前端笔记

yugang@myhexin.com

目录

关于 FE

1.1 移动端 rem.js、图片懒加载的不足

rem.js：用淘宝封装的 flexable.js
图片懒加载：未考虑 img.onload，参考文档(http://172.20.200.191:8003/pages/viewpage.action?pageId=9516154)

1.2 UI 字体

字体优先原则，尽量与设计稿字体一致。

1.3 落地页之类顶图加载方式

Base64 模糊加载

```
<img id="topload" class="u-w100" src=""
 data-original="images/banner" data-format="jpg">
<script type="text/javascript">
    loadSpecImg("topload");
</script>
```

```
//load special image
function loadSpecImg (id) {
    var _t = document.getElementById(id),
        _to = _t.getAttribute("data-original"),
        _tf = _t.getAttribute("data-format");

    var dpr = window.devicePixelRatio;
    dpr = dpr > 2 ? 2 : dpr;

    var _img = new Image();
    _img.onload = function(){
            _t.setAttribute('src', _img.src);
            _t.removeAttribute('data-original');
            _t.removeAttribute('data-format');
    };
    if(dpr < 2){
        _img.src = _to + '_1x.' + _tf;
    }else{
        _img.src = _to + '_2x.' + _tf;
    }
}
```

1.4 图片格式选择（色彩度 + 大小）

1x 图：宽 480

2x 图：原稿
模糊图：宽 200 以下，模糊度最高，大小保证 5k 以下

1.5 图片加载方式

针对图片过多或过大的页面，选择合适的图片加载方式，目前支持顺序加载、延时加载、懒加载
url:http://172.20.200.191:8003/pages/viewpage.action?pageId=9516154

关键字	参数	意义
type（配置options）	onebyone	顺序加载
	timeout	延时加载
	lazy	懒加载
domclass（配置options）		图片检索class
container（配置options）		加载容器
init()函数		触发加载器
	param（可选）	type为"lazy"时，param表示页面加载完时最多渲染图片的数量； type为"timeout"时，param表示延时的时间间隔
justrender()函数		单加载某张图片
	obj	需要加载的img节点

依赖 js 见本地文件

1.6 mustche.js 模块抽离

例子见本地文件

1.7 CSS 命名规范，HTML 语义化

1.8 JS 面向对象的封装（原型、对象自变量表示法）

a：module 模式
b：原型 + 原型链 (03/28 笔记)
c：this 指向 + 实例化 (03/29 笔记)
d：JS 文件结构（公有变量，公有方法，私有....，dom 操作、样式、功能（分离-SRP 原则））

1.9 JS 编写注意

a：采用面向对象的方式编写
b：注意模块化的思想，如果需要进行压缩打包之类操作，需注意依赖关系（NODEJS 模块导出）。
Ps：类似落地页之类页面，一个 js 文件的只需要使用 gulp 进行压缩处理即可，若逻辑复杂，js 文件较多，则使用 webpack 进行打包。

1.10 插件化思想

(03/30 笔记)

1.11 chart

pc：amchart.js、echart.js、highchart.js
mobile：chart.js

1.12 Icon 之类图片用 background

原因：此类图片一般较小，转为 base64 格式后，若图片复用，则增加 HTML 大小，若写入 css 里面，则不会如此。

关于测试

2.1 样式的严谨性

例如宽度不写，iPhone 跟低版本 Android 样式不一致

2.2 属性的兼容性

例如安卓部分机型不支持 flex 布局
方案：用浮动布局、宽度百分比

2.3 关于 REM

宽度（图片、模块）：用百分比
高度及其他：用 rem 单位

2.4 iPhone 跟 Android 使用字体不一致，所占空间大小不

a：用（max-width:百分比）解决。
b：用 @media 改字体大小

2.5 弹框滚动，os8.1 出现不可滚动的情况

原因：弹框子模块高度未明确，用 overflow:auto 样式
方案：弹框子模块高度用 rem 单位写出

2.6 控制浮动按钮的显示隐藏

问题：按钮显示时机不对，有的时候正常，有的时候在到达指定位置之前就显示
原因：specDom 上方存在模糊加载顶图且图片高度未设置，在 2x 图 onload 之前，图片高度会出现 0 的情况
解决方案：模糊加载顶图设置最小高度

关于其他

3.1 touchwatch.js

监听手势以及滑动过程中变向

3.2 ScrollHandle.js

弹框禁止滚动

3.2 滚动选择 demo

移动端选择日期（不可小于当前日期）

ES5：

4.1 定义变量

```
var $obj = JQuery对象;
var obj = Dom对象;’
$(obj) == $obj;
$obj.get(0) == $obj[0] == obj;
```

4.2 变量声明提前

4.3 函数声明提前

4.4 Arguments

ECMAScript6：

5.1 Let const --------> var

let: 代码块内有效,“暂时性死区”

```
if (true) {
    tmp = 'abc' ; // ReferenceError
    let tmp;
}
```

const:声明一个只读的常量，只声明不赋值，就会报错。“暂时性死区”

let 和 const 申明的变量都不提升。

5.2 箭头函数

```
es5:
    function (job) {
        return job;
    }
es6:
    job => job;
```

5.3 箭头函数的 this 绑定

```
es5:
    function foo () {
        var self = this;
        setTimeout ( function () {
            console.log( "id:" + self.id )
        },100);
    }
es6:
    function foo () {
        setTimeout ( function () {
            console.log( "id:" + this.id )
        }.bind( this ) , 100);
    }

    function foo () {
        return () => {
            return () => {
                return () => {
                    console.log("id:", this.id);
                };
            };
```

```
        };
    }
```

5.4 …

```
    function foo (x,y,…args) {
        console.log(args);
    }
    foo (1,2,3,4,5) //[3,4,5]

    let arr = [1,2,3]
    function demo (x,y,z){
        console.log(x,y,z);
    }
    demo(…arr); // 1 2 3
```

5.5 类

```
    class Point {
        constructor (x,y) {
            this.x = x;
            this.y = y;
        }
        toString () {
            return '(' + this.x + ',' + this.y + ')';
        }
    }

    var p = new Point (1,2) ;
```

5.6 继承

```
    class ColorPoint extends Point {
        constructor (x,y,color){
            super (x,y) ;
            this.color = color;
        }
        to String () {
            return this.color + ' ' + super.toString();
        }
    }
```

5.7 Promise

```
es5:
    $.ajax({
        url: '…' ,
        success: function (data) {
```

```
            //…….to do something
        }
    })
es6:
    function printHello (ready) {
        return new Promise ( function (resolve,reject) {
            if (ready) {
                resolve(“hello”);
            } else {
                reject(“goodBye”);
            }
        })
    }
    printHello (true)
        .then (printHello)
        .then (printHello)
        .then (printHello);
```

Gulp:

gulp 是基于 Nodejs 的自动任务运行器,她能自动化地完成 javascript/coffee/ sass/less/html/image/css 等文件的的测试、检查、合并、压缩、格式化、浏览器自动刷新、部署文件生成，并监听文件在改动后重复指定的这些步骤。

6.1 安装 nodejs

6.2 安装 npm / cnpm

6.3 全局安装 gulp

```
npm install gulp -g
```

6.4 新建 package.json

换电脑的时候直接运行 npm install，会自动安装 package 内的依赖

6.5 本地安装 gulp

```
npm install –save-dev gulp
```

6.6 安装需要的 gulp 插件

```
npm install –save-dev
```

6.7 新建 gulpfire.js（配置文件）

```
var gulp = require ('gulp'),
minifycss = require ('gulp-minify-css');

gulp.task("minify-css",function () {
    gulp.src("src/*.css")
        .pipe(minifycss())
        .pipe(gulp.dest("css/"))
})

gulp.task("default" , ["minify-css" , … ]);
```

6.7 自动监听 刷新浏览器

```
//编写服务 监听
gulp.task("watch",function () {
    $.connect.server({
            root:["src/"],//开始读取路径
            livereload:true, //自动刷新浏览器(IE 不支持)
            port:3000 //默认端口
    });
```

```
        open('http://localhost:3000');//默认打开此链接

        //监听以下文件，自动执行对应任务 构建
        gulp.watch("src/*.css", ['minify-css']);
        //ps:若监听到以上文件有变化，对应任务自动执行，但未刷新浏览器
        //在每个任务后面加上'.pipe($.connect.reload())'通知浏览器刷新界面
    })
```

webPack:

webPPack 可以看做是**模块打包机**：它做的事情是，分析你的项目结构，找到 JavaScript 模块以及其它的一些浏览器不能直接运行的拓展语言（Less，TypeScript 等），并将其打包为合适的格式以供浏览器使用。

7.1 安装 webpack

```
npm install –g webpack //全局安装
```

7.2 新建 package.json

7.3 本地安装 webpack

```
npm install –save-dev webpack
```

7.4 新建 webpack.config.js（配置文件）

```
module.exports = {
 devtool: 'eval-source-map',

 entry: __dirname + "/app/main.js"
 output: {
     path: __dirname + "/dist",
     filename: "bundle.js"
 },

 module: {
     loaders: [
         {
             test: /\.json$/,
             use: "json-loader"
         },
         {
             test: /\.js$/,
             exclude: /(node_modules|bower_components)/,
             use: "babel-loader"
             query: {
                 presets: ['react' , 'es2015']
             }
         }
     ]
 }
 // Plugins
}
```

7.5 开发环境

webpack 打包速度比较慢，如果不想每次修改后都重新运行的话,可以启动监听模式—watch，当然，使用 webpack-dev-server 开发服务是一个更好的选择，他会在 localhost:8080 启动一个 express 静态资源 web 服务器，并以监听模式自动运行 webpack。

NodeJS:

8.1 事件驱动和异步 I/O

nodejs 中的事件驱动和异步 I/O 是如何实现的？

nodejs 是**单线程**(single thread)运行的，通过一个**事件循环**(event-loop)来循环取出**消息队列**(event-queue)中的消息进行处理,处理过程基本上就是去调用该消息对应的回调函数。**消息队列**就是当一个事件状态发生变化时，就将一个消息压入队列中。

nodejs 的时间驱动模型一般要注意下面几个点：

a：因为是**单线程**的，所以当顺序执行 js 文件中的代码的时候，**事件循环**是被暂停的。

b：当 js 文件执行完以后，**事件循环**开始运行，并从**消息队列**中取出消息，开始执行回调函数

c：因为是**单线程**的，所以当回调函数被执行的时候，**事件循环**是被暂停的

d：当涉及到 I/O 操作的时候，nodejs 会开一个独立的线程来进行异步 I/O 操作，操作结束以后将消息压入消息队列。

NODE 利用单线程，远离多线程死锁、状态同步等问题；利用异步 I/O，让单线程原理阻塞，以更好地使用 CPU。

8.2 路由

所谓路由，就是如何处理 HTTP 请求中的路径部分。

回顾 node 学习中：

```
var handle = {};
handle["/"] = requestHandle.start;
handle["/start"] = requestHandle.start;
handle["/upload"] = requestHandle.upload;
```

以及 Express 中：

```
app.get('/', function (req, res) {
    res.send('Hello World!');
});
```

上面的 hanlde[]以及 app.get()调用，实际上就为我们的网站添加了一条路由，指定路径以及函数来处理。

8.3 中间件

Express 里有个中间件的概念。所谓中间件，就是在收到请求后和发送响应之前这个阶段执行的一些函数。

中间件函数的原型如下：

```
function (req, res, next) { // 中间件 }
```

第一个参数是 Request 对象 req。第二个参数是 Response 对象 res。第三个则是用来驱动中间件调用链的函数 next，如果你想让后面的中间件继续处理请求，就需

要调用 next 方法。
给某个路径应用中间件函数的典型调用是这样的：

```
app.use('/abcd', function (req, res, next) {
    console.log(req.baseUrl);
    next();
})
```

补充：
Express 还提供了一个叫做 Router 的对象，行为很像中间件，你可以把 Router 直接传递给 app.use，像使用中间件那样使用 Router。另外你还可以使用 router 来处理针对 GET、POST 等的路由，也可以用它来添加中间件，总之你可以将 Router 看作一个微缩版的 app。定义了 router 后，也可以将其作为中间件传递给 app.use。

```
// ---- ROUTES ----
// 舊方法
app.get('/sample', function(req, res) {
    res.send('this is a sample!');
});

// Express Router
// 建立 Router 物件
var router = express.Router();
router.get('/', function(req, res) {
    res.send('home page!');
});

router.get('/about', function(req, res) {
    res.send('about page!');
});

// 將路由套用至應用程式
app.use('/', router);

// ---- 啟動伺服器 ----
app.listen(port);
```

以上建立了 http://localhost:8080 和 http://localhost:8080/about 两个网页，若 app.use 如果改成

```
app.use('/app', router);
```

则网页变为 http://localhost:8080/app 和 http://localhost:8080/app/about 這樣的特性可以讓我們很方便地將不同功能的路由區分開來，分別建立不同的 Router 物件，以不同的路徑套用至應用程式中，讓程式結構模組化且更有彈性

8.4 模块

8.4.1 exports? module.exports?

许多初学者都曾经纠结过为何存在 exports 的情况下，还有 module.exports。理想情况下，只要赋值给 exports 即可：

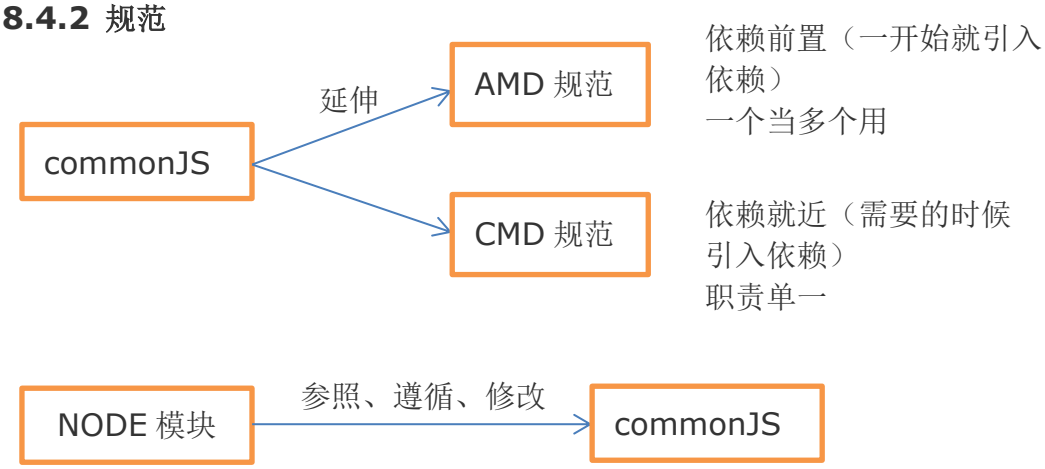
```
exports = function(){
    //my class
};
```

但是通常都会得到一个失败的结果。其原因在于，exports 对象是通过形参的方式传入的，直接赋值形参会改变形参的引用，但并不能改变作用域外的值。测试代码如下：

```
var chage = (a)=>{
    a=100;
    console.log(a); // => 100
};
var a = 10;
chage(a);
console.log(a); // => 10
```

如果要达到 require 引入一个类的效果，请赋值给 module.exports 对象。这个迂回的方案不改变形参的引用。

8.4.2 规范



8.5 应用场景

a：I/O 密集型(读取文件较多)
b：不是很擅长 CPU 密集型(计算较多)业务，但是可以合理调度
c：与遗留系统问题和平共处
d：分布式应用

8.6 NODE 的使用者

a：前后端编程语言环境统一：这类倚重点的代表是雅虎。雅虎开放了 Cocktail 框架，利用自己深厚的前端沉淀，将 YUI3 这个前端框架的能力借助 Node 延伸到服务器端，使得使用者摆脱了日常工作中一边些 JavaScript 一边些 PHP 所带来的上下文交换负担。

b：Node 带来的高性能的 I/O 用于实时应用：Voxer 将 Node 应用在实时语

音上。国内腾讯的朋友网将 Node 应用在长链接中，以提供实时功能，花瓣网、蘑菇街等公司通过 socket.io 实现实时通知的功能。

c：并行 I/O 使得使用者可以更高效地利用分布式环境：阿里巴巴和 eBay 是这方面的典型。阿里巴巴的 NodeFox 和 eBay 的 q1.io 都是借用 Node 并行 I/O 的能力，更高效的使用已有的数据。

d：并行 I/O，有效利用稳定接口提升 Web 渲染能力：雪球财经和 LinkedIn 的移动版网站均是这种案例，撇弃同步等待式的顺序请求，大胆采用并行 I/O，加速数据的获取进而提升 Web 的渲染速度。

e：云计算平台提供 Node 支持：微软将 Node 引入 Azure 的开发中，阿里云、百度均纷纷在云服务器上提供 Node 应用托管服务，Joynet 更是云计算中提供 Node 支持的代表。这类平台看重 JavaScript 带来的开发上的优势，以及低资源占用、高性能的特点

f：游戏开发领域：游戏领域对实时和并发有很高的要求，网易开源了 pomelo 实时框架，可以应用在游戏和高实时应用中。

g：工具类应用：前端构建工具 GULP 之类

8.7 使用 superagent 与 cheerio 完成简单爬虫

superagent 是个 http 方面的库，可以发起 get 或 post 请求。用 superagent 去抓取 https://cnodejs.org 的内容。

cheerio 大家可以理解成一个 Node.js 版的 jquery，用来从网页中以 css selector 取数据，使用方式跟 jquery 一样一样的。

```
var express = require("express"),
    superagent = require("superagent"),
    cheerio = require("cheerio");

var app = express();

app.get("/",function (req,res,next) {
    //用 superagent 去抓取 https://cnodejs.org 的内容
    var crawUrl = "https://cnodejs.org";
    superagent.get(crawUrl)
        .end(function (err,sres) {
            // 常规的错误处理
            if(err){
                return next(err);
            }
            // sres.text 里面存储着网页的 html 内容，将它传给 cheerio.load 之后
            // 就可以得到一个实现了 jquery 接口的变量，我们习惯性地将它命名为 `$`
            // 剩下就都是 jquery 的内容了
            var $ = cheerio.load(sres.text);
            var items = [];
            $('#topic_list .topic_title').each(function (index,el
ement) {
                var $element = $(element);
```

```
                    items.push({
                        title: $element.attr("title"),
                        href: crawUrl + $element.attr("href")
                    })
                })

                res.send(items);
            })
    })

    app.listen(8080,function () {
        console.log("app is listening at port 8080");
    })
```

8.8 异步编程

8.8.1 异步编程的难点

难点 1：异常处理

异步方法的定义如下：

```
var async = function ( callback ) {
 process.nextTick( callback );
}
```

调用 async()方法后，callback 被存放起来，直到下一个事件循环(tick)才会取出来执行。尝试对异步方法进行 try/catch 操作只能捕获当次事件循环内的异常，对 callback 执行时抛出的异常将无能为力，示例代码如下：

```
try{
 async(callback);
}catch (e){
 // TODO
}
```

Node 在处理异常上形成了一种约定,将异常作为回调函数的第一个实参传回，如果为空值，则表示异步调用没有异常抛出：

```
Async( function (err,results) {
 // TODO
})
```

在我们自行编写的异步方法上，也需要去遵循这样一些原则：

原则一：必须执行调用者传入的回调函数；

原则二：正确传递回异常供调用者判断。

示例代码如下：

```
var async = function (callback) {
 process.nextTick( function() {
        var results = something;
        if (error) {
                return callback(error);
```

```
            }
                    callback(null, results);
    });
};
```

难点 2：函数嵌套过深
难点 3：阻塞代码
难点 4：多线程编程
难点 5：异步转同步

8.8.2 异步编程解决方案

目前，异步编程的主要解决方案有以下 3 种：

1) 事件发布/订阅模式

Node 自身提供的 events 模块是发布/订阅模式的一个简单实现，Node 中部分模块都继承自它。简单实例代码如下：

```
var events = require('events');
var emitter = new events.EventEmitter();
// 订阅/监听
emitter.on('someEvent', function( message ) {
 console.log( message );
});
// 发布/事件触发
emitter.emit('someEvent', 'arg1 参数');
```

2) Promise/Deferred 模式
3) 流程控制库

ANGULAR:

9.1 model 模型

ng-model 指令用于绑定应用程序数据到 HTML 控制器(input, select, textarea)的值。

1、可以将输入域的值与 AngularJS 创建的变量绑定。
2、可以为应用数据提供状态值(invalid, dirty, touched, error)
3、ng-model 指令基于它们的状态为 HTML 元素提供了 CSS 类

9.2 scope

AngularJS 应用组成如下：

View(视图), 即 HTML。
Model(模型), 当前视图中可用的数据。
Controller(控制器), 即 JavaScript 函数，可以添加或修改属性。

scope 是模型。

scope 是一个 JavaScript 对象，带有属性和方法，这些属性和方法可以在视图和控制器中使用。

根作用域

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。

9.3 控制器

ng-controller 指令定义了应用程序控制器。
控制器是 JavaScript 对象，由标准的 JavaScript 对象的构造函数创建。

9.4 过滤器

过滤器可以使用一个管道字符（|）添加到表达式和指令中

9.5 service

服务是作为一个参数传递到 controller 中。如果要使用它，需要在 controller 中定义。例如：$location,$timeout…

```
app.controller("myCtrl",function($scope,$location,$timeout) {
    $scope.myUrl = $location.absUrl();
    $timeout(function () {
            alert('2s');
    },2000);
})
```

补充: $watch 持续监听数据上的变化，更新界面

9.6 $Http

$http 是 AngularJS 中的一个核心服务，用于读取远程服务器的数据。
v1.5 中$http 的 success 和 error 方法已废弃。使用 then 方法替代。

```
app.controller("siteCtrl",function ($scope,$http) {
    $http({
        method: 'GET',
        url: 'json/data.json'
    }).then(function successCallback(response) {
        $scope.names = response.data.sites;
    }, function errorCallback(response) {
        // 请求失败执行代码
    });
    // 简写
    /*$http({
        method:'GET',
        url: 'json/data.json'
    }).then(function (response) {
        $scope.names = response.data.sites;
    })*/

    //1.5 以下版本写法
    $http.get("json/data.json").success(function (response) {
        $scope.names = response.sites;
    })
})
```

9.7 $emit、$broadcast、$on、$digest、$watch

$emit：向上广播冒泡，只能向 parent controller 传递 event 与 data
$broadcast ：向下广播，只能向 child controller 传递 event 与 data
$on：用于接收 event 与 data
$digest（数据绑定失效时用）：绝对不要在 controller 里用原生之类方法操作 dom 元素，最好通过指令、双向绑定之类实现，万一要用原生方法，最好在指令里面，因为用原生方法可能导致双向绑定失效，若失效则需要调用$digest 同步视图。
$watch：是一个 scope 函数，用于监听模型变化，当你的模型部分发生变化时它会通知你。太多的$watch 将会导致性能问题，$watch 如果不再使用，我们最好将其释放掉。

9.8 Select

可以使用数组或对象创建一个下拉列表选项。
ng-repeat 也可以实现，但是 ng-repeat 有局限性，选择的值是一个字符串。

```
<div ng-app="myApp" ng-controller="myCtrl">
    <select ng-init="selectName = names[0]" ng-model="sele
ctName" ng-options="x.site for x in sites">
```

```
        </select>
        <p>你选择的是{{selectName.site}}
           <span>网址为{{selectName.url}}</span>
        </p>

        <select ng-model="selectedRep">
              <option ng-repeat="x in sites">{{x.site}}</option>
        </select>
        <p>你选择的是{{ selectedRep }}</p>

    </div>
```

```
    <script type="text/javascript">
        var app = angular.module("myApp",[]);
        app.controller("myCtrl",function ($scope) {
            $scope.sites = [
                {site : "Google", url : "http://www.google.com"},
                {site : "Runoob", url : "http://www.runoob.com"},
                {site : "Taobao", url : "http://www.taobao.com"}
            ];
        })
    </script>
```

9.9 表格

$even $odd $index

9.10 HTML DOM

ng-disabled n-show n-hide

```
    <div ng-app="" ng-init="mySwitch=true;hour=13">
        <button ng-disabled="mySwitch">点我</button>
        <p><input type="checkbox" ng-model="mySwitch"></p>
        <p>{{mySwitch}}</p>

        <p ng-show="hour > 12">我是可见的</p>
    </div>
```

9.11 点击事件

ng-click 指令定义了 AngularJS 点击事件。

9.12 模块

模块定义了一个应用程序。
模块是应用程序中不同部分的容器。
模块是应用控制器的容器。
控制器通常属于一个模块。

可以通过 AngularJS 的 angular.module 函数来创建模块。

9.13 包含

可以使用 ng-include 指令来包含 HTML 内容。
ng-include 指令除了可以包含 HTML 文件外，还可以包含 AngularJS 代码，包含的文件中有 AngularJS 代码，它将被正常执行。

默认情况下， ng-include 指令不允许包含其他域名的文件。
如果你需要包含其他域名的文件，你需要设置域名访问白名单：

```
app.config(function($sceDelegateProvider){
    $sceDelegateProvider.resourceUrlWhitelist([ '白名单域名' ]);
});
```

此外，你还需要设置服务端允许跨域访问。

9.14 依赖注入

AngularJS 使用$injetor（注入器服务）来管理依赖关系的查询和实例化。 事实上，$injetor 负责实例化 AngularJS 中所有的组件，包括应用的模块、指令和控制器等。 在运行时，任何模块启动时$injetor 都会负责实例化，并将其需要的所有依赖传递进去

9.14.1 推断式注入

这种注入方式，需要在保证参数名称与服务名称相同。如果代码要经过压缩等操作，就会导致注入失败。

```
app.controller("myCtrl1", function($scope,hello1,hello2){
    $scope.hello = function(){
        hello1.hello();
     hello2.hello();
    }
});
```

9.14.2 标记式注入

这种注入方式，需要设置一个依赖数组，数组内是依赖的服务名字，在函数参数中，可以随意设置参数名称，但是必须保证顺序的一致性

```
Var myCtrl2 = function($scope,hello1,hello2){
    $scope.hello = function(){
        hello1.hello();
     hello2.hello();
    }
}
myCtrl2.$injector = ['hello1','hello2'];
app.controller("myCtrl2", myCtrl2);
```

9.14.3 内联式注入

这种注入方式直接传入两个参数，一个是名字，另一个是一个数组。这个数组的最后一个参数是真正的方法体，其他的都是依赖的目标，但是要保证与方法体的参数顺序一致（与标记注入一样）。

```
app.controller("myCtrl3",['$scope','hello1','hello2',function($scope,hello1,hello2){
    $scope.hello = function(){
        hello1.hello();
        hello2.hello();
    }
}]);
```

9.14.4 $injector 常用的方法

在 angular 中，可以通过 angular.injector()获得注入器。

```
var $injector = angular.injector();
```

通过$injector.get('serviceName')获得依赖的服务名字。

```
$injector.get('$scope');
```

通过$injector.annotate('xxx')获得 xxx 的所有依赖项。

```
$injector.annotate(xxx);
```

9.15 动画

1) **animate.css** ： https://daneden.github.io/animate.css/
2) **angular-animate** ： ng-enter、ng-leave

AngularJS 提供了动画效果，可以配合 CSS 使用。
AngularJS 使用动画需要引入 angular-animate.min.js 库。

```
<body ng-app="myApp">
    <!-- 需要配合 css 使用 -->
    <input type="checkBox" ng-model="myCheck">
    <div ng-hide="myCheck"></div>
</body>
```

9.16 路由

路由允许我们通过不同的 URL 访问不同的内容。通常我们的 URL 形式为 http://runoob.com/first/page，但在单页 Web 应用中 AngularJS 通过 # + 标记 实现当我们点击以上的任意一个链接时，向服务端请的地址都是一样的(http://runoob.com/)。 因为 # 号之后的内容在向服务端请求时会被浏览器忽略掉。 所以我们就需要在客户端实现 # 号后面内容的功能实现。 AngularJS 路由 就通过 # + 标记 帮助我们区分不同的逻辑页面并将不同的页面绑定到对应的控制器上。
例如：

```
<body ng-app="routingDemo">
    <ul>
            <li><a href="#/">首页</a></li>
            <li><a href="#/test1">测试页 1</a></li>
            <li><a href="#/test2">测试页 2</a></li>
            <li><a href="#/test3">测试页 3</a></li>
    </ul>
    <div ng-view></div>
    <script type="text/javascript">
        var app = angular.module("routingDemo",['ngRoute']);
       app.config(["$routeProvider",function ($routeProvider) {
               $routeProvider.when('/',{template:'这是首页'})
                   .when('/test1',{template:'这是测试页 1'})
                   .when('/test2',{template:'这是测试页 2'})
                   .when('/test3',{template:'这是测试页 3'})
                   .otherwise({redirectTo:'/'});
        }])
    </script>
</body>
```

1、载入了实现路由的 js 文件：angular-route.js。
2、包含了 ngRoute 模块作为主应用模块的依赖模块。
3、使用 ngView 指令，该 div 内的 HTML 内容会根据路由的变化而变化。
4、配置 $routeProvider，AngularJS $routeProvider 用来定义路由规则。

config 函数用于配置路由规则。通过使用 configAPI，我们请求把 $routeProvider 注入到我们的配置函数并且使用$routeProvider.whenAPI 来定义我们的路由规则。
$routeProvider 为我们提供了 when(path,object) & otherwise(object) 函数按顺序定义所有路由。
路由配置对象语法规则如下：

```
$routeProvider.when(url, {
    template: string, // ng-view 中插入简单的 HTML 内容
    //ng-view 中插入 HTML 模板文件,eg:templateUrl: 'views/computers.html'
    templateUrl: string,
    //在当前模板上执行的 controller 函数，生成新的 scope
    controller: string, function 或 array,
    controllerAs: string, // 为 controller 指定别名
    redirectTo: string, function, // 重定向的地址。
    resolve: object<key, function> //指定当前 controller 所依赖的其他模块。
});
```

9.17 小 Demo 心得

ng 特性：数据绑定、指令

Model:

```
var app = angular.module("myapp",[]);
```

Control:

```
app.controller("myCtrl",function ($scope) { //… })
```

View:

ng-model(针对表单)　ng-bind　ng-show/ng-hide

ng-repeat　ng-click ……

9.18 Angular 调试工具 Batarang

Batarang

主要功能：查看作用域、输出调试信息、性能监控

9.19 项目

9.19.1 起步(环境)

一、Angular 调试工具：batarang

二、第三方依赖管理工具：bower

用 bower 管理前端的包，二用 npm 去管理一些后端的包和构建工具，例如用 yeoman、grunt、gulp、jshint 等

三、css 预编译处理：less

四、自动化构建工具：gulp

9.19.2 规划

路由管理模块的配置和使用(UI-ROUTER)

内置指令的使用/自定义指令，编写组件（过滤器），promise 和 ajax 内置服务，第三方编写自定义服务，表单校验，AngularJS 装饰器修改 Ajax 服务

1. AngularJS 内置组件

   $q 服务：实现 promise 功能

   $http 服务：处理 ajax 请求

   ng-repeat、ng-model 指令

2. 复杂指令服务等组件

9.19.3 ui-router

路由 UI-ROUTER(http://www.jb51.net/article/78895.htm)总结扩展

调用方法 ui.router：http://jsrun.net/d5kKp/edit

路由参数：http://jsrun.net/F5kKp/edit

重要指令和服务：ui-sref、$state http://jsrun.net/J5kKp/edit

9.19.4 **Directive** 自定义指令

例如：自定义一个 app-position-list 指令
HTML

```
<article app-position-list></article>
```

PositionList.html

```
<ul class="position-list u-w100">
    <li class="item" ng-repeat="x in list">
            <img class="logo f-fl" ng-src="{{x.logoSrc}}"/>
            <h3 class="title fs36" ng-bind="x.name"></h3>
            <p class="text fs26" ng-bind="x.companyName+ '
[' + x.city + ']' + ' ' + x.industry"></p>
            <p class="text fs26" ng-bind="x.time"></p>
    </li>
</ul>
```

JS

```
angular.module("app").directive("appPositionList",[function () {
    return {
            restrict:'A',
            replace:true,
            templateUrl:'view/template/positionList.html'
    }
}])

angular.module("app").controller("mainCtrl",function ($scope) {
    $scope.list = [
            {
                    id:'1',
                    name:'测试 1',
                    companyName:'爱基金',
                    logoSrc:'images/company-1.jpg',
                    city:'杭州',
                    industry:'核新同花顺',
                    time:'2017-04-28 14:54'
            }
    ];
})
```

这样的话，如果 app-position-list 重复使用，数据也是一模一样的，那么有什么好的方法解决**自定义指令的复用**呢？

Direction 隔离 Scope 数据交互

参考文档：
https://blog.coding.net/blog/angularjs-directive-isolate-scope?type=early

HTML

```
    <article app-position-list data=”list1”></article>
    <article app-position-list data=”list2”></article>
```

PositionList.html

```
    <ul class="position-list u-w100">
        <li class="item" ng-repeat="x in data">
                <img class="logo f-fl" ng-src="{{x.logoSrc}}"/>
                <h3 class="title fs36" ng-bind="x.name"></h3>
                <p class="text fs26" ng-bind="x.companyName+ '
[' + x.city + ']' + ' ' + x.industry"></p>
                <p class="text fs26" ng-bind="x.time"></p>
        </li>
    </ul>
```

JS

```
    angular.module("app").directive("appPositionList",[function () {
        return {
                restrict:'A',
                replace:true,
                templateUrl:'view/template/positionList.html',
                scope:{
                        data: ‘=’
                }
        }
    }])

    angular.module("app").controller("mainCtrl",function ($scope) {
        $scope.list1 = [
                {
                        id:'1',
                        name:'测试 1',
                        companyName:'爱基金',
                        logoSrc:'images/company-1.jpg',
                        city:'杭州',
                        industry:'核新同花顺',
                        time:'2017-04-28 14:54'
                }
        ];
```

```
        $scope.list2 = [
                {
                        id:'1',
                        name:'测试 2',
                        companyName:'爱基金 2',
                        logoSrc:'images/company-2.jpg',
                        city:'杭州',
                        industry:'核新同花顺',
                        time:'2017-04-28 15:54'
                }
        ];
    })
```

补充：

Directive 在使用隔离 scope 的时候，提供了三种方法同隔离之外的地方交互。这三种方法分别是：

@ ：绑定一个局部 scope 属性到当前 dom 节点的属性值。结果始终是一个字符串，因为 dom 属性是字符串。

& ：提供一种方式执行一个表达式在父 scope 的上下文中。如果没有指定 attr 名称，则属性名位相同的本地名称。

= ：通过 directive 的 attr 属性的值在局部 scope 的属性和父 scope 属性名之间建立双向绑定。

9.19.6 项目代码

Demo: http://daceyu.com/static/ng/dist/index.html
Src: https://github.com/daceYu/ng

9.19.7 总结

创建唯一主页面 → 创建 app.js → 配置路由 → 控制器。

模块页面(例如 main.html)自定义指令 → 配置路由 → 控制器
→ 创建指令页面 → 自定义指令配置

LESS:

Less 是一门 CSS 预处理语言，它扩展了 CSS 语言，增加了变量、Mixin、函数等特性，使 CSS 更易维护和扩展。（定义变量、后代选择器、文件引用、函数）
Less 可以运行在 Node 或浏览器端。
例子：

```
@base: #f938ab;

.box-shadow(@style, @c) when (iscolor(@c)) {
    -webkit-box-shadow: @style @c;
    box-shadow: @style @c;
}
.box-shadow(@style, @alpha: 50%) when (isnumber(@alpha))
{
    .box-shadow(@style, rgba(0, 0, 0, @alpha));
}
.box {
    color: saturate(@base, 5%);
    border-color: lighten(@base, 30%);
    div {
            .box-shadow(0 0 5px, 30%)
    }
}
```

输出：

```
.box {
    color: #fe33ac;
    border-color: #fdcdea;
}
.box div {
    -webkit-box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
    box-shadow: 0 0 5px rgba(0, 0, 0, 0.3);
}
```