分布式&微服务架构

1 前言

面试时，面试官可能会问到你未来的规划的？

有同学可能张口就来，我要在 5 年成为一名出色的架构师、技术负责人等。

那面试官又问出了如下的问题：

如何成为一名优秀的架构师？

个人认为一名优秀的架构师应该具备如下几个特点：

第一、强烈的好奇心。

不只是对软件技术本身，对大千世界都保持强烈的好奇心，强烈的好奇心能够让你敏锐地发现有潜力的重要的新技术；

第二、敏锐的业务嗅觉。

工程技术不同于科学研究，工程技术最终要服务于实际业务的，要产生实际价值，是要赚钱的，那么业务需要什么样的功能，需要什么样的技术去实现，都需要有敏锐的业务嗅觉；

第三、扎实的技术基础。

基本功一定要扎实，如操作系统、数据结构、数据库原理，编程语言和算法原理，设计模式和设计原则等。只要这些软件技术基础都扎实了，才能构建起敏锐的技术嗅觉，才能构建起自己坚实的技术体系；

第四、出色的编程能力。

虽然可能再需要你写代码，但是要有出色的编程能力，这样才能对架构中那些最敏感的技术点保持敏锐的技术嗅觉，能够抓住软件的关键点，不会在纷繁复杂的问题中迷失方向；

第五、深刻领悟主流技术产品和模式。

架构师不是凭空进行架构设计，是站在巨人的肩膀上，在现有的其他优秀架构基础上进一步设计出符合自己业务特点的架构系统。只有深刻领悟主流技术产品和模式是如何设计的，才能根据自己的业务特点，去其糟粕，做最好的匹配和改进，从而设计出属于自己的优秀的系统。

架构师的学习可从如下几个方面着手：

第一、基础方面：包括数据结构、操作系统、算法应用、设计模式等一切拥有优秀编程能力所应该熟知的软件基础知识；

第二、技术方面：如何使用优秀的技术产品去构建自己的系统，这些技术产品各自的特点是什么,有什么优缺点、具体原理是怎样的？都要有深刻掌握和理解。对大型互联网系统而言，主要包括缓存、异步、分布式存储、微服务等；

第三、架构方面：考虑点主要包括高可用、高性能、高扩展这三部分。

总之，作为一名技术人，我们既然选择了这个职业，就一定要有上进的决心，不能只顾写代码，一定要提升架构设计能力。因为即使代码写得再好，做的也是执行层面的事儿，就会有收入的天花板，想要突破它，就要突破你做事儿的边界，让自己成为一个架构师或技术负责人。

2 分布式

2.1什么是分布式架构？

当系统的并发处理能力，存储能力等不足时，我们可能会创建多个 Web 服务(多个 tomcat 服务器),多个数据库服务(主动架构等)，这些服务器通过网络进行连接，然后协同处理客户端的并发请求，这样的系统我们称之为分布式系统。（这里要说明的一点是微服务架构是分布式架构，但分布式架构不一定是微服务架构）

2.2为什么需要分布式架构？

当一个系统的业务量越来越大时，我们需要垂直或是水平拆分业务系统，同时为了避免所有

业务都部署在一台机器上时，一旦机器出现故障从而导致整体不可用，就需要将这些业务部署在多台计算机上，来构建一个分布式架构。

分布式架构可以更好的提高系统的容量、可靠性（避免单点故障）、性能。同时因为模块化，系统的可重用性以及并行开发的效率也会提高。

2.3分布式架构有哪些优势？

1. 可以实现更大数据量的存储。（每天几十个 PB 的数据）
2. 可以更好提高系统的高可用性。（业务冗余、业务拆分、限流、....）
3. 可以更好提高系统的可重用性。(公共模块-例如订单系统、支付系统，日志收集系统，监控系统，。。。)
4. 可以更好提高系统的性能。（并行处理能力）

2.4分布式架构有什么劣势？

分布式系统架构虽然解决了“单点”、“性能”和”容量“的问题，但却新增了一些新的问题。

- 会增加架构设计的难度。
- 部署和维护的成本也会加大。

很多技术方案都是“按下葫芦浮起瓢”，都是有得有失，分布式架构也是如此，理性面对即可。

2.5分布式架构有哪些关键技术？

- **服务治理。**

服务治理的最大意义是需要把服务间的依赖关系、服务调用链，以及关键的服务给梳理出来，并对这些服务进行性能和可用性方面的管理。

- **架构管理。**

基于服务所形成的架构需要有架构版本管理、整体架构的生命周期管理，以及对服务的编排、聚合、事务处理等服务调度功能。

- **DevOps。**

分布式系统可以更为快速地更新服务，但是对于服务的测试和部署都会是挑战。所以，还需要 DevOps 的全流程，其中包括环境构建、持续集成、持续部署等。自动化运维。有了 DevOps 后，我们就可以对服务进行自动伸缩、故障迁移、配置管理、状态管理等 一系列的自动化运维技术了。

- **资源调度管理。**

应用层的自动化运维需要基础层的调度支持，也就是云计算 IaaS 层的计算、存储、网络等资源调度、隔离和管理。

- **整体架构监控。**

如果没有一个好的监控系统，那么自动化运维和资源调度管理只可能成为一个泡影， 因为监控系统是你的眼睛。没有眼睛，没有数据，就无法进行高效的运维。所以说，监控是非常重要 的部分。这里的监控需要对三层系统（应用层、中间件层、基础层）进行监控。

- **流量控制。**

我们的流量控制，负载均衡、服务路由、熔断、降级、限流等和流量相关的调度都 会在这里，包括灰度发布之类的功能也在这里。

2.6基于分布式架构如何提高其高性能？

一般面对这样的问题，从整体维度去思考时，通常要分析问题，例如影响系统性能的因素有哪些？
1）请求数据传输时间
2）请求数据的处理时间
3）响应数据的传输时间
4）响应数据的渲染时间

解决方案如下：
1）减少数据传输时间？(加带宽，减少数据传输量，减少传输距离，数据压缩)
2）提高请求数据的处理速度？(分布式架构，缓存，算法，sql 调优，索引的设计，异步，硬件)
3）提高数据在客户端的渲染时间？(局部更新，减少不必要的元素等)

具体从架构层面进行设计的话可以从如下几个维度进行思考：

- **应用缓存。**

为系统添加缓存，可以有效地提高系统的访问能力。从前端的浏览器，到网络，再到后端的服务，底层的数据库、文件系统、硬盘和 CPU，全都有缓存，这是提高快速访问能力最有效的手段。

- **负载均衡。**

负载均衡是做水平扩展的关键技术，我们可以使用多台机器来共同分担一部分流量请求。

- **异步调用。**

异步系统主要通过消息队列来对请求做排队处理，这样可以把前端的请求的峰值给“削平”了，而后端通过自己能够处理的速度来处理请求。

- **数据分区和数据镜像。**

数据分区是把数据按一定的方式分成多个区（比如通过地理位置），不同的数据区来分担不同区的流量。

2.7 如何基于架构提高系统的稳定性？

- **服务拆分。**

服务拆分可以更好的实现故障隔离，同时也可以重用服务模块。

- **服务冗余。**

服务冗余是为了去除单点故障，支持服务的弹性伸缩，以及故障迁移。

- **限流降级。**

当系统扛不住压力时，只能通过限流或者功能降级的方式来停掉一部分服务，或是拒绝一部分用户，以确保整个架构不会挂掉。

- **高可用架构。**

高可用就是从冗余架构的角度来保障可用性。比如多租户隔离，灾备多活等。总之，就是为了不出单点故障。

- **高可用运维。**

指的是 DevOps 中的 CI（持续集成）/CD（持续部署）。一个良好的运维应做了足够的自动化测试，做了相应的灰度发布，以及对线上系统的自动化控制。这样，可以做到"计划内"或是"非计划内"的宕机事件的时长最短。

2.8分布式架构有什么难点？

- **异构系统存在很多不标准问题。**

构建软件时使用的编程语言、通讯协议、数据格式、运维标准可能不同，进而导致架构设计的复杂度越来越高。

- **系统架构中的服务依赖问题。**

传统的单体应用，一台机器挂了，整个软件就挂掉了，分布式架构下也可能会出现这样的问题，因为一个服务可能会依赖于另一个服务，某个服务挂掉了，会导致调用链上的服务都出现故障（这就是多米诺骨牌效应）

- **故障发生的概率更大。**

分布式系统中，服务和机器都会比较多，故障发生的频率会更大，只是影响面没有单体应用的影响面大，分布式系统中故障可以被隔离。还有就是分布式架构管理相对于单体架构也更加复杂，没有优秀的架构管理人员，故障的频率还是会非常高。

- **多层架构的运维复杂度更大。**

分布式架构中，我们可以将系统分为四层(基础层、平台层、应用层、接入层)。
1)基础层:包括机器、网络和存储设备。
2)平台层:就是中间件层包括 tomcat、MySQL、Redis、Kafka 类似的软件。
3)应用层:就是我们的业务软件，包括各种业务服务。

4)接入层:就是接入用户请求的网关、负载均衡、CDN、DNS 等。

3 微服务入门

3.1什么是微服务?

微服务是一种分布式架构，简单点就是将整体大应用，基于业务打散为更为微小的服务。然后作为独立的进程进行开发、测试、部署、运行、维护。

3.2微服务架构诞生的背景?

服务大了太臃肿，要拆成若干个小系统，然后进行分而治之（例如北京一个火车站到多个火车站）。
这样分了以后,可以把每个服务作为一个独立的开发项目,由团队进行快速开发、迭代升级。

3.3为什么需要微服务架构?

对系统分而治之,故障隔离、同时解决因并发访问过大带来的系统复杂性(例如:业务,开发,测试,升级,可靠性等)

3.4微服务架构可能带来什么问题?

数据一致性问题、网络通信故障、限流与熔断机制、调用链路跟踪、集群监控、用户登录与权限管理。

3.5微服务解决方案有哪些?

大厂一般会自研,中小企业采用开源 Spring Cloud Netfix（大部分组件停止更新）,Spring Cloud Alibaba（一站式解决方案）等

3.6微服务架构中有哪些关键组件?

服务的注册,发现,配置,限流降级,API 网关,分布式事务管理等

4 注册中心部分

4.1服务注册中心诞生的背景?

注册中心本质上就是存储服务信息的一个服务,也可以理解为一个中介(这里用到了中介者模式)。我们知道服务多了,需要统一管理。例如所有公司需要在工商局进行备案,淘宝也可以理解为买家和卖家的注册中心。

4.2服务注册中心你是如何选型的?

选型时,我们首先要从这样的几个维度进行思考:
1) 社区活跃度(用户群的大小)
2) 稳定性
3) 功能
4) 性能
5) 学习成本。

4.3你了解哪些服务注册中心?

Zookeeper,Eureka,Nacos,Consul

4.4说说你对 Nacos 的理解?

Nacos 是 Alibaba 基于 SpringBoot 技术实现的一个注册中心,配置中心,本质上也是一个 web 服务,
提供了服务的注册,发现,配置等功能。可以作为各个服务的一个中介,也就是所谓的中介者模式。

4.5Nacos 如何检测服务状态?

通过心跳包实现,服务启动时会定时向 nacos 发送心跳包-BeatInfo。

4.6Nacos 注册中心有哪些常用配置?

我们可以在项目的 application.yml 中进行服务的注册配置，例如:

```
   spring:
      application:
       # 定义服务名称(注册到 nacos 中的服务)
        name: xxxx
      cloud:
        nacos:
          discovery:
            # 可以配置多个，逗号分隔
            server-addr: localhost:8848
            # nacos 客户端向服务端发送心跳的时间间隔，时间单位其实是 ms
            heart-beat-interval: 5000
            # 服务端没有接收客户端心跳请求就将其设为不健康的时间间隔，默认为 15s
            # 注：推荐值该值为 15s 即可，如果有的业务线希望服务下线或者出故障时希望尽快被
发现，可以适当减少该值
            heart-beat-timeout: 20000
            # 客户端在启动时是否读取本地配置项(一个文件)来获取服务列表
            # 注：推荐该值为 false，若改成 true。则客户端会在本地的一个文件中保存服务信
息，
            # 当下次宕机启动时，会优先读取本地的配置对外提供服务。
            naming-load-cache-at-start: false
```

5 服务调用部分

5.1服务之间进行调用你是如何实现的?

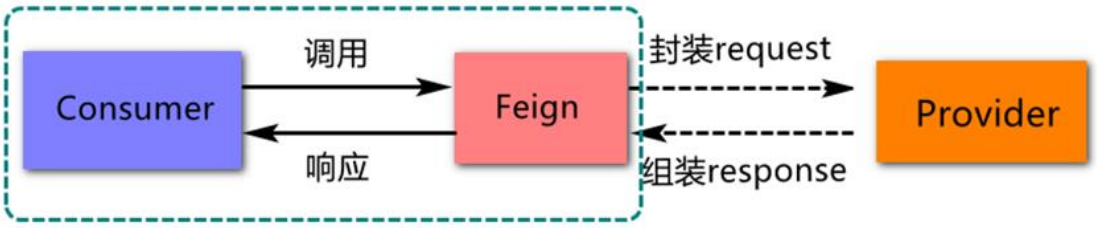
在微服务架构中，服务之间调用一般有两种方式:

1) Restful 方式（直接基于 http 协议进行远端服务调用，数据格式标准，通用性强，代表作品有 SpringCloud 中的 OpenFeign）

2) RPC 方式(直接基于 tcp 协议进行远端服务调用，协议数据格式不够通用，但是效率会比较好，代表组件有 Dubbo)

5.2你是如何理解 OpenFeign 的？

它基于 restful 方式进行声明式服务调用的一个服务调用组件，它可以嵌入到服务调用的客户端服务中，然后由客户端基于 Feign 代理以及 Restful 规范进行服务调用。



5.3微服务调用过程中你用的负载均衡组件是什么？

SpringCloud 中基于 Feign 方式进行远程服务调用时,负载均衡组件默认使用的是 Ribbon。

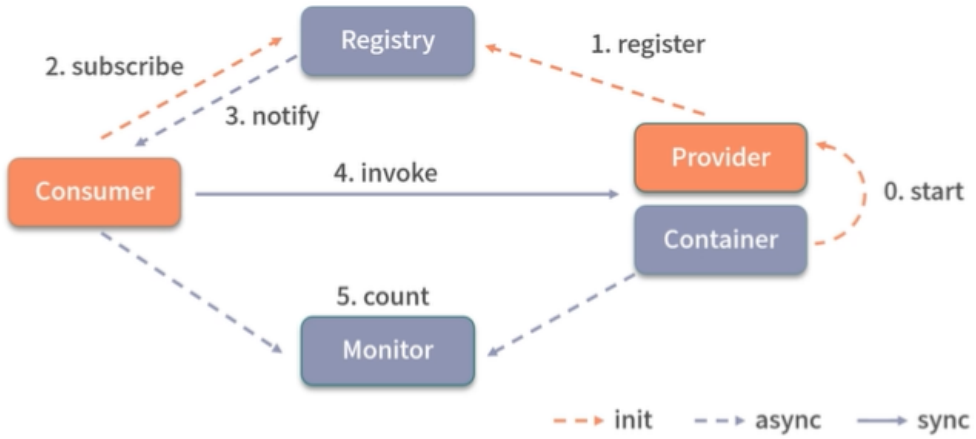
5.4常用负载均衡算法有哪些？

轮询、哈希、权重、重试、随机等

5.5什么是 Dubbo？

Apache Dubbo 是一款高性能、轻量级的开源 Java RPC 框架，提供了面向接口的远程方法调用，可靠智能的容错和负载均衡，服务自动注册和发现能力。总之，我们可以将 Dubbo 看成是一个分布式服务框架，致力于提供高性能、透明化的 RPC 远程调用方法以及服务治理方案,解决微服务架构落地时的问题。Dubbo 由阿里开源,后加入 Apache 基金会进行孵化，目前已经孵化完成，是 Apache 顶级开源项目。很多互联网大厂（例如阿里、滴滴、去哪儿网）都使用 Dubbo 作为 RPC 框架。还有一些大厂基于 Dubbo 进行二次开发实现了自己的 RPC 框架，例如当当网的 DubboX。

5.1 Dubbo 架构是怎样的？



5.2 Dubbo 源码从哪里去下载？

https://github.com/apache/dubbo

6 配置中心部分

6.1诞生的背景？

实际项目中我们会有很多配置，例如连接数据库的配置，日志级别的配置，负载均衡的配置，限流算法的配置，缓存是否开启等，这些配置起初可能是写在配置文件，但是项目上线以后，我们直接再通过配置文件方式修改具体配置就不太方便了，即使可以，那我们修改了配置件是否要重启系统呢？你重启气筒时，系统是否要停止对外界服务呢？，基于这些问题“配置中心”就诞生了。

6.2什么是配置中心？

配置中心是存储项目（服务）配置信息的一个服务，这个服务可以实现配置的动态发布和更

新。

6.3为什么要使用配置中心？

集中管理配置信息，动态发布配置信息，服务自动感知配置,提高服务可用性。例如实际项目中我们数据库的账号和密码可能每隔两周或 1 个月都要更新一下（更新的目的是要保证系统的安全），假如应用了配置中心，我们可以直接在配置中心进行更新即可，它会自动同步到具体的服务，我们不需要重启服务就可以获取最新的配置了。

6.4你在配置中心配置过什么内容？

负载均衡策略、连接池，日志、缓存、限流、熔断规则。

6.5为什么要定义 bootstrap.yml 文件？

此文件被读取的优先级比较高，可以在服务启动时读取配置中心的数据。然后通过这些数据来更新配置。

6.6配置中心宕机了，还可以读到配置信息吗？

配置中心的数据可以在服务本地存储一份，所以配置中心宕机了，可以从本地内存读取。

6.7微服务配置中心如何感知配置中心的配置的变化？

1.4.x 版本中的 nacos 客户端会基于长轮询机制从 nacos 获取配置信息。这个轮询可以这样去理解，
我们骑着共享单车去火车站买票，但是票卖完了，一种方式是直接返回（这种方式称之为短轮询）。还有一种方式在火车站的椅子上等一会，看看是否还会放票，是否有人会退票等。那这种机制就是长轮询。

6.8Nacos 的配置管理模型是怎样的？

在 nacos 中的配置管理模型中，一个 namespace 可以有多个分组，每个分组下又可以有多个服务实例。
例如：namespace>group>service/data-id

6.9Nacos 配置中心基础配置有哪些？

```
spring:
  application:
    name: xxxx
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yml
        # prefix: 文件名前缀，默认是 spring.application.name
        # 如果没有指定命令空间，则默认命令空间为 PUBLIC
        namespace: dev
        # 如果没有配置 Group，则默认值为 DEFAULT_GROUP
        group: DEFAULT_GROUP
        # 从 Nacos 读取配置项的超时时间
        timeout: 5000
        # 长轮训超时时间
        config-long-poll-timeout: 1000
        # 重试时间
        config-retry-time: 100000
        # 长轮训重试次数
        max-retry: 3
```

7 限流降级部分

7.1为什么要限流？

请求量比较大,但是系统资源处理能力不足。类似车辆限号。

7.2你了解哪些限流组件

阿里巴巴的 Sentinel,....

7.3你了解哪些限流算法？

计数器,令牌桶,漏桶,滑动窗口算法。

7.4Sentinel 如何对请求进行限流？

可以基于 Spring MVC 拦截器以及 aop 方式对请求进行限流。当请求传递到服务端，服务器可以调用拦截器对请求进行拦截，判定此请求是否允许放行，请求量太大可能就要被限制了。当然在热点数据限流上还可以通过 AOP 方式进行限制。

7.5Sentinel 出现限流时的异常类型是什么？

BlockException?

7.6Sentinel 中默认的异常处理器是什么？

DefaultBlockExceptionHandler

7.7Sentinel 限流的阈值类型有哪些？

QPS（每秒请求次数），线程数

7.8Sentinel 的流控模式有哪些？

1)直接模式：直接对请求 url 进行限流，
2)关联模式：一种霸权主义，要保证我(核心业务)先行。
3)链路模式：对同一个资源的访问有多条链路，然后对指定链路进行限流。

7.9如何理解 Sentinel 的关联限流？

霸权方式,当对A的资源的访问量比较大时,限流其它资源的访问(例如我们可以优先保证创建订单可以成功、查询订单可以被限流。)

7.10 如何理解 Sentinel 的链路限流?

对同一个资源的访问,可能会有多条链路,可以对指定链路进行限流。(例如对 Google 的访问)。

7.11 @SentinelResource 注解的作用是什么?

定义限流切入点方法,底层可以基于 aop 方式对请求链路进行限制(一般应用于链路限流、热点数据限流)。对应的切面(SentinelResourceAspect)。

7.12 Sentinel 常见的限流效果是什么?

快速失败,warm up(预热方式),排队等待

7.13 什么是服务降级?

系统出现大量的慢调用(一个请求响应的时间比较长)或一些异常(经常运行过程中会出现异常),可以对这些服务进行熔断-暂时关闭系统

7.14 如何理解慢调用?

客户端发起一个请求,得到服务端的响应比较慢。当然一般会给出一个时间标准

7.15 如何理解热点数据?

频繁访问的数据-例如文章、视频、图片、....

7.16 Sentinel 如何判断哪些数据是热点数据?

LRU 算法(最近最少使用算法-一般应用于缓存淘汰策略。)

7.17 对热点数据限流,底层基于什么机制去实现?

AOP（@SentinelResuource 注解定义切入点方法，然后在 SentinelResourceAspect 切面中基于限流策略，进行限流。）

8 API 网关部分

8.1为什么要使用 API 网关？

1)统一 url 访问（Restful，RPC），
2)服务保护（不对外暴露内部服务的真实地址）
3)统一身份认证（单点登陆系统-不用每个服务都编写一个登陆认证模块）
4)统一跨域设计
5）限流降级(Sentinel)
6）黑白名单(GlobalFilter)

8.2API 网关都可以配置什么？

1)路由（uri，predicate，filter）
2)统一身份认证
3)负载均衡(Ribbon)
4)黑白名单设计(自定义)
5)限流熔断(Sentinel)
6)跨域等

8.3API 网关中的负载均衡如何实现？

Ribbon

8.4SpringCloud Gateway 处理请求的基本流程？

1)客户端向 Spring Cloud Gateway 发出请求。
2)基于 GatewayHandlerMapping 调用谓词 predicates(predicates)的集合判定请求与路由(Routers)是否匹配，不匹配则抛出异常。
3)将请求其发送到 Gateway Web Handler。此对象基于路由配置调用过滤链中的过滤器（也就是所谓的责任链模式）进一步的处理请求。
4)将请求转发到具体的服务。

8.5网关中谓词(Predicate)的作用是什么？

对请求 url、请求数据进行校验，符合规则再交给过滤器去处理。

8.6你知道哪些谓词对象(Predicate)对象？

1）Path 相关的
2）日期时间相关的
3）IP 相关的
4）Cookie 相关的
5）请求参数相关的
6）请求方式相关
7）请求头相关的
8）上传文件大小相关的
9）…

8.7网关中的过滤器是如何分类的？

1）全局过滤器(不需要配置)，作用于所有路由。
2）局部过滤器（这个需要针对具体路由进行配置），只作用于具体路由。

8.8你知道 Gateway 中的哪些过滤器？

1）负载均衡相关的
2）请求转发相关的
3）限流相关的
4）请求前缀、后缀相关的
5）。。。

8.9API Gateway 如何基于 Sentinel 进行限流？

1）依赖
2）配置

9 分布式事务

9.1什么是分布式事务？

分布式事务是分布式架构下的一种事务处理方式，是指位于不同的节点之上的事务参与者，执行一系列操作时，要确保这些操作要么都执行成功，要么都执行失败。本质上来说，就是为了保证不同数据库的数据一致性。

9.2CAP 定理是什么？

分布式系统有三个指标：

1)Consistency：一致性(多方对数据读写时要保证数据的一致性)
2)Availability：可用性(只要有请求,就必须要有响应)
3)Partition tolerance：分区容错性(分布式架构系统中必须要满足的一个指标,在分布式架构下的一台服务器出了问题,其它服务必须依旧可以持续提供业务服务)

这三个指标不可能同时满足，这个定理就叫 CAP 定理。
在分布式系统中,通常分区容错性是必须满足的,可用性(AP)和一致性(CP)只能选择其一。

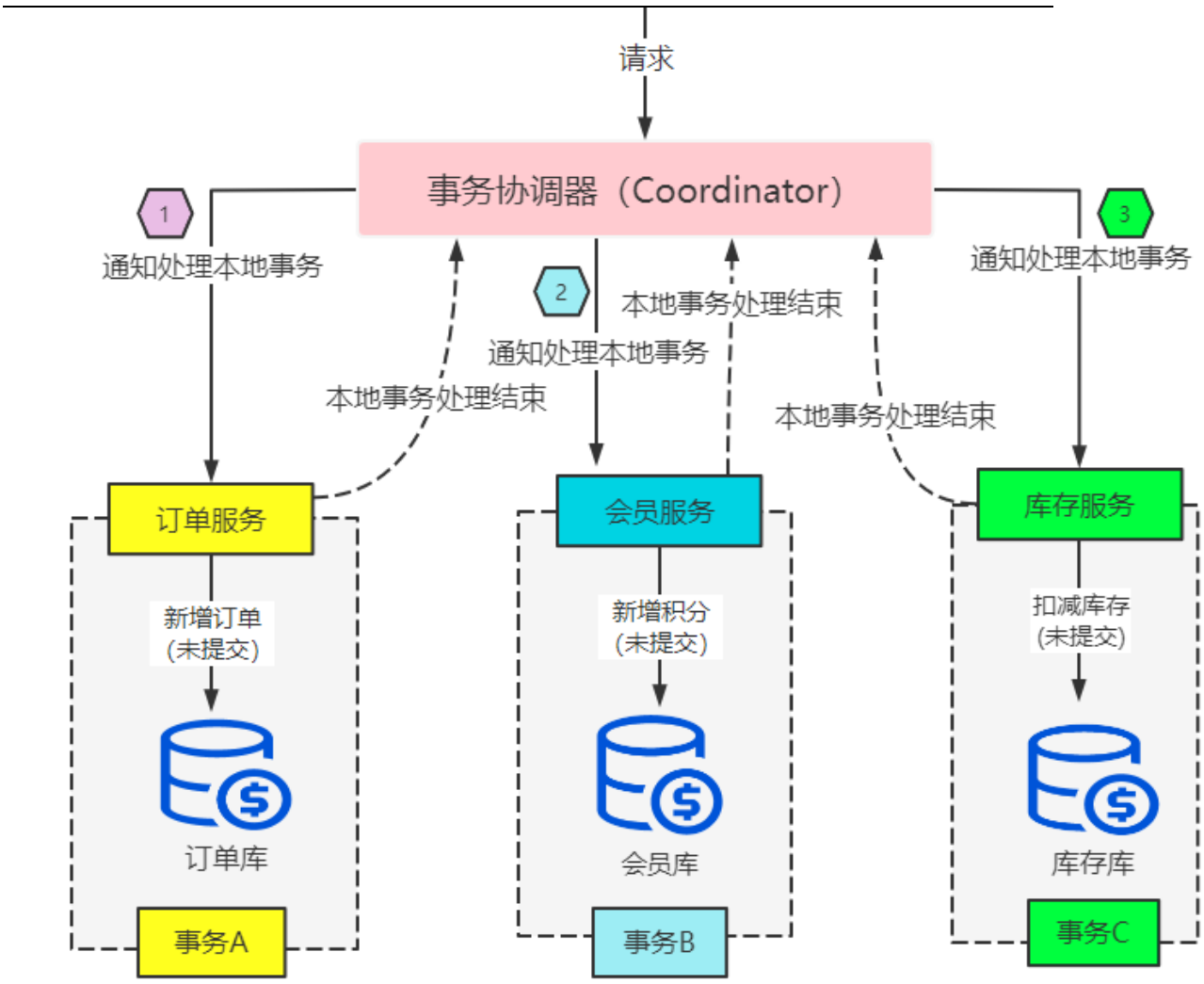
9.3 BASE 理论是什么？

Base 全称是 Basically Available（基本可用），Soft state（软状态），和 Eventually consistent（最终一致性）三个短语的缩写，BASE 理论是对 CAP 中一致性和可用性权衡的结果，核心思想是：即使无法做到强一致性，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（通常可以将这种事务方式理解柔性事务）。

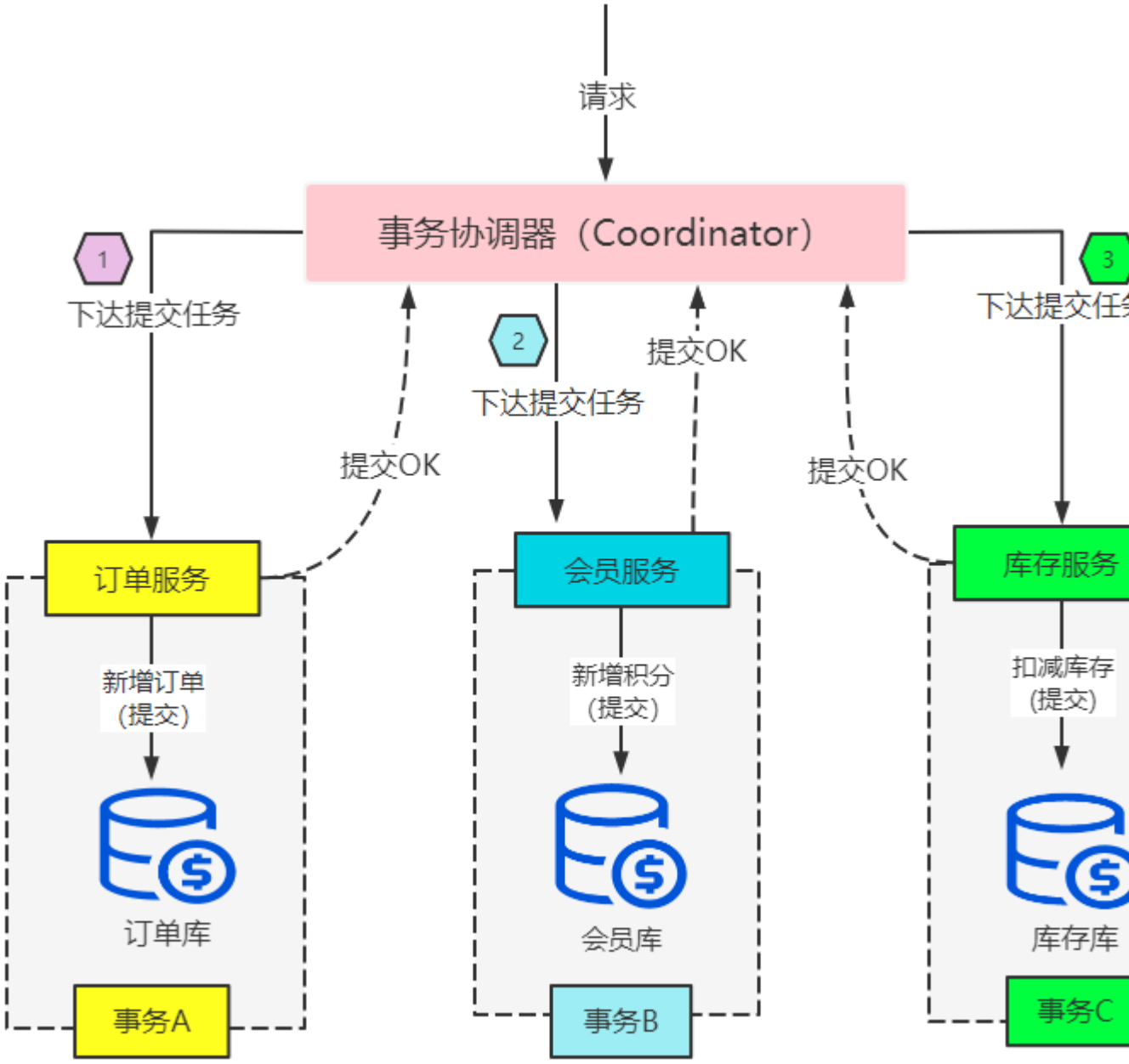
9.4分布式事务中的二阶段提交？

分布式事务中增加了一个新的角色，事务协调者（Coordinator），它的职责就是协调各个分支事务的开启与提交、回滚的处理。

当商城应用订单创建时，首先事务协调者会向各服务下达“处理本地事务”的通知，所谓本地事务就是每个服务应该做的事情，如订单服务中负责创建新的订单记录；会员服务负责增加会员的积分；库存服务负责减少库存数量。在这个阶段，被操作的所有数据都处于未提交（uncommit）的状态，会被排它锁锁定。这个阶段也是二阶段提交中的第一阶段（也称之为预处理阶段），如图所示：



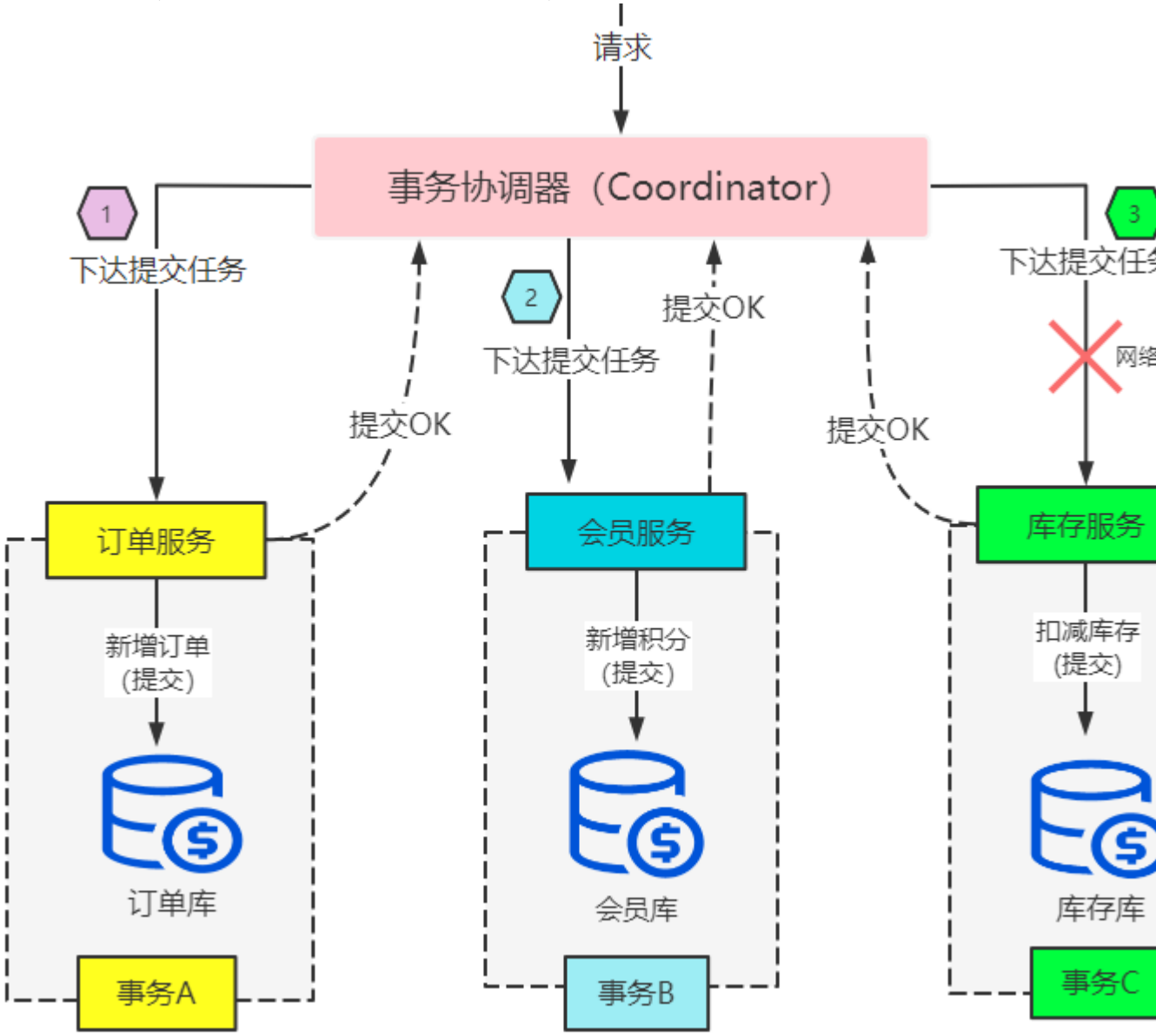
当本地事务都处理完成后，会通知事务协调者“本地事务处理完毕”。当事务协调者陆续收到订单、会员、库存服务的处理完毕通知后，便进入“阶段二：提交阶段”。



在提交阶段，事务协调者会向每一个服务下达提交命令，每个服务收到提交命令后在本地事务中对阶段一未提交的数据执行 Commit 提交以完成数据最终的写入，之后服务便向事务协调者上报“提交成功”的通知。当事务协调者收到所有服务“提交成功”的通知后，就意味着一次分布式事务处理已完成。
这便是二阶段提交的正常执行过程，但假设在阶段一有任何一个服务因某种原因向事务协调者上报“事务处理失败”，就意味着整体业务处理出现问题，阶段二的操作就自动改为回滚（Rollback）处理，将所有未提交的数据撤销，使数据还原以保证完整性。

9.5分布式事务二阶段提交有什么缺陷？

对于二阶段提交来说，它有一个致命问题，当阶段二某个服务因为网络原因无法收到协调者下达的提交命令，则未提交的数据就会被长时间阻塞，可能导致系统崩溃。



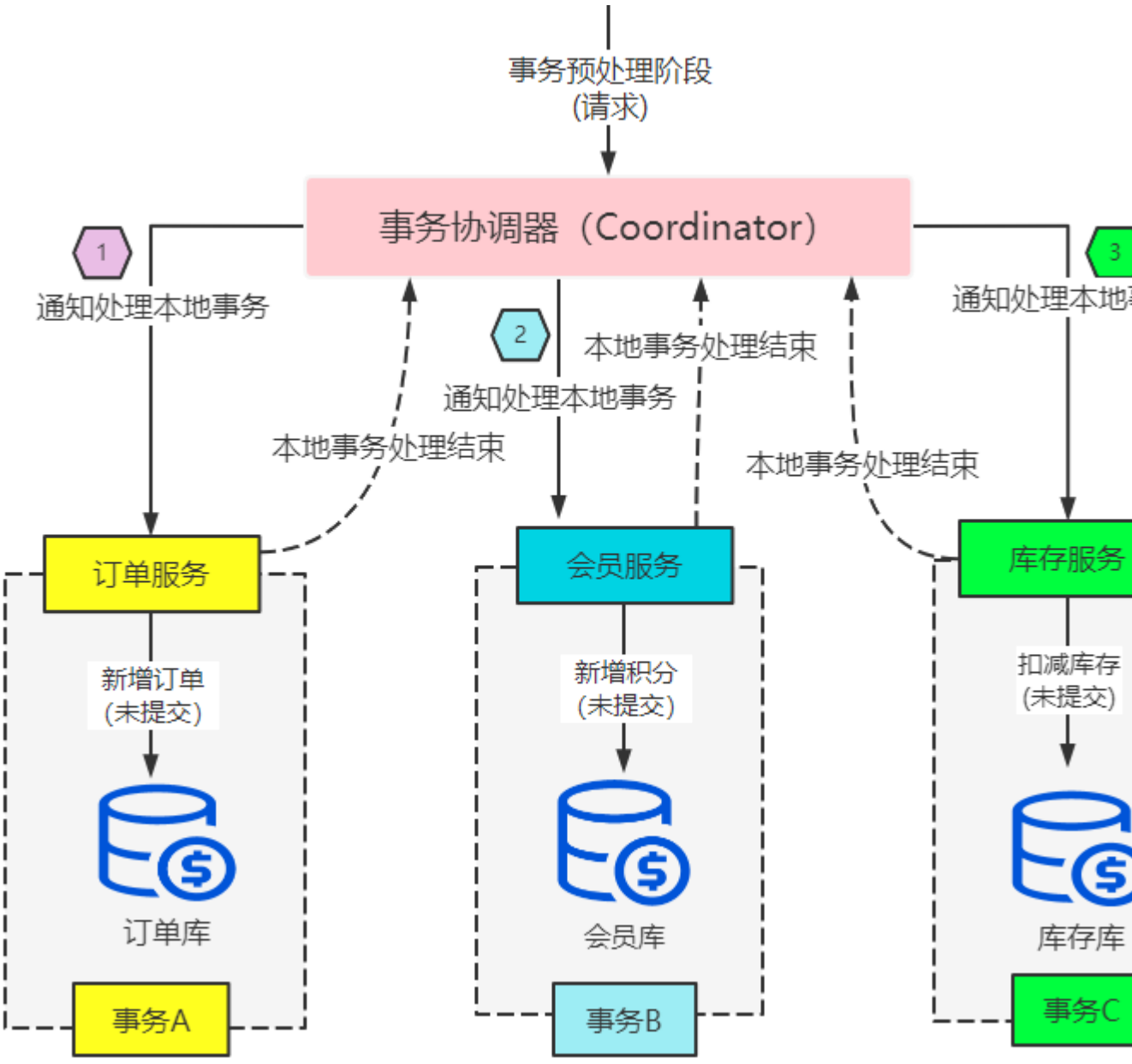
以上图为例，假如在提交阶段，库存服务实例与事务协调者之间断网。提交指令无法下达，这会导致商品库存记录会长期处于未提交的状态，因为这条记录被数据库排他锁长期独占，之后再有其他线程要访问“飞科剃须刀”库存数据，该线程就会长期处于阻塞状态，随着阻塞线程的不断增加，库存服务会面临崩溃的风险。

那这个问题要怎么解决呢？其实只要在服务这一侧增加超时机制，过一段时间被锁定的“飞科剃须刀”数据因超时自动执行提交操作，释放锁定资源。尽管这样做会导致数据不一致，但也比线程积压导致服务崩溃要好，出于此目的，三阶段提交（3PC）便应运而生。

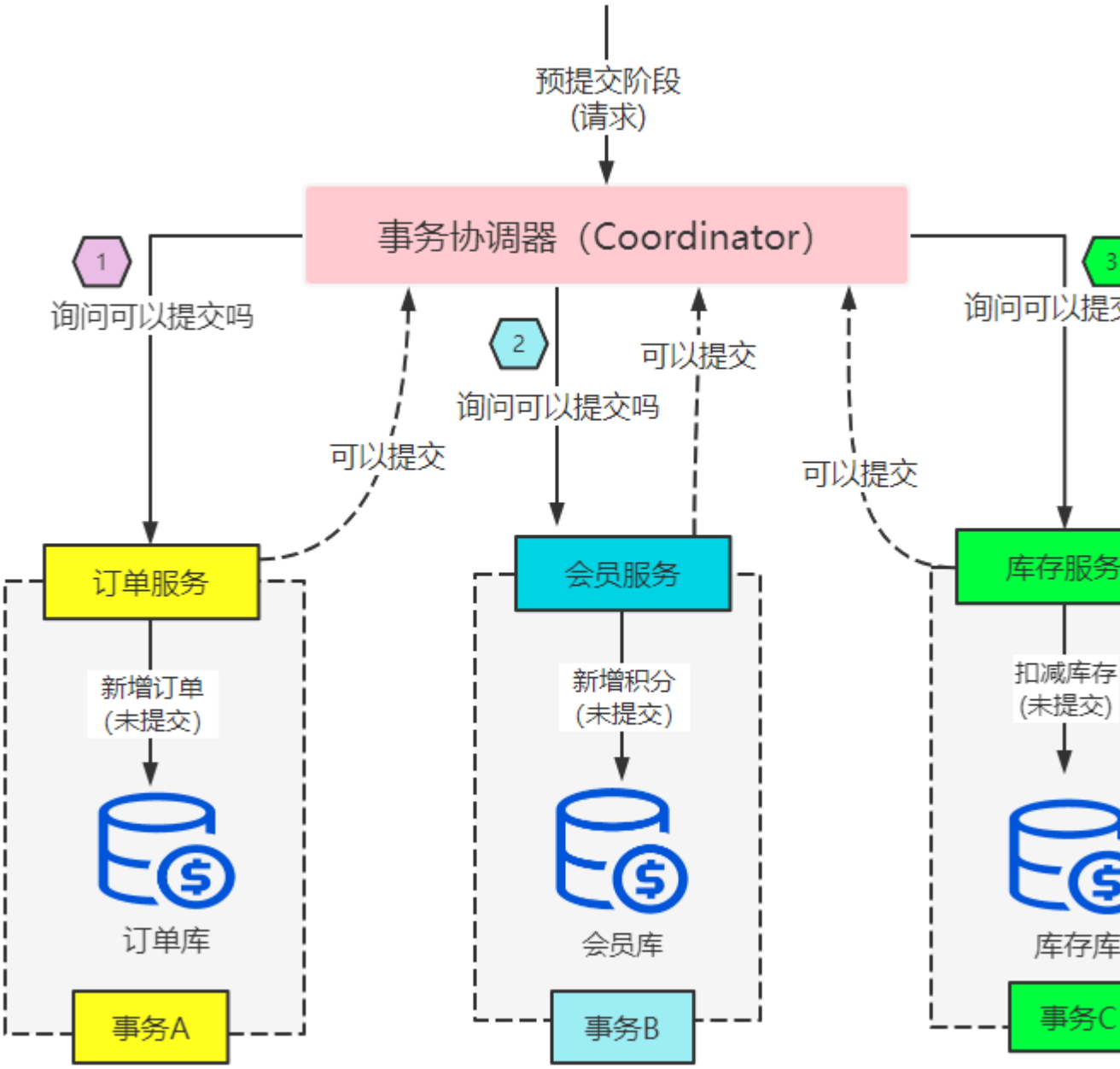
9.6分布式事务中的三阶段提交？

三阶段提交实质是将二阶段中的提交阶段拆分为“预提交阶段”与“提交阶段”，同时在服务端都引入超时机制，保证数据库资源不会被长时间锁定。下面是三阶段提交的示意流程：

阶段 1：事务预处理阶段，3PC 的事务预处理阶段与 2PC 是一样的，用于处理本地事务，锁定数据库资源，例如：



阶段 2:当所有服务返回成功后，进入阶段二。预提交阶段只是一个询问机制，以确认所有服务都已准备好,同时在此阶段协调者和参与者都设置了超时时间以防止出现长时间资源锁定。



当阶段二所有服务返回“可以提交”，进入阶段三“提交阶段”。
3PC 的提交阶段与 2PC 的提交阶段是一致的，在每一个数据库中执行提交实现数据的资源写入，如果协调者与服务通信中断导致无法提交，在服务端超时后在也会自动执行提交操作来保证资源释放。

三阶段提交本质上是二阶段提交的优化版本，主要通过加入预提交阶段引入了超时机制，让数据库资源不会被长期锁定，但这也会带来一个新问题，数据一致性也很可能因为超时后的强制提交被破坏，对于这个问题各大软件公司都在各显神通，常见的做法有：增加异步的数据补偿任务、更完善的业务数据完整性的校验代码、引入数据监控及时通知人工补录这些都是不错的补救措施。

9.7Seata 是什么

Seata 是一款开源的分布式事务解决方案，致力于在微服务架构下提供高性能和简单易用的分布式事务服务。它的官网是 http://seata.io/。

9.8Seata 提供了哪些分布式事务方案？

1) AT 模式（是 Seata 主推的分布式事务解决方案，对业务无侵入，真正做到业务与事务分离）
2) TCC 模式（对业务代码侵入性太强。没有 AT 模式全局锁，加锁逻辑需要根据业务自行实现。因此 TCC 的性能会比 AT 模式更好）
3) SAGA 模式（Saga 模式的正向服务和补偿服务都需要手动实现，因此有很强的侵入性。能保证隔离性，不容易进行并发控制）
4) XA 模式 （利用事务资源实现对 XA 协议的支持，是传统分布式强一致性的解决方案，性能较低，在实际业务中使用较少。）

9.9什么是 AT 模式事务？

AT(Automatic Transaction) 全自动事务，是 Seata 提供的一种事务方案。采用了简单易用且无侵入的事务处理机制，通过自动生成反向 SQL 实现事务回滚。

9.10 AT 模式事务有什么特点?

AT 模式是 Seata 独创的模式，它是基于 2PC 的方案，核心理念是利用数据库 JDBC 加上 Oracle、MySQL 自带的事务方式来对我们分布式事务进行管理。其主要特点如下：

1) 对业务无侵入，通过配置即可实现。
2) AT 事务使用一个数据源代理对象来实现自动化事务处理。
3) AT 事务执行分两个阶段：
   第一阶段：执行本地事务。
   第二阶段：全局事务提交，或全局事务回滚。
4) AT 事务使用 undo_log 表保存回滚日志(反向操作)，事务执行失败时，会根据回滚日志表来回滚数据。

9.11 Seata 的三大组件是什么？

Seata 有三个组成部分：

1) **第一个是事务协调者（TC）**，它的作用是维护全局和分支事务的状态，驱动全局事务提交或者回滚，这正是前面所说 2PC 或者 3PC 方案时提到的事务协调者组件的具体实现，TC 由 SEATA 官方提供。
2) **第二个是事务管理器（TM）**，事务管理器用于定义全局事务的范围，开始全局事务提交或者回滚全局事务都是由 TM 来决定。
3) **第三个是资源管理器（RM）**，他用于管理分支事务处理的资源，并且报告分支事务的状态，并驱动分支事务提交或者回滚。

9.12 Seata 事务协调器(TC)的作用是什么？

协调各个模块的执行，例如：
- 启动全局事务。
- 收集各模块第一阶段的运行状态。
- 向各个模块发送第二阶段提交或回滚指令。

9.13 Seata 事务管理器(TM)的主要作用是什么？

负责主要全局事务决策，例如：
- 向协调器申请启动全局事务。
- 收集所有模块第一阶段的运行状态，并对全局事务的成功或失败进行决策。
- 将决策结果提交给事务协调器。

9.14 资源管理器(RM)的作用是什么？

在各个模块中与协调器通信，并控制第二阶段的执行：
- 在每个模块中向协调器注册分支事务。
- 向协调器提交第一阶段的执行状态。
- 接收协调器第二阶段的指令，控制执行二阶段的提交或回滚。

9.15 AT 事务中数据源代理对象作用是什么？

- 自动保存事务回滚日志。
- 第二阶段提交时，执行根据回滚日志回滚数据，并删除日志。
- 第二阶段回滚时，删除回滚日志。

9.16 说说 TCC 模式的事务？

TCC 模式，全称 Try-Confirm-Cancel，通过名称也能看出来其流程主要有三个步骤：

1) 预处理 Try：实现业务检查和资源预留
2) 确认/提交 Confirm：业务确认和提交
3) 撤销/回滚 Cancel：业务回滚

TCC 模式本身就是二阶段提交的一种改进，不一样的是，这次就没有 AT 模式那么方便了，因为他需要我们自己写代码来实现了。Seata 中的 TCC 模式的实现关键在于拆分二阶段，也就是如何把一步操作拆分为两步，比如库存扣减，本身就是一个 update 语句，但是 TCC 下却需要我们拆分为先冻结库存，然后再扣减这部分库存

10 案例分析：

10.1 交易系统服务拆分问题？

Xxx 面试者提到负责了一个交易系统的实现，并将系统拆分成了报价系统、促销系统、订单系统。那现在的问题是，为什么要进行系统拆分，而且拆分后带来的其它复杂度，你是怎么考虑的？

答复：

1） 从订单系统层面来看，由于交易流程中的订单系统相对来说业务稳定，不存在很多的迭代需求，如果耦合到整个交易系统中，在其他功能发布上线的时候会影响订单系统，比如订单中心的稳定性。基于这样的考虑，需要拆分出一个独立的子系统。
2） 从促销系统层面来看，由于促销系统是交易流程中的非核心系统，出于保障交易流程稳定性的考虑，将促销系统单独拆分出来，在发生异常的时候能让促销系统具有可降级的能力。
3） 从报价系统层面来看，报价是业务交易流程中最为复杂和灵活的系统，出于专业化和快速迭代的考虑，拆分出一个独立的报价系统，目的就是为了快速响应需求的变化。

最后，从复杂度评估层面来看，系统拆分虽然会导致系统交互更加复杂，但在规范了 API 的格式定义和调用方式后，系统的复杂度可以维持在可控的范围内。

我觉得，这样的回答很好地表达了应聘者对系统设计的思考与理解。因为他说出了原有系统中关于订单、促销和报价功能耦合在一起带来的实际问题，这是**立足于点，**又从交易流程的角度做系统设计串联起三个系统的拆分逻辑，这是**连接成线**，最后从复杂度和成本考量的方向夯实了设计的原则，这是**扩展成面。**

10.2 点评系统逻辑设计是怎样的？

在电商中，当用户发表一条商品评论，后台的逻辑是，点评系统会调用一系列的远程 API 接口，如调用风控系统、广告系统、图片系统、消息系统等很多个外部系统接口，这样的逻辑如何实现？

你可能回说，我引入了 MQ 消息队列，做了系统解耦，采用异步消息通知的方式来触发系统调用。

面对类似问题，我觉得应该考虑这样几个层面？
第一：复杂度问题？（功能复杂度-耦合和非功能复杂度-高可用）
1) 功能上假如这些系统之间的通讯是直接的 RPC，那就意味着耦合可能会比较大，不利于扩展。
2) 非功能上(高性能-TPS/QPS、高可用)

第二：有哪些可选解决方案？
1） 引入第三方消息队列 （会带来新的复杂度）
2） 基于 Redis 实现消息队列
3） 基于内存中数组实现消息队列

第三：你评估的标准是什么？
1） 功能复杂度
2） 非功能复杂度(无单点、可水平扩展、可降级)

第四：说一下你的技术实现？

当你在确定了具体的架构解决方案之后，需要进一步说明你技术上的落地实现方式和原理，假如你最终选择基于 Redis 来实现消息队列，那么可以有几种实现方式？各自的优缺点有哪些？对于这些问题，要做到心里有数。比如，基于 Redis List 的 LPUSH 和 RPOP 的实现方式、基于 Redis 的订阅或发布模式，或者基于 Redis 的有序集合（Sorted Set）的实现方式等等。