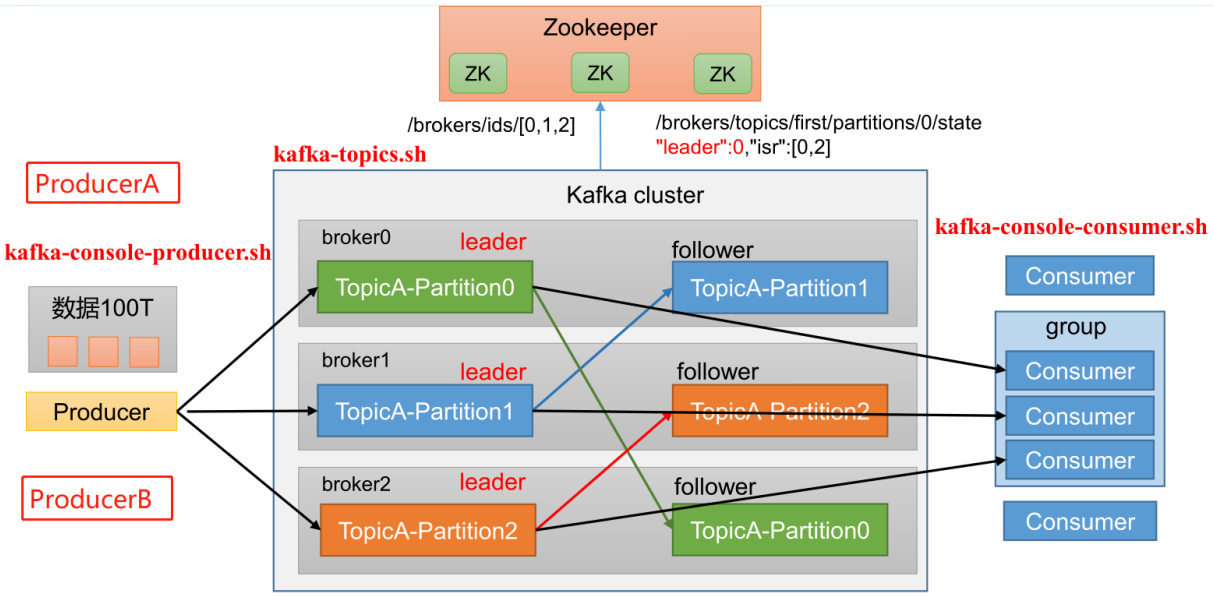
# 其他八股文详解

## 一、 Kafka 相关

参考：https://www.cnblogs.com/huangdh/p/16886327.html

### 1.1 Kafka 基础架构




1. **生产者 Producer**：消息生产者，就是向 Kafka broker 发消息的客户端。

2. **消费者 Consumer**：消息消费者，向 Kafka broker 获取消息的客户端。

3. **消费者组 Consumer Group（CG）**：消费者组，由多个 consumer 组成。**消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费**；消费者组之间互不影响，所有的消费者都属于某个消费者组，**即消费者组是逻辑上的一个订阅者。**

4. **Broker**：一台 Kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker 可以容纳多个 topic。

5. **Topic**：可以理解为一个队列，生产者和消费者面向的都是一个 topic。

6. **Partition**：**为了实现扩展性，一个非常大的 topic 可以分布到多个 broker（即服务器）上，一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列。**

7. **Replica**：副本。**一个 topic 的每个分区都有若干个副本，一个 Leader 和若干个 Follower。**

8. Leader：**每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是 Leader。**

9. Follower：每个分区多个副本中的“从”，实时从 Leader 中同步数据，保持和 Leader 数据的同步。Leader 发生故障时，某个 Follower 会成为新的 Leader。

10. in-syncreplicaset（ISR）：由 Leader 维护，意为和 Leader 保持同步的 Follower+Leader 集合(leader：0，isr:0,1,2)，如果某个 Follower 节点在指定时间内没有应答 Leader，则将这个节点从 ISR 队列中踢出。

**1.为方便扩展，并提高吞吐量，一个 topic 分为多个 partition。**

**2.配合分区的设计，提出消费者组的概念，组内每个消费者并行消费。**

**3.为提高可用性，为每个 partition 增加若干副本，类似 NameNodeHA。**

**4.ZK 中记录谁是 leader，Kafka2.8.0 以后也可以配置不采用 ZK。**

消息传递语义：

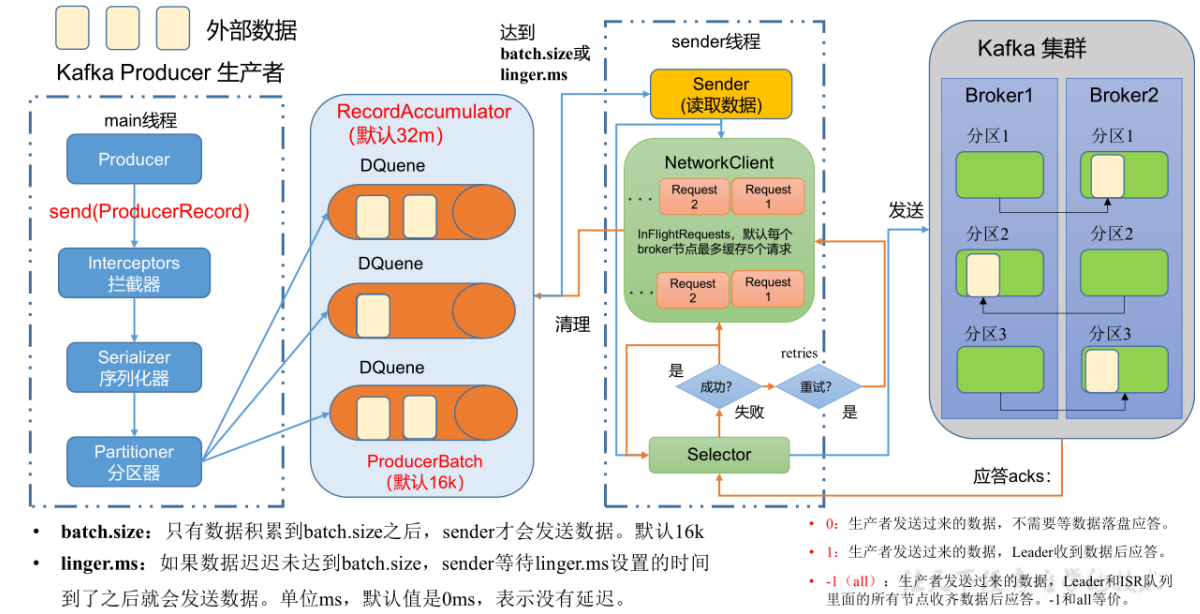
- 至少一次（AtLeastOnce）=ACK 级别设置为-1+分区副本大于等于 2+ISR 里应答的最小副本数量大于等于 2。可以保证数据不丢失，但是不能保证数据不重复；

- 最多一次（AtMostOnce）=ACK 级别设置为 0。可以保证数据不重复，但是不能保证数据不丢失。

- 精确一次（ExactlyOnce）：**精确一次（ExactlyOnce）=幂等性+至少一次（ack=-1+分区副本数>=2+ISR 最小副本数量>=2）**。对于一些非常重要的信息，比如和钱相关的数据，**要求数据既不能重复也不丢失。**

## 1.2 Kafka 生产者

### 1.2.1 生产者消息发送流程




**在消息发送的过程中，涉及到了两个线程——main 线程和 Sender 线程。在 main 线程中创建了一个双端队列 RecordAccumulator（默认 32m）。main 线程将消息发送给 RecordAccumulator，Sender 线程不断从 RecordAccumulator 中拉取消息发送到 Kafka Broker。**

1. 发送消息的第一步就是执行拦截器

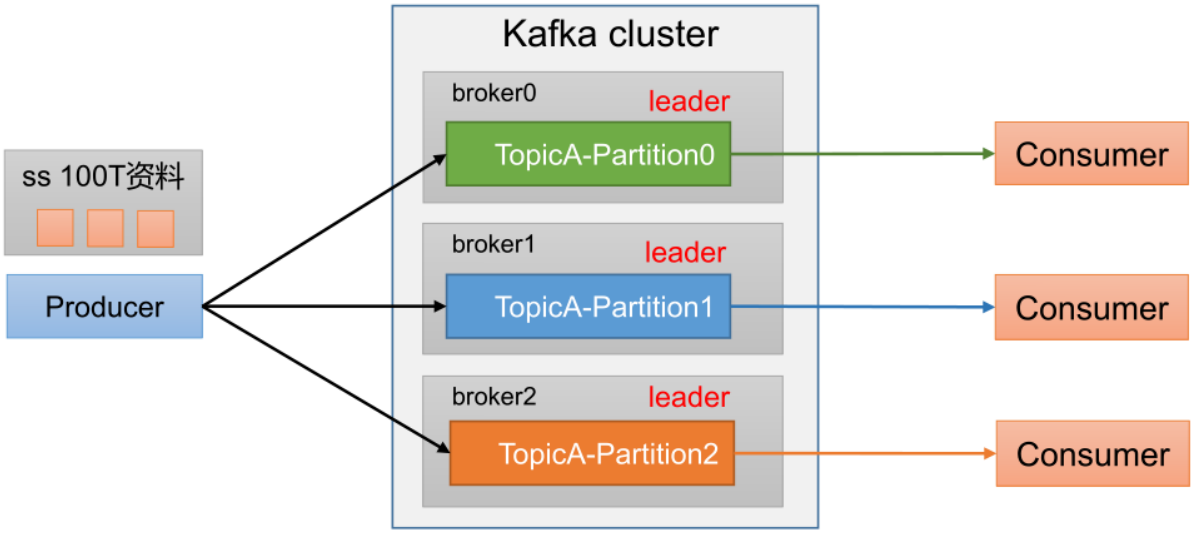
2.

### 1.2.2 生产者分区

#### 1.2.2.1 分区的好处

1. **便于合理使用存储资源，每个 Partition 在一个 Broker 上存储，可以把海量的数据按照分区切割成一块一块数据存储在多台 Broker 上。合理控制分区的任务，可以实现负载均衡的效果。**

2. **提高并行度**，生产者可以**以分区为单位发送数据**；消费者可以**以分区为单位进行消费数据**。




#### 1.2.2.2 生产者消息发送分区策略

（1）指明 partition 的情况下，直接将指明的值作为 partition 值；例如 partition=0，所有数据写入分区 0。

（2）没有指明 partition 值但有 key 的情况下，**将 key 的 hash 值与 topic 的 partition 数进行取余得到 partition 值；例如：key1 的 hash 值=5，key2 的 hash 值=6，topic 的 partition 数=2，那么 key1 对应的 value1 写入 1 号分区，key2 对应的 value2 写入 0 号分区。**

（3）**既没有 partition 值又没有 key 值的情况下，Kafka 采用 StickyPartition（黏性分区器），会随机选择一个分区，并尽可能一直使用该分区，待该分区的 batch 已满或者已完成，Kafka 再随机选择一个分区进行使用（和上一次的分区不同）。**例如：第一次随机选择 0 号分区，等 0 号分区当前批次满了（默认 16k）或者 linger.ms 设置的时间到，Kafka 再随机选择一个分区进行使用（如果还是 0 会继续随机）。

### 1.2.3 生产者角度--如何提高生产者吞吐量

1. **批量写入 batch.size：修改批次大小，默认 16K。**

2. **linger.ms：修改等待时间，默认 0。**

3. **修改 Kafka 生产者内存缓冲区大小 RecordAccumulator：修改缓冲区大小，默认 32M：buffer.memory**

4. **compression.type：修改压缩，默认 none，可配置值：gzip、snappy、lz4 和 zstd**

### 1.2.4 生产者角度--如何提高数据可靠性

ACK 应答原理：

1. acks：0，生产者发送过来的数据，不需要等数据落盘应答。Leader 节点收到消息之后，在数据落盘之前就挂掉了，则会导致数据丢失。**数据可靠性分析：数据丢失。**

2. acks：1，生产者发送过来的数据，Leader 收到数据后应答。应答完成之后，还没开始同步副本，Leader 节点就挂掉了，新的 Leader 不会再收到之前发送的消息，因为生产者已经认为消息发送成功了。 **数据可靠性分析：数据丢失。**

3. **acks：-1（all）：生产者发送过来的数据，Leader 和 ISR 队列里面的所有节点收齐数据后应答。**

思考：

1. acks：-1（all），Leader 收到数据，所有 Follower 都开始同步数据，但有一个 Follower，因为某种故障，迟迟不能与 Leader 进行同步，那这个问题怎么解决呢？

答：Leader 维护了一个动态的 in-syncreplicaset（ISR），意为和 Leader 保持同步的 Follower+Leader 集合(leader：0，isr:0,1,2)。如果 Follower 长时间未向 Leader 发送通信请求或同步数据，则该 Follower 将被踢出 ISR。**该时间阈值由 replica.lag.time.max.ms 参数设定，默认 30s**。例如 2 超时，(leader:0, isr:0,1)。这样就不用等长期联系不上或者已经故障的节点。

2. acks：-1（all）是否表示数据完全可靠呢？

答：不一定。**如果分区副本设置为 1 个，或者 ISR 里应答的最小副本数量（min.insync.replicas 默认为 1）设置为 1，和 ack=1 的效果是一样的，仍然有丢数的风险（leader：0，isr:0）**

生产者侧数据可靠性总结：

- **acks=0，生产者发送过来数据就不管了，可靠性差，效率高。**
- **acks=1，生产者发送过来数据 Leader 应答，可靠性中等，效率中等。**
- **acks=-1，生产者发送过来数据 Leader 和 ISR 队列里面所有 Follwer 应答，可**

**靠性高，效率低。**

- **数据完全可靠条件=ACK 级别设置为-1+分区副本大于等于 2+ISR 里应答的最小副本数量大于等于 2。**

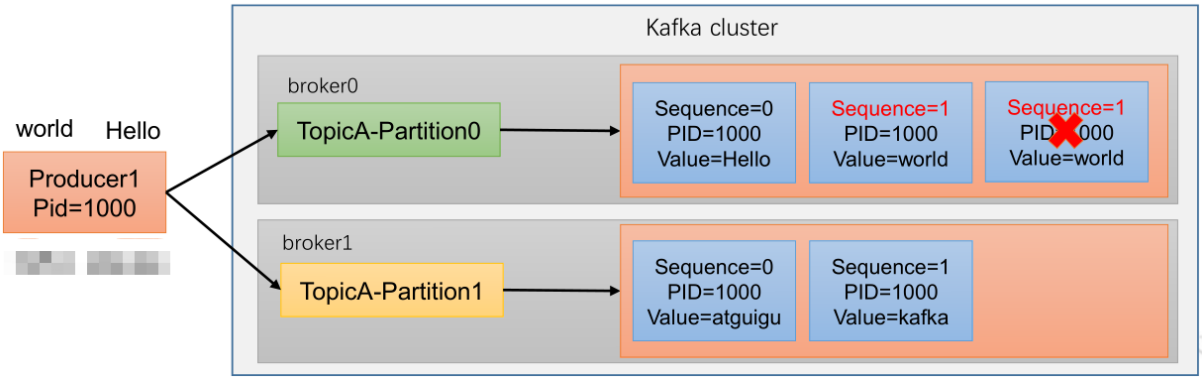
在生产环境中，acks=0 很少使用；acks=1，一般用于传输普通日志，允许丢个别数据；acks=-1，一般用于传输和钱相关的数据，对可靠性要求比较高的场景。

### 1.2.5 数据幂等性保障

**幂等性：就是指 Producer 不论向 Broker 发送多少次重复数据，Broker 端都只会持久化一条，保证了不重复。**

**重复数据的判断标准**：具有<**PID,Partition,SeqNumber**>相同主键的消息提交时，Broker 只会持久化一条。其中 **PID 是 Kafka 每次重启都会分配一个新的**；**Partition 表示分区号；SequenceNumber 是单调自增的。**

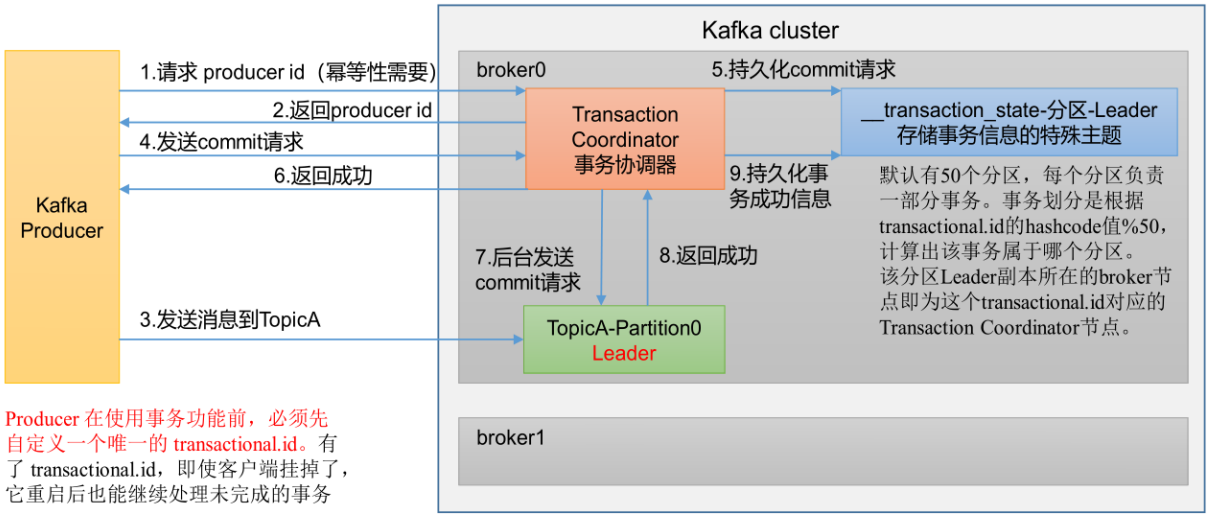
**所以幂等性只能保证的是在单分区单会话内不重复。**




使用幂等性的方法：

开启参数 enable.idempotence 默认为 true

## 1.2.6 生产者事务




## 1.2.7 数据有序/数据乱序

1. **单分区内，有序。多分区内，分区与分区间无序。**

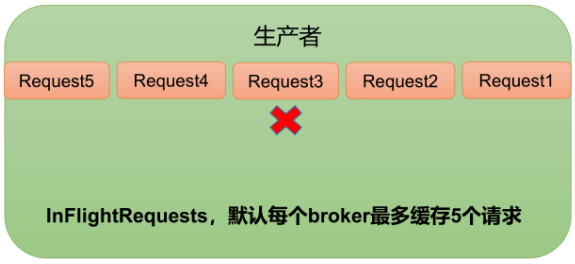
2. **kafka 在 1.x 版本之前**，保证数据单分区内有序，条件如下：

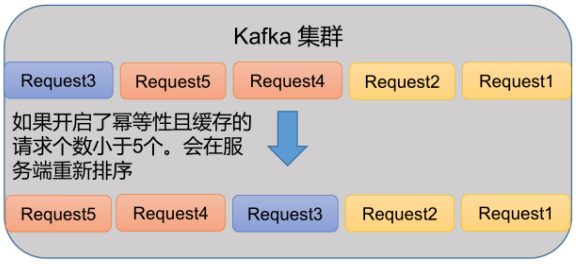
   a. **max.in.flight.requests.per.connection=1（不需要考虑是否开启幂等性）。**

3. **kafka 在 1.x 及以后版本**保证数据单分区有序，条件如下：

   a. **未开启幂等性**：max.in.flight.requests.per.connection 需要设置为 1。

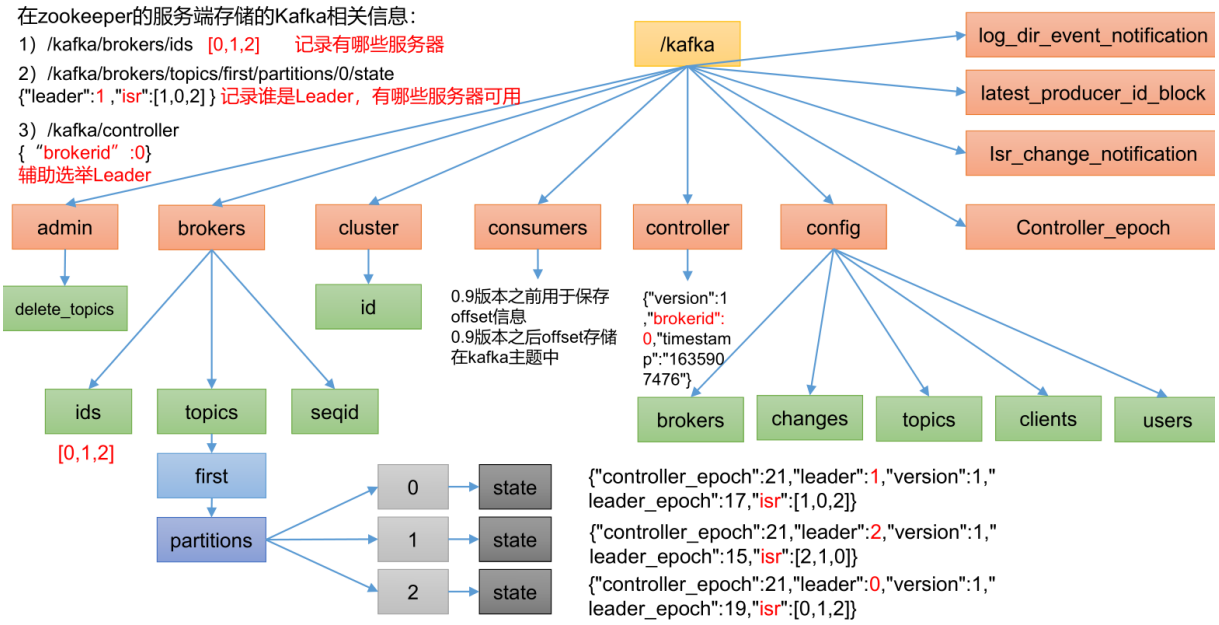
   b. **开启幂等性**：**max.in.flight.requests.per.connection 需要设置小于等于 5**。因为在 kafka1.x 以后，启用幂等后，kafka 服务端会缓存 producer 发来的最近 5 个 request 的元数据，故无论如何，都可以保证最近 5 个 request 的数据都是有序的。



## 1.3 Kafka Broker

### 1.3.1 ZooKeeper 存储的 kafka 信息

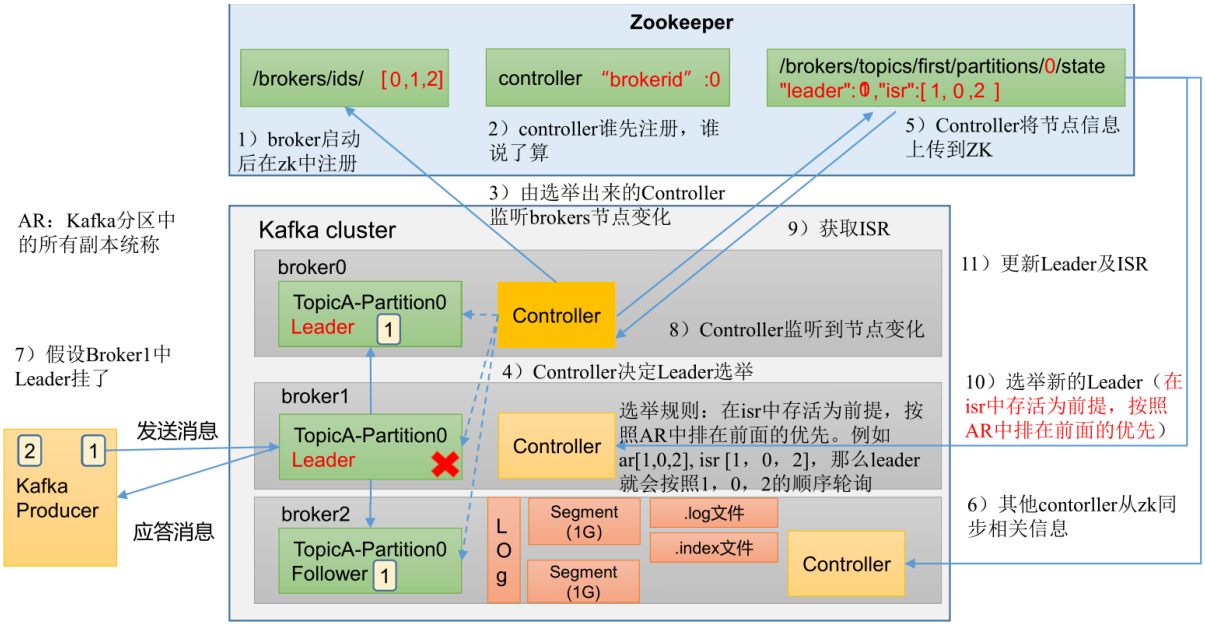



1. **/kafka/brokers/ids：[0,1,2]，记录有哪些服务器。**

2. **/kafka/brokers/topics/first/partitions/0/state：{"leader":1,"isr":[1,0,2] } 记录谁是 Leader，有哪些服务器可用。**

3. **/kafka/controller：{ “brokerid” :0} 辅助选举 Leader**

## 1.3.2 Kafka Broker 工作流程




1. broker 启动后在 zk 中注册

2. controller 谁先注册，谁说了算

3. 由选举出来的 controller 监听 brokers 节点变化

4. controller 决定 Leader 选举

5. controller 将节点信息上传到 zk 中

6. 其他 controller 从 zk 同步相关信息

7. 假设 broker1 中 Leader 挂了

8. controller 监听到节点发生变化

9. 获取 ISR

10. 选举新的 Leader（在 isr 中存活为前提，按照 AR 中排在前面的优先，例如：ar[1,0,2]，那么 leader 就会按照 1,0,2 的顺序轮询）

11. 更新 Leader 及 ISR

# TODO

## 1.3.3 Kafka 副本

### 1.3.3.1 Kafka 集群中分区分布原理分析

https://www.cnblogs.com/wxd0108/p/6519973.html

副本分配逻辑规则如下：

- 在 Kafka 集群中，每个 Broker 都有均等分配 Partition 的 Leader 机会。

- 上述图 Broker Partition 中，箭头指向为副本，以 Partition-0 为例:broker1 中 parition-0 为 Leader，Broker2 中 Partition-0 为副本。

- 上述图种每个 Broker(按照 BrokerId 有序)依次分配主 Partition,下一个 Broker 为副本，如此循环迭代分配，多副本都遵循此规则。

副本分配算法如下：

- 将所有 N Broker 和待分配的 i 个 Partition 排序.

- 将第 i 个 Partition 分配到第(i mod n)个 Broker 上.

- 将第 i 个 Partition 的第 j 个副本分配到第((i + j) mod n)个 Broker 上.

## 1.3.4 文件存储

https://cloud.tencent.com/developer/article/1057763

Kafka 关于文件存储的部分名词解释如下：

- Partition：topic 物理上的分组，一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列。
- Segment：partition 物理上由多个 segment 组成，

从以下四个方面分析：

1. topic 中 partition 存储分布
2. partiton 中文件存储方式
3. partiton 中 segment 文件存储结构
4. 在 partition 中如何通过 offset 查找 message

### 1.3.4.1 topic 中 partition 存储分布

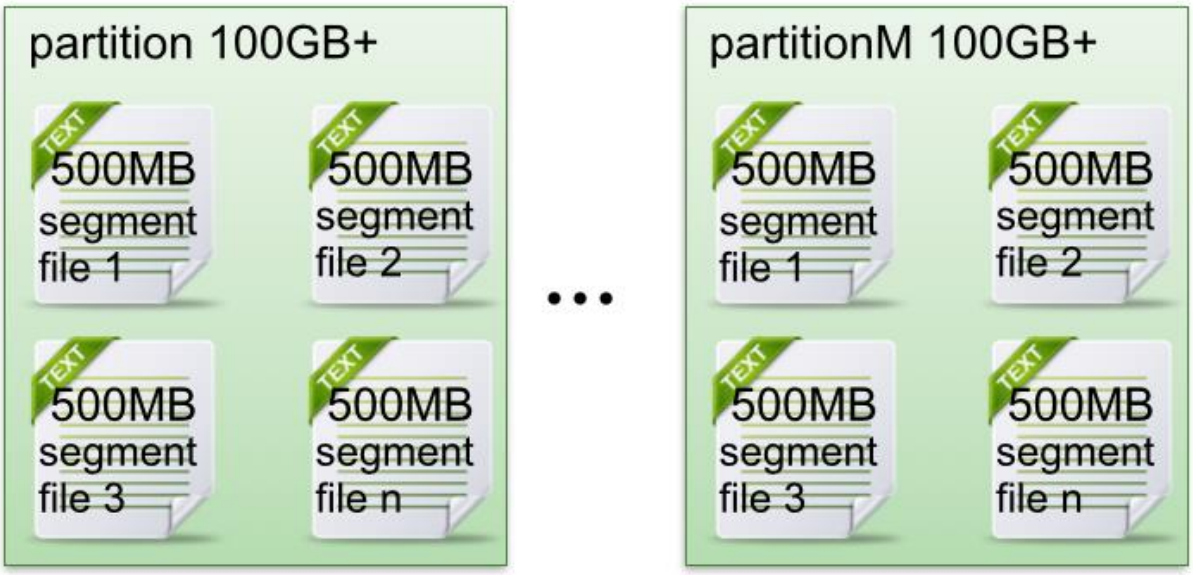
一个 broker，例如创建 2 个 topic 名称分别为 report_push、launch_info, partitions 数量都为

partitions=4，数据文件存储根目录下文件夹如下所示：

```
|--report_push-0
    |--report_push-1
    |--report_push-2
    |--report_push-3
    |--launch_info-0
    |--launch_info-1
    |--launch_info-2
    |--launch_info-3
```

在 Kafka 文件存储中，同一个 topic 下有多个不同 partition，每个 partition 为一个目录，partiton 命名规则为 topic 名称+有序序号，第一个 partiton 序号从 0 开始，序号最大值为 partitions 数量减 1。多副本同理

### 1.3.4.2 partition 中文件存储方式




- 每个 partion(目录)相当于一个巨型文件被平均分配到多个大小相等 segment(段)数据文件中。但每个段 segment file 消息数量不一定相等，这种特性方便 old segment file 快速被删除。
- 每个 partiton 只需要支持顺序读写就行了，segment 文件生命周期由服务端配置参数决定。

这样做的好处是快速删除无用文件，有效提高磁盘利用率。

### 1.3.4.3 partition 中 segment 文件存储结构

本节深入分析 partion 中 segment file 组成和物理结构。

- **segment file 组成：**由 2 大部分组成，分别为 index file 和 data file，此 2 个文件一一对应，成对出现，后缀".index"和“.log”分别表示为 segment 索引文件、数据文件.
- **segment 文件命名规则：**partion 全局的第一个 segment 从 0 开始，后续每个 segment 文件名为上一个全局 partion 的最大 offset(偏移 message 数)。数值最大为 64 位 long 大小，19 位数字字符长度，没有数字用 0 填充。

下面文件列表是笔者在 Kafka broker 上做的一个实验，创建一个 topicXXX 包含 1 partition，设置每个 segment 大小为 500MB,并启动 producer 向 Kafka broker 写入大量数据,如下图 2 所示 segment 文件列表形象说明了上述 2 个规则：

00000000000000000000.index
00000000000000000000.log
00000000000000368769.index
00000000000000368769.log
00000000000000737337.index
00000000000000737337.log
00000000000001105814.index
00000000000001105814.log
……

图 2

以上图中一对 segment file 文件为例，说明 segment 中 index<—->data file 对应关系物理结构如下

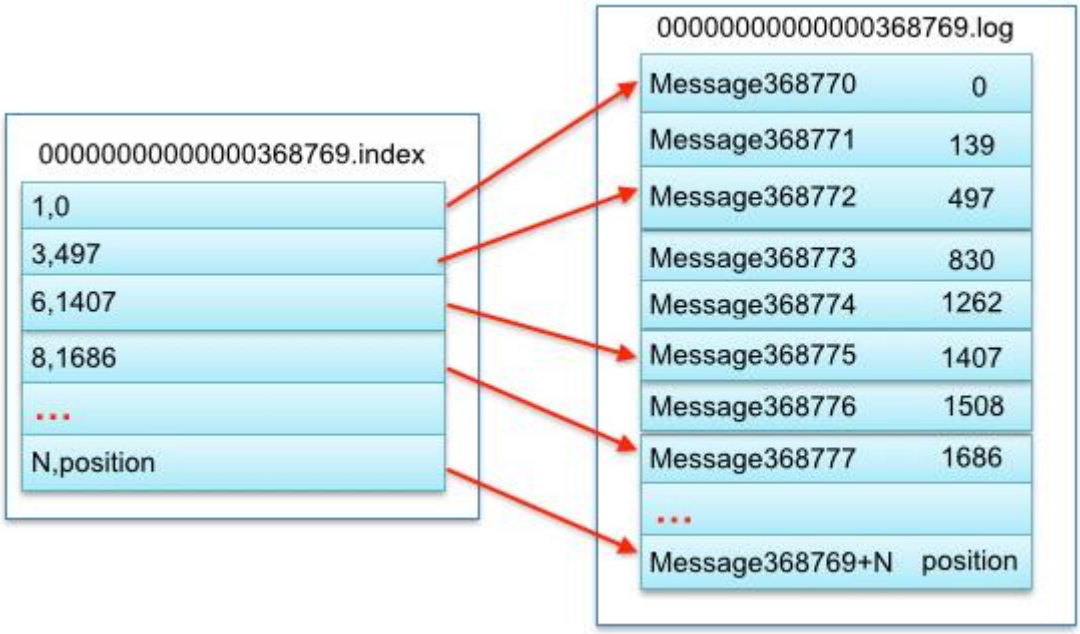
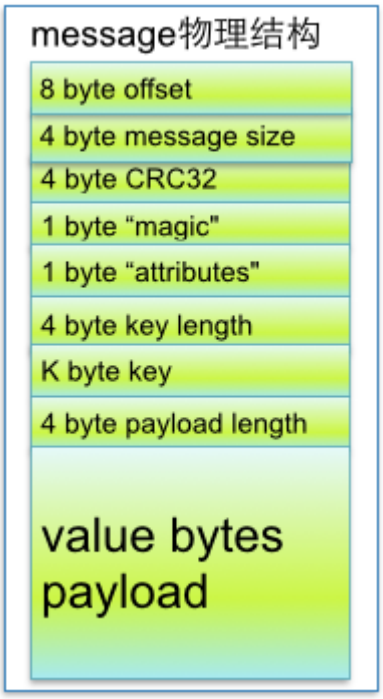



图 3

上图中索引文件存储大量元数据，数据文件存储大量消息，索引文件中元数据指向对应数据文件中 message 的物理偏移地址。

其中以索引文件中元数据 3,497 为例，依次在数据文件中表示第 3 个 message(在全局 partiton 表示第 368772 个 message)、以及该消息的物理偏移地址为 497。

从上述图了解到 segment data file 由许多 message 组成，下面详细说明 message 物理结构如下：




参数说明：

| 关键字 | 解释说明 |
| --- | --- |
| 8 byte **offset（重要，msg 的 offset）** | 在 parition(分区)内的每条消息都有一个有序的 id 号，这个 id 号被称为偏移(offset),它可以唯定每条消息在 parition(分区)内的位置。即 offset 表示 partiion 的第多少 message |
| 4 byte message size | message 大小 |
| 4 byte CRC32 | 用 crc32 校验 message |
| 1 byte “magic" | 表示本次发布 Kafka 服务程序协议版本号 |
| 1 byte “attributes" | 表示为独立版本、或标识压缩类型、或编码类型。 |
| 4 byte key length | 表示 key 的长度,当 key 为-1 时，K byte key 字段不填 |
| K byte key | 可选 |
| value bytes payload | 表示实际消息数据。 |

### 1.3.4.4 在 partition 中如何通过 offset 查找 message

例如读取 offset=368776 的 message，需要通过下面 2 个步骤查找。

1. 查找 segment file

上述图 2 为例，其中 00000000000000000000.index 表示最开始的文件，起始偏移量(offset)为 0.第二个文件 00000000000000368769.index 的消息量起始偏移量为 368770 = 368769 + 1.同样，第三个文件 00000000000000737337.index 的起始偏移量为 737338=737337 + 1，其他后续文件依次类推，以起始偏移量命名并排序这些文件，只要根据 offset **二分查找**文件列表，就可以快速定位到具体文件。

2. 当 offset=368776 时定位到 00000000000000368769.index|log 通过 segment file 查找 message

通过第一步定位到 segment file，当 offset=368776 时，依次定位到 00000000000000368769.index 的元数据物理位置和 00000000000000368769.log 的物理偏移地址，然后再通过 00000000000000368769.log 顺序查找直到 offset=368776 为止。从上述图 3 可知这样做的优点，segment index file 采取稀疏索引存储方式，它减少索引文件大小，通过 mmap 可以直接内存操作，稀疏索引为数据文件的每个对应 message 设置一个元数据指针,它比稠密索引节省了更多的存储空间，但查找起来需要消耗更多的时间。

### 1.3.4.5 Kafka 读写消息具体方式

写 message

- 消息从 java 堆转入 page cache(即物理内存)。
- 由异步线程刷盘,消息从 page cache 刷入磁盘。

读 message

- 消息直接从 page cache 转入 socket 发送出去。
- 当从 page cache 没有找到相应数据时，此时会产生磁盘 IO,从磁盘 Load 消息到 page cache,然后直接从 socket 发出去

### 1.3.4.6 总结

Kafka 高效文件存储设计特点

- Kafka 把 topic 中一个 parition 大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。
- 通过索引信息可以快速定位 message 和确定 response 的最大大小。
- 通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作。
- 通过索引文件稀疏存储，可以大幅降低 index 文件元数据占用空间大小。

## 1.3.5 Controller

一文理解 Kafka 的选举机制与 Rebalance 机制-腾讯云开发者社区-腾讯云 (tencent.com)

Controller 是从 Broker 中选举出来的，负责分区 Leader 和 Follower 的管理。当某个分区的 leader 副本发生故障时，由 Controller 负责为该分区选举新的 leader 副本。当检测到某个分区的 ISR(In-Sync Replica)集合发生变化时，由控制器负责通知所有 broker 更新其元数据信息。当使用 kafka-topics.sh 脚本为某个 topic 增加分区数量时，同样还是由控制器负责分区的重新分配。

Kafka 中 Contorller 的选举的工作依赖于 Zookeeper，成功竞选为控制器的 broker 会在

Zookeeper 中创建/controller 这个临时（EPHEMERAL）节点。

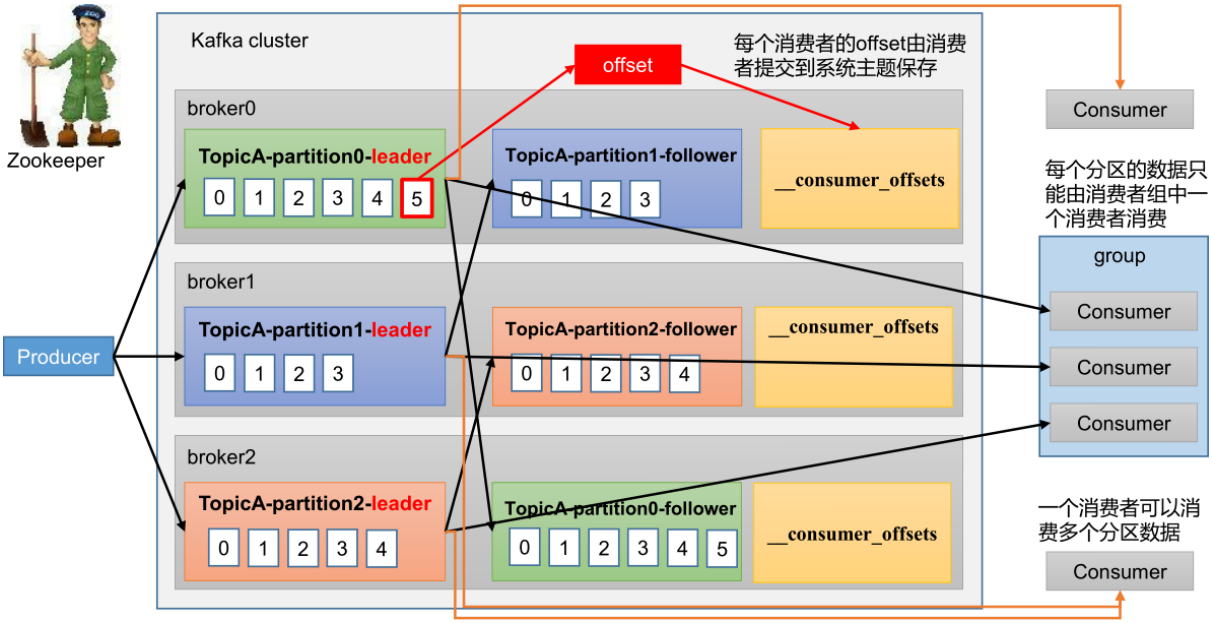
选举过程：

# 1.4 Kafka 消费者

## 1.4.1 Kafka 消费方式

Kafka 采用 pull（拉）模式。**consumer 采用从 broker 中主动拉取数据。Kafka 采用这种方式。pull 模式不足之处是，如果 Kafka 没有数据，消费者可能会陷入循环中，一直返回空数据。**

## 1.4.2 Kafka 消费者工作流程




## 1.4.3 分区的分配以及再平衡

https://cloud.tencent.com/developer/article/1852157

分区分配：

一个 consumer group 中有多个 consumer 组成，一个 topic 有多个 partition 组成。那么问题来了，到底由哪个 consumer 来消费哪个 partition 的数据？

原则如下：

同一个消费组中 给不同消费者分配能够消费的分区数;

同一个消费组中,一个分区只会被一个消费者消费。

### 1.4.3.1 分区策略的选择与配置

https://blog.csdn.net/weixin_40482816/article/details/127513743

由 GroupCoordinator（组协调器）选择使用哪个分配策略去分配分区。有以下两个原则：

1. 选择所有 Member 都支持的分配策略
2. 在 1 的基础上,优先选择每个 `partition.assignment.strategy` 配置靠前的策略。

如果新加入的消费者带上的分配策略跟现有 Group 中所有 Member(Group 有 Member 的情况下)都支持的协议都不交叉就会抛出异常：**INCONSISTENT_GROUP_PROTOCOL**

3. 每个消费者都可以配置一个 `partition.assignment.strategy` 属性并且可以配置多个自己支持的分配策略

**Kafka 有四种主流的分区分配策略：Range、RoundRobin、Sticky、CooperativeSticky。**

可以通过配置参数 **partition.assignment.strategy**，修改分区的分配策略。**默认策略是 Range + CooperativeSticky**。Kafka 可以同时使用多个分区分配策略。

### 1.4.3.2 分区再平衡

分区再平衡部分

就 Kafka 消费端而言，有一个难以避免的问题就是消费者的重平衡即 Rebalance。Rebalance 是让一个消费组的所有消费者就如何消费订阅 topic 的所有分区达成共识的过程，在 Rebalance 过程中，所有 Consumer 实例都会停止消费，等待 Rebalance 的完成。因为要停止消费等待重平衡完成，因此 Rebalance 会严重影响消费端的 TPS，是应当尽量避免的。

**触发再平衡的时机**：

Rebalance 的触发条件有 3 个。

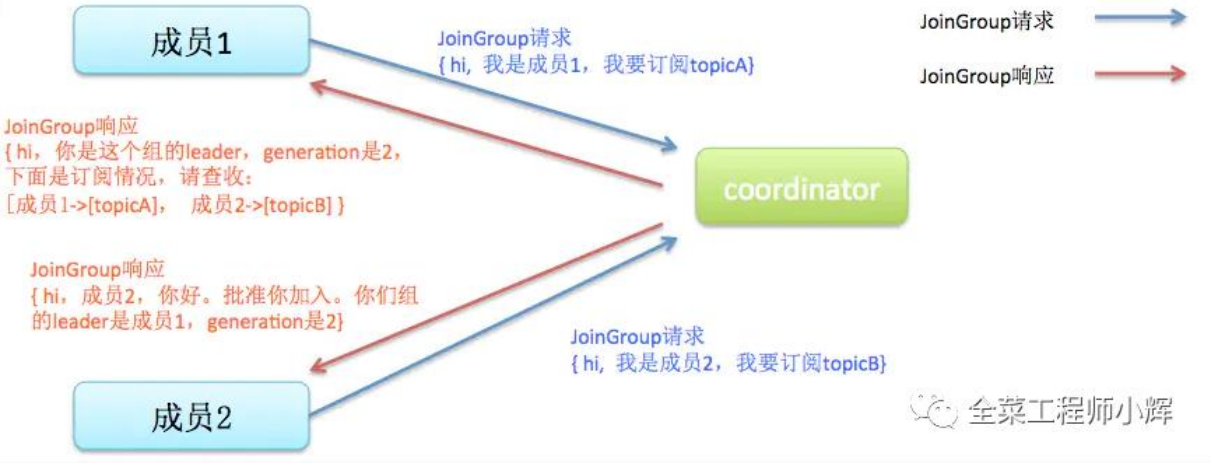
1. 消费组成员个数发生变化。例如有新的 Consumer 实例加入或离开该消费组。
2. 订阅的 Topic 个数发生变化。
3. 订阅 Topic 的分区数发生变化。

Rebalance 发生时，Group 下所有 Consumer 实例都会协调在一起共同参与，kafka 能够保证尽量达到最公平的分配。但是 Rebalance 过程对 consumer group 会造成比较严重的影响。**在 Rebalance 的过程中 consumer group 下的所有消费者实例都会停止工作**，等待 Rebalance 过程完成。

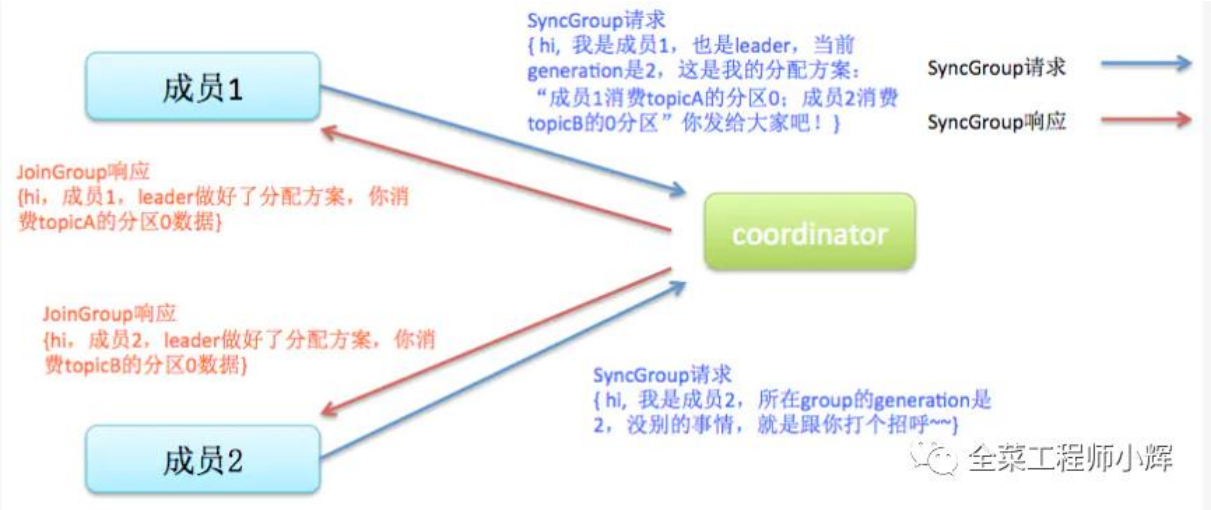
**再平衡过程**：

Rebalance 过程分为两步：Join 和 Sync。

1. Join。所有成员都向 Group Coordinator 发送 JoinGroup 请求，请求加入消费组。一旦所有成员都发送了 JoinGroup 请求，Coordinator 会从中选择一个 Consumer 担任 leader 的角色，并把组成员信息以及订阅信息发给 leader——注意 leader 和 coordinator 不是一个概念。leader 负责消费分配方案的制定。




2. Sync。这一步 leader 开始分配消费方案，即哪个 consumer 负责消费哪些 topic 的哪些 partition。一旦完成分配，leader 会将这个方案封装进 SyncGroup 请求中发给 coordinator，非 leader 也会发 SyncGroup 请求，只是内容为空。coordinator 接收到分配方案之后会把方案塞进 SyncGroup 的 response 中发给各个 consumer。这样组内的所有成员就都知道自己应该消费哪些分区了。




**避免不必要的 Rebalance**

前面说过 Rebalance 发生的时机有三个，后两个时机是可以人为避免的。发生 Rebalance 最常见的原因是消费组成员个数发生变化。

这其中消费者成员正常的添加和停掉导致 Rebalance，也是无法避免。但是在某些情况下，Consumer 实例会被 Coordinator 错误地认为已停止从而被踢出 Group。从而导致 rebalance。

这种情况可以通过 Consumer 端的参数 `session.timeout.ms` 和 `max.poll.interval.ms` 进行配置。

有关这种情况，可以查看博客《一文理解 Kafka 重复消费的原因和解决方案》

除了这个参数，Consumer 还提供了控制发送心跳请求频率的参数，就是 `heartbeat.interval.ms`。这个值设置得越小，Consumer 实例发送心跳请求的频率就越高。频繁

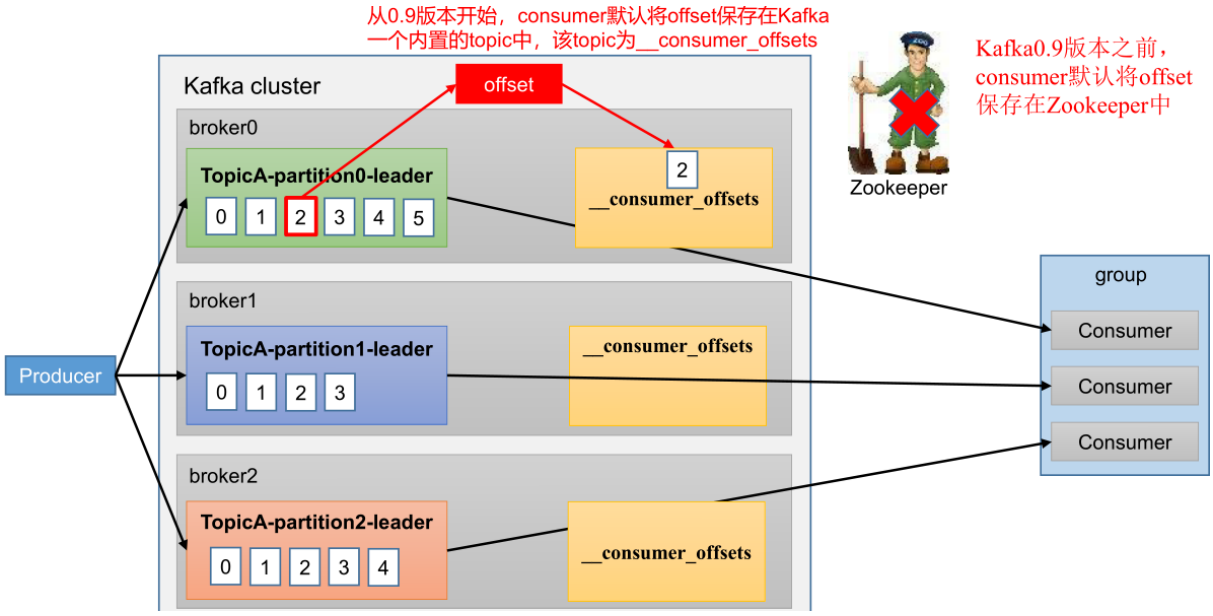
地发送心跳请求会额外消耗带宽资源，但好处是能够更快地知道是否开启 Rebalance，因为 Coordinator 通知各个 Consumer 实例是否开启 Rebalance 就是将 REBALANCE_NEEDED 标志封装进心跳请求的响应体中。

总之，要为业务处理逻辑留下充足的时间使 Consumer 不会因为处理这些消息的时间太长而引发 Rebalance，但也不能时间设置过长导致 Consumer 宕机但迟迟没有被踢出 Group。

## 1.4.4 Offset 位移

### 1.4.4.1 offset 默认维护位置

Kafka0.9 版本之前，consumer 默认将 offset 保存在 Zookeeper 中。从 0.9 版本开始，consumer 默认将 offset 保存在 Kafka 一个内置的 topic 中，该 topic 为**__consumer_offsets**。




**__consumer_offsets 主题**里面采用 **key 和 value 的方式存储数据。key 是 group.id+topic+分区号，value 就是当前 offset 的值。每隔一段时间，kafka 内部会对这个 topic 进行 compact，也就是每个 group.id+topic+分区号就保留最新数据。**

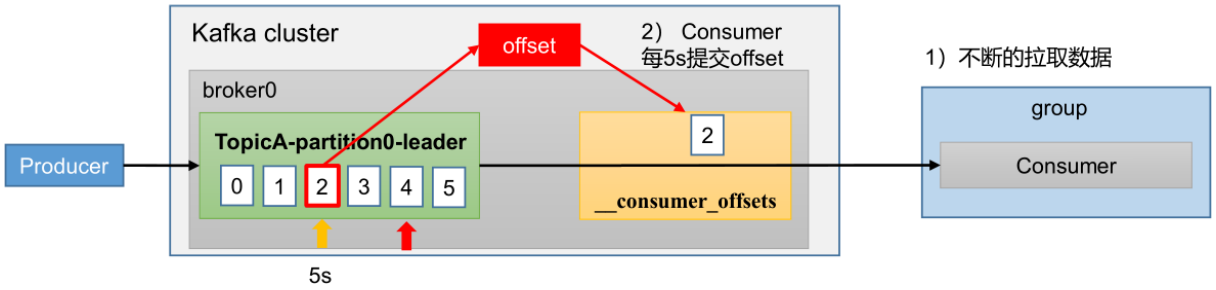
__consumer_offsets 为 Kafka 中的 topic，那就可以通过消费者进行消费。

### 1.4.4.2 自动提交 offset

自动提交 offset 的相关参数：

enable.auto.commit：是否开启自动提交 offset 功能，默认是 true，消费者会自动周期性地向服务器提交偏移量。。

auto.commit.interval.ms：自动提交 offset 的时间间隔，默认是 5s。如果设置了 enable.auto.commit 的值为 true， 则该值定义了消费者偏移量向 Kafka 提交的频率，默认 5s




### 1.4.4.3 手动提交 offset

**为什么要手动提交 offset？**

虽然自动提交 offset 十分简单便利，但由于其是**基于时间提交**的，开发人员难以把握 offset 提交的时机。因此 Kafka 还提供了手动提交 offset 的 API。

**手动提交 offset 的方法有两种**：分别是 **commitSync（同步提交）和 commitAsync（异步提交）**。两者的**相同点是**，都会**将本次提交的一批数据最高的偏移量提交**；**不同点是**，同步提交阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；而异步提交则没有失败重试机制，故有可能提交失败。

**commitSync（同步提交）**：必须等待 offset 提交完毕，再去消费下一批数据。**虽然同步提交 offset 更可靠一些，但是由于其会阻塞当前线程，直到提交成功。因此吞吐量会受到很大的影响。**因此更多的情况下，会选用异步提交 offset 的方式。

**commitAsync（异步提交）**：发送完提交 offset 请求后，就开始消费下一批数据了。

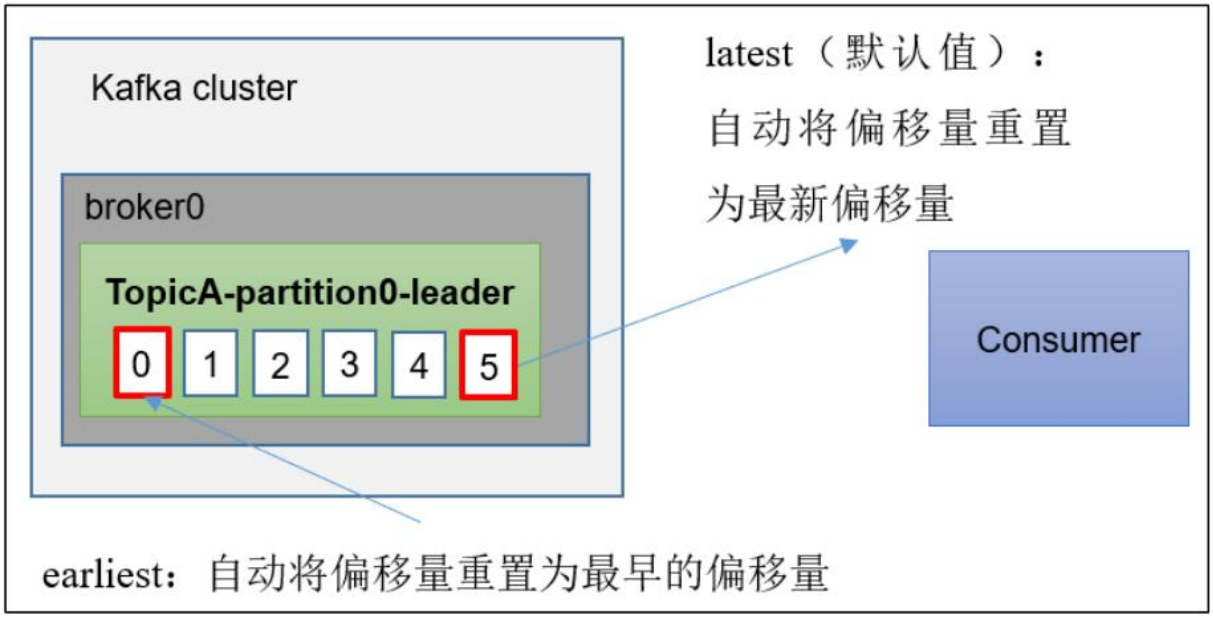
### 1.4.4.4 指定 Offset 消费

**auto.offset.reset = earliest | latest | none　默认是 latest。**

**（1）earliest：自动将偏移量重置为最早的偏移量，--from-beginning。**

**（2）latest（默认值）：自动将偏移量重置为最新偏移量。**

**（3）none：如果未找到消费者组的先前偏移量，则向消费者抛出异常。**




### 1.4.4.5 指定时间消费

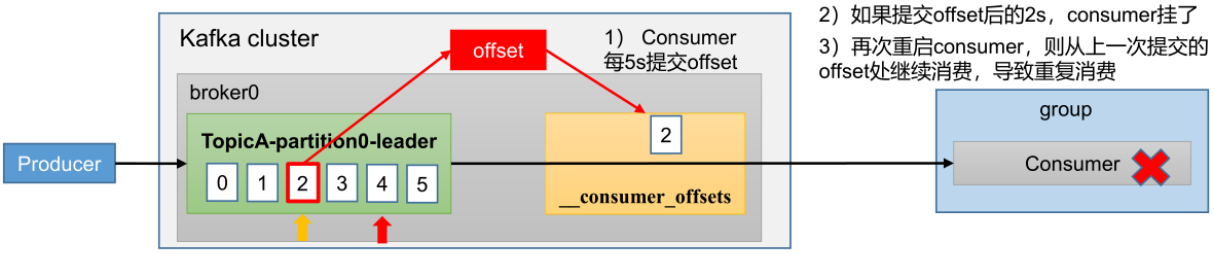
需求：在生产环境中，会遇到最近消费的几个小时数据异常，想重新按照时间消费。例如要求按照时间消费前一天的数据，怎么处理？

### 1.4.4.6 重复消费和漏消费

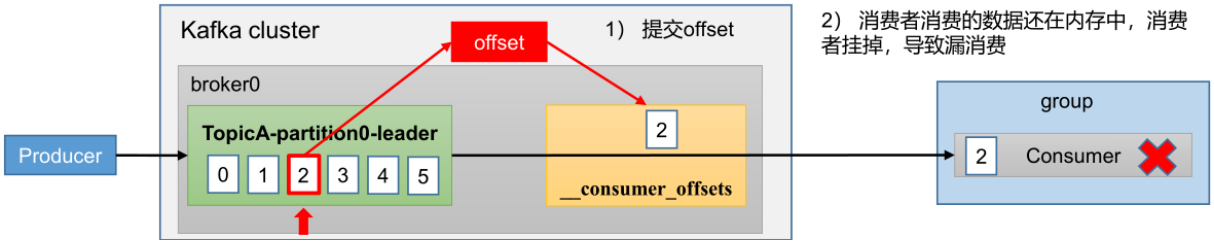
**重复消费：已经消费了数据，但是 offset 没提交。**

**漏消费：先提交 offset 后消费，有可能会造成数据的漏消费。**

（1）场景1：重复消费。自动提交offset引起。




（2）场景1：漏消费。设置offset为手动提交，当offset被提交时，数据还在内存中未落盘，此时刚好消费者线程被kill掉，那么offset已经提交，但是数据未处理，导致这部分内存中的数据丢失。



如何做到既不漏消费也不重复消费呢？**消费者事务**

### 1.4.5 消费者事务

**如果想完成 Consumer 端的精准一次性消费，那么需要 Kafka 消费端将消费过程和提交 offset 过程做原子绑定。**

### 1.4.6 消费者角度-如何提高吞吐量（数据积压）

1）如果是 Kafka 消费能力不足，则可以考虑**增加 Topic 的分区数，并且同时提升消费组的消费者数量，消费者数=分区数（两者缺一不可）**。

2）如果是下游的数据处理不及时：**提高每批次拉取的数量**。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。

## 1.5 Kafka 补充知识

### 1.5.1 高水位 HW 和 Leader Epoch 机制

https://www.lixueduan.com/posts/kafka/12-hw-leader-epoch/

### 1.5.2 Kafka 消费组/协调器

https://juejin.cn/post/7155297634339520526#heading-9

### 1.5.3 Kafka 选主

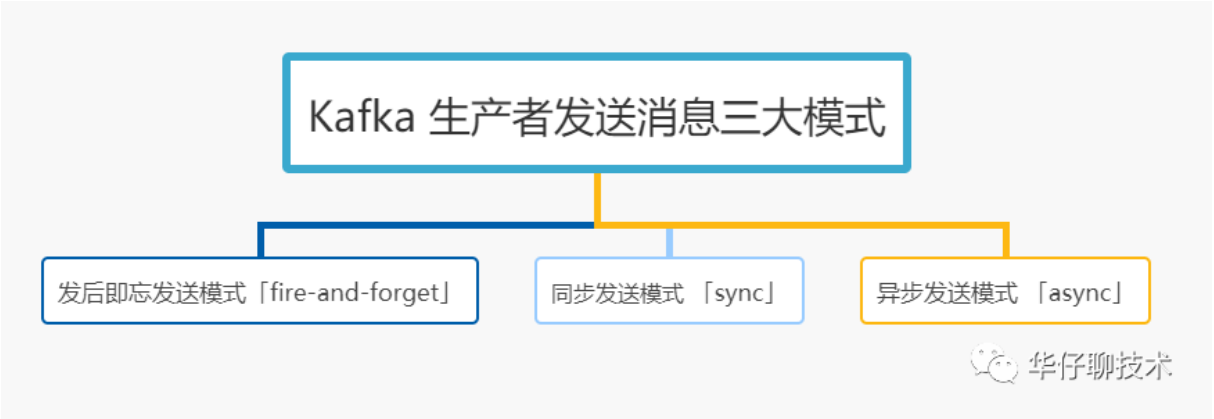
### 1.5.4 Kafka 分区同步

### 1.5.5 Kafka 面试题集合

https://ost.51cto.com/posts/11148?u_atoken=dcbbf899-6d73-40b4-8107-f022b01b513e&u_asession=01tHVwov9jR55LYuuMjyYLa4pOgN_5flAcJz7UnRQURo3ExSrJJVfRlpSNL36763H3X0KNBwm7Lovlpxjd_P_q4JsKWYrT3W_NKPr8w6oU7K_t_f1bbQEeflKg8sjRCqyxqKyj-Ku2w-s9yzCW7aTvhWBkFo3NEHBv0PZUm6pbxQU&u_asig=056pv1ppK7eLlCmy2-NPy3nAUx3uIvIykr1jgy5Kv2nUZWRUyev0OxeJTsX_VSly-uMjJYsTsc26R4NXpgTxHhwWeq5_ZQru0zhzRhPbiLbj74OjZNwBgB8PII3LHfEIDxU-7ED1Hjf-aIUQkDrosD3QqbTo2xXcfuZ15Usb22oD39JS7q8ZD7Xtz2Ly-

b0kmuyAKRFSVJkkdwVUnyHAIJzSzZswywNtarrQL837rrLfaTQ9VY540X-2KO9VTG5DESggefkcQmY4QIKRVIaahK_-3h9VXwMyh6PgyDIVSG1W_vHOvpUpM_df-NuMUr8xSK6rN4WnkP27a0CQVvwR6dMmXWQ2iHn8TOsy3RVLHdoZaUIcpioi0yOALHFTAsn2w6mWspDxyAEEo4kbsryBKb9Q&u_aref=30NWxdLM0FhXKjKrk8HMKebxLBc%3D

#### 1.5.5.1 生产者发送消息的模式




**01 发后即忘发送模式**

发后即忘模式「fire-and-forget」，它只管发送消息，并不需要关心消息是否发送成功。其本质上也是一种异步发送的方式，消息先存储在缓冲区中，达到设定条件后再批量进行发送。这是 kafka 吞吐量最高的方式，但同时也是消息最不可靠的方式，因为对于发送失败的消息并没有做任何处理，某些异常情况下会导致消息丢失。

**02 同步发送模式**

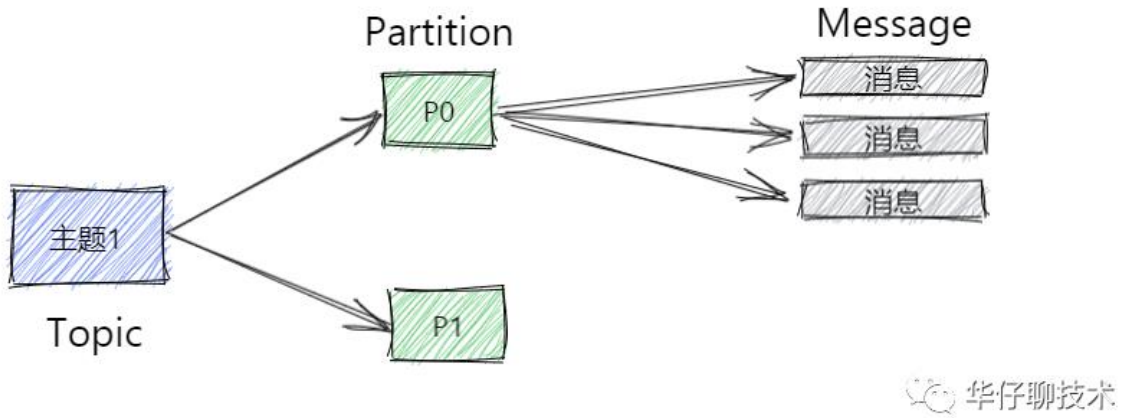
同步发送模式 「sync」，调用 send() 方法会返回一个 Future 对象，再通过调用 Future 对象的 get() 方法，等待结果返回，根据返回的结果可以判断消息是否发送成功， 由于是同步发送会阻塞，只有当消息通过 get() 返回数据时，才会继续下一条消息的发送。

**03 异步发送模式**

异步发送模式「async」，在调用 send() 方法的时候指定一个 callback 函数，当 Broker 接收到返回的时候，该 callback 函数会被触发执行，通过回调函数能够对异常情况进行处理，当调用了回调函数时，只有回调函数执行完毕生产者才会结束，否则一直会阻塞

#### 1.5.5.2 Kafka 为什么要设计分区

其实这个问题说来很简单, 假如不进行分区的话就如同 MySQL 单表存储一样，发消息就会被集中存储，这样会导致某台 Kafka 服务器存储 Topic 消息过多，如果在写消息压力很大的情况下，最终会导致这台 Kafka 服务器吞吐量出现瓶颈, 因此 Kafka 设计了分区的概念，同时也带来了「**负载均衡**」、「**横向扩展**」的能力，如下图所示：

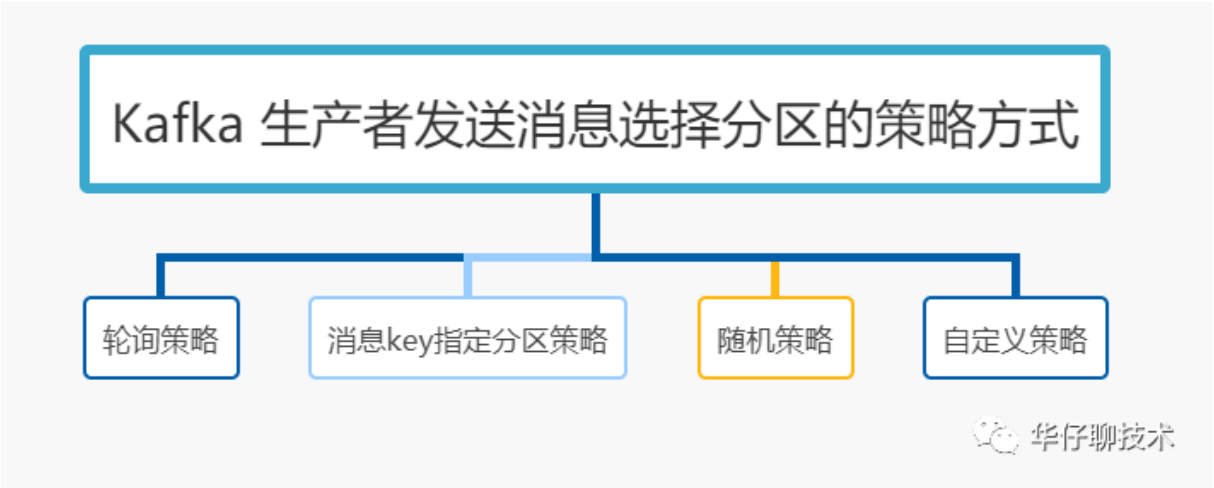



1) **负载均衡**：发送消息时可以根据分区的数量进行数据均匀分布，使其落在不同的分区上, 这样可以提高并发写性能;同时消费的时候多个订阅者可以从一个或者多个分区中同时消费数据，以支撑海量数据处理能力，提高读消息性能。

2)**横向扩展**：可以将一个 Topic 分成了多个 Partition，将不同的 Partition 尽可能的部署在不同的物理节点上，这样扩展起来非常方便，另外一个消费者可以消费多个分区中的数据，但是这样还是不能够充分的发挥横向扩展，这时候消费者组就出现了，我们用消费者组，来消费整个的 Topic，一个消费者消费 Topic 中的一个分区。

#### 1.5.5.3 生产者发送消息时如何选择分区

生产者发送消息的时候选择分区的策略方式主要有以下 4 种：




1) **轮询策略（Round-robin）**：顺序分配消息，即按照消息顺序依次发送到某 Topic 下不同的分区，它总是能保证消息最大限度地被平均分配到所有分区上，如果消息在创建的时候 key 为 null， 那么 Kafka 默认会采用这种策略。

2) **消息 key 指定分区策略**：Kafka 允许为每条消息定义 key，即消息在创建的时候 key 不为空，此时 Kafka 会根据消息的 key 进行 hash, 然后根据 hash 值对 Partition 进行取模映射到指定的分区上， 这样的好处就是相同 key 的消息会发送到同一个分区上， 这样 Kafka 虽然不能保证全局有序，但是可以保证每个分区的消息是有序的，这就是消息分区有序性，适应场景有下单支付的时候希望消息有序，可以通过订单 id 作为 key 发送消息达到分区有序性。

3) **随机策略**：随机发送到某个分区上，看似也是将消息均匀打散分配到各个分区，但是性能还是无法跟轮询策略比，「如果追求数据的均匀分布，最好还是使用轮询策略」。

4) **自定义策略**：可以通过实现 org.apache.kafka.clients.producer.Partitioner 接口，重写 partition 方法来达到自定义分区效果。

如果指定了 Key，则采用消息 key 指定分区策略；如果没有指定 Key，采用轮询策略：

- 指明 partition 的情况下，直接将指明的值作为 partition 值；
- 没有指明 partition 值，但有 key 的情况下，将 key 的 hash 值与 topic 的 partition 数目进行取余操作，得到 partition 值；
- 即没有 partition 值又没有 key 的情况下，第一次调用时随机生成一个整数（后面每次调用在这个整数上自增），将这个值与 topic 可用的 partition 总数取余得到 partition 值，也就是常说的 round-robin 算法。

#### 1.5.5.4 Kafka 如何合理设置分区数，越多越好吗？

**Kafka 如何合理设置分区数**

首先我们要了解在 Partition 级别上达到负载均衡是实现高吞吐量的关键，合适的 Partition 数量可以达到并行读写和负载均衡的目的，需要根据每个分区的生产者和消费者的目标吞吐量进行估计。

此时我们可以遵循一定的步骤来计算确定分区数：

1) 首先根据某个 Topic 当前接收的数据量等经验来确定分区的初始值。
2) 然后针对这个 Topic，进行测试 Producer 端吞吐量和 Consumer 端的吞吐量。
3) 测试的结果， 假设此时他们的值分别是 Tp「Producer 端吞吐量」、Tc「Consumer 端吞吐量」，总的目标吞吐量是 Tt， 单位是 MB/s， 那么结果 numPartition = Tt / max (Tp, Tc)。
4) 特殊说明：测试 Tp 通常是很容易的，因为它的逻辑非常简单，就是直接发送消息到 Kafka 就好了。而测试 Tc 通常与应用消费消息后进行其他什么处理有关，相对复杂一些

**分区设置越多越好吗?**

不是的！

**01** 使用内存方面分析

1)Broker 端：有很多组件都在内存中维护了分区级别的缓存，比如 Controller，FetcherManager 等，因此分区数越多，这类缓存的成本就越大。

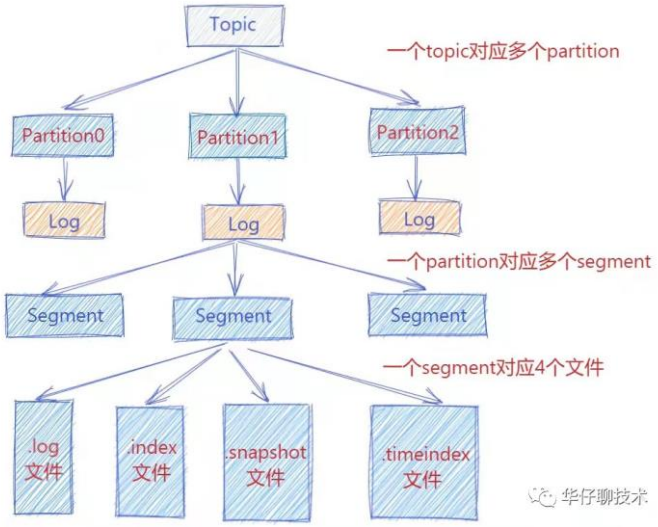
2)Producer 端：比如参数 batch.size，默认是 16KB。它会为每个分区缓存消息，在数据积累到一定大小或者足够的时间时，累积的消息将会从缓存中移除并发往

Broker 节点。这个功能是为了提高性能而设计，但是随着分区数增多，这部分缓存所需的内存占用也会更多。

3)Consumer 端：消费者数跟分区数是直接挂钩的，在消费消息时的内存占用、以及为达到更高的吞吐性能需要开启的 Consumer 数也会随着分区数增加而增加

## 02 消耗文件句柄方面分析

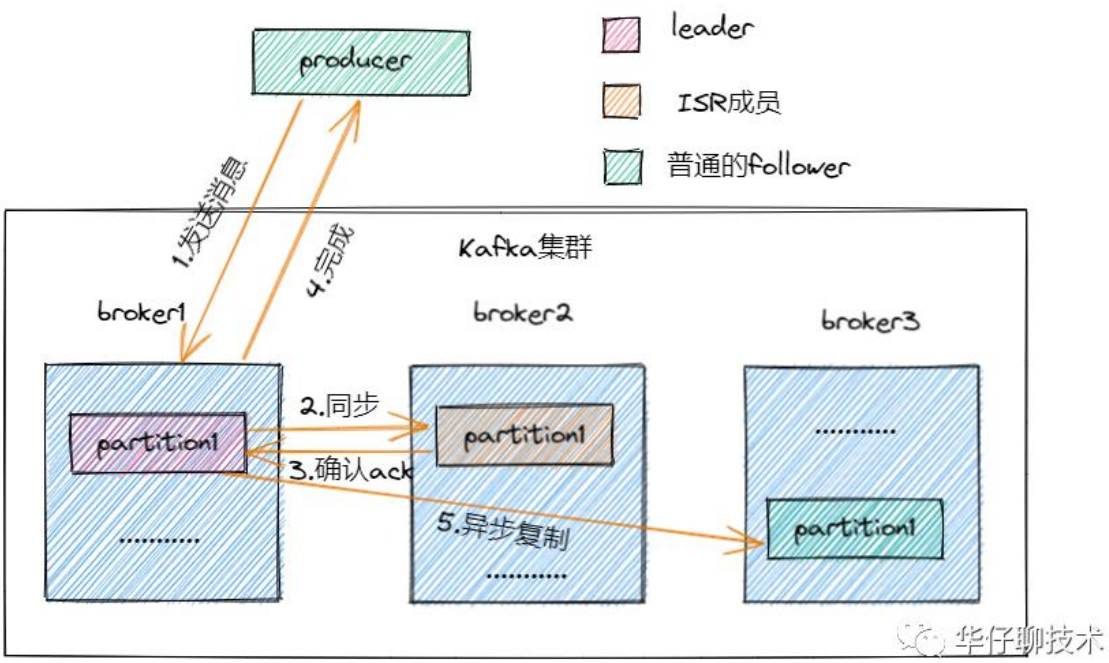
每个 partition 会对应到磁盘的一个目录，每个日志数据段都两个 index 文件和一个数据文件。 每个 Broker 需要为每个日志段文件打开 3 个句柄（2 个 index，1log），partition 越多，占用操作系统句柄数成倍增加。




## 03 端到端的延迟方面分析

Kafka 端对端延迟指 Producer 端发布消息到 Consumer 端接收消息所需要的时间，即 Consumer 端接收消息的时间减去 Producer 端发布消息的时间。

分区越多，副本之间需要同步的数据就越多。消息需要在所有 ISR 副本列表同步复制完成，才会暴露给消费者，那复制数据所花的时间就是延迟时间的主要部分




## 04 高可用性方面分析

在 Kafka 集群内部，所有的数据副本采用自动化的方式管理且会确保所有副本之间的数据是保持同步状态的。当有 Broker 节点出现故障，需要重新选举 leader，那对应故障副本的分区均无法使用，选举完成后，需要做数据同步，分区越多，服务不可用时间就会变长。

### 1.5.5.5 如何保证 Kafka 中消息有序

Kafka 不能保证消息全局有序，但可以保证分区有序性，分区与分区之间是无序的。

可以从以下三个方面来入手分析：

## 01 生产端 Producer

**消息重试可能会导致乱序**

解决办法：

设置 `max.in.flight.requests.per.connection` 该参数指定了生产者在收到 Broker 响应之前可以发送多少个消息。它的值越高，就会占用越多的内存，同时也会提升吞吐量。把它设为 1 就可以保证消息是按照发送的顺序写入服务器的。

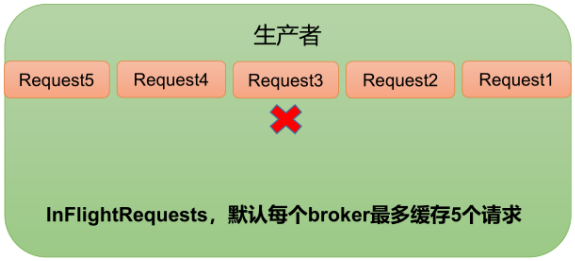
1. kafka 在 1.x 版本之前，保证数据单分区内有序，条件如下：

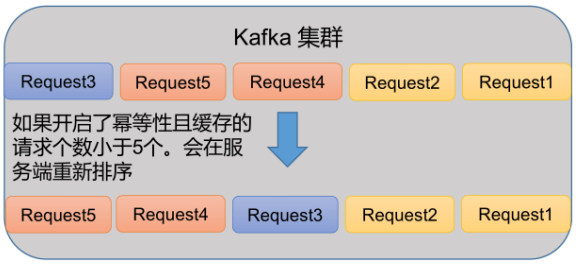
   max.in.flight.requests.per.connection=1（不需要考虑是否开启幂等性）。

2. kafka 在 1.x 及以后版本保证数据单分区有序，条件如下：

未开启幂等性：max.in.flight.requests.per.connection 需要设置为 1。

开启幂等性：max.in.flight.requests.per.connection 需要设置小于等于 5。因为在 kafka1.x 以后，启用幂等后，kafka 服务端会缓存 producer 发来的最近 5 个 request 的元数据，故无论如何，都可以保证最近 5 个 request 的数据都是有序的。




## 02 服务端 Broker

**分区 Partition 进行数量改变的时候，就不能保证消息顺序了**

解决办法：需要在动态变更分区的时候，考虑对业务的影响。有可能需要根据业务和当前分区需求，重新划分消息类别。

## 03 消费端 Consumer

**消费者多线程并发消费消息，不做处理的话可能就会导致乱序**

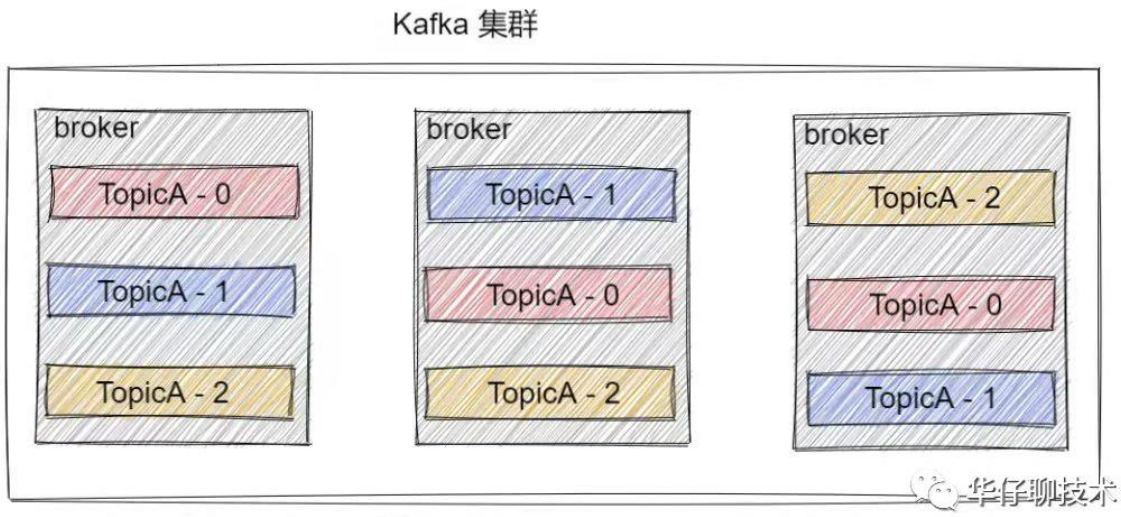
解决办法：可以通过写多个内存队列，将消息根据 key 进行哈希写到同一个队列中，然后对于多个线程，每个线程分别消息一个队列即可保证消息顺序。

### 1.5.5.6 Kafka 为什么不支持读写分离？

1) 场景不一致：读写分离架构适用于那种读操作负载很大，但写操作相对不频繁的场景，但是 Kafka 显然不适合这种场景。
2) 延迟问题：Kafka 通过 PULL 方式来实现数据同步，因此 Leader 副本和 Follower 副本存在数据不一致的情况， 如果允许 Follower 副本提供读服务的话，就会带来消息滞后的问题。

#### 1.5.5.7 Kafka 副本有哪两种，作用是什么？

为实现【数据备份】的功能，保证集群中的某个节点发生故障时，该节点上的 Partition 数据不丢失，且 Kafka 仍然能够继续工作，为此 Kafka 提供了副本机制，**一个 Topic 的每个 Partition 都有若干个副本：一个 Leader 副本和若干个 Follower 副本**。




1)Leader 主副本负责对外提供读写数据服务。
2)Follower 从副本只负责和 Leader 副本保持数据同步，并不对外提供任何服务。

#### 1.5.5.8 Kafka 能否手动删除消息？

Kafka 是支持手动删除消息的， 当然它本身提供了消息留存策略，能够自动删除过期的消息。可以从两方面答：

手动删除消息：

1. 对于普通消息：我们可以使用 Kafka-delete-records 命令或者通过程序调用 Admin.deleteRecords 方法来删除消息。两者底层都是调用 Admin 的 deleteRecords 的方法，**通过将分区的 LEO 值抬高来间接删除消息**。
2. 对于设置了 key 并且参数 cleanup.policy=compact 的消息：我们可以**构造一条 <Key， null> 的消息发送给 Broker，依靠 Log Cleaner 组件提供的功能删除掉该 Key 的消息**。

自动删除消息：

**日志删除(Log Retention)**：按照一定的保留策略直接删除不符合条件的日志分段(LogSegment)。

**日志压缩(Log Compaction)**：针对每个消息的 key 进行整合，对于有相同 key 的不同 value 值，只保留最后一个版本。

## 01 日志删除

Kafka 的日志管理器(LogManager)中有一个专门的日志清理任务通过周期性检测和删除不符合条件的日志分段文件(LogSegment)，这里我们可以通过设置 Kafka Broker 端的参数「log.retention.check.interval.ms」，默认值为 300000，即 5 分钟。

**在 Kafka 中一共有 3 种保留策略：**

**基于时间策略**

日志删除任务会周期检查当前日志文件中是否有保留时间超过设定的阈值(retentionMs) 来寻找可删除的日志段文件集合(deletableSegments)。

其中 retentionMs 可以通过 Kafka Broker 端的这几个参数的大小判断的 log.retention.ms > log.retention.minutes > log.retention.hours 优先级来设置，默认情况只会配置 log.retention.hours 参数，值为 168 即为 7 天。

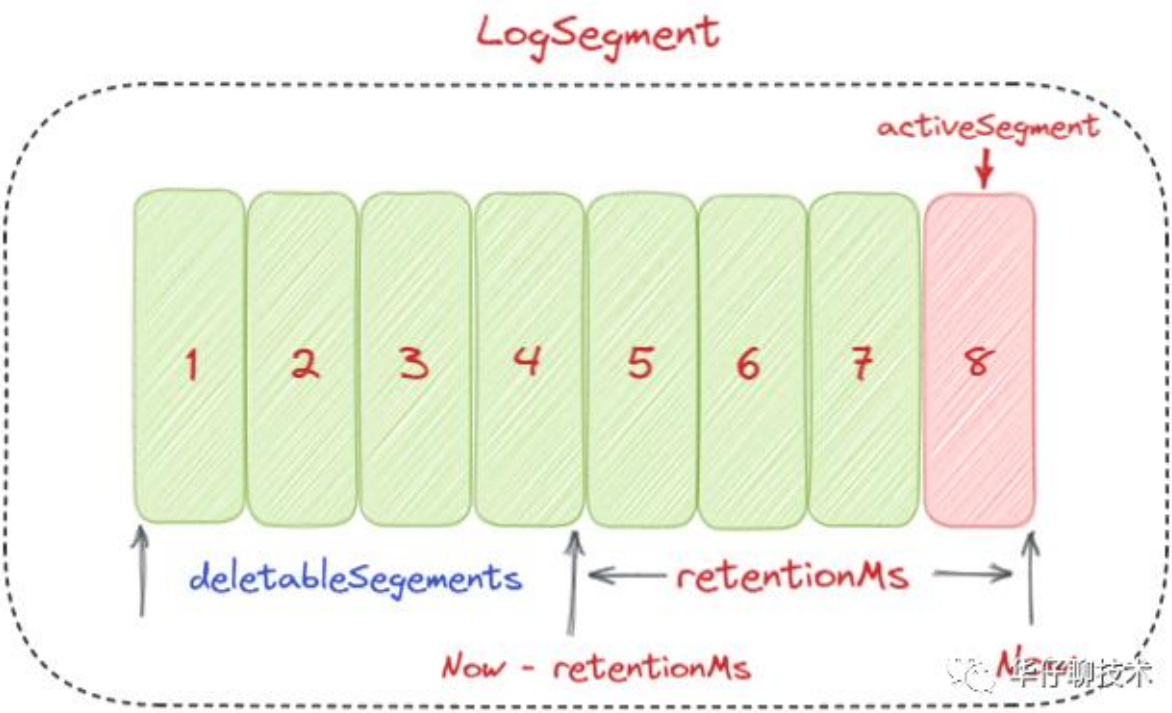
这里需要注意：删除过期的日志段文件，并不是简单的根据该日志段文件的修改时间计算的，而是要根据该日志段中最大的时间戳 largestTimeStamp 来计算的，**首先要查询该日志分段所对应的时间戳索引文件，查找该时间戳索引文件的最后一条索引数据，如果时间戳值大于 0，则取值，否则才会使用最近修改时间(lastModifiedTime)。**

**【删除步骤】：**

1. 首先从 Log 对象所维护的日志段的跳跃表中移除要删除的日志段，用来确保已经没有线程来读取这些日志段。

2. 将日志段所对应的所有文件，包括索引文件都添加上“.deleted”的后缀。

3. 最后交给一个以“delete-file”命名的延迟任务来删除这些以“ .deleted ”为后缀的文件。默认 1 分钟执行一次， 可以通过 file.delete.delay.ms 来配置。




**基于日志大小策略**

日志删除任务会周期检查当前日志大小是否超过设定的阈值(retentionSize) 来寻找可删除的日志段文件集合(deletableSegments)。

其中 retentionSize 这里我们可以通过 Kafka Broker 端的参数 log.retention.bytes 来设置， 默认值为-1，即无穷大。

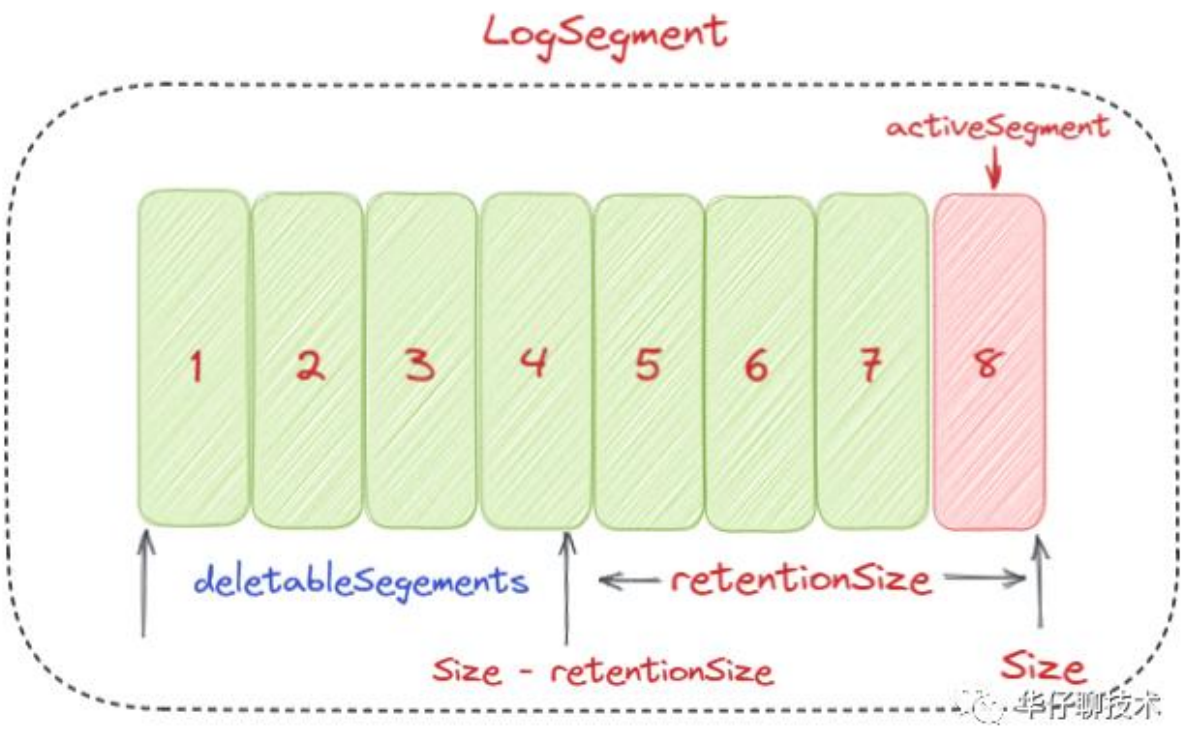
这里需要注意的是 **log.retention.bytes 设置的是 Log 中所有日志文件的大小**，而不是单个日志段的大小。单个日志段可以通过参数 log.segment.bytes 来设置，默认大小为 1G。

**【删除步骤】：**

1. 首先计算日志文件的总大小 Size 和 retentionSize 的差值，即需要删除的日志总大小。

2. 然后从日志文件中的第一个日志段开始进行查找可删除的日志段的文件集合(deletableSegments)

3. 找到后就可以进行删除操作了。




**基于日志起始偏移量**

该策略判断依据是日志段的下一个日志段的起始偏移量 baseOffset 是否小于等于 logStartOffset，如果是，则可以删除此日志分段。

**【如下图所示 删除步骤】：**

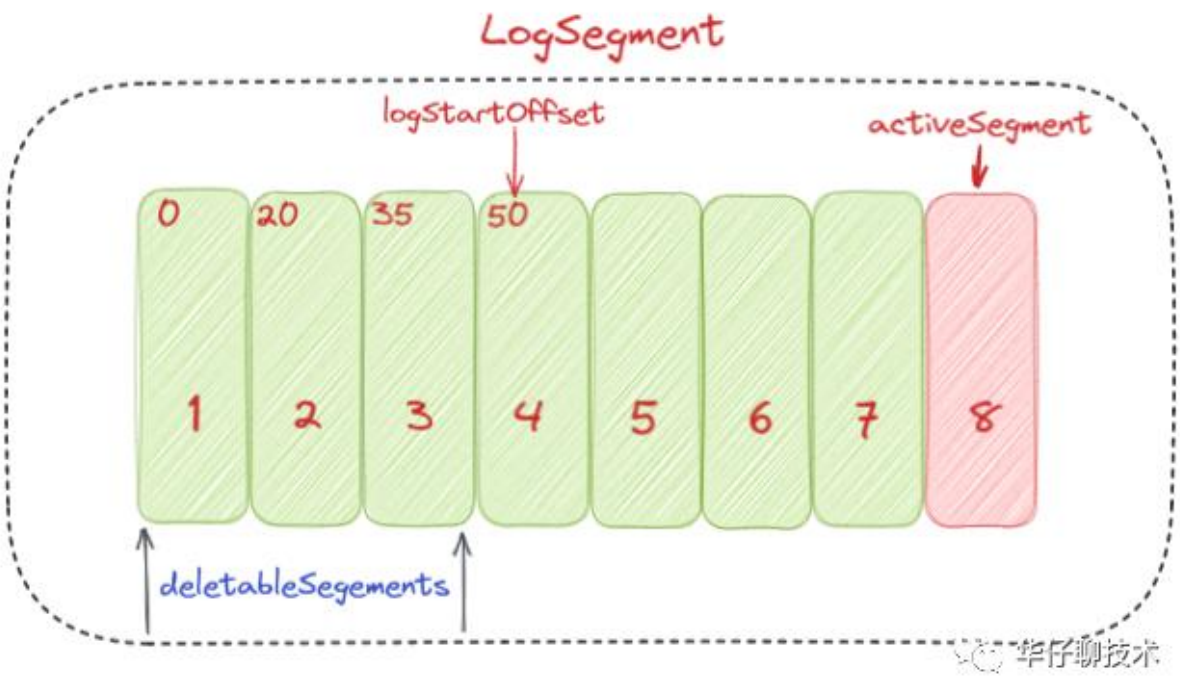
1. 首先从头开始遍历每个日志段，日志段 1 的下一个日志分段的起始偏移量为 20，小于 logStartOffset 的大小，将日志段 1 加入 deletableSegments。

2. 日志段 2 的下一个日志偏移量的起始偏移量为 35，也小于 logStartOffset 的大小，将日志分段 2 页加入 deletableSegments。

3. 日志段 3 的下一个日志偏移量的起始偏移量为 50，也小于 logStartOffset 的大小，将日志分段 3 页加入 deletableSegments。

4. 日志段 4 的下一个日志偏移量通过对比后，在 logStartOffset 的右侧，那么从日志段 4 开始的所有日志段都不会加入 deletableSegments。

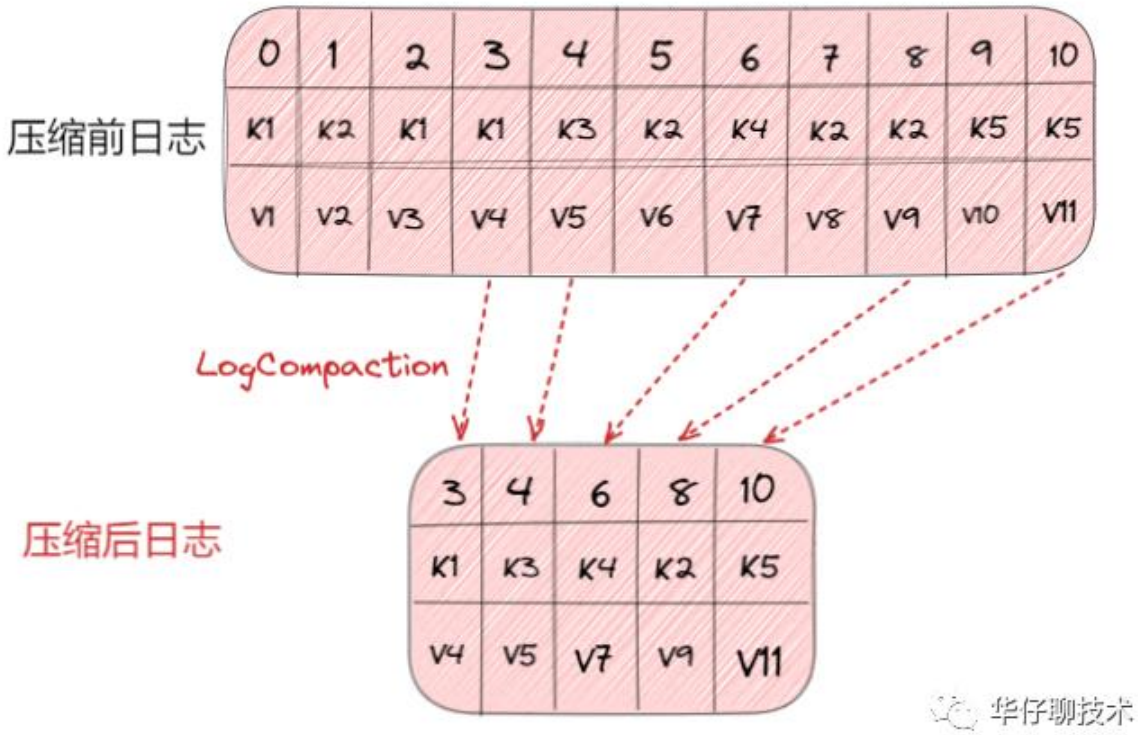
5. 待收集完所有的可删除的日志集合后就可以直接删除了。




**02** 日志压缩

日志压缩 Log Compaction 对于有相同 key 的不同 value 值，只保留最后一个版本。如果应用只关心 key 对应的最新 value 值，则可以开启 Kafka 相应的日志清理功能，Kafka 会定期将相同 key 的消息进行合并，只保留最新的 value 值。

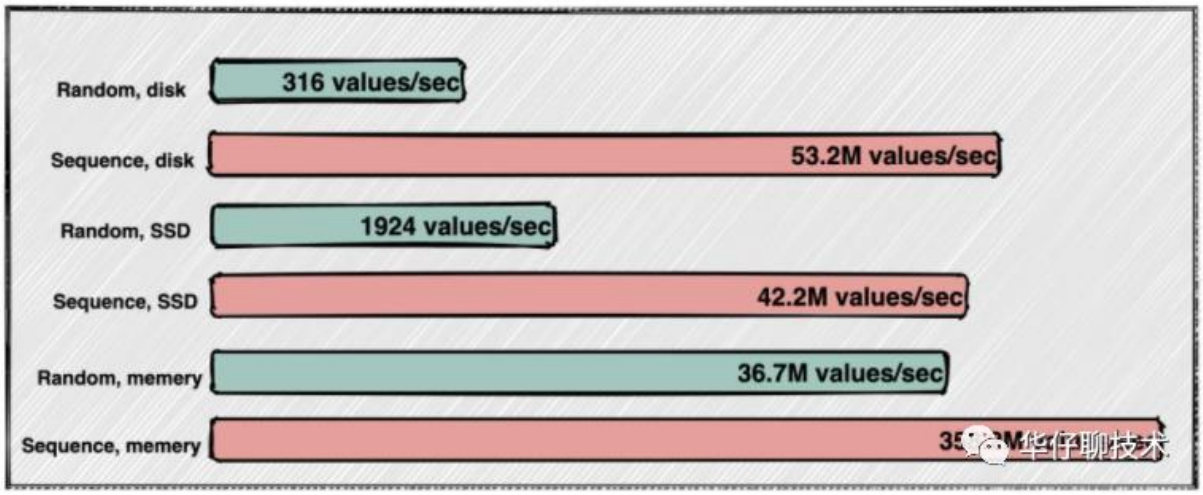
Log Compaction 可以类比 Redis 中的 RDB 的持久化模式。我们可以想象下，如果每次消息变更都存 Kafka，在某一时刻， Kafka 异常崩溃后，如果想快速恢复，可以直接使用日志压缩策略， 这样在恢复的时候只需要恢复最新的数据即可，这样可以加快恢复速度。




### 1.5.5.9 Kafka 读写数据为什么这么快？

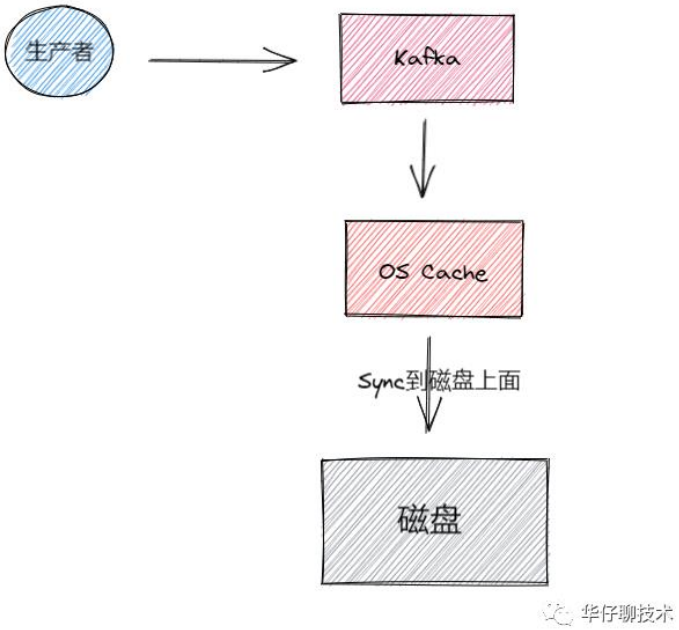
**01** 顺序追加写

**kafka 在写数据的时是以「磁盘顺序写」的方式来进行落盘的，即将数据追加到文件的末尾**。对于普通机械磁盘，如果是随机写的话，涉及到磁盘寻址的问题，导致性能极低，但是如果只是按照顺序的方式追加文件末尾的话，这种磁盘顺序写的性能基本可以跟写内存的性能差不多的。下图所示普通机械磁盘的顺序 I/O 性能指标是 53.2M values/s。



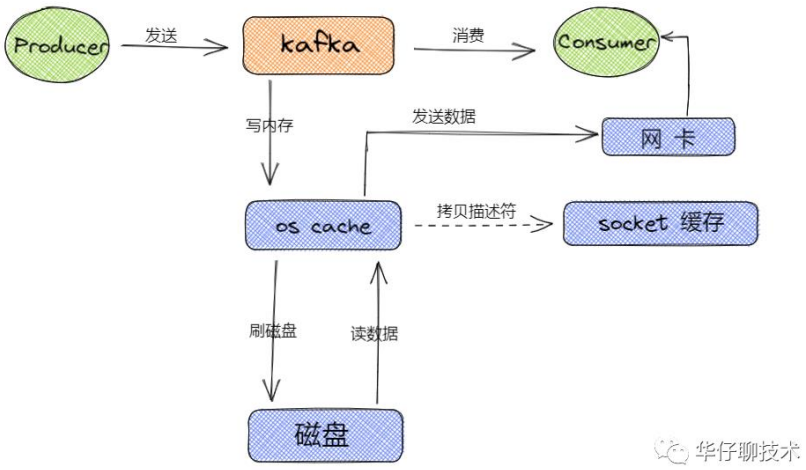
## 02Page Cache

首先 Kafka **为了保证磁盘写入性能，通过 mmap 内存映射的方式利用操作系统的 Page Cache 异步写入**。也可以称为 os cache，意思就是操作系统自己管理的缓存。那么在写磁盘文件的时候，就可以先直接写入 os cache 中，接下来由操作系统自己决定什么时候把 os cache 里的数据真正刷入到磁盘中, 这样大大提高写入效率和性能。如下图所示:




## 03 零拷贝技术

kafka 为了解决内核态和用户态数据不必要 Copy 这个问题, **在读取数据的时候就引入了「零拷贝技术」**。即让操作系统的 os cache 中的数据直接发送到网卡后传出给下游的消费者，中间跳过了两次拷贝数据的步骤，从而减少拷贝的 CPU 开销, 减少用户态内核态的上下文切换次数, 从而优化数据传输的性能, 而 Socket 缓存中仅仅会拷贝一个描述符过去，不会拷贝数据到 Socket 缓存，如下图所示：

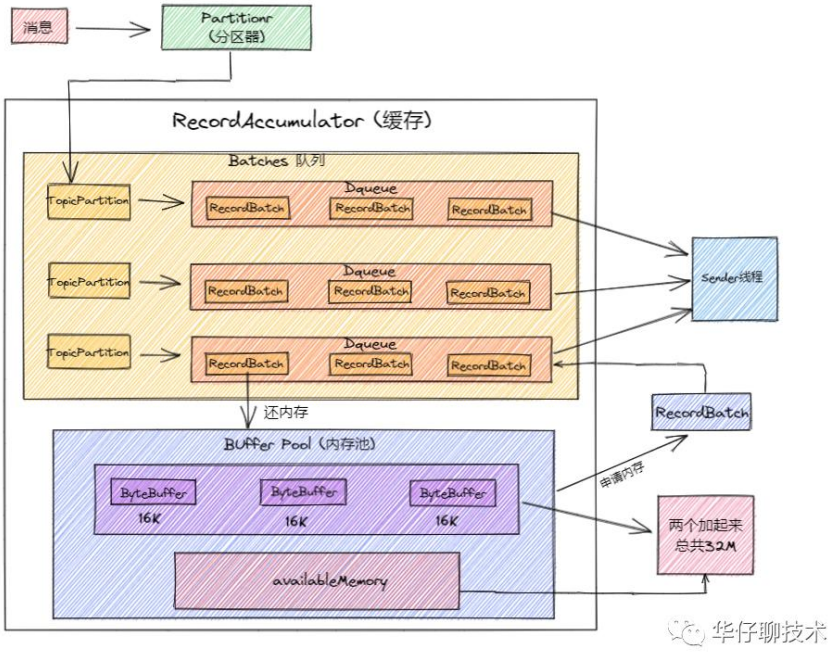



在 Kafka 中主要有以下两个地方使用到了「零拷贝技术」：

1) **基于 mmap 机制实现的索引文件**：首先索引文件都是基于 MappedByBuffer 实现，即让用户态和内核态来共享内核态的数据缓冲区，此时数据不需要 Copy 到用户态空间。虽然 mmap 避免了不必要的 Copy，但是在不同操作系统下， 其创建和销毁成功是不一样的，不一定都能保证高性能。所以在 Kafka 中只有索引文件使用了 mmap。
2) **基于 sendfile 机制实现的日志文件读写**：在 Kafka 传输层接口中有个 TransportLayer 接口，它的实现类中有使用了 Java FileChannel 中 transferTo 方法。该方法底层就是使用 sendfile 实现的零拷贝机制， 目前只是在 I/O 通道是普通的 PLAINTEXT 的时候才会使用到零拷贝机制。

## 04 消息批量发送

Kafka 在发送消息的时候并不是一条条的发送的，而是会把多条消息合并成一个批次 Batch 进行处理发送，消费消息也是同样，一次拉取一批次的消息进行消费。如下图所示：




## 05 数据压缩

在 Kafka 中三个端都使用了优化后的压缩算法，压缩有助于提高吞吐量, 降低延迟并提高磁盘利用率。Kafka 底层支持多种压缩算法: lz4, snappy, gzip, 从 Kafka 2.1.0 开始新增了 ZStandard 算法, 该算法是 Facebook 开源的压缩算法, 能提供超高的压缩比。

在 Kafka 中, 压缩可能会发生在两个地方: 生产者端和 Broker 端。一句话总结下压缩和解压缩, 即 **Producer 端压缩, Broker 端保持, Consumer 端解压缩，这样可以节省大量的网络和磁盘开销**。

### 1.5.5.10 什么是消费者组，有什么用？

首先我来看看什么是「消费者组」：

消费者组 Consumer Group，顾名思义就是由多个 Consumer 组成，且拥有一个公共且唯一的 Group ID。组内每个消费者负责消费不同分区的数据，但一个分区只能由一个组内消费者消费，消费者组之间互不影响。

**为什么 Kafka 要设计 Consumer Group, 只有 Consumer 不可以吗?**

我们知道 Kafka 是一款高吞吐量，低延迟，高并发, 高可扩展的消息队列产品， 那么如果某个 Topic 拥有数百万到数千万的数据量， 仅仅依靠 Consumer 进程消费， 消费速度可想而知， 所以需要一个扩展性较好的机制来保障消费进度， 这个时候 Consumer Group 应运而生， Consumer Group 是 Kafka 提供的可扩展且具有容错性的消费者机制。

Kafka Consumer Group 特点如下:

1) 每个 Consumer Group 有一个或者多个 Consumer。
2) 每个 Consumer Group 拥有一个公共且唯一的 Group ID。
3) Consumer Group 在消费 Topic 的时候，Topic 的每个 Partition 只能分配给组内的某个 Consumer，只要被任何 Consumer 消费一次, 那么这条数据就可以认为被当前 Consumer Group 消费成功。

为什么要有消费者组的原因：

1. 实现消息的负载均衡：消费组中的每个消费者只消费其中的一部分消息，从而实现消息的负载均衡。

2. 实现消息的高可用性：消费组中的多个消费者可以共同消费所有分区中的消息，当某个消费者出现故障时，其他消费者可以接管该消费者的分区，从而实现消息的高可用性。

3. 实现消息的顺序性：消费组中的每个消费者只消费其中的一部分消息，从而保证每个消费者消费的消息是有序的。

#### 1.5.5.11 Kafka 中 Offset 的作用是什么，如何进行维护？

在 Kafka 中每个 Topic 分区下面的每条消息都被赋予了一个唯一的 ID 值，用来标识它在分区中的位置。这个 ID 值就被称为位移「Offset」或者叫偏移量，一旦消息被写入到日志分区中，它的位移值将不能被修改。

**01 位移 Offset 管理方式**

**Kafka 目前采用内部 Topic：__consumer_offsets 来保存 Offset**

**__consumer_offsets**：**用来保存 Kafka Consumer 提交的位移信息**，另外它是由 Kafka 自动创建的，和普通的 Topic 相同，它的消息格式也是 Kafka 自己定义的，我们无法进行修改。

**__consumer_offsets 有 3 种消息格式**：

1) 用来保存 Consumer Group 信息的消息。
2) 用来删除 Group 过期位移甚至是删除 Group 的消息，也可以称为 tombstone 消息，即墓碑消息，它的主要特点是空消息体，一旦某个 Consumer Group 下的所

有 Consumer 位移数据都已被删除时，Kafka 会向 __consumer_offsets 主题的对应分区写入 tombstone 消息，表明要彻底删除这个 Group 的信息。

3) 用来保存位移值 Offset。

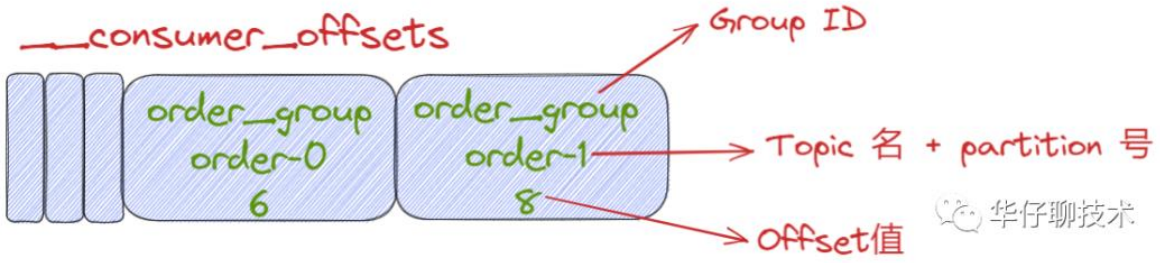
**__consumer_offsets 消息格式分析揭秘：**

1) 消息格式我们可以简单理解为是一个 KV 对。Key 和 Value 分别表示消息的键值和消息体。
2) 那么 Key 存什么呢?既然是存储 Consumer 的位移信息，在 Kafka 中，Consumer 数量会很多，必须有字段来标识位移数据是属于哪个 Consumer 的，怎么来标识 Consumer 字段呢?我们知道 Consumer Group 会共享一个公共且唯一的 Group ID，那么只保存它就可以了吗?我们知道 Consumer 提交位移是在分区的维度进行的，很显然，key 中还应该保存 Consumer 要提交位移的分区。

总结：

1. 位移主题的 **Key** 中应该保存 3 部分内容：消费者组 ID、Topic 名、分区号
2. **value** 可以简单认为存储的是 offset 值，当然底层还存储其他一些元数据，帮助 Kafka 来完成一些其他操作，比如删除过期位移数据等。

**__consumer_offsets 消息格式示意图：**




## 02__consumer_offsets 创建

__consumer_offsets 是怎么被创建出来的呢? 当 Kafka 集群中的第一个 Consumer 启动时，Kafka 会自动创建__consumer_offsets。

分区数默认是 50，副本个数默认是 3；Kafka 会自动创建有 50 个分区的__consumer_offsets，具体 Consumer Group 的消费情况存储到哪个分区计算公式如下：

abs(GroupId.hashCode()) % NumPartitions

这样就可以保证 Consumer Offset 信息与 Consumer Group 对应的 Coordinator

处于同一个 Broker 节点上。

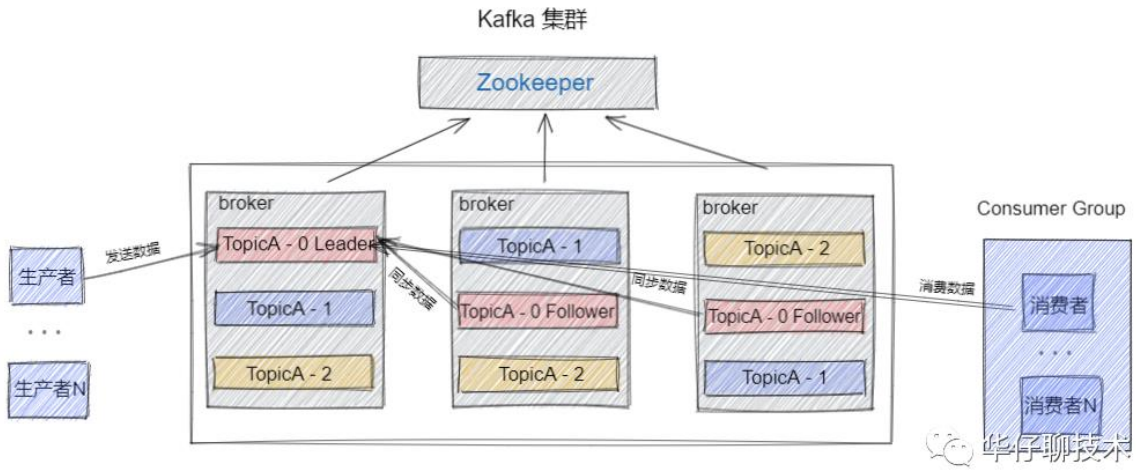
### 03 offset 提交方式

### 注意：提交时不是当前 Offset，是 offset + 1。

1. **自动提交 offset（默认 1s）**：默认的提交方式，消费者会定期自动提交 offset。这种方式的优点是简单方便，不需要额外的代码来处理 offset，但是缺点是不够灵活，无法控制提交的时间和频率，可能会导致 offset 提交不及时或者提交过于频繁。
2. **手动同步提交 offset**：手动同步提交 offset 是消费者在消费完一批消息后，手动调用 commitSync() 方法来提交 offset。这种方式的优点是比较可靠，可以确保 offset 提交成功，但是缺点是比较耗时，需要等待 offset 提交成功后才能继续消费。
3. **手动异步提交 offset**：手动异步提交 offset 是消费者在消费完一批消息后，手动调用 commitAsync() 方法来提交 offset。这种方式的优点是比较快速，不需要等待 offset 提交成功就可以继续消费，但是缺点是不够可靠，可能会出现 offset 提交失败的情况。
4. **手动提交指定 partition 的 offset**：手动提交指定 partition 的 offset 是消费者在消费完一个 partition 的消息后，手动调用 commitSync(Map<TopicPartition, OffsetAndMetadata> offsets) 方法来提交指定 partition 的 offset。这种方式的优点是比较灵活，可以控制每个 partition 的 offset，但是缺点是比较麻烦，需要额外的代码来处理 offset。

### 1.5.5.12 谈谈你对 Kafka 集群架构是如何理解的？

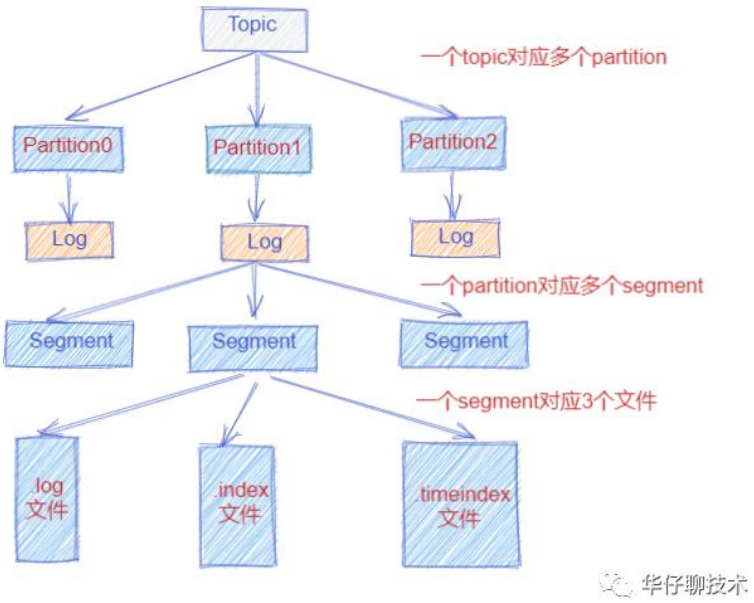
### 01 Kafka 整体架构图




一个典型的 Kafka 集群中包含若干 Producer，若干 Broker「Kafka 支持水平扩展，一般 Broker 数量越多，集群吞吐率越高」，若干 Consumer Group，以及一个 Zookeeper 集群。

Kafka 通过 Zookeeper 管理集群配置，选举 Leader，以及在 Consumer Group 发生变化时进行 Rebalance。Producer 使用 push 模式将消息发布到 Broker，Consumer 使用 pull 模式从 Broker 订阅并消费消息。

## 02 Kafka 存储机制




Producer 端生产的消息会不断追加到 log 文件末尾，这样文件就会越来越大, 为了防止 log 文件过大导致数据定位效率低下，Kafka 采取了分片和索引机制。

它将每个 Partition 分为多个 Segment 每个 Segment 对应 3 个文件：

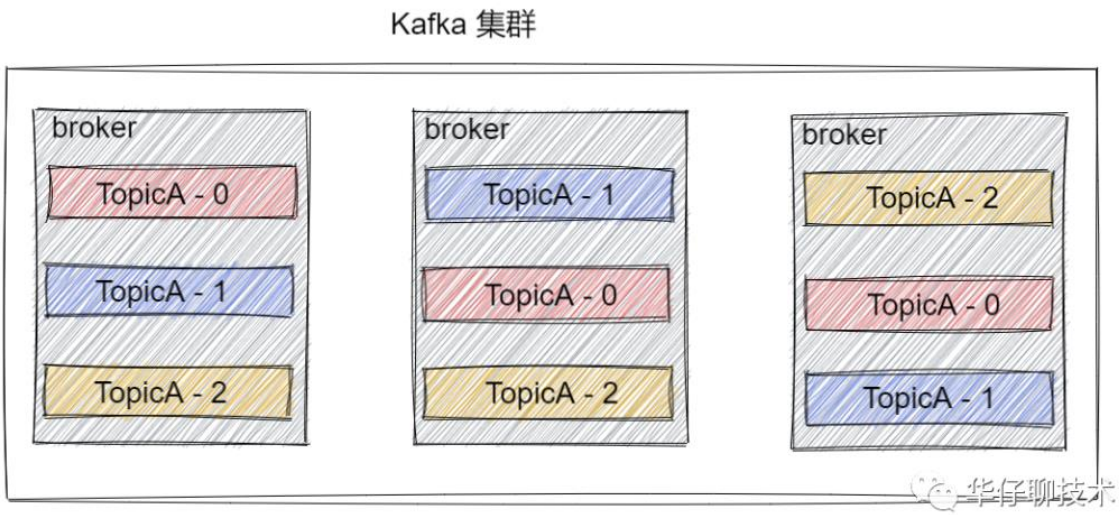
1) .index 索引文件

2) .log 数据文件

3) .timeindex 时间索引文件

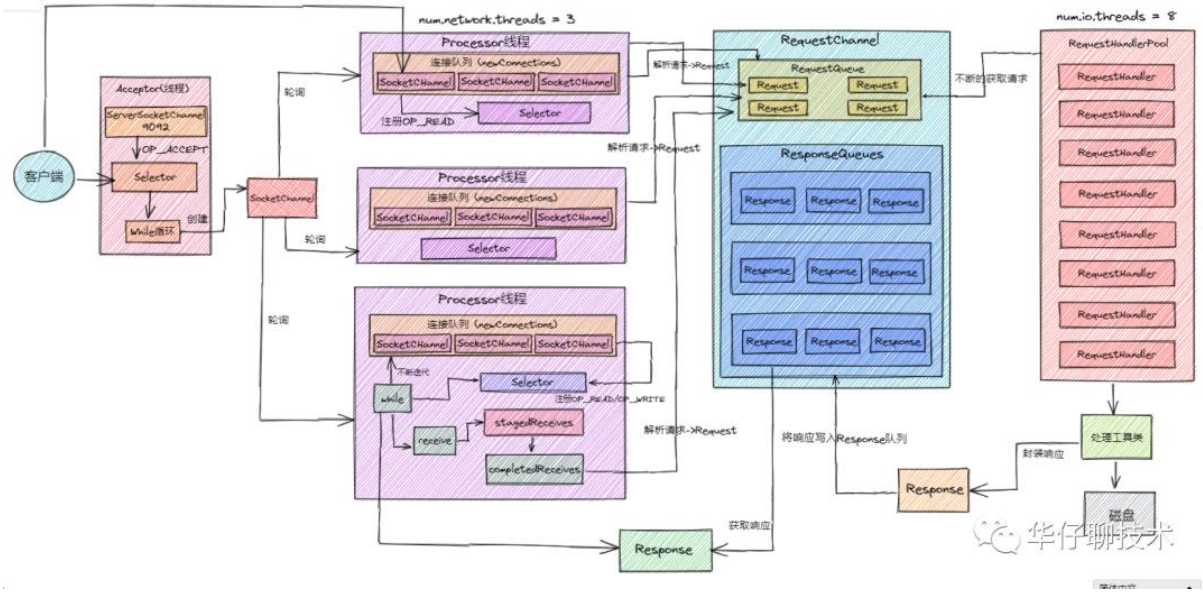
这些文件都位于同一文件夹下面，该文件夹的命名规则为：topic 名称-分区号。

## 03 Kafka 副本机制




Kafka 中的 Partition 为了保证数据安全，每个 Partition 可以设置多个副本。此时我们对分区 0,1,2 分别设置 3 个副本。而且每个副本都是有「角色」之分的，它们会选取一个副本作为 Leader 副本，而其他的作为 Follower 副本，我们的 Producer 端在发送数据的时候，只能发送到 Leader Partition 里面 ，然后 Follower Partition 会去 Leader Partition 自行同步数据, Consumer 消费数据的时候，也只能从 Leader 副本那去消费数据的。

## 04 Kafka 网络模型

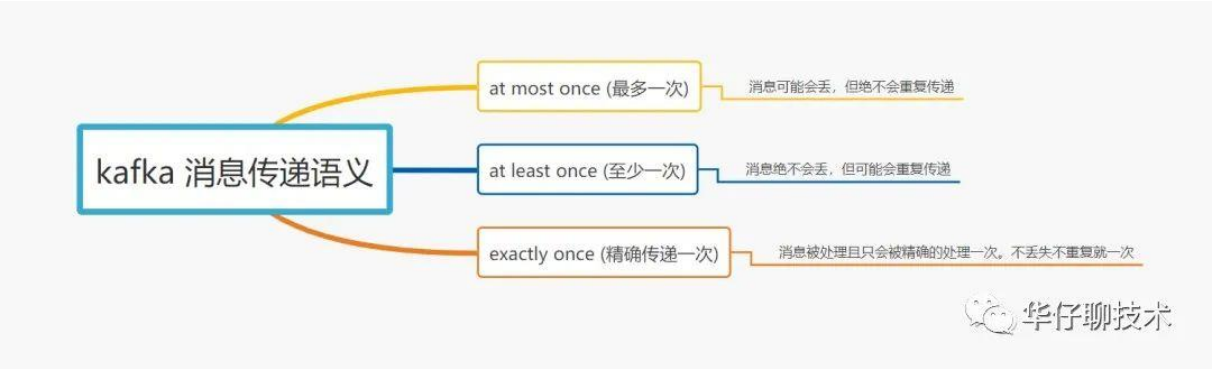



Kafka 采用多路复用方案，Reactor 设计模式，并引用 Java NIO 的方式更好的解决网

络超高并发请求问题。

#### 1.5.5.13 谈谈你对 Kafka 消息语义是如何理解的

「消息传递语义」就是 Kafka 提供的 Producer 和 Consumer 之间的消息传递过程中消息传递的保证性。主要分为三种：



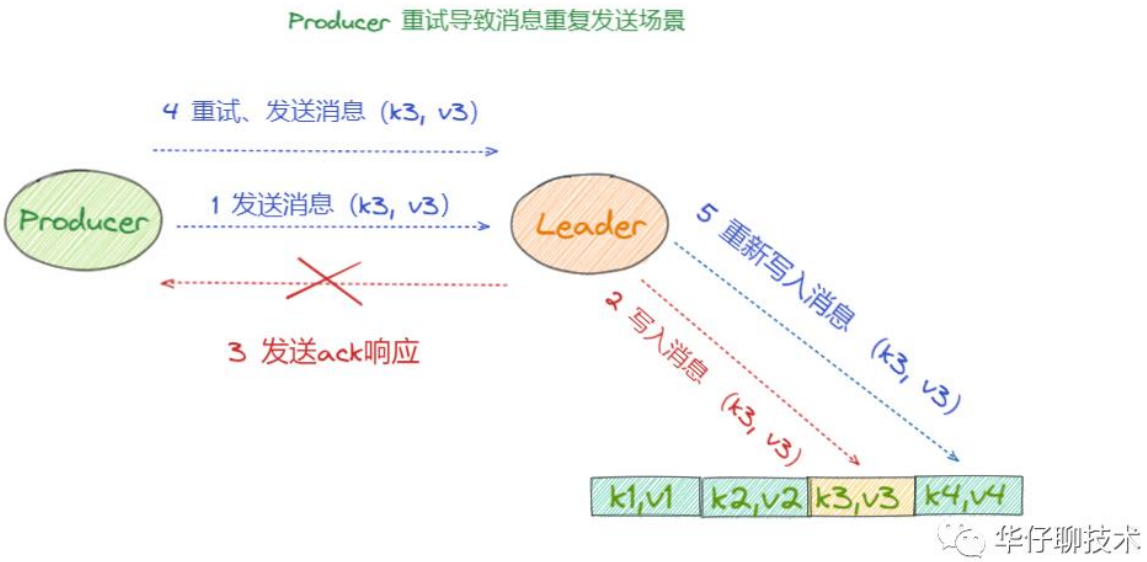

**01 Producer 端**

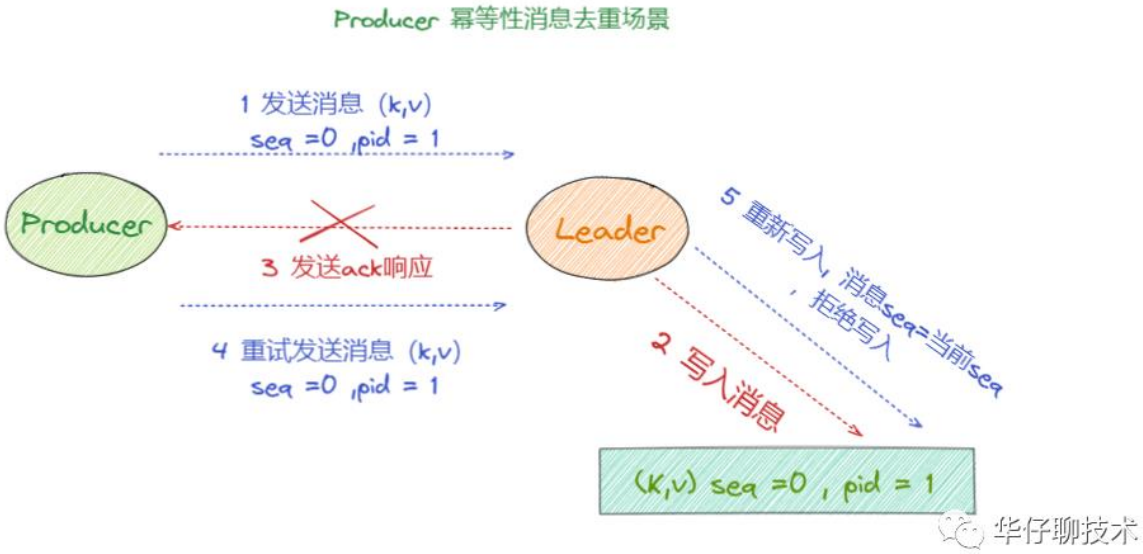
生产者发送语义：首先当 Producer 向 Broker 发送数据后，会进行消息提交，如果成功则消息不会丢失。因此发送一条消息后，可能会有几种情况发生：

1) 遇到网络问题造成通信中断， 导致 Producer 端无法收到 ack，Producer 无法准确判断该消息是否已经被提交， 又重新发送消息，这就可能造成 「at least once」语义。
2) 在 Kafka 0.11 之前的版本，会导致消息在 Broker 上重复写入(保证至少一次语义)，但在 0.11 版本开始，**通过引入「PID 及 Sequence Number」支持幂等性，保证精确一次「exactly once」语义**。

   其中启用幂等传递的方法配置：enable.idempotence = true。启用事务支持的方法配置：设置属性 transcational.id = "指定值"。
3) 可以根据 Producer 端 request.required.acks 的配置来取值。
   - **acks = 0**：由于发送后就自认为发送成功，这时如果发生网络抖动，Producer 端并不会校验 ACK 自然也就丢了，且无法重试。
   - **acks = 1**：消息发送 Leader Parition 接收成功就表示发送成功，这时只要 Leader Partition 不 Crash 掉，就可以保证 Leader Partition 不丢数据，保证「at least once」语义。
   - **acks = -1 或者 all**： 消息发送需要等待 ISR 中 Leader Partition 和 所有的

Follower Partition 都确认收到消息才算发送成功, 可靠性最高, 但也不能保证不丢数据,比如当 ISR 中只剩下 Leader Partition 了, 这样就变成 acks = 1 的情况了，保证 「at least once」语义。



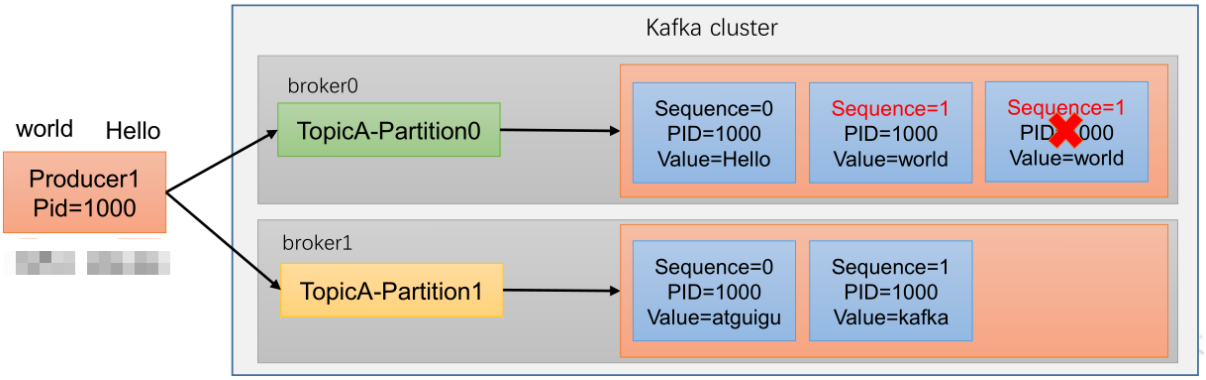




**生产者消息幂等性**

**幂等性：就是指 Producer 不论向 Broker 发送多少次重复数据，Broker 端都只会持久化一条，保证了不重复。**

**重复数据的判断标准**：具有<**PID,Partition,SeqNumber**>相同主键的消息提交时，Broker 只会持久化一条。其中 **PID 是 Kafka 每次重启都会分配一个新的**；**Partition 表示分区号；SequenceNumber 是单调自增的。**

**所以幂等性只能保证的是在单分区单会话内不重复。**




使用幂等性的方法：

开启参数 enable.idempotence 默认为 true

### 02 Consumer 端

消费者接收语义：从 Consumer 角度来剖析，我们知道 Offset 是由 Consumer 自己来维护的。

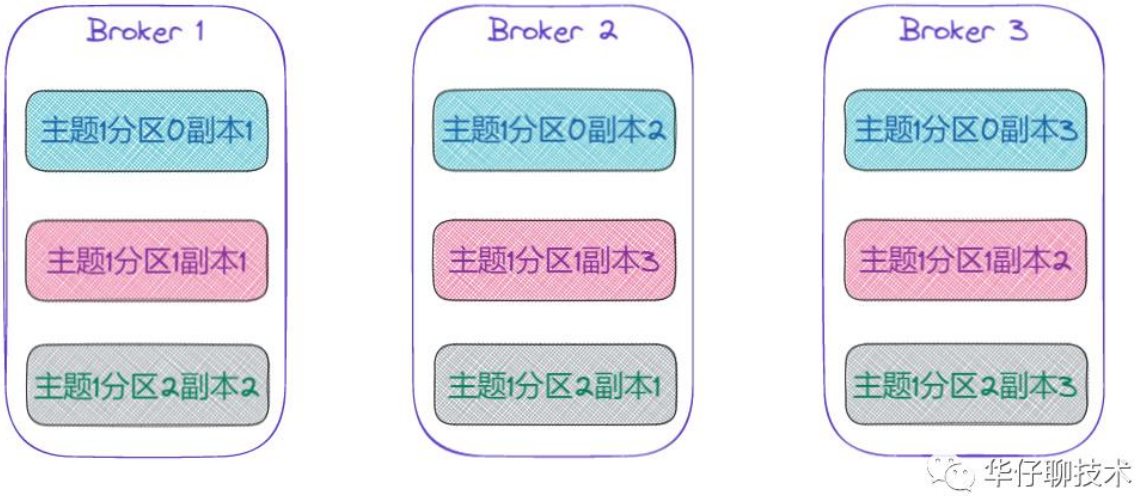
Consumer 消费消息时，有以下 2 种选择:

1) 读取消息 -> 提交 offset -> 处理消息: 如果此时保存 offset 成功，但处理消息失败，Consumer 又挂了，会发生 Reblance，新接管的 Consumer 将从上次保存的 offset 的下一条继续消费，导致消息丢失，保证「at most once」语义。
2) 读取消息 -> 处理消息 -> 提交 offset：如果此时消息处理成功，但保存 offset 失败，Consumer 又挂了，导致刚才消费的 offset 没有被成功提交，会发生 Reblance，新接管的 Consumer 将从上次保存的 offset 的下一条继续消费，导致消息重复消费，保证「at least once」语义。

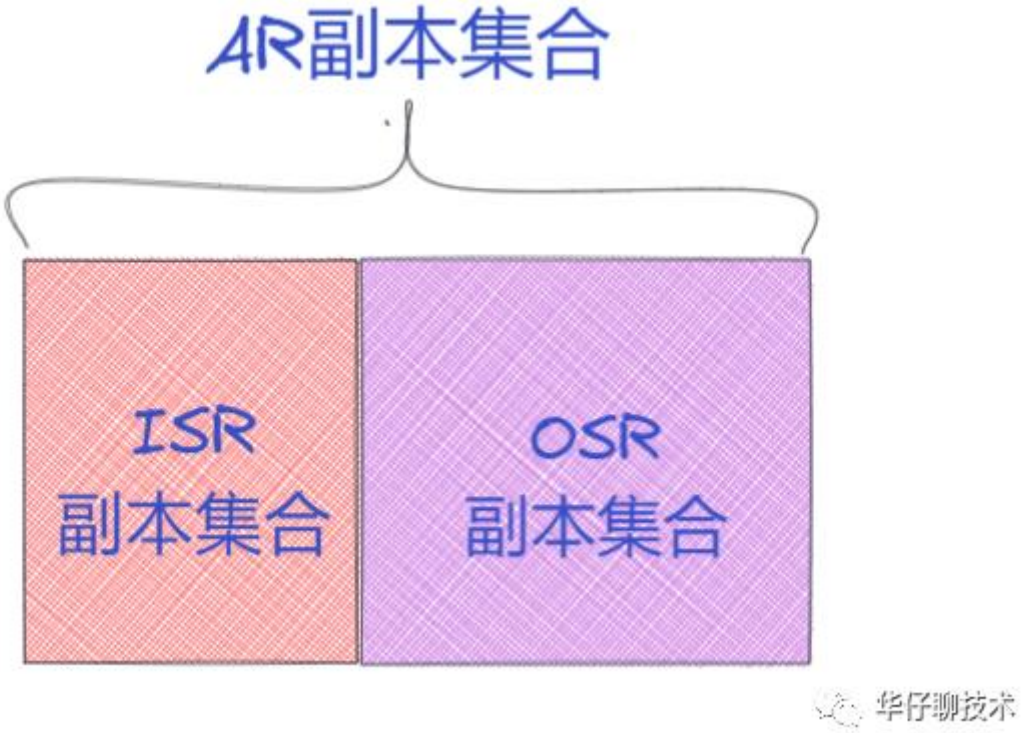
总结：默认 Kafka 提供 「at least once」语义的消息传递，允许用户通过在处理消息之前保存 Offset 的方式提供 「at most once」 语义。如果我们可以自己实现消费幂等，理想情况下这个系统的消息传递就是严格的「exactly once」，也就是保证不丢失、且只会被精确的处理一次，但是这样是很难做到的（一个思路是用 redis 存一下 Key 或者唯一 ID，消费时先去 Redis 查一下看看是不是已经消费过了）

#### 1.5.5.14 谈谈你对 Kafka 副本机制是如何理解的

Kafka 的副本机制，说白了就是一个「数据备份机制」。保证集群中的某个节点发生故障时，该节点上的 Partition 数据不丢失，且 Kafka 仍然能够继续工作，提高了系统可用性和数据持久性。同一个分区下的所有副本保存相同的消息数据，这些副本分散保存在不同的 Broker 上，保证了 Broker 的整体可用性。




Kafka 中 只有 Leader 副本才能对外进行读写服务，一个 Topic 的每个 Partition 都有若干个副本，副本分成两类：领导者副本「Leader Replica」和追随者副本「Follower Replica」。每个分区在创建时都要选举一个副本作为领导者副本，其余的副本作为追随者副本。在 Kafka 中，Follower 副本是不对外提供服务的。也就是说，任何一个 Follower 副本都不能响应客户端的读写请求。所有的读写请求都必须先发往 Leader 副本所在的 Broker，由该 Broker 负责处理。Follower 副本不处理客户端请求，它唯一的任务就是从 Leader 副本异步拉取消息，并写入到自己的提交日志中，从而实现与 Leader 副本的同步。




如上图所示，与副本相关的集合有三种：

1) AR 副本集合: 分区 Partition 中的所有 Replica 组成 AR 副本集合。
2) ISR 副本集合: 所有与 Leader 副本能保持一定程度同步的 Replica 组成 ISR 副本集合， 其中也包括 Leader 副本。
3) OSR 副本集合: 与 Leader 副本同步滞后过多的 Replica 组成 OSR 副本集合。

AR：一个分区的 AR 副本集合在分配的时候就被指定，并且只要不发生重分配集合内部副本的顺序是保持不变的

ISR：其中，Leader 副本天然就在 ISR 副本集合中，ISR 不只是有 Follower 副本集合，它必然包括 Leader 副本，另外，能够进入到 ISR 副本集合的 Follower 副本要满足一定的条件：

**完全依赖于 Broker 端参数 replica.lag.time.max.ms 参数值**；从 2.5 版本开始，默认值从 10 秒增加到 30 秒。即只要一个 Follower 副本落后 Leader 副本的时间不连续超过 30 秒，Kafka 就认为该 Follower 副本与 Leader 是同步的，即使 Follower 副本中保存的消息明显少于 Leader 副本中的消息。

如果某副本同步过程的速度持续慢于 Leader 副本的消息写入速度的时候，那么在 replica.lag.time.max.ms 时间后，该 Follower 副本就会被认为与 Leader 副本是不

同步的，因此 Kafka 会自动收缩，将其踢出 ISR 副本集合中。后续如果该副本追上了 Leader 副本的进度的话，那么它是能够重新被加回 ISR 副本集合的。

**在默认情况下，当 Leader 副本发生故障时，只有在 ISR 副本集合中的 Follower 副本才有资格被选举为新 Leader，而 OSR 中副本集合的副本是没有机会的(可以通过 unclean.leader.election.enable 进行配置执行脏选举)。**

Leader 副本如何选举出来

总结：首先说结论，分区 Leader 的选举由 Controller 进行，会同时参考 ISR 和 AR，策略主要就是选择 AR 中第一个在 ISR 中存活的副本作为 Leader；要注意的是，ISR 是动态变化的，AR 一般来说不变，只有在分区重分配时才会变化。

Leader 选举是指分区 Leader 副本的选举，它是由 Kafka Controller 负责具体执行的，当创建分区或分区副本上线的时候都需要执行 Leader 的选举动作。

有以下场景可能会触发选举：

1) **当 Controller 感知到分区 Leader 下线需要执行 Leader 选举。**

此时的选举策略是：Controller 会从 AR 副本集合(同时也在 ISR 副本集合)中按照副本的顺序取出第一个存活的副本作为 Leader。

一个分区的 AR 副本集合在分配的时候就被指定，并且只要不发生重分配集合内部副本的顺序是保持不变的，而分区的 ISR 副本集合中副本的顺序可能会改变。

注意这里是根据 AR 副本集合的顺序而不是 ISR 副本结合的顺序进行选举的。

此时如果 ISR 副本集合中没有可用的副本，还需要再检查一下所配置的 unclean.leader.election.enable 参数「默认值为 false」。如果这个参数配置为 true，那么表示允许从非 ISR 副本集合中选举 Leader，从 AR 副本集合列表中找到第一个存活的副本即为 Leader。

2) **当分区进行重分配的时候也需要进行 Leader 选举。**

此时的选举策略是：从重分配的 AR 副本集合中找到第一个存活的副本，且这个副本在当前的 ISR 副本集合中。当发生优先副本的选举时，直接将优先副本设置为 Leader 即可，AR 副本集合中的第一个副本即为优先副本。

3) **当某节点执行 ControlledShutdown 被优雅地关闭时，位于这个节点上的 Leader 副本都会下线，所以与此对应的分区需要执行 Leader 的选举。**

此时的选举策略是：从 AR 副本集合中找到第一个存活的副本，且这个副本在当前的 ISR 副本集合中，同时还要确保这个副本不处于正在被关闭的节点上。

#### 1.5.5.15 谈谈你对 Kafka 控制器及选举机制是如何理解的？

所谓的控制器「Controller」就是通过 ZooKeeper 来管理和协调整个 Kafka 集群的组件。集群中任意一台 Broker 都可以充当控制器的角色，但是在正常运行过程中，只能有一个 Broker 成为控制器。

控制器的职责主要包括：

1) 集群元信息管理及更新同步 (Topic 路由信息等)。
2) 主题管理(创建、删除、增加分区等)。
3) 分区重新分配。
4) 副本故障转移、 Leader 选举、ISR 变更。
5) 集群成员管理(通过 watch 机制自动检测新增 Broker、Broker 主动关闭、Broker 宕机等)。

**01 控制器机制**

1) Kafka 使用 Zookeeper 的临时节点来选举 Controller。
2) Zookeeper 在 Broker 加入集群或退出集群时通知 Controller。
3) Controller 负责在 Broker 加入或离开集群时进行分区 Leader 选举。

**02 控制器数据分布**

从上面表格可以看出,存储的大概有 3 大类:

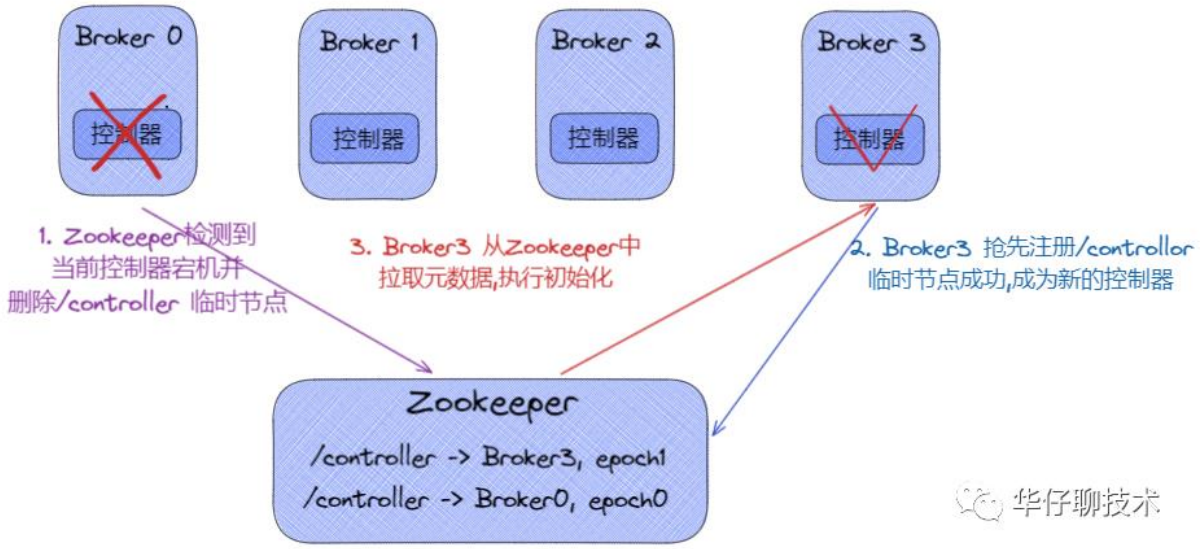
1) 所有 topic 信息：包括具体的分区信息，比如 Leader 副本是谁，ISR 集合中有哪些副本等。
2) 所有 Broker 信息：包括当前都有哪些运行中的 Broker，哪些正在关闭中的 Broker 等。
3) 涉及运维任务的副本分区：包括当前正在进行 Leader 选举以及分区重分配的分区列表等。

## 03 控制器故障转移

在 Kafka 集群运行过程中，只能有一台 Broker 充当控制器的角色，存在「单点故障」的风险，Kafka 如何应对单点故障呢?

其实 Kafka 为控制器提供故障转移功能「Failover」。指**当运行中的控制器突然宕机时，Kafka 能够快速地感知到，并立即启用备用控制器来代替之前失败控制器的过程**。

下面通过一张图来展示控制器故障转移的过程：




## 04 控制器触发选举场景

**场景一、集群首次启动时：**

**场景二、Broker 监听 /controller 节点消失时**

**场景三、Broker 监听 /controller 节点数据变化时**

## 05 控制器选举机制

Kafka Controller 节点的初始化依赖 Zookeeper 实现，具体流程如下：

1. **注册 Controller 节点**：当 Kafka 集群启动时，每个 Broker 都会尝试在 Zookeeper 中的 /controller 路径下创建一个临时节点。因为同一时刻只能存在一个 /controller 节点，所以只有一个 Broker 成功创建节点并成为 Controller。其他 Broker 会收到节点创建失败的通知，然后转为观察者（Observer）状态，监视 Controller 节点路径的变化。
2. **监听 Controller 节点**：所有非 Controller 的 Broker 都会在 Zookeeper 中对 /controller 路径设置一个 Watcher 事件。这样当 Controller 节点发生变化时（例如，Controller 失

效），所有非 Controller 就会收到一个 Watcher 事件。

3. **选举新的 Controller**：当某个 Broker 接收到 Controller 节点变化的通知后，它会再次尝试在 Zookeeper 中的 /controller 路径下创建一个临时节点。与启动时的过程类似，只有一个 Broker 能够成功创建节点并成为新的 Controller。新 Controller 会在选举成功后接管集群元数据的管理工作。

4. **更新集群元数据**：新 Controller 在选举成功后需要更新集群元数据，包括分区状态、副本状态等。同时，新控制器会通知所有相关的 Broker 更新它们的元数据信息。这样，集群中的所有 Broker 都能够知道新 Controller 的身份，并进行协同工作。

注意：临时节点的特点是在创建它的客户端（即 Broker 节点）断开连接时，它会自动被 Zookeeper 删除。这种机制保证了只有一个 Broker 节点能够成为控制器，以避免多个控制器同时对集群元数据进行操作引发的问题。

## 06 控制器脑裂问题

脑裂问题是分布式系统中经常出现的现象，Kafka 脑列问题是由于网络或其他原因导致多个 Broker 认为自己是 Controller，从而导致元数据不一致和分区状态混乱的问题。

Kafka 是通过 **epoch number**（纪元编号）来解决脑裂问题，epoch number 是一个单调递增的版本号。

脑裂问题产生和处理过程如下：

- 假设有三个 Broker，分别是 Broker 0，Broker 1 和 Broker 2。Broker 0 是 Controller，它在 ZooKeeper 中创建了/controller 节点，并设置 epoch number 值为 1。Broker 1 和 Broker 2 在/controller 节点设置了 Watcher。
- 由于某种原因，Broker 0 出现了 Full GC，导致它与 ZooKeeper 的会话超时。ZooKeeper 删除了/controller 节点，并通知 Broker 1 和 Broker 2 进行新的 Controller 选举。
- Broker 1 和 Broker 2 同时尝试在 ZooKeeper 中创建/controller 节点，假设 Broker 1 成功了，那么它就成为了新的 Controller，设置 epoch number 值为 2，并向 Broker 2 同步数据。
- Broker 0 的 Full GC 结束后，继续向 Broker 1 和 Broker 2 同步数据，Broker 1 和 Broker 2 接收到数据后，发现 epoch number 小于当前值，就会拒绝这些消息。并通知 Broker 0 最新的 epoch number，然后 Broker 0 发现自己已经不是 Controller 了，最后与新的 Controller 建立连接。

#### 1.5.5.16 TODO 谈谈 Kafka 的数据可靠性是怎么保证的？

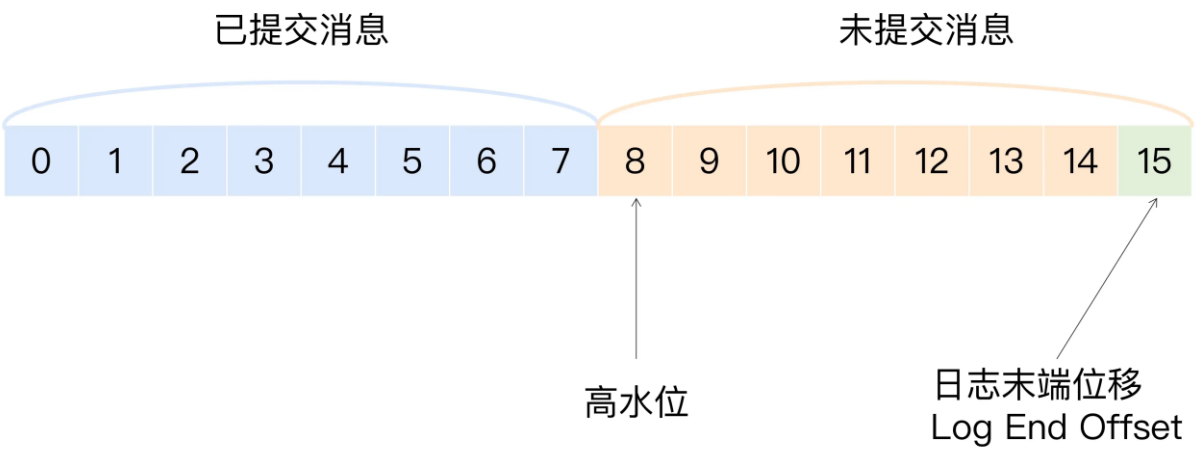
开始数据可靠性之前先看几个重要的概念：AR、OSR、ISR、HW、LEO，前面已经讲了 AR、OSR、ISR。这里我们重点讲下 HW、LEO。

- **HW**：全称「Hign WaterMark」，即高水位，它标识了一个特定的消息偏移量 offset，**消费者只能拉取到这个水位 offset 之前的消息**。
- LEO：全称「Log End Offset」，它标识当前日志文件中下一条待写入的消息的 offset，在 ISR 副本集合中的每个副本都会维护自身的 LEO。

高水位作用

在 Kafka 中，高水位的作用主要有 2 个。

- 1）定义消息可见性，即用来标识分区下的哪些消息是可以被消费者消费的。
- 2）帮助 Kafka 完成副本同步。



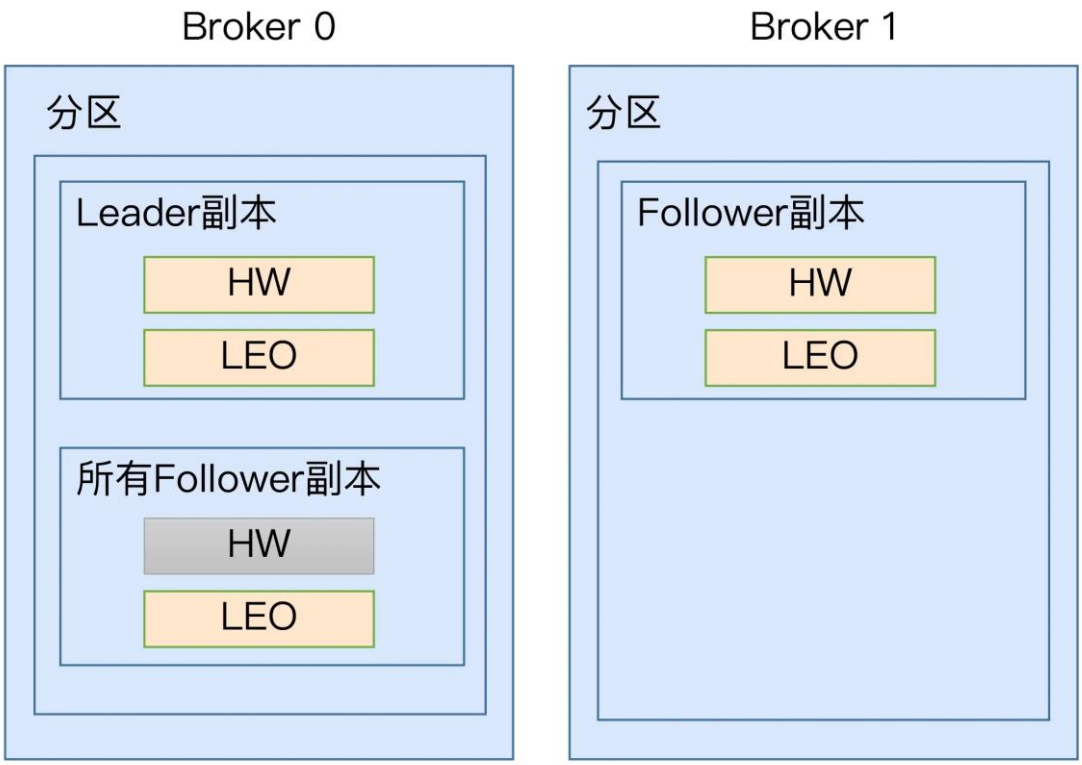

**在分区高水位以下的消息被认为是已提交消息，反之就是未提交消息。消费者只能消费已提交消息**，即图中位移小于 8 的所有消息。

**高水位和 LEO 是副本对象的两个重要属性。**Kafka 所有副本都有对应的高水位和 LEO 值，而不仅仅是 Leader 副本。只不过 Leader 副本比较特殊，Kafka 使用 Leader 副本的高水位来定义所在分区的高水位。换句话说，**分区的高水位就是其 Leader 副本的高水位**。

**注意：当集群中副本所在的 Broker 发生故障而后恢复时，副本先将数据截断（Truncation）到其 HW 处（LEO 等于 HW），然后再开始向 Leader 同步数据。**

高水位更新机制

Leader 副本所在的 Broker 上，还保存了其他 Follower 副本（也称为远程副本）的 LEO 值。主要为了**帮助 Leader 副本确定其高水位，也就是分区高水位。如下图所示。**




多副本下，各个副本中的 HW 和 LEO 演变过程

总结：只有所有副本都同步好的消息才能对外可见，而 HW 表示可见消息 offset 的高水位，理解这个前提后，就不难理解 HW 和 LEO 是如何变化的了

过程见 HW 和 LEO 的详细更新过程

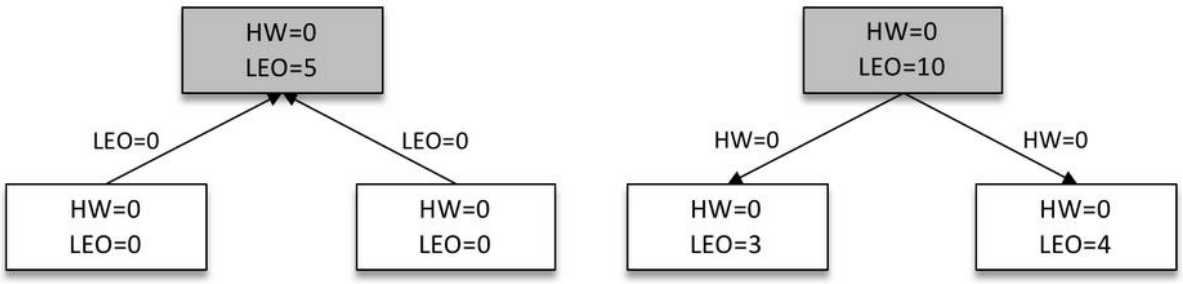
为什么要去 fetch 两次才能更新 HW 呢，因为在第二次 Follower 才会把拉完消息后的 LEO 传给 Leader，Leader 才能计算新的 HW，然后传给 Follower 后才能让 Follower 计算 HW，也就是第一次只是更新了 Follower 的 LEO，第二次才会更新 Leader 远程副本的 LEO 和 Leader 的 HW 和 Follower 的 HW

某个分区有 3 个副本分别位于 broker0、broker1 和 broker2 节点中，假设 broker0 上的副本 1 为当前分区的 Leader 副本，那么副本 2 和副本 3 就是 Follower 副本，整个消息追加的过程可以概括如下：

1. 生产者客户端发送消息至 Leader 副本（副本 1）中。
2. 消息被追加到 Leader 副本的本地日志，并且会更新日志的偏移量。
3. Follower 副本（副本 2 和副本 3）向 Leader 副本请求同步数据。
4. Leader 副本所在的服务器读取本地日志，并更新对应拉取的 Follower 副本的信息。
5. Leader 副本所在的服务器将拉取结果返回给 Follower 副本。
6. Follower 副本收到 Leader 副本返回的拉取结果，将消息追加到本地日志中，并更新日志的偏移量信息。

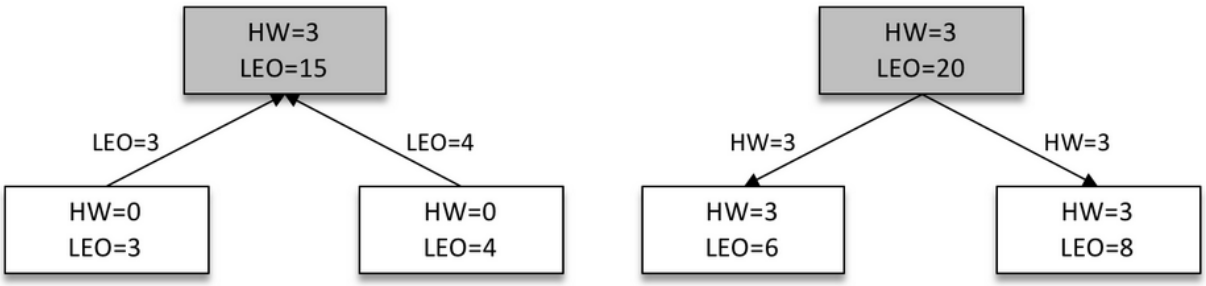
HW 和 LEO 详细过程如下：

1. 假设某时刻生产者向分区写了 5 条消息，Leader 副本的 LEO 增加至 5，并且所有副本的 HW 还都为 0。




情形1（初识状态）　　情形2

2. 第一次 Fetch：Follower 副本向 Leader 副本拉取消息，会带上自身的 LEO 信息，如上图左所示；随后 Leader 副本更新本地远程副本的 LEO 值，并计算 HW=max(HW,min(LEO1,LEO2...))**（这里取个 max 是为了防止新加入的同步副本导致 HW 受污染，保证 HW 都是单调递增的）**，返回给 Follower 副本相应的消息，并且带上自身的 HW 信息，如上图右所示。
3. Follower 副本各自都拉取了消息，并且更新各自的 LEO 为 3 和 4，同时更新 HW，算法是**比较当前 LEO 和 Leader 副本中传送过来的 HW 的值，取较小值作为自己的 HW 值，即 HW=min(HW,LEO1)；**
4. 第二次 Fetch：接下来，Follower 副本再次请求拉取 Leader 副本中的消息，同时带有 Follower 副本的 LEO 信息。如下图左所示。




情形3　　情形4

5. Leader 副本收到请求后，更新本地远程副本的 LEO 值，并计算 HW=max(HW,min(LEO1,LEO2...)).然后连同消息和 HW 一起返回给 Follower 副本，如上图右所示。两个 Follower 副本会在拉取消息后更新 LEO 并且更新自己的 HW 为 3（min（LEO，3）=3）。

由上可知，在写入数据到 Leader 后，第一轮 Fetch 只会更新 Follower 的 LEO 值，而 Leader 的 HW 和 Follower 的 HW 是在第二次 Follower 的 Fetch 中才会分别更新，并且更新是有延迟的，

Leader 的先更新，在响应之后 Follower 才会更新，所以可能会有消息丢失的问题。为了规避因高水位更新错配导致的各种不一致的问题，引入了 Leader Epoch 概念。

Leader Epoch

所谓 Leader Epoch，我们大致可以认为是 Leader 版本。它由两部分数据组成。

- 1）**Epoch**。一个单调增加的版本号。每当副本领导权发生变更时，都会增加该版本号。小版本号的 Leader 被认为是过期 Leader，不能再行使 Leader 权力。
- 2）**起始位移（Start Offset）**。Leader 副本在该 Epoch 值上写入的首条消息的位移。

具体例子：假设有两个 leader epoch <0, 0> 和 <1, 120>，那么，第一个 Leader Epoch 表示版本号是 0，这个版本的 Leader 从位移 0 开始保存消息，一共保存了 120 条消息。之后，Leader 发生了变更，版本号增加到 1，新版本的起始位移是 120；

Broker 会在内存中为每个分区都缓存 Leader Epoch 数据，同时它还会定期地将这些信息持久化到一个 checkpoint 文件中。当 Leader 副本写入消息到磁盘时，Broker 会尝试更新这部分缓存。如果该 Leader 是首次写入消息，那么 Broker 会向缓存中增加一个 Leader Epoch 条目，否则就不做更新。这样，每次有 Leader 变更时，新的 Leader 副本会查询这部分缓存，取出对应的 Leader Epoch 的起始位移，以避免数据丢失和不一致的情况。

假设仅使用 HW 会有什么问题？

假设会截断

### 1.5.5.17 谈谈 Kafka 消息分配策略有哪些

https://www.modb.pro/db/174527

这里指的是消费者的分区分配策略。即「确定哪个 Partition 由哪个 Consumer 来消费的问题」。

消费者分区分配策略有四种：

```
第一种分区策略：RoundRobinAssignor

第二种分区策略：RangeAssignor

第三种分区策略：StickyAssignor

第四种分区策略：CooperativeStickyAssignor（kafka2.4.0 版本引入的策略）
```

### 01 RangeAssignor

它是**以单个 Topic 为一个维度**来计算分配的, 只负责将每一个 Topic 的分区尽可能均衡的分配给消费者:

1. 消费组里面所有消费者 consumer 按照字母排序，给 Topic 的分区按照分区号排序。
2. 先计算每个分区最少平均分配多少个分区数, 然后余下的逐个分。

即：首先会计算每个 consumer 可以消费的分区个数，然后按照顺序将指定个数范围的分区分配给各个 consumer。

**计算过程：**对于每一个 Topic，将该 Topic 的所有可用 partitions 和订阅该 Topic 的所有 consumers 展开(字典排序)，然后将 partitions 数量除以 consumers 数量，算数除的结果分别分配给订阅该 Topic 的 consumers，算数除的余数分配给前一个或者前几个 consumers。

**缺点**：

**Range 针对单个 Topic 的情况下显得比较均衡，但是假如 Topic 很多的话, consumer 排序靠前的可能会比 consumer 排序靠后的负载多很多。**

### 02 RoundRobinAssignor

**RoundRobin 是针对所有 topic 分区**。它是采用轮询分区策略，**是把所有的 partition、所有的 consumer 列举出来** 进行排序，最后再通过**轮询**分配 partition 给每个消费者（**如果该消费者没有订阅该主题则跳到下一个消费者**）

### 03 StickyAssignor

**两个主要目的**是:

1、分区的分配要尽可能的均匀，分配给消费者者的主题分区数最多相差一个；

2、分区的分配尽可能的与上次分配的保持相同。

当两者发生冲突时，第一个目标优先于第二个目标。

**类似于 RoundRobin，区别在于在重分配时，StickyAssignor 会尽量保存之前的分区分配方案，使得分区重分配具备黏性，减少不必要的分区移动。**

### 04 CooperativeStickyAssignor 策略

上述三种分区分配策略均是基于 eager 协议，**Kafka2.4.0** 开始引入 **CooperativeStickyAssignor 策略。**

CooperativeStickyAssignor 与之前的 StickyAssignor 虽然都是维持原来的分区分配方案，**最大的区别**是：

```
StickyAssignor 仍然是基于 eager 协议，分区重分配时候，都需要 consumers 先放弃当前持有的分区，重新加入 consumer group；
而 CooperativeStickyAssignor 基于 cooperative 协议，该协议将原来的一次全局分区重平衡，改成多次小规模分区重平衡。渐进式的重平衡。
```

两类分区策略详细介绍如下：

```
EAGER ：重新平衡协议要求消费者在参与重新平衡事件之前始终撤销其拥有的所有分区。因此，它允许完全改组分配。

COOPERATIVE：协议允许消费者在参与再平衡事件之前保留其当前拥有的分区。分配者不应该立即重新分配任何拥有的分区，而是可以指示消费者需要撤销分区，以便可以在下一次重新平衡事件中将被撤销的分区重新分配给其他消费者。
```

可以看到：

- EAGER 协议中各个消费者参与重平衡操作之前都需要先放弃自己拥有的全部分区；

- COOPERATIVE 协议中 将重平衡分成了两次，以加入一个消费者为例：
  - 第一次 各个消费者会将各自已经在消费的分区传给组协调器，最终计算出需要撤销的分区，并把最终方案传给之前存活的消费者重新开始消费；
  - 第二次 将撤销完成的分区分配给新加入的分区。

**COOPERATIVE 协议将一次全局重平衡，改成每次小规模重平衡，直至最终收敛平衡的过程。**

### 1.5.5.18 谈谈 Kafka 消费者重平衡机制（Rebalance）

分区再平衡部分

就 Kafka 消费端而言，有一个难以避免的问题就是消费者的重平衡即 Rebalance。Rebalance 是让一个消费组的所有消费者就如何消费订阅 topic 的所有分区达成共识的过程，在 Rebalance 过程中，所有 Consumer 实例都会停止消费，等待 Rebalance 的完成。因为要停止消费等待重平衡完成，因此 Rebalance 会严重影响消费端的 TPS，是应当尽量避免的。

**触发再平衡的时机**：

Rebalance 的触发条件有 3 个。

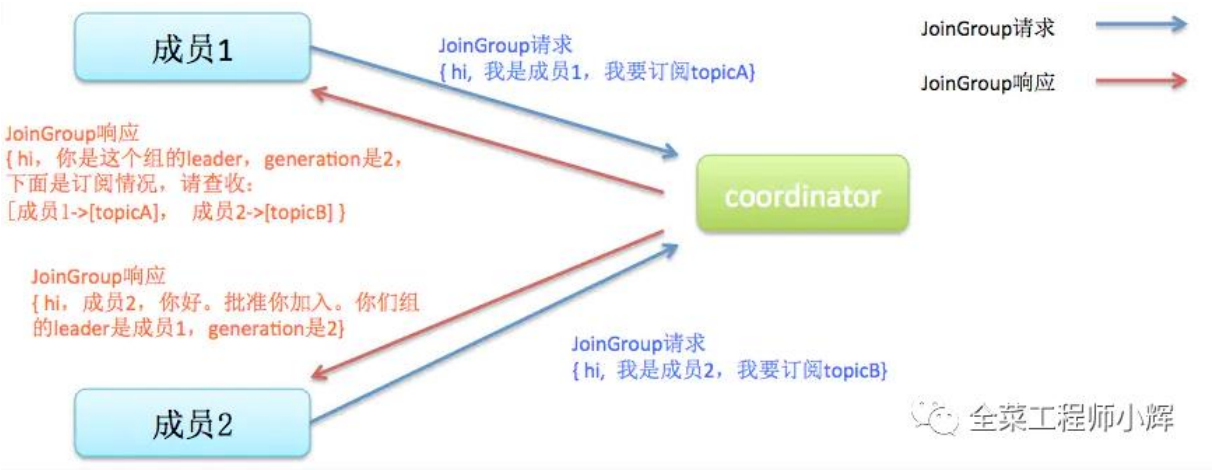
1. 消费组成员个数发生变化。例如有新的 Consumer 实例加入或离开该消费组。
2. 订阅的 Topic 个数发生变化。（比如按照正则订阅的某类 topic）
3. 订阅 Topic 的分区数发生变化。

Rebalance 发生时，Group 下所有 Consumer 实例都会协调在一起共同参与，kafka 能够保证尽量达到最公平的分配。但是 Rebalance 过程对 consumer group 会造成比较严重的影响。**在 Rebalance 的过程中 consumer group 下的所有消费者实例都会停止工作**，等待 Rebalance 过程完成。

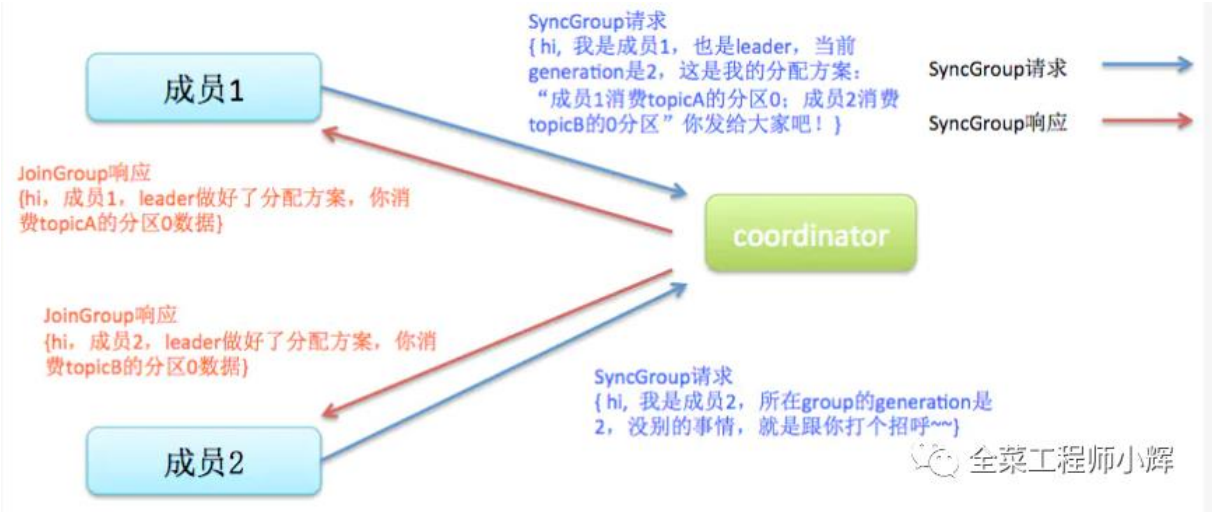
**再平衡过程**：

Rebalance 过程分为两步：Join 和 Sync。在开始两步之前，协调者会要求组内成员必须在这段缓冲时间内快速地提交自己的位移信息，然后再开启正常的 JoinGroup/SyncGroup 请求发送。

1. Join。所有成员都向 Group Coordinator 发送 JoinGroup 请求，请求加入消费组。一旦所有成员都发送了 JoinGroup 请求，Coordinator 会从中选择一个 Consumer 担任 leader 的角色，并把组成员信息以及订阅信息发给 leader——注意 leader 和 coordinator 不是一个概念。leader 负责消费分配方案的制定。




2. Sync。这一步 leader 开始分配消费方案，即哪个 consumer 负责消费哪些 topic 的哪些 partition。一旦完成分配，leader 会将这个方案封装进 SyncGroup 请求中发给 coordinator，非 leader 也会发 SyncGroup 请求，只是内容为空。coordinator 接收到分配方案之后会把方案塞进 SyncGroup 的 response 中发给各个 consumer。这样组内的所有成员就都知道自己应该消费哪些分区了。




**避免不必要的 Rebalance**

前面说过 Rebalance 发生的时机有三个，后两个时机是可以人为避免的。发生 Rebalance 最常见的原因是消费组成员个数发生变化。

这其中消费者成员正常的添加和停掉导致 Rebalance，也是无法避免。但是在某些情况下，Consumer 实例会被 Coordinator 错误地认为已停止从而被踢出 Group。从而导致 rebalance。

这种情况可以通过 Consumer 端的参数 `session.timeout.ms` 和 `max.poll.interval.ms` 进行配置。

除了这个参数，Consumer 还提供了控制发送心跳请求频率的参数，就是 `heartbeat.interval.ms`。这个值设置得越小，Consumer 实例发送心跳请求的频率就越高。频繁地发送心跳请求会额外消耗带宽资源，但好处是能够更快地知道是否开启 Rebalance，因为 Coordinator 通知各个 Consumer 实例是否开启 Rebalance 就是将 REBALANCE_NEEDED 标志封装进心跳请求的响应体中。

总之，要为业务处理逻辑留下充足的时间使 Consumer 不会因为处理这些消息的时间太长而引发 Rebalance，但也不能时间设置过长导致 Consumer 宕机但迟迟没有被踢出 Group。

### 1.5.5.19 谈谈 Kafka 线上大量消息积压应该如何处理

#### 01 优化性能来避免消息积压

对于 Kafka 性能的优化，主要体现在生产者和消费者两部分业务逻辑中。

发送端性能优化

消费端性能优化

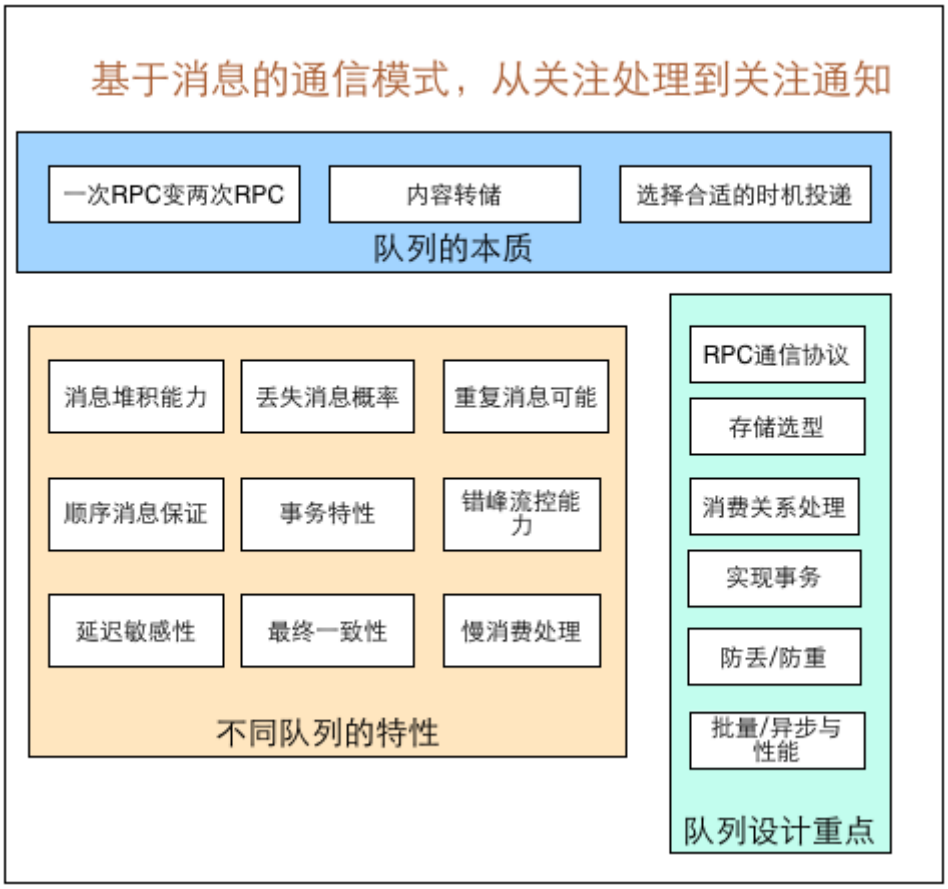
## 二、 Pulsar 相关

https://mp.weixin.qq.com/s/-jeUzx9eX7l9XLR_95frPQ?scene=25#wechat_redirect

https://alexstocks.github.io/html/pulsar.html

https://www.cnblogs.com/labuladong/p/17023978.html

## 2.0 消息队列学习角度

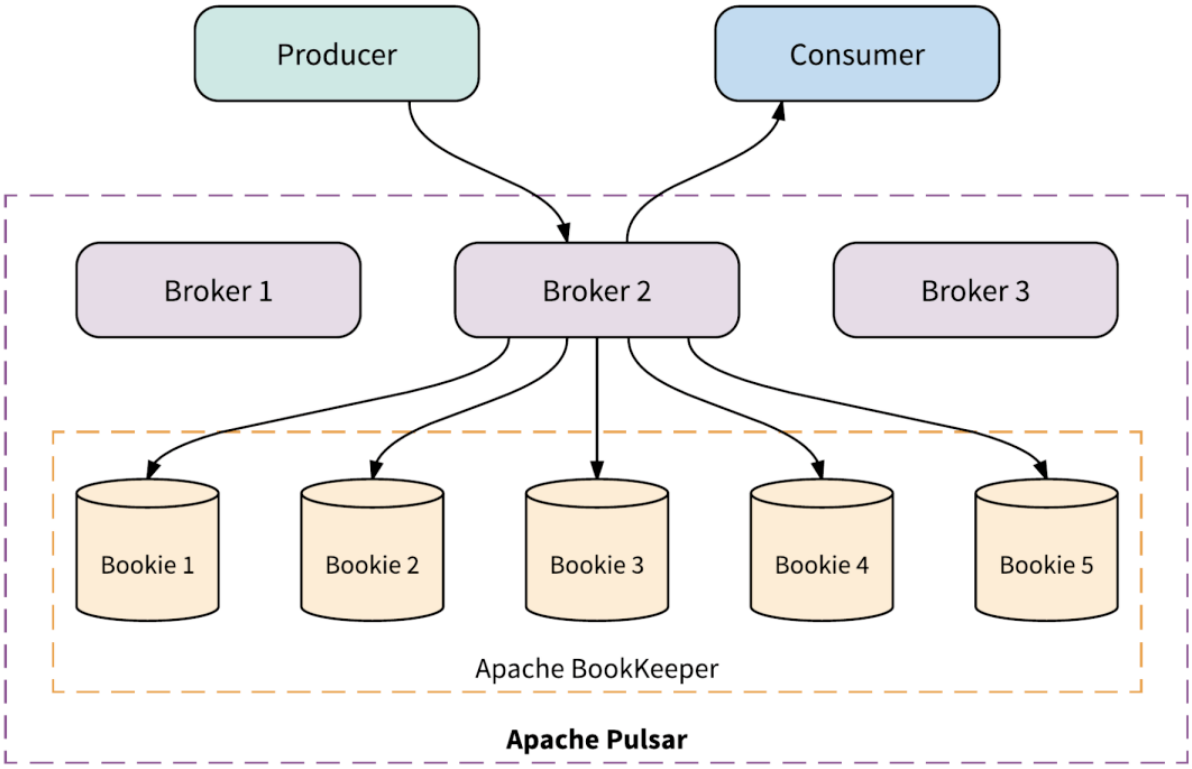



类比一下 kafka，先从整体架构入手，

## 2.1 Pulsar 架构

https://juejin.cn/post/7213983057899946040

### 2.1.1 存储与计算分离




Pulsar 架构中主要有三个组件：ZooKeeper、Pulsar Broker 和 Apache BookKeeper Bookie。与 Kafka 一样，ZooKeeper 提供服务发现、领导者选举和元数据存储。与 Kafka 不同的是，**Pulsar 通过 Broker 和 BookKeeper Bookie 组件分离了消息处理计算与消息存储功能。**

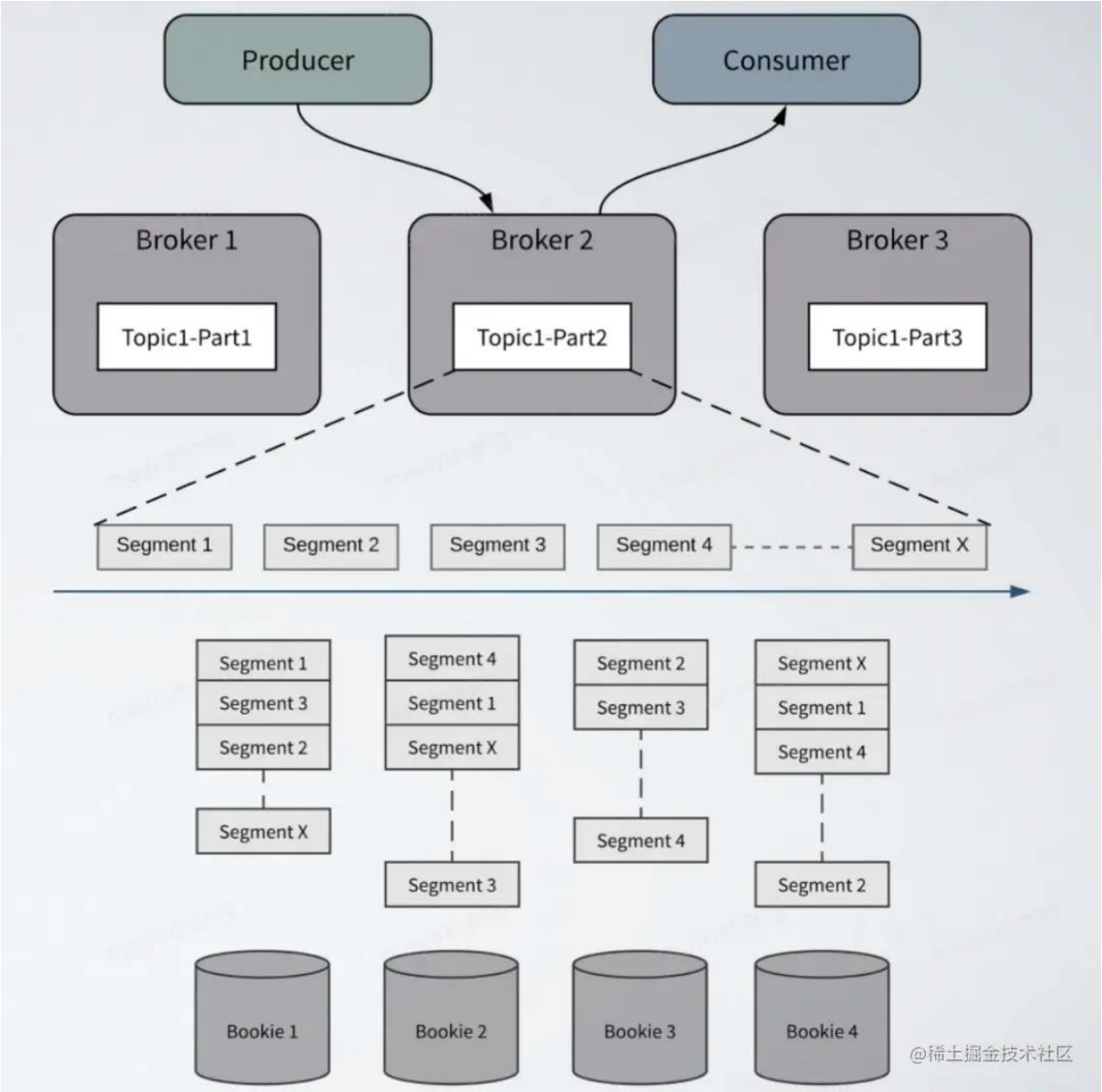
- **Broker：**无状态服务层，负责接收和传递消息，集群负载均衡等工作，Broker 不会持久化保存元数据，因此可以快速的上、下线。
- **Apache BookKeeper：**有状态持久层，由一组名为 Bookie 的存储节点组成，持久化地存储消息。
- **Producer：**数据生产者，负责发布数据到 Topic。
- **Consumer：**数据消费者，负责从 Topic 订阅数据。
- **Zookeeper**：元数据存储。

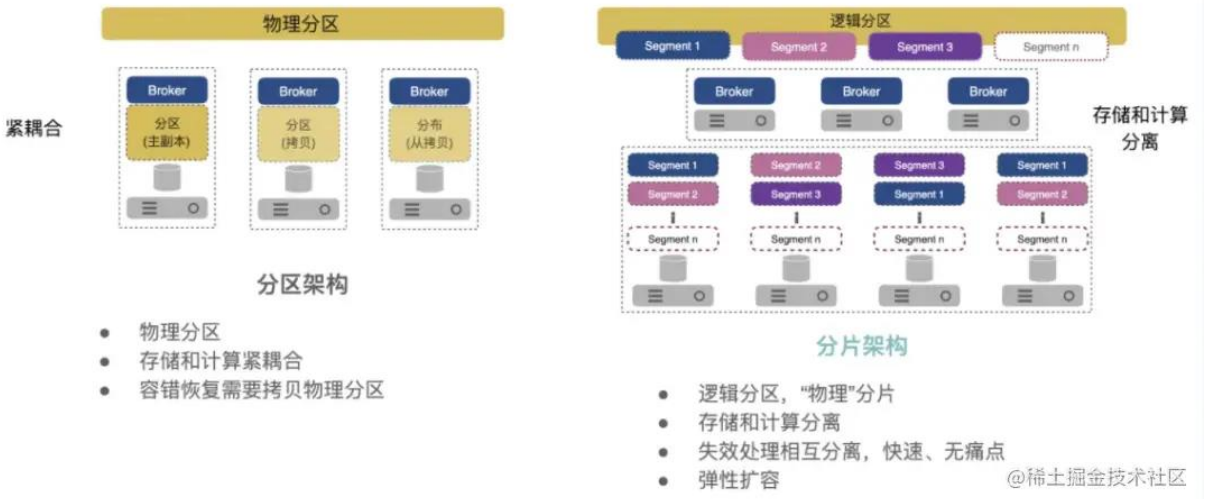
Pulsar 在架构设计上采用了计算与存储分离的模式，Pub/Sub 相关的计算逻辑在 Broker 上完成，数据存储在 BookKeeper 的 Bookie 节点上。

### 2.1.2 分片存储

Apache Pulsar 中的每个 Topic 分区本质上都是存储在 Apache BookKeeper 中的分布式日志。

Topic 可以有多个分区，分区数据持久化时，分区是逻辑上的概念，实际存储的单位是分片（Segment），一个分区 Topic1-Part2 的数据由多个 Segment 组成， 每个 Segment 作为 Apache BookKeeper 中的一个 Ledger，均匀分布并存储在 Apache BookKeeper 群集中的多个 Bookie 节点中， 每个 Segment 具有 3 个副本。






## 2.1.3 架构设计的优势

Apache Pulsar 计算与存储分离的架构，以及分片存储的设计为 Apache Pulsar 带来了相比于传统基于分区存储 MQ 的一些优势：

- Broker 和 Bookie 相互独立，方便实现独立的扩展以及独立的容错。
- Broker 无状态，便于快速上、下线，更加适合于云原生场景。
- 分区存储不受限于单个节点存储容量。
- 分区数据分布均匀。

可扩展性

由于 Broker 和 Bookie 是分开的，因此可以独立地扩展存储层和服务层。

**Broker 扩展**

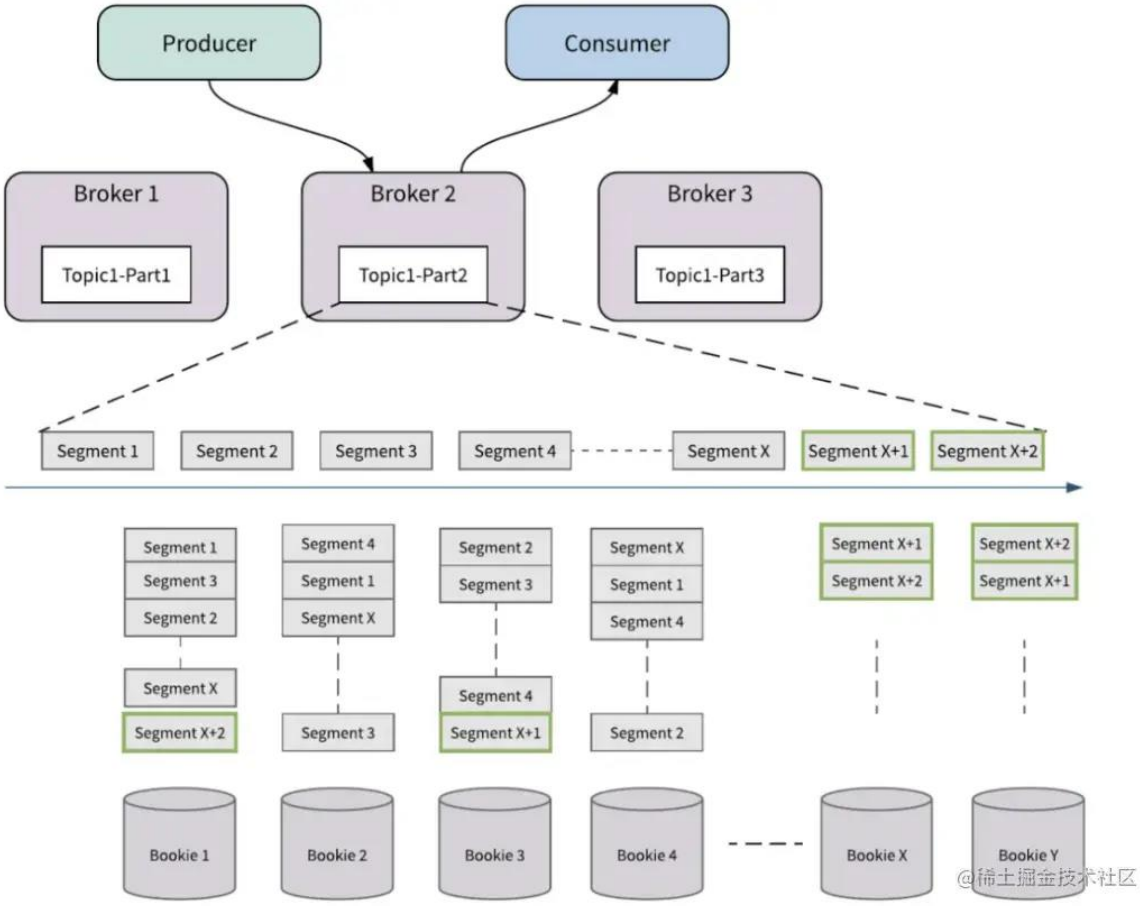
总结：发生在需要支持更多的消费者或生产者时；Broker 扩展的过程中，一些分区的所有权可以转移到新的 Broker 节点，不涉及数据迁移，而 Kafka 在扩展的过程中，增加 Broker 节点需要迁移分区的数据，如果分区数量很多耗时就很久！

**计算层的扩展通过增加 Broker 来实现**。在 Pulsar 中 Broker 是无状态的，可以通过增加节点的方式实现快速扩容。当需要支持更多的消费者或生产者时，可以简单地添加更多的 Broker 节点来满足业务需求。Pulsar 支持自动的分区负载均衡，在 Broker 节点的资源使用率达到阈值时，会将负载迁移到负载较低的 Broker 节点，这个过程中分区也将在多个 Broker 节点中做平衡迁移，一些分区的所有权会转移到新的 Broker 节点。不涉及到数据的迁移。

**Bookie 扩展**

总结：发生在需要扩展存储节点，增加存储容量时；Bookie 扩展的过程中，不会涉及到旧 Segment 的迁移，而是在写入新 Segment 时优先写入新的 Bookie

存储层的扩容，通过增加 Bookie 节点来实现。




起始状态有四个存储节点，Bookie1, Bookie2, Bookie3, Bookie4，以 Topic1-Part2 为例，当这个分区的最新的存储分片是 SegmentX 时，对存储层扩容，添加了新的 Bookie 节点，BookieX,BookieY，那么在存储分片滚动之后，新生成的存储分片，SegmentX+1,SegmentX+2，会优先选择新的 Bookie 节点（BookieX,BookieY）来保存数据。

容错性

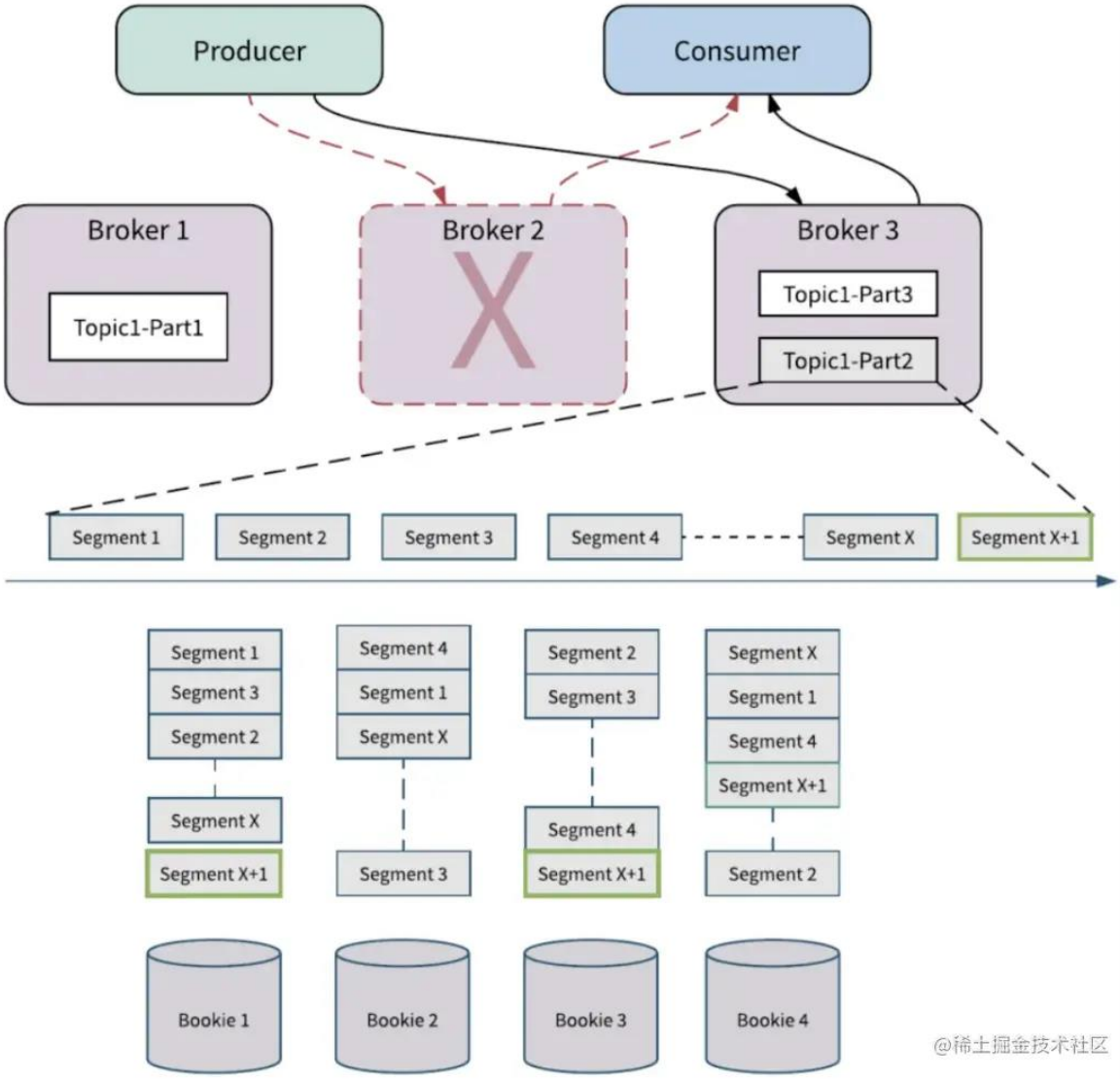
得益于计算与存储分离以及分片存储的设计，Pulsar 可以实现独立、灵活的容错。

**Broker 容错**

**总结：Broker 挂掉时，所拥有的分区会转移到其他 Broker 上，也就是只迁移分区的所有权到正常的 Broker，但是分区对应的 Segment 不会迁移，不会影响分区的可用性。**

当 Broker 挂掉时，以下图为例，假设 Broker2 存储分片为 SegmentX，此时 Broker2 挂掉了，此时如果生产者和消费者向其他的 Broker 发起请求，则会触发分区的所有权转移，即将挂掉

的 Broker2 拥有的分区 Topic1-Part2 的所有权转移到其他的 Broker（Broker3），新 Broker 接管分区所有权时，不需要复制分区的数据，新的分区 Owner（Broker3）会产生一个新的分片 SegmentX+1，如果有新数据到来，会存储到新的分片 SegmentX+1 上，不会影响分区可用性。




**Bookie 容错**

总结：Bookie 挂掉时，BookKeeper Auditor 会在后台修复这个 Bookie 上的 Segment，这个过程对 Broker 和应用透明，Broker 会产生新的 Segment 来处理写入请求，不会影响分区的可用性。具体做法是会从其他 Bookie 读取对应的 Segment 副本，并写入另外的 Bookie 中，直到达到冗余数量，

如下图所示，假设 Bookie2 上的 Segment4 损坏。Apache BookKeeper Auditor 会检测到这个错误并进行复制修复。Apache BookKeeper 中的副本修复是 Segment 级别的多对多快速修复，BookKeeper 可以从 Bookie 3 和 Bookie 4 读取 Segment 4 中的消息，并在 Bookie 1 处修复 Segment 4。如果是 Bookie 节点故障，这个 Bookie 节点上所有的 Segment 会按照上述方式复制到其他的 Bookie 节点。所有的副本修复都在后台进行，对 Broker 和应用透明，Broker 会

产生新的 Segment 来处理写入请求，不会影响分区的可用性。

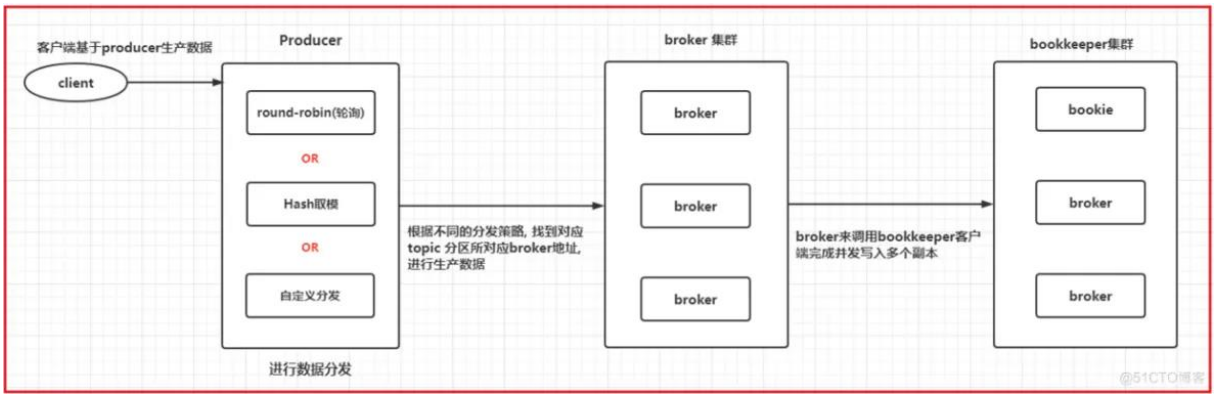
总结：分片存储解决了分区容量受单节点存储空间限制的问题，当容量不够时，可以通过扩容 Bookie 节点的方式支撑更多的分区数据，也解决了分区数据倾斜的问题，数据可以均匀的分配在 Bookie 节点上。Broker 和 Bookie 灵活的容错以及无缝的扩容能力让 Apache Pulsar 具备非常高的可用性。

## 2.2 两次 RPC，一个队列

### 2.2.1 生产者到 broker

## 2.2 BookKeeper

## 2.3 Pulsar 消息发送流程




## 2.4 Pulsar 存储模型

## 2.5 如何保证消息顺序性

顺序性表示 保证消息是按顺序发送，按顺序消费，一个接着一个

## 2.6 如何保证消息幂等性

## 2.7 如何保证消息不丢失、不重复

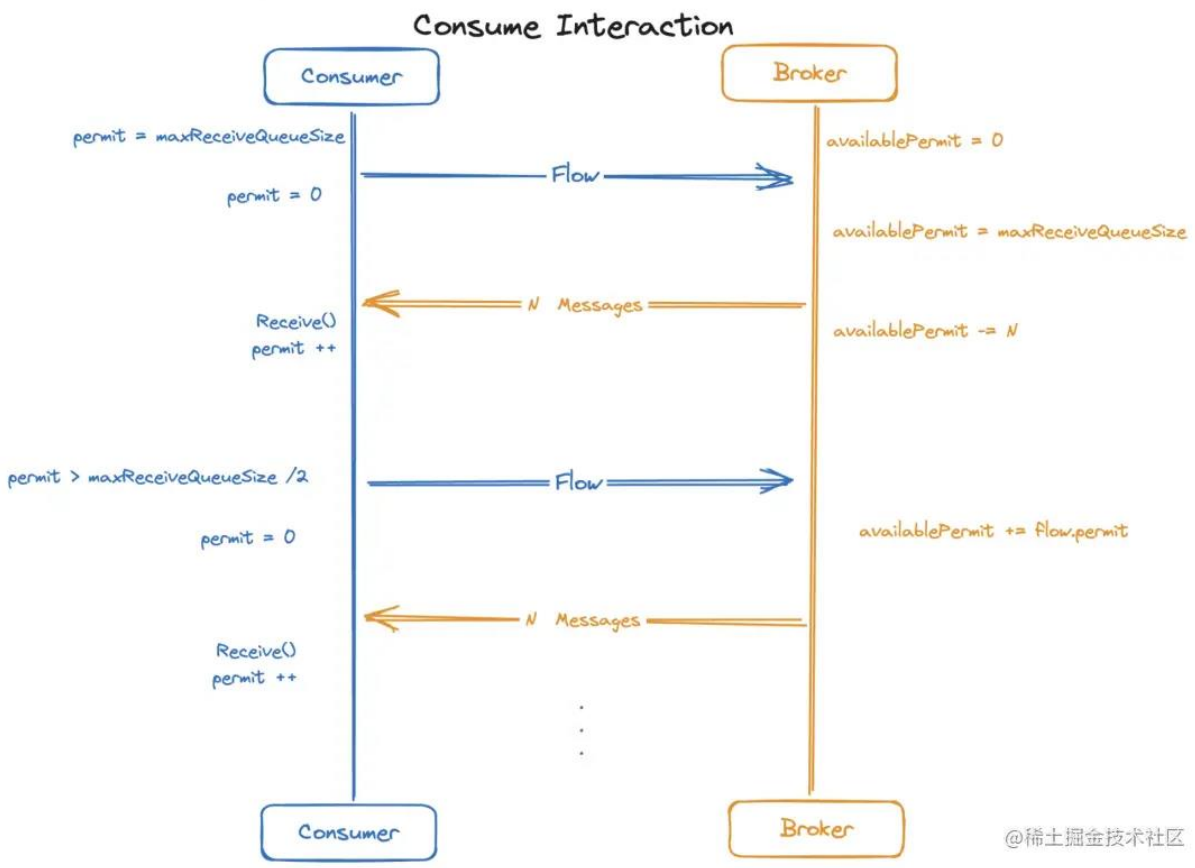
## 2.8 消息重推的几种方式

https://juejin.cn/post/7260024054065446970

### 2.8.1 消息获取（拉取/推送）机制

Broker 会根据最大缓存队列数量（配置：ReceiveQueue）+消费者当前可接收的消息数量（Permit）来推送消息给消费者。

详细交互流程如下：




- 消费者创建后以 最大缓存队列数量 为 Permit 的值。
- 消费者向 Broker 发送 FLOW 请求，携带 Permit 信息，同时消费者处的 Permit 减少到 0，Broker 以收到的 Permit 为依据向消费者推送消息，并将自身保存的 Permit 减少到 0
- 消费者收到消息后，将消息放到缓存队列中，用户调用 Receive 方法获取消息后消费者将 Permit+1
- 当消费者处的 Permit > MaxReceiveQueueSize / 2 时，消费者会再次发起 FLOW 请求，并携带新的 Permit 值。

### 2.8.2 SDK 统一的超时重推

超过一定时间，如果消息没有 Ack 就重新推送。

通过配置：ackTimeOut 来确认

- 消费者可以设置 AckTimeout（默认关闭）
- 设置 AckTimeout 之后，消费者会构建 UnAckedMessageTracker，追踪用户 Receive()的所有消息
- 用户 Ack 消息时，会从 UnAckedMessageTracker 删除

- 没有 Ack 的消息会定时轮询，超过 AckTimeout 会触发重推

重推通过 RedeliverUnackMessage 实现，会主动发起 Redeliver 请求，Broker 会根据请求的 MessageId 信息重新推送。

### 2.8.3 用户决策的重推--NACK

通过 NegativeAcksTracker 来管理消息的重推，只会跟踪调用 NegatineAcknowledge()方法的 MessageID，

重推通过 RedeliverUnackMessage 实现

NACK 还可以设置重投递的延迟时间。

### 2.8.4 用户决策的重推--RLQ

## 2.9 如何保证消息至少被消费一次

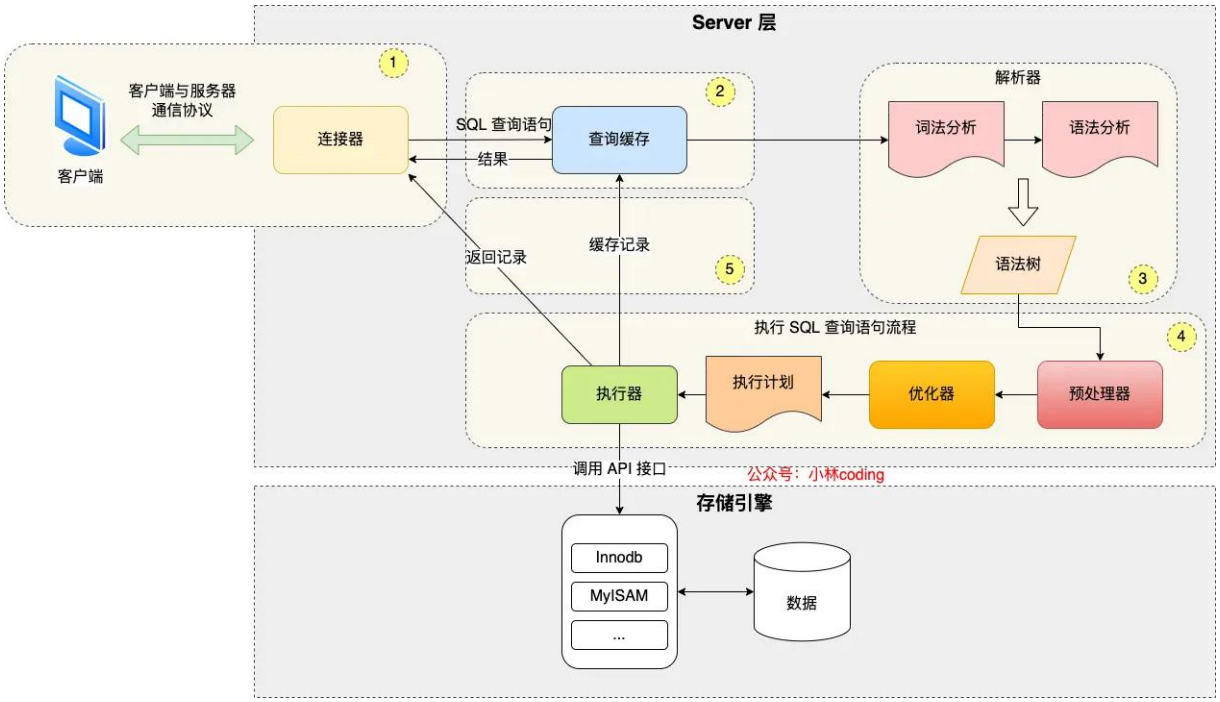
## 2.10 如何确保消息有且只能被消费一次

## 2.11 Pulsar 消费语义

# 三、 MySQL 相关

## 3.0 MySQL 数据结构

## 3.1 MySQL 执行流程




- Server 层：建立连接、分析和执行 SQL。包括连接器、查询缓存、分析器、优化器、执行器等，涵盖 MySQL 核心服务功能以及所有内置函数（如日期、时间、数学和加密函数等），还包括存储过程、触发器、视图等。
- 存储引擎层：数据的存储和提取

一条 SQL 查询语句的执行过程：

1.第一步，首先连接到数据库上，也就是先由连接器处理。连接器负责跟客户端建立连接、获取权限、维持和管理连接。

数据库中的长连接和短连接：

长连接：连接成功后，如果客户端持续有请求，则一直使用同一个连接。

短连接：每次执行完很少的几次查询就断开连接，下次查询再重新建立一个。

使用长连接后，为什么 MySQL 占用内存涨的特别快？

MySQL 在执行过程中临时使用的内存是管理在连接对象里面的，这些资源会在连接断开的时候才会释放，所以如果长连接累积下来，可能导致内存占用太大。

如何解决上述问题呢？

1.定期断开长连接。使用一段时间，或者程序里面判断执行过一个占用内存的大查询后，断开连接，之后要查询再重连。

2. 在 MySQL5.7 或更新版本，可以在每次执行一个比较大的操作后，通过执行 mysql_reset_connection 来重新初始化连接资源。这个过程不需要重连或者重新做权限验证，但是会将连接恢复到刚刚创建完时的状态。

2.第二步，**查询缓存**，连接建立完成后，执行逻辑就会来到第二步：**查询缓存**。

MySQL8.0 之后已经将查询缓存整个模块删除了，原因如下：

查询缓存的失效非常频繁，只要有对表的更新，这个表上的所有查询缓存都会被清空，对于更新压力大的数据库来说，查询缓存的命中率会非常低。

3.第三步，**对 SQL 语句进行解析**，也就是 MySQL 需要知道你要做什么，此处是由分析器完成的。

词法分析：分析 SQL 语句，识别出字符串分别是什么，代表什么

语法分析：根据语法规则判断 SQL 语句是否满足 MySQL 语法。

4. 第四步，解析完 SQL 后，就需要进入执行 SQL 流程，主要分为以下三个阶段：
   - 预处理阶段
   - 优化阶段
   - 执行阶段

预处理阶段：

- 检查表或者字段是否存在
- 将 select *中的*扩展为表上所有列

优化阶段，主要负责将 SQL 查询语句的执行方案确定下来：

- 表中**有多个索引时，决定使用哪个索引**，或者一个语句中存在多表关联时（join），**决**

**定各个表的连接顺序。**

执行阶段，由执行器完成，和存储引擎交互，交互以记录为单位，分为以下三种情况的交互方式：

- 执行前还需要判断是否对要操作的表有相应的权限
- 实际查询有以下三种情况：
  - 索引查询：
    - 调用存储引擎的接口，通过索引定位符合条件的第一条记录，结束后将查询结果返回给客户端，继续循环读下一条符合条件的记录，直到结束。
  - 全表扫描：
    - 调用存储引擎接口，读取表中第一条记录，判断是否满足，满足则直接发送给客户端（读一条返回一条）；循环读出所有的记录
  - **索引下推，减少回表次数，所谓索引下推就是针对联合索引的情况，把理应 Server 做的判断逻辑推到了存储引擎**：
    - 假设：如果联合索引设置了 age 和 reward，查询条件是 age>20 and reward=1000。
    - 1. Server 定位到第一条满足索引查询的记录
    - 2. 存储引擎定位到联合索引后，先不执行回表操作，而是判断一下该索引包含的列条件是否满足查询要求，如果不成立则直接跳过该二级索引，如果**成立才执行回表操作**，将记录返回给 server 层。（如果没有索引下推，则存储引擎不会判断联合索引其他列是否满足条件，而是直接回表查询并返回给 Server，由 Server 判断所有的查询条件是否成立）
    - 3. Server 判断其他查询条件是否成立，成立则发送给客户端，否则跳过进行下一轮查询，直到查询完成。

## 3.2 MySQL 底层数据存储方式

### 01 MySQL 数据存放在哪个文件？

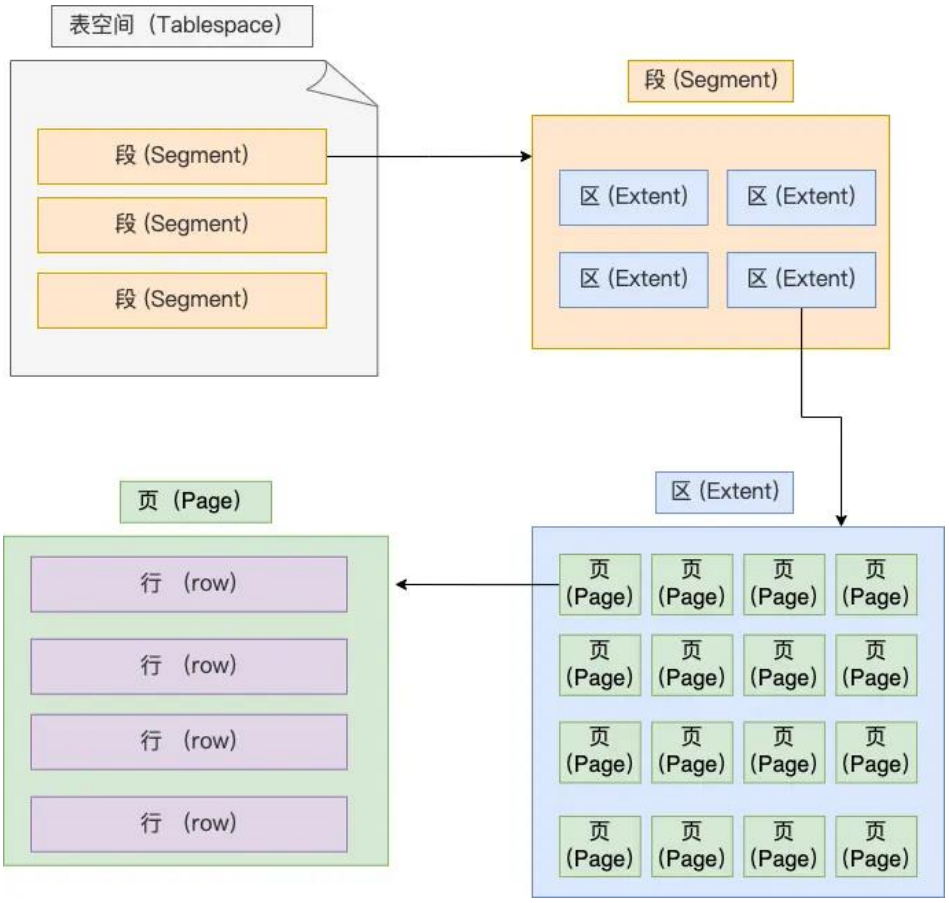
```
[root@xiaolin ~]#ls /var/lib/mysql/my_test
db.opt
t_order.frm
t_order.ibd
```

- *.opt：数据库的默认字符集和字符校验规则

- *.frm：表的元数据信息，主要包含表结构定义
- *.ibd：存放表数据

可以看到表空间的数据全部都存储在*.ibd 表数据文件，那么在此文件内部是如何组织数据的呢？

**02 表空间的文件结构**




可以看到，一个 ibd 文件从上到下可以分为段---》区----》页----》行，下面我们从下往上介绍各个概念：

- **行（row）**：数据表中记录按行存储，每行记录按照不同的格式有不同的存储结构。
- **页（page）**：包括数据页、undo 日志页、溢出页等；Innodb 以【页】为单位来读写的，读一条记录时是从磁盘中取一个【页】到内存，写入时也是以【页】为单位整体写入。
- **区（extent）**：数据量大时，为某索引分配空间会按照区为单位分配，每个区大小为 1MB，对于 16KB 的页来说，连续 64 个页会被划为一个区。使得链表中相邻页的物理位置也相邻，就可以使用顺序 I/O。
- **段（segment）**：表空间由各个段组成，段一般分为数据段、索引段和回滚段
  - 索引段：存放 B+树的非叶子节点的区的集合；

- 数据段：存放 B+树的叶子节点的区的集合；
- 回滚段：存放回滚数据的区的集合。

了解完表空间文件的组成，继续看在 InnoDB 中数据行是如何组织的

**03 InnoDB 的行格式**

InnoDB 提供了四种行格式，分别为 Redundant（5.0 以后已废弃）、Compact（5.7 版本后默认此格式）、Dynamic 和 Compressed（这两个行格式和 Compact 比较像）

Compact 行格式

一条完整的记录可以分为【记录的额外信息】和【记录的真实数据】两个部分

**记录的额外信息**

1. 变长字段长度列表
   - **存储变长字段的真实数据占用的字节数，按列的顺序逆序存放。**
   - 只出现在数据表有变长字段的时候
2. NULL 值列表
   - 只出现在数据表中有允许为 null 的时候
   - 每一列对应一个二进制位，二进制位按照列的顺序逆序排列。二进制的值为 1 时，该列的值为 null，二进制位的值为 0 时，该列的值不为 null
   - NULL 值列表必须用整数个字节的位表示，不足时要在高位补；超过 8 个字段即用两个字节表示，以此类推
3. 记录头信息
   - delete_mask：标识数据是否被删除；删除数据时并不会真正删除记录，而是将记录的 delete_mask 标记为 1。
   - next_record：下一条记录的位置；记录和记录之间通过链表组织，指向的是下一条记录的【记录头信息】和【真实数据】之间的位置，好处是向左读就是记录头信息，向右读

就是真实数据。

- record_type：当前记录的类型，0 表示普通记录，1 表示 B+树非叶子节点记录，2 表示最小记录，3 表示最大记录

**记录的真实数据**

除了列值，还有三个隐藏字段，后两个用于 MVCC：

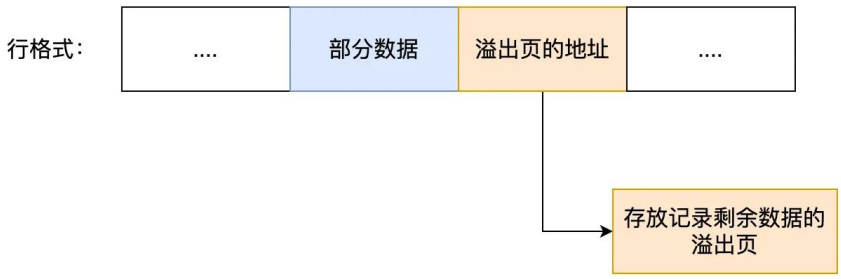
- **row_id：**没有指定主键或者唯一约束列才有这个字段
- **trx_id（必需）：**事务 id，表示这个数据由哪个事务生成
- **roll_pointer（必需）：**本记录上一个版本的指针。

**04 行溢出后，MySQL 是怎么处理的？**

如果一个数据页存不了一条记录，就会**发生行溢出，多的数据会存到另外的【溢出页】中**

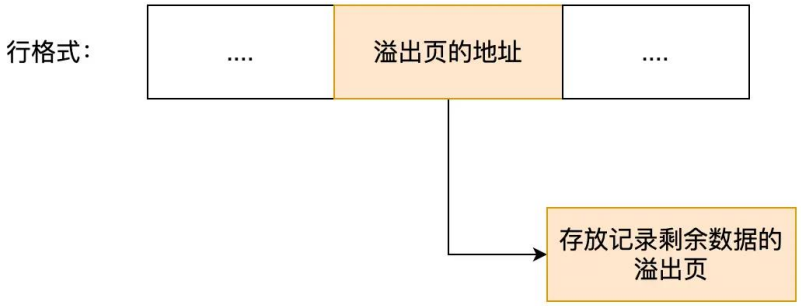
**Compact**

对于 Compact 行格式，记录的真实数据处只会保存该列一部分数据，剩余部分放在【溢出页】中，然后真实数据处用 20 字节存储指向溢出页的地址。



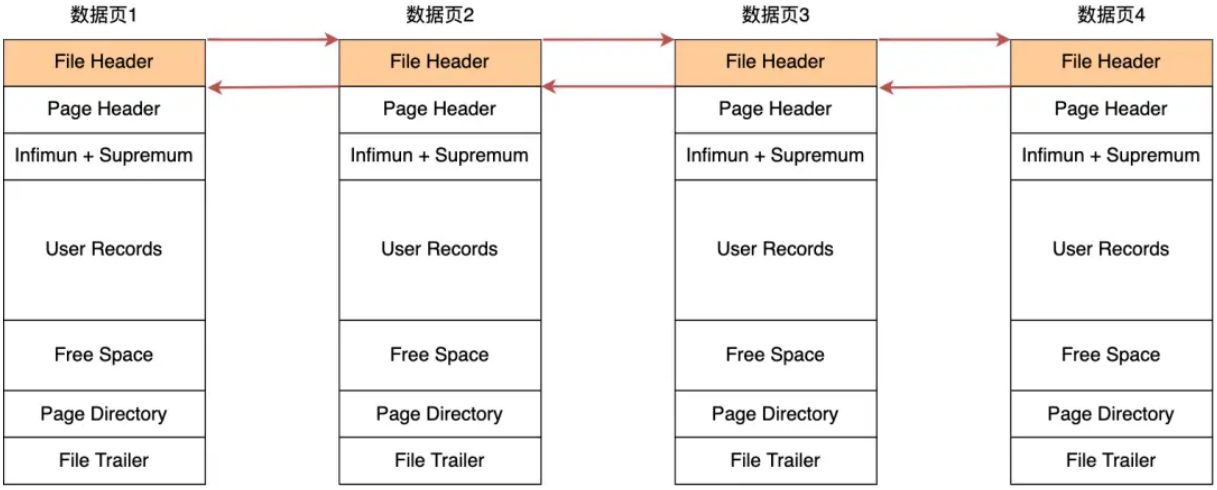

Compressed 和 Dynamic

记录的真实数据处不会存储该列一部分数据，只会用 20 个字节的指针指向溢出页，实际的数据存储在溢出页中。



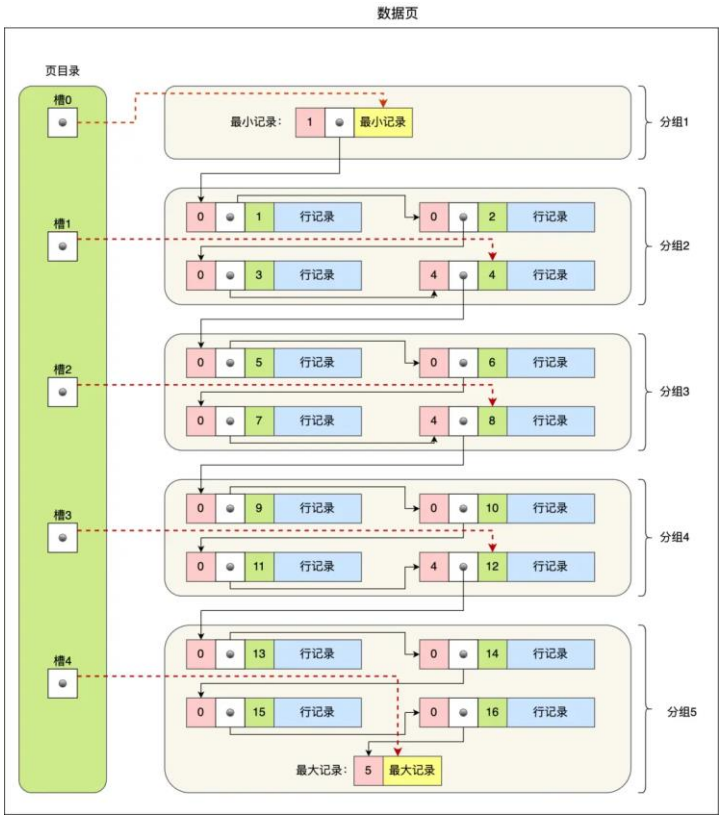

## 3.3 MySQL 索引

### 01 InnoDB 中数据页的格式




**数据页之间采用双向链表**连接，数据主要存储在 User Records

- **数据页中记录**按照【主键】顺序组成**单向链表**
- **数据页中有一个页目录**，起到记录索引的作用

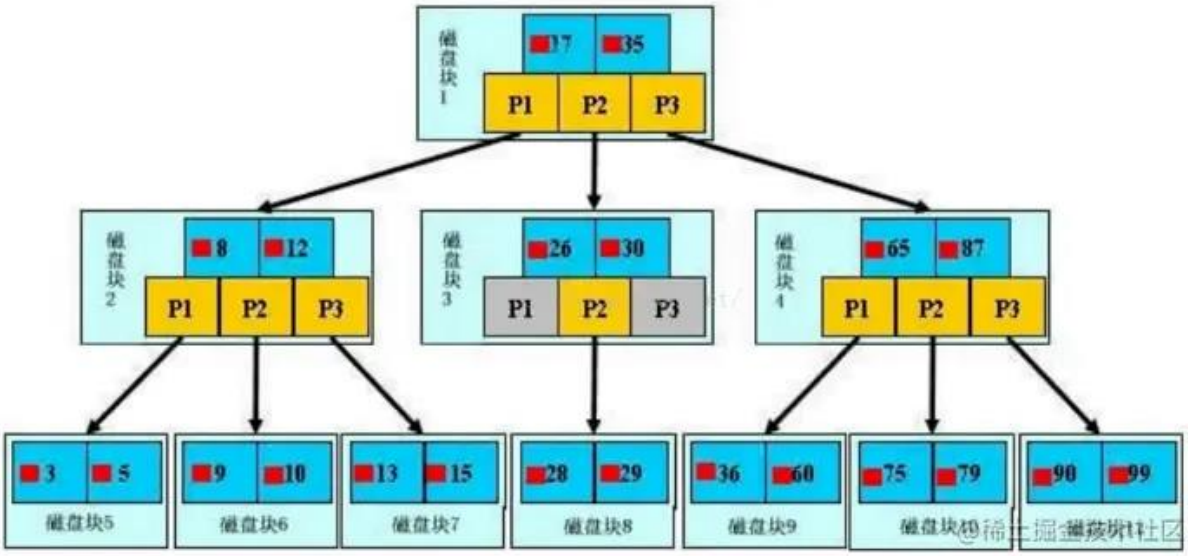


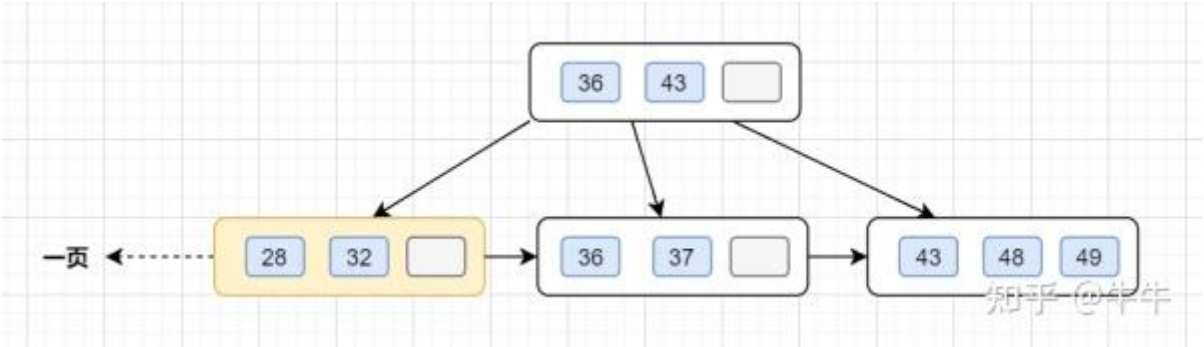

页目录不包含【已删除】的记录

- 页目录由多个槽组成，相当于分组记录的索引，每个槽相当于指针指向了不同组的最后一个记录。
- 通过二分查找定位到记录在哪个槽里，再遍历槽内的所有记录，找到对应的记录。

### 02 B+树如何查询

InnoDB 采用 **B+树**作为索引，**每个节点都是一个数据页**






- 只有叶子节点存放数据，非叶子节点仅存放目录项（只存储主键值和对应的页号）
- 非叶子节点分层降低每一层的搜索量
- 节点按索引建大小排序，双向链表

**03 为什么使用 B+树作为索引的数据结构**

下面我们讨论二叉搜索树、AVL 树、红黑树、B 树哈希索引为什么不能作为数据库索引的数据结构

DB 这种数据库索引的数据结构，需要考虑两方面的因素：1. 增删改查的时间复杂度要低 2. 磁盘 I/O 操作次数要尽可能少

- 二叉搜索树：查找时间复杂度是 O(logn)，但是极端情况下二叉搜索树会退化成一个链表，树结构不稳定，时间复杂度会降低为 O(n)
- AVL 树和红黑树：都是尽量保持树结构稳定，但是随着插入元素增多，树会变高，意味着磁盘 I/O 次数会增多
- B 树：多叉树，有效降低树的高度，但是每个节点都包含真实数据，每一层存储的索引会变少，树的高度较高，会花费更多的 I/O 操作；同时不方便进行范围查询，需要使用中序遍历
- B+树（**InnoDB 使用的数据结构**）：
    - 非叶子节点仅存放索引，仅叶子节点存储真实数据，相比于 B 树更加矮胖，查询时磁盘 I/O 次数会更少，并且查询效率更加稳定。
    - 所有非叶子节点都是冗余索引，使得 B+树在插入、删除效率更高
    - 叶子节点之间使用了双向链表，有利于范围查询

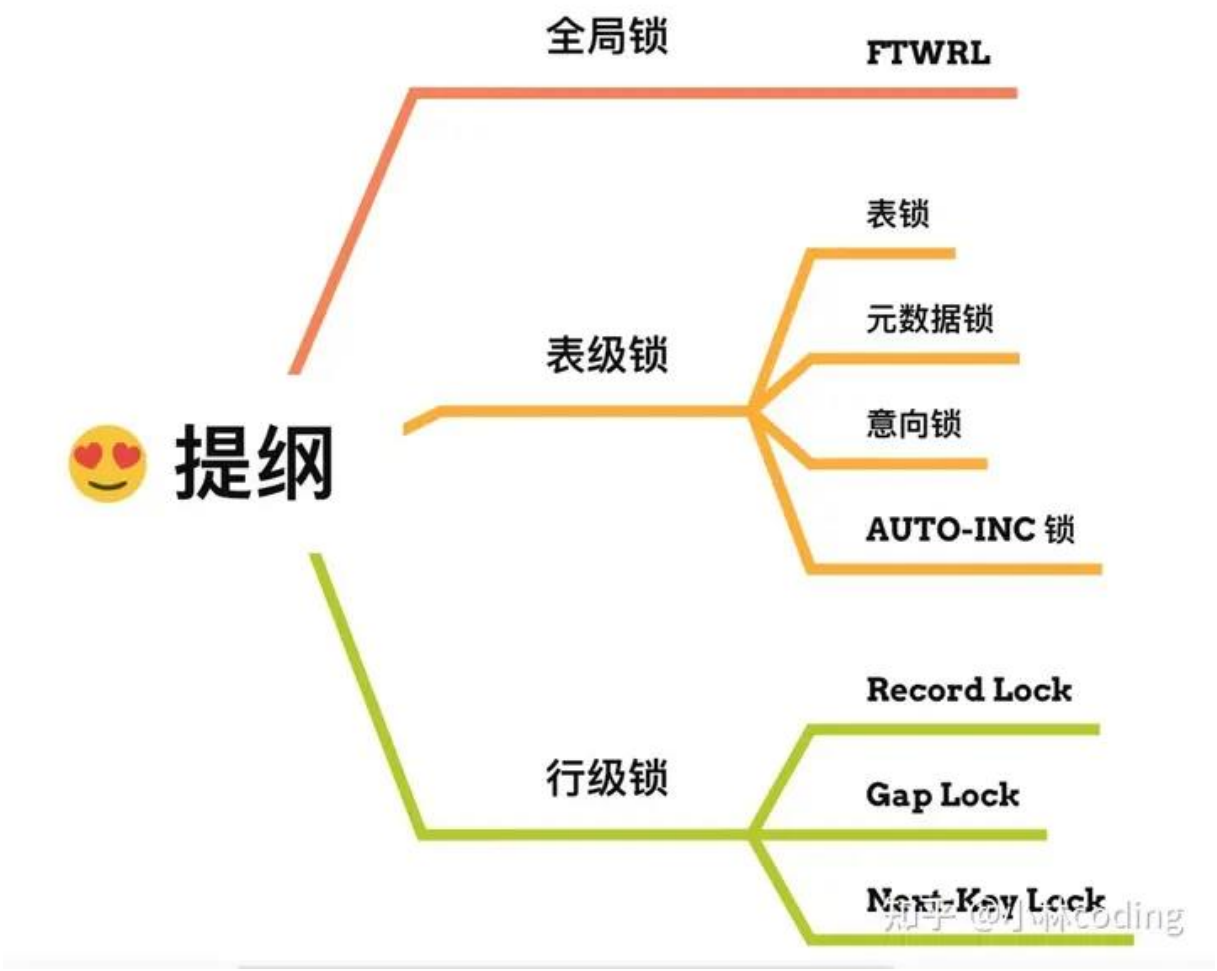
**04 索引的数据结构**

索引的数据结构和数据页差不多，区别就是数据页记录的是真实数据，索引页记录的是对应页的指针。

## 3.4 MySQL 事务

## 3.5 MySQL 锁

### 3.5.1 MySQL 有哪些锁




| 锁粒度 | 锁名称 | 加锁时机 | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

全局锁

- FTWRL(flush table with read lock)

```
flush table with read lock
```

执行上述命令就可以使用全局锁了，整个数据库都处于只读状态

全局锁主要用于做**全库逻辑备份**，这样在备份数据库期间，不会因为数据或表结构更新而出现备份数据和预期不一样。

**缺点**：整个数据库只能读，不能写，会造成业务停滞

备份时更好的解决办法：

mysqldump 时加上 -single-transaction 参数，会在备份前开启事务，只适用于【可重复读隔离级别的事务】的存储引擎

表级锁

表级锁有以下几种：

- 表锁；
- 元数据锁（MDL）；
- 意向锁；
- AUTO-INC 锁；

**表锁**

## 3.6 MySQL 日志

MySQL 有三种日志：undo log（回滚日志）、redo log（重做日志）、binlog（归档日志）

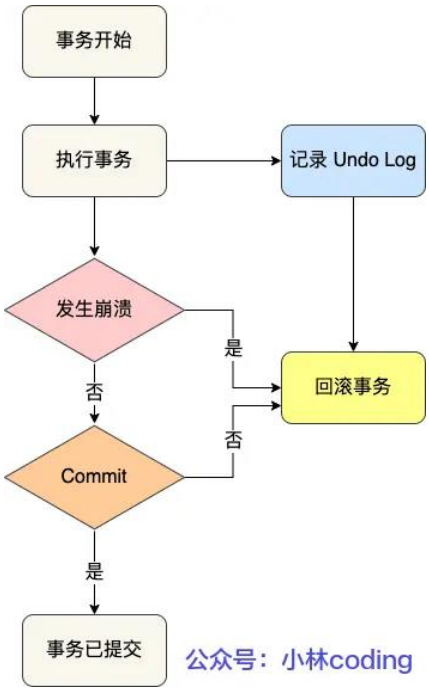
- undo log：InnoDB 生成的日志，实现事务中的**原子性**，主要**用于事务回滚和 MVCC**
- redo log：InnoDB 生成的日志，实现事务中的**持久性**，主要**用于掉电等故障恢复**
- binlog：Server 层生成的日志，主要用于数据备份和主从复制

### 3.6.1 undo log

主要功能：

undo log 是一种用于**撤销回退**的日志。在事务没提交之前，MySQL 会先记录更新（修改、删除、新增）前的数据到 undo log 日志文件里面，**当事务回滚时，可以利用**

**undo log 来进行回滚**。




实现形式：

- trx_id：记录是被哪个事务修改的
- roll_pointer：将 undo log 串成一个链表，这个链表被称为版本链

undo log 两大作用：

- **实现事务回滚，保障事务的原子性**：可以利用 undo log 中的历史数据将数据恢复到事务开始之前的状态。
- **实现 MVCC**：执行快照读（普通 select 语句）的时候，会根据事务中视图的信息，顺着 undo log 版本链找到满足其可见性的记录。

如何持久化到磁盘：

undo 页和数据页一样，**通过 redo log 保证持久化**；

### 3.6.2 redo log

主要功能：

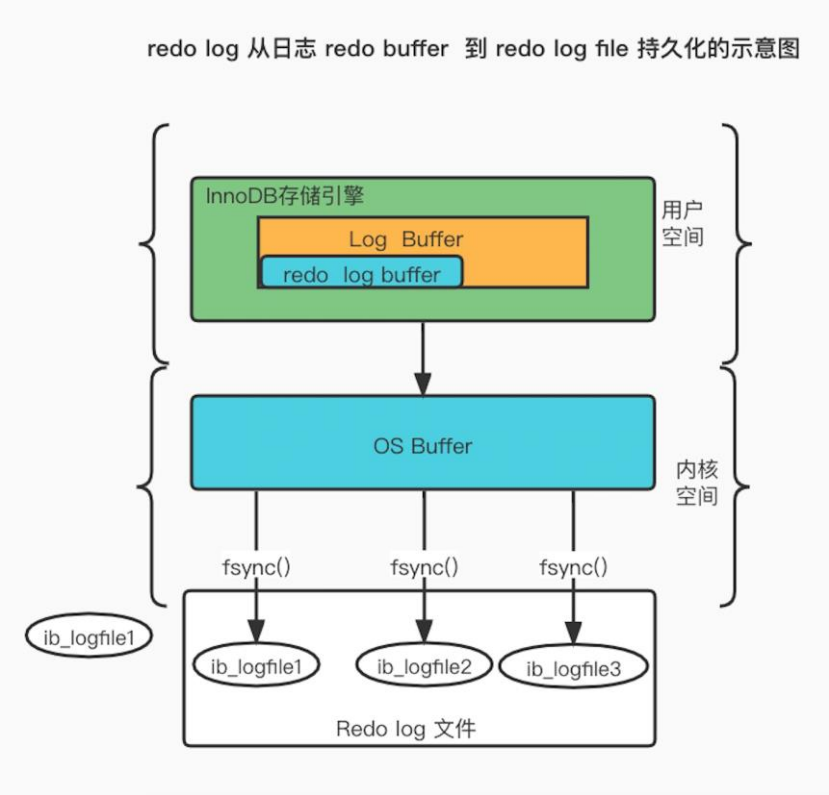
MySQL 通过 WAL（日志先行）技术，配合 redo log 保证了事务的持久性，redo log 用于 **crash-safe（崩溃恢复）**，记录的是物理日志，记录了某个数据页做了什么修改，比如对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了 AAA 更新，每执行一个事务就会产生一条或多条物理日志。

redo log 的作用：

- redo log 实际是 WAL 技术中的写日志部分，实现事务的持久性，让 MySQL 有 crash-safe 的能力
- redo log 是顺序循环写，将写操作从【随机写】变成了【顺序写】

redo log 是直接写入磁盘的吗？

**不是！！redo log 有自己的缓存---redo log buffer**，每当产生一条 redo log，会先写入 redo log buffer，后续再持久化到磁盘中



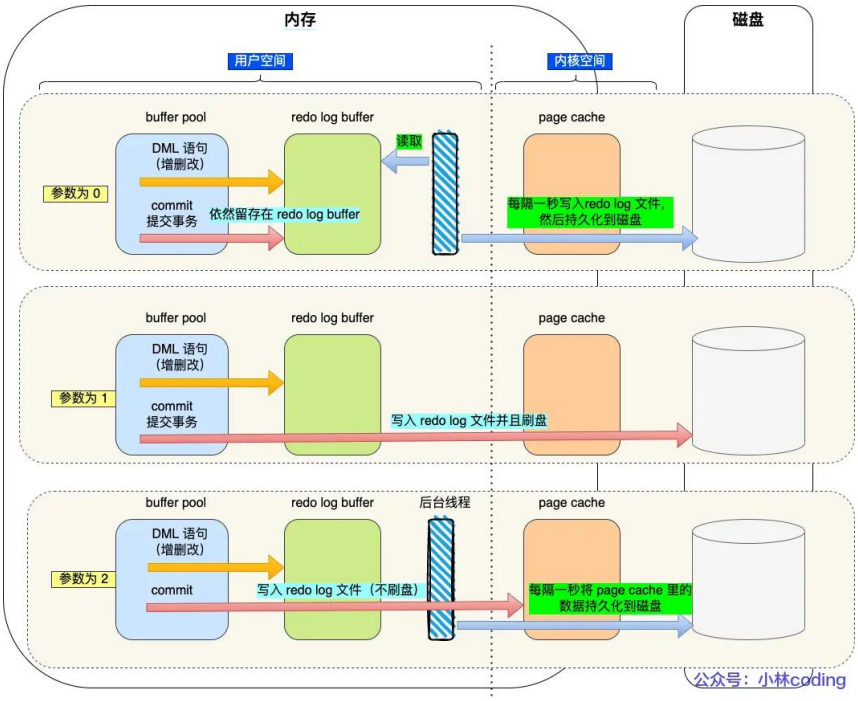
redo log 刷盘策略

刷盘时机：

- MySQL 正常关闭时
- 超过 redo log buffer 一半时
- 后台线程每 1 秒持久化一次
- 事务提交时：通过 innodb_flush_log_at_trx_commit 参数，控制在每次事务提交时的落盘策略
- write pos 追上 checkpoint 时，也就是执行一次 checkpoint 时，详情见下

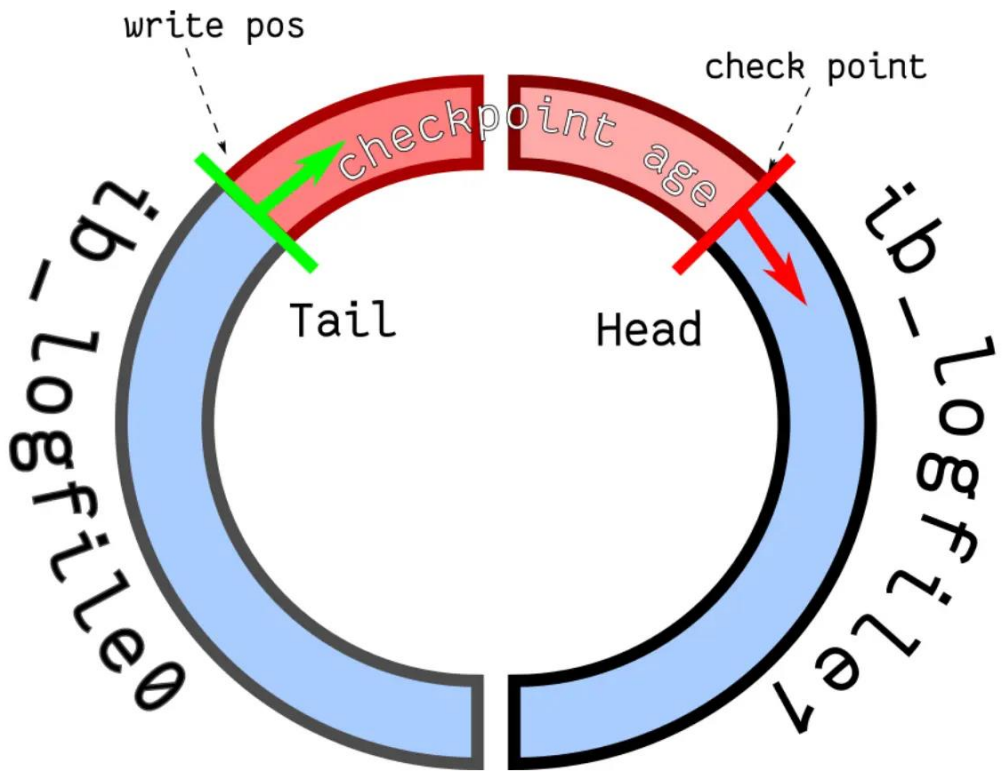
每次事务提交时落盘策略，也就是 innodb_flush_log_at_trx_commit 参数控制了什么：

- 参数为 0：事务提交不会触发落盘，还是留在 redo log buffer 中，由后台线程每 1 秒先调用 write()写到 Page Cache，再调用 fsync()持久化到磁盘。（MySQL 进程崩溃会导致上一秒所有事务数据丢失）
- 参数为 1（默认值）：事务提交时将缓存在 redo log buffer 中的 redo log 直接持久化到磁盘。
- 参数为 2：事务提交时，将 redo log buffer 中的 redo log 写到操作系统缓存（Page Cache），由后台线程每 1 秒调用 fsync()持久化到磁盘（MySQL 进程崩溃不会丢数据，只有 os 崩溃或断电的情况下，上一秒所有事务的数据才会丢失）



redo log 文件写满了怎么办？

redo log 是循环写的方式，相当于**一个环形**，InnoDB 用 write pos 表示 redo log 当前记录写到的位置，用 checkpoint 表示当前要擦除的位置，如下图：




- write pos 和 checkpoint 都是顺时针移动
- write pos ~ checkpoint 之间（图中红色部分）用来记录新的更新操作。
- checkpoint ~ write pos 之间（图中蓝色部分）：待落盘的脏数据页记录

当 write pos 追上 checkpoint 时，意味 redo log 文件满了，MySQL 会阻塞，将 buffer pool 中脏页刷新到磁盘中，并向后移动 checkpoint。

一次 checkpoint 的过程就是脏页刷新到磁盘中变成干净页，然后标记 redo log 哪些记录可以被覆盖的过程。

### 3.6.3 binlog

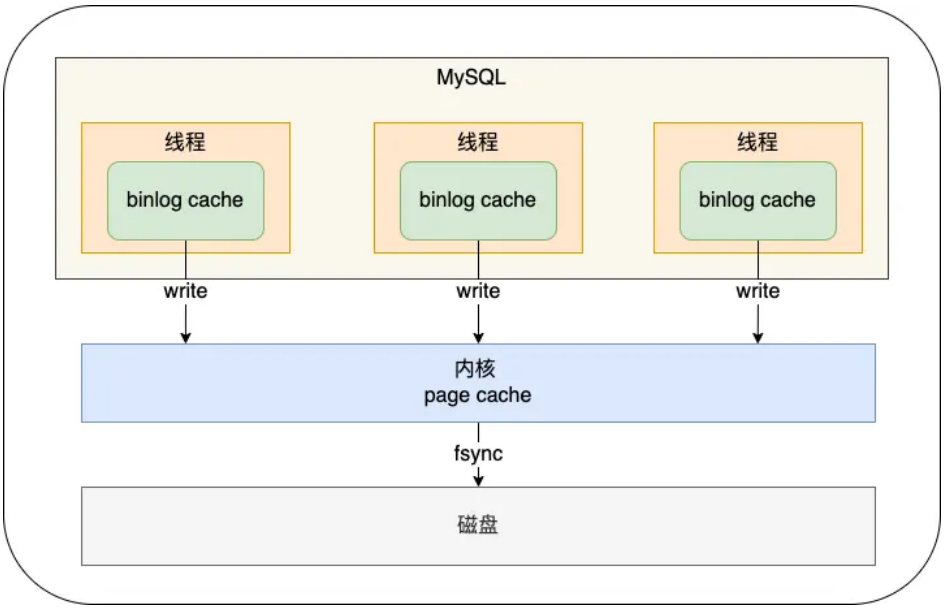
binlog 记录了表结构变更和表数据修改的日志，用于备份恢复、主从复制（不能用于 crash-safe）。

binlog 和 redo log 的区别？

1. **适用对象**：binlog 是 Server 层的实现，所有存储引擎都可以用；redo log 是 InnoDB 的实现。
2. **文件格式不同**：redo log 是物理日志，记录了某个数据页做了什么修改；binlog 是逻辑日志（也不完全是），记录了每行 SQL，分为三种格式
   - STATEMENT：**记录了逻辑操作**，每一条修改的 SQL 都记录到 binlog 中
     - 缺点：存在动态函数问题，如 now 函数，主库上执行的结果和从库执行的结果不一致，导致主动复制的数据不一致。
   - ROW：**记录行数据最终被修改成什么样子**。
     - 缺点：每行的变化结果都会记录，如执行批量 update 语句，更新多少行就有多少行 ROW 记录，会使 binlog 过大；而 STATEMENT 只会记录一个 update
   - MIXED：**包含 STATEMENT 和 ROW 模式**，会根据不同情况自动使用这两个模式
3. **写入方式不同**：binlog 是追加写，写满一个文件就创建一个新的，**保存的全量日志**，不会循环写；redo log 是循环写，写满就从头开始写，**保存未被刷入磁盘的脏页日志**。
4. **用途不同**：binlog 用于备份恢复、主从复制；redolog 用于掉电恢复

binlog 刷盘时机

每个线程有一个 binlog cache（Server 层的 cache），事务执行过程中，先把日志写到 binlog cache 中，事务提交时，再把 binlog cache 写到 binlog 文件中。




最终 binlog cache 都是写到同一个 binlog 文件中。

MySQL 提供 sync_binlog 参数来控制数据库的 binlog 刷到磁盘上的频率：

- sync_binlog=0：每次提交事务只 write，不 fsync，由操作系统决定何时将数据持久化

到磁盘。

- sync_binlog=1：每次提交事务执行完 write 马上执行 fsync
- sync_binlog=N（N>1）：每次提交事务都 write，但是累积 N 个事务后才 fsync。

### 3.6.4 MySQL 更新一条记录的过程

具体更新一条记录 UPDATE t_user SET name = 'xiaolin' WHERE id = 1; 的流程如下：

1. Server 层执行器具体执行，调用存储引擎接口，通过查询条件获取记录：
   - 如果记录本身就在 buffer pool 中，则直接返回给执行器；
   - 记录不在 buffer pool 中，则将数据页从磁盘中读入 buffer pool，并返回给执行器；
2. 执行器获取记录后，判断更新前后是否一致：
   - 一致则不进行后续流程；
   - 不一致则将更新前和更新后的数据都传给存储引擎层，让 InnoDB 真正执行记录更新
3. 开启事务，记录相应的 undo log，同时写 undo 页对应的 redo log
4. 更新记录，会先更新内存（同时标记为脏页），然后写入 redo log，这时更新就完成了；后续由后台线程将脏页写入到磁盘；
5. 至此一条记录更新完毕
6. 记录该语句对应的 binlog，此时 binlog 保存在线程的 binlog cache 中
7. 事务提交（两阶段提交）：
   - prepare 阶段：将 redo log 对应的事务状态置为 prepare，随后刷新到磁盘；
   - commit 阶段：将 binlog 刷新到磁盘；随后调用存储引擎提交事务接口，将 redo log 状态设置为 commit，write 到 page cache 中（只要 binlog 写入成功，则认为事务执行成功了）；
8. 至此，一条更新语句执行完成。

可以看到，事务提交的过程中，采用了两阶段提交的方式持久化两种日志，主要是为了避免两份日志之间的逻辑不一致。

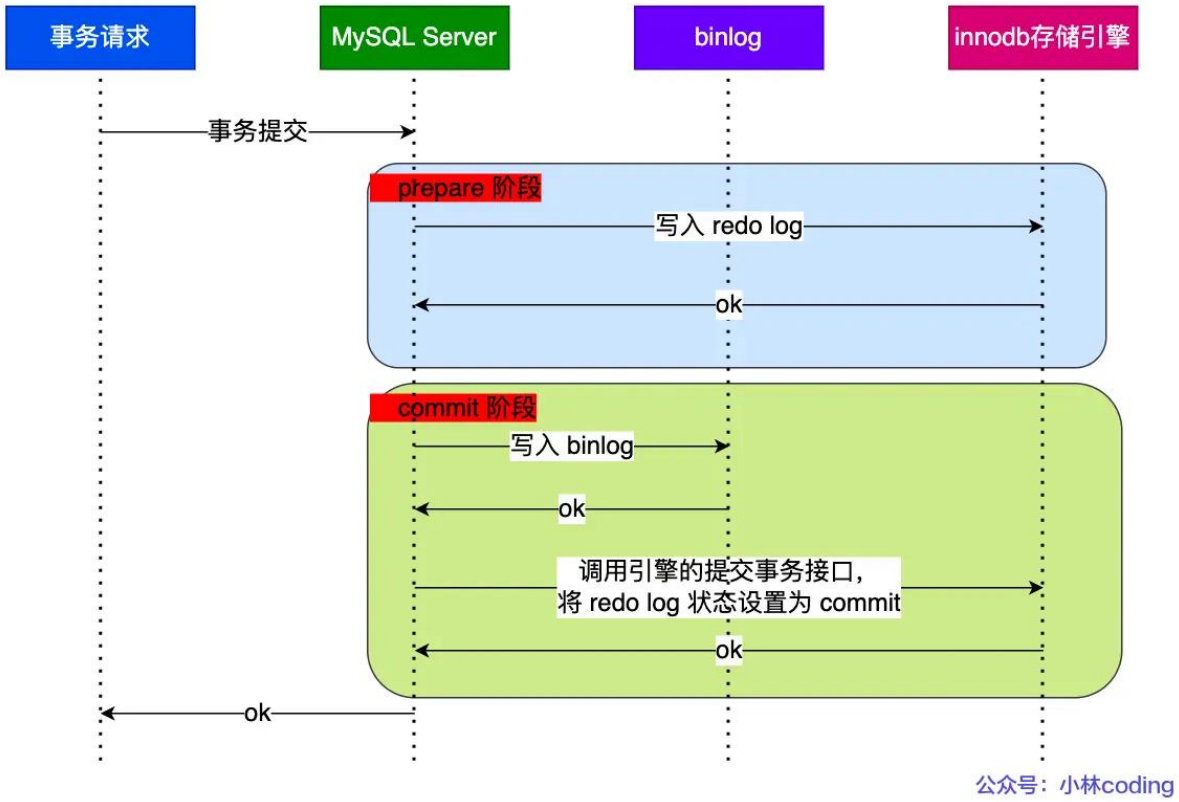
为什么要有两阶段提交？

- **redo log 刷入磁盘但是 MySQL 宕机了，binlog 没来得及刷**：MySQL 重启后，通过 redo log 可以恢复，但是 binlog 没有记录到这条更新语句，在主从架构中，binlog 会复制到从库，从库会保存旧值，**会导致主从不一致**；

- **binlog 刷入磁盘但是 MySQL 宕机了，redo log 没来得及刷**：崩溃恢复后此事务无效，主库中还是旧值，但是 binlog 记录了更新语句，在主从架构中，从库会保存更新后的语句，实际上这个更新事务是无效的，**也会导致主从不一致**；

为了解决这种问题，MySQL 采用两阶段提交的方式解决，单个事务的提交拆分成两个阶段，分别是【准备（prepare）阶段】和【提交（commit）阶段】

两阶段提交的过程



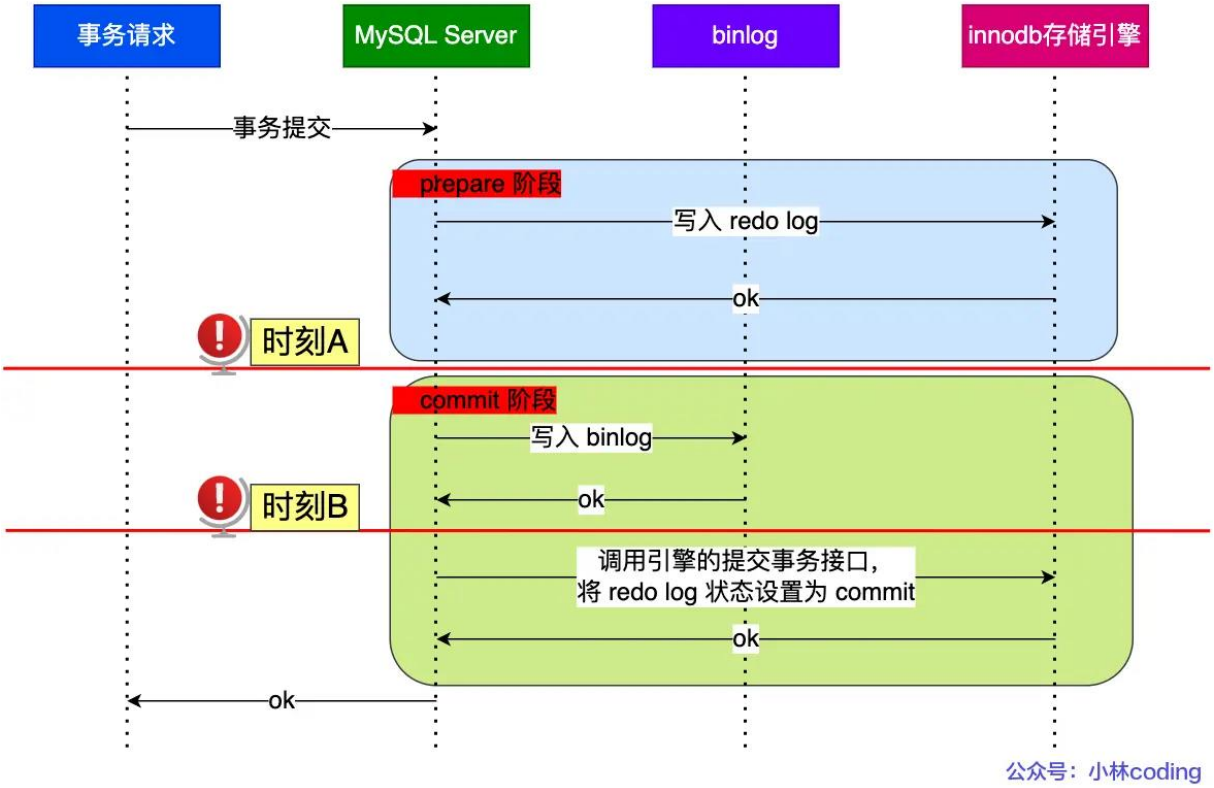

在事务提交过程中有两个阶段，实际就是将 redo log 的写入拆成两个步骤：prepare 和 commit，中间再穿插写入 binlog，具体如下：

- prepare 阶段：写入 redo log，并将对应事务状态设置为 prepare，然后将 redo log 持久化到磁盘（innodb_flush_log_at_trx_commit=1 的作用）；
- commit 阶段：写入 binlog，然后将 binlog 持久化到磁盘（sync_binlog=1 的作用），接着调用存储引擎的提交事务接口，将 redo log 状态设置为 commit，此时不需要保证刷新到磁盘，因为只要 binlog 持久化成功则说明事务成功了。

可以看到，MySQL 是以 binlog 是否写成功来标识事务是否提交成功了，因为 binlog 写入

后就会被从库（或者从 binlog 恢复出来的库）使用，所以在主库上也要提交此事务，就可以保证主从库的数据一致性。

接下来就可以讨论当两阶段提交不同阶段发生宕机重启会出现什么现象：




不管是 A（redo log 已写入磁盘，binlog 还没写入），还是 B（redo log 和 binlog 都写入磁盘，但是 redo log 还没有设置为 commit）崩溃，**此时 redo log 都是 prepare 状态**。

**宕机重启后步骤**：

- 扫描 redo log 文件，如果 redo log 处于 prepare 状态，则通过事务 ID 去 binlog 中查看。
  - binlog 中有：说明 redo log 和 binlog 都刷盘了，则**提交事务**。对应 B
  - binlog 中没有：说明 redo log 有但是 binlog 没有，则**回滚事务**。对应 A

所以是否回滚就看 binlog 有没有这个事务 ID 就可以了。

两阶段提交有什么问题？

主要有两个问题：

- 磁盘 I/O 次数高：如果 sync_binlog 和 innodb_flush_log_at_trx_commit 都为 1，则在提交过程中至少会通过 fsync 调用两次刷盘操作
- 锁竞争激烈：为了解决多事务的情况下日志提交一致性的问题，早期 MySQL 会通过 prepare_commit_mutex 锁来保证事务提交的顺序；事务进入 prepare 阶段前获取锁，commit 阶段结束后释放锁。

怎么解决两阶段提交的两个问题？

**组提交：**

各阶段都有一个队列，先进入队列的为 Leader，后续为 Follower，由 Leader 全权负责整队的操作，完成后通知队中其他事务操作结束。

MySQL 5.6 之前，prepare 阶段不变，将 commit 阶段分为以下三个过程：

- **flush 阶段**：多事务按进入的顺序将 binlog 从 cache 写入文件（不刷盘）
- **sync 阶段**：对 binlog 文件做 fsync 操作（多事务 binlog 合并一次刷盘）
- **commit 阶段**：各事务按顺序做 innoDB commit 操作

上述每个阶段都有一个队列，锁只针对每个队列进行保护，不再锁住提交事务的整个过程，锁粒度减小了，多阶段可以并发执行，同时多事务提交时会将多个 binlog 刷盘操作合并成一个，减少了磁盘 I/O 次数。

MySQL 5.7 之后，prepare 阶段也做了改变，多事务不再各自执行 redo log 刷盘操作，而是推迟到了组 flush 阶段。先进入队列的是 Leader

- **flush 阶段**：**支撑 redo log 的组提交**；先对 redo log 做一次 write+fsync，刷盘。再对 binlog 做一次 write，不刷盘
- **sync 阶段**：**支撑 binlog 的组提交**；达到指定时间或者队列中事务数达到 N 个，直接调用 fsync 对 binlog 刷盘
- commit 阶段：承接 sync 阶段的事务，完成最后的提交；将 redo log 状态置为 commit

MySQL 磁盘 I/O 很高，有什么优化的办法？

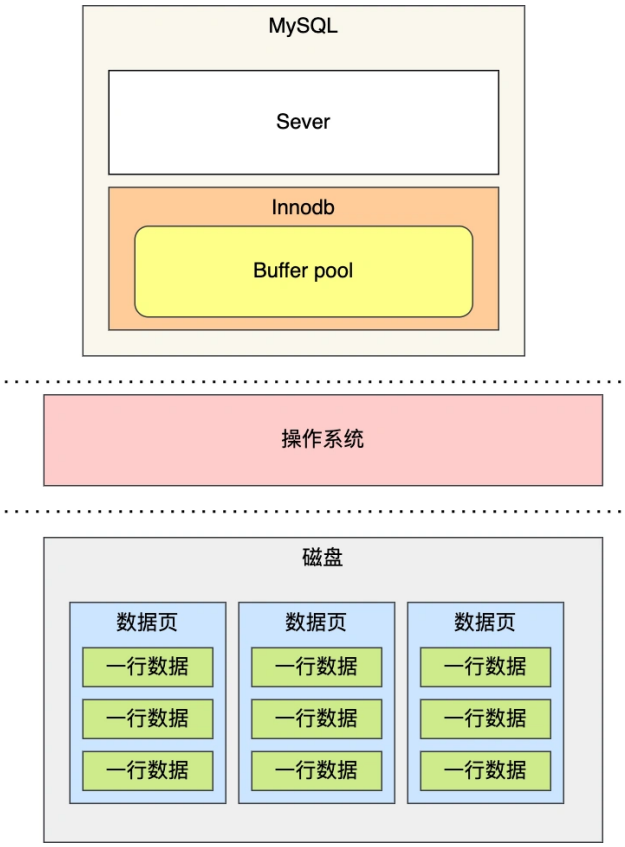
- **延迟 binlog 刷盘时机**：将 sync_binlog 设置为大于 1 的值；缺点是主机掉电时会丢失 N 个事务的 binlog 日志。
- **延迟 redo log 的刷盘时机**：将 innodb_flush_log_at_trx_commit 设置为 2；表示只写到 page cache，不刷盘；缺点是主机掉电会丢数据。

- **延迟 binlog 组提交的时机（只有开启了组提交才需要）**：增加等待事务的个数或者时间；缺点是会增加语句的响应时间；但是可能不会丢数据，因为 binlog 已经写入了 page cache，只要系统没宕机，就会持久化到磁盘

## 3.7 MySQL buffer pool

### 3.7.1 buffer pool 简介

InnoDB 搞了个**缓冲池（Buffer Pool）**，来提高数据库的读写性能。在 MySQL 启动时就会向 OS 申请一片连续的内存空间，默认 128MB，此时缓冲页是空闲的，随着程序运行，才会通过缺页中断将虚拟地址和物理地址建立映射关系。

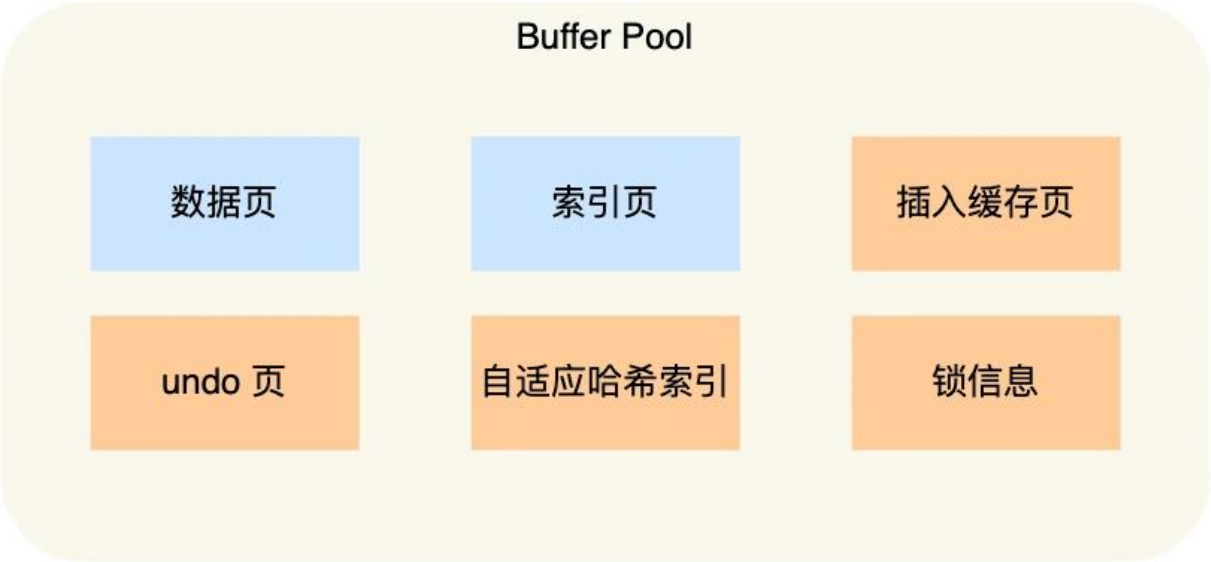



有了缓冲池之后，读写大致流程如下：

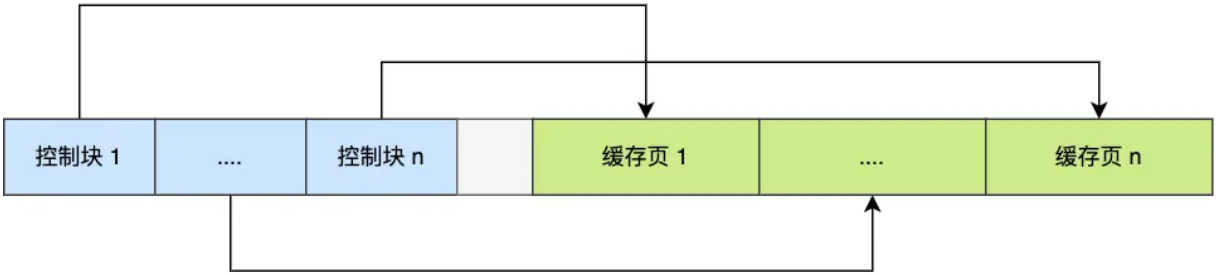
- **读取数据时**：如果存在于 Buffer Pool 中，则客户端直接读取 Buffer Pool 中数据，否则再去磁盘读取。
- **修改数据时**：如果存在于 Buffer Pool 中，则首先修改 Buffer Pool 中数据所在页，并把该页设置为脏页，最后由后台线程写入磁盘；如果不在 Buffer Pool 中，则先读到 Buffer Pool 中，再执行上述操作。

Buffer Pool 缓存什么呢？

按页缓存，Buffer Pool 中的页叫做缓存页。除了缓存【索引页】和【数据页】，还包括 undo 页、插入缓存页、自适应哈希索引、锁信息等。




每一个缓存页都有一个控制块，控制块信息包括【缓存页地址、缓存页表空间、页号、链表节点】等，当剩余空间不足以分配一对控制块和缓存页时，会产生碎片空间。



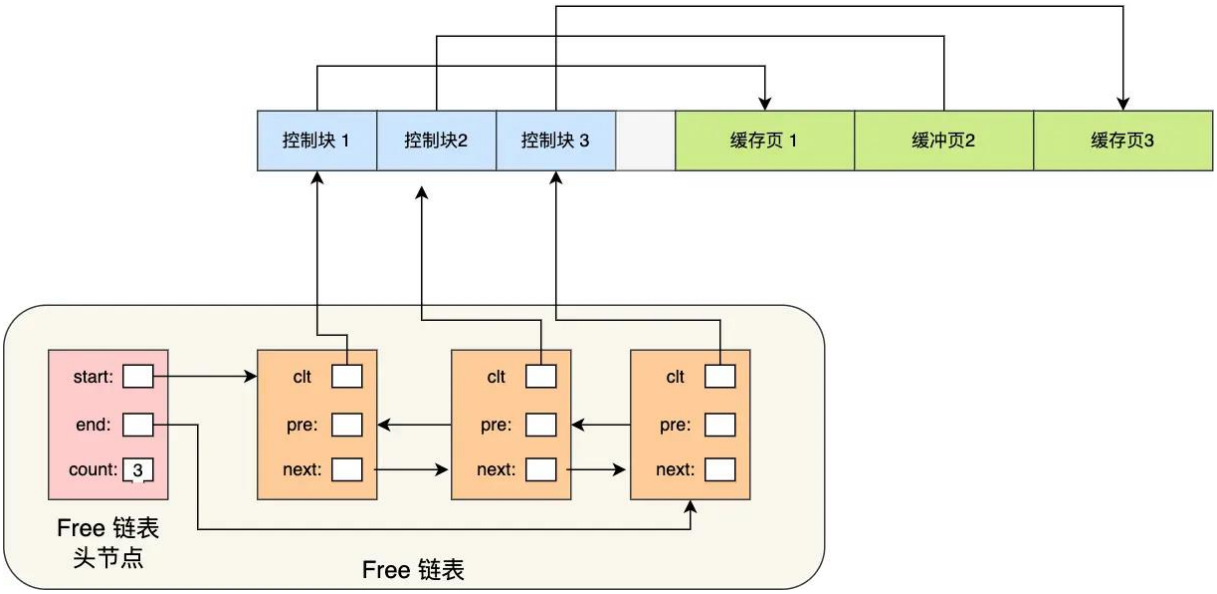

## 3.7.2 管理 buffer pool

InnoDB 通过三种页和链表管理缓存页：

- Free Page（空闲页）：表示此页未被使用，位于 Free 链表。
- Clean Page（干净页）：表示此页已被使用但是页面未发生修改，位于 LRU 链表。
- Dirty Page（脏页）：表示此页【已被使用】且【已经被修改】，脏页同时位于 LRU 链表和 Flush 链表。

### 3.7.2.1 管理空闲页

通过链表结构，将空闲缓存页的【控制块】作为链表节点，这个链表成为 **Free 链表（空闲链表）**。

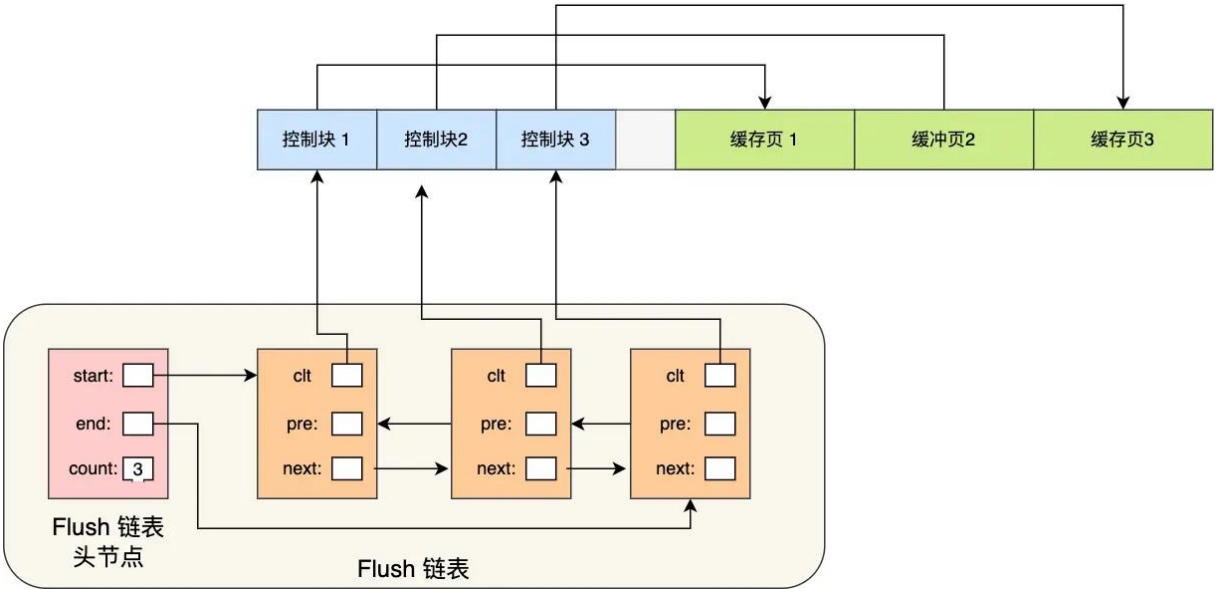



Free 链表有一个头节点，包含链表的头结点地址、尾结点地址、当前链表节点数量；Free 链表节点是一个个控制块，控制块中包含空闲缓存页的地址，相当于链表中每个节点都对应一个空闲的缓存页。

当磁盘中加载一个页到 buffer pool 中时，就从 Free 链表中取一个空闲的缓存页，填充控制块信息，并将该缓存页对应的控制块从 Free 链表中移除。

### 3.7.2.2 管理脏页

类似于管理空闲页的方式，也有一个链表，称为 **Flush 链表**，链表节点也是控制块，区别是 **Flush 链表的元素都是脏页**。



后台线程可以遍历 Flush 链表，将脏页写入磁盘。

脏页什么时候会刷入磁盘呢？

- redo log 日志满了会主动触发脏页刷新
- Buffer Pool 空间不足需要将数据页淘汰时，如果淘汰的是脏页需要先刷盘
- MySQL 空闲时后台线程会定期将适量的脏页刷盘
- MySQL 正常关闭之前，会将所有脏页刷盘。

#### 3.7.2.3 【重点 ！】如何提高缓存命中率

第一印象我们是使用 LRU 算法来提高缓存命中率：

- 思路：链表头部是最近使用的，链表末尾的节点是最久没有被使用的，空间不足时要从链表尾部淘汰最久没有被使用的节点，腾出空间。
- 当访问的页在 Buffer Pool 里，那么就直接把该页对应的 LRU 链表节点移动到链表头部
- 当访问的页不在 Buffer Pool 里，那么要取一个空闲页放入从磁盘中读取的页，并将该页移动到 LRU 链表头里，如果空间不足还要先淘汰 LRU 链表末尾的节点。

上述为传统的 LRU 的实现思路，但是传统的 LRU 有以下两个问题是无法避免的：

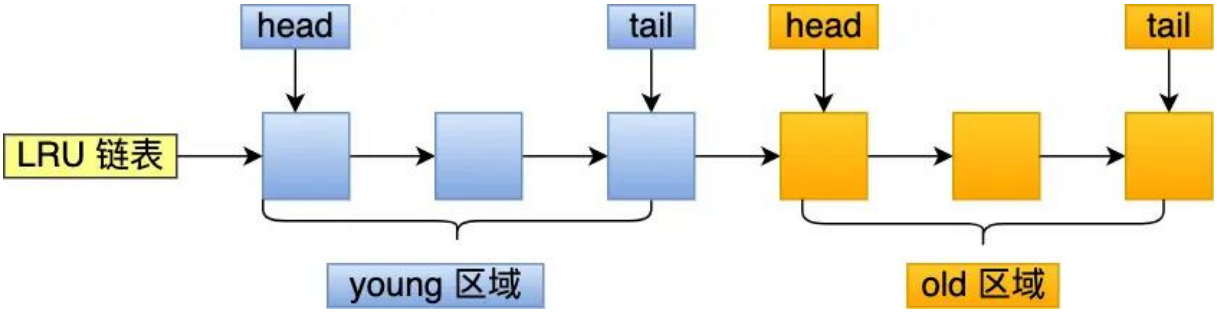
- **预读失效**：提前加载的页没有被访问，但是占据了链表的头部位置，空间不足时反而淘汰了末尾很可能频繁访问的页，会降低缓存命中率
- **Buffer Pool 污染**：一个 SQL 扫描了大量数据，可能会将 Buffer Pool 数据全部替换出去，导致大量热数据被淘汰，等热数据被再次访问时，会产生大量的磁盘 I/O，性能会急剧下降，这个过程成为 Buffer Pool 污染。

MySQL 是如何解决预读失效的？

优化方向是：让预读的页停留在 Buffer Pool 里的时间尽可能的短，并且**让真正被访问的页才移动到 LRU 链表的头部，保证真正被读取的热数据留在 Buffer Pool 里的时间尽可能长**。

MySQL 改进 LRU 的方式：

将 LRU 划分了 2 个区域：old 区域和 young 区域。young 区域在 LRU 链表的前半部分，old 区域在后半部分，预读的页只加入到 old 区域的头部，只有当预读页真正被访问时，才将该页插入到 young 区域的头部。




注意看，仅采用划分两个区域的方式不会解决 Buffer Pool 污染的问题，设想这样一个场景，一条 SQL 语句读取了大量的页，这些页虽然过程中先会在 old 区停留一会，但是最终都会放到 young 区的头部，导致缓存污染，关键点在于，页读到 old 区之后，只要有访问立马就搞到了 young 区头部，那么是不是提高进入 young 区的门槛，就可以解决缓存污染的问题呢？我们看一下 MySQL 是如何解决的？

MySQL 是如何解决缓存污染的？

优化方向：young 区作为热点数据的集中地，需要尽量提高进入 young 区的门槛。

MySQL 改进的方式：

进入 young 区条件增加了一个停留在 old 区域的时间判断。

具体实现：

- 对 old 区某数据进行第一次访问时，记录下第一次访问时间
  - 如果后续访问时间与第一次访问时间的时间间隔在某时间间隔内（默认 1s）：则不会将该缓存页从 old 区域移动到 young 区域。
  - 如果不在某时间间隔内（默认 1s）：将该缓存页从 old 区域移动到 young 区域的头部。

也就是同时满足【被访问】和【在 old 区域停留时间超过 1 秒】两个条件，才会插入到 young 区域头部。

MySQL 额外优化

针对 young 区做了一个额外的优化，当访问前面 1/4 区域时不会移动到链表头部，只有后面 3/4 被访问了才会。

MySQL 提高缓存命中率方式的总结：

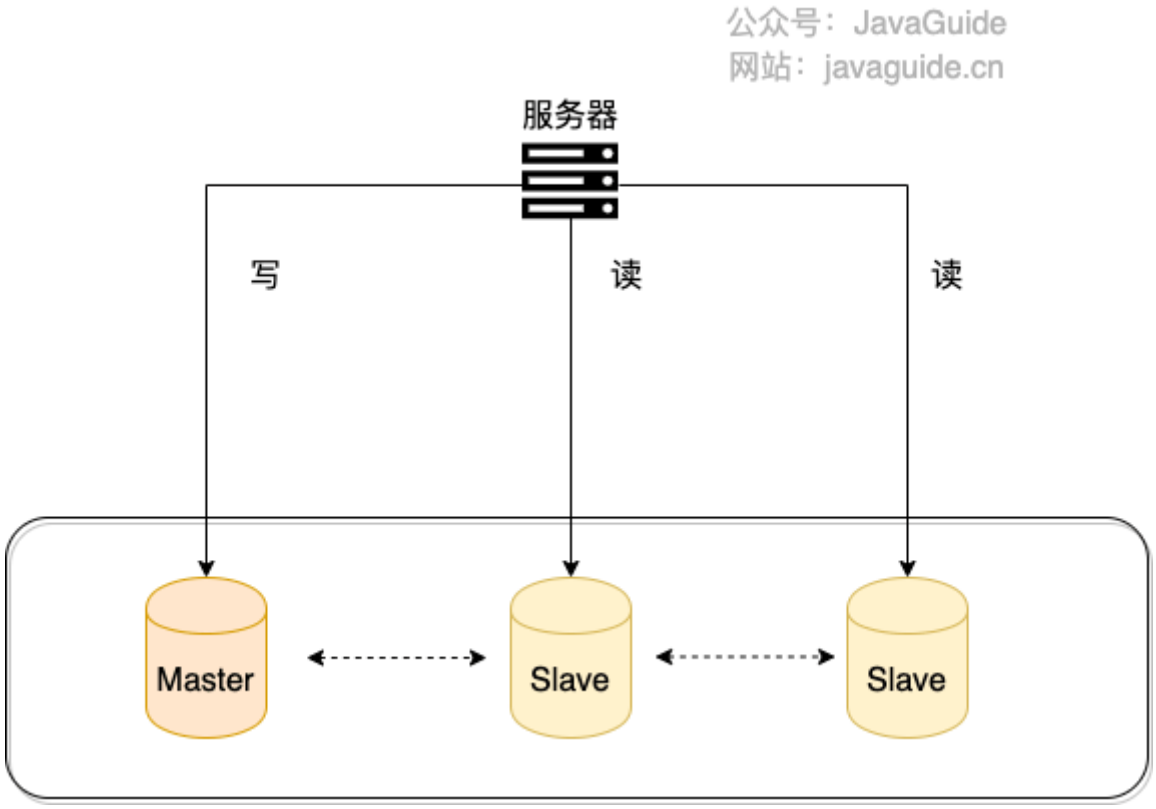
- 采用改进后的 LRU 算法
  - 将 LRU 链表划分为 young 区域和 old 区域，当页从磁盘中读取到 LRU 链表时，先放到 old 区域的头部，只有被访问且后续访问相比于第一次访问间隔 1s 以上（也就是在 old 区域停留超过 1s）时，才会移动到 young 区域的头部。
  - young 区域前 1/4 部分被访问时不会移动到 young 头部，只有后面 3/4 被访问了才会，目的是防止 young 区域节点频繁移动到头部。

## 3.8 MySQL 进阶

### 3.8.1 读写分离

读写分离含义

**读写分离主要是为了将对数据库的读写操作分散到不同的数据库节点上。**




一般情况下，我们会选择一主多从，一台主数据库负责写，其他的从数据库负责读。主库和从库之间会进行数据同步。

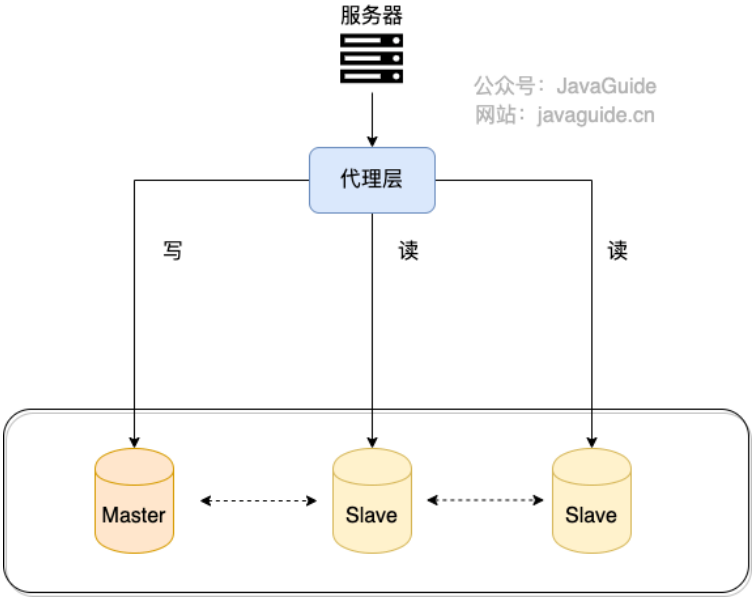
如何实现读写分离？

分为以下几步：

1. 部署多台数据库，选择其中一台作为主数据库，其他作为从数据库
2. 保证主库和从库数据实时同步，也就是**主从复制**
3. 系统将**写请求交给主库处理，读请求交给从库处理**

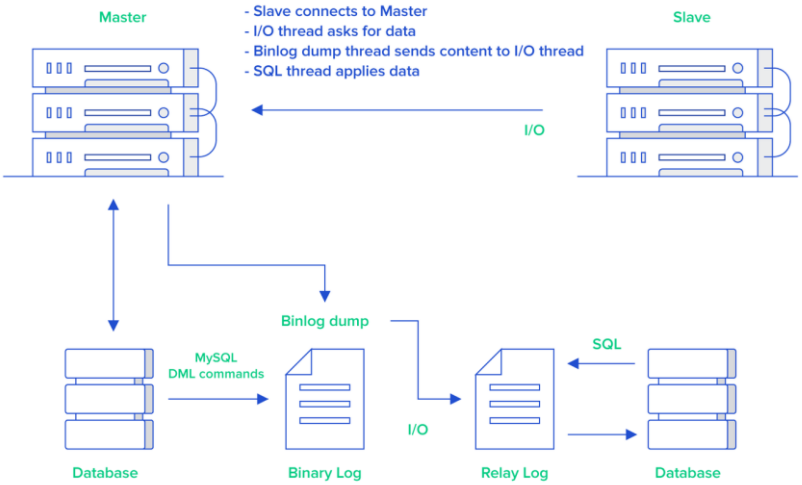
落地方案有两种：

1. 代理方式：应用和数据库中间加一层代理，应用所有数据都交给代理层处理，代理层负责分离读写请求，将它们路由到对应的数据库中。
   - 优点：加了一层代理，客户端不需要关注连接维护、信息维护等细节
   - 缺点：架构比较复杂，代理需要高可用架构



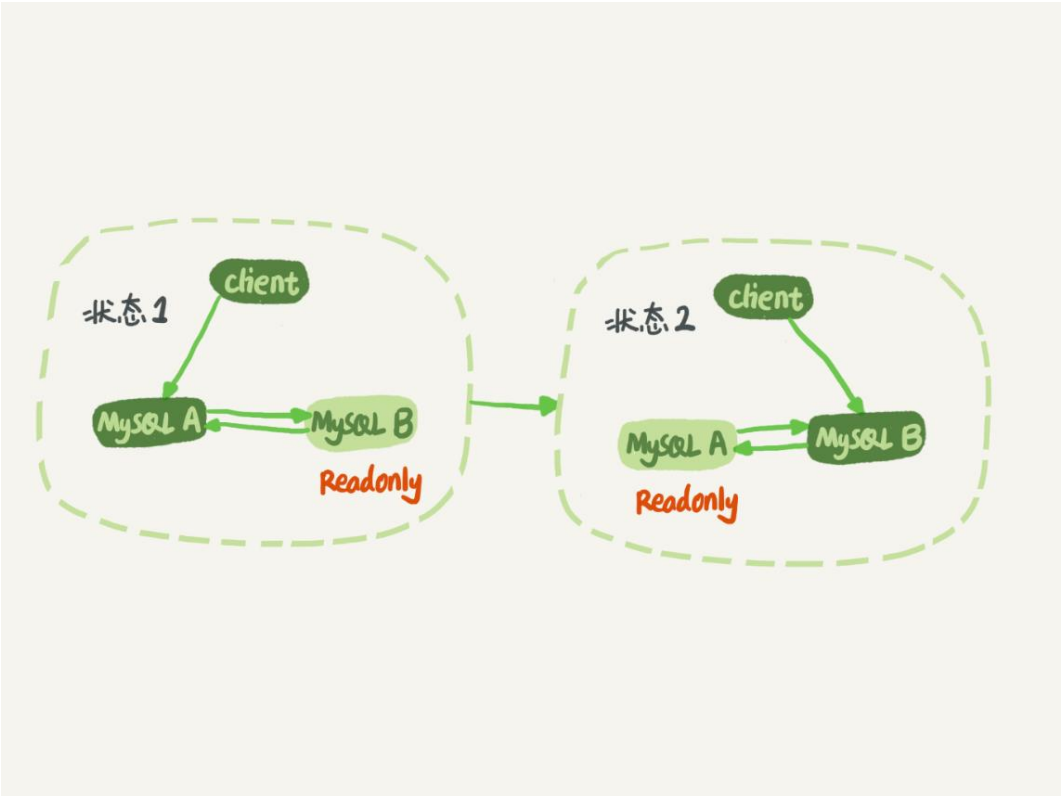

2. 客户端直连（组件方式）：在应用程序中引入第三方组件来帮助读写请求。
   - 优点：查询性能较好，且整体架构简单，排查问题较方便
   - 缺点：应用程序要了解部署细节

主从复制的原理




1. 主库将数据库中数据的变化写入到 binlog
2. 从库连接主库
3. 在备库上通过 change master 命令，设置主库的服务器信息（IP、端口、用户名、密码等）以及要从哪个位置开始请求 binlog（包含文件名和日志偏移量）
4. 在从库上执行 start slave 命令，从库会启动两个线程，io 线程和 sql 线程，其中 io 线程负责与主库建立连接。
5. 主库校验完信息后，开始按照从库传来的位置，从本地读取 binlog，发送给从库
6. 从库拿到 binlog 后，写到本地文件，称为中转日志（relay log）
7. sql 线程读取 relay log，解析命令并执行

双主复制架构




生产中使用较多的是双主架构，也就是两个节点之间互为主备，这样在切换主备时就不需要修改主备关系了

但是有个**循环复制**的问题：

- 在主 A 上更新了一个语句，把生成的 binlog 发送给 B，节点 B 执行完更新语句后同样会写 binlog，此时主 A 是 B 的从节点，那么又会把 B 新生成的 binlog 拿过来执行一次，A 和 B 之间会不断循环执行更新语句，即循环复制。

解决办法：

通过判断 server id 判断是否在本节点执行更新语句（binlog 中的每一个命令都会记录第一次执行时所在实例的 server id），所以可以规定

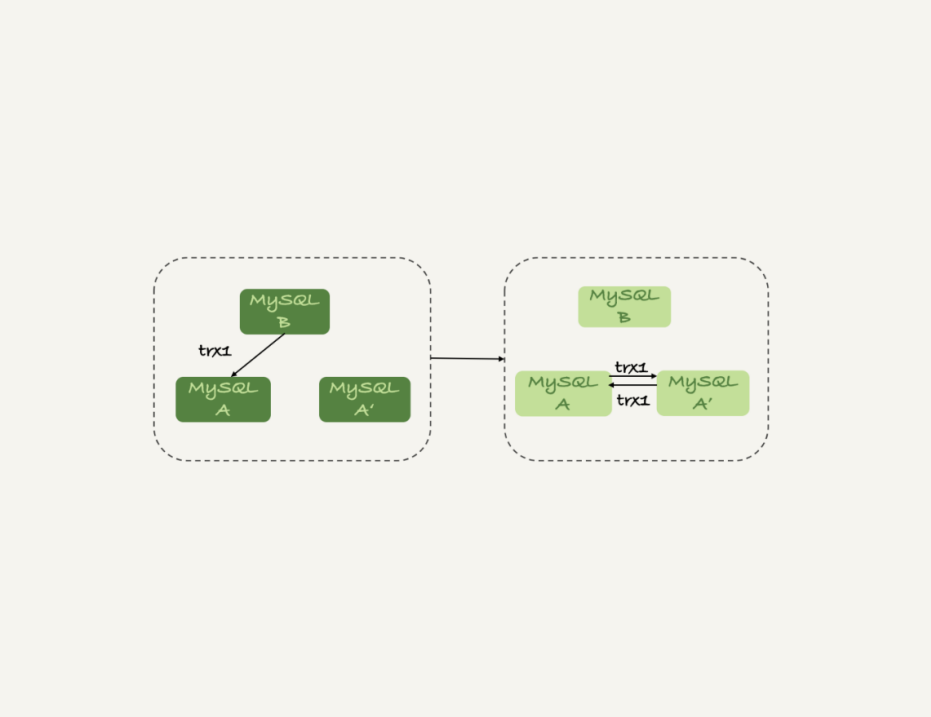
1. 每个库 server id 必须不同
2. 备库接收到 binlog 并在重放过程中，这条语句在新 binlog 中设置的 server id 与原 server id 相同
3. 每个库收到从主库发来的 binlog 后，先判断 server id，如果和自己相同则直接丢弃

具体逻辑如下：

1. 节点 A 更新事务，binlog 中记录的都是 A 的 server id
2. 传到 B 执行一次后，B 生成的 binlog 中的 server id 也是 A 的 server id
3. 再传回给 A，A 判断 server id 和自己相同，则直接丢弃不处理

什么情况下通过 server id 也无法解决循环复制的问题呢？

1. 主库 A 更新事务后修改了 server_id：在一个主库更新事务后，用命令 set global server_id=x 修改了 server_id
2. 数据迁移，出现三节点复制的场景：事务 1 在节点 B 执行，binlog 上 server_id 是 B，binlog 传给节点 A，随后 A 和 A'搭建了双主架构，就会出现循环复制，见下图。




针对场景 2 的解答 TODO

主从切换

TODO

读写分离会带来什么问题？如何解决？

**主从同步延迟**：写完主库后，主库的数据同步到从库需要时间，也就是主库和从库的数据存在延迟，这个时间差就导致了主库和从库的数据不一致性问题。

解决办法：

1. **强制走主库**：必须拿到最新结果的走主库，可以读到旧数据的走从库

- 必须拿到最新结果的请求，强制走主库：比如卖家发布商品后查看商品是否发布成功
- 可以读到旧数据的请求，可以走从库：买家逛商铺查看发布的商品，可以走从库
- 缺点：对于必须全部读最新结果比如“金融类业务”，读写压力都在主库，等于没做读写分离

2. **sleep 方案**：读从库前先 sleep 一下，比如 1 秒
    - 缺点：不精确，sleep 时间不好把握
3. **判断主备无延迟方案**：实际有三种方案，即判断备库收到的日志是否执行完成了（有明显的缺陷就是：存在一种客户端收到了提交确认，但是备库还没收到日志的状态）
    a. 对比主备延迟时间确保主备无延迟：从库执行查询请求前，先判断 seconds_behind_master（单位是秒）是否等于 0，必须要等这个参数变为 0 才能继续查询。
    b. 对比位点确保主备无延迟：读到的主库最新位点和备库执行的最新位点，两组值完全相同则表示日志已同步完成。
    c. 对比 GTID 集合确保主备无延迟：备库收到的所有日志的 GTID 集合和备库所有已经执行完成的 GTID 集合相同则表示日志已同步完成。
4. **semi-sync（半同步复制机制）+ 判断主备无延迟方案**：
    - 半同步复制机制流程：事务提交时主库把 binlog 发送给从库；从库收到 binlog 后，返回主库一个 ack，表示已收到；主库收到 ack 后，才会返回“事务完成”的确认给客户端。
    - !!! 缺点：
        1. 一主多从的情况下，在某些从库执行查询请求可能还会存在过期读的现象（因为只收到一个 ack 主就会返回事务成功给客户端了）
        2. 2. 在持续延迟的情况下，可能出现过度等待的问题（比如一直给主写，从和主就一直有延迟，从就一直无法提供读的服务）
5. 等主库位点方案：

```
select master_pos_wait(file, pos[, timeout]);
```

这条命令的逻辑：

- 在从库上执行
- 参数 file 和 pos 指主库上的文件名和位置
- timeout 参数可选，设置为正整数 N 表示这个函数最多等待 N 秒。

这个命令会返回正整数 M，表示从命令开始执行，到应用完 file 和 pos 表示的 binlog 位置，执

行了多少事务，主要是看**从库是否已经执行到了指定的主库上的 binlog position**

当事务 1 执行完后，再执行一个查询请求的逻辑如下：

1. trx1 事务更新完成后，马上在主库执行 show master status 得到当前主库执行到的 File 和 Pos
2. 选定一个从库执行查询语句
3. 在从库上执行 select master_pos_wait(File,Position,1)
4. 如果返回值>=0，则在从库执行查询语句
5. 否则到主库执行查询语句

6. GTID 方案：在第 5 条等主库位点方案中，还需要先在主库执行 show master status，在 5.7.6 之后，允许在执行完更新类事务后，将事务的 GTID 返回给客户端，所以等 GTID 方案可以少一次查询。

```
select wait_for_executed_gtid_set(gtid_set,1)
```

这条命令逻辑是：

- 等到，直到从库执行的事务中包含传入的 gtid_set，返回 0
- 超时返回 1

当事务 1 执行完成后，再执行一个查询请求的逻辑如下：

1. trx1 事务更新完成后，从返回包直接获取这个事务的 GTID，记为 gtid1
2. 选定一个从库执行查询语句
3. 在从库上执行 select wait_for_executed_gtid_set(gtid1,1)
4. 如果返回值是 0，则在从库中执行查询语句
5. 否则到主库执行查询语句

## 3.8.2 分库分表

分库分表概念

分库：将数据库中的数据分散到不同的数据库上

- **垂直分库**：将单一数据库按业务划分，不同的业务使用不同的数据库，例如将数据库中用户表、订单表、商品表分别拆为用户数据库、订单数据库、商品数据库
- **水平分库**：将同一个表按照一定规则拆分到不同的数据库，每个库位于不同服务器上

分表：对单表的数据进行拆分

- **垂直分表**：对数据表列的拆分，把一张列比较多的表拆分成多张表
- **水平分表**：将一张行比较多的表拆分成多张表，解决单一表数据量过大的问题。

什么时候需要分库分表？

- 单表数据达到千万级别以上，数据库读写速度比较换满
- DB 中数据占用空间大，备份时间久
- 应用并发量太大

常用的数据分片算法

- **哈希分片**：求指定 key（比如 id） 的哈希，然后根据哈希值确定数据应被放置在哪个表中。哈希分片比较适合随机读写的场景，不太适合经常需要范围查询的场景。
- **范围分片**：按照特性的范围区间（比如时间区间、ID 区间）来分配数据，比如 将 id 为 1~299999 的记录分到第一个库， 300000~599999 的分到第二个库。范围分片适合需要经常进行范围查找的场景，不太适合随机读写的场景（数据未被分散，容易出现热点数据的问题）。
- **地理位置分片**：很多 NewSQL 数据库都支持地理位置分片算法，也就是根据地理位置（如城市、地域）来分配数据。
- **融合算法**：灵活组合多种分片算法，比如将哈希分片和范围分片组合。

数据分片原则：先确定尽量减少扫描分片的数量，也就是根据某维度查询时，可以定位到具体是哪个分片。对于查询需求不是很频繁的维度，可以考虑采用冷热数据分离的思路，比如双写到 ES，根据 ES 查。

以淘宝订单号为例设置分片 key

https://developer.aliyun.com/article/1256632

有两种思路：1. 按照订单号做 hash 分散数据 2. 按照用户号做 hash 分散数据

主要是根据用户号来 hash 分片，可以保证每个用户的订单分到同一个表中

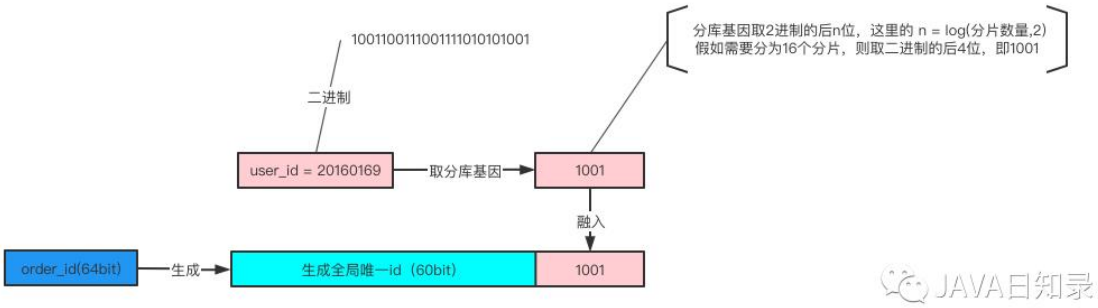
针对非分片 key 的列如何查询呢？

将分片 key 的信息冗余到想要查询的列中，可以通过查询的列直接知道所在分片的信息，这种

方法也叫做**基因法**。

原理出自一个理论：**对一个数取余 2 的 n 次方，那么余数就是这个数的二进制的最后 n 位数。**

假如我们现在根据 user_id 进行分片，采用 (user_id % 16) 的方式来进行数据库路由，这里的 user_id%16，其本质是 user_id 的最后 4 个 bit 位 **log(16,2) = 4** 决定这行数据落在哪个分片上，这 4 个 bit 就是分片基因




- 使用 user_id%16 分片，决定这行数据要插入到哪个分片中
- 分库基因是 user_id 的最后 4 个 bit，log(16,2) = 4，即 1001
- 在生成 order_id 时，先使用一种分布式 ID 生成算法生成前 60bit（上图中绿色部分）
- 将分库基因加入到 order_id 的最后 4 个 bit（上图中粉色部分）
- 拼装成最终的 64bit 订单 order_id（上图中蓝色部分）

这样保证了同一个用户创建的所有订单都落到了同一个分片上，order_id 的最后 4 个 bit 都相同，于是：

- 通过 user_id %16 能够定位到分片
- 通过 order_id % 16 也能定位到分片

由于字段 order_id 的设计中直接包含了分片键信息，所以我们可以直接通过分片键部分直接定位到分片上。

同样地，在插入时，由于可以知道插入时 user_id 对应的值，所以只要在业务层做一次字符的拼接，然后再插入数据库就行了。

只需要查询一次即可

分库分表之后的问题

1. join 操作：同一个数据库中的表分布在不同数据库中，导致无法使用 join 操作。

   解决办法：应用程序侧手动进行数据的封装，比如在一个 DB 中查询完，再根据这个数据

去另一个 DB 中找对应的数据。

2. 事务问题：引入分布式事务来解决。

3. 分布式 ID

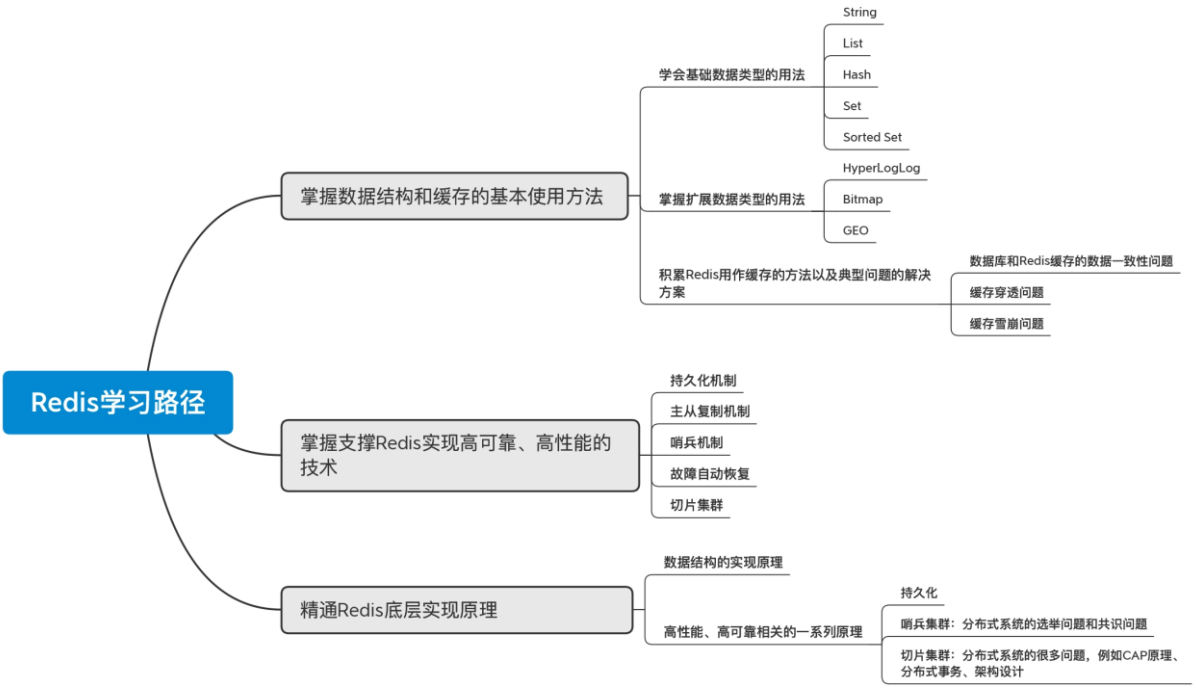
分库分表后的数据迁移

1. 比较 low 的方法，停机迁移。
2. **双写**：
    - 数据库双写（事务成功以老库为准，操作的数据不存在于新库的话，需要插入到新库中），查询走老模型；每天脚本对账，补平差异，并在新库导入历史数据；
    - 历史数据导入完毕且数据对账无误后，依然数据库双写（事务成功以新库为准），查询切新 DB，每天脚本对账，补平差异
    - 直到新老库数据一致为止，老库不再同步写入

### 3.8.3 深分页问题

where

# 四、 Redis 相关

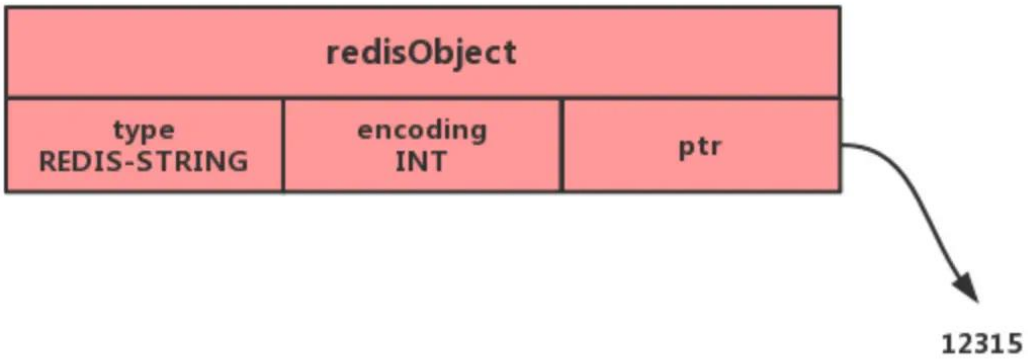



## 4.1 数据结构

https://xiaolincoding.com/redis/data_struct/command.html#string

redis 的 key 固定就是 string，redis 的 value 会有一个 redisobject，会标识 value 的类型以及 value 的指针，value 的类型包括 string、list、hash、set、zset、bitmap、hyperloglog、geo、stream

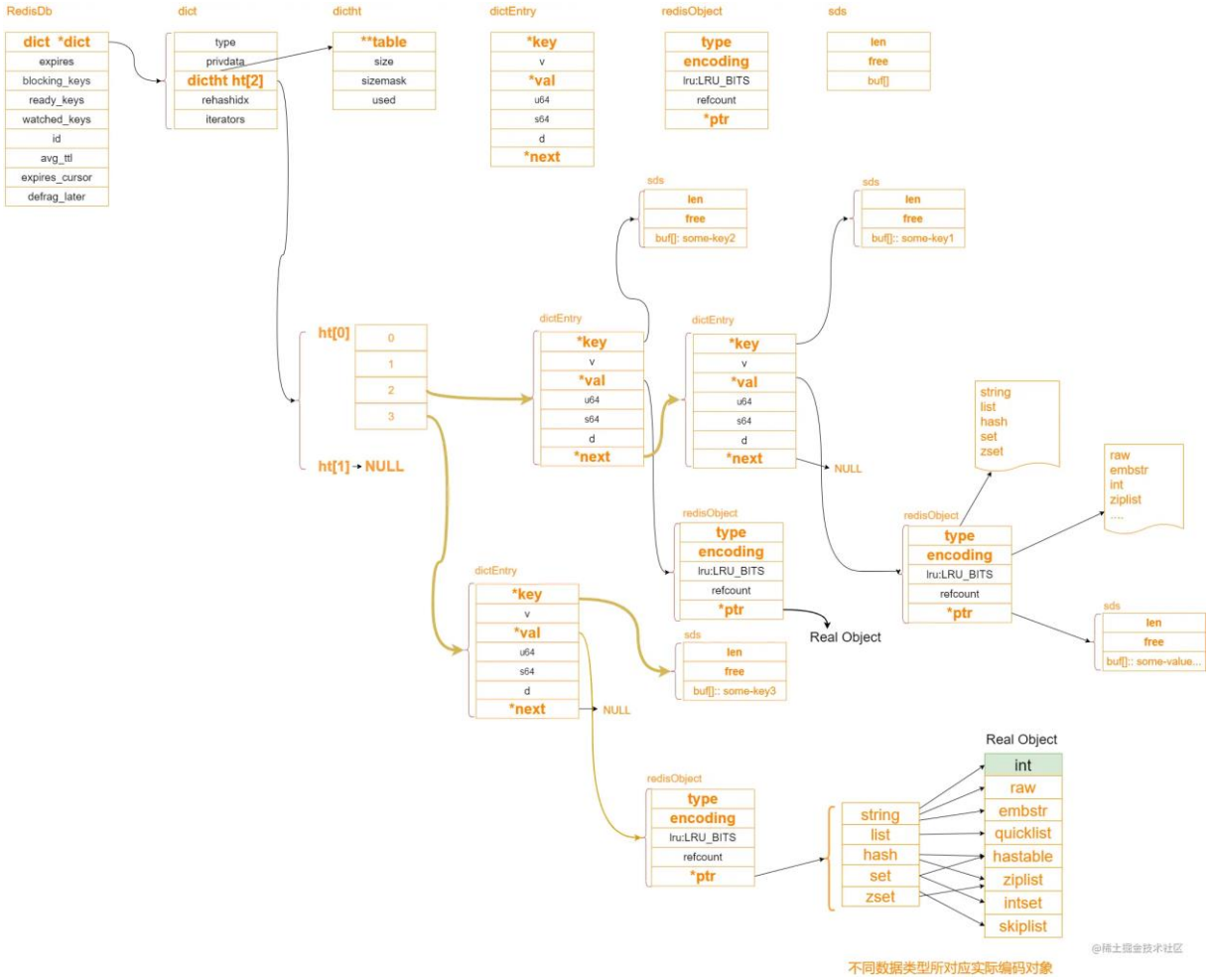
以 string 类型为例的 redisobject 结构如下：



# 4.1.0 redis 底层数据存储原理

## 01 概述

Redis 底层采用**数组**， key 就是对应数组的**索引**， 采用Hash(key)映射到数组上。解决冲突采用链地址法。具体可看参考下文。




## 02 底层存储原理

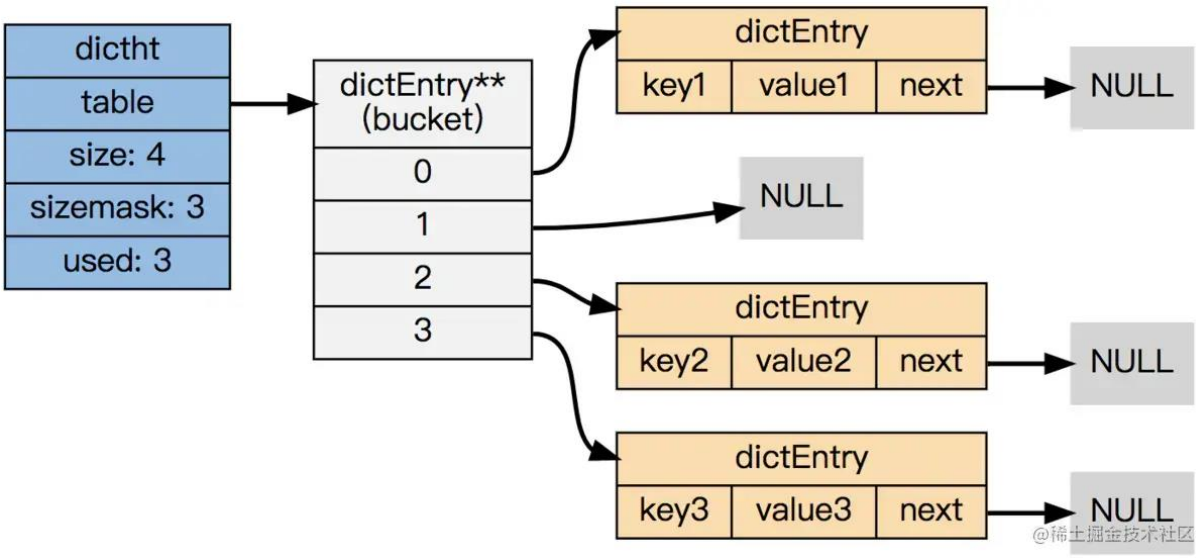
redis 中以 **redisDb** 作为整个缓存存储的核心，保存着我们**客户端需要的缓存**数据。

其结构如下：

```
typedef struct redisDb {
 dict *dict; // 最重要--字典类型，保存数据库的键值对
 dict *expires; // 重要--字典类型，保存过期的时间  dict
*blocking_keys; // 和 ready_key 实现 BLPOP 等阻塞命令  dict
*ready_keys; // 同上
 dict *watched_keys; // 实现 watch 命令，记录正在被 watch 的 key
 int id; // 数据库 id，默认 16 个，支持单个  long long avg_ttl; /*
Average TTL, just for stats */
} redisDb;
```

- **dict** 和 **expires** 是 redisDB 中最主要的两个属性，*分别保存了对象数据键值对和 key 的过期时间*，其底层数据结构都是 dict 字典。之所以分开存储，由于过期时间并不是数据的固有属性，虽然分开存储需要两次查找，但是却能节省内存开销。
- blocking_keys 和 ready_keys 主要为了*实现 BLPOP 等阻塞命令*
- watched_keys 用于实现 watch 命令，*记录正在被 watch 的一些 key*
- eviction_pool 是记录*可能被 lru 淘汰的一些备选 key*
- id 为*当前数据库的 id*，redis 支持单个服务多数据库，默认有 16 个

**03 dict**



```
/* 字典 : 每个字典使用两个哈希表，用于实现渐进式 rehash
*/typedef struct dict {
    // type 存储了 hash 函数，key 和 value 的复制函数等，比较以及销毁函数
dictType *type;
    // privdata 保存一些私有数据，决定了*type 保存的函数，实现了【多态】
void *privdata;
    dictht ht[2]; // 哈希表（2 个）, 正常使用ht[0],rehash 就会扩容使用
ht[1]
    int rehashidx; // 记录 rehash 进度的标志，值为 -1 表示 rehash 未
进
    int iterators; // 当前正在运作的安全迭代器数量
 } dict;
```

上述 dictht 就是个 hash 表，包含：

```
typedef struct dictht {
    // 哈希表节点指针数组（俗称桶，bucket）
    dictEntry **table;
    // 指针数组的大小
    unsigned long size;
    // 指针数组的长度掩码，用于计算索引值，其实永远都是 size-1
    unsigned long sizemask;
    // 哈希表现有的节点数量
    unsigned long used;
} dictht;
```

**size：标识 dictEntry 指针数组的长度。它总是 2 的指数次幂**

**dictEntry 指针数组（table）。key 的哈希值最终映射到这个数组的某个位置上（对应一个 bucket）。如果多个 key 映射到同一个位置，就发生了冲突，那么就拉出一个 dictEntry 链表。**

```
// 哈希表节点 dictEntry
typedef struct dictEntry {
     void *key; // redis 的键
     union {
     void *val; // 存储了对应 string/set/list/hash/zset 的数据
uint64_t u64;
     int64_t s64;
     } v;
     struct dictEntry *next; //链表后续节点
}dictEntry;
```

**上述 dictEntry 的 value 最终指向了 redisObject 对象，我们来观察下其结构。**

```
typedef struct redisObject {
     unsigned type:4; // 类型 ，比如 string，set 等，才能确定是哪种数据结构使用什么 API 操作
     unsigned encoding:4; // encoding 表示 ptr 指向的具体数据结构,这个对象使用什么数据结构实现
     unsigned lru:REDIS_LRU_BITS; // 对象最后一次被访问的时
     int refcount; // 引用计数
     void *ptr; // 指向底层数据结构的指针
robj;
```

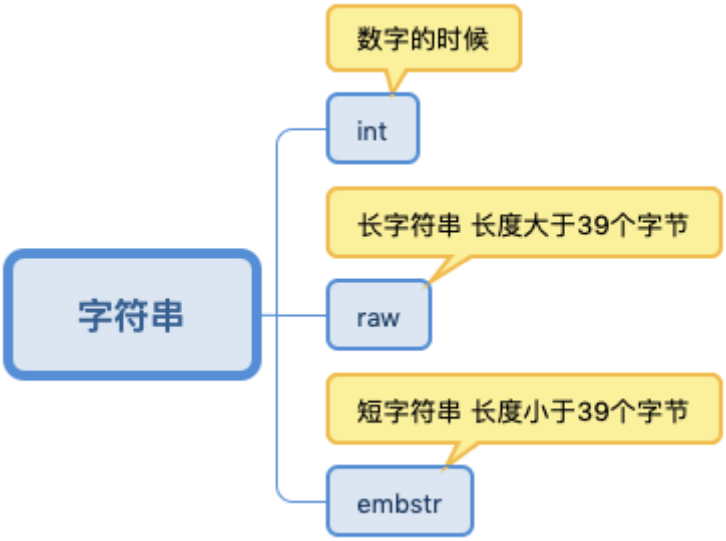
### 4.1.1 redis 提供的数据类型

| 数据类型 | 使用场景 |
| --- | --- |
| String | 存储 key-value 键值对，注意 redis 中 String 可修改。统计在线人数；也可以存储**视频、图片**等 |
| hash | 购物车： hset [key] [field] [value] 命令， 存放键值对，一般可以用来存某个对象的基本属性信息，例如，用户信息，商品信息等 |
| set | **全局去重**，JVM自带的 set 不适合分布式集群情况 |
| zset | **排行榜**，比如微信运动排行榜 |

| | |
|---|---|
| list | 分页功能，lrange 做基于 redis 的分页功能，性能很好；**模仿一个消息队列** |

下面以 3.0 为例，介绍各个数据类型

#### 4.1.1.1 String




##### 4.1.1.1.1 内部实现

不同于原生 C 语言的字符串，Redis 中的 String 类型底层的数据结构是 SDS（简单动态字符串），结构体定义如下：

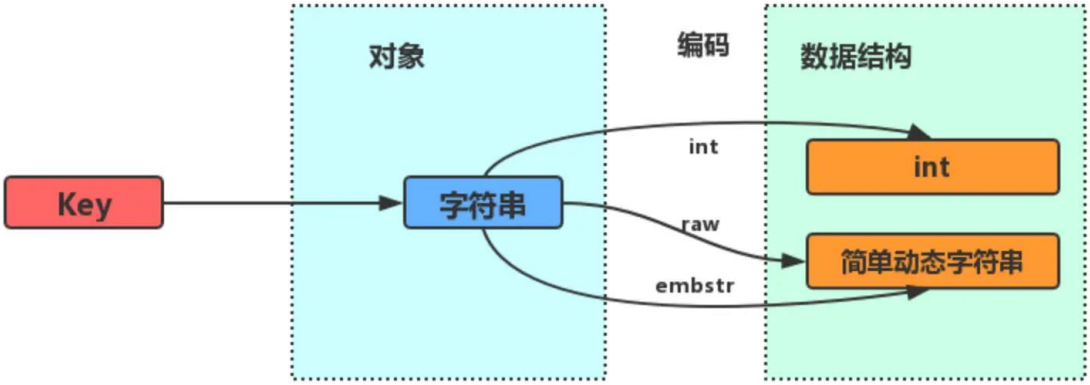
```
struct sdshdr {
    //buf 中已经使用的长度
    //等于 SDS 所保存字符串的长度
    unsigned int len;
    //buf 中未使用的长度
    unsigned int free;
    //字节数组，用于保存字符串
    char buf[];          、
};
```

相比于原生 C 语言的字符串，SDS 有如下优势：

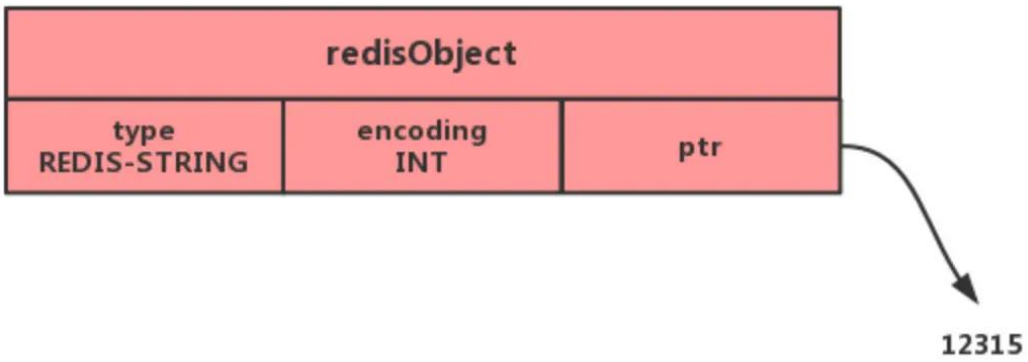
- **SDS 不仅可以保存文本数据，还可以保存二进制数据**。因为 SDS 使用 len 属性的值而不是空字符来判断字符串是否结束，并且 SDS 的所有 API 都会以处理二进制的方式来处理 SDS 存放在 buf[] 数组里的数据。所以 SDS 不光能存放文本数据，而且能保存图片、音频、视频、压缩文件这样的二进制数据。

- **SDS 获取字符串长度的时间复杂度是 O(1)**。因为 C 语言的字符串并不记录自身长度，所以获取长度的复杂度为 O(n)；而 SDS 结构里用 len 属性记录了字符串长度，所以复杂度为 O(1)。
- **Redis 的 SDS API 是安全的，拼接字符串不会造成缓冲区溢出**。因为 SDS 在拼接字符串之前会检查 SDS 空间是否满足要求，如果空间不够会自动扩容，所以不会导致缓冲区溢出的问题。

字符串对象的内部编码（encoding）有 3 种：**int、raw 和 embstr**。

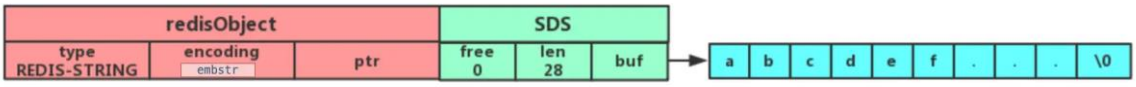



1. **int 编码**：字符串对象**保存的是整数值**，并且整数可以用 long 类型表示的话，会将 redisobject 中 encoding 字段设置为 int，并且将整数值保存在 redisobject 的 ptr 字段里（将 void*转换成 long）：

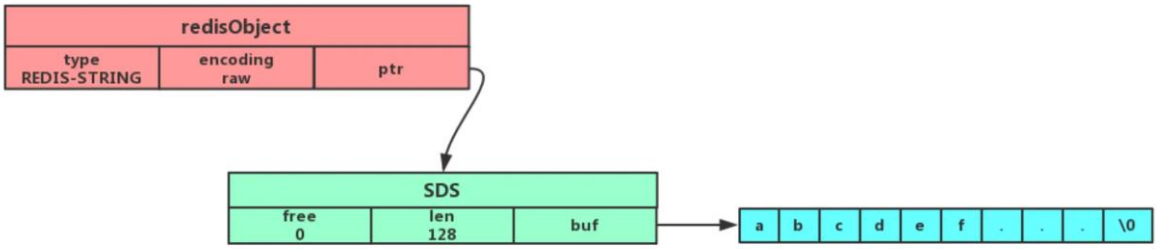



2. **embstr 编码**：**专门用于保存短字符串的一种优化编码方式**，字符串对象保存的是**字符串并且字符串长度小于等于 32 字节**（边界在 redis 版本不一样，redis 2.+是 32 字节，redis 3.0-

4.0 是 39 字节，redis 5.0 是 44 字节），会使用 SDS 保存这个字符串，并将对象的 encoding 置为 embstr，embstr 编码是专门用于保存短字符串的一种优化编码方式：




3. **raw 编码**：字符串对象保存的是一个字符串并且字符串长度大于 32 字节（redis 2.+），会用 SDS 保存这个字符串，并将对象的 encoding 置为 raw。




可以看到 embstr 和 raw 编码都会使用 SDS 来保存值，但不同之处在于 embstr 会通过一次内存分配函数来分配一块连续的内存空间来保存 redisObject 和 SDS，而 raw 编码会通过调用两次内存分配函数来分别分配两块空间来保存 redisObject 和 SDS。Redis 这样做会有很多好处：

- embstr 编码将创建字符串对象所需的内存分配次数从 raw 编码的两次降低为一次；
- 释放 embstr 编码的字符串对象同样只需要调用一次内存释放函数；
- 因为 embstr 编码的字符串对象的所有数据都保存在一块连续的内存里面可以更好的利用 CPU 缓存提升性能。

embstr 的缺点：

- 如果字符串的长度增加需要重新分配内存时，整个 redisObject 和 sds 都需要重新分配空间，所以 **embstr 编码的字符串对象实际上是只读的**，redis 没有为 embstr 编码的字符串对象编写任何相应的修改程序。当我们对 embstr 编码的字符串对象执行任何修改命令（例如 append）时，程序会先将对象的编码从 embstr 转换成 raw，然后再执行修改命令。

#### 4.1.1.1.2 常用指令

基本操作

```
# 设置 key-value 类型的值
> SET name lin
OK
# 根据 key 获得对应的 value
> GET name
"lin"
# 判断某个 key 是否存在
> EXISTS name
(integer) 1
# 返回 key 所储存的字符串值的长度
> STRLEN name
(integer) 3
# 删除某个 key 对应的值
> DEL name
(integer) 1
```

批量操作：

```
# 批量设置 key-value 类型的值
> MSET key1 value1 key2 value2
OK
# 批量获取多个 key 对应的 value
> MGET key1 key2
1) "value1"
2) "value2"
```

计数器（字符串为整数时可以使用）

```
# 设置 key-value 类型的值
> SET number 0
OK
# 将 key 中储存的数字值增一
> INCR number
(integer) 1
# 将 key 中存储的数字值加 10
> INCRBY number 10
(integer) 11
# 将 key 中储存的数字值减一
> DECR number
(integer) 10
# 将 key 中存储的数字值键 10
> DECRBY number 10
(integer) 0
```

过期（默认为永不过期）：

```
# 设置 key 在 60 秒后过期（该方法是针对已经存在的 key 设置过期时间）
> EXPIRE name  60
(integer) 1
# 查看数据还有多久过期
> TTL name
(integer) 51

#设置 key-value 类型的值，并设置该 key 的过期时间为 60 秒
> SET key  value EX 60
OK
> SETEX key  60 value
OK
```

不存在就插入：

```
# 不存在就插入（not exists）
>SETNX key value
(integer) 1
```

项目中实现的分布式锁：

```
# 如果不存在就插入并且设置该 key 的过期时间为 60 秒
>SET lock_key unique_value NX PX 60000
(integer) 1
```

- lock_key 就是 key 键；（在项目中一般是 appid+zoneid）
- unique_value 是客户端生成的唯一的标识；（我们不限制仅锁住的客户端删除锁，故项目中是 1）
- NX 代表只在 lock_key 不存在时，才对 lock_key 进行设置操作；
- PX 10000 表示设置 lock_key 的过期时间为 10s，这是为了避免客户端发生异常而无法释放锁。

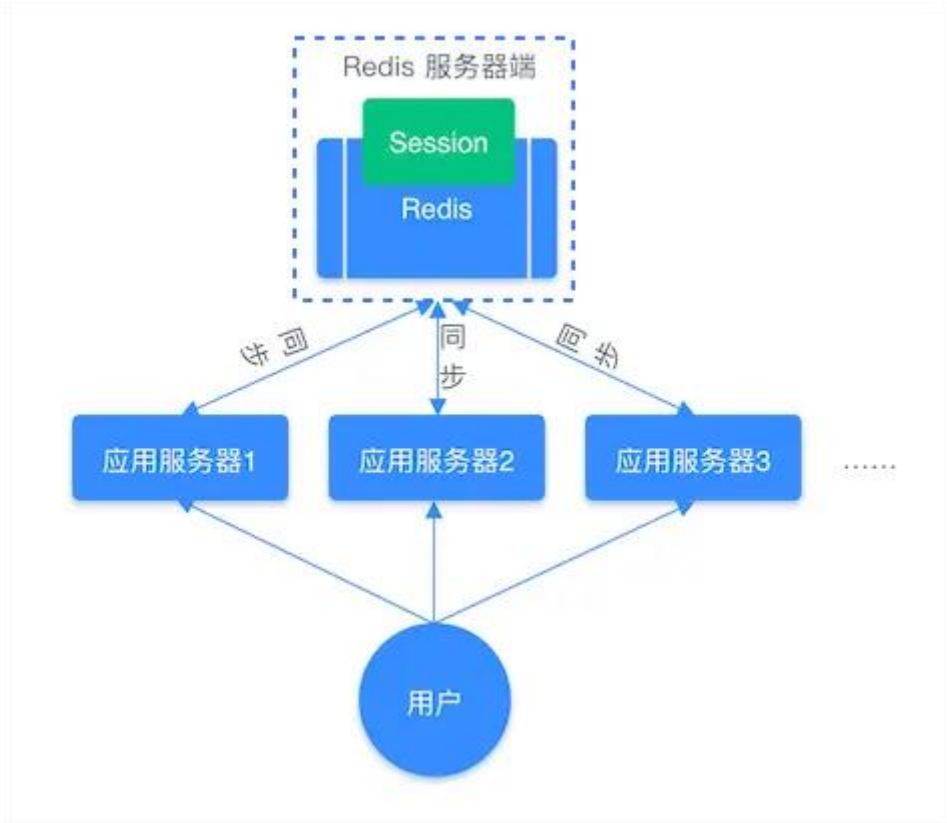
**4.1.1.1.3 应用场景**

1. 缓存对象：缓存个 json 对象，或者把对象的属性分离到每个 key 中，用 MSET 存，用 MGET 取

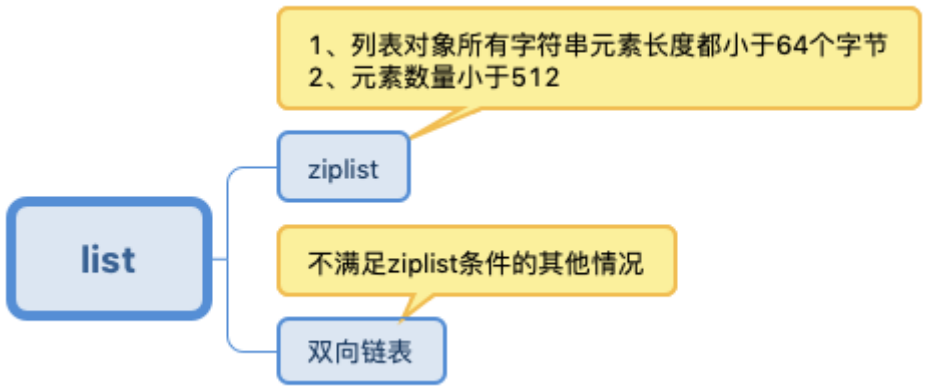
2. 常规计数：redis 单线程，执行命令过程中是原子的。String 数据类型适合计数场景，比如计算访问次数、点赞、转发、库存数量等等。

3. 分布式锁（项目中有用）

4. **共享 Session 信息：**可以借助 Redis 对这些 Session 信息进行统一的存储和管理，这样无论请求发送到那台服务器，服务器都会去同一个 Redis 获取相关的 Session 信息，这样就解决了分布式系统下 Session 存储的问题。




#### 4.1.1.2 List




List 列表是简单的字符串列表，**按照插入顺序排序**，可以从头部或尾部向 List 列表添加元素。

列表的最大长度为 2^32 - 1，也即每个列表支持超过 40 亿个元素。

##### 4.1.1.2.1 内部实现

List 类型的底层数据结构是由**双向链表或压缩列表**实现的：

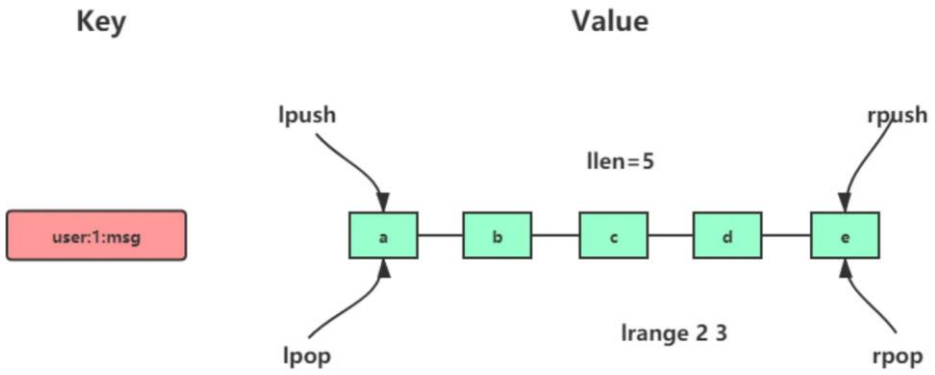
- 如果列表的元素个数小于 512 个（默认值，可由 list-max-ziplist-entries 配置），

列表每个元素的值都小于 64 字节（默认值，可由 list-max-ziplist-value 配置），Redis 会使用压缩列表作为 List 类型的底层数据结构；

- 如果列表的元素不满足上面的条件，Redis 会使用双向链表作为 List 类型的底层数据结构；

但是在 **Redis 3.2 版本之后，List 数据类型底层数据结构就只由 quicklist 实现了，替代了双向链表和压缩列表**。

#### 4.1.1.2.2 常用指令

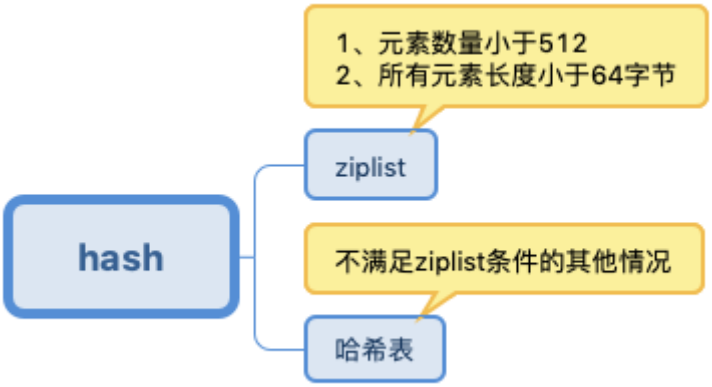


```
# 将一个或多个值 value 插入到 key 列表的表头(最左边)，最后的值在最前面
LPUSH key value [value ...]
# 将一个或多个值 value 插入到 key 列表的表尾(最右边)
RPUSH key value [value ...]
# 移除并返回 key 列表的头元素
LPOP key
# 移除并返回 key 列表的尾元素
RPOP key

# 返回列表 key 中指定区间内的元素，区间以偏移量 start 和 stop 指定，从 0 开始
LRANGE key start stop

# 从 key 列表表头弹出一个元素，没有就阻塞 timeout 秒，如果 timeout=0 则一直阻塞
BLPOP key [key ...] timeout
# 从 key 列表表尾弹出一个元素，没有就阻塞 timeout 秒，如果 timeout=0 则一直阻塞
BRPOP key [key ...] timeout
```

#### 4.1.1.2.3 应用场景

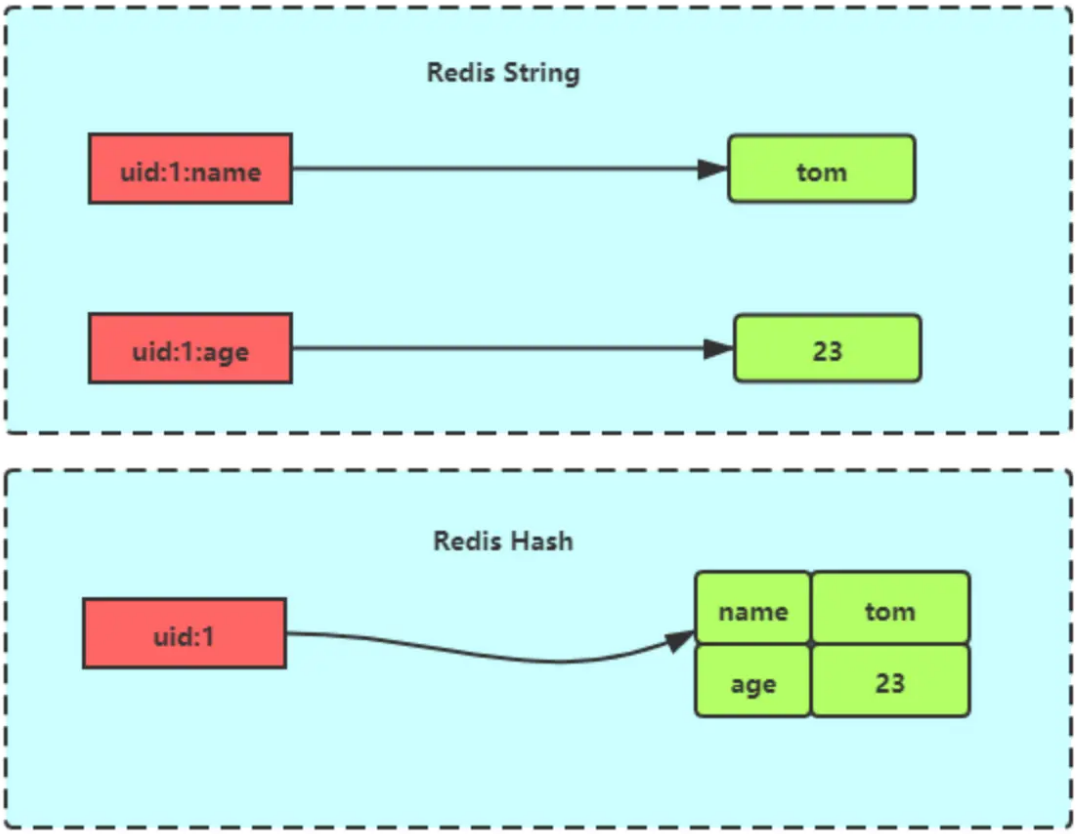
1. 消息队列

### 4.1.1.3 Hash




Hash 是一个键值对（key - value）集合，其中 value 的形式如： value=[{field1，value1}，...{fieldN，valueN}]。Hash 特别适合用于存储对象。

Hash 与 String 对象的区别如下图所示：




#### 4.1.1.3.1 内部实现

Hash 类型的底层数据结构是由压缩列表或哈希表实现的：

- 如果哈希类型元素个数小于 512 个（默认值，可由 hash-max-ziplist-entries 配置），所有值小于 64 字节（默认值，可由 hash-max-ziplist-value 配置）的话，Redis 会使用压缩列表作为 Hash 类型的底层数据结构；
- 如果哈希类型元素不满足上面条件，Redis 会使用哈希表作为 Hash 类型的 底层数据结构。

**在 Redis 7.0 中，压缩列表数据结构已经废弃了，交由 listpack 数据结构来实现了。**

#### 4.1.1.3.2 常用指令

```
# 存储一个哈希表 key 的键值
HSET key field value
# 获取哈希表 key 对应的 field 键值
HGET key field

# 在一个哈希表 key 中存储多个键值对
HMSET key field value [field value...]
# 批量获取哈希表 key 中多个 field 键值
HMGET key field [field ...]
# 删除哈希表 key 中的 field 键值
HDEL key field [field ...]

# 返回哈希表 key 中 field 的数量
HLEN key
# 返回哈希表 key 中所有的键值
HGETALL key

# 为哈希表 key 中 field 键的值加上增量 n
HINCRBY key field n
```

#### 4.1.1.3.3 应用场景

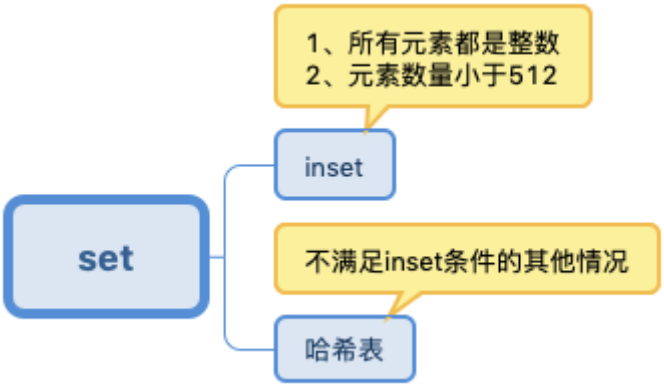
1. 缓存对象
2. 购物车：以用户 id 为 key，商品 id 为 field，商品数量为 value，恰好构成了购物车的 3 个要素。

涉及的命令如下：

- 添加商品：HSET cart:{用户 id} {商品 id} 1
- 添加数量：HINCRBY cart:{用户 id} {商品 id} 1
- 商品总数：HLEN cart:{用户 id}
- 删除商品：HDEL cart:{用户 id} {商品 id}
- 获取购物车所有商品：HGETALL cart:{用户 id}

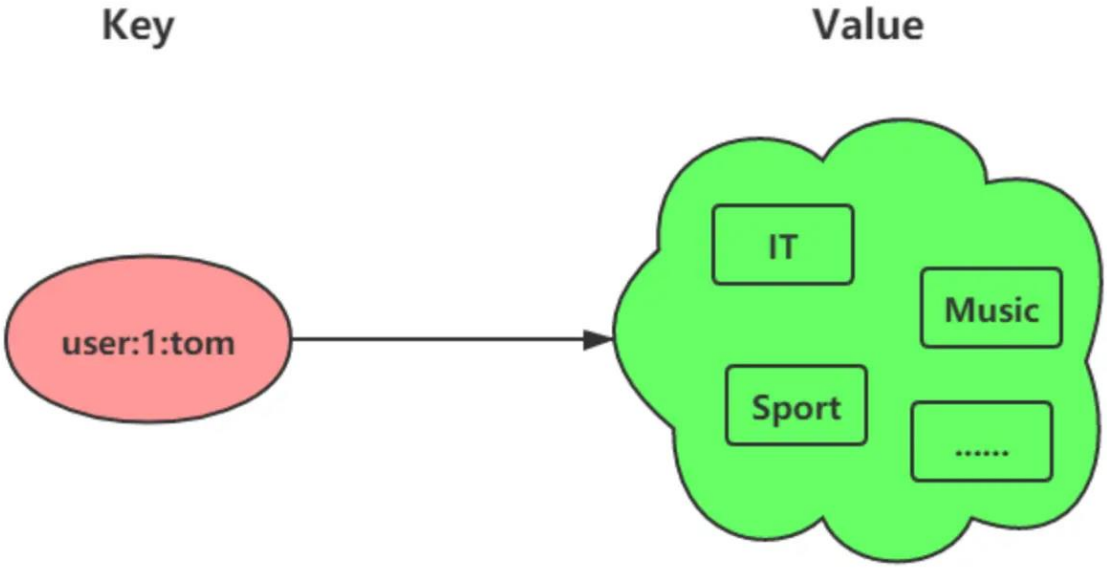
当前仅仅是将商品 ID 存储到了 Redis 中，在回显商品具体信息的时候，还需要拿着商品 id 查询一次数据库，获取完整的商品的信息。

#### 4.1.1.4 Set




Set 类型是一个无序并唯一的键值集合，它的存储顺序不会按照插入的先后顺序进行存储。

一个集合最多可以存储 2^32-1 个元素。概念和数学中个的集合基本类似，可以交集，并集，差集等等，所以 Set 类型除了支持集合内的增删改查，同时还支持多个集合取交集、并集、差集。



#### 4.1.1.4.1 内部实现

Set 类型的底层数据结构是由**哈希表或整数集合**实现的：

- 如果集合中的元素都是整数且元素个数小于 512 （默认值，set-maxintset-entries 配置）个，Redis 会使用**整数集合**作为 Set 类型的底层数据结构；
- 如果集合中的元素不满足上面条件，则 Redis 使用**哈希表**作为 Set 类型的底层数据结构。

#### 4.1.1.4.2 常用指令

Set 常用操作：

```
# 往集合 key 中存入元素，元素存在则忽略，若 key 不存在则新建
SADD key member [member ...]
# 从集合 key 中删除元素
SREM key member [member ...]
# 获取集合 key 中所有元素
SMEMBERS key
# 获取集合 key 中的元素个数
SCARD key

# 判断 member 元素是否存在于集合 key 中
SISMEMBER key member

# 从集合 key 中随机选出 count 个元素，元素不从 key 中删除
SRANDMEMBER key [count]
# 从集合 key 中随机选出 count 个元素，元素从 key 中删除
SPOP key [count]
```

Set 运算操作：

```
# 交集运算
SINTER key [key ...]
# 将交集结果存入新集合 destination 中
SINTERSTORE destination key [key ...]

# 并集运算
SUNION key [key ...]
# 将并集结果存入新集合 destination 中
SUNIONSTORE destination key [key ...]

# 差集运算
SDIFF key [key ...]
# 将差集结果存入新集合 destination 中
SDIFFSTORE destination key [key ...]
```

#### 4.1.1.4.3 应用场景

集合的主要几个特性，无序、不可重复、支持并交差等操作。

因此 Set 类型比较适合用来数据去重和保障数据的唯一性，还可以用来统计多个集合的交集、错集和并集等，当我们存储的数据是无序并且需要去重的情况下，比较适合使用集合类型进行存储。

但是要提醒你一下，这里有一个潜在的风险。**Set 的差集、并集和交集的计算复杂度较高，在数据量较大的情况下，如果直接执行这些计算，会导致 Redis 实例阻塞**。

在主从集群中，为了避免主库因为 Set 做聚合计算（交集、差集、并集）时导致主库被阻塞，我们可以选择一个从库完成聚合统计，或者把数据返回给客户端，由客户端来完成聚合统计。

**点赞**

Set 类型可以保证一个用户只能点一个赞，这里举例子一个场景，key 是文章 id，value 是用户 id。

**共同关注**

Set 类型支持交集运算，所以可以用来计算共同关注的好友、公众号等。

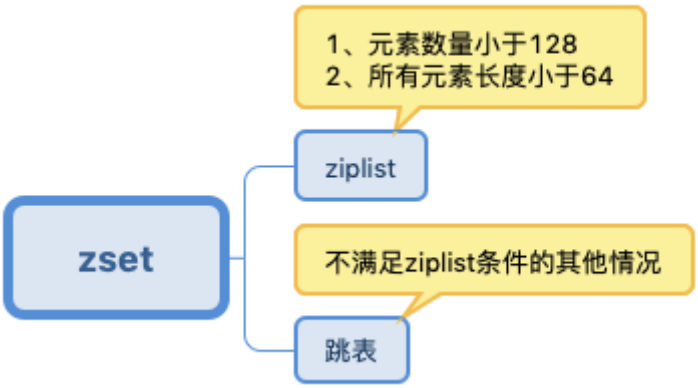
key 可以是用户 id，value 则是已关注的公众号的 id。

**抽奖活动**

存储某活动中中奖的用户名，Set 类型因为有去重功能，可以保证同一个用户不会中奖两次。

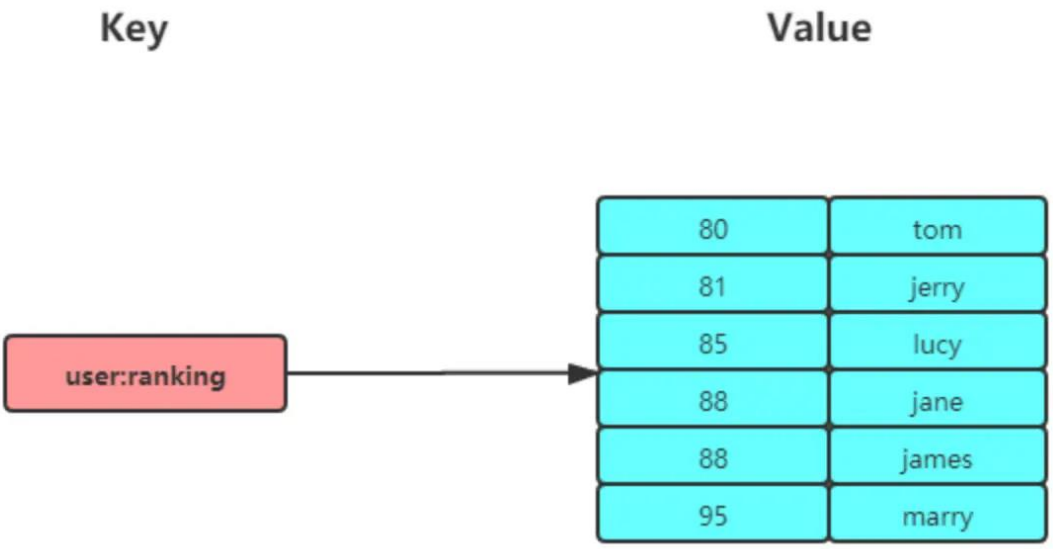
key 为抽奖活动名，value 为员工名称，把所有员工名称放入抽奖箱

#### 4.1.1.5 Zset




Zset 类型（有序集合类型）相比于 Set 类型**多了一个排序属性 score（分值）**，对于有序集合 ZSet 来说，每个存储元素相当于有两个值组成的，一个是有序集合的元素值，一个是排序值。

有序集合保留了集合不能有重复成员的特性（分值可以重复），但不同的是，有序集合中的元素可以排序。




#### 4.1.1.5.1 内部实现

Zset 类型的底层数据结构是由**压缩列表或跳表**实现的：

- 如果有序集合的元素个数小于 128 个，并且每个元素的值小于 64 字节时，Redis 会使用**压缩列表**作为 Zset 类型的底层数据结构；
- 如果有序集合的元素不满足上面的条件，Redis 会使用**跳表**作为 Zset 类型的底层数据结构；

**在 Redis 7.0 中，压缩列表数据结构已经废弃了，交由 listpack 数据结构来实现了。**

```
typedef struct zset {
    dict *dict;
    zskiplist *zsl;
} zset;
```

注意：上述所示 zset 结构体中既有跳表又有哈希表，zset 插入元素时会依次在跳表和哈希表中插入或更新数据，因为引入跳表，zset 可以支持范围查询，因为引入哈希表，zset 能以常数复杂度获取元素权重。不提哈希表是因为哈希表只用于获取元素权重，大部分操作是跳表实现的。

#### 4.1.1.5.2 常用命令

Zset 常用操作：

```
# 往有序集合 key 中加入带分值元素
ZADD key score member [[score member]...]
# 往有序集合 key 中删除元素
ZREM key member [member...]
# 返回有序集合 key 中元素 member 的分值
ZSCORE key member
# 返回有序集合 key 中元素个数
ZCARD key

# 为有序集合 key 中元素 member 的分值加上 increment
ZINCRBY key increment member

# 正序获取有序集合 key 从 start 下标到 stop 下标的元素
ZRANGE key start stop [WITHSCORES]
# 倒序获取有序集合 key 从 start 下标到 stop 下标的元素
ZREVRANGE key start stop [WITHSCORES]

# 返回有序集合中指定分数区间内的成员，分数由低到高排序。
ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]

# 返回指定成员区间内的成员，按字典正序排列，分数必须相同。
ZRANGEBYLEX key min max [LIMIT offset count]
# 返回指定成员区间内的成员，按字典倒序排列，分数必须相同
ZREVRANGEBYLEX key max min [LIMIT offset count]
```

Zset 运算操作（相比于 Set 类操作，Zset 类型没有支持差集运算）

```
# 并集计算(相同元素分值相加)，numberkeys 一共多少个 key，WEIGHTS 每个 key 对应的分值乘积
ZUNIONSTORE destkey numberkeys key [key...]
# 交集计算(相同元素分值相加)，numberkeys 一共多少个 key，WEIGHTS 每个 key 对应的分值乘积
ZINTERSTORE destkey numberkeys key [key...]
```

#### 4.1.1.5.3 应用场景

Zset 类型（Sorted Set，有序集合） 可以根据元素的权重来排序，我们可以自己来决定每个元素的权重值。比如说，我们可以根据元素插入 Sorted Set 的时间确定权重值，先插入的元素权重小，后插入的元素权重大。

在面对需要展示最新列表、排行榜等场景时，如果数据更新频繁或者需要分页显示，可以优先考虑使用 Sorted Set。

**排行榜**

有序集合比较典型的使用场景就是排行榜。例如学生成绩的排名榜、游戏积分排行榜、视频播放排名、电商系统中商品的销量排名等。

**电话、姓名排序**

使用有序集合的 ZRANGEBYLEX 或 ZREVRANGEBYLEX 可以帮助我们实现电话号码或姓名的排序，我们以 ZRANGEBYLEX （返回指定成员区间内的成员，按 key 正序排列，分数必须相同）为例。

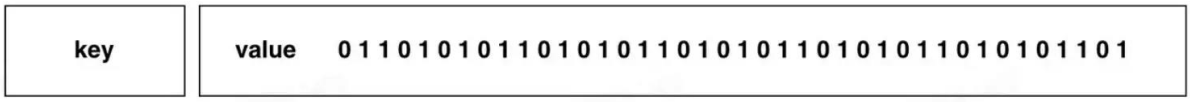
**注意：不要在分数不一致的 SortSet 集合中去使用 ZRANGEBYLEX 和 ZREVRANGEBYLEX 指令，因为获取的结果会不准确。**

### 4.1.1.6 BitMap

Bitmap，即位图，是一串连续的二进制数组（0 和 1），可以通过偏移量（offset）定位元素。BitMap 通过最小的单位 bit 来进行 0|1 的设置，表示某个元素的值或者状态，时间复杂度为 O(1)。

由于 bit 是计算机中最小的单位，使用它进行储存将非常节省空间，特别适合一些数据量大且使用二值统计的场景。

#### 4.1.1.6.1 内部实现




Bitmap 本身是用 String 类型作为底层数据结构实现的一种统计二值状态的数据类型。

String 类型是会保存为二进制的字节数组，所以，Redis 就把字节数组的每个 bit 位利用起来，用来表示一个元素的二值状态，你可以把 Bitmap 看作是一个 bit 数组。

#### 4.1.1.6.2 常用命令

bitmap 基本操作：

```
# 设置值，其中 value 只能是 0 和 1
SETBIT key offset value

# 获取值
GETBIT key offset

# 获取指定范围内值为 1 的个数
# start 和 end 以字节为单位
BITCOUNT key start end
```

bitmap 运算操作：

```
# BitMap 间的运算
# operations 位移操作符，枚举值
  AND 与运算 &
  OR 或运算 |
  XOR 异或 ^
  NOT 取反 ~
# result 计算的结果，会存储在该 key 中
# key1 … keyn 参与运算的 key，可以有多个，空格分割，not 运算只能一个 key
# 当 BITOP 处理不同长度的字符串时，较短的那个字符串所缺少的部分会被看作 0。
返回值是保存到 destkey 的字符串的长度（以字节 byte 为单位），和输入 key 中最
长的字符串长度相等。
BITOP [operations] [result] [key1] [keyn…]

# 返回指定 key 中第一次出现指定 value(0/1)的位置
BITPOS [key] [value]
```

#### 4.1.1.6.3 应用场景

Bitmap 类型非常适合二值状态统计的场景，这里的二值状态就是指集合元素的取值就只有 0 和 1 两种，在记录海量数据时，Bitmap 能够有效地节省内存空间。

**签到统计**

在签到打卡的场景中，我们只用记录签到（1）或未签到（0），所以它就是非常典型的二值状态。

签到统计时，每个用户一天的签到用 1 个 bit 位就能表示，一个月（假设是 31 天）的签到情况用 31 个 bit 位就可以，而一年的签到也只需要用 365 个 bit 位，根本不用太复杂的集合类型。

**判断用户登录态**

Bitmap 提供了 GETBIT、SETBIT 操作，通过一个偏移值 offset 对 bit 数组的 offset 位置的 bit 位进行读写操作，需要注意的是 offset 从 0 开始。

只需要一个 key = login_status 表示存储用户登陆状态集合数据， 将用户 ID 作为

offset，在线就设置为 1，下线设置 0。通过 GETBIT 判断对应的用户是否在线。 5000 万用户只需要 6 MB 的空间。

**连续签到用户总数**

如何统计出这连续 7 天连续打卡用户总数呢？

我们把每天的日期作为 Bitmap 的 key，userId 作为 offset，若是打卡则将 offset 位置的 bit 设置成 1。

key 对应的集合的每个 bit 位的数据则是一个用户在该日期的打卡记录。

一共有 7 个这样的 Bitmap，如果我们能对这 7 个 Bitmap 的对应的 bit 位做 『与』运算。同样的 UserID offset 都是一样的，当一个 userID 在 7 个 Bitmap 对应对应的 offset 位置的 bit = 1 就说明该用户 7 天连续打卡。

结果保存到一个新 Bitmap 中，我们再通过 BITCOUNT 统计 bit = 1 的个数便得到了连续打卡 7 天的用户总数了。

#### 4.1.1.7 HyperLogLog

Redis HyperLogLog 是 Redis 2.8.9 版本新增的数据类型，是一种用于「统计基数」的数据集合类型，基数统计就是指统计一个集合中不重复的元素个数。但要注意，HyperLogLog 是统计规则是基于概率完成的，不是非常准确，标准误算率是 0.81%。所以，简单来说 HyperLogLog **提供不精确的去重计数**。

HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的内存空间总是固定的、并且是很小的。

在 Redis 里面，**每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基数**，和元素越多就越耗费内存的 Set 和 Hash 类型相比，HyperLogLog 就非常节省空间。

#### 4.1.1.7.1 内部实现

略过，不会考这个东西的

#### 4.1.1.7.2 常用命令

HyperLogLog 命令很少，就三个。

```
# 添加指定元素到 HyperLogLog 中
PFADD key element [element ...]

# 返回给定 HyperLogLog 的基数估算值。
PFCOUNT key [key ...]

# 将多个 HyperLogLog 合并为一个 HyperLogLog
PFMERGE destkey sourcekey [sourcekey ...]
```

#### 4.1.1.7.3 应用场景

**百万级网页 UV 计数**

Redis HyperLogLog 优势在于只需要花费 12 KB 内存，就可以计算接近 2^64 个元素的基数，和元素越多就越耗费内存的 Set 和 Hash 类型相比，HyperLogLog 就非常节省空间。

所以，非常适合统计百万级以上的网页 UV 的场景。

在统计 UV 时，你可以用 PFADD 命令（用于向 HyperLogLog 中添加新元素）把访问页面的每个用户都添加到 HyperLogLog 中。

```
PFADD page1:uv user1 user2 user3 user4 user5
```

接下来，就可以用 PFCOUNT 命令直接获得 page1 的 UV 值了，这个命令的作用就是返回 HyperLogLog 的统计结果。

```
PFCOUNT page1:uv
```

不过，有一点需要你注意一下，HyperLogLog 的统计规则是基于概率完成的，所以它

给出的统计结果是有一定误差的，标准误算率是 0.81%。

这也就意味着，你使用 HyperLogLog 统计的 UV 是 100 万，但实际的 UV 可能是 101 万。虽然误差率不算大，但是，如果你需要精确统计结果的话，最好还是继续用 Set 或 Hash 类型。

#### 4.1.1.8 GEO

Redis GEO 是 Redis 3.2 版本新增的数据类型，主要用于存储地理位置信息，并对存储的信息进行操作。GEO 非常适合应用在基于位置信息服务（Location-Based Service，LBS）的应用。LBS 应用访问的数据是和人或物关联的一组经纬度信息，而且要能查询相邻的经纬度范围，GEO 就非常适合应用在 LBS 服务的场景中。

##### 4.1.1.8.1 内部实现

GEO 本身并没有设计新的底层数据结构，而是**直接使用了 Sorted Set 集合类型**。

**GEO 类型使用 GeoHash 编码方法实现了经纬度到 Sorted Set 中元素权重分数的转换**，这其中的两个关键机制就是「对二维地图做区间划分」和「对区间进行编码」。一组经纬度落在某个区间后，就用区间的编码值来表示，并把编码值作为 Sorted Set 元素的权重分数。

这样一来，我们就可以把经纬度保存到 Sorted Set 中，利用 Sorted Set 提供的“按权重进行有序范围查找”的特性，实现 LBS 服务中频繁使用的“搜索附近”的需求。

#### 4.1.1.8.2 常用命令

```
# 存储指定的地理空间位置，可以将一个或多个经度(longitude)、纬度(latitude)、位置名称(member)添加到指定的 key 中。
GEOADD key longitude latitude member [longitude latitude member ...]

# 从给定的 key 里返回所有指定名称(member)的位置（经度和纬度），不存在的返回 nil。
GEOPOS key member [member ...]

# 返回两个给定位置之间的距离。
GEODIST key member1 member2 [m|km|ft|mi]

# 根据用户给定的经纬度坐标来获取指定范围内的地理位置集合。
GEORADIUS key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count] [ASC|DESC] [STORE key] [STOREDIST key]
```

#### 4.1.1.8.3 应用场景

**滴滴叫车**

这里以滴滴叫车的场景为例，介绍下具体如何使用 GEO 命令：GEOADD 和 GEORADIUS 这两个命令。

假设车辆 ID 是 33，经纬度位置是（116.034579，39.030452），我们可以用一个 GEO 集合保存所有车辆的经纬度，集合 key 是 cars:locations。

执行下面的这个命令，就可以把 ID 号为 33 的车辆的当前经纬度位置存入 GEO 集合中：

```
GEOADD cars:locations 116.034579 39.030452 33
```

当用户想要寻找自己附近的网约车时，LBS 应用就可以使用 GEORADIUS 命令。

例如，LBS 应用执行下面的命令时，Redis 会根据输入的用户的经纬度信息

（116.054579，39.030452 ），查找以这个经纬度为中心的 5 公里内的车辆信息，并返回给 LBS 应用。

```
GEORADIUS cars:locations 116.054579 39.030452 5 km ASC COUNT 10
```

## 4.1.2 redis 底层数据结构

### 4.1.2.1 SDS

#### 4.1.2.1.1 C 语言字符串的缺陷

1. 取长度时需要遍历字符串，O(N)的操作
2. 字符串中不能含有“\0”字符，否则会被误认为是字符串结尾，使得 C 字符串只能保存文本数据，不能保存图片、音频、视频等二进制数据
3. 不会记录自身缓冲区大小，可能会发生缓冲区溢出导致程序运行终止

#### 4.1.2.1.2 SDS 结构设计

- **len，记录了字符串长度**。这样获取字符串长度的时候，只需要返回这个成员变量值就行，时间复杂度只需要 O（1）。

- **alloc，分配给字符数组的空间长度**。这样在修改字符串的时候，可以通过 alloc - len 计算出剩余的空间大小，可以用来判断空间是否满足修改需求，如果不满足的话，就会自动将 SDS 的空间扩展至执行修改所需的大小，然后才执行实际的修改操作，所以使用 SDS 既不需要手动修改 SDS 的空间大小，也不会出现前面所说的缓冲区溢出的问题。
- **flags，用来表示不同类型的 SDS**。一共设计了 5 种类型，分别是 sdshdr5、sdshdr8、sdshdr16、sdshdr32 和 sdshdr64，后面在说明区别之处。
- **buf[]，字符数组，用来保存实际数据**。不仅可以保存字符串，也可以保存二进制数据。

总的来说，SDS 通过增加三个元数据：len、alloc、flags 来解决 C 语言字符串的缺陷

O（1）复杂度获取字符串长度

通过 len 这个成员变量直接返回字符串长度，复杂度只有 O（1）

二进制安全

因为有 len 这个成员变量，SDS 不需要通过“\0”来判断字符串是否结尾，所以 SDS 可以保存任意格式的二进制数据。

不会发生缓冲区溢出

SDS 中引入了 len 和 alloc 成员变量，可以通过 alloc-len 计算出剩余可用空间的大小，**当判断出缓冲区大小不够用时，Redis 会自动将扩大 SDS 的空间大小。**

扩容规则：

- 如果所需的 sds 长度**小于 1 MB**，那么最后的扩容是按照**翻倍扩容**来执行的，即 2 倍的 newlen
- 如果所需的 sds 长度**超过 1 MB**，那么最后的扩容长度应该是 newlen **+ 1MB**

节省内存空间

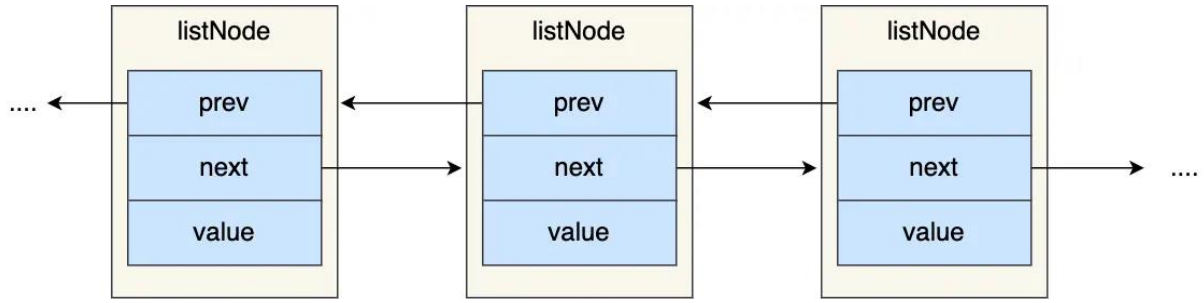
通过 flags 成员变量设置了 5 种类型，分别是 sdshdr5、sdshdr8、sdshdr16、sdshdr32 和 sdshdr64，**区别就在于，它们数据结构中的 len 和 alloc 成员变量的数据类型不同**。除了设计不同类型的结构体，Redis 在编程上还**使用了专门的编译优化来节省内存空间**，取消了字节对齐，按照实际占用来分配内存

#### 4.1.2.2 链表

##### 4.1.2.2.1 链表节点结构设计

```
typedef struct listNode {
    //前置节点
    struct listNode *prev;
    //后置节点
    struct listNode *next;
    //节点的值
    void *value;
} listNode;
```

可以看出是一个双向链表。



#### 4.1.2.2.2 链表结构设计

```
typedef struct list {
    //链表头节点
    listNode *head;
    //链表尾节点
    listNode *tail;
    //节点值复制函数
    void *(*dup)(void *ptr);
    //节点值释放函数
    void (*free)(void *ptr);
    //节点值比较函数
    int (*match)(void *ptr, void *key);
    //链表节点数量
    unsigned long len;
} list;
```

- 链表头指针 head、链表尾指针 tail、链表节点数量 len
- 自定义实现的复制函数 dup、节点释放函数 free、节点比较函数 match

Redis 链表优点如下：

- listnode 结构体有 pre 和 next，**获取某节点前置或后置节点的时间复杂度是 O（1）**
- list 结构体提供了 head 和 tail，**获取表头节点和表尾节点的时间复杂度是 O（1）**
- list 结构体提供了 len，**获取链表中节点数量的时间复杂度是 O（1）**
- 链表节点使用指针保存节点值，通过 list 结构中的 dup、free、match 函数指针为该节点类型设置特定的函数，因此**链表节点可以保存各种不同类型的值**。

Redis 链表缺点如下：

- 链表节点间内存不连续，**无法很好利用 CPU 缓存**。
- 每个链表节点都要分配链表节点的结构头，**内存开销较大**。

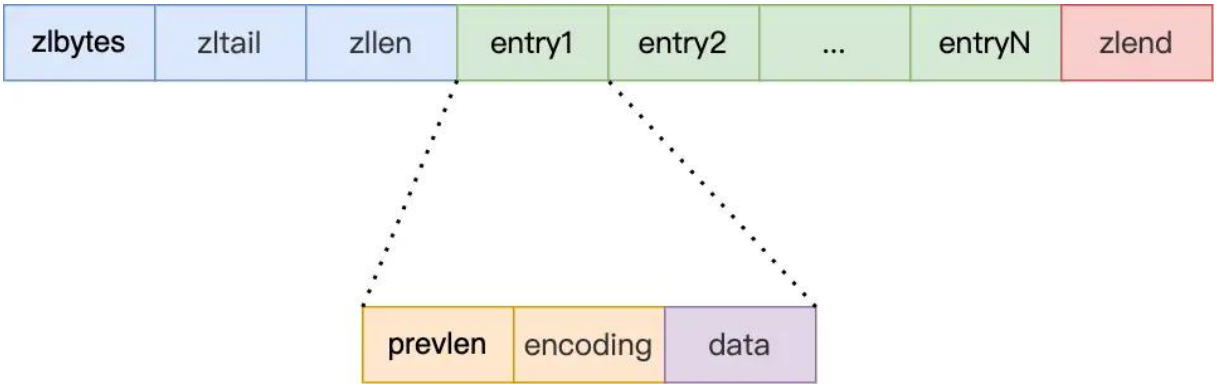
#### 4.1.2.3 压缩列表

#### 4.1.2.3.1 压缩列表结构设计

**由连续内存块组成的顺序性数据结构**

- ***zlbytes***，记录整个压缩列表占用对内存字节数；
- ***zltail***，记录压缩列表「尾部」节点距离起始地址由多少字节，也就是列表尾的偏移量；
- ***zllen***，记录压缩列表包含的节点数量；
- ***zlend***，标记压缩列表的结束点，固定值 0xFF（十进制 255）

第一个元素和最后一个元素可以 O（1）查找到，剩余元素只能遍历，所以压缩列表不适合保存过多元素。




压缩列表节点包含三部分内容：

- ***prevlen***，记录了「前一个节点」的长度，目的是为了实现从后向前遍历；
- ***encoding***，记录了当前节点实际数据的「类型和长度」，类型主要有两种：字符串和整数。
- ***data***，记录了当前节点的实际数据，类型和长度都由 encoding 决定；

其中，**prelen 和 encoding 会根据压缩列表中存储的数据类型和数据大小使用不同空间大小**的 prelen 和 encoding；这种根据数据大小和类型进行不同空间大小分配的设计思想，正是 Redis 为了节省内存采用的。

压缩列表缺点：

1. 除了第一个元素和最后一个元素，其他元素查找需要遍历
2. 可能会引发连锁更新（比如插入或修改了一条比较长的数据，那可能后续的 entry 头部也要跟着修改）

#### 4.1.2.4 哈希表

Redis 哈希表能以 O（1）的复杂度查询数据，并且采用拉链法来解决哈希冲突

##### 4.1.2.4.1 哈希表结构设计

Redis 中哈希表结构体定义是同一个，都是有两个哈希数组，这里不做赘述

##### 4.1.2.4.2 渐进式 rehash

上述所说，Redis 定义了**两个哈希表（ht[2]）**

普通的 rehash 过程可能如下：

- 给 h[1]分配空间，一般会比 h[0] 大 2 倍（第一个比 h[0]大的 2 的 n 次方，原因是可以将 key 的 hash 值和 size-1 做位与操作即可得到数组中的下标）；
- 将「哈希表 1 」的数据迁移到「哈希表 2」 中；
- 迁移完成后，「哈希表 1 」的空间会被释放，并把「哈希表 2」 设置为「哈希表 1」，然后在「哈希表 2」 新创建一个空白的哈希表，为下次 rehash 做准备。

这样会有一个明显的问题，当「哈希表 1」数据量非常大时，可能在迁移过程中涉及大量的数据拷贝，导致 Redis 无法正常工作，所以 Redis 采用了渐进式 rehash

渐进式 rehash 过程：

- 给 ht[1]分配空间，一般会比 ht[0] 大 2 倍（扩容时：第一个比 h[0]大的 2 的 n 次方，原因是可以将 key 的 hash 值和 size-1 做位与操作即可得到数组中的下标；缩容时：第一个比当前 key 数量大的 2 的 n 次方，最小容量是 4）；
- 修改 dict 结构体中的 rehashidx 字段，设置为 0，表示 rehash 工作正式开始
- **在 rehash 进行期间，每次哈希表元素进行新增、删除、查找或者更新操作时，**

**Redis 除了会执行对应的操作之外，还会顺带将 ht[0]中索引位置上的所有 key-value 迁移到 ht[1]上**；当 rehash 工作完成后，程序将 rehashinx 的值增加一

- 随着处理客户端发起的哈希表操作请求数量越多，最终在某个时间点会把 ht[0]的所有 key-value 迁移到 ht[1]，从而完成 rehash 操作。同时将 rehashidx 的值置为-1，表示 rehash 操作已完成。
- 如果服务器长时间空闲，redis 会在很长时间内都一直使用两个哈希表，所以在 redis 周期函数中，如果发现有字典正在进行渐进式 rehash，那么就会花费 **1 毫秒**的时间，帮助一起进行渐进式 rehash 操作。

注意：在 rehash 的过程中，如果是新增一个 key-value，则会被保存在 ht[1]中，ht[0]不会有任何添加操作；在删除、查找或更新操作时，会在两个哈希表中进行，并顺便将该索引位置上的所有 key-value 迁移到目标哈希表中，比如查找时会先在 h[0]中查找，没找到的话再继续在 h[1]上查找

渐进式 rehash 触发过程：

触发条件和负载因子有关：

| 负载因子 = 哈希表已保存节点数量 / 哈希表大小 |
| --- |

扩容时主要有两个条件：

- **当负载因子大于等于 1，并且 Redis 没有在执行 bgsave 命令或者 bgrewiteaof 命令，也就是没有执行 RDB 快照或没有进行 AOF 重写的时候，就会进行 rehash 操作。**
- **当负载因子大于等于 5 时，此时说明哈希冲突非常严重了，不管有没有有在执行 RDB 快照或 AOF 重写，都会强制进行 rehash 操作。**

缩容时主要有一个条件：

- **当负载因子小于 0.1，并且 Redis 没有执行 bgsave 或者 bgrewriteaof 命令时会执行 rehash**

渐进式 rehash 的优缺点：

- 优点：避免 redis 阻塞；
- 缺点：rehash 需要分配一个新的 hash 表，会使内存暴增，使得大量 key 被驱逐

#### 4.1.2.5 整数集合

整数集合是 Set 对象的底层实现之一。当一个 Set 对象只包含整数值元素，并且元素数量不大时，就会使用整数集这个数据结构作为底层实现。

##### 4.1.2.5.1 结构设计

```
typedef struct intset {
    //编码方式
    uint32_t encoding;
    //集合包含的元素数量
    uint32_t length;
    //保存元素的数组
    int8_t contents[];
} intset;
```

contents 保存真实数据，保存的数据类型取决于 encoding 字段，如果是 int16，则数组中所有元素的类型都是 int16，int32、int64 同理。

##### 4.1.2.5.2 升级操作

注意：整数集合只支持升级，不支持降级！!!

升级发生时机：

当新插入的元素类型（int32）比现在数组保存的元素类型（int16）都长时，需要先按照新的元素类型升级整数集合，再插入新数据。

升级过程：

**升级是在原本的数组上扩展空间，不会重新分配一个新类型的数组**。

简单来说，是将数组扩容，从后往前依次转换数组元素，并将数组元素放到正确的位上，同时会保持底层数组的有序性不变

整数集合升级的好处：

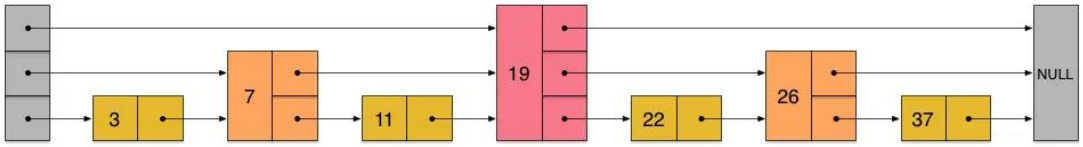
不在一开始就按照 int64 元素大小分配空间，而是采用升级的方式最大的好处就是会**节省内存资源**。

### 4.1.2.6 跳表

Redis 只有 Zset 采用了跳表，可以支持平均 O（logN）复杂度的节点查找。

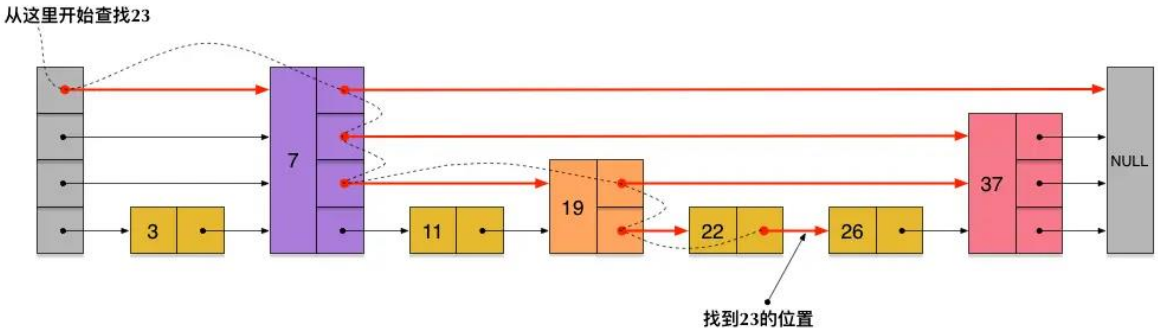
#### 4.1.2.6.1 结构设计

**跳表是在链表基础上改进过来的，实现了一种「多层」的有序链表。实际上就是在单链表上进行多级索引。**




跳表查询过程

从最高层开始，比较传入元素的 score 值和层级中元素的 score，根据类似于二分的思想，依次向下查找指定 score 值的节点。



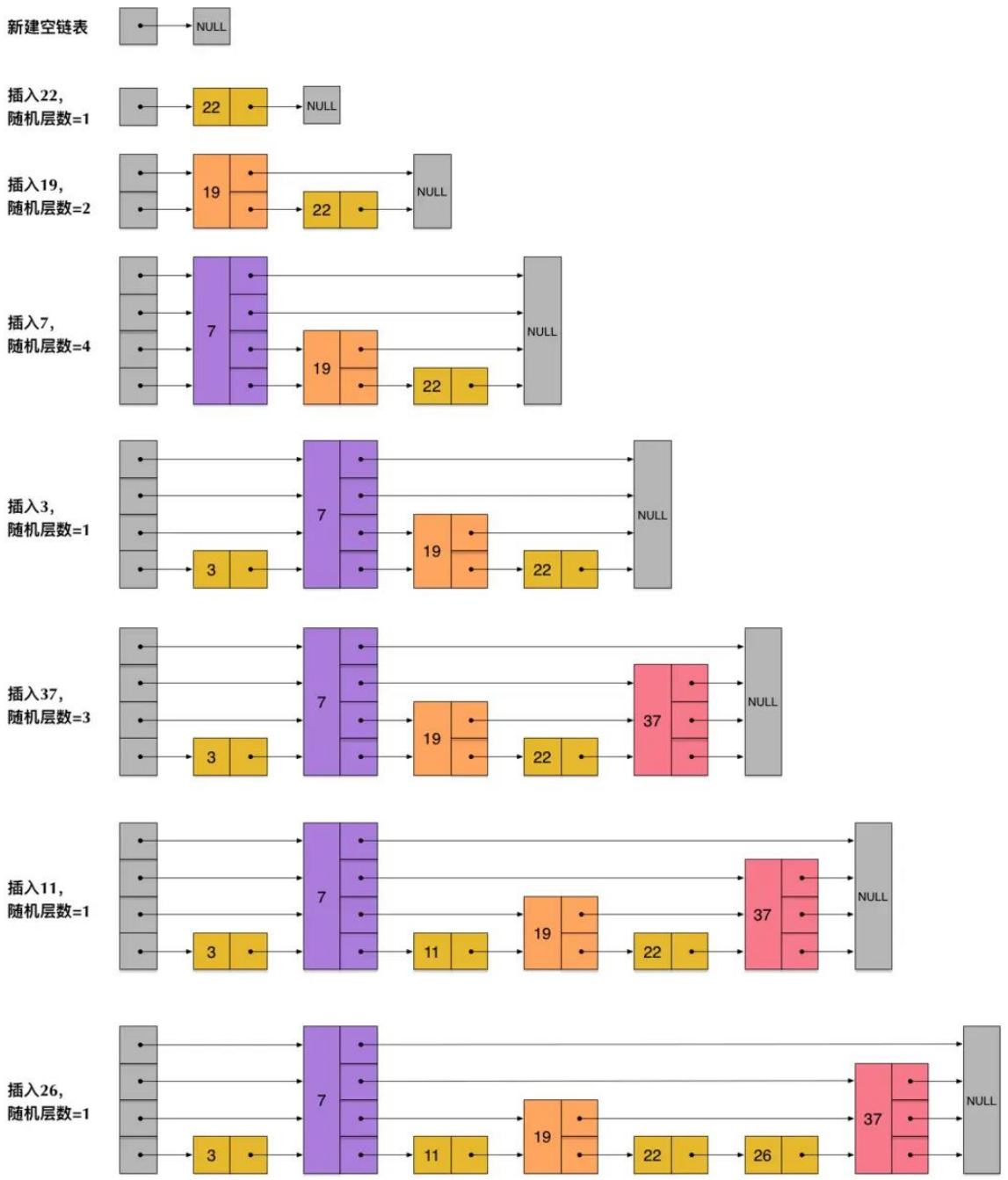

插入时链表节点的层级确定

Redis 为了避免上下两层出现严格 1:2 数量对应关系后，新插入节点会打乱这种关系，而需要把新插入节点后所以节点都进行调整。

采用的是**在创建节点的时候，随机生成每个节点的层数**，并没有严格维持相邻两层的节点数量比例为 2 : 1 的情况。

具体的做法是，**跳表在创建节点时候，会生成范围为[0-1]的一个随机数，如果这个随**

机数小于 0.25（相当于概率 25%），那么层数就增加 1 层，然后继续生成下一个随机数，直到随机数的结果大于 0.25 结束，最终确定该节点的层数。




层高有限制，7.0 是 32，6.0 是 64，3.0 是 32

#### 4.1.2.6.2 为什么用跳表不用平衡树

- 从内存占用上来比较，跳表比平衡树更灵活一些。平衡树每个节点 2 个指针，Redis

中每个节点平均 1.33 个指针。

- **在做范围查找的时候，跳表比平衡树操作要简单**。在平衡树上，我们找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。而在跳表上进行范围查找就非常简单，只需要在找到小值之后，对第 1 层链表进行若干步的遍历就可以实现。
- **从算法实现难度上来比较，跳表比平衡树要简单得多**。平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而跳表的插入和删除只需要修改相邻节点的指针，操作简单又快速。

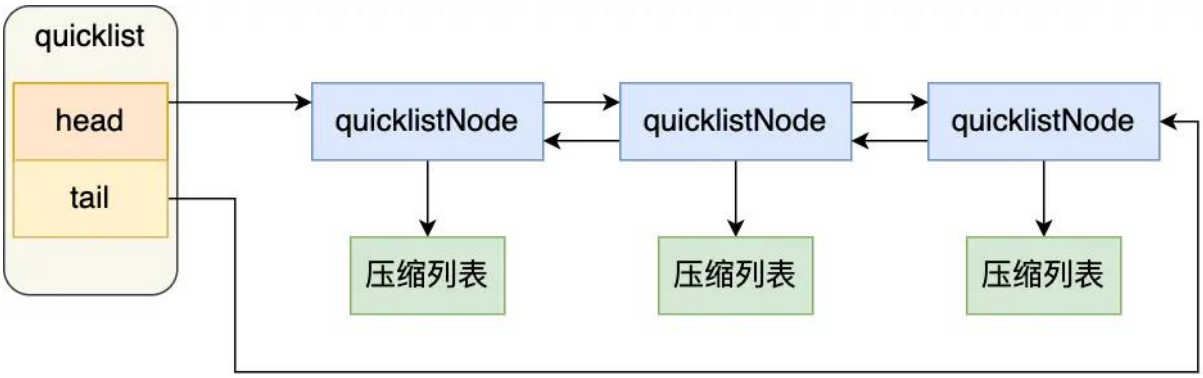
#### 4.1.2.7 quicklist

List 对象 3.0 之前底层是双向链表或者压缩列表，3.2 之后是由 quicklist 实现。

quicklist 本质上是**双向链表+压缩链表**的组合，一个 quicklist 就是一个链表，链表中每个元素又是一个压缩列表。

其中，quicklist 通过控制每个链表节点中的压缩列表的大小或者元素个数，来规避连锁更新的问题。因为压缩列表元素越少或越小，连锁更新带来的影响就越小，从而提供了更好的访问性能

##### 4.1.2.7.1 结构设计




类似于链表，但是 quicklist 的节点是 quicklistNode

quicklist 添加元素的过程：

在向 quicklist 添加一个元素的时候，不会像普通的链表那样，直接新建一个链表节点。而是会检查插入位置的压缩列表是否能容纳该元素，如果能容纳就直接保存到 quicklistNode 结构里的压缩列表，如果不能容纳，才会新建一个新的 quicklistNode 结构。

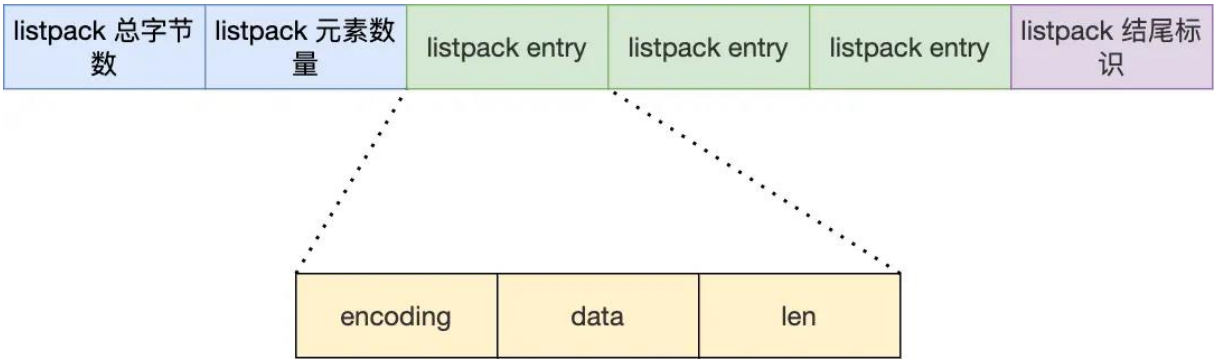
quicklist 会控制 quicklistNode 结构里的压缩列表的大小或者元素个数，来规避潜在的连锁更新的风险，但是这并没有完全解决连锁更新的问题。

#### 4.1.2.8 listpack

**listpack 解决了压缩列表中可能存在的连锁更新的隐患**：每个节点不再包含前一个节点的长度了，压缩列表每个节点正因为需要保存前一个节点的长度字段，就会有连锁更新的隐患。

##### 4.1.2.8.1 结构设计

类似于压缩列表，仍然是连续内存，并且为了节省内存的开销，listpack 节点会采用不同的编码方式保存不同大小的数据




- encoding，定义该元素的编码类型，会对不同长度的整数和字符串进行编码；
- data，实际存放的数据；
- len，encoding+data 的总长度；

可以看到，**listpack 没有压缩列表中记录前一个节点长度的字段了，listpack 只记录当前节点的长度，当我们向 listpack 加入一个新元素的时候，不会影响其他节点的长度字段的变化，从而避免了压缩列表的连锁更新问题**。

## 4.2 Redis 缓存相关

### 4.2.1 Redis 缓存雪崩、缓存穿透、缓存击穿、缓存预热、缓存更新、缓存降级

| | 原因 | 解决办法 |
|---|---|---|
| **缓存雪崩** | 大量请求无法在 Redis 中进行处理<br>1. 缓存中大量数据同时过期<br>2. Redis 挂了 | 针对原因 1：<br>1）避免大量数据同时过期，给过期时间增加较小的随机数；<br>2）服务降级：核心数据查缓存，非核心数据不查缓存，返回预定义信息、空值或错误信息<br>针对原因 2：<br>1）服务熔断：暂停应用对缓存的访问，影响过大<br>2）请求限流：限制每秒进入系统的请求数<br>3）事前预防：构建 Redis 高可用集群（主从架构+cluster） |
| **缓存穿透** | 查询不存在，要访问的数据既不在 Redis 中也不在数据库中（查询一个一定不存在的数据，导致每次请求都要去数据库查询） | 1. 使用布隆过滤器提前拦截：数据写入 DB 时更新布隆过滤器，缓存缺失查询数据库时先查布隆过滤器，在 DB 中可能存在才去查 DB<br>2. 从 DB 中查不到时缓存空对象 |
| **缓存击穿** | 高热 key 失效 | 1. 高热 key 不设置过期时间：过期时间存在 value 里，通过程序后台异步线程进行缓存构建<br>2. 使用互斥锁：只让一个线程构建缓存，其他线程等待构建完直接从缓存获取 |
| 缓存预热 | | 将相关的缓存数据直接加载到缓存系统，避免在用户请求时先查 DB 再更新 |

| | | |
|---|---|---|
| | | 缓存 |
| 缓存更新 | | LRU（访问时间最旧淘汰）/LFU（频次低的淘汰）<br>超时剔除（设置 key 过期时间）<br>主动更新（程序里设置生命周期） |
| 缓存降级 | | 缓存失效或缓存服务挂掉不去访问数据库，而是直接返回内存中保存的数据或默认值 |

### 4.2.2 过期 key 删除以及缓存淘汰（内存淘汰）缓存满了怎么办？

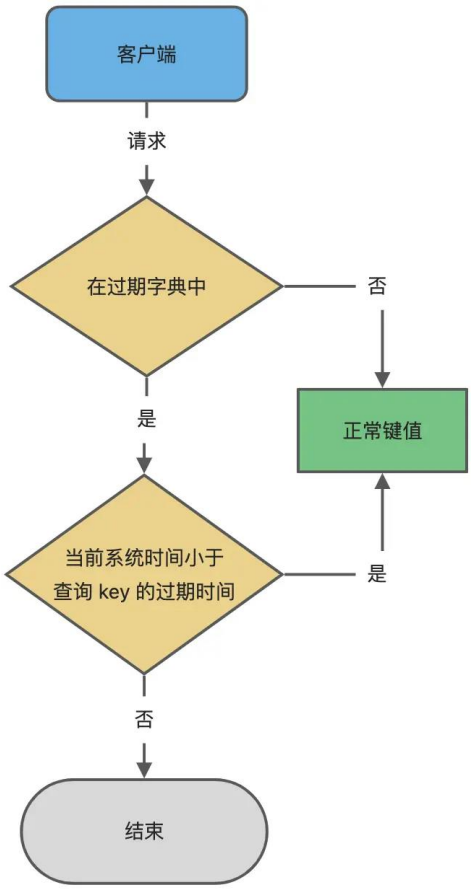
**01 key 过期后如何处理**

获取 key 过期时间的方式

我们可以对一个 key 设置过期时间，Redis 会把该 key 带上过期时间存储到一个**过期字典**（**expires dict**）中，也就是说「过期字典」保存了数据库中所有 key 的过期时间。

当我们查询一个 key 时，Redis 首先检查该 key 是否存在于过期字典中：

- 如果不在，则正常读取键值；
- 如果存在，则会获取该 key 的过期时间，然后与当前系统时间进行比对，如果比系统时间大，那就没有过期，否则判定该 key 已过期。




Redis 使用的过期删除策略是「**惰性删除+定期删除**」这两种策略配和使用。

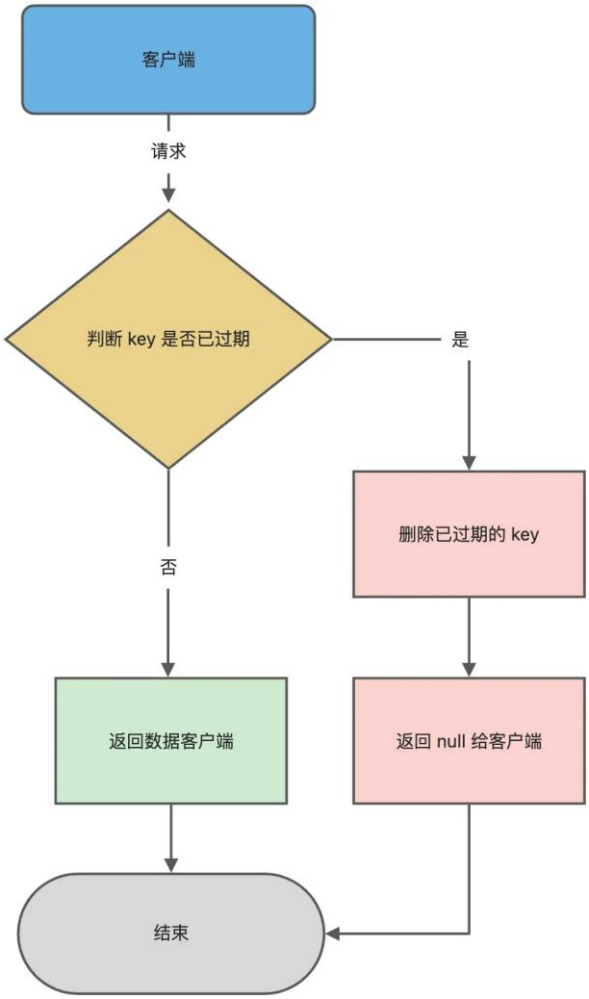
定时删除

**对每一个设置了过期时间的 key 都会创建一个定时器，一旦到达过期时间就会立即删除**

缺点：占用了大量 CPU 资源去处理过期数据，会影响 Redis 的吞吐量和响应时间，所以 Redis 没有采用这种方式。

惰性删除

**不主动删除过期键，每次从数据库访问 key 时，都检测 key 是否过期，如果过期则删除该 key。**




优点：每次访问时才检查 key 是否过期，占用 CPU 资源少。

缺点：key 过期后如果一直不被访问则会一直在内存中，造成内存空间浪费，对内存不友好。

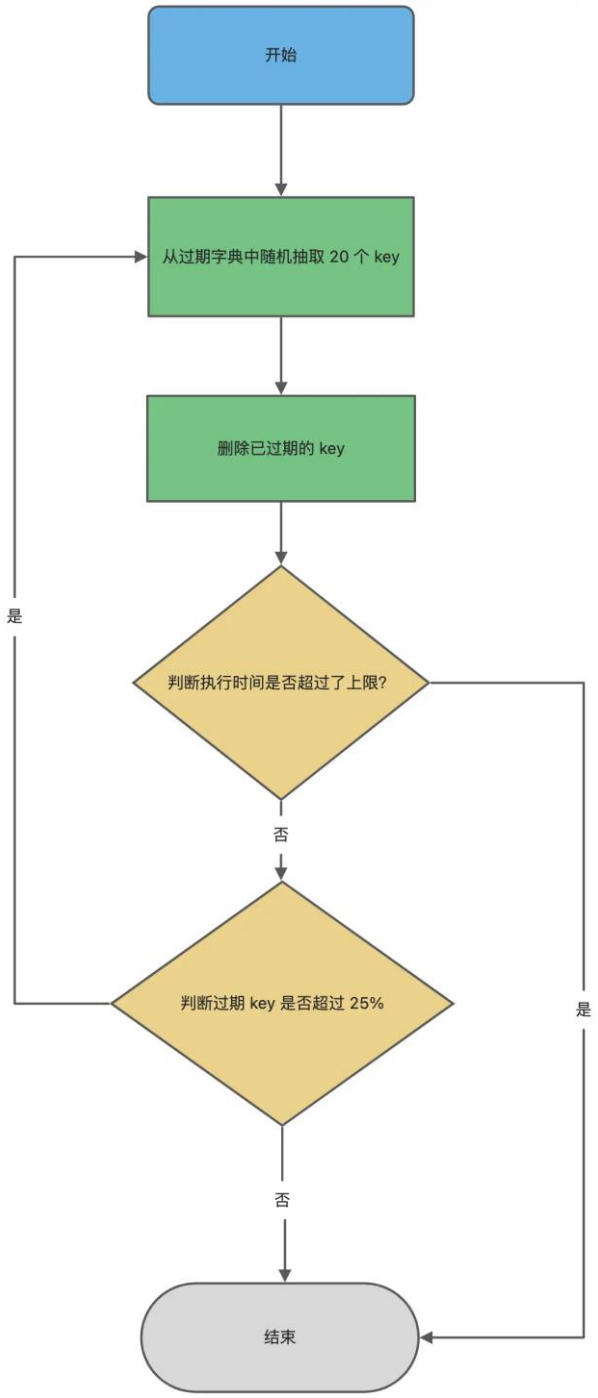
定期删除

**每隔一段时间（默认 100ms，每秒 10 次）「随机」从数据库中取出一定数量（代码里写死 20 个）的 key 进行检查，并删除其中的过期 key。**

定期删除循环流程的时间上限，默认不会超过 25ms。

**详细流程：**

- 从 DB 的过期字典中随机取 20 个 key，判断是否过期，如果过期则清理；
- 如果有 5 个以上（也就是超过 25%）key 过期，则循环继续清理
- 在清理过程中，如果达到 CPU 的 25%时间（默认情况下是 25ms）则退出清理过程。




优点：折中方案

删除 key 的方式

Redis4.0 以前，删除指令是 **del**，del 会直接释放对象的内存，但是如果删除的 key 是一个非常大的对象（大 key），那么删除操作就会导致单线程卡顿，Redis 的响应就慢了。

Redis4.0 版本引入了 **unlink** 指令，能对删除操作进行“懒”处理，只是将该 key 标记为待删除状态，将删除操作丢给后台线程，由后台线程 BIO（background IO service）来异步回收内

存。

Redis 持久化时，对过期 key 如何处理？

RDB 文件分为两个阶段，RDB 文件生成阶段和加载阶段

- RDB 文件生成阶段：过期 key 不会被保存到新的 RDB 文件中。
- RDB 加载阶段：
  - Redis 是【主服务器】：载入 RDB 文件时，会对文件中保存的 key 进行检查，过期 key 不会被载入
  - Redis 是【从服务器】：载入 RDB 文件时，不论 key 是否过期都会被载入；不过没影响，主从同步时，从服务器的数据会先被清空。

AOF 文件分为两个阶段：AOF 文件写入阶段和 AOF 重写阶段

- AOF 文件写入阶段：如果过期 key 没被删除，AOF 会保留此 key，当此过期 key 被删除后，Redis 会向 AOF 文件追加一条 DEL 命令来显式删除该 key。
- AOF 重写阶段：已过期的 key 不会被保存到重写后的 AOF 文件。

Redis 主从模式中，对过期 key 如何处理？

从库不会主动进行过期 key 处理，客户端访问从库，依然可以像未过期的 key 一样返回对应的值。

从库依靠主库控制：**主库在 key 到期时，会在 AOF 文件里增加一条 del 指令，同步到所有的从库**，从库通过执行这条 del 指令来删除过期的 key

**02 内存淘汰策略**

Redis 的内存淘汰策略，是指内存达到 maxmemory 极限时，使用某种算法来决定清理掉哪些数据，以保证新数据的存入。

- 64 位系统中 maxmemory 默认值是 0，即没有内存大小限制，直到 Redis 实例因内存不足而崩溃也不淘汰
- 32 位系统中，maxmemory 默认值是 3G

| | | |
|---|---|---|
| 不进行数据淘汰 | noeviction | 不淘汰任何数据，直接返回错误 |
| 进行数据淘汰且仅针对设置了过期时间的 key | volatile-random | 随机淘汰设置了过期时间的任意 key |

| | | |
|---|---|---|
| | volatile-ttl | 优先淘汰最早过期的 key |
| | volatile-lru(3.0 之前默认) | 淘汰所有设置了过期时间的 key 中，最久未使用的 key |
| | volatile-lfu(4.0 之后新增) | 淘汰所有设置了过期时间的 key 中，最少使用的 key |
| 进行数据淘汰且针对所有数据 | allkeys-random | 随机淘汰任意 key |
| | allkeys-lru | 淘汰所有 key 中最久未使用的 key |
| | allkeys-lfu(4.0 之后新增) | 淘汰所有 key 中最少使用的 key |

Redis 是如何实现 LRU 和 LFU 的？

LRU：Redis 实现了一种近似 LRU 的算法，在 redisObject 结构体中有一个 **lru 字段**，记录了此数据最后一次访问时间（24 位记录了 key 的访问时间戳）。

```
typedef struct redisObject {
    ...
    unsigned lru:REDIS_LRU_BITS; // 对象最后一次被访问的时
    ...
}robj;
```

当 Redis 进行内存淘汰时，会使用随机采样的方式来淘汰数据，随机取 5 个值（可配置），然后淘汰最久没有使用的那一个。

优点：

- 不需要维护一个大链表，节省空间
- 不用每次访问数据都移动链表项，提升缓存性能。

缺点：

- **无法解决缓存污染问题**：即应用一次读取大量数据，然而这些数据只会被读一次，造成缓存污染。

TODO: Redis4.0 后引入 LFU 解决此问题

LFU：redisObject 结构体中 lru 字段分成两段存储，高 16 位存储 ldt（Last Decrement Time），低 8 位记录 logc（Logistic Counter）。

- ldt：key 访问时间戳
- logc：记录 key 的访问频次，值越小表示使用频率越低，每个新加入的 key 初始值是 5

### 4.2.3 如何保证数据库和缓存的一致性？

**01 常见的缓存更新策略**

常见缓存更新策略有 3 种：

- Cache Aside（旁路缓存）策略
- Read/Write Through（读写/写穿）策略
- Write Back（写回）策略

Cache Aside（旁路缓存）策略

应用直接与【数据库、缓存】交互，并负责对缓存的维护，细分为【读策略】和【写策略】

**读策略步骤**：

- 命中缓存，则直接返回数据
- 未命中缓存，则从数据库中读取数据，再将数据写入缓存，并返回给用户

**不用更新数据库，只用更新缓存**

**写策略步骤**：先更新数据库中数据，再删除缓存中数据。

对于写操作，其实有两种方式：

- **先删除缓存，再更新数据库**
  - 高并发下可能会导致数据长时间不一致
    - 采用异步更新缓存的策略，读线程不更新缓存，由写线程更新完数据库后通过 binlog 异步更新缓存；不会导致数据不一致，但是在数据库更新完成之前，都要到数据库层面读数据----- 适合一致性要求高的场景
    - 采用延时双删的策略，写线程更新完数据库后再次删除缓存，其他读线程发现缓存无数据，会从数据库中读取更新后的数据，又再次一致
- **先更新数据库，再删除缓存**
  - 高并发不会长时间不一致（可能少部分时间读到旧值，保证最终一致性）
    - 写线程可能会删除缓存失败，此时可以采用消息队列进行删除补偿

下面对上述两种情况进行详细分析

先删除缓存，再更新数据库

- 正常情况：
  1. A 线程写操作，先删除缓存，再更新数据库
  2. B 线程读操作，发现缓存中没有，先从数据库中读更新后的数据，再更新缓存
- 高并发异常
  1. A 线程写操作，先删除缓存，但是**由于网络原因等未及时更新数据库**
  2. B 线程读操作，会去数据库读到旧值，并将**旧值放入缓存**！！（核心问题）
  3. A 线程更新数据库成功（同步情况下写操作不会更新 Redis），此时缓存（旧）和数据库（新）不一致
- 解决方案
  - **异步更新缓存**：B 线程读操作不更新缓存，而是由 A 线程写操作更新数据库成功后，通过 binlog 异步更新缓存。
  - **延时双删**：A 线程写操作更新数据库时会休眠直到事务提交成功，再次删除缓存。其他线程读时会从数据库中读取更新后的新数据，会再次一致。

先更新数据库，再删除缓存

- 正常情况：
  1. A 线程写操作，先更新数据库，再删除缓存
  2. B 线程读操作，发现缓存中没有，则从数据库读取更新后的新数据，并更新缓存
- 高并发异常
  1. A 线程写操作，先更新数据库，但是没来得及删除缓存
  2. B 线程读操作，读取缓存的旧数据，此时数据不一致
  3. A 线程删缓存

  但是这种场景下，其他读线程会更新缓存，更新后数据就一致了，**不一致的时间很短**

  如果 A 线程删除失败的话，读取的就会一直是旧数据了。
- 解决方案
  - **消息队列进行删除补偿**：如果 Redis 删除发现报错，就把 Redis 的 key 作为消息发送到消息队列中，以便后续再次对 Redis 进行删除操作。

## 4.3 Redis 持久化机制

### 4.3.1 Redis 如何实现数据不丢失？

Redis 有三种数据持久化方式：

- **AOF 日志**：每执行一条写操作命令，就把该**命令追加到一个文件**里

- **RDB 快照**：将某一时刻的**内存数据**，以**二进制的方式写入磁盘**
- **混合持久化**：Redis4.0 新增，集成了 AOF 和 RDB 的优点

### 4.3.2 AOF 日志如何实现

Redis 执行完写操作命令后，将该命令以追加的方式写到文件里，重启时会读取该文件的命令，逐一执行命令的方式进行数据恢复

Redis 是先执行写操作后写日志，写 AOF 日志时在主线程中执行

优点：

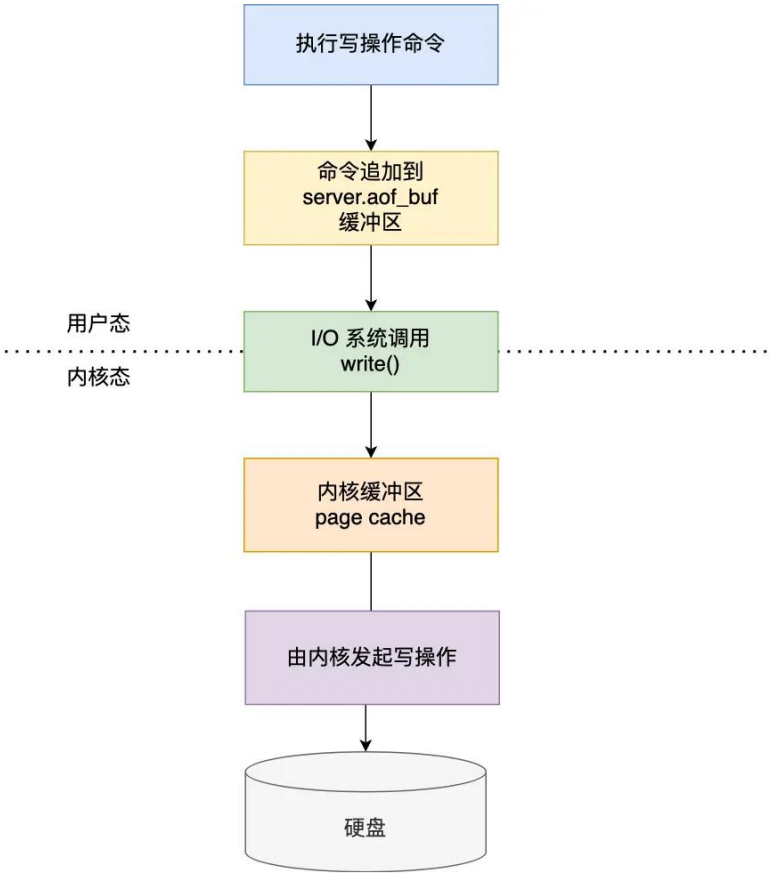
1. 避免额外的检查开销
2. 不会阻塞当前写操作命令的执行

缺点：

1. 数据可能会丢失：因为如果没有将写命令写入硬盘就宕机，有丢失的风险
2. **AOF 日志是在主线程中执行**，在写入磁盘时还是会阻塞后续操作

AOF 写入过程

AOF 写入过程如下：




1. Redis 执行完写操作命令后，会将命令追加到 server.aof_buf 缓冲区（用户态缓冲区）；
2. 然后通过 write() 系统调用，将 aof_buf 缓冲区的数据写入到 AOF 文件，此时数据并没有写入到硬盘，而是拷贝到了内核缓冲区 page cache，等待内核将数据写入硬盘；
3. 具体内核缓冲区的数据什么时候写入到硬盘，由内核决定。

可以看到，AOF 是先写到内核的缓冲区，具体的写回磁盘有 3 种策略：

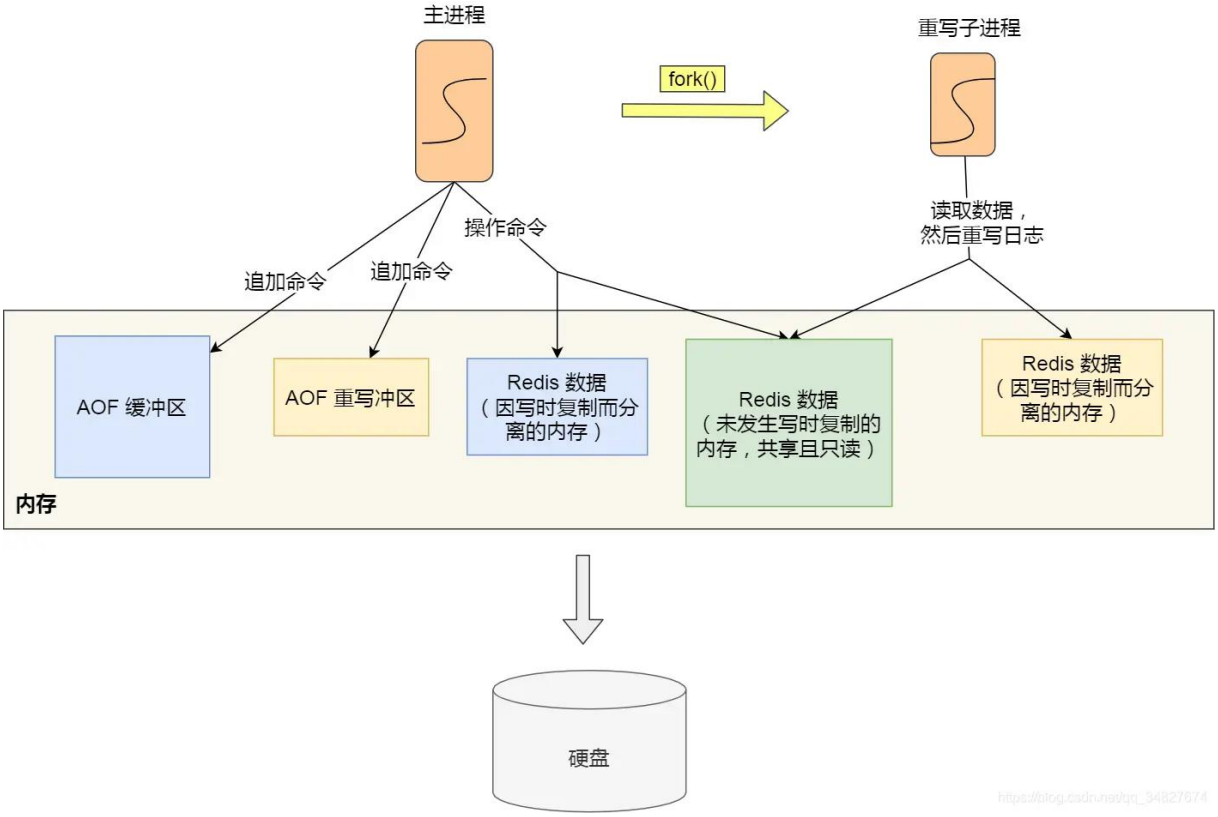
- **Always：**每次写操作命令执行完后，同步将 AOF 日志数据写回硬盘；
- **Everysec**：每次写操作命令执行完后，每隔一秒将内核缓冲区里的内容写回到硬盘；
- **No：**意味着不由 Redis 控制写回硬盘的时机，每次写操作命令执行完后，先写入内核缓冲区，由操作系统决定何时将缓冲区内容写回硬盘。

| 写回策略 | 写回时机 | 优点 | 缺点 |
| --- | --- | --- | --- |
| Always | 同步写回 | 可靠性高、最大程度保证数不丢失 | 每个写命令都要写回硬盘，性能开销大 |
| Everysec | 每秒写回 | 性能适中 | 宕机时会丢失1秒内的数据 |
| No | 由操作系统控制写回 | 性能好 | 宕机时丢失的数据可能会很多 |

AOF 过大会触发重写机制

AOF 重写机制是在重写时，读取当前数据库中的所有键值对，然后将每一个键值对用一条命令记录到「新的 AOF 文件」，等到全部记录完后，就将新的 AOF 文件替换掉现有的 AOF 文件。

Redis 的**重写 AOF 过程是由后台子进程 *bgrewriteaof* 来完成的，在重写过程中，主进程仍然可以正常服务**



具体过程如下：

1. 执行重写时，**主进程拷贝出后台的 bgrewriteaof 子进程即 AOF 重写子进程，父子进程共享内存**（采用 copy on write 机制，并不会将父进程所有数据拷贝给子进程，而是将父进程的内存页表拷贝给子进程，子进程对这块内存只读）
2. AOF 重写**子进程会读取 Redis 所有的数据**，并把 kv 转换成一条命令，**将命令记录到 AOF 重写文件**。
3. 如果此时父进程发生写操作，则会将**父进程写操作同时写入内核中的【AOF 缓冲区】和【AOF 重写缓冲区】**，如果写操作是针对一个已经存在的 key，那么父进程会申请新的内存空间，并将该 key 对应的页表项修改为新申请的内存地址（子进程中该 key 的页表项还是指向原来的内存地址，父子的内存数据开始分离）
4. 子进程完成 AOF 重写工作后，会向主进程发送一条信号，主进程收到信号后，调用信号处理函数，做如下工作：
   a. 将【AOF 重写缓冲区】所有内容追加到新的 AOF 文件中
   b. 新 AOF 文件改名，覆盖现有的 AOF 文件

注意：

1. AOF 日志重写时可能发生阻塞的点：
   a. **父进程 fork 创建子进程时**：拷贝完成前父进程是阻塞的。如果父进程内存过大，则 fork 时间会越久
   b. **父进程操作大 key 时**：大 key 重写申请大块内存耗时会比较久，可能有阻塞风险。
   c. **Redis 机器上开启了内存大页机制（Huge Page，页面大小为 2M）（Huge Page 可以提升 TLB 命中率，因为在相同内存容量下，huge page 可以减少页表项，TLB 可以缓存更多页表项，进而减少 TLB miss 的开销）**：父进程申请新内存时可能会发生阻塞
2. 为什么不对之前的 AOF 进行操作：
   a. 父子进程写同一个文件必然会产生竞争，进而影响父进程性能
   b. AOF 重写失败的话会污染原 AOF 文件，无法再做故障恢复

### 4.3.3 RDB 快照如何实现

由于 AOF 中记录的都是一条条命令，使用 AOF 做故障恢复时需要将文件中所有命令都执行一遍，可能会恢复的很缓慢，进而影响正常使用，所以有另一种持久化方法：**RDB，内存快照，就是指内存中的数据在某一个时刻的状态（全量数据）记录**。

**RDB 记录的是某一时刻的内存数据，当数据恢复时，我们直接将 RDB 文件读入内存，就可以**

**很快地恢复**。

RDB 如何生成

生成有两个命令：

- save：主线程生成，会阻塞主线程
- bgsave：创建子进程生成 RDB 文件，避免主线程阻塞

RDB 在服务器启动时自动加载，无命令

可以通过配置来实现每隔一段时间自动执行一次 bgsave 命令：

```
save time count
```

表示在 time 时间内，对 redis 进行了至少 count 次修改

RDB 执行过程中，数据可以被修改，修改操作会在 RDB 生成过程结束后写入 RDB 文件

使用 **COW 技术**，当父进程发生写操作时才会新申请内存，子进程仍然使用之前的内存写 RDB 文件。

### 4.3.4 混合持久化

单纯用 AOF 或者 RDB 做持久化都不太好，原因是：

1. 单纯用 AOF：需要执行文件中所有的命令，恢复速度慢
2. 单纯用 RDB：频率不好把控，相比于 AOF 可能会丢失更多的数据

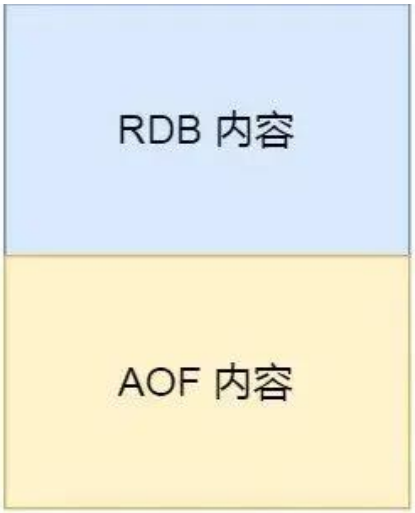
混合持久化工作在 AOF 日志重写过程。

详细过程：

1. AOF 重写日志时，会先将与主进程共享的内存数据以 RDB 的方式写入 AOF，同时主线程的写操作命令会被记录在【AOF 重写缓冲区】
2. 【AOF 重写缓冲区】的命令会以增量的方式追加的 AOF 文件中
3. 替换旧的 AOF 文件

也就是说，使用了混合持久化，AOF 文件**前半部分是 RDB 格式的全量数据，后半部分是 AOF**

**格式的增量数据**。




混合持久化的 AOF 文件内容

好处：

重启 Redis 加载数据时，前半段是 RDB，加载速度很快，后半段是 AOF，使数据丢失更少。

## 4.4 Redis 单线程模型

### 4.4.1 说说 Redis 单线程模型以及为什么用单线程

Redis 为什么要用单线程？

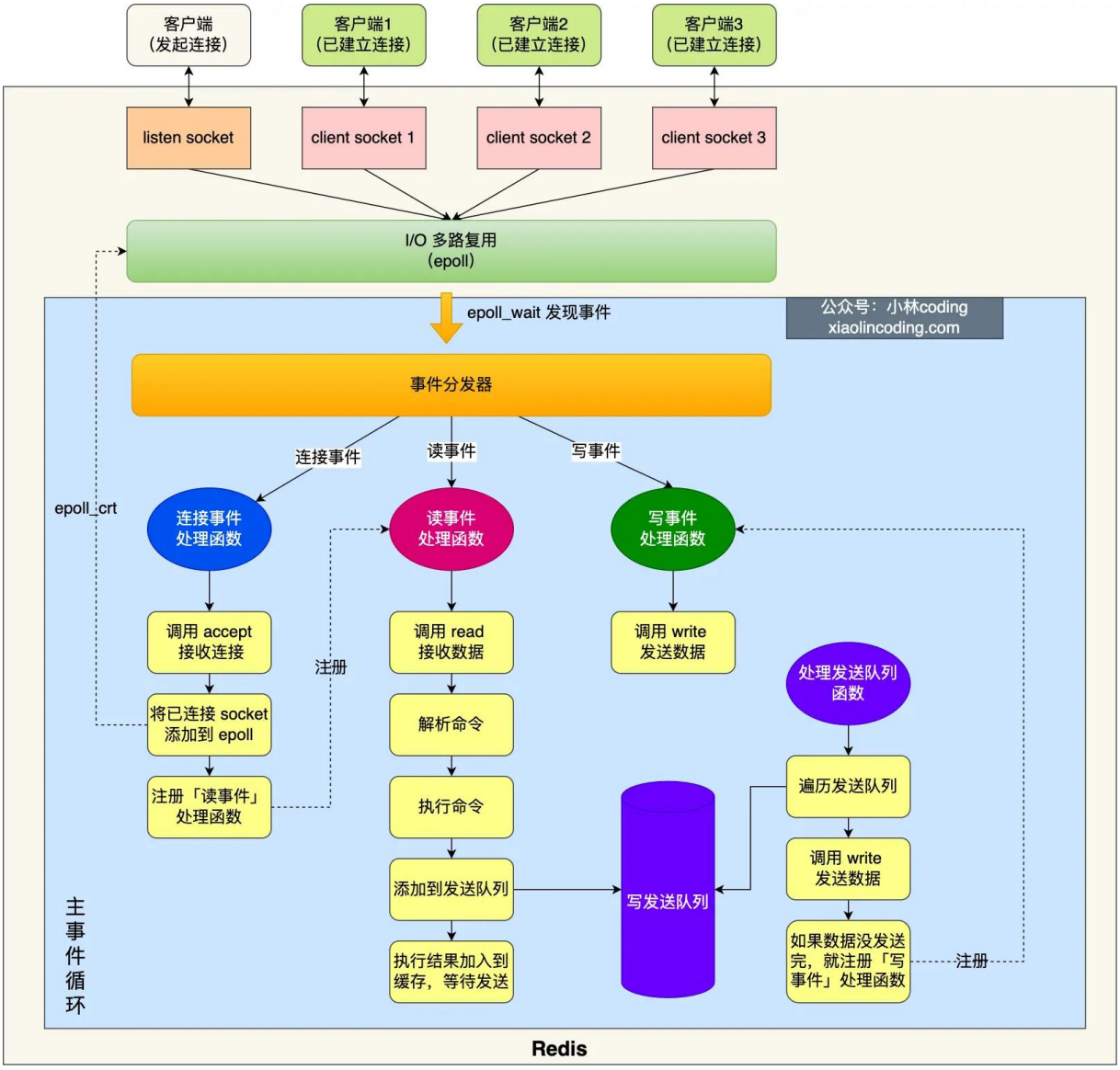
**Redis 单线程指的是「接收客户端请求->解析请求 ->进行数据读写等操作->发送数据给客户端」这个过程是由一个线程（主线程）来完成的。**

实际上，Redis 还有几个后台线程：

1. 关闭文件 2. AOF 刷盘 3. 异步释放 Redis 内存，也就是 lazyfree

Redis 单线程模型

Redis 采用的是单 Reactor 单进程的方案




图中蓝色部分是一个事件循环，由**主线程负责**

初始化：

1. 调用 epoll_create 对象和调用 socket()创建一个服务端 socket
2. 调用 bind()绑定端口和调用 listen()监听该 socket
3. 调用 epoll_ctl()将 listen socket 加入到 epoll，同时注册【**连接事件处理函数**】。

初始化完成后，开始了事件循环，主要做以下事：

1. 调用处理发送队列函数（beforesleep）遍历发送队列，如果有发送任务，则通过 write 函数将客户端发送缓冲区里的数据发送出去；如果没发送完，则注册【写事件处理函数】，等待 epoll_wait 可写后处理。

2. 接着，调用 epoll_wait 函数等待事件到来：
   - **连接事件到来**：调用【**连接事件处理函数**】，该处理函数的逻辑包括：调用 accept 获取已连接的 socket---》调用 epoll_ctl 将已连接的 socket 加入到 epoll---》注册【**读事件处理函数**】
   - **读事件到来**：调用【**读事件处理函数**】，该处理函数逻辑包括：调用 read 获取客户端发送的数据---》解析命令---》处理命令---》将客户端对象添加到发送队列----》将执行结果写入客户端发送缓冲区等待发送。
   - **写事件到来**：调用【**写事件处理函数**】，该处理函数逻辑包括：调用 write 将客户端发送缓冲区的数据发出去---》如果这一轮数据没发送完则注册【写事件处理函数】，等待 epoll_wait 发现可写后再处理。

### 4.4.2 Redis 用单线程为什么这么快？

1. Redis 大部分操作**都在内存中完成**，并且采用了高效的数据结构，CPU 不是瓶颈
2. 可以**避免多线程之间的竞争**，多线程会增加系统复杂度，同时可能存在线程切换、加锁解锁甚至死锁造成的性能损耗
3. 采用了 **I/O 多路复用机制**，一个线程可以处理多个连接

### 4.4.3 Redis 6.0 之后为什么引入多线程

TODO:为了提高网络 I/O 并行度，Redis 6.0 之后也采用了**多个 I/O 线程处理网络请求**，但是**对于命令的执行，还是单线程**

## 4.5 Redis 集群

### 4.5.1 Redis 主从架构

Redis 主从服务器之间采用【读写分离】的方式，主服务器可以读写，从服务器只读并接受主服务器同步过来的写操作命令。

总结：主从复制分为三种情况：

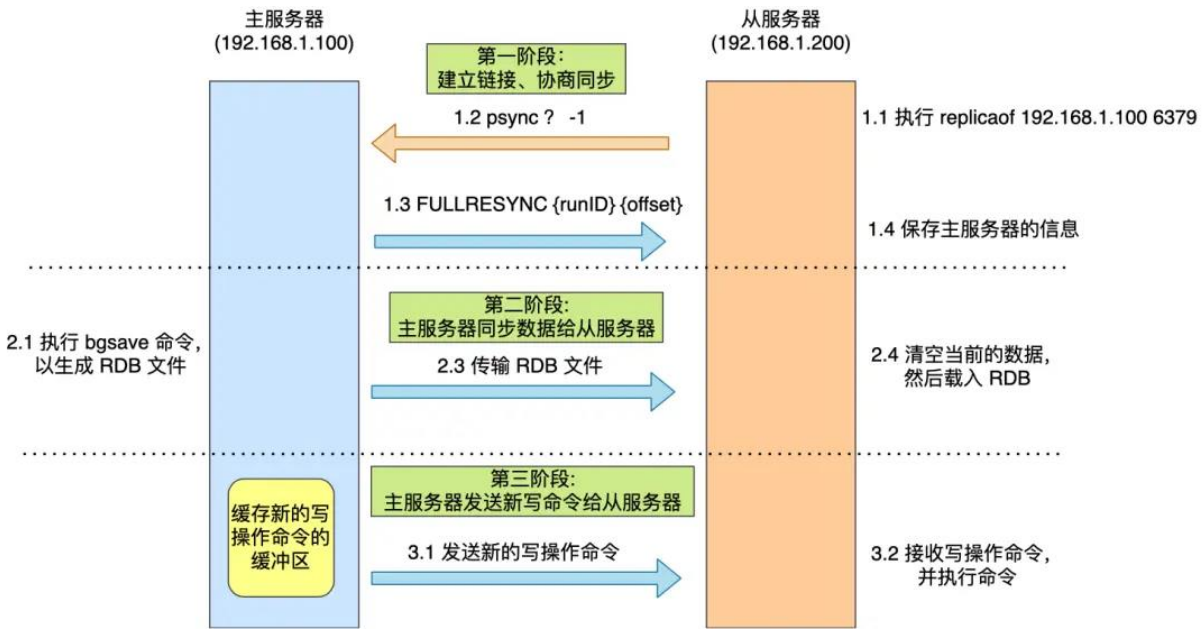
1. 第一次同步：发生在第一次成为主的从节点时；这时从会发送 psync 命令，指定参数 offset 为-1 表示本次为全量复制，随后主将 RDB 通过 replication buffer 发送给从，从加载 RDB 保持主从同步。

2. 命令传播：发生在第一次同步之后；主和从会维持一个长连接，主中每一个写命令都会发送给从，从执行该命令保持和主的同步
3. 增量同步：发生在主从断连之后重新连接的情况；从会发送 psync 命令给主，指定 runid 和 offset，主通过判断 runid 和 offset 以及自己的 offset 决定采用增量同步还是全量同步的方式
    - 如果 runid 和自己的 id 不一致，说明换主了，只能全量同步
    - 如果 repl_backlog_buffer 中 master_repl_offset 和 slave_repl_offset 之间的差值大于缓冲区大小，说明已经发生了覆盖，只能全量同步；否则只发送差异数据给从

**01 第一次同步**

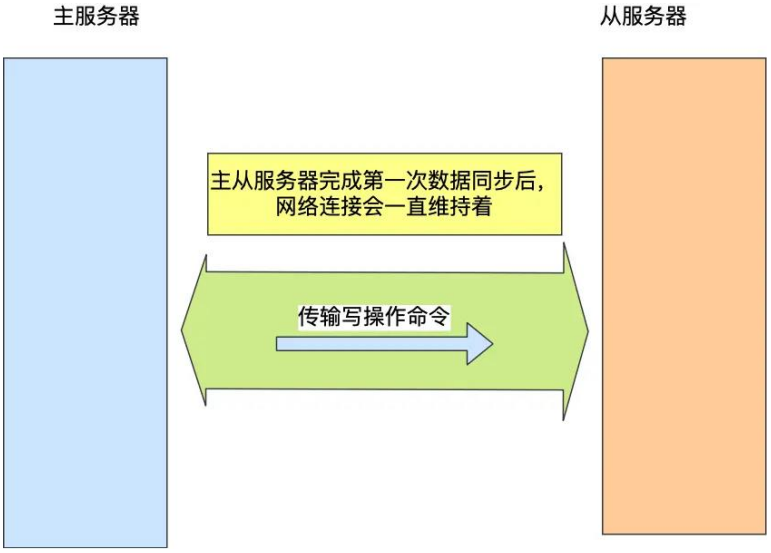
主从服务器之间第一次同步可分为三个阶段：

1. **建立链接、协商同步**：
    a. 从给主发送 psync 命令（包含两个参数，主服务器 runID（redis 启动时自动生成一个随机的唯一 ID，第一次同步值为"?"）和复制进度 offset（第一次同步值为-1）），发起数据同步；
    b. 主响应 FULLRESYNC（表示全量复制）；
2. **主服务器同步数据给从服务器**：
    a. 主执行 bgsave 生成 RDB 并发给从
    b. 从先清空当前数据，然后载入 RDB 文件
    c. 在收到从的回复确认消息之前，主会将期间收到的写操作命令写入到 **replication buffer** 缓冲区里
3. **主服务器发送新写操作命令给从服务器**：
    a. 从完成 RDB 加载后，回复确认消息给主
    b. 主将 replication buffer 缓冲区里记录的写操作命令发送给从，从执行命令




## 02 命令传播

基于长连接的命令传播保证第一次同步后的主从数据的一致性：主从第一次同步后，双方会维护一个 TCP 长连接，后续主的写操作命令会通过此长连接传播给从，从执行命令使得与主的数据库状态相同。



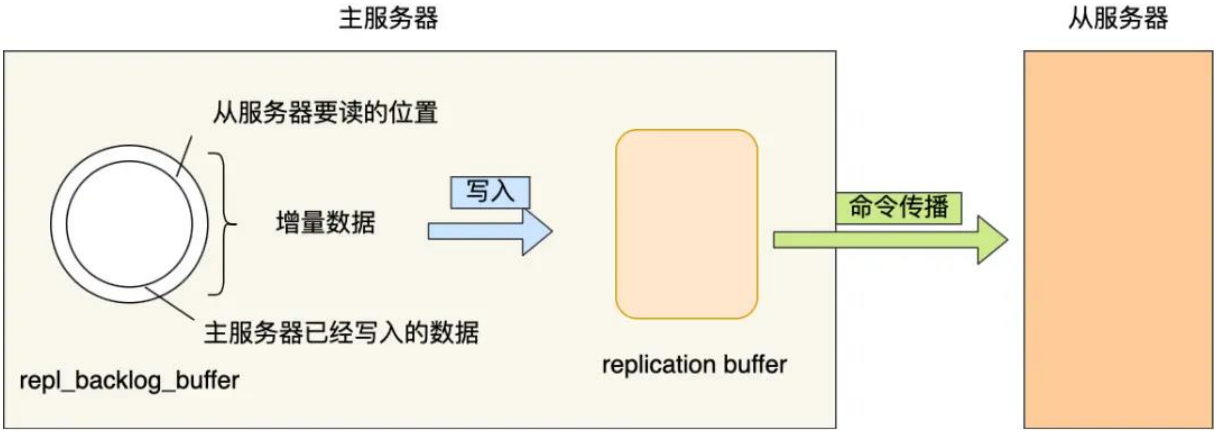

## 03 增量同步

**主从断开后，当连接恢复时，会视情况采用增量同步**

首先介绍两个概念：

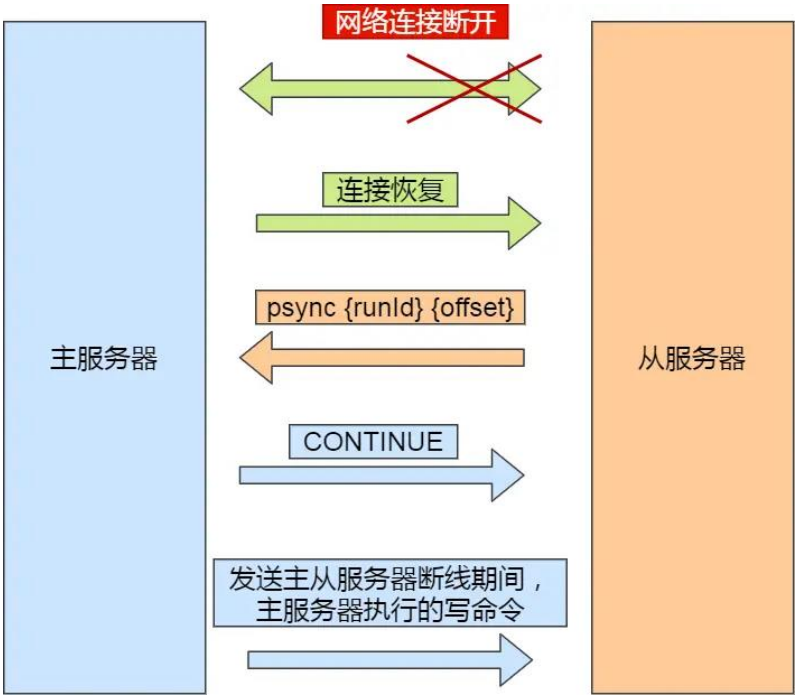
- **repl_backlog_buffer**，是一个「环形」缓冲区，用于主从服务器断连后，从中找到差异的数据；在命令传播过程中，主不仅将写操作发送给从，也会写入 repl_backlog_buffer 缓冲区。

- **replication offset**，标记上面那个缓冲区的同步进度，主从服务器都有各自的偏移量，主服务器使用 master_repl_offset 来记录自己「写」到的位置，从服务器使用 slave_repl_offset 来记录自己「读」到的位置。




**主要流程：**

1. 从服务器恢复网络后，发送 psync 命令给主服务器，此时 psync 命令里 offset 是从的复制偏移量 slave_repl_offset，runid 是主的 id。
2. 主收到命令后，告诉从接下来采用增量复制的方式同步数据。
   a. 主判断有哪些增量数据的方式：
      i. 主根据 master_repl_offset 和从的 slave_repl_offset 之间的差距，决定对从采用哪种同步操作（当 master 覆盖 slave 时，采用全量复制）。
      ii. 如果 runid 不是主的 id，则说明换了个主，只能全量复制
   b.
3. 主将主从断连期间所执行的写命令通过 replication buffer 发送给从，然后从执行这些命令




#### 4.5.1.2 其他面试题

TODO

### 4.5.2 哨兵机制

Redis 主从架构中，如果主节点挂了需要手动切换某个从节点为主节点，带来很高的运维负担以及风险，故**提供了哨兵机制，作用是实现主从节点故障转移**

主要干什么：监控主节点是否存活，如果主节点挂了，哨兵会选举一个从节点切换为主节点，并把新主节点的相关信息通知给从节点和客户端。

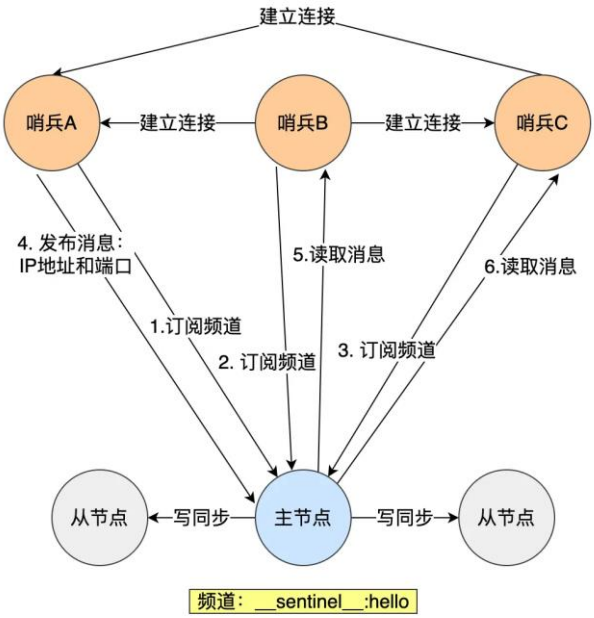
简单来说，就是三件事：监控、选主、通知

**01 哨兵集群如何组成**

哨兵节点之间通过 Redis 的发布者/订阅者机制来相互发现。

具体过程：

- 哨兵发布自己的 IP 地址和端口信息到主节点的__sentinel__:hello 频道上
- 其他哨兵会订阅该频道，就可以从这个频道直接获取到 IP 地址和端口号
- 哨兵之间建立网络连接




哨兵集群会对从节点的运行状态也进行监控，那么如何获取从节点信息：

每 10 秒向主节点发送 INFO 命令获取所有从节点信息。

**02 如何判断主节点真的故障了**

详细过程如下：

1. 哨兵每隔 1 秒发送一个 ping 命令给主从节点，如果没有在规定时间内响应哨兵的 PING 命令，哨兵会将该节点标记为【主观下线】
2. 当一个哨兵判断主节点为【主观下线】后，会向其他哨兵发起命令，其他哨兵收到命令后会 ping 一下主节点，投票赞成或者拒绝
3. 赞同票到达阈值后，主节点就会被该哨兵标记为【客观下线】；阈值一般设置为哨兵个数/2+1

**03 由哪个哨兵进行主从故障转移（哨兵选主）**

标记主节点为【客观下线】之后，需要有个主哨兵来执行主从故障转移

详细流程如下：

1. 将主节点标记为【客观下线】的哨兵是一个 Leader 的候选者（可能同时有多个哨兵成为候选者）

2. 候选者先给自己投一票，然后向其他哨兵发送命令，希望成为 Leader
3. 需要满足两个条件，才能当选 Leader：
    - 拿到半数以上的赞成票
    - 拿到的票数同时需要大于等于哨兵配置文件中的 quorum 值。

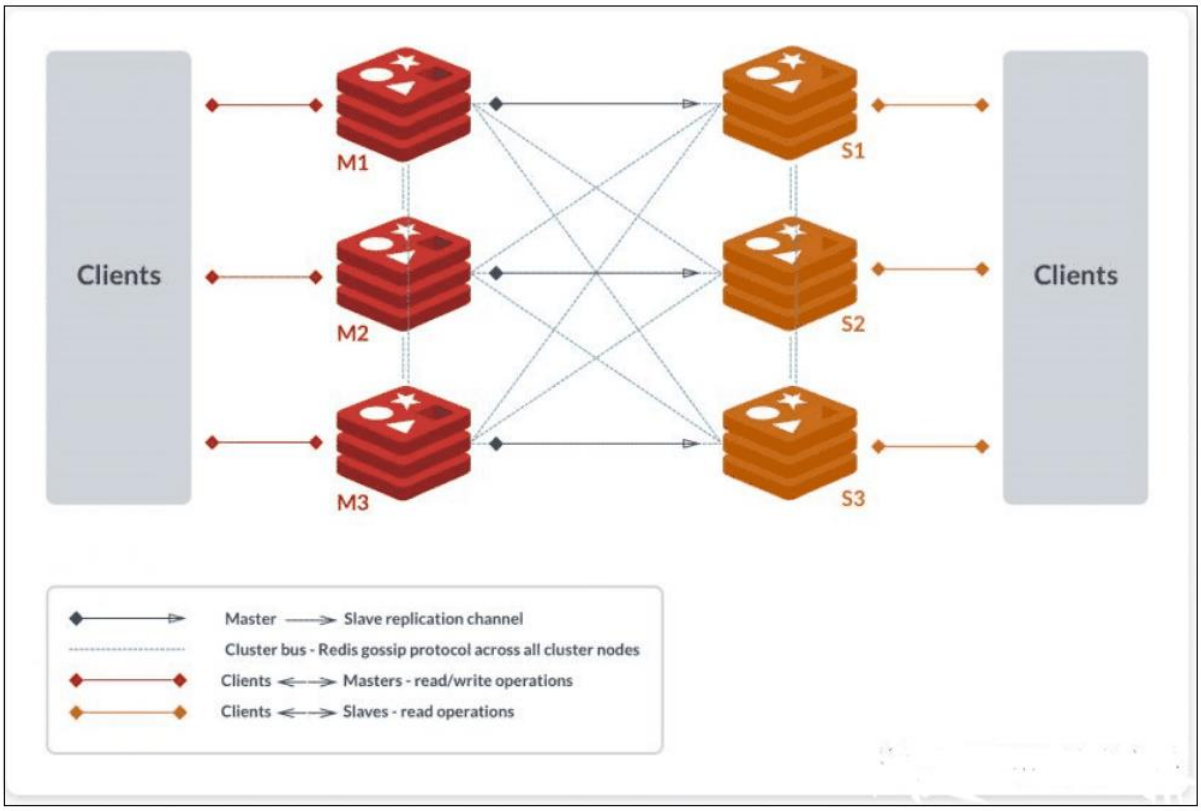
哨兵节点的数量应该是奇数并且 quorum 的值建议设置为哨兵个数的二分之一加 1

**04 主从故障转移的过程**

详细流程：

1. 选出新的主节点：在已经下线主节点属下的所有从节点里面，挑选出一个从节点，并将其转换为主节点。
    a. 首先剔除网络状态不好的从节点；判断依据是主从间断连次数，如果断连次数超过 10 次则将该从节点剔除
    b. 对于剩下的从节点，依次（有先后顺序）从三个方面考虑：**优先级**、**复制进度**、**ID 号**
        i. 优先级越小的从节点排名越靠前；通过 slave-priority 配置项给从节点设置优先级。
        ii. 复制进度最靠前的从节点胜出；如果某个从节点的 slave_repl_offset 最接近 master_repl_offset，说明它的复制进度最靠前，可以选为新主节点。
        iii. ID 号小的从节点胜出；
    c. 哨兵向选出来的从节点发送命令，使其升级为新主节点，并以每秒一次的频率向升级的从节点发送 Info 命令，被升级节点的角色信息从 slave 变为 master 时升级完毕。
2. 将从节点指向新的主节点：向剩下所有的从节点发送 slaveof 或 replicaof 命令，让它们成为新主节点的从节点。
3. 通知客户端主节点已变更：**通过发布者/订阅者机制来实现。**对+switch-master 频道，哨兵发送新主节点的 IP 地址和端口信息，客户端就可以通过这个频道拿到新主节点的信息了。
4. 旧主节点重新上线时将该节点变为从节点：哨兵继续监视旧的主节点，待旧主节点重新上线时，哨兵集群发送 replica of 或 slave of 命令，将旧主节点变成新主节点的从节点

## 4.5.2 Redis 切片集群




集群分片概念

集群模式就是将数据进行分片存储，当一个分片数据达到上限时，还可以分成多个分片。

每一个主节点负责维护一部分槽以及槽所映射的键值数据

Redis 哈希槽的概念以及槽位定位算法

Redis 没有采用一致性哈希算法，而是使用哈希槽。Redis 哈西槽一共有 16384 个，计算给定 key 的哈希槽，我们只需要对 key 的 CRC16 取模 16384

HASH_SLOT = CRC16(key) % 16384

假设集群中有 A、B、C 三个集群节点，**不存在复制模式**下，每个集群的节点包含的哈希槽如下：

- 节点 A 包含从 0 到 5500 的哈希槽；
- 节点 B 包含从 5501 到 11000 的哈希槽；
- 节点 C 包含从 11001 到 16383 的哈希槽；

这时，如果**节点 B 出现故障**，整个集群就会出现缺少 5501 到 11000 的哈希槽范围而不可用。

为什么不直接维护 key 和实例的关系

- key 的数量无法预估：映射表可能非常大
- Redis Cluster 采用去中心化模式，实例间会交换自己的槽位配置，存储了整个集群完整的映射关系：节点之间交换的信息会变得庞大，消耗过多网络资源
- 集群扩缩容时，节点间发生数据迁移，迁移时会修改每个 key 的映射关系，维护成本高

哈希槽配置信息

客户端上线连接到集群某个实例时，就可以得到一份集群槽位配置信息，集群槽位配置信息存储于每个节点中，客户端请求时，会先计算 key 对应的哈希槽，然后给对应的实例发送请求。

重定向

回顾一下客户端的请求流程，客户端在本地缓存了集群的槽位分配信息，会先从本地拿到对应的实例发送请求，但是槽位分配不是一成不变的，集群中新增或者下线某个实例后，需要进行槽位的重新分片，所以客户端请求的流程如下：

- 节点计算槽后查映射发现是本节点服务的：当前节点直接处理并返回数据就可以了
- 节点计算槽后查映射发现槽不是本节点服务的：
  - 当前没有正在迁移：说明之前已经迁移完了，当前节点返回 MOVED 重定向错误
  - 当前正在发生迁移：
    - **客户端请求的哈希槽已经迁移完成**：直接返回 MOVED 重定向，客户端收到后，会修改本地缓存的映射表，然后向新节点请求
    - **客户端请求的哈希槽在迁移过程中**：
      - 请求的 key 还在当前节点：当前节点处理并响应客户端。
      - 请求的 key 已经被迁移走：
        1. 返回给客户端 ASK 重定向错误，告诉客户端新节点的信息
        2. 客户端收到 ASK 后，会临时（一次性）重定向，自动给新节点发送一次 ASKING 命令
        3. 新节点收到请求后判断：
           a. 哈希槽导入完毕：请求的 keys 已经全部导入新节点，新节点处理请求并返回执行结果；
           b. 哈希槽正在导入中：返回重试错误（TRYAGAIN）

4. 如果之前返回的是重试，那么客户端稍后重试继续请求新节点

Redis Cluster 中节点是如何进行通信的？

节点间使用 Gossip 协议进行通信共享

Gossip 消息：

- MEET：给集群中某节点发送 MEET 消息，用于将 Redis 节点添加到切片集群中
- PING/PONG：集群中节点会定时向其他节点发送 PING 消息，来交换各个节点状态信息，检查节点状态，包括在线状态、疑似下线状态 PFAIL、已下线状态 FAIL
- FAIL：

## 4.6 其他

### 4.6.1 大 key 专题

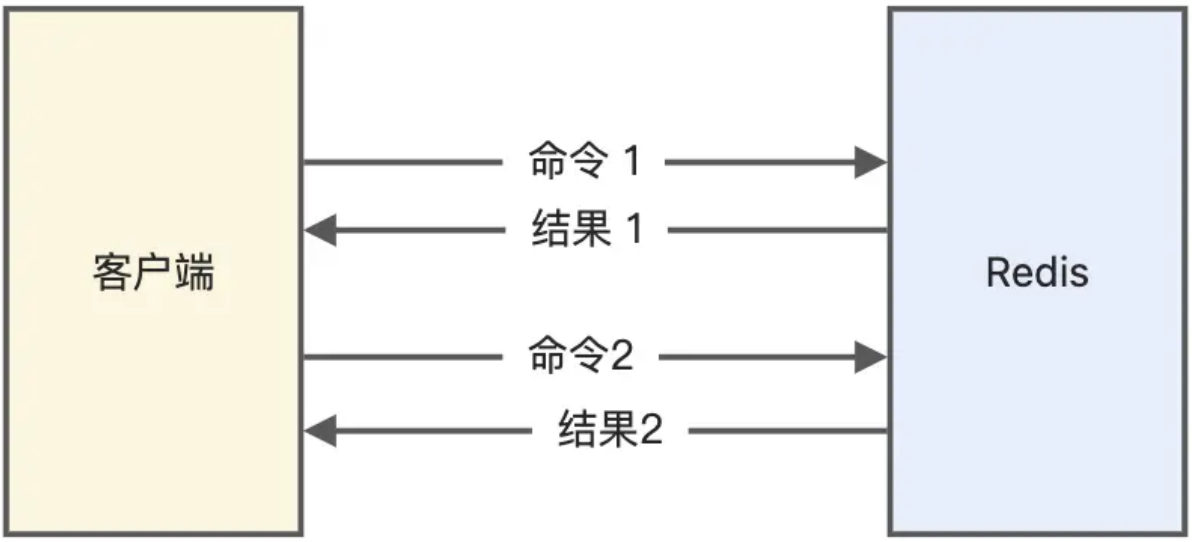
### 4.6.2 Redis 延迟高专题

### 4.6.3 Redis 分布式锁

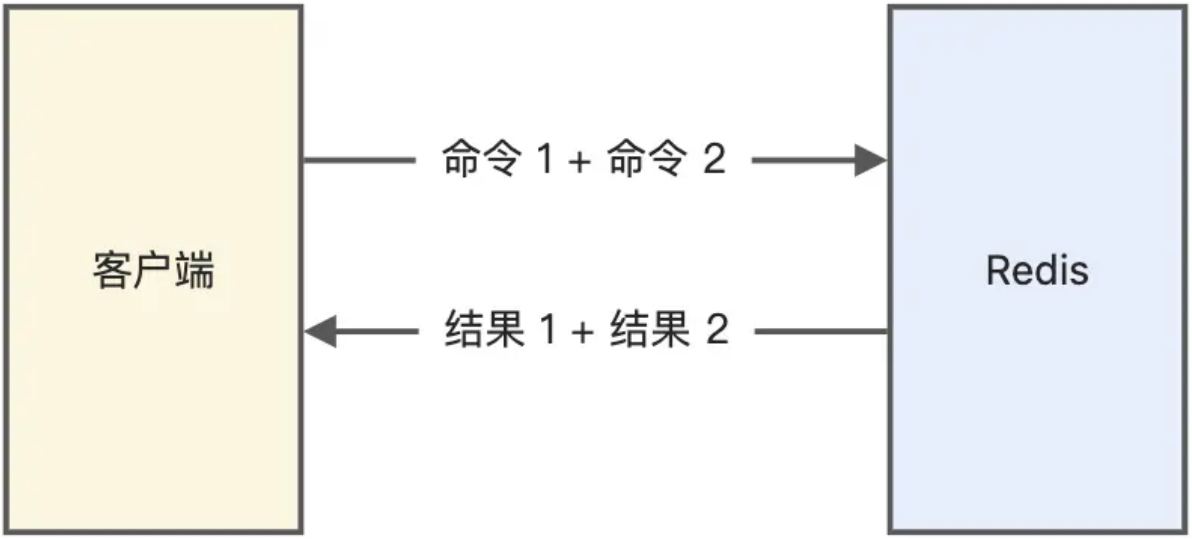
### 4.6.4 Redis 批量操作（pipeline && 原生批量命令 && lua 脚本）

Redis pipeline

普通命令模式：




redis pipeline 模式：




管道技术（Pipeline）是客户端提供的一种批处理技术，用于一次处理多个 Redis 命令，从而提高整个交互的性能。

使用管道技术可以解决多个命令执行时的网络等待，它是把多个命令整合到一起发送给服务器端处理之后统一返回给客户端，这样就免去了每条命令执行后都要等待的情况，从而有效地提高了程序的执行效率。

原生批量操作命令

Redis 提供了一些原生支持批量操作的命令：

- MGET（获取一个或多个指定 key 的值）、MSET（设置一个或多个指定 key 的值）、
- HMGET（获取指定哈希表中一个或多个指定字段的值）、HMSET（同时将一个或多个 field-value 对设置到指定哈希表中）、
- SADD（向指定集合添加一个或多个元素）

在非分片集群模式下命令是原子性的。

集群模式下可以这样请求：

1. 找到 key 对应的所有 hash slot；
2. 分别向对应的 Redis 节点发起 MGET 请求获取数据；
3. 等待所有请求执行结束，重新组装结果数据，保持跟入参 key 的顺序一致，然后返回结果。

Lua 脚本

一段 Lua 脚本可以视作一条命令执行，可以看作是 **原子操作**（一段 Lua 脚本执行过程中不会有其他脚本或者 Redis 命令同时执行）

在非分片集群模式下 Lua 脚本的执行也是原子性的。

集群模式下

管道技术和 mset、mget 的区别

### 4.6.5 Redis 充当消息队列

使用 redis stream 作为消息队列

# 五、 操作系统相关

## 5.1 网络 IO

# 六、 算法题

## 6.1 排序

**重点**：快排、堆排

归并排序

## 6.2 LRU/LFU

## 6.3 golang 并发编程

## 6.4 树的非递归写法

## 6.5 回溯

# 七、 系统设计题

https://soulmachine.gitbooks.io/system-design/content/cn/

# 八、 分布式问题

## 8.0 分布式理论 CAP && BASE

CAP 理论

- C：一致性，所有客户端看到的都是同一份数据，即使在数据更新和删除之后
- A：可用性，即使部分节点发生故障，所有客户端也能找到可用的数据备份
- P：分区容忍性，即使发生网络分区故障，系统仍然能够按照预期正常工作。

一般来说，P 不可避免，只能在 C 和 A 中进行选择

BASE 理论

在选择 AP 模型的同时最大限度的满足一致性。

- BA：基本可用性，即在部分节点出现故障时不要求整个系统完全可用，允许系统出现部分功能和性能上的损失，比如增加响应时间，引导用户到一个降级提示页面等
- S：软状态：不要求数据的变化立即反映到所有的服务器节点上
- E：最终一致性：不要求系统数据始终保持一致状态，只要求系统经过一段时间后最终达到一致状态即可。

核心思想：**强一致性无法得到保障时，我们可以根据业务自身的特点，采用适当的方式来达到最终一致性**。

## 8.1 分布式锁

分布式锁需要满足以下特性：

1. 互斥性：在相同时刻，只有一个客户端能持有相同资源的锁
2. 安全性：对同一把锁，只能由同一个客户端进行加锁和解锁
3. 可用性：避免死锁，相同资源的锁不能无限期被某个客户端持有，导致后续客户端不能加锁。

目前常用的方案有以下三类：

1. 基于数据库事务实现的锁服务
2. 基于分布式缓存实现的锁服务，典型代表有 Redis 等
3. 基于分布式一致性算法实现的锁服务，典型代表有 ZooKeeper

经验之谈：小型系统中，可以使用 mysql 等数据库分布式锁；注重 CP，可以用基于分布式一致性算法实现的锁服务；注重 AP，可以使用基于分布式缓存实现的锁服务。

**01 基于数据库**

创建一张锁表，获取锁的时候在表中增加一条记录，释放锁的时候删除这条记录，利用数据库的唯一索引来保障互斥性。

有以下几个点需要注意：

1. 锁释放失败：需要定时清理或监控运营
2. 锁是非阻塞的，如果需要阻塞则可以使用循环语句直至 INSERT 成功
3. 大规模并发下，数据库并发可能成为瓶颈，适用于并发不是特别高的场景

**02 基于 Redis**

**有两大类，一类是基于 Redis 主从模式的单机实现，另一类是基于 Redis 集群（不是切片集群，就是很多个主从 Redis 组成的集群）的多机实现**

基于 Redis 单机实现的分布式锁

1. 使用原子性命令进行加锁：

```
SET {appid}-{zoneid} 1 NX EX 30
```

2. 为了避免执行业务逻辑时间过长，超过 redis 的失效时间，可以启动守护线程，对加锁的 key 进行续期
3. 执行业务逻辑，操作共享资源
4. 为了避免非原子性操作，使用 lua 脚本对锁进行删除和释放，删除时判断加锁的来源，避免删除其他客户端的锁

```
//解锁
if (redis.call('get',KEYS[1]) == ARGV[1])
then return redis.call('del',KEYS[1])
else return 0
end
```

**优先**：性能比较好，也有比较成熟的业界开源方案

**缺点：**

1. 锁是非阻塞的，无论成功还是失败都会直接返回：可以通过循环重复执行并设置重试次数，直到获取锁为止。
2. 删除锁失败守护线程一直执行，可能导致后续逻辑获取不到锁：自行设定一下续期逻

辑，例如设置多长续期时间

3. 主从切换时有一定时间差，可能出现同时获取到锁的情况：一般需要监控此类场景以及运营处理。场景如：机器 A 申请到一把锁之后，如果 Redis 主宕机，这时候从机并没有同步到这一把锁，那么机器 B 再次申请的时候就会再申请到这把锁。

基于 Redis 多机实现的分布式锁 Redlock

假设有 N 个 Redis 节点，节点间完全独立，在每个节点上获取和释放锁的方式相同，RedLock 获取锁的过程，客户端应执行如下操作：

1. 获取当前 Unix 时间，以毫秒为单位
2. 依次从 N 个实例中使用相同的 key 获取锁，同时客户端会设置一个获取锁的超时时间（应该远小于锁的失效时间），如果获取锁超时了，则向下一个 Redis 实例获取锁。
3. 当且仅当从大多数（大于 N/2+1，一半以上）Redis 节点都获取到了锁，并且获取锁消耗时间（当前时间-开始获取锁的时间）小于锁失效时间，该客户端才叫获取了锁。并且锁的真正有效时间 == 锁失效时间-获取锁消耗时间
4. 如果获取锁失败，客户端在所有 Redis 实例上解锁。

**优点**：相比于单机实现，Redlock 提供了更高的可用性

**缺点**：

1. 需要多台 Redis 实例，维护成本高，并且实现复杂
2. 如果出现服务器挂掉的情况，可能出现脑裂等情况

**03 基于 ZooKeeper**

利用 ZooKeeper 临时顺序节点的特性实现一个分布式锁。

1. 客户端连接 ZooKeeper，并在/locks 下相应资源/key 下创建临时有序子节点，假设第一个客户端的子节点为 001，第二个为 002，以此类推
2. 客户端获取子节点列表，判断创建的节点是否为当前列表中序号最小的节点。如果是则认为获得了锁，否则监听前一个子节点的删除消息。
3. 获取锁的客户端，执行业务代码后，删除当前客户端对应的子节点释放锁。
4. 监听当前子节点的删除消息后一个客户端会收到删除消息，则获取到锁，以此类推

优点：

1. ZooKeeper 具备良好的故障恢复能力和数据一致性保障，leader 宕机后会根据一致性算法选出新的 Leader
2. ZooKeeper 有序临时节点在客户端宕机或断开后能自动删除，不会因为节点不删除而引发后续流程阻塞问题

缺点：

1. 高并发场景性能上不如 Redis 分布式锁
2. 运营成本高

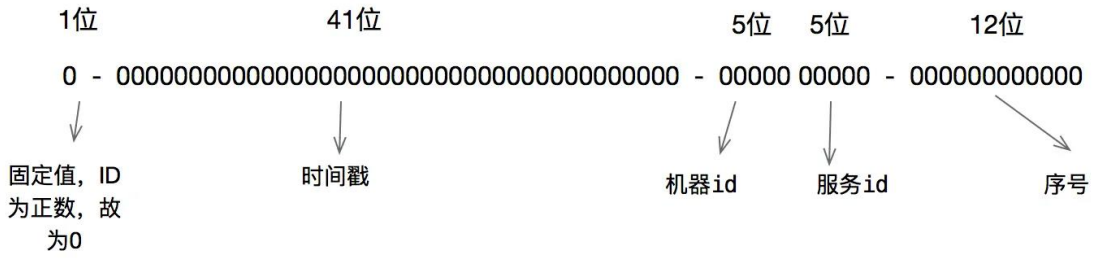
## 8.2 分布式 ID

https://zhuanlan.zhihu.com/p/107939861

分布式唯一 ID 的特性如下：

- 全局唯一：基本要求，必须保证生成的 ID 是全局性唯一的
- 有序性：生成的 ID 需要按某规则有序，便于数据库写入和排序
- 可用性
- 自主性：不依赖中心认证即可自行生成 ID
- 安全性：不暴露系统和业务的信息

snowflake 算法

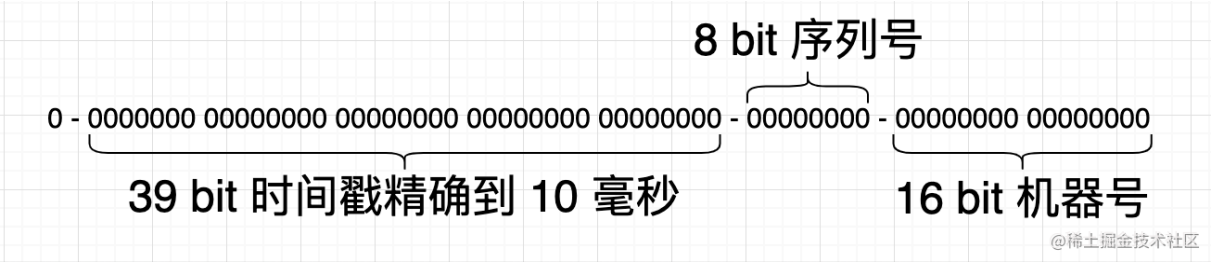



- 最高位是 0
- 41 位毫秒级时间戳：大概可以用 69 年
- 10 位机器码：可根据业务自行设定，如机房号+机器号、机器号+服务号等
- 12 位存储序号：同一时间戳内按递增序号区分，即每节点每毫秒 4096 个 ID。

缺点是：

- 强依赖机器时钟，如果机器上时钟回拨，会导致发号重复或服务不可用；
  - 解决办法：1. 时钟回拨很短则等待或报错 2. 匀几 bit 位作为时钟回拨位，发现时钟回拨则加一，达到最大位后从 0 开始循环。
- 分布式环境中每个节点时钟不一致可能导致 ID 不是全局递增

sonyflake 算法（项目用的）




- 最高位是 0
- 39 位以 10ms 为单位保存时间戳，最大 174 年
- 8 位作为序列号：每 10ms 最多 256 个
- 16 位作为机器号，默认机器私有 IP 的最后两位

如何解决时钟回拨问题的：sleep 等待时钟恢复到上一次生成 ID 的时间戳

数据库自增主键 ID

利用数据库自增主键的特性，向表中插入一条记录返回主键 ID 作为分布式 ID

缺点：DB 存在宕机风险，无法抗住高并发

基于数据库集群模式

数据库多主集群，每个 MySQL 实例都可以单独生成自增 ID。

会生成重复 ID 怎么办：设置不同的起始值和步长

优点：解决 DB 单点问题

缺点：不利于后续扩容，而且依旧无法满足高并发场景

数据库号段模式

号段模式可以理解为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，具体的业务服务将本号段，生成 1~1000 的自增 ID 并加载到内存。

采用版本号乐观锁方式更新，SQL 示例如下：

```
update id_generator set max_id = #{max_id+step}, version =
version + 1 where version = # {version} and biz_type = XXX
```

## 8.3 分布式事务

### 8.3.1 强一致性方案

#### 8.3.1.1 2PC-二阶段提交

二阶段提交算法思路可以概括为：参与者将操作成功或者失败通知协调者，再由协调者根据所有参与者的反馈决定各参与者是否要提交操作还是终止操作。

所谓两阶段是指：

**第一阶段：投票阶段**

事务协调者给参与者发送 Prepare 消息，参与者要么直接返回失败，要么在本地执行事务，写本地的 redo 和 undo 日志，但是不提交

**第二阶段：提交阶段**

协调者收到了参与者的失败消息或者超时，则向每一个参与者发送回滚（Rollback）消息，否则发送提交消息；参与者根据协调者的执行执行提交或回滚操作，释放所有事务处理过程中使用的锁资源。（注意：必须在最后阶段释放锁资源）

有两个前提：

- **网络在提交阶段的短时间内是可靠的**，即提交阶段不会丢失消息

- 因为网络分区、机器崩溃导致失联的**节点最终能够恢复，不会永久性处于失联状态。**

2PC 缺点：

1. **单点故障问题**：协调者等待参与者恢复可以有超时机制，可以允许参与者宕机，但是协调者宕机，则所有参与者必须等待。
2. **同步阻塞问题**：从投票阶段到提交阶段完成之间，资源是被锁住的。
3. **数据一致性问题**：不满足上述两个前提时，两阶段提交会出现一致性问题。
   - 网络不可靠：协调者提交阶段时，如果因为网络分区导致部分参与者已提交，部分未提交则数据不一致。
   - 节点宕机不恢复：数据肯定不一致。

#### 8.3.1.2 3PC-三阶段提交

为了解决 2PC 的单点故障、同步阻塞和数据一致性问题，引出了 3PC 协议

3PC 有两个改动点：

- **引入超时机制**：同时在协调者和参与者中引入超时机制
- **2PC 的准备阶段细分为两个阶段**

**第一阶段：CanCommit 阶段**

协调者向参与者发送 commit 请求，参与者如果可以提交则返回 Yes，否则返回 No

**第二阶段：PreCommit 阶段**

- 所有参与者都返回 Yes，执行事务的预执行，发送 commit
- 任何一个参与者返回了 No 或者等待超时，执行事务的中断，发送 abort

**第三阶段：doCommit 阶段**

该阶段进行真正的事务提交，也有两种情况

- 针对第二阶段的第一种情况，协调者向参与者发送事务提交请求
- 针对第二阶段的第二种情况，执行中断事务

优点：**解决了 2PC 的单点故障问题以及减少阻塞**，因为如果参与者没有及时收到协调者的信息之后，参与者会执行 commit，而不会一直持有事务资源并处于阻塞状态。

缺点：**对数据一致性没有任何改进**。如果第二阶段协调者发送的是 abort，但是由于网络问题，参与者等待超时后执行的 commit，就会数据不一致。

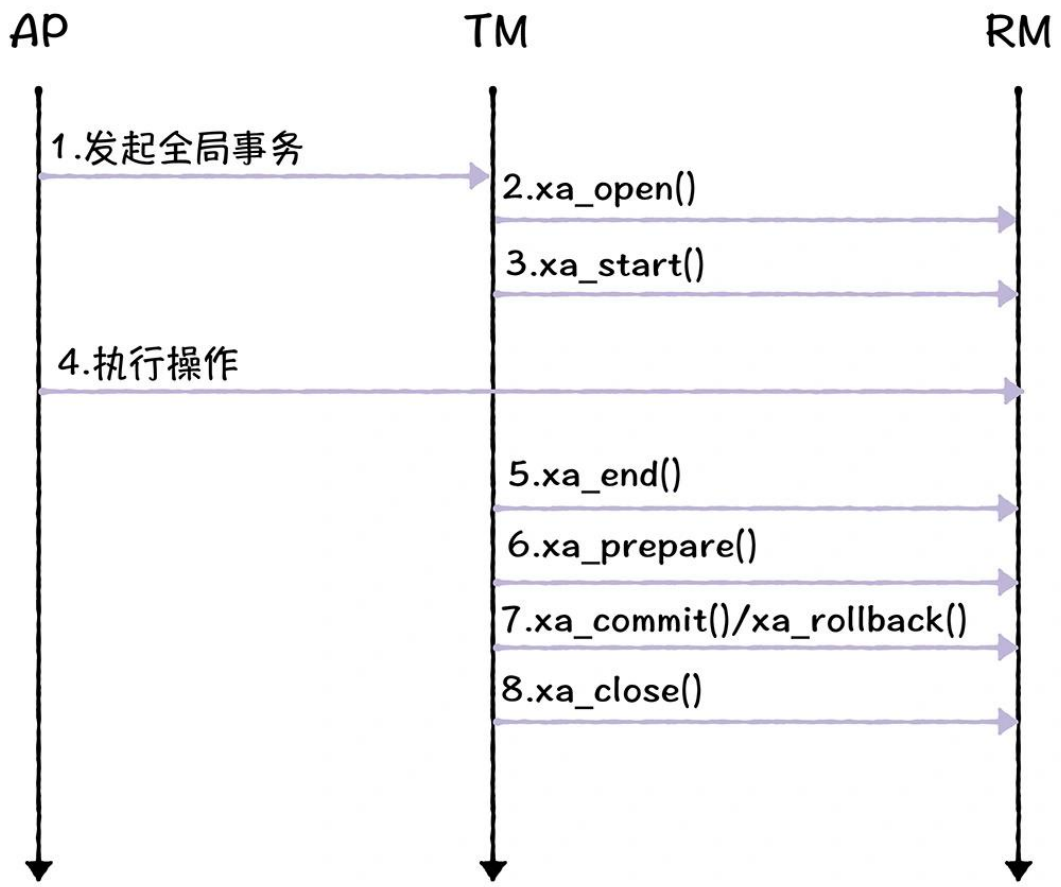
#### 8.3.1.3 XA 协议

MySQL 自带的 2PC 协议

首先了解下分布式事务处理模型 DTP，定义了三大组件：

- **AP**：应用程序，分布式事务的**发起者**
- **RM**：资源管理器，分布式事务的**参与者，**如数据库，应该具有事务提交或回滚能力
- **TM**：事务管理器，分布式事务的**协调者，**与每个 RM 通信，协调事务的提交和回滚，并协助进行故障恢复。

应用程序访问、使用资源管理器的资源，并通过事务管理器的事务接口（TX interface）定义需要执行的事务操作，然后事务管理器和资源管理器会基于 XA 规范，执行二阶段提交协议




1. AP（应用程序）联系 TM（事务管理器）发起全局事务；
2. TM 调用 xa_open() 建立与资源管理器的会话；

3. TM 调用 xa_start() 标记事务分支（Transaction branch）的开头；
4. AP 访问 RM（资源管理器），并定义具体事务分支的操作，比如更新一条数据记录（UPDATE executed_table SET status = true WHERE id=100）和插入一条数据记录（INSERT into operation_table SET id = 100, op = ‘get-cdn-log’）；
5. TM 调用 xa_end() 标记事务分支的结尾；
6. TM 调用 xa_prepare() 通知 RM 做好事务分支提交的准备工作，比如锁定相关资源，也就是执行二阶段提交协议的提交请求阶段；
7. TM 调用 xa_commit() 通知 RM 提交事务分支（xa_rollback() 通知 RM 回滚事务），也就是执行二阶段提交协议的提交执行阶段；
8. TM 调用 xa_close() 关闭与 RM 的会话。

**XA 是资源层面的底层分布式事务解决方案**

缺点：客观上延后了提交事务的时间点，延长了事务的执行时间，导致事务在访问共享资源时发生冲突和死锁的概率提高；因此 **XA 并发性能不理想**

## 8.3.2 最终一致性方案

### 8.3.2.1 本地事务状态表

**概念**：调用方请求外部系统前将待执行的事务流程及其状态信息存储到 DB 中，依赖 DB 本地事务的原子性保证本地事务和调用外部系统事务的一致性，**存储事务执行状态信息的表称为本地事务状态表**。

大致流程：

1. 事务状态信息存储到 DB 后，调用方才开始继续后面的流程，同时调用外部系统，并且每次调用成功后会更新相应的子事务状态，某一步失败则终止执行。
2. 后台运行一个定时任务，定期扫描事务状态表中未完成的子事务，并重新发起调用或者执行回滚，或者在失败重试指定次数后触发告警。

本地事务表的设计由业务方自己设定，可以多个子事务管理，也**可以只有一条记录，通过状态流转来控制程序调用不同的外部系统（**项目中任务中心使用的这种方式**）**。

### 8.3.2.2 可靠消息队列

利用消息中间件完成

事务发起方（消息生产方）将消息发送给消息中间件，事务参与方从消息中间件接收消息，事务发起方、事务参与方分别和消息中间件之间通过网络通信，由于网络的不确定性会导致分布式事务问题。

基于此方案需要解决以下问题：

- **本地事务和消息发送的原子性问题**：事务发起方在本地事务执行成功后必须将消息发送出去，否则丢弃消息。
- **事务参与方接收消息的可靠性**：事务参与方必须能从消息队列中接收到消息，如果消息接收失败则可以重复接收消息。
- **消息重复消费问题**：需要事务参与方实现消费幂等。

事务消息方案

基于事务消息，先预发送消息，向 MQ 中写入一条 Prepare 消息，执行业务逻辑 && 本地事务，向 MQ 发送 Confirm/Rollback 消息），

- 如果收到 Confirm 消息则提交事务消息，将消息转为对消费者可见，并开始投递；
- 如果收到 Rollback 消息，则删除之前的事务消息；
- 如果未收到消息，则会通过事务回查机制定时检查本地事务的状态，决定是否可以提交投递。

#### 8.3.2.3 最大努力通知

最简单的一种柔性事务，**适用于一些最终一致性时间敏感度低的业务，且被动方处理结果不影响主动方的处理结果**。如银行通知、商户通知等。

一般符合以下特点：

- **不可靠消息**：业务活动主动方，完成业务处理后向业务活动被动方发送消息，直到通知 N 次后不再通知，允许消息丢失（不可靠消息）。
- **定期校对**：业务活动被动方，根据定时策略，向业务活动主动方查询（主动方提供查询接口），恢复丢失的业务消息。

与可靠消息队列方案的区别：

- **解决方案思想不同**：可靠消息队列，发起通知方要保证将消息发出去，消息可靠性由发起通知方保证；最大努力通知，发起通知方尽最大努力将业务处理结果通知给接收通知方，但是消息可能接收不到，需要接收通知方主动调用发起通知方的接口查询业务处理结果，消息可靠性关键在接收通知方；
- **两者业务应用场景不同**：可靠消息队列关注的是交易过程的事务一致，以异步的方式完成交易；最大努力通知关注的是交易后的通知事务，即将交易结果可靠的通知出去。
- **技术解决方向不同**：可靠消息队列要解决消息从发出到接收的一致性，即消息发出并被接收到；最大努力通知无法保证消息从发出到接收的一致性，只提供消息接收的可靠性机制，可靠性机制指的是最大努力将消息通知给接收方，当消息无法被接收方接收时，由接收方主动查询消息（业务处理结果）。

#### 8.3.2.4 TCC

**TCC 天生适用于需要强隔离性的分布式事务**

具体实现：

业务侵入性较强，要求业务处理过程必须拆分为“预留业务资源”和“确认/释放消费资源”两个子过程，实现过程可分为三个阶段：

- **Try：尝试执行阶段**，完成所有业务可执行性的检查（保障一致性），并且预留好事务需要用到的所有业务资源（保障隔离性）。
- **Confirm：确认执行阶段**，不进行业务检查，直接使用 Try 阶段准备的资源来完成业务的处理，Confirm 阶段可能重复执行，需要满足幂等性。
- **Cancel：取消执行阶段**，释放 Try 阶段预留的业务资源。Cancel 阶段也可能会重复执行，同样需要满足幂等性。

## 8.4 分布式限流