微服务相关面试题

1 Springboot

1.1 讲一讲SpringBoot自动装配的原理

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

嗯，好的，它是这样的。

在Spring Boot项目中的引导类上有一个注解@SpringBootApplication，这个注解是对三个注解进行了封装，分别是：

- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan

其中`@EnableAutoConfiguration`是实现自动化配置的核心注解。

该注解通过`@Import`注解导入对应的配置选择器。关键的是内部就是读取了该项目和该项目引用的Jar包的的classpath路径下**META-INF/spring.factories**文件中的所配置的类的全类名。

在这些配置类中所定义的Bean会根据条件注解所**指定的条件来决定**是否需要将其导入到Spring容器中。

一般条件判断会有像`@ConditionalOnClass`这样的注解，判断是否有对应的class文件，如果有则加载该类，把这个配置类的所有的Bean放入spring容器中使用。

1.2 讲一讲SpringBoot启动流程

> 难易程度：☆☆☆

> 出现频率：☆☆☆

springboot项目在启动的时候, 首先会执行启动引导类里面的

`SpringApplication.run(AdminApplication.class, args)` 方法

```
@SpringBootApplication
@MapperScan("com.heima.admin.dao")
public class AdminApplication {

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

这个run方法主要做的事情可以分为三个部分：

第一部分进行SpringApplication的初始化模块，配置一些基本的环境变量、资源、构造器、监听器

第二部分实现了应用具体的启动方案，包括启动流程的监听模块、加载配置环境模块、及核心的创建上下文环境模块

第三部分是自动化配置模块，该模块作为springboot自动配置核心

1.3 你们常用的SpringBoot起步依赖有哪些

> 难易程度：☆☆
>
> 出现频率：☆☆☆☆

1.4 springBoot支持的配置文件有哪些？加载顺序是什么样的

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

springboot项目支持很多的配置文件，在源码可以看到

```
<resources>
  <resource>
    <directory>${basedir}/src/main/resources</directory>
    <filtering>true</filtering>
    <includes>
      <include>**/application*.yml</include>
      <include>**/application*.yaml</include>
      <include>**/application*.properties</include>
    </includes>
  </resource>
```

它会按照这个不同类型的文件从上到下的顺序进行加载，也就说，会先加载application.yml然后加载application.properties文件。如果有相同的配置，先加载的会被后加载的文件覆盖

假如在启动项目的时候给了启动参数，则最后生效，会覆盖前面所有相同的配置

```
java  -jar --server.port=8089 xx.jar
```

1.5 运行一个SpringBoot项目有哪些方式

> 难易程度：☆☆
>
> 出现频率：☆☆☆

1. 直接使用jar -jar 运行
2. 开发过程中运行main方法
3. 可以配置插件 , 将springboot项目打war包, 部署到Tomcat中运行
4. 直接用maven插件运行 maven spring-boot：run

1.6 Spring Boot的核心注解是哪个？他由哪几个注解组成的？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

Spring Boot的核心注解是@SpringBootApplication , 他由几个注解组成 :

- @SpringBootConfiguration： 组合了- @Configuration注解，实现配置文件的功能；
- @EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项
- @ComponentScan：Spring组件扫描

1.7 你们项目中使用的SpringBoot是哪个版本？

> 难易程度：☆☆
>
> 出现频率：☆☆

- SpringBoot : 2.3.9.RELEASE
- SpringCloud : Hoxton.SR10
- SpringCloudAlibaba : 2.2.5.RELEASE

1.8 Spring Boot如何定义多套不同环境配置？

> 难易程度：☆☆
>
> 出现频率：☆☆

提供多套配置文件，如：

```
applcation.properties
application-dev.properties
application-test.properties
application-prod.properties
```

然后在applcation.properties文件中指定当前的环境**spring.profiles.active=test**,这时候读取的就是application-test.properties文件。

2 SpringCloud

2.1 什么是微服务?微服务的优缺点是什么?

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

微服务

将单体服务拆分成一组小型服务。拆分完成之后，每个小型服务都运行在独立的进程中。服务与服务之间采用轻量级的通信机制来进行沟通（Spring Cloud 中是基于Http请求）

每一个服务都按照具体的业务进行构建，如电商系统中，订单服务，会员服务，支付服务等。这些拆分出来的服务都是独立的应用服务，可以独立的部署到上产环境中。相互之间不会受影响。所以一个微服务项目就可以根据业务场景进行开发。这在单体类项目中是无法实现的。

优点：松耦合，聚焦单一业务功能，无关开发语言，团队规模降低。在开发中，不需要了解多有业务，只专注于当前功能，便利集中，功能小而精。微服务一个功能受损，对其他功能影响并不是太大，可以快速定位问题。微服务只专注于当前业务逻辑代码，不会和 html、css 或其他界面进行混合。可以灵活搭配技术，独立性比较好。

缺点：随着服务数量增加，管理复杂，部署复杂，服务器需要增多，服务通信和调用压力增大，运维工程师压力增大，人力资源增多，系统依赖增强，数据一致性，性能监控。

2.2 Spring Cloud 5大组件有哪些？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

早期我们一般认为的Spring Cloud五大组件是

- Eureka ：注册中心
- Ribbon ：负载均衡
- Feign ：远程调用
- Hystrix： 服务熔断
- Zuul/Gateway ：网关

随着SpringCloudAlibba在国内兴起，我们项目中使用了一些阿里巴巴的组件

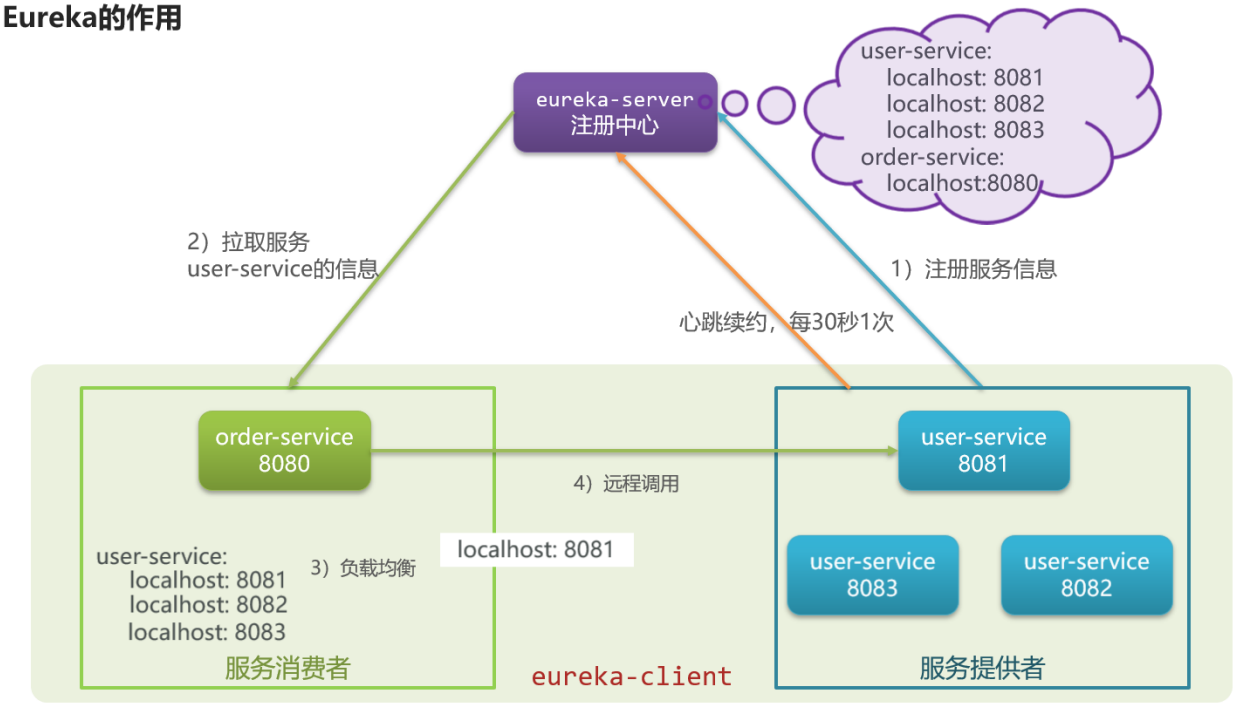
- 注册中心/配置中心 Nacos
- 负载均衡 Ribbon
- 服务调用 Feign
- 服务保护 sentinel
- 服务网关 Gateway

2.3 服务注册和发现是什么意思？Spring Cloud 如何实现服务注册发现？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

各种注册中心组件的原理和流程其实大体上类似

Eureka的作用



核心的功能就一下几个：

1. 服务注册：服务启动的时候会将服务的信息注册到注册中心，比如：服务名称，服务的IP，端口号等
2. 服务发现：服务调用方调用服务的时候，根据服务名称从注册中心拉取服务列表，然后根据负载均衡策略，选择一个服务，获取服务的IP和端口号，发起远程调用
3. 服务状态监控：服务提供者会定时向注册中心发送心跳，注册中心也会主动向服务提供者发送心跳探测，如果长时间没有接收到心跳，就将服务实例从注册中心下线或者移除

使用的话，首先需要部署注册中心服务，然后在我们自己的微服务中引入注册中心依赖，然后再配置文件中配置注册中心地址 就可以了

```yaml
spring:
  application:
    name: leadnews-admin
  cloud:
    nacos:
      # 注册中心地址
      discovery:
        server-addr: 124.221.75.8:8848
      # 配置中心地址
      config:
        server-addr: 124.221.75.8:8848
        file-extension: yml
```

2.4 nacos、eureka的区别？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

① Nacos支持服务端主动检测提供者状态：临时实例采用心跳模式，非临时实例采用主动检测模式

② 临时实例心跳不正常会被剔除，非临时实例则不会被剔除

③ Nacos支持服务列表变更的消息推送模式，服务列表更新更及时

④ Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；

总结：

- Eureka采用AP方式
- naocs默认是AP模式，可以采用CP模式

2.5 你们项目中微服务之间是如何通讯的?

> 难易程度：☆☆
>
> 出现频率：☆☆

1．同步通信：通过Feign发送http请求调用

2．异步：消息队列，如RabbitMq、KafKa等

2.6 你们项目负载均衡如何实现的？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

服务调用过程中的负载均衡一般使用SpringCloud的Ribbon 组件实现，Feign的底层已经自动集成了Ribbon，使用起来非常简单

客户端调用的话一般会通过网关, 通过网关实现请求的路由和负载均衡

```
spring:
  cloud:
    gateway:
      routes:
        # 平台管理
        - id: wemedia
          uri: lb://leadnews-wemedia
          predicates:
            - Path=/wemedia/**
          filters:
            - StripPrefix= 1
```

2.7 Ribbon负载均衡策略有哪些？如果想自定义负载均衡策略如何实现？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆

常用的负载均衡算法：

1、RoundRobinRule：简单轮询服务列表来选择服务器

2、AvailabilityFilteringRule：对以下两种服务器进行忽略：

（1）在默认情况下，这台服务器如果3次连接失败，这台服务器就会被设置为“短路”状态。短路状态将持续30秒，如果再次连接失败，短路的持续时间就会几何级地增加。

（2）并发数过高的服务器、如果一个服务器的并发连接数过高，配置了AvailabilityFilteringRule规则的客户端也会将其忽略。并发连接数的上限，可以由客户端的.ActiveConnectionsLimit属性进行配置。

3、WeightedResponseTimeRule： 为每一个服务器赋予一个权重值。服务器响应时间越长，这个服务器的权重就越小。这个规则会随机选择服务器，这个权重值会影响服务器的选择。

4、**ZoneAvoidanceRule**：以区域可用的服务器为基础进行服务器的选择。使用Zone对服务器进行分类，这个Zone可以理解为一个机房、一个机架等。

而后再对Zone内的多个服务做轮询。它是Ribbon默认的负载均衡规则。

5、BestAvailableRule：忽略那些短路的服务器，并选择并发数较低的服务器。

6、RandomRule： 随机选择一个可用的服务器。

7、RetryRule：重试机制的选择逻辑。

默认的负载均衡算法：ZoneAvoidanceRule

自定义负载均衡

如果想要自定义负载均衡，可以自己创建类实现IRule接口，然后再通过配置类或者配置文件配置即可：

通过定义IRule实现可以修改负载均衡规则，有两种方式：

1. 代码方式：在order-service中的OrderApplication类中，定义一个新的IRule：

```
@Bean
public IRule randomRule(){
    return new RandomRule();
}
```

2. 配置文件方式：在order-service的application.yml文件中，添加新的配置也可以修改规则：

```
userservice: # 给某个微服务配置负载均衡规则，这里是userservice服务
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
# 负载均衡规则
```

2.8什么是Spring Cloud Gateway以及在你们的项目中如何去应用该组件的？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

Spring Cloud Gateway：是Spring Cloud中所提供的一个服务网关组件，是整个微服务的统一入口，在服务网关中可以实现请求路由、统一的日志记录，流量监控、权限校验等一系列的相关功能！

项目应用：权限的校验

具体实现思路：使用Spring Cloud Gateway中的全局过滤器拦截请求(GlobalFilter、Order)，从请求头中获取token，然后解析token。如果可以进行正常解析，此时进行放行；如果解析不到直接返回。

2.9 你们项目的配置文件是怎么管理的？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

大部分的固定的配置文件都放在服务本地，一些根据环境不同可能会变化的部分，放到Nacos中

Naocs中主要存放的是各个微服务共享的配置，需要随着需求动态变更的配置。

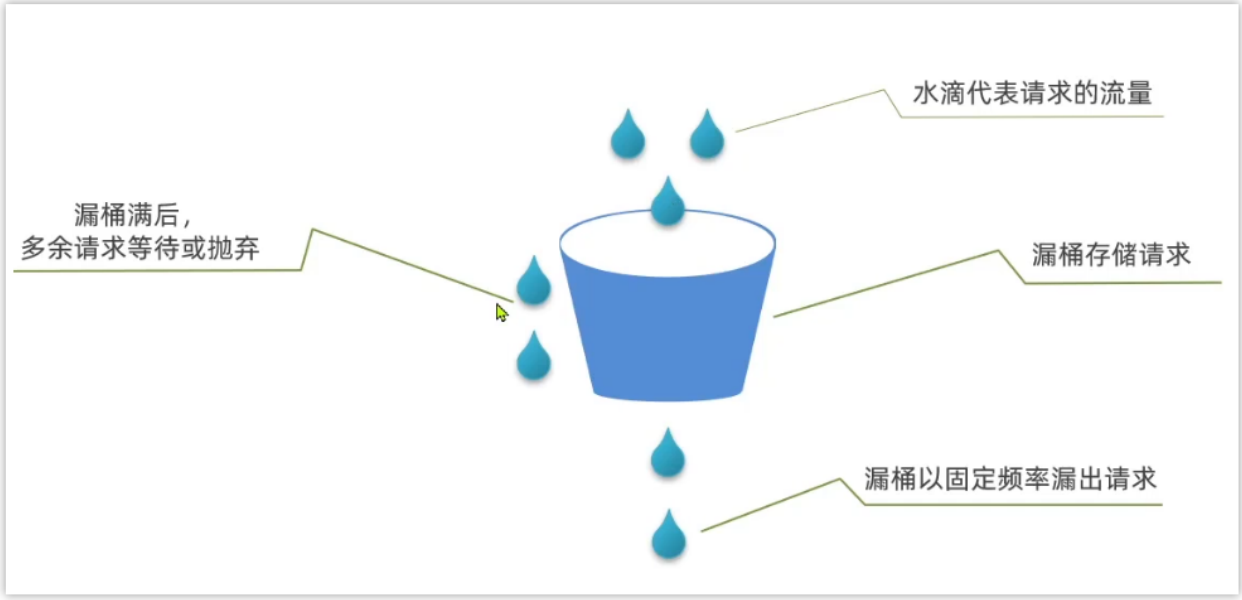
2.10 你们项目中有没有做过限流？怎么做的？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

常见的限流算法：**漏桶算法、令牌桶算法**

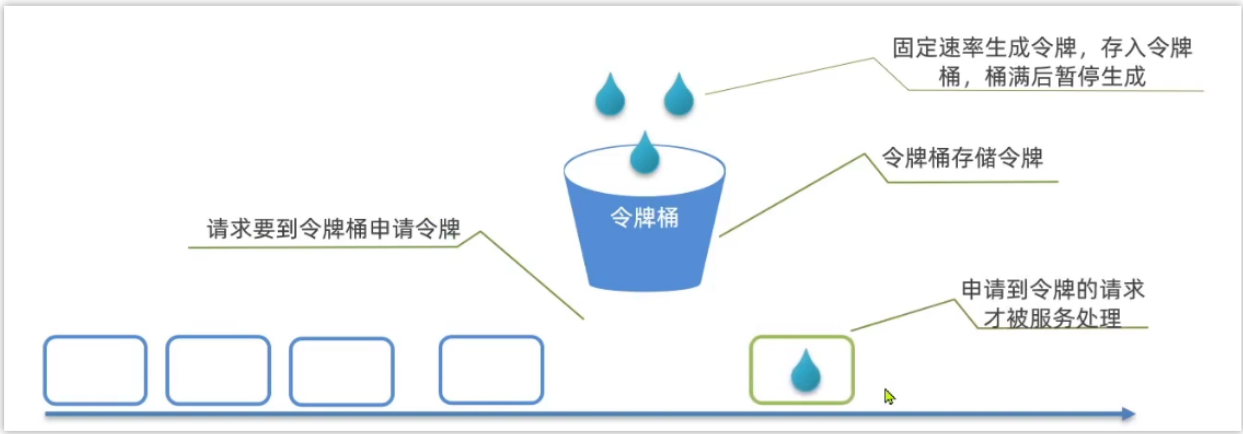
漏桶算法：漏桶算法其实很简单，可以粗略的认为就是注水漏水过程，往桶中以一定速率流出水，以任意速率流入水，当水超过桶流量则丢弃，因为桶容量是不变的，保证了整体的速率。

如下图所示：



令牌桶算法：令牌桶是一个存放固定容量令牌的桶，按照固定速率r往桶里添加令牌；桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃；当一个请求达到时，会尝试从桶中获取令牌；如果有，则继续处理请求；如果没有则排队等待或者直接丢弃；可以发现，漏桶算法的流出速率恒定，而令牌桶算法的流

出速率却有可能大于r；



从作用上来说，漏桶和令牌桶算法最明显的区别就是是否允许突发流量(burst)的处理，漏桶算法能够强行限制数据的实时传输（处理）速率，对突发流量不做额外处理；而令牌桶算法能够在限制数据的平均传输速率的同时允许某种程度的突发传输。

限流算法区别

对比项	令牌桶	漏桶
能否保证流量曲线平滑	基本能，在请求量持续高于令牌生成速度时，流量平滑。但请求量在令牌生成速率上下波动时，无法保证曲线平滑	能，所有请求进入桶内，以恒定速率放行，绝对平滑
能否应对突增流量	能，桶内积累的令牌可以应对突增流量	能，请求可以暂存在桶内
流量控制精确度	高	高

参考回答：

我们项目的流量还是比较大的，我们项目中用的令牌桶算法来进行限流的，在gateway中进行设置。

令牌桶是一个存放固定容量令牌的桶，按照固定速率r往桶里添加令牌；桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃；当一个请求达到时，会尝试从桶中获取令牌；如果有，则继续处理请求；如果没有则排队等待或者直接丢弃；可以发现，漏桶算法的流出速率恒定，而令牌桶算法的流出速率却有可能大于r； 也就说对于突发流量令牌桶也能应付。

具体使用是，在网关路由中进行过滤器配置，可以设置桶的带下，和固定速率。我们通常也会按照用户访问的ip进行限制，这个令牌需要存入redis，所以也需要集成redis使用。

详细使用令牌桶算法限流，详细查看资料中的《网关限流》讲义

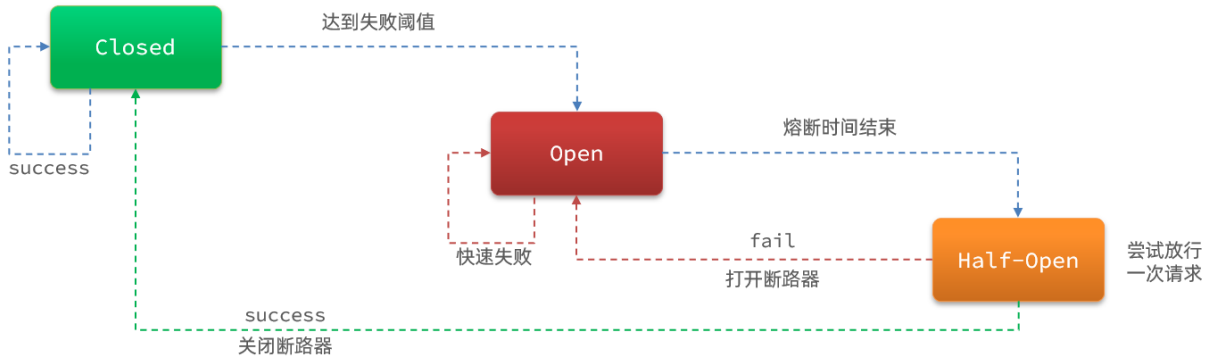
2.11 断路器/熔断器用过嘛？断路器的状态有哪些？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

我们项目中使用Hystrix实现的断路器 ,默认是关闭的，如果需要开启需要在引导类上添加注解：

`@EnableCircuitBreaker`

断路器状态机包括三个状态：



- closed：关闭状态，断路器放行所有请求，并开始统计异常比例、慢请求比例。超过阈值则切换到open状态
  - 请求错误率超过 5%（默认）
- open：打开状态，服务调用被**熔断**，访问被熔断服务的请求会被拒绝，快速失败，直接走降级逻辑。Open状态5秒后（默认值）会进入half-open状态
- half-open：半开状态，放行一次请求，根据执行结果来判断接下来的操作。
  - 请求成功：则切换到closed状态
  - 请求失败：则切换到open状态

2.12 你们项目中有做过服务降级嘛？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

我们项目中涉及到服务调用得地方都会定义降级，一般降级逻辑就是返回默认值，降级的实现也非常简单，就是创建一个类实现 `FallbackFactory` 接口，然后再对应的Feign客户端接口上面，通过@FeignClient指定降级类

2.13 你们的微服务是怎么监控的？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

skywalking

请查看资料中的skywalking的讲义

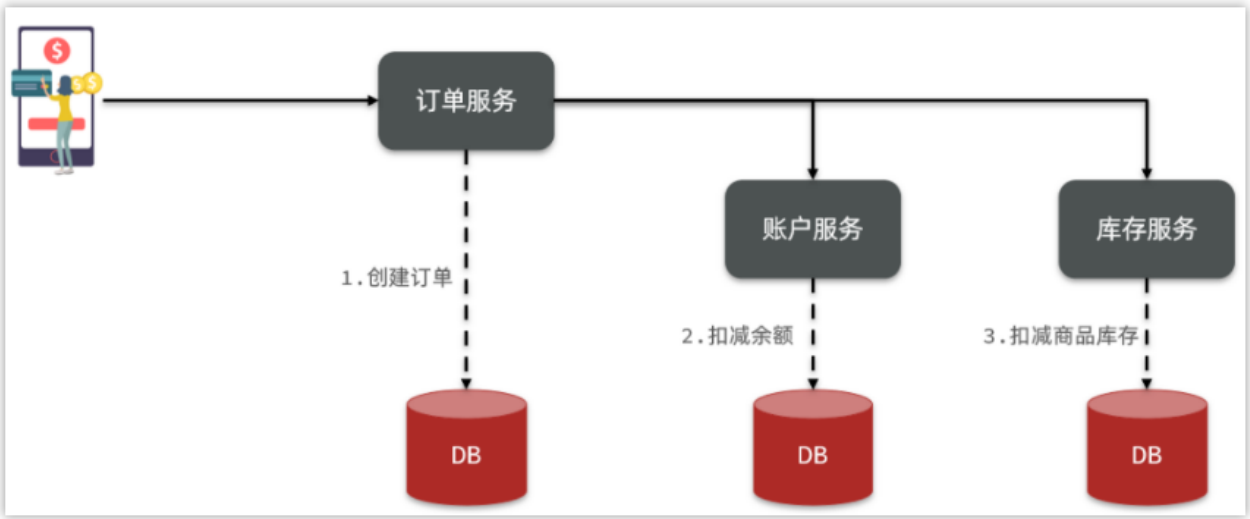
3 分布式事务篇

3.1 什么是分布式事务？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

概述：在分布式系统中，一个业务因为跨越不同数据库或者跨越不同微服务而包含多个子事务，要求所有子事务同时成功或失败，这就是分布式事务。

如下所示：



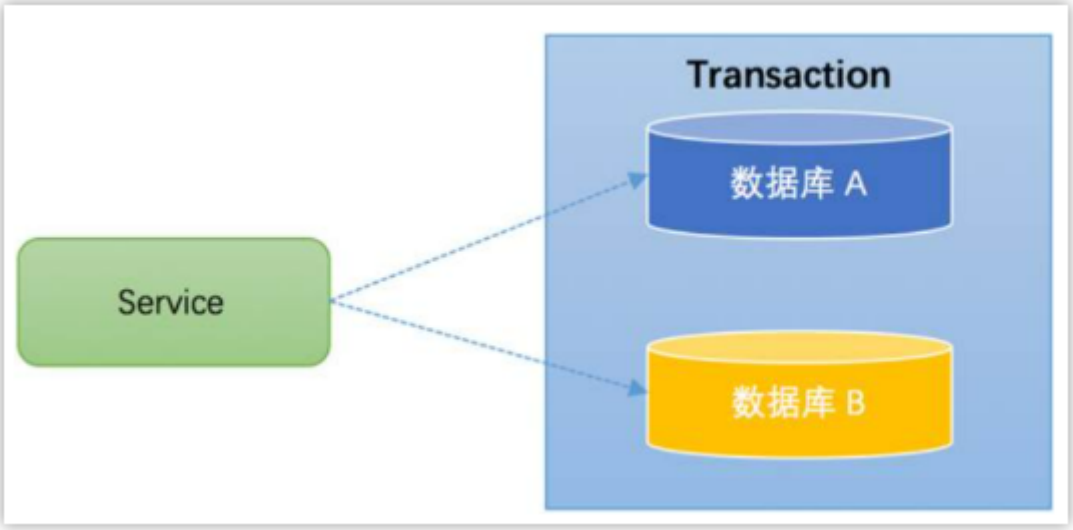
某电商系统的下单操作，需要请求三个服务来完成，这三个服务分别是：订单服务，账户服务，库存服务。当订单生成完毕以后，就需要分别请求账户服务和库存服务进行进行账户余额的扣减和库存扣减。假设都扣减成功了，此时在执行下单的后续操作时出现了问题，那么订单数据库就进行事务回滚，订单生成失败，而账户余额和扣减则都扣减成功了。这就出现了问题，而分布式事务就是解决上述这种不一致问题的。

3.2 哪些场景下都会产生分布式事务？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

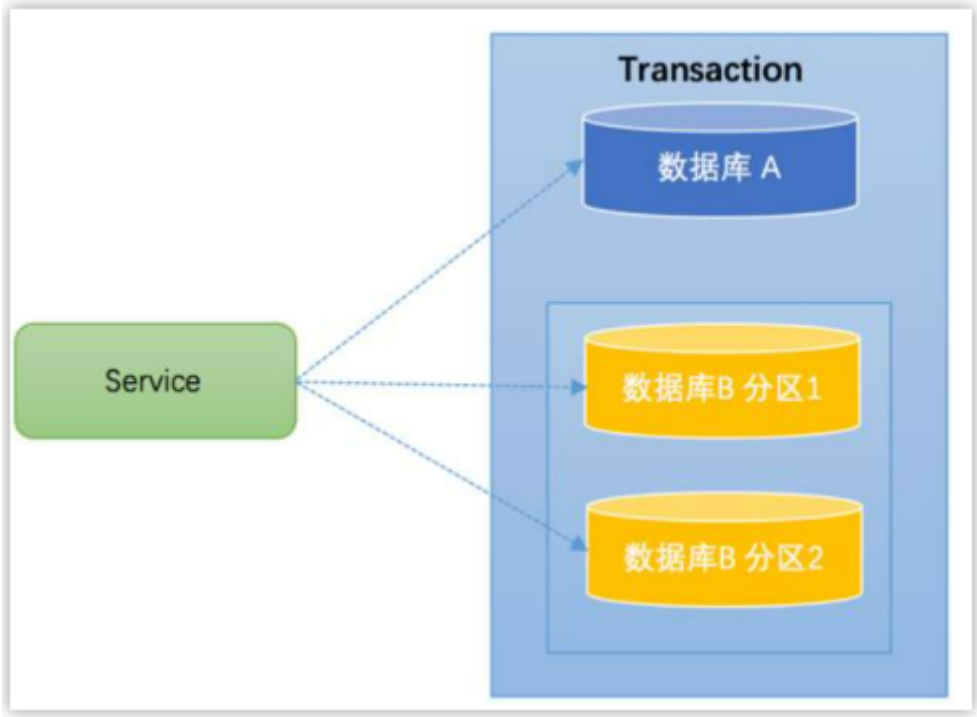
场景1：跨库事务

跨库事务指的是，一个应用某个功能需要操作多个库，不同的库中存储不同的业务数据。如下所示：



场景二：分库分表

通常一个库数据量比较大或者预期未来的数据量比较大，都会进行水平拆分，也就是分库分表。如下图，将数据库B拆分成了2个库：



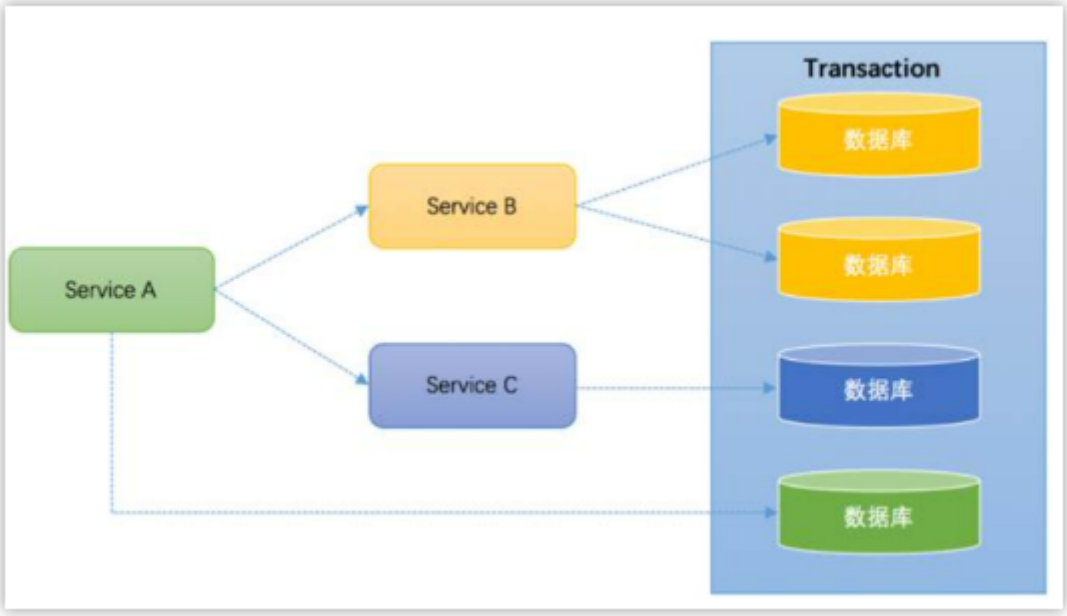
对于分库分表的情况，一般开发人员都会使用一些数据库中间件来降低sql操作的复杂性。

如，对于sql：insert into user(id,name) values (1,"tianshouzhi"), (2,"wangxiaoxiao")。这条sql是操作单库的语法，单库情况下，可以保证事务的一致性。

但是由于现在进行了分库分表，开发人员希望将1号记录插入分库1，2号记录插入分库2。所以数据库中间件要将其改写为2条sql，分别插入两个不同的分库，此时要保证两个库要不都成功，要不都失败，因此基本上所有的数据库中间件都面临着分布式事务的问题。

场景三：跨服务事务

跨服务事务指的是，一个应用某个功能需要调用多个微服务进行实现，不同的微服务操作的是不同的数据库。如下所示：



Service A完成某个功能需要直接操作数据库，同时需要调用Service B和Service C，而Service B又同时操作了2个数据库，Service C也操作了一个库。需要保证这些跨服务的对多个数据库的操作要不都成功，要不都失败，实际上这可能是最典型的分布式事务场景。

3.3 什么是CAP理论？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

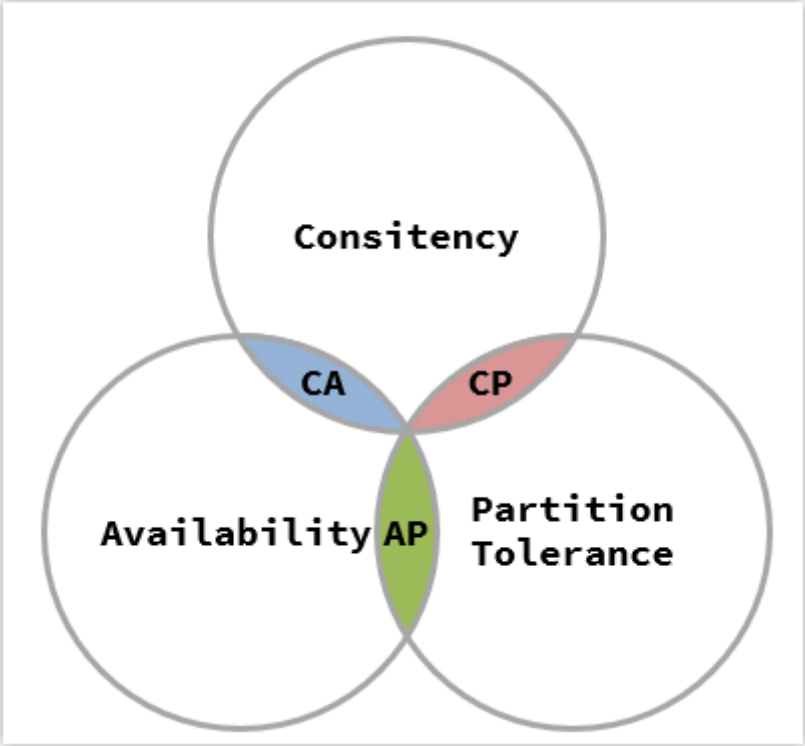
CAP定理是由加州大学伯克利分校Eric Brewer教授提出来的，他指出WEB服务无法同时满足一下3个属性：

1、一致性(Consistency) ： 更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致(强一致性)，不能存在中间状态。

2、可用性(Availability) ： 系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。

3、分区容错性(Partition tolerance) ： 分布式系统在遇到任何网络分区故障时，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

如下所示：



3.4 为什么分布式系统中无法同时保证一致性和可用性？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

对于分布式系统而言，各节点之间一定会存在网络交互，首先网络存在延迟，其次无法100%确保网络的可用，因此可以认为分区网络故障不可避免。
在此条件下，如果要保证各节点的一致性，就必须在一个节点数据变更时，将数据同步给另一个节点，同时，在数据同步过程中，被同步的节点是不能对外提供服务的，否则就会出现数据不一致。而节点不可对外提供服务，就违背了可用性。
所以，在存在系统分区的场景下，可用性和一致性无法同时满足

3.5 什么是BASE理论？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

CAP是分布式系统设计理论，BASE是CAP理论中AP方案的延伸，核心思想是即使无法做到强一致性（StrongConsistency，CAP的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。它的思想包含三方面：

1、Basically Available（基本可用）：基本可用是指分布式系统在出现不可预知的故障的时候，允许损失部分可用性，但不等于系统不可用。

2、Soft state（软状态）：即是指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

3、Eventually consistent（最终一致性）：强调系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。其本质是需要系统

保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

3.6 分布式事务的常见的解决方案有哪些？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

方案一：2PC

两阶段提交又称2PC，2PC是一个非常经典的强一致、中心化的原子提交协议 。

中心化是指协议中有两类节点：一个是中心化协调者节点 （coordinator）和 N 个参与者节点 （partcipant）。

两个阶段 ：

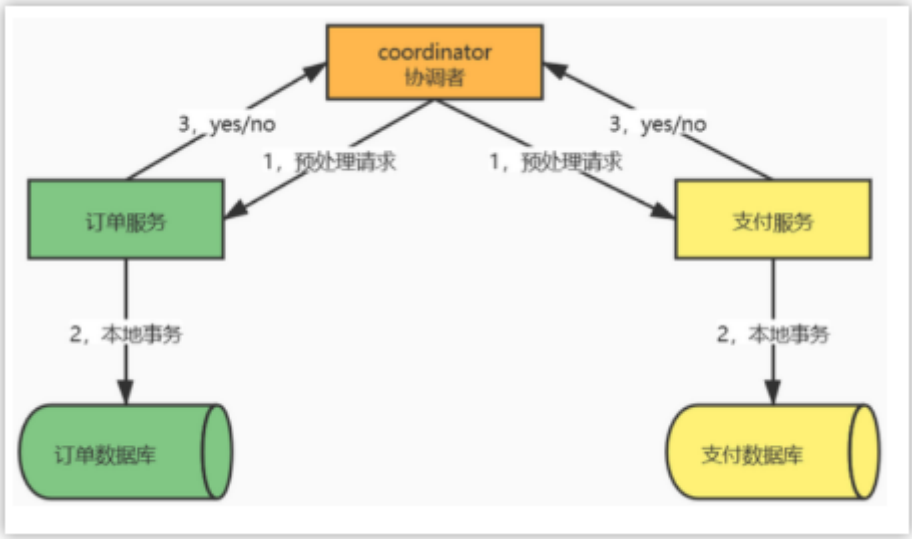
1、第一阶段：投票阶段

2、第二阶段：提交/执行阶段。

举例订单服务A，需要调用支付服务B 去支付，支付成功则处理订单状态为待发货状态，否则就需要将购物订单处理为失败状态。 那么看2PC阶段是如何处

理的。

阶段一：



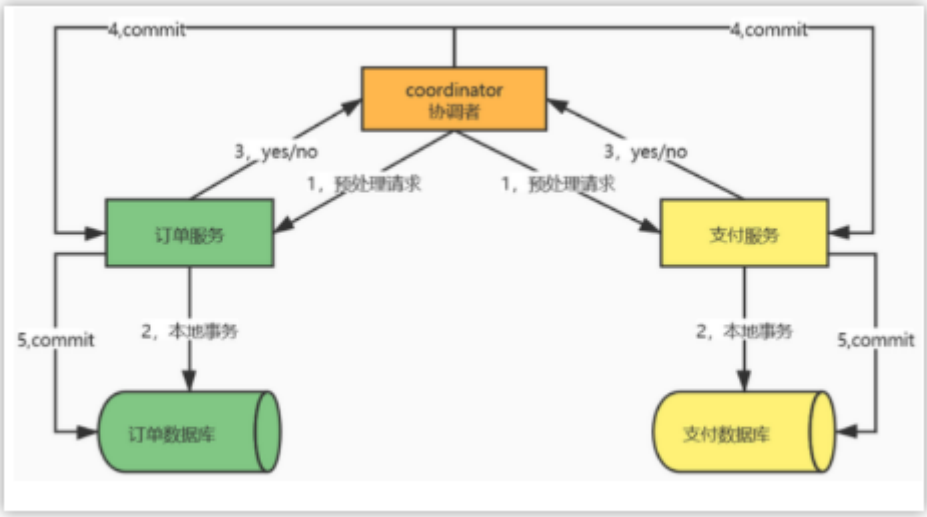
阶段一执行流程：

1、事务询问协调者向所有的参与者发送事务预处理请求，称之为Prepare，并开始等待各参与者的响应。

2、执行本地事务各个参与者节点执行本地事务操作，但在执行完成后并不会真正提交数据库本地事务，而是先向协调者报告说：“我这边可以处理了/我这边不能处理”。

3、各参与者向协调者反馈事务询问的响应如果参与者成功执行了事务操作,那么就反馈给协调者Yes响应,表示事务可以执行,如果没有参与者成功执行事务,那么就反馈给协调者 No 响应,表示事务不可以执行。

阶段二：



阶段二执行流程：

1、所有的参与者反馈给协调者的信息都是Yes,那么就会执行事务提交协调者向所有参与者节点发出Commit请求

2、事务提交参与者收到Commit请求之后,就会正式执行本地事务Commit操作,并在完成提交之后释放整个事务执行期间占用的事务资源。

方案二：TCC

TCC（Try-Confirm-Cancel）又称补偿事务。其核心思想是："针对每个操作都要注册一个与其对应的确认和补偿（撤销操作）"。

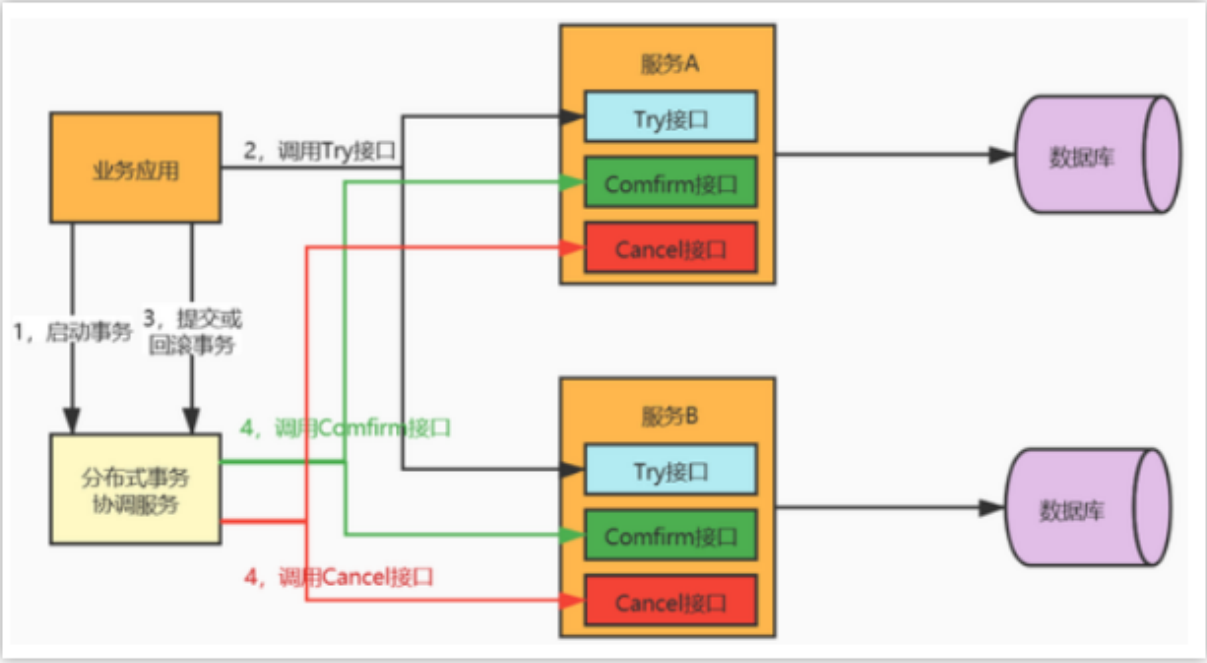
它分为三个操作：

1、Try阶段：主要是对业务系统做检测及资源预留。

2、Confirm阶段：确认执行业务操作。

3、Cancel阶段：取消执行业务操作。

如下所示：

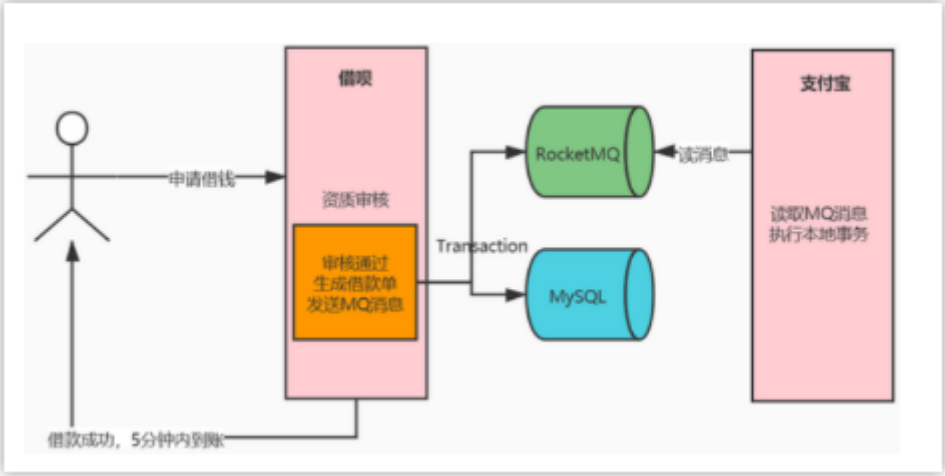


TCC事务的处理流程与2PC两阶段提交类似，不过2PC通常都是在跨库的DB层面，而TCC本质上就是一个应用层面的2PC，需要通过业务逻辑来实现。这种分布式事务的实现方式的优势在于，可以让应用自己定义数据库操作的粒度，使得降低锁冲突、提高吞吐量成为可能。 不足之处则在于对应用的侵入性非常强，业务逻辑的每个分支都需要实现try、confirm、cancel三个操作。此外，其实现难度也比较大，需要按照网络状态、系统故障等不同的失败原因实现不同的回滚策略。为了满足一致性的要求，confirm和cancel接口还必须实现幂等。

方案三：MQ分布式事务

上面的三种分布式事务的解决方案适用于对数据一致性要求很高的场景。如果数据强一致性要求没那么高，可以采用消息中间件（MQ）实现事务最终一致。 在支付系统中，常常使用的分布式事务解决方案就是基于MQ实现的，它对数据强一致性要求没那么高，但要求数据最终一致即可。

例如：向借呗申请借钱，借呗审核通过后支付宝的余额才会增加，但借呗和支付宝有可能不是同一个系统，这时候如何实现事务呢？实现方案如下图：



执行流程如下所示：

1、找花呗借钱

2、花呗借钱审核通过，同步生成借款单

3、借款单生成后，向MQ发送消息，通知支付宝转账

4、支付宝读取MQ消息，并增加账户余额

上图最复杂的其实是如何保障2、3在同一个事务中执行（本地事务和MQ消息发送在同一个事务执行），借款结束后，借呗数据处理就完成了，接下来支付宝才能读到消息，然后执行余额增加，这才完成整个操作。如果中途操作发生异常，例如支付宝余额增加发生问题怎么办？此时需要人工解决，没有特别好的办法，但这种事故概率极低。

3.7 Seata的架构是什么？

> 难易程度：☆☆
>
> 出现频率：☆☆

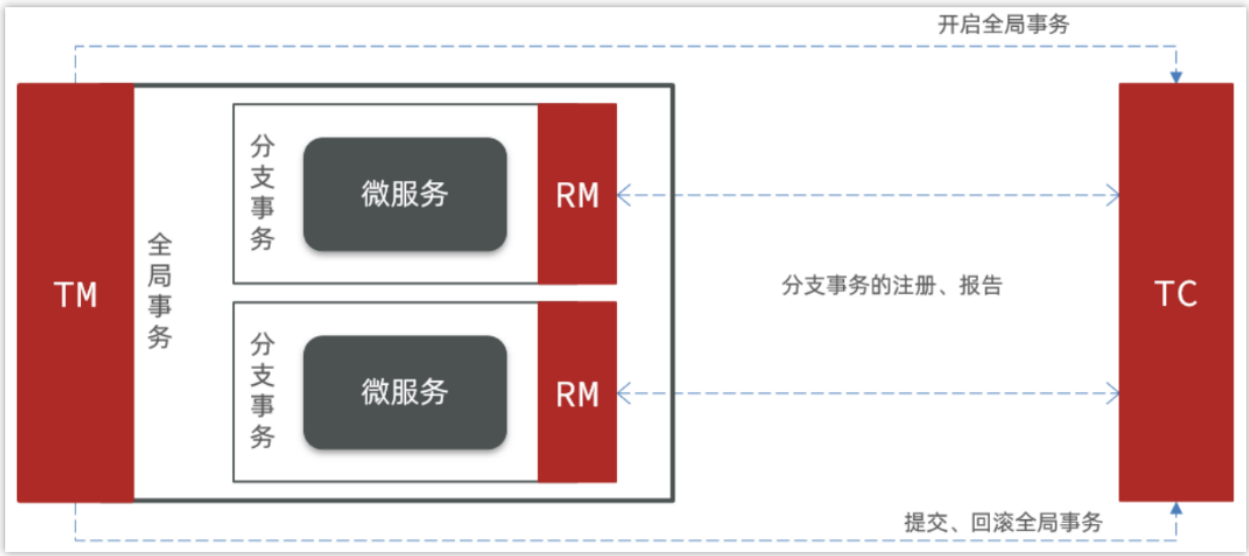
Seata事务管理中有三个重要的角色：

1、TC (Transaction Coordinator) -事务协调者：维护全局和分支事务的状态，协调全局事务提交或回滚。

2、TM (Transaction Manager) -事务管理器：定义全局事务的范围、开始全局事务、提交或回滚全局事务。

3、RM (Resource Manager) -资源管理器：管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。

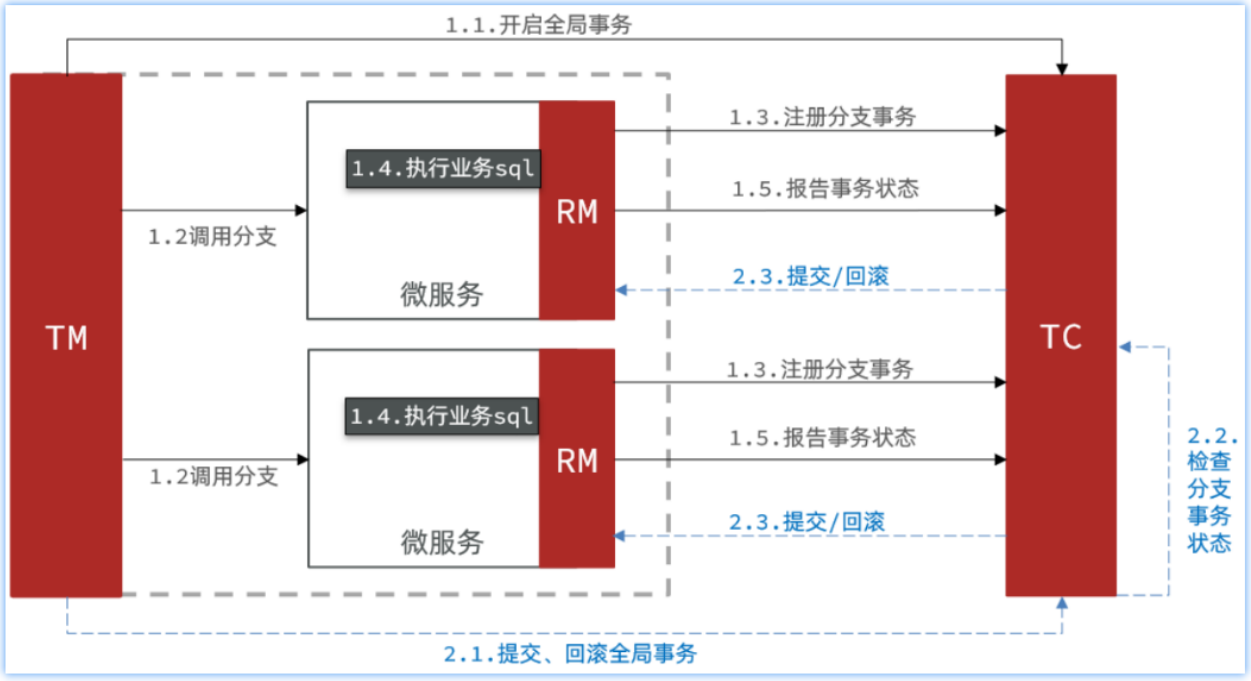
如下所示：



3.8 XA模式的工作流程是什么？

> 难易程度：☆☆☆
>
> 出现频率：☆☆

xa模式整个工作流程图如下所示：



分为两个阶段：

1、RM一阶段的工作：① 注册分支事务到TC ② 执行分支业务sql但不提交 ③ 报告执行状态到TC

2、TC二阶段的工作：TC检测各分支事务执行状态 ①如果都成功，通知所有RM提交事务 ②如果有失败，通知所有RM回滚事务

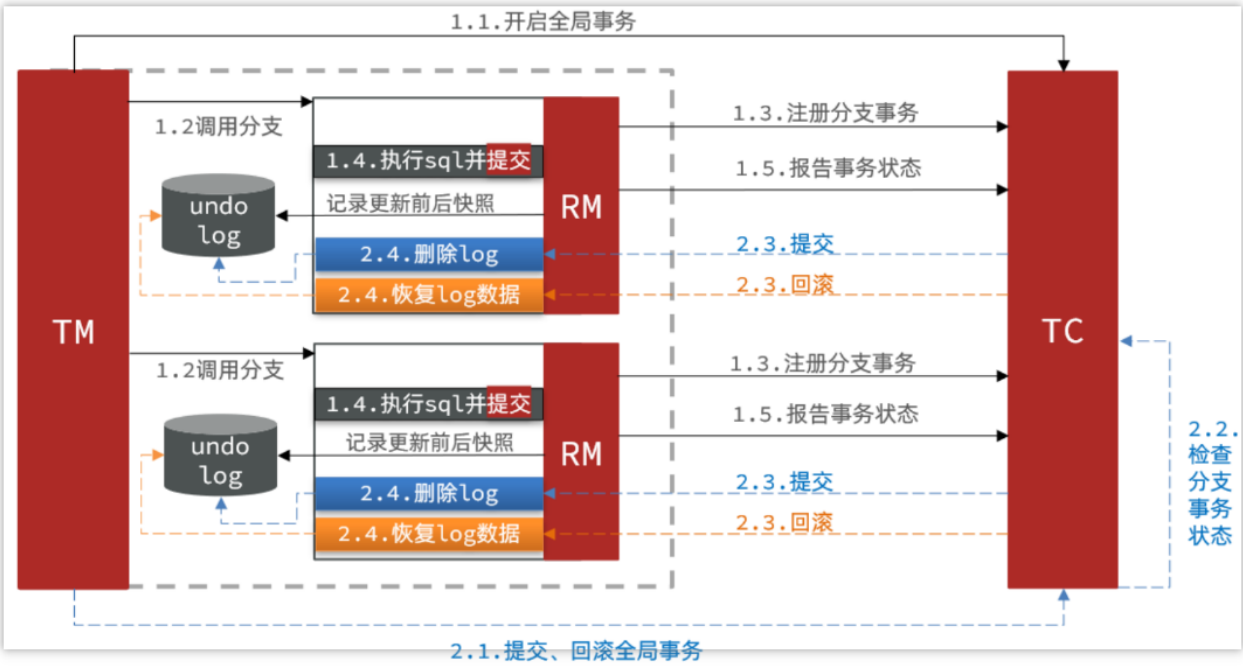
3、RM二阶段的工作：接收TC指令，提交或回滚事务

xa模式牺牲了可用性，保证了强一致性

3.9 AT模型的工作原理是什么？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

at模式的整个工作流程图如下所示：



1、阶段一RM的工作：① 注册分支事务 ② 记录undo-log（数据快照）③ 执行业务sql并提交 ④报告事务状态

2、阶段二提交时RM的工作：删除undo-log即可

3、阶段二回滚时RM的工作：根据undo-log恢复数据到更新前

at模式牺牲了一致性，保证了可用性

3.10 TCC模型的工作原理是什么？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

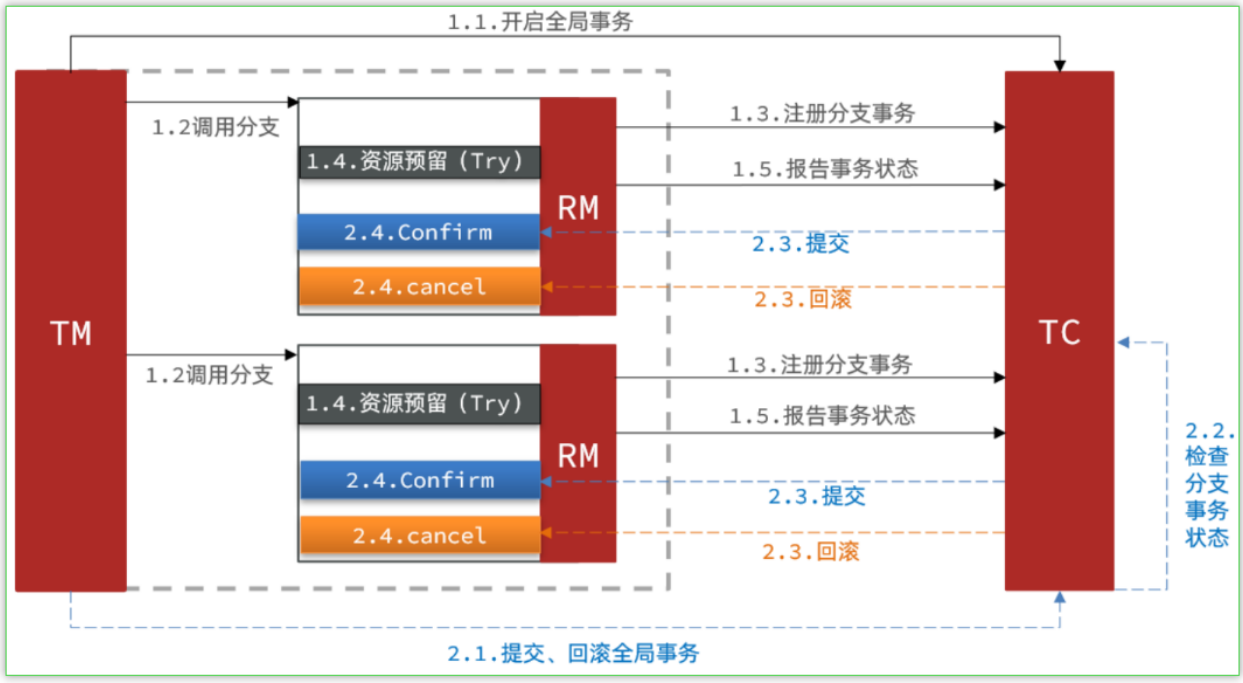
TCC模式与AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复。需要实现三个方法：

1、Try：资源的检测和预留；

2、Confirm：完成资源操作业务；要求 Try 成功 Confirm 一定要能成功。

3、Cancel：预留资源释放，可以理解为try的反向操作。

Seata中的tcc模型的执行流程如下所示：



1、阶段一RM的工作：① 注册分支事务 ② 执行try操作预留资源 ④报告事务状态

2、阶段二提交时RM的工作：根据各分支事务的状态执行confirm或者cancel

4 面试现场

4.1 Springboot

> **面试官：**讲一讲SpringBoot自动装配的原理？
>
> **候选人：**
>
> 嗯，好的，它是这样的。
>
> 在Spring Boot项目中的引导类上有一个注解@SpringBootApplication，这个注解是对三个注解进行了封装，分别是：
>
> - @SpringBootConfiguration
> - @EnableAutoConfiguration
> - @ComponentScan
>
> 其中`@EnableAutoConfiguration`是实现自动化配置的核心注解。

该注解通过`@Import`注解导入对应的配置选择器。关键的是内部就是读取了该项目和该项目引用的Jar包的的classpath路径下**META-INF/spring.factories**文件中的所配置的类的全类名。

在这些配置类中所定义的Bean会根据条件注解所**指定的条件来决定**是否需要将其导入到Spring容器中。

一般条件判断会有像`@ConditionalOnClass`这样的注解，判断是否有对应的class文件，如果有则加载该类，把这个配置类的所有的Bean放入spring容器中使用。

面试官：讲一讲SpringBoot启动流程

候选人：

嗯，是这样的

springboot项目在启动的时候, 首先会执行启动引导类里面的

`SpringApplication.run(AdminApplication.class, args)`方法

这个run方法主要做的事情可以分为三个部分：

第一部分进行SpringApplication的初始化模块，配置一些基本的环境变量、资源、构造器、监听器

第二部分实现了应用具体的启动方案，包括启动流程的监听模块、加载配置环境模块、及核心的创建上下文环境模块

第三部分是自动化配置模块，该模块作为springboot自动配置核心

面试官：你们常用的SpringBoot起步依赖有哪些

候选人：

嗯，现在使用springboot的情况比较多，这些起步依赖也用过很多

比如：关于redis、mabatis-plus、test、mongodb、apqp等等吧

面试官：springBoot支持的配置文件有哪些？加载顺序是什么样的

候选人：

嗯，这个主要支持文件类型的加载，yml和properties文件，优先加载yml，侯然后加载properties

如果有相同的配置，先加载的会被后加载的文件覆盖

假如在启动项目的时候给了启动参数，则最后生效，会覆盖前面所有相同的配置

```
java  -jar --server.port=8089 xx.jar
```

面试官：运行一个SpringBoot项目有哪些方式

候选人：

嗯，这个也有很种方式

第一种，也是比较常见的直接使用jar -jar 运行

第二种是可以配置插件 , 将springboot项目打war包, 部署到Tomcat中运行

第三种是直接用maven插件运行 maven spring-boot：run运行项目

面试官：Spring Boot的核心注解是哪个？他由哪几个注解组成的？

候选人：

嗯~~

Spring Boot的核心注解是@SpringBootApplication , 他由几个注解组成 :

- @SpringBootConfiguration： 组合了- @Configuration注解，实现配置文件的功能；
- @EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项
- @ComponentScan：Spring组件扫描

面试官：你们项目中使用的SpringBoot是哪个版本？

候选人：

嗯，我们的项目中使用的

- SpringBoot : 2.3.9.RELEASE
- SpringCloud : Hoxton.SR10
- SpringCloudAlibaba : 2.2.5.RELEASE

面试官：Spring Boot如何定义多套不同环境配置？

候选人：

这个我们项目提供多套配置文件，比如说application-dev.properties、application-prod.properties等等

然后在applcation.properties文件中指定当前的环境**spring.profiles.active=prod**,这时候读取的就是application-prod.properties文件。

4.2 SpringCloud

面试官：什么是微服务?微服务的优缺点是什么?

候选人：

嗯~~

微服务就是一个独立的职责单一的服务应用程序，一个模块

优点：松耦合，聚焦单一业务功能，无关开发语言，团队规模降低。在开发中，不需要了解多有业务，只专注于当前功能，便利集中，功能小而精。微服务一个功能受损，对其他功能影响并不是太大，可以快速定位问题。微服务只专注于当前业务逻辑代码，不会和 html、css 或其他界面进行混合。可以灵活搭配技术，独立性比较好。

缺点：随着服务数量增加，管理复杂，部署复杂，服务器需要增多，服务通信和调用压力增大，运维工程师压力增大，人力资源增多，系统依赖增强，数据一致性，性能监控。

面试官：Spring Cloud 5大组件有哪些？

候选人：

早期我们一般认为的Spring Cloud五大组件是

- Eureka ：注册中心
- Ribbon ：负载均衡
- Feign ：远程调用
- Hystrix： 服务熔断

- Zuul/Gateway ：网关

随着SpringCloudAlibba在国内兴起，我们项目中使用了一些阿里巴巴的组件

- 注册中心/配置中心 Nacos
- 负载均衡 Ribbon
- 服务调用 Feign
- 服务保护 sentinel
- 服务网关 Gateway

面试官：服务注册和发现是什么意思？Spring Cloud 如何实现服务注册发现？

候选人：

我理解的是主要三块大功能，分别是服务注册 、服务发现、服务状态监控

1. 服务注册 : 服务启动的时候会将服务的信息注册到注册中心, 比如: 服务名称 , 服务的IP , 端口号等
2. 服务发现 : 服务调用方调用服务的时候, 根据服务名称从注册中心拉取服务列表 , 然后根据负载均衡策略 , 选择一个服务, 获取服务的IP和端口号, 发起远程调用
3. 服务状态监控 : 服务提供者会定时向注册中心发送心跳 , 注册中心也会主动向服务提供者发送心跳探测, 如果长时间没有接收到心跳, 就将服务实例从注册中心下线或者移除

使用的话, 首先需要部署注册中心服务 , 然后在我们自己的微服务中引入注册中心依赖, 然后再配置文件中配置注册中心地址 就可以了

面试官：nacos、eureka的区别？

候选人：

我们项目中采用的nacos作为注册中心的，同时nacos也可以做为配置中心使用，而eureka只是有注册中心功能，不过关于注册中心它们也有一些不同

① Nacos支持服务端主动检测提供者状态：临时实例采用心跳模式，非临时实例采用主动检测模式

② 临时实例心跳不正常会被剔除，非临时实例则不会被剔除

③ Nacos支持服务列表变更的消息推送模式，服务列表更新更及时

④ Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；

也就说，Eureka采用AP方式，而naocs默认是AP模式，也可以采用CP模式

面试官：你们项目中微服务之间是如何通讯的?

候选人：

嗯，如果是同步通信的话，我们一般使用springCloud的组件Feign发送http请求调用

如果是异步通信的话，使用的消息队列中间件，如RabbitMq、Kafka这些都用过

面试官：你们项目负载均衡如何实现的？

候选人：

是这样~~

在服务调用过程中的负载均衡一般使用SpringCloud的Ribbon 组件实现，Feign的底层已经自动集成了Ribbon，使用起来非常简单

客户端调用的话一般会通过网关, 通过网关实现请求的路由和负载均衡

面试官：Ribbon负载均衡策略有哪些？如果想自定义负载均衡策略如何实现？

候选人：

嗯，是这样~~

Ribbon提供了很多负载均衡算法，比如有轮询（RoundRobinRule）、随机策略（RandomRule）、还可以按照权重（WeightedResponseTimeRule）进行设置，默认采用的是区域可用的服务器为基础进行服务器的选择（ZoneAvoidanceRule），底层采用的一个区域内的机器进行轮询

如果想要自定义负载均衡策略可以有两种方式

第一是：定义一个@Bean，返回值是IRule，可以采用不同的实现来设置不同的策略，这个全局生效的。我们一般也很少采用

第二是：通过配置文件的方式添加某一个服务的负载均衡策略，这种方式比较常见

面试官：什么是Spring Cloud Gateway以及在你们的项目中如何去应用该组件的？

候选人：

Spring Cloud Gateway是Spring Cloud中所提供的一个服务网关组件，是整个微服务的统一入口，在服务网关中可以实现请求路由、统一的日志记录，流量监控、权限校验等一系列的相关功能！

我们在项目使用网关主要是权限校验

具体实现思路：使用Spring Cloud Gateway中的全局过滤器拦截请求(GlobalFilter、Order)，从请求头中获取token，然后解析token。如果可以进行正常解析，此时进行放行；如果解析不到直接返回。

面试官：你们项目的配置文件是怎么管理的？

候选人：

嗯，我们的项目中大部分的固定的配置文件都放在服务本地，一些根据环境不同可能会变化的部分，放到Nacos中

面试官：你们项目中有没有做过限流？怎么做的？

候选人：

回答方式一：

我们的项目目前访问量不大，并没设置限流操作，不过我知道怎么去用，常见的限流算法：**漏桶算法、令牌桶算法**

漏桶算法：漏桶算法其实很简单，可以粗略的认为就是注水漏水过程，往桶中以一定速率流出水，以任意速率流入水，当水超过桶流量则丢弃，因为桶容量是不变的，保证了整体的速率。

令牌桶算法：令牌桶是一个存放固定容量令牌的桶，按照固定速率r往桶里添加令牌；桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃；当一个请求达到时，会尝试从桶中获取令牌；如果有，则继续处理请求；如果没有则排队等待或者直接丢弃；可以发现，漏桶算法的流出速率恒定，而令牌桶算法的流出速率却有可能大于r；

从作用上来说，漏桶和令牌桶算法最明显的区别就是是否允许突发流量(burst)的处理，漏桶算法能够强行限制数据的实时传输（处理）速率，对突发流量不做额外处理；而令牌桶算法能够在限制数据的平均传输速率的同时允许某种程度的突发传输。

回答方式二：

我们项目的流量还是比较大的，我们项目中用的令牌桶算法来进行限流的，在gateway中进行设置。

令牌桶是一个存放固定容量令牌的桶，按照固定速率r往桶里添加令牌；桶中最多存放b个令牌，当桶满时，新添加的令牌被丢弃；当一个请求达到时，会尝试从桶中获取令牌；如果有，则继续处理请求；如果没有则排队等待或者直接丢弃；可以发现，漏桶算法的流出速率恒定，而令牌桶算法的流出速率却有可能大于r； 也就说对于突发流量令牌桶也能应付。

具体使用是，在网关路由中进行过滤器配置，可以设置桶的带下，和固定速率。我们通常也会按照用户访问的ip进行限制，这个令牌需要存入redis，所以也需要集成redis使用。

面试官：断路器/熔断器用过嘛？断路器的状态有哪些？

候选人：

我们项目中使用Hystrix实现的断路器 ,默认是关闭的，如果需要开启需要在引导类上添加注解：

```
@EnableCircuitBreaker
```

断路器状态机包括三个状态：

- closed：关闭状态，断路器放行所有请求，并开始统计异常比例、慢请求比例。超过阈值则切换到open状态
- 通常一般判断是服务响应时间太久会进行开启
- open：打开状态，服务调用被**熔断**，访问被熔断服务的请求会被拒绝，快速失败，直接走降级逻辑。Open状态5秒后（默认值）会进入half-open状态
- half-open：半开状态，放行一次请求，根据执行结果来判断接下来的操作。
- 请求成功：则切换到closed状态
- 请求失败：则切换到open状态

面试官：你们项目中有做过服务降级嘛？

候选人：

我们项目中涉及到服务调用得地方都会定义降级,一般降级逻辑就是返回默认值 ,降级的实现也非常简单,就是创建一个类实现 `FallbackFactory` 接口,然后再对应的Feign客户端接口上面,通过@FeignClient指定降级类

面试官：你们的微服务是这么监控的？

候选人：

嗯，有的，因为我们的微服务也比较多，需要进行监控，我们采用的skywalking进行监控的

skywalking主要可以监控接口、服务、物理实例的一些状态。特别是在压测的时候可以看到众多服务中哪些服务和接口比较慢，我们可以针对性的分析和优化。

我们还在skywalking设置了告警规则，特别是在项目上线以后，如果报错，我们分别设置了可以给相关负责人发短信和发邮件，第一时间知道项目的bug情况，第一时间修复

4.3 分布式事务

面试官：什么是分布式事务？

候选人：

嗯，清楚的~

分布式系统上一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务节点上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。

面试官：哪些场景下都会产生分布式事务？

候选人：

嗯，这个一般有两种情况

第一个是跨库操作，一个应用某个功能需要操作多个库，不同的库中存储不同的业务数据

第二个是跨服务事务，一个应用某个功能需要调用多个微服务进行实现，不同的微服务操作的是不同的数据库

像我们的项目中更多的跨服务事务比较多。

面试官：什么是CAP理论？

候选人：

CAP主要是在分布式项目下的一个理论。包含了三项，一致性、可用性、分区容错性

- 一致性(Consistency)是指更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致(强一致性)，不能存在中间状态。
- 可用性(Availability) 是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。
- 分区容错性(Partition tolerance) 是指分布式系统在遇到任何网络分区故障时，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

面试官：为什么分布式系统中无法同时保证一致性和可用性？

候选人：

嗯，是这样的~~

首先一个前提，对于分布式系统而言，分区容错性是一个最基本的要求，因此基本上我们在设计分布式系统的时候只能从一致性（C）和可用性（A）之间进行取舍。

如果保证了一致性（C）：对于节点N1和N2，当往N1里写数据时，N2上的操作必须被暂停，只有当N1同步数据到N2时才能对N2进行读写请求，在N2被暂停操作期间客户端提交的请求会收到失败或超时。显然，这与可用性是相悖的。

如果保证了可用性（A）：那就不能暂停N2的读写操作，但同时N1在写数据的话，这就违背了一致性的要求。

面试官：什么是BASE理论？

候选人：

嗯，这个也是CAP分布式系统设计理论

BASE是CAP理论中AP方案的延伸，核心思想是即使无法做到强一致性（StrongConsistency，CAP的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。它的思想包含三方面：

1、Basically Available（基本可用）：基本可用是指分布式系统在出现不可预知的故障的时候，允许损失部分可用性，但不等于系统不可用。

2、Soft state（软状态）：即是指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

3、Eventually consistent（最终一致性）：强调系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。其本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

面试官：分布式事务的常见的解决方案有哪些？

候选人：

嗯，这个有多种方案都可以解决分布式事务的问题。

第一个是2PC两阶段提交

1、第一阶段：投票阶段

2、第二阶段：提交/执行阶段。

举例订单服务A，需要调用支付服务B 去支付，支付成功则处理订单状态为待发货状态，否则就需要将购物订单处理为失败状态。 那么看2PC阶段是如何处理的。

第二个方案是TCC

TCC（Try-Confirm-Cancel）又称补偿事务。其核心思想是："针对每个操作都要注册一个与其对应的确认和补偿（撤销操作）"。

它分为三个操作：

1、Try阶段：主要是对业务系统做检测及资源预留。

2、Confirm阶段：确认执行业务操作。

3、Cancel阶段：取消执行业务操作。

比如说，转账，A向B转1000元

try阶段主要是检查和冻结，检查A的余额是否足够，进行冻结操作，检查B的账户是否正常

Confirm阶段，假如try阶段的检查和冻结都没问题，则AB都执行提交操作即可

Cancel阶段，假如try阶段的检查和冻结有问题，则执行回滚操作

第三，也可以采用MQ解决分布式事务

上面的两种分布式事务的解决方案适用于对数据一致性要求很高的场景。如果数据强一致性要求没那么高，可以采用消息中间件（MQ）实现事务最终一致。 在支付系统中，常常使用的分布式事务解决方案就是基于MQ实现的，它对数据强一致性要求没那么高，但要求数据最终一致即可。

面试官：Seata的架构是什么？

候选人：

Seata事务管理中有三个重要的角色：

1、TC (Transaction Coordinator) -事务协调者：维护全局和分支事务的状态，协调全局事务提交或回滚。

2、TM (Transaction Manager) -事务管理器：定义全局事务的范围、开始全局事务、提交或回滚全局事务。

3、RM (Resource Manager) -资源管理器：管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。

面试官：XA模式的工作流程是什么？

候选人：

嗯，seata的xa模式分为两个阶段：

1、RM一阶段的工作：① 注册分支事务到TC ② 执行分支业务sql但不提交 ③ 报告执行状态到TC

2、TC二阶段的工作：TC检测各分支事务执行状态①如果都成功，通知所有RM提交事务 ②如果有失败，通知所有RM回滚事务

3、RM二阶段的工作：接收TC指令，提交或回滚事务

xa模式牺牲了可用性（需要互相等待，共同提交事务），保证了强一致性

面试官：AT模型的工作原理是什么？

候选人：

嗯，seata的AT模型分为两个阶段：

1、阶段一RM的工作：① 注册分支事务 ② 记录undo-log（数据快照）③ 执行业务sql并提交 ④报告事务状态

2、阶段二提交时RM的工作：删除undo-log即可

3、阶段二回滚时RM的工作：根据undo-log恢复数据到更新前

at模式牺牲了一致性，保证了可用性

面试官：TCC模型的工作原理是什么？

候选人：

TCC模式与AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复。需要实现三个方法：

1、Try：资源的检测和预留；

2、Confirm：完成资源操作业务；要求 Try 成功 Confirm 一定要能成功。

3、Cancel：预留资源释放，可以理解为try的反向操作。

Seata中的tcc模型的执行流程也是两步

1、阶段一RM的工作：① 注册分支事务 ② 执行try操作预留资源 ④报告事务状态

2、阶段二提交时RM的工作：根据各分支事务的状态执行confirm或者cancel