## 微服务环境下的集成测试探索（一）——服务Stub&Mock

## 引子

现在微服务很流行，企业架构微服务化的确能解决不少问题，但是在微服务环境下，服务之间的依赖以及由此造成的开发、测试和集成的问题，一直都是微服务最大的痛点。

传统的解决方案是，除了测试、预发布和生产环境，还会部署多套用于开发和集成的环境。这样存在的问题是，只要有一组服务出现问题，就会影响其他使用该环境的团队的日常开发和测试。而且常常出现问题后，需要耗费很多时间定位，结果还常常是因为某个服务的版本没有同步。并且多套环境维护起来也是一个麻烦重重，即使有了容器。

这次我们一起来探索一下 API 模拟工具以及基于契约的测试，也许会是解决这个问题的一个方案。

## WireMock 介绍

我们开发应用也好、服务也好，常常需要依赖后端或者服务的接口。例如开发移动应用 App，可能后端接口还在开发中，这时 App 的开发因为无法调用后端，很不方便。又或者程序会依赖第三方的接口，例如微信支付，在本地开发时不能直接调用。

这时我们就会需要一个工具来模拟这些服务，WireMock 就是这样的一个工具，主要针对的是最常见的 HTTP 服务。

### WireMock 用于开发调试

WireMock 首先自身就是一个可以独立运行的服务。下载 Standalone Jar 文件后，即可可以直接运行。

java -jar wiremock-standalone-2.11.0.jar

此时可以通过 Json 映射文件来定义 Stub 服务。例如下面是一个映射文件，request 部分设置匹配的 Url 路径、请求方法及参数，如果匹配到了，则会返回 response 部分设置的内容。把该文件放到 WireMock 同路径下的 mappings目录下即可。

{  
  "request" : {

"urlPath" : "/api/order/find",

"method" : "GET",

"queryParameters" : {

"orderId" : {

"matches" : "^[0-9]{16}$"

}

}

},

"response" : {

"status" : 200,

"bodyFileName" : "body-order-find-1.json",

"headers" : {

"Content-Type" : "application/json;charset=UTF-8"

}

}  
}

Response 的内容可以直接在映射文件里设置，也可以引用了另一个文件。这里是引用了一个名为 body-order-find-1.json 的文件，该文件放置在 WireMock 同路径下的 \_\_files 目录下。

{  
    "success": **true**,

"data": {

"id": 781202,

"buyerId": -2,

"status": 0,

// 略...

}  
}

下面我们用 curl 测试一下。第一次我们请求的参数 orderId 无法匹配指定的正则，WireMock 会返回 Request was not matched，而且还会很贴心的告诉你最接近的匹配是什么。

$ curl http://localhost:8080/api/order/find?orderId=abcdefghijklmnop

          Request was not matched

          =======================

----------------------------------------------

| Closest stub         | Request

----------------------------------------------

GET                    | GET

/api/order/find        | /api/order/find

----------------------------------------------

第二次我们参数 orderId 匹配的话，WireMock 会直接返回设置的结果。

$ curl http://localhost:8080/api/order/find?orderId=9999999999999999

{

   "success": true,

   "data": {

       "id": 781202,

       "buyerId": -2,

       "status": 0

   }

}

上面的例子是 WireMock 最基本的用法，除了请求匹配响应，WireMock 也能支持：

通过 RESTFul 的接口提交和管理请求映射和相应。

支持响应模板，返回内容时会将变量填充到响应模板中。当然，这里的模板功能是比较简单的，但对于大部分 Stub 的场景应该是足够了。

支持模拟异常返回，例如设置有一定比例的超时返回等等，这个功能用于测试非常方便。

为了方便编写请求映射文件，WireMock 还可以运行在代理模式，只需要运行时添加 --enable-browser-proxying 参数即可。此时 WireMock 匹配到请求后，不是返回指定的内容，而是把请求 Forword 到指定的 URL，获得 Response 后再返回给调用方。同时，WireMock 会记录请求和返回的内容，生成 Json 映射文件。使用时只要根据需求对这些映射文件做一定修改，既可以用来模拟目标服务。

### WireMock 用于集成测试

除了独立运行，WireMock 也可以直接嵌入到代码中。最方便的就是在 JUnit 中使用，WireMock 提供了 WireMockRule， 可以很方便的在测试时嵌入一个 Stub 服务。

下面是一个支付相关的集成测试，被测试方法会调用微信的支付服务。stubForUnifiedOrderSuccess 设置了一个很简单的 Stub，一旦匹配到请求的 URL 为 /pay/unifiedorder，那就返回指定的 XML 内容。这样我就可以在集成测试里测试整个支付流程，而不必依赖真正的微信支付。当然，测试时微信支付接口的 Host 也要改成 WireMockRule 配置的本地端口。并且，通过这种方式也很容易测试一些异常情况，根据需要修改 Stub 返回的内容即可。

**public** **class** **OrderTest** {  
    @Rule

**public** WireMockRule wireMockRule = **new** WireMockRule(9090);

/\*\*       
     \* 统一下单 Stub       
     \* 参考 https://pay.weixin.qq.com/wiki/doc/api/jsapi.php?chapter=9\_1       
     \*       
     \* @param tradeType 交易类型, 可以是JSAPI、NATIVE或APP       
     \*/

**public** **void** stubForUnifiedOrderSuccess(String tradeType) {

String unifiedOrderResp = "<xml>\n" +

"    <return\_code><![CDATA[SUCCESS]]></return\_code>\n" +

"    <return\_msg><![CDATA[OK]]></return\_msg>\n" +

"    <appid><![CDATA[wxxxxxxxxxxxxxxxxx]]></appid>\n" +

"    <mch\_id><![CDATA[9999999999]]></mch\_id>\n" +

"    ...... \n" +

"    <trade\_type><![CDATA[" + tradeType + "]]></trade\_type>\n" +

"</xml>";

stubFor(post(urlEqualTo("/pay/unifiedorder"))

.withHeader("Content-Type", equalTo("text/xml;charset=UTF-8"))

.willReturn(aResponse()

.withStatus(200)

.withHeader("Content-Type", "text/plain")

.withBody(unifiedOrderResp)));

}

@Test

**public** **void** test001\_doPay() {

stubForUnifiedOrderSuccess("JSAPI");

payServices.pay();

// 测试代码...

}  
}

有时候在集成测试里，我们还需要验证系统的行为，例如是否调用了某个 API，调用了几次，调用的参数和内容是否符合要求等。区别于前面说的 Stub，其实这就是常说的 Mock 功能。WireMock 对此也有很强大的支持。

verify(postRequestedFor(urlEqualTo("/pay/unifiedorder"))  
        .withHeader("Content-Type", equalTo("text/xml;charset=UTF-8"))  
 .withQueryParam("param", equalTo("param1"))

.withRequestBody(containing("success"));

这样，有了 WireMock，集成测试时处理第三方的依赖就非常方便了。不需要直接调用依赖的服务，也不需要专门创建用于集成测试的 Stub 或 Mock，直接代码中根据需要设置即可。

### WireMock 总结

**总结一下， WireMock 可以：**

●作为代理运行，此时可以录制请求和返回的脚本，用于后继 Stub 和 Mock 使用。

●独立运行，作为一个 Stub 服务，根据匹配的请求返回数据。

●作为 Stub，通过代码嵌入 HTTP 模拟服务，在指定端口监听，并根据匹配的请求返回数据。

●作为 Mock，在单元测试和集成测试中，验证请求逻辑。例如是否进行了调用、参数是否正确等。

这里再强调下 Stub 和 Mock 的区别，很多人经常搞混。Stub 就是一个纯粹的模拟服务，用于替代真实的服务，收到请求返回指定结果，不会记录任何信息。Mock 则更进一步，还会记录调用行为，可以根据行为来验证系统的正确性。

我们可以用 WireMock 来优化开发和集成的流程。

在外部服务尚未开发完成时，模拟服务，方便开发。

在本地开发时，模拟外部服务避免直接依赖。

在单元测试中模拟外部服务，同时验证业务逻辑。

## 契约式测试

本文主要以 WireMock 为例介绍了 API 模拟工具的使用方法。其实除了 WireMock，还有不少类似的工具，例如最早的 MounteBank，以及 MockServer、Moco 等也都是很强大的工具。

不过，在微服务环境下，光有 API 模拟工具还不够。对于 WireMock，首先必须考虑如何来管理大量的映射文件。一个方法是开发一个专用的 Stub 平台，来管理所有的映射文件，同时作为 Stub 运行。另外一个方法是通过 Git 来管理映射文件，需要的时候同步下来运行 WireMock 即可。

另外，我们上面提到 WireMock 的两大作用，调用方模拟服务以及服务方集成测试，是否可以统一两者呢？也就是说，调用方和服务方约定好接口，生成映射文件，这个文件即可以用于客户端模拟服务，也可以用于服务方集成测试，这样双方开发也好、集成也好都会方便很多。下一篇我们来研究一下 Spring Cloud Contract，它就是基于 WireMock 实现了契约式的测试，上文中双方约定好的接口，其实就是双方的契约。