微服务的集成测试，怎么做才高效？

真实的故事

某天，项目组测试mm对大家抱怨道，“都说微服务好，我咋发现自从咱产品微服务化后，忙的一塌糊涂……”。

“集成测试又挂了！API team，你们提供的接口变了吗？晕死，改接口提前说一声啊~……”

“老大，集成测试环境出问题了，新版本的服务部署不上去，做不了服务间的集成测试，得找人先修环境……”

……

不知大家在微服务化的演进中，有没有遇到如上的场景，可能不是你，但你身边一定有这样的影子。

今天这篇文章，我就和大家聊聊，“微服务架构下，如何有效执行集成测试?”

传统集成测试之殇

传统的集成测试是将不同的单元/部件按照业务组合，对其间的协作进行正确性检验的工作。通常，集成测试发生在单元测试之后，端到端测试之前。由于集成测试阶段开始较晚，团队会花费几周甚至数月来验证各个部件之间的协作，发现问题和缺陷修复的成本很高。

对于集成测试产生的收益，社区褒贬不一。J.B. Rainsberge博士在其文Integrated tests are scam中列举了集成测试的弊端：

> “Integrated tests are scam - a self-replicating virus that threatens to infect your code base, your project, and your team with endless pain and suffering.”
>
> 集成测试像自我扩散的病毒，无情的威胁着代码库、项目和团队”。

随着系统分支以及系统间依赖变得愈发复杂，集成测试导致的测试成本高、环境搭建耗时等问题也愈发明显。

当集成测试遇上微服务

微服务架构是近几年受到社区热捧的一种应用架构模式，其核心是基于业务上下文，构建细粒度、可独立部署的单元。Martin Fowler在其博文microservices中对微服务给出了通俗易懂的解释:

> 微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相协作（通常是基于HTTP协议的RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。

既然每个服务都可以被独立部署，那就意味着，随着服务数量及调用关系复杂度的增加，如果依然使用传统的集成测试方式对服务协作进行验证，必然会面临成本呈指数级增长的挑战，具体体现在:

- 验证成本高

为了验证多个服务协作后的功能正确与否，需要为每个服务搭建基础设施(包括其依赖的数据库、缓存等)，并执行部署、配置等步骤，以确保服务能正确运行。

- 结果不稳定

微服务构建的系统本质上是分布式系统，服务间通信通常都是跨网络调用的。当对服务间协作进行测试时，网络延迟、超时、带宽等因素都会影响到测试结果，极易导致结果不稳定。

- 反馈周期长

相比于传统的单体应用，微服务架构下的可独立部署单元多，因此集成测试的反馈周期比传统的方式更长，定位问题所花费的时间也更长。

因此，如何提升微服务中集成测试的有效性，成了服务规模化后必须要面对的挑战。

什么是契约测试

契约，是服务消费者(Consumer)与提供者(Provider)之间协作的规约。契约通常包括：

- **请求**

  指消费者发出的请求。包括请求头(Header)、请求内容(URI、Path、HTTP Verb)以及请求参数等。

- **响应**

指提供者应答的响应。可能包括响应的状态码、响应体的内容(XML/JSON)或者错误的信息描述等。

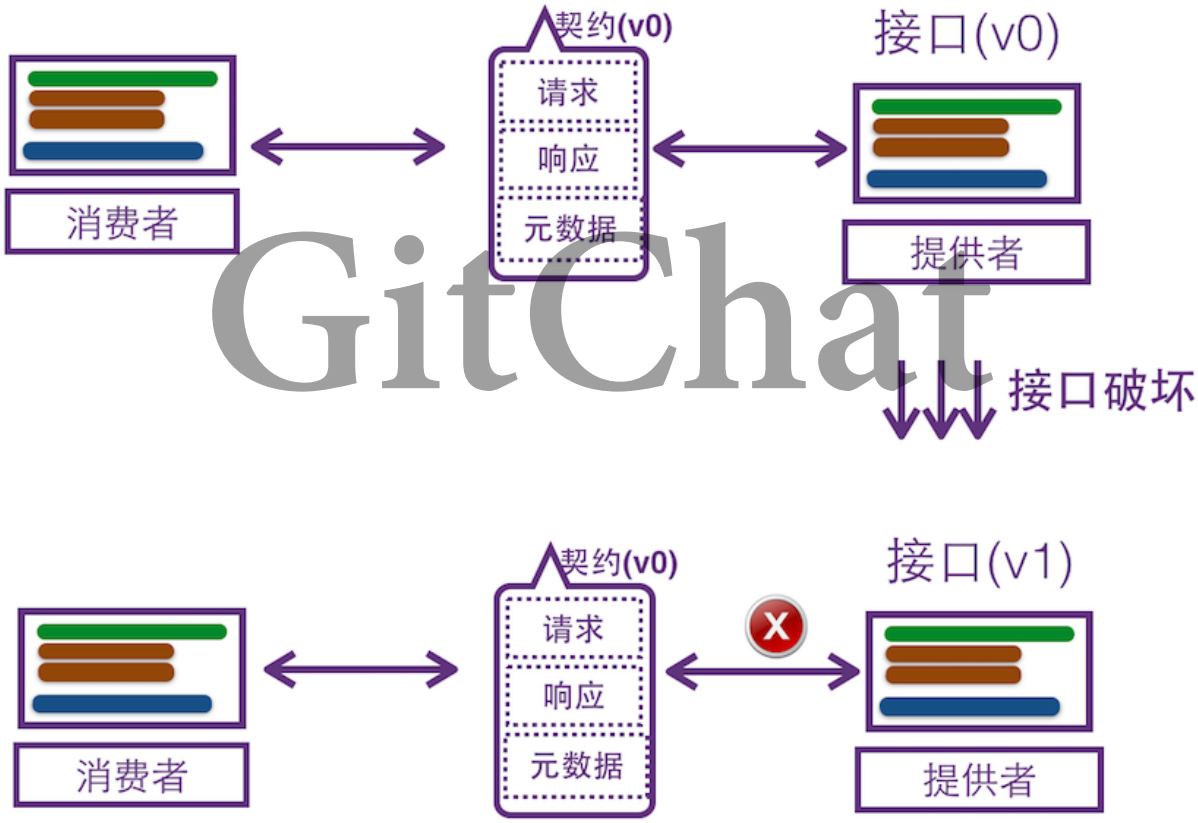
- **元数据**

  指对消费者与提供者间一次协作过程的描述。譬如消费者/提供者的名称、上下文及场景描述等。

契约测试，是基于契约，对消费者与提供者间协作的验证。

通过契约测试，团队能以一种离线的方式(不需要消费者、提供者同时在线)，通过契约作为中间的标准，验证提供者提供的内容是否满足消费者的期望。

如下图所示，当消费者与提供者之间建立契约(v0)后，如果提供者提供的内容被意外修改(譬如从v0变化成v1)，则提供者的v1版本显然无法满足之前定义的契约(v0)，因此就能够及时发现提供者接口变化导致的错误，并对其进行修正。



契约测试的出现，能够将集成测试转换成离线的、隔离的单元测试。

什么是消费者驱动的契约测试

消费者驱动的契约测试(Consumer-Driven Contracts，简称CDC)，是指从消费者业务实现的角度出发，驱动出契约，再基于契约，对提供者验证的一种测试方式。

CDC的核心流程包括如下两步：

1. 对消费者的业务逻辑进行验证时，先对其期望的响应做Mock；并将请求(消费者)-响应(基于Mock)的协作过程记录为契约；

2. 通过契约，对提供者进行回放，保证提供者提供的内容满足消费者的期望。

实际上，CDC是典型的Outside In在架构层面的延伸，和多年前XP提出的TDD(测试驱动开发)、BDD(行为驱动开发)的思路如出一辙。它的本质是从利益相关者的目标和动机出发，最大限度地满足需求方的业务价值实现。

通过这种机制，当我们在消费者侧构建出契约后，就能基于契约验证提供者，尽早捕获协作中出现的问题。

业界常用的CDC测试框架有：

- Janus
- Pact
- Pacto
- Spring Cloud Contract

关于这些框架的比较，有兴趣的朋友可以查看相关网站。

使用Pact实现消费者驱动的契约测试

Pact是实现CDC的框架之一，起源于REA(澳洲一家房产互联网门户)在微服务演进过程中所面临的服务间测试方面的挑战。Pact主要支持服务间REST接口的验证，经过几年的发展，Pact已经提供了Ruby、JVM/Scala、JS、Swift等多个版本。

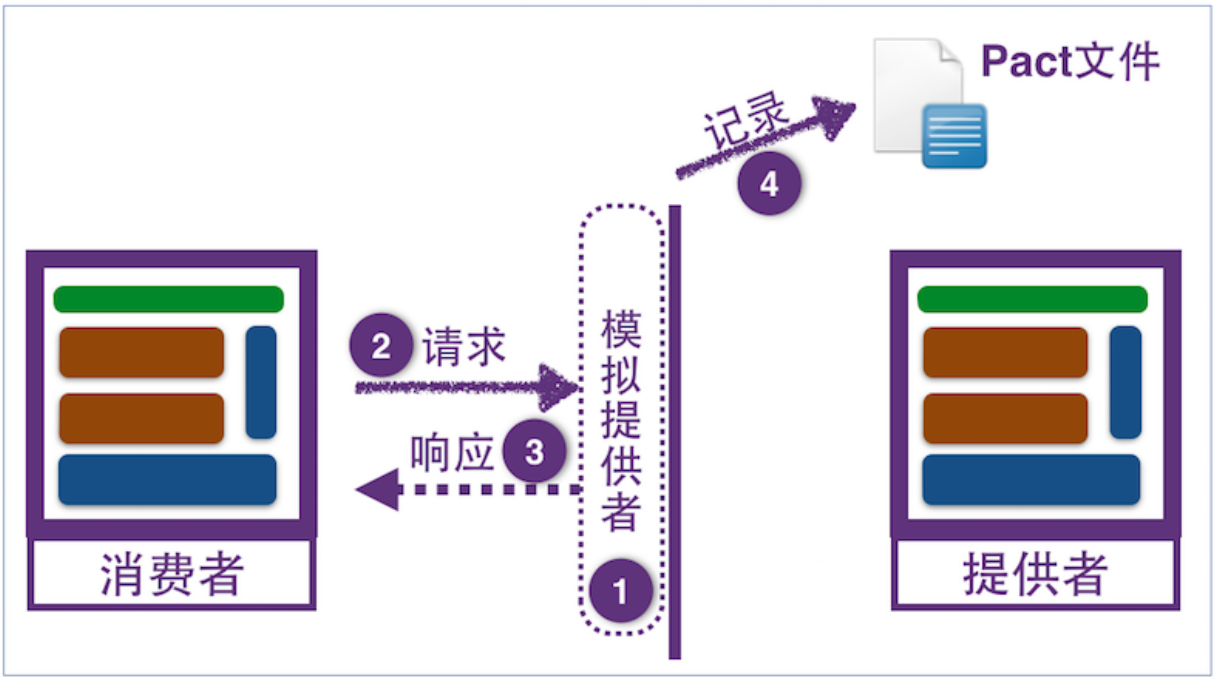
最近几年，随着微服务的快速发展，很多公司开始使用Pact，构建微服务的集成测试体系，像SoundCloud、Redhat、Pivotal Labs、ThoughtWorks等。

在我经历的几个微服务项目中，都曾使用Pact作为服务间协作的验证机制，屡试不爽。

Pact遵循CDC的主要流程，其验证过程分为如下两部分：

基于消费者的业务逻辑，驱动出契约

其实现步骤如下所示：



消费者端生成契约

1. 使用Pact的DSL，Mock提供者。
2. 消费者对Mock的提供者发送请求。
3. 使用Pact的DSL，定义响应(包括Headers、Status以及Body等)。
4. 使用@PactVerification运行单元测试(Pact集成了JUnit、RSpec等框架)。

代码如下图所示：

1. 定义Mock的提供者localhost:8080。
2. 将Mock地址传给消费者。
3. 使用DSL，定义响应内容。
4. 在消费者端运行单元测试，生成契约。

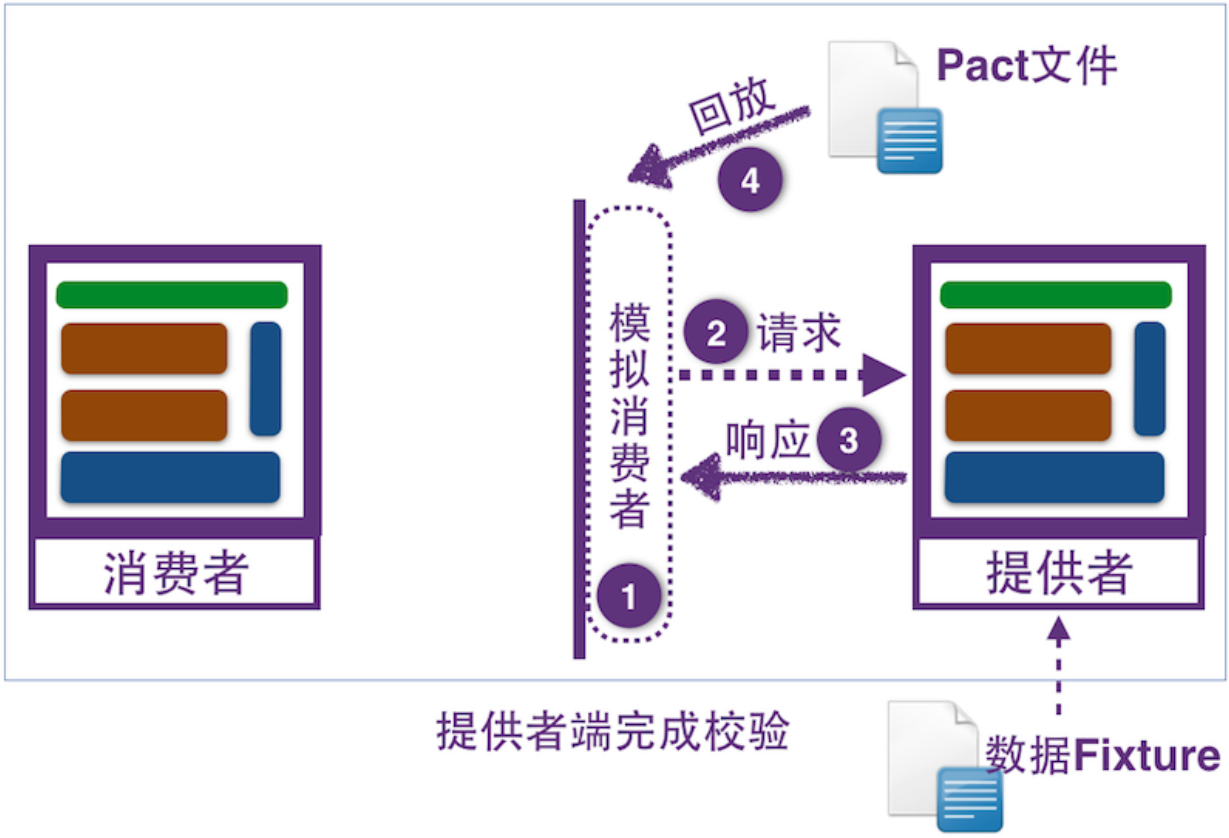
```
(1) @Rule
    public PactRule rule = new PactRule("localhost", 8080, this);

(2) OrderService orderService = new OrderService("http://localhost:8080");

    private static Map<String, String> getHTTPHeaders(){...}

(3) @Pact(state="WhenOneOrderExists", provider="orderProvider", consumer="orderConsumer")
    public PactFragment createOrderFragment(ConsumerPactBuilder.PactDslWithProvider.PactDslWithState builder) {

        PactDslJsonBody body = new PactDslJsonBody()
                .integerType("id",36)
                .stringValue("title","The order is created with ID [36]")
                .asBody();

        return builder.uponReceiving("a request to get one order with id 36")
                .path("/orders/36")
                .method("GET")
                .willRespondWith()
                .headers(getHTTPHeaders())
                .status(200)
                .body(body).toFragment();
    }

    @Test
(4) @PactVerification("WhenOneOrderExists")
    public void testOneOrderExist() {
        assertEquals(orderService.getOrder(36), new Order(36));
    }
```

当运行测试后，Pact框架记录消费者的名称、发送的请求、期望的响应以及元数据，将其保存为当前场景下的契约文件，通常命名为[Consumer]-[Provider].json，本例为orderConsumer-orderProvider.json，内容如下所示：


```
{
  "provider" : {
    "name" : "orderProvider"
  },
  "consumer" : {
    "name" : "orderConsumer"
  },
  "interactions" : [ {
    "providerState" : "WhenOneOrderExists",
    "description" : "a request to get one order with id 24",
    "request" : {
      "method" : "GET",
      "path" : "/orders/24"
    },
    "response" : {
      "status" : 200,
      "headers" : {
        "Content-Type" : "application/json;charset=UTF-8"
      },
      "body" : {
        "id" : 24,
        "title" : "The order is created with ID [24]"
      },
      "responseMatchingRules" : {
        "$.body.id" : {
          "match" : "integer"
        }
      }
    }
  }],
  "metadata" : {
    "pact-specification" : {
      "version" : "2.0.0"
    },
    "pact-jvm" : {
      "version" : "2.1.13"
    }
  }
}
```

到此，契约就生成了。我们可以将其保存在文件系统或者Pact-Broker(Pact提供的中间件，用来管理契约文件)中，以便后续提供者使用。

基于消费者驱动出的契约，对提供者进行验证



在提供者端，我们不需要写任何验证的相关代码，Pact已经提供了验证的接口，我们只需要做好如下配置：

- 为提供者指定契约文件的存储源(如文件系统或者Pact-Broker)。
- 启动提供者。
- 运行PactVerify(Pact有Maven、Gradle或者Rake插件，提供pactVerify命令)。

当执行pactVerify时，Pact将按照如下步骤，**自动**完成对提供者的验证：

1. 构建Mock的消费者。
2. 根据契约文件记录的请求内容，向提供者发送请求。
3. 从提供者获取响应结果。
4. 验证提供者的响应结果与Pact契约文件定义的契约中是否一致。

验证完成后，结果类似如下图所示：

同时，运行的结果也将显示消费者测试时设置好的上下文。

在本例中，即

> “WhenOrderExists”

可以看出，基于Pact框架，能低成本的帮助团队验证服务间接口的协作，其优势主要体现在：

1. 通过消费者驱动出契约，能有效保障提供者提供的响应一直是满足利益相关者的目标和动机的。
2. 将传统的集成测试的在线验证场景(要求消费者、提供者同时在线，都运行在测试环境中)，转变成离线的隔离场景(消费者、提供者不需要同时在线，可以分别验证)。
3. 借助Pact与其JUnit/RSpec等框架的整合，能使用单元测试，而非集成测试的方式验证服务间协作，极大的缩短了反馈周期，提升了测试效率。

我们的Pact实践

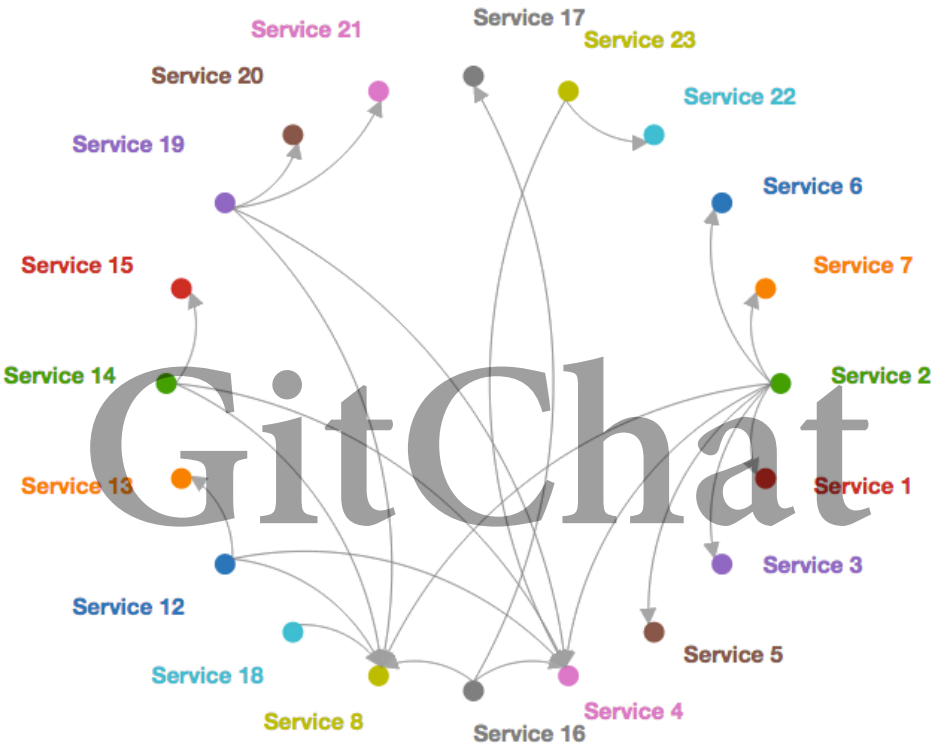
在使用Pact的过程中，我们团队通常主要遵循如下的实践：

消费者端

- 先从消费者开始，实现业务逻辑，并使用Pact生成消费者期望的契约。
- 当消费者运行完测试，生成契约后，使用Gradle/mvn publish(或Rake publish)将Pact契约文件上传到Pact-Broker中。使用Pact-Broker作为Pact契约文件的集中存储机制。
- 基于如上步骤，消费者可以独立于提供者进行开发、测试和部署。

提供者端

- 当提供者端的代码发生变更，提供者从Pact-Broker中获取契约文件，并进行验证。如果团队意外修改了API的接口，则运行pactVerify就能很容易的发现错误(不需要等到非要将二者部署到集成环境上，启动并进行联调)。
- 提供者端定义Fixture数据集，当运行pactVerify时，加载预先定义的Fixture数据，目的是匹配契约中定义的消费者期望的响应。
- 定义启动/停止提供者的任务。这样，就能在CI中启动提供者并对其自动回放PactVerify验证。具体可参见本文的代码示例。
- 借助Pact-Broker提供的重绘工具，静态显示服务间的依赖关系，类似如下图所示。



- 在某些场景下，如果产品只涉及提供者，不涉及消费者。譬如，提供API给第三方使用。此时，可以构建假的消费者并使用Pact，尽早验证提供者的契约内容是否发生变化，避免上线后破坏第三方消费者。

总结

随着微服务概念的热捧，不少组织开始使用微服务架构。但是，当服务规逐渐模化后，传统的集成测试方式面临越来越多的挑战。

理解消费者驱动的契约测试概念，并使用Pact这类框架，能有效帮助团队降低服务间的集成测试成本，尽早验证当提供者接口被修改时，是否破坏了消费者的期望。

当消费者与提供者之间采用REST通信时，Pact是不错的选择。但对于RPC的通信机制，可能无法使用Pact。

基于Pact-Broker，能够将服务间协作的契约文件集中化存储，并重绘服务间的静态关系图，为团队提供了全局的服务依赖关系图。

最后，附本文的示例代码，供大家参考，欢迎反馈。