简介

服务契约，指基于OpenAPI规范的微服务接口契约，是服务端与消费端对于接口的定义。服务契约用于服务端和消费端的解耦，服务端围绕契约进行服务的实现，消费端根据契约进行服务的调用。CSE Java SDK使用yaml文件格式定义服务契约，可支持多种风格开发微服务。

CSE-Codegen是基于Swagger Codegen实现的代码生成工具，用户只需在微服务工程的服务端和消费端的pom文件分别引入插件依赖，就可以根据定义好的契约文件生成服务端和消费端的框架代码，快速构建微服务应用。

特性描述

1. 新增契约同步功能

支持从远端的Git仓库同步一个或多个契约到微服务工程中，每次运行都可以根据最新的契约生成框架代码。

2. 服务提供者推荐使用SpringMVC风格，服务消费者推荐使用透明RPC风格

在不指明language的情况下，为服务提供者生成SpringMVC风格的框架代码，为服务消费者生成透明RPC风格的框架代码。

3. 为服务提供者和服务消费者生成完备的框架性代码

通过在微服务工程的pom文件中配置插件依赖，运行插件后生成以下文件：服务提供者provider生成model + delegate + controller + impl，服务消费者consumer生成model + delegate + impl。

4. 适应多服务多契约的场景

在配置中增加参数，可以适应多服务多契约的场景。

5. 使用契约中的x-java-class参数，避免consumer和provider的model路径不一致导致调用失败

x-java-class作为契约中一个重要的参数，存在于definitions中的每一个model，标志着model的package路径，能够保证服务消费者consumer和服务提供者provider的model的package路径统一。要求契约的每个model都具备x-java-class，根据x-java-class > service.packageName > packageName的优先级生成model的pacakge路径，避免consumer和provider的model路径不一致导致调用失败。

6. 最大程度保证显式契约和隐式契约的一致性

契约必须按照标准的Swagger API Spec语法来描述，使用yaml来表示。
契约中建议model都使用x-java-class参数，保证package路径统一。否则，使用REST方式会调用失败，使用RPC方式会使性能下降。
契约中不建议使用default返回码，CSE Java SDK不支持default返回码，插件也默认屏蔽default返回码。
插件严格按照契约定义，针对契约的不同返回码，生成的框架代码有所体现，最大程度保证显式契约和隐式契约一致。

版本

目前CSE-Codegen插件版本是2.2.8，可以到华为镜像站的huaweicloudsdk仓库获取。

使用CSE-Codegen插件快速开发微服务

介绍概念性的东西，可能有点枯燥，接下来我们就赶紧来看看如何使用CSE-Codegen插件来开发微服务应用。其实，使用CSE-Codegen插件开发微服务很简单。

首先，我们需要准备一个契约仓库，比如github，上传我们准备好的契约文件；配置一下maven的settings文件，然后在微服务工程里面引入插件；运行插件就能同步远端契约到本地并根据契约生成框架代码，用户自己填写必要的业务逻辑，就OK了。

为了方便大家体验CSE-Codegen插件的功能，我们提供了一个项目示例代码，按照如下步骤完成一个简单的微服务的开发。

1. Git仓库归档契约

创建远端Git仓库，用于契约管控，上传契约文件到Git仓库中。可以参考契约仓库示例。

2. 配置maven的settings文件

到华为镜像站里面找到HuaweiCloud SDK，下载settings.xml，替换掉原来的settings文件。在settings文件中增加如下配置。

```
<profiles>
  <profile>
    <id>MyProfile</id>
    <pluginRepositories>
      <pluginRepository>
        <id>HuaweiCloudSDK</id>
        <url>https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
        <releases>
          <enabled>true</enabled>
```

```
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>
```

3. 配置pom文件，引入插件

使用eclipse的用户可能需要在标签外层再套一层<pluginManagement> 标签。参数说明如下表所示。

参数	说明
skip	是否跳过执行该插件功能，默认是true，所以这里需要手动将skip设为false。
skipOverWrite	是否跳过文件覆盖，默认是false，即每次运行插件都可以更新框架代码。
repositories	定义多个契约仓库。
repository	定义单个契约仓库，即远端契约所在的git仓库。
userName	契约仓库的用户名（选填）。
password	契约仓库密码（选填）。
repoUrl	契约仓库地址，http、https、ssh格式都适用。
branch	契约仓库分支名。
services	定义多个服务。
service	定义单个服务，每个服务可以有多个契约文件。
appId	应用Id（选填，只在consumer这边指定，consumer跨应用调用provider的时候可以填对应的provider的appId）。
serviceName	服务名(服务消费者consumer和服务提供者provider都填provider的服务名)。
packageName（service层）	生成的框架代码（delegate、impl、controller）的包路径，当契约中的model里面没有x-java-class，也作为model的包路径。
schemaType	指定服务是consumer还是provider，根据契约生成相应的框架代码。
schemas	定义多个契约文件。
schema	定义单个契约文件。
schemaPath	契约文件在契约仓库的相对路径。
packageName	生成代码的包路径，优先级小于service里面的packageName，当两者都没有设置，插件运行会报错。

- consumer模块引入huawei-swagger-codegen-maven-plugin，插件版本号是2.2.8。schemaType指定参数“consumer”，生成服务消费者框架代码。

```
<plugins>
    <plugin>
        <groupId>io.swagger</groupId>
        <artifactId>huawei-swagger-codegen-maven-plugin</artifactId>
        <version>2.2.8</version>
        <executions>
            <execution>
                <goals>
                    <goal>generate</goal>
```

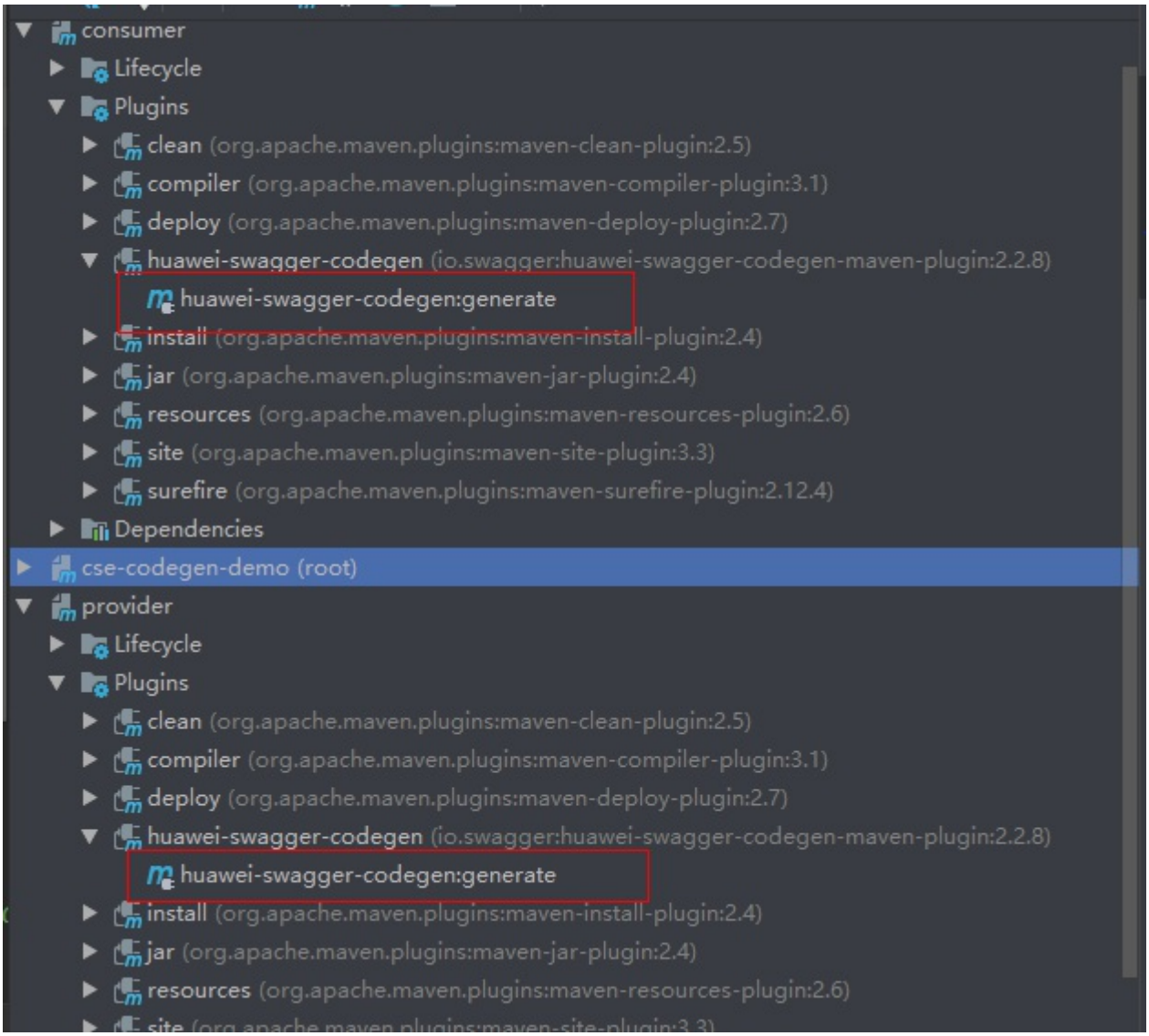
```
                </goals>
            </execution>
        </executions>
        <configuration>
            <skip>false</skip>
            <!--<skipOverwrite>false</skipOverwrite>-->
            <repositories>
                <repository>
                    <!--<userName></userName>-->
                    <!--<password></password>-->
                    <repoUrl> https://github.com/huaweicse/cse-codegen-schemas.git</repoUrl>
                    <branch>master</branch>
                    <services>
                        <service>
                            <!--<appId>lala</appId>-->
                            <serviceName>myprovider</serviceName>
                            <packageName>com.huawei.paas.consumer</packageName>
                            <schemaType>consumer</schemaType>
                            <schemas>
                                <schema>
                                    <schemaPath>dir/myservice.yaml</schemaPath>
                                </schema>
                            </schemas>
                        </service>
                    </services>
                </repository>
            </repositories>
            <packageName>com.huawei.paas.consumer</packageName>
        </configuration>
    </plugin>
</plugins>
```

- provider模块引入huawei-swagger-codegen-maven-plugin，插件版本号是2.2.8。schemaType指定参数“provider”，生成服务提供者框架代码。

```
<plugins>
    <plugin>
        <groupId>io.swagger</groupId>
        <artifactId>huawei-swagger-codegen-maven-plugin</artifactId>
        <version>2.2.8</version>
        <executions>
            <execution>
                <goals>
                    <goal>generate</goal>
                </goals>
            </execution>
        </executions>
        <configuration>
            <skip>false</skip>
            <skipOverwrite>false</skipOverwrite>
            <repositories>
                <repository>
                    <!--<userName></userName>-->
                    <!--<password></password>-->
                    <repoUrl> https://github.com/huaweicse/cse-codegen-schemas.git</repoUrl>
                    <branch>master</branch>
                    <services>
                        <service>
                            <serviceName>myprovider</serviceName>
                            <packageName>com.huawei.paas.provider</packageName>
                            <schemaType>provider</schemaType>
                            <schemas>
                                <schema>
                                    <schemaPath>dir/myservice.yaml</schemaPath>
                                </schema>
                            </schemas>
                        </service>
                    </services>
                </repository>
            </repositories>
            <packageName>com.huawei.paas.provider</packageName>
        </configuration>
    </plugin>
</plugins>
```

4. 运行插件，同步契约并生成框架代码

分别运行consumer和provider的CSE-Codegen插件，可以在命令行中执行mvn huawei-swagger-codegen:generate，或者一些IDE提供了运行插件的快捷方法，比如idea。或者直接编译整个微服务工程，运行插件（当然，不是说每次编译都要运行插件，所以插件配置提供了一个skip参数，默认skip为false，如果想要阻止插件运行导致重复生成框架代码，我们只需要将skip设置为true）。



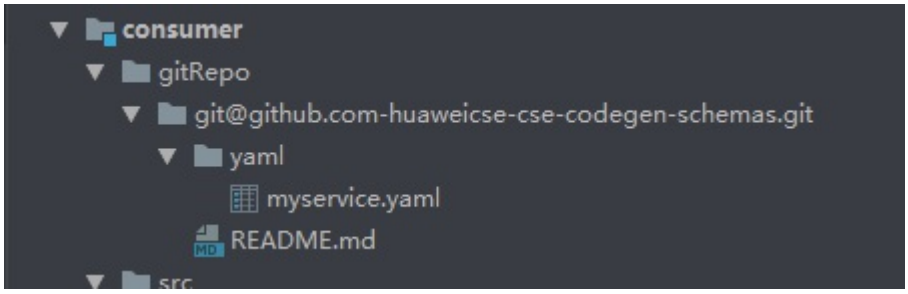
接下来就来看一下CSE-Codegen插件运行后带来了什么效果。

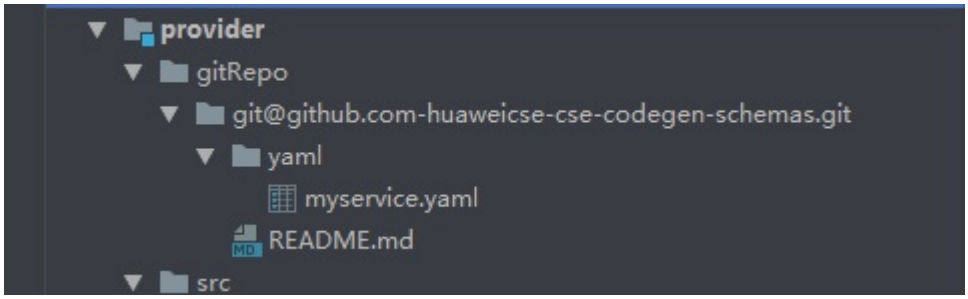
- 同步契约

  首先插件会检查工程的gitRepo目录下是否存在同名的Git仓库，如果存在则进行删除。

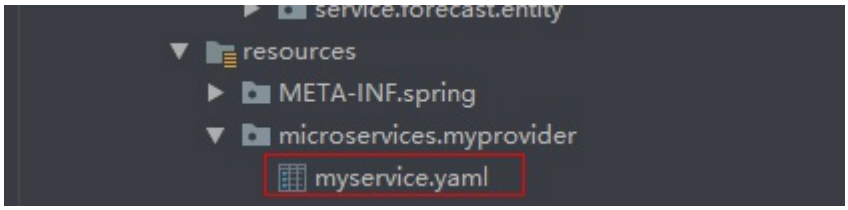
```
[INFO] --- huawei-swagger-codegen-maven-plugin:2.2.8:generate (default-cli) @ provider ---
[INFO] Delete existing repository: D:/契约生成代码插件/cse-codegen-demo/provider/gitRepo/git@github.com-huaweicse-cse-codegen-schemas.git
```

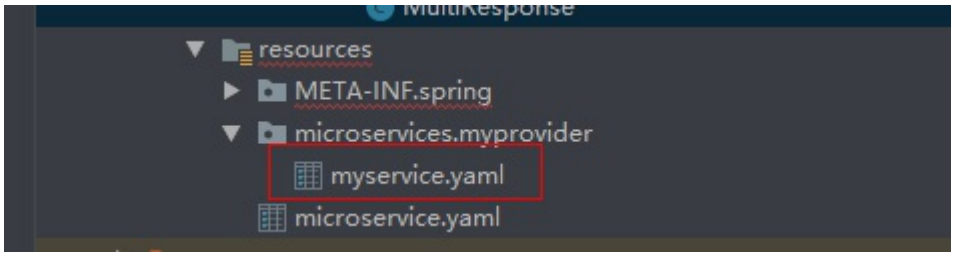
  然后下载整个Git仓库到工程的gitRepo目录下。



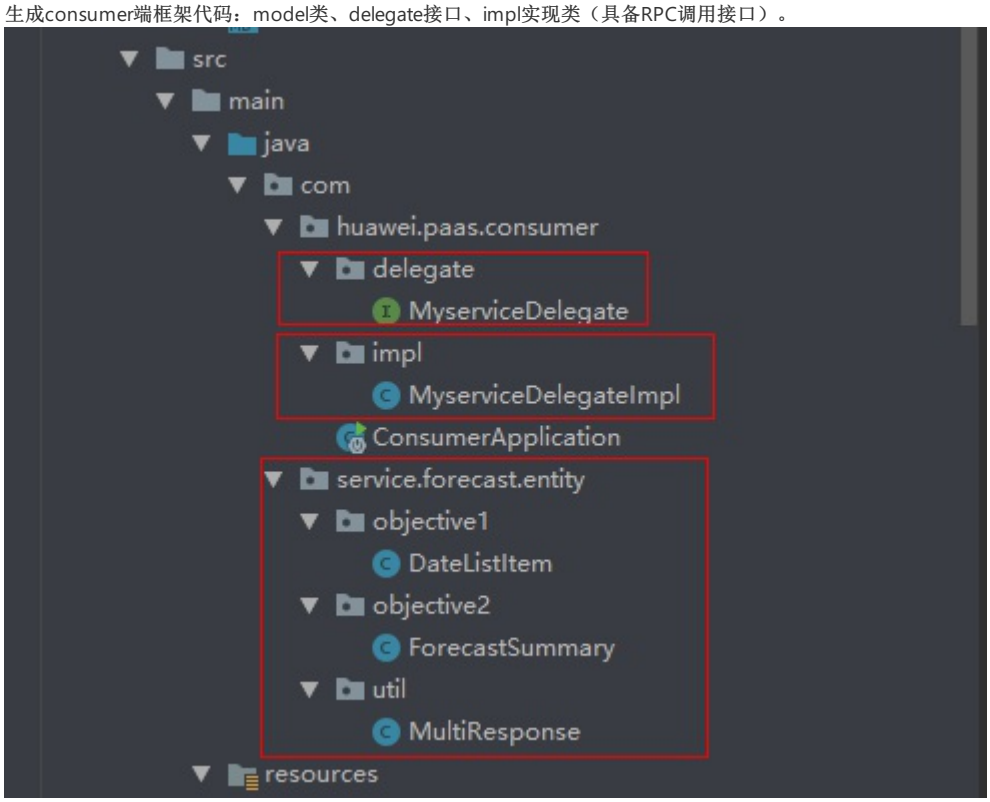


接下来，插件根据scchemaPath查找gitRepo里的契约文件，复制到契约文件到工程的microservices目录下，如目录中存在同名契约则替换掉。





- 生成框架代码
  - 生成consumer端框架代码：model类、delegate接口、impl实现类（具备RPC调用接口）。



接口MyserviceDelegate

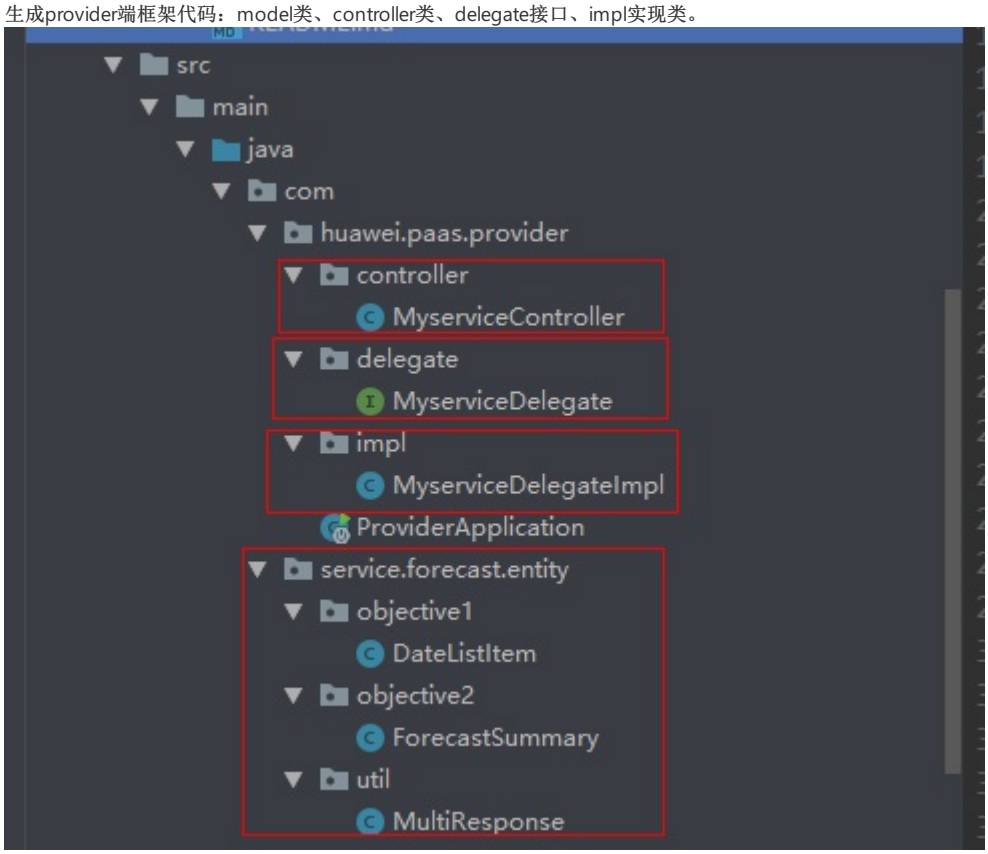
```
package com.huawei.paas.consumer.delegate;

import com.service.forecast.entity.objective2.ForecastSummary;
import com.service.forecast.entity.util.MultiResponse;
```

```
public interface MyserviceDelegate {

    String extra(String city);

    ForecastSummary show(String city);
}
```

接口实现类MyServiceDelegateImpl，除了根据契约生成的extra和show方法，consumer端这边还生成了useMyserviceDelegate的RPC调用接口，用户在实现MyserviceDelegate接口的时候，可以使用RPC方式调用provider端的服务。当然用户可以选择使用REST方式调用。

```
package com.huawei.paas.consumer.impl;

import org.springframework.stereotype.Component;
import org.apache.servicecomb.provider.pojo.RpcReference;
import com.huawei.paas.consumer.delegate.MyserviceDelegate;
import com.service.forecast.entity.objective2.ForecastSummary;
import com.service.forecast.entity.util.MultiResponse;

@Component
public class MyserviceDelegateImpl implements MyserviceDelegate {

    @RpcReference(microserviceName = "myprovider", schemaId = "myservice")
    private MyserviceDelegate useMyserviceDelegate;

    public String extra(String city){

        // Return MyserviceDelegate.extra()
        return null;
    }

    public ForecastSummary show(String city){

        // Return MyserviceDelegate.show()
        return null;
    }
}
```

- 生成provider端框架代码：model类、controller类、delegate接口、impl实现类。



MyServiceController

```
package com.huawei.paas.provider.controller;

import com.service.forecast.entity.objective2.ForecastSummary;
import com.service.forecast.entity.util.MultiResponse;
import com.huawei.paas.provider.delegate.MyserviceDelegate;

import javax.ws.rs.core.MediaType;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import org.apache.servicecomb.provider.rest.common.RestSchema;

@javax.annotation.Generated(value = "io.swagger.codegen.languages.CseSpringBootProviderCodegen", date = "2018-11-
21T20:03:08.654+08:00")

@RestSchema(schemaId = "myservice")
@RequestMapping(path = "/forecast" ,produces = { "application/json" }, consumes = { "application/json" })
public class MyserviceController {

    @Autowired
    private MyserviceDelegate userMyserviceDelegate;

    @RequestMapping(value = "/extra",
        produces = { "application/json" },
        consumes = { "application/json" },
        method = RequestMethod.POST)
    @ApiResponses({
        @ApiResponse(code = 200, response = String.class, message = "response of 200"),
        @ApiResponse(code = 400, response = MultiResponse.class, message = "parameter error"),
        @ApiResponse(code = 401, response = MultiResponse.class, message = "parameter empty")
        })
    public String extra( @RequestParam(value = "city", required = true) String city){

        return userMyserviceDelegate.extra(city);
    }

    @RequestMapping(value = "/show",
        produces = { "application/json" },
        consumes = { "application/json" },
        method = RequestMethod.GET)
    @ApiResponses({
        @ApiResponse(code = 200, response = ForecastSummary.class, message = "response of 200")
        })
    public ForecastSummary show( @RequestParam(value = "city", required = false) String city){

        return userMyserviceDelegate.show(city);
    }

}
```

接口MyServiceDelegate

```
package com.huawei.paas.provider.delegate;

import com.service.forecast.entity.objective2.ForecastSummary;
import com.service.forecast.entity.util.MultiResponse;

public interface MyserviceDelegate {

    String extra(String city);

    ForecastSummary show(String city);
}
```

接口实现类MyServiceDelegateImpl，用户可以在方法里面填写服务提供者provider的业务逻辑，最终数据可以返回到服务消费者consumer。

```
package com.huawei.paas.provider.impl;

import org.springframework.stereotype.Component;
import com.huawei.paas.provider.delegate.MyserviceDelegate;
import com.service.forecast.entity.objective2.ForecastSummary;
import com.service.forecast.entity.util.MultiResponse;

@Component
```

```
public class MyserviceDelegateImpl implements MyserviceDelegate {

    public String extra(String city){

        // Do some magic here
        return null;
    }

    public ForecastSummary show(String city){

        // Do some magic here
        return null;
    }
}
```

5. 根据框架代码，用户实现自己的业务逻辑

如果impl目录不存在或者impl目录下不存在契约对应的impl实现类，则生成impl实现类，否则不生成（避免覆盖用户已有的业务逻辑）。用户可以在impl实现类中增加自己的业务逻辑。如果用户想要使用框架提供的impl实现类，那么就必须将现有的实现类删除，重新运行插件。

下面提供了一个业务逻辑实现的示例。
服务提供者provider的接口实现类MyServiceDelegateImpl

```
@Component
public class MyserviceDelegateImpl implements MyserviceDelegate {
    @Override
    public ForecastSummary show(String city) {

        ForecastSummary forecastSummary = new ForecastSummary();

        List<DateListItem> dateListItemList = new ArrayList<>();

        for (long i = 1; i <= 30; i++) {
            DateListItem dateListItem = new DateListItem();
            dateListItem.date(i).image("image").dateTxt("dateTxt").temperatureMax(30.0).weather("sunny")
                    .temperatureMin(20.0).temperature(25.0).humidity(20.0).pressure(90.0).windSpeed(5.0)
                    .cloudsDeg(2.0);
            dateListItemList.add(dateListItem);
        }


forecastSummary.country("China").cityName(city).coordinatesLat(30.0).coordinatesLon(70.0).dateList(dateListItemList).currentTime(10l);

        return forecastSummary;
    }

    public String extra(String city) {

        MultiResponse multiResponse = new MultiResponse();

        if (city == null || city.equals("")) {

            multiResponse.setCode(401);
            multiResponse.setMessage("city不能为空");

//            throw new InvocationException(401, "parameter empty", "city不能为空");
            throw new InvocationException(multiResponse.getCode(), "parameter empty", multiResponse);
        }

        if (!city.equals("Shenzhen")) {

            multiResponse.setCode(400);
            multiResponse.setMessage("city必须是Shenzhen");

//            throw new InvocationException(400, "parameter error", "city必须是Shenzhen");
            throw new InvocationException(multiResponse.getCode(), "parameter empty", multiResponse);
        }

        return city;
    }
}
```

服务消费者provider的接口实现类MyServiceDelegateImpl

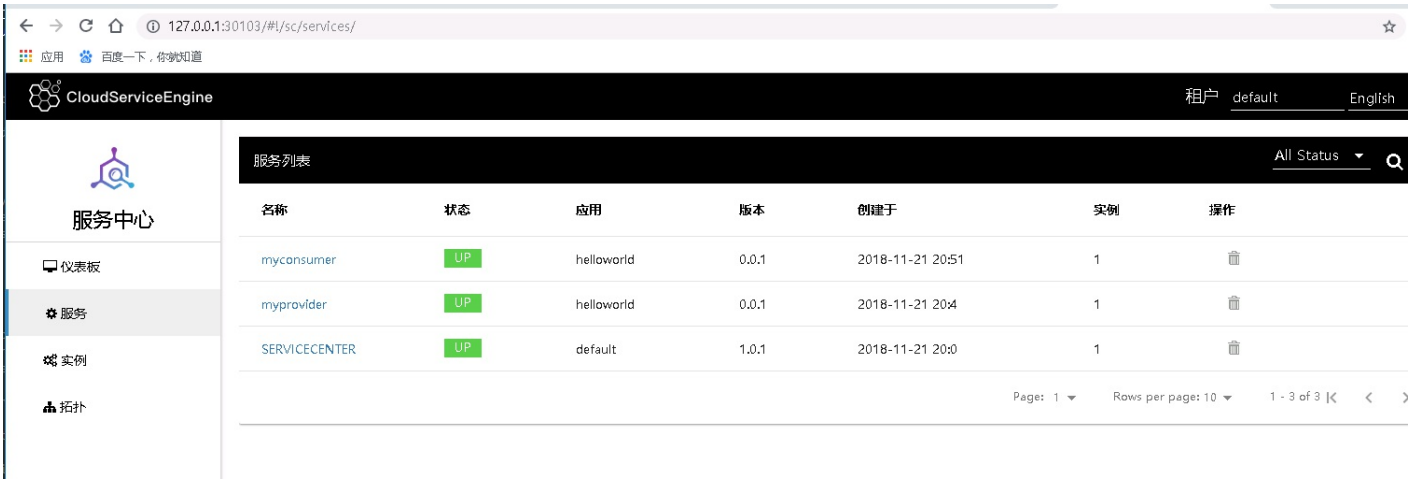
```
@RestSchema(schemaId = "myservice")
```

```
@RequestMapping(path = "/consumer")
@Component
public class MyserviceDelegateImpl implements MyserviceDelegate {

    RestTemplate restTemplate = RestTemplateBuilder.create();

    @RpcReference(microserviceName = "provider", schemaId = "myservice")
    private MyserviceDelegate useMyserviceDelegate;

    @RequestMapping(value = "/show",
            produces = { "application/json" },
            consumes = { "application/json" },
            method = RequestMethod.GET)
    public ForecastSummary show(@RequestParam(value = "city", required = false) String city){

        return useMyserviceDelegate.show(city);
//        return restTemplate.getForObject("cse://provider/forecast/show?city=" + city, ForecastSummary.class);
    }

    @RequestMapping(value = "/extra",
            produces = { "application/json" },
            consumes = { "application/json" },
            method = RequestMethod.POST)
    public String extra( @RequestParam(value = "city", required = true) String city) {
        return useMyserviceDelegate.extra(city);
//        return restTemplate.postForObject("cse://provider/forecast/extra?city=" + city, null, String.class);
    }
}
```

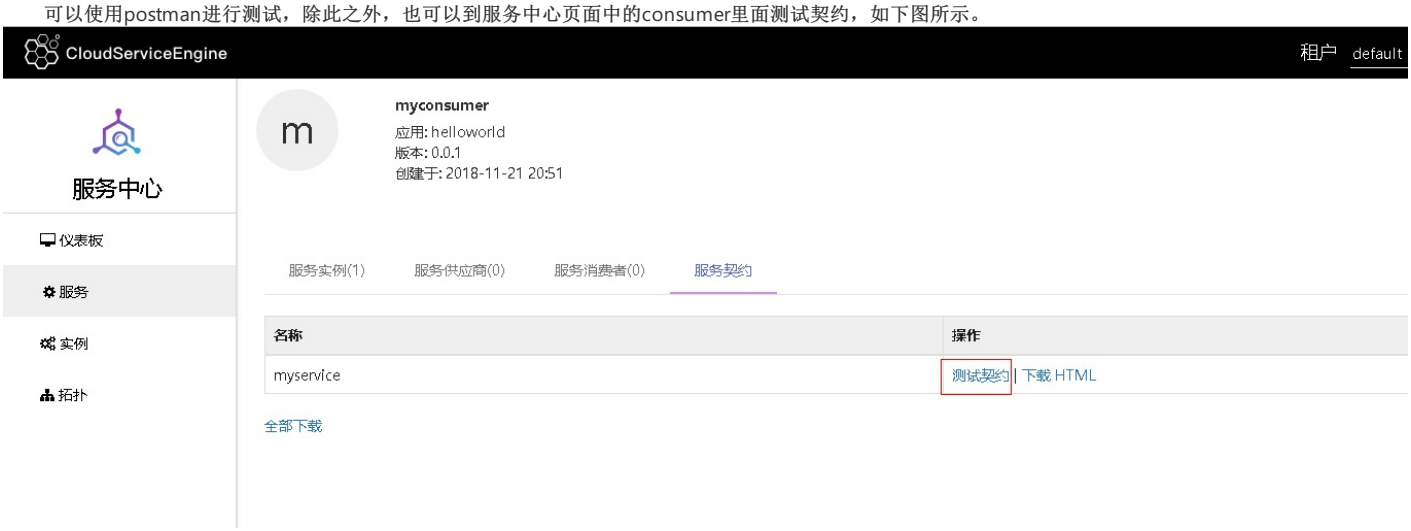
6. 测试服务消费者consumer和服务提供者provider的通信情况

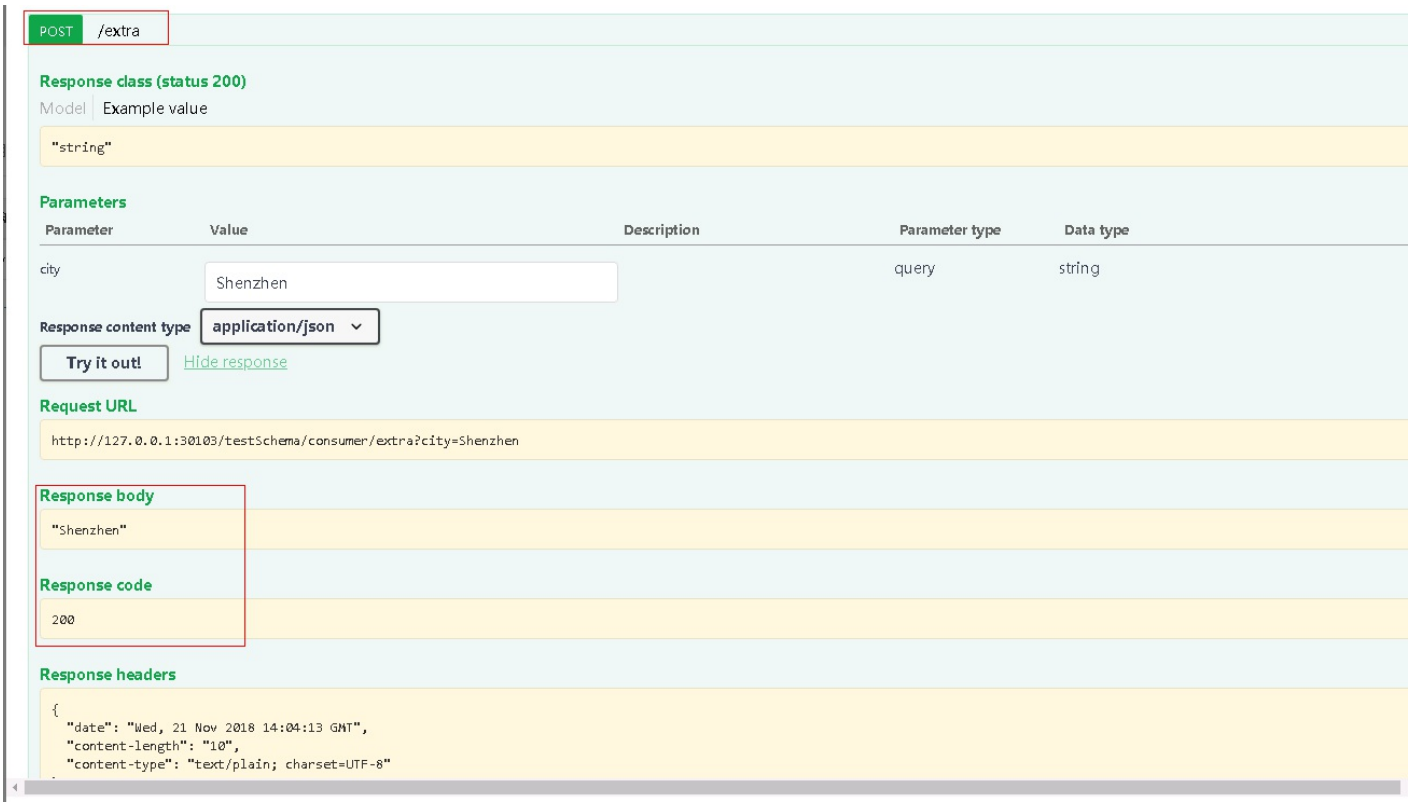
测试consumer和provider是否可以通信，可以下载本地服务中心，如下图所示。

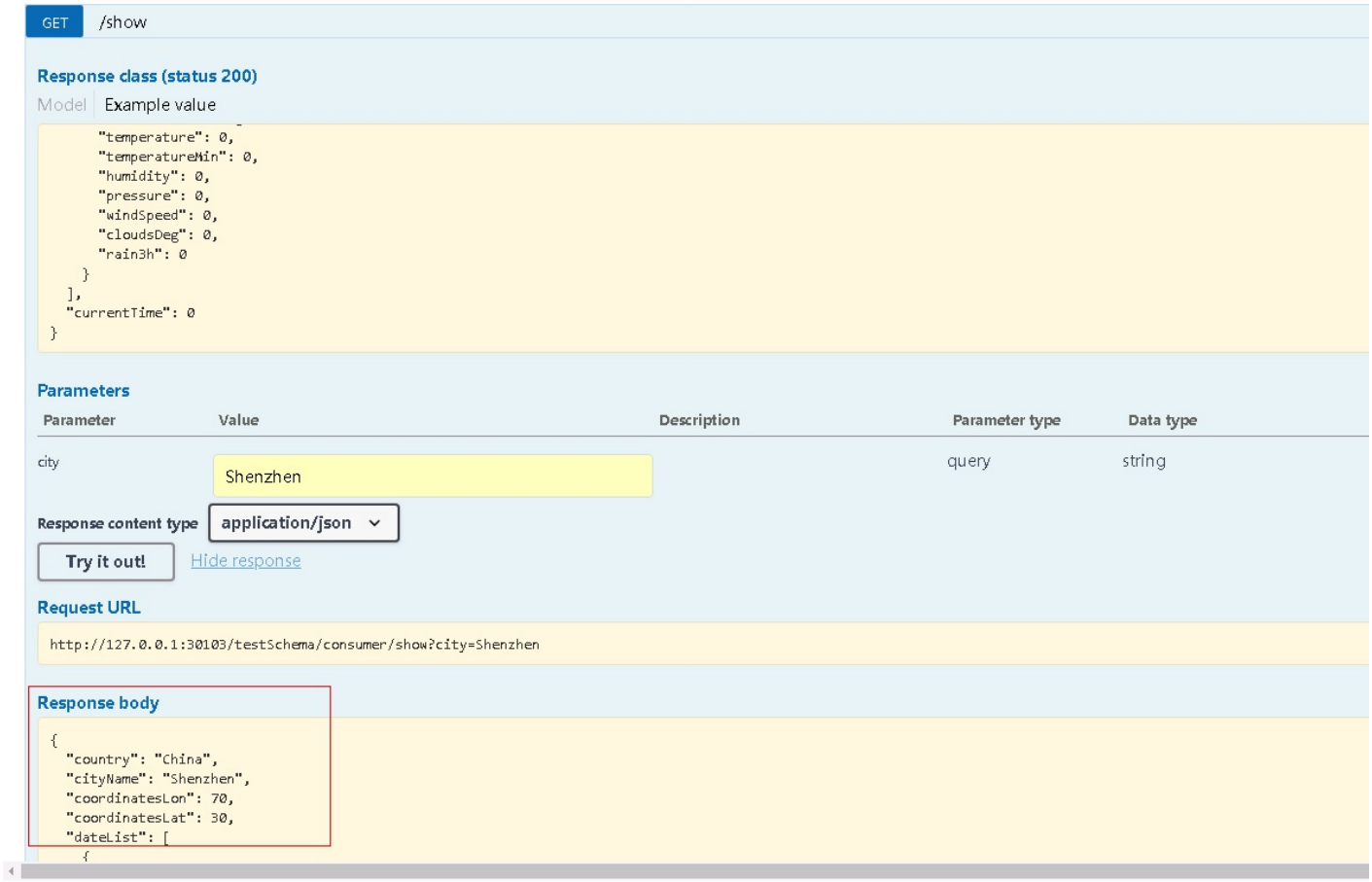
配置好工程里的microservice.yaml，启动服务中心，启动consumer和provider，可以到127.0.0.1:30103访问服务中心，如下图所示。



可以使用postman进行测试，除此之外，也可以到服务中心页面中的consumer里面测试契约，如下图所示。







测试成功，一个简单的微服务就开发完成了。