# 分布式面试题

**分布式优缺点**

优：增大系统业务量和高并发下的性能

一台服务器坏了只会让部分服务停止，其他服务器可以正常使用

服务被拆分成各个模块，重用度更高

缺：结构复杂，设计和部署比较难，维护难度增大

**分布式锁：解决分布式环境下进程维度的并发问题**

分布式系统中，多个节点同时发起一个任务调度请求，就需要使用分布式锁来保证只有一个节点能够成功获取到任务调度的锁，避免任务被多次执行

**分布式事务**

分布式事务中，需要保证不同节点对共享资源的访问是互斥的，否则会导致数据不一致，使用分布式锁来保证在事务执行过程中，各个节点之间对共享资源的访问是互斥的

**分布式缓存**

分布式缓存中，需要保证各个节点对缓存的访问是互斥的，否则会导致缓存中的数据不一致，可以使用分布式锁来保证各个节点对缓存的访问是互斥的

**超卖问题**

电商等场景中，需要对某个商品的购买数量进行限制，如果多个用户同时发起购买请求，那么就需要使用分布式锁来保证只有一个用户能够成功获取到购买锁，避免超卖等问题

**redis分布式锁为什么要加超时时间**

一个服务端一直持有这个锁，其他线程就无法获取，会发生死锁，所以需要设置超时时间，过期后自动释放锁

网络抖动:客户端A中的线程获取到了锁，然后执行finally中的释放锁的代码时，网络出问题，导致客户端A没有成功释放锁。redis服务端来会一直把锁给客户端A，其他客户端就不能获取到这个锁

redis宕机:客户端A获取锁，Redis服务器突然宕机，锁没有释放。等到Redis再次恢复的时候，Redis服务端还会把锁给到客户端A，这样也会发生锁死的情况

**使用分布式锁需要注意的点**

互斥性：任意时刻只有一个客户端可以获取锁

防死锁：假如客户端在持有锁的时候崩溃了，没有释放锁，那么别的客户端无法获得锁，会造成死锁，要保证客户端一定会释放锁。Redis中可以设置锁的过期时间来保证不会发生死锁

持锁人解锁：加锁和解锁必须是同一个客户端，客户端A的线程加的锁必须是客户端A的线程来解锁，不能解开别的客户端的锁

可重入：当客户端获取对象锁之后，这个客户端可以再次获取这个对象上的锁

**分布式锁加了超时时间，锁过期后，另一线程又能拿到锁，该怎么办**

用守护线程，快过期时如果任务还没结束，重置过期时间，主线程执行完成后，一并销毁守护线程

延长过期时间

**分布式事务，二阶段提交,用于更新多个数据库(强一致性)**

准备阶段：协调者开启事务，每个参与者记录日志，执行操作后向协调者发成功/失败消息

提交阶段：所有参与者都返回成功,则向参与者发送提交请求，参与者将本地事务提交

某个失败后，则向参与者发送回滚请求

协调者可能出现单点故障，存在提交请求由于网络延迟未被部分实例收到的问题

**三阶段提交(强一致性)**

询问阶段：协调者开启事务，询问参与者是否执行分布式事务，参与者返回同意/终止，尽可能早发现问题

准备阶段: 同上

提交阶段:同上，如果超时未收到所有参与者的已提交消息则成功，否则这发送回滚消息

**什么是TCC(强一致性)**

常用的分布式事务解决方案，try成功执行确认，失败执行回滚

Try:业务检查，进行业务校验和资源预留， A减100元，A,B都预留+100元，

Confirm:业务确认，对try阶段校验过的业务进行确认，A,B都预留-100, B+100

Cancel:回滚阶段，用于回滚try阶段执行的业务

**TCC空回滚**

由于网络原因try调接口失败，并未进行资源预留，但是执行了cancel回滚

解决方案：当try异常时，记录try是否成功调用了接口，如果没调成功cancel就走空方法

**TCC悬挂**

try执行某个服务A超时，然后执行cancel，但是服务A在此时执行完成了(预留了资源)，导致预留的资源无法释放

解决方案：额外记录分支事务记录表，执行try调用的接口前判断cancel是否已执行过，执行过则不再执行try

Seata中AT,TCC,Saga,XA模式

AT:事务对表数据修改，由Seata自动生成事务和二阶段提交/回滚

TCC:用户根据业务场景实现，try,Confirm,cancel，不知对表数据修改，如发送短信，无法回滚撤回需要在cancel中补发一条抱歉的短信等

Saga:适用长事务，

XA：

Nginx

**负载均衡是什么**

多个服务器，将请求均匀分配到每个服务器，提高服务器使用率和吞吐量，避免部分节点压力过大

**客户端负和服务端的负载均衡**

客户端负载均衡(正向代理)：客户端根据微服务列表，挑一个进行请求

服务端负载均衡(反向代理)：服务端只暴露一个接口，请求到服务端后进行负载均衡

**OpenFeign是客户端还是服务端做负载均衡**

客户端负载均衡，内部集成LoadBalancer，从微服务列表中根据负载均衡算法选一个服务节点进行请求

**gateway是客户端还是服务端负载均衡**

服务端负载均衡

**请求中的连接超时和读取超时**

连接超时：http请求三次握手建立连接的时间

读取超时：连接建立后发送数据给服务器到服务器响应的时间

**负载均衡算法**

静态：

**轮询**：按服务器列表顺序，循环的将请求发到服务器

**随机**：随机分发请求

**源Hash算法**：根据IP地址哈希后进行路由，相同用户的请求会到同个服务器(避免服务器上的缓存失效,如登录信息)

动态：服务器在时间T内的平均 CPU占用率， 内存，响应时间， 定时上报给负载均衡调度器，并计算出权重

**加权轮询**：根据权重，轮询到的多发些请求

**加权随机**：根据权重，随机到的多发点请求

**最少连接**：根据连接数分配权重，连接数少的多发点请求

**最快响应**：根据接口响应时间分配权重，响应快的多发点请求

**CAP理论**

一致性：一个分区修改数据后，其他分区数据都要一致

可用性：提供正常服务

分区容错性：允许分区之间通信失败

**最终一致性(弱一致性)**

允许短时间内不同分区的数据出现不一致，来换取高可用

**微服务优缺点**

优：

每个服务职责单一，高内聚，低耦合

每个服务有自己是数据库，压力小

每个服务有自己的服务器集群，数据库集群，吞吐量高，高可用

每个服务可用采用不同的框架，语言，开发更加灵活

缺：

服务器之间需要发送请求来调用，会影响部分性能

微服务设计起来比较复杂

需要更多人力来运维

**eureka和nacos的区别**

Eureka:提供服务注册和拉取，Eureka会定时向注册中心发送心跳，短期内没有发送心跳，就会直接剔除；如果不主动拉去服务，注册中心不会主动推送

**Eureka能做集群吗，可以相互注册吗**

能，能

**Nacos的功能**

**服务注册：**服务端启动时，将自生上报给注册中心，并定时发送心跳维持健康状态

**服务发现**：客户端向注册中心(nacos)发起请求，获取服务实例列表

**动态配置管理**：类似配置中心，可以配置多套环境，微服务的配置都从nacos配置中心拉去

****DNS服务**：根据实例权重分配请求，实现负载均衡**

****多环境隔离**：确保不同环境之间的配置和服务互不影响**

****心跳机制和主动探测****

**心跳：服务端主动定时发送心跳请求，表示自己正常运行，连续无心跳表示异常**

**主动探测：中心主动请求每个服务端的健康检查接口，判断服务端是否正常运行**

****Nacos服务注册和发现****

**Yml添加配置nacos地址和服务名称**

**@EnableDiscoveryClient 开启服务注册发现**

**首次调用后会从注册中心获取微服务列表缓存在本地，再根据实例ip调用**

****Nacos注册中心宕机后，远程调用是否还会成功?****

**是否首次调用，第一次调用微服务后，会从注册中心获取微服务实例列表并缓存在客户端(本地)，再根据实例的ip调用。如果缓存存在实例信息则可以调用成功;如果是首次调用，本地缓存没有实例信息则无法调用**

****Nacos配置中心和项目yml中存在相同的配置时，哪个优先级高****

**配置中心的优先级高**

****RestTemplate和openFeign的区别****

**RestTemplate要先获得微服务所有地址列表，然后挑一个发起请求**

**OpenFeign: 通过注解(封装了restTemplate)**

****熔断器实现有哪些****

**Hystrix和Resilience4j**

****服务熔断****

**当某服务的接口调用失败达到一定次数就不再调用，而是本地服务设置的默认值，等检测到服务正常后再恢复调用(每隔一段时间调用看是否正常)**

****熔断器有哪些状态****

**关闭(请求正常通过)**

**打开（请求多次失败后，执行熔断，之后不再请求，直接执行降级的逻辑，如返回缓存中默认值）**

**半开（一段时间后，允许少量请求发出，成功则关闭，失败则回到打开状态）**

# Zookeeper

**zookeeper集群为什么推荐奇数个节点部署**

最少3个节点: 才能存活数大于一半

3个节点更省资源：leader选举时，要求存活节点必须大于总结点的一半，3个节点最多可失效2个，4个节点也最多可失效2两个