# 今日大纲

1. 统一开发环境
2. 了解微服务架构
3. 了解Spring Cloud
4. Spring Cloud的快速入门
5. Eureka服务注册中心的使用
6. 使用Ribbon实现负载均衡
7. 使用Hystrix实现容错

# 统一开发环境

JDK：1.8

Eclipse：4.4.1 luna

Maven：3.2.3

安装文件在课前资料中。

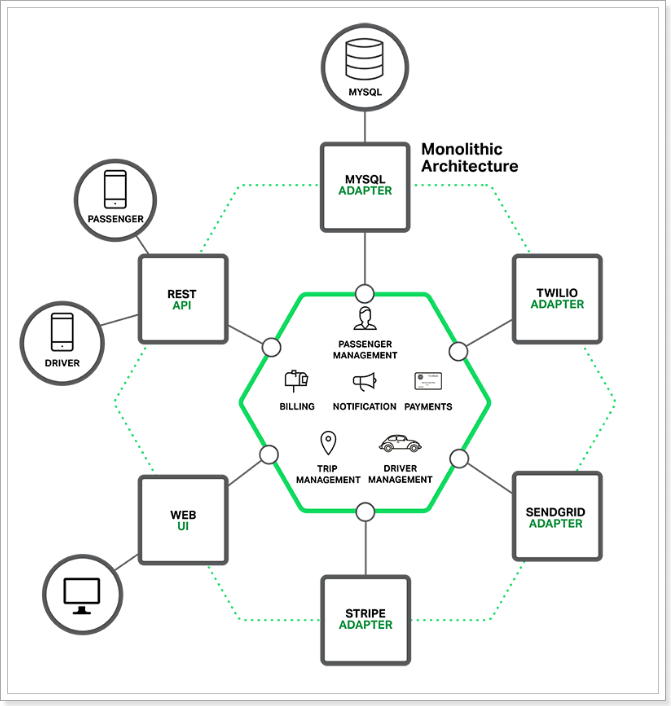
# 微服务架构

目前微服务是非常火的架构或者说概念，也是在构建大型互联网项目时采用的架构方式。

## 单体架构

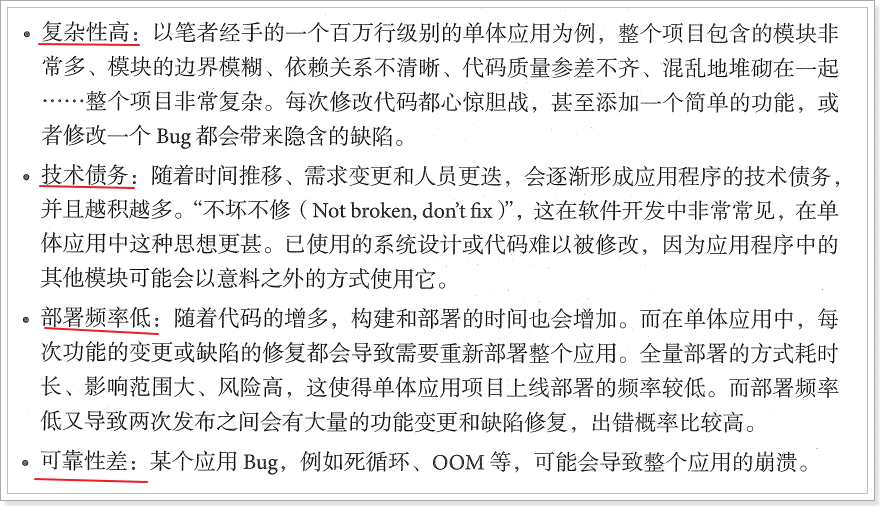
单体架构，是指将开发好的项目打成war包，然后发布到tomcat等容器中的应用。

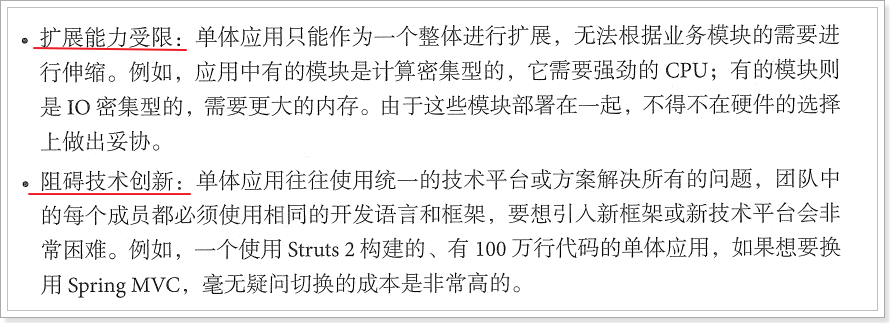
假设你正准备开发一款与Uber和Hailo竞争的出租车调度软件，经过初步会议和需求分析，你可能会手动或者使用基于Spring Boot、Play或者Maven的生成器开始这个新项目，它的六边形架构是模块化的 ，架构图如下：



* 应用核心是业务逻辑，由定义服务、域对象和事件的模块完成。围绕着核心的是与外界打交道的适配器。适配器包括数据库访问组件、生产和处理消息的消息组件，以及提供API或者UI访问支持的web模块等。
* 尽管也是模块化逻辑，但是最终它还是会打包并部署为单体式应用。具体的格式依赖于应用语言和框架。例如，许多Java应用会被打包为WAR格式，部署在Tomcat或者Jetty上，而另外一些Java应用会被打包成自包含的JAR格式，同样，Rails和Node.js会被打包成层级目录。
* 这种应用开发风格很常见，因为IDE和其它工具都擅长开发一个简单应用，这类应用也很易于调试，只需要简单运行此应用，用Selenium链接UI就可以完成端到端测试。单体式应用也易于部署，只需要把打包应用拷贝到服务器端，通过在负载均衡器后端运行多个拷贝就可以轻松实现应用扩展。在早期这类应用运行的很好。

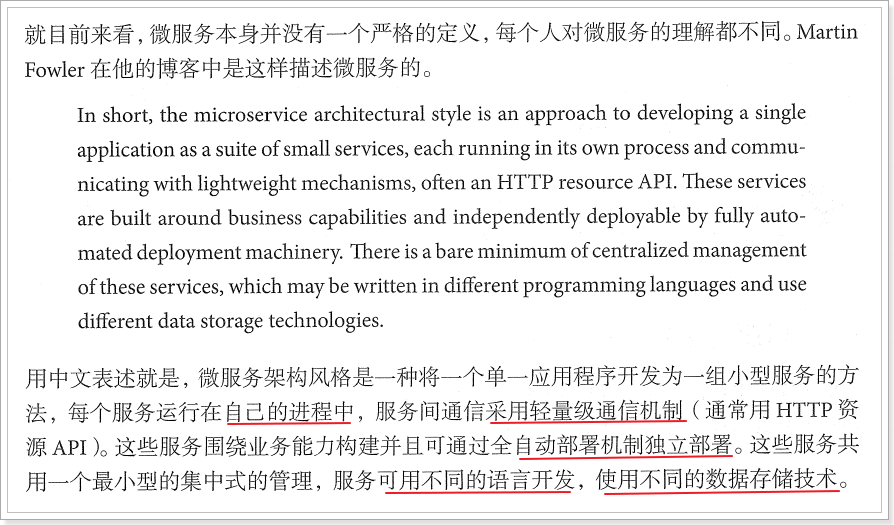
## 单体架构存在的问题





如何解决以上问题呢？ -- 使用微服务架构。

## 什么是微服务？

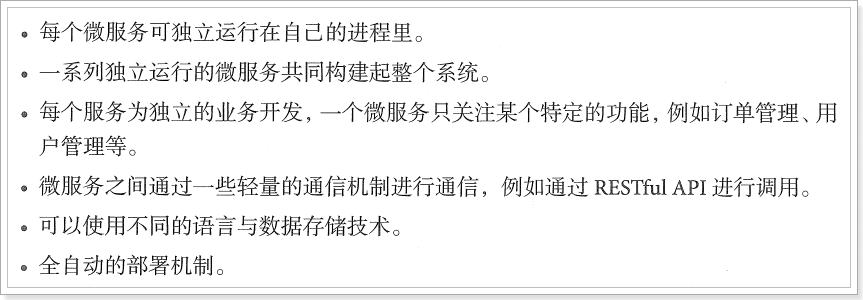


作者：Martin Fowler

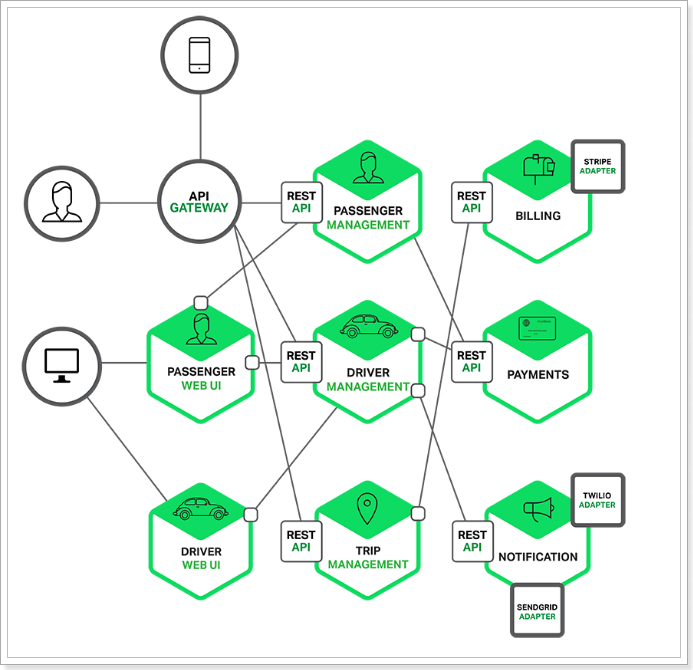




## 微服务架构的特征



## 微服务架构示例



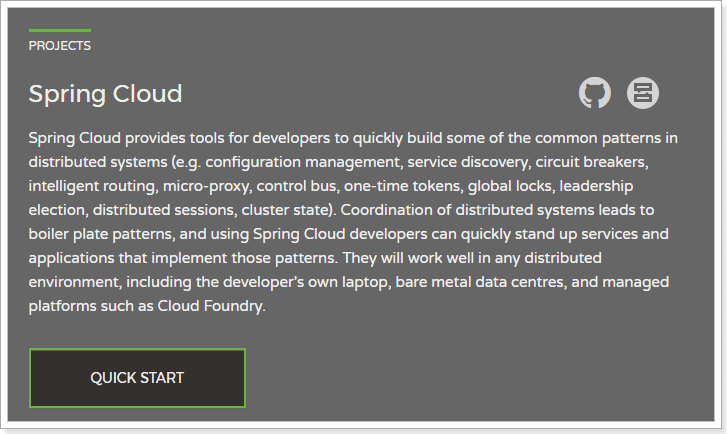
每一个应用功能区都使用微服务完成。

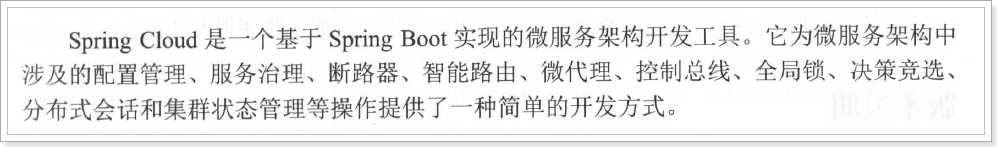
# Spring Cloud简介

## 简介

Spring Cloud项目的官方网址：

<http://projects.spring.io/spring-cloud/>

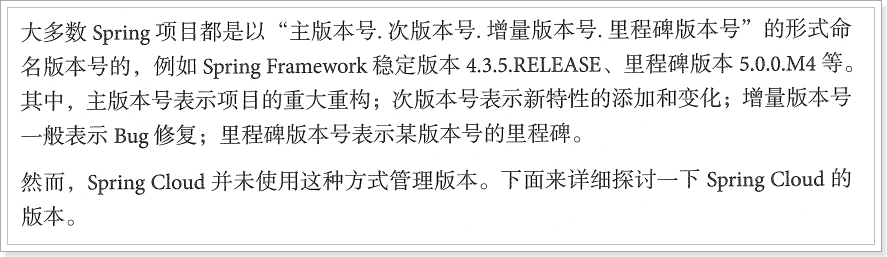




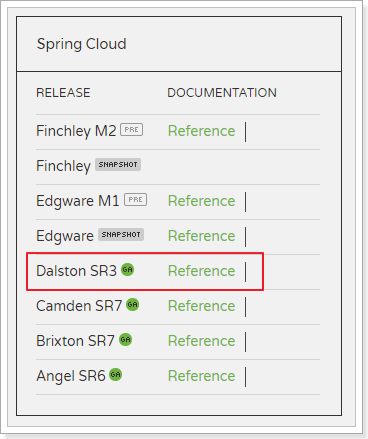
## Spring Cloud子项目

| **Component** | **Camden.SR7** | **Dalston.SR3** | **Edgware.M1** | **Finchley.M2** | **Finchley.BUILD-SNAPSHOT** | **备注** |
| --- | --- | --- | --- | --- | --- | --- |
| spring-cloud-aws | 1.1.4.RELEASE | 1.2.1.RELEASE | 1.2.1.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT | 用于简化整合Amazon Web Service的组件 |
| spring-cloud-bus | 1.2.2.RELEASE | 1.3.1.RELEASE | 1.3.1.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT | 事件、消息总线，用于传播集群中的状态变化或事件。 |
| spring-cloud-cli | 1.2.4.RELEASE | 1.3.4.RELEASE | 1.4.0.M1 | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT | 用于在Groovy平台创建Spring Cloud应用。 |
| spring-cloud-commons | 1.1.9.RELEASE | 1.2.3.RELEASE | 1.3.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT | 服务发现、负载均衡、熔断机制这种模式为Spring Cloud客户端提供了一个通用的抽象层。 |
| spring-cloud-contract | 1.0.5.RELEASE | 1.1.3.RELEASE | 1.2.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT |  |
| spring-cloud-config | 1.2.3.RELEASE | 1.3.2.RELEASE | 1.4.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT | 配置管理工具，支持使用git、svn等存储配置文件。并在支持客户端配置信息的刷新，加密解密配置内容等。 |
| spring-cloud-netflix | 1.2.7.RELEASE | 1.3.4.RELEASE | 1.4.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT | 核心组件，对多个Netflix OSS开源套件进行整合。 |
| spring-cloud-security | 1.1.4.RELEASE | 1.2.1.RELEASE | 1.2.1.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT | 安全工具包。 |
| spring-cloud-cloudfoundry | 1.0.1.RELEASE | 1.1.0.RELEASE | 1.1.0.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT | 整合Pivotal Cloudfoundry（Vmware推出的业界第一个开源PaaS云平台）支持。 |
| spring-cloud-consul | 1.1.4.RELEASE | 1.2.1.RELEASE | 1.2.1.RELEASE | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT | 服务发现与配置管理工具 |
| spring-cloud-sleuth | 1.1.3.RELEASE | 1.2.4.RELEASE | 1.3.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT | Spring Cloud应用的分布式跟踪实现。 |
| spring-cloud-stream | Brooklyn.SR3 | Chelsea.SR2 | Ditmars.M2 | Elmhurst.M1 | Elmhurst.BUILD-SNAPSHOT | 通过Redis、RabbitMQ、Kafka实现的消息微服务。 |
| spring-cloud-zookeeper | 1.0.4.RELEASE | 1.1.2.RELEASE | 1.2.0.M1 | 2.0.0.M1 | 2.0.0.BUILD-SNAPSHOT | 基于ZooKeeper的服务发现与配置管理组件。 |
| spring-boot | 1.4.5.RELEASE | 1.5.4.RELEASE | 1.5.6.RELEASE | 2.0.0.M3 | 2.0.0.M3 |  |
| spring-cloud-task | 1.0.3.RELEASE | 1.1.2.RELEASE | 1.2.0.RELEASE | 2.0.0.M1 | 2.0.0.RELEASE | 用于快速构建数据处理的应用。 |
| spring-cloud-vault |  | 1.0.2.RELEASE | 1.1.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT |  |
| spring-cloud-gateway |  |  | 1.0.0.M1 | 2.0.0.M2 | 2.0.0.BUILD-SNAPSHOT | Spring Cloud网关相关的整合实现。 |

## 版本说明

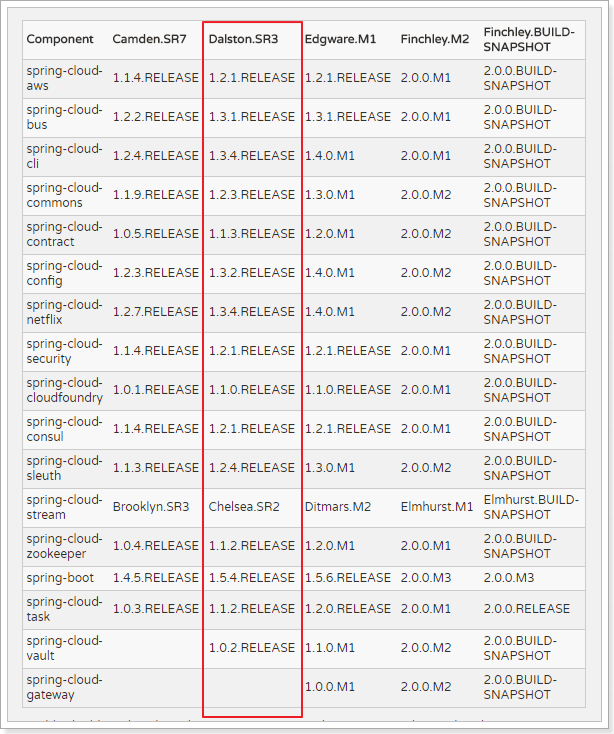


官方版本：

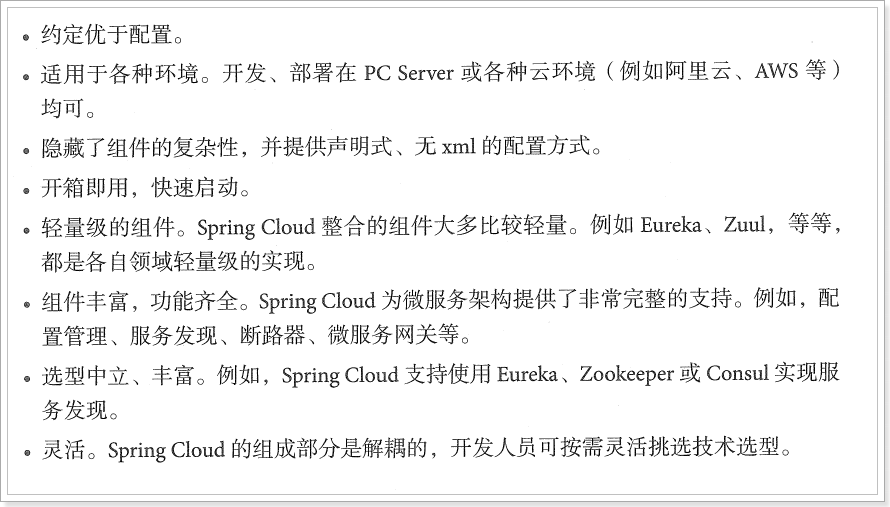




可见，目前Dalston SR3版本是最新的稳定版，所以我们学习的过程中，就是使用的这个版本。



## Spring Cloud框架特点



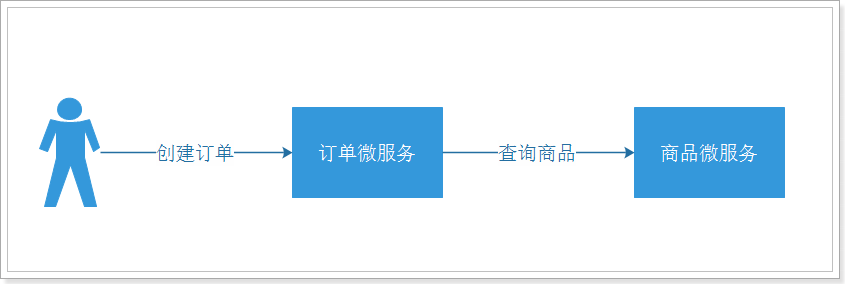
# 使用Spring Boot实现微服务

在正式学习Spring Cloud之前我们先使用Spring Boot实现一个微服务。

业务非常简单：

1. 商品微服务：通过商品id查询商品的服务；
2. 订单微服务：创建订单时通时，通过调用商品的微服务进行查询商品数据；

图示：

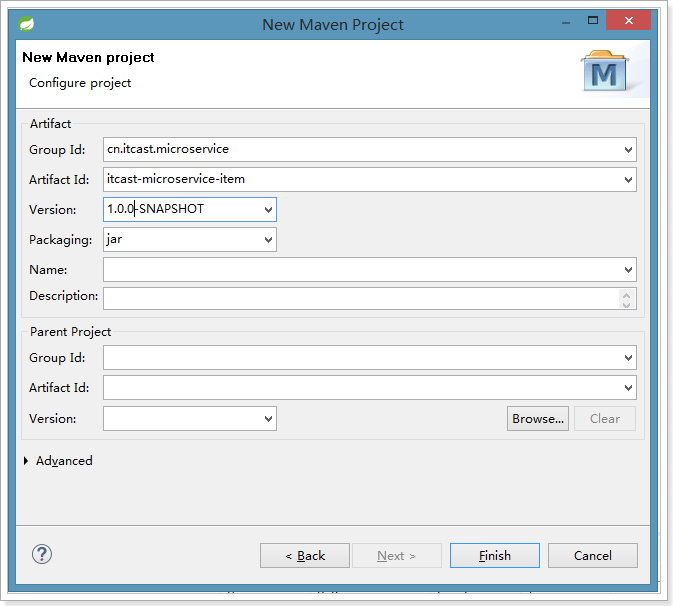


说明：

1. 对于商品微服务而言，商品微服务是服务的提供者，订单微服务是服务的消费者；
2. 对于订单微服务而言，订单微服务是服务的提供者，人是服务的消费者。

## 实现商品微服务

### 创建工程



### 导入依赖

重点是导入Spring Boot的依赖：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>cn.itcast.microservice</groupId>

<artifactId>itcast-microservice-item</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<!-- 资源文件拷贝插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

<!-- java编译插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

</project>

### 创建实体Item

**package** cn.itcast.microservice.pojo;

**public** **class** Item {

**private** Long id;

**private** String title;

**private** String pic;

**private** String desc;

**private** Long price;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getTitle() {

**return** title;

}

**public** **void** setTitle(String title) {

**this**.title = title;

}

**public** String getPic() {

**return** pic;

}

**public** **void** setPic(String pic) {

**this**.pic = pic;

}

**public** String getDesc() {

**return** desc;

}

**public** **void** setDesc(String desc) {

**this**.desc = desc;

}

**public** Long getPrice() {

**return** price;

}

**public** **void** setPrice(Long price) {

**this**.price = price;

}

@Override

**public** String toString() {

**return** "Item [id=" + id + ", title=" + title + ", pic=" + pic + ", desc=" + desc + ", price=" + price + "]";

}

}

### 编写ItemService

编写ItemService用于实现具体的商品查询逻辑，为了演示方便，我们并不真正的连接数据库，而是做模拟实现。

**package** cn.itcast.microservice.service;

**import** java.util.HashMap;

**import** java.util.Map;

**import** org.springframework.stereotype.Service;

**import** cn.itcast.microservice.pojo.Item;

@Service

**public** **class** ItemService {

**private** **static** **final** Map<Long, Item> ***MAP*** = **new** HashMap<Long, Item>();

**static** { // 准备一些静态数据

***MAP***.put(1L, **new** Item(1L, "商品标题1", "http://图片1", "商品描述1", 1000L));

***MAP***.put(2L, **new** Item(1L, "商品标题2", "http://图片2", "商品描述2", 2000L));

***MAP***.put(3L, **new** Item(1L, "商品标题3", "http://图片3", "商品描述3", 3000L));

***MAP***.put(4L, **new** Item(1L, "商品标题4", "http://图片4", "商品描述4", 4000L));

***MAP***.put(5L, **new** Item(1L, "商品标题5", "http://图片5", "商品描述5", 5000L));

***MAP***.put(6L, **new** Item(1L, "商品标题6", "http://图片6", "商品描述6", 6000L));

***MAP***.put(7L, **new** Item(1L, "商品标题7", "http://图片7", "商品描述7", 7000L));

***MAP***.put(8L, **new** Item(1L, "商品标题8", "http://图片8", "商品描述8", 8000L));

}

/\*\*

\* 模拟实现商品查询

\*

\* **@param** id

\* **@return**

\*/

**public** Item queryItemById(Long id) {

**return** ***MAP***.get(id);

}

}

### 编写ItemController

**package** cn.itcast.microservice.controller;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RestController;

**import** cn.itcast.microservice.pojo.Item;

**import** cn.itcast.microservice.service.ItemService;

@RestController

**public** **class** ItemController {

@Autowired

**private** ItemService itemService;

/\*\*

\* 对外提供接口服务，查询商品信息

\*

\* **@param** id

\* **@return**

\*/

@GetMapping(value = "item/{id}")

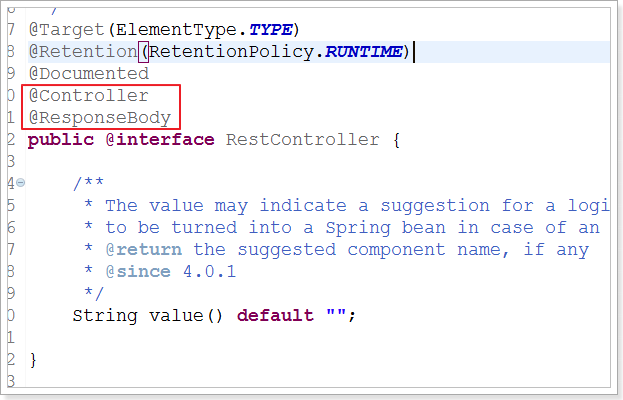
**public** Item queryItemById(@PathVariable("id") Long id) {

**return** **this**.itemService.queryItemById(id);

}

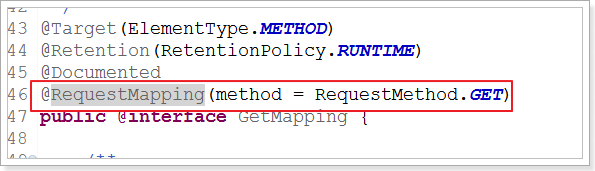
}

@RestController注解的说明：



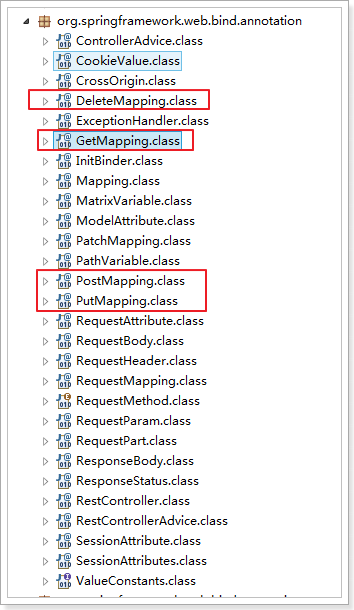
从源码可以看出，这是一个组合注解，组合了@Controller和@Response注解。相当于我们同时写了这2个注解。

@GetMapping注解的说明：



@GetMapping注解是 @RequestMapping(method = RequestMethod.GET) 简写方式。其功能都是一样的。

同理还有其它注解：



### 编写程序入口

**package** cn.itcast.microservice;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication //申明这是一个Spring Boot项目

**public** **class** ItemApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ItemApplication.**class**, args);

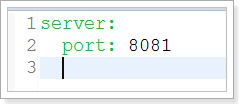
}

}

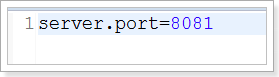
### 创建application.yml配置文件

Spring Boot以及Spring Cloud项目支持yml和properties格式的配置文件。

yml格式是YAML（Yet Another Markup Language）编写的格式，YAML和properties格式的文件是可以相互转化的。如：



等价于：



配置文件的示例：

server:

port: 8081 #服务端口

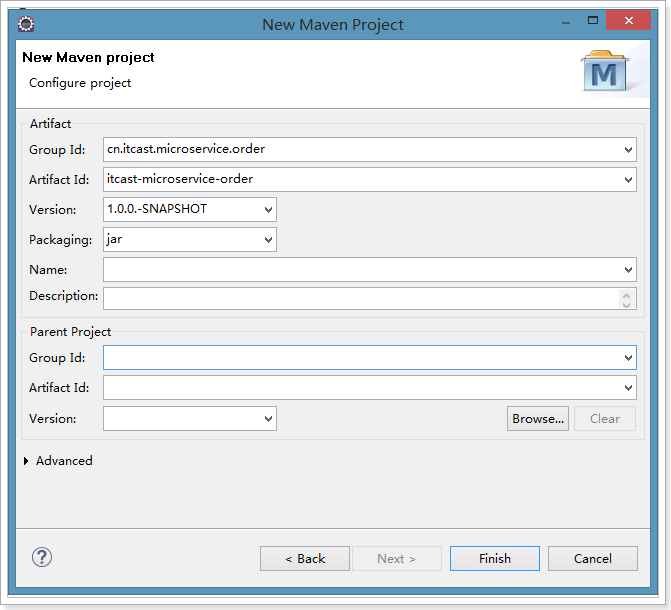
### 启动程序测试



可以看到已经通过微服务查询到数据。

## 实现订单微服务

### 创建工程



### 导入依赖

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>cn.itcast.microservice.order</groupId>

<artifactId>itcast-microservice-order</artifactId>

<version>1.0.0.-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<!-- 资源文件拷贝插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

<!-- java编译插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

</project>

### 创建订单Order实体

**package** cn.itcast.microservice.order.pojo;

**import** java.util.Date;

**import** java.util.List;

**public** **class** Order {

**private** String orderId;

**private** Long userId;

**private** Date createDate;

**private** Date updateDate;

**private** List<OrderDetail> orderDetails;

**public** Order() {

}

**public** Order(String orderId, Long userId, Date createDate, Date updateDate) {

**this**.orderId = orderId;

**this**.userId = userId;

**this**.createDate = createDate;

**this**.updateDate = updateDate;

}

**public** String getOrderId() {

**return** orderId;

}

**public** **void** setOrderId(String orderId) {

**this**.orderId = orderId;

}

**public** Long getUserId() {

**return** userId;

}

**public** **void** setUserId(Long userId) {

**this**.userId = userId;

}

**public** Date getCreateDate() {

**return** createDate;

}

**public** **void** setCreateDate(Date createDate) {

**this**.createDate = createDate;

}

**public** Date getUpdateDate() {

**return** updateDate;

}

**public** **void** setUpdateDate(Date updateDate) {

**this**.updateDate = updateDate;

}

**public** List<OrderDetail> getOrderDetails() {

**return** orderDetails;

}

**public** **void** setOrderDetails(List<OrderDetail> orderDetails) {

**this**.orderDetails = orderDetails;

}

@Override

**public** String toString() {

**return** "Order [orderId=" + orderId + ", userId=" + userId

+ ", createDate=" + createDate + ", updateDate=" + updateDate

+ "]";

}

}

### 创建订单详情OrderDetail实体

订单与订单详情是一对多的关系。

**package** cn.itcast.microservice.order.pojo;

**public** **class** OrderDetail {

**private** String orderId;

**private** Item item = **new** Item();

**public** OrderDetail() {

}

**public** OrderDetail(String orderId, Item item) {

**this**.orderId = orderId;

**this**.item = item;

}

**public** String getOrderId() {

**return** orderId;

}

**public** **void** setOrderId(String orderId) {

**this**.orderId = orderId;

}

**public** Item getItem() {

**return** item;

}

**public** **void** setItem(Item item) {

**this**.item = item;

}

@Override

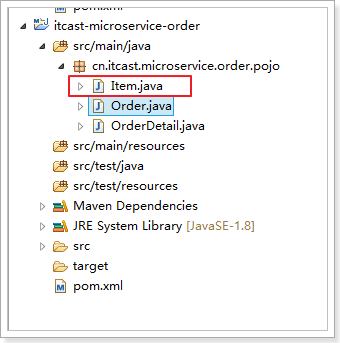
**public** String toString() {

**return** "OrderDetail [orderId=" + orderId + ", item=" + item + "]";

}

}

### 将商品微服务中的Item类拷贝到当前工程



### 编写OrderService

该Service实现的根据订单Id查询订单的服务，为了方便测试，我们将构造数据实现，不采用查询数据库的方式。

**package** cn.itcast.microservice.order.service;

**import** java.util.ArrayList;

**import** java.util.Date;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** cn.itcast.microservice.order.pojo.Item;

**import** cn.itcast.microservice.order.pojo.Order;

**import** cn.itcast.microservice.order.pojo.OrderDetail;

@Service

**public** **class** OrderService {

**private** **static** **final** Map<String, Order> ***MAP*** = **new** HashMap<String, Order>();

**static** {

// 构造测试数据

Order order = **new** Order();

order.setOrderId("59193738268961441");

order.setCreateDate(**new** Date());

order.setUpdateDate(order.getCreateDate());

order.setUserId(1L);

List<OrderDetail> orderDetails = **new** ArrayList<OrderDetail>();

Item item = **new** Item();// 此处并没有商品的数据，需要调用商品微服务获取

item.setId(1L);

orderDetails.add(**new** OrderDetail(order.getOrderId(), item));

item = **new** Item(); // 构造第二个商品数据

item.setId(2L);

orderDetails.add(**new** OrderDetail(order.getOrderId(), item));

order.setOrderDetails(orderDetails);

***MAP***.put(order.getOrderId(), order);

}

@Autowired

**private** ItemService itemService;

/\*\*

\* 根据订单id查询订单数据

\*

\* **@param** orderId

\* **@return**

\*/

**public** Order queryOrderById(String orderId) {

Order order = ***MAP***.get(orderId);

**if** (**null** == order) {

**return** **null**;

}

List<OrderDetail> orderDetails = order.getOrderDetails();

**for** (OrderDetail orderDetail : orderDetails) {

// 通过商品微服务查询商品数据

Item item = **this**.itemService.queryItemById(orderDetail.getItem()

.getId());

**if** (**null** == item) {

**continue**;

}

orderDetail.setItem(item);

}

**return** order;

}

}

### 实现ItemService

**package** cn.itcast.microservice.order.service;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** org.springframework.web.client.RestTemplate;

**import** cn.itcast.microservice.order.pojo.Item;

@Service

**public** **class** ItemService {

// Spring框架对RESTful方式的http请求做了封装，来简化操作

@Autowired

**private** RestTemplate restTemplate;

**public** Item queryItemById(Long id) {

**return** **this**.restTemplate.getForObject("http://127.0.0.1:8081/item/"

+ id, Item.**class**);

}

}

### 编写OrderController

**package** cn.itcast.microservice.order.controller;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RestController;

**import** cn.itcast.microservice.order.pojo.Order;

**import** cn.itcast.microservice.order.service.OrderService;

@RestController

**public** **class** OrderController {

@Autowired

**private** OrderService orderService;

@GetMapping(value = "order/{orderId}")

**public** Order queryOrderById(@PathVariable("orderId") String orderId) {

**return** **this**.orderService.queryOrderById(orderId);

}

}

### 编写程序入口

**package** cn.itcast.microservice;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.web.client.RestTemplate;

@SpringBootApplication

**public** **class** OrderApplication {

@Bean // 向Spring容器中定义RestTemplate对象

**public** RestTemplate restTemplate(){

**return** **new** RestTemplate();

}

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(OrderApplication.**class**, args);

}

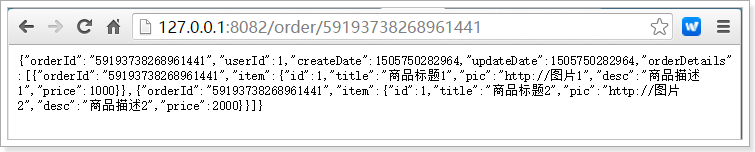
}

### 编写application.yml配置文件

server:

port: 8082 #服务端口

### 启动测试

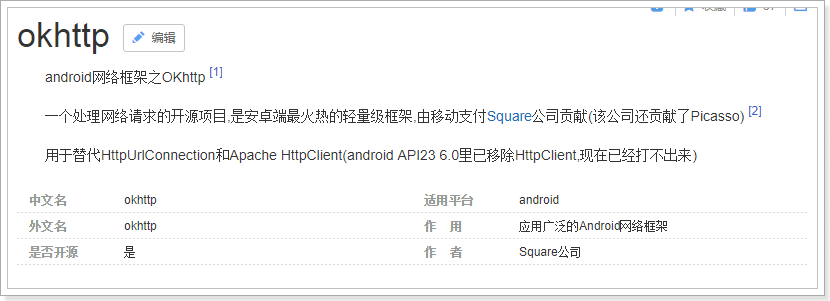


测试结果可见，查询订单时，同时也将商品数据查询到。

## 添加okHttp的支持

okhttp是一个封装URL,比HttpClient更友好易用的工具。目前似乎okhttp更流行一些。

官网：<http://square.github.io/okhttp/>



RestTemplate底层默认使用的jdk的标准实现，如果我们想让RestTemplate的底层使用okhttp，非常简单：

1. 添加okhttp依赖

<dependency>

<groupId>com.squareup.okhttp3</groupId>

<artifactId>okhttp</artifactId>

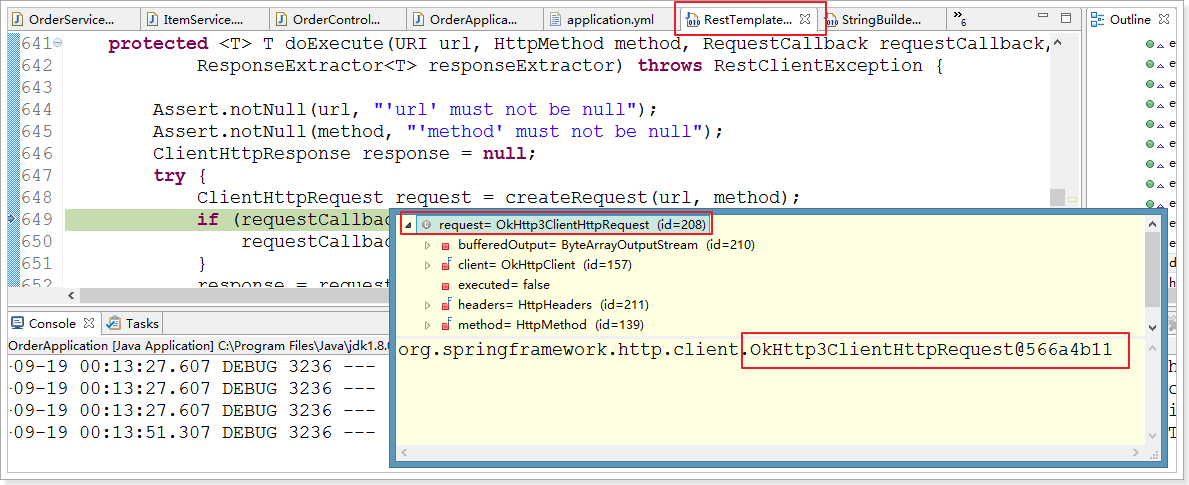
<version>3.9.0</version>

</dependency>

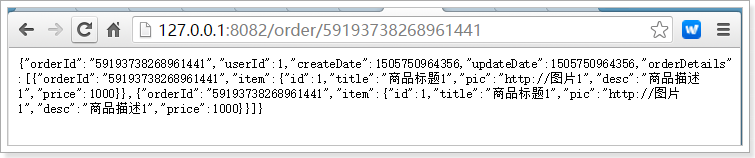
1. 设置requestFactory

return new RestTemplate(new OkHttp3ClientHttpRequestFactory());  


测试：



结果：



测试结果是一样的。

## 解决订单系统中的url硬编码问题

通过以上的测试我们发现，在订单系统中要调用商品微服务中的查询接口来获取数据，在订单微服务中将url硬编码到代码中，这样显然不好，因为，运行环境一旦发生变化这个url地址将不可用。

如何解决呢？

解决方案：将url地址写入到application.yml配置文件中。

实现：

修改application.yml文件：

server:

port: 8082 #服务端口

itcast:

item:

url: http://127.0.0.1:8081/item/

修改ItemService中的实现：



测试：



## 继续优化解决硬编码的问题

在SpringBoot中使用@ConfigurationProperties注解可以非常简单的将配置文件中的值映射成对象。

第一步，创建ItemProperties类：

**package** cn.itcast.microservice.order.properties;

**public** **class** ItemProperties {

**private** String url;

**public** String getUrl() {

**return** url;

}

**public** **void** setUrl(String url) {

**this**.url = url;

}

}

第二步，创建OrderProperties类：

**package** cn.itcast.microservice.order.properties;

**import** org.springframework.boot.context.properties.ConfigurationProperties;

**import** org.springframework.stereotype.Component;

@Component

@ConfigurationProperties(prefix="itcast") //以itcast开头的配置被匹配到

**public** **class** OrderProerties {

**private** ItemProperties item = **new** ItemProperties();

**public** ItemProperties getItem() {

**return** item;

}

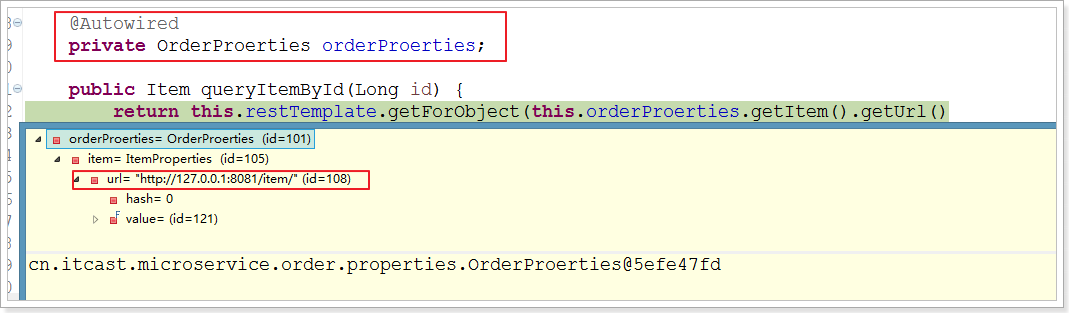
**public** **void** setItem(ItemProperties item) {

**this**.item = item;

}

}

第三步，在Itemservice中注入该对象：



可以看出，这种解决方案比第一种好很多。更加的方便的。

思考：这样是否还存在问题？如果商品的微服务有多个怎么办？

# Spring Cloud快速入门

## 分析硬编码的问题

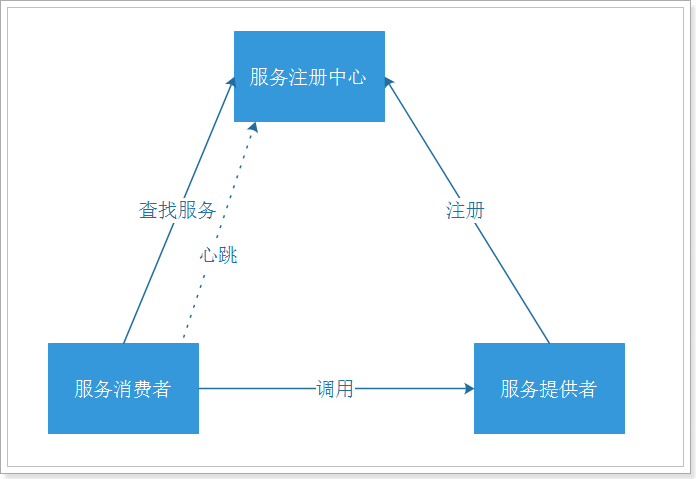
通过前面5.4、5.5的实现，我们视乎已经解决了url硬编码的问题，但是我们想想：

1. 如果商品微服务的ip地址发生了变更，订单微服务中的配置文件也需要跟着修改
2. 如果商品微服务有多个，那么在订单微服务中又该如何写地址？

那应该怎么解决呢？ -- 通过服务注册、发现的机制来完成。

## 微服务注册与发现

原理示意图：

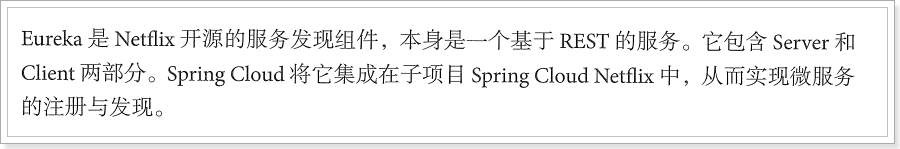


由上图可以看出：

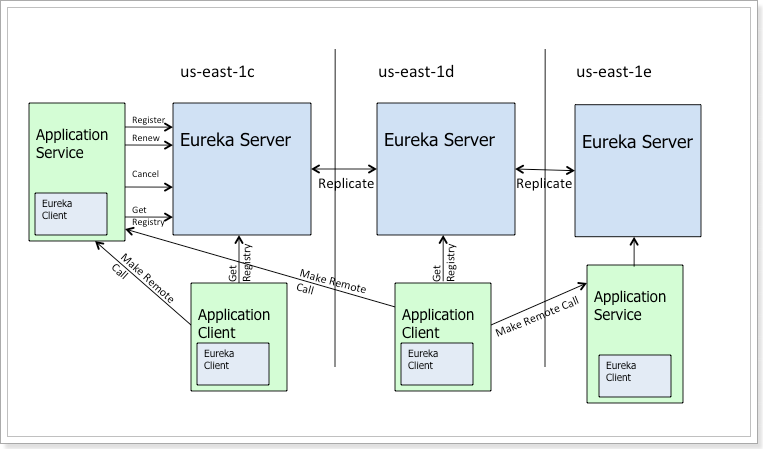
1. 服务提供者将服务注册到注册中心
2. 服务消费者通过注册中心查找服务
3. 查找到服务后进行调用（这里就是无需硬编码url的解决方案）
4. 服务的消费者与服务注册中心保持心跳连接，一旦服务提供者的地址发生变更时，注册中心会通知服务消费者

## 注册中心Eureka

Spring Cloud提供了多种注册中心的支持，如：Eureka、ZooKeeper等。推荐使用Eureka。



### 原理



Eureka包含两个组件：Eureka Server和Eureka Client。

Eureka Server提供服务注册服务，各个节点启动后，会在Eureka Server中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到。

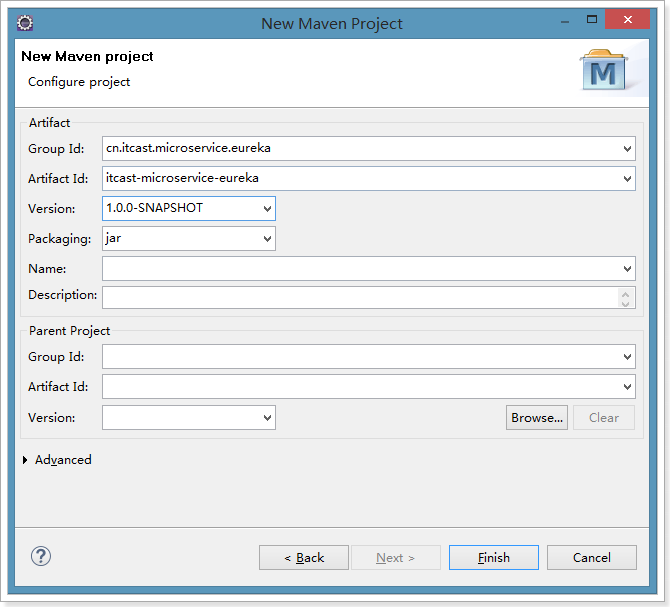
Eureka Client是一个java客户端，用于简化与Eureka Server的交互，客户端同时也就别一个内置的、使用轮询(round-robin)负载算法的负载均衡器。

在应用启动后，将会向Eureka Server发送心跳,默认周期为30秒，如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，Eureka Server将会从服务注册表中把这个服务节点移除(默认90秒)。

Eureka Server之间通过复制的方式完成数据的同步，Eureka还提供了客户端缓存机制，即使所有的Eureka Server都挂掉，客户端依然可以利用缓存中的信息消费其他服务的API。综上，Eureka通过心跳检查、客户端缓存等机制，确保了系统的高可用性、灵活性和可伸缩性。

### 编写Eureka Server

第一步：创建Maven工程：



第二步，导入依赖：

这里需要导入Spring Cloud的管理依赖。

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>cn.itcast.microservice.eureka</groupId>

<artifactId>itcast-microservice-eureka</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<!-- 导入Spring Cloud的依赖管理 -->

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR3</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<!-- 导入Eureka服务的依赖 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<!-- 资源文件拷贝插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

<!-- java编译插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

</project>

第三步，编写程序启动类：

**package** cn.itcat.microservice;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer //申明这是一个Eureka服务

@SpringBootApplication

**public** **class** EurekaServer {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(EurekaServer.**class**, args);

}

}

第四步，编写application.yml配置文件：

server:

port: 6868 #服务端口

eureka:

client:

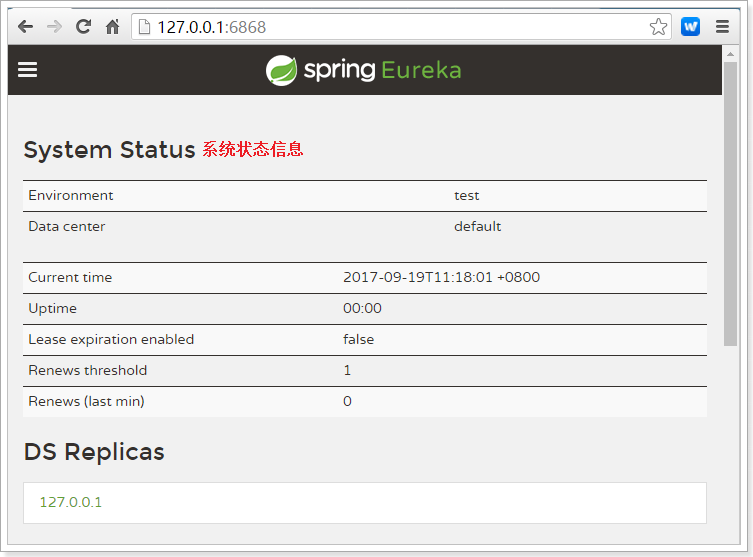
registerWithEureka: **false** #是否将自己注册到Eureka服务中，本身就是所有无需注册

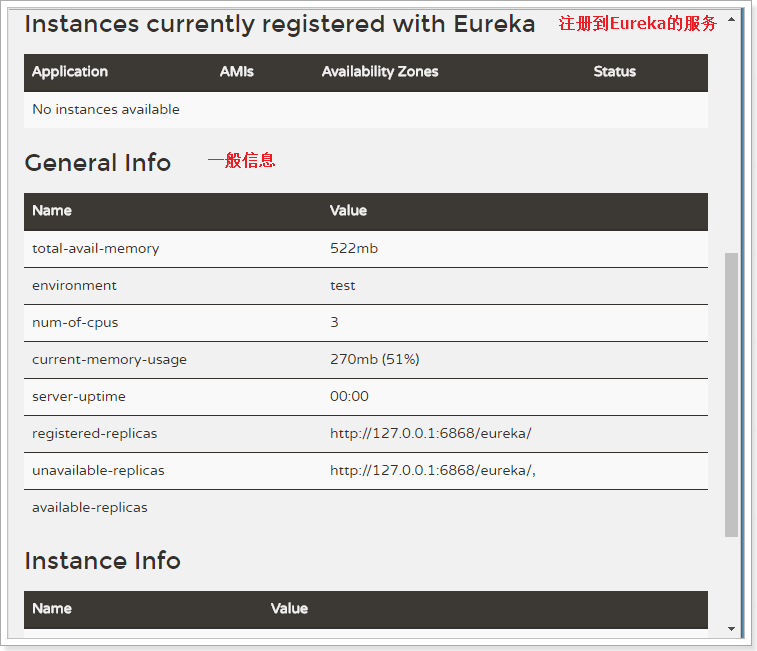
fetchRegistry: **false** #是否从Eureka中获取注册信息

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: [http://127.0.0.1:${server.port}/eureka/](http://127.0.0.1:$%7bserver.port%7d/eureka/)

第五步，启动程序做测试：





### 将商品微服务注册到Eureka

接下来，我们需要将商品的微服务注册到Eureka服务中。

第一步：修改pom文件，引入Spring Cloud的管理依赖以及eureka服务依赖。

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>cn.itcast.microservice</groupId>

<artifactId>itcast-microservice-item</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<!-- 导入Spring Cloud的依赖管理 -->

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR3</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- 导入Eureka服务的依赖 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<!-- 资源文件拷贝插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

<!-- java编译插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

</project>

第二步，修改application.yml配置文件：

server:

port: 8081 #服务端口

spring:

application:

name: itcast-microservice-item #指定服务名

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

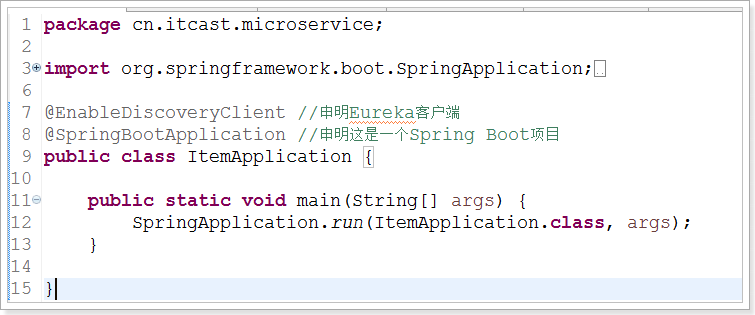
serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://127.0.0.1:6868/eureka/

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

第三步，修改启动类，增加@EnableDiscoveryClient注解：



第四步，启动测试：



至此，我们已经将自己的微服务注册到Eureka server中了。

## 订单系统从Eureka发现服务

之前我们在订单系统中是将商品微服务的地址进行了硬编码，现在，由于已经将商品服务注册到Eureka中，所以，只需要从Eureka中发现服务即可。

第一步，在订单系统中添加依赖：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>cn.itcast.microservice.order</groupId>

<artifactId>itcast-microservice-order</artifactId>

<version>1.0.0.-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<!-- 导入Spring Cloud的依赖管理 -->

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR3</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.squareup.okhttp3</groupId>

<artifactId>okhttp</artifactId>

<version>3.9.0</version>

</dependency>

<!-- 导入Eureka服务的依赖 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<!-- 资源文件拷贝插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

<!-- java编译插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

</project>

第二步，修改application.yml配置文件：

server:

port: 8082 #服务端口

itcast:

item:

url: http://127.0.0.1:8081/item/

spring:

application:

name: itcasst-microservice-order #指定服务名

eureka:

client:

registerWithEureka: **false** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://127.0.0.1:6868/eureka/

第三步，修改ItemService的实现逻辑：

**package** cn.itcast.microservice.order.service;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.cloud.client.ServiceInstance;

**import** org.springframework.cloud.client.discovery.DiscoveryClient;

**import** org.springframework.stereotype.Service;

**import** org.springframework.web.client.RestTemplate;

**import** cn.itcast.microservice.order.pojo.Item;

@Service

**public** **class** ItemService {

// Spring框架对RESTful方式的http请求做了封装，来简化操作

@Autowired

**private** RestTemplate restTemplate;

@Autowired

**private** DiscoveryClient discoveryClient;

@Value("${itcast.item.url}")

**private** String itcastItemUrl;

**public** Item queryItemById(Long id) {

String serviceId = "itcast-microservice-item";

List<ServiceInstance> instances = **this**.discoveryClient.getInstances(serviceId);

**if**(instances.isEmpty()){

**return** **null**;

}

// 为了演示，在这里只获取一个实例

ServiceInstance serviceInstance = instances.get(0);

String url = serviceInstance.getHost() + ":" + serviceInstance.getPort();

**return** **this**.restTemplate.getForObject("http://" + url + "/item/" + id, Item.**class**);

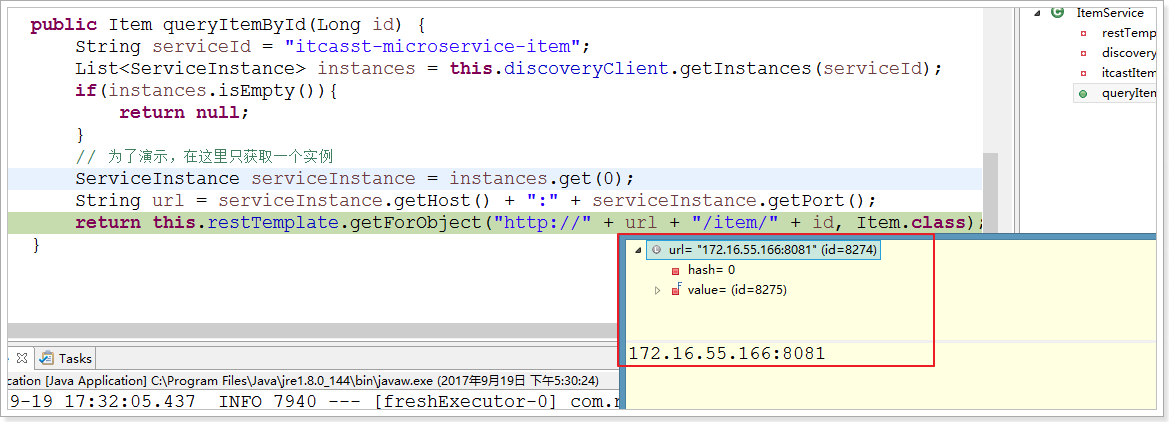
}

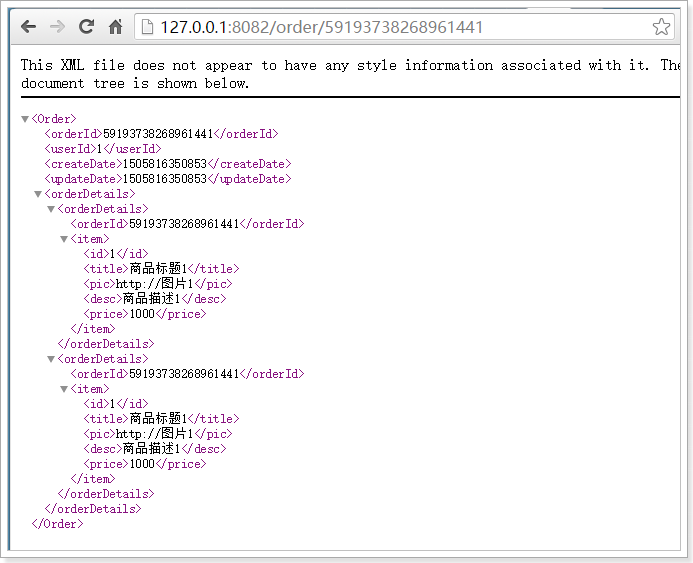
}

第四步，在启动类中添加@EnableDiscoveryClient注解



第五步，启动测试





可以看到以及获取到数据，但是，我们发现响应的数据变成了xml结构。

## 解决响应变成xml的问题

由于我们引入了eureka server的依赖，导致破坏了之前SpringMVC默认的配置，从而导致了响应成了xml。

解决方法：排除eureka server中的xml依赖，如下：

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

<exclusions>

<exclusion>

<groupId>com.fasterxml.jackson.dataformat</groupId>

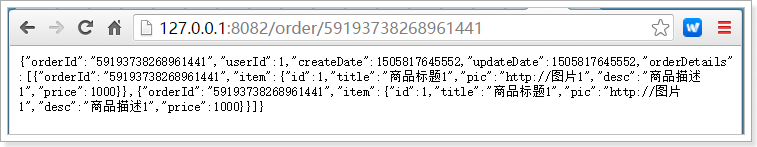
<artifactId>jackson-dataformat-xml</artifactId>

</exclusion>

</exclusions>

</dependency>

测试结果：



# 注册中心Eureka

## 为Eureka添加用户认证

在前面的示例中，我们可以看到我们需要登录即可访问到Eureka服务，这样其实是不安全的。

接下来，我们为Eureka添加用户认证。

第一步，为itcast-microservice-eureka添加安全认证依赖：

<!-- 安全认证 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

第二步，增加application.yml配置文件：

security:

basic:

enable: **true** #开启基于HTTP basic的认证

user: #配置用户的账号信息

name: itcast

password: itcast123

第三步，重新启动Eureka服务进行测试：



输入正确的用户名密码即可登录。

这时，服务提供者注册到Eureka时会报错：

2017-09-20 18:29:17.144 WARN 7200 --- [freshExecutor-0] c.n.d.s.t.d.RetryableEurekaHttpClient : Request execution failure with status code 401; retrying on another server if available

2017-09-20 18:29:17.155 ERROR 7200 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient\_ITCAST-MICROSERVICE-ITEM/172.16.55.166:itcast-microservice-item:8081 - was unable to refresh its cache! status = Cannot execute request on any known server

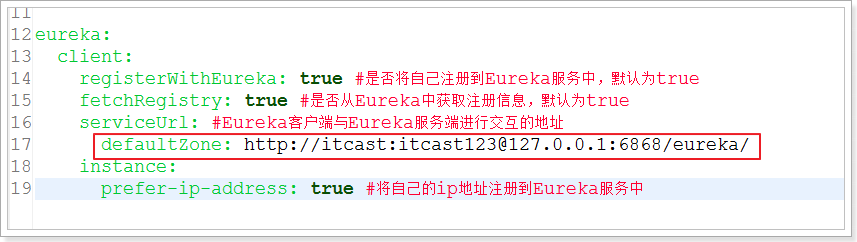
所以，需要在服务注册时也需要设置用户名和密码。

### 服务注册时设置账户信息

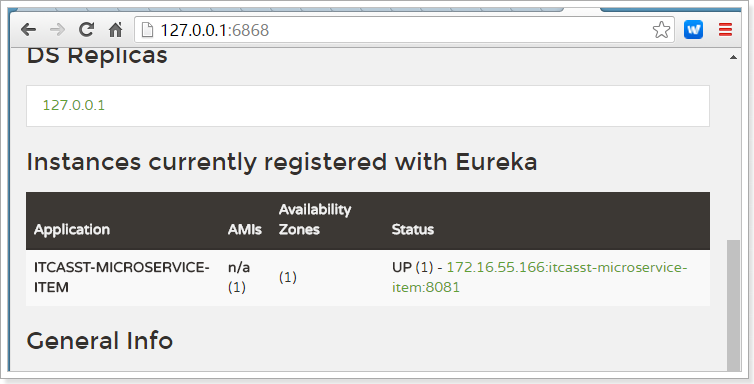
服务注册到有认证需求的注册中心时，需要设置如下信息：

<http://USER:PASSWORD@127.0.0.1:6868/eureka/>

配置：

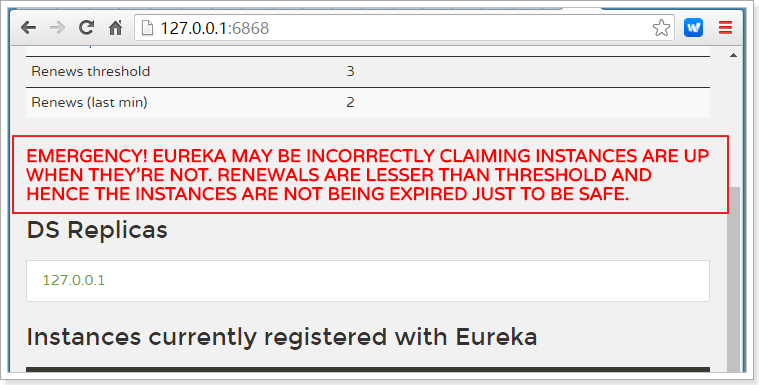


重新启动测试：

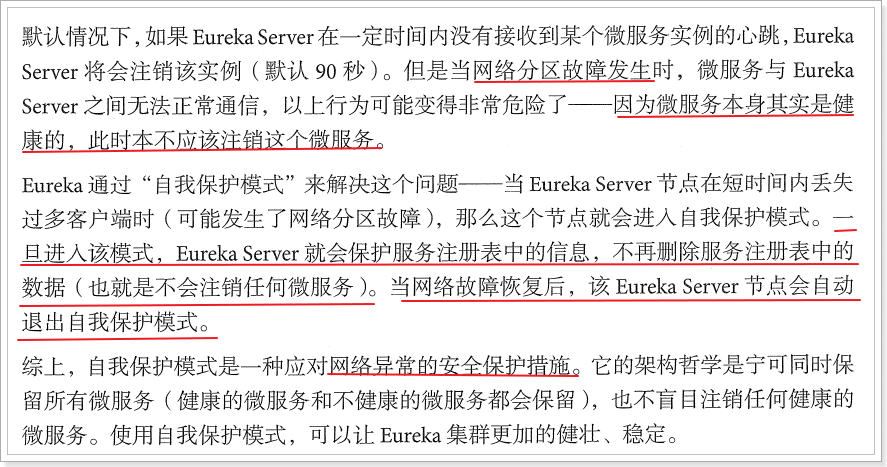


可以看到已经注册到了Eureka服务注册中心。

## Eureka的自我保护模式



如图，当前Eureka进入了自我保护模式。



所以，一般进入自我保护模式，无需处理。如果，需要禁用自我保护模式，只需要在配置文件中添加配置即可：

eureka:

client:

registerWithEureka: **false** #是否将自己注册到Eureka服务中，本身就是所有无需注册

fetchRegistry: **false** #是否从Eureka中获取注册信息

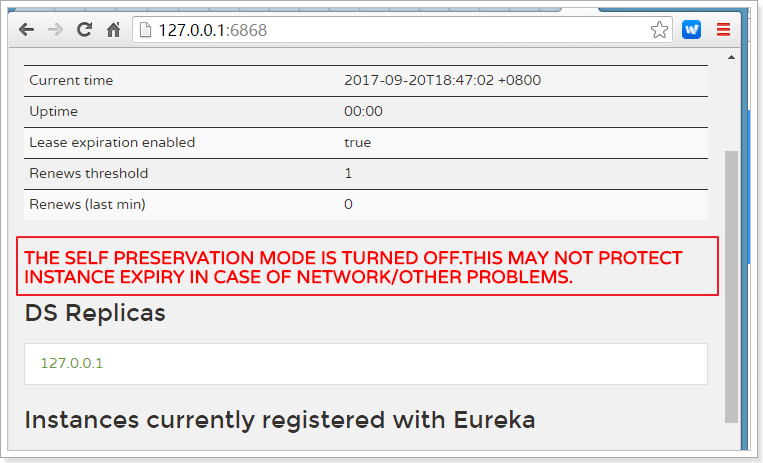
serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://127.0.0.1:${server.port}/eureka/

server:

enable-self-preservation: **false** #禁用自我保护模式

重新启动服务查看效果：



提示，如果禁用自我保护模式，在网络通信故障下会出现问题。

## Eureka的高可用

前面的测试，我们会发现，Eureka服务是一个单点服务，在生产环境就会出现单点故障，为了确保Eureka服务的高可用，我需要搭建Eureka服务的集群。

搭建Eureka集群非常简单，只要启动多个Eureka服务并且让这些服务之间彼此进行注册即可实现。

第一步，修改itcast-microservice-eureka的application.yml文件：

server:

port: 6868 #服务端口

spring:

application:

name: itcasst-microservice-eureka #指定服务名

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，本身就是所有无需注册

fetchRegistry: **true** #是否从Eureka中获取注册信息

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://itcast:itcast123@127.0.0.1:6869/eureka/

server:

enable-self-preservation: **true** #禁用自我保护模式

security:

basic:

enable: **true** #开启基于HTTP basic的认证

user: #配置用户的账号信息

name: itcast

password: itcast123

第二步，修改配置文件，再启动一个Eureka服务，进行重启测试：

server:

port: 6869 #服务端口

spring:

application:

name: itcast-microservice-eureka #指定服务名

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，本身就是所有无需注册

fetchRegistry: **true** #是否从Eureka中获取注册信息

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

server:

enable-self-preservation: **true** #禁用自我保护模式

security:

basic:

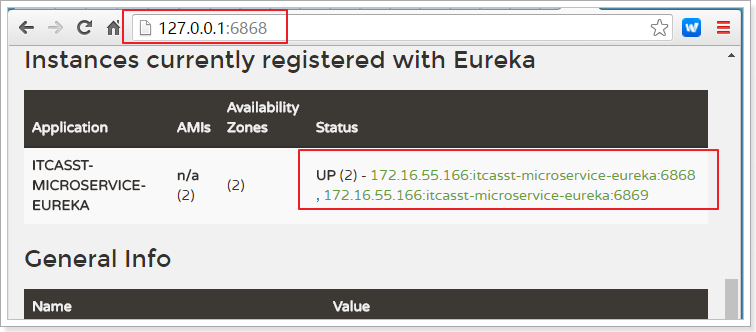
enable: **true** #开启基于HTTP basic的认证

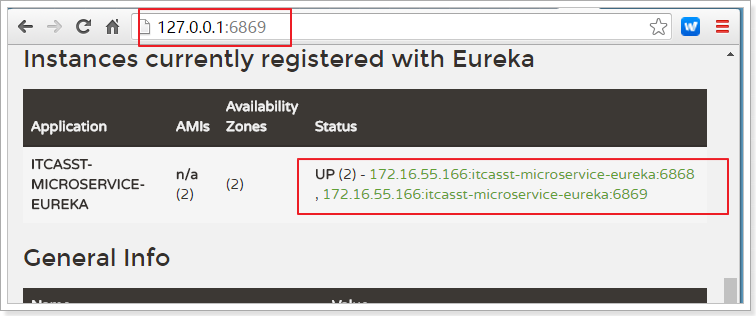
user: #配置用户的账号信息

name: itcast

password: itcast123

测试结果：





可以看到，2个Eureka服务进行了彼此注册。

## 将服务注册到Eureka集群

服务注册到Eureka集群时，可以指定多个，也可以指定一个Eureka服务（因为Eureka服务集群间彼此互联）。

修改itcast-microservice-item的application.yml配置文件：

server:

port: 8081 #服务端口

spring:

application:

name: itcast-microservice-item #指定服务名

logging:

level:

org.springframework: DEBUG

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

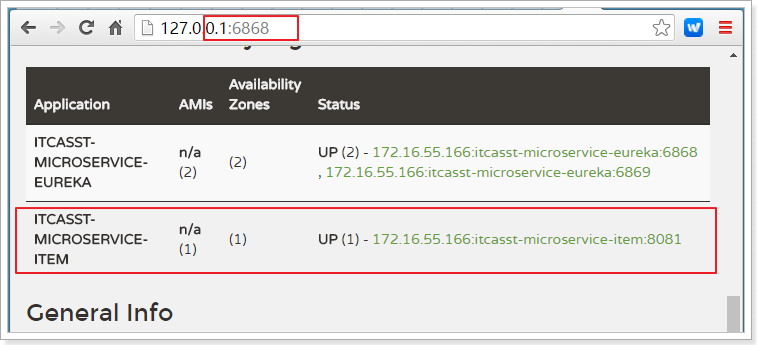
serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

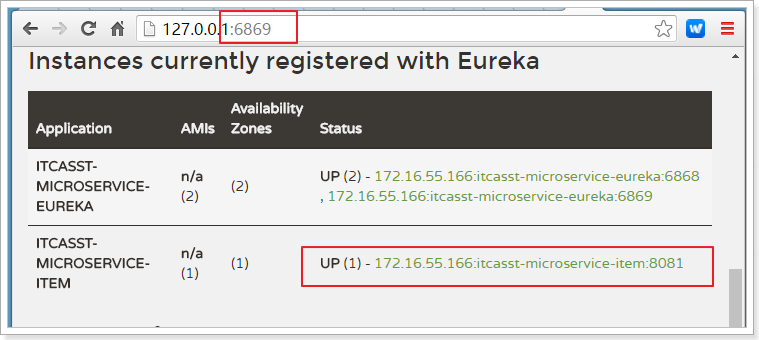
defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/,http://itcast:itcast123@127.0.0.1:6869/eureka/

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

重启启动，测试：





可以通过停止Eureka服务进行测试，结果会发现集群是高可用。

## 指定服务的IP地址

在服务的提供者配置文件中可以指定ip地址，如下：

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

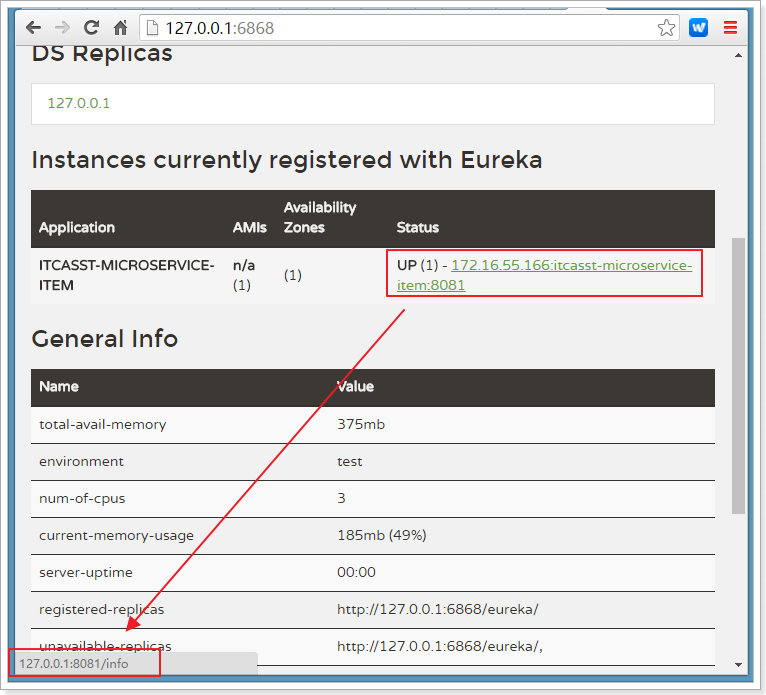
serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

ip-address: 127.0.0.1



## 指定实例id

通过instance-id 参数指定服务注册到Eureka中的服务实例id：

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

ip-address: 127.0.0.1

instance-id: ${spring.application.name}:${server.port} #指定实例id

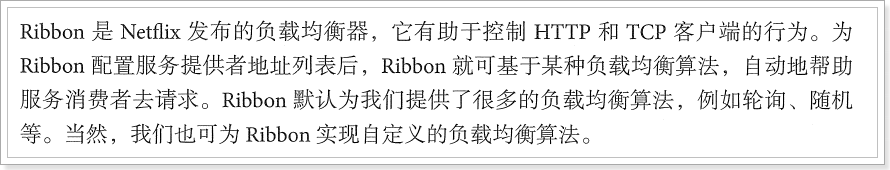
# 使用Ribbon实现负载均衡

首先，我们思考一个问题，如果为同一个的提供者在Eureka中注册了多个服务，那么客户端该如何选择服务呢？

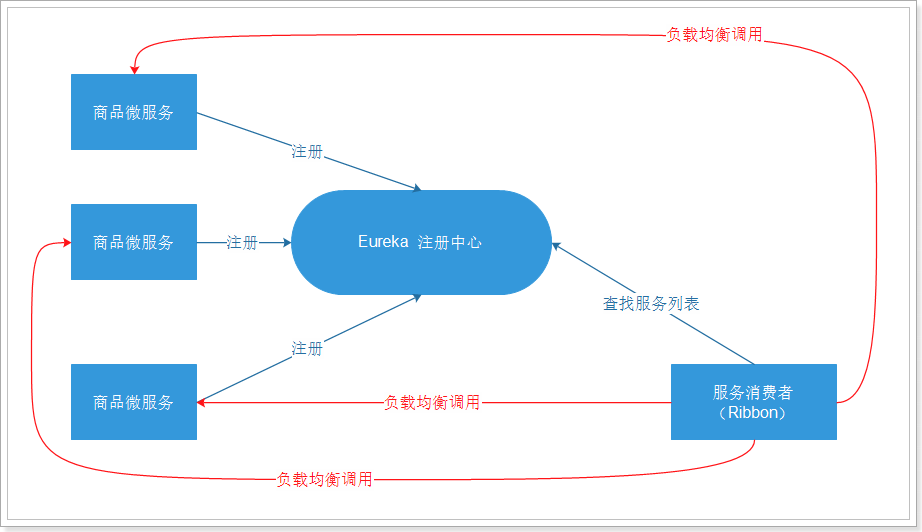
这时，就需要在客户端实现服务的负载均衡。

在Spring Cloud中推荐使用Ribbon来实现负载均衡。

## Ribbon简介



## 架构



## 开始使用Ribbon

### 为itcast-microservice-order增加ribbon依赖

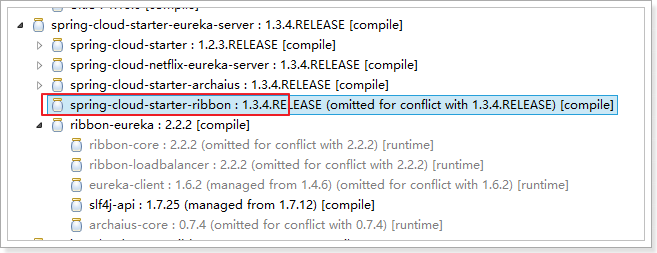
<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

其实，该依赖是可以省略的，因为>spring-cloud-starter-eureka-server中已经包含了spring-cloud-starter-ribbon：



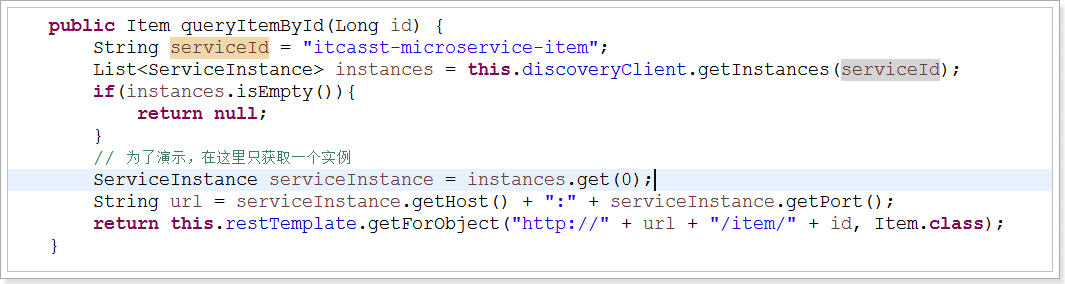
### 为RestTemplate设置@LoadBalanced注解



这样，RestTemplate就具备了负载均衡的功能。

### 改造ItemService的实现

之前的实现：



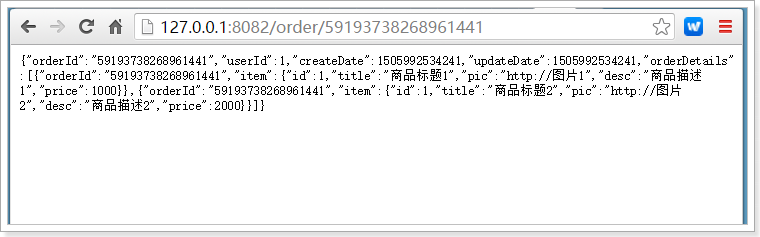
改造后的实现：



可以发现，实现更加简化了。

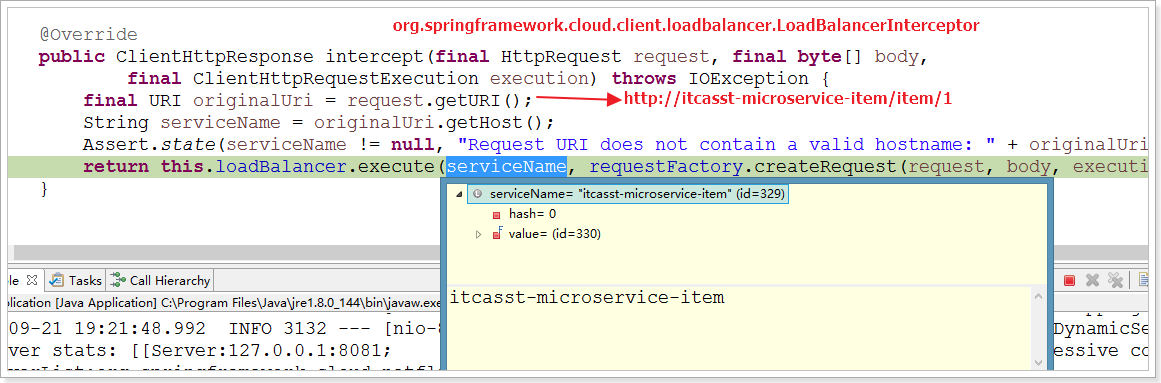
### 重启订单服务进行测试

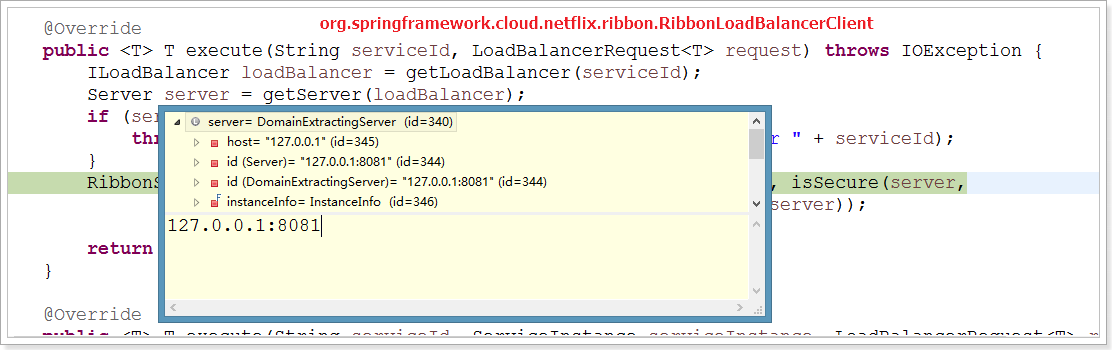
测试结果：



结果显示，可以正常获取到商品数据。

内部原理：

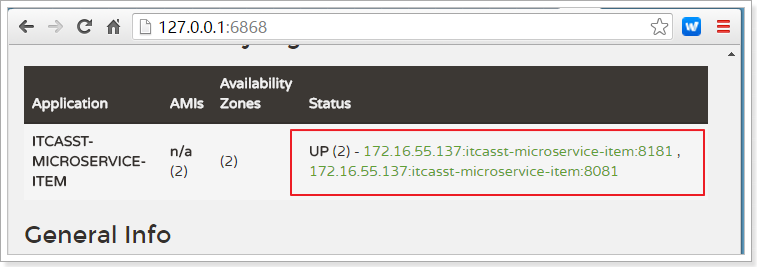




在执行请求前会经过org.springframework.cloud.client.loadbalancer.LoadBalancerInterceptor这个拦截器，并且通过org.springframework.cloud.netflix.ribbon.RibbonLoadBalancerClient中，通过serverId查找服务地址，然后在去做真正的请求。

### 测试负载均衡

测试方法：

第一步，启动2个itcast-microservice-item服务（多个也可以）  


第二步，编写测试单元测试用例，导入测试依赖：

<!-- 引入SpringBoot的单元测试 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

</dependency>

第三步，编写测试用例：

**package** cn.itcast.microservice.order.service;

**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.cloud.client.ServiceInstance;

**import** org.springframework.cloud.client.loadbalancer.LoadBalancerClient;

**import** org.springframework.context.annotation.Import;

**import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

**import** cn.itcast.microservice.OrderApplication;

@RunWith(SpringJUnit4ClassRunner.**class**)

@SpringBootTest

@Import(OrderApplication.**class**)

**public** **class** ItemServiceTest {

@Autowired

**private** LoadBalancerClient loadBalancerClient;

@Test

**public** **void** test(){

String serviceId = "itcast-microservice-item";

**for** (**int** i = 0; i < 100; i++) {

ServiceInstance serviceInstance = **this**.loadBalancerClient.choose(serviceId);

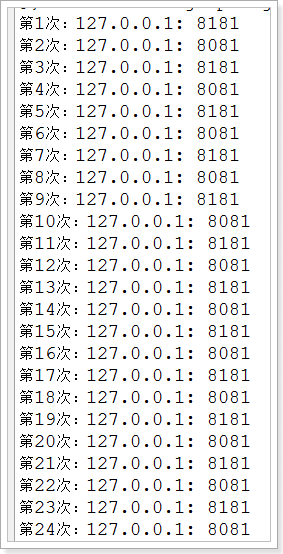
System.***out***.println("第"+(i+1)+"次：" + serviceInstance.getHost() + ": " + serviceInstance.getPort());

}

}

}

测试结果：



## 设置负载均衡的为随机

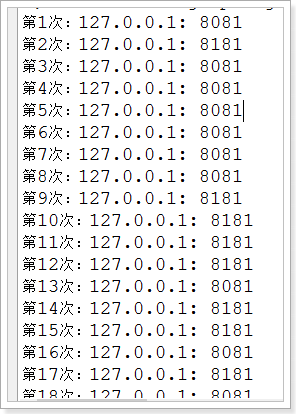
配置：

itcast-microservice-item:

ribbon:

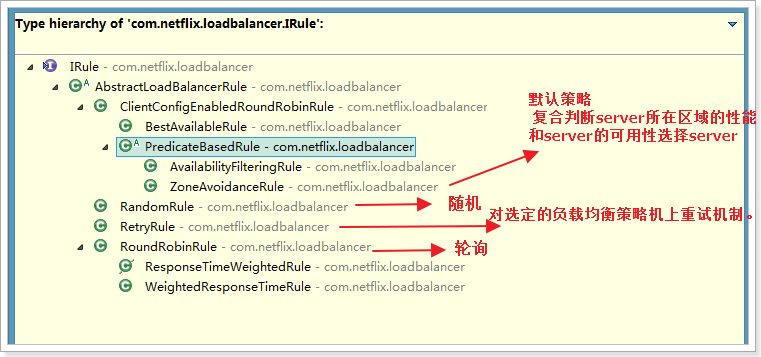
NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

测试：

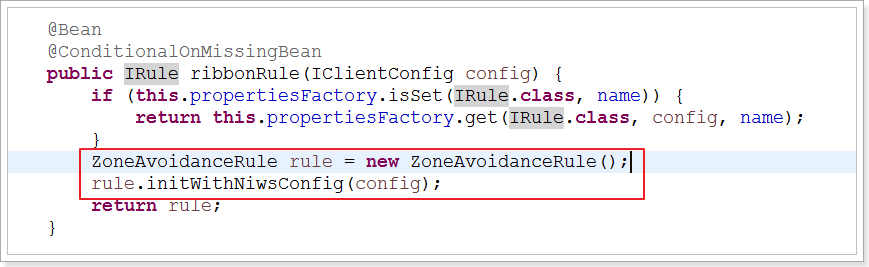


## 其它策略

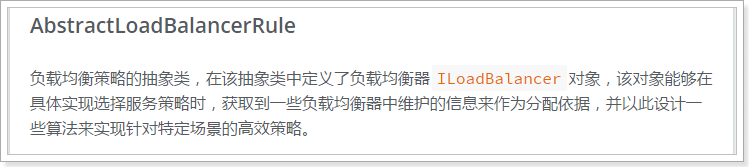
接口：com.netflix.loadbalancer.IRule，其实现类：

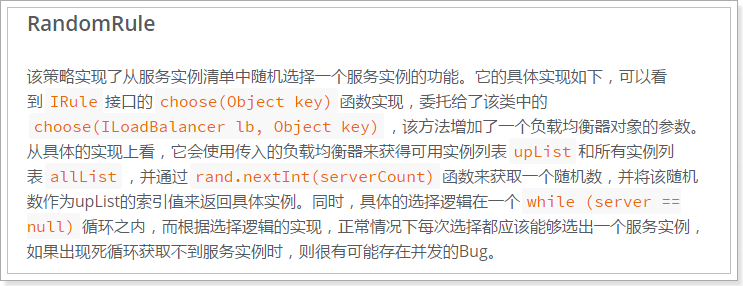


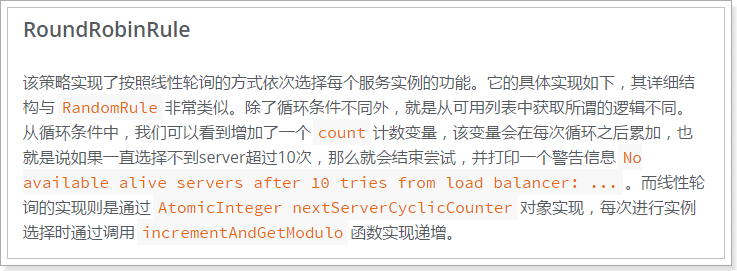
默认策略：



其它策略：





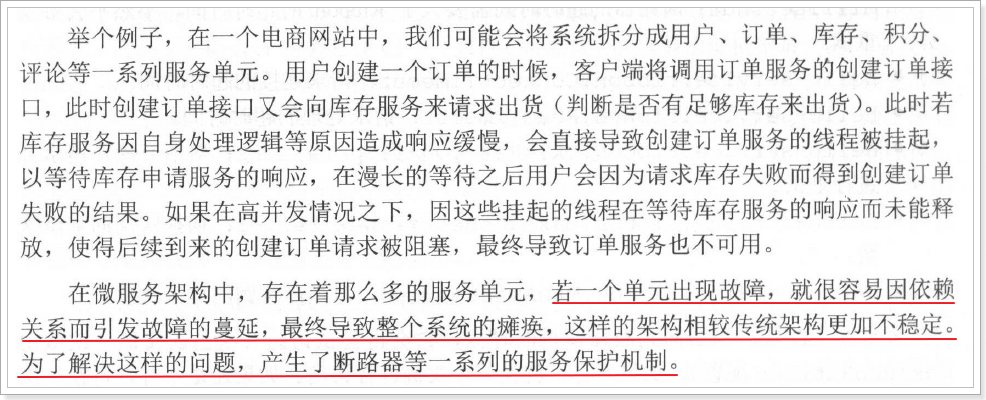


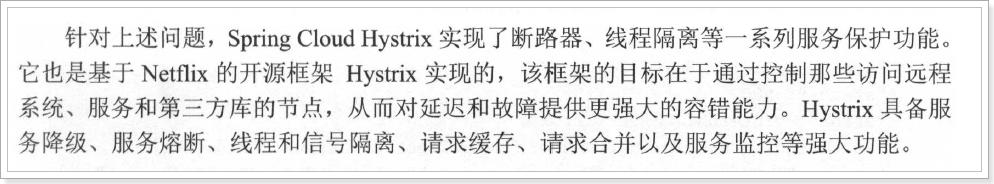


建议：采用默认策略即可。

# 容错保护：Hystrix

## 分析

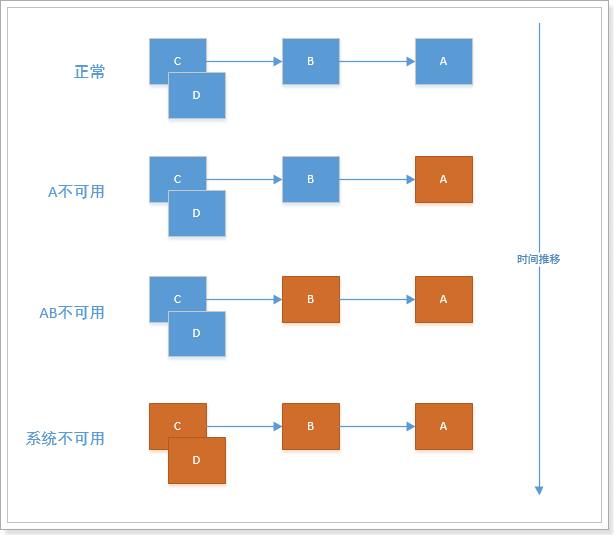




## 雪崩效应

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个系统不可用的情况，这种现象被称为服务雪崩效应。服务雪崩效应是一种因“服务提供者”的不可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。

如果下图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。

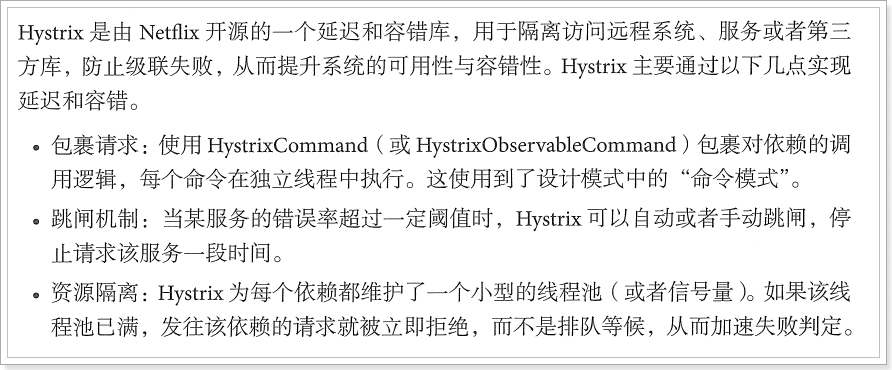


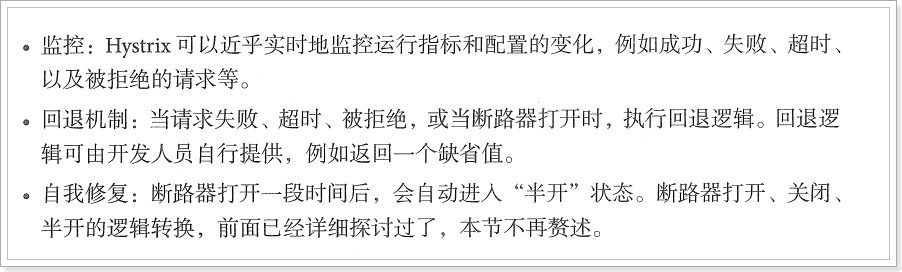
## Hystrix简介

主页：<https://github.com/Netflix/Hystrix/>



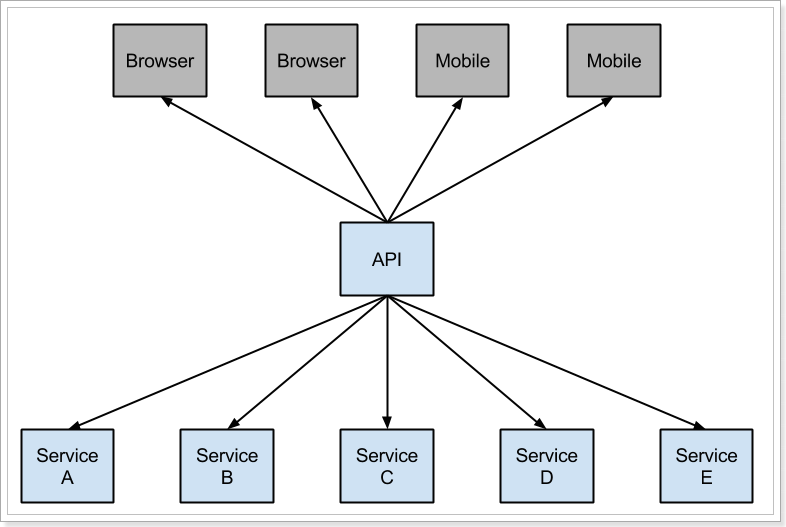




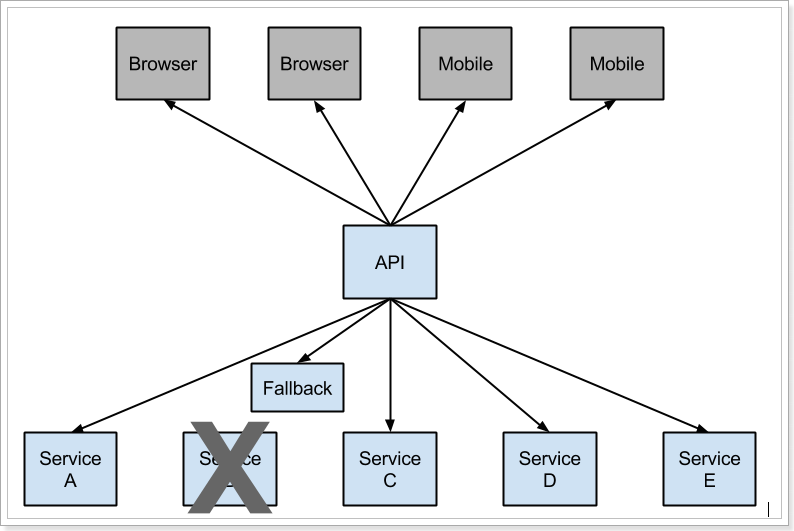


## 原理说明

正常情况：



当对特定服务的呼叫达到一定阈值时（Hystrix中的默认值为5秒内的20次故障），电路打开，不进行通讯。并且是一个隔离的线程中进行的。



## 快速入门

在itcast-microservice-order系统中增加Hystrix实现容错。

### 导入依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

### 修改ItemService的queryItemById方法

@HystrixCommand(fallbackMethod = "queryItemByIdFallbackMethod") // 进行容错处理

**public** Item queryItemById(Long id) {

String serviceId = "itcast-microservice-item";

**return** **this**.restTemplate.getForObject("http://" + serviceId + "/item/" + id, Item.**class**);

}

**public** Item queryItemByIdFallbackMethod(Long id){ // 请求失败执行的方法

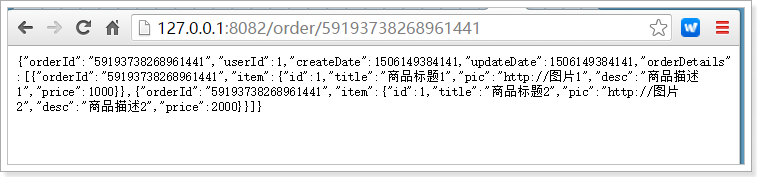
**return** **new** Item(id, "查询商品信息出错!", **null**, **null**, **null**);

}

### 在启动类OrderApplication添加@EnableHystrix注解

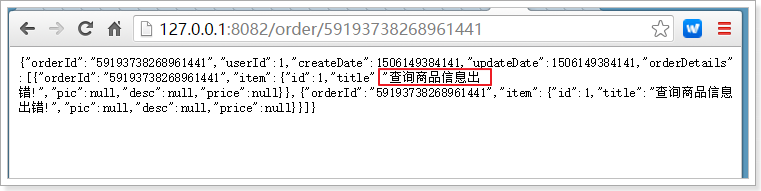


### 重新启动进行测试



测试一切正常。

接下来，我们把商品服务停止进行测试：



可以看到，订单服务正常，但是查询商品服务已停止服务，查询到的是错误信息。

由此可见，商品服务的宕机并没有影响订单服务的正常工作，起到的容错效果。