

Spring Cloud 是分布式微服务架构的一站式解决方案, 也可以理解为Spring Cloud是帮助我们管理这些微服务的, 那么在学习Spring Cloud之前, 我们需要先具备一些微服务.

本课程的学习, 是基于JDK17+来进行开发的, 本章节内容主要分两个方面:

1. 开发环境安装
2. 项目搭建

## 1. 开发环境安装

### 1.1 JDK

#### 1.1.1 JDK版本介绍

Oracle从JDK9开始每半年发布一个新版本, 新版本发布后, 老版本就不再进行维护. 但是会有几个长期维护的版本.

目前长期维护的版本有: JDK8, JDK11, JDK17, JDK21

在 JDK版本的选择上, 尽量选择长期维护的版本.

#### 为什么选择JDK17?

Spring Cloud 是基于 SpringBoot 进行开发的, SpringBoot 3.X以下的版本, Spring官方已不再进行维护(还可以继续使用), SpringBoot 3.X的版本, 使用的JDK版本基线为JDK17. 鉴于JDK21 是2023.09月发布的, 很多功能还没有在生产环境验证, 所以本课程选择使用JDK17来学习

#### 1.1.2 JDK17安装

参考 [📖JDK17安装](#)

### 1.2 MySQL安装

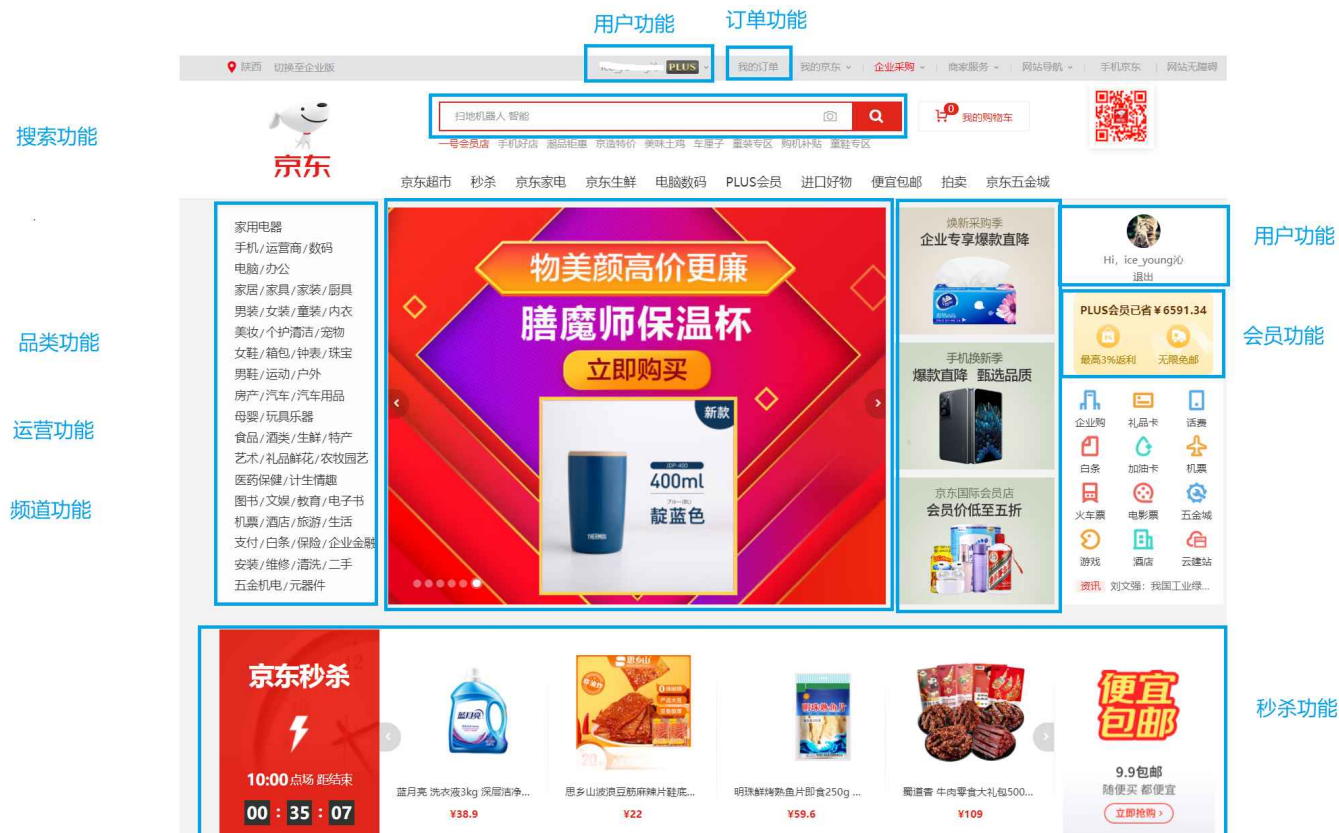
参考 [📖MySQL安装](#)

## 2. 案例介绍

### 2.1 需求

实现一个电商平台(不真实实现, 仅为演示)

一个电商平台包含的内容非常多, 以京东为例, 仅从首页上就可以看到巨多的功能



我们该如何实现呢? 如果把这些功能全部写在一个服务里, 这个服务将是巨大的.

巨多的会员, 巨大的流量, 微服务架构是最好的选择.

微服务应用开发的第一步, 就是服务拆分. 拆分后才能进行"各自开发"

## 2.2 服务拆分

### 服务拆分原则

微服务到底多小才算"微", 这个在业界并没有明确的标准. 微服务并不是越小越好, 服务越小, 微服务架构的优点和缺点都会越来越明显.

服务越小, 微服务的独立性就会越来越高, 但同时, 微服务的数量也会越多, 管理这些微服务的难度也会提高. 所以服务拆分也要考虑场景.

还是以企业管理为例

企业中一个员工的工作内容与企业规模, 项目规模等都有关系.

在小公司, 一个员工可能需要负责很多部门的事情, 大公司的话, 一个部门的工作可能需要多个员工来处理.

拆分微服务一般遵循如下原则:

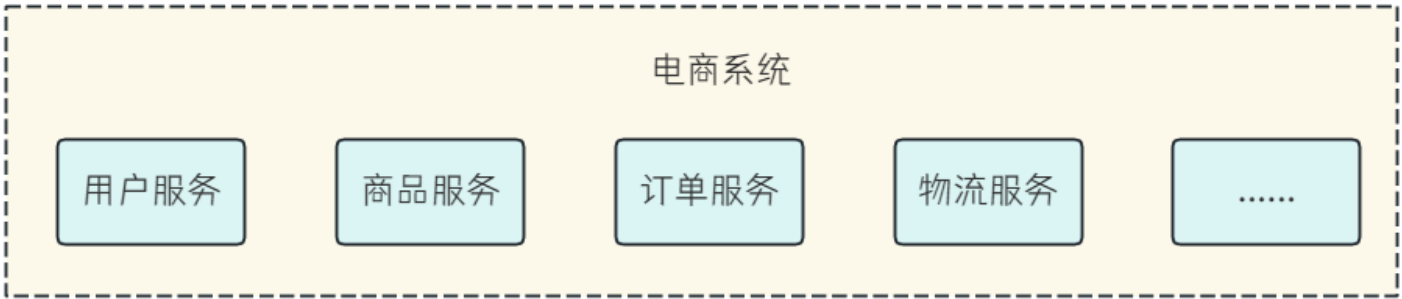
#### 1. 单一职责原则

单一职责原则原本是面向对象设计中的一个基本原则, 它指的是一个类应该专注于单一功能. 不要存在多于一个导致类变更的原因.

在微服务架构中, 一个微服务也应该只负责一个功能或业务领域, 每个服务应该有清晰的定义和边界, 只关注自己的特定业务领域.

组织团队也是, 一个人专注做一件事情的效率远高于同时关注多件事情.  
比如一个人同时管理和维护一份代码, 要比多个人同时维护多份代码的效率高.

比如电商系统

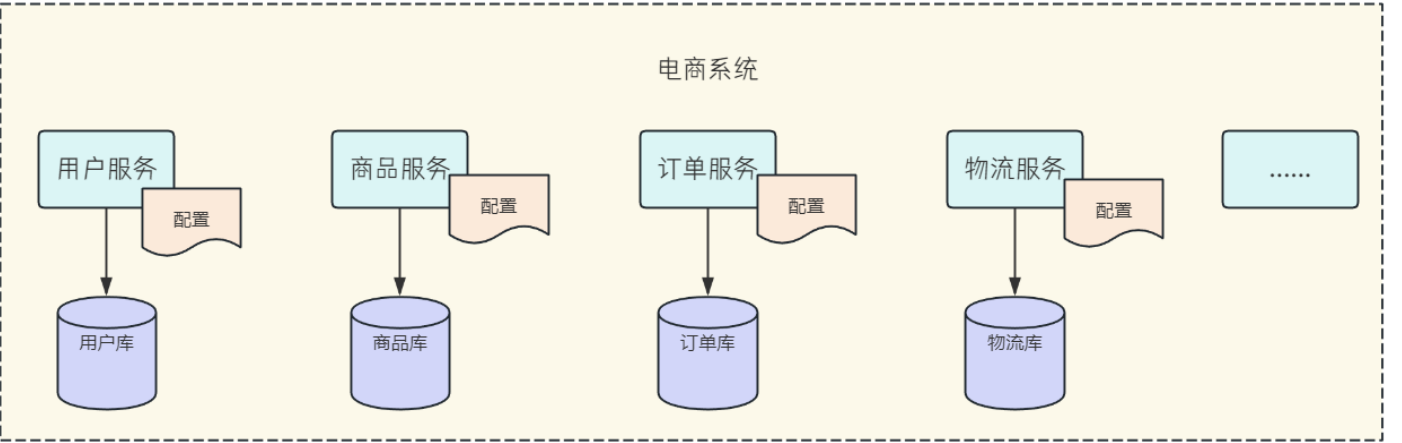


2. 服务自治

服务自治是指每个微服务都应该具备高度自治的能力, 即每个服务要能做到独立开发, 独立测试, 独立构建, 独立部署, 独立运行.

以上面的电商系统为例, 每一个微服务应该有自己的存储, 配置, 在进行开发, 构建, 部署, 运行和测试时, 并不需要过多关注其他微服务的状态和数据

比如企业管理  
每个部分负责每个部门的事情, 并且尽可能少的受其他团队影响  
研发部门只负责需求功能的开发, 而不负责需求文档的书写和UI的设计. 并且其他部门的人员变动, 流程变更, 也尽可能少的影响研发部门. 部门和部门之间尽可能自治.

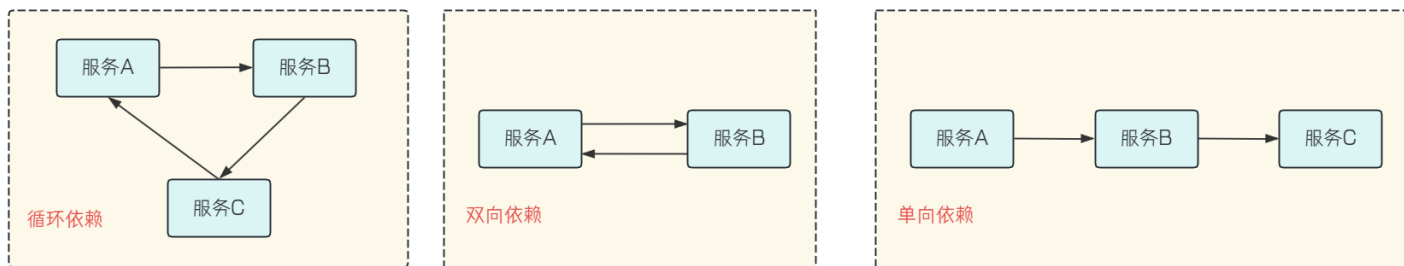


3. 单向依赖

微服务之间需要做到单向依赖, 严禁循环依赖, 双向依赖

循环依赖: A -> B -> C ->A

双向依赖: A -> B, B -> A



如果一些场景确实无法避免循环依赖或者双向依赖, 可以考虑使用消息队列等其他方式来实现

💡 微服务架构并无标准架构, 合适的就是最好的, 不然架构师大会也不会各个系统架构百花齐放了.

在架构设计的过程中, 坚持 "**合适优于业界领先**", 避免 "**过度设计**" (为了设计而设计).

罗马不是一天建成的, 很多业界领先方案并不是一群天才在某个时期一下子做出来的, 而是经过数年的发展逐步完善. 业界领先的方案大多是 "逼" 出来的, 随着业务的发展, 量变导致质变, 新的问题出现了, 当前的方案无法满足需求, 需要用新的方案来解决. 通过不断的创新和尝试, 业界领先的方案才得以形成.

## 服务拆分示例

一个完整的电商系统是庞大的, 当然这也不是咱们课程的重点, 咱们课程中重点关注如何使用Spring Cloud解决微服务架构中遇到的问题.

以订单列表为例:

2023-12-18 13:29:41	订单号: 9	过节热门小商品小店		过年过节生日发光户外玩具 孔雀开平3个	x1	泡泡鱼	¥63.00 在线支付	正在出库 跟踪 订单详情	选购京东服务 取消订单 查看发票
2023-12-18 13:25:28	订单号: 3	过节热门小商品小店		过年过节生日发光户外玩具 苟尾巴108根	x1	泡泡鱼	应付 ¥44.90 在线支付	等待付款 跟踪 订单详情	剩余22时49分 付款 取消订单 查看发票
2023-12-18 13:24:46	订单号: 1	过节热门小商品小店		过年过节生日发光户外玩具 满弟珍珠五盒共50个	x1	泡泡鱼	应付 ¥59.00 在线支付	等待付款 跟踪 订单详情	剩余22时48分 付款 取消订单 查看发票

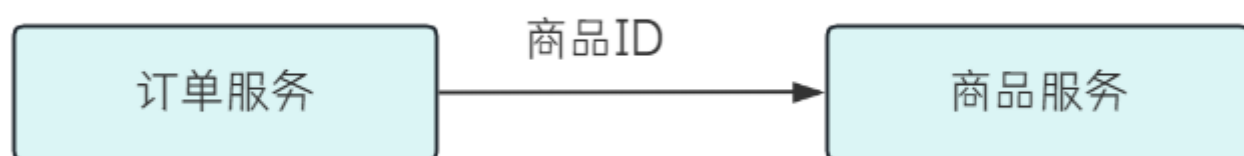
简单来看, 这个页面提供了以下信息:

1. 订单列表
2. 商品信息

根据服务的单一职责原则, 我们把服务进行拆分为: 订单服务, 商品服务

订单服务: 提供订单ID, 获取订单详细信息

商品服务: 根据商品ID, 返回商品详细信息.



💡 真实的订单服务和咱们课程中的并不一致, 企业中拆分的粒度更加细, 此处仅为举例, 不影响我们课程的学习, 咱们主要学习微服务的管理.

### 3. 数据准备

根据服务自治原则, 每个服务都应有自己独立的数据库

订单服务:

```
1 -- 建库
2 create database if not exists cloud_order charset utf8mb4;
3
4 -- 订单表
5 DROP TABLE IF EXISTS order_detail;
6 CREATE TABLE order_detail (
7     `id` INT NOT NULL AUTO_INCREMENT COMMENT '订单id',
8     `user_id` BIGINT ( 20 ) NOT NULL COMMENT '用户ID',
9     `product_id` BIGINT ( 20 ) NULL COMMENT '产品id',
10    `num` INT ( 10 ) NULL DEFAULT 0 COMMENT '下单数量',
11    `price` BIGINT ( 20 ) NOT NULL COMMENT '实付款',
12    `delete_flag` TINYINT ( 4 ) NULL DEFAULT 0,
13    `create_time` DATETIME DEFAULT now(),
14    `update_time` DATETIME DEFAULT now(),
15    PRIMARY KEY ( id )) ENGINE = INNODB DEFAULT CHARACTER
16    SET = utf8mb4 COMMENT = '订单表';
17
18 -- 数据初始化
19 insert into order_detail (user_id,product_id,num,price)
20 values
21 (2001, 1001,1,99), (2002, 1002,1,30), (2001, 1003,1,40),
22 (2003, 1004,3,58), (2004, 1005,7,85), (2005, 1006,7,94);
23
```

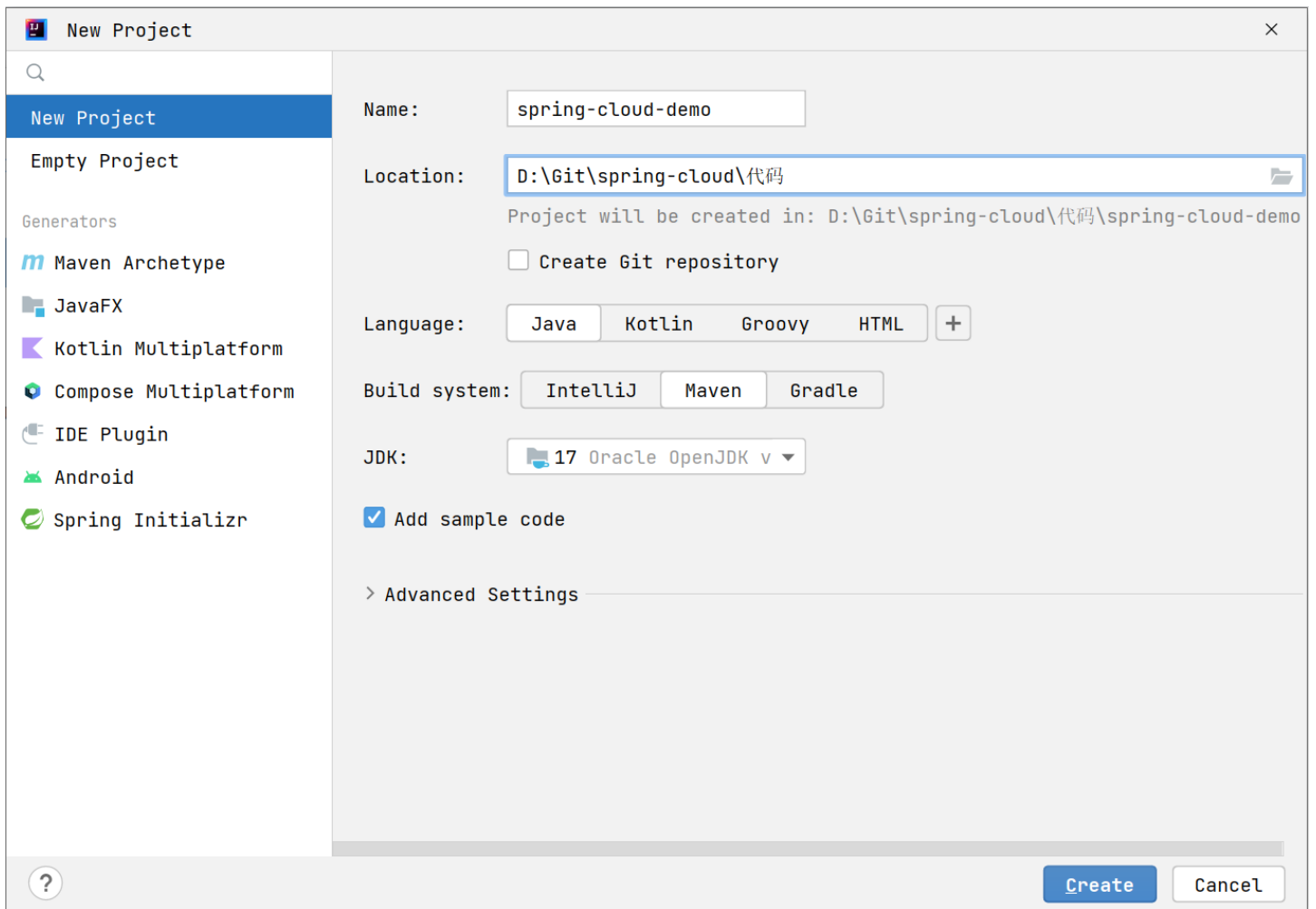
```
1 create database if not exists cloud_product charset utf8mb4;
2
3 -- 产品表
4 DROP TABLE IF EXISTS product_detail;
5 CREATE TABLE product_detail (
6     `id` INT NOT NULL AUTO_INCREMENT COMMENT '产品id',
7     `product_name` varchar ( 128 ) NULL COMMENT '产品名称',
8     `product_price` BIGINT ( 20 ) NOT NULL COMMENT '产品价格',
9     `state` TINYINT ( 4 ) NULL DEFAULT 0 COMMENT '产品状态 0-有效 1-下架',
10    `create_time` DATETIME DEFAULT now(),
11    `update_time` DATETIME DEFAULT now(),
12    PRIMARY KEY ( id )) ENGINE = INNODB DEFAULT CHARACTER
13    SET = utf8mb4 COMMENT = '产品表';
14
15 -- 数据初始化
16 insert into product_detail (id, product_name,product_price,state)
17 values
18 (1001,"T恤", 101, 0), (1002, "短袖",30, 0), (1003, "短裤",44, 0),
19 (1004, "卫衣",58, 0), (1005, "马甲",98, 0),(1006,"羽绒服", 101, 0),
20 (1007, "冲锋衣",30, 0), (1008, "袜子",44, 0), (1009, "鞋子",58, 0),
21 (10010, "毛衣",98, 0)
```

## 4. 工程搭建

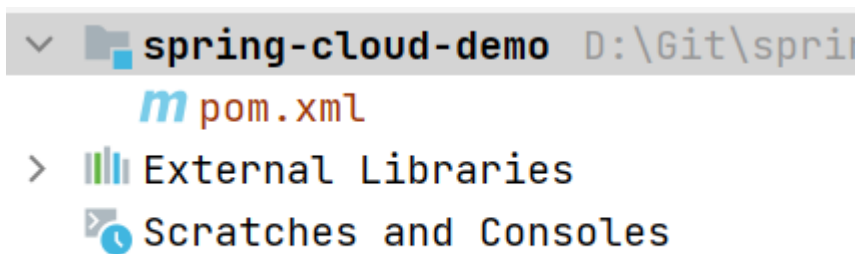
### 4.1 构建父子工程

#### 4.1.1 创建父工程

1. 创建一个空的Maven项目, 删除所有代码, 只保留pom.xml



目录结构:



## 2. 完善pom文件

使用properties来进行版本号的统一管理, 使用dependencyManagement来管理依赖, 声明父工程的打包方式为pom

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5             http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <groupId>org.example</groupId>
```

```
8     <artifactId>spring-cloud-demo</artifactId>
9     <packaging>pom</packaging>
10    <version>1.0-SNAPSHOT</version>
11
12    <parent>
13        <groupId>org.springframework.boot</groupId>
14        <artifactId>spring-boot-starter-parent</artifactId>
15        <version>3.1.6</version>
16        <relativePath/> <!-- lookup parent from repository -->
17    </parent>
18    <properties>
19        <maven.compiler.source>17</maven.compiler.source>
20        <maven.compiler.target>17</maven.compiler.target>
21        <java.version>17</java.version>
22        <mybatis.version>3.0.3</mybatis.version>
23        <mysql.version>8.0.33</mysql.version>
24        <spring-cloud.version>2022.0.3</spring-cloud.version>
25    </properties>
26
27    <dependencies>
28        <dependency>
29            <groupId>org.projectlombok</groupId>
30            <artifactId>lombok</artifactId>
31            <optional>true</optional>
32        </dependency>
33    </dependencies>
34
35    <dependencyManagement>
36        <dependencies>
37            <dependency>
38                <groupId>org.springframework.cloud</groupId>
39                <artifactId>spring-cloud-dependencies</artifactId>
40                <version>${spring-cloud.version}</version>
41                <type>pom</type>
42                <scope>import</scope>
43            </dependency>
44            <dependency>
45                <groupId>org.mybatis.spring.boot</groupId>
46                <artifactId>mybatis-spring-boot-starter</artifactId>
47                <version>${mybatis.version}</version>
48            </dependency>
49            <dependency>
50                <groupId>com.mysql</groupId>
51                <artifactId>mysql-connector-j</artifactId>
52                <version>${mysql.version}</version>
53            </dependency>
54            <dependency>
```



```

55         <groupId>org.mybatis.spring.boot</groupId>
56         <artifactId>mybatis-spring-boot-starter-test</artifactId>
57         <version>${mybatis.version}</version>
58         <scope>test</scope>
59     </dependency>
60 </dependencies>
61 </dependencyManagement>
62 </project>

```

## DependencyManagement 和 Dependencies

1. `dependencies`：将所依赖的jar直接加到项目中。子项目也会继承该依赖。
2. `dependencyManagement`：只是声明依赖，并不实现Jar包引入。如果子项目需要用到相关依赖，需要显式声明。如果子项目没有指定具体版本，会从父项目中读取version。如果子项目中指定了版本号，就会使用子项目中指定的jar版本。此外父工程的打包方式应该是pom，不是jar，这里需要手动使用 `packaging` 来声明。

SpringBoot 实现依赖jar包版本的管理，也是这种方式

```

205 <webjars-locator-core.version>0.52</webjars-locator-core.version>
206 <wsdl4j.version>1.6.3</wsdl4j.version>
207 <xml-maven-plugin.version>1.0.2</xml-maven-plugin.version>
208 <xmlunit2.version>2.9.1</xmlunit2.version>
209 <yasson.version>3.0.3</yasson.version>
210 </properties>
211 <dependencyManagement>
212 <dependencies>
213 <dependency>
214 <groupId>org.apache.activemq</groupId>
215 <artifactId>activemq-amqp</artifactId>
216 <version>${activemq.version}</version>
217 </dependency>
218 <dependency>
219 <groupId>org.apache.activemq</groupId>
220 <artifactId>activemq-blueprint</artifactId>
221 <version>${activemq.version}</version>
222 </dependency>
223 <dependency>
224 <groupId>org.apache.activemq</groupId>
225 <artifactId>activemq-broker</artifactId>

```

## 依赖Jar的版本判断

1 <dependencies>

```

2      <dependency>
3          <groupId>org.projectlombok</groupId>
4          <artifactId>lombok</artifactId>
5          <optional>true</optional>
6      </dependency>
7 </dependencies>
8
9 <dependencyManagement>
10     <dependencies>
11         <dependency>
12             <groupId>com.mysql</groupId>
13             <artifactId>mysql-connector-j</artifactId>
14             <version>${mysql.version}</version>
15         </dependency>
16     </dependencies>
17 </dependencyManagement>

```

上述代码中, lombok 会被直接引入到当前项目以及子项目中, mysql-connector-j 不会实际引入jar, 子项目继承时必须显式声明.

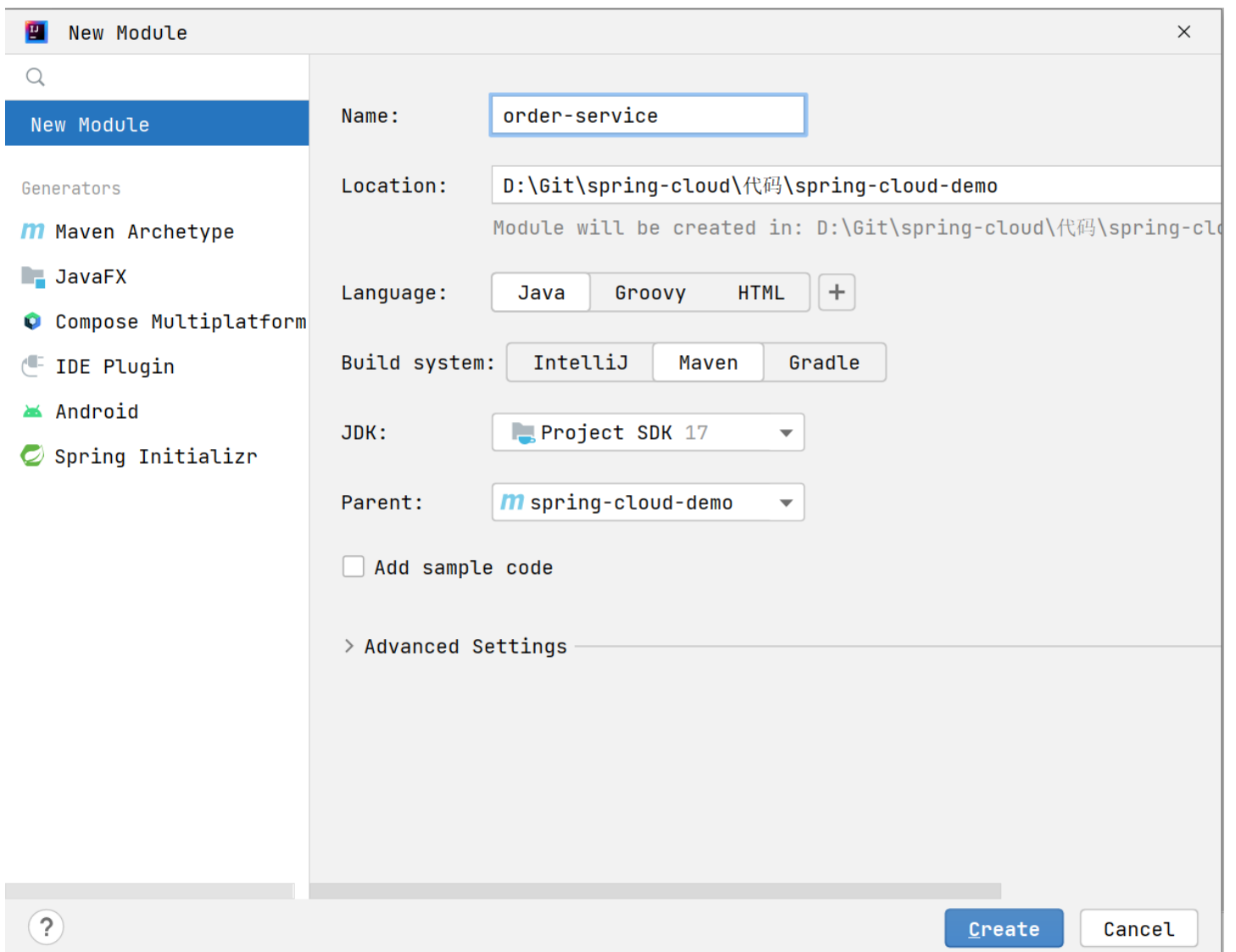
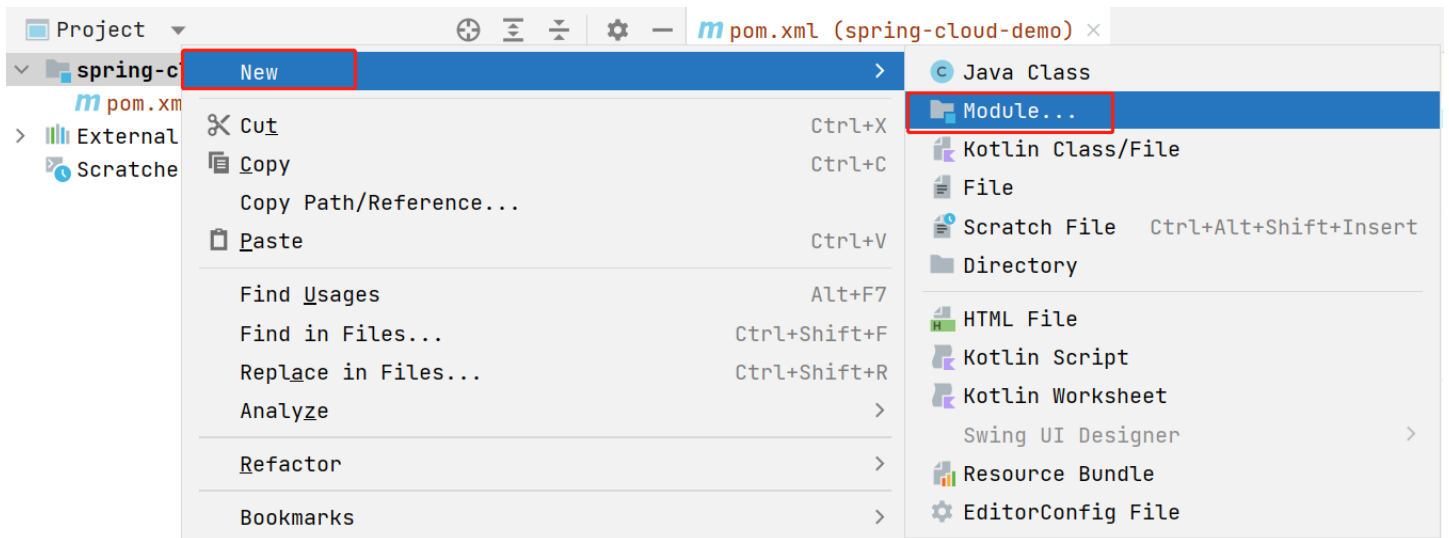
## Spring Cloud版本

Spring Cloud 是基于SpringBoot搭建的, 所以Spring Cloud 版本与SpringBoot版本有关

Release Train	Spring Boot Generation
2023.0.x aka Leyton	3.2.x
2022.0.x aka Kilburn	3.0.x, 3.1.x (Starting with 2022.0.3)
2021.0.x aka Jubilee	2.6.x, 2.7.x (Starting with 2021.0.3)
2020.0.x aka Ilford	2.4.x, 2.5.x (Starting with 2020.0.3)
Hoxton	2.2.x, 2.3.x (Starting with SR5)
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

咱们项目中使用的SpringBoot 版本为 3.1.6, 对应的Spring Cloud版本应该为2022.0.x, 选择任一就可以

### 4.1.2 创建子项目-订单服务



## 声明项目依赖 和 项目构建插件

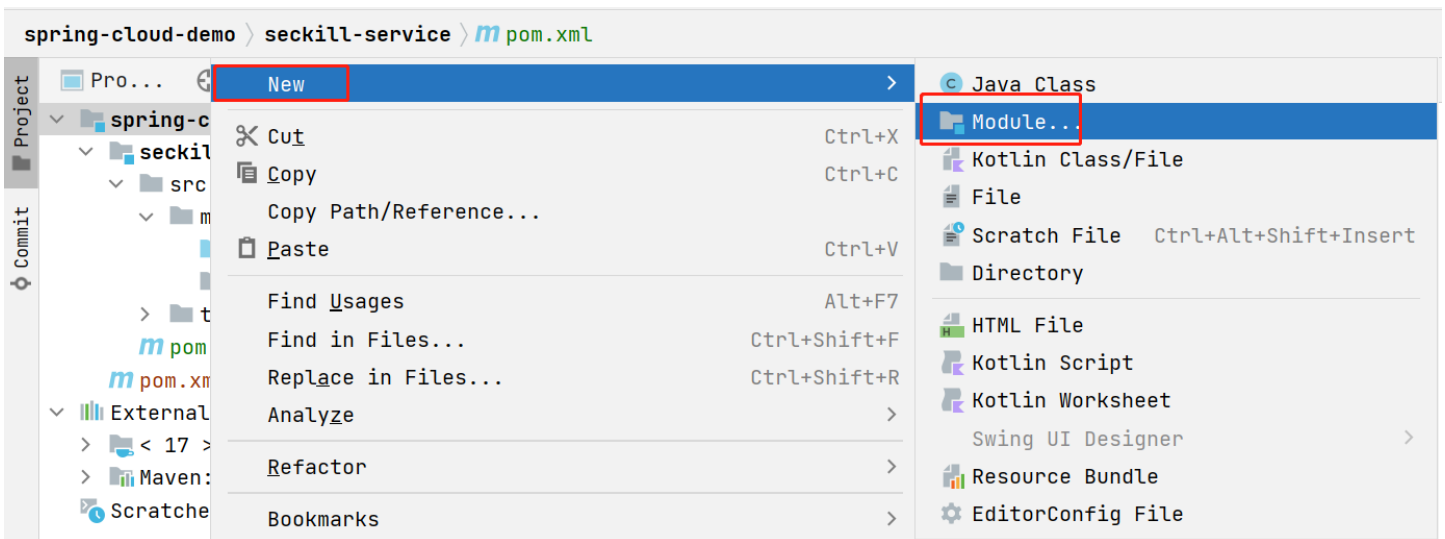
```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5   </dependency>
```

```

6     <dependency>
7         <groupId>com.mysql</groupId>
8         <artifactId>mysql-connector-j</artifactId>
9     </dependency>
10    <!--mybatis-->
11    <dependency>
12        <groupId>org.mybatis.spring.boot</groupId>
13        <artifactId>mybatis-spring-boot-starter</artifactId>
14    </dependency>
15 </dependencies>
16
17 <build>
18     <plugins>
19         <plugin>
20             <groupId>org.springframework.boot</groupId>
21             <artifactId>spring-boot-maven-plugin</artifactId>
22         </plugin>
23     </plugins>
24 </build>

```

### 4.1.3 创建子项目-商品服务



New Module

New Module

Generators

m

 Maven Archetype

JavaFX

Compose Multiplatform

IDE Plugin

Android

Spring Initializr

Name:

product-service

Location:

D:\Git\spring-cloud\代码\spring-cloud-demo

Module will be created in: D:\Git\spring-cloud\代码\spring-cl

Language:

Java

Groovy

HTML

+

Build system:

IntelliJ

Maven

Gradle

JDK:

Project SDK 17

Parent:

m

 spring-cloud-demo

☐ Add sample code

> Advanced Settings

?

Create

Cancel

## 声明项目依赖 和项目构建插件

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5   </dependency>
6   <dependency>
7     <groupId>com.mysql</groupId>
8     <artifactId>mysql-connector-j</artifactId>
9   </dependency>
10  <!--mybatis-->
11  <dependency>
12    <groupId>org.mybatis.spring.boot</groupId>
13    <artifactId>mybatis-spring-boot-starter</artifactId>
14  </dependency>
15 </dependencies>
16
17 <build>
18   <plugins>
```

```
19     <plugin>
20         <groupId>org.springframework.boot</groupId>
21         <artifactId>spring-boot-maven-plugin</artifactId>
22     </plugin>
23 </plugins>
24 </build>
```

## 4.2 完善订单服务

### 4.2.1 完善启动类, 配置文件

启动类

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3
4 @SpringBootApplication
5 public class OrderServiceApplication {
6     public static void main(String[] args) {
7         SpringApplication.run(OrderServiceApplication.class, args);
8     }
9 }
```

配置文件

```
1 server:
2     port: 8080
3 spring:
4     datasource:
5         url: jdbc:mysql://127.0.0.1:3306/cloud_order?
6             characterEncoding=utf8&useSSL=false
7         username: root
8         password: root
9         driver-class-name: com.mysql.cj.jdbc.Driver
10 mybatis:
11     configuration: # 配置打印 MyBatis日志
12         log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
13         map-underscore-to-camel-case: true #配置驼峰自动转换
```

### 4.2.2 业务代码

#### 1. 实体类

```
1 import lombok.Data;
2
3 import java.util.Date;
4
5 @Data
6 public class OrderInfo {
7     private Integer id;
8     private Integer userId;
9     private Integer productId;
10    private Integer num;
11    private Integer price;
12    private Integer deleteFlag;
13    private Date createTime;
14    private Date updateTime;
15 }
```

## 2. Controller

```
1 @RequestMapping("/order")
2 @RestController
3 public class OrderController {
4     @Autowired
5     private OrderService orderService;
6
7     @RequestMapping("/{orderId}")
8     public OrderInfo getOrderById(@PathVariable("orderId") Integer orderId){
9         return orderService.selectOrderById(orderId);
10    }
11 }
```

## 3. Service

```
1 import com.bite.order.mapper.OrderMapper;
2 import com.bite.order.model.OrderInfo;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class OrderService {
8     @Autowired
9     private OrderMapper orderMapper;
10 }
```

```

11     public OrderInfo selectOrderById(Integer orderId) {
12         OrderInfo orderInfo = orderMapper.selectOrderById(orderId);
13         return orderInfo;
14     }
15 }

```

## 4. Mapper

```

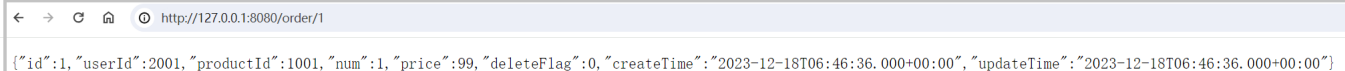
1 import com.bite.order.model.OrderInfo;
2 import org.apache.ibatis.annotations.Mapper;
3 import org.apache.ibatis.annotations.Select;
4
5 @Mapper
6 public interface OrderMapper {
7     @Select("select * from order_detail where id=#{orderId}")
8     OrderInfo selectOrderById(Integer orderId);
9 }

```

### 4.2.3 测试

访问url: <http://127.0.0.1:8080/order/1>

页面正常返回结果:



```

{"id":1,"userId":2001,"productId":1001,"num":1,"price":99,"deleteFlag":0,"createTime":"2023-12-18T06:46:36.000+00:00","updateTime":"2023-12-18T06:46:36.000+00:00"}

```

## 4.3 完善商品服务

### 4.3.1 完善启动类, 配置文件

启动类

```

1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3
4 @SpringBootApplication
5 public class ProductServiceApplication {
6     public static void main(String[] args) {
7         SpringApplication.run(ProductServiceApplication.class, args);
8     }
9 }

```



## 配置文件

```
1 server:
2   port: 9090
3 spring:
4   datasource:
5     url: jdbc:mysql://127.0.0.1:3306/cloud_product?
      characterEncoding=utf8&useSSL=false
6     username: root
7     password: root
8     driver-class-name: com.mysql.cj.jdbc.Driver
9 mybatis:
10  configuration: # 配置打印 MyBatis日志
11    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
12    map-underscore-to-camel-case: true #配置驼峰自动转换
```

后面需要多个服务一起启动, 所以设置为不同的端口号

## 4.3.2 业务代码

### 1. 实体类

```
1 import lombok.Data;
2
3 import java.util.Date;
4
5 @Data
6 public class ProductInfo {
7     private Integer id;
8     private String productName;
9     private Integer productPrice;
10    private Integer state;
11    private Date createTime;
12    private Date updateTime;
13 }
```

### 2. Controller

```
1 @RequestMapping("/product")
2 @RestController
3 public class ProductController {
4     @Autowired
```

```

5     private ProductService productService;
6
7     @RequestMapping("/{productId}")
8     public ProductInfo getProductById(@PathVariable("productId") Integer
productId){
9         System.out.println("收到请求,Id:"+productId);
10        return productService.selectProductById(productId);
11    }
12 }

```

### 3. Service

```

1 @Service
2 public class ProductService {
3     @Autowired
4     private ProductMapper productMapper;
5
6     public ProductInfo selectProductById(Integer id){
7         return productMapper.selectProductById(id);
8     }
9 }

```

### 4. Mapper

```

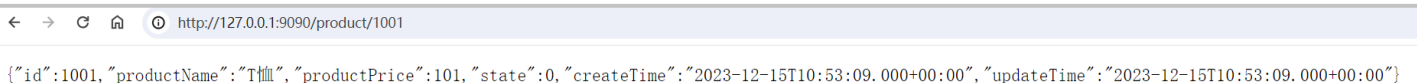
1 @Mapper
2 public interface ProductMapper {
3     @Select("select * from product_detail where id=#{id}")
4     ProductInfo selectProductById(Integer id);
5 }

```

#### 4.3.3 测试

访问url: <http://127.0.0.1:9090/product/1001>

页面正常返回结果:



```

{"id":1001,"productName":"T恤","productPrice":101,"state":0,"createTime":"2023-12-15T10:53:09.000+00:00","updateTime":"2023-12-15T10:53:09.000+00:00"}

```

### 4.4 远程调用

## 4.4.1 需求

根据订单查询订单信息时, 根据订单里产品ID, 获取产品的详细信息.



## 4.4.2 实现

**实现思路:** order-service服务向product-service服务发送一个http请求, 把得到的返回结果, 和订单结果融合在一起, 返回给调用方.

**实现方式:** 采用Spring 提供的RestTemplate

实现http请求的方式, 有很多, 可参考: <https://zhuanlan.zhihu.com/p/670101467>

### 1. 定义RestTemplate

```
1 @Configuration
2 public class BeanConfig {
3     @Bean
4     public RestTemplate restTemplate(){
5         return new RestTemplate();
6     }
7 }
```

### 2. 修改order-service中的 OrderService

```
1 @Service
2 public class OrderService {
3     @Autowired
```

```

4     private OrderMapper orderMapper;
5
6     @Autowired
7     private RestTemplate restTemplate;
8
9     public OrderInfo selectOrderById(Integer orderId) {
10         OrderInfo orderInfo = orderMapper.selectOrderById(orderId);
11         String url = "http://127.0.0.1:9090/product/" +
            orderInfo.getProductId();
12         ProductInfo productInfo = restTemplate.getForObject(url,
            ProductInfo.class);
13         orderInfo.setProductInfo(productInfo);
14         return orderInfo;
15     }
16 }

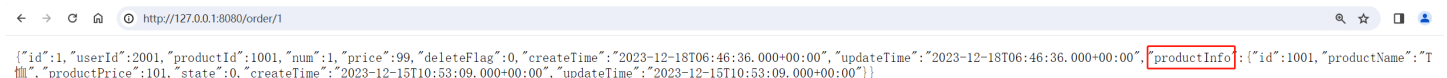
```

RestTemplate 详细使用可参考: [RestTemplate 最详解 - 程序员自由之路 - 博客园](#)

### 4.4.3 测试

访问url: <http://127.0.0.1:8080/order/1>

页面返回结果:



```

{"id":1,"userId":2001,"productId":1001,"num":1,"price":99,"deleteFlag":0,"createTime":"2023-12-18T06:46:36.000+00:00","updateTime":"2023-12-18T06:46:36.000+00:00","productInfo":{"id":1001,"productName":"T恤","productPrice":101,"state":0,"createTime":"2023-12-15T10:53:09.000+00:00","updateTime":"2023-12-15T10:53:09.000+00:00"}}

```

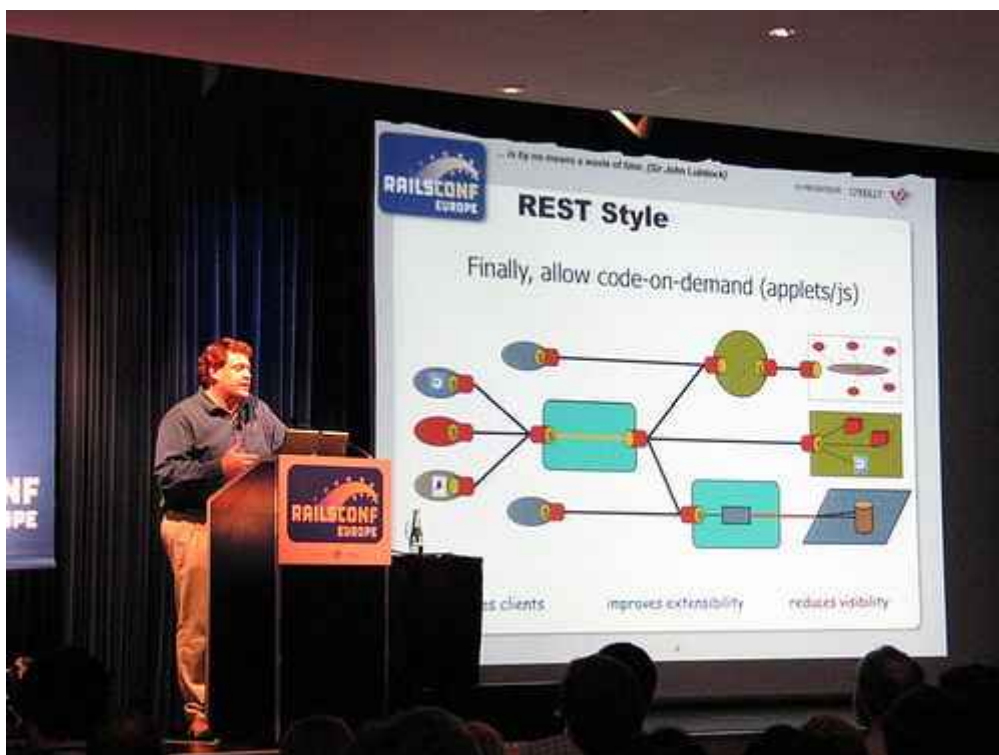
## 5. RestTemplate

RestTemplate 是从 Spring3.0 开始支持的一个 HTTP 请求工具, 它是一个同步的 REST API 客户端, 提供了常见的REST请求方案的模版.

### 什么是REST?

REST(**R**epresentational **S**tate **T**ransfer), 表现层资源状态转移.

REST是由HTTP的主要设计者Roy Fielding博士在2000年他的博士论文中提出来的一种软件架构风格.



这里面主要有三个概念:

1. 资源: 网络上的所有事物都可以抽象为资源, 每个资源都有一个唯一的资源标识符(URI)
2. 表现层: 资源的表现形式, 比如文本作为资源, 可以用txt格式表现, 也可以通过HTML, XML, JSON等格式来表现, 甚至以二进制的格式表现.
3. 状态转移: 访问URI, 也就是客户端和服务器的交互过程. 客户端用到的手段, 只能是HTTP协议. 这个过程中, 可能会涉及到数据状态的变化. 比如对数据的增删改查, 都是状态的转移.

**REST 是一种设计风格, 指资源在网络中以某种表现形式进行状态转移.**

简单来说: REST描述的是在网络中Client和Server的一种交互形式, REST本身不实用, 实用的是如何设计 RESTful API (REST风格的网络接口)

## 什么是RESTful?

REST 是一种设计风格, 并没有一个明确的标准. 满足这种设计风格的程序或接口我们称之为RESTful(从单词字面来看就是一个形容词). 所以RESTful API 就是满足REST架构风格的接口.

RESTful 风格大致有以下几个主要特征:

1. **资源:** 资源可以是一个图片, 音频, 视频或者JSON格式等网络上的一个实体, 除了一些二进制的资源外普通的文本资源更多以JSON为载体、面向用户的一组数据(通常从数据库中查询而得到)

2. **统一接口**: 对资源的操作. 比如获取, 创建, 修改和删除. 这些操作正好对应HTTP协议提供的GET、POST、PUT和DELETE方法. 换言之, 如果使用RESTful风格的接口, 从接口上你可能只能定位其资源, 但是无法知晓它具体进行了什么操作, 需要具体了解其发生了什么操作动作要从其HTTP请求方法类型上进行判断

比如同一个的URL:

GET /blog/{blogId}: 查询博客

DELETE /blog/{blogId}: 删除博客

这些内容都是通过HTTP协议来呈现的. 所以RESTful是基于HTTP协议的.

**RestTemplate** 是Spring提供, 封装HTTP调用, 并强制使用RESTful风格. 它会处理HTTP连接和关闭, 只需要使用者提供资源的地址和参数即可.

## RESTful实践

RESTful风格的API 固然很好很规范, 但大多数互联网公司并没有按照其规则来设计, 因为REST是一种风格, 而不是一种约束或规则, 过于理想的RESTful API 会付出太多的成本.

RESTful API 缺点:

1. 操作方式繁琐, RESTful API通常根据GET, POST, PUT, DELETE 来区分对资源的操作动作. 但是HTTP Method 并不可直接见到, 需要通过抓包等工具才能观察. 如果把动作放在URL上反而更加直观, 更利于团队的理解和交流.
2. 一些浏览器对GET, POST之外的请求支持不太友好, 需要额外处理.
3. 过分强调资源. 而实际业务需求可能比较复杂, 并不能单纯使用增删改查就能满足需求, 强行使用RESTful API会增加开发难度和成本.

所以, 在实际开发中, 如果业务需求和RESTful API不太匹配或者很麻烦时, 也可以不用RESTful API. 如果使用场景和REST风格比较匹配, 就可以采用RESTful API.

总之: 无论哪种风格的API, 都是为了方便团队开发, 协商以及管理, 不能墨守成规. 尽信书不如无书, 尽信规范不如无规范.

## 6. 项目存在问题

- 远程调用时, URL的IP和端口号是写死的(<http://127.0.0.1:9090/product/>), 如果更换IP, 需要修改代码
  - 调用方如何可以不依赖服务提供方的IP?
- 多机部署, 如何分摊压力?
- 远程调用时, URL非常容易写错, 而且复用性不高, 如何优雅的实现远程调用
- 所有的服务都可以调用该接口, 是否有风险?

• .....

除此之外, 微服务架构还面临很多问题, 接下来我们学习如何使用Spring Cloud 来解决这些问题.