Hand Enterprise Solutions

HMALL微服务开发手册

作者: 唐磊

建档日期: 2017-04-27

上次更新:

版本:  1.0

2017年04月

上海汉得信息技术股份有限公司



## 文档控制

### 记录更改

| 日期 | 作者 | 版本 | 文档状态 | 更改参考 |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 2017-04-27 | 唐磊 | 1.0 | Draft | No previous version |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

### 审阅人

| 姓名 | 职位 |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

### 分发

| 拷贝号 | 姓名 | 职位 |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

目录

[文档控制 2](#_Toc481076562)

[记录更改 2](#_Toc481076563)

[审阅人 2](#_Toc481076564)

[分发 2](#_Toc481076565)

[1 微服务环境准备 4](#_Toc481076566)

[1.1 配置maven 4](#_Toc481076567)

[1.2 使用intellij导入项目 5](#_Toc481076568)

[1.3 创建微服务的module 8](#_Toc481076569)

[1.4 安装postman 10](#_Toc481076570)

[1.5 微服务概念 14](#_Toc481076571)

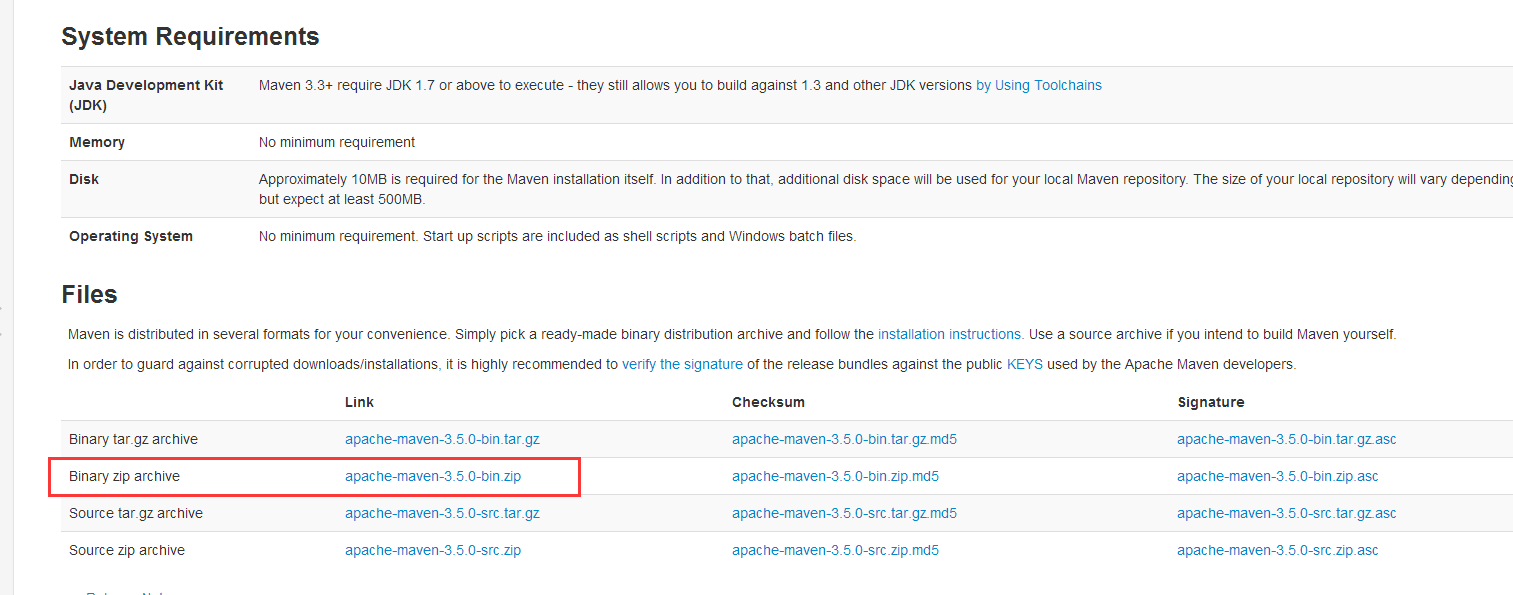
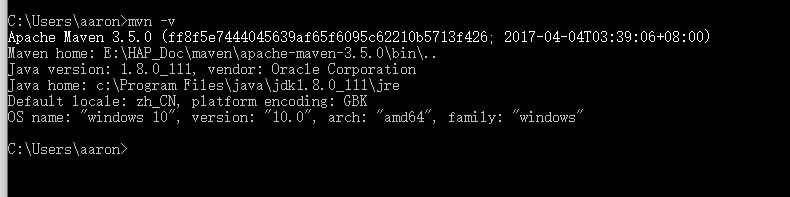
[2 Springboot 16](#_Toc481076572)

[2.1 Springboot快速入门 16](#_Toc481076573)

# 1. 微服务环境准备

特殊说明：java同一采用jdk1.8版本的。

## 1.1 配置maven

1. 首先，先到官网去下载maven。这里是官网的地址：<http://maven.apache.org/download.cgi>。
2. 下载这个版本apache-maven-3.5.0-bin.zip
3. 
4. 解压后，配置环境变量。
5. 新建系统变量 MAVEN\_HOME 变量值：D:\maven\apache-maven-3.5.0
6. 编辑系统变量 Path 添加变量值： ;%MAVEN\_HOME%\bin
7. 测试maven是否安装好
8. 执行mvn -v查看maven的版本
9. 

10. **MVN install下载jar包速度太慢解决方法（引入镜像）**

创建maven项目后，修改pom.xml文件后，运行mvn install时，因为jar包默认是从国外的服务器上下载，所以速度较慢。

解决方法：国内 有一些速度较快的镜像，将镜像导入maven的配置文件，即可快速下载jar包。

具体操作：进入maven安装目录->进入conf文件->打开settings.xml->找到<mirrors> .....</mirrors>，在之间加入

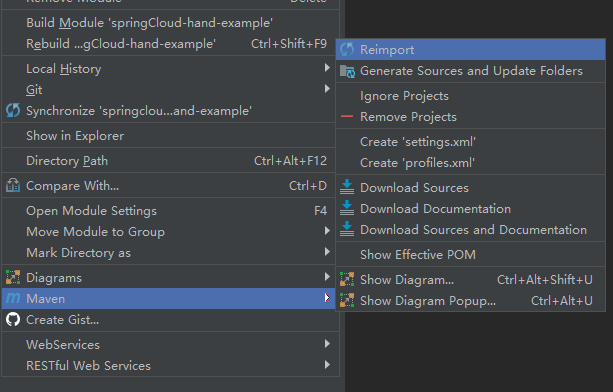
|  |
| --- |
| <mirror>  <id>alimaven</id>  <name>aliyun maven</name>  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>  <mirrorOf>central</mirrorOf>  </mirror>  <mirror>  <id>repo3</id>  <mirrorOf>central</mirrorOf>  <name>Human Readable Name for this Mirror.</name>  <url>https://repo.spring.io/libs-snapshot/</url>  </mirror>  <mirror>  <id>repo2</id>  <mirrorOf>central</mirrorOf>  <name>Human Readable Name for this Mirror.</name>  <url>http://repo2.maven.org/maven2/</url>  </mirror>  <mirror>  <id>repo1</id>  <mirrorOf>central</mirrorOf>  <name>Human Readable Name for this Mirror.</name>  <url>http://repo1.maven.org/maven2/</url>  </mirror> |

四个我测试了都可以，下载速度第一个最快。可以把四个按顺序都添加到<mirrors>.....</mirrors>中间，下载依赖库时默认选第一个。

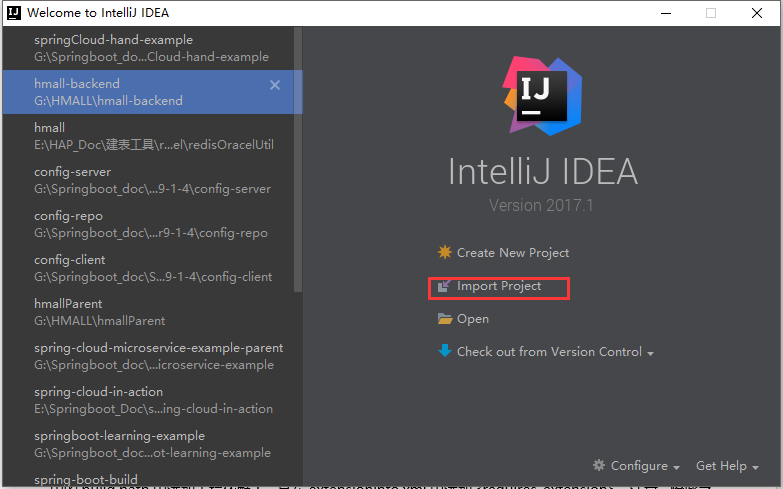
保存，OK。

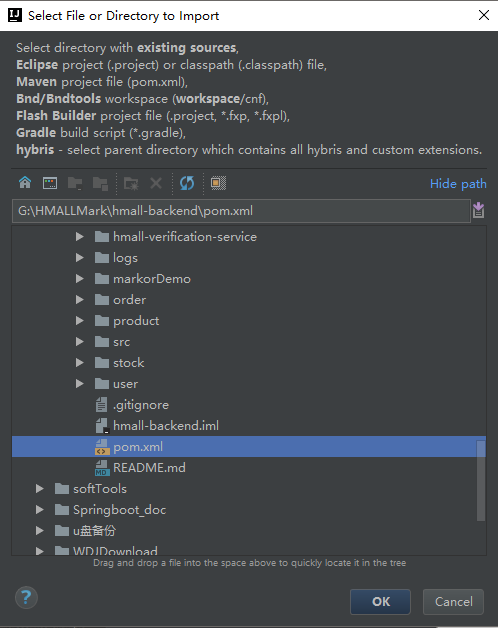
11.使用intellij导入maven的工程的jar包

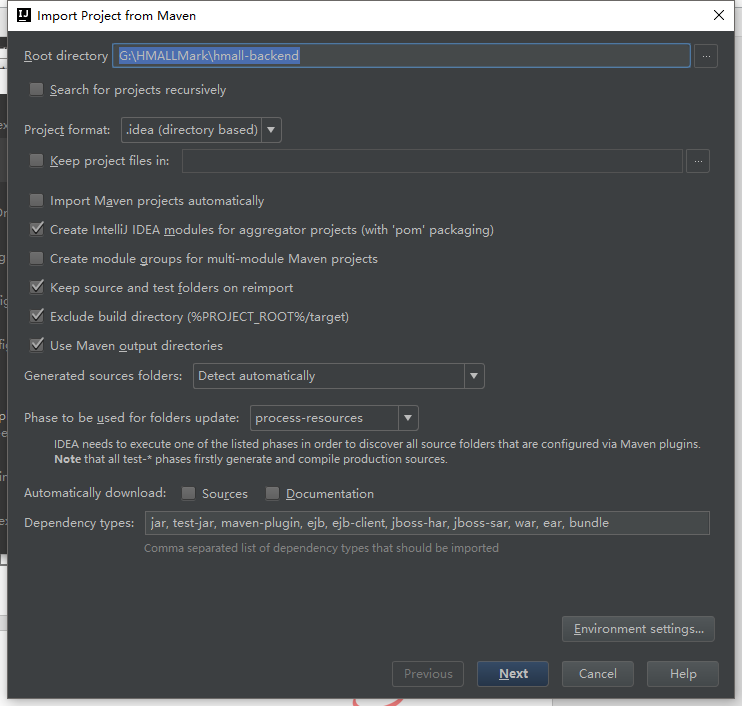
可以使用maven->reimport导入

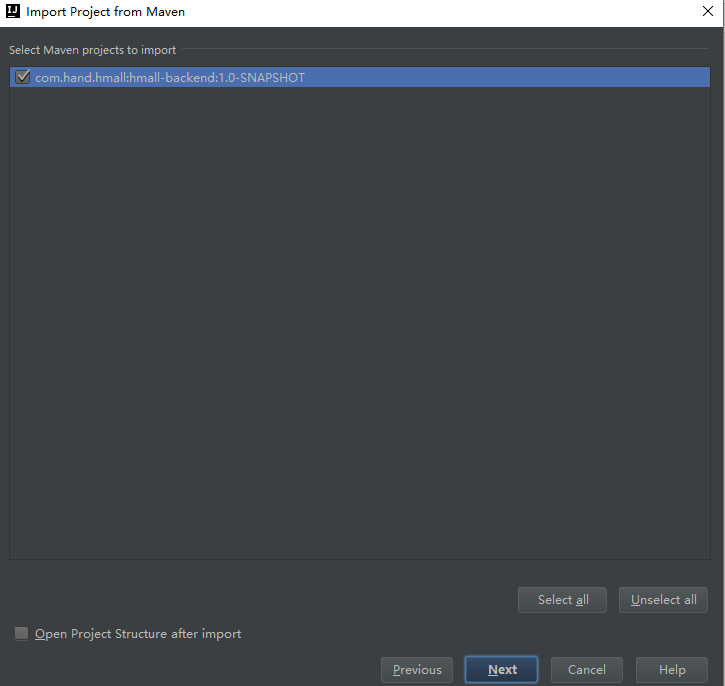


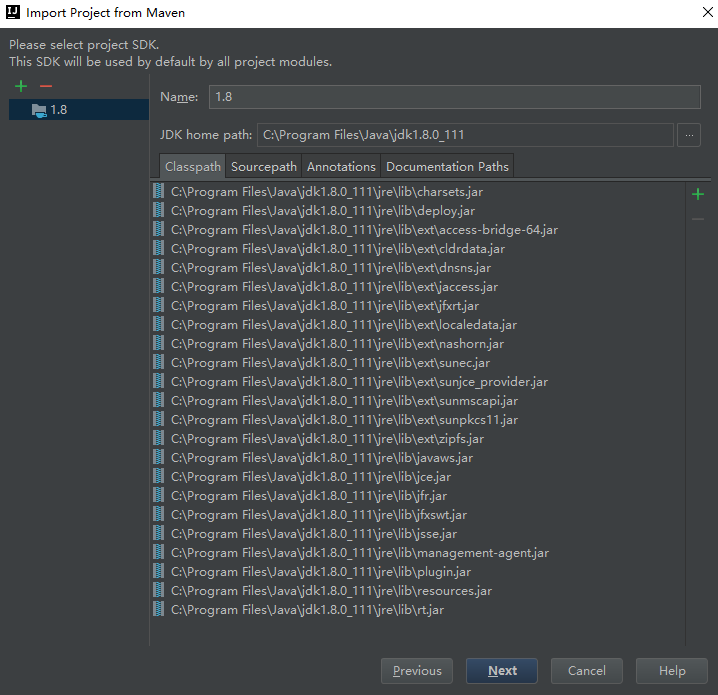
## 1.2 使用intellij导入项目

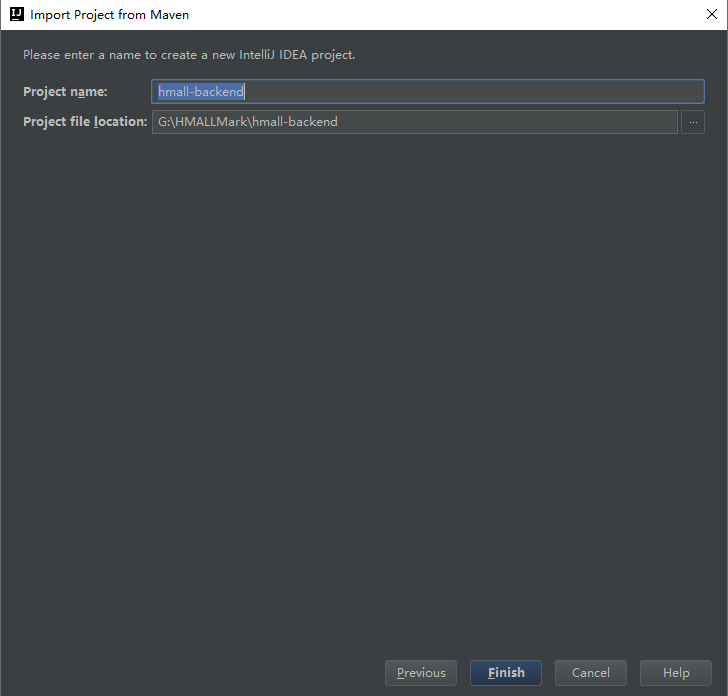
1. 先安装好intellij
2. 

3、

4、

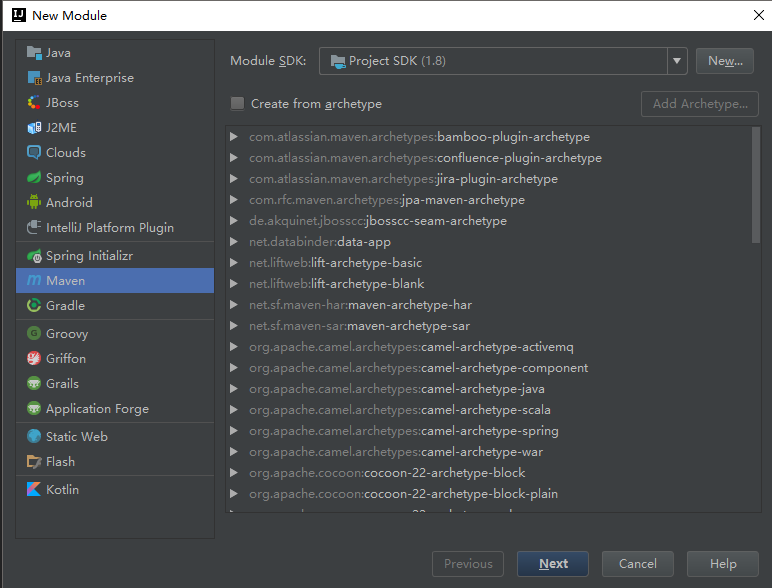
5、

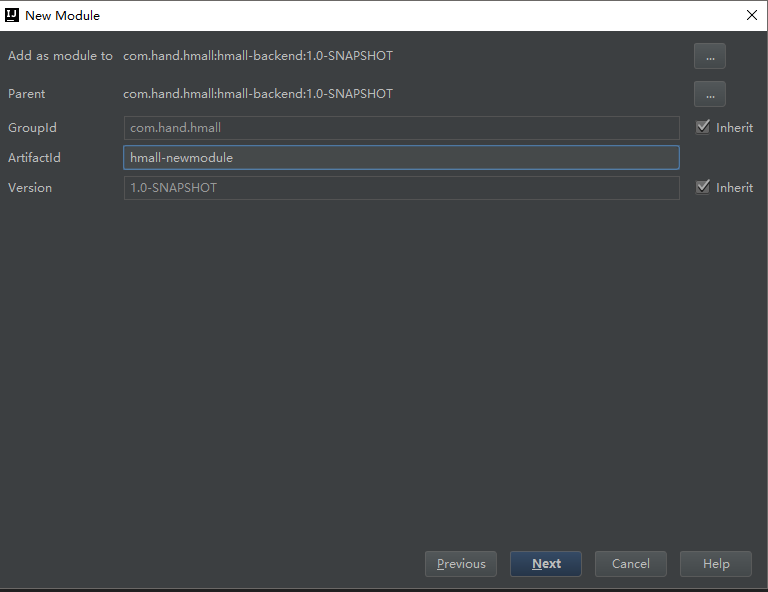
6、

7、

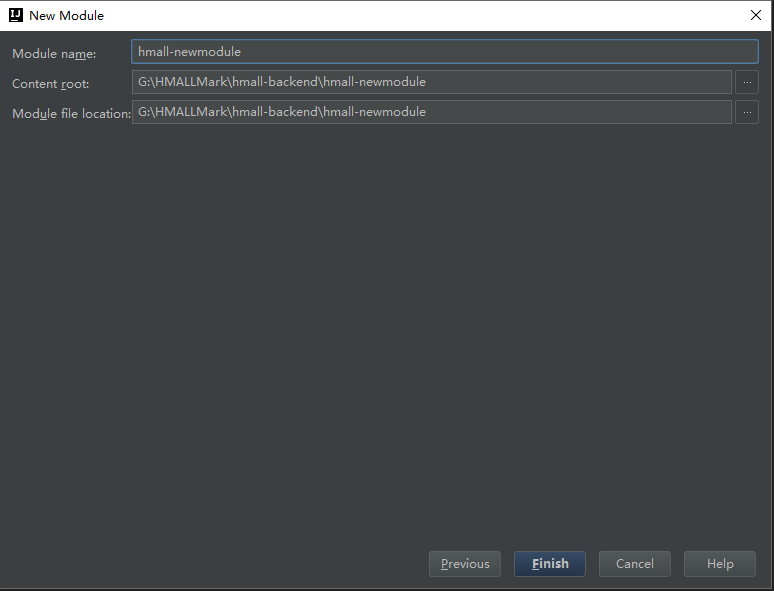
8、导入后，intellij会自动下载项目依赖的jar包。也可以右键->maven->reimport来自动导入jar包。

## 1.3 创建微服务的module

1. 右键项目工程，选择module，选择maven
2. 

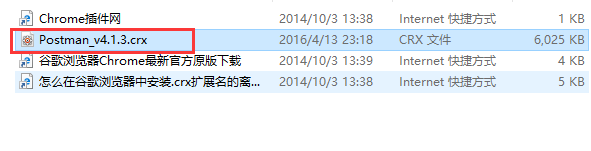
3、在ArifactiId填入微服务的命名。

3、填入module的名字



## 1.4 安装postman

1. 在谷歌浏览器中安装postman的插件，这个插件的作用是测试微服务的接口是否能正常访问。
2. 有两种安装方法，一种是翻墙用谷歌浏览器自带的插件应用商城下载postman插件。另一种方法是安装下载好的插件。
3. 使用第二种方式安装，这个压缩包另存后，解压里面有Postman\_v4.1.3.crx。



1. 打开Chrome，依次选择“选项”>>"更多工具">>“扩展程序”，也可以在地址栏里直接输入：“chrome://extensions/”，进入插件管理页面，直接将Postman\_v4.1.3.crx拖进去就可以安装。使用的时候将postman启用。

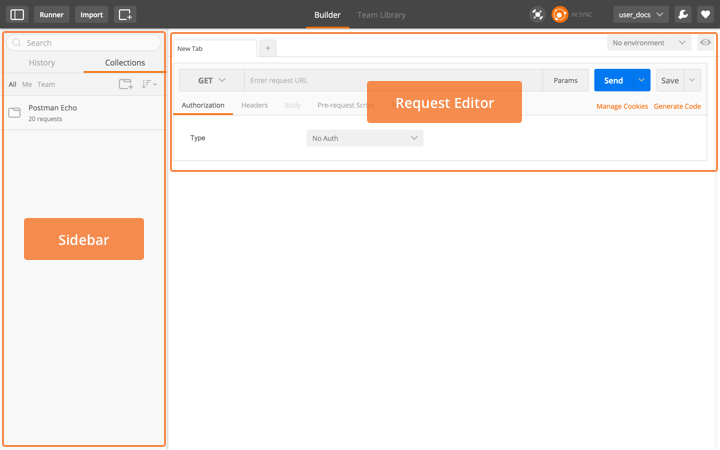


1. 使用Restful请求测试

打开chrome的“应用”，或者直接在地址栏里输入“chrome://apps/”也可以打开应用页面

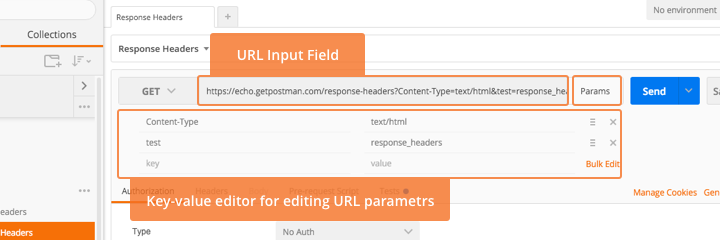
打开postman

安装好之后，我们先打开Postman，可以看到界面分成左右两个部分，右边是我们后头要讲的collection，左边是现在要讲的request builder。在request builder中，我们可以通过Postman快速的随意组装出我们希望的request。一般来说，所有的HTTP Request都分成4个部分，URL, method, headers和body。而Postman针对这几部分都有针对性的工具。



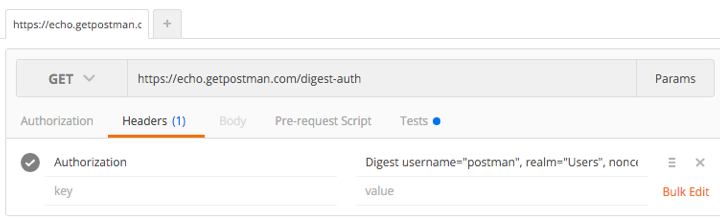
URL

要组装一条Request, URL永远是你首先要填的内容，在Postman里面你曾输入过的URL是可以通过下拉自动补全的哦。如果你点击Params按钮，Postman会弹出一个键值编辑器，你可以在哪里输入URL的Parameter，Postman会帮你自动加入到URL当中，反之，如果你的URL当中已经有了参数，那Postman会在你打开键值编辑器的时候把参数自动载入。



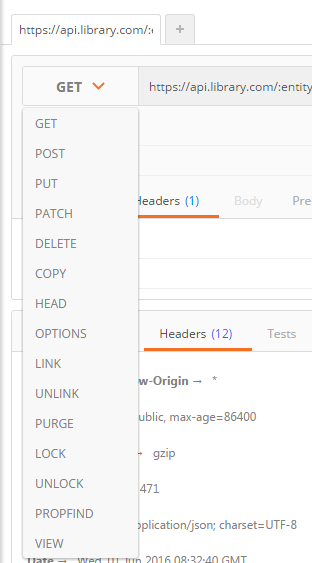
Headers

点击’Headers’按钮，Postman同样会弹出一个键值编辑器。在这里，你可以随意添加你想要的Header attribute，同样Postman为我们通过了很贴心的auto-complete功能，敲入一个字母，你可以从下拉菜单里选择你想要的标准atrribute



Method

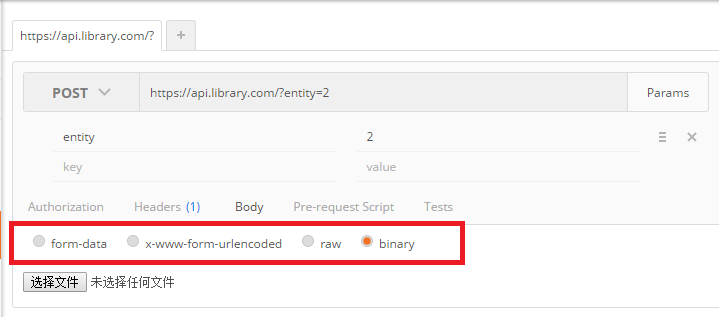
要选择Request的Method是很简单的，Postman支持所有的Method，而一旦你选择了Method，Postman的request body编辑器会根据的你选择，自动的发生改变。



Request Body

如果我们要创建的request是类似于POST，那我们就需要编辑Request Body，Postman根据body type的不同，提供了4中编辑方式：

* form-data
* x-www-form-urlencoded
* raw
* binary

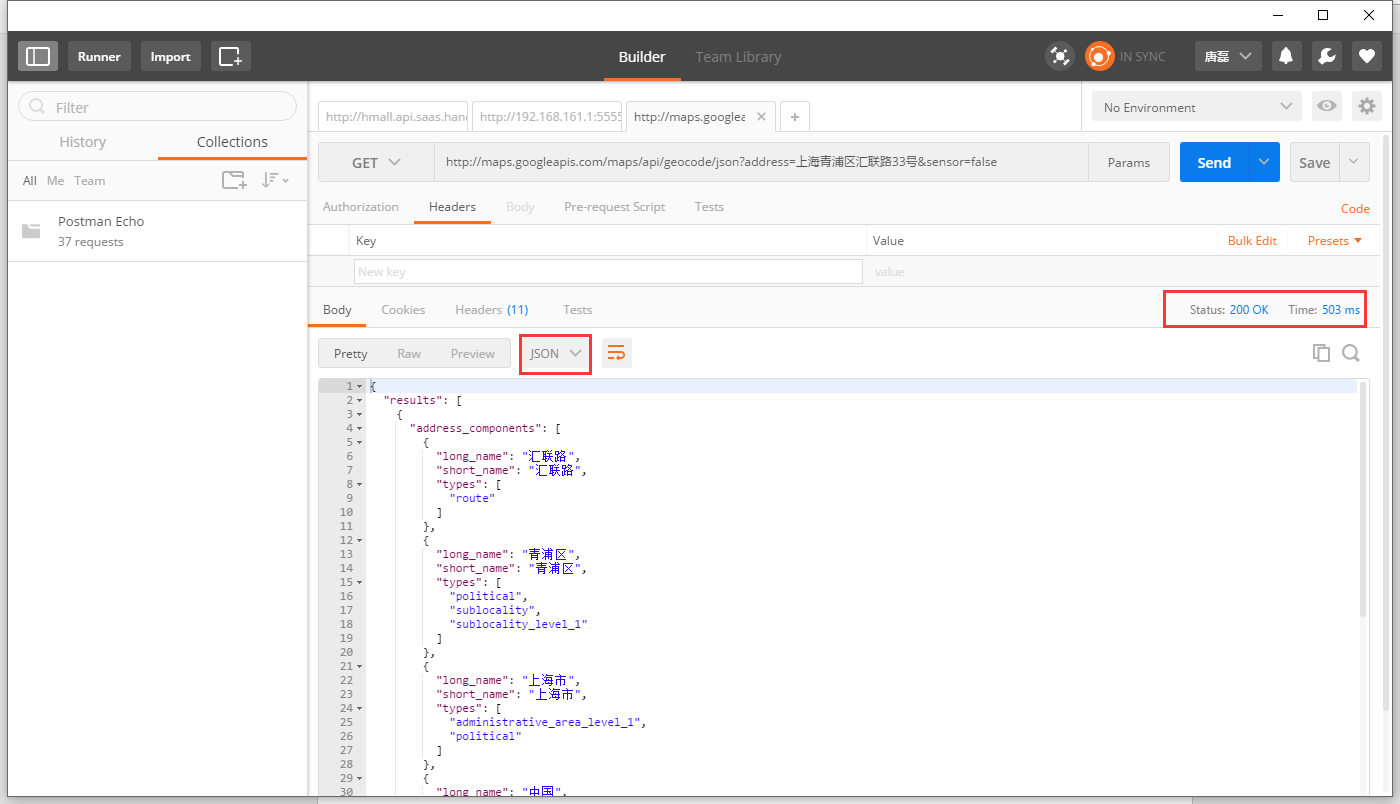


演示

我这里创建一条发送给google geocode的request，看看是啥结果：

请求的url地址是：http://maps.googleapis.com/maps/api/geocode/json?address=上海青浦区汇联路33号&sensor=false

点击了Send之后，可以在Postman上直接看到response的内容，内容很漂亮，Postman根据内容检索自动按JSON的格式显示出来，同时我们可以清楚的看到status code和花费的时间。

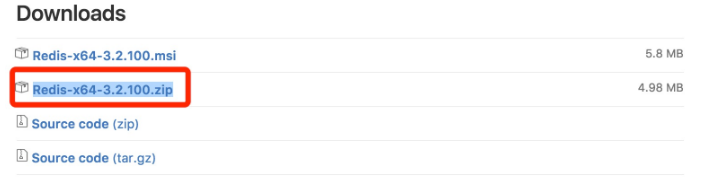


更详细的postman操作可以参照：<http://www.cnplugins.com/tool/specify-postman-methods.html>

## 1.5 redis安装与连接

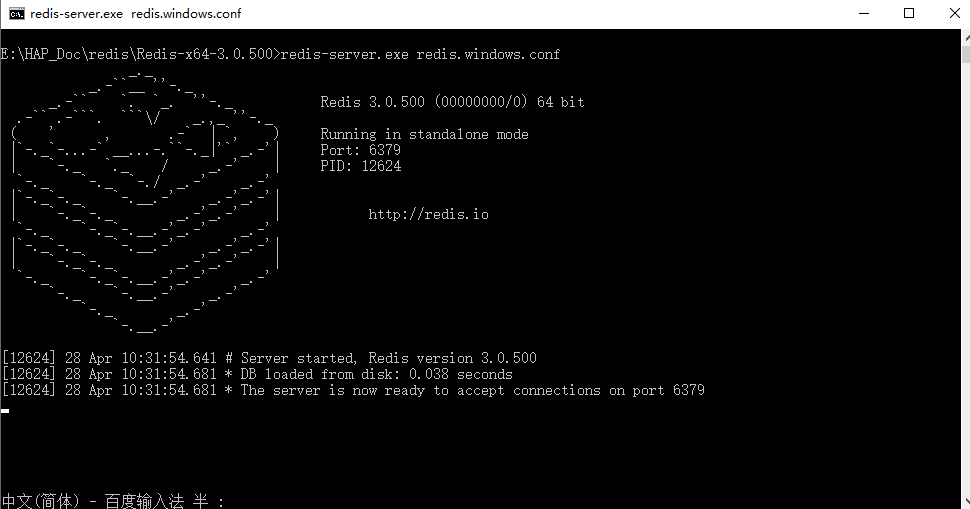
1、下载地址：https://github.com/MSOpenTech/redis/releases。

Redis 支持 32 位和 64 位。这个需要根据你系统平台的实际情况选择，这里我们下载 Redis-x64-xxx.zip压缩包到 C 盘，解压后，将文件夹重新命名为 redis。



打开一个 cmd 窗口 使用cd命令切换目录到 C:\redis 运行 redis-server.exe redis.windows.conf 。

如果想方便的话，可以把 redis 的路径加到系统的环境变量里，这样就省得再输路径了，后面的那个 redis.windows.conf 可以省略，如果省略，会启用默认的。输入之后，会显示如下界面：

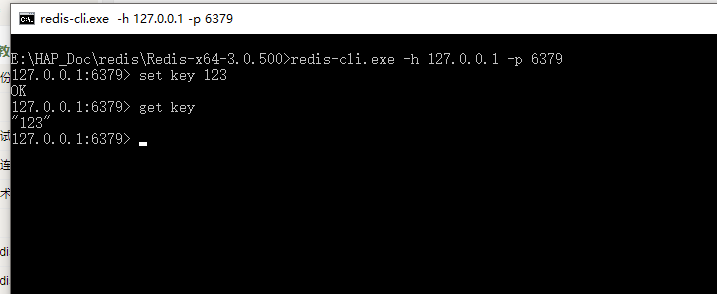


这时候另启一个cmd窗口，原来的不要关闭，不然就无法访问服务端了。

切换到redis目录下运行 redis-cli.exe -h 127.0.0.1 -p 6379 。

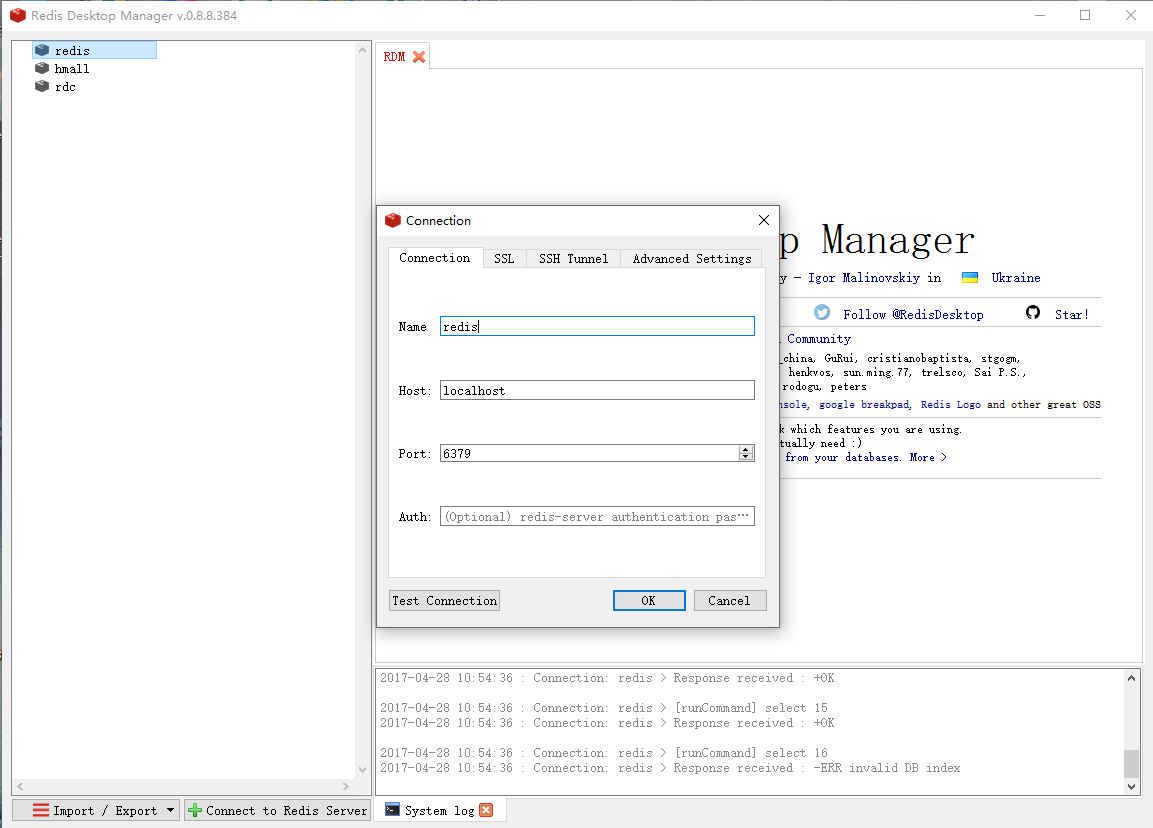
设置键值对 set key 123

取出键值对 get key

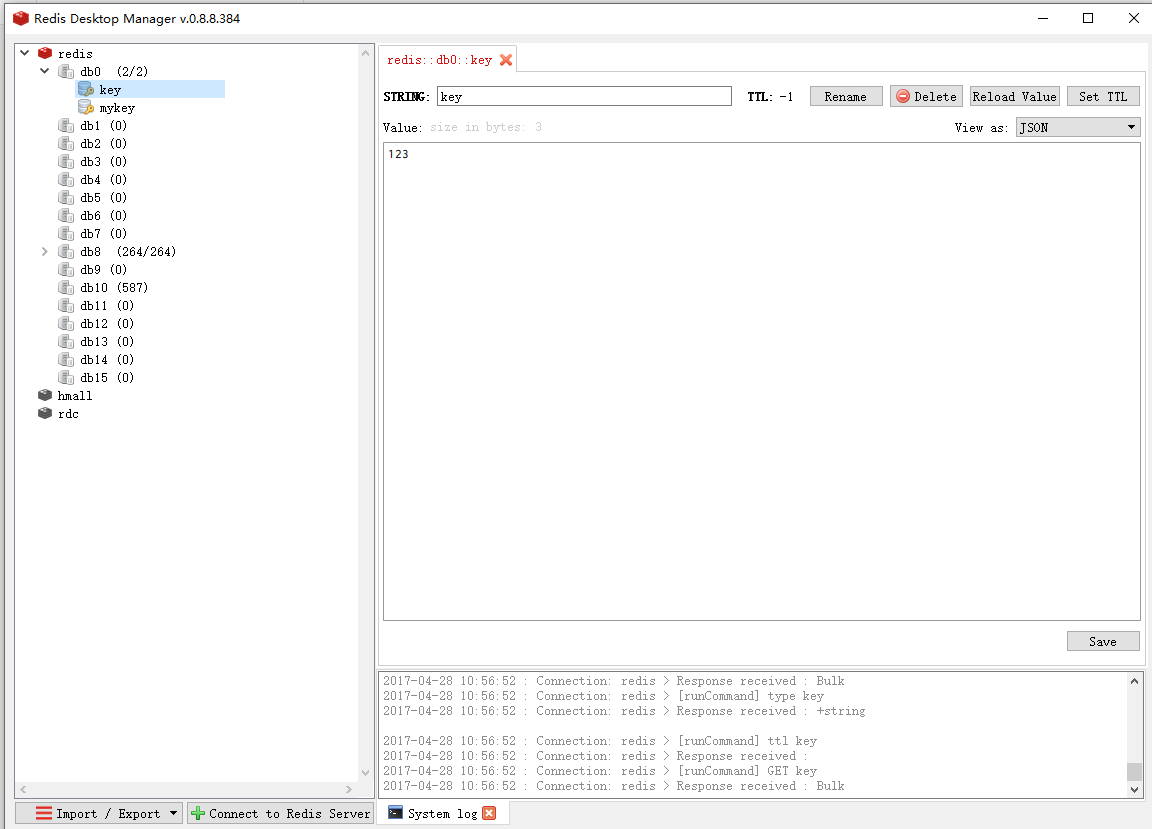


至此，redis安装完毕。

本地安装RedisDesktopManager，下载地址<https://redisdesktop.com/download>，本地连接参照下面的截图。



连接好后，可以进入下面的截图

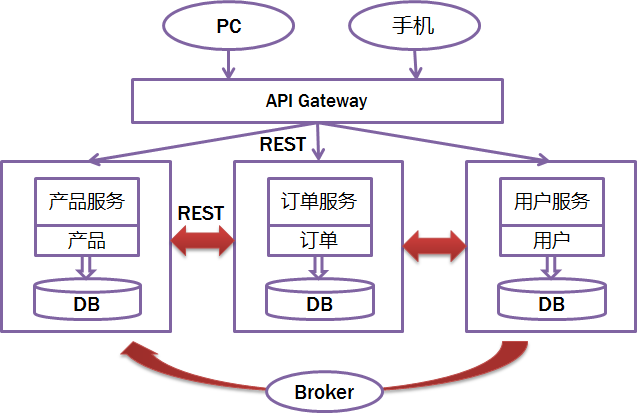


## 1.6微服务概念

1. 先介绍传统的web开发方式，所有的功能打包在一个 WAR包里，基本没有外部依赖（除了容器），部署在一个JEE容器（Tomcat，JBoss，WebLogic）里，包含了 DO/DAO，Service，UI等所有逻辑。



2、微服务的目的是有效的拆分应用，实现敏捷开发和部署 。



一般在后台N个服务和UI之间一般会一个代理或者叫API Gateway，他的作用包括

* 提供统一服务入口，让微服务对前台透明
* 聚合后台的服务，节省流量，提升性能
* 提供安全，过滤，流控等API管理功能

3、服务之间的通讯用RESTful，一般REST基于HTTP，更容易实现，更容易被接受，服务端实现技术也更灵活些，各个语言都能支持，同时能跨客户端，对客户端没有特殊的要求，只要封装了HTTP的SDK就能调用，所以相对使用的广一些。

4、hmall的微服务架构主要用到了springboot和springcloud等。

# 2. Springboot

## 2.1 Springboot快速入门

1、Spring Boot让我们的Spring应用变的更轻量化。比如：你可以仅仅依靠一个Java类来运行一个Spring引用。你也可以打包你的应用为jar并通过使用java -jar来运行你的Spring Web应用。

Spring Boot的主要优点：

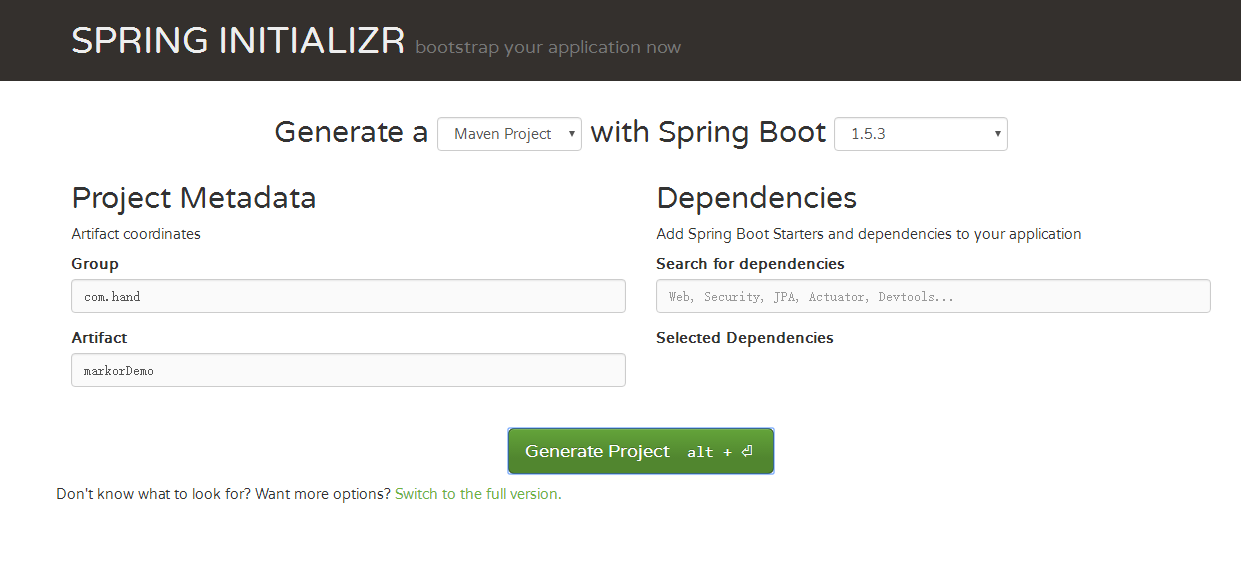
* 为所有Spring开发者更快的入门
* 开箱即用，提供各种默认配置来简化项目配置
* 内嵌式容器简化Web项目
* 没有冗余代码生成和XML配置的要求

本小节的目标完成Spring Boot基础项目的构建，并且实现一个简单的Http请求处理，通过这个例子Spring Boot有一个初步的了解，并体验其结构简单、开发快速的特性。

2、通过SPRING INITIALIZR工具产生基础项目

访问：http://start.spring.io/

选择构建工具Maven Project、Spring Boot版本1.3.2以及一些工程基本信息，可参考下图所示SPRING INITIALIZR



SPRING INITIALIZR

点击Generate Project下载项目压缩包

解压项目包，并用IDE以Maven项目导入，以IntelliJ IDEA 为例：

菜单中选择File–>New–>Project from Existing Sources...

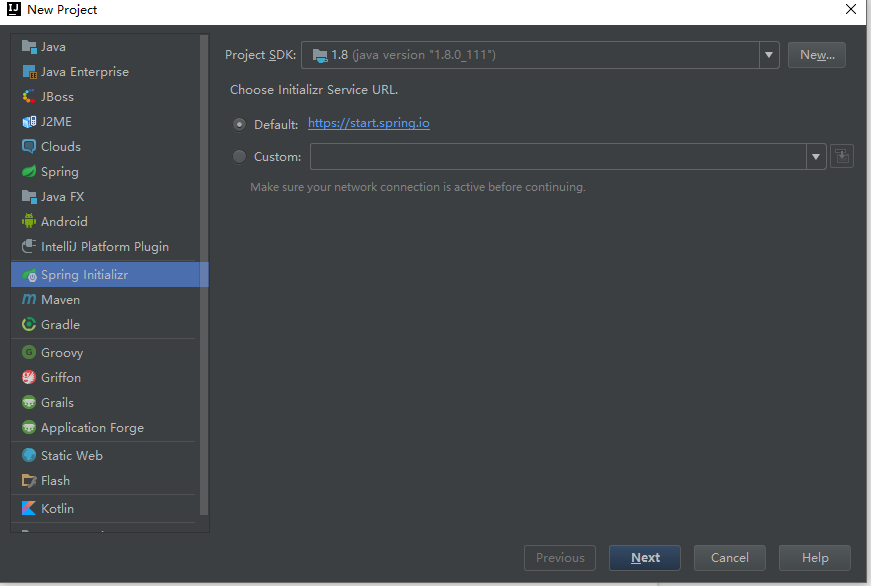
选择解压后的项目文件夹，点击OK

点击Import project from external model并选择Maven，点击Next到底为止。

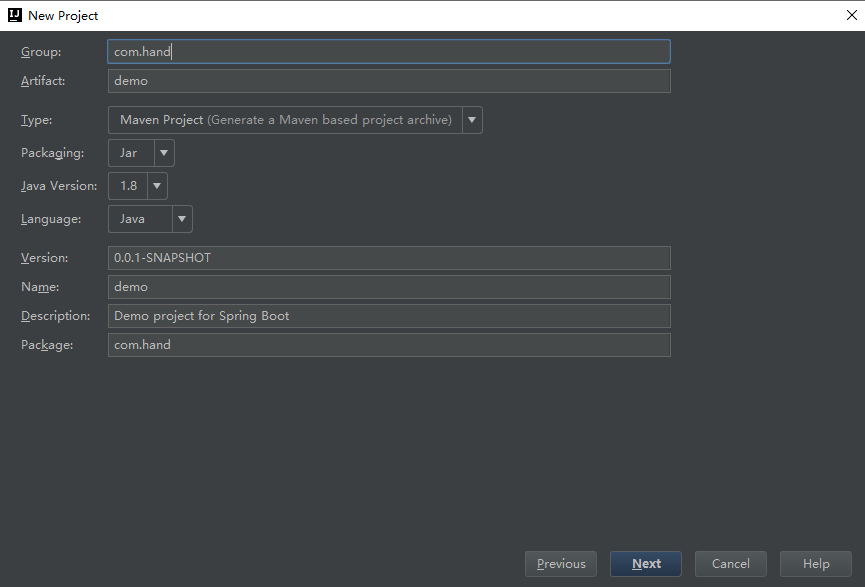
若你的环境有多个版本的JDK，注意到选择Java SDK的时候请选择Java 8的版本

1. 使用使用Intellij中的Spring Initializr来快速构建Spring Boot/Cloud工程

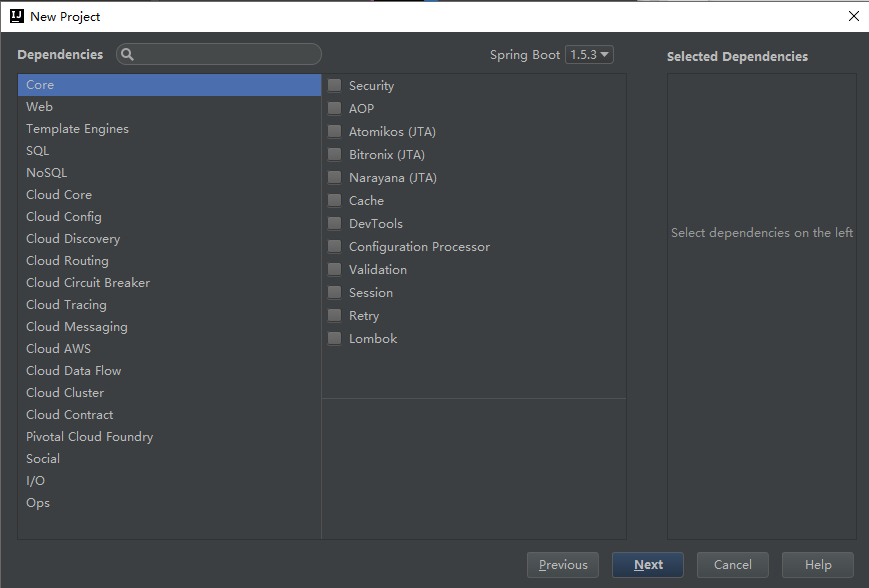
菜单栏中选择File=>New=>Project..，我们可以看到如下图所示的创建功能窗口。其中Initial Service Url指向的地址就是Spring官方提供的Spring Initializr工具地址，所以这里创建的工程实际上也是基于它的Web工具来实现的。



点击Next，等待片刻后，我们可以看到如下图所示的工程信息窗口，在这里我们可以编辑我们想要创建的工程信息。其中，Type可以改变我们要构建的工程类型，比如：Maven；Language可以选择：Java。

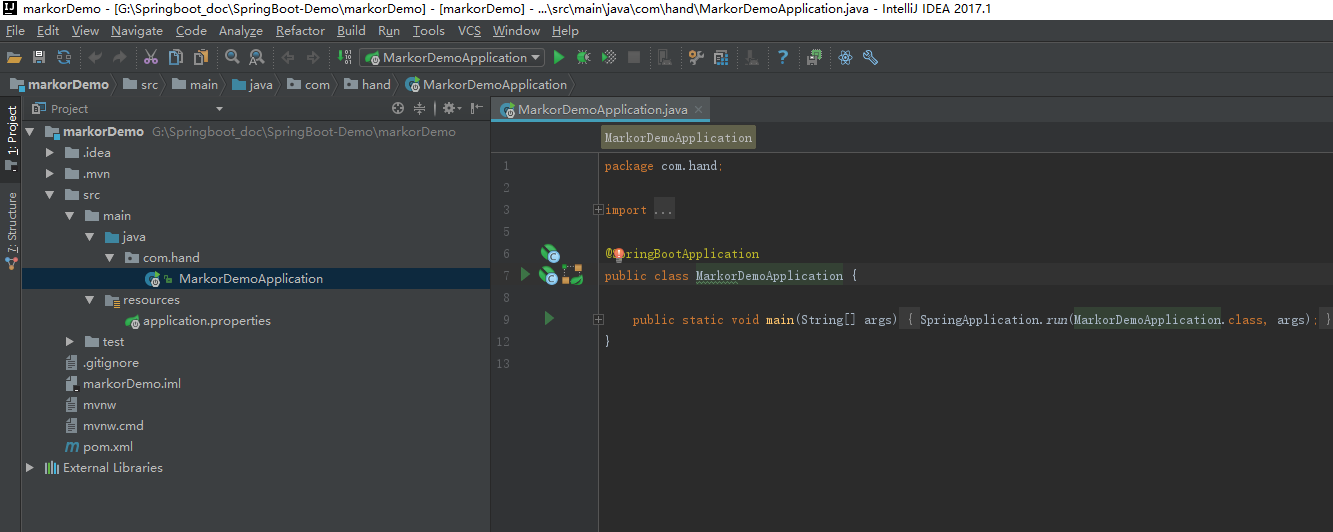


点击Next，进入选择Spring Boot版本和依赖管理的窗口。在这里值的我们关注的是，它不仅包含了Spring Boot Starter POMs中的各个依赖，还包含了Spring Cloud的各种依赖。



点击Next，进入最后关于工程物理存储的一些细节。最后，点击Finish就能完成工程的构建了。ntellij中的Spring Initializr虽然还是基于官方Web实现，但是通过工具来进行调用并直接将结果构建到我们的本地文件系统中，让整个构建流程变得更加顺畅

1. 项目结构解析



通过上面步骤完成了基础项目的创建，如上图所示，Spring Boot的基础结构共三个文件（具体路径根据用户生成项目时填写的Group所有差异）：

src/main/java下的程序入口：MarkorDemoApplication

src/main/resources下的配置文件：application.properties

src/test/下的测试入口：MarkorDemoApplicationTests

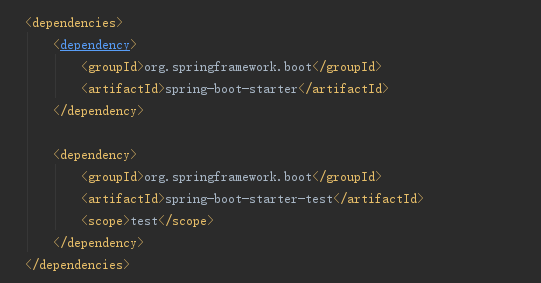
生成的MarkorDemoApplication和MarkorDemoApplicationTests类都可以直接运行来启动当前创建的项目，由于目前该项目未配合任何数据访问或Web模块，程序会在加载完Spring之后结束运行。

3、引入Web模块

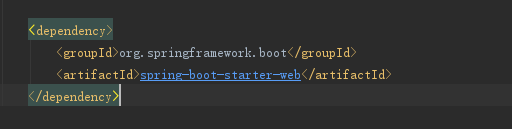
当前的pom.xml内容如下，仅引入了两个模块：

spring-boot-starter：核心模块，包括自动配置支持、日志和YAML

spring-boot-starter-test：测试模块，包括JUnit、Hamcrest、Mockito



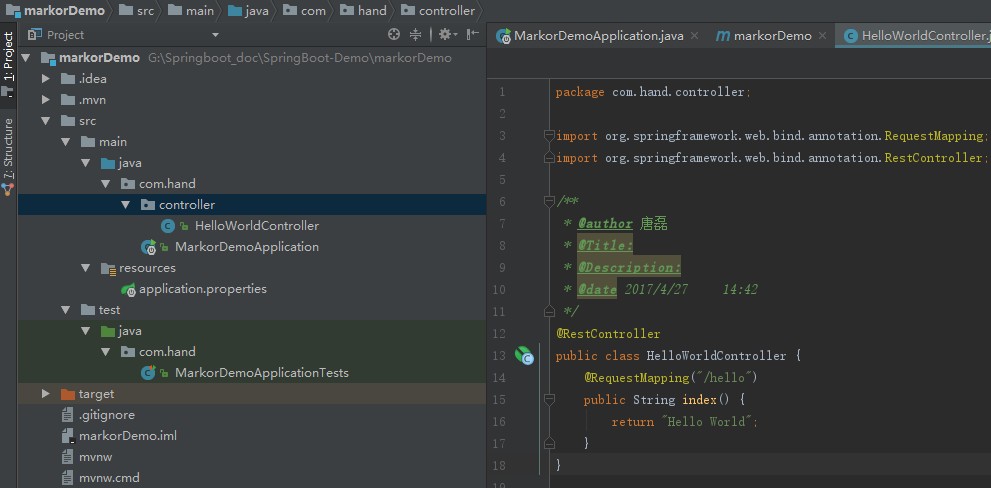
引入Web模块，需添加spring-boot-starter-web模块：



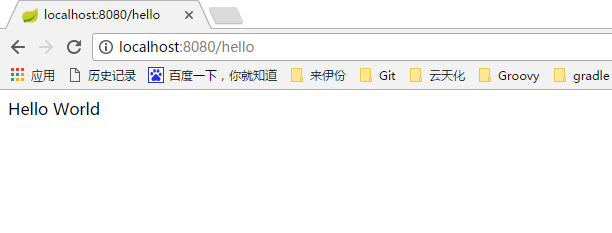
4、编写HelloWorld服务

创建package命名为com.hand.controller（根据实际情况修改）

创建HelloWorldController类，内容如下

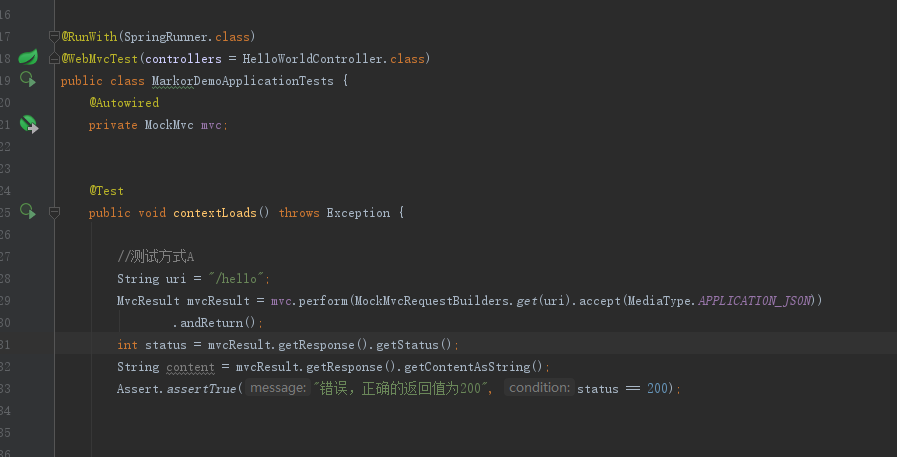


启动主程序，打开浏览器访问http://localhost:8080/hello，可以看到页面输出Hello World



5、编写单元测试用例

打开的src/test/下的测试入口MarkorDemoApplicationTests类。下面编写一个简单的单元测试来模拟http请求，具体如下：



@RunWith(SpringRunner.class) 告诉JUnit运行使用Spring的测试支持

@WebMvcTest注解来测试应用程序的Spring MVC插件

## 2.2 Spring Boot构建RESTful API与单元测试

1. 首先详细说明一下在快速入门中使用的@Controller、@RestController、@RequestMapping注解。

@Controller：修饰class，用来创建处理http请求的对象

@RestController：Spring4之后加入的注解，原来在@Controller中返回json需要@ResponseBody来配合，如果直接用@RestController替代@Controller就不需要再配置@ResponseBody，默认返回json格式。

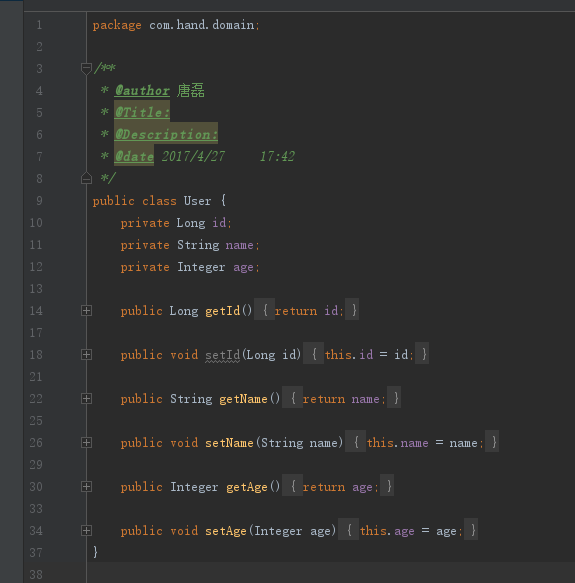
@RequestMapping：配置url映射

2、下面我们尝试使用Spring MVC来实现一组对User对象操作的RESTful API，配合注释详细说明在Spring MVC中如何映射HTTP请求、如何传参、如何编写单元测试。

RESTful API具体设计如下：



3、User实体定义：



4、实现对User对象的操作接口

|  |
| --- |
| package com.hand.controller;  import com.hand.domain.User; import org.springframework.web.bind.annotation.\*;  import java.util.\*;  */\*\*  \** ***@author*** *唐磊  \** ***@Title:*** *\** ***@Description:*** *\** ***@date*** *2017/4/27 17:43  \*/* @RestController @RequestMapping(value="/users") // 通过这里配置使下面的映射都在/users下 public class UserController {  static Map<Long, User> *users* = Collections.*synchronizedMap*(new HashMap<Long, User>());   @RequestMapping(value="/", method= RequestMethod.*GET*)  public List<User> getUserList() {  // 处理"/users/"的GET请求，用来获取用户列表  // 还可以通过@RequestParam从页面中传递参数来进行查询条件或者翻页信息的传递  List<User> r = new ArrayList<User>(*users*.values());  return r;  }   @RequestMapping(value="/", method=RequestMethod.*POST*)  public String postUser(@ModelAttribute User user) {  // 处理"/users/"的POST请求，用来创建User  // 除了@ModelAttribute绑定参数之外，还可以通过@RequestParam从页面中传递参数  *users*.put(user.getId(), user);  return "success";  }   @RequestMapping(value="/{id}", method=RequestMethod.*GET*)  public User getUser(@PathVariable Long id) {  // 处理"/users/{id}"的GET请求，用来获取url中id值的User信息  // url中的id可通过@PathVariable绑定到函数的参数中  return *users*.get(id);  }   @RequestMapping(value="/{id}", method=RequestMethod.*PUT*)  public String putUser(@PathVariable Long id, @ModelAttribute User user) {  // 处理"/users/{id}"的PUT请求，用来更新User信息  User u = *users*.get(id);  u.setName(user.getName());  u.setAge(user.getAge());  *users*.put(id, u);  return "success";  }   @RequestMapping(value="/{id}", method=RequestMethod.*DELETE*)  public String deleteUser(@PathVariable Long id) {  // 处理"/users/{id}"的DELETE请求，用来删除User  *users*.remove(id);  return "success";  } } |

至此，我们通过引入web模块（没有做其他的任何配置），就可以轻松利用Spring MVC的功能，以非常简洁的代码完成了对User对象的RESTful API的创建以及单元测试的编写。其中同时介绍了Spring MVC中最为常用的几个核心注解：@Controller,@RestController,RequestMapping以及一些参数绑定的注解：@PathVariable,@ModelAttribute,@RequestParam等。当使用@RequestMapping URI template 样式映射时， 即 someUrl/{paramId}, 这时的paramId可通过 @Pathvariable注解绑定它传过来的值到方法的参数上。

1. 下面针对该Controller编写测试用例验证正确性，具体如下。当然也可以通过浏览器插件等进行请求提交验证。

|  |
| --- |
| 1. package com.hand;  import com.hand.controller.HelloController; import com.hand.controller.UserController; import org.junit.Before; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.boot.test.web.client.TestRestTemplate; import org.springframework.http.MediaType; import org.springframework.test.context.junit4.SpringRunner; import org.springframework.test.web.servlet.MockMvc; import org.springframework.test.web.servlet.RequestBuilder; import org.springframework.test.web.servlet.request.MockMvcRequestBuilders; import org.springframework.test.web.servlet.setup.MockMvcBuilders;  import java.net.URL;  import static org.hamcrest.Matchers.*equalTo*; import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*; import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*get*; import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*content*; import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*status*;  @RunWith(SpringRunner.class) @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.*RANDOM\_PORT*) public class DemoApplicationTests {   private MockMvc mvc;   @Autowired  private TestRestTemplate template;   @Before  public void setUp() throws Exception {  mvc = MockMvcBuilders.*standaloneSetup*(  new HelloController(),  new UserController()).build();  }   @Test  public void getHello() throws Exception {  mvc.perform(MockMvcRequestBuilders.*get*("/hello").accept(MediaType.*APPLICATION\_JSON*))  .andExpect(*status*().isOk())  .andExpect(*content*().string(*equalTo*("Hello World")));  }  @Test  public void testUserController() throws Exception { // 测试UserController  RequestBuilder request = null;   // 1、get查一下user列表，应该为空  request = *get*("/users/");  mvc.perform(request)  .andExpect(*status*().isOk())  .andExpect(*content*().string(*equalTo*("[]")));   // 2、post提交一个user  request = *post*("/users/")  .param("id", "1")  .param("name", "测试大师")  .param("age", "20");  mvc.perform(request) // .andDo(MockMvcResultHandlers.print())  .andExpect(*content*().string(*equalTo*("success")));   // 3、get获取user列表，应该有刚才插入的数据  request = *get*("/users/");  mvc.perform(request)  .andExpect(*status*().isOk())  .andExpect(*content*().string(*equalTo*("[{\"id\":1,\"name\":\"测试大师\",\"age\":20}]")));   // 4、put修改id为1的user  request = *put*("/users/1")  .param("name", "测试终极大师")  .param("age", "30");  mvc.perform(request)  .andExpect(*content*().string(*equalTo*("success")));   // 5、get一个id为1的user  request = *get*("/users/1");  mvc.perform(request)  .andExpect(*content*().string(*equalTo*("{\"id\":1,\"name\":\"测试终极大师\",\"age\":30}")));   // 6、del删除id为1的user  request = *delete*("/users/1");  mvc.perform(request)  .andExpect(*content*().string(*equalTo*("success")));   // 7、get查一下user列表，应该为空  request = *get*("/users/");  mvc.perform(request)  .andExpect(*status*().isOk())  .andExpect(*content*().string(*equalTo*("[]")));   }  } |

# 3. SpringCloud构建微服务架构

## 3.1 SpringCloud简介

Spring Cloud是一个基于Spring Boot实现的云应用开发工具，详细官方文档：<https://springcloud.cc/> 。

微服务架构

“微服务架构”在这几年非常的火热，以至于关于微服务架构相关的产品社区也变得越来越活跃（比如：netflix、dubbo），Spring Cloud也因Spring社区的强大知名度和影响力也被广大架构师与开发者备受关注。

那么什么是“微服务架构”呢？简单的说，微服务架构就是将一个完整的应用从数据存储开始垂直拆分成多个不同的服务，每个服务都能独立部署、独立维护、独立扩展，服务与服务间通过诸如RESTful API的方式互相调用。

对于“微服务架构”，大家在互联网可以搜索到很多相关的介绍和研究文章来进行学习和了解。也可以阅读始祖Martin Fowler的《Microservices》，本文不做更多的介绍和描述。

## 3.2 服务注册与发现

使用Spring Cloud搭建服务注册与发现模块。

这里我们会用到Spring Cloud Netflix，该项目是Spring Cloud的子项目之一，主要内容是对Netflix公司一系列开源产品的包装，它为Spring Boot应用提供了自配置的Netflix OSS整合。通过一些简单的注解，开发者就可以快速的在应用中配置一下常用模块并构建庞大的分布式系统。它主要提供的模块包括：服务发现（Eureka），断路器（Hystrix），智能路有（Zuul），客户端负载均衡（Ribbon）等。所以，我们这里的核心内容就是服务发现模块：Eureka

#### 3.2.1 创建“服务注册中心”

创建一个基础的Spring Boot工程，并在pom.xml中引入需要的依赖内容：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.3.5.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Brixton.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

通过@EnableEurekaServer注解启动一个服务注册中心提供给其他应用进行对话。这一步非常的简单，只需要在一个普通的Spring Boot应用中添加这个注解就能开启此功能，比如下面的例子：

|  |
| --- |
| @EnableEurekaServer  @SpringBootApplication  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  **new** SpringApplicationBuilder(Application.class).web(**true**).run(args);  }  } |

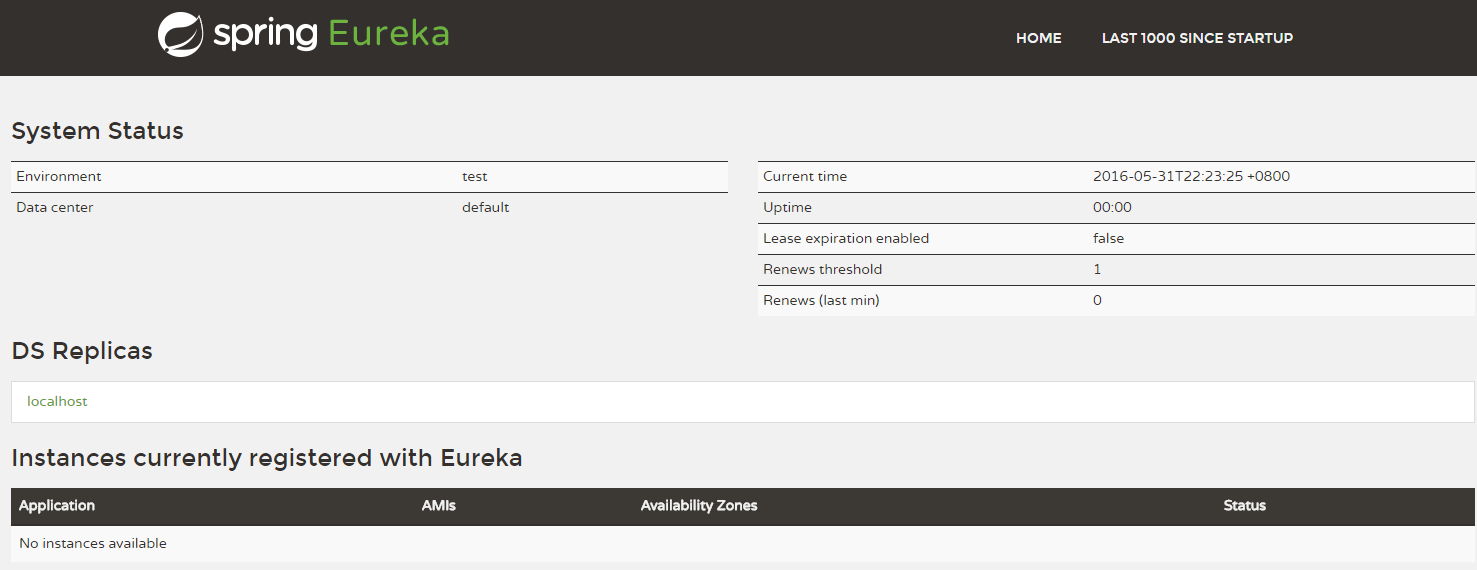
在默认设置下，该服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为，只需要在application.properties中问增加如下配置：

|  |
| --- |
| server.port=1111  eureka.client.register-with-eureka=**false**  eureka.client.fetch-registry=**false**  eureka.client.serviceUrl.defaultZone=http://localhost:${server.port}/eureka/ |

为了与后续要进行注册的服务区分，这里将服务注册中心的端口通过server.port属性设置为1111。

启动工程后，访问：http://localhost:1111/

可以看到下面的页面，其中还没有发现任何服务



#### 3.2.2 创建“服务提供方”

下面我们创建提供服务的客户端，并向服务注册中心注册自己。

假设我们有一个提供计算功能的微服务模块，我们实现一个RESTful API，通过传入两个参数a和b，最后返回a + b的结果。

首先，创建一个基本的Spring Boot应用，在pom.xml中，加入如下配置：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.3.5.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Brixton.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

其次，实现/add请求处理接口，通过DiscoveryClient对象，在日志中打印出服务实例的相关内容。

|  |
| --- |
| @RestController  **public** **class** **ComputeController** {  **private** **final** Logger logger = Logger.getLogger(getClass());  @Autowired  **private** DiscoveryClient client;  @RequestMapping(value = "/add" ,method = RequestMethod.GET)  **public** Integer **add**(@RequestParam Integer a, @RequestParam Integer b) {  ServiceInstance instance = client.getLocalServiceInstance();  Integer r = a + b;  logger.info("/add, host:" + instance.getHost() + ", service\_id:" + instance.getServiceId() + ", result:" + r);  **return** r;  }  } |

最后在主类中通过加上@EnableDiscoveryClient注解，该注解能激活Eureka中的DiscoveryClient实现，才能实现Controller中对服务信息的输出。

|  |
| --- |
| @EnableDiscoveryClient  @SpringBootApplication  **public** **class** **ComputeServiceApplication** {  **public** **static** **void** **main**(String[] args) {  **new** SpringApplicationBuilder(ComputeServiceApplication.class).web(**true**).run(args);  }  } |

我们在完成了服务内容的实现之后，再继续对application.properties做一些配置工作，具体如下：

|  |
| --- |
| spring.application.name=compute-service  server.port=2222  eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/ |

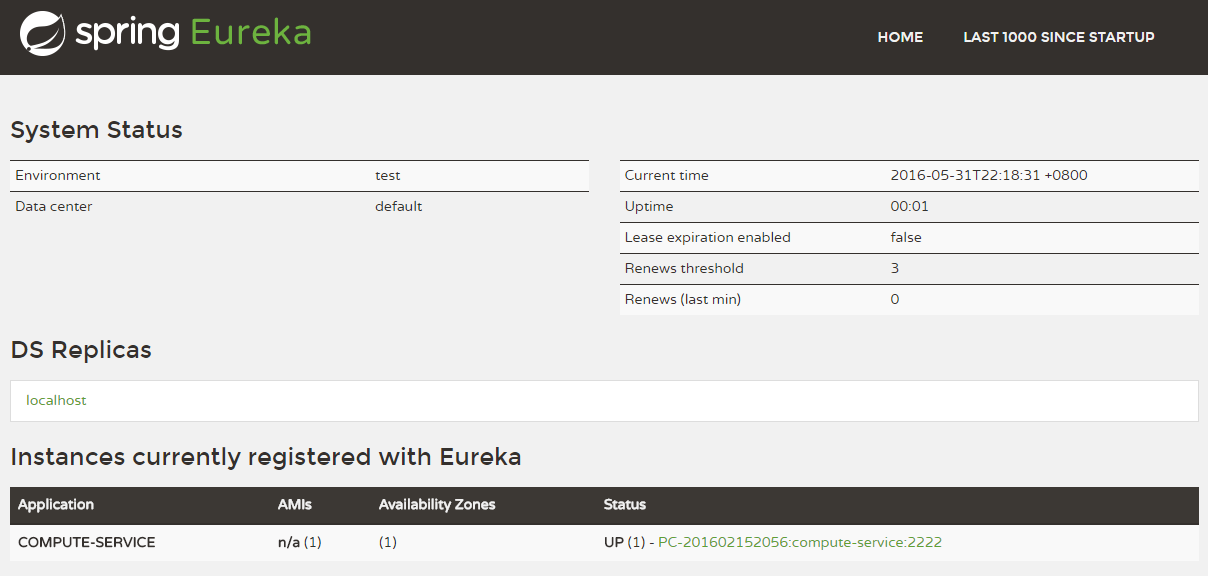
通过spring.application.name属性，我们可以指定微服务的名称后续在调用的时候只需要使用该名称就可以进行服务的访问。

eureka.client.serviceUrl.defaultZone属性对应服务注册中心的配置内容，指定服务注册中心的位置。

为了在本机上测试区分服务提供方和服务注册中心，使用server.port属性设置不同的端口。

启动该工程后，再次访问：http://localhost:1111/

可以看到，我们定义的服务被注册了。



## 3.3 服务消费者

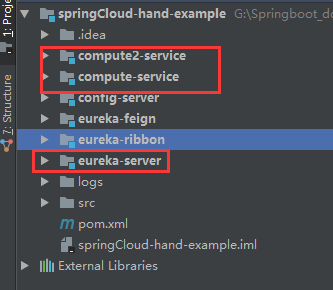
**Ribbon**

Ribbon是一个基于HTTP和TCP客户端的负载均衡器。Feign中也使用Ribbon，后续会介绍Feign的使用。Ribbon可以在通过客户端中配置的ribbonServerList服务端列表去轮询访问以达到均衡负载的作用。

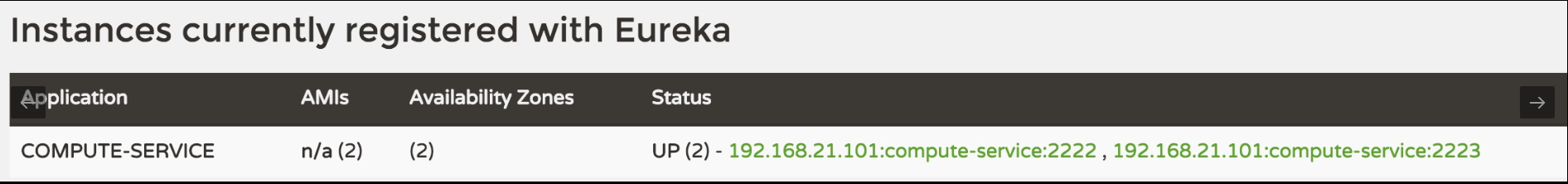
当Ribbon与Eureka联合使用时，ribbonServerList会被DiscoveryEnabledNIWSServerList重写，扩展成从Eureka注册中心中获取服务端列表。同时它也会用NIWSDiscoveryPing来取代IPing，它将职责委托给Eureka来确定服务端是否已经启动。

下面我们通过实例看看如何使用Ribbon来调用服务，并实现客户端的均衡负载。

前面3.2小节的服务需要启动，服务注册中心需要启动（eureka-server），服务提供方（compute-service）需要启动，修改compute-service中的server-port为2223，再启动一个服务提供方：compute2-service



此时访问：<http://localhost:1111/>



可以看到COMPUTE-SERVICE服务有两个单元正在运行：

192.168.21.101:compute-service:2222

192.168.21.101:compute-service:2223

#### 3.3.1 使用Ribbon实现客户端负载均衡的消费者

构建一个基本Spring Boot项目，并在pom.xml中加入如下内容：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.3.5.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-ribbon</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Brixton.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

在应用主类中，通过@EnableDiscoveryClient注解来添加发现服务能力。创建RestTemplate实例，并通过@LoadBalanced注解开启均衡负载能力。

|  |
| --- |
| @SpringBootApplication  @EnableDiscoveryClient  **public** **class** **RibbonApplication** {  @Bean  @LoadBalanced  RestTemplate **restTemplate**() {  **return** **new** RestTemplate();  }  **public** **static** **void** **main**(String[] args) {  SpringApplication.run(RibbonApplication.class, args);  }  } |

创建ConsumerController来消费COMPUTE-SERVICE的add服务。通过直接RestTemplate来调用服务，计算10 + 20的值。

|  |
| --- |
| @RestController  **public** **class** **ConsumerController** {  @Autowired  RestTemplate restTemplate;  @RequestMapping(value = "/add", method = RequestMethod.GET)  **public** String **add**() {  **return** restTemplate.getForEntity("http://COMPUTE-SERVICE/add?a=10&b=20", String.class).getBody();  }  } |

application.properties中配置eureka服务注册中心。

|  |
| --- |
| spring.application.name=ribbon-consumer  server.port=3333  eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/ |

启动该应用，并访问两次：http://localhost:3333/add

然后，打开compute-service的两个服务提供方，分别输出了类似下面的日志内容：

* 端口为2222服务提供端的日志：

|  |
| --- |
| 2017-04-27 11:16:26.787 INFO 90014 --- [io-2222-exec-10] com.didispace.web.ComputeController : /add, host:192.168.21.101, service\_id:compute-service, result:30 |

* 端口为2223服务提供端的日志：

|  |
| --- |
| 2017-04-27 11:19:41.241 INFO 90122 --- [nio-2223-exec-1] com.didispace.web.ComputeController : /add, host:192.168.21.101, service\_id:compute-service, result:30 |

可以看到，之前启动的两个compute-service服务端分别被调用了一次。到这里，我们已经通过Ribbon在客户端已经实现了对服务调用的均衡负载。

#### 3.3.2 Feign

Feign是一个声明式的Web Service客户端，它使得编写Web Serivce客户端变得更加简单。我们只需要使用Feign来创建一个接口并用注解来配置它既可完成。它具备可插拔的注解支持，包括Feign注解和JAX-RS注解。Feign也支持可插拔的编码器和解码器。Spring Cloud为Feign增加了对Spring MVC注解的支持，还整合了Ribbon和Eureka来提供均衡负载的HTTP客户端实现。

下面，通过一个例子来展现Feign如何方便的声明对上述computer-service服务的定义和调用。

创建一个Spring Boot工程，配置pom.xml，将上述的配置中的ribbon依赖替换成feign的依赖即可，具体如下：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.3.5.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-feign</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Brixton.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

在应用主类中通过@EnableFeignClients注解开启Feign功能，具体如下：

|  |
| --- |
| @SpringBootApplication  @EnableDiscoveryClient  @EnableFeignClients  **public** **class** **FeignApplication** {  **public** **static** **void** **main**(String[] args) {  SpringApplication.run(FeignApplication.class, args);  }  } |

定义compute-service服务的接口，具体如下：

|  |
| --- |
| @FeignClient("compute-service")  **public** **interface** **ComputeClient** {  @RequestMapping(method = RequestMethod.GET, value = "/add")  Integer **add**(@RequestParam(value = "a") Integer a, @**RequestParam**(value = "b") Integer b);  } |

* 使用@FeignClient("compute-service")注解来绑定该接口对应compute-service服务
* 通过Spring MVC的注解来配置compute-service服务下的具体实现。

在web层中调用上面定义的ComputeClient，具体如下：

|  |
| --- |
| @RestController  **public** **class** **ConsumerController** {  @Autowired  ComputeClient computeClient;  @RequestMapping(value = "/add", method = RequestMethod.GET)  **public** Integer **add**() {  **return** computeClient.add(10, 20);  }  } |

application.properties中不用变，指定eureka服务注册中心即可，如：

|  |
| --- |
| spring.application.name=feign-consumer  server.port=3333  eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/ |

启动该应用，访问几次：http://localhost:3333/add

再观察日志，可以得到之前使用Ribbon时一样的结果，对服务提供方实现了均衡负载。

这一节我们通过Feign以接口和注解配置的方式，轻松实现了对compute-service服务的绑定，这样我们就可以在本地应用中像本地服务一下的调用它，并且做到了客户端均衡负载。

## 3.4 断路器

在微服务架构中，我们将系统拆分成了一个个的服务单元，各单元间通过服务注册与订阅的方式互相依赖。由于每个单元都在不同的进程中运行，依赖通过远程调用的方式执行，这样就有可能因为网络原因或是依赖服务自身问题出现调用故障或延迟，而这些问题会直接导致调用方的对外服务也出现延迟，若此时调用方的请求不断增加，最后就会出现因等待出现故障的依赖方响应而形成任务积压，最终导致自身服务的瘫痪。

举个例子，在一个电商网站中，我们可能会将系统拆分成，用户、订单、库存、积分、评论等一系列的服务单元。用户创建一个订单的时候，在调用订单服务创建订单的时候，会向库存服务来请求出货（判断是否有足够库存来出货）。此时若库存服务因网络原因无法被访问到，导致创建订单服务的线程进入等待库存申请服务的响应，在漫长的等待之后用户会因为请求库存失败而得到创建订单失败的结果。如果在高并发情况之下，因这些等待线程在等待库存服务的响应而未能释放，使得后续到来的创建订单请求被阻塞，最终导致订单服务也不可用。

在微服务架构中，存在着那么多的服务单元，若一个单元出现故障，就会因依赖关系形成故障蔓延，最终导致整个系统的瘫痪，这样的架构相较传统架构就更加的不稳定。为了解决这样的问题，因此产生了断路器模式。

#### 3.4.1 什么是断路器

断路器模式源于Martin Fowler的Circuit Breaker一文。“断路器”本身是一种开关装置，用于在电路上保护线路过载，当线路中有电器发生短路时，“断路器”能够及时的切断故障电路，防止发生过载、发热、甚至起火等严重后果。

在分布式架构中，断路器模式的作用也是类似的，当某个服务单元发生故障（类似用电器发生短路）之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个错误响应，而不是长时间的等待。这样就不会使得线程因调用故障服务被长时间占用不释放，避免了故障在分布式系统中的蔓延。

#### 3.4.2 Netflix Hystrix

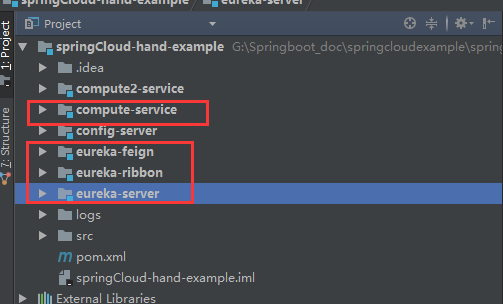
在Spring Cloud中使用了Hystrix 来实现断路器的功能。Hystrix是Netflix开源的微服务框架套件之一，该框架目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。Hystrix具备拥有回退机制和断路器功能的线程和信号隔离，请求缓存和请求打包，以及监控和配置等功能。

下面我们来看看如何使用Hystrix。

#### 3.4.3 准备工作

在开始加入断路器之前，我们先拿之前构建两个微服务为基础进行下面的操作，主要使用下面几个工程：

* eureka-server工程：服务注册中心，端口1111
* compute-service工程：服务单元，端口2222
* eureka-ribbon：通过ribbon实现的服务单元，依赖compute-service的服务，端口3333
* eureka-feign：通过feign实现的服务单元，依赖compute-service的服务，端口3333



#### 3.4.4 在Ribbon中引入Hystrix

* 依次启动eureka-server、compute-service、eureka-ribbon工程
* 访问http://localhost:1111/可以看到注册中心的状态
* 访问http://localhost:3333/add，调用eureka-ribbon的服务，该服务会去调用compute-service的服务，计算出10+20的值，页面显示30
* 关闭compute-service服务，访问http://localhost:3333/add，我们获得了下面的报错信息

|  |
| --- |
| Whitelabel Error Page  This application has no explicit mapping for /error, so you are seeing this as a fallback.  Thu Apr 25 21:16:59 CST 2017  There was an unexpected error (type=Internal Server Error, status=500).  I/O error on GET request for "http://COMPUTE-SERVICE/add?a=10&b=20": Connection refused: connect; nested exception is java.net.ConnectException: Connection refused: connect |

pom.xml中引入依赖hystrix依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix</artifactId>  </dependency> |

在eureka-ribbon的主类RibbonApplication中使用@EnableCircuitBreaker注解开启断路器功能：

|  |
| --- |
| @SpringBootApplication  @EnableDiscoveryClient  @EnableCircuitBreaker  **public** **class** **RibbonApplication** {  @Bean  @LoadBalanced  RestTemplate **restTemplate**() {  **return** **new** RestTemplate();  }  **public** **static** **void** **main**(String[] args) {  SpringApplication.run(RibbonApplication.class, args);  }  } |

改造原来的服务消费方式，新增ComputeService类，在使用ribbon消费服务的函数上增加@HystrixCommand注解来指定回调方法。

|  |
| --- |
| @Service  **public** **class** **ComputeService** {  @Autowired  RestTemplate restTemplate;  @HystrixCommand(fallbackMethod = "addServiceFallback")  **public** String **addService**() {  **return** restTemplate.getForEntity("http://COMPUTE-SERVICE/add?a=10&b=20", String.class).getBody();  }  **public** String **addServiceFallback**() {  **return** "error";  }  } |

提供rest接口的Controller改为调用ComputeService的addService

|  |
| --- |
| @RestController  **public** **class** **ConsumerController** {  @Autowired  **private** ComputeService computeService;  @RequestMapping(value = "/add", method = RequestMethod.GET)  **public** String **add**() {  **return** computeService.addService();  }  } |

验证断路器的回调

* 依次启动eureka-server、compute-service、eureka-ribbon工程
* 访问http://localhost:1111/可以看到注册中心的状态
* 访问http://localhost:3333/add，页面显示：30
* 关闭compute-service服务后再访问http://localhost:3333/add，页面显示：error

#### 3.4.5 Feign使用Hystrix

注意这里说的是“使用”，没有错，我们不需要在Feigh工程中引入Hystix，Feign中已经依赖了Hystrix，我们可以在未做任何改造前，尝试下面你的操作：

* 依次启动eureka-server、compute-service、eureka-feign工程
* 访问http://localhost:1111/可以看到注册中心的状态
* 访问http://localhost:3333/add，调用eureka-feign的服务，该服务会去调用compute-service的服务，计算出10+20的值，页面显示30
* 关闭compute-service服务，访问<http://localhost:3333/add，我们获得了下面的报错信息>

|  |
| --- |
| Whitelabel Error Page  This application has no explicit mapping for /error, so you are seeing this as a fallback.  Sat Jun 25 22:10:05 CST 2016  There was an unexpected error (type=Internal Server Error, status=500).  add timed-out and no fallback available. |

如果您够仔细，会发现与在ribbon中的报错是不同的，看到add timed-out and no fallback available这句，或许您已经猜到什么，看看我们的控制台，可以看到报错信息来自hystrix-core-1.5.2.jar，所以在这个工程中，我们要学习的就是如何使用Feign中集成的Hystrix。

使用@FeignClient注解中的fallback属性指定回调类

|  |
| --- |
| @FeignClient(value = "compute-service", fallback = ComputeClientHystrix.class)  **public** **interface** **ComputeClient** {  @RequestMapping(method = RequestMethod.GET, value = "/add")  Integer **add**(@RequestParam(value = "a") Integer a, @**RequestParam**(value = "b") Integer b);  } |

创建回调类ComputeClientHystrix，实现@FeignClient的接口，此时实现的方法就是对应@FeignClient接口中映射的fallback函数。

|  |
| --- |
| @Component  **public** **class** **ComputeClientHystrix** **implements** **ComputeClient** {  @Override  **public** Integer **add**(@RequestParam(value = "a") Integer a, @**RequestParam**(value = "b") Integer b) {  **return** -9999;  }  } |

再用之前的方法验证一下，是否在compute-service服务不可用的情况下，页面返回了-9999。

## 3.4 分布式配置中心

Spring Cloud Config为服务端和客户端提供了分布式系统的外部化配置支持。配置服务器为各应用的所有环境提供了一个中心化的外部配置。它实现了对服务端和客户端对Spring Environment和PropertySource抽象的映射，所以它除了适用于Spring构建的应用程序，也可以在任何其他语言运行的应用程序中使用。作为一个应用可以通过部署管道来进行测试或者投入生产，我们可以分别为这些环境创建配置，并且在需要迁移环境的时候获取对应环境的配置来运行。

配置服务器默认采用git来存储配置信息，这样就有助于对环境配置进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容。当然他也提供本地化文件系统的存储方式，下面从这两方面介绍如何使用分布式配置来存储微服务应用多环境的配置内容。

#### 3.4.1 构建Config Server

通过Spring Cloud构建一个Config Server，非常简单，只需要三步：

pom.xml中引入spring-cloud-config-server依赖，完整依赖配置如下：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.3.5.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Brixton.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement> |

创建Spring Boot的程序主类，并添加@EnableConfigServer注解，开启Config Server

|  |
| --- |
| @EnableConfigServer  @SpringBootApplication  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  **new** SpringApplicationBuilder(Application.class).web(**true**).run(args);  }  } |

application.properties中配置服务信息以及git信息，例如：

|  |
| --- |
| spring.application.name=config-server  server.port=7001  # git管理配置  spring.cloud.config.server.git.uri=url  spring.cloud.config.server.git.searchPaths=path  spring.cloud.config.server.git.username=username  spring.cloud.config.server.git.password=password |

* spring.cloud.config.server.git.uri：配置git仓库位置
* spring.cloud.config.server.git.searchPaths：配置仓库路径下的相对搜索位置，可以配置多个
* spring.cloud.config.server.git.username：访问git仓库的用户名
* spring.cloud.config.server.git.password：访问git仓库的用户密码

到这里，使用一个通过Spring Cloud Config实现，并使用git管理内容的配置中心已经完成了，启动该应用，成功后开始下面的内容。

Spring Cloud Config也提供本地存储配置的方式。我们只需要设置属性spring.profiles.active=native，Config Server会默认从应用的src/main/resource目录下检索配置文件。也可以通过spring.cloud.config.server.native.searchLocations=file:F:/properties/属性来指定配置文件的位置。虽然Spring Cloud Config提供了这样的功能，但是为了支持更好的管理内容和版本控制的功能，还是推荐使用git的方式。

为了验证上面完成的配置服务器，在git仓库下创建了一个config-repo目录作为配置仓库，并根据不同环境新建了下面四个配置文件：

config.properties

config -dev.properties

config -realease.properties

config -prod.properties

其中设置了一个from属性，为每个配置文件分别设置了不同的值，如：

from=git-default-1.0

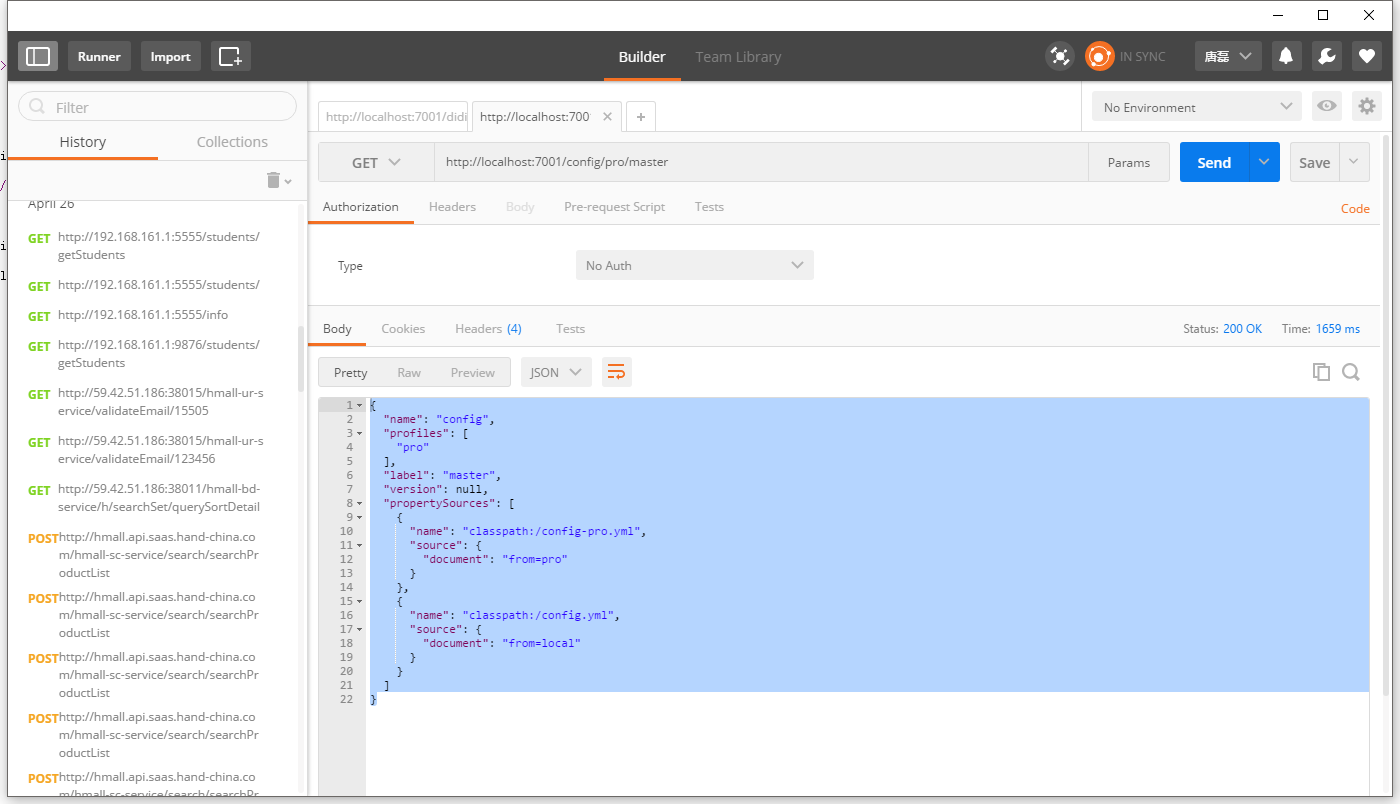
from=git-dev-1.0

from=git-realease-1.0

from=git-prod-1.0

为了测试版本控制，在master中，我们都加入1.0的后缀，同时创建一个config-label-test分支，并将各配置文件中的值用2.0作为后缀。

完成了这些准备工作之后，我们就可以通过POSTMAN工具直接来访问到我们的配置内容了。



URL与配置文件的映射关系如下：

* /{application}/{profile}[/{label}]
* /{application}-{profile}.yml
* /{label}/{application}-{profile}.yml
* /{application}-{profile}.properties
* /{label}/{application}-{profile}.properties

上面的url会映射{application}-{profile}.properties对应的配置文件，{label}对应git上不同的分支，默认为master。

我们可以尝试构造不同的url来访问不同的配置内容，比如：要访问config-label-test分支，didispace应用的prod环境，可以通过这个url：<http://localhost:7001/config/pro/master>

|  |
| --- |
| {  "name": "config",  "profiles": [  "pro"  ],  "label": "master",  "version": null,  "propertySources": [  {  "name": "classpath:/config-pro.yml",  "source": {  "document": "from=pro"  }  },  {  "name": "classpath:/config.yml",  "source": {  "document": "from=local"  }  }  ]  } |

#### 3.4.2 微服务端映射配置

在完成并验证了配置服务中心之后，下面看看我们如何在微服务应用中获取配置信息。

创建一个Spring Boot应用，在pom.xml中引入spring-cloud-starter-config依赖，完整依赖关系如下：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.3.5.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  </dependencies> |

创建最基本的Spring Boot启动主类

|  |
| --- |
| @SpringBootApplication  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  **new** SpringApplicationBuilder(Application.class).web(**true**).run(args);  }  } |

创建bootstrap.properties配置，来指定config server，例如：

|  |
| --- |
| # 这里的application name必须与配置文件一致，规则同上面提到的映射表一致  spring.application.name=config  spring.cloud.config.profile=dev  spring.cloud.config.label=master  spring.cloud.config.uri=http://localhost:7001/  server.port=7002 |

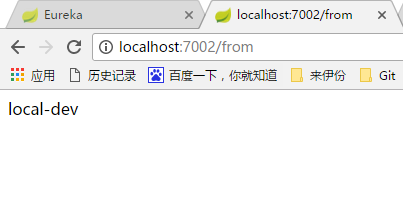
这里需要格外注意：上面这些属性必须配置在bootstrap.properties中，config部分内容才能被正确加载。因为config的相关配置会先于application.properties，而bootstrap.properties的加载也是先于application.properties。

创建一个Rest Api来返回配置中心的from属性，具体如下：

|  |
| --- |
| @RefreshScope  @RestController  **class** **TestController** {  @Value("${from}")  **private** String from;  @RequestMapping("/from")  **public** String **from**() {  **return** **this**.from;  }  } |

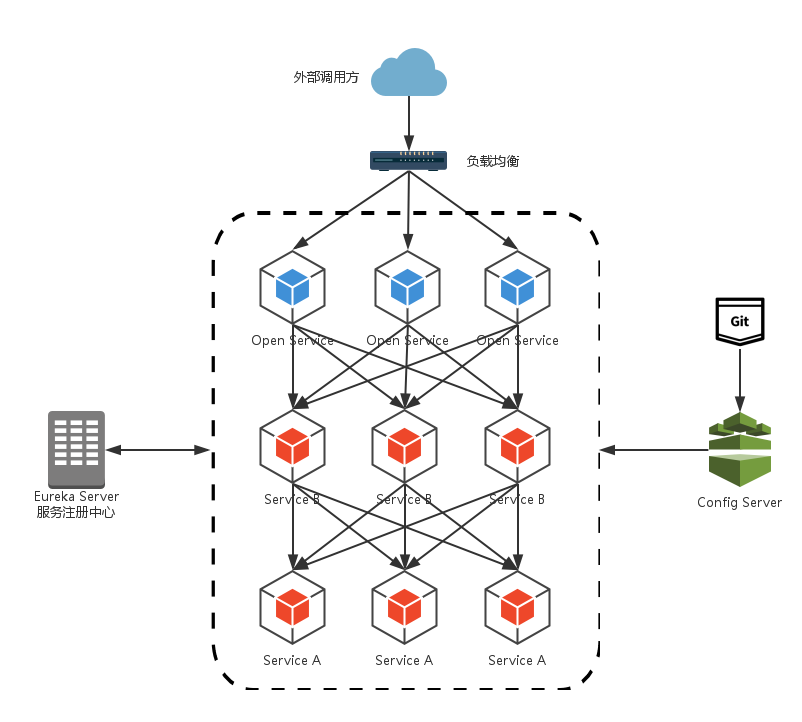
通过@Value("${from}")绑定配置服务中配置的from属性。

启动该应用，并访问：http://localhost:7002/from ，我们就可以根据配置内容输出对应环境的from内容了。



## 3.5 服务网关

通过之前几篇Spring Cloud中几个核心组件的介绍，我们已经可以构建一个简略的（不够完善）微服务架构了。比如下图所示：



我们使用Spring Cloud Netflix中的Eureka实现了服务注册中心以及服务注册与发现；而服务间通过Ribbon或Feign实现服务的消费以及均衡负载；通过Spring Cloud Config实现了应用多环境的外部化配置以及版本管理。为了使得服务集群更为健壮，使用Hystrix的融断机制来避免在微服务架构中个别服务出现异常时引起的故障蔓延。

在该架构中，我们的服务集群包含：内部服务Service A和Service B，他们都会注册与订阅服务至Eureka Server，而Open Service是一个对外的服务，通过均衡负载公开至服务调用方。本文我们把焦点聚集在对外服务这块，这样的实现是否合理，或者是否有更好的实现方式呢？

先来说说这样架构需要做的一些事儿以及存在的不足：

首先，破坏了服务无状态特点。为了保证对外服务的安全性，我们需要实现对服务访问的权限控制，而开放服务的权限控制机制将会贯穿并污染整个开放服务的业务逻辑，这会带来的最直接问题是，破坏了服务集群中REST API无状态的特点。从具体开发和测试的角度来说，在工作中除了要考虑实际的业务逻辑之外，还需要额外可续对接口访问的控制处理。

其次，无法直接复用既有接口。当我们需要对一个即有的集群内访问接口，实现外部服务访问时，我们不得不通过在原有接口上增加校验逻辑，或增加一个代理调用来实现权限控制，无法直接复用原有的接口。

面对类似上面的问题，我们要如何解决呢？下面进入本文的正题：服务网关！

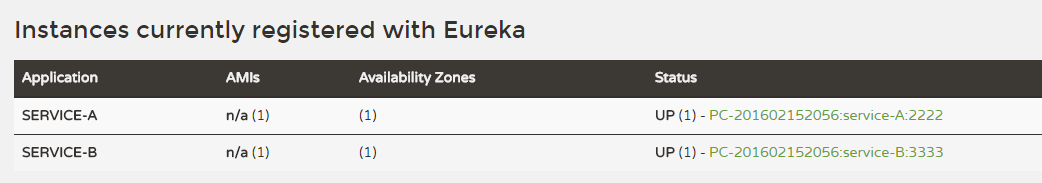
为了解决上面这些问题，我们需要将权限控制这样的东西从我们的服务单元中抽离出去，而最适合这些逻辑的地方就是处于对外访问最前端的地方，我们需要一个更强大一些的均衡负载器，它就是本文将来介绍的：服务网关。

服务网关是微服务架构中一个不可或缺的部分。通过服务网关统一向外系统提供REST API的过程中，除了具备服务路由、均衡负载功能之外，它还具备了权限控制等功能。Spring Cloud Netflix中的Zuul就担任了这样的一个角色，为微服务架构提供了前门保护的作用，同时将权限控制这些较重的非业务逻辑内容迁移到服务路由层面，使得服务集群主体能够具备更高的可复用性和可测试性。

下面我们通过实例例子来使用一下Zuul来作为服务的路有功能。

#### 3.5.1 准备工作

在使用Zuul之前，我们先构建一个服务注册中心、以及两个简单的服务，比如：我构建了一个service-A，一个service-B。然后启动eureka-server和这两个服务。通过访问eureka-server，我们可以看到service-A和service-B已经注册到了服务中心。



#### 3.5.2 开始使用Zuul

引入依赖spring-cloud-starter-zuul、spring-cloud-starter-eureka，如果不是通过指定serviceId的方式，eureka依赖不需要，但是为了对服务集群细节的透明性，还是用serviceId来避免直接引用url的方式吧。

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-zuul</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency> |

应用主类使用@EnableZuulProxy注解开启Zuul

|  |
| --- |
| @EnableZuulProxy  @SpringCloudApplication  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  **new** SpringApplicationBuilder(Application.class).web(**true**).run(args);  }  } |

这里用了@SpringCloudApplication注解，之前没有提过，通过源码我们看到，它整合了@SpringBootApplication、@EnableDiscoveryClient、@EnableCircuitBreaker，主要目的还是简化配置。这几个注解的具体作用这里就不做详细介绍了，之前的文章已经都介绍过。

application.properties中配置Zuul应用的基础信息，如：应用名、服务端口等。

|  |
| --- |
| spring.application.name=api-gateway  server.port=5555 |

#### 3.5.3 Zuul配置

完成上面的工作后，Zuul已经可以运行了，但是如何让它为我们的微服务集群服务，还需要我们另行配置，下面详细的介绍一些常用配置内容。

**服务路由**

通过服务路由的功能，我们在对外提供服务的时候，只需要通过暴露Zuul中配置的调用地址就可以让调用方统一的来访问我们的服务，而不需要了解具体提供服务的主机信息了。

在Zuul中提供了两种映射方式：

通过url直接映射，我们可以如下配置：

|  |
| --- |
| # routes to url  zuul.routes.api-a-url.path=/api-a-url/\*\*  zuul.routes.api-a-url.url=http://localhost:2222/ |
|  |

该配置，定义了，所有到Zuul的中规则为：/api-a-url/\*\*的访问都映射到http://localhost:2222/上，也就是说当我们访问http://localhost:5555/api-a-url/add?a=1&b=2的时候，Zuul会将该请求路由到：http://localhost:2222/add?a=1&b=2上。

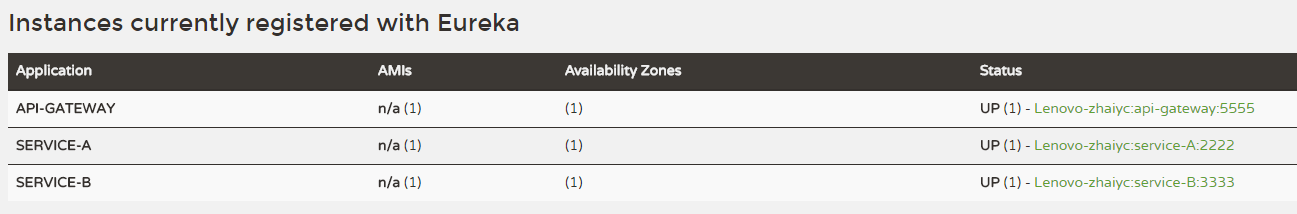
其中，配置属性zuul.routes.api-a-url.path中的api-a-url部分为路由的名字，可以任意定义，但是一组映射关系的path和url要相同，下面讲serviceId时候也是如此。

通过url映射的方式对于Zuul来说，并不是特别友好，Zuul需要知道我们所有为服务的地址，才能完成所有的映射配置。而实际上，我们在实现微服务架构时，服务名与服务实例地址的关系在eureka server中已经存在了，所以只需要将Zuul注册到eureka server上去发现其他服务，我们就可以实现对serviceId的映射。例如，我们可以如下配置：

|  |
| --- |
| zuul.routes.api-a.path=/api-a/\*\*  zuul.routes.api-a.serviceId=service-A  zuul.routes.api-b.path=/api-b/\*\*  zuul.routes.api-b.serviceId=service-B  eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/ |

针对我们在准备工作中实现的两个微服务service-A和service-B，定义了两个路由api-a和api-b来分别映射。另外为了让Zuul能发现service-A和service-B，也加入了eureka的配置。

接下来，我们将eureka-server、service-A、service-B以及这里用Zuul实现的服务网关启动起来，在eureka-server的控制页面中，我们可以看到分别注册了service-A、service-B以及api-gateway



尝试通过服务网关来访问service-A和service-B，根据配置的映射关系，分别访问下面的url

http://localhost:5555/api-a/add?a=1&b=2：通过serviceId映射访问service-A中的add服务

http://localhost:5555/api-b/add?a=1&b=2：通过serviceId映射访问service-B中的add服务

http://localhost:5555/api-a-url/add?a=1&b=2：通过url映射访问service-A中的add服务

推荐使用serviceId的映射方式，除了对Zuul维护上更加友好之外，serviceId映射方式还支持了断路器，对于服务故障的情况下，可以有效的防止故障蔓延到服务网关上而影响整个系统的对外服务

#### 3.5.4 服务过滤

在完成了服务路由之后，我们对外开放服务还需要一些安全措施来保护客户端只能访问它应该访问到的资源。所以我们需要利用Zuul的过滤器来实现我们对外服务的安全控制。

在服务网关中定义过滤器只需要继承ZuulFilter抽象类实现其定义的四个抽象函数就可对请求进行拦截与过滤。

比如下面的例子，定义了一个Zuul过滤器，实现了在请求被路由之前检查请求中是否有accessToken参数，若有就进行路由，若没有就拒绝访问，返回401 Unauthorized错误。

|  |
| --- |
| **public** **class** **AccessFilter** **extends** **ZuulFilter** {  **private** **static** Logger log = LoggerFactory.getLogger(AccessFilter.class);  @Override  **public** String **filterType**() {  **return** "pre";  }  @Override  **public** **int** **filterOrder**() {  **return** 0;  }  @Override  **public** **boolean** **shouldFilter**() {  **return** **true**;  }  @Override  **public** Object **run**() {  RequestContext ctx = RequestContext.getCurrentContext();  HttpServletRequest request = ctx.getRequest();  log.info(String.format("%s request to %s", request.getMethod(), request.getRequestURL().toString()));  Object accessToken = request.getParameter("accessToken");  **if**(accessToken == **null**) {  log.warn("access token is empty");  ctx.setSendZuulResponse(**false**);  ctx.setResponseStatusCode(401);  **return** **null**;  }  log.info("access token ok");  **return** **null**;  }  } |

自定义过滤器的实现，需要继承ZuulFilter，需要重写实现下面四个方法：

filterType：返回一个字符串代表过滤器的类型，在zuul中定义了四种不同生命周期的过滤器类型，具体如下：

pre：可以在请求被路由之前调用

routing：在路由请求时候被调用

post：在routing和error过滤器之后被调用

error：处理请求时发生错误时被调用

filterOrder：通过int值来定义过滤器的执行顺序

shouldFilter：返回一个boolean类型来判断该过滤器是否要执行，所以通过此函数可实现过滤器的开关。在上例中，我们直接返回true，所以该过滤器总是生效。

run：过滤器的具体逻辑。需要注意，这里我们通过ctx.setSendZuulResponse(false)令zuul过滤该请求，不对其进行路由，然后通过ctx.setResponseStatusCode(401)设置了其返回的错误码，当然我们也可以进一步优化我们的返回，比如，通过ctx.setResponseBody(body)对返回body内容进行编辑等。

在实现了自定义过滤器之后，还需要实例化该过滤器才能生效，我们只需要在应用主类中增加如下内容：

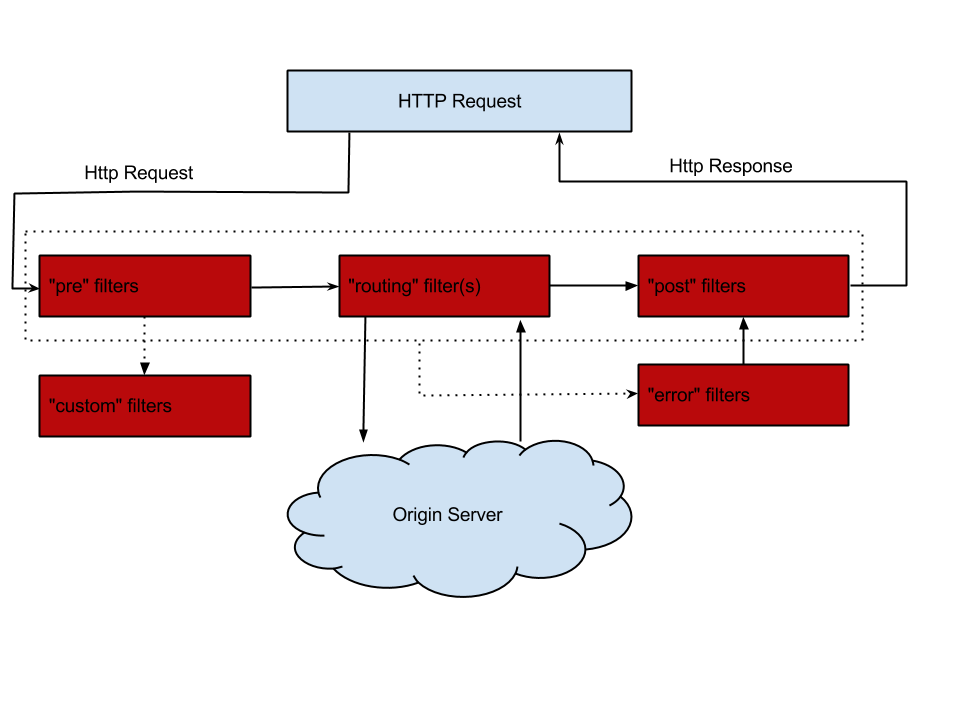
|  |
| --- |
| @EnableZuulProxy  @SpringCloudApplication  public class Application {  public static void main(String[] args) {  new SpringApplicationBuilder(Application.class).web(true).run(args);  }  @Bean  public AccessFilter accessFilter() {  return new AccessFilter();  }  } |

启动该服务网关后，访问：

http://localhost:5555/api-a/add?a=1&b=2：返回401错误

http://localhost:5555/api-a/add?a=1&b=2&accessToken=token：正确路由到server-A，并返回计算内容

对于其他一些过滤类型，这里就不一一展开了，根据之前对filterType生命周期介绍，可以参考下图去理解，并根据自己的需要在不同的生命周期中去实现不同类型的过滤器。



最后，总结一下为什么服务网关是微服务架构的重要部分，是我们必须要去做的原因：

不仅仅实现了路由功能来屏蔽诸多服务细节，更实现了服务级别、均衡负载的路由。

实现了接口权限校验与微服务业务逻辑的解耦。通过服务网关中的过滤器，在各生命周期中去校验请求的内容，将原本在对外服务层做的校验前移，保证了微服务的无状态性，同时降低了微服务的测试难度，让服务本身更集中关注业务逻辑的处理。

实现了断路器，不会因为具体微服务的故障而导致服务网关的阻塞，依然可以对外服务。