厦门大学信息学院2021级中间件技术期末试卷

（复现）

概念题　　5题　　4分
简答题　　8题　　5分
综合题　　2题　　10分
开发设计题1题　　20分

概念题

1. 什么是中间件？

答:中间件是处于系统软件(操作系统和网络软件)和应用软件之间的**一种起连接作用的分布式软件**，它能使应用软件之间进行**跨网络的协同工作(也就是互操作)**。

2. 比较图解释容器和vm的区别

答：虚拟化是一种通过在物理硬件上创建多个虚拟环境，使多个操作系统或应用程序能够共享同一硬件资源的技术，通过抽象硬件资源来提高资源利用率和灵活性，减少硬件需求，增强系统的隔离性和安全性，实现快速部署和弹性扩展，简化集中管理和灾难恢复，降低IT基础设施的总拥有成本，并广泛应用于服务器、桌面、存储和网络等领域。

容器:为特定组件提供服务的运行时环境，比如JAVA虚拟机，封装底层J2EEAPI服务，容器可以管理对象的生命周期、对象与对象之间的依赖关系。

虚拟机:虚拟整个计算机，包括OS和磁盘，一台物理计算机上模拟多台计算机运行任务

容器:在OS之上的虚拟，容器共享OS

两个基本的手段：

比较图解释不同（画图看ppt），虚拟机(VM)是计算机系统的仿真，它可以在计算机硬件上运行看似很多单独的计算机，每个VM都需要自己的底层操作系统，并虚拟化硬件；容器不像VM那样虚拟化底层计算机，而只是虚拟化操作系统，其位于物理服务器及其主机系统之上(通常是Linux或Windows)，每个容器共享操作系统内核。

3. 两段锁协议

两端锁协议:所有的事务必须分两个阶段对数据项加锁和解锁。即**事务分两个阶段**，第一个阶段是获得锁。事务可以获得任何数据项上的任何类型的锁，但是不能释放:第二阶段是释放锁，事务可以释放任何数据项上的任何类型的锁，但不能申请。

获得锁的阶段称为扩展阶段，在这个阶段可以进行加锁操作，**对任何数据进行读操作之前要申请获得S锁，在进行写操作之前要申请并获得X锁，加锁不成功，则事务进入等待状态，**直到加锁成功才继续执行。加锁后无法解锁。

释放锁的阶段称为收缩阶段，当事务释放一个锁后，事务进入收缩阶段，在该阶段只能解锁不能加锁。

4. +5.英语解释。IDL、IIOP、WebService、CORBA

答:

IDL(Interface Definition Language):是用于描述分布式对象接口的定义语言，利用 IDL 进行接口定义之后，就确定了客户端与服务器之间的接口，这样即使客户端和服务器独立进行开发，也能够正确地定义和调用所需要的分布式方法。**定义 CORBA 接口**

IIOP(Internet Inter. ORB Protocol)，互联网内部对象请求代理协议，它是一个用于 CORBA 2.0 及兼容平台上的协议。用来在 CORBA 对象请求代理之间交流的协议。Java 中使得程序可以和其他语言的 CORBA 实现互操作性的协议。**支持两阶段提交 如何在 TCP /IP 网络交换 GIOP 信息**

Web 服务(Web service)是一种面向服务的架构的技术，通过标准的 Web 协议提供服务，目的是保证不同平台的应用服务可以互操作。是无状态的。客户端和服务器之间的交互在请求之间是无状态的。

CORBA：扩展了 RPC 完全面向对象，底层是 IIOP 协议，难以穿越防火墙。

简答题

1. 什么是负载均衡？举例说明负载均衡的作用&使用场景。

答:负载均衡是指将工作任务、访问请求等负载进行平衡，分摊到多个服务器和组件等操作元上进行执行，是解决高性能，单点故障，提高可用性和可扩展性，进行水平伸缩的终极解决方案。

负载哪些层面？可以是 chip 芯片层面、任务在 CPU 中的分配和 GPU 层面、任务、网络、web 请求、数据库请求（查询）。

处理的负载在 CPU 和 GPU 之间的分配。

负载均衡几种方式:轮询，dns

轮叫调度(1: 1)、加权轮叫调度、最少链接调度、加权最小链接调度。

2 层(数据链路层) :采用虚拟 IP (VIP)，集群中不同的机器采用相同 IP 地址

3 层(网络层) VIP,但是集群中不同的机器采用不同的 IP 地址。

4 层(传输层)修改 IP+端口号

7 层(应用层) DNS HTTP Redis 等

2. 从技术和商业角度解释计算架构演进。怎么演进？为什么会这么演进？

答：演进顺序:单机——C/S、 B/S——分布式——云计算——端管边云融合，其背后驱动力有（4 个:单机->网络->云->端云）

技术逻辑：计算要求、计算分配

商业逻辑：单位计算资源越来越便宜，按量计价如果再加一句，就是谁掏钱的问题，是由 Service Provider (消息提供者)来掏钱，还是由 Consumer (客户)来掏这些计算资源的钱。

(C/S 架构:客户端/服务器架构，每一个客户端的实例都可以向一个服务器发出请求；B/S 架构:与 C/S 不同，客户端不需要安装专门的软件，只需要浏览器就可。浏览器和 Web 服务器交互，Web 服务器和后端数据库交互。)

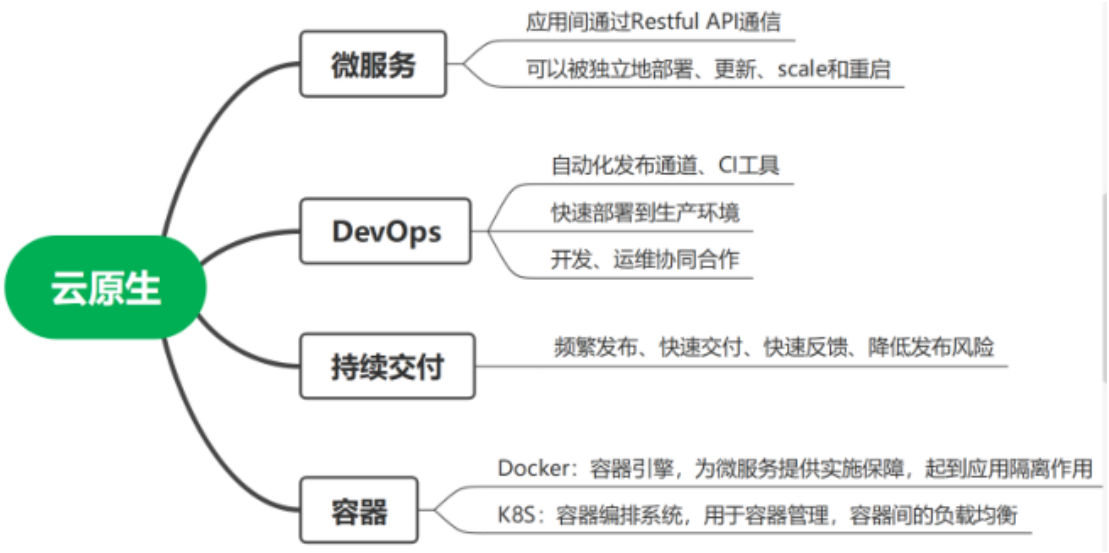
3. 解释云原生、AI 原生的概念。举例子云原生，云原生的概念怎么实现。相对传统技术的特点。

答：云原生是一种构建和运行应用程序的方法，是一套技术体系和方法论。**云原生(CloudNative) 是一个组合词，Cloud+Native。** Cloud 表示应用程序位于中，而不是传统的数据中心；Native 表示应用程序从设计之初即考虑到云的环境，原生为云而设计，在云上以最佳姿势运行，充分利用和发挥云平台的弹

性+分布式优势。

AI 原生应用（AI Native）思维是一种基于 AI 构建和运行应用程序的方法，是一套技术体系和方法论。AI 表示应用程序需要更好的利用 AI 能力和适应 AI，而不是传统的 AI 来适应人；Native 表示应用程序从设计之初即考虑到 AI 的应用和能力，原生为 AI 而设计，在用好 AI 以最佳姿势运行，充分利用和发挥 AI 大模型的智能+CUI 优势，提高应用程序的智能化水平。

四个特点：DevOps+ 持续交付+微服务+容器。



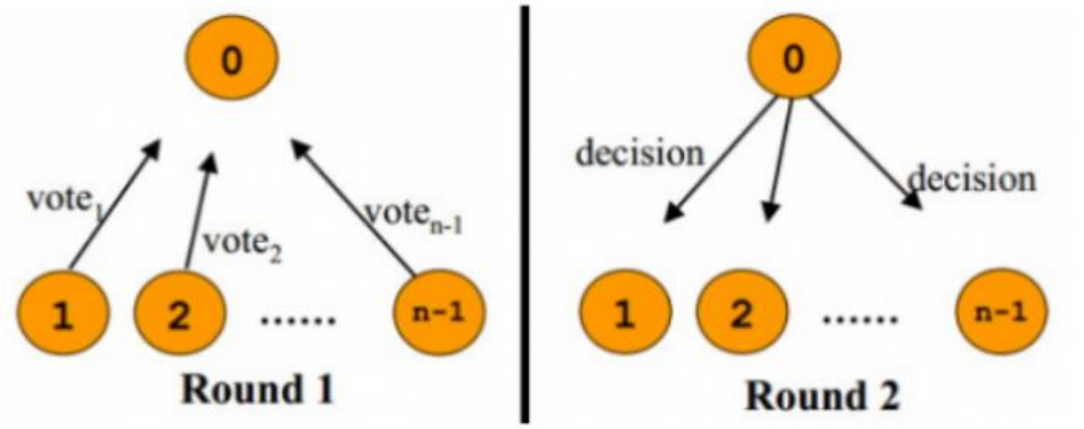
总而言之，符合云原生架构的应用程序应该是:采用开源堆栈(K8S+Docker) 进行容器化，基于微服务架构提高灵活性和可维护性，借助敏捷方法、DevOps 支持持续迭代和运维自动化，利用云平台设施实现弹性伸缩、动态调度、优化资源利用率。

4. 说明两段式提交（画 ppt 的图）。作用。解释典型的异常（3 例子）和异常处理方法。

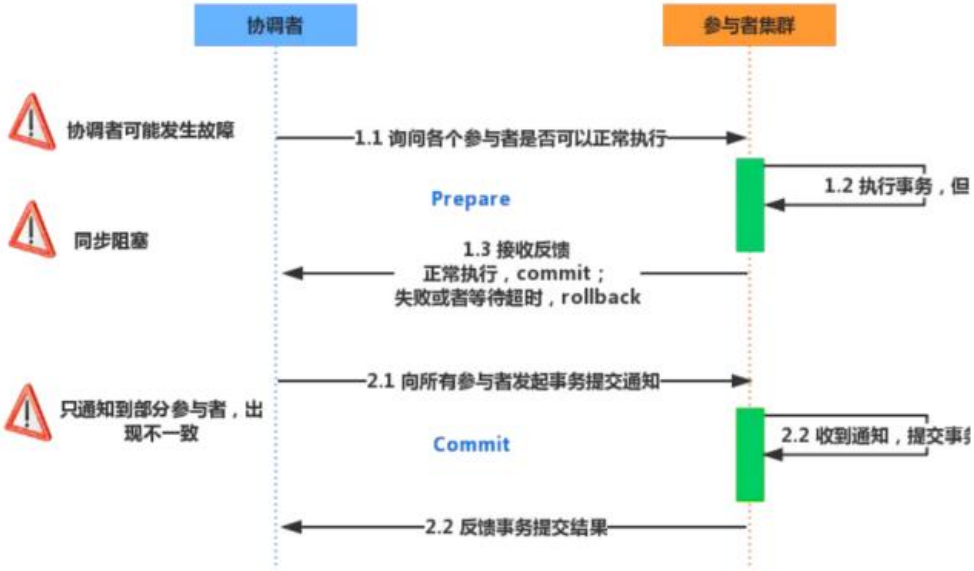
答：在分布式系统中，事务往往包含有多个参与者的活动，单个参与者上的活动是能够保证原子性的，而多个参与者之间原子性的保证则需要通过两阶段提交来实现，**两阶段提交是分布式事务实现的关键。**

在分布式事务两阶段提交协议中，有一个主事务管理器负责充当**分布式事务协调器**的角色。

事务协调器负责整个事务，并使之与网络中的其他事务管理器(**参与者**)协同工作协调者可以看做成事务的发起者，同时也是事务的一个参与者。对于一个分布式事务来说，一个事务是涉及到多个参与者的。



时间轴出现的特殊异常情况的处理，按时间轴列举出异常情况以及如何处理



提交请求(投票)阶段

- 协调者向所有参与者发送 prepare 请求与事务内容，询问是否可以准备事务提交，并等待参与者的响应。
- 参与者执行事务中包含的操作，并记录 undo 日志(用于回滚)和 redo 日志(用于重放)，但不真正提交。
- 参与者向协调者返回事务操作的执行结果，执行成功返回 yes，否则返回 no。

提交(执行)阶段

1 成功情况：若所有参与者都返回 yes，说明事务可以提交

- 协调者向所有参与者发送 commit 请求
- 参与者收到 commit 请求后，将事务真正地提交上去，并释放占用的事务资源，并向协调者返回 ack
- 协调者收到所有参与者的 ack 消息，事务成功完成

2 失败情况：若有参与者返回 no 或者超时未返回，说明事务中断，需要回滚

- 协调者向所有参与者发送 rollback 请求
- 参与者收到 rollback 请求后，根据 undo 日志回滚到事务执行前的状态，释放占用的事务资源，并向协调者返回 ack
- 协调者收到所有参与者的 ack 消息，事务回滚完成

5. 接口。控制反转 & 依赖注入。说明作用和使用场景。

接口是一种定义一组方法规范的抽象类型。接口不包含方法的实现，而是由具体的类来实现这些方法。接口用于实现多态性和解耦。

依赖注入是一种设计模式，它允许将对象的依赖关系注入到对象中，而不是在对象内部创建依赖。这样可以实现更好的解耦和可测试性。

DI Dependency Injection。比如 A 用到了 B 类的对象 b， 但 A 不需要 new 一个 b，而是定义一个 B 的引用，由容器在外部 new，并注入到 A 类的引用中。减少了类里面实例化的次数。

控制反转是一种设计原则，它将对象的控制权从内部转移到外部容器或框架，典型实现方式是依赖注入（DI）。IoC 容器负责创建和管理对象的生命周期以及它们之间的依赖关系。

通过接口、依赖注入和控制反转，可以实现灵活、可扩展和易于测试的应用程序设计。这种设计模式有助于减少代码的耦合度，提高代码的可维护性和可测试性。

场景：增强灵活性：依赖注入使得依赖关系可以在运行时动态配置，支持不同的实现类和配置需求。例如，日志服务可以通过依赖注入切换不同的日志框架（如 Log4j、SLF4J）。

6. 与单体应用的结构相比，微服务架构具有哪些特点。

答：a. 传统架构存在的问题

- 项目过于臃肿，部署效率低下
- 开发成本高
- 无法灵活扩展

b. 微服务的特点

- 服务拆分粒度更细

微服务可以说是更细维度的服务化，小到一个子模块，只要该模块依赖的资源与其他模块都没有关系，那么就可以拆分为一个微服务。

- 服务独立部署

传统的单体架构是以整个系统为单位进行部署，而微服务则是以每一个独立组件为单位进行部署。

- 服务独立维护，分工明确

每个微服务都可以交由一个小团队进行开发，测试维护部署，并对整个生命周期负责，当我们将每个微服务都隔离为独立的运行单元之后，任何一个或者多个微服务的失败都将只影响自己或者少量其他微服务，而不会大面积地波及整个服务运行体系。

（同微服务架构、微服务和服务的区别）

7. 解释命名服务中的黄页、白页和绿页

答：白页：命名服务，通过外部名字定位构件。

黄页：目录服务，通过服务特性定位构件。

绿页：合约服务，通过技术规范定位构件。

8. 高内聚和低耦合。高扇入和低扇出？

高内聚和低耦合 是软件设计的两个重要原则，其中高内聚指的是模块或类中的所有功能都紧密相关，完成单一任务或职责，提升代码的可维护性和可重用性；低耦合指的是模块或类之间的依赖关系较少，彼此独立，修改一个模块不会影响其他模块，从而提高系统的灵活性和稳定性。**高扇入和低扇出** 描述的是模块间的依赖关系，高扇入指的是有很多其他模块依赖于该模块，表示其重要性和稳定性；低扇出指的是该模块依赖于较少的其他模块，表示其独立性和简化的依赖关系。

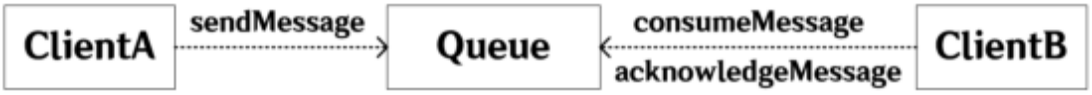
综合题

1. 消息型中间件中的 Topic 和 Queue。如何使用消息型中间件实现解耦、异步和削峰。

答： P2P (Point to Point) 点对点模型（Queue 队列模型）
Publish/Subscribe(Pub/Sub) 发布/订阅模型(Topic 主题模型)

a. 点对点模型：生产者和消费者之间的信息往来

每个消息都被发送到特定的消息队列，接收者从队列中获取消息。队列保留着消息，直到他们被消费或超时。

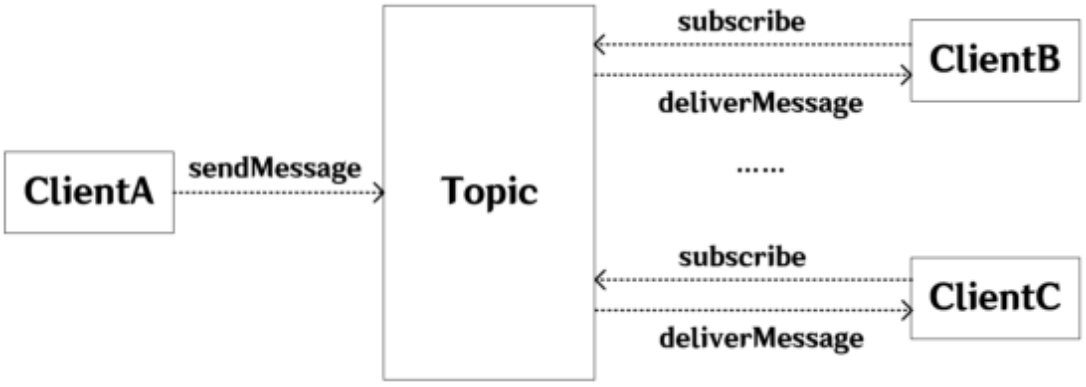


点对点模型的特点

- **每个消息只有一个消费者**（Consumer）(即一旦被消费，消息就不再在消息队列中)；
- **发送者和接收者之间在时间上没有依赖性**，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列；
- **接收者在成功接收消息之后需向队列应答成功**。

b. 发布订阅模型

包含三个角色：主题(Topic)，发布者(Publisher)，订阅者(Subscriber)，多个发布者将消息发送到 topic，系统将这些消息投递到订阅此 topic 的订阅者。



发布者发送到 topic 的消息，只有订阅了 topic 的订阅者才会收到消息。topic 实现了发布和订阅，当你发布一个消息，所有订阅这个 topic 的服务都能得到这个消息，所以从 1 到 N 个订阅者都能得到这个消息的拷贝。

发布/订阅模型的特点

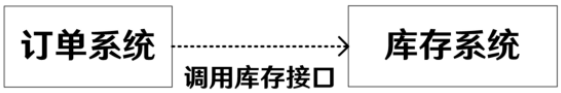
- 每个消息可以有多个消费者

- 发布者和订阅者之间有时间上的依赖性（先订阅主题，再来发送消息）
- 订阅者必须保持运行的状态，才能接受发布者发布的消息

a. 解耦

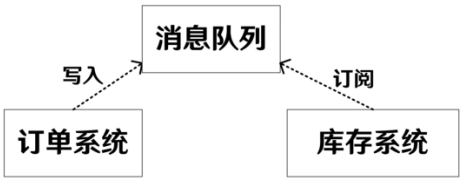
场景说明：用户下单后，订单系统需要通知库存系统。

传统做法：订单系统调用库存系统的接口



传统模式的缺点：假如库存系统无法访问，则订单减库存将失败，从而导致订单失败，订单系统与库存系统耦合。

消息队列解决方案



订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功

库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作

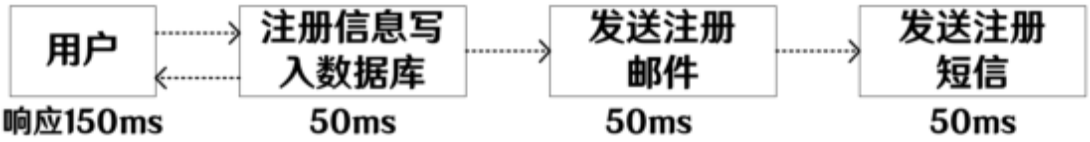
假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

b. 异步

场景说明：用户注册，需要执行三个业务逻辑，分别为写入用户表，发注册邮件以及注册短信。

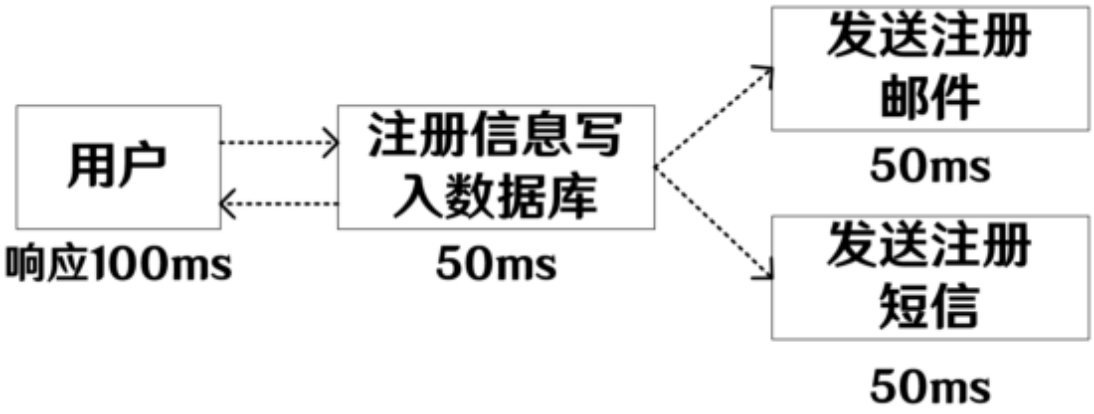
串行方式

将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端。



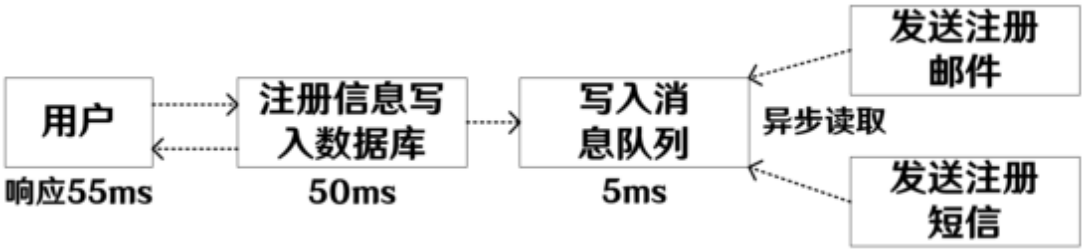
并行方式

将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间。



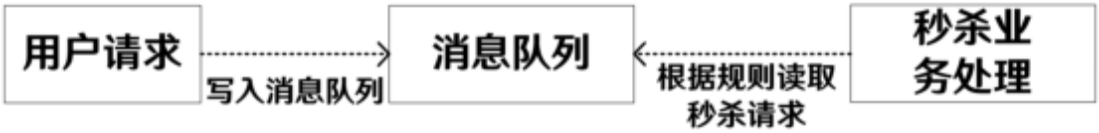
异步处理

引入消息中间件，将部分的业务逻辑，进行异步处理。改造后的架构如下：



c. 削峰

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛。应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。秒杀业务根据消息队列中的请求信息，再做后续处理

2. 技术和商业角度阐述微服务产生的原因和意义&未来发展的看法。

答：从技术角度，微服务通过将应用拆分为小型、独立部署的服务，提升了系统的灵活性、可扩展性和维护性，解决了单体应用的复杂性和扩展瓶颈；从商业角度，微服务能够加快开发速度、提高生产效率和响应市场需求的能力，促进业务创新和竞争力的提升。 未来，随着云计算、容器化和 DevOps 的发展，微服务将继续演进，推动企业数字化转型，进一步增强系统的弹性和自动化管理能力。

开放设计题

秒杀&大规模电子商务为背景

1. 假如微服务架构&哪几个功能可以做成微服务，举 3 个例子说明。

答：需要遵循的原则:高内聚低耦合，服务粒度适中(横向拆分)、围绕业务概念建模(确定功能边界)、演进式拆分(先粗后细)

在微服务架构中，可以将不同的业务功能拆分为独立的微服务，每个微服务围绕特定的业务功能构建，具有高度的自治性和灵活性。结合以上内容，以下是五个可以做成微服务的功能示例：

1. 用户管理服务（User Management Service）

功能：

- 处理用户的注册、登录和管理。
- 提供用户信息的查询和修改功能。
- 处理用户认证和授权。

2. 订单管理服务（Order Management Service）

功能：

- 处理用户的订单请求，包括订单创建、查询和取消。
- 管理订单的状态（如待支付、已支付、已发货、已完成）。
- 处理订单的支付和退款。

3. 库存管理服务（Inventory Management Service）

功能：

- 管理产品库存信息，包括库存的查询和更新。
- 处理库存的锁定和释放，确保库存的一致性。
- 监控库存水平，自动生成补货请求。

4. 支付服务（Payment Service）

功能：

- 处理支付请求，与支付网关集成。
- 支持多种支付方式（如信用卡、PayPal、支付宝等）。
- 处理支付状态的更新和支付记录的管理。

5. 通知服务（Notification Service）

功能：

- 处理系统中的通知和消息推送。
- 支持多种通知渠道（如邮件、短信、推送通知等）。
- 处理通知的发送和状态跟踪。

2. 中间件的角度说明哪些层次采用哪些方案，可以应对短时间大容量的需求。举 3 个例子。

答：

1. Web 层

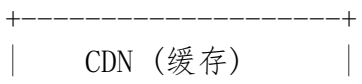
问题：高并发请求导致 Web 服务器负载过高，响应速度慢，甚至导致服务器崩溃。

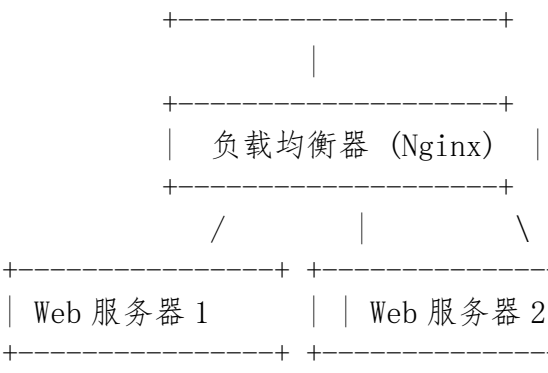
解决方案：使用负载均衡和缓存策略。

设计：

- **负载均衡**：通过负载均衡器（如 Nginx、HAProxy）将请求分发到多个 Web 服务器实例，避免单点过载。
- **缓存**：使用反向代理缓存（如 Varnish、Nginx 缓存）和内容分发网络（CDN）缓存静态内容，减轻 Web 服务器的负载。

示例：





在这个设计中，负载均衡器将请求分发到多个 Web 服务器，而 CDN 和反向代理缓存则减轻了服务器的负载，提升了响应速度。

2. 应用层

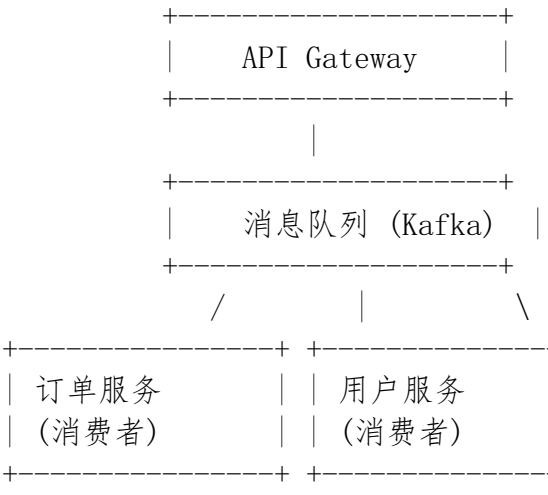
问题：应用层处理大量并发请求时，服务响应时间延长，可能导致超时或失败。

解决方案：使用消息队列和异步处理。

设计：

- **消息队列**：通过消息队列（如 RabbitMQ、Kafka）将请求进行排队，异步处理，平滑处理高并发请求。
- **异步处理**：应用服务将请求发送到消息队列，后台工作进程从队列中读取请求并处理，确保系统稳定性。

示例：



在这个设计中，API Gateway 将请求发送到消息队列，订单服务、用户服务和支付服务作为消费者从队列中读取请求并处理，平滑了高峰期间的请求处理。

3. 数据层

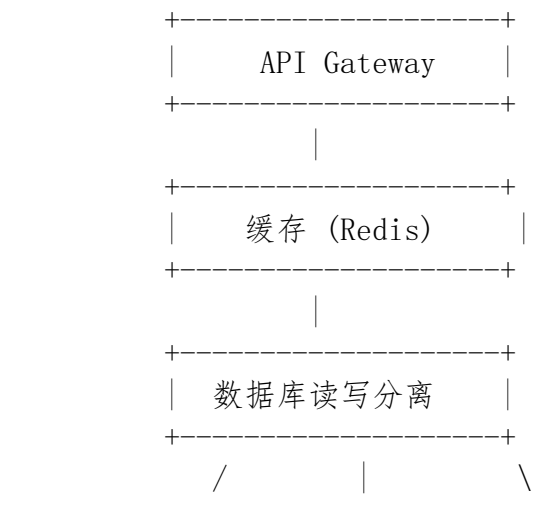
问题：高并发读写请求导致数据库压力过大，响应时间延长，甚至导致数据库崩溃。

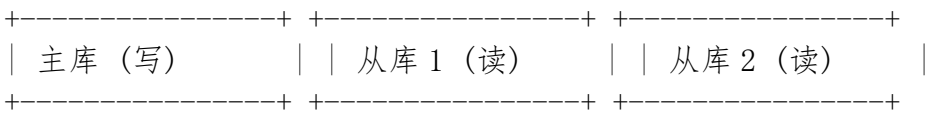
解决方案：使用数据库读写分离和缓存。

设计：

- **读写分离**：通过主从复制（如 MySQL Replication）实现读写分离，主库处理写请求，从库处理读请求，减轻主库压力。
- **缓存**：使用分布式缓存（如 Redis、Memcached）缓存频繁访问的数据，减少数据库查询次数。

示例：





在这个设计中，API Gateway 将读请求优先发送到缓存，缓存未命中时再查询从库，而写请求直接发送到主库。通过读写分离和缓存，显著提高了数据库的处理能力和响应速度。