Cucumber 高阶用法

活用 Cucumber 测试服务端开放 API

吕 双涛, 刘 娟, 和 柴 瑞亚

2016 年 12 月 28 日发布

一般而言，对于 OpenAPI 的测试通常采用 Google Chrome 的 PostMan 插件或者采用 Firefo
测试方式对于简单的功能验证是可行的，但是对于返回值的验证也仅限于肉眼观察，对于返
却有明显的不足。本文介绍了一种基于 Cucumber 和 RestAssured 来验证 OpenAPI 的方式，
能提供有效的返回值验证手段。关于 Cucumber 的认识和初级使用，请详见文章《Cucumbe

服务端开放 API 简介

所谓的开放 API 是服务型网站一种常见的应用，网站的服务商将自己的网站服务封装成一系
Interface，应用编程接口）开放出去，供第三方开发者使用，所开放的API就被称作 OpenA

本文中所涉及到的开放 API 主要实现了对数据的查询：根据卡号查询银行卡信息、分页获取
对应的账单地址信息等。读者可以通过类 io.cucumber.samples.dw.controller.AppControlle
续行文中主要使用了一个开放 API：“/card/query”。

样例应用的实现和启动方式

本文中所述被测试应用是基于Spring boot实现，采用 Spring boot 可以加快开发和部署速度，能够在开发环境中迅速启动并验证功能的实现是否符合预期设计。

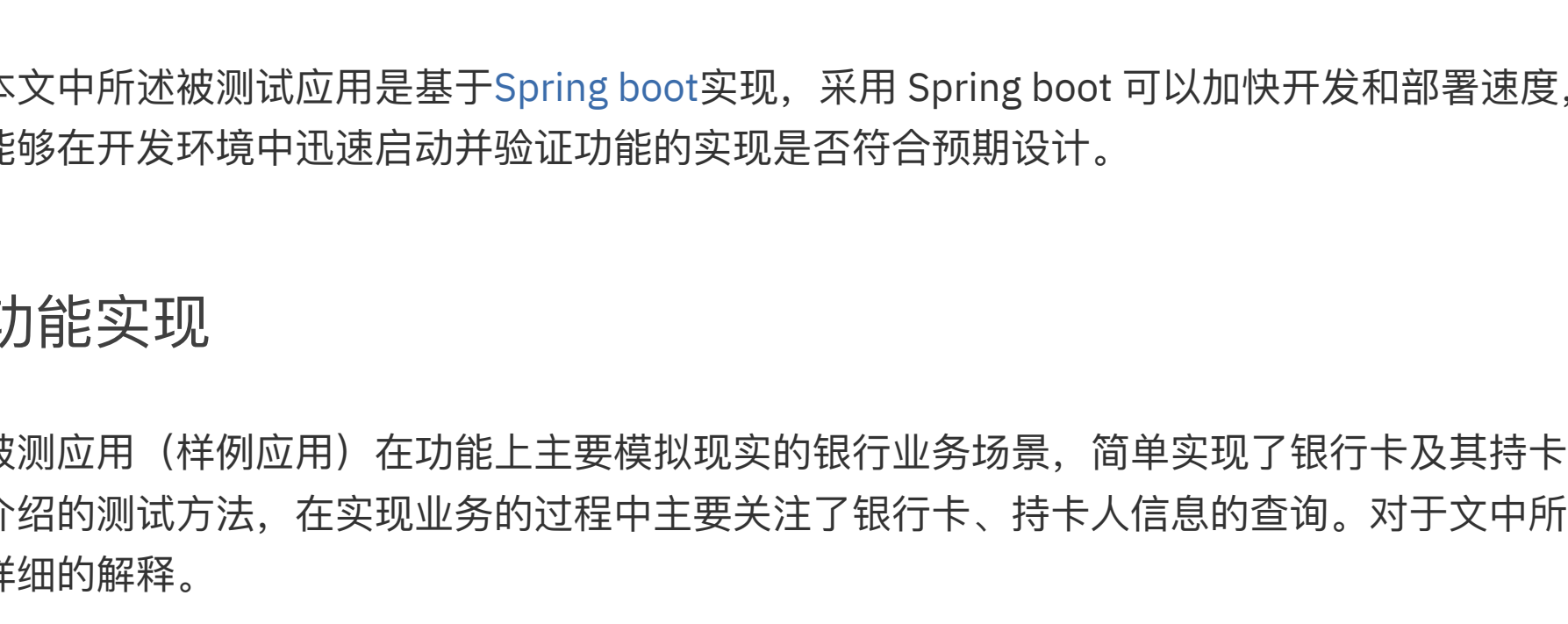
功能实现

被测应用（样例应用）在功能上主要模拟现实的银行业务场景，简单实现了银行卡及其持卡介绍的测试方法，在实现业务的过程中主要关注了银行卡、持卡人信息的查询。对于文中所详细的解释。

系统架构

在实现开放 API 功能的过程中，本文主要采用了如图 1 所示的架构，并使用主流的工具加以

图 1. 系统架构图



点击查看大图

系统基于 Spring-boot 结合 Spring-context 和 Spring-jdbc 开发而实现，在其上，搭建了提供启动时，使用 AppStarter 初始化数据和清理环境。

启动样例应用

AppStarter 类是整个应用的启动点，启动过程中主要做了：

1. 初始化 Spring；
2. 启动 Spring boot 及其内嵌的 Web Container 以接收 HTTP 请求；
3. 为了便于读者搭建环境，减少不必要的数据库配置过程，同时也为了保证每次运行的时候
   的是 Derby 的 Memory 模式，这样，在 Sample app 每次启动时都需要重新初始化数据库

当应用成功启动之后，可以从控制台看到类似如下的输出信息：

清单 1. 控制台输出

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.3.2.RELEASE)
… … … … … …
/*** Database initialization ***/
CREATE TABLE CARD
(
  ID              INT PRIMARY KEY          NOT NULL GENERATED ALWAYS AS IDE
  CARD_NUM        CHAR(8)                  NOT NULL UNIQUE,
  IS_PRIMARY_CARD SMALLINT DEFAULT 0       NOT NULL,
  CARD_OWNER_NAME VARCHAR(64)              NOT NULL,
  CARD_TYPE       SMALLINT DEFAULT 0       NOT NULL,
  STAR_POINTS     DECIMAL(10) DEFAULT 0.00 NOT NULL
)

INSERT INTO CARD (CARD_NUM, IS_PRIMARY_CARD, CARD_OWNER_NAME, CARD_TYPE, ST
VALUES ('C0000001', 1, 'CENT LUI', 0, 1024.64)

INSERT INTO CARD (CARD_NUM, IS_PRIMARY_CARD, CARD_OWNER_NAME, CARD_TYPE, ST
VALUES ('C0000002', 1, 'ROD JOHN', 0, 1048576.16)

INSERT INTO CARD (CARD_NUM, IS_PRIMARY_CARD, CARD_OWNER_NAME, CARD_TYPE, ST
VALUES ('C0000003', 1, 'STEVE JOBS', 0, 1048576.16)

INSERT INTO CARD (CARD_NUM, IS_PRIMARY_CARD, CARD_OWNER_NAME, CARD_TYPE, ST
VALUES ('S0000001', 0, 'CENT LUI', 1, 0.00)

INSERT INTO CARD (CARD_NUM, IS_PRIMARY_CARD, CARD_OWNER_NAME, CARD_TYPE, ST
VALUES ('S0000002', 0, 'ROD JOHN', 1, 512.64)

INSERT INTO CARD (CARD_NUM, IS_PRIMARY_CARD, CARD_OWNER_NAME, CARD_TYPE, ST
VALUES ('S0000003', 0, 'STEVE JOBS', 1, 1024.64)

```

```
38
39  CREATE TABLE ADDRESS
40  (
41    ID         INT PRIMARY KEY NOT NULL GENERATED ALWAYS AS IDENTITY ( START
42    CARD_NUM   CHAR(8)         NOT NULL,
43    REGION     VARCHAR(128)    NOT NULL,
44    COUNTRY    VARCHAR(6)      NOT NULL DEFAULT 'CHN',
45    STATE      VARCHAR(64)     NOT NULL,
46    CITY       VARCHAR(64)     NOT NULL,
47    STREET     VARCHAR(64)     NOT NULL,
48    EXT_DETAIL VARCHAR(128)    NOT NULL,
49    FOREIGN KEY (CARD_NUM) REFERENCES CARD (CARD_NUM)
50  )
51
52  INSERT INTO ADDRESS (CARD_NUM, REGION, COUNTRY, STATE, CITY, STREET, EXT_DE
53    SELECT
54      CARD_NUM,
55      'AP',
56      'CN',
57      'HeNan',
58      'LuoYang',
59      'Peking Rd',
60      'Apartment 1-13-01 No.777'
61    FROM CARD
62
63  INSERT INTO ADDRESS (CARD_NUM, REGION, COUNTRY, STATE, CITY, STREET, EXT_DE
64    SELECT
65      CARD_NUM,
66      'EU',
67      'ES',
68      'Madrid',
69      'Sol',
70      'Century Rd',
71      'Apartment 1-13-01 No.777'
72    FROM CARD
73  … … … … … …
74  /*** Open API URI mappings ***/
75  Mapped "{[/address/count],methods=[GET],produces=[application/json]}" onto
76  Mapped "{[/card/count],methods=[GET],produces=[application/json]}" onto org
77  Mapped "{[/address/query],methods=[GET],produces=[application/json]}" onto
78  Mapped "{[/address/all],methods=[GET],produces=[application/json]}" onto or
79  Mapped "{[/address/paged],methods=[GET],produces=[application/json]}" onto
80  Mapped "{[/card/query],methods=[GET],produces=[application/json]}" onto org
81  Mapped "{[/card/all],methods=[GET],produces=[application/json]}" onto org.s
82  Mapped "{[/card/paged],methods=[GET],produces=[application/json]}" onto org
83  Mapped "{[/database/test],methods=[GET],produces=[application/json]}" onto
84  … … … … … …
85  Tomcat started on port(s): 8080 (http)
```

应用启动验证

当应用启动之后，可以通过应用提供的 Open API/database/test 来验证系统是否正确启动。
当以 GET 方法访问 Open API/database/test 时，能够看到如 清单 2 的返回内容：

清单 2. API 返回内容

```
[
  {
    "cardCount": 6
  },
  {
    "addressCount": 12
  }
]
```

开放 API “/card/query”

在本例中,这个开放 API 提供了根据卡号查询银行卡信息以及对应账单地址信息的功能，通过
发现它接收一个 String 类型的参数"cardNum"；其返回值是一个银行卡列表所序列化之后的

清单 3. API 返回内容

```
@RequestMapping(value = "/card/query", method = RequestMethod.GET,
    produces = "application/json")
@ResponseBody
ResponseEntity<StandardJsonResponse> getCardByCardNum(
    @RequestParam("cardNum") String cardNum) {
    StandardJsonResponse<List<Card>> response = new StandardJsonResponse<>(
    List<Card> cardList = cardService.queryCardsByCardNum(cardNum);
    response.setData(cardList);
ResponseEntity<StandardJsonResponse> entity =
   new ResponseEntity(response, HttpStatus.OK);
    return entity;
}
```

在一次测试调用中，该 Open API 返回了如 清单 4 所示的测试内容：

清单 4. Open API 返回内容

```
{
  "errName": null,
  "errMsg": "SUCCESS",
  "errCode": 0,
  "data": [
    {
      "id": 5,
      "cardNum": "S0000002",
      "cardOwnerName": "ROD JOHN",
      "cardType": "1",
```

```
        "cardSeqNum": 0,
        "starPoints": 512,
        "cardBillingAddressList": [
          {
            "id": 5,
            "cardNum": "S0000002",
            "region": "AP",
            "country": "CN",
            "state": "HeNan",
            "city": "LuoYang",
            "street": "Peking Rd",
            "extDetail": "Apartment 1-13-01 No.777"
          },
          {
            "id": 11,
            "cardNum": "S0000002",
            "region": "EU",
            "country": "ES",
            "state": "Madrid",
            "city": "Sol",
            "street": "Century Rd",
            "extDetail": "Apartment 1-13-01 No.777"
          }
        ],
        "primaryCard": false
      }
    ]
}
```


测试服务端 API 面临的问题

通过以上对 Open API 的简介和本例中所主要使用的 API“/card/query” 的介绍，相信读者对
理解。如果一个测试人员拿到了一个对这样的 Open API 的测试需求，他会如何检查这个 AP

测试面对的问题

作为测试人员，在面对这个 Open API 测试需求时，通常需要弄明白以下的问题：

1. 该 API 的主要功能是什么?
2. 调用该 API 的参数是什么?
3. 调用该 API 的 HTTP Method 方法是什么?
4. 该 API 的返回内容和结构如何?
5. 调用该 API 所产生的影响是否符合期望?

诚然，对于真实情况下的 Open API，除去正向的功能性验证之外还有很多其他需要验证的点
证相关的部分。这里我们暂时先不考虑这些方面。

其实，上述 5 个问题的答案已经在前面有所介绍了，手动验证该 API,获取上述问题的答案也
却也有其局限性，尤其是有大量待验证的 API 的情形下。通常，在解决了 API 能否被被验证
种简单而且行之有效的方式自动化的验证这些 API。

现实工作中，由于开发迭代速度快，产品代码变更速度快、幅度大，更会出现 API 文档更新
决这些问题的考虑，我们首先想到了 Live documentation，进而想到了基于 Cucumber 来实

基于 Cucumber 测试服务端 API

之所以选择基于 Cucumber 来测试 Open API 主要是因为：

1. 基于 Cucumber 的功能描述可以作为 Live documentation：既能保持明确清晰的功能描
2. 基于 Cucumber-JVM 能够轻松实现自动化验证，并且有类似 Rest-Assured 这样的开源工
   据的 Open API 的验证会非常有效、非常方便。

接下来，我们将以 step-by-step 的方式讲述如何实现基于 Cucumber 测试 Open API。文中
预装 Java7 和 Maven3.2，对于如何安装 Java7 和 Maven3.2 以及对应的环境设置，已经超
Oracle JDK 和 Apache Maven 的相关文档。

创建 Maven 项目并启用 Cucumber

首先，从创建 Maven 项目（Intellij IDEA 下称作模块）开始：

创建 Maven 项目可以采用可视化的方式来做：Eclipse 和 Intellij IDEA 都已经充分支持，且
令行方式来创建 Maven 项目，待配置好 JDK 和 Maven 环境之后，在命令行下执行 清单 5 所
目：

清单 5. 创建 Maven 项目命令

```
mvn -B archetype:generate -DgroupId=io.cucumber.samples.dw -DartifactId=oper
```

其次，添加 project dependencies。完整的代码清单，查看清单 6。

待上述命令执行完成之后，在命令执行的目录下，会发现有一个新建的项目"open-api-test"
所需要的 dependencies，此处将测试所需要的所有 dependencies 都添加进来，后续行文中
的 dependencies 加以介绍。

清单 6. 添加 Maven 项目 dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven
    <modelVersion>4.0.0</modelVersion>
    <groupId>io.cucumber.samples.dw</groupId>
    <artifactId>open-api-test</artifactId>
    <packaging>jar</packaging>
    <version>1.0.0-SNAPSHOT</version>
    <name>open-api-test</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEn
        <java.version>1.7</java.version>
        <maven.compiler.version>3.3</maven.compiler.version>
        <junit.version>4.12</junit.version>
        <cucumber.version>1.2.4</cucumber.version>
        <spring.version>4.1.7.RELEASE</spring.version>
        <restassured.version>2.9.0</restassured.version>
        <jodaTime.version>2.9.3</jodaTime.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>info.cukes</groupId>
            <artifactId>cucumber-java</artifactId>
            <version>${cucumber.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>info.cukes</groupId>
            <artifactId>cucumber-junit</artifactId>
            <version>${cucumber.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>info.cukes</groupId>
            <artifactId>cucumber-spring</artifactId>
            <version>${cucumber.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
```

```
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>com.jayway.restassured</groupId>
            <artifactId>rest-assured</artifactId>
            <version>${restassured.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>com.jayway.restassured</groupId>
            <artifactId>json-schema-validator</artifactId>
            <version>${restassured.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven.compiler.version}</version>
                <configuration>
                    <encoding>UTF-8</encoding>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                    <compilerArgument>-Werror</compilerArgument>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-resources-plugin</artifactId>
                <version>2.7</version>
                <configuration>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.19</version>
                <configuration>
                    <argLine>-Dfile.encoding=UTF-8</argLine>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

做完上述工作之后，可以通过执行 清单7 所示命令下载对应dependencies到本地，并验证配

清单 7. 使用 Maven 命令编译项目

```
mvn -U clean compile
```

最后，启用 Cucumber。

启用 Cucumber 可以有很多层次，上述步骤中添加了 Cucumber-JVM 到 Maven 的项目依赖

使用 Cucumber DI 容器简化设计

对于致力于以 Cucumber 实现 Live Documentation 的读者来说，本文建议是用 Cucumber 高
Injection）容器。是的，这里所说的 Dependency Injection 就是大家在使用 Spring Framew

Cucumber DI 容器简介

DI 容器可以帮我们管理类实例的创建和维护，使得我们在使用类的时候不需要再使用 new 关
按照指定的生命周期去管理类的实例，这样，我们在设计和实现测试用例时，能够更加关注

Cucumber 几乎支持所有主流的 DI 容器：

1. PicoContainer：这是 Cucumber-JVM 作者 Aslak Hellesøy 所贡献的一个开源 DI 容器，
   果读者在此之前尚未接触过 DI 容器，建议以此 DI 容器入门；
2. Guice：Google 提供的轻量级 DI 容器，如果读者已经使用过诸如 Spring 的 DI 容器，不
   因是：
   - 有一定的学习成本；
   - 在设计上与 Spring 不太一致，易于混淆。
3. Spring：一系列非常流行的 Java 框架，不仅包括 DI 容器，而且包含了其他非常多的功能
4. CDI/Weld：Context and Dependency Injection 规范在 J2EE 平台的实现版本；
5. OpenEJB：Apache 的一款 stand-alone EJB Server 实现，其中包含了 Context and Dep

本文采用 Spring 作为 DI 容器，原因是 Spring Framework 在 Java 开发者中使用极其广泛，
群众基础。

使用 Spring 作为 Cucumber DI 容器

将 Spring 与 Cucumber 集成的过程其实相当简单，Cucumber-JVM 提供了一个 cucumber-s
steps、helper utilities 的创建，并注入到对应的测试场景中。上述 Maven 项目的 depender
Cucumber 和 Spring 集成的：

清单 8. Maven 依赖项

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-spring</artifactId>
    <version>${cucumber.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
```

其中"spring-context"部分可以有不同的表现形式，可以以上述 dependency 的形式出现，也
式出现，如清单 9 所示 java snippet 中的 ContextConfiguration annotation：

清单 9. Spring-context

```
@ContextConfiguration("classpath:cucumber.xml")
public class AppStarterTest {

}
```

以 Spring 作为 DI 容器，还需要给出 Spring beans/context 的配置文件，cucumber-spring
于“cucumber.xml”中的内容，其符合 Spring 定义规范，其中你可以定义各种 bean， 引用各
packages，配置是否采用 annotation。

下面是本文采用的一个样例，采用了 annotation config 方式，因此在 cucumber.xml 需要的

清单 10. cucumber.xml

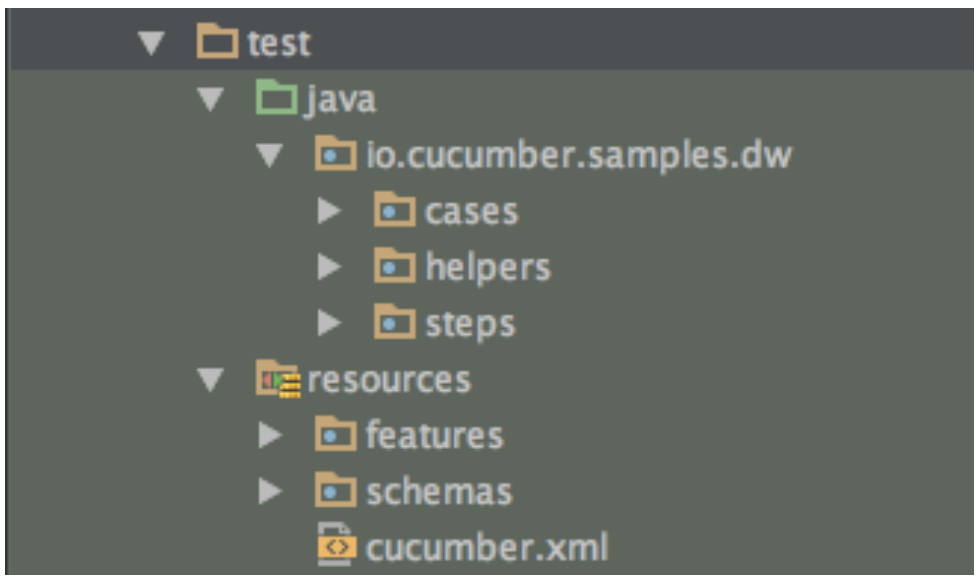
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframew
```

```
        http://www.springframework.org/schema/context http://www.springframe
    ">
     <context:annotation-config/>
     <context:component-scan base-package="io.cucumber.samples.dw.helpers"/>
</beans>
```

何时初始化 DI 容器

既然 DI 容器中包含了测试用例运行所需要的各种要素，那么何时初始化各种资源是至关重要
为您推荐一个较为实用的测试用例组织结构：如 图 2 所示：

图 2. 推荐的测试用例组织结构图



点击查看大图

- resources/features：定义各种功能 specifications；
- resources/schemas：定义 JSON Schema validation 相关文件；
- steps：feature 文件对应的 Java Steps；
- helpers：用于定义测试用例所共用的工具类、方法等；
- cases：测试用例入口点。

之所以采用这样的组织结构在于能够：

1. 能够清晰的组织测试用例；
2. 对于同一类 artifact，通过命名规范，能够快速定位；
3. 统一的测试入口点便于设置 Cucumber options，对于用例开发过程中的 debug、error a

那么 DI 容器应该在什么时候初始化呢?

首先，使用 helpers 初始化 DI 容器肯定是不合适的，因为 helper 类和方法自身不能够描述

其次，如果采用 steps 类来初始化 DI 容器，会造成一个问题：对于有多个 steps 类的情况，

最后，也是最合适的地方：cases package 下的测试用例入口点，其中使用了 @ContextConfiguration("classpath:cucumber.xml")出使初始化了 DI 容器。 清单11 为 DI 容

清单 11. DI 容器初始化

```
package io.cucumber.samples.dw.cases;
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;

/**
 * Created by stlv for developerworks article
 */
@RunWith(Cucumber.class)
@CucumberOptions(
        format = {
            "pretty",
            "html:target/html-report/",
            "json:target/json-report/dw.json"
        },
        features = {
            "classpath:features"
        },
        glue = {
            "io.cucumber.samples.dw.steps"
        },
        tags = {
            "@api",
            "~@ui"
        }
)
@ContextConfiguration("classpath:cucumber.xml")
public class AppStarterTest {

}
```

对于 CucumberOptions，请参考Cucumber 使用进阶文章中的介绍，此处不再赘述。

覆盖 CucumberOptions 默认值

为什么要 override 已经定义好的 CucumberOptions? 最常见的情况可能是这样的:

1. 有一个（些）测试用例失败了，在不改变任何文件的前提下，通过 override CucumberO
   例，这个是实际场景中最常见的情况；
2. 对于不同的测试环境，可以通过 override CucumberOptions 来实现在不同的环境下运行

因此，能够掌握 override CucumberOptions，对于熟练掌握 Cucumber 是非常有益处的。O
法有如下两种：

1. 通过 override Java 系统属性 cucumber.options 来实现。
   - 将 cucumber.options 直接传递给 Java 命令，例如：

     java -Dcucumber.options="-g step_definitions features" cucumber.api.cli.Main

   - 将 cucumber.options 传递给 Maven 命令，例如：

     mvn test -Dcucumber.options="-g step_definitions features"

2. 通过定义环境变量 CUCUMBER_OPTIONS 来实现：

   export CUCUMBER_OPTIONS="-g step_definitions features"

以上两种方式的效果是等价的，读者可以依据实际情况采用不同的实现方式。

截止到这里，读者应该已经能够成功搭建出基于 Spring DI 容器的自动化用例测试工程了，能
测试。

但是，对于本文中待测的 Open API，它返回的是 JSON 数据，因此，并不建议读者止步于此
Rest-Assured 工具，将其集成到测试环境，以便实现 JSON Schema 验证。

集成 Rest-Assured 和 JSON Schema 验证

JSON Schema 是一个非常强大的 JSON 结构验证工具，它通过定义 JSON 的结构、数据取值

常用类型

JSON Schema 将 JSON 数据类型划分为 6 种类型：

1. string：文本类型，可以包含 Unicode 字符；
2. number/integer：数字型或整数型；
3. object：类似于 Java 中 map 的概念，包含 key 机器对应的 value，key 在 JSON 中对应
   中对应的是包含零个或多个的 properties 的复杂结构；
4. array：对应于数组或 Java 中 list 的概念。
5. boolean：布尔数据类型。
6. null：null 通常用来表示一个不存在的值，当 Schema 中定义某一个 property 是 null，那
   null。

另外，加之 JSON Schema 也支持"引用"的概念，实现 Schema 定义的复用，因此，使用上述
义出各种复杂的数据类型。

定义 Schema 验证 Card 这一数据模型

本文所述的 Open API 的返回值也是 JSON 数据，样例如下 snippet 所示。从中可以看出，返
的结构，用于表示本次 API 调用结果；然后，"data"property 是 API 调用所返回的业务数据
数据，包括了"id","cardNum"等诸多 properties。同时，还包含了一个"cardBillingAddressL
表。

清单 12. Card data response

```
{
    "errName": null,
    "errMsg": "SUCCESS",
    "errCode": 0,
    "data": [
        {
            "id": 1,
            "cardNum": "C0000001",
            "cardOwnerName": "CENT LUI",
            "cardType": "0",
            "cardSeqNum": 0,
            "starPoints": 1024,
            "cardBillingAddressList": [
                {
                    "id": 1,
                    "cardNum": "C0000001",
                    "region": "AP",
                    "country": "CN",
                    "state": "HeNan",
                    "city": "LuoYang",
                    "street": "Peking Rd",
                    "extDetail": "Apartment 1-13-01 No.777"
                },
```

```
                {
                    "id": 7,
                    "cardNum": "C0000001",
                    "region": "EU",
                    "country": "ES",
                    "state": "Madrid",
                    "city": "Sol",
                    "street": "Century Rd",
                    "extDetail": "Apartment 1-13-01 No.777"
                }
            ],
            "primaryCard": true
        }
    ]
}
```

对于这样的返回值，根据上面所述的 JSON Schema 知识，定义出的 Schema 信息如下：

清单 13. Card data JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "银行卡数据格式验证 Schema",
  "definitions": {
    "eleInnerData": {
      "properties": {
        "id": {
          "type": "integer",
          "minimum": 1
        },
        "cardNum": {
          "$ref":"common-schema.json#/definitions/cardNum"
        },
        "cardOwnerName": {
          "type": "string",
          "minLength": 2,
          "maxLength": 128
        },
        "cardType": {
          "type": "string",
          "minLength": 1,
          "maxLength": 1,
          "enum": [
            "0",
            "1"
          ]
        },
        "cardSeqNum": {
          "type": "integer",
          "minimum": 0,
          "maximum": 127
        },
        "starPoints": {
          "type": "number",
          "minimum": 0.00
        },
```

```
        "cardBillingAddressList": {
          "$ref": "address-schema.json"
        },
        "primaryCard": {
          "type": "boolean",
          "enum": [
            true,
            false
          ]
        }
      },
      "required": [
        "id",
        "cardNum",
        "cardOwnerName",
        "cardType",
        "cardSeqNum",
        "starPoints",
        "cardBillingAddressList",
        "primaryCard"
      ],
      "additionalProperties": false
    },
    "eleData": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/eleInnerData"
      },
      "minItems": 0
    }
  },
  "allOf": [
    {
      "$ref": "common-schema.json"
    },
    {
      "type": "object",
      "properties": {
        "data": {
          "$ref": "#/definitions/eleData"
        }
      },
      "required": [
        "data"
      ],
      "additionalProperties": true
    }
  ]
}
```

其中引用了 common-schema 的定义如下：

清单 14. common-schema 定义

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
```

```
  "title": "通用交互数据格式验证 Schema",
  "definitions": {
    "cardNum": {
      "type": "string",
      "minLength": 8,
      "maxLength": 8,
      "pattern": "[C|S](0*)\\d+"
    },
    "errName": {
      "anyOf": [
        {
          "type": "string",
          "minLength": 1
        },
        {
          "type": "null"
        }
      ]
    },
    "errMsg": {
      "type": "string",
      "minLength": 1
    },
    "errCode": {
      "type": "integer",
      "maximum": 0
    }
  },
  "type": "object",
  "properties": {
    "errName": {
      "$ref": "#/definitions/errName"
    },
    "errMsg": {
      "$ref": "#/definitions/errMsg"
    },
    "errCode": {
      "$ref": "#/definitions/errCode"
    }
  },
  "required": ["errName","errMsg","errCode"],
  "additionalProperties": true
}
```

本文对于 JSON Schema 的定义并未详细描述，读者可以参考Understanding JSON Schema Schema。

在 Rest-Assured 中对返回数据执行 JSON Schema Validation

Rest-Assured 从 version 2.10 开始支持 JSON Schema Validation，读者只需要在 pom 文件 持 JSON Schema Validation 了：

清单 15. JSON Schema Validation 所需的 Maven dependencies

```
<dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>json-schema-validator</artifactId>
    <version>2.9.0</version>
    <scope>test</scope>
</dependency>
```

使用 Rest-Assured 提供的 Schema Validator 验证 Rest-Assured Response 返回数据是非常
代码就能实现以 schemaFile 所指定的 JSON Schema 来验证 response 的 body。

清单 16. 使用 Rest-Assured 做 JSON Schema Validation

```
public void assertThatRepliedCardDataMetSchemaDefinedSpecs(String schemaFile
    response.body(JsonSchemaValidator.
        matchesJsonSchemaInClasspath("schemas/" + schemaFile));
}
```

小结

本文首先介绍了服务端开放 API 的交互参数和返回格式，进而介绍如何使用 Cucumber 结合
API。本文介绍了如何使用 JSON Schema 来做数据结构和数据有效性验证，从而保证即使在
轻松地验证数据结构是否符合期望。

参考资源(resources)

- Understanding JSON Schema 通过丰富的例子介绍如何能够定义一个有效的 JSON Sche
- The Cucumber for Java Book: Behaviour-Driven Development for Testers and Develop
  语言描述；
- Cucumber documents 了解 Cucumber 不同语言的实现；
- Cucumber-JVM 使用 Cucumber 在 Java 中的实现。

下载资源

本文相关源代码 (dw_art2.tar | 296 KB)

评论

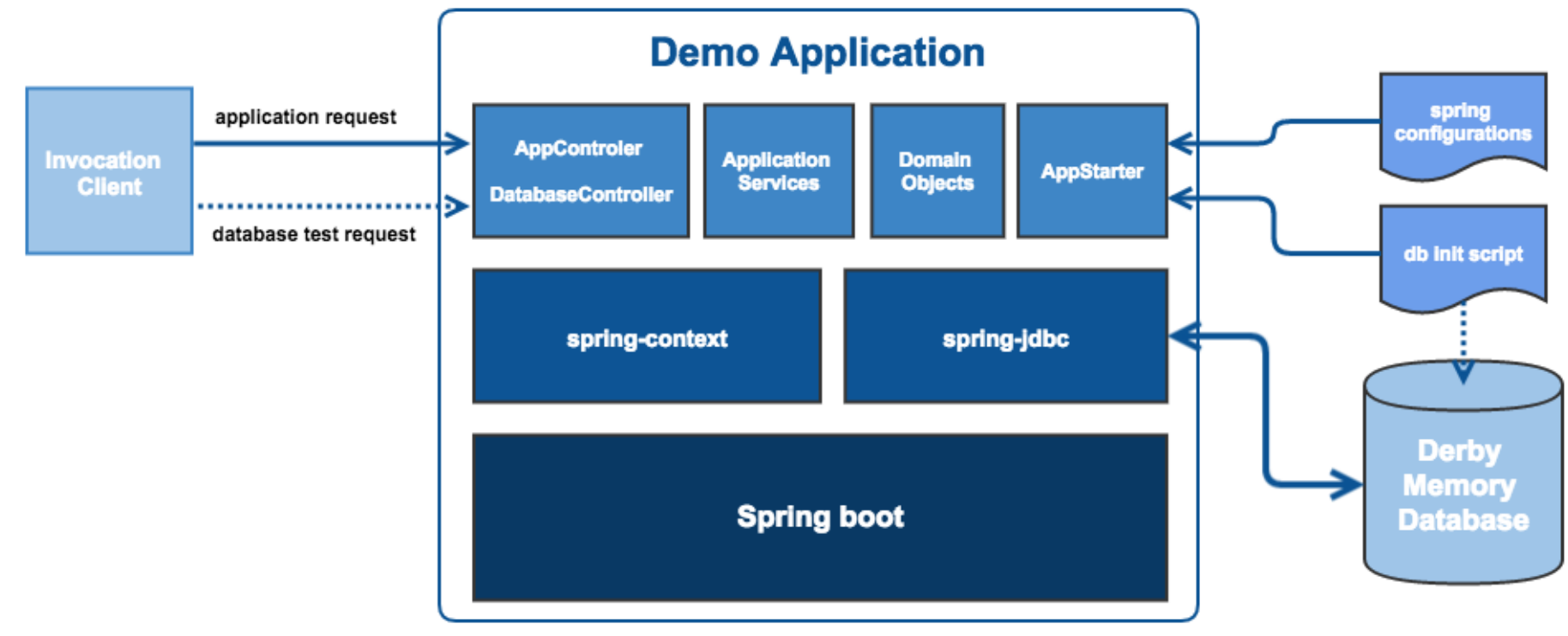
添加或订阅评论，请先登录或注册。

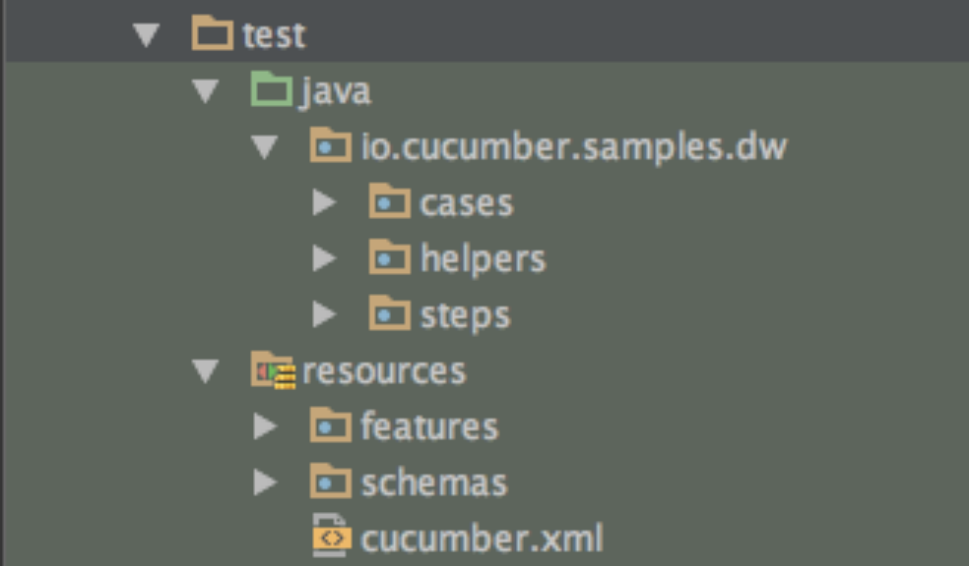
有新评论时提醒我

社区

dW 中国时事通讯

软件下载




test
java
io.cucumber.samples.dw
cases
helpers
steps
resources
features
schemas
cucumber.xml