# 消息队列

### 为什么使用消息队列，消息队列有什么优缺点？

消息队列核心有三个作用：解耦、异步、削峰

**解耦**：A通过接口向BCD三个系统发送数据，如果E也需要呢，C不需要呢，导致A与其他系统耦合度过高，如果适用MQ，A产生一条消息后发送到MQ，哪个系统需要直接去MQ消费。

**异步**：A收到请求在本地写库，还需要再BCD三个系统写库，三个系统依次写库耗时太长，如果适用MQ，A系统本地写库后连续发送3条消息到MQ，直接返回响应给用户，用户体验极好。

**削峰**：高并发每秒有5k+请求，mysql每秒最多扛2k左右请求，同时涌入大量请求会导致mysql系统崩溃，如果使用MQ，每秒5k请求写入MQ，A系统从MQ拉去请求，每秒拉取2K个，保证系统不会崩溃。

**缺点**：

系统可用性降低：外部依赖越多系统越容易挂掉

系统复杂性提高：需要考虑消息有没有重复消费，消息丢失，消息传递顺序等问题

一致性问题：A处理完请求返回成功，但是BCD三个系统中不能保证都执行成功

### Kafka、ActiveMQ、RabbitMQ、RocketMQ有什么区别以及适用那些场景

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 特性 | Kafka | ActiveMQ | RabbitMQ | RocketMQ |
| 单机吞吐量 | 10万级 | 万级 | 万级 | 10万级 |
| Topic数量对吞吐量影响 | Topic到几十到几百时吞吐量大幅度下降 |  |  | Topic可以达到几百到几千级别，吞吐量小幅度下降 |
| 失效性 | ms级 | ms级 | 微秒级 | ms级 |
| 可用性 | 非常高，分布式 | 高，主从架构的高可用 | 高，主从架构的高可用 | 非常高，分布式 |
| 消息可靠性 | 经过优化配置，0丢失 | 有较低概率丢失 | 基本不丢 | 经过优化配置，0丢失 |

### 消息队列如何保证高可用性？

RabbitMQ，是基于主从（非分布式）做高可用性

RabbitMQ有三种模式，单机模式、普通集群模式、镜像集群模式

**普通集群模式**：在多台机器上启动多个RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。

**镜像集群模式**（高可用性）

在镜像集群模式下，创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue的一个完整镜像，包含 queue 的全部数据。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。

### 如何保证消息不被重复消费，即如何保证消费的幂等性？

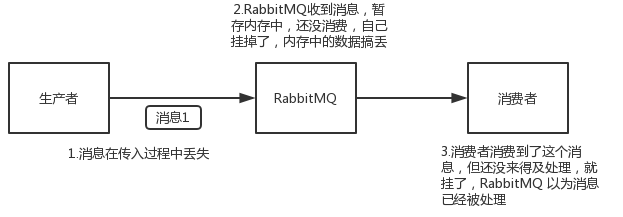
结合业务来：

如让生产者发送数据时候，里面加一个全局唯一id，类似订单id，消费到了之后，先根据id去redis查一下是否消费过，没消费过就处理，然后id写入reids，如果消费过则跳过；

再如通过数据库唯一键来保证数据不会重复插入多条

### 如何保证消息可靠性传输？（如何处理消息丢失）

数据丢失的3种情况：



**生产者丢失**：

方案1：开启RabbitMQ事务机制，不建议使用，太消耗性能

方案2：开启生产者confirm机制，消息写入RabbitMQ，如果成功接收会返回一个ack消息，如果没能处理，会返回一个nack，可以进行重发

**RabbitMQ丢失**：

开启RabbitMQ持久化，保证Queue和发送的消息都设置为持久化

**消费者丢失**：

关闭RabbitMQ的自动ack确认

### 如何保证消息的顺序性？

# 如何设计一个高并发系统？

**系统拆分**

将一个系统拆分为多个子系统，每一个系统连接一个数据库

**缓存**

大部分的高并发场景，都是读多写少，可以在数据库和缓存里都写一份，读的时候大量走缓存， redis 轻轻松松单机几万的并发。

**MQ**

MySQL扛不住高并发，用MQ大量的写请求灌入 MQ 里，排队慢慢玩儿，后边系统消费后慢慢写，控制在 mysql 承载范围之内。

**分库分表**

将一个数据库拆分为多个库，多个库来扛更高的并发；然后将一个表拆分为多个表，每个表的数据量保持少一点，提高 sql 跑的性能。

**读写分离**

大部分时候数据库是读多写少，没必要所有请求都集中在一个库上，可以搞主从架构，主库写入，从库读取，读写分离。读流量太多的时候，还可以加更多的从库。

**ES**

es 是分布式的，可以随便扩容，一些比较简单的查询、统计类的操作，可以考虑用 es 来承载，还有一些全文搜索类的操作，也可以考虑用 es 来承载。

# 读写分离

### 如何实现MySQL的读写分离？

基于主从架构，一个主库，挂多个从库，只写主库，主库会自动把数据同步到从库上去

### MySQL主从复制原理是什么？

主库将变更写入binlog日志，从库连接到主库，有一个线程将主库的binlog日志拷贝到自己本地，写入一个relay中继日志种，接着从库就会启用一个SQL线程从中继日志读取binlog，执行binlog种的内容

### MySQL主从同延时问题怎么解决

通过MySQL命令show status 查看Seconds\_Behind\_Master，可以卡拿到从库复制主库数据落后多少

半同步复制：解决主库图软宕机，数据没有同步到从库上，数据丢失问题

所谓半同步复制，指主库吸入binlog日志后，会强制立即将数据同步到从库，从库将日志写入relay log之后，会返回一个ack给主库，主库接收到至少一个从库的ack才认为写操作完成

并行复制：解决时延问题

指的是从库开启多个线程并行读取relay log种不同库的日志，并行回放不同库的日志

分库：一个主库拆分为多个主库，每个主库的并发就减少了几倍

打开并行复制，多个库并行复制

# 分库分表

**分库**：

一个库，并发最多2000，分库是将一个库种的数据拆分到多个库中，减轻并发压力

**分表**：

单表数据太多严重影响SQL执行性能，需要将一个表的数据放在多个表中，单表控制在200w以内

**分库分表中间件：**

Cobar 阿里，proxy层方案

Sharding-jdbc，client层方案

MyCat 基于Cobar改造，proxy层方案

### 有一个未分库分表的系统，未来要分库分表，如何才能从未分库分表动态切换到分库分表上？

**停机迁移方案**：用户量不大，数据不多使用

**双写迁移方案**：比较靠谱，不用停机，就是同时写老库和新库，系统部署后，写一个导数工具，跑起来读老库写新库，写的时候判断数据最后修改时间，若这条数据在新库种不存在，或者这条数据比新库中的数据新才可以写，不允许老数据覆盖新数据，当数据完全一致后，重新部署代码

### 分布式ID生成方式

1.UUID 性能差

2.数据库自增ID 需要一个单独的MYSQL来提供服务，无法抗住高并发

3.基于数据库的号段模式，从数据库批量的获取自增ID，每次从数据库取出一个号段范围，例如 (1,1000] 代表1000个ID



1. 基于redis实现，利用redis的incr命令
2. 基于雪花算法snowflake模式

# 分布式锁

### Zookeeper的应用场景

1. 分布式协调

可以基于其临时节点的特性，不同机器在 ZooKeeper 的一个指定节点下创建临时子节点，不同机器之间可以根据这个临时节点来判断客户端机器是否存活。

1. 分布式锁
2. 配置信息管理（注册中心）

zookeeper 可以用作很多系统的配置信息的管理，比如 kafka、storm 等等很多分布式系统都会选用 zookeeper 来做一些元数据、配置信息的管理，包括 dubbo 注册中心

### Redis分布式锁和zk分布式锁的对比

redis 分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能。

zk 分布式锁，获取不到锁，注册个监听器，不需要不断主动尝试获取锁，性能开销较小。

如果是 redis 获取锁的那个客户端 挂了，只能等待超时时间之后才能释放锁；而 zk 的话，因为创建的是临时 znode，只要客户端挂了，znode 就没了，此时就自动释放锁

# Dubbo

### Duobbo的工作原理？注册中心挂了可以继续通信吗？

**工作原理**：

**工作流程**：

第一步：provider向注册中心注册服务

第二步：consumer从注册中心订阅服务，注册中心会通知consumer注册好的服务

第三步：consumer调用provider

第四步：provider和consumer都异步通知监控中心

注册中心挂了可以继续通信，初始化的时候，消费者会将提供者的地址信息拉去到本地缓存

### Dubbo支持哪些通信协议？支持哪些序列化协议？Hessian的数据结构？PB是什么？

Dubbo支持的通信协议

Dubbo协议：单一长连接，NIO异步通信，hessian作为序列化协议。并发量高。

通俗来说，就是消费者与提供者简历请求后持续发送请求，无需再建立连接。

Rmi协议：多个短连接，适合消费者和提供者数量差不多的情况，适用于文件传输，走java 二进制序列化，不常用。

Hessian协议：多个短链接，适用与提供者比消费者还多的情况，适用于文件传输，hessian 序列化协议，不常用

http协议：走json序列化，不常用。

Webservice：走SOAP文本序列化协议。

Hessian的数据结构

原始二进制数据、boolean、64-bit date（64位毫秒值的日期）、64-bit double、 32-bit int、

64bit-long、 null、utf-8编码的string

3种递归类型： list for lists and arrays、map for maps and dictionaries、object for objects

特殊类型：ref 表示对共享对象的引用

PB是什么

Protocol buffer谷歌出品的一种轻量并且搞笑的结构化数据存储格式，性能比json和xml快上了20~100倍，因为它适用proto编译器，自动序列化和反序列化，并且压缩效果好，数据量体积小，因此传输带宽和速度上有优势。

### Dubbo负载均衡策略和集群容错策略有哪些？动态代理策略？

**负载均衡策略4种**

Random loadbalance:

默认得负载均衡策略，随机调用实现负载均衡，可以对provider设置不通权重，权重越大分配得流量越高

Roundrobin loadbalance:

将流量均匀得打到每个机器，如果机器性能不一样，容易造成性能差的机器负载过高，需要调整权重

Leastactive loadbalance:

自动感知，机器性能越差收到的请求就会越少

Consistanthash loadbalance:

一类请求都到一个节点，基于一致性 Hash算法

**集群容错策略5种**

Failover(失败自动切换)：

失败自动切换，自动重试其他机器，默认就是这个，常见于读操作。

Failfast(快速失败)：

一次调用失败就立即失败，常见于非幂等性的写操作，比如新增一条记录

Failsafe(失败安全)

在实现上，当出现调用失败时，会忽略此错误，并记录一条日志，同时返回一个空结果，在上游看来调用是成功的。应用场景，可以用于写入审计日志等操作。

Failback(失败自动恢复)：

失败了后台自动记录请求，然后定时重发，比较适合于写消息队列这种。

Forking(并行调用)：

并行调用多个provider，只要一个成功就立即返回。常用于实时性要求比较高的读操作，但是会浪费更多的服务资源。

Broadcast(广播调用)：

逐个调用所有的 provider。任何一个 provider 出错则报错（从2.1.0 版本开始支持）。通常用于通知所有提供者更新缓存或日志等本地资源信息。

**Dubbo动态代理策略**

默认使用 javassist 动态字节码生成，创建代理类。但是可以通过 spi 扩展机制配置自己的动态代理策略

# 如何自己实现一个rpc框架？

分布式事务如何解决？