第3章

接口自动化测试

接口自动化测试是集成自动化测试的主要体现。在实际项目中，较流行的接口是基于HTTP的REST接口和基于RPC的Dubbo接口。

3.1 基础知识

3.1.1 HTTP和REST

HTTP的全称为Hypertext Transfer Protocol，即超文本传输协议。在TCP/IP分层模型中，HTTP属于应用层协议。HTTP采用CS(Client/Server，客户端/服务端)模型，即客户端连接服务端发送请求后，需要等待直到收到服务端的响应。另外，HTTP是无状态的，即后续的请求如果需要用到前面的数据(状态)，那么必须重传。

HTTPS是HTTP通过SSL/TLS加密而成的。与HTTP相比，HTTPS的安全性更高，但传输速度更慢。

1. HTTP消息体

以访问笔者官网(网址详见前言二维码)为例来看一个HTTP消息的结构，HTTP请求的消息如下：

```
GET http://www.lujiatao.com/ HTTP/1.1
Host: www.lujiatao.com
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/89.0.4389.114 Safari/537.36
```

```
Accept: text/html, application/xhtml + xml, application/xml; q = 0. 9, image/avif, image/webp,
image/apng, * / * ;q = 0.8,application/signed - exchange;v = b3;q = 0.9
Accept - Encoding: gzip, deflate
Accept - Language: zh - CN,zh;q = 0.9,en;q = 0.8
```

HTTP 响应的消息如下：

```
HTTP/1.1 200 OK
Server: nginx/1.19.6
Date: Wed, 05 May 2021 07:34:28 GMT
Content - Type: text/html
Content - Length: 895
Last - Modified: Sun, 14 Mar 2021 08:04:52 GMT
Connection: keep - alive
ETag: "604dc3a4 - 37f"
Accept - Ranges: bytes

<! DOCTYPE html >< html lang = "">< head >< meta charset = "utf - 8">< meta http - equiv = "X - UA -
Compatible" content = " IE = edge">< meta name = " viewport" content = " width = device - width,
initial - scale = 1">< link rel = "icon" href = "favicon. ico">< title >卢家涛</title >< link href =
"/css/app. bfbdc5aa. css" rel = " preload" as = " style">< link href = "/css/chunk - vendors.
e17c3475. css" rel = "preload" as = "style">< link href = "/js/app. aa6644b9. js" rel = "preload"
as = "script">< link href = "/js/chunk - vendors. d61eecd2. js" rel = " preload" as = " script">
< link href = "/css/chunk - vendors. e17c3475. css" rel = " stylesheet">< link href = "/css/app.
bfbdc5aa. css" rel = "stylesheet"></head >< body >< noscript >< strong > We're sorry but 卢家涛
doesn't work properly without JavaScript enabled. Please enable it to continue. </strong >
</noscript >< div id = "app"></div >< script src = "/js/chunk - vendors. d61eecd2. js"></script >
< script src = "/js/app. aa6644b9. js"></script ></body ></html >
```

HTTP 的请求消息和响应消息具有类似的结构，它们包括以下三部分。

（1）请求行/响应行：HTTP 请求消息的第一行被称为起始行或请求行，它包括请求方法（以上示例的 GET）、请求 URL 和 HTTP 版本（以上示例的 HTTP/1.1）。HTTP 响应消息的第一行被称为状态行或响应行，它包括 HTTP 版本（以上示例的 HTTP/1.1）、状态码（以上示例的 200）和状态文本（以上示例的 OK）。

（2）请求头/响应头：HTTP 请求头/响应头即请求消息/响应消息中的 Headers，它们位于第一行之后和空行之前，并以“名称：值”对的形式出现。对于以上示例，Host：www.lujiatao.com 到 Accept-Language：zh-CN,zh;q=0.9,en;q=0.8 之间的所有内容都是请求头，而 Server：nginx/1.19.6 到 Accept-Ranges：bytes 之间的所有内容都是响应头。

（3）请求体/响应体：HTTP 请求体/响应体即请求消息/响应消息中的 Body，它们位于空行之后。对于以上示例，请求体是空的（即空行之后没有内容），而响应体是一个 HTML 文本。

2. HTTP 请求方法

HTTP 请求方法用于描述对给定资源执行的期望操作，包括以下 9 种请求方法。

（1）GET：GET 方法用于获取指定资源，可理解为“读取”资源。在 GET 方法的 URL 中可以携带参数。

（2）HEAD：与 GET 方法一样，HEAD 方法也用于获取指定资源，但 HEAD 方法的响应消息没有响应体。

(3) POST：POST 方法用于创建指定资源，比如常见的提交表单或上传文件等操作。POST 方法既可以在 URL 中携带参数，也可以在请求体中携带数据。

(4) PUT：PUT 方法用于修改指定资源，通常是将资源进行整体替换。PUT 方法是幂等的，即同样的请求调用一次与调用多次的效果是一样的。

(5) PATCH：PATCH 方法也用于修改指定资源，但通常是将资源进行局部修改。PATCH 方法和 POST 方法一样，它们都是非幂等的。

(6) DELETE：DELETE 方法用于删除指定的资源。

(7) OPTIONS：OPTIONS 方法一般用于检测服务器支持的请求方法。在 OPTIONS 方法的响应消息中包含一个名为 Allow 的响应头，该响应头的值表示了服务器支持的 HTTP 方法。

(8) TRACE：TRACE 方法主要用于调试或测试，是对服务器的一种连通性测试方法。

(9) CONNECT：CONNECT 方法一般用于代理服务器。比如目标服务器使用了 HTTPS 进行数据传输，若客户端(比如浏览器)使用代理服务器，那么客户端会首先使用 CONNECT 方法向代理服务器发送目标服务器的 IP 地址、端口和身份认证信息，在代理服务器与目标服务器建立连接后再进行后续的数据传输。

3. HTTP 状态码

由于 HTTP 状态码存在于 HTTP 响应消息中，因此又被称为 HTTP 响应码。

HTTP 状态码由 3 位数字组成，其中第一位数字代表当前响应的类型，共包含 5 种类型。

(1) 1××：表示一些提示性的响应信息。这类响应通常无须过多关注。

(2) 2××：表示请求成功。例如，200 OK(请求成功)、201 Created(请求成功并创建了一个资源)、204 No Content(请求成功但响应体为空)。

(3) 3××：表示重定向。例如，302 Found(资源被重定向到其他地址)、304 Not Modified(资源未修改，客户端应使用缓存)。

(4) 4××：表示客户端错误。例如，400 Bad Request(请求的语义或参数错误)、401 Unauthorized(未进行身份认证)、403 Forbidden(禁止访问资源)、404 Not Found(请求的资源不存在)、405 Method Not Allowed(不支持的请求方法)。

(5) 5××：表示服务器错误。例如，500 Internal Server Error(服务器发生了无法处理的内部错误)、502 Bad Gateway(网关无法得到应用服务器的正确响应)、503 Service Unavailable(服务器还未准备好处理该请求)、504 Gateway Timeout(网关等待应用服务器的响应超时)。

4. REST 和 REST 接口

(1) REST。

REST(Representational State Transfer，表现层状态转换)是使用了 HTTP 子集的一种软件架构风格，其被设计用于代替 SOAP(Simple Object Access Protocol，简单对象访问协议)。在这种架构风格中，用户通过资源标识符及对资源的操作(如 GET、POST 等)使资源的表现形式(状态)被转换，并呈现给最终用户以供使用。REST 最初源于 Roy T. Fielding 在 2000 年发表的博士论文 *Architectural Styles and the Design of Network-based Software Architectures*，有兴趣的读者可查阅该论文。

（2）REST 接口。

本节的重点聚焦到 REST 接口上。

REST 接口是指遵循 REST 架构风格的接口，这种接口也被称为 RESTful 接口或 RESTful API。

如果服务器的主域名为 www.example.com，那么 REST 接口的基 URL 一般为 http(s)://api.example.com 或 http(s)://www.example.com/api。考虑到接口的更新换代，一般还需要加入接口的版本（如 v1、v2 等），此时 REST 接口的 URL 演变为 http(s)://api.example.com/v1 或 http(s)://www.example.com/api/v1。

接着需要在 URL 中体现要操作的资源。资源应该使用名词来表示，且由于资源可以有多个，因此通常使用名词的复数形式来表示资源。例如，/users 表示用户资源，/users/1 表示用户 ID 为“1”的用户资源。加入资源后，REST 接口的完整 URL 已经被构建完成，完整 URL 如下：

```
http(s)://api.example.com/v1/users
http(s)://api.example.com/v1/users/1
```

或者如下：

```
http(s)://www.example.com/api/v1/users
http(s)://www.example.com/api/v1/users/1
```

在 REST 接口中，对资源的操作需要使用部分 HTTP 请求方法来表示，针对不同的操作对应的 HTTP 请求方法如下所述。

① 获取资源：使用 GET 请求，服务器应该返回 0 个、1 个或多个资源。

② 创建资源：使用 POST 请求，指示服务器创建一个资源。

③ 修改资源：使用 PUT 或 PATCH 请求，PUT 请求用于整体修改资源，而 PATCH 请求同于局部修改资源。

④ 删除资源：使用 DELETE 请求，指示服务器删除一个资源。

REST 接口的 7 个完整示例如表 3-1 所示。

表 3-1 REST 接口的 7 个完整示例

请求示例	说明	响应体示例	状态码

GET /users	查询全部用户，服务器应该返回一个用户列表	[{
		"id": 1,
		"idCard": "510105199612278838",
		"name": "张三"
		}, {
		"id": 2,
		"idCard": "510105199612276891",
		"name": "李四"
		}, {
		…
		}]
			200

续表

请求示例	说明	响应体示例	状态码

GET /users/1	查询用户ID为1的用户，服务器应该返回一个用户	{
		"id": 1,
		"idCard": "510105199612278838",
		"name": "张三"
		}
			200

GET
/users?page = 2&per_page = 10
	根据过滤条件查询用户，服务器应该返回一个用户列表	[{
		"id": 11,
		"idCard": ...,
		"name": ...
		}, {
		...
		}, {
		"id": 20,
		"idCard": ...,
		"name": ...
		}]
			200

POST /users
{
"idCard": "510105199612278838",
"name": "张三"
}
	创建一个用户，服务器应该返回该创建的用户	{
		"id": 1,
		"idCard": "510105199612278838",
		"name": "张三"
		}
			201

PUT /users/1
{
"idCard": "51010519961227883X",
"name": "张三(新)"
}
	整体修改一个用户，服务器应该返回修改后的用户	{
		"id": 1,
		"idCard": "51010519961227883X",
		"name": "张三(新)"
		}
			200

PATCH /users/1
{
"name": "张三(新)"
}
	局部修改一个用户，服务器应该返回修改后的用户	{
		"id": 1,
		"idCard": "510105199612278838",
		"name": "张三(新)"
		}
			200

DELETE /users/1	删除一个用户，服务器应该返回为空		204

在实际项目中，通常还会将数据封装到 data 中，并提供 code 和 message 字段以增加返回数据的实用性。比如重新封装 GET /users/1 请求的响应体，内容如下：

```
{
    "code": 1,
    "message": "成功",
    "data": {
        "id": 1,
```

```
        "idCard": "510105199612278838",
        "name": "张三"
    }
}
```

3.1.2 RPC 和 Dubbo

1. RPC

RPC(Remote Procedure Call,远程过程调用)协议是计算机通信协议,它允许本地计算机上的应用程序调用远程计算机上的应用程序,且这个远程调用过程类似于本地调用。RPC 的优势在于为开发人员屏蔽了远程调用的底层技术细节,让开发人员将精力聚焦于上层业务逻辑。在面向对象编程的应用程序中,RPC 通常体现为 RMI(Remote Method Invocation,远程方法调用)。RPC 有很多具体的实现形式,常见的有如下 5 种。

(1) NFS(Network File System,网络文件系统)。NFS 被设计用于跨计算机、操作系统、网络结构和传输协议的文件共享,这种可移植性底层是基于 RPC 来实现的。

(2) Java RMI(Java Remote Method Invocation,Java 远程方法调用)。Java RMI 允许在一个 JVM 中运行的应用程序调用另一个 JVM 中运行的应用程序提供的方法,因此它实现了 Java 应用程序之间的远程通信功能。

(3) JSON-RPC(JavaScript Object Notation-Remote Method Invocation,JavaScript 对象表示法-远程方法调用)。JSON-RPC 是一种无状态、轻量级的 RPC 实现,它使用 JSON 作为数据格式。

(4) XML-RPC(Extensible Markup Language-Remote Method Invocation,可扩展标记语言-远程方法调用)。XML-RPC 是使用 XML 作为数据格式、HTTP 作为传输机制的 RPC 实现。

(5) Apache Dubbo(以下简称为 Dubbo)。其是一个高性能、轻量级的开源 Java 微服务框架。

2. Dubbo 和 Dubbo 接口

Dubbo 是一个高性能、轻量级的开源 Java 微服务框架,它是一个 RPC 的实现框架。Dubbo 提供了以下六大核心能力。

(1) 面向接口代理的高性能 RPC 调用:基于代理的高性能远程调用能力,服务以接口为粒度,为开发人员屏蔽了远程调用的底层细节。

(2) 智能负载均衡:内置了多种负载均衡策略,可智能感知下游节点的健康状况,从而显著减少调用延迟,以提高系统的吞吐量。

(3) 服务自动注册及发现:支持多种类型的注册中心,可实时感知服务实例的上下线。

(4) 高度可扩展能力:遵循“微内核+插件”的设计原则,所有核心能力如协议、网络传输及序列化等被设计为扩展点,平等对待内置实现和第三方实现。

(5) 运行期间流量调度:内置条件、脚本等路由策略,通过配置不同的路由策略,可轻松实现灰度发布、同机房优先等功能。

(6) 可视化的服务治理与运维:提供丰富的服务治理和运维工具——随时查询服务元数据、服务健康状态及调用统计,并可以实时下发路由策略及调整配置参数。

Dubbo 的基本架构如图 3-1 所示。

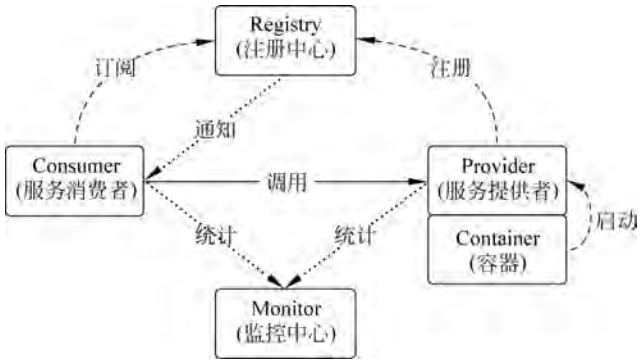

图 3-1　Dubbo 的基本架构

注：虚线表示初始化；实线表示同步调用；点线表示异步调用

首先，由容器启动服务提供者，并将服务提供者注册到注册中心，即向注册中心声明可以提供的服务。然后，服务消费者向注册中心请求需要使用的服务，注册中心将可用的服务返回给服务消费者。由于当可用的服务发生变化时，注册中心需要通知服务消费者，因此服务消费者对于注册中心来说是一个订阅者的角色，而注册中心对于服务消费者来说是一个发布者的角色。最后，服务消费者调用服务提供者提供的服务完成应用程序的请求。对于监控中心而言，需要同时统计服务消费者和服务提供者的调用信息（包括调用次数和调用时间）。另外，注册中心和监控中心都是可选的，服务消费者可以直连服务提供者。

以上都是对 Dubbo 框架的介绍，那么 Dubbo 接口又是什么呢？在 Dubbo 框架中，服务提供者将接口暴露为可远程调用的服务，因此 Dubbo 接口本质上就是 Java 接口，只是这些接口需要通过 Dubbo 框架暴露给服务消费者以供调用。

3.2　查看接口的辅助工具

3.2.1　浏览器开发者工具

使用开发者工具可以方便地看到当前浏览器中的 HTTP 请求和响应数据，提供开发者工具的主要浏览器包括 Chrome、Edge、Safari 和 Firefox 等。

以 Chrome 为例，在 Windows 计算机中使用 F12 键或 Ctrl＋Shift＋I 组合键打开开发者工具在 Network 标签页可看到 HTTP 请求和响应数据，如图 3-2 所示。

说明

如果读者使用的是笔记本计算机，通常就需要使用 fn＋F12 组合键才能打开开发者工具。另外，在 macOS 计算机中，需要将 Ctrl＋Shift＋I 组合键换为 option（alt）＋command＋I 组合键。

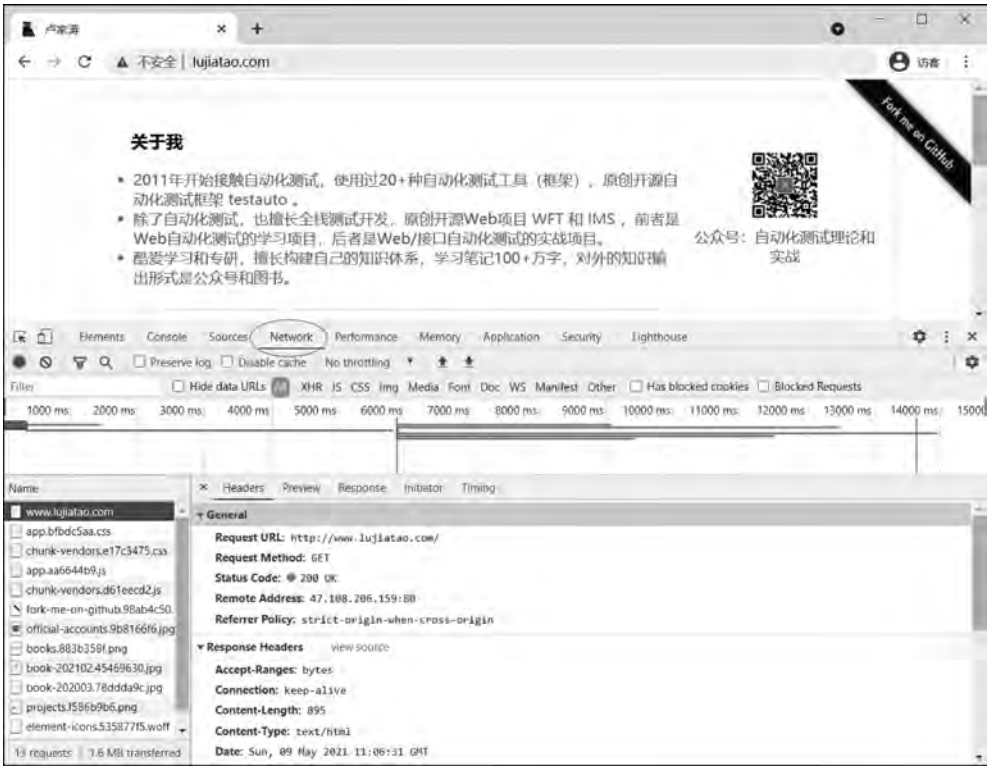

图 3-2　使用 Chrome 开发者工具查看 HTTP 请求数据

3.2.2　HTTP 代理和调试工具

浏览器开发者工具只能针对 Web 网页发起的 HTTP 请求进行数据展示，其无法显示手机端发起的 HTTP 请求数据。因此如果需要查看多个端（如 Web 网页、手机等）的 HTTP 请求数据，需要使用专业的 HTTP 代理工具，如 Fiddler、Charles 和 Burp Suite 等。这类工具通常还带有调试功能，如拦截请求、篡改响应等。

目前，Fiddler 已经分为 Fiddler Everywhere、Fiddler Classic、Fiddler Jam、FiddlerCap 和 FiddlerCore 5 个版本，本节以 Fiddler Everywhere 为例介绍如何查看 HTTP 请求数据。

首先，访问 Fiddler 官方网站下载 Fiddler Everywhere（地址详见前言二维码）。

说明

下载 Fiddler Everywhere 时需填写邮箱地址并选择国家，还需要选择 I accept the Fiddler End User License Agreement 选项，读者按照自身情况填写即可。

下载后是一个.exe 安装文件，直接双击安装即可。安装完成后，Fiddler Everywhere 会自动打开并要求登录，读者可使用已有账户或创建新账户来进行登录。

如果使用 Fiddler Everywhere 查看 Web 网页的 HTTP 请求数据，就不需要额外的配

置操作，直接在浏览器中访问网页即可在 Fiddler Everywhere 中查看到数据，如图 3-3 所示。

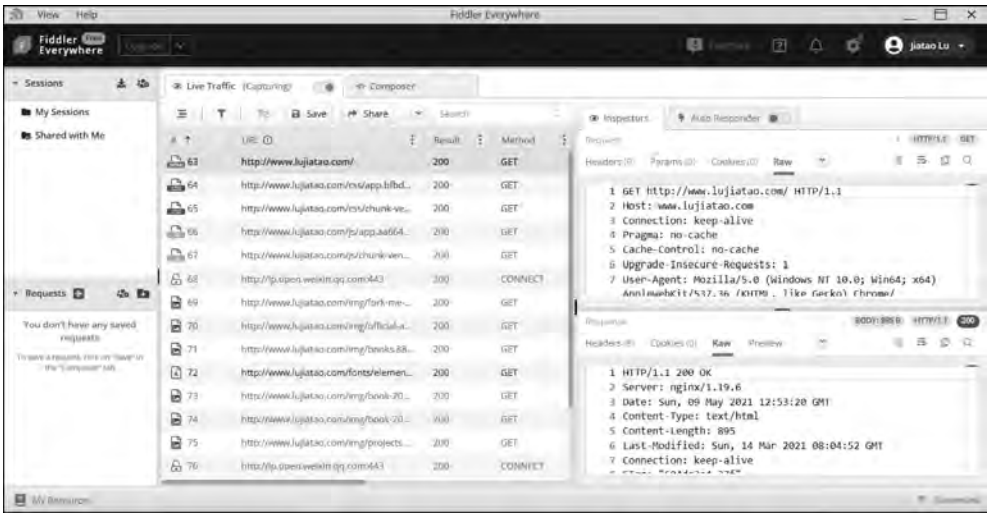

图 3-3　使用 Fiddler Everywhere 查看 HTTP 请求数据

由于本节并不是介绍如何使用 Fiddler Everywhere 查看 HTTP 请求数据的详细指南，因此如果读者需要使用它来查看 HTTPS 请求数据或手机的 HTTP(S)请求数据，需要进行额外配置。配置方法可查阅官方文档，查看 HTTPS 请求数据的配置方法官方文档地址详见前言二维码。

另外，查看 Android 和 iOS 设备 HTTP(S)请求数据的配置方法也请参考官方文档，官方文档地址详见前言二维码。

3.3　使用 Requests 测试 HTTP 接口

在 Python 中，Requests 是最流行的 HTTP 函数库，本节将介绍使用 Requests 来对 HTTP 接口进行测试。

Requests 支持 Python 2.7/3.5＋及 PyPy，其主要特性如下所述。

(1) 自动创建连接池和设置 Connection 为 keep-alive。

(2) 将 Cookie 通过“键-值”对的形式进行持久化，以便在多个请求之间共用会话。

(3) 类似于浏览器的 SSL/TLS 验证机制。

(4) 自动进行内容的解码和解压缩。

(5) 支持 Basic 和 Digest 身份认证机制。

(6) Unicode 编码的响应体。

(7) 支持 HTTP 和 HTTPS 代理。

(8) 支持 Multipart 类型的文件上传，支持流媒体下载。

(9) 支持设置请求超时时间。

(10) 支持分块传输。

本节会使用到两个示例：Web 应用程序 httpbin.org 和 IMS(Inventory Management System,库存管理系统)。前者由 Requests 的作者 Kenneth Reitz 提供,后者是笔者开发的一个用于学习自动化测试的项目。

3.3.1 简单请求和响应

在使用 Requests 之前需要先安装它,安装命令如下：

```
pip install requests
```

在 chapter_03 包中新增 learning_requests 子包,在 learning_requests 子包中新增 simple_request_and_response 模块,然后编写一些简单的请求。

【例 3-1】 编写一些简单的请求。

```
import requests

response_01 = requests.get('https://httpbin.org/get')
response_02 = requests.post('https://httpbin.org/post')
response_03 = requests.put('https://httpbin.org/put')
response_04 = requests.patch('https://httpbin.org/patch')
response_05 = requests.delete('https://httpbin.org/delete')
print(response_01.request.method, response_02.request.method, response_03.request.method,
response_04.request.method,response_05.request.method)
```

以上代码在导入了 requests 模块后,直接调用其中的函数便可以执行 GET、POST、PUT、PATCH 和 DELETE 请求。在 Requests 中,HTTP 响应被表示为一个 Response 对象,访问 request 属性可以得到响应的请求信息。以上代码在获得该请求信息后从中又提取了 HTTP 请求方法。

执行以上代码,执行结果如下：

```
GET POST PUT PATCH DELETE
```

当然还可以从 Response 对象中获取状态码、响应头和响应体。比如提取 response_01 状态码、响应头和响应体,代码如下：

```
print(f'状态码:{response_01.status_code}')
print(f'响应头:{response_01.headers}')
print(f'响应体:{response_01.json()}')
```

执行结果如下：

```
状态码:200
响应头:{'Date': 'Mon, 10 May 2021 22:50:42 GMT', 'Content-Type': 'application/json',
'Content-Length': '307', 'Connection': 'keep-alive', 'Server': 'gunicorn/19.9.0',
'Access-Control-Allow-Origin': '*', 'Access-Control-Allow-Credentials': 'true'}
响应体:{'args': {}, 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate',
'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.25.1', 'X-Amzn-Trace-Id':
'Root=1-6099b8c2-26e8ec416670ffff528aaee5'}, 'origin': '222.209.71.15', 'url':
'https://httpbin.org/get'}
```

从以上代码和输出可以看出，分别访问 Response 对象的 status_code 和 headers 属性可以获取状态码和响应头，而调用 Response 对象的 json()方法可以获取响应体的 JSON 表示形式。

除了调用 Response 对象的 json()方法，还可以访问它的 text 属性以直接获取响应体的文本表示形式，代码如下：

```
print(response_01.text)
```

执行结果如下：

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.25.1",
    "X-Amzn-Trace-Id": "Root=1-6099bd5b-3ce8749f1aad443f3528ca14"
  },
  "origin": "222.209.71.15",
  "url": "https://httpbin.org/get"
}
```

说明，Requests 会自动解码服务器的响应内容，但有时候不一定能处理正确，可以自己手动处理编码问题，代码如下：

```
response.encoding = 'utf-8'
print(response.text)
```

以上代码首先把响应内容设置为 UTF-8 编码，然后再获取响应体。

3.3.2 构建请求参数

1. 构建 URL 参数

URL 参数常用于 GET 请求的参数携带。

新增 build_request_param 模块，编写构建 URL 参数的代码。

【例 3-2】 编写构建 URL 参数的代码。

```
import requests

response = requests.get('https://httpbin.org/get?key=value')
print(response.json())
```

以上代码直接将 URL 参数添加在 URL 后面，这也是使用浏览器直接访问 GET 请求的方式。执行结果如下：

```
{'args': {'key': 'value'}, 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate',
'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.25.1', 'X-Amzn-Trace-Id':
```

```
'Root = 1 - 6099c6ca - 2f24cd593d9b6d486486f5b9 '}, 'origin ': '222. 209. 71. 15 ', 'url ':
'https://httpbin. org/get?key = value'}
```

从以上输出结果可以看出，httpbin. org 对带 URL 参数的请求做了特殊处理，即将请求传入的参数用 args 原样返回给调用者。

虽然可以直接将 URL 参数添加到 URL 中，但是更通用的方式是使用 params 参数将 URL 参数添加到请求中，代码如下：

```
my_params = {
    'key': 'value'
}
response = requests.get('https://httpbin.org/get', params = my_params)
```

此时可以调用 Response 对象的 url 属性查看构建后的 URL，代码如下：

```
print(response.url)
```

执行结果如下：

```
https://httpbin.org/get?key = value
```

从以上代码和输出可以看出，使用 params 参数构建的 URL 参数与直接将 URL 参数添加到 URL 后面的效果是一样的。不过推荐使用 params 参数的形式，这样便于将 URL 与参数解耦，方便自动化测试用例的维护。

2. 构建请求体参数

以 POST 请求为例介绍请求体参数的构建。常见的构建请求体参数有使用表单和 JSON 两种方式。

如果使用表单方式构建请求体参数，那么需要将数据传递给 data 参数，代码如下：

```
my_datas = {
    'key': 'value'
}
response = requests.post('https://httpbin.org/post', data = my_datas)
print(response.json())
```

执行结果如下：

```
{'args': {}, 'data': '', 'files': {}, 'form': {'key': 'value'}, 'headers': {'Accept': '*/
*', 'Accept - Encoding': 'gzip, deflate', 'Content - Length': '9', 'Content - Type':
'application/x - www - form - urlencoded', 'Host': 'httpbin.org', 'User - Agent': 'python -
requests/2.25.1', 'X - Amzn - Trace - Id': 'Root = 1 - 6099cbc1 - 1257865c33309bfd7ea2eeac'},
'json': None, 'origin': '222.209.71.15', 'url': 'https://httpbin.org/post'}
```

从以上输出结果可以看出，httpbin. org 对表单参数也做了特殊处理，即将请求传入的参数用 form 原样返回给调用者。

如果使用 JSON 方式构建请求体参数，那么需要将数据传递给 json 参数，代码如下：

```
my_json = {
    'key': 'value'
}
```

```
response = requests.post('https://httpbin.org/post', json=my_json)
print(response.json())
```

执行结果如下：

```
{'args': {}, 'data': '{"key": "value"}', 'files': {}, 'form': {}, 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate', 'Content-Length': '16', 'Content-Type': 'application/json', 'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.25.1', 'X-Amzn-Trace-Id': 'Root=1-6099ccde-7a06a7b213cb935122e1da43'}, 'json': {'key': 'value'}, 'origin': '222.209.71.15', 'url': 'https://httpbin.org/post'}
```

httpbin.org 将 json 参数用 data 原样返回给了调用者。

3. 自定义请求头

除了常见的 URL 形式或请求体形式的参数携带，有时需要将参数以自定义请求头的方式传递，例如，传递一个名称为 key，值为 value 的请求头，代码如下：

```
my_headers = {
    'key': 'value'
}
response = requests.post('https://httpbin.org/post', headers=my_headers)
print(response.json())
```

执行结果如下：

```
{'args': {}, 'data': '', 'files': {}, 'form': {}, 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate', 'Content-Length': '0', 'Host': 'httpbin.org', 'Key': 'value', 'User-Agent': 'python-requests/2.25.1', 'X-Amzn-Trace-Id': 'Root=1-6099ce05-0702cbd92292679d2faf7199'}, 'json': None, 'origin': '222.209.71.15', 'url': 'https://httpbin.org/post'}
```

httpbin.org 将自定义的请求头附加到 headers 中，并返回给调用者。

说明

请求头的名称对字母的大小写不敏感，如 User-Agent 和 user-agent 表示的是一样的。

以上示例中的返回值都是经过 httpbin.org 特殊处理的，实际项目的返回值肯定是不同的，这点需要读者特别注意。另外，对于文件上传的参数构建会有所区别，详见 3.3.6 节。

3.3.3 操作 Cookie

客户端（比如浏览器）在与服务器的交互过程中，为了验证客户端的合法性，当客户端与服务器建立连接并认证通过后，服务器会将会话 ID 放在响应头的 Set-Cookie 中，客户端在获取会话 ID 后，在后续请求中将会话 ID 通过 Cookie 的形式携带并发送回服务器。

在 Requests 中，访问 Response 对象的 cookies 属性可以获取 Cookie，而 Cookie 由一个 RequestsCookieJar 对象表示。新增 operate_cookie 模块，并以登录 IMS 为例演示 Cookie 的获取。

【例 3-3】 获取登录 IMS 的 Cookie。

```
import requests
from requests.cookies import RequestsCookieJar

body = {
    'username': 'admin',
    'password': 'admin123456'
}
response = requests.post('http://ims.lujiatao.com/api/login', data=body)
cookie = response.cookies
assert isinstance(cookie, RequestsCookieJar)
print(f'Cookie:{cookie.items()}')
```

执行以上代码，执行结果如下：

```
Cookie:[('JSESSIONID', 'BE96D3C41CA7644F438C5B1AC07439D5')]
```

以上代码调用了 RequestsCookieJar 对象的 items()方法来获取全部 Cookie。

说明

会话 ID 是由服务器根据一定算法生成的，因此读者看到的会话 ID 会与笔者的不同。

当然也可以只获取 Cookie 的全部键或全部值，分别使用 keys()和 values()方法即可，代码如下：

```
print(f'Cookie Keys:{cookie.keys()}')
print(f'Cookie Values:{cookie.values()}')
```

执行结果如下：

```
Cookie Keys:['JSESSIONID']
Cookie Values:['BE96D3C41CA7644F438C5B1AC07439D5']
```

还可以获取指定的 Cookie 值，代码如下：

```
print(f'会话 ID:{cookie.get("JSESSIONID")}')
```

执行结果如下：

```
会话 ID:BE96D3C41CA7644F438C5B1AC07439D5
```

以上代码是调用 get()方法来获取指定的 Cookie 值，也可以直接以字典形式来访问，代码如下：

```
print(f'会话 ID:{cookie["JSESSIONID"]}')
```

一旦获取到 Cookie 后，就可以在后续请求中将 Cookie 作为 cookies 参数传递给其他请求了，比如登录 IMS 后获取物品分类，代码如下：

```
requests.get('http://ims.lujiatao.com/api/goods-category', cookies=cookie)
```

当然在获取了 Cookie 后，也可对 Cookie 进行修改后再使用。修改 Cookie 需要使用

RequestsCookieJar 对象的 set()方法，代码如下：

```
cookie.set('key', 'value')
print(f'新 Cookie:{cookie.items()}')
```

执行以上代码，执行结果如下：

```
新 Cookie:[('key', 'value'), ('JSESSIONID', 'BE96D3C41CA7644F438C5B1AC07439D5')]
```

当 Cookie 作为 cookies 参数传递时，可使用字典来代替 RequestsCookieJar 对象，代码如下：

```
dict_cookie = {
    'JSESSIONID': cookie.get('JSESSIONID')
}
requests.get('http://ims.lujiatao.com/api/goods-category', cookies=dict_cookie)
```

3.3.4 详解 request()函数

由于 requests 模块的 get()、post()、put()、patch()、delete()、head()和 options()函数都是对 request()函数的进一步封装，因此有必要详细了解 request()函数的使用。

在 3.3.1 节中，笔者使用 get()函数向 httpbin.org 发起了一个 GET 请求，代码如下：

```
requests.get('https://httpbin.org/get')
```

但也可直接使用 request()函数来替代 get()函数，代码如下：

```
requests.request('get', 'https://httpbin.org/get')
```

从以上代码可以看出，通过将请求方法以字符串形式传递给 request()函数的第一个参数可以替代 get()函数。同理，也可以使用同样方式替代 post()、put()、patch()、delete()、head()和 options()函数。

request()函数支持许多参数，但除了 method 和 url 外，其他都是非必填参数。具体详见表 3-2。

表 3-2 request()函数参数

参数名称	参数含义	必填
method	HTTP 请求方法，支持 GET、POST、PUT、PATCH、DELETE、HEAD、OPTIONS 和 TRACE。对大小写不敏感，即传入 get、Get 或 GET 都表示 GET 请求	是
url	请求的 URL	是
params	URL 查询参数，通常使用字典类型，也可使用列表(列表元素为元组)等	否
data	请求体，通常使用字典类型，也可使用列表(列表元素为元组)等。Content-Type 默认为 application/x-www-form-urlencoded	否

续表

参数名称	参数含义	必填
json	请求体，通常使用字典类型，也可使用可序列化的 Python 对象。Content-Type 默认为 application/json	否
headers	请求头，使用字典类型	否
cookies	Cookie，使用字典类型或 CookieJar 类型对象。当然也可以使用 RequestsCookieJar 类型对象，因为 RequestsCookieJar 是 CookieJar 的子类	否
files	待上传的文件，Content-Type 为 multipart/form-data。详见 3.3.6 节	否
auth	身份认证数据，可使用元组类型或可调用对象，如('username', 'password')	否
timeout	请求超时时间（单位为秒），可使用浮点类型或元组类型。若使用浮点类型，则表示总超时时间；若使用元组类型，则表示连接和读取超时时间，比如：(connect_timeout, read_timeout)。总时间超时、连接超时和读取超时分别会抛出 Timeout、ConnectTimeout 和 ReadTimeout 异常	否
allow_redirects	是否允许重定向，使用布尔类型，默认为 True	否
proxies	代理服务器，使用字典类型，比如{'http': 'http_target_url', 'https': 'https_target_url'}或{'http_src_url': 'http_target_url'}	否
stream	是否立即下载响应内容，使用布尔类型，默认为 False	否
verify	服务端安全验证，使用布尔类型（是否验证服务器的 TLS 证书）或字符串类型（CA bundle 文件路径），默认为 True	否
cert	客户端安全验证，使用字符串类型（SSL 客户端证书.pem 文件的路径）或元组类型。元组类型如('/path/to/filename.cert', '/path/to/filename.key')	否
hooks	设置请求的 Hook 回调函数，详见 5.2 节	否

查看 request()函数的源代码后发现，它的实现很简单，除去注释内容后，就只有 3 行代码，代码如下：

```
def request(method, url, **kwargs):
    with sessions.Session() as session:
        return session.request(method=method, url=url, **kwargs)
```

从以上代码可以看出，request()函数实际上是将发送请求的动作交给了 Session 对象的 request()方法。关于 Session 详见 3.3.5 节。

3.3.5 使用会话

如果直接使用 get()、post()等函数发送请求，每个请求都需要建立和断开会话，这无形之中增加了系统开销。更好的方式是在多个请求中共用会话，在 Requests 中的会话用 Session 对象来表示。

仍然以登录 IMS 为例，看看在不使用会话的情况下请求的调用耗时。为此新增 use_session 模块，代码如下：

```
import time

import requests
```

```
body = {
    'username': 'admin',
    'password': 'admin123456'
}
start = time.time()
for i in range(10):
    response = requests.post('http://ims.lujiatao.com/api/login', data = body)
    cookie = response.cookies
    requests.get('http://ims.lujiatao.com/api/goods/all', cookies = cookie)
print(f'耗时:{time.time() - start}')
```

/goods/all 接口用于获取 IMS 中的全部商品。多次执行以上代码,在笔者的计算机上耗时基本维持在 1.3～1.5s。

接着使用会话做同样的操作,观察耗时情况,代码如下:

```
start = time.time()
for i in range(10):
    with Session() as session:
        response = session.post('http://ims.lujiatao.com/api/login', data = body)
        cookie = response.cookies
        session.get('http://ims.lujiatao.com/api/goods/all', cookies = cookie)
print(f'耗时:{time.time() - start}')
```

以上代码使用了 with…as 语句将使用会话的代码进行了包裹,以便使用完后可以自动断开会话。

要执行以上代码,还需要增加导入语句,代码如下:

```
from requests import Session
```

多次执行以上代码,在笔者的计算机上耗时基本维持在 1.2～1.4s。因此使用会话的耗时低于不使用会话的耗时,即使用会话提高了多个请求的执行效率。

除了提高效率,使用会话的另一个重要作用是共用数据,最常见的是共用 Cookie。在以上共用会话的代码中,可以省略操作 Cookie 的部分,修改后 with…as 语句中的代码如下:

```
session.post('http://ims.lujiatao.com/api/login', data = body)
session.get('http://ims.lujiatao.com/api/goods/all')
```

重新执行以上代码,执行结果仍然是成功的,因此使用会话后不再需要显式保存和传递 Cookie,即 Cookie 在会话中被自动共用了。

另一种数据共用是直接在 Session 对象中设置会话级默认数据,这样会话中的每个请求都默认携带该数据,代码如下:

```
with Session() as session:
    session.headers = {
        'key_01': 'value_01'
    }
    response = session.post('http://ims.lujiatao.com/api/login', data = body)
    first = response.request.headers['key_01']
    response = session.get('http://ims.lujiatao.com/api/goods/all')
    second = response.request.headers['key_01']
```

```
    assert first == second
```

对于会话中的具体请求而言，其传入的参数优先级更高，可以覆盖会话级默认数据，为此在以上 with...as 语句中增加 request_headers 作为具体请求的请求头来演示这个过程，代码如下：

```
request_headers = {
    'key_01': 'value_01_new',
    'key_02': 'value_02',
}
response = session.get('http://ims.lujiatao.com/api/goods/all', headers=request_headers)
print(response.request.headers)
```

重新执行测试代码，执行结果如下：

```
{'key_01': 'value_01_new', 'key_02': 'value_02', 'Cookie': 'JSESSIONID=
22872B2AA4E7B8A4AA6B6605C09065D2'}
```

从以上输出可以看出，当会话中的具体请求传入的数据与默认数据相同时，会覆盖默认数据（以上输出中的'key_01': 'value_01_new'）；当不相同时，就追加传入的数据（以上输出中的'key_02': 'value_02'）。

3.3.6 上传和下载文件

文件的上传和下载是应用程序的常见功能，因此必须掌握如何通过接口来上传和下载文件。

1. 上传文件

最常见的是 Multipart 类型的文件上传一般通过表单提交，其 Content-Type 为 multipart/form-data，示例表单代码如下：

```
<form action="/upload" method="POST" enctype="multipart/form-data">
    <input type="file" name="my-file"/>
    <input type="submit"/>
</form>
```

在 Requests 中，可以使用 files 参数模拟文件的上传，代码如下：

```
import requests

requests.post('http://your-ip:your-port/upload', files={'my-file': open('path/to/file
', 'rb')})
```

注意：建议使用二进制方式打开文件，因为 Requests 会尝试自动设置 Content-Length，其值为文件中的字节数，如果以文本方式打开可能会出错。

以上代码中的 your-ip、your-port 和 path/to/file 分别表示服务器的 IP 地址、端口号和待上传文件的全路径，读者应根据实际情况进行替换。my-file 对应的是表单中 input 标签的 name 属性值。

可以为待上传的文件提供更多信息，如文件名、Content-Type 和自定义的请求头等，代码如下：

```
request_headers = {
    'key_01': 'value_01',
    'key_02': 'value_02',
}
requests.post('http://your-ip:your-port/upload',
              files={'my-file': ('my-filename.png', open('path/to/file', 'rb'), '
image/png', request_headers)})
```

以上代码将文件名指定为 my-filename.png，将 Content-Type 指定为 image/png，并提供了两个自定义的请求头 key_01 和 key_02。

除了单个文件上传，Requests 也支持多个文件上传，代码如下：

```
files = [
    ('my-file', open('path/to/file_01', 'rb')),
    ('my-file', open('path/to/file_02', 'rb')),
]
requests.post('http://your-ip:your-port/upload', files=files)
```

2. 下载文件

下载文件需要访问 Response 对象的 content 属性，以便以二进制形式获取响应体。

以下载 httpbin.org 的收藏夹图标(favicon.ico)为例，代码如下：

```
response = requests.get('https://httpbin.org/static/favicon.ico')
with open(r'E:\favicon.ico', 'wb') as file:
    file.write(response.content)
```

执行以上代码后，E 盘根目录会生成一个 favicon.ico 文件。

对于大文件的下载建议使用分段传输，代码如下：

```
response = requests.get('https://httpbin.org/static/favicon.ico', stream=True)
with open(r'E:\favicon.ico', 'wb') as file:
    for chunk in response.iter_content(chunk_size=128):
        file.write(chunk)
```

使用分段传输时，需将请求的 stream 参数指定为 True，并使用 iter_content()方法来迭代 Response 对象的二进制内容。另外还可以使用 iter_content()方法的 chunk_size 参数来设置分段大小，默认的分段大小为 1。

以上代码虽然实现了分段传输，但是有一个问题：延迟下载响应内容(即 stream 参数指定为 True)后，Requests 并不会自动释放连接。为了解决这个问题，可以手动调用 Response 对象的 close()方法，但更好的方式是使用 with...as 语句自动释放连接，代码如下：

```
with requests.get('https://httpbin.org/static/favicon.ico', stream=True) as response:
    with open(r'E:\favicon.ico', 'wb') as file:
        for chunk in response.iter_content(chunk_size=128):
            file.write(chunk)
```

3.4 测试 Dubbo 接口

在开始测试 Dubbo 接口之前，请先搭建好 IMS Dubbo 环境，详见电子版附录 A.4 节。

说明

IMS(Inventory Management System，库存管理系统)是笔者开发的一个用于学习自动化测试的项目，分为 Spring Cloud 和 Dubbo 两个版本，前者已经部署到公网(网址详见前言二维码)，而后者需要读者自己部署。

有多种方式可以对 Dubbo 接口进行测试，包括使用 Java API、Spring XML、Spring 注解、Spring Boot、泛化调用和 Python 客户端等。那么这些方式应该如何选择呢？

(1) Dubbo 接口对测试人员公开：测试人员能获得 Dubbo 接口相关依赖时，推荐使用 Java API 或 Spring(Spring XML、Spring 注解或 Spring Boot)来测试 Dubbo 接口。

(2) Dubbo 接口对测试人员隐藏：测试人员不能获得 Dubbo 接口相关依赖时，推荐使用泛化调用来测试 Dubbo 接口。

(3) 使用其他语言来测试 Dubbo 接口：比如若使用 Python 来测试 Dubbo 接口，则需要使用 Python 客户端。

3.4.1 使用 Java API

由于本节需要使用到 Java，因此笔者首先创建了一个新的 Maven 工程 mastering-test-automation-for-dubbo。然后修改工程的 pom.xml 文件引入相关依赖，依赖如下：

```
<dependencies>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo</artifactId>
        <version>2.7.11</version>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-configcenter-zookeeper</artifactId>
        <version>2.7.11</version>
    </dependency>
    <dependency>
        <groupId>com.lujiatao</groupId>
        <artifactId>ims-api</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

以上配置中，dubbo 是 Dubbo 框架的基础依赖，dubbo-configcenter-zookeeper 是 Dubbo Zookeeper 注册中心的依赖，而 ims-api 是 IMS 的 Dubbo 接口。

待依赖下载完成后，读者可以先熟悉一下 IMS 的 Dubbo 接口源代码。其中，只有 GoodsCategoryService(物品类别)、GoodsService(物品)和 UserService(用户)3 个简单的接口，如图 3-4 所示。

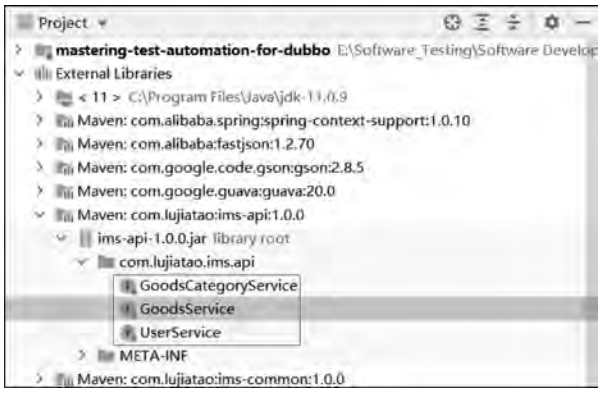

图 3-4　IMS 的 Dubbo 接口

接着笔者以测试 GoodsService 接口的 getById()方法为例介绍 Dubbo 接口的测试，该方法用于根据物品 ID 获取物品。为此先在/src/test/java 目录创建 com.lujiatao.dubbo 包，并新增 JavaAPI 类。

说明

根据 Maven 工程的规范，测试代码应该存放在/src/test/java 目录，非测试代码应该存放在/src/main/java 目录。

使用 Java API 测试 Dubbo 接口可以看作是模拟一个服务消费者来调用服务提供者提供的 Dubbo 接口，而服务消费者本质上属于一个应用程序，因此首先需要配置该应用程序。在 Dubbo 中，使用 ApplicationConfig 类来配置应用程序的代码如下：

```
ApplicationConfig applicationConfig = new ApplicationConfig();
applicationConfig.setName("JavaAPI");
```

以上代码将应用程序的名称配置为 JavaAPI。

然后需要配置注册中心。为了防止超时，笔者将超时时间设置为 10s(10000ms)，代码如下：

```
RegistryConfig registryConfig = new RegistryConfig();
registryConfig.setTimeout(10000);
registryConfig.setAddress("zookeeper://192.168.3.102:10003");
```

由于 IMS 使用 ZooKeeper 作为注册中心，因此以上代码使用了 zookeeper 作为注册中心地址的前缀，而 192.168.3.102 和 10003 是笔者搭建的注册中心 IP 地址和端口，读者应根据实际情况进行替换。

接着创建 Dubbo 接口引用的环节，需要使用到 ReferenceConfig 类，代码如下：

```
ReferenceConfig<GoodsService> referenceConfig = new ReferenceConfig<>();
```

```
referenceConfig.setApplication(applicationConfig);
referenceConfig.setRegistry(registryConfig);
referenceConfig.setInterface(GoodsService.class);
referenceConfig.setVersion("1.0.0");
```

ReferenceConfig 是一个泛型类，由于这里要测试 GoodsService 接口，因此将 GoodsService 作为类型形参传递给 ReferenceConfig。另外需要使用之前创建的应用程序和注册中心配置，并且还需要设置接口及其版本号。

最后调用 ReferenceConfig 对象的 get()方法获取到目标接口 GoodsService，然后调用其中的 getById()方法，代码如下：

```
GoodsService goodsService = referenceConfig.get();
Goods goods = goodsService.getById(1);
Gson gson = new GsonBuilder().create();
System.out.println(gson.toJson(goods));
```

以上代码使用到了 Gson，其用于将对象序列化为 JSON。

运行 JavaAPI 类，执行结果如下：

```
{"id":1,"brand":"HUAWEI","model":"Mate 40","desc":"","count":0,"goodsCategoryId":1,
"createTime":"2021-05-19 08:51:14","updateTime":"2021-05-19 08:51:14"}
```

由于 IMS 每次启动时会重新初始化数据库，因此读者看到的 createTime 和 updateTime 会与上述结果不一致。

有时出于测试目的，希望绕开注册中心直接调用 Dubbo 接口，这种方式被称为点对点直连。点对点直接需要删除配置注册中心的代码，并调用 ReferenceConfig 对象的 setUrl()方法设置直连 URL 即可。

【例 3-4】 Dubbo 接口的点对点直接。

```
package com.lujiatao.dubbo;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.lujiatao.ims.api.GoodsService;
import com.lujiatao.ims.common.entity.Goods;
import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.ReferenceConfig;

public class JavaAPI {

    public static void main(String[] args) {
// 配置应用程序
        ApplicationConfig applicationConfig = new ApplicationConfig();
        applicationConfig.setName("JavaAPI");
// 创建 Dubbo 接口引用
        ReferenceConfig<GoodsService> referenceConfig = new ReferenceConfig<>();
        referenceConfig.setApplication(applicationConfig);
        referenceConfig.setUrl("dubbo://192.168.3.102:10001");
        referenceConfig.setInterface(GoodsService.class);
```

```
        referenceConfig.setVersion("1.0.0");
// 调用 Dubbo 接口
        GoodsService goodsService = referenceConfig.get();
        Goods goods = goodsService.getById(1);
        Gson gson = new GsonBuilder().create();
        System.out.println(gson.toJson(goods));
    }

}
```

不管是使用注册中心还是点对点直连方式，当调用了 Dubbo 接口后，就可以使用单元测试框架编写自动化测试用例了。Java 中的单元测试框架有 TestNG、JUnit 等。

3.4.2 使用 Spring XML

使用 XML 是 Spring 的传统配置方式，因此也可以使用这种方法来配置一个调用 Dubbo 接口的测试应用程序。

首先创建 XML 配置文件，为此在/src/test 目录中新增 resources 目录，在 resources 目录中新增 consumer.xml 配置文件，文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://dubbo.apache.org/schema/dubbo
       http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

    <dubbo:application name="SpringXML"/>
    <dubbo:registry timeout="10000" address="zookeeper://192.168.3.102:10003"/>
    <dubbo:reference id="goodsService" interface="com.lujiatao.ims.api.GoodsService"
version="1.0.0"/>

</beans>
```

以上配置可以看成 3.4.1 节中的 XML 配置版本，其中也配置了应用程序名称、注册中心超时时间和地址、接口及其版本号。另外，配置中的 id 用于在代码中引用该接口。

以上配置完成后，就可以在 com.lujiatao.dubbo 包中新增 SpringXML 类用于 Dubbo 接口测试了。

【例 3-5】 使用 Spring XML 进行 Dubbo 接口测试。

```
package com.lujiatao.dubbo;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.lujiatao.ims.api.GoodsService;
import com.lujiatao.ims.common.entity.Goods;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringXML {

    public static void main(String[ ] args) {
          ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext ( "
consumer.xml");
        context.start();
        GoodsService goodsService = (GoodsService) context.getBean("goodsService");
        Goods goods = goodsService.getById(1);
        Gson gson = new GsonBuilder().create();
        System.out.println(gson.toJson(goods));
        context.close();
    }

}
```

以上代码使用了 ClassPathXmlApplicationContext 来加载 XML 配置，它是 Spring 中的一种应用程序上下文。getBean()方法用于获取一个 Bean（以上示例的 GoodsService），其参数 goodsService 对应 XML 配置中的 id。另外在使用应用程序上下文时，需使用 start()和 close()方法来分别执行启动和关闭操作。

XML 配置同样支持点对点直连，为此删除注册中心的相关配置。内容如下：

```
<dubbo:registry timeout = "10000" address = "zookeeper://192.168.3.102:10003"/>
```

接着增加直连 URL，内容如下：

```
<dubbo:reference url = "dubbo://192.168.3.102:10001" id = "goodsService" interface = "com.
lujiatao.ims.api.GoodsService"
                  version = "1.0.0"/>
```

3.4.3 使用 Spring 注解

对于 XML 中的部分配置可以使用 Properties 来替代，Dubbo 会默认加载 dubbo.properties 配置文件。可被替代的 XML 配置如下：

```
<dubbo:application name = "SpringXML"/>
<dubbo:registry timeout = "10000" address = "zookeeper://192.168.3.102:10003"/>
```

为了替代以上 XML 配置，在/src/test/resources 目录中新增 dubbo.properties 配置文件，文件内容如下：

```
dubbo.application.name = SpringXML
dubbo.registry.timeout = 10000
dubbo.registry.address = zookeeper://192.168.3.102:10003
```

在将应用程序和注册中心的配置换成使用 Properties 配置文件后，还可以直接使用注解在 Java 类中替换 XML 配置中的 Dubbo 接口引用部分。为此新增 Consumer 类，代码如下：

```
package com.lujiatao.dubbo;

import com.lujiatao.ims.api.GoodsService;
import org.apache.dubbo.config.annotation.DubboReference;
import org.apache.dubbo.config.spring.context.annotation.EnableDubbo;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableDubbo
public class Consumer {

    @DubboReference(version = "1.0.0")
    private GoodsService goodsService;

    @Bean("goodsService")
    public GoodsService getGoodsService() {
        return goodsService;
    }

}
```

以上代码中的@Configuration 注解用于修饰一个 Spring 配置类，而@EnableDubbo 注解用于使 Dubbo 组件成为一个 Spring Bean。

说明

如果 Properties 配置文件名称不是 dubbo.properties，就需要显式引入该配置文件，比如配置文件名为 my-dubbo.properties，则需要使用@PropertySource("/my-dubbo.properties")注解来修饰 Consumer 类。

接着新增 SpringAnnotation 类用于 Dubbo 接口的调用。

【例 3-6】 使用 Spring 注解进行 Dubbo 接口测试。

```
package com.lujiatao.dubbo;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.lujiatao.ims.api.GoodsService;
import com.lujiatao.ims.common.entity.Goods;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringAnnotation {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(Consumer.class);
        context.start();
        GoodsService goodsService = (GoodsService) context.getBean("goodsService");
```

```
        Goods goods = goodsService.getById(1);
        Gson gson = new GsonBuilder().create();
        System.out.println(gson.toJson(goods));
        context.close();
    }

}
```

以上代码跟 3.4.2 节中的 SpringXML 类很相似，唯一不同的是应用程序上下文的初始化方式，这里使用的是 AnnotationConfigApplicationContext 类。

若需使用点对点直连，则需要修改 Consumer 类中的@DubboReference 注解，新增 url 参数，代码如下：

```
@DubboReference(url = "dubbo://192.168.3.102:10001", version = "1.0.0")
```

然后删除 dubbo.properties 文件中的注册中心相关配置即可。

3.4.4 使用 Spring Boot

Spring Boot 可以看成对 Spring 的一个开箱即用的封装，它同时也是构建 Spring 微服务的基础，因此使用它来调用 Dubbo 接口是目前最常见的方式。

要使用 Spring Boot 调用 Dubbo 接口，首先需要在 pom.xml 文件中增加相关的依赖，依赖如下：

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.7.11</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.3.8.RELEASE</version>
    <exclusions>
        <exclusion>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

dubbo-spring-boot-starter 是 Dubbo 的 Spring Boot 配置依赖，而 spring-boot-starter 是 Spring Boot 的基础依赖。以上配置排除了 logback-classic 依赖，否则在 Spring Boot 启动时会抛出 IllegalArgumentException 异常，并提示 LoggerFactory is not a Logback LoggerContext but Logback is on the classpath.。

另外，由于 spring-boot-starter 与 dubbo 的 snakeyaml 依赖版本有冲突，需要把 dubbo 中的 snakeyaml 依赖排除掉。mastering-test-automation-for-dubbo 工程的完整依赖如下：

```
<dependencies>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo</artifactId>
        <version>2.7.11</version>
        <exclusions>
            <exclusion>
                <groupId>org.yaml</groupId>
                <artifactId>snakeyaml</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-configcenter-zookeeper</artifactId>
        <version>2.7.11</version>
    </dependency>
    <dependency>
        <groupId>com.lujiatao</groupId>
        <artifactId>ims-api</artifactId>
        <version>1.0.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
        <version>2.7.11</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>2.3.8.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>ch.qos.logback</groupId>
                <artifactId>logback-classic</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

接着在 com.lujiatao.dubbo 包中新增 SpringBoot 类用于 Dubbo 接口测试。

【例 3-7】 使用 Spring Boot 进行 Dubbo 接口测试。

```
package com.lujiatao.dubbo;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.lujiatao.ims.api.GoodsService;
import com.lujiatao.ims.common.entity.Goods;
import org.apache.dubbo.config.annotation.DubboReference;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SpringBoot {

    @DubboReference(version = "1.0.0")
    private GoodsService goodsService;

    public static void main(String[ ] args) {
        ConfigurableApplicationContext context = SpringApplication.run(SpringBoot.class,
args);
        GoodsService goodsService = (GoodsService) context.getBean("goodsService");
        Goods goods = goodsService.getById(1);
        Gson gson = new GsonBuilder().create();
        System.out.println(gson.toJson(goods));
    }

    @Bean("goodsService")
    public GoodsService getGoodsService() {
        return goodsService;
    }

}
```

以上代码可以看成 3.4.3 节中的 SpringAnnotation 和 Consumer 类的合并版本。不同之处在于应用程序上下文的初始化方式，这里使用的是 ConfigurableApplicationContext 类。

仅仅有以上代码还无法启动该 Spring Boot 应用程序，还需要增加配置信息。在 Spring Boot 中，通常使用 YAML 文件来配置应用程序。为此在/src/test/resources 目录新增 application.yml 文件，文件内容如下：

```
dubbo:
  application:
    name: SpringBoot
  registry:
    timeout: 10000
    address: zookeeper://192.168.3.102:10003
server:
  port: 8081
```

8081 是笔者使用的 Spring Boot 应用程序端口号，读者可根据实际情况修改端口号。

做完以上配置后，运行 Spring Boot 类便可以调用 Dubbo 接口了。

若需使用点对点直连，需要修改 Spring Boot 类中的@DubboReference 注解，新增 url 参数，代码如下：

```
@DubboReference(url = "dubbo://192.168.3.102:10001", version = "1.0.0")
```

然后删除 application.yml 文件中的注册中心相关配置即可。

3.4.5 使用泛化调用

泛化调用可以使用 Java API、Spring(Spring XML、Spring 注解或 Spring Boot)或框架/工具来调用 Dubbo 接口。本节介绍使用 Java API 和 JMeter 两种方式来调用 Dubbo 接口。

1. 使用 Java API 调用 Dubbo 接口

使用 Java API 通过泛化调用方式来调用 Dubbo 接口需要用到 GenericService 接口。服务消费者(客户端)使用 GenericService 实现的是泛化调用,而服务提供者(服务端)使用 GenericService 实现的是泛化实现(详见 3.5.2 节)。

在 com.lujiatao.dubbo 包中新增 GenericInvoke 类用于测试 Dubbo 接口。

【例 3-8】 使用泛化调用进行 Dubbo 接口测试。

```
package com.lujiatao.dubbo;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.ReferenceConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.rpc.service.GenericService;

public class GenericInvoke {

    public static void main(String[ ] args) {
        ApplicationConfig applicationConfig = new ApplicationConfig();
        applicationConfig.setName("GenericInvoke");
        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setTimeout(10000);
        registryConfig.setAddress("zookeeper://192.168.3.102:10003");
        ReferenceConfig<GenericService> referenceConfig = new ReferenceConfig<>();
        referenceConfig.setApplication(applicationConfig);
        referenceConfig.setRegistry(registryConfig);
        referenceConfig.setInterface("com.lujiatao.ims.api.GoodsService");
        referenceConfig.setVersion("1.0.0");
        referenceConfig.setGeneric(true);
        GenericService genericService = referenceConfig.get();
        Object goods = genericService.$invoke("getById", new String[ ]{"int"}, new Object[
]{1});
        Gson gson = new GsonBuilder().create();
        System.out.println(gson.toJson(goods));
    }

}
```

以上代码整体跟 3.4.1 节中的例 3-4 相似度很高,笔者重点介绍下它们之间的不同点。首先 ReferenceConfig 的类型形参被替换成了 GenericService,GoodsService.class 也被替换

成了字符串 com.lujiatao.ims.api.GoodsService，因为泛化调用是不依赖具体的 Dubbo 接口的。然后需要调用 ReferenceConfig 对象的 setGeneric()方法，并传入 true 以告诉 Dubbo 此时需要使用泛化调用。泛化调用的核心是 $invoke()方法，它支持 3 个参数，第一个参数（“getById”）表示方法名，第二个参数（“int”）表示方法的参数类型，第三个参数（1）表示方法的参数值。由于参数可能有多个，因此使用了字符串数组来传递参数的类型，使用对象数组来传递参数的值。

运行 GenericInvoke 类，执行结果如下：

```
{"goodsCategoryId":1,"createTime":"2021-05-19 08:51:14","count":0,"model":"Mate 40",
"updateTime":"2021-05-19 08:51:14","id":1,"class":"com.lujiatao.ims.common.entity.
Goods","brand":"HUAWEI","desc":""}
```

从以上输出可以看出，使用泛化调用的返回结果会多一个 class，其值表示该返回对象的类型。以上示例返回了一个 Goods 对象。

若需使用点对点直连，直接参考 3.4.1 节即可。

使用泛化调用时，需要注意传参方式。如果是 Java 的 8 种基本类型，直接传递即可；如果不是基本类型，就需要使用参数类型的全名，比如 String 类型必须写成 java.lang.String。针对参数为 POJO（Plain Old Java Object，简单的 Java 对象）的情况，参数值需要使用 Map<String, Object>类型进行传递。以在 IMS 中新增一个物品为例，传参的写法如下：

```
Map<String, Object> params = new HashMap<>();
params.put("goodsCategoryId", 1);
params.put("model", "New Model");
params.put("count", 0);
params.put("brand", "New Brand");
params.put("desc", "");
genericService.$invoke("add", new String[]{"com.lujiatao.ims.common.entity.Goods"}, new
Object[]{params});
```

以上 createTime、updateTime 和 id 3 个字段都没有传递，因为 IMS 的数据库表会自动生成这些字段的默认值。

2. 使用 JMeter 调用 Dubbo 接口

Dubbo 官方推荐使用 Apache JMeter Plugin For Apache Dubbo 插件为 JMeter 提供 Dubbo 接口的调用能力。

以 Windows 操作系统为例，首先访问 Apache JMeter Plugin For Apache Dubbo 的下载地址，地址详见前言二维码。

截至笔者写作本书时，最新的插件版本为 2.7.8，因此笔者下载的插件文件名为 jmeter-plugins-dubbo-2.7.8-jar-with-dependencies.jar，下载后将该文件复制到 JMeter 的/lib/ext 目录，接着使用插件提供的 Dubbo Sample 进行 Dubbo 接口调用即可。参数填写如图 3-5 所示。

执行结果与使用 Java API 进行泛化调用的执行结果是一致的。

若需使用点对点直连，则将 Register Center 的 Protocol 改为 none，并将 Address 改为 192.168.3.102:10003 即可。

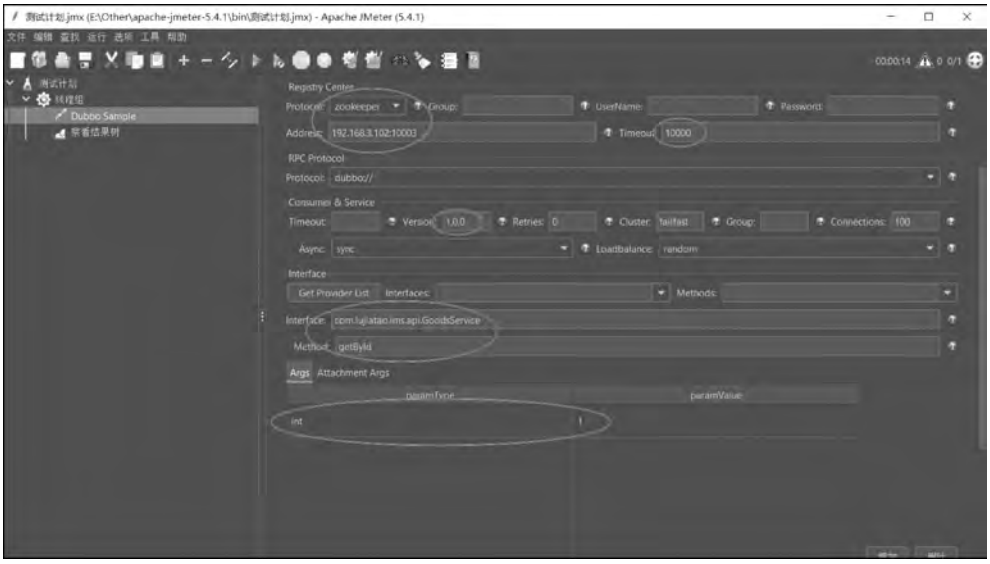

图 3-5　Dubbo Sample 的参数填写

相比使用 Java API，使用 Apache JMeter Plugin For Apache Dubbo 来传递 POJO 类型的参数就简单得多了。仍然以在 IMS 中新增一个物品为例，直接将 paramType 和 paramValue 分别修改为 com. lujiatao. ims. common. entity. Goods 和{"goodsCategoryId":1,"count":0,"model":"New Model","brand":"New Brand","desc":""}即可。

3.4.6　使用 Python 客户端

由于 Dubbo 官方推荐的 Python 客户端 Python Client For Apache Dubbo 并不支持 Python 3 且需要服务提供者使用 JSON-RPC，因此笔者使用了另一个 Python 客户端 python-dubbo-support-python 3 来替代。

执行命令即可安装 python-dubbo-support-python3，命令如下：

```
pip install python-dubbo-support-python3
```

本节继续使用 mastering-test-automation 工程，并在 chapter_03 包中新增 learning_dubbo_test 子包，在 learning_dubbo_test 子包中新增 learning_dubbo_test 模块，并编写 Dubbo 接口测试代码。

【例 3-9】 使用 Python 客户端进行 Dubbo 接口测试。

```
from dubbo.client import ZkRegister, DubboClient

zookeeper = ZkRegister('192.168.3.102:10003')
dubbo_client = DubboClient('com.lujiatao.ims.api.GoodsService', zk_register=zookeeper)
result = dubbo_client.call('getById', 1)
print(result)
```

ZkRegister 和 DubboClient 类分别表示注册中心和 Dubbo 客户端。DubboClient 对象

的 call()方法用于调用 Dubbo 接口，其第一个参数是方法名，第二个参数是元组类型，用于表示方法的参数，第三个参数表示超时时间(单位为秒)。

执行以上代码，执行结果如下：

```
{'updateTime': '2021-05-19 08:51:14', 'createTime': '2021-05-19 08:51:14',
'goodsCategoryId': 1, 'count': 0, 'desc': '', 'model': 'Mate 40', 'brand': 'HUAWEI', 'id': 1}
```

从以上输出可以看出，使用 Python 客户端调用 Dubbo 接口的返回值与使用 Java 是一致的。

若需使用点对点直连，则删除掉注册中心相关配置，并在创建 DubboClient 对象时提供直连 URL 即可，代码如下：

```
dubbo_client = DubboClient('com.lujiatao.ims.api.GoodsService', host = '192.168.3.102:
10001')
```

python-dubbo-support-python3 同样支持 POJO 类型的参数传递。仍以在 IMS 中新增一个物品为例，代码如下：

```
goods = Object('com.lujiatao.ims.common.entity.Goods')
goods['goodsCategoryId'] = 1
goods['count'] = 0
goods['model'] = 'New Model'
goods['brand'] = 'New Brand'
goods['desc'] = ''
dubbo_client.call('add', goods)
```

以上代码中的 Object 类用于表示一个 Java 对象，其需要单独导入，代码如下：

```
from dubbo.codec.encoder import Object
```

3.5 Mock 测试

3.5.1 HTTP 接口测试的 Mock

用于 HTTP 接口测试的 Mock 框架/工具有很多，比如 Responses、WireMock 等，还有像 RAP 这类接口管理工具，其提供了包括 Mock 在内的多种功能。本节以 Responses 为例介绍 Mock 是如何进行 HTTP 接口测试的。

Responses 用于模拟 Requests 的响应数据，因此可以使用 Responses 模拟 HTTP 接口的返回值。

要使用 Responses，需要执行命令安装它，命令如下：

```
pip install responses
```

以登录 IMS 为例，在不使用 Mock 的情况下，编写一个正常的测试函数，代码如下：

```
import requests
```

```
body = {
    'username': 'admin',
    'password': 'admin123456'
}


def test_normal():
    response = requests.post('http://ims.lujiatao.com/api/login', data=body)
    assert response.json() == {'code': 0, 'msg': '', 'data': None}
```

使用 Responses 可以轻松模拟 IMS 的返回数据，为此新增一个测试函数 test_mock_01()，代码如下：

```
@responses.activate
def test_mock_01():
    responses.add(responses.POST, 'http://ims.lujiatao.com/api/login', json={'code': 0,
'msg': '', 'data': None})
    response = requests.post('http://ims.lujiatao.com/api/login', data=body)
    assert response.json() == {'code': 0, 'msg': '', 'data': None}
```

以上代码首先使用@responses.activate 装饰器激活 Responses 的 Mock 功能，然后添加需要 Mock 的请求，包括请求方法、URL 和返回数据。

当然在执行以上测试代码之前还需要导入 responses，代码如下：

```
import responses
```

现在的问题是如何证明发起的接口请求命中的是 Mock 接口，而非真正的 IMS 接口呢？Responses 提供了一个 CallList 对象用于存放调用信息，可使用 responses.calls 对其进行快捷访问，代码如下：

```
@responses.activate
def test_mock_01():
    assert len(responses.calls) == 0 # 新增
    responses.add(responses.POST, 'http://ims.lujiatao.com/api/login', json={'code': 0,
'msg': '', 'data': None})
    response = requests.post('http://ims.lujiatao.com/api/login', data=body)
    assert response.json() == {'code': 0, 'msg': '', 'data': None}
    assert len(responses.calls) == 1 # 新增
```

以上代码在调用接口的前后判断了 CallList 对象的长度，以此证明发起的接口请求确实命中的是 Mock 接口。

除了使用@responses.activate 装饰器激活 Responses 的 Mock 功能，也可以使用 with…as 语句，代码如下：

```
def test_mock_02():
    with responses.RequestsMock() as response:
        response.add(responses.POST, 'http://ims.lujiatao.com/api/login', json={'code':
0, 'msg': '', 'data': None})
        response = requests.post('http://ims.lujiatao.com/api/login', data=body)
        assert response.json() == {'code': 0, 'msg': '', 'data': None}
```

这种情况适合测试函数/方法中只有部分请求需要 Mock 的场景。

以上只是对 Responses 的简单介绍，有兴趣的读者可查阅 Responses 的 GitHub 文档，地址详见前言二维码。

3.5.2 Dubbo 接口测试的 Mock

由于目前 Dubbo 接口没有成熟的开源 Mock 测试框架/工具，因此在本节，笔者将通过实现 GenericService 接口来实现一个 Dubbo 的 Mock 服务器。

首先在 mastering-test-automation-for-dubbo 工程中新增一个名为 mock-server 的 Maven 模块，在该模块的/src/main/java 目录中新增 com.lujiatao.mockserver 包，在 com.lujiatao.mockserver 包中新增 GenericServiceImpl 类，代码如下：

```
package com.lujiatao.mockserver;

import org.apache.dubbo.rpc.service.GenericException;
import org.apache.dubbo.rpc.service.GenericService;

import java.util.HashMap;
import java.util.Map;

public class GenericServiceImpl implements GenericService {

    @Override
    public Object $invoke(String method, String[] parameterTypes, Object[] args) throws
GenericException {
        if (method.equals("getById")) {
            if (parameterTypes.length == 1) {
                if (parameterTypes[0].equals("int")) {
                    Map<String, Object> params = new HashMap<>();
                    params.put("id", 1);
                    params.put("brand", "HUAWEI");
                    params.put("model", "Mate 40");
                    params.put("desc", "");
                    params.put("count", 0);
                    params.put("goodsCategoryId", 1);
                    params.put("createTime", "2021-05-19 08:51:14");
                    params.put("updateTime", "2021-05-19 08:51:14");
                    params.put("class", "com.lujiatao.ims.common.entity.Goods");
                    return params;
                }
            }
        }
        throw new IllegalArgumentException("不存在该 Mock 方法");
    }

}
```

以上代码重写了 GenericService 接口的 $invoke()方法，在方法体中判断了泛化调用时的方法名、参数数量及类型，最后使用 Map<String, Object>代替 POJO(以上示例中的 Goods)返回给方法调用者。

新增 MockServer 类，代码如下：

```
package com.lujiatao.mockserver;

import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.config.ServiceConfig;
import org.apache.dubbo.rpc.service.GenericService;

import java.util.concurrent.CountDownLatch;

public class MockServer {

    public static void main(String[ ] args) throws InterruptedException {
        ApplicationConfig applicationConfig = new ApplicationConfig();
        applicationConfig.setName("MockServer");
        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setTimeout(10000);
        registryConfig.setAddress("zookeeper://192.168.3.102:10003");
        ServiceConfig<GenericService> serviceConfig = new ServiceConfig<>();
        serviceConfig.setApplication(applicationConfig);
        serviceConfig.setRegistry(registryConfig);
        serviceConfig.setInterface("com.lujiatao.ims.api.GoodsService");
        serviceConfig.setVersion("1.0.0");
        serviceConfig.setRef(new GenericServiceImpl());
        serviceConfig.export();
        new CountDownLatch(1).await();
    }

}
```

ServiceConfig 类与 ReferenceConfig 类功能相对应，前者用于暴露接口，后者用于引用接口。以上代码创建了一个 ServiceConfig 对象，并将对 GoodsService 接口的调用映射到 GenericServiceImpl 类，即将对真实接口的调用映射到 GenericService 接口的实现类。配置好映射关系后，需要使用 export() 方法暴露该服务提供者。另外，在最后创建 CountDownLatch 对象并调用其 await() 方法的目的是阻塞主线程，否则在执行 MockServer 类时，主线程将很快结束，即无法使 Mock 服务器持续地运行。

最后到了验证 Mock 服务器的时候了。首先停止已启动的 IMS 服务提供者，然后启动 Mock 服务器(即运行 MockServer 类)。接着便可以使用泛化调用的方式来调用 Dubbo 接口了。

说明

出于演示目的，该 Mock 服务器仅提供了 GoodsService 接口 getById() 方法的 Mock，且返回值也仅仅是硬编码的。