AngularJS开发指南33：单元测试

03-29 18:42 09-16 14:36 6617 1350 4 AngularJS AngularJS 开发指南

Javascript是动态类型语言，它的表达式有强大的威力，但也因此使得编译器对它没有任何帮助。所以我们强烈的觉得它需要强大的测试框架。我们已经建好了这样一个框架，所以没有理由不用它吧。

不要把问题都揽在一起

单元测试，如名字一样就是单独测试每个部分的代码。单元测试视图解决的问题是：我的逻辑正确了吗？排序函数正确排序了吗？为了解决这样的问题我们非常需要将它们鼓励出来。因为当我们解决排序问题的时候，我们不希望还要考虑到其他的如果DOM操作的问题，或者需要执行异步请求来获取数据之类的。但是在通常的项目中，几乎很难去调用一个单独的纯粹的函数。因为开发者常常将问题揽在一起，最后写出的代码职责太多。比如异步读取数据同时又要排序又要操作DOM。在AngularJS中，我们试图让开发者正容易地使用正确的开发方法，所以我们通过依赖注入来让你异步获取数据（你也可以模拟），我们创建了数据抽象让你操作模型数据的时候不用去管DOM。所以，最后你可以写出纯粹的排序函数了，并且能去测试它。测试不需要等待异步请求，也不需要去管理DOM或者验证DOM是否也正确反应。AngularJS是以可测试为宗旨之一写的，但是仍然需要你自己正确地操作。我们试图让事情变得简单，但是如果你不照章办事，你还是会把你的应用搞的完全没办法测。formatDate

依赖注入

有几种你可以获取依赖的方式：1.你直接使用new运算符建立实例。2.你查找已知的域，比如全局对象。3.你使用一个已有的注册系统来获取（但你如何来获取注册系统呢，貌似你还是必须查找一个已知域）。4.等待依赖被传递给你。

上面的选项只有最后是可测的。让我们看看为什么：

使用new操作符

new操作符本身来说并没有什么错误，但是使用它的地方会将自己和被调用者

new操作符本身来说并没有什么错误，但是待用它的地方会将自己和被调用者永久绑定。比如说，我们现在实例化一个XHR对象来取数据。

```
function MyClass() {
  this.doWork = function() {
    var xhr = new XHR();
    xhr.open(method, url, true);
    xhr.onreadystatechange = function() {...}
    xhr.send();
  }
}
```

问题来了，在测试中，我们很可能想要模拟一个XHR来返回虚拟的数据。但是调用 `new XHT()` 使得我们被绑定在了这个真实的对象上，别且没有很好地方法能替换它。呃，其实有一些方法，但是都很丑陋并且会导致其他问题，这个已经超出本章范围，暂不讨论。

上面的类很难用于测试，我们不得不使用一些歪招：

```
var oldXHR = XHR;
XHR = function MockXHR() {};
var myClass = new MyClass();
myClass.doWork();
// assert that MockXHR got called with the right arguments
XHR = oldXHR; // if you forget this bad things will happen
```

全局查找

另一种解决上述问题的方法是在一个已知域查找服务

```
function MyClass() {
  this.doWork = function() {
    global.xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
  }
}
```

虽然没有使用new来创建实例，但本质上和new是一样。在代码中还是没有办

法为了测试的需要而去拦截global.xhr的调用，除非又使用些怪招。这里面最基本的问题在于我们需要有一个方法来模拟依赖。更多这方面的知识请参阅 Brittle Global State & Singletons ：

上面的类难测试是因为我们需要改变全局状态：

```
var oldXHR = global.xhr;
global.xhr = function mockXHR() {};
var myClass = new MyClass();
myClass.doWork();
// assert that mockXHR got called with the right arguments
global.xhr = oldXHR; // if you forget this bad things will happen
```

服务注册

看起来好像我们可以用一个注册系统来解决这个问题，在需要时替换掉不需要测试的部分就行了。

```
function MyClass() {
  var serviceRegistry = ????;
  this.doWork = function() {
    var xhr = serviceRegistry.get('xhr');
    xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
}
```

但是，这个serviceRegistry是哪里来的呢？貌似又要new一个。而且在测试全局的时候，我们无法重置服务。

上面的例子难以测试是因为我们必须改变全局状态：

```
var oldServiceLocator = global.serviceLocator;
global.serviceLocator.set('xhr', function mockXHR() {});
var myClass = new MyClass();
myClass.doWork();
// assert that mockXHR got called with the right arguments
global.serviceLocator = oldServiceLocator; // if you forget this bad
things will happen
```

传递依赖

终于说到这个了。

```
function MyClass(xhr) {
  this.doWork = function() {
    xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
}
```

这是最好的一种方式，因为代码不关心xhr从哪里来，谁穿件来的谁负责实例化它。因为它的创建者和用它的地方肯定是不一样的，所以它在逻辑上分离了两者，简言之这就是依赖注入。

上面的例子是很好测的，我们可以这样写：

```
function xhrMock(args) {...}
var myClass = new MyClass(xhrMock);
myClass.doWork();
// assert that xhrMock got called with the right arguments
```

注意我们没有用到任何全局变量。

AngularJS已经内建好依赖注入系统来让事情比那的简单，但是你仍然需要按照规矩来，才能让你的应用更好测试。

控制器

应用的本质区别在于它的逻辑，这也正是我们希望测试的。如果你应用的逻辑里混杂着DOM操作，那就会很难测试，像下面这样：

```
function PasswordController() {
  // get references to DOM elements
  var msg = $('.ex1 span');
  var input = $('.ex1 input');
  var strength;

  this.grade = function() {
```

```
      msg.removeClass(strength);
      var pwd = input.val();
      password.text(pwd);
      if (pwd.length > 8) {
        strength = 'strong';
      } else if (pwd.length > 3) {
        strength = 'medium';
      } else {
        strength = 'weak';
      }
      msg
       .addClass(strength)
       .text(strength);
    }
  }
```

上面的代码难测在于你需要同时去测试DOM是否也正确反应了。你的测试可能会像下面这样：

```
var input = $('<input type="text"/>');
var span = $('<span>');
$('body').html('<div class="ex1">')
  .find('div')
    .append(input)
    .append(span);
var pc = new PasswordController();
input.val('abc');
pc.grade();
expect(span.text()).toEqual('weak');
$('body').html('');
```

在AngularJS控制器是和DOM操作完全分离的，这使得我们能像下面这样更好的测试：

```
function PasswordCntrl($scope) {
  $scope.password = '';
  $scope.grade = function() {
    var size = $scope.password.length;
    if (size > 8) {
      $scope.strength = 'strong';
    } else if (size > 3) {
      $scope.strength = 'medium';
    } else {
```

```
    } else {
      $scope.strength = 'weak';
    }
  };
}
```

测试会变得这样直白：

```
var pc = new PasswordController();
pc.password('abc');
pc.grade();
expect(span.strength).toEqual('weak');
```

注意我们的测试不只是变短了，而是更加清楚了。这样的测试代码告诉了你究竟是怎么运行的，而不是一堆看不懂的符号。

过滤器

过滤器是用来将输出给用户的数据变得更可读的。它们的重要性在于它们将格式化的工作从应用逻辑中抽离出来了，进一步的简化了应用逻辑。

```
myModule.filter('length', function() {
  return function(text){
    return (''+(text||'')).length;
  }
});

var length = $filter('length');
expect(length(null)).toEqual(0);
expect(length('abc')).toEqual(3);
```

指令

指令是在当模型数据改变时负责更新DOM的。

评论：AngularJS开发指南33：单元测试

您还没有登录，不能发表评论哦

[使用MarkDown语法，24 到 20480 字节，当前0字节]

编辑 / 预览 提交

评论：AngularJS开发指南33：单元测试

建议向百度百科一样，能有修改，然后管理员审核的功能，读起来太费力了，错别字，语句不通顺啊！

繁华落尽 09-16 14:36发表

1 0 0 回复

评论：AngularJS开发指南33：单元测试

靠，，没写完吧？

aijse 07-01 11:08发表

0 0 0 回复

评论：AngularJS开发指南33：单元测试

错别字真是太多了，，

aijse 07-01 11:05发表

0 0 0 回复

评论：AngularJS开发指南33：单元测试

> 为了解决这样的问题我们非常需要将它们鼓励出来。

应该是独立出来而非鼓励出来，方便时候修改下好些。：）

hanks 04-22 10:20发表

0 0 0 回复

作者信息

angularjs

关注

管理员 积分：**36033** 粉丝：**713** 关注：**1** 文章/评论：**150**

作者文章

03-09 00:12 thunk-redis：支持 Redis Cluster 的客户端（node.js）

11-24 23:29 thunks 的作用域和异常处理设计

08-17 11:08 记一次 OSX 下从源码安装 git——折腾~

08-03 00:24 网站用户登录系统设计——jsGen实现版

01-02 19:22 人人有奖中：牛头不对马嘴，七牛土豪大礼年度回馈！

05-24 21:01 基于jQuery UI Autocomplete Widget的ui-autocomplete

05-08 18:19 前端技术

05-06 17:45 JavaScript 的怪癖 1：两个「空值」——undefined 和 null

04-29 17:02 UnitedStack第一次公开招聘(招聘AngularJS经验高级前端)

04-17 18:07 JavaScript 的怪癖 1：隐式类型转换