Table of Contents

前端小分队读书计划

时间： **2016-12-01**

内容：前端知识及技术

组长：何纪成

成员： 郭乃豪，王立银，刘沈，宋健

计划：

每周五下午三点左右进行读书分享活动，每周两个人进行分享，分享类型为前端涉及知识以及相关技术！

2016年12月02日 第一周

分享人

- 郭乃豪
  - 理解AngularJS
- 宋健
  - 初识NodeJS

一、 结合背景理解AngularJS

1.1 理解AngularJS的擅长之处

1.1.1 回合式和单页面应用程序

广义上来讲，存在两种类型的Web应用程序：回合式和单页面。
回合式：由浏览器向服务器请求一个厨师的HTML文档，用户交互会使浏览器请求一个新想HTML文档。在这类应用程序中，浏览器基本上是一个HTML内容的解析引擎，所有的应用程序逻辑和数据都保留在服务器上。

回合式不足：用户在下一个HTML文档被请求和加载之前必须等待，它需要大型的服务端基础设施来处理所有请求并管理所有的应用程序状态，需要大型的服务器基础设施来处理所有请求并管理所有的应用程序状态，需要许多带宽。

单页面：一个初始的HTML文档被发送给浏览器，但是用户交互的所产生的ajax请求只会请求较小的HTML片段，或者要插入到已有的显示给用户元素中的数据。初始的HTML文档不会被再次加载或者替换，在Ajax请求被异步执行的时候用户还可以继续与已有的HTML进行交互。

AngularJS 以单页面应用程序和复杂的回合式应用程序见长。对于较简单的项目，一般来说jQuery或者类似的替代者会是更好的选择。

1.2 MVC模式

MVC全名是Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。

1.2.1 模型

模型包含用户赖以工作的数据，有两种广义上的模型：视图模型，只表示从控制器传往视图的数据；领域模型，包含业务领域的数据，以及用于创建、存储和操纵这些数据的各种操作、转换和规则，统称模型逻辑。

1.2.2 控制器

在一个AngularJS Web应用中，控制器作为数据模型和视图之间的渠道。控制器会想作用域中添加业务领域逻辑，作用域是模型的子集。
控制器应当：包含初始化作用域所需的逻辑；包含视图所需的用于表示作用域中的数据的逻辑/行为；包含根据用户交互来更新作用域所需的逻辑/行为。

1.2.3 视图

视图是通过HTML元素定义的，这些元素是通过数据绑定或者指令来进行增强或者生成的。
视图应当：包含将数据呈现给用户所需的逻辑和标记。

1.3 RESTful服务

常见的使用ajax请求调用服务器端代码。

1.4 常见的设计陷阱

1.4.1 讲逻辑放到错误的地方

常见为：将业务逻辑放到视图中而不是控制器中；将领域逻辑放到控制器中而不是模型中；在使用RESTful服务是将数据存储逻辑放到客户端模型中。
开发需遵守的三条规则： 视图逻辑应该仅为显示准备数据，而且永远不应该修改模型； 控制器逻辑永远都不应该直接创建、更新或删除模型中的数据； 客户端永远都不应该直接访问数据存储。

1.4.2 采用数据存储所依赖的数据格式

在一个设计良好的从RESTful服务获取初级的AngularJS 应用程序中，服务端的职责应当是隐藏数据存储的实现细节，影响客户端一一种合适的数据格式表达数据，使客户端使用起来简洁易用。

1.4.3 默守成规

如果在一个AngularJS 应用中你在直接使用jQuery操作DOM，那么你就遇到问题了。因为使用jQuery不易分离MVC的各个组件，并使得所创建的Web应用难以测试、优化和维护。

什么是Node.js

1、Node.js就是运行在服务端的JavaScrip。

2、Node.js是一个基于Chrome JavaScrip运行时简历的一个平台。

3、Node.js是一个非阻塞I/O服务端JavaScript环境，基于Google的V8引擎，V8引擎执行Javascript的速度非常快。

看下官网的介绍：

```
Node.js is a platform built on Chrome's JavaScript runtime for easily building fast,
scalable network applications. Node.js uses an event-driven, non-blocking I/O model
that makes it lightweight and efficient, perfect for data-intensive real-time
applications that run across distributed devices.
```

Node.js的使用场景

在看他的使用场景之前先了解下Node js 的优缺点吧

一、Node js特点

```
1、他是一个JavaScript运行环境

2、依赖于Chrome V8引擎进行代码解析

3、事件驱动非阻塞I/O

4、轻量、可伸缩、适用于实时数据交互应用

5、单进程，单线程
```

二、Node js优、缺点

优点：

```
1、并发高是选择Node js重要的优点
2、适合I/O密集型应用
```

缺点：

```
1、不适合CPU密集型应用：CPU密集型应用给Node带来的挑战主要是：由于JavaScript单线程的原因，
   如果有长时间运行的计算（比如大循环），将会导致CPU时间片不能释放，使得后续I/O无法发起：

解决方案：分解大型运算任务为多个小任务，使得运算能够适时释放，不阻塞I/O调用的发起。

2、因为是单进程单线程的只能使用一个CPU，不能充分使用CPU导致资源的浪费

3、可靠性低、一呆代码某个环节崩溃，整个系统都将会崩溃
   原因：他是单线程、但单进程的

   解决方法：

       3.1、Nginx反向代理负载均衡开多个进程，绑定多个端口

       3.2、开多个进程监听同一个端口，使用cluster模块

 4、开源组件库质量参差不齐，更新快，向下不兼容

 5、Debug不方便，错误没有stack trace
```

三、适合Node js的场景

1、Restful API

这是NodeJS最理想的应用场景，可以处理数万条连接，本身没有太多的逻辑，只需要请求API，组织数据进行返回即可。它本质上只是从某个数据库中查找一些值并将它们组成一个响应。由于响应是少量文本，入站请求也是少量的文本，因此流量不高，一台机器甚至也可以处理最繁忙的公司的API需求。

2、统一Web应用的UI层

目前MVC的架构，在某种意义上来说，Web开发有两个UI层，一个是在浏览器里面我们最终看到的，另一个在server端，负责生成和拼接页面。

不讨论这种架构是好是坏，但是有另外一种实践，面向服务的架构，更好的做前后端的依赖分离。如果所有的关键业务逻辑都封装成REST调用，就意味着在上层只需要考虑如何用这些REST接口构建具体的应用。那些后端程序员们根本不操心具体数据是如何从一个页面传递到另一个页面的，他们也不用管用户数据更新是通过Ajax异步获取的还是通过刷新页面。

3、大量的Ajax请求

例如个性化应用，每个用户看到的页面都不一样，缓存失效，需要在页面加载的时候发起Ajax请求，或者弹幕系统大量的用户同时通过评论，NodeJS能响应大量的并发请求。

总而言之，NodeJS适合运用在高并发、I/O密集、少量业务逻辑的场景。

Node js安装配置

1、下载源码 更多版本：https://nodejs.org/en/download/

```
cd /work/app/
wget http://nodejs.org/dist/v0.10.24/node-v0.10.24.tar.gz
```

2、解压源码

```
tar -zxvf node-v0.10.24.tar.gz
```

3、编译安装

```
cd node-v0.10.24
./configure --prefix=/work/app
make
make install
```

4、配置NODE_HOME，进入profile编辑环境变量

```
vim /etc/profile
```

设置nodejs环境变量，在 export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL 一行的上面添加如下内容

```
export NODE_HOME=/work/app/node/0.10.24
export PATH=$NODE_HOME/bin:$PATH
```

:wq 保存并退出，编译/etc/profile 使配置生效

source /etc/profile 验证是否安装配置成功

node -v 输出 v0.10.24 表示配置成功

npm模块安装路径

```
/work/app/node/0.10.24/lib/node_modules
```

2016年12月09日 第2周

分享人

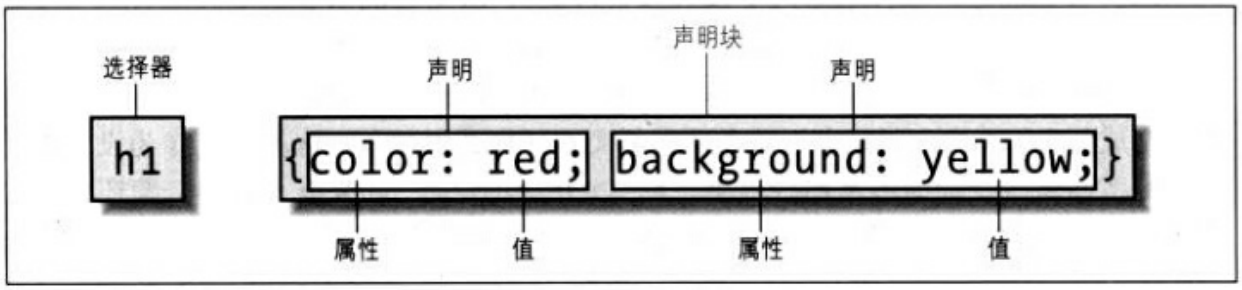
- 王立银
  - **css**优先级规则
- 刘沈
  - **AngularJS**指令的**Scope(**作用域**)**

css优先级计算规则

特殊性

CSS继承是从一个元素向其后代元素传递属性值所采用的机制。确定应当向一个元素应用哪些值时，浏览器不仅要考虑继承，还要考虑声明的特殊性，另外需要考虑声明本身的来源。这个过程就称为层叠。——《css权威指南》

上面这句话有两个词需要稍作解释，“声明”和“特殊性”。如下图，css规则由选择器和声明块组成，写在选择器后面大括号里的就叫声明。



实际上，同一个元素可以使用多个规则来指定它的字体颜色，每个规则都有自己的选择器。显然最终只有一个规则起作用（不可能一个字既是红色又是绿色），那么该规则的特殊性最高，特殊性即css优先级。很多同学仅仅知道选择器优先级ID>class>元素选择器，而不知道ID的优先级为什么大于class的优先级。那么css优先级到底是怎么计算的呢？

特殊性分类

选择器的特殊性值表述为4个部分，用0,0,0,0表示。

1. 内联样式的特殊性值,加1,0,0,0。
2. ID选择器的特殊性值，加0,1,0,0。
3. 类选择器、属性选择器或伪类，加0,0,1,0。
4. 元素和伪元素，加0,0,0,1。
5. 通配选择器*，子选择器（>）和相邻同胞选择器（+）对特殊性没有贡献,即0,0,0,0。
6. 比较特殊的一个标志!important（权重），它没有特殊性值，但它的优先级是最高的，为了方便记忆，可以认为它的特殊性值为1,0,0,0,0。

易混淆

伪类(:link,:hover,:active,:visited,:focus,:first-child,:first,:left,:right,:lang)
伪元素(:first-letter,:first-line,:before,:after)

例子

```
1 a{color: yellow;} /*特殊性值：0,0,0,1*/
2 div a{color: green;} /*特殊性值：0,0,0,2*/
3 .demo a{color: black;} /*特殊性值：0,0,1,1*/
4 .demo input[type="text"]{color: blue;} /*特殊性值：0,0,2,1*/
5 .demo *[type="text"]{color: grey;} /*特殊性值：0,0,2,0*/
6 #demo a{color: orange;} /*特殊性值：0,1,0,1*/
7 div#demo a{color: red;} /*特殊性值：0,1,0,2*/
```

分析上面的demo，要注意特殊性是怎么排序的，上面第4行和第5行规则，第4行之所以优先级比第5行高，是因为第四行特殊性值最后面是1，而第5行特殊性值最后面是0。回过头来回答文章最开始的问题，为什么ID选择器的优先级比类选择器的优先级高？实际上是因为选择器特殊性值是从左向右排序的，特殊性值1,0,0,0大于以0开头的所有特殊性值，即便它是0,99,99,99，优先级依然比1,0,0,0要低。

补充

通配选择器*的特殊性值是0,0,0,0，而元素通过父元素继承过来的样式是没有特殊性值的，所以，通配选择器定义的规则优先级高于元素继承过来的规则的优先级

层叠

假如特殊性相同的两条规则应用到同一个元素会怎样？css会先查看规则的权重（!important），加了权重的优先级最高，当权重相同的时候，会比较规则的特殊性，根据前面的优先级计算规则决定哪条规则起作用，当特殊性值也一样的时候，css规则会按顺序排序，后声明的规则优先级高，成为人生赢家，当上总经理出任CEO迎娶白富美。

$scope

$scope 的使用贯穿整个 AngularJS App 应用,它与数据模型相关联,同时也是表达式执行的上下文.有了 $scope 就在视图和控制器之间建立了一个通道,基于作用域视图在修改数据时会立刻更新 $scope,同样的 $scope 发生改变时也会立刻重新渲染视图.（相当于作用域、控制范围）

有了 $scope 这样一个桥梁,应用的业务代码可以都在 controller 中,而数据都存放在controller 的 $scope 中.

$rootScope

AngularJS 应用启动并生成视图时,会将根 ng-app 元素与 $rootScope 进行绑定.$rootScope 是所有 $scope 的最上层对象,可以理解为一个 AngularJS应用中得全局作用域对象,所以为它附加太多逻辑或者变量并不是一个好主意,和污染 Javascript 全局作用域是一样的.

$scope 的作用

$scope 对象在 AngularJS 中充当数据模型的作用,也就是一般 MVC 框架中Model得角色.但又不完全与通常意义上的数据模型一样,因为 $scope 并不处理和操作数据,它只是建立了视图和 HTML 之间的桥梁,让视图和 Controller 之间可以友好的通讯.

再进一步系统的划分它的作用和功能:

1.提供了观察者可以监听数据模型的变化

2.可以将数据模型的变化通知给整个 App

3.可以进行嵌套,隔离业务功能和数据

4.给表达式提供上下文执行环境

在 Javascript 中创建一个新的执行上下文,实际就是用函数创建了一个新的本地上下文,在 AngularJS 中当为子 DOM 元素创建新的作用域时,其实就是为子 DOM 元素创建了一个新的执行上下文.

$scope 生命周期(作用域)

AngularJS 中也有一个'事件'的概念,比如当一个绑定了 ng-model 的 input 值发生变化时,或者一个 ng-click 的 button 被点击时,AngularJS 的事件循环就会启动.事件循环是 AngularJS 中非常非常核心的一个概念,因为不是本文主旨所以不多说,感兴趣的可以自己看看资料.这里事件就在 AngularJS 执行上下文中处理,$scope 就会对定义的表达式求值.此时事件循环被启动, AngularJS 会监控应用程序内所有对象,脏值检查循环也会启动.

$scope 的生命周期有4个阶段:

1. 创建

控制器或者指令创建时, AngularJS 会使用 $injector 创建一个新的作用域,然后在控制器或指令运行时,将作用域传递进去.

2. 链接

AngularJS 启动后会将所有 $scope 对象附加或者说链接到视图上,所有创建 $scope 对象的函数也会被附加到视图上.这些作用域将会注册当 AngularJS 上下文发生变化时需要运行的函数.也就是 $watch 函数, AngularJS 通过这些函数或者何时开始事件循环.$ $scope.$watch

3. 更新

一旦事件循环开始运行,就会开始执行自己的脏值检测.一旦检测到变化,就会触发 $scope 上指定的回调函数 $http,$timeout,$interval

4. 销毁

通常来讲如果一个 $scope 在视图中不再需要, AngularJS 会自己清理它.当然也可以通过 $destroy() 函数手动清理.

2016年12月16日 第3周

分享人

- 刘沈
  - **Scope**补充
- 宋健
  - **javascript**之闭包

$scope

Scope(作用域) 是应用在 HTML (视图) 和 JavaScript (控制器)之间的纽带。Scope 是一个对象，有可用的方法和属性。Scope 可应用在视图和控制器上。当在控制器中添加 $scope 对象时，视图 (HTML) 可以获取了这些属性。视图中，你不需要添加 $scope 前缀, 只需要添加属性名即可，如： 。

1.1 根作用域$rootScope

所有的应用都有一个 $rootScope，它可以作用在 ng-app 指令包含的所有 HTML 元素中。$rootScope 可作用于整个应用中。是各个 controller 中 scope 的桥梁。用 rootscope 定义的值，可以在各个 controller 中使用。
例：

```
<div ng-app="myApp" ng-controller="myCtrl">

<h1> 家族成员:</h1>

<ul>
    <li ng-repeat="x in names"> </li>
</ul>

</div>

<script>
  var app = angular.module('myApp', []);

  app.controller('myCtrl', function($scope, $rootScope) {
      $scope.names = ["Emil", "Tobias", "Linus"];
      $rootScope.lastname = "Refsnes";
  });
</script>
```

1.2 $scope 生命周期(作用域)

1.2.1 $scope 的生命周期有4个阶段:

1. 创建
   ng-controller指令是作用域创建指令, 当在DOM树中遇到作用域创建指令时,AngularJs都会创建Scope类的新实例$scope.

2. 监视器注册阶段
   模板链接阶段为在模板中表示的作用域中的值注册监视器。这些监视器自动将模型的更改传播到DOM元素。也可以利用$watch()方法在一个作用域值上注册你自己的监视函数。
   例:

   ```
   $scope.watchedItem = 'myItem';
   $scope.counter = 0;
   $scope.$watch('name',function(newValue, oldValue){
   $scope.watchedItem = $scope.counter+1;
   })
   ```

3. 变化阶段
   该阶段发生在该作用域内的数据发生变化时。
   当你在angularJS代码进行更改时，一个名为$apply()的作用域函数更新模型，并调用$digest()函数来更新DOM

和监视器。

4. 销毁

$destroy()方法从浏览器内存中删除作用域。在作用域不再需要时angularJS库自动调用此方法。

javascript之闭包

闭包的概念

闭包（closure）是 JavaScript 的一种语法特性。

关于闭包，有一种经典的提法——“闭包是代码块和创建该代码块的上下文（环境）中数据的结合”。

闭包就是在函数内部定义函数，内部的函数可访问其外部函数的作用域。下面是在程序中实现闭包的例子。

```
function outer(name){ // 外部的函数

 var msg="hello";
  function inner(){ // 内部函数
   alert(msg+" "+name);
  }
  return inner(); // 返回内部函数
 }
 var clos=outer("WANGERN");
 clos();
```

执行代码，将弹出警告“hello WANGERN”。

前提

要理解闭包，还有一个很关键性概念—— JavaScript 的作用域规则。先解释一下作用域（scope）。在运行函数都会创建属于函数的上下文环境（context）及作用域，作用域即当前环境范围内的变量。JavaScript 中最外围的环境为 window 对象，也就是全局作用域所在的环境。当执行到下一级环境时，下一级环境会主动包含上一级的作用域，最终形成一级一级关联的作用域链（对象的 [[Scope]] 属性指向该作用域链）。当有下一级环境生成时，上一级环境会失活，但不会自动销毁而保存在一种“栈”式结构中，这样可以保证作用域链的延续性，也可以环境回退时再次激活。当前环境可访问当前作用域链中的全部变量，比如上面代码中的 inner() 函数可访问 outer() 函数中的 msg 和 name 变量。

闭包就是借助这种作用域链，一方面可使内部函数可访问外部函数的变量；另一方面，闭包还可以抑制外部函数环境的销毁，使其变量始终保存在内存中，直至不需要时再销毁。

闭包的实例

先看下面的例子，当function里嵌套function时，内部的function可以访问外部function里的变量。

```
function foo(x) {
    var tmp = 3;
    function bar(y) {
        alert(x + y + (++tmp));
    }
    bar(10);
}
foo(2)
```

不管执行多少次，都会alert 16，因为bar能访问foo的参数x，也能访问foo的变量tmp。

但，这还不是闭包。当你return的是内部function时，就是一个闭包。内部function会close-over外部function的变量直到内部function结束。

```
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);
```

上面的脚本最终也会alert 16，因为虽然bar不直接处于foo的内部作用域，但bar还是能访问x和tmp。

但是，由于tmp仍存在于bar闭包的内部，所以它还是会自加1，而且你每次调用bar时它都会自加1.

(我们其实可以建立不止一个闭包方法，比如return它们的数组，也可以把它们设置为全局变量。它们全都指向相同的x和相同的tmp，而不是各自有一份副本。)

上面的x是一个字面值(值传递)，和JS里其他的字面值一样，当调用foo时，实参x的值被复制了一份，复制的那一份作为了foo的参数x。

那么问题来了，JS里处理object时是用到引用传递的，那么，你调用foo时传递一个object，foo函数return的闭包也会引用最初那个object！

```
function foo(x) {
var tmp = 3;
return function (y) {
    alert(x + y + tmp);
    x.memb = x.memb ? x.memb + 1 : 1;
    alert(x.memb);
    }
}
var age = new Number(2);
var bar = foo(age); // bar 现在是一个引用了age的闭包
bar(10);
```

不出我们意料，每次运行bar(10)，x.memb都会自加1。但需要注意的是x每次都指向同一个object变量——age，运行两次bar(10)后，age.memb会变成2.

这和HTML对象的内存泄漏有关，呃，不过貌似超出了答题的范围。

这里有一个不用return关键字的闭包例子：

```
function closureExample(objID, text, timedelay) {
    setTimeout(function() {
        document.getElementById(objID).innerHTML = text;
    }, timedelay);
}
closureExample('myDiv', 'Closure is created', 500);
```

JS里的function能访问它们的：

1. 参数
2. 局部变量或函数
3. 外部变量（环境变量？），包括

   3.1 全局变量，包括DOM。

   3.2 外部函数的变量或函数。

如果一个函数访问了它的外部变量，那么它就是一个闭包。

注意，外部函数不是必需的。通过访问外部变量，一个闭包可以维持（keep alive）这些变量。在内部函数和外部函数的例子中，外部函数可以创建局部变量，并且最终退出；但是，如果任何一个或多个内部函数在它退出后却没有退出，那么内部函数就维持了外部函数的局部数据。

一个典型的例子就是全局变量的使用。

从技术上来讲，在JS中，每个function都是闭包，因为它总是能访问在它外部定义的数据。

闭包经常用于创建含有隐藏数据的函数（但并不总是这样）。

```
var db = (function() {
// 创建一个隐藏的object，这个object持有一些数据
// 从外部是不能访问这个object的
var data = {};
// 创建一个函数，这个函数提供一些访问data的数据的方法
return function(key, val) {
    if (val === undefined) { return data[key] } // get
    else { return data[key] = val } // set
    }
// 我们可以调用这个匿名方法
// 返回这个内部函数，它是一个闭包
})();

db('x'); // 返回 undefined
db('x', 1); // 设置data['x']为1
db('x'); // 返回 1
// 我们不可能访问data这个object本身
// 但是我们可以设置它的成员
```

闭包的总结

(1) 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；

(2) 网上主流的对闭包剖析的文章实际上是和闭包原则反向而驰的，如果需要知道闭包细节才能用好的话，这个闭包是设计失败的；