主讲老师： Fox

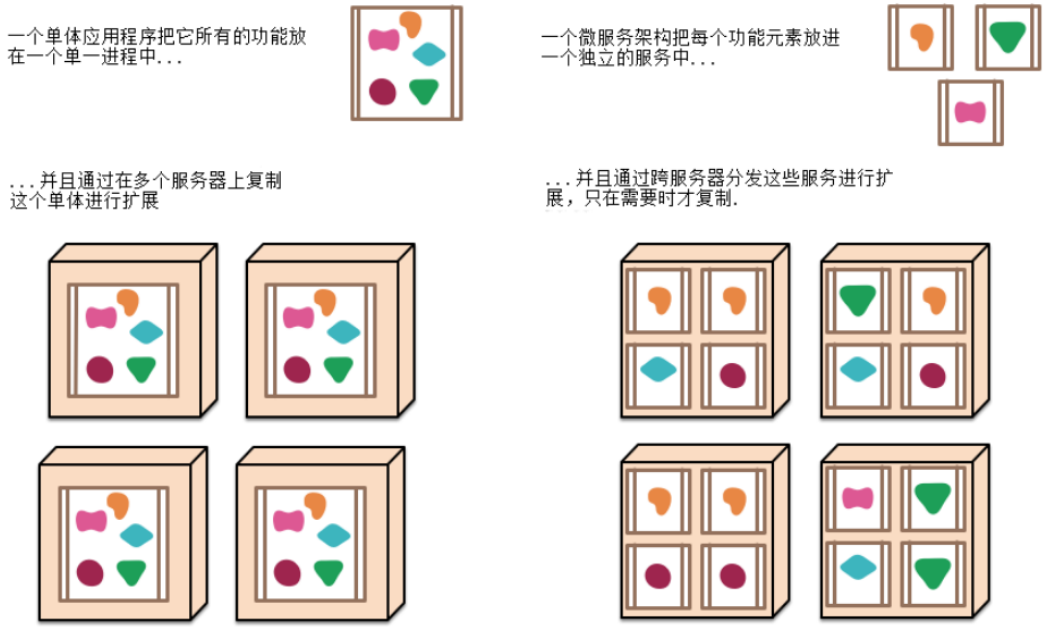
学习本课程前提： 理解微服务，熟悉常用微服务组件的使用，没接触过微服务的同学可以去补一下微服务专题的课程

0. 微服务拆分

微服务介绍

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

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



拆分的目的：将复杂的问题简单化

单体优势

- 公司业务处刚开始阶段
- 开发简单直接，代码和项目集中式管理。
- 排查问题时只需要排查这个应用就可以了，更有针对性。
- 只需要维护一个工程，节省维护系统运行的人力成本

单体劣势

- 伴随着公司业务功能越来越多，开发团队的规模越来越大
- 在技术层面，数据库的连接数成为应用服务器扩容的瓶颈
- 单体架构增加了研发的成本抑制了研发效率的提升
- 单体架构对于系统的运维也会有很大的影响

微服务优缺点出现时机刚好与单体相反

微服务能解决的问题：

- 快速迭代
- 三高

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

			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
资源成本	低	高	吞吐量的下降意味着要增加更多的资源，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

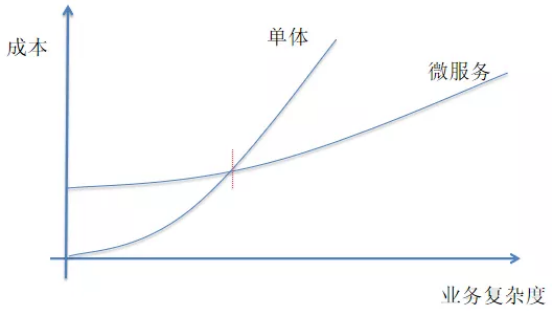
思考：

1. 什么时候使用微服务架构？ 或者微服务拆分的时机是什么？
2. 如何决定服务的拆分粒度？

微服务拆分时机

如下场景是否需要进行微服务拆分？

- 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。

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

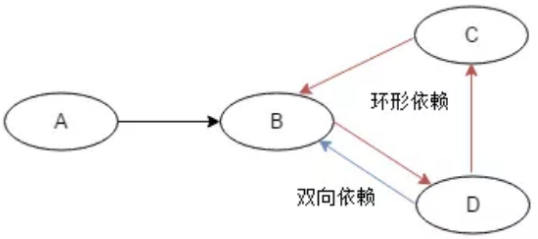
服务自治、接口隔离原则：尽量消除对其他服务的强依赖，这样可以降低沟通成本，提升服务稳定性。服务通过标准的接口隔离，隐藏内部实现细节。这使得服务可以独立开发、测试、部署、运行，以服务为单位持续交付。

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化分清楚或者有通用的功能没有下沉下来。



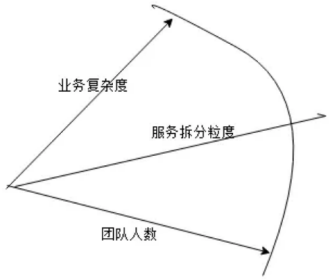
阶段性合并：随着你对业务领域理解的逐渐深入或者业务本身逻辑发生了比较大的变化，亦或者之前的拆分没有考虑的很清楚，导致拆分后的服务边界变得越来越混乱，这时就要重新梳理领域边界，不断纠正拆分的合理性。

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。

拆分粒度控制

思考： 拆分的粒度是不是越细越好?

目前很多传统的单体应用再向微服务架构进行升级改造，如果拆分粒度太细会增加运维复杂度，粒度过大又起不到效果，那么改造过程中如何平衡拆分粒度呢？平衡拆分粒度可以从两方面进行权衡，一是业务发展的复杂度，二是团队规模的人数



人员和服务数量的不匹配，会导致维护成本增加，也会导致服务合并。

前期设计和开发阶段： 3个人负责一个微服务
后期维护阶段：每个微服务可以安排2个人维护，每个人可以维护多个微服务

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

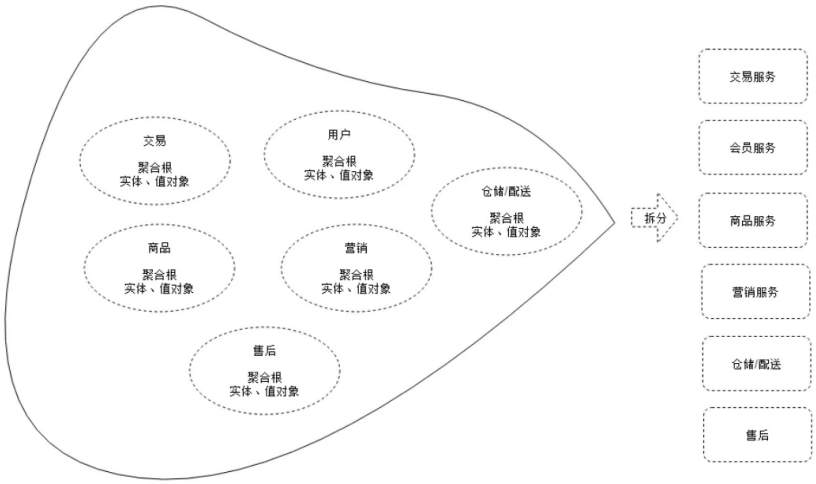
- 基于数据驱动拆分服务： 自下而上的架构设计方法，通过分析需求，确定整体数据结构，根据表之间的关系拆分服务。

  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（如单点登录），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，也就是微服务里面的 API 网关。

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。常见的拆分方式和具体的性能瓶颈有关，例如电商的抢购，性能压力最大的是入口的排队功能，可以将排队功能独立为一个服务。

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 端主要是商详有大量的读取操作，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。

拆分注意的风险

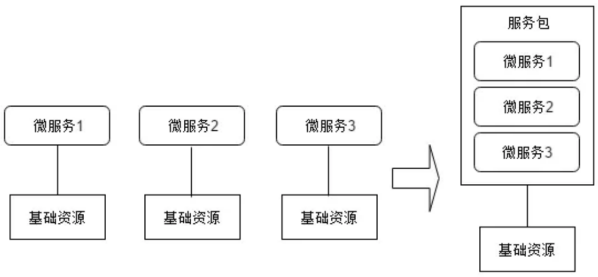
不打无准备之仗：开发团队是否具备足够的经验，能否驾驭微服务的技术栈，可能是第一个需要考虑的点。

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未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

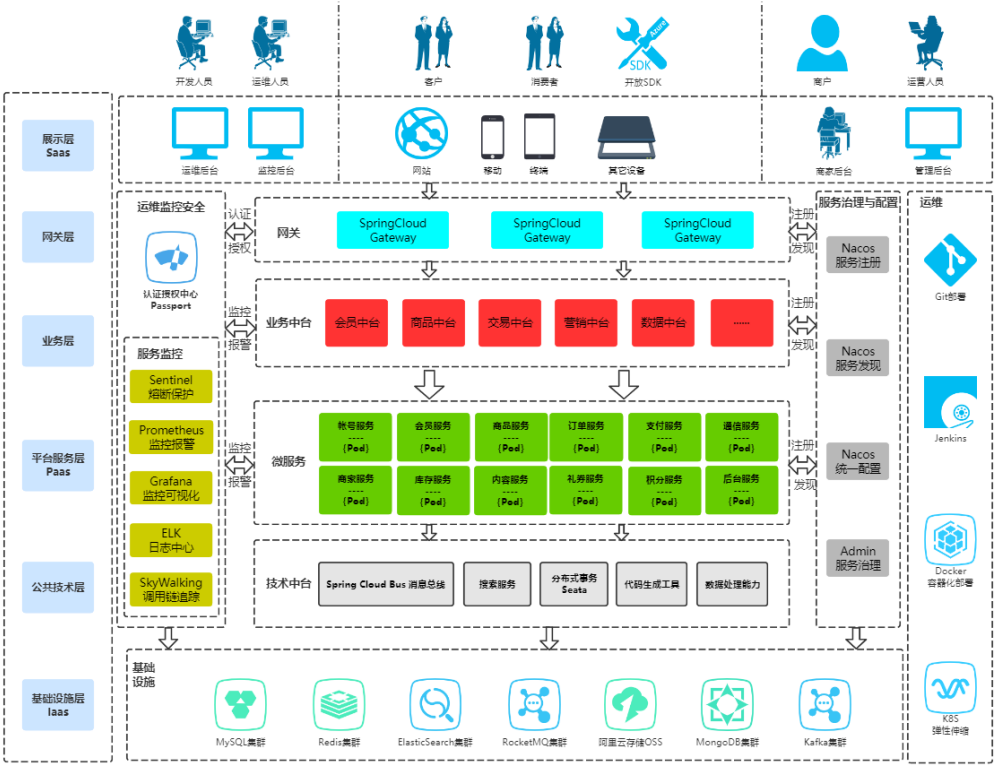
服务只拆不合：

- 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开的时候可以很快就分离



1. 电商项目拆分

电商项目微服务架构： https://www.processon.com/view/link/5e69e768e4b07fc7a6841488



1.1 Spring Cloud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:

Spring Cloud Version	Spring Cloud Alibaba Version	Spring Boot Version
Spring Cloud 2020.0	2020.0.RC1	2.4.2.RELEASE
Spring Cloud Hoxton.SR8	2.2.5.RELEASE	2.3.2.RELEASE
Spring Cloud Greenwich.SR6	2.1.4.RELEASE	2.1.13.RELEASE
Spring Cloud Hoxton.SR3	2.2.1.RELEASE	2.2.5.RELEASE
Spring Cloud Hoxton.RELEASE	2.2.0.RELEASE	2.2.X.RELEASE
Spring Cloud Greenwich	2.1.2.RELEASE	2.1.X.RELEASE
Spring Cloud Finchley	2.0.4.RELEASE	2.0.X.RELEASE
Spring Cloud Edgware	1.5.1.RELEASE(停止维护，建议升级)	1.5.X.RELEASE

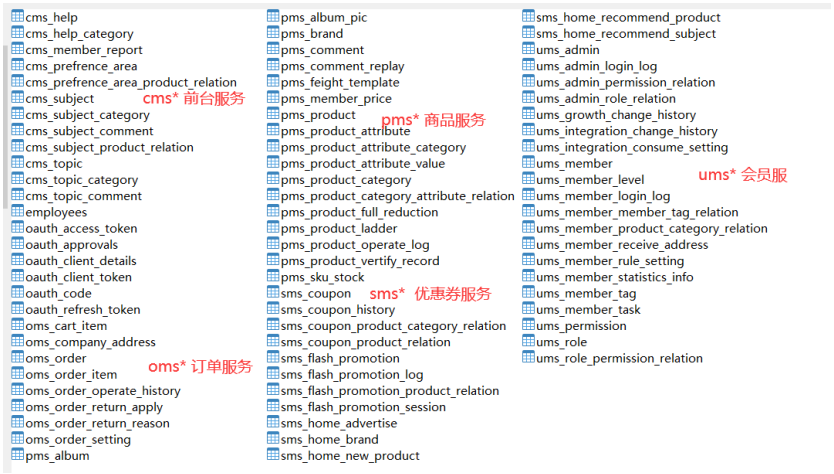
```
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>2.1.7.RELEASE</version>
 <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.tuling</groupId>
<artifactId>tuling-mall</artifactId>
<version>0.0.1-SNAPSHOT</version>
 <name>tuling-mall</name>
 <packaging>pom</packaging>

<properties>
 <spring-cloud.version>Greenwich.SR3</spring-cloud.version>
 <spring-cloud-alibaba.version>2.1.2.RELEASE</spring-cloud-alibaba.version>
</properties>

<dependencyManagement>
 <dependencies>
 <!--Spring Cloud 相关依赖-->
 <dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-dependencies</artifactId>
 <version>${spring-cloud.version}</version>
 <type>pom</type>
 <scope>import</scope>
 </dependency>
 <!--Spring Cloud Alibaba 相关依赖-->
 <dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-alibaba-dependencies</artifactId>
```

```
<version>${spring-cloud-alibaba.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

```

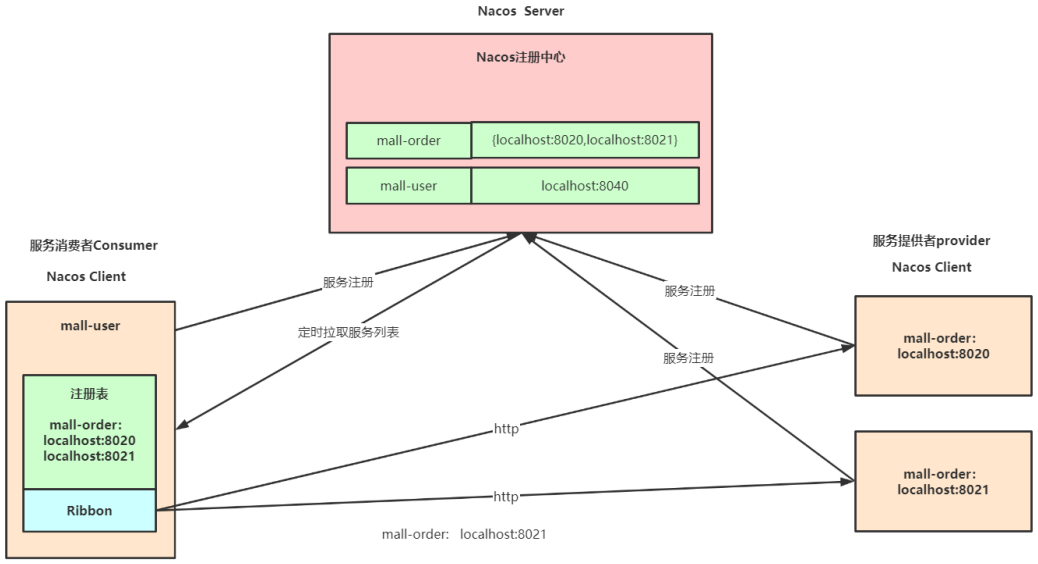
1.2 模块的拆分

名称	服务IP	端口	描述
Mysql数据库	tlshopdb.com	3306	
Nacos注册中心&配置中心	tlshop.com	8848	
tulingmall-authcenter	auth.tuling.com	9999	认证授权中心
tulingmall-gateway	gateway.tuling.com	8888	网关服务
tulingmall-member	member.tuling.com	8877	用户服务
tulingmall-product	product.tuling.com	8866	商品服务
tulingmall-coupons	coupons.tuling.com	8855	优惠券服务
tulingmall-order	order.tuling.com	8844	订单服务
tulingmall-portal		8887	前台服务

1.3 数据库的拆分



1.4 注册中心的配置



将微服务注册到Nacos Server

1.引入依赖

```
<!--nacos 注册中心 -->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2.在yml中配置注册中心地址

```
spring:
 application:
 name: mall-order #微服务的名称
 cloud:
 nacos:
 discovery:
 server-addr: 192.168.65.232:8848 #注册中心地址
 namespace: 80a98d11-492c-4008-85aa-32d889e9b0d0 #环境隔离
```

1.5 服务间的调用实现

使用openfeign作为服务间调用组件

1.引入依赖

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

```
@FeignClient(value = "tulingmall-coupons",path = "/coupon")
public interface CouponsFeignService {

 @RequestMapping(value = "/list", method = RequestMethod.GET)
 @ResponseBody
 CommonResult<List<SmsCouponHistory>> list(@RequestParam(value = "useStatus", required = false) Integer useStatus
 ,@RequestHeader("memberId") Long memberId);

}
```

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

```
@SpringBootApplication
@EnableFeignClients
public class TulingmallMemberApplication {

```

```
 public static void main(String[] args) {
 SpringApplication.run(TulingmallMemberApplication.class, args);
 }

}
```

4. 测试，发起远程服务调用

```
@Autowired
private CouponsFeignService couponsFeignService;

@RequestMapping(value = "/coupons", method = RequestMethod.GET)
public CommonResult getCoupons(@RequestParam(value = "useStatus", required = false) Integer useStatus
 ,@RequestHeader("memberId") Long memberId){
 // 通过openfeign从远程微服务tulingmall-coupons获取优惠券信息
 return couponsFeignService.list(useStatus, memberId);
}
```

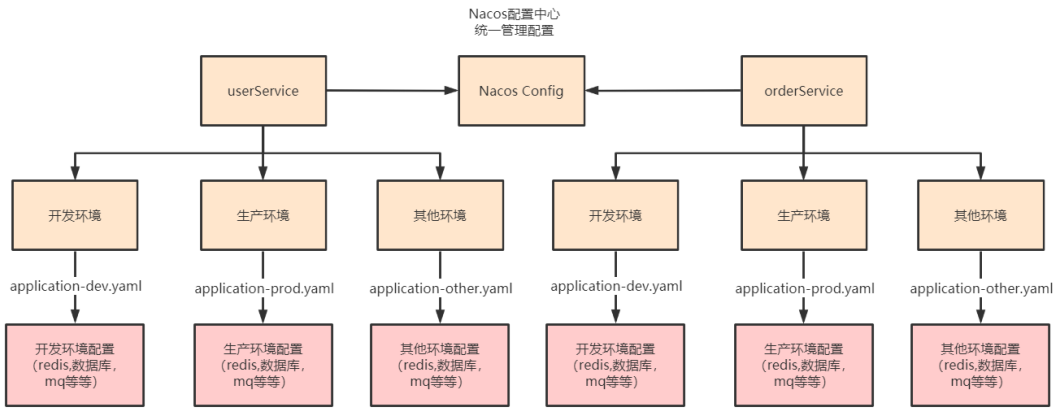
5. 开启openfeign日志配置

```
@Configuration
public class FeignConfig {

 @Bean
 public Logger.Level feignLoggerLevel() {
 return Logger.Level.FULL;
 }

}
```

如果日志不显示，可以在yml中通过logging.level设置日志级别

```
logging:
 level:
 com.tuling: debug
```

1.6 配置中心的配置



使用nacos config作为配置中心

1.引入依赖

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

2. 创建bootstrap.yml文件，添加配置中心的配置

```
spring:
 application:
 name: tulingmall-member #微服务的名称
 cloud:
 nacos:
 config:
```

```
 serverAddr: 192.168.65.232:8848 #配置中心的地址
 namespace: 741b4a7b-c610-4f88-8b83-e9ec87e68319
 # dataid 为 yml 的文件扩展名配置方式
  # `${spring.application.name}.${file-extension:properties}`
  file-extension: yml

  #通用配置
  shared-dataids: nacos.yml,mybatis.yml,actuator.yml,redis.yml,mongodb.yml
  refreshable-dataids: nacos.yml,mybatis.yml,actuator.yml,redis.yml,mongodb.yml

#profile粒度的配置
#`${spring.application.name}-${profile}.${file-extension:properties}`
  profiles:
  active: dev
```

3. 添加微服务配置和公共的配置到nacos配置中心

1.7 网关服务搭建

创建tulingmall-gateway服务

1. 引入依赖

```
<!-- gateway网关 -->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>

<!-- nacos服务注册与发现 -->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
<!-- nacos 配置中心 -->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>

<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

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

2. 编写yml配置文件

application.yml

```yaml
server:
 port: 8888
spring:
 application:
 name: tulingmall-gateway
 #配置nacos注册中心地址
 cloud:
 nacos:
 discovery:
  server-addr: 192.168.65.232:8848 #注册中心地址
  namespace: 80a98d11-492c-4008-85aa-32d889e9b0d0 #环境隔离

 gateway:
 discovery:
 locator:
 # 默认为false，设为true开启通过微服务创建路由的功能，即可以通过微服务名访问服务
 enabled: true
 # 是否开启网关
 enabled: true
 routes:
 - id: tulingmall-member #路由ID，全局唯一
 uri: lb://tulingmall-member
 predicates:
 - Path=/member/**,/sso/**
 - id: tulingmall-coupons
 uri: lb://tulingmall-coupons
 predicates:
 - Path=/coupon/**

logging:
 level:
 org.springframework.cloud.gateway: debug
```

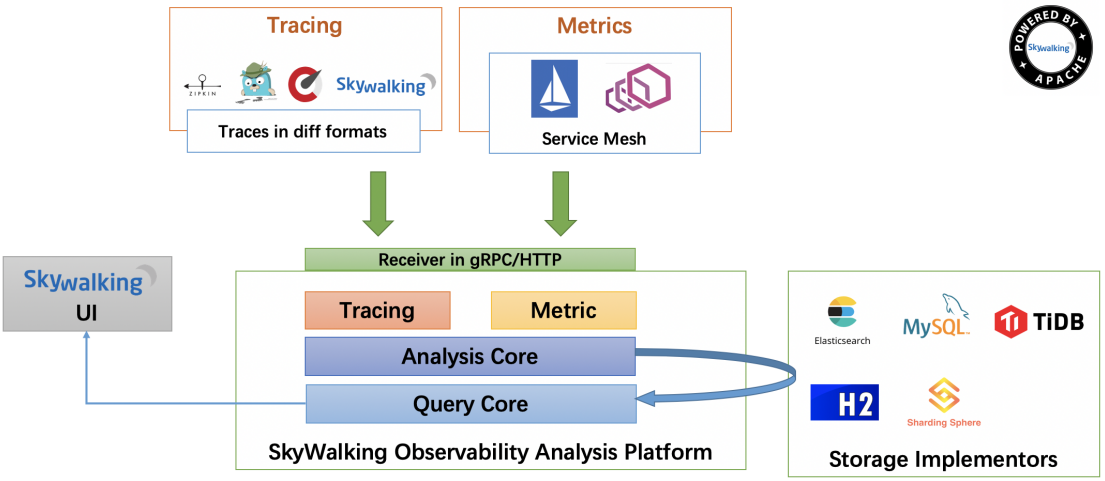
gateway配置移植到配置中心，添加bootstrap.yml：

```yaml
spring:
 application:
 name: tulingmall-gateway #微服务的名称
 cloud:
 nacos:
 config:
 serverAddr: 192.168.65.232:8848 #配置中心的地址
 namespace: 741b4a7b-c610-4f88-8b83-e9ec87e68319
 # dataid 为 yml 的文件扩展名配置方式
 # `${spring.application.name}.${file-extension:properties}`
 file-extension: yml

 #通用配置
 shared-configs[0]:
 data-id: nacos.yml
 refresh: true

 #profile粒度的配置
 #`${spring.application.name}-${profile}.${file-extension:properties}`
 profiles:
 active: dev
```

1.8 微服务接入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

```
-javaagent:D:\apache\apache-skywalking-apm-es7-8.4.0\apache-skywalking-apm-bin-es7\agent\skywalking-agent.jar
-Dskywalking.agent.service_name=tulingmall-gateway
-Dskywalking.collector.backend_service=192.168.3.100:11800
```

访问Skywalking UI： http://192.168.3.100:8080/

3. 集成日志框架，生成traceId

logback官方配置

引入依赖

```
<!-- apm-toolkit-logback-1.x -->
<dependency>
 <groupId>org.apache.skywalking</groupId>
 <artifactId>apm-toolkit-logback-1.x</artifactId>
 <version>8.4.0</version>
</dependency>
```

添加logback日志文件logback-spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

 <!-- 引入 Spring Boot 默认的 logback XML 配置文件 -->
 <include resource="org/springframework/boot/logging/logback/defaults.xml"/>

 <!-- 控制台 Appender -->
 <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
 <!-- 日志的格式化 -->
  <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
  <layout class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.TraceIdPatternLogbackLayout">
  <Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level %tid %t %logger{36}: %msg%n</Pattern>
  </layout>
  </encoder>
  </appender>

  <!-- 从 Spring Boot 配置文件中，读取 spring.application.name 应用名 -->
  <springProperty name="applicationName" scope="context" source="spring.application.name" />


  <!-- 日志文件 Appender -->
  <appender name="file" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <!-- 日志文件的路径 -->
  <file>/logs/tulingmall/${applicationName}.log</file>
  <!--滚动策略，基于时间 + 大小的分包策略 -->
  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
```

```
<fileNamePattern>/logs/tulingmall/${applicationName}-%d{yyyy-MM-dd}-%i.log</fileNamePattern>
<MaxHistory>20</MaxHistory>
<timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
<maxFileSize>500MB</maxFileSize>
</timeBasedFileNamingAndTriggeringPolicy>
</rollingPolicy>
<!-- 日志的格式化 -->
<encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
<layout class="org.apache.skywalking.apm.toolkit.log.logback.v1.x.TraceIdPatternLogbackLayout">
<!-- 日志格式中添加 %tid 即可输出 trace id -->
<Pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level %tid %t %logger{36}: %msg%n</Pattern>
</layout>
</encoder>
</appender>

<!-- 设置 Appender -->
<root level="DEBUG">
<appender-ref ref="console"/>
<appender-ref ref="file"/>
</root>

</configuration>
```

整合ELK

4. 启动FileBeats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

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

启动FileBeats

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

5. 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.f
>	2021-06-17 15:19:59.237	ERROR	812f0a005c1245c0l 2cb0adcc754.91.16239143992210013	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:\l \tul log
>	2021-06-17 15:19:59.233	ERROR	812f0a005c1245c0b6a5b 2cb0adcc754.91.16239143992210013	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] with root cause java.lang.IllegalArgumentException: 非法参数异常 at com.tuling.tulingmall.controller.UmsCouponController.list$original$EjUHD4lb(UmsCouponController.java:54) at com.tuling.tulingmall.controller.UmsCouponController.list$original$EjUHD4l	f:\l \tul s.lo

```
input {
```

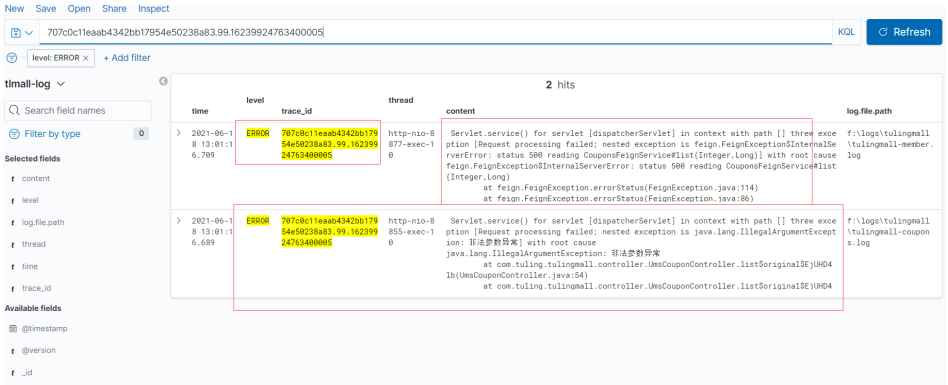
```
  beats {
  port => 5044
  codec => "json"
  }
}

filter {
  grok {
   match => {
   "message" => "(?<time>\d{4}\-\d{2}\-\d{2}\s\d{2}\:\d{2}\:\d{2}\.\d{3})\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}"
   }
   }
   mutate {
   remove_field => "message" # 删除原始日志内容节省存储和带宽
   }
}

output {
  elasticsearch {
   hosts => ["192.168.65.220:9200"]
   index => "tlmall-log" # ES 重建立索引
   }
}


```

启动Logstash

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

6. 在kibana中根据trace_id搜索对应的系统日志

http://192.168.65.220:5601/app/kibana#/home



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。

课后作业：

实现徐庶老师的电商项目单体版的微服务拆分。

文档：01 电商项目微服务架构拆分.note

链接：http://note.youdao.com/noteshare?id=c491404ad8f61122e4ff5f518b1ec68f&sub=73DEBEE3BF4248EF9DABFA59C2795158