

主讲老师： Fox

学习本课程前提：

- 1.熟悉常用微服务组件（nacos, feign, gateway, skywalking）的使用，没接触过微服务的同学可以先补一下[微服务专题的基础使用课程](#)
- 2.电商项目环境运行文档：<https://docs.qq.com/doc/DUXVuckZsQW5uUkdm>
3. 扩展：想要从0开始学习项目开发的同学可以学习[图灵商城项目实战-基础版](#)

本课程核心内容：

1. 微服务拆分时机和策略
2. 微服务技术栈选型以及常用组件的接入
3. 扩展知识：微服务全链路灰度解决方案实战

- 1 文档：[1 电商项目微服务架构拆分实战.note](#)
- 2 链接：<http://note.youdao.com/noteshare?id=500a8cecf089f175cb3310270e4beb21&sub=D9071B4F5C0641A5829F76BEAF2C2EDE>

1. 微服务架构拆分

1.1 微服务介绍

1.2 微服务拆分时机

1.3 微服务拆分的一些通用原则

1.4 微服务拆分策略

功能维度拆分策略

非功能维度拆分策略

1.5 拆分注意的风险

2. 电商项目微服务拆分

2.1 电商项目微服务架构图

2.2 Spring Cloud&Spring Cloud Alibaba技术栈选型

2.3 服务模块的拆分

2.4 数据库的拆分

2.5 接入Nacos注册中心

2.6 接入Nacos配置中心

2.7 基于openfeign实现微服务间的调用

2.8 网关服务搭建

2.9 接入Skywalking

2.10 整合ELK收集微服务链路日志

方案一：使用logstash日志插件

方案二：标准的ELK收集方案：通过FileBeat收集本地日志

测试：在kibana中根据trace_id搜索对应的系统日志

3. 微服务全链路灰度解决方案

3.1 灰度发布

3.2 微服务全链路灰度

3.3 全链路灰度设计思路

标签路由

节点打标

流量染色

分布式链路追踪

总结

3.4 基于Discovery实现全链路灰度

Discovery版本选择

网关服务接入Discovery

应用微服务接入Discovery

全链路版本权重灰度发布

全链路版本条件权重灰度发布

3.5 MSE 微服务治理全链路灰度

1. 微服务架构拆分

1.1 微服务介绍

英文:<https://martinfowler.com/articles/microservices.html>

中文:<http://blog.cuicc.com/blog/2015/07/22/microservices>

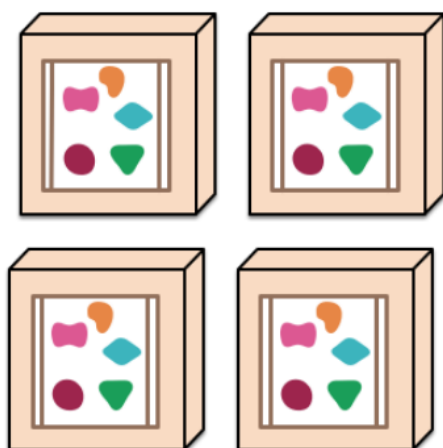
一个单体应用程序把它所有的功能放在一个单一进程中...



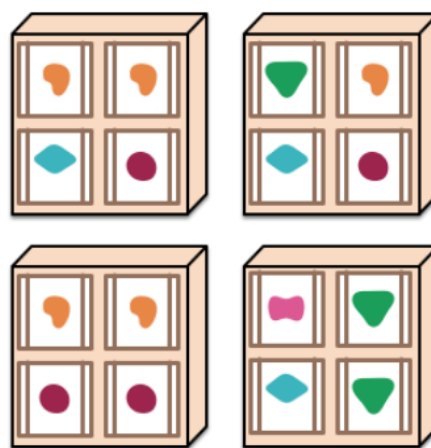
一个微服务架构把每个功能元素放进一个独立的服务中...



...并且通过在多个服务器上复制这个单体进行扩展



...并且通过跨服务器分发这些服务进行扩展，只在需要时才复制。



因素	单体架构	微服务架构	说明
交付速度	较慢	较快	服务拆分后，各个服务可以独立并行开发、测试、部署，交付效率提升，产品的更新速度会更快，用户体验更好。代码规模越大，微服务的优势越明显
故障隔离范围	线程级	进程级	服务独立运行，通过进程的方式隔离，使故障范围得到有效控制。架构变得更简单可靠。根据业务的重要程度划分服务，把核心的业务划分为独立的服务。这样

			可以从数据库到服务，保持有效的故障隔离，进而保持稳定
整体可用性	较低	更高	微服务架构由于故障范围得到有效隔离，整体可用性更高，降低一点故障对整体的影响
架构持续演进	困难	简单	由于微服务的粒度更小，架构演进的影响面就更小。不存在大规模重构导致的各种问题。微服务架构对架构演进更友好
沟通效率	低	高	业界普遍认为团队规模越大，沟通效率越低，微服务架构按业务构建全功能团队，把权利下放，不会出现决策瓶颈点，降低沟通规模，提升沟通效率
技术栈选择	受限	灵活	如果某个业务需要独立的技术栈，可以通过服务划分，接口集成的方式实现。例如搜索，技术栈、专业细分领域都不相同，通常采用独立的服务实现
可扩展性	受限	灵活	微服务架构可以根据服务对资源的要求以服务为粒度扩展，符合AKF扩展立方体中的Y轴扩展，而单体架构只能整体扩展，只能做到AKF扩展立方体中的X轴扩展
可重用性	低	高	微服务架构可以实现以服务为粒度通过接口共享重用
实现业务复杂性分解难度	困难	容易	微服务架构通过将业务分解为更多的服务，业务边界更清晰，更容易把一个复杂的问题分解为简单的小问题
产品创新复杂度	困难	容易	微服务架构以服务为粒度独立演进，团队有更多的自主决策权，更多的试错机会，更利于创新
一致性实现成本	低	高	服务划分后，如果服务A同时调用服务B和服务C，如何保证同时成功或失败？单体架构下的单库事务变成了分布式事务问题
时延	低	高	服务划分后，调用次数增加，响应时间必然升高，吞吐量降低，如何弥补
			吞吐量的下降意味着要增加更多

资源成本	低	高	的资源，对于交付型项目，特别是小规模部署的场景下，是比较致命的
关联查询复杂度	简单	复杂	微服务架构的一个非常明显的特征就是一个服务所拥有的数据只能通过这个服务的API来访问。通过这种方式来解耦，这样就会带来查询问题。以前通过join就可以满足要求，现在如果需要跨多个服务集成查询就会非常麻烦
远程调用	不涉及	涉及	微服务存在更多的远程调用，需要额外考虑序列化、通信协议、数据压缩、服务间的负载均衡、容错等问题
服务治理	不涉及	涉及	由于服务数量变多，微服务架构需要额外考虑服务的注册发现、依赖关系、治理等问题
对开发人员的要求	低	高	微服务架构更复杂，开发人员端到端负责，既要考虑接口定义，又要考虑数据库设计，对开发人员的水平要求更高
对工具的依赖	较低	较高	微服务架构中服务的数量较多，使用工具的效果更明显，依赖程度更高
运维复杂度	低	高	微服务架构中服务的数量较多，对服务的监控、健康检查要求更高，整体运维复杂度更高

思考：

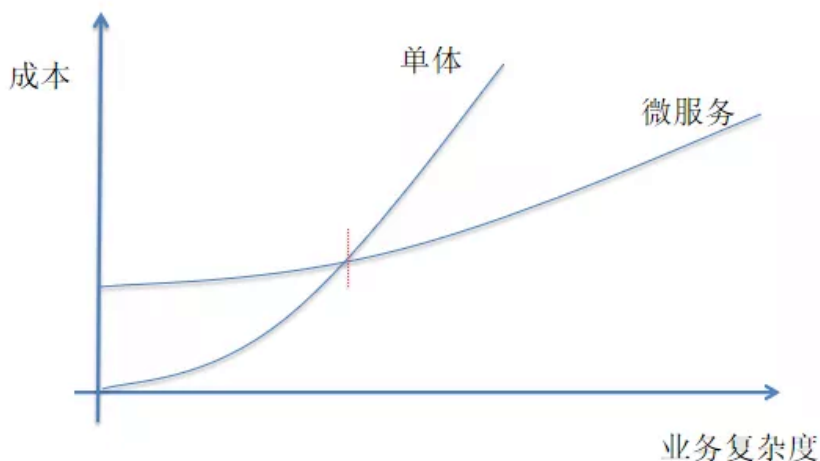
微服务拆分的时机是什么？ 如何决定微服务的拆分粒度？

1.2 微服务拆分时机

如下场景是否需要微服务拆分？

- 代码维护困难，几百人同时开发一个模块，提交代码频繁出现大量冲突；
- 模块耦合严重，互相依赖，小功能修改也必须累计到大版本才能上线，上线还需要总监协调各个团队开会确定；

- 横向扩展流程复杂，主要业务和次要业务耦合。例如下单和支付业务需要扩容，而注册业务不需要扩容



微服务不仅仅是技术的升级，更是开发方式、组织架构、开发观念的转变。

何时进行微服务的拆分：

- **业务规模**：业务模式得到市场的验证，需要进一步加快脚步快速占领市场，这时业务的规模变得越来越大，按产品生命周期来划分（导入期、成长期、成熟期、衰退期）这时一般在成长期阶段。如果是导入期，尽量采用单体架构。
- **团队规模**：一般是团队达到百人的时候，主要还是要结合业务复杂度
- **技术储备**：领域驱动设计、注册中心、配置中心、日志系统、持续交付、监控系统、分布式定时任务、CAP 理论、分布式调用链、API 网关等等。
- **人才储备**：精通微服务落地经验的架构师及相应开发人员。
- **研发效率**：研发效率大幅下降。

1.3 微服务拆分的一些通用原则

单一服务内部功能高内聚低耦合：每个服务只完成自己职责内的任务，对于不是自己职责的功能交给其它服务来完成

闭包原则 (CCP)：微服务的闭包原则就是当我们需要改变一个微服务的时候，所有依赖都在这个微服务的组件内，不需要修改其他微服务

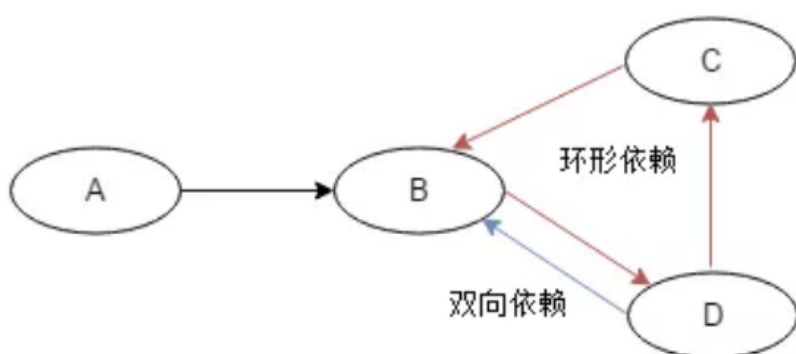
服务自治、接口隔离原则：尽量消除对其他服务的强依赖，这样可以降低沟通成本，提升服务稳定性。服务通过标准的接口隔离，隐藏内部实现细节。这使得服务可以独立开发、测试、部署、运行，以服务为单位持续交付。

持续演进原则：在服务拆分的初期，你其实很难确定服务究竟要拆成什么样。应逐步划分，持续演进，避免服务数量的爆炸性增长。

拆分的过程尽量避免影响产品的日常功能迭代：也就是说要一边做产品功能迭代，一边完成服务化拆分。比如优先剥离比较独立的边界服务（如短信服务等），从非核心的服务出发减少拆分对现有业务的影响，也给团队一个练习、试错的机会。同时当两个服务存在依赖关系时优先拆分被依赖的服务。

服务接口的定义要具备可扩展性：比如微服务的接口因为升级把之前的三个参数改成了四个，上线后导致调用方大量报错，推荐做法服务接口的参数类型最好是封装类，这样如果增加参数就不必变更接口的签名

避免环形依赖与双向依赖：尽量不要有服务之间的环形依赖或双向依赖，原因是存在这种情况说明我们的功能边界没有划分清楚或者有通用的功能没有下沉下来。



阶段性合并：随着你对业务领域理解的逐渐深入或者业务本身逻辑发生了比较大的变化，亦或者之前的拆分没有考虑的很清楚，导致拆分后的服务边界变得越来越混乱，这时就要重新梳理领域边界，不断纠正拆分的合理性。

自动化驱动：部署和运维的成本会随着服务的增多呈指数级增长，每个服务都需要部署、监控、日志分析等运维工作，成本会显著提升。因此，在服务划分之前，应该首先构建自动化的工具及环境。开发人员应该以自动化为驱动力，简化服务在创建、开发、测试、部署、运维上的重复性工作，通过工具实现更可靠的操作，避免微服务数量增多带来的开发、管理复杂度问题。

1.4 微服务拆分策略

功能维度拆分策略

大的原则是基于**业务复杂度**拆分服务： 业务复杂度足够高，应该基于领域驱动拆分服务。业务复杂度较低，选择基于数据驱动拆分服务

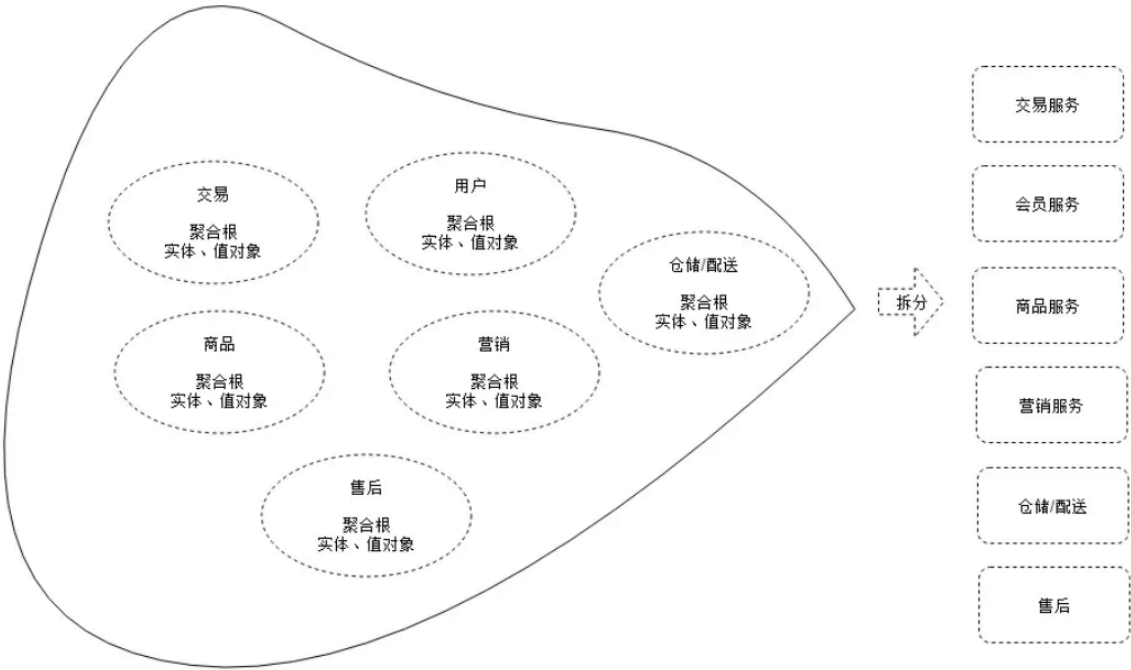
- **基于数据驱动拆分服务：** 自下而上的架构设计方法，通过分析需求，确定整体数据结构，根据表之间的关系拆分服务。

拆分步骤： 需求分析，抽象数据结构，划分服务，确定调用关系和业务流程验证。

- **基于领域驱动拆分服务：** 自上而下的架构设计方法，通过和领域专家建立统一的语言，不断交流，确定关键业务场景，逐步确定边界上下文。领域驱动更强调业务实现效果，认为自下而上的设计可能会导致技术人员不能更好地理解业务方向，进而偏离业务目标。

拆分步骤： 通过模型和领域专家建立统一语言，业务分析，寻找聚合，确定服务调用关系，业务流程验证和持续优化。

以电商的场景为例，交易链路划分的限界上下文如下图左半部分，根据一个限界上下文可以设计一个微服务，拆解出来的微服务如下图右侧部分。



- **还有一种常见拆分场景，从已有单体架构中逐步拆分服务。**

拆分步骤： 前后端分离，提取公共基础服务（如授权服务，分布式ID服务），不断从老系统抽取服务，垂直划分优先，适当水平切分

以上几种拆分方式不是多选一，而是可以根据实际情况自由排列组合。**同时拆分不仅是架构上的调整，也意味着要在组织结构上做出相应的适应性优化，以确保拆分后的服务由相对独立的团队负责维护。**

非功能维度拆分策略

主要考虑六点包括扩展性、复用性、高性能、高可用、安全性、异构性

扩展性

区分系统中变与不变的部分，不变的部分一般是成熟的、通用的服务功能，变的部分一般是改动比较多、满足业务迭代扩展性需要的功能，**我们可以将不变的部分拆分出来，作为共用的服务，将变的部分独立出来满足个性化扩展需要**

同时根据二八原则，系统中经常变动的部分大约只占 20%，而剩下的 80% 基本不变或极少变化，这样的拆分也解决了发布频率过多而影响成熟服务稳定性的问题。

复用性

不同的业务里或服务里经常会出现重复的功能，比如每个服务都有鉴权、限流、安全及日志监控等功能，可以将这些通过的功能拆分出来形成独立的服务。

高性能

将性能要求高或者性能压力大的模块拆分出来，避免性能压力大的服务影响其它服务。

我们也可以**基于读写分离来拆分**，比如电商的商品信息，在 App 端主要是商品详情有大量的读取操作，但是写入端商家中心访问量确很少。因此可以对流量较大或较为核心的服务做读写分离，拆分为两个服务发布，一个负责读，另外一个负责写。

数据一致性是另一个基于性能维度拆分需要考虑的点，对于强一致的数据，属于强耦合，尽量放在同一个服务中（但是有时会因为各种原因需要进行拆分，那就需要有响应的机制进行保证），弱一致性通常可以拆分为不同的服务。

高可用

将可靠性要求高的核心服务和可靠性要求低的非核心服务拆分开来，然后重点保证核心服务的高可用。具体拆分的时候，核心服务可以是一个也可以是多个，只要最终的服务数量满足“三个火枪手”的原则就可以。

安全性

不同的服务可能对信息安全有不同的要求，因此**把需要高度安全的服务拆分出来**，进行区别部署，比如设置特定的 DMZ 区域对服务进行分区部署，可以更有针对性地满足信息安全的要求，也可以降低对防火墙等安全设备吞吐量、并发性等方面的要求，降低成本，提高效率。

异构性

对于对开发语言种类有要求的业务场景，可以用不同的语言将其功能独立出来实现一个独立服务。

1.5 拆分注意的风险

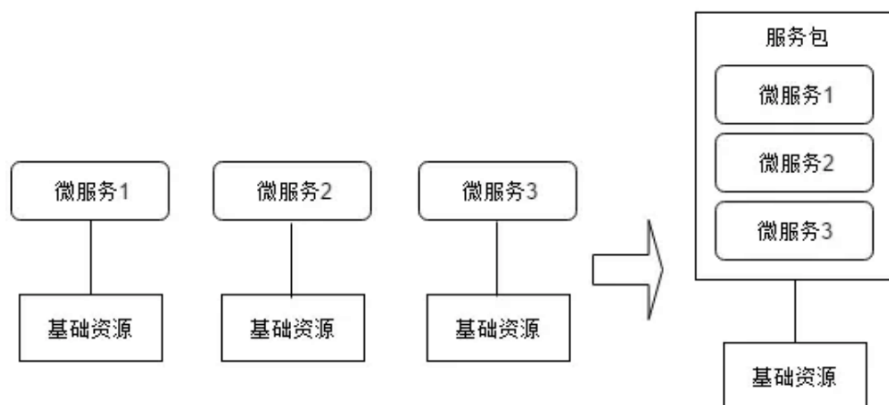
不打无准备之仗：开发团队是否具备足够的经验，能否驾驭微服务的技术栈，可能是第一个需要考虑的点。

不断纠正：我们需要承认我们的认知是有限的，只能基于目前的业务状态和有限的对未来的预测来制定出一个相对合适的拆分方案，而不是所谓的最优方案，任何方案都只能保证在当下提供了相对合适的粒度和划分原则，要时刻做好在未来的某一个时刻会变得不和时宜、需要再次调整的准备。

要做行动派，而不是理论派：**在具体怎么拆分上，也不要太纠结于是否合适，如果拆了之后发现真的不合适，在重新调整就好了。**如果要灵活调整，可以针对服务化架构搭建起一套完成的能力体系，比如服务治理平台、数据迁移工具、数据双写等等

服务只拆不合：

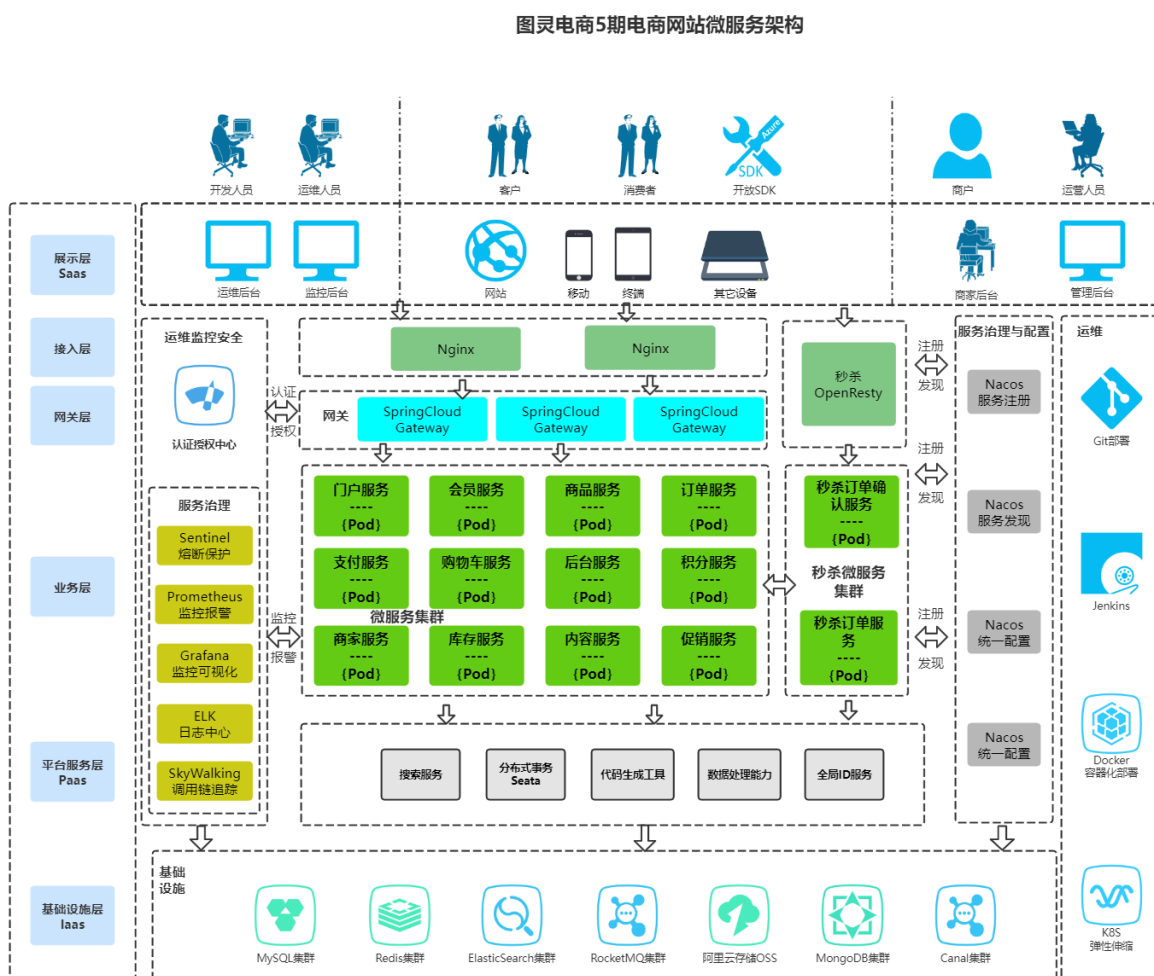
- 拆相当于我们开发代码，合相当于重构代码。随着我们对应用程序领域的了解越来越深，它们需要随着时间的推移而变化。
- 人员和服务数量的不匹配，导致的维护成本增加，也是导致服务合并的一个重要原因。
- **如果微服务数量过多和资源不匹配，则可以考虑合并多个微服务到服务包，部署到一台服务器，这样可以节省服务运行时的基础资源消耗也降低了维护成本。**需要注意的是，虽然服务包是运行在一个进程中，但是服务包内的服务依然要满足微服务定义，以便在未来某一天要重新拆分的时候可以很快就分离



2. 电商项目微服务拆分

2.1 电商项目微服务架构图

五期项目的各个模块和中间件组合起来，发挥着各自的作用，组成了如下图 所示的[架构图](#)：



2.2 Spring Cloud&Spring Cloud Alibaba技术栈选型

Spring Cloud Alibaba官网: <https://github.com/alibaba/spring-cloud-alibaba/wiki>

SpringCloud的几大痛点:

- SpringCloud部分组件停止维护和更新, 给开发带来不便;
- SpringCloud部分环境搭建复杂, 没有完善的可视化界面, 我们需要大量的二次开发和定制
- SpringCloud配置复杂, 难以上手, 部分配置差别难以区分和合理应用

SpringCloud Alibaba的优势:

- 阿里使用过的组件经历了考验, 性能强悍, 设计合理, 现在开源出来大家用成套的产品搭配完善的可视化界面给开发运维带来极大的便利
- 搭建简单, 学习曲线低。

所以我们优先选择Spring Cloud Alibaba提供的微服务组件

Spring Cloud Alibaba官方推荐版本选择:

<https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E>

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version
2.2.6.RELEASE	Spring Cloud Hoxton.SR9	2.3.2.RELEASE


关于依赖下载不下来的问题:





























1. idea使用专业版, 检查idea的maven环境配置
2. maven配置[阿里云镜像](#)

2.3 服务模块的拆分

五期项目是在以前的项目上发展而来, 在业务上基本保持不动, 但做了更细致的微服务拆分, 大体上秉承了前面所说的微服务设计原则, 同时在具体的业务模块实现上, 根据实际资源情况和教学、授课要求做了一定程度的取舍和合并。具体来说, 相对一般的电商系统, 我们没有单独实现账户、库存、支付服务, 报表往往需要和 BI (Business Intelligence) 报表框架集成, 所以也未实现。因为说到底, 我们学习本项目的目的是为了学习技术而不是为了学习电商业务。

业务服务模块

 **tlmallV5** D:\TuLing\VIPL\Mall\code\tlmallV5

- >  .idea
- ✓  doc
 - >  htmljss
 - >  nginx
- >  **tulingmall-admin**
- >  **tulingmall-authcenter**
- >  **tulingmall-canal**
- >  **tulingmall-cart**
- >  **tulingmall-common**
- >  **tulingmall-core**
- >  **tulingmall-gateway**
- >  **tulingmall-member**
- ✓  **tulingmall-order**
 - >  **tulingmall-order-curr**
 - >  **tulingmall-order-history**
 -  pom.xml
- >  **tulingmall-portal**
- >  **tulingmall-product**
- >  **tulingmall-promotion**
- >  **tulingmall-redis-comm**
- >  **tulingmall-redis-multi**
- >  **tulingmall-search**
- >  **tulingmall-security**
- ✓  **tulingmall-sk**
 - >  **tulingmall-sk-cart**
 - >  **tulingmall-sk-order**
 -  pom.xml
- >  **tulingmall-unqid**

- tulingmall-authcenter 认证中心程序
- tulingmall-canal 数据同步程序
- tulingmall-cart 购物车程序
- tulingmall-common 通用模块，被其他程序以 jar 包形式使用
- tulingmall-core 遗留模块，主要包含 model 的声明，被其他程序以 jar 包形式使用
- tulingmall-gateway 网关程序
- tulingmall-member 用户管理程序
- tulingmall-order-curr 订单程序
- tulingmall-order-history 历史订单处理程序
- tulingmall-portal 商城首页入口程序
- tulingmall-product 商品管理程序

- tulingmall-promotion 促销管理程序
- tulingmall-redis-comm 缓存模块，被其他程序以 jar 包形式使用
- tulingmall-redis-multi 多源缓存模块，被其他程序以 jar 包形式使用
- tulingmall-search 商品搜索程序
- tulingmall-security 安全模块，被其他程序以 jar 包形式使用
- tulingmall-sk-cart 秒杀确认单处理
- tulingmall-sk-order 秒杀订单处理
- tulingmall-unqid 分布式 ID 生成程序

本节课会以tulingmall-member, tulingmall-promotion模块演示接入微服务Spring Cloud&Spring Cloud Alibaba技术栈

课后作业

电商项目单体架构版本进行微服务拆分

https://vip.tulingxueyuan.cn/detail/p_607e83a2e4b09134c989f5cd/8?product_id=p_607e83a2e4b09134c989f5cd

2.4 数据库的拆分

整个项目中数据库被拆分为：

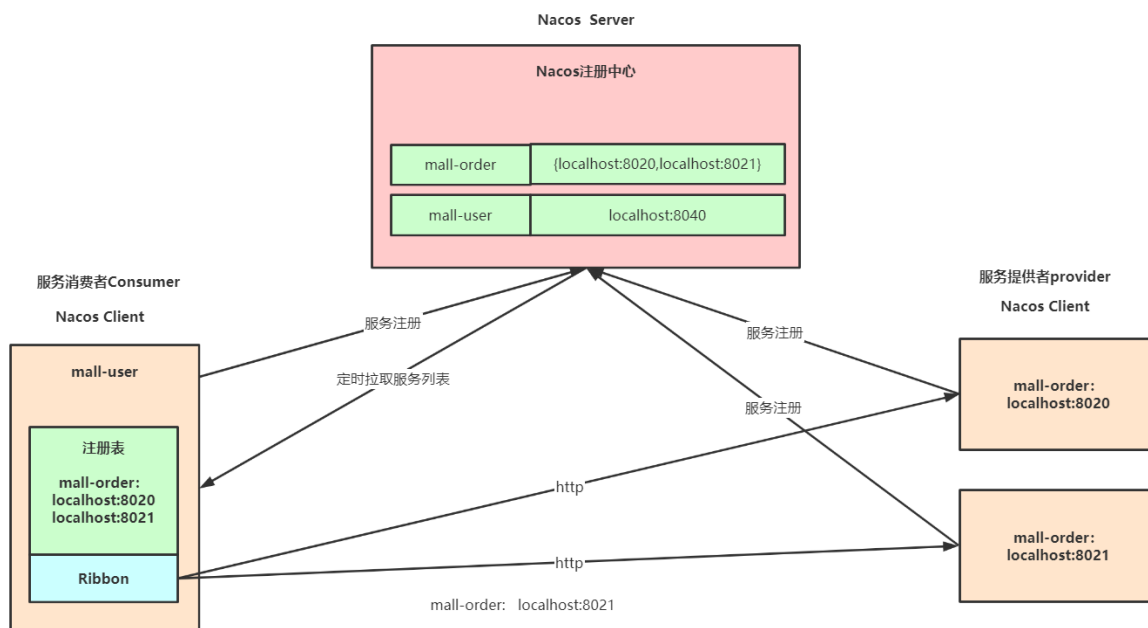
- 商品库：tl_mall_goods
- 促销库：tl_mall_promotion
- 用户库：tl_mall_user
- 其他库：tl_mall_normal
- 订单库：tl_mall_order
- 购物车库：tl_mall_cart

课程关键 Table

- 用户库 ums_member，用户/会员表
- 商品库 pms_product 商品表，主要包括四部分：商品的基本信息、商品的促销信息、商品的属性信息、商品的关联
- 商品库 pms_sku_stock 商品 SKU 库存表
- 订单库 oms_order 订单表
- 订单库 oms_order_item 订单详情表，订单中包含的商品信息，一个订单中会有多个订单商品信息，所以 oms_order 和它是一对多的关系

- 促销库 sms_home_brand 首页品牌推荐表
- 促销库 sms_home_new_product 首页显示的新鲜好物信息
- 促销库 sms_home_recommend_product 首页显示的人气推荐信息
- 促销库 sms_home_recommend_subject 首页显示的专题精选信息
- 促销库 sms_home_advertise 首页显示的轮播广告信息
- 促销库 sms_flash_promotion 秒杀/限时购活动的信息
- 促销库 sms_flash_promotion_product_relation 存储与秒杀/限时购相关的商品信息，也包含了秒杀/限时购的库存信息

2.5 接入Nacos注册中心



注意：项目使用的nacos server版本：v1.4.3

<https://github.com/alibaba/nacos/releases/download/1.4.3/nacos-server-1.4.3.zip>

将微服务注册到Nacos Server

1) 引入依赖

```

1 <!--nacos 注册中心 -->
2 <dependency>
3   <groupId>com.alibaba.cloud</groupId>
4   <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
5 </dependency>

```

2) 在yml中配置注册中心地址

```

1 spring:
2   application:
3     name: mall-order #微服务名

```

```
4  cloud:
5  nacos:
6  discovery:
7  server-addr: 192.168.65.103:8848 #注册中心地址
8  namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618 #环境隔离
```

2.6 接入Nacos配置中心

使用nacos作为微服务配置中心

1) 引入依赖

```
1  <!-- nacos 配置中心 -->
2  <dependency>
3  <groupId>com.alibaba.cloud</groupId>
4  <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
5  </dependency>
```

2) 创建bootstrap.yml文件，添加配置中心的配置

```
1  spring:
2  application:
3  name: tulingmall-member #微服务的名称
4  cloud:
5  nacos:
6  config:
7  serverAddr: 192.168.65.103:8848 #配置中心的地址
8  namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618
9  # dataid 为 yml 的文件扩展名配置方式
10 # `${spring.application.name}.${file-extension:properties}`
11 file-extension: yml
12
13 #通用配置
14 shared-configs[0]:
15 data-id: tulingmall-nacos.yml
16 group: DEFAULT_GROUP
17 refresh: true
18 shared-configs[1]:
19 data-id: tulingmall-redis.yml # redis服务集群配置
20 group: DEFAULT_GROUP
21 refresh: true
22
23 #profile粒度的配置
24 #`${spring.application.name}-${profile}.${file-extension:properties}`
```

```
25 profiles:
26 active: dev
```

3) 添加微服务配置和公共的配置到nacos配置中心

public fox				
配置管理 fox 6cd8d896-4d19-4e33-9840-26e4bee9a618 查询结果: 共查询到 7 条满足要求的配置。				
Data ID:	添加通配符"*"进行模糊查询	Group:	添加通配符"*"进行模糊查询	查询 高级查询 导出查询结果 导入配置
<input type="checkbox"/>	Data Id <small>↓↑</small>	Group <small>↓↑</small>	归属应用: <small>↓↑</small>	操作
<input type="checkbox"/>	tulingmall-nacos.yml	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	tulingmall-db-common.yml	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	tulingmall-gateway-dev.yml	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	tulingmall-member-dev.yml	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	tulingmall-promotion-dev.yml	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	tulingmall-redis.yml	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	tulingmall-redis-key-dev.yml	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多

2.7 基于openfeign实现微服务间的调用

使用openfeign作为服务间调用组件，**业务场景：用户查询优惠券**

1) 引入依赖

```
1 <!-- 服务远程调用 -->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-openfeign</artifactId>
5 </dependency>
```

2) 编写调用接口+@FeignClient注解，指定要调用的微服务及其接口方法

```
1 @FeignClient(value = "tulingmall-coupons", path = "/coupon")
2 public interface CouponsFeignService {
3
4   @RequestMapping(value = "/list", method = RequestMethod.GET)
5   @ResponseBody
6   CommonResult<List<SmsCouponHistory>> list(@RequestParam(value = "useStatus", required = false) Integer useStatus
7   , @RequestHeader("memberId") Long memberId);
8
9 }
```

3) 启动类添加@EnableFeignClients注解，开启openFeign远程调用功能

```
1 @SpringBootApplication
2 @EnableFeignClients
3 public class TulingmallMemberApplication {
4
```

```

5 public static void main(String[] args) {
6     SpringApplication.run(TulingmallMemberApplication.class, args);
7 }
8
9 }

```

4) 测试，发起远程服务调用

```

1 @Autowired
2 private CouponsFeignService couponsFeignService;
3
4 @RequestMapping(value = "/coupons", method = RequestMethod.GET)
5 public CommonResult getCoupons(@RequestParam(value = "useStatus", required = false) Integer useStatus
6 ,@RequestHeader("memberId") Long memberId){
7     // 通过openfeign从远程微服务tulingmall-coupons获取优惠券信息
8     return couponsFeignService.list(useStatus, memberId);
9 }

```

5) 开启openfeign日志配置

```

1 @Configuration
2 public class FeignConfig {
3
4     @Bean
5     public Logger.Level feignLoggerLevel() {
6         return Logger.Level.FULL;
7     }
8
9 }

```

如果日志不显示，可以在yml中通过logging.level设置日志级别

```

1 logging:
2   level:
3     com.tuling: debug

```

扩展场景：实现RequestInterceptor，添加请求头参数用于传递memberId

```

1 @Slf4j
2 public class HeaderInterceptor implements RequestInterceptor {
3     @Override
4     public void apply(RequestTemplate template) {
5
6         ServletRequestAttributes attributes = (ServletRequestAttributes) Request
            ContextHolder.getRequestAttributes();
7         if(null != attributes){

```



```

8  HttpServletRequest request = attributes.getRequest();
9  log.info("从Request中解析请求头");
10  template.header("memberId", request.getHeader("memberId"));
11  }
12  }
13  }
14
15  # FeignConfig.java中添加拦截器配置
16  @Bean
17  public RequestInterceptor requestInterceptor() {
18      return new HeaderInterceptor();
19  }

```

RequestInterceptor作用：在发送请求前，对发送的模板进行操作，例如设置请求头等属性。

2.8 网关服务搭建

创建tulingmall-gateway服务

1) 引入依赖

```

1  <!-- gateway网关 -->
2  <dependency>
3      <groupId>org.springframework.cloud</groupId>
4      <artifactId>spring-cloud-starter-gateway</artifactId>
5  </dependency>
6
7  <!-- nacos服务注册与发现 -->
8  <dependency>
9      <groupId>com.alibaba.cloud</groupId>
10     <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
11 </dependency>
12 <!-- nacos 配置中心 -->
13 <dependency>
14     <groupId>com.alibaba.cloud</groupId>
15     <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
16 </dependency>
17
18 <dependency>
19     <groupId>org.springframework.boot</groupId>
20     <artifactId>spring-boot-starter-actuator</artifactId>
21 </dependency>

```

注意：会和spring-webmvc的依赖冲突，需要排除spring-webmvc

2) 编写yaml配置文件

application.yml

```
1 server:
2   port: 8888
3 spring:
4   application:
5     name: tulingmall-gateway
6     #配置nacos注册中心地址
7   cloud:
8     nacos:
9     discovery:
10      server-addr: 192.168.65.103:8848 #注册中心地址
11      namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618 #环境隔离
12
13 gateway:
14   routes:
15     - id: tulingmall-member #路由ID，全局唯一
16       uri: lb://tulingmall-member
17       predicates:
18         - Path=/member/**,/sso/**
19         - id: tulingmall-promotion
20           uri: lb://tulingmall-promotion
21           predicates:
22             - Path=/coupon/**
23
24 logging:
25   level:
26     org.springframework.cloud.gateway: debug
```

3) gateway配置移植到配置中心，添加bootstrap.yml：

```
1 spring:
2   application:
3     name: tulingmall-gateway #微服务的名称
4   cloud:
5     nacos:
6     config:
7       serverAddr: 192.168.65.103:8848 #配置中心的地址
8       namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618
```

```

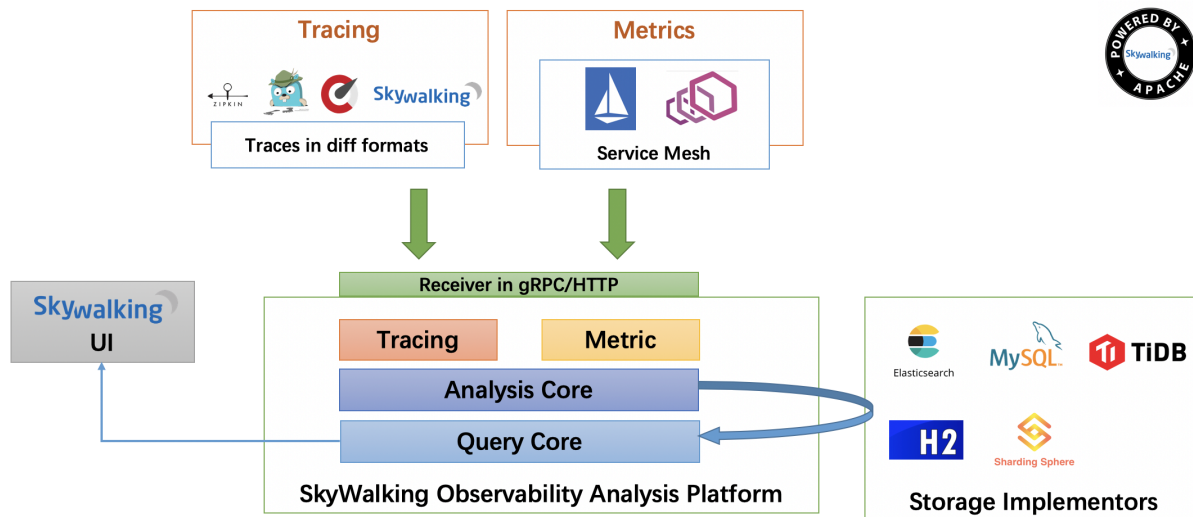
9  # dataid 为 yml 的文件扩展名配置方式
10 # `${spring.application.name}.${file-extension:properties}`
11 file-extension: yml
12
13 #通用配置
14 shared-configs[0]:
15 data-id: tulingmall-nacos.yml
16 group: DEFAULT_GROUP
17 refresh: true
18
19 #profile粒度的配置
20 #`${spring.application.name}-${profile}.${file-extension:properties}`
21 profiles:
22 active: dev

```

2.9 接入Skywalking

1) 搭建Skywalking OAP服务

参考微服务专题Skywalking课程



2) 微服务配置Skywalking Agent

以tulingmall-member为例，在jvm参数中增加agent配置

```

1 -javaagent:D:\apache\apache-skywalking-java-agent-8.11.0\skywalking-agent
  \skywalking-agent.jar
2 -Dskywalking.agent.service_name=tulingmall-member
3 -Dskywalking.collector.backend_service=192.168.65.103:11800

```

访问Skywalking UI : <http://192.168.65.103:8080/>

3) 集成日志框架, 生成traceId

[logback官方配置](#)

引入依赖

```
1 <!-- apm-toolkit-logback-1.x -->
2 <dependency>
3   <groupId>org.apache.skywalking</groupId>
4   <artifactId>apm-toolkit-logback-1.x</artifactId>
5   <version>8.9.0</version>
6 </dependency>
```

添加logback日志文件logback-spring.xml

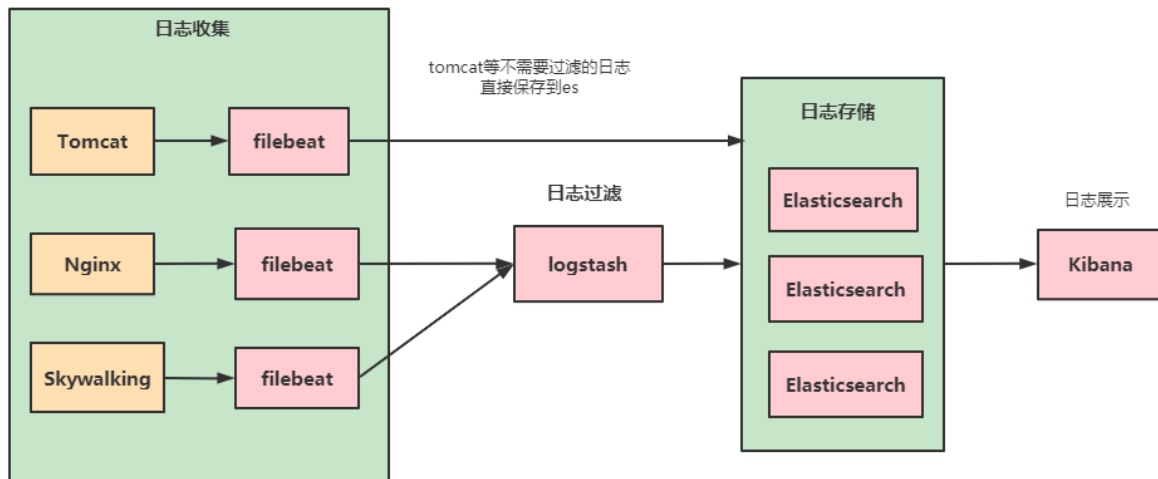
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
3
4   <!-- 引入 Spring Boot 默认的 logback XML 配置文件 -->
5   <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
6
7   <!-- 控制台 Appender -->
8   <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
9     <!-- 日志的格式化 -->
10    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
11      <layout class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.TraceIdPatternLogbackLayout">
12        <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level %tid %t %logger{36}: %msg%n</Pattern>
13      </layout>
14    </encoder>
15  </appender>
16
17  <!-- 从 Spring Boot 配置文件中, 读取 spring.application.name 应用名 -->
18  <springProperty name="applicationName" scope="context" source="spring.application.name" />
19
20  <!-- 日志文件 Appender -->
21  <appender name="file" class="ch.qos.logback.core.rolling.RollingFileAppender">
22    <!-- 日志文件的路径 -->
23    <file>/logs/tulingmall/${applicationName}.log</file>
24    <!--滚动策略, 基于时间 + 大小的分包策略 -->
```

```

25 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
26 <fileNamePattern>/logs/tulingmall/${applicationName}-${d{yyyy-MM-dd}}-
%i.log</fileNamePattern>
27 <MaxHistory>20</MaxHistory>
28 <timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
29 <maxFileSize>500MB</maxFileSize>
30 </timeBasedFileNamingAndTriggeringPolicy>
31 </rollingPolicy>
32 <!-- 日志的格式化 -->
33 <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
34 <layout class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.Trace
IdPatternLogbackLayout">
35 <!-- 日志格式中添加 %tid 即可输出 trace id -->
36 <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level %tid %t %logger{36}:
%msg%n</Pattern>
37 </layout>
38 </encoder>
39 </appender>
40
41 <!-- 设置 Appender -->
42 <root level="DEBUG">
43 <appender-ref ref="console"/>
44 <appender-ref ref="file"/>
45 </root>
46
47 </configuration>

```

2.10 整合ELK收集微服务链路日志



方案一：使用logstash日志插件

1) 引入依赖

```

1 <dependency>
2   <groupId>net.logstash.logback</groupId>
3   <artifactId>logstash-logback-encoder</artifactId>
4   <version>6.3</version>
5 </dependency>

```

2) logback-spring.xml中添加logstash配置

```

1 <!-- add converter for %tid -->
2 <conversionRule conversionWord="tid" converterClass="org.apache.skywalking
3   g.apm.toolkit.log.logback.v1.x.LogbackPatternConverter"/>
4 <!-- add converter for %sw_ctx -->
5 <conversionRule conversionWord="sw_ctx" converterClass="org.apache.skywal
6   king.apm.toolkit.log.logback.v1.x.LogbackSkyWalkingContextPatternConverter",
7
8 <appender name="LOGSTASH" class="net.logstash.logback.appender.LogstashTc
9   pSocketAppender">
10   <destination>192.168.65.25:9527</destination>
11   <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEn
12     coder">
13     <providers>
14     <timestamp>
15     <timeZone>UTC</timeZone>
16   </timestamp>
17   <pattern>
18   <pattern>
19     {
20     "level": "%level",
21     "tid": "%tid",

```

```

18   "skyWalkingContext": "%sw_ctx",
19   "thread": "%thread",
20   "class": "%logger{1.}:%L",
21   "message": "%message",
22   "stackTrace": "%exception{10}"
23 }
24 </pattern>
25 </pattern>
26 </providers>
27 </encoder>
28 </appender>
29
30 <!-- 设置 Appender -->
31 <root level="INFO">
32   <appender-ref ref="LOGSTASH"/>
33 </root>

```

3) 添加tulingmall-logstash.conf配置, 启动logstash

```

1 input {
2   tcp {
3     # 在9527端口接收logback传来的日志
4     host => "0.0.0.0"
5     port => 9527
6     mode => "server"
7     tags => ["tulingmall"]
8     codec => json_lines
9   }
10 }
11 filter {
12 }
13 output {
14   #控制台输出
15   stdout { codec => rubydebug }
16   #输出到es
17   elasticsearch {
18     hosts => ["127.0.0.1:9200"]
19     user => "elastic"
20     password => "123456"
21     index => "tulingmall-%{+YYYY.MM.dd}"
22   }
23 }

```

启动logstash

```
1 bin/logstash -f tulingmall-logstash.conf --config.reload.automatic
2 # 后台启动
3 nohup bin/logstash -f tulingmall-logstash.conf >/dev/null 2>tulingmall.log &
```

方案二：标准的ELK收集方案：通过FileBeat收集本地日志

1) 启动FileBeat收集本地日志

修改filebeat.yml的配置，指定日志收集地址和logstash地址

```
1 filebeat.inputs:
2 - type: log
3   enabled: true
4   paths:
5   - f:\logs\tulingmall\*.log
6   multiline.pattern: '^\\d{4}\\-\\d{2}\\-\\d{2}\\s\\d{2}:\\d{2}:\\d{2}\\.\\d{3}'
7   multiline.negate: true
8   multiline.match: after
9 #----- Logstash output -----
10 output.logstash:
11   hosts: ["192.168.65.220:5044"]
```

启动FileBeats

```
1 #启动FileBeats
2 filebeat.exe -e -c filebeat.yml
```

2) Logstash 解析 Trace ID

通过 grok 自定义正则表达式，可以从日志行中抽取出 trace id，就可以在 es 中建立索引，方便日志检索。使用 (?<trace_id>[0-9a-f.]{53,54}) 即可抽取出 trace id。

2 hits

time	level	trace_id	thread	content	log.file.path
> 2021-06-17 15:19:59.237	ERROR	812f0a005c1245c0b2cb0adcc754.91.16239143992210013	http-nio-8877-exec-3	Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is feign.FeignException\$InternalServerError: status 500 reading CouponsFeignService#list(Integer,Long)] with root cause feign.FeignException\$InternalServerError: status 500 reading CouponsFeignService#list(Integer,Long) at feign.FeignException.errorStatus(FeignException.java:114) at feign.FeignException.errorStatus(FeignException.java:86)	f:\logs\tulingmall\tulingmall-member.log
> 2021-06-17 15:19:59.233	ERROR	812f0a005c1245c0b2cb0adcc754.91.16239143992210013	http-nio-8855-exec-7	Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is java.lang.IllegalArgumentException: 非法参数异常] java.lang.IllegalArgumentException: 非法参数异常 at com.tuling.tulingmall.controller.UmsCouponController.list\$original\$ejUHD41b(UmsCouponController.java:54) at com.tuling.tulingmall.controller.UmsCouponController.list\$original\$ejUHD41b(UmsCouponController.java:54)	f:\logs\tulingmall\tulingmall-coupons.log

```
1 input {
```

```

2  beats {
3    port => 5044
4    codec => "json"
5  }
6 }
7
8 filter {
9   grok {
10    match => {
11      "message" => "(?<time>\d{4}\-\d{2}\-\d{2}\s\d{2}\:\d{2}\:\d{2}\.\d{3})
12      \s(?<level>\w{4,5})\s+\T\I\D\:\s*(?<trace_id>[0-9a-f.]{53,54})\s%{DATA:thread}\s%{DATA:class}\: %{GREEDYDATA:content}"
13    }
14  }
15  mutate {
16    remove_field => "message" # 删除原始日志内容节省存储和带宽
17  }
18
19 output {
20   elasticsearch {
21     hosts => ["192.168.65.220:9200"]
22     index => "tlmall-log" # ES 重建索引
23   }
24 }
25
26

```

启动Logstash

```

1 # 测试配置是否正确
2 bin/logstash -f config/tlmall-skywalking.conf --config.test_and_exit
3 #启动Logstash
4 bin/logstash -f config/tlmall-skywalking.conf --config.reload.automatic

```

测试：在kibana中根据trace_id搜索对应的系统日志

整合 Skywalking 和 ELK 后，通过 trace id，在 Skywaling 中快速看到链路中哪个环节出了问题，然后在 ELK 中按 trace id 搜索对应的系统日志，这样就可以很方便的定位出问题，为线上排障提供了方便。

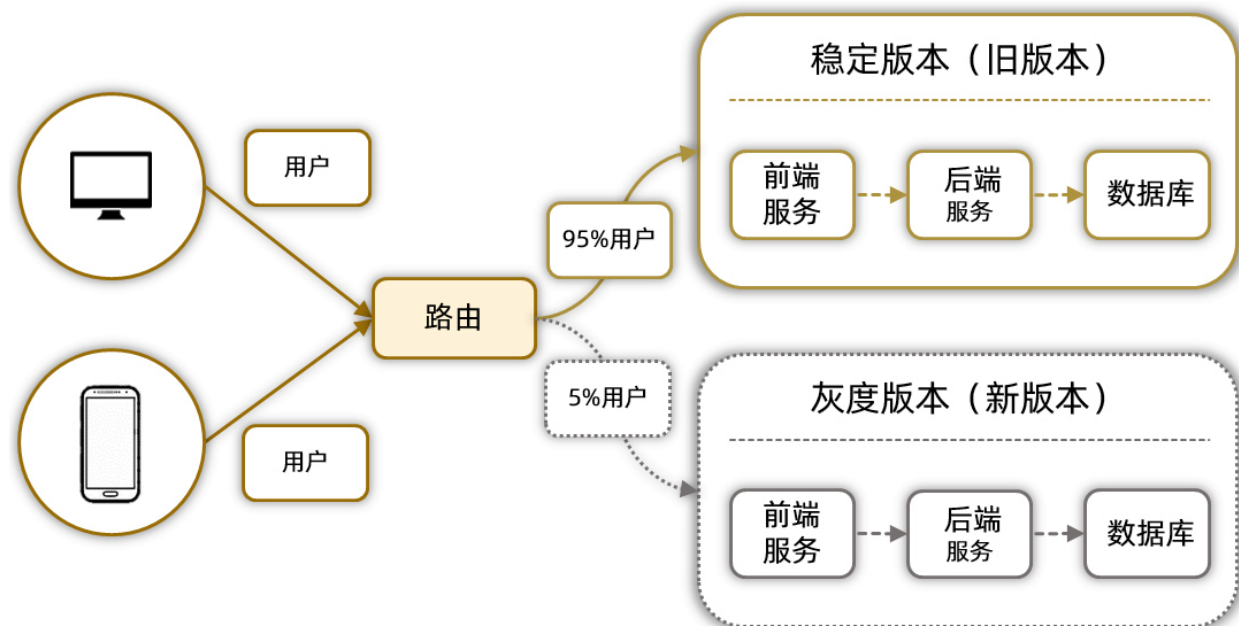
a88c9113cc2244eabdf3aa9699f9a4cc.155.16652943724880021						
level: ERROR × + 添加筛选						
tulingmall-2022.10* 2 个命中						
<div>搜索字段名称</div> <div>按类型筛选 0</div> <div>选定字段 6</div> <div> <div>tid</div> <div>level</div> <div>thread</div> <div>skyWalkingContext</div> <div>class</div> <div>message</div> </div>						
TID:a88c9113cc2244eabdf3aa9699f9a4cc.155.16652943724880021	ERROR	http-nio-8877-exec-5	SW_CTX:[tulingmall-member,c6a3ff849d6844f3a08845f8e9d8f780192.168.65.103,a88c9113cc2244eabdf3aa9699f9a4cc.155.16652943724880021,d1d873c2d69a4bcd2e1d15f94fa45a.216.16652943724920000,0]	org.apache.catalina.core.ContainerBase.[Tomcat].[localhost].[/].[/].dispatcherServlet:?	Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is feign.FeignException\$InternalServerError: [500] during [GET] to [http://tulingmall-promotion/coupon/list] [CouponsFeignService\$list(Integer,Long)]: [{"timestamp":"2022-10-09T05:46:12.539+08:00","status":500,"error":"Internal Server Error","message":""}]]	
TID:a88c9113cc2244eabdf3aa9699f9a4cc.155.16652943724880021	ERROR	http-nio-8822-exec-4	SW_CTX:[tulingmall-promotion,6726e1b14c4b416a48ca693b178d7440192.168.65.103,a88c9113cc2244eabdf3aa9699f9a4cc.155.16652943724880021,70a46094a2e04119bce4324c55d91b1e.180.16652943724940000,0]	org.apache.catalina.core.ContainerBase.[Tomcat].[localhost].[/].[/].dispatcherServlet:?	Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is java.lang.IllegalArgumentException: 非法参数异常] with root cause	

3. 微服务全链路灰度解决方案

3.1 灰度发布

概念

灰度发布 Gray Release（又名金丝雀发布 Canary Release）。不停机旧版本，部署新版本，高比例流量（例如：95%）走旧版本，低比例流量（例如：5%）切换到新版本，通过监控观察无问题，逐步扩大范围，最终把所有流量都迁移到新版本上。属无损发布。



优点

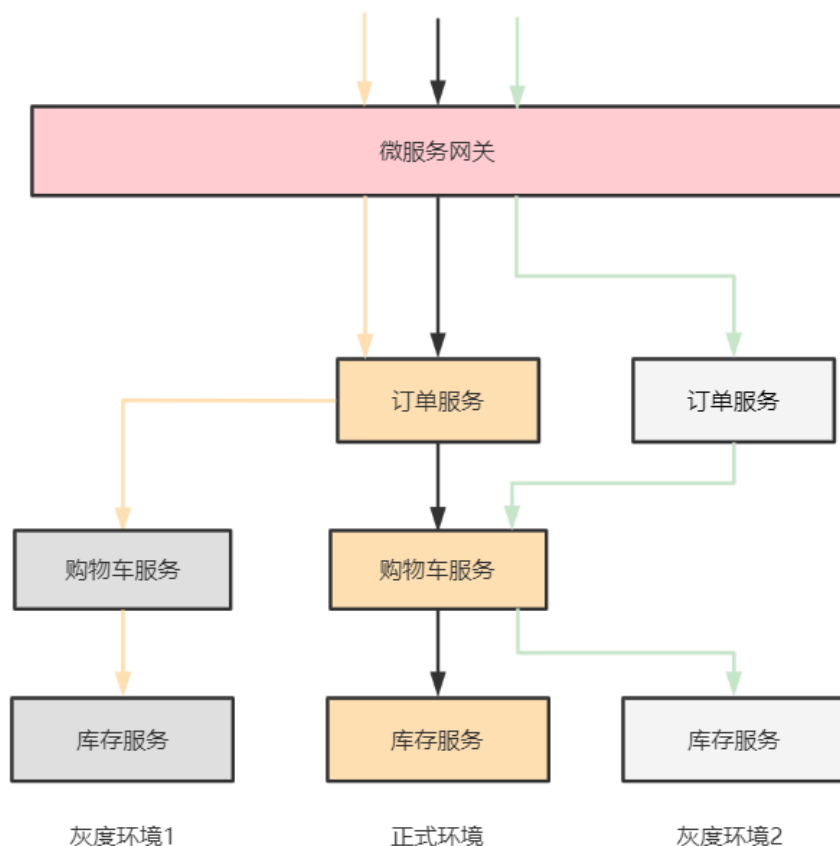
灵活简单，不需要用户标记驱动。安全性高，新版本如果出现问题，只会发生在低比例的流量上

缺点

成本较高，需要部署稳定/灰度两套环境

3.2 微服务全链路灰度

微服务体系架构中，服务之间的依赖关系错综复杂，有时某个功能发版依赖多个服务同时升级上线。我们希望对这些服务的新版本同时进行小流量灰度验证，这就是微服务架构中特有的全链路灰度场景，通过构建从网关到整个后端服务的环境隔离来对多个不同版本的服务进行灰度验证。在发布过程中，我们只需部署服务的灰度版本，流量在调用链路上流转时，由流经的网关、各个中间件以及各个微服务来识别灰度流量，并动态转发至对应服务的灰度版本。如下图：



上图可以很好展示这种方案的效果，我们用不同的颜色来表示不同版本的灰度流量，可以看出无论是微服务网关还是微服务本身都需要识别流量，根据治理规则做出动态决策。当服务版本发生变化时，这个调用链路的转发也会实时改变。相比于利用机器搭建的灰度环境，这种方案不仅可以节省大量的机器成本和运维人力，而且可以帮助开发者实时快速的对线上流量进行精细化的全链路控制。

3.3 全链路灰度设计思路

那么全链路灰度具体是如何实现呢？

我们需要解决以下问题：

- 1.链路上各个组件和服务能够根据请求流量特征进行动态路由。
- 2.需要对服务下的所有节点进行分组，能够区分版本。

- 3.需要对流量进行灰度标识、版本标识。
- 4.需要识别出不同版本的灰度流量。

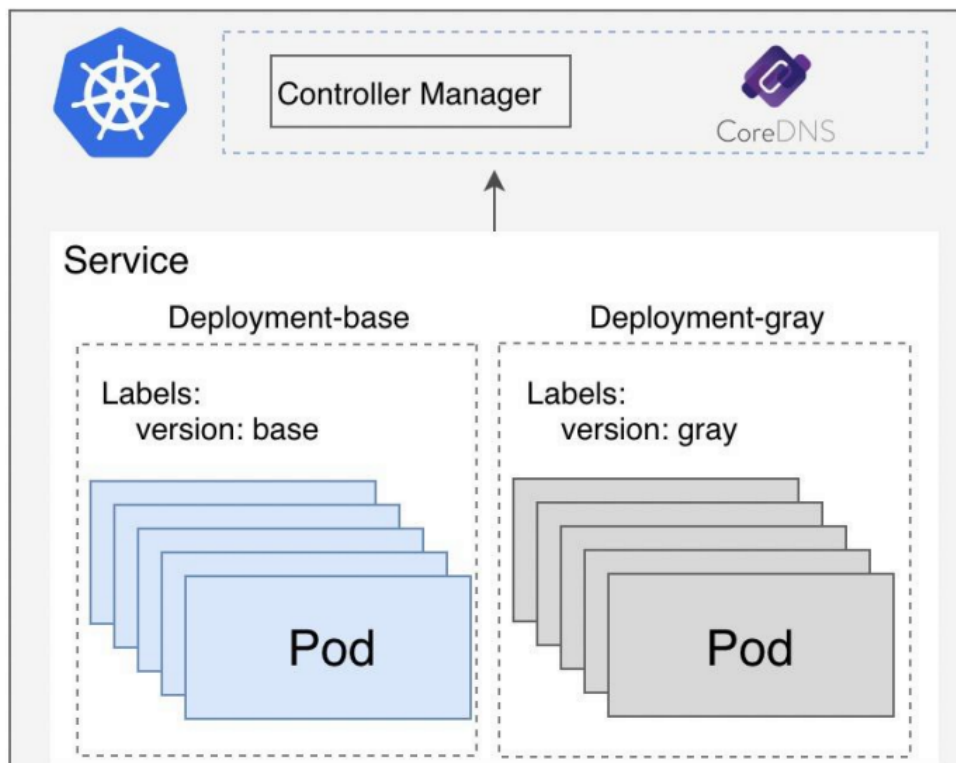
标签路由

标签路由通过对服务下所有节点按照标签名和标签值不同进行分组，使得订阅该服务节点信息的服务消费端可以按需访问该服务的某个分组，即所有节点的一个子集。服务消费端可以使用服务提供者节点上的任何标签信息，根据所选标签的实际含义，消费端可以将标签路由应用到更多的业务场景中。

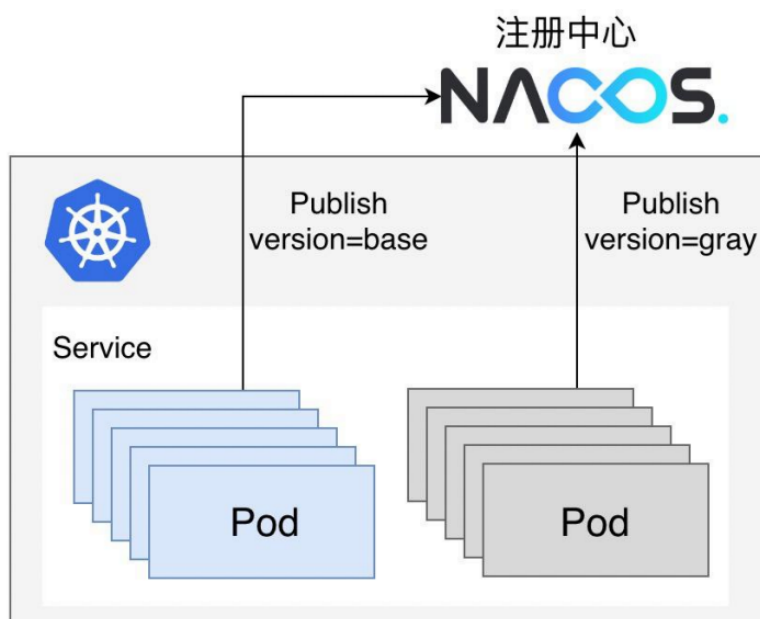
节点打标

那么如何给服务节点添加不同的标签呢？

在使用Kubernetes Service作为服务发现的业务系统中，服务提供者通过向ApiServer提交Service资源完成服务暴露，服务消费端监听与该Service资源下关联的Endpoint资源，从Endpoint资源中获取关联的业务Pod 资源，读取上面的Labels数据并作为该节点的元数据信息。所以，我们只要在业务应用描述资源Deployment中的Pod模板中为节点添加标签即可。



在使用Nacos作为服务发现的业务系统中，一般是需要业务根据其使用的微服务框架来决定打标方式。如果Java应用使用的Spring Cloud微服务开发框架，我们可以为业务容器添加对应的环境变量来完成标签的添加操作。比如我们希望为节点添加版本灰度标，那么为业务容器添加`spring.cloud.nacos.discovery.metadata.version=gray`，这样框架向Nacos注册该节点时会为其添加一个标签`version=gray`。



流量染色

请求链路上各个组件如何识别出不同的灰度流量？

答案就是流量染色，为请求流量添加不同灰度标识来方便区分。我们可以在请求的源头上对流量进行染色，前端在发起请求时根据用户信息或者平台信息的不同对流量进行打标。如果前端无法做到，我们也可以在微服务网关上对匹配特定路由规则的请求动态添加流量标识。此外，流量在链路中流经灰度节点时，如果请求信息中不含有灰度标识，需要自动为其染色，接下来流量就可以在后续的流转过程中优先访问服务的灰度版本。

分布式链路追踪

如何保证灰度标识能够在链路中一直传递下去呢?

借助于分布式链路追踪思想，我们也可以传递一些自定义信息，比如灰度标识。

总结

首先，需要支持动态路由功能，对于Spring Cloud、Dubbo开发框架，可以对出口流量实现自定义Filter，在该Filter中完成流量识别以及标签路由。同时需要借助分布式链路追踪技术完成流量标识链路传递以及流量自动染色。此外，需要引入一个中心化的流量治理平台，方便各个业务线的开发者定义自己的全链路灰度规则。

实现全链路灰度的能力，无论是成本还是技术复杂度都是比较高的，以及后期的维护、扩展都是非常大的成本。

3.4 基于Discovery实现全链路灰度

Discovery 【探索】企业级云原生微服务开源解决方案

官方文档：[全链路灰度发布](https://github.com/Nepxion/Discovery/wiki)
<https://github.com/Nepxion/Discovery/wiki>

Discovery版本选择

当前项目可以选择Discovery版本6.12.1

发布策略

提醒：版本号右边，↑ 表示>=该版本号，↓ 表示<=该版本号

版本	状态	SC	SB	SCA
8.0.0 (商业版)	🟢	2021.x.x	2.7.x 2.6.x	2022.x
7.0.0 (商业版)	🟢	2020.x.x	2.5.x 2.4.1 ↑	2021.x
6.13.1 ↑	🟢	H.SR5 ↑ H G F	2.3.x 2.2.x 2.1.x 2.0.x	2.2.7.RELEASE ↑
6.12.1 ↓	🟢	H.SR5 ↑ H G F	2.3.x 2.2.x 2.1.x 2.0.x	2.2.6.RELEASE ↓ 2.1.x 2.0.x
5.6.0	🔴	G	2.1.x	2.1.x
4.15.0	🔴	F	2.0.x	2.0.x
3.29.0	🔵	E	1.5.x	1.5.x
2.0.x	🔴	D	1.x.x	1.5.x
1.0.x	🔴	C	1.x.x	1.5.x

🟢 表示维护中 | 🔵 表示不维护，但可用，强烈建议升级 | 🔴 表示不维护，不可用，已废弃

父pom引入Discovery

```
1
2 <!--discovery 相关依赖 -->
3 <dependency>
4   <groupId>com.nepxion</groupId>
5   <artifactId>discovery</artifactId>
```

```
6 <version>6.12.1</version>
7 <type>pom</type>
8 <scope>import</scope>
9 </dependency>
```

网关服务接入Discovery

1) 引入依赖

```
1 <!-- 1.注册中心插件 -->
2 <dependency>
3 <groupId>com.nepxion</groupId>
4 <artifactId>discovery-plugin-register-center-starter-nacos</artifactId>
5 </dependency>
6
7 <!-- 2.配置中心插件 -->
8 <dependency>
9 <groupId>com.nepxion</groupId>
10 <artifactId>discovery-plugin-config-center-starter-nacos</artifactId>
11 </dependency>
12
13 <!-- 3.管理中心插件 -->
14 <dependency>
15 <groupId>com.nepxion</groupId>
16 <artifactId>discovery-plugin-admin-center-starter</artifactId>
17 </dependency>
18
19 <!-- 4.网关策略编排插件 -->
20 <dependency>
21 <groupId>com.nepxion</groupId>
22 <artifactId>discovery-plugin-strategy-starter-gateway</artifactId>
23 </dependency>
```

2) application.yml中添加discovery配置

```
1 spring:
2   application:
3     name: tulingmall-gateway
4     strategy: #discovery配置
5     gateway:
6     dynamic:
7     route:
8     enabled: true # 开启网关订阅配置中心的动态路由策略,默认为false
```

```

9  cloud:
10  nacos: #配置nacos注册中心地址
11  discovery:
12  server-addr: 192.168.65.103:8848 #注册中心地址
13  namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618 #环境隔离
14
15  discovery: #discovery配置，设置流量染色元数据
16  metadata:
17  group: discovery-group #组名必须配置

```

根据实际的灰度发布维度和场景，配置染色方式的元数据。

组名必须要配置，版本、区域、环境和可用区根据具体场景选择其中一种或者几种进行配置

spring.cloud.discovery.metadata.group=discovery-guide-group

spring.cloud.discovery.metadata.version=1.0

spring.cloud.discovery.metadata.region=dev

spring.cloud.discovery.metadata.env=env1

spring.cloud.discovery.metadata.zone=zone1

spring.cloud.discovery.metadata.active=true

应用微服务接入Discovery

1) 引入依赖

```

1  <!-- 1.注册中心插件 -->
2  <dependency>
3    <groupId>com.nepxion</groupId>
4    <artifactId>discovery-plugin-register-center-starter-nacos</artifactId>
5  </dependency>
6
7  <!-- 2.配置中心插件 -->
8  <dependency>
9    <groupId>com.nepxion</groupId>
10   <artifactId>discovery-plugin-config-center-starter-nacos</artifactId>
11 </dependency>
12
13 <!-- 3.管理中心插件 -->
14 <dependency>
15   <groupId>com.nepxion</groupId>
16   <artifactId>discovery-plugin-admin-center-starter</artifactId>
17 </dependency>
18
19 <!-- 4.服务策略编排插件 -->

```

```

20 <dependency>
21   <groupId>com.nepxion</groupId>
22   <artifactId>discovery-plugin-strategy-starter-service</artifactId>
23 </dependency>

```

2) application.yml中添加discovery配置

```

1 spring:
2   main:
3     allow-bean-definition-overriding: true
4   application:
5     name: tulingmall-member
6   cloud:
7     nacos:
8     discovery:
9       server-addr: 192.168.65.103:8848
10      namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618
11
12      discovery: #discovery配置，设置流量染色元数据
13      metadata:
14        group: discovery-group #组名必须配置
15        version: 1.0 #指定版本号

```

全链路版本权重灰度发布

在nacos配置中心中增加网关的版本权重灰度发布策略，Group为discovery-group，Data Id为tulingmall-gateway，策略内容如下，实现从网关发起的调用全链路1.0版本流量权重为90%，1.1版本流量权重为10%

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <rule>
3   <strategy>
4     <version-weight>1.0=90;1.1=10</version-weight>
5   </strategy>
6 </rule>

```

也可以指定具体每个微服务的版本权重

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <rule>
3   <strategy>
4     <version-weight>{"tulingmall-member":"1.0=90;1.1=10", "tulingmall-promotion":"1.0=90;1.1=10"}</version-weight>
5   </strategy>

```


6 </rule>

* Data ID: 网关微服务名

* Group: discovery指定的Group

[更多高级选项](#)

描述:

Beta发布: ☐ 默认不要勾选。

配置格式: ☐ TEXT ☐ JSON ☒ XML ☐ YAML ☐ HTML ☐ Properties 灰度发布规则

配置内容?:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <rule>
3   <strategy>
4     <version-weight>1.0=90;1.1=10</version-weight>
5   </strategy>
6 </rule>
```

结合课堂代码进行测试

全链路版本条件权重灰度发布

指定百分比流量分配

- 灰度路由，即服务a和b 1.1版本被调用到的概率为5%
- 稳定路由，即服务a和b 1.0版本被调用到的概率为95%

在nacos配置中心修改网关控制的灰度发布策略

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <rule>
3   <strategy-release>
4     <conditions type="gray">
5       <condition id="gray-condition" version-id="gray-route=5;stable-
6         route=95"/>
7     </conditions>
8   <routes>
9     <route id="gray-route" type="version">{"tulingmall-member":"1.1", "tulin
10      gmall-promotion":"1.1"}</route>
11     <route id="stable-route" type="version">{"tulingmall-member":"1.0", "tu
12      lingmall-promotion":"1.0"}</route>
13   </routes>
```

```
12 </strategy-release>
13 </rule>
```

结合课堂代码进行测试

根据前端传递的Header参数动态选择百分比流量分配

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <rule>
3   <strategy-release>
4     <conditions type="gray">
5       <!-- 灰度路由1，条件expression驱动 -->
6       <condition id="gray-condition-1" expression="#H['a'] == '1'" version-
id="gray-route=10;stable-route=90"/>
7       <!-- 灰度路由2，条件expression驱动 -->
8       <condition id="gray-condition-2" expression="#H['a'] == '1' and #H['b']
== '2'" version-id="gray-route=85;stable-route=15"/>
9       <!-- 兜底路由，无条件expression驱动 -->
10      <condition id="basic-condition" version-id="gray-route=0;stable-route=1
00"/>
11    </conditions>
12
13    <routes>
14      <route id="gray-route" type="version">{"tulingmall-member":"1.1", "tuli
ngmall-promotion":"1.1"}</route>
15      <route id="stable-route" type="version">{"tulingmall-member":"1.0", "tu
lingmall-promotion":"1.0"}</route>
16    </routes>
17  </strategy-release>
18 </rule>
```

3.5 MSE 微服务治理全链路灰度

阿里云MSE服务治理产品就是一款基于Java Agent实现的无侵入式企业生产级服务治理产品，您不需要修改任何一行业务代码，即可拥有不限于全链路灰度的治理能力，并且支持近5年内所有的 Spring Boot、Spring Cloud和Dubbo。

最佳实践：[基于MSE云原生网关实现全链路灰度](#)

全链路灰度作为MSE服务治理专业版中的拳头功能，具备以下六大特点：

- 可通过定制规则引入精细化流量

流量规则 *

框架类型 *

☒ Spring Cloud ☐ Dubbo

Path

HTTP相对路径。例如/a/b,注意严格匹配。留空代表任何路径。 切换为自定义输入

条件模式 *

☒ 同时满足下列条件 ☐ 满足下列任一条件

条件列表 *

参数类型	参数	条件	值	操作
Cookie	name	=	xiaoming	删除

+ 添加新的规则条件

- 全链路隔离流量泳道

- 通过设置流量规则对所需流量进行'染色', '染色'流量会路由到灰度机器。
- 灰度流量携带灰度标往下游传递, 形成灰度专属环境流量泳道, 无灰度环境应用会默认选择未打标的基线环境。

- 端到端的稳定基线环境

未打标的应用属于基线稳定版本的应用, 即稳定的线上环境。当我们将发布对应的灰度版本 代码, 然后可以配置规则定向引入特定的线上流量, 控制灰度代码的风险。

- 流量一键动态切流

流量规则定制后, 可根据需求进行一键停启, 增删改查, 实时生效。灰度引流更便捷。

- 低成本接入, 基于Java Agent技术实现无需修改一行业务代码

MSE微服务治理能力基于Java Agent字节码增强的技术实现, 无缝支持市面上近5年的所有Spring Cloud 和Dubbo的版本, 用户不用改一行代码就可以使用, 不需要改变业务的现有架构, 随时可上可下, 没有绑定。只需开启MSE微服务治理专业版, 在线配置, 实时生效。

- 具备无损上下线能力, 使得发布更加丝滑

应用开启MSE微服务治理后就具备无损上下线能力, 大流量下的发布、回滚、扩容、缩容等场景, 均能保证流量无损。