689 words • 2~3 min read

为什么你的Angular代码很难测试

Angular推出有好几年的时候了, 跟其他的MV*框架相比, 它的双向绑定, 无须显式声明Model, 模块管理, 依赖注入等特点都给Web应用开发带来了极大的便利, 另外, 借助于它众多强大的原生directive, 我们几乎可以避免麻烦的DOM操作了, 除了这些, Angular还有一个很大的亮点, 那就是高度的可测试性.

今天的Web开发已经不同往日, 更多的交互与逻辑都需要在前端完成, 有时候, 前端的代码量甚至在后端之上. 怎么去保证如此多的前端逻辑不被破坏, 依赖于功能测试? 这显示不现实, 功能测试很耗时, 而且它的创建成本较高, 所以通常只用它来覆盖最基本的那部分逻辑, 另一方面, 功能测试是依赖于流程的, 如果你想验证购买页面上的某个前端逻辑, 那么你就不得不一路从产品详情页面老老实实点过来, 反馈时间太长了, 可能你要等一分多钟才知道某个功能出错了, 我们自然不想把宝贵的开发时间浪费在等待上.

我在过去一段比较长的时候里都在项目上使用Angular, 在感受到Angular带来的便利的同时, 也饱受了Angular测试的折磨, 因为我一直觉得Angular的单元测试很难写, 跟JUnit + Mockito比起来, Angular代码的单元测试真是感觉写起来不得心应手, 更别说用TDD的方式来驱动开发.

我一直在思考为什么Angular社区说Angular的测试性很高, 但是在项目上实现用起来却是另一番境地. 经过分析项目上的代码, 我觉得要想驱动测试开发Angular代码, 那么其实是对你的Angular代码提出了比较高的要求, 你要遵循Angular的风格来开发你的应用, 只有你了解了其中的思想, 你的测试写起来才会轻松.

如果你已经使用Angular有一段时间了, 但是还没有读过这篇文章, 那么我强烈推荐你去读一下: Thinking in Angular

先来看一看怎么样的Angular代码才是苗正根红的Angular代码.

避免使用任何的DOM操作

像DOM操作这样的脏活累活都应该交给Angular的原生directive去做, 我们的Angular代码应该只处理与DOM无关的业务逻辑. 来看一个简单的例子, 我们想创建一个简单的邮箱地址验证的directive, 它要实现的功能是, 当焦点从邮箱地址输入框移出的时候, 对输入框中的邮箱地址进行验证, 如果验证失败, 则向输入框添加一个样式表示输入的地址不合法, 比较糟糕的实现可能是这样的

```
<label for="email">Your email:</label>
<input id="email" type="text" ng-model="email" valid
ate-on-blur/>

angular.module("TheNG", [])
  .directive("validateOnBlur", function() {
    return {
      link: function(scope, element, attrs) {
        var validate = function(email) {
          return email.indexOf('@gmail.com') !== -1;
        };
        element.on('blur', function() {
          if (!validate(scope.email)) {
            element.addClass('error - box');
          }
        });
      }
   };
});
```

上面的代码应该可以满足我们的要求(验证逻辑因为不是我们关注的重点, 所以并不完善), 而且这个directive实现起来也挺简单的, 但是现在让我们一起来分析一下为什么我们认为这种写法是比较糟糕的.

最简单的办法就是在你的directive里面去找所有与DOM操作相关的代码.

首先看到的就是on()这个事件监听器. 完全没有必要自己去监听发生在被directive修饰的元素上的事件, angular有一整套的原生directive来干这个事情, 这里正确的做法应该是使用ng-blur来处理blur事件.

下一个有问题的地方就是addClass(), angular除了提供了事件监听相关的directive外, 也提供了操作元素本身属性的directive, ng-class就可以用来替换addClass()方法.

按照这个思路修改后的代码:

```
<label for="email">Your email:</label>
<input id="email" type="text" ng-model="email" ng-bl
ur="validate()" ng-class="{'error-box': !isValid}" v
alidate-on-blur/>

angular.module("TheNG", [])
  .directive("validateOnBlur", function () {
    return {
      link: function (scope, element, attrs) {
        scope.isValid = true;
        scope.validate = function () {
        scope.isValid = scope.email.indexOf('@gmail.
com') !== -1;
      };
    }
   };
  }
);
```

比较一下这两个版本的实现, 是不是修改后的版本更简短, 更容易理解一些. 在新的版本里面, 我们只处理了业务逻辑, 即判断一个邮箱地址是否合法, 至于何时触发验证, 验证失败或成功之后应该有怎样的样式, 我们都统统交给了angular原生directive去处理了.

从测试的角度来看, 如果想给第一个版本的实现写单元测试, 那么要准备和验证的东西都很多, 我们需要设法去触发对应元素的blur事件, 然后再验证这个元素上是否添加了error-box这个class, 根据我的经验, 有时候为了验证这些DOM更新, 你还不得不创建真实的DOM结构添加到DOM tree上去, 又增加了一部分工作量.

而版本二就简单多了, 只定义了一个Model值isValid来标识当前的邮箱地址是否合法, validate()方法会在每次失焦之后自动执行, 要为它添加单元测试, 则只需要调用一下它的validate()方法, 然后验证isValid的值就可以了. SO EASY!~

将所有第三方服务封装成Service

一个Web项目中总是无法避免地要使用一些第三方的服务, 这里讨论的主要是前端的一些第三方服务, 比如在线客服, 站点统计等, 这些代码都在我们的控制之外, 大多数时候下都是从服务提供商的服务器上下载下来的, 而我们需要在业务代码中调用这些代码.如果我们每次都是赤裸裸地以全局变量的形式来使用这些服务, 那么造成的问题就是这样的代码很难测试, 因为这些代码是不存在于我们的代码库中的, 而且内容应该也是不定时更新的, 大多数情况很多人会因为这些原因放弃到对这类操作的测试.假设我们现在需要在某些动作发生之后调用一个第三方服务, 这个第三方服务叫做serviceLoadedFromExternal, 它提供了一个API叫做makeServiceCall, 如果直接使用这个API, 那么在测试中很难去验证这个服务被执行了(因为在单元测试环境中这个服务根本不存在), 但是如果我们将这个服务包装成一个angular service, 那么就可以在测试中轻易地将它替换成一个mock对象, 然后验证这个mock对象上的方法被调用了就可以了.比较下面的两段代码:

- 直接使用第三方服务

```
angular.module(“TheNG”, [])
  .controller(“AController”, function ($scope) {
    $scope.someAction = function () {
    // handle some logic here
    serviceLoadedFromExternal.makeServiceCall();
   };
});
```

- 使用封装成service的第三方服务

```
angular.module(“TheNG”, [])
  .factory(“wrappedService”, function () {
    return {
      makeServiceCall: function () {
        serviceLoadedFromExternal.makeServiceCall();
      }
    };
 })
 .controller(“AController”, function ($scope, wrappe
dService) {
   $scope.someAction = function () {
   // handle some logic here
   wrappedService.makeServiceCall();
```

```
    };
  });
```

Angular是高度模块化的, 它希望通过这种模块的形式来解决JS代码管理上的混乱, 并且使用依赖注入来自动装配, 这一点与Spring IOC很像, 带来的好处就是你的依赖是可以随意替换的, 这就极大的增加了代码的可测试性.

尽量将Ajax请求放到service中去做

Angular中使用service来组织那些可被复用的逻辑, 除此之外, 我们也可以将service理解为是对应一个领域对象的操作的集合, 因此, 通常会将一组Ajax操作放在一个service中去统一管理.

当然了,你也可以通过向你的directive或是controller中注入$http, 但是我个人不喜欢这种做法. 首先, $http是一个比较初级的依赖, 与其实注入的业务服务不是一个抽象层级, 如果在你的业务代码中直接操作http请求, 给人的一种感觉就像是在Spring MVC的request method中直接使用HttpServeletRequest一样, 有点突兀, 另外会让整个方法失衡, 因为这些操作的抽象层次是不一样的. 其次就是给测试带来的麻烦, 我们不得不使用$httpBackend来模拟一个HTTP请求的发送.

我们应该设法让测试更简单,通过将Ajax请求封装到service中, 我们只需要让被mock的service返回我们期望的结果就可以了. 只有这样大家才会喜欢写测试, 甚至是做到测试驱动开发, 要去mock$http这样的东西, 显然是增加了测试的负担.

使用Promise处理Ajax的返回值, 而不是传递回调函数

Angular中所有的Ajax请求默认都返回一个Promise对象, 不建议将处理Ajax返回值的逻辑通过回调函数的形式传递给发送http请求的service, 而应该是在调用service的地方利用返回的promise对象来决定如何处理.

让我们通过下面的例子来感受一下:

```
angular.module("TheNG", [])
  .service("deliveryService", function ($http) {
    this.validateAddress = function (address, succes
```

```
s, failure) {
      $http.post('/unknown', address).then(success,
failure);
    };
  })
  .controller("DeliveryController", function ($scope
, deliveryService) {
     var acceptAddress = function () {
       console.log('address accepted');
     };
     var rejectAddress = function () {
       console.log('address rejected');
     };
     $scope.validateAddress = function () {
     deliveryService.validateAddress($scope.address,
 acceptAddress, rejectAddress);
   };
});
```

这里的处理办法是将快递地址验证失败或成功之后的处理函数都传给了deliveryService, 当验证结果从服务器端返回之后, 相应的处理函数会被执行. 这做写法其实是比较常见的, 但是问题出在哪里呢?

其实, 作为一个service的接口, validateAddress应该只接收一个待验证的地址, 验证完成之后返回一个验证结果就可以了, 本来应该是一个很干净的接口, 我们之所以丑陋把对应的处理函数也传进去, 原因就在于这是一个异步的请求, 所以需要在发请求的时候就将对处理函数绑定上去.

你应该已经猜到了第二个问题我会说一说对它的测试, 通常来说, 如果一个service创建成本较高或是存在外部依赖/请求的话, 我们会将这个service mock掉, 通过让mocked service直接返回我们想要的结果来让我们只关注被验证的业务逻辑.我们回头看一下上面的实现, 如果我们把deliveryService的validateAddress()方法mock掉, 那么我们根本没有办法去验证acceptAddress()和rejectAddress()里面的逻辑!

所以, 如果你的处理函数是传递给service中的API的话, 那么你的测试其实就已经跟这个API的实现绑定了, 你只有去创建一个真实的service并且让它发送HTTP请求, 你的处理函数才会被执行到.

经过这一番折腾, 你一定要说, 这测试比实现代码难写多了. 正确的打开方式应该是这样的: service的API只需要返回promise, 对应的处理函数的绑定

在这个返回的promise上, 这样我们只需要mock那个service的接口让它返回一个我们期望的promise, 然后控制promise的结果让对应的处理函数被执行:

```
angular.module("TheNG", [])
  .service("deliveryService", function ($http) {
    this.validateAddress = function (address) {
      return $http.post('/unknown', address);
    };
  })
  .controller("DeliveryController", function ($scope
, deliveryService) {
     var acceptAddress = function () {
       console.log('address accepted');
     };
     var rejectAddress = function () {
       console.log('address rejected');
     };
     $scope.validateAddress = function () {
       deliveryService.validateAddress($scope.addres
s).then(acceptAddress, rejectAddress);
     };
 });
```

本来打算接下来介绍一下Angular代码的单元测试的各种模式的, 写着写着篇幅有点多了, 期待下一篇吧.

软件测试 | Angular, 冯尔东 | 2015年7月16日 · pm2:28