大数据面试题（二）

Zookeeper

1. 简介ZAB

ZAB协议是为Zookeeper专门设计的一种支持崩溃恢复的消息广播协议。ZAB协议只允许有一个主进程接受客户事务请求并处理，即leader。当leader收到请求后，将请求事务转化为事务proposal，由于leader会为每一个follower创建一个队列，将该事务放入响应队列，保证事务的顺序性。之后会在队列中顺序向其他节点广播该提案，follower收到后会将其以事务的形式写入到本地日志中，并向leader发送反馈ack确认。leader会等待其他followe的回复，当收到一半以上的follower响应时，leader会向其他节点发送commit消息，同时提交该提案。

ZAB有两种模式，分别为故障恢复模式以及消息广播。当系统启动或者leader服务器出现故障等现象时，进入故障恢复模式。将会开启新的一轮选举，选举产生的leader会与过半的follower进行同步，使数据一致。

当同步结束后，退出恢复模式，进入消息广播模式。当一台遵从ZAB协议的服务器启动后，如果检测到有leader在广播消息，会自动进入恢复模式，当其完成与leader的同步之后，进入消息广播模式；如果集群中的非leader节点收到客户端请求，非leader节点会先将请求发送到leader服务器。

故障恢复发时候，ZAB协议两个需要保证的地方，第一就是ZAB协议需要保证已经被leader提交的事务最终被所有的机器提交；第二就是需要保证丢弃那些只在leader上提交的事务。为了保证以上两点，选举时如果选择ZXID最大的节点可以解决上述问题。

leader重新选举的条件是当leader宕机或发生故障，集群中少于一半的节点与当前leader保持连接。

2. Zookeeper如何保证数据的一致性？

1）顺序一致性
来自客户端的更新将严格按照客户端发送的顺序处理；
2）原子性
更新或者成功或者失败，不存在部分成功或者部分失败的场景；
3）单一视图
无论客户端连接到哪个服务器，看到的都是一样的视图；
4）可靠性
一旦一个更新生效，它将一直保留，直到再次更新；
5）实时性
在特定的一段时间内，任何系统的改变都能被客户端看到，或者被监听到。

3. 叙述ZAB集群数据同步的过程

第一阶段（准leader 生成初始化事务集合）

所有follower 向leader 发送自己最后接收的事务的epoch；
leader 选取最大的epoch，加1得到e1，将e1 发送给follower；
follower 收到leader 发送的epoch 值之后，与自己的epoch 值作比较，若小于，则将自己的epoch更新为e1，并向准leader 发送反馈ACK信息（epoch 信息、历史事务集合）；
leader 接收到ACK 消息之后，会在所有历史事务集合中选择其中一个历史事务集合作为初始化事务集合，该事务集合满足ZXID最大；

第二阶段（数据同步）

leader 将epoch 与 初始化事务集合发送给集群中过半的follower；每个follower 会分配到一个队列，leader 会将那些没有被各个follower 同步的事务以proposal 形式发送给各个follower，并在后面追加commit 消息，表示事务已被提交；

follower 接收后，会执行初始化事务集合中的事务（执行过跳过，未执行执行），反馈给leader 表明自己已经处理

leader 收到后过半反馈后，发送commit 消息

follower 接收到commit 消息后，提交事

4.使用Redis开发分布式锁和使用Zookeeper开发分布式锁的区别是什么？

- Redis setnx 开发分布式锁，无法保证线程顺序，有可能让某些线程 一直处于等待状态（自己动手实现）
- Zookeeper 开发分布式锁，主要借助Zookeeper的临时顺序节点（可以使用Curator-framework实现）

Kafka

1. kafka介绍

(1).工作原理

- 首先`Producer`产生`Record`发送给指定的==Kafka Topic（Topic实质是有多个分区构成，每一个分区都会相应的复制分区）==，在真正存放到Kafka集群时会进行计算==key.hashCode%topicPartitionNums==等于要存放的分区序号。
- `Leader`分区中的数据会自动同步到`Follower`分区中,`ZooKeeper`会实时监控服务健康信息，一旦发生故障，会立即进行故障转移操作（将一个Follower复制分区自动升级为Leader主分区）
- Kafka一个分区实际上是一个有序的Record的`Queue`（符合队列的数据结构，先进先出），分区中新增的数据，会添加到队列的末尾，在处理时，会从队列的头部开始消费数据。`Queue`在标识读写操作位置时，会使用一个`offset`（读的offset <= 写的offset）
- 最后`Consumer`会订阅一个Kafka Topic，一旦Topic中有新的数据产生，Conumser立即拉取最新的记录，进行相应的业务处理。

2.Kafka如何保证高性能读写 √

- 写入性能：分区、磁盘顺序写入、Memory Map File(内核空间内存)
- 写出性能：分区、Zero Copy

总的来说Kafka快的原因：

1、partition顺序读写，充分利用磁盘特性，这是基础；

2、Producer生产的数据持久化到broker，采用mmap文件映射，实现顺序的快速写入；

3、Customer从broker读取数据，采用sendfile，将磁盘文件读到OS内核缓冲区后，直接转到socket buffer进行网络发送。

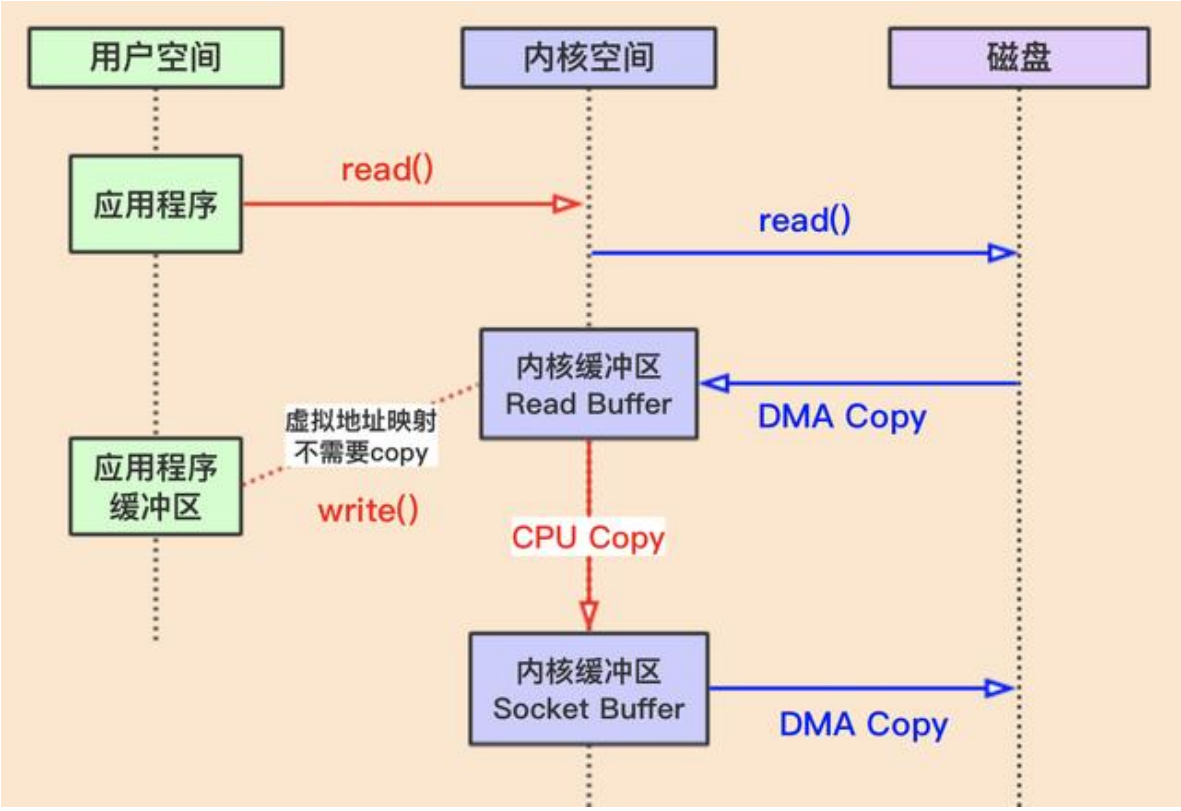
1、顺序读写
磁盘顺序读或写的速度400M/s，能够发挥磁盘最大的速度。
随机读写，磁盘速度慢的时候十几到几百K/s。这就看出了差距。
kafka将来自Producer的数据，顺序追加在partition，partition就是一个文件，以此实现顺序写入。
Consumer从broker读取数据时，因为自带了偏移量，接着上次读取的位置继续读，以此实现顺序读。
顺序读写，是kafka利用磁盘特性的一个重要体现。

2、零拷贝 sendfile(in,out)
数据直接在内核完成输入和输出，不需要拷贝到用户空间再写出去。
kafka数据写入磁盘前，数据先写到进程的内存空间。

3、mmap文件映射
虚拟映射只支持文件；
在进程 的非堆内存开辟一块内存空间，和OS内核空间的一块内存进行映射，
kafka数据写入、是写入这块内存空间，但实际这块内存和OS内核内存有映射，也就是相当于写在内核内存空间了，且这块内核空间、内核直接能够访问到，直接落入磁盘。

使用mmap+write方式替换原来的传统IO方式，就是利用了虚拟内存的特性

整体流程的核心区别就是，把数据读取到内核缓冲区后，应用程序进行写入操作时，直接是把**内核的Read Buffer的数据 复制到 Socket Buffer 以便进行写入，这次内核之间的复制也是需要CPU参与的**



sendfile

这种方式可以替换mmap+write方式，如：

```
mmap();
write();
```

替换为

```
sendfile();
```

这样就减少了一次上下文切换，因为少了一个应用程序发起write操作，直接发起sendfile操作。

3.kafka的零拷贝

zero copy技术就是减少不必要的内核缓冲区跟用户缓冲区间的拷贝，从而减少CPU的开销　和内核态切换开销，达到性能的提升

传统IO的流程

1、第一次：将磁盘文件，读取到操作系统内核缓冲区；
2、第二次：将内核缓冲区的数据，copy到application应用程序的buffer；
3、第三步：将application应用程序buffer中的数据，copy到socket网络发送缓冲区(属于操作系统内核的缓冲区)；
4、第四次：将socket buffer的数据，copy到网卡，由网卡进行网络传输。

传统方式，读取磁盘文件并进行网络发送，经过的四次数据copy是非常繁琐的。实际IO读写，需要进行IO中断，需要CPU响应中断(带来上下文切换)，尽管后来引入DMA来接管CPU的中断请求，但四次copy是存在“不必要的拷贝”的。

重新思考传统IO方式，会注意到实际上并不需要第二个和第三个数据副本。应用程序除了缓存数据并将其传输回套接字缓冲区之外什么都不做。相反，数据可以直接从读缓冲区传输到套接字缓冲区

4.Kafka数据同步机制 √

- LEO（LogEndOffset）：表示每个partition的log最后一条Message的位置。
- HW（High Watermark）：表示partition各个replicas数据间同步且一致的offset位置，即表示allreplicas已经commit位置，每个Broker缓存中维护此信息，并不断更新。是指consumer能够看到的此partition位置。 ISR 集合中最小的 LEO 即为分区的 HW
- ISR(In-Sync replicas): 处于同步中的副本集合，Kafka分区Leader维系了一份ISR的列表，该列表会把一些慢节点移除

```
replica.lag.time.max.ms= 最大允许Replicas 同步时间间隔 大于该参数，该节点会被移除。
```

kafka 的数据同步过程

Producer 在发布消息到某个 Partition 时，先通过ZooKeeper 找到该 Partition 的 Leader 【get /brokers/topics//partitions/2/state】，然后无论该Topic 的 Replication Factor 为多少（也即该 Partition 有多 少个 Replica(副本)），Producer 只将该消息发送到该 Partition 的Leader。Leader 会将该消息写入其本地 Log。每个 Follower都从 Leader pull 数据。这种方式上，Follower 存储的数据顺序与 Leader 保持一致。Follower 在收到该消息并写入其Log 后，向 Leader 发送 ACK。一旦 Leader 收到了 ISR 中的所有 Replica 的 ACK，该消息就被认为已经 commit 了，Leader 将增加 HW(HighWatermark)并且向 Producer 发送ACK。

这里会分两种情况：

第一种是 leader 处理完 producer 请求之后，follower 发送一个 fetch 请求过来、

第二种是follower 阻塞在 leader 指定时间之内，leader 副本收到producer 的请求。

数据丢失的问题

前提：min.insync.replicas=1 的时候。->设定 ISR 中的最小副本数是多少，默认值为 1, 当且仅当 acks 参数设置为-1（表示需要所有副本确认）时，此参数才生效. 表达的含义是，至少需要多少个副本同步才能表示消息是提交的所以，当 min.insync.replicas=1 的时候一旦消息被写入 leader 端 log 即被认为是“已提交”，而延迟一轮 FETCH RPC 更新 HW 值的设计使得 follower HW值是异步延迟更新的，倘若在这个过程中 leader 发生变更，那么成为新 leader 的 follower 的 HW 值就有可能是过期的，使得 clients 端认为是成功提交的消息被删除。

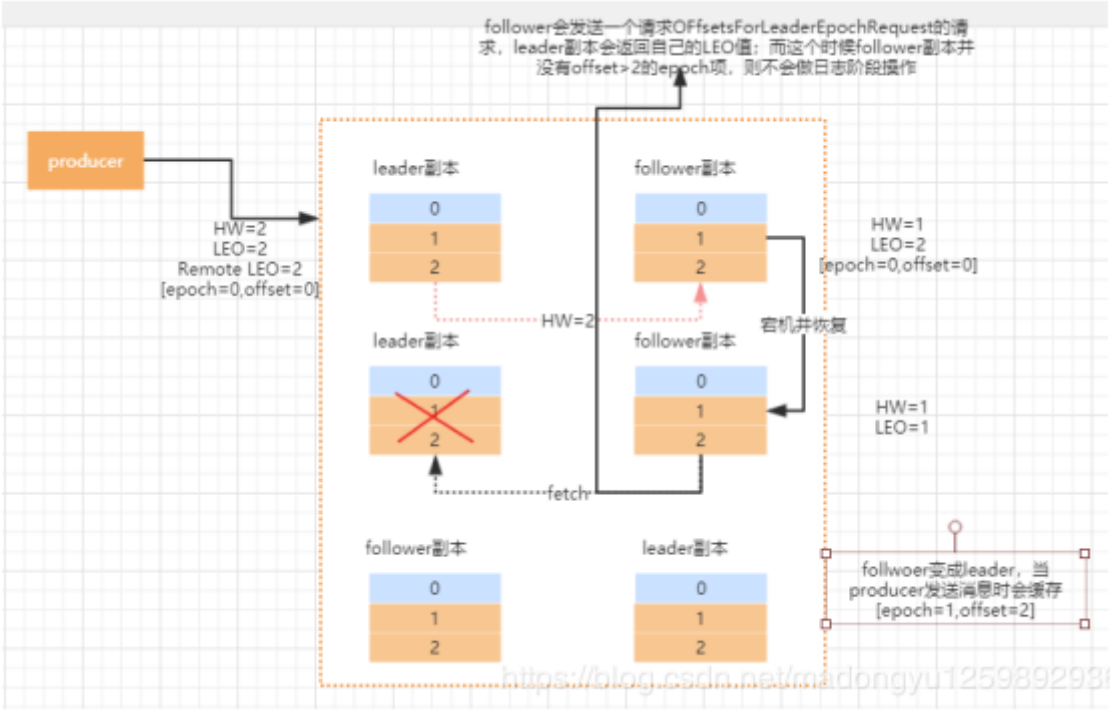
数据丢失的解决方案

在 kafka0.11.0.0 版本以后，提供了一个新的解决方案，使用 leader epoch 来解决这个问题，leader epoch 实际上是一对值(epoch,offset), epoch 表示 leader 的版本号，从 0开始，当 leader 变更过 1 次时 epoch 就会+1，而 offset 则对应于该 epoch 版本的 leader 写入第一条消息的位移。比如说

(0,0) ; (1,50); 表示第一个 leader 从 offset=0 开始写消息，一共写了 50 条，第二个 leader 版本号是 1，从 50 条处开始写消息。这个信息保存在对应分区的本地磁盘文件中，文 件 名 为 ： /tml/kafka-log/topic/leader-epochcheckpoint

leader broker 中会保存这样的一个缓存，并定期地写入到一个 checkpoint 文件中。

当 leader 写 log 时它会尝试更新整个缓存——如果这个leader 首次写消息，则会在缓存中增加一个条目；否则就 不做更新。而每次副本重新成为 leader 时会查询这部分缓存，获取出对应 leader 版本的 offset



如何处理所有的 Replica 不工作的情况

在 ISR 中至少有一个 follower 时，Kafka 可以确保已经commit 的数据不丢失，但如果某个 Partition 的所有 Replica 都宕机了，就无法保证数据不丢失了

等待 ISR 中的任一个 Replica“活”过来，并且选它作为Leader

这就需要在可用性和一致性当中作出一个简单的折衷。如果一定要等待 ISR 中的 Replica“活”过来，那不可用的时 间就可能会相对较长。而且如果 ISR 中的所有 Replica 都无法“活”过来了，或者数据都丢失了，这个 Partition 将永远不可用。

```
2.选择第一个“活”过来的 Replica（不一定是 ISR 中的）作为 Leader
```

选择第一个“活”过来的 Replica 作为 Leader，而这个Replica 不是 ISR 中的 Replica，那即使它并不保证已经包 含了所有已 commit 的消息，它也会成为 Leader 而作为consumer 的数据源（前文有说明，所有读写都由 Leader完成）。使用的是第一种策略

https://blog.csdn.net/madongyu1259892936/article/details/99596335

https://www.cnblogs.com/yoke/p/11486196.html

https://blog.csdn.net/qq_36142114/article/details/80314947

5.Kafka Streaming:

- Kafka流计算是基于应用端计算
- Kafka流计算不存在阶段划分
- Kafka流计算的并行度和分区保持一致，所需要资源由分区决定。
- 不具备Shuffle功能，，不涉及任务间通信。
- Kafka 流计算的Shuffle借助Topic分区的特性实现的。

6.幂等性

幂等：多次操作的结果和一次操作的结果相同，就称为幂等性操作**。读操作一定是幂等性操作，写操作一定不是幂等性操作。

Kafka的producer和broker之间默认有应答（ack）机制，当kafka的producer发送数据给broker，如果在规定的时间没有收到应答，生产者会自动重发数据，这样的操作可能造成重复数据（at least onnce语义）的产生。

```
enable.idempotence = true //开启幂等性

properties.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,true);
```

7.Memory Map FIle（内核空间内存）

简称mmap，简单描述其作用就是：将磁盘文件映射到内存, 用户通过修改内存就能修改磁盘文件。
它的工作原理是直接利用操作系统的Page来实现文件到物理内存的直接映射。完成映射之后你对物理内存的操作会被同步到硬盘上（操作系统在适当的时候）。

通过mmap，进程像读写硬盘一样读写内存（当然是虚拟机内存），也不必关心内存的大小有虚拟内存为我们兜底。
使用这种方式可以获取很大的I/O提升，省去了用户空间到内核空间复制的开销。

mmap也有一个很明显的缺陷——不可靠，写到mmap中的数据并没有被真正的写到硬盘，操作系统会在程序主动调用flush的时候才把数据真正的写到硬盘。Kafka提供了一个参数——producer.type来控制是不是主动flush；如果Kafka写入到mmap之后就立即flush然后再返回Producer叫同步(sync)；写入mmap之后立即返回Producer不调用flush叫异步(async)。

8.Kafka,Flume中组件包括哪些？

```
Kafka组件：

        Topic ：消息根据Topic进行归类

        Producer：发送消息者

        Consumer：消息接受者

        broker：每个kafka实例(server)

        Zookeeper：依赖集群保存meta信息。

Flume组件：

        Agent:

        Source:
```

```
        Channel:

        Sink:
```

9.Kafka事务

数据库事务回顾

数据库事务：多个操作是一个原子操作，不可分割，同时成功或者同时失败。

事务问题： 脏读、不重复读、幻影读

事务隔离级别（解决事务问题）：读未提交、读提交、可重复读、序列化读

Kafka事务

kafka事务指的是在一个事务内多个Record发送或者处理是一个原子操作，不可分割，同时成功，或者同时失败。

kafka事务的隔离级别：读未提交（默认：`read_uncommitted`）、读提交（`read_commmitted`）

仅生产者事务

Kafka生产者在一个事务内生产的Record是一个不可分割的整体，要么同时写入Kafka集群，或者某个出错，回滚撤销所有的写操作

> 开启生产者事务：
>
> - `ENABLE_IDEMPOTENCE_CONFIG=true`
> - `每一个生产者，需要唯一的事务编号：TransactionID`
> - `事务的超时时间（可选）：TRANSACTION_TIMEOUT_CONFIG`

消费生产并存事务（重点）

> 特点：消费kafka的A主题进行业务操作，将操作的结果写入到B主题中。如果开启`consume-transfer-produce`事务，读A和写B在一个事务环境中，不可分割，要么同时成功，要么同时失败

10.kafka和rabbitmq的区别

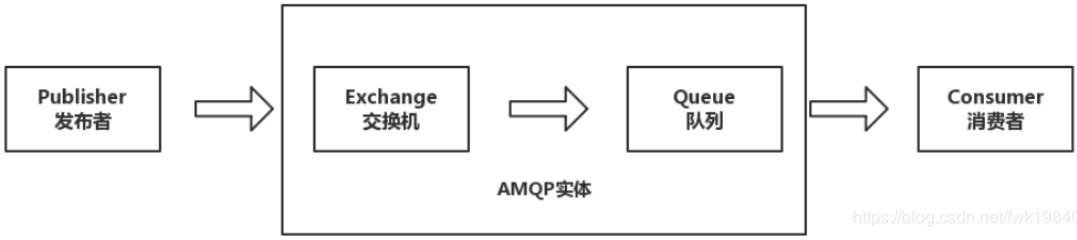
一、语言不同

RabbitMQ是由内在高并发的erlanng语言开发，用在实时的对可靠性要求比较高的消息传递上。

kafka是采用Scala语言开发，它主要用于处理活跃的流式数据,大数据量的数据处理上

二、结构不同

RabbitMQ采用AMQP（Advanced Message Queuing Protocol，高级消息队列协议）是一个进程间传递异步消息的网络协议



RabbitMQ的broker由Exchange,Binding,queue组成

kafka采用mq结构：broker 有part 分区的概念

三、Broker与Consume交互方式不同

RabbitMQ 采用push的方式

kafka采用pull的方式

四、在集群负载均衡方面，

rabbitMQ的负载均衡需要单独的loadbalancer进行支持。

kafka采用zookeeper对集群中的broker、consumer进行管理

五、使用场景

rabbitMQ支持对消息的可靠的传递，支持事务，不支持批量的操作；基于存储的可靠性的要求存储可以采用内存或者硬盘。

金融场景中经常使用

kafka具有高的吞吐量，内部采用消息的批量处理，zero-copy机制，数据的存储和获取是本地磁盘顺序批量操作，具有O(1)的复杂度（与分区上的存储大小无关），消息处理的效率很高。（大数据）