，单例，工厂，代理，装饰，观察者模式等**

参考答案：

1) 单例：　任意对象都是单例，无须特别处理

var obj = {name: 'michaelqin', age: 30};

2) 工厂: 就是同样形式参数返回不同的实例

function Person() { this.name = 'Person1'; }

function Animal() { this.name = 'Animal1'; }

function Factory() {}

Factory.prototype.getInstance = function(className) {

return eval('new ' + className + '()');

}

var factory = new Factory();

var obj1 = factory.getInstance('Person');

var obj2 = factory.getInstance('Animal');

console.log(obj1.name); // Person1

console.log(obj2.name); // Animal1

3) 代理: 就是新建个类调用老类的接口,包一下

function Person() { }

Person.prototype.sayName = function() { console.log('michaelqin'); }

Person.prototype.sayAge = function() { console.log(30); }

function PersonProxy() {

this.person = new Person();

var that = this;

this.callMethod = function(functionName) {

console.log('before proxy:', functionName);

that.person[functionName](); // 代理

console.log('after proxy:', functionName);

}

}

var pp = new PersonProxy();

pp.callMethod('sayName'); // 代理调用Person的方法sayName()

pp.callMethod('sayAge'); // 代理调用Person的方法sayAge()

4) 观察者: 就是事件模式，比如按钮的onclick这样的应用.

function Publisher() {

this.listeners = [];

}

Publisher.prototype = {

'addListener': function(listener) {

this.listeners.push(listener);

},

'removeListener': function(listener) {

delete this.listeners[listener];

},

'notify': function(obj) {

for(var i = 0; i < this.listeners.length; i++) {

var listener = this.listeners[i];

if (typeof listener !== 'undefined') {

listener.process(obj);

}

}

}

}; // 发布者

function Subscriber() {

}

Subscriber.prototype = {

'process': function(obj) {

console.log(obj);

}

};　// 订阅者

var publisher = new Publisher();

publisher.addListener(new Subscriber());

publisher.addListener(new Subscriber());

publisher.notify({name: 'michaelqin', ageo: 30}); // 发布一个对象到所有订阅者

publisher.notify('2 subscribers will both perform process'); // 发布一个字符串到所有订阅者

**11. 列举数组相关的常用方法**

参考答案: push/pop, shift/unshift, split/join, slice/splice/concat, sort/reverse, map/reduce, forEach, filter

**12. 列举字符串相关的常用方法**

参考答案: indexOf/lastIndexOf/charAt, split/match/test, slice/substring/substr, toLowerCase/toUpperCase

**Node面试题**

**1. 为什么要用node?**

参考答案: 总结起来node有以下几个特点:简单强大，轻量可扩展．简单体现在node使用的是javascript,json来进行编码，人人都会；强大体现在非阻塞IO,可以适应分块传输数据，较慢的网络环境，尤其擅长高并发访问；轻量体现在node本身既是代码，又是服务器，前后端使用统一语言;可扩展体现在可以轻松应对多实例，多服务器架构，同时有海量的第三方应用组件．

**2. node的构架是什么样子的?**

参考答案: 主要分为三层，应用app >> V8及node内置架构 >> 操作系统. V8是node运行的环境，可以理解为node虚拟机．node内置架构又可分为三层: 核心模块(javascript实现) >> c++绑定 >> libuv + CAes + http.

**3. node有哪些核心模块?**

参考答案: EventEmitter, Stream, FS, Net和全局对象

node全局对象

**1. node有哪些全局对象?**

参考答案: process, console, Buffer和exports

**2. process有哪些常用方法?**

参考答案: process.stdin, process.stdout, process.stderr, process.on, process.env, process.argv, process.arch, process.platform, process.exit

**3. console有哪些常用方法?**

参考答案: console.log/console.info, console.error/console.warning, console.time/console.timeEnd, console.trace, console.table

**4. node有哪些定时功能?**

参考答案: setTimeout/clearTimeout, setInterval/clearInterval, setImmediate/clearImmediate, process.nextTick

node中的事件循环是什么样子的?

总体上执行顺序是：process.nextTick >> setImmidate >> setTimeout/SetInterval 看官网吧： https://github.com/nodejs/node/blob/master/doc/topics/event-loop-timers-and-nexttick.md

node中的Buffer如何应用?

参考答案: Buffer是用来处理二进制数据的，比如图片，mp3,数据库文件等.Buffer支持各种编码解码，二进制字符串互转．

**EventEmitter**

**1. 什么是EventEmitter?**

参考答案: EventEmitter是node中一个实现观察者模式的类，主要功能是监听和发射消息，用于处理多模块交互问题.

**2. 如何实现一个EventEmitter?**

参考答案: 主要分三步：定义一个子类，调用构造函数，继承EventEmitter

代码演示

var util = require('util');

var EventEmitter = require('events').EventEmitter;

function MyEmitter() {

EventEmitter.call(this);

} // 构造函数

util.inherits(MyEmitter, EventEmitter); // 继承

var em = new MyEmitter();

em.on('hello', function(data) {

console.log('收到事件hello的数据:', data);

}); // 接收事件，并打印到控制台

em.emit('hello', 'EventEmitter传递消息真方便!');

**3. EventEmitter有哪些典型应用?**

参考答案: 1) 模块间传递消息 2) 回调函数内外传递消息 3) 处理流数据，因为流是在EventEmitter基础上实现的. 4) 观察者模式发射触发机制相关应用

**4. 怎么捕获EventEmitter的错误事件?**

参考答案: 监听error事件即可．如果有多个EventEmitter,也可以用domain来统一处理错误事件.

代码演示

var domain = require('domain');

var myDomain = domain.create();

myDomain.on('error', function(err){

console.log('domain接收到的错误事件:', err);

}); // 接收事件并打印

myDomain.run(function(){

var emitter1 = new MyEmitter();

emitter1.emit('error', '错误事件来自emitter1');

emitter2 = new MyEmitter();

emitter2.emit('error', '错误事件来自emitter2');

});

**5. EventEmitter中的newListenser事件有什么用处?**

参考答案: newListener可以用来做事件机制的反射，特殊应用，事件管理等．当任何on事件添加到EventEmitter时，就会触发newListener事件，基于这种模式，我们可以做很多自定义处理.

代码演示

var emitter3 = new MyEmitter();

emitter3.on('newListener', function(name, listener) {

console.log("新事件的名字:", name);

console.log("新事件的代码:", listener);

setTimeout(function(){ console.log("我是自定义延时处理机制"); }, 1000);

});

emitter3.on('hello', function(){

console.log('hello　node');

});

Stream

**1. 什么是Stream?**

参考答案: stream是基于事件EventEmitter的数据管理模式．由各种不同的抽象接口组成，主要包括可写，可读，可读写，可转换等几种类型．

**2. Stream有什么好处?**

参考答案: 非阻塞式数据处理提升效率，片断处理节省内存，管道处理方便可扩展等.

**3. Stream有哪些典型应用?**

参考答案: 文件，网络，数据转换，音频视频等.

**4. 怎么捕获Stream的错误事件?**

参考答案: 监听error事件，方法同EventEmitter.

**5. 有哪些常用Stream,分别什么时候使用?**

参考答案: Readable为可被读流，在作为输入数据源时使用；Writable为可被写流,在作为输出源时使用；Duplex为可读写流,它作为输出源接受被写入，同时又作为输入源被后面的流读出．Transform机制和Duplex一样，都是双向流，区别时Transfrom只需要实现一个函数\_transfrom(chunk, encoding, callback);而Duplex需要分别实现\_read(size)函数和\_write(chunk, encoding, callback)函数.

**6. 实现一个Writable Stream?**

参考答案: 三步走:1)构造函数call Writable 2)　继承Writable 3) 实现\_write(chunk, encoding, callback)函数

代码演示

var Writable = require('stream').Writable;

var util = require('util');

function MyWritable(options) {

Writable.call(this, options);

} // 构造函数

util.inherits(MyWritable, Writable); // 继承自Writable

MyWritable.prototype.\_write = function(chunk, encoding, callback) {

console.log("被写入的数据是:", chunk.toString()); // 此处可对写入的数据进行处理

callback();

};

process.stdin.pipe(new MyWritable()); // stdin作为输入源，MyWritable作为输出源

**文件系统**

**1. 内置的fs模块架构是什么样子的?**

参考答案: fs模块主要由下面几部分组成: 1) POSIX文件Wrapper,对应于操作系统的原生文件操作 2) 文件流 fs.createReadStream和fs.createWriteStream 3) 同步文件读写,fs.readFileSync和fs.writeFileSync 4) 异步文件读写, fs.readFile和fs.writeFile

**2. 读写一个文件有多少种方法?**

参考答案: 总体来说有四种: 1) POSIX式低层读写 2) 流式读写 3) 同步文件读写 4) 异步文件读写

**3. 怎么读取json配置文件?**

参考答案: 主要有两种方式，第一种是利用node内置的require('data.json')机制，直接得到js对象; 第二种是读入文件入内容，然后用JSON.parse(content)转换成js对象．二者的区别是require机制情况下，如果多个模块都加载了同一个json文件，那么其中一个改变了js对象，其它跟着改变，这是由node模块的缓存机制造成的，只有一个js模块对象; 第二种方式则可以随意改变加载后的js变量，而且各模块互不影响，因为他们都是独立的，是多个js对象.

**4. fs.watch和fs.watchFile有什么区别，怎么应用**?

参考答案: 二者主要用来监听文件变动．fs.watch利用操作系统原生机制来监听，可能不适用网络文件系统; fs.watchFile则是定期检查文件状态变更，适用于网络文件系统，但是相比fs.watch有些慢，因为不是实时机制．

**网络**

**1. node的网络模块架构是什么样子的?**

参考答案: node全面支持各种网络服务器和客户端，包括tcp, http/https, tcp, udp, dns, tls/ssl等.

**2. node是怎样支持https,tls的?**

参考答案: 主要实现以下几个步骤即可: 1) openssl生成公钥私钥 2) 服务器或客户端使用https替代http 3) 服务器或客户端加载公钥私钥证书

**3. 实现一个简单的http服务器?(必须会手写)**

参考答案: 经典又很没毛意义的一个题目．思路是加载http模块，创建服务器，监听端口.

代码演示

var http = require('http'); // 加载http模块

http.createServer(function(req, res) {

res.writeHead(200, {'Content-Type': 'text/html'}); // 200代表状态成功, 文档类型是给浏览器识别用的

res.write('<meta charset="UTF-8"> <h1>我是标题啊！</h1> <font color="red">这么原生，初级的服务器，下辈子能用着吗?!</font>'); // 返回给客户端的html数据

res.end(); // 结束输出流

}).listen(3000); // 绑定3ooo, 查看效果请访问 http://localhost:3000

child-process

**1. 为什么需要child-process?**

参考答案: node是异步非阻塞的，这对高并发非常有效．可是我们还有其它一些常用需求，比如和操作系统shell命令交互，调用可执行文件，创建子进程进行阻塞式访问或高CPU计算等，child-process就是为满足这些需求而生的．child-process顾名思义，就是把node阻塞的工作交给子进程去做．

**2. exec,execFile,spawn和fork都是做什么用的?**

参考答案: exec可以用操作系统原生的方式执行各种命令，如管道 cat ab.txt | grep hello; execFile是执行一个文件; spawn是流式和操作系统进行交互; fork是两个node程序(javascript)之间时行交互.

**3. 实现一个简单的命令行交互程序?**

参考答案: 那就用spawn吧.

代码演示

var cp = require('child\_process');

var child = cp.spawn('echo', ['你好', "钩子"]); // 执行命令

child.stdout.pipe(process.stdout); // child.stdout是输入流，process.stdout是输出流

// 这句的意思是将子进程的输出作为当前程序的输入流，然后重定向到当前程序的标准输出，即控制台

**4. 两个node程序之间怎样交互?**

参考答案: 用fork嘛，上面讲过了．原理是子程序用process.on, process.send，父程序里用child.on,child.send进行交互.

代码演示

1) fork-parent.js

var cp = require('child\_process');

var child = cp.fork('./fork-child.js');

child.on('message', function(msg){

console.log('老爸从儿子接受到数据:', msg);

});

child.send('我是你爸爸，送关怀来了!');

2) fork-child.js

process.on('message', function(msg){

console.log("儿子从老爸接收到的数据:", msg);

process.send("我不要关怀，我要银民币！");

});

**5. 怎样让一个js文件变得像linux命令一样可执行**?

参考答案: 1) 在myCommand.js文件头部加入 #!/usr/bin/env node 2) chmod命令把js文件改为可执行即可 3) 进入文件目录，命令行输入myComand就是相当于node myComand.js了

**6. child-process和process的stdin,stdout,stderror是一样的吗?**

参考答案: 概念都是一样的，输入，输出，错误，都是流．区别是在父程序眼里，子程序的stdout是输入流，stdin是输出流．

**node高级话题(异步，部署，性能调优，异常调试等)**

**1. node中的异步和同步怎么理解**

参考答案: node是单线程的，异步是通过一次次的循环事件队列来实现的．同步则是说阻塞式的IO,这在高并发环境会是一个很大的性能问题，所以同步一般只在基础框架的启动时使用，用来加载配置文件，初始化程序什么的．

**2. 有哪些方法可以进行异步流程的控制?**

参考答案: 1) 多层嵌套回调 2)　为每一个回调写单独的函数，函数里边再回调 3) 用第三方框架比方async, q, promise等

**3. 怎样绑定node程序到80端口?**

参考答案: 多种方式 1) sudo 2) apache/nginx代理 3) 用操作系统的firewall iptables进行端口重定向

**4. 有哪些方法可以让node程序遇到错误后自动重启?**

参考答案: 1) runit 2) forever 3) nohup npm start &

**5. 怎样充分利用多个CPU?**

参考答案: 一个CPU运行一个node实例

**6. 怎样调节node执行单元的内存大小?**

参考答案: 用--max-old-space-size 和 --max-new-space-size 来设置 v8 使用内存的上限

**7. 程序总是崩溃，怎样找出问题在哪里?**

参考答案: 1) node --prof 查看哪些函数调用次数多 2) memwatch和heapdump获得内存快照进行对比，查找内存溢出

**8. 有哪些常用方法可以防止程序崩溃**?

参考答案: 1) try-catch-finally 2) EventEmitter/Stream error事件处理 3) domain统一控制 4) jshint静态检查 5) jasmine/mocha进行单元测试

**9. 怎样调试node程序?**

参考答案: node --debug app.js 和node-inspector

**常用知名第三方类库(Async, Express等**)

**1. async都有哪些常用方法，分别是怎么用?**

参考答案: async是一个js类库，它的目的是解决js中异常流程难以控制的问题．async不仅适用在node.js里，浏览器中也可以使用． 1) async.parallel并行执行完多个函数后，调用结束函数

async.parallel([

function(){ ... },

function(){ ... }

], callback);

**2) async.series串行执行完多个函数后，调用结束函数**

async.series([

function(){ ... },

function(){ ... }

]);

**3) async.waterfall依次执行多个函数，后一个函数以前面函数的结果作为输入参数**

async.waterfall([

function(callback) {

callback(null, 'one', 'two');

},

function(arg1, arg2, callback) {

// arg1 now equals 'one' and arg2 now equals 'two'

callback(null, 'three');

},

function(arg1, callback) {

// arg1 now equals 'three'

callback(null, 'done');

}

], function (err, result) {

// result now equals 'done'

});

**4) async.map异步执行多个数组，返回结果数组**

async.map(['file1','file2','file3'], fs.stat, function(err, results){

// results is now an array of stats for each file

});

**5) async.filter异步过滤多个数组，返回结果数组**

async.filter(['file1','file2','file3'], fs.exists, function(results){

// results now equals an array of the existing files

});

**2. express项目的目录大致是什么样子的**

参考答案: app.js, package.json, bin/www, public, routes, views.

**3. express常用函数**

参考答案: express.Router路由组件,app.get路由定向，app.configure配置，app.set设定参数,app.use使用中间件

**4. express中如何获取路由的参数**

参考答案: /users/:name使用req.params.name来获取; req.body.username则是获得表单传入参数username; express路由支持常用通配符 ?, +, \*, and ()

**5. express response有哪些常用方法**

参考答案: res.download() 弹出文件下载

res.end() 结束response

res.json() 返回json

res.jsonp() 返回jsonp

res.redirect() 重定向请求

res.render() 渲染模板

res.send() 返回多种形式数据

res.sendFile 返回文件

res.sendStatus() 返回状态

其它相关后端常用技术(MongoDB, Redis, Apache, Nginx等)

**1. mongodb有哪些常用优化措施**

参考答案: 类似传统数据库，索引和分区．

**2. mongoose是什么？有支持哪些特性?**

参考答案: mongoose是mongodb的文档映射模型．主要由Schema, Model和Instance三个方面组成．Schema就是定义数据类型，Model就是把Schema和js类绑定到一起，Instance就是一个对象实例．常见mongoose操作有,save, update, find. findOne, findById, static方法等．

**2. redis支持哪些功能**

参考答案: set/get, mset/hset/hmset/hmget/hgetall/hkeys, sadd/smembers, publish/subscribe, expire

**3. redis最简单的应用**

参考答案:

var redis = require("redis"),

client = redis.createClient();

client.set("foo\_rand000000000000", "some fantastic value");

client.get("foo\_rand000000000000", function (err, reply) {

console.log(reply.toString());

});

client.end();

**4. apache,nginx有什么区别?**

参考答案: 二者都是代理服务器，功能类似．apache应用简单，相当广泛．nginx在分布式，静态转发方面比较有优势．