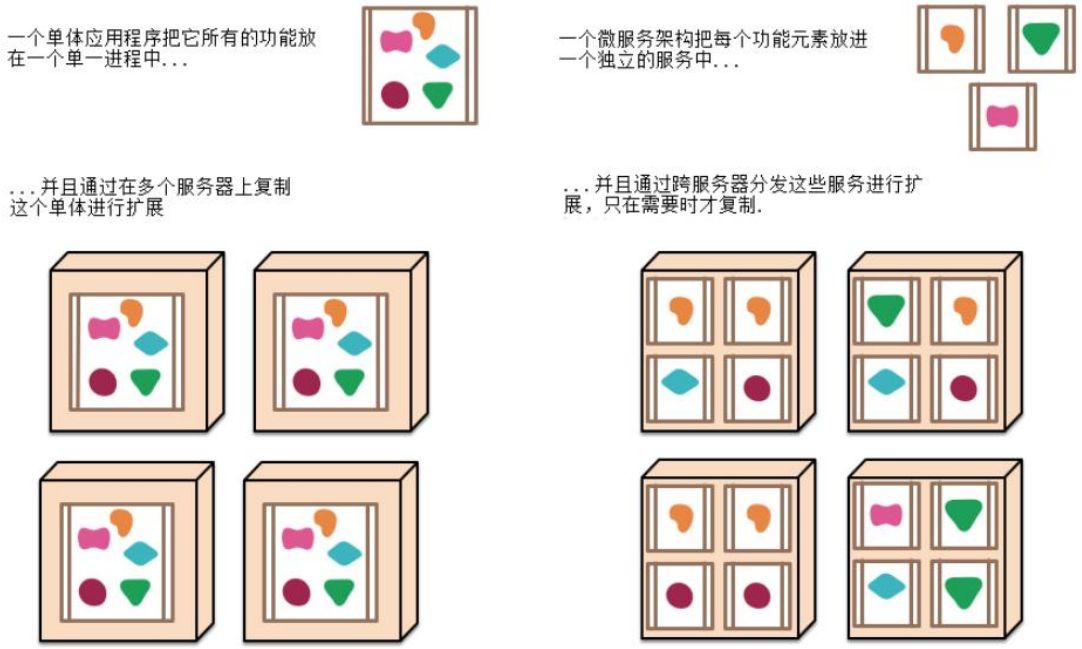
主讲老师： Fox 老师
有道笔记地址：https://note.youdao.com/s/BkIo32M0

1. 微服务架构拆分设计

1.1 微服务架构

微服务架构风格是一种将一个单一应用程序开发为一组小型服务的方法，每个服务运行在自己的进程中，服务间通信采用轻量级通信机制(通常用 HTTP 资源 API)。这些服务围绕业务能力构建并且可通过全自动部署机制独立部署。这些服务共用一个最小型的集中式的管理，服务可用不同的语言开发，使用不同的数据存储技术。
英文:https://martinfowler.com/articles/microservices.html
中文:http://blog.cuicc.com/blog/2015/07/22/microservices



单体架构 vs 微服务架构

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

1.2 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

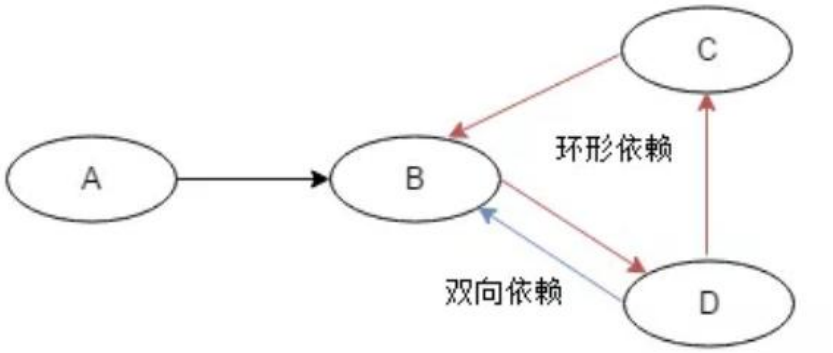
服务自治、接口隔离原则：尽量消除对其他服务的强依赖，这样可以降低沟通成本，提升服务稳定性。服务通过标准的接口隔离，隐藏内部实现细节。这使得服务可以独立开发、测试、部署、运行，以服务为单位持续交付。

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。

1.3 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

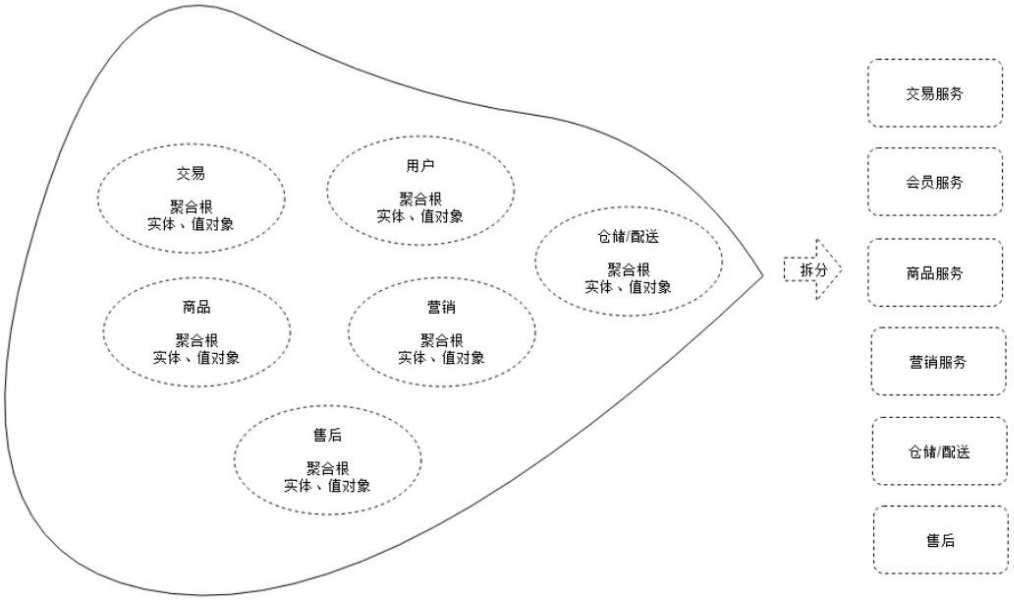
基于数据驱动拆分服务： 自下而上的架构设计方法，通过分析需求，确定整体数据结构，根据表之间的关系拆分服务。

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。



互联网未来架构之道 DDD 领域驱动设计

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。

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。

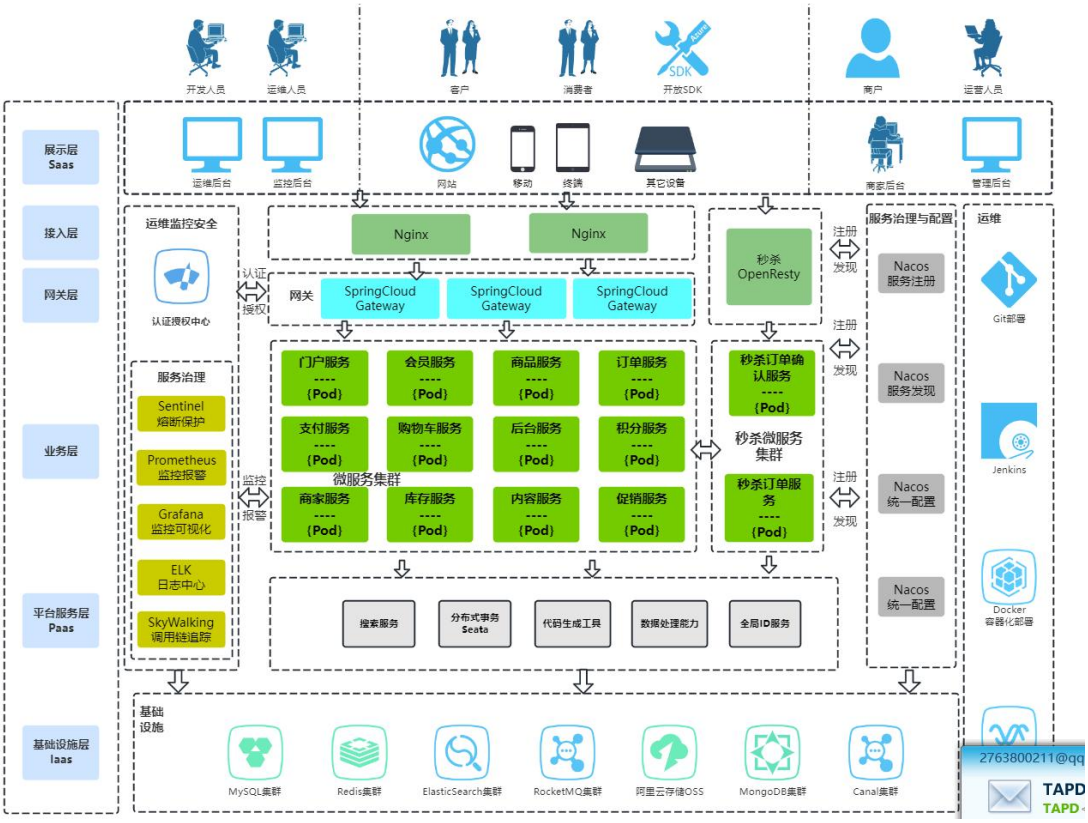
异构性

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。

2. 电商项目微服务拆分实战

2.1 电商项目微服务架构图

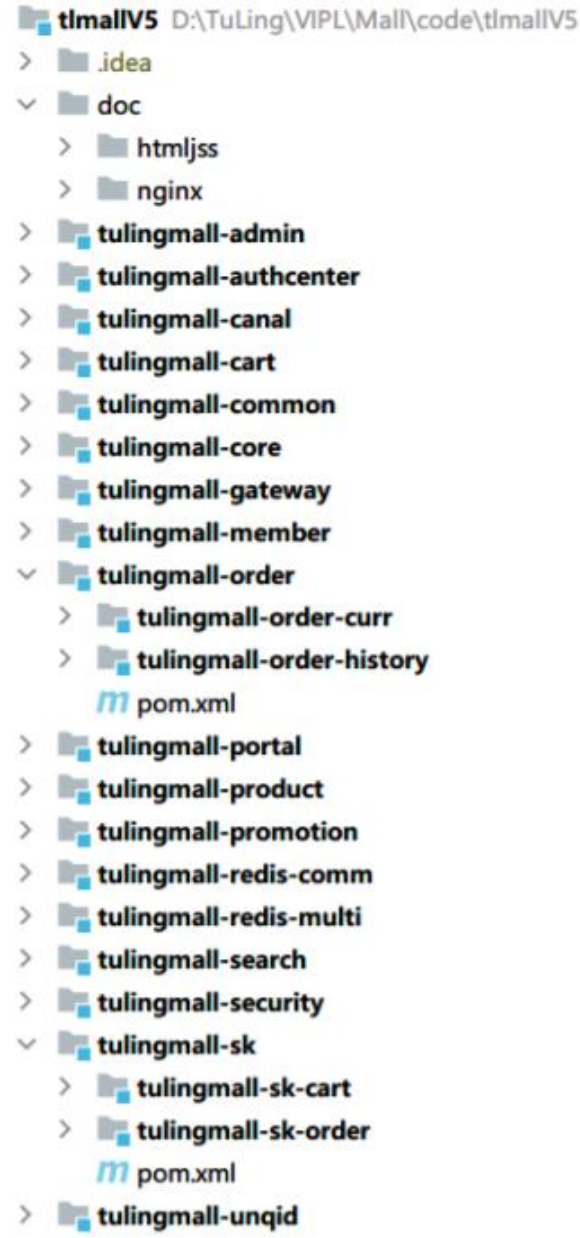
五期项目的各个模块和中间件组合起来，发挥着各自的作用，组成了如下图 所示的架构图：



2.2 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



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

2.3 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

2.4 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:

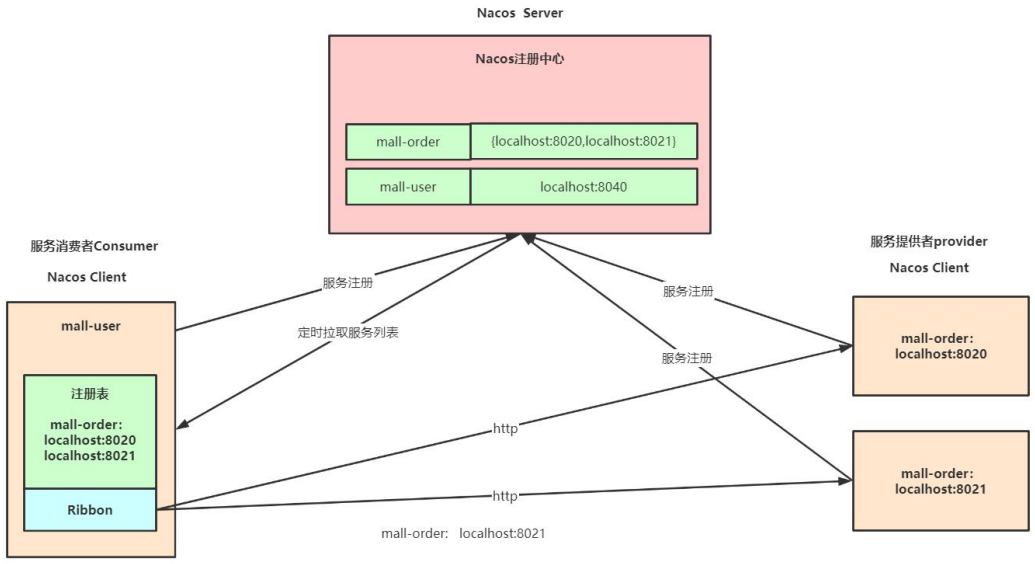
https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version
2.2.9.RELEASE	Spring Cloud Hoxton.SR12	2.3.12.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

2.5 接入 Nacos 注册中心



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   #微服务名
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.65.103:8848      #注册中心地址
        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

```
<!-- 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: 192.168.65.103:8848   #配置中心的地址
        namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618
        # 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: tulingmall-nacos.yml
          group: DEFAULT_GROUP
```

```
            refresh: true
          shared-configs[1]:
            data-id: tulingmall-redis.yml # redis 服务集群配置
            group: DEFAULT_GROUP
            refresh: true

#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

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

```
<!-- 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
```

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

```
@Slf4j
public class HeaderInterceptor implements RequestInterceptor {
    @Override
    public void apply(RequestTemplate template) {

        ServletRequestAttributes attributes = (ServletRequestAttributes)
```

```
RequestContextHolder.getRequestAttributes();
            if(null != attributes){
                HttpServletRequest request = attributes.getRequest();
                log.info("从 Request 中解析请求头");
                template.header("memberId",request.getHeader("memberId"));
            }
        }
}

# FeignConfig.java 中添加拦截器配置
@Bean
public RequestInterceptor requestInterceptor() {
        return new HeaderInterceptor();
}
```

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

```
<!-- 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

```
server:
  port: 8888
spring:
  application:
    name: tulingmall-gateway
  #配置 nacos 注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.65.103:8848   #注册中心地址
        namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618   #环境隔离

    gateway:
      routes:
      - id: tulingmall-member     #路由 ID，全局唯一
        uri: lb://tulingmall-member
        predicates:
        - Path=/member/**,/sso/**
      - id: tulingmall-promotion
        uri: lb://tulingmall-promotion
        predicates:
        - Path=/coupon/**

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

3）gateway 配置移植到配置中心，添加 bootstrap.yml ：

```
spring:
  application:
    name: tulingmall-gateway   #微服务的名称
  cloud:
    nacos:
      config:
        serverAddr: 192.168.65.103:8848   #配置中心的地址
        namespace: 6cd8d896-4d19-4e33-9840-26e4bee9a618
        # 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: tulingmall-nacos.yml
            group: DEFAULT_GROUP
            refresh: true

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