

微服务



黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

微服务

岗位要求:

- 1、全日制本科及以上学历，6年以上工作经验;
- 2、Java技术基础扎实，精通面向对象设计。熟悉Java EE开发框架(Strues Spring ibatis 等)，熟练使用WAS等应用中间件;
- 3、熟悉分布式及微服务框架，有丰富的高并发、高可用分布式系统设计及应用经验。
- 3、熟悉分布式及微服务框架（对。SpringCloud、Dubbo等有过深入使用及研究），有相关项目实施经验者优先
- 4、擅长主流分布式中间件例如RocketMQ、Consul、Redis等，有高并发高可用分布式系统开发经验。
- 5、工作态度积极，好学、责任心强、思维缜密敏捷、良好的对外沟通和团队协作能力，有中大型项目开发经验优先;
- 6、有银行、互联网金融类业务系统或产品开发经验者优先

公司介绍

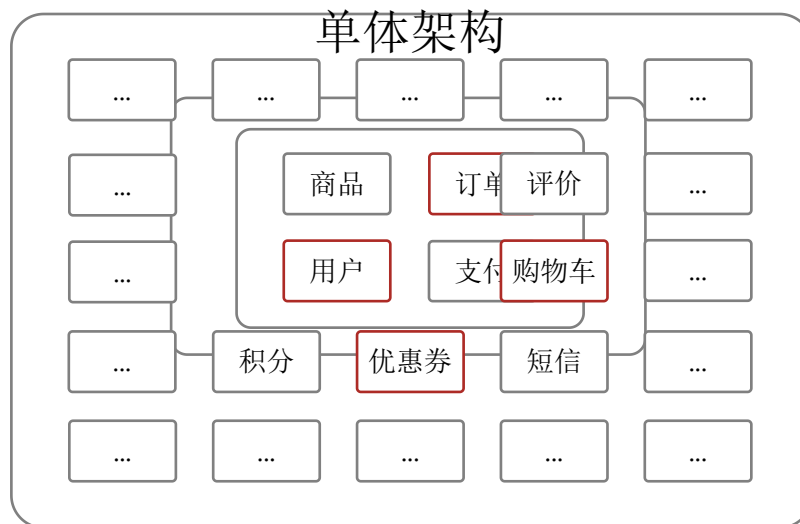
平安金服全称深圳平安综合金融服务有限公司线上个人客户综合金融经营服务平台，为集团子公司及机构、平安客户及用户、外部企业客户提供综合金融服务。

微服务



微服务是一种软件架构风格，它是以专注于单一职责的很多小型项目为基础，组合出复杂的大型应用。

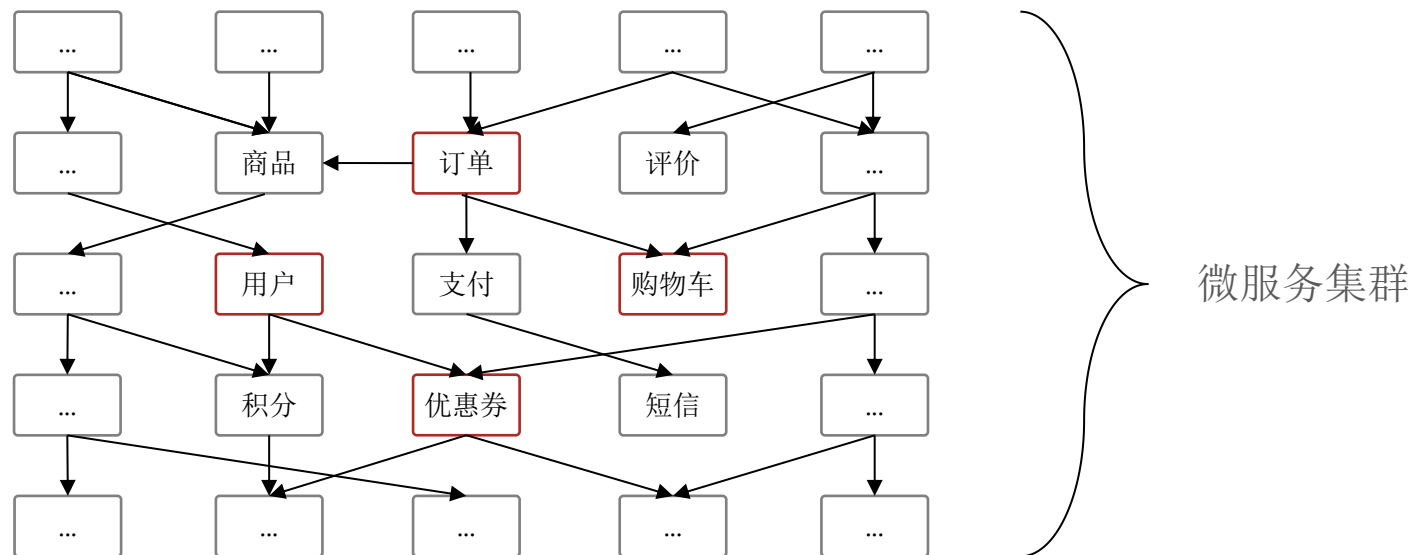
单体架构



微服务是一种软件架构风格，它是以专注于单一职责的很多小型项目为基础，组合出复杂的大型应用。

微服务

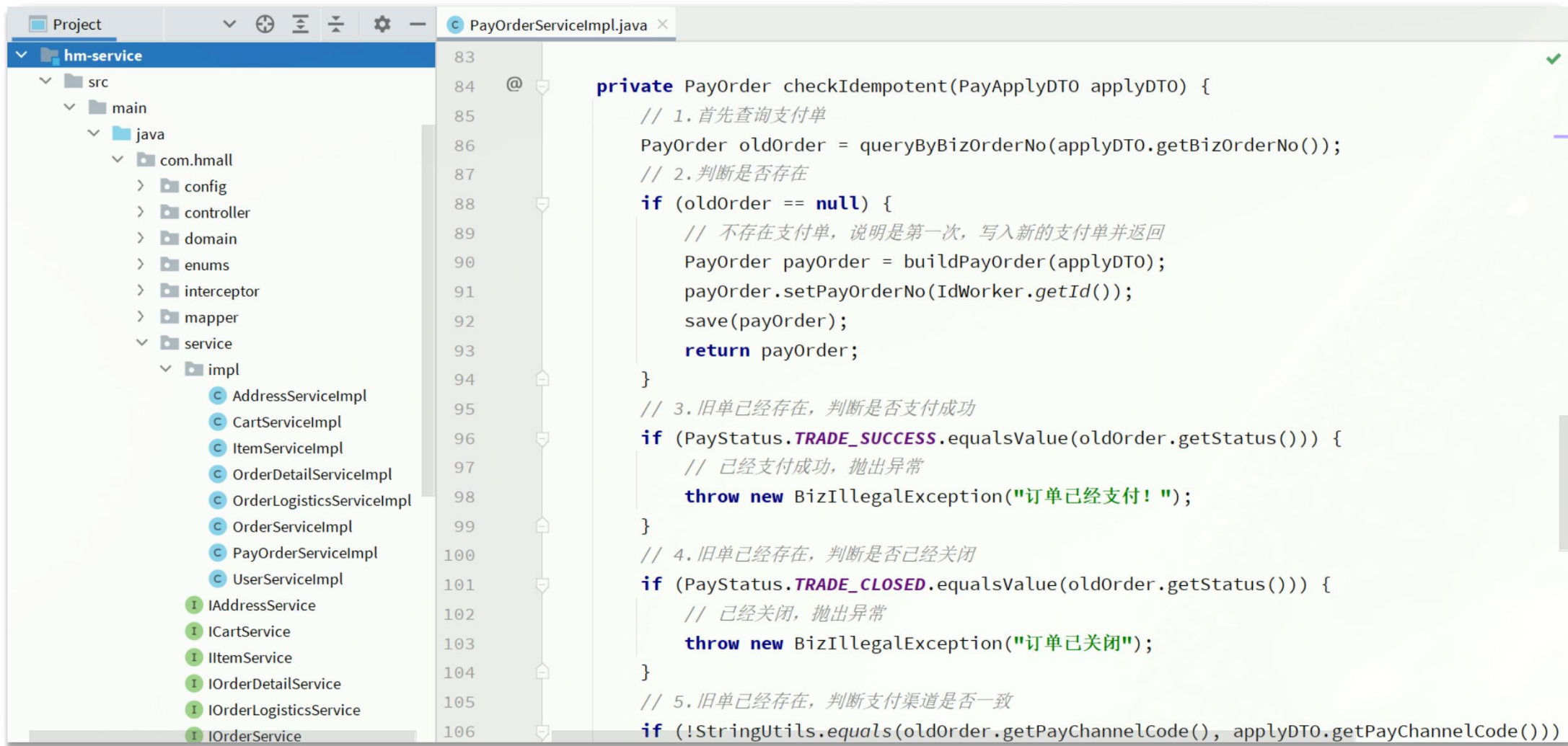
- 服务拆分
- 远程调用
- 服务治理
- 请求路由
- 身份认证
- 配置管理
- 服务保护
- 分布式事务
- 异步通信
- 消息可靠性
- 延迟消息
- 分布式搜索
- 倒排索引
- 数据聚合



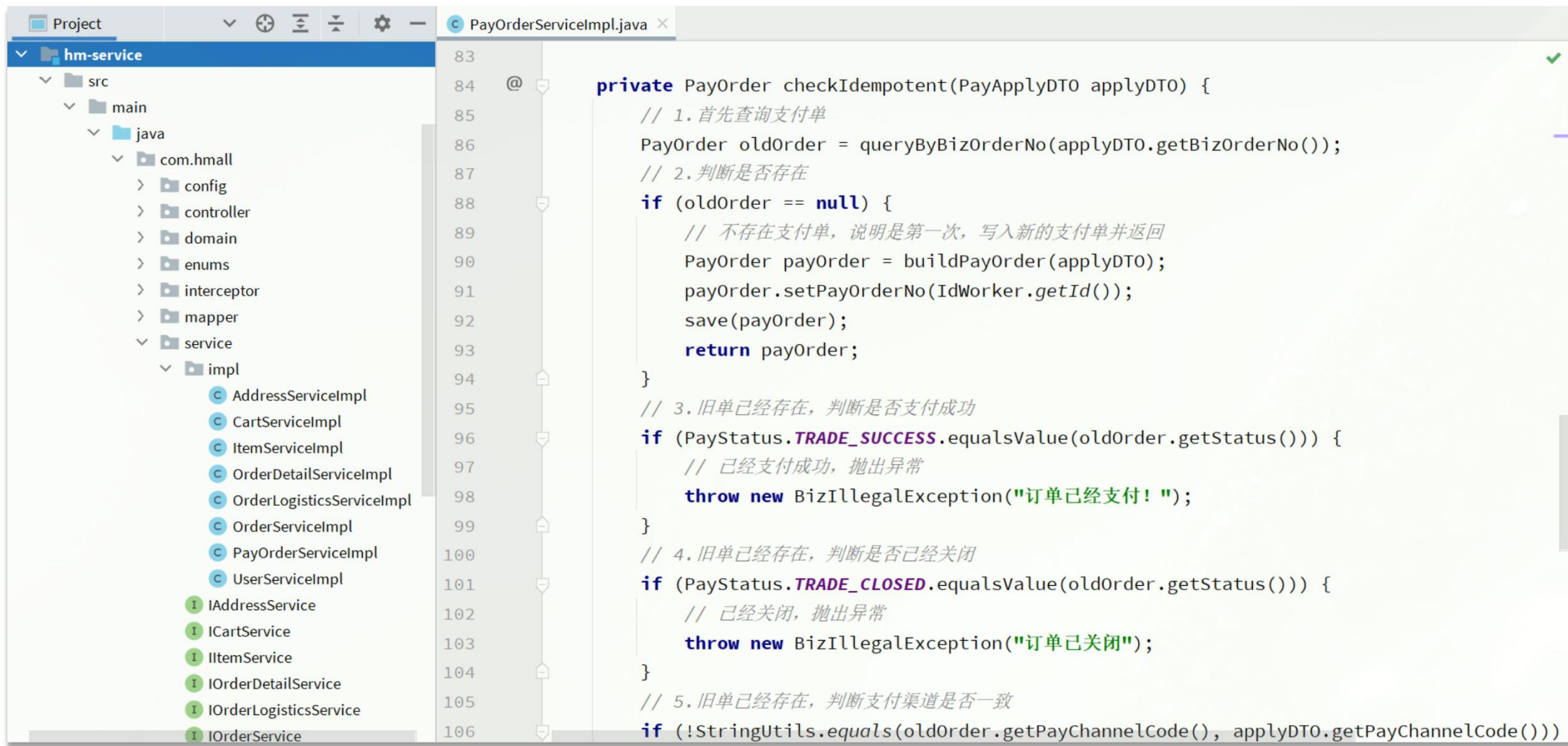
维基百科
自由的百科全书

微服务是一种软件架构风格，它是以专注于单一职责的很多小型项目为基础，组合出复杂的大型应用。

微服务



微服务



The screenshot shows an IDE with a project named 'hm-service'. The project structure is as follows:

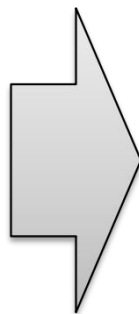
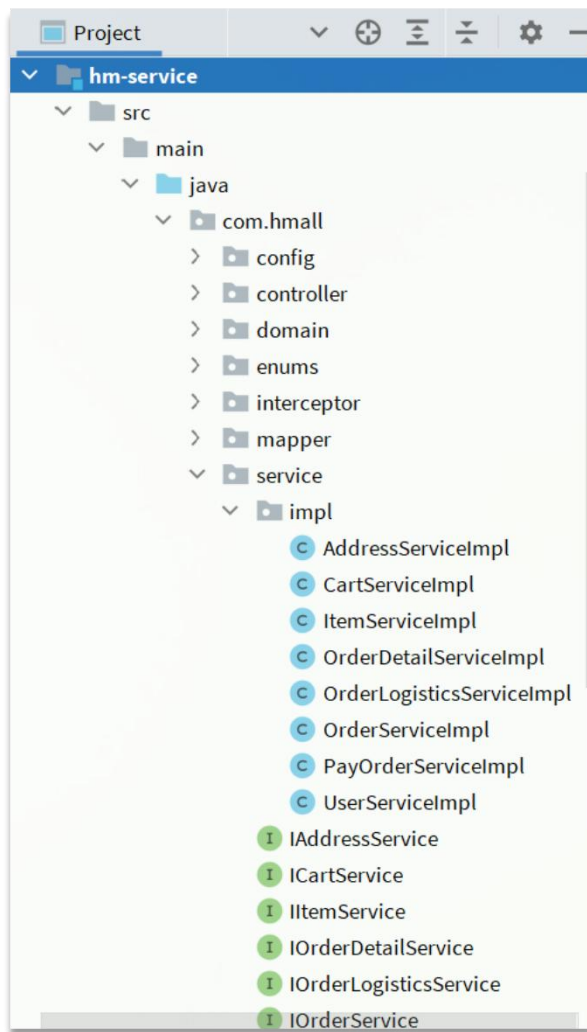
- src
 - main
 - java
 - com.hmall
 - config
 - controller
 - domain
 - enums
 - interceptor
 - mapper
 - service
 - impl
 - AddressServiceImpl
 - CartServiceImpl
 - ItemServiceImpl
 - OrderDetailServiceImpl
 - OrderLogisticsServiceImpl
 - OrderServiceImpl
 - PayOrderServiceImpl
 - UserServiceImpl
 - IAddressService
 - ICartService
 - IItemService
 - IOrderDetailService
 - IOrderLogisticsService
 - IOrderService

The code editor shows the file 'PayOrderServiceImpl.java' with the following code:

```
83
84 @
85
86 // 1. 首先查询支付单
PayOrder oldOrder = queryByBizOrderNo(applyDTO.getBizOrderNo());
87 // 2. 判断是否存在
88 if (oldOrder == null) {
89     // 不存在支付单，说明是第一次，写入新的支付单并返回
90     PayOrder payOrder = buildPayOrder(applyDTO);
91     payOrder.setPayOrderNo(IdWorker.getId());
92     save(payOrder);
93     return payOrder;
94 }
95 // 3. 旧单已经存在，判断是否支付成功
96 if (PayStatus.TRADE_SUCCESS.equalsValue(oldOrder.getStatus())) {
97     // 已经支付成功，抛出异常
98     throw new BizIllegalException("订单已经支付!");
99 }
100 // 4. 旧单已经存在，判断是否已经关闭
101 if (PayStatus.TRADE_CLOSED.equalsValue(oldOrder.getStatus())) {
102     // 已经关闭，抛出异常
103     throw new BizIllegalException("订单已关闭");
104 }
105 // 5. 旧单已经存在，判断支付渠道是否一致
106 if (!StringUtils.equals(oldOrder.getPayChannelCode(), applyDTO.getPayChannelCode()))
```

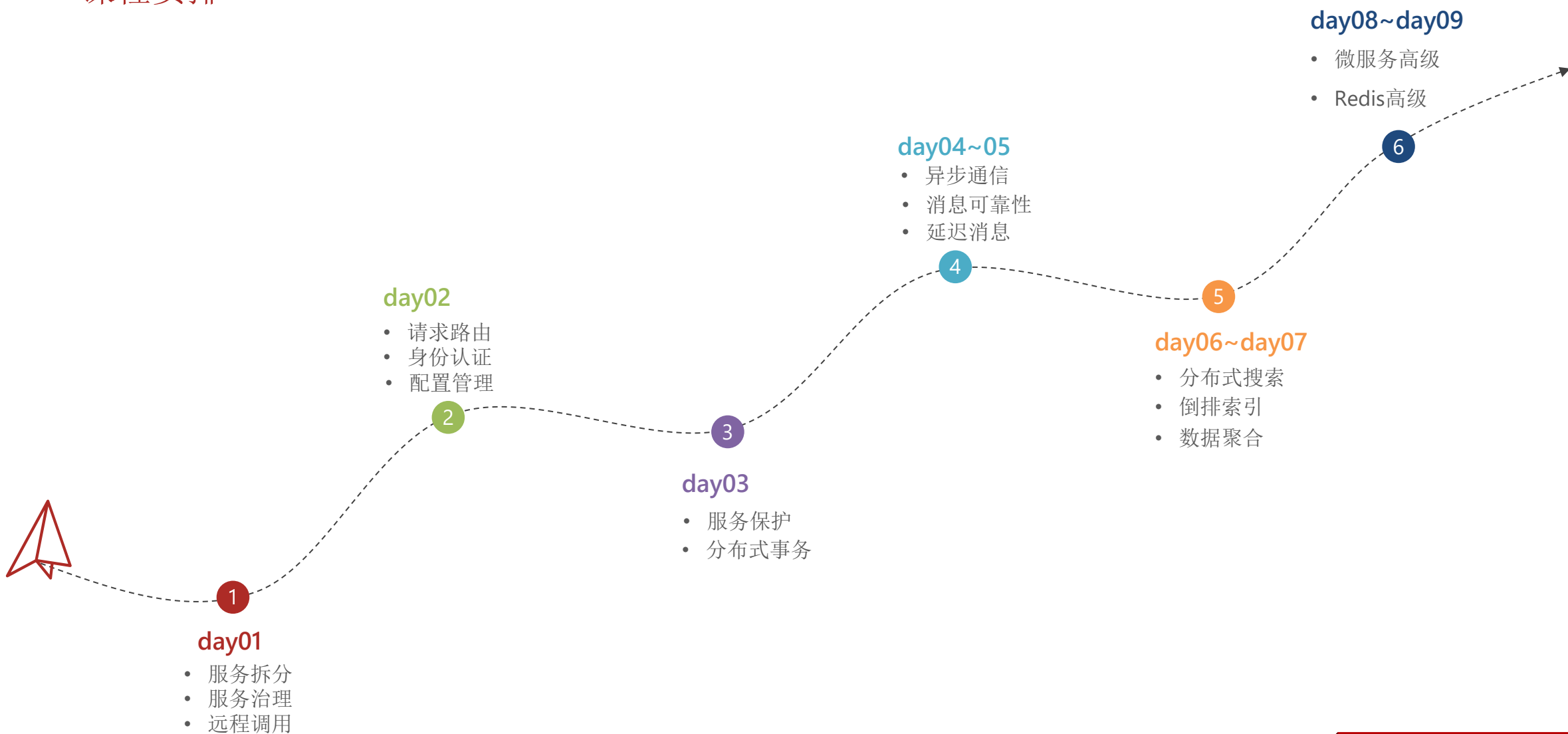
微服务

- 服务拆分
- 远程调用
- 服务治理
- 请求路由
- 身份认证
- 配置管理
- 服务保护
- 分布式事务
- 异步通信
- 消息可靠性
- 延迟消息
- 分布式搜索
- 倒排索引
- 数据聚合

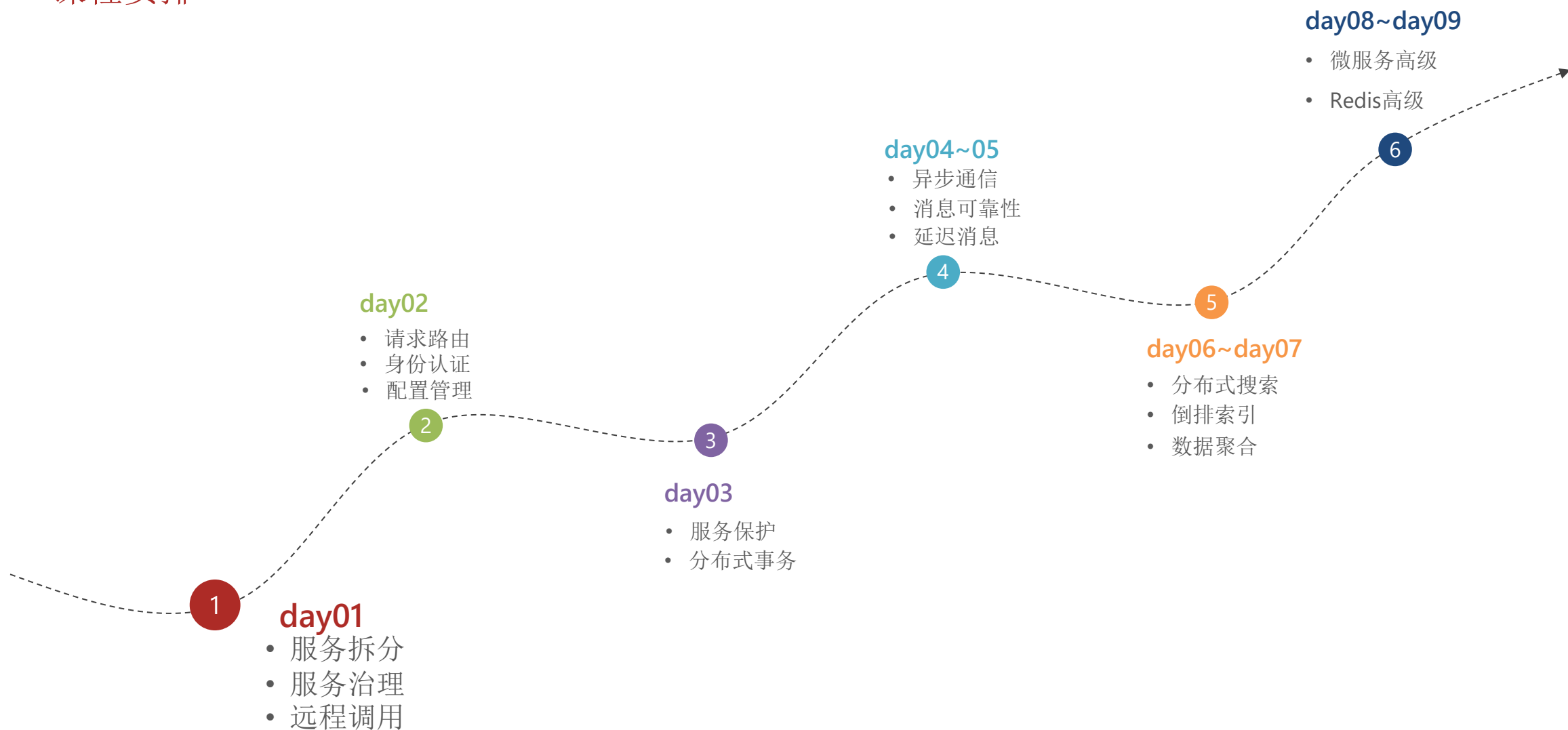


黑马程序员-研究院

课程安排



课程安排





认识微服务



目录

Contents

- ◆ 单体架构
- ◆ 微服务
- ◆ SpringCloud

单体架构

单体架构：将业务的所有功能集中在一个项目中开发，打成一个包部署。

优点：

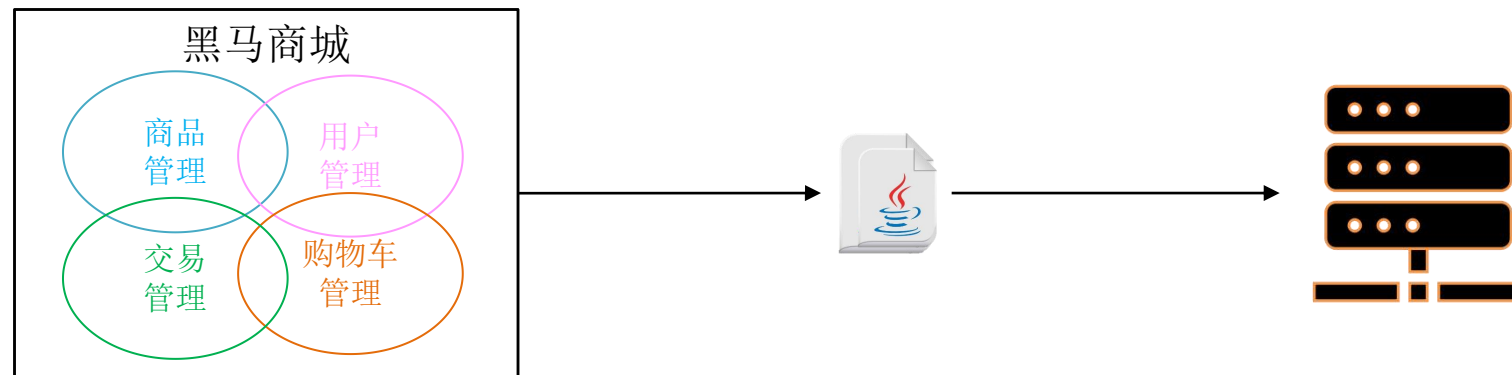
- 架构简单
- 部署成本低

缺点：

- 团队协作成本高
- 系统发布效率低
- 系统可用性差

总结：

单体架构适合开发功能相对简单，规模较小的项目。





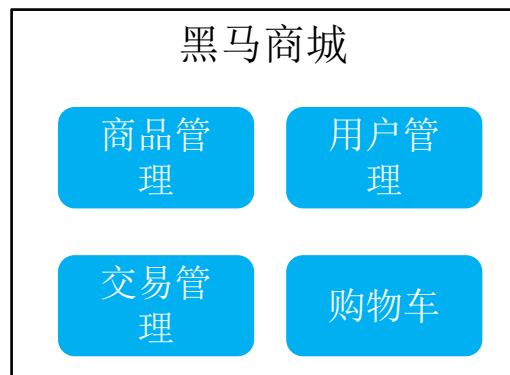
目录

Contents

- ◆ 单体架构
- ◆ 微服务
- ◆ SpringCloud

微服务

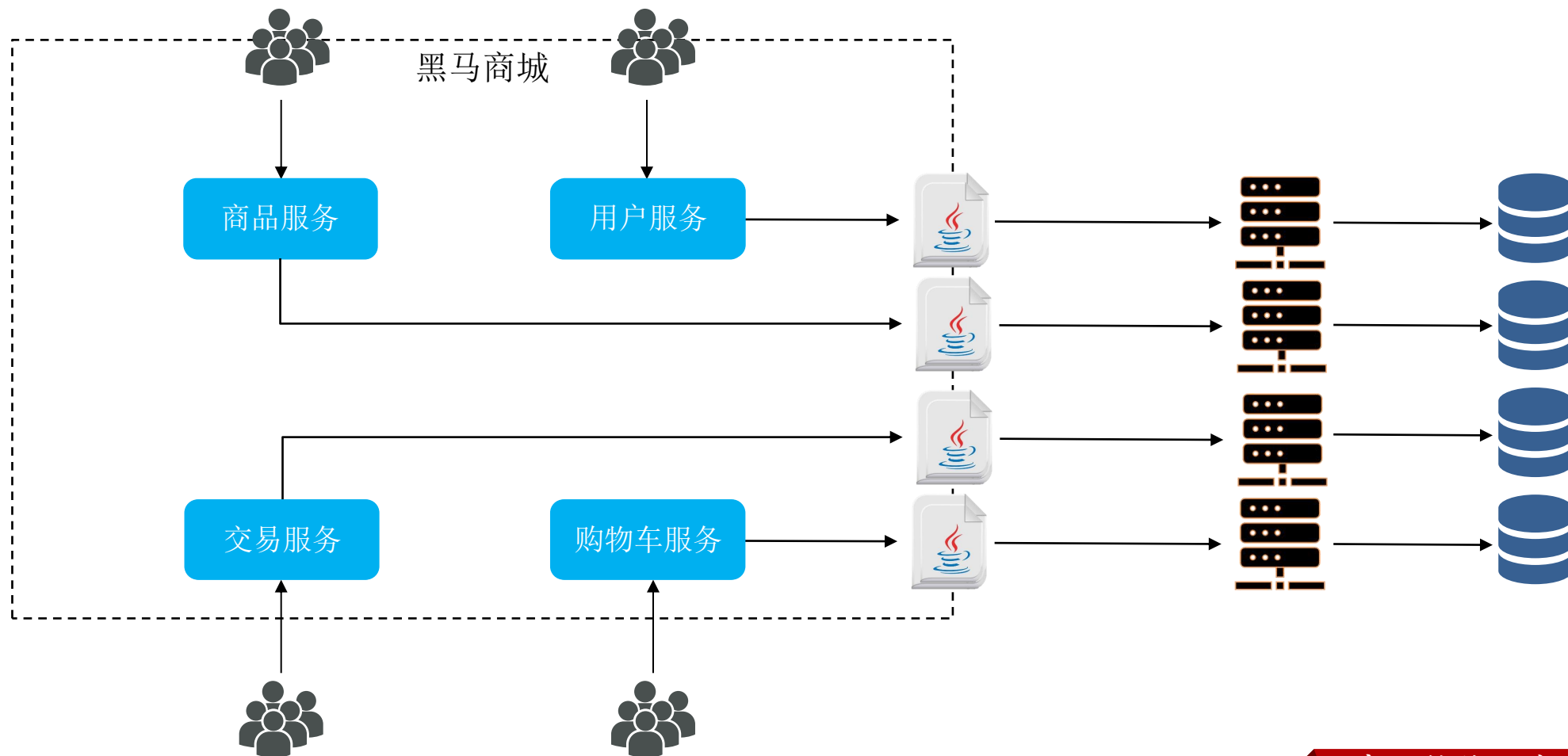
微服务架构，是服务化思想指导下的一套最佳实践架构方案。服务化，就是把单体架构中的功能模块拆分为多个独立项目。



微服务

微服务架构，是服务化思想指导下的一套最佳实践架构方案。服务化，就是把单体架构中的功能模块拆分为多个独立项目。

- 粒度小
- 团队自治
- 服务自治





目录

Contents

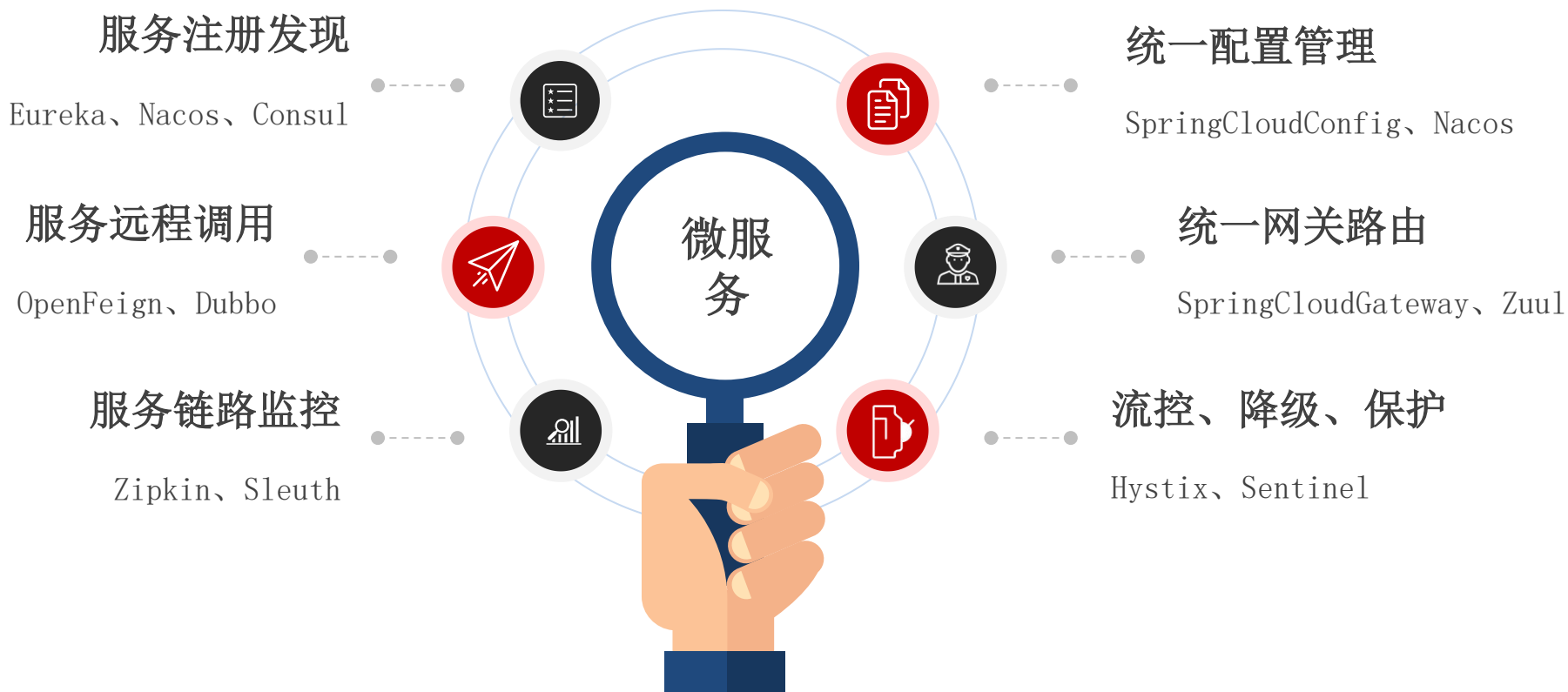
- ◆ 单体架构
- ◆ 微服务
- ◆ SpringCloud



SpringCloud

SpringCloud是目前国内使用最广泛的微服务框架。官网地址：<https://spring.io/projects/spring-cloud>。

SpringCloud集成了各种微服务功能组件，并基于SpringBoot实现了这些组件的自动装配，从而提供了良好的开箱即用体验：





SpringCloud

SpringCloud基于SpringBoot实现了微服务组件的自动装配，从而提供了良好的开箱即用体验。但对于*SpringBoot*的版本也有要求：

SpringCloud版本	SpringBoot版本
2022.0.x aka Kilburn	3.0.x
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



SpringCloud

SpringCloud基于SpringBoot实现了微服务组件的自动装配，从而提供了良好的开箱即用体验。但对于*SpringBoot*的版本也有要求：

SpringCloud版本	SpringBoot版本
2022.0.x aka Kilburn	3.0.x
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



微服务拆分

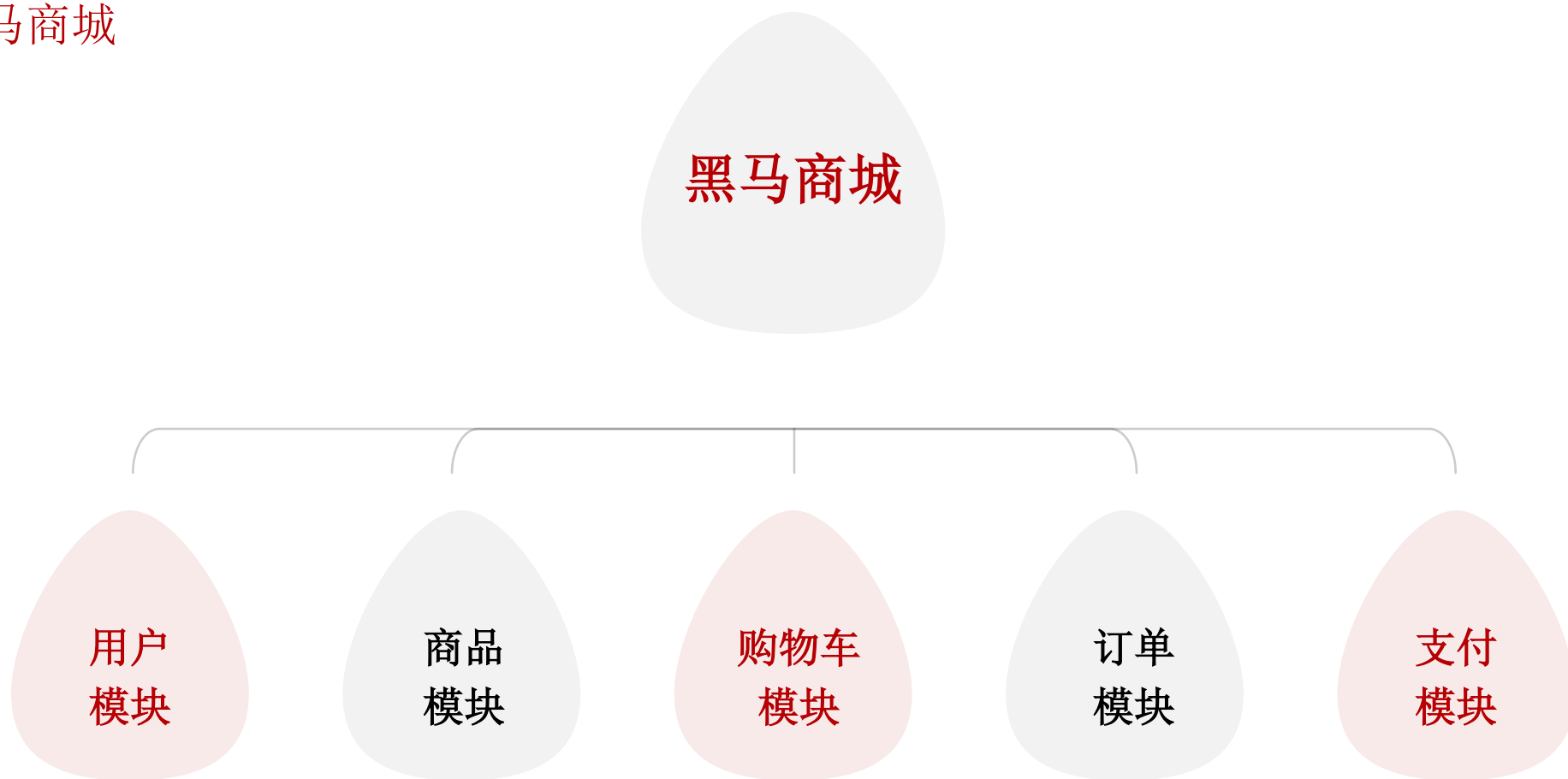


目录

Contents

- ◆ 熟悉黑马商城
- ◆ 服务拆分原则
- ◆ 拆分服务
- ◆ 服务调用

熟悉黑马商城





目录

Contents

- ◆ 熟悉黑马商城
- ◆ 服务拆分原则
- ◆ 拆分服务
- ◆ 服务调用

服务拆分原则



什么时候拆分?

- 创业型项目：先采用单体架构，快速开发，快速试错。随着规模扩大，逐渐拆分。
- 确定的大型项目：资金充足，目标明确，可以直接选择微服务架构，避免后续拆分的麻烦。

服务拆分原则



怎么拆分?

从拆分目标来说，要做到：

- **高内聚**：每个微服务的职责要尽量单一，包含的业务相互关联度高、完整度高。
- **低耦合**：每个微服务的功能要相对独立，尽量减少对其它微服务的依赖。

从拆分方式来说，一般包含两种方式：

- **纵向拆分**：按照业务模块来拆分
- **横向拆分**：抽取公共服务，提高复用性



目录

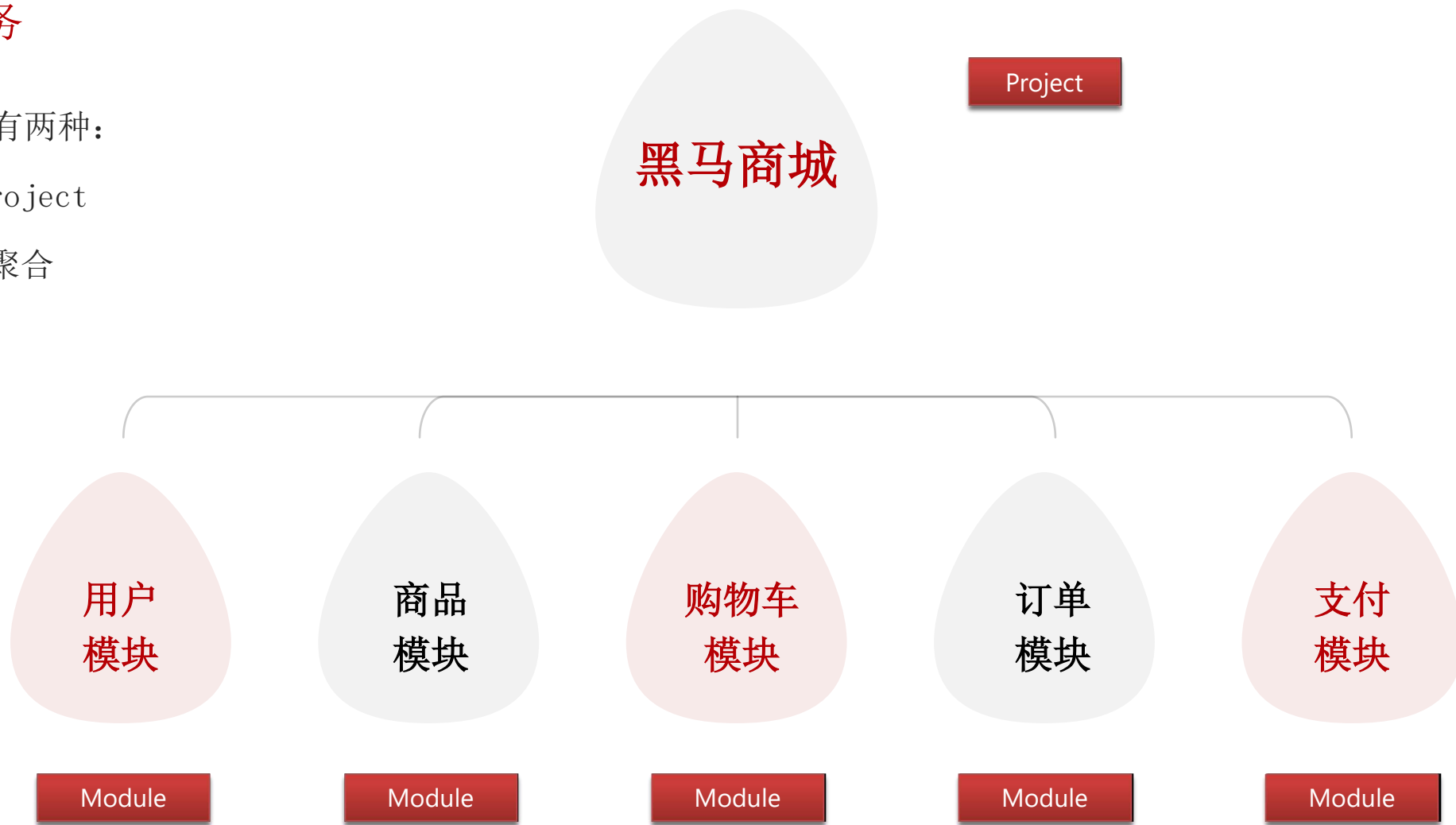
Contents

- ◆ 熟悉黑马商城
- ◆ 服务拆分原则
- ◆ 拆分服务
- ◆ 服务调用

拆分服务

工程结构有两种:

- 独立Project
- Maven聚合



案例

拆分服务

需求:

- 将hm-service中与商品管理相关功能拆分到一个微服务module中，命名为item-service
- 将hm-service中与购物车有关的功能拆分到一个微服务module中，命名为cart-service

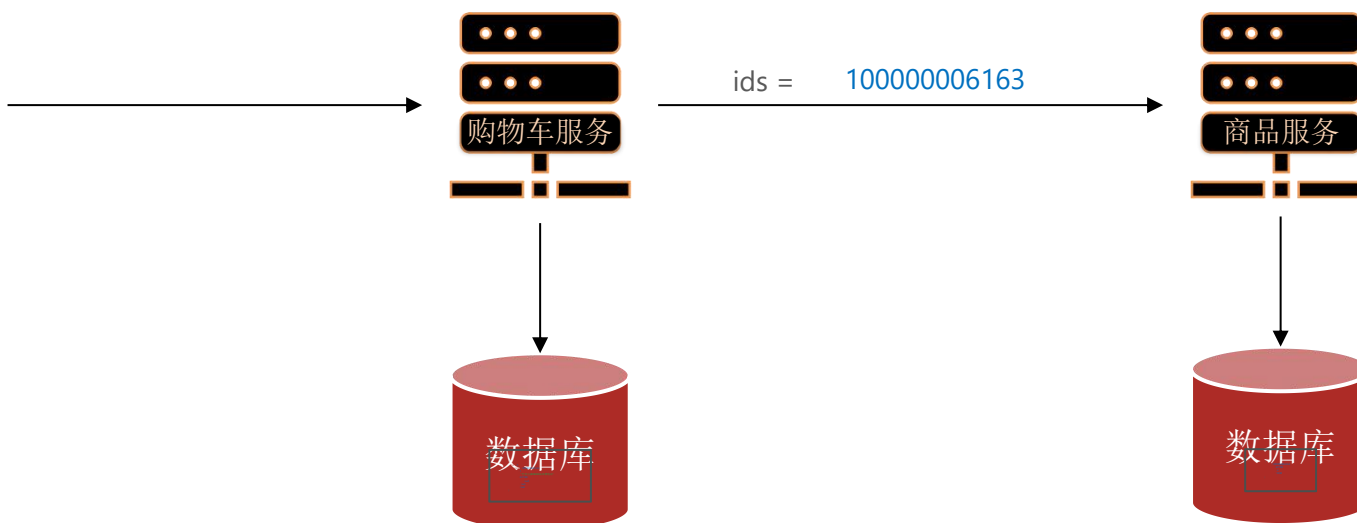
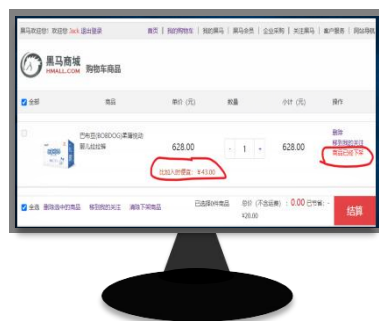


目录

Contents

- ◆ 熟悉黑马商城
- ◆ 服务拆分原则
- ◆ 拆分服务
- ◆ 远程调用

远程调用



```
{
  "id": 7,
  "itemId": 100000006163,
  "num": 1,
  "name": "巴布豆(BOBDOG)柔薄悦动",
  "price": 67100,
  "newPrice": null,
  "status": null,
  "stock": null
}
```

```
{
  "id": 100000006163,
  "price": 62800,
  "status": 2,
  "stock": 1000
}
```


远程调用

Spring给我们提供了一个RestTemplate工具，可以方便的实现Http请求的发送。使用步骤如下：

① 注入RestTemplate到Spring容器

```
@Bean
public RestTemplate restTemplate(){
    return new RestTemplate();
}
```

② 发起远程调用

```
public <T> ResponseEntity<T> exchange(
    String url, // 请求路径
    HttpMethod method, // 请求方式
    @Nullable HttpEntity<?> requestEntity, // 请求实体，可以为空
    Class<T> responseType, // 返回值类型
    Map<String, ?> uriVariables // 请求参数
) {
    // "http://localhost:8081/items?id={id}"
    // HttpMethod.GET
    // ItemDTO.class
    // Map.of("id", "1")
}
```



总结

什么时候拆分微服务?

- 初创型公司或项目尽量采用单体项目，快速试错。随着项目发展到一定规模再做拆分

如何拆分微服务?

- 目标：高内聚、低耦合。
- 方式：纵向拆分、横向拆分

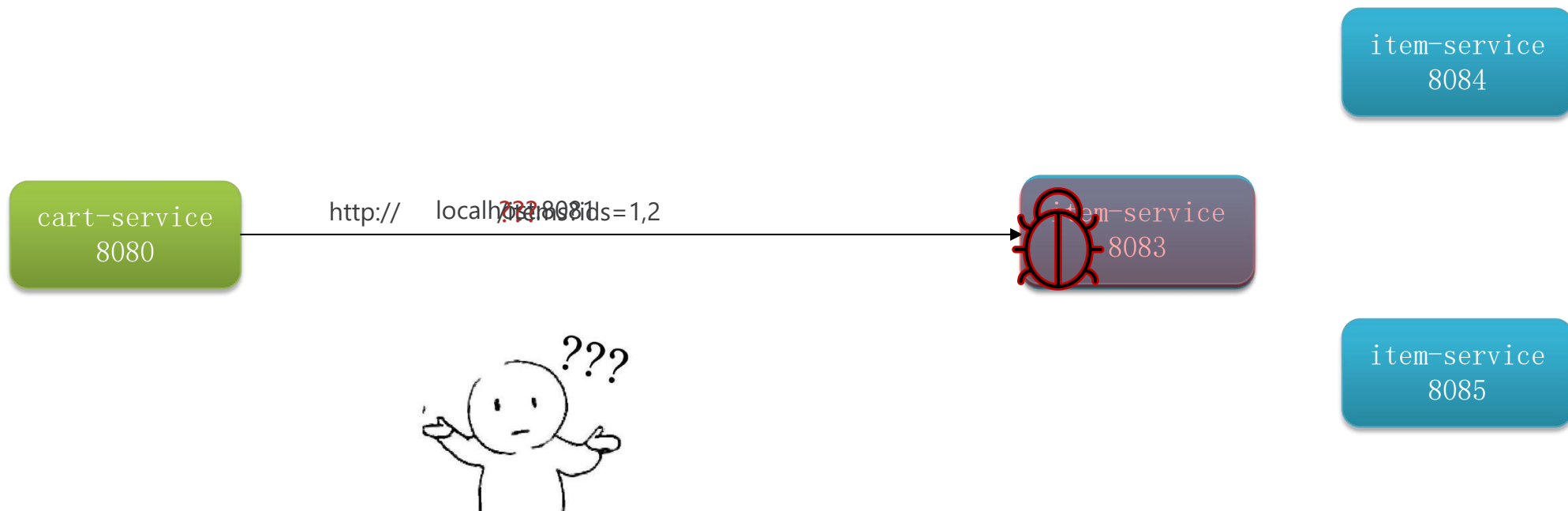
拆分后碰到的第一个问题是什么，如何解决?

- 拆分后，某些数据在不同服务，无法直接调用本地方法查询数据
- 利用RestTemplate发送Http请求，实现远程调用



服务治理

服务远程调用时存在的问题



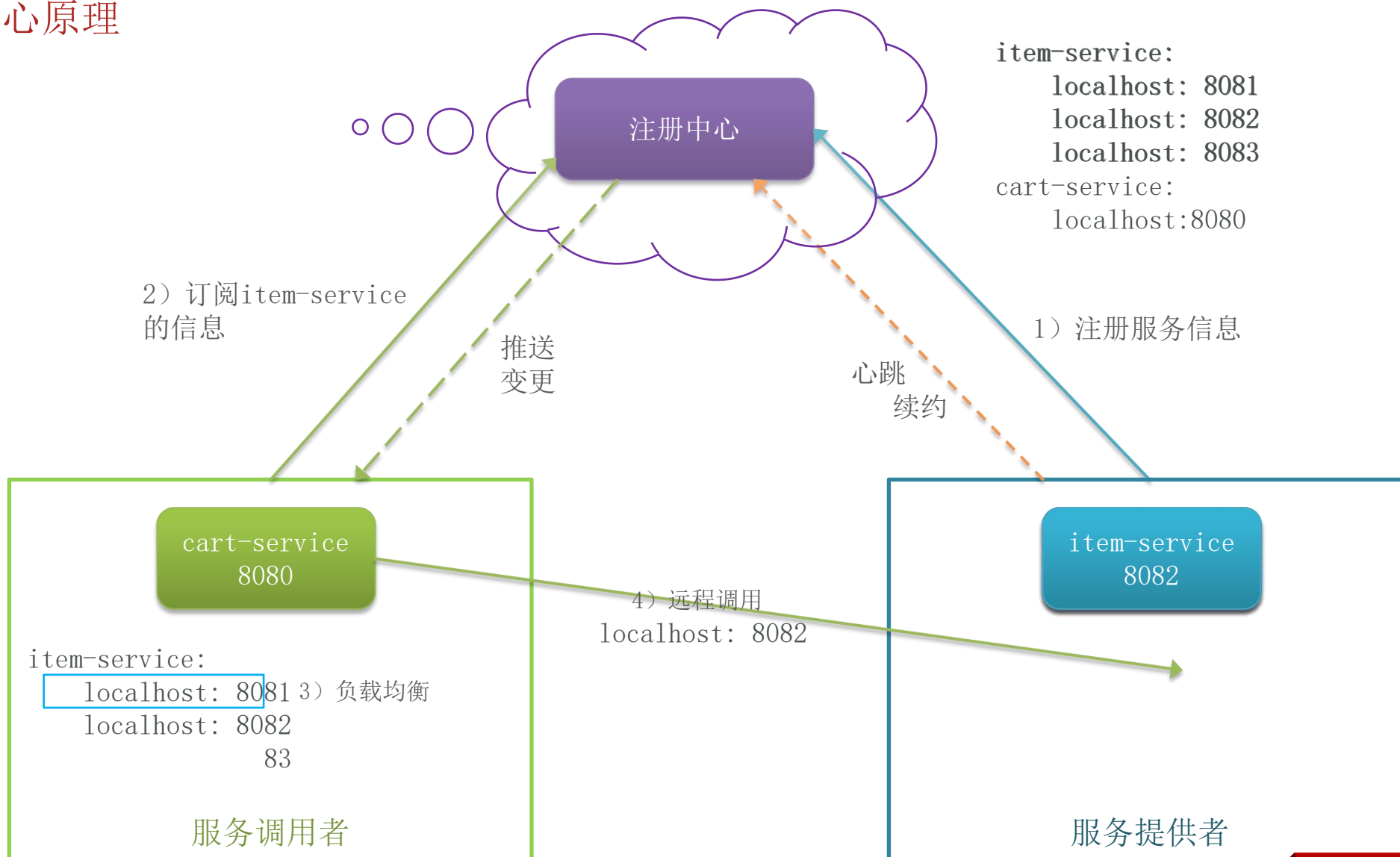


目录

Contents

- ◆ 注册中心原理
- ◆ Nacos注册中心
- ◆ 服务注册
- ◆ 服务发现

注册中心原理





总结

服务治理中的三个角色分别是什么？

- 服务提供者：暴露服务接口，供其它服务调用
- 服务消费者：调用其它服务提供的接口
- 注册中心：记录并监控微服务各实例状态，推送服务变更信息

消费者如何知道提供者的地址？

- 服务提供者会在启动时注册自己信息到注册中心，消费者可以从注册中心订阅和拉取服务信息

消费者如何得知服务状态变更？

- 服务提供者通过心跳机制向注册中心报告自己的健康状态，当心跳异常时注册中心会将异常服务剔除，并通知订阅了该服务的消费者

当提供者有多个实例时，消费者该选择哪一个？

- 消费者可以通过负载均衡算法，从多个实例中选择一个



目录

Contents

- ◆ 注册中心原理
- ◆ Nacos注册中心
- ◆ 服务注册
- ◆ 服务发现

Nacos注册中心

Nacos是目前国内企业中占比最多的注册中心组件。它是阿里巴巴的产品，目前已经加入SpringCloudAlibaba中。





目录

Contents

- ◆ 注册中心原理
- ◆ Nacos注册中心
- ◆ 服务注册
- ◆ 服务发现

服务注册

服务注册步骤如下：

① 引入nacos discovery依赖：

```
<!--nacos 服务注册发现-->
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

② 配置Nacos地址

```
spring:
  application:
    name: item-service # 服务名称
  cloud:
    nacos:
      server-addr: 192.168.150.101:8848 # nacos地址
```



目录

Contents

- ◆ 注册中心原理
- ◆ Nacos注册中心
- ◆ 服务注册
- ◆ 服务发现

服务发现

消费者需要连接nacos以拉取和订阅服务，因此服务发现的前两步与服务注册是一样，后面再加上服务调用即可：

① 引入nacos discovery依赖

...

② 配置nacos地址

...

③ 服务发现

```
private final DiscoveryClient discoveryClient;

private void handleCartItems(List<CartVO> vos) {
    // 1.根据服务名称，拉取服务的实例列表
    List<ServiceInstance> instances = discoveryClient.getInstances("item-service");
    // 2.负载均衡，挑选一个实例
    ServiceInstance instance = instances.get(RandomUtil.randomInt(instances.size()));
    // 3.获取实例的IP和端口
    URI uri = instance.getUri();
    // ... 略
}
```



OpenFeign



目录

Contents

- ◆ 快速入门
- ◆ 连接池
- ◆ 最佳实践
- ◆ 日志

快速入门

OpenFeign是一个声明式的http客户端，是SpringCloud在Eureka公司开源的Feign基础上改造而来。官方地址：

<https://github.com/OpenFeign/feign>

其作用就是基于SpringMVC的常见注解，帮我们优雅的实现http请求的发送。

Feign makes writing java http clients easier

[gitter](#)[join chat](#) **PASSED**[maven central](#)**11.1**

Feign is a Java to HTTP client binder inspired by [Retrofit](#), [JAXRS-2.0](#), and [WebSocket](#). Feign's first goal was reducing the complexity of binding [Denominator](#) uniformly to HTTP APIs regardless of [ReSTfulness](#).

快速入门

// 2.1.发现item-service服务的实例列表

```
List<ServiceInstance> instances = discoveryClient.getInstances("item-service");
```

// 2.2.负载均衡，挑选一个实例

```
ServiceInstance instance = instances.get(RandomUtil.randomInt(instances.size()));
```

// 2.3.发送请求，查询商品

```
ResponseEntity<List<ItemDTO>> response = restTemplate.exchange(
```

```
instance.getUri() + "/items?ids={ids}", // 请求路径
```

```
HttpMethod.GET, // 请求方式
```

```
null, // 请求实体
```

```
new ParameterizedTypeReference<List<ItemDTO>>() {}, // 返回值类型
```

```
Map.of("ids", CollUtil.join(itemIds, ",")) // 请求参数
```

```
);
```

服务名称

根据服务名称才能
拉取服务的实例列
表

请求方式和路径

通过请求方式和路径确
定要请求的资源地址，
访问到对应的接口

返回值类型

知道了返回值类型，才
能将接收到的JSON结果
处理为对应的Java实体

请求参数

请求参数，是调用接口
必不可少的要素

快速入门

OpenFeign已经被SpringCloud自动装配，实现起来非常简单：

- ① 引入依赖，包括OpenFeign和负载均衡组件SpringCloudLoadBalancer

```
<!--OpenFeign-->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<!--负载均衡-->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

- ② 通过@EnableFeignClients注解，启用OpenFeign功能

```
@EnableFeignClients
@SpringBootApplication
public class CartApplication { // ... 略 }
```

快速入门

[注册中心](#)

OpenFeign已经被SpringCloud自动装配，实现起来非常简单：

item-service

- localhost:8081
- localhost:8083

③ 编写FeignClient

```
@FeignClient(value = "item-service")
public interface ItemClient {

    @GetMapping("/items")
    List<ItemDTO> queryItemByIds(@RequestParam("ids") Collection<Long> ids);
}
```

Get http:// localhost:8081 /items ? ids = 1,2,3

④ 使用FeignClient，实现远程调用

```
List<ItemDTO> items = itemClient.queryItemByIds(List.of(1,2,3));
```

1,2,3

快速入门

OpenFeign已经被SpringCloud自动装配，实现起来非常简单：

- ① 引入依赖，包括OpenFeign和负载均衡组件SpringCloudLoadBalancer

```
<!--OpenFeign-->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<!--负载均衡-->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

- ② 通过@EnableFeignClients注解，启用OpenFeign功能

```
@EnableFeignClients
@SpringBootApplication
public class CartApplication { // ... 略 }
```

快速入门

OpenFeign已经被SpringCloud自动装配，实现起来非常简单：

③ 编写FeignClient

```
@FeignClient(value = "item-service")
public interface ItemClient {

    @GetMapping("/items")
    List<ItemDTO> queryItemByIds(@RequestParam("ids") Collection<Long> ids);
}
```

④ 使用FeignClient，实现远程调用

```
List<ItemDTO> items = itemClient.queryItemByIds(List.of(1,2,3));
```



目录

Contents

- ◆ 快速入门
- ◆ 连接池
- ◆ 最佳实践
- ◆ 日志

连接池

OpenFeign对Http请求做了优雅的伪装，不过其底层发起http请求，依赖于其它的框架。这些框架可以自己选择，包括以下三种：

- `HttpURLConnection`：默认实现，不支持连接池
- `Apache HttpClient`：支持连接池
- `OKHttp`：支持连接池

具体源码可以参考`FeignBlockingLoadBalancerClient`类中的`delegate`成员变量。

连接池

OpenFeign整合OKHttp的步骤如下：

① 引入依赖

```
<!--ok-http-->
<dependency>
  <groupId>io.github.openfeign</groupId>
  <artifactId>feign-okhttp</artifactId>
</dependency>
```

② 开启连接池功能

```
feign:
  okhttp:
    enabled: true # 开启OKHttp连接池支持
```

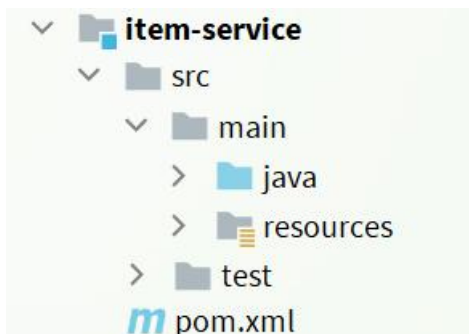



目录

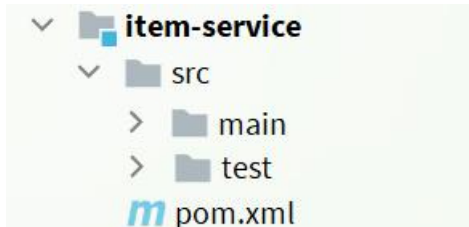
Contents

- ◆ 快速入门
- ◆ 连接池
- ◆ 最佳实践
- ◆ 日志

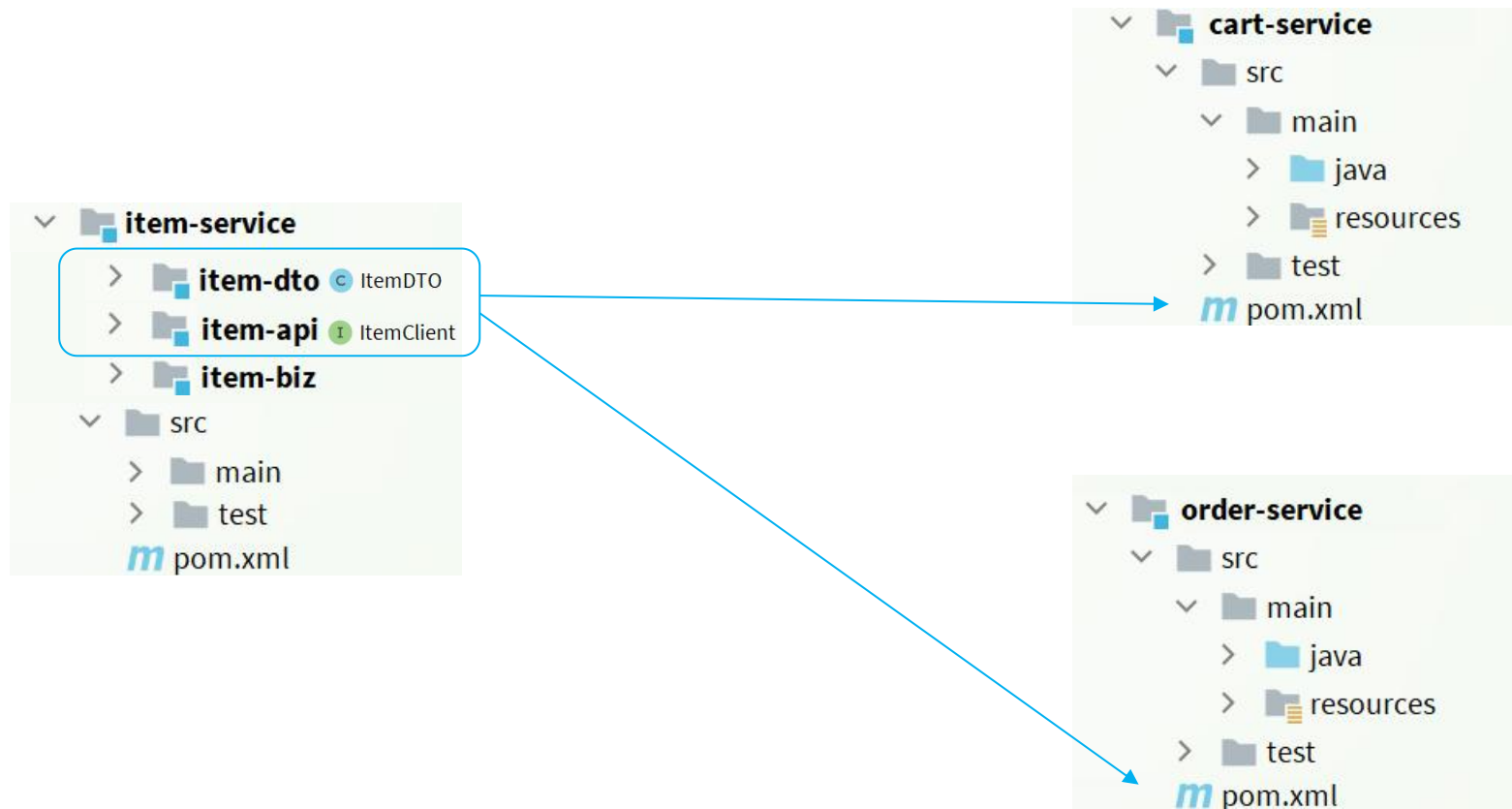
最佳实践



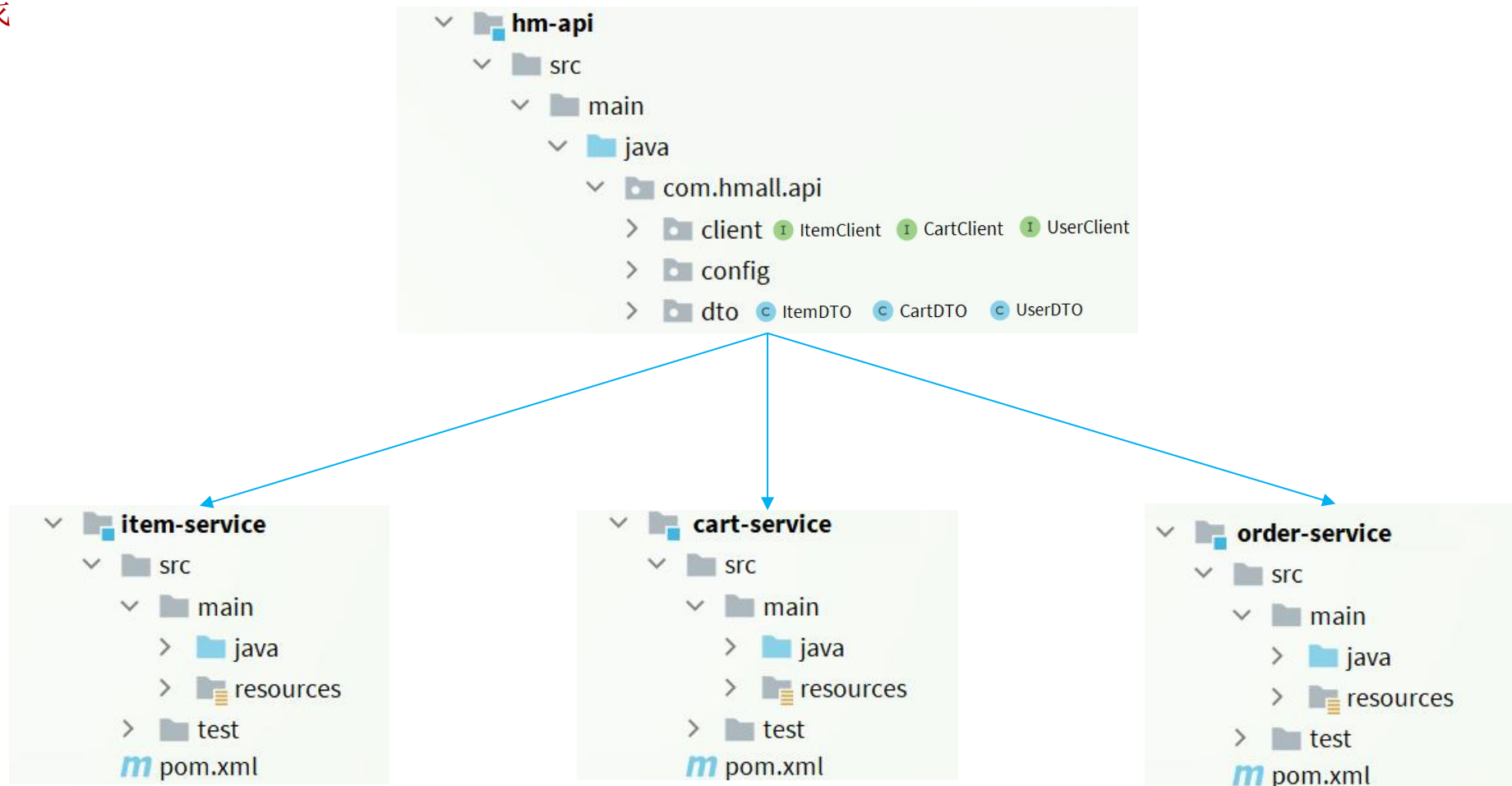
最佳实践



最佳实践



最佳实践



最佳实践

当定义的FeignClient不在SpringBootApplication的扫描包范围时，这些FeignClient无法使用。有两种方式解决：

方式一：指定FeignClient所在包

```
@EnableFeignClients(basePackages = "com.hmall.api.clients")
```

方式二：指定FeignClient字节码

```
@EnableFeignClients(clients = {UserClient.class})
```



目录

Contents

- ◆ 快速入门
- ◆ 连接池
- ◆ 最佳实践
- ◆ 日志

日志

OpenFeign只会在FeignClient所在包的日志级别为DEBUG时，才会输出日志。而且其日志级别有4级：

- **NONE**：不记录任何日志信息，这是默认值。
- **BASIC**：仅记录请求的方法，URL以及响应状态码和执行时间
- **HEADERS**：在BASIC的基础上，额外记录了请求和响应的头信息
- **FULL**：记录所有请求和响应的明细，包括头信息、请求体、元数据。

由于Feign默认的日志级别就是NONE，所以默认我们看不到请求日志。

日志

要自定义日志级别需要声明一个类型为Logger.Level的Bean，在其中定义日志级别：

```
public class DefaultFeignConfig {  
    @Bean  
    public Logger.Level feignLogLevel(){  
        return Logger.Level.FULL;  
    }  
}
```

但此时这个Bean并未生效，要想配置某个FeignClient的日志，可以在@FeignClient注解中声明：

```
@FeignClient(value = "item-service", configuration = DefaultFeignConfig.class)
```

如果想要全局配置，让所有FeignClient都按照这个日志配置，则需要在@EnableFeignClients注解中声明：

```
@EnableFeignClients(defaultConfiguration = DefaultFeignConfig.class)
```



总结

如何利用OpenFeign实现远程调用？

- 引入OpenFeign和SpringCloudLoadBalancer依赖
- 利用@EnableFeignClients注解开启OpenFeign功能
- 编写FeignClient

如何配置OpenFeign的连接池？

- 引入http客户端依赖，例如OKHttp、HttpClient
- 配置yaml文件，打开OpenFeign连接池开关

OpenFeign使用的最佳实践方式是什么？

- 由服务提供者编写独立module，将FeignClient及DTO抽取

如何配置OpenFeign输出日志的级别？

- 声明类型为Logger.Level的Bean
- 在@FeignClient或@EnableFeignClients注解上使用