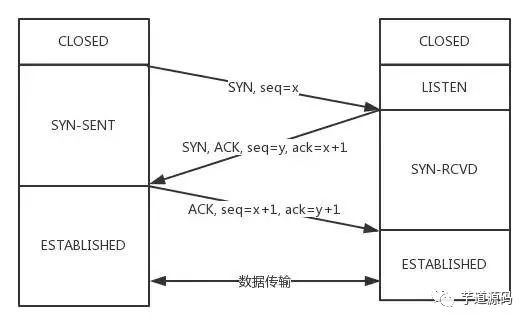
# TCP协议

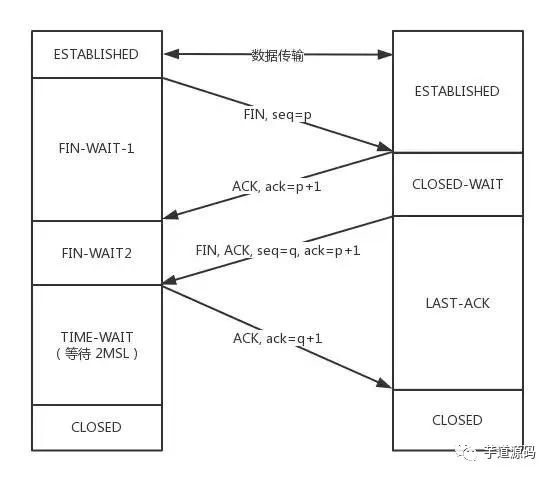
## 能不能说一说 TCP 和 UDP 的区别？

* TCP是一个面向连接的，可靠的，基于字节流的传输层协议
* UDP是一个面向无连接的传输层协议
* TCP的特性：
  + 面向连接。指的是客户端和服务器的连接，在双方通信之前，TCP需要三次握手建立连接，而UDP不需要
  + 可靠性。TCP会保证包按序到达，不允许半点差错，这是有状态。当意识到丢了包或者网络环境不佳，TCP会根据具体环境情况调整自己的行为，控制自己的发送速度或者重发，这是可控制。
  + 面向字节流。UDP的数据传输时基于数据报的。这是因为仅仅只是继承了IP层的特性，而TCP为了维护状态，将一个个IP包变成了字节流。
* UDP的特性：
  + 无状态
  + 不可控

## TCP三次握手的过程，为什么不是两次？四次？

* 三次握手的过程：从最开始双方都属于close状态。然后服务端开始监听某个端口，进入了listen状态。然后客户端主动发起连接，发送SYN，自己变成了SYN-SENT状态。服务端接收到，返回SYN和ACK，服务端变成了SYN-REVD。之后客户端再发送ACK给服务端，自己变成了established状态，服务端收到ACK之后，也变成established状态。（SYN是需要消耗一个序列号，凡是需要对端确认的，一定消耗TCP报文的序列号）
* 为什么不是两次握手：
  + 原因：无法确认客户端的接收能力。
  + 如果是两次，当客户端发了SYN报文想握手，但是因为网络原因，没有到达，TCP以为是包丢了，重试，建立连接。此时看上去没有什么问题，如果这个包在连接关闭之后到达了，由于是两次握手，服务端只要收到包就建立连接，造成了资源的浪费。
* 为什么不是四次握手：三次握手的目的是确认双方发送和接收的能力，四次也是可以的。但是为了解决问题，三次就够了。
* 三次握手的过程可以携带数据吗？第三次握手的时候可以携带。如果前两次可以携带数据，如果有人想攻击服务器，只需要在第一次握手的SYN报文中放入大量数据，服务端就必须消耗更多的时间和空间去处理数据，增加了服务器被攻击的风险。
* 

## 说说TCP四次挥手的过程

* 刚开始双方都处于ESTABLISHED状态，客户端要断开了，向服务器发送FIN报文，发送后客户端变成了FIN-WAIT-1状态。这时候客户端变成了half-close(半开关)状态，只能接收报文。服务端接收后向客户端确认，变成了close-wait状态。客户端收到了服务端的确认，变成了FIN-wait2状态。随后服务端向客户端发送FIN报文，自己进入last-ack状态。客户端收到服务端发来的FIN后，自己变成了TIME-WAIT状态，然后发送ACK给服务端。这个时候，客户端需要等待足够长的时间，一般是2个MSL，在这段时间内如果客户端没有收到服务端的重发请求，那么表示ACK成功到达，挥手结束，否则客户端重发ACK。
* 等待2MSL的意义：如果不等待，客户端直接关闭连接，当服务端还有很多数据需要发送给客户端时，此时有新的连接占用了这个端口，会造成数据的混乱。1个MSL确保四次挥手中主动关闭方最后的ACK报文最终能达到对端。1个SML确保对端没有收到ACK重传的FIN报文可以到达。
* 为什么是四次挥手而不是三次：因为服务端收到FIN后，不会立即返回FIN，需要必须等到服务端所有的报文都发送完毕，才能发FIN。因此先发一个ACK表示服务端已经收到客户端的FIN，延迟一段时间再发FIN，这就造成了四次挥手。如果是三次挥手，ACK和FIN合并，长时间的延迟会认为客户端的FIN没有到达而重传。
* 

## 说说半连接队列和SYN Flood攻击的关系

* 三次握手前，服务端从close变成listen时，同时在内部建立两个队列：半连接队列（SYN队列）和全连接队列（ACCEPT队列）。当客户端发送SYN给服务端，服务端回复SYN和ACK之后，由状态listen变成SYN-RCVD，此时连接进入SYN队列。当客户端返回ACK之后，三次握手完成，这个连接在等待被取走之前，会进入ACCEPT队列。
* SYN Flood攻击：利用客户端再短时间内伪造大量不存在的IP地址，向服务端发送SYN。会造成两个后果：
  + 处理大量的SYN包并返回ACK，会有大量的连接处于SYN-RCVD状态，占满半连接队列，无法处理请求。
  + 由于不存在的IP，服务端短时间内收不到ACK，会一直重发数据，直到耗尽资源。
* 如何应对SYN Flood攻击：
  + 增加 SYN 连接，也就是增加半连接队列的容量。
  + 减少 SYN + ACK 重试次数，避免大量的超时重发。
  + 利用 SYN Cookie 技术，在服务端接收到SYN后不立即分配连接资源，而是根据这个SYN计算出一个Cookie，连同第二次握手回复给客户端，在客户端回复ACK的时候带上这个Cookie值，服务端验证 Cookie 合法之后才分配连接资源。

## TCP的拥塞控制

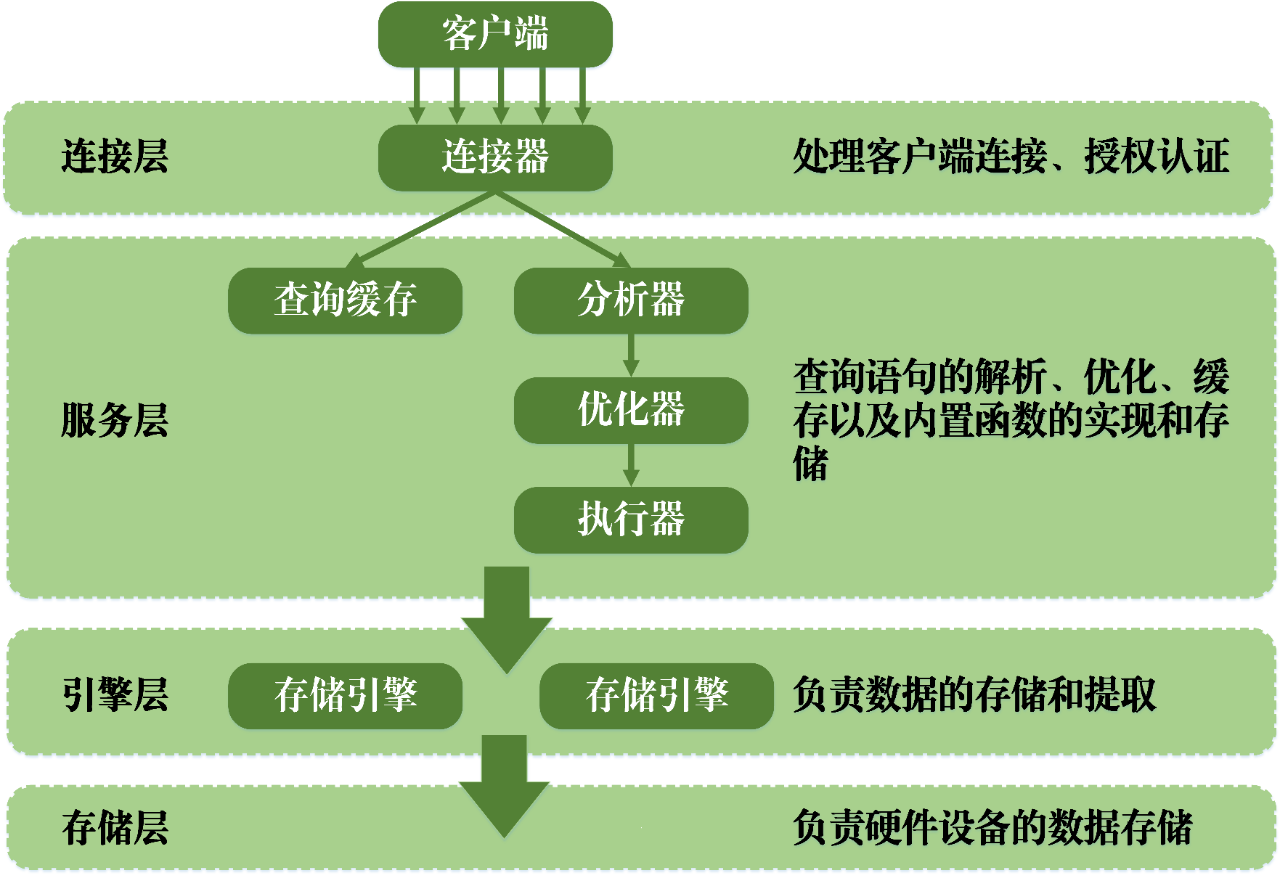
* 对用拥塞控制来说，TCP每条连接都需要维护两个核心状态：
  + 拥塞窗口：指目前自己还能传输的数据量大小。
  + 慢启动阙值
* 涉及的算法：
  + 慢启动：刚进入传输数据的时候，不知道现在的网络是稳定还是拥堵的，如果发包太急，会疯狂丢包，造成雪崩的网络灾难。因此，拥塞控制首先采用一种保守的算法来慢慢地使用网络。这种算法叫慢启动，运行过程如下：
    - 首先，三次握手，双方宣告自己的接受窗口大小
    - 双方初始化自己的拥塞窗口大小
    - 在开始传输的一段时间，发送端每收到一个ACK，拥塞窗口加1，也就是说，每经过一个RTT，cwnd翻倍。达到一个阙值就停止。这个阙值叫做慢启动阙值。
  + 拥塞避免
  + 快速重传
    - 快速重传：在 TCP 传输的过程中，如果发生了丢包，即接收端发现数据段不是按序到达的时候，接收端的处理是重复发送之前的 ACK。比如第 5 个包丢了，即使第 6、7 个包到达的接收端，接收端也一律返回第 4 个包的 ACK。当发送端收到 3 个重复的 ACK 时，意识到丢包了，于是马上进行重传，不用等到一个 RTO 的时间到了才重传。这就是快速重传，它解决的是是否需要重传的问题。
    - 选择性重传：在收到发送端的报文后，接收端回复一个 ACK 报文，那么在这个报文首部的可选项中，就可以加上SACK这个属性，通过left edge和right edge告知发送端已经收到了哪些区间的数据报。因此，即使第 5 个包丢包了，当收到第 6、7 个包之后，接收端依然会告诉发送端，这两个包到了。剩下第 5 个包没到，就重传这个包。这个过程也叫做选择性重传(SACK，Selective Acknowledgment)，它解决的是如何重传的问题。
  + 快速恢复：发送端收到三次重复ACK之后，发现丢包，觉得网络有些拥塞，会进入快速恢复阶段。发送端将拥塞阙值降低为cwnd的一办，cwnd的大小变为拥塞阙值，cwnd线性增加

# Mysql

## 关系型数据库与NoSQL的优缺点

* 关系型数据库
  + 优点
    - 二维表格，关系模式容易理解
    - 可维护性高，统一使用表结构存储
    - 采用通用的查询语言SQL，学习成本低
    - 通过SQL之间可以实现多个表之间非常复杂的查询
  + 缺点
    - 读写性能较差，尤其是海量数据的读写
    - 以固定的表结构存储，灵活性较差
    - 难以满足高并发的读写需求，磁盘IO瓶颈
* NoSQL
  + 优点
    - 格式灵活，没有固定的存储格式
    - 可扩展性高，容易进行水平扩展
    - 无需SQL解析层，读写性能高
  + 缺点
    - 不提供SQL支持，学习和使用成功高
    - 无事务处理，不支持ACID特性
    - 数据结构相对复杂，复杂查询方面欠缺

## Mysql的逻辑架构

* 

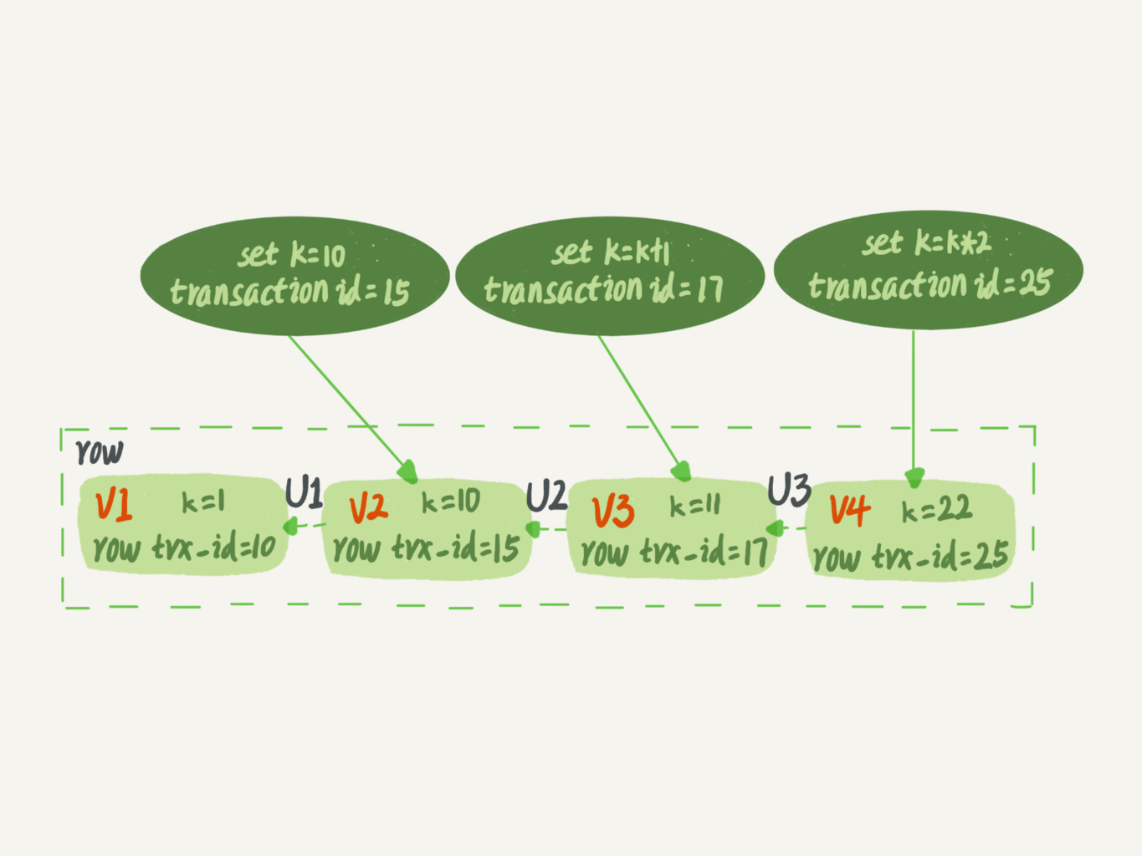
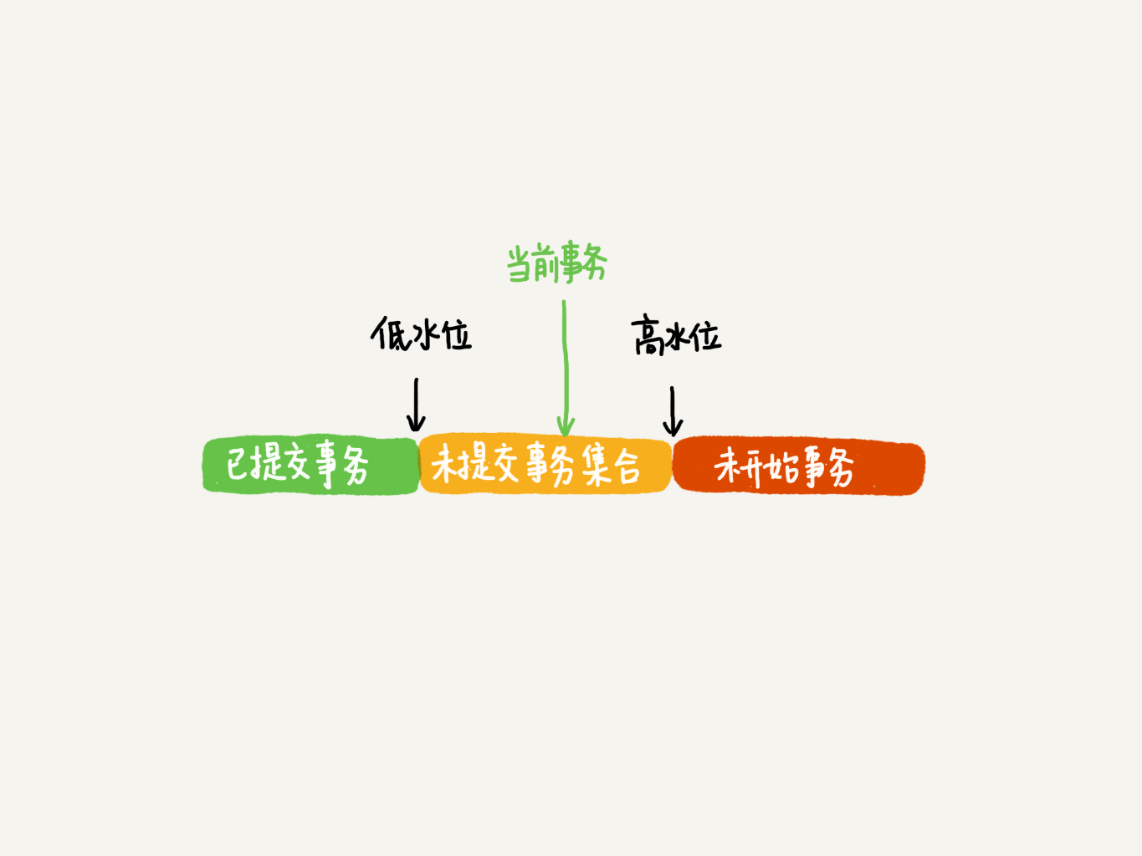
## 聚簇索引/非聚簇索引

* InnoDB是聚簇索引
  + 聚簇索引的主索引的叶子结点存储的是键值对应的数据本身，辅助索引的叶子结点存储的是键值对应的数据的主键键值。因此主键的值长度越小越好，类型越简单越好。
  + 聚簇索引的数据和主键索引存储在一起
  + 聚簇索引的数据是根据主键的顺序保存。因此适合按主键索引的区间查找，可以有更少的磁盘I/O，加快查询速度。但是也是因为这个原因，聚簇索引的插入顺序最好按照主键单调的顺序插入，否则会频繁的引起页分裂，严重影响性能。
  + 在InnoDB中，如果只需要查找索引的列，就尽量不要加入其它的列，这样会提高查询效率
* MyISAM是非聚簇索引
  + MyISAM存储引擎采用的是非聚簇索引，非聚簇索引的主索引和辅助索引几乎是一样的，只是主索引不允许重复，不允许空值，他们的叶子结点的key都存储指向键值对应的数据的物理地址
  + 非聚簇索引的数据表和索引表是分开存储的
  + 非聚簇索引中的数据是根据数据的插入顺序保存。因此非聚簇索引更适合单个数据的查询。插入顺序不受键值影响
  + 只有在MyISAM中才能使用FULLTEXT索引。(mysql5.6以后innoDB也支持全文索引)

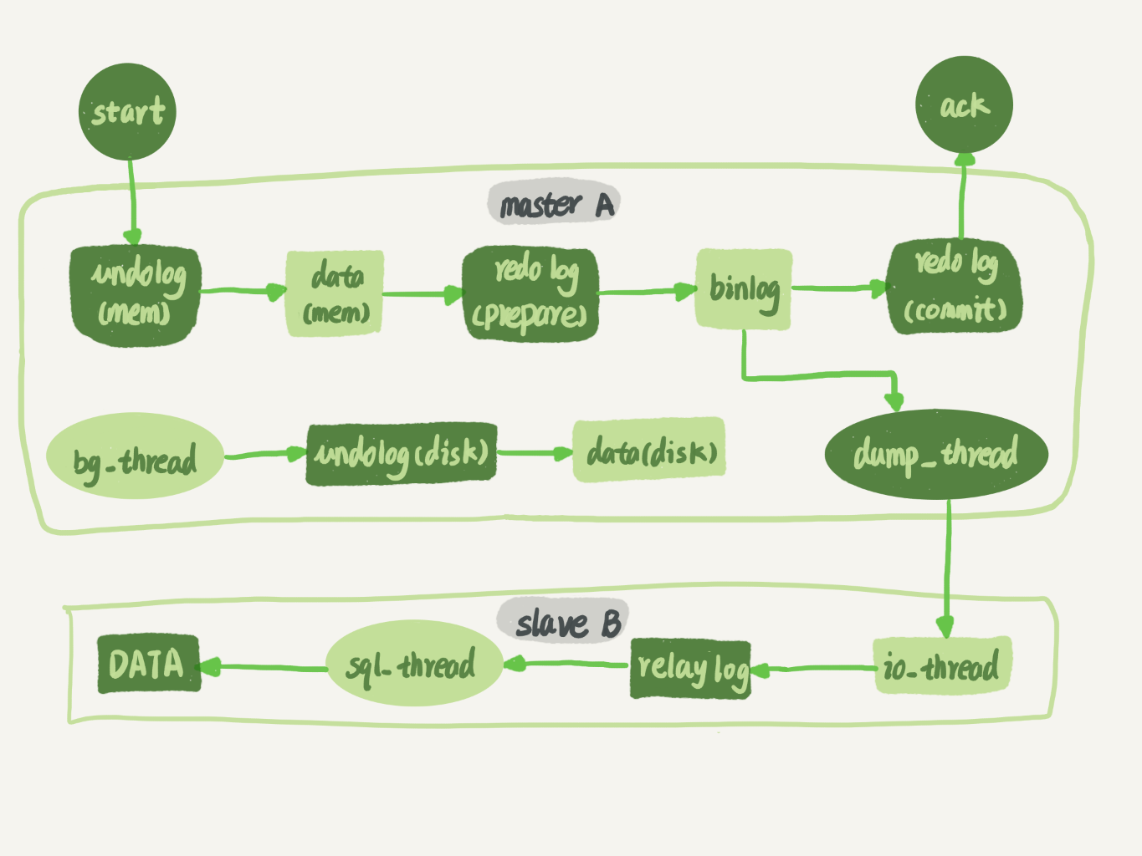
## Mysql索引底层实现

* 使用B+Tree树，非叶子节点不存储数据。主键索引的叶子节点存储是数据，普通索引叶子节点存储是主键值。
* B+Tree和B-Tree的区别
  + B+Tree
    - 所有叶子节点之间都有一个链指针
    - 数据记录都存放在叶子节点中
    - 所有的叶子结点中包含了全部关键字的信息，非叶子节点只存储键值信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接，所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。 (而B树的非终节点也包含需要查找的有效信息)
  + B-Tree
    - 每个节点都存储key和data，所有节点组成这棵树，并且叶子节点指针为null，叶子结点不包含任何关键字信息
  + 为什么用B+Tree而不用B-Tree
    - 因为B-Tree的每个节点都存储了数据，当数据量比较大时，磁盘IO一次读出的数据量大小是固定的，单个数据变大，每次读出的数据就少，增加了磁盘IO

## MVCC，InnoDB中MVCC的实现

* 基本原理：MVCC的实现，通过保存数据在某个时间点的快照来实现的。这意味着一个事务无论运行多长时间，在同一个事务里能够看到数据一致的视图。根据事务开始的时间不同，同时也意味着在同一个时刻不同事务看到的相同表里的数据可能是不同的。
* 基本特征：
  + 每行数据都存在一个版本，每次数据更新时都更新版本
  + 修改时Copy出当前版本随意修改，各个事务直接无干扰
  + 保存时比较版本号，如果成功，则覆盖，失败则放弃
* InnoDB存储引擎MVCC的实现
  + 
  + InnoDB 里面每个事务有一个唯一的事务 ID，叫作 transaction id。它是在事务开始的时候向 InnoDB 的事务系统申请的，是按申请顺序严格递增的。而每行数据也都是有多个版本的。每次事务更新数据的时候，都会生成一个新的数据版本，并且把 transaction id 赋值给这个数据版本的事务 ID，记为 row trx\_id。同时，旧的数据版本要保留，并且在新的数据版本中，能够有信息可以直接拿到它。也就是说，数据表中的一行记录，其实可能有多个版本 (row)，每个版本有自己的 row trx\_id。上图中的虚线表示undo log。
  + 
    - 对于当前事务的启动瞬间，一个数据版本的row\_id，有以下几种可能
      * 如果落在绿色部分，表示这个版本是已提交的事务或者是当前事务自己生成的，这个数据是可见的；
      * 如果落在红色部分，表示这个版本是由将来启动的事务生成的，是肯定不可见的；
      * 如果落在黄色部分，那就包括两种情况
        + 若 row trx\_id 在数组中，表示这个版本是由还没提交的事务生成的，不可见；
        + 若 row trx\_id 不在数组中，表示这个版本是已经提交了的事务生成的，可见。
* InnoDB 的行数据有多个版本，每个数据版本有自己的 row trx\_id，每个事务或者语句有自己的一致性视图。普通查询语句是一致性读，一致性读会根据 row trx\_id 和一致性视图确定数据版本的可见性。
  + 对于可重复读，查询只承认在事务启动前就已经提交完成的数据；
  + 对于读提交，查询只承认在语句启动前就已经提交完成的数据；
  + 在可重复读隔离级别下，只需要在事务开始的时候创建一致性视图，之后事务里的其他查询都共用这个一致性视图；
  + 在读提交隔离级别下，每一个语句执行前都会重新算出一个新的视图。

## 主从复制流程

* 
* 事务日志同步的完整过程：
  + 在备库 B 上通过 change master 命令，设置主库 A 的 IP、端口、用户名、密码，以及要从哪个位置开始请求 binlog，这个位置包含文件名和日志偏移量。
  + 在备库 B 上执行 start slave 命令，这时候备库会启动两个线程，就是图中的 io\_thread 和 sql\_thread。其中 io\_thread 负责与主库建立连接。
  + 主库 A 校验完用户名、密码后，开始按照备库 B 传过来的位置，从本地读取 binlog，发给 B。
  + 备库 B 拿到 binlog 后，写到本地文件，称为中转日志（relay log）。
  + sql\_thread 读取中转日志，解析出日志里的命令，并执行。
* 主从复制的好处：
  + 通过复制实现数据的异地备份，当主数据库故障时，可切换从数据库，避免数据丢失。
  + 可实现架构的扩展，当业务量越来越大，I/O访问频率过高时，采用多库的存储，可以降低磁盘I/O访问的频率，提高单个机器的I/O性能。
  + 可实现读写分离，使数据库能支持更大的并发。
  + 实现服务器的负载均衡，通过在主服务器和从服务器之间切分处理客户查询的负荷。

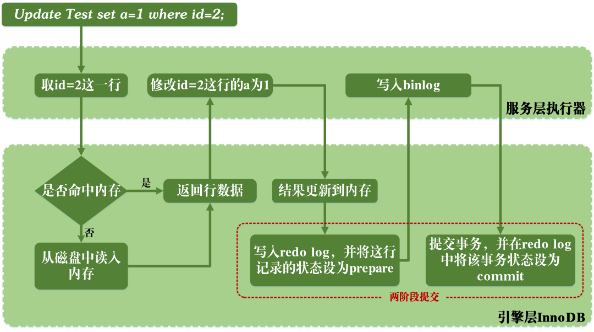
## 间隙锁是啥？它怎么避免出现幻读的？它引入了什么新的问题？

* 什么是幻读：幻读是指在同一个事务中，存在前后两次查询同一个范围的数据，但是第二次查询却看到了第一次查询没看到的行。出现的场景：
  + 事务的隔离级别为可重复读，且是当前读
  + 幻读仅专指新插入的行
* 为什么会出现幻读：行锁只能锁定存在的行，针对新插入的操作没有限定
* 间隙锁是专门用于解决幻读这种问题的锁，它锁的了行与行之间的间隙，能够阻塞新插入的操作
* 间隙锁的引入也带来了一些新的问题，比如：降低并发度，可能导致死锁。

## Mysql隔离级别，ACID特性

* 读未提交（read uncommitted）
  + 读未提交是指，一个事务还没提交时，它做的变更就能被别的事务看到。
* 读提交（read committed）
  + 读提交是指，一个事务提交之后，它做的变更才会被其他事务看到。
* 可重复读（repeatable read）
  + 可重复读是指，一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。当然在可重复读隔离级别下，未提交变更对其他事务也是不可见的。
* 串行化（serializable ）
  + 串行化，顾名思义是对于同一行记录，“写”会加“写锁”，“读”会加“读锁”。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。
* 原子性：使用 undo log来实现，如果事务执行过程中出错或者用户执行了rollback，系统通过undo log日志返回事务开始的状态。
* 持久性：使用 redo log来实现，只要redo log日志持久化了，当系统崩溃，即可通过redo log把数据恢复。
* 隔离性：通过锁以及MVCC来实现。
* 一致性：通过回滚、恢复以及并发情况下的隔离性，从而实现一致性。

## Mysql日志

* 日志类型
  + Redo log是物理日志
  + Binlog是逻辑日志
* 逻辑架构层
  + Redo log是InnoDB引擎特有的
  + Binlog是mysql server层的。所有引擎共享
* 文件空间
  + Redo log日志文件的空间固定
  + Binlog日志文件的空间不固定，写完会切换下一个文件
* 写入方式
  + Redo log日志是循环写入和擦除
  + Binlog日志是追加写入，不会覆盖已写入文件
* 

# Kafka(待完善)

## Kafka的用途有哪些？使用场景如何？

## Kafka中的ISR、AR又代表什么？ISR的伸缩又指什么

## Kafka中的HW、LEO、LSO、LW等分别代表什么？

## Kafka中是怎么体现消息顺序性的？

## Kafka中的分区器、序列化器、拦截器是否了解？它们之间的处理顺序是什么？

## Kafka生产者客户端的整体结构是什么样子的？

## Kafka生产者客户端中使用了几个线程来处理？分别是什么？

## Kafka的旧版Scala的消费者客户端的设计有什么缺陷？

## “消费组中的消费者个数如果超过topic的分区，那么就会有消费者消费不到数据”这句话是否正确？如果不正确，那么有没有什么hack的手段？

## 消费者提交消费位移时提交的是当前消费到的最新消息的offset还是offset+1?

## 有哪些情形会造成重复消费？

## 那些情景下会造成消息漏消费？

## KafkaConsumer是非线程安全的，那么怎么样实现多线程消费？

## 简述消费者与消费组之间的关系

## 当你使用kafka-topics.sh创建（删除）了一个topic之后，Kafka背后会执行什么逻辑？

## topic的分区数可不可以增加？如果可以怎么增加？如果不可以，那又是为什么？

## topic的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么？

## 创建topic时如何选择合适的分区数？

## Kafka目前有那些内部topic，它们都有什么特征？各自的作用又是什么？

## 优先副本是什么？它有什么特殊的作用？

## Kafka有哪几处地方有分区分配的概念？简述大致的过程及原理

## 简述Kafka的日志目录结构

## Kafka中有那些索引文件？

## 如果我指定了一个offset，Kafka怎么查找到对应的消息？

## 如果我指定了一个timestamp，Kafka怎么查找到对应的消息？

## 聊一聊你对Kafka的Log Retention的理解

## 聊一聊你对Kafka的Log Compaction的理解

## 聊一聊你对Kafka底层存储的理解（页缓存、内核层、块层、设备层）

## 聊一聊Kafka的延时操作的原理

## 聊一聊Kafka控制器的作用

## 消费再均衡的原理是什么？（提示：消费者协调器和消费组协调器）

## Kafka中的幂等是怎么实现的

## Kafka中的事务是怎么实现的

## Kafka中有那些地方需要选举？这些地方的选举策略又有哪些？

## 失效副本是指什么？有那些应对措施？

## 多副本下，各个副本中的HW和LEO的演变过程

## 为什么Kafka不支持读写分离？

## Kafka在可靠性方面做了哪些改进？（HW, LeaderEpoch）

## Kafka中怎么实现死信队列和重试队列？

## Kafka中的延迟队列怎么实现

## Kafka中怎么做消息审计？

## Kafka中怎么做消息轨迹？

## Kafka中有那些配置参数比较有意思？聊一聊你的看法

## Kafka中有那些命名比较有意思？聊一聊你的看法

## Kafka有哪些指标需要着重关注？

## 怎么计算Lag？(注意read\_uncommitted和read\_committed状态下的不同)

## Kafka的那些设计让它有如此高的性能？

## Kafka有什么优缺点？

## 还用过什么同质类的其它产品，与Kafka相比有什么优缺点？

## 为什么选择Kafka?

## 在使用Kafka的过程中遇到过什么困难？怎么解决的？

## 怎么样才能确保Kafka极大程度上的可靠性？

## 聊一聊你对Kafka生态的理解

# linux

## 工作中常用的几个命令

* 是否频繁gc：jstat -gc pid 1000(表示1秒钟采样一次)
* 是否频繁上下文切换：vmstat

# 分布式事务

## 2PC(二阶段提交)

* 有一个TM(事务管理器，也称为协调者)和多个RM(资源管理器，也称为参与者)，第一阶段首先TM通知各个RM准备他们的事务，然后RM把事务的状态改为可提交。第二阶段TM根据RM返回的结果，决定是提交事务还是回滚事务。
* 缺点：
  + 单点故障：由于TM的重要性，一旦TM发生故障，RM会一直阻塞。如果在第二阶段，TM发生故障，RM会一直处于锁定事务资源的状态中，而无法继续完成事务。
  + 同步阻塞：由于所有节点在执行操作时都是阻塞，当RM占有公共资源时，其他第三方因访问不到公共资源处于阻塞状态。
  + 数据不一致：在第二阶段中，当TM向RM发送提交事务的请求后。由于网络原因或者TM发生了故障，一部分RM收到了提交事务的请求。其他RM未接收到请求无法提交事务。造成数据的不一致

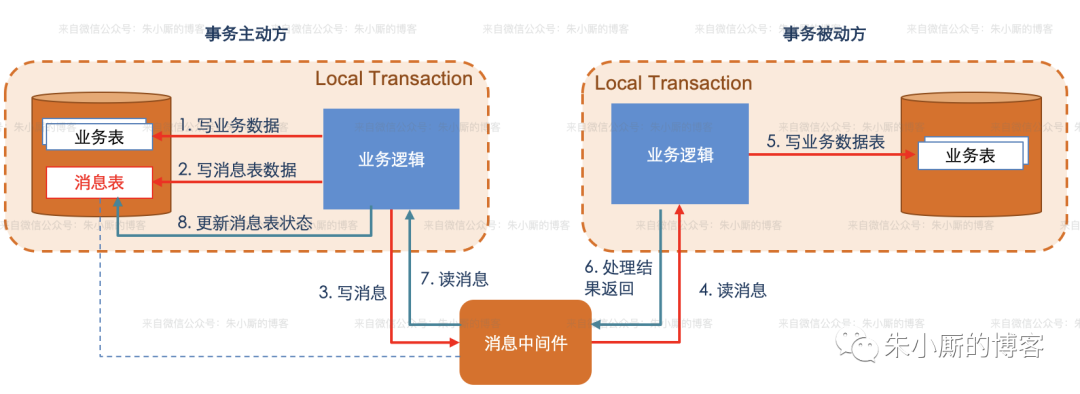
## 3PC(3阶段提交)

* 对于2PC，改动如下：
  + 引入超时机制。在TM和RM直接都引入了超时机制。
  + 在2PC中间插入了一个准备阶段，使得原先在两阶段提交中，参与者在投票之后，由于协调者发生崩溃或错误而导致参与者处于无法知晓是否提交或者中止的“不确定状态”所产生的可能相当长的延时的问题得以解决。
* 第一阶段CanCommit，TM向RM发送commit请求，如果RM可以提交返回yes响应，否则返回no响应。
* 第二阶段PreCommit
  + 假如TM收到RM的响应都是yes ,TM向RM发送PreCommit请求，TM进入Prepared阶段。RM收到PreCommit请求后，会执行事务操作，并将undo 和redo信息记录到事务日志中。如果RM成功执行了事务操作，则返回ACK响应，同时等待最终操作。
  + 如果有任何一个RM向TM响应了no，或者等待超时后，TM都没有收到RM的响应，则进行事务的中断。TM向RM发送Abort请求。RM收到TM的Abort请求后(或者超时，或者没有收到TM发送的消息)，执行事务的中断。
* 第三阶段DoCommit
  + 执行提交：
    - TM收到RM的第二阶段ACK响应后，TM进入提交状态，并向所有的RM发送DoCommit请求。RM在接收到DoCommit请求后，正式执行事务的提交，并释放所有资源，并向TM发送ACK响应，TM收到RM的ACK响应，完成事务。
  + 中断事务
    - TM没有收到RM的第二阶段ACK响应，TM向所有的RM发送Abort请求，RM接收到Abort请求后，用第二阶段记录的undo信息来执行事物的回滚操作，完成后释放所有事务资源。并向TM发送ACK响应。TM收到RM的ACK响应，执行事务的中断。
  + 缺点
    - 数据不一致：在第三阶段中，当TM向RM发送提交事务的请求后。由于网络原因或者TM发生了故障，一部分RM收到了提交事务的请求。其他RM未接收到请求无法提交事务。会造成数据的不一致

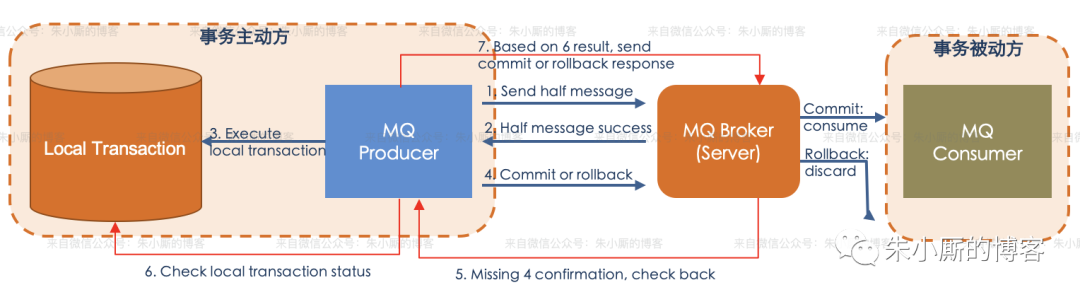
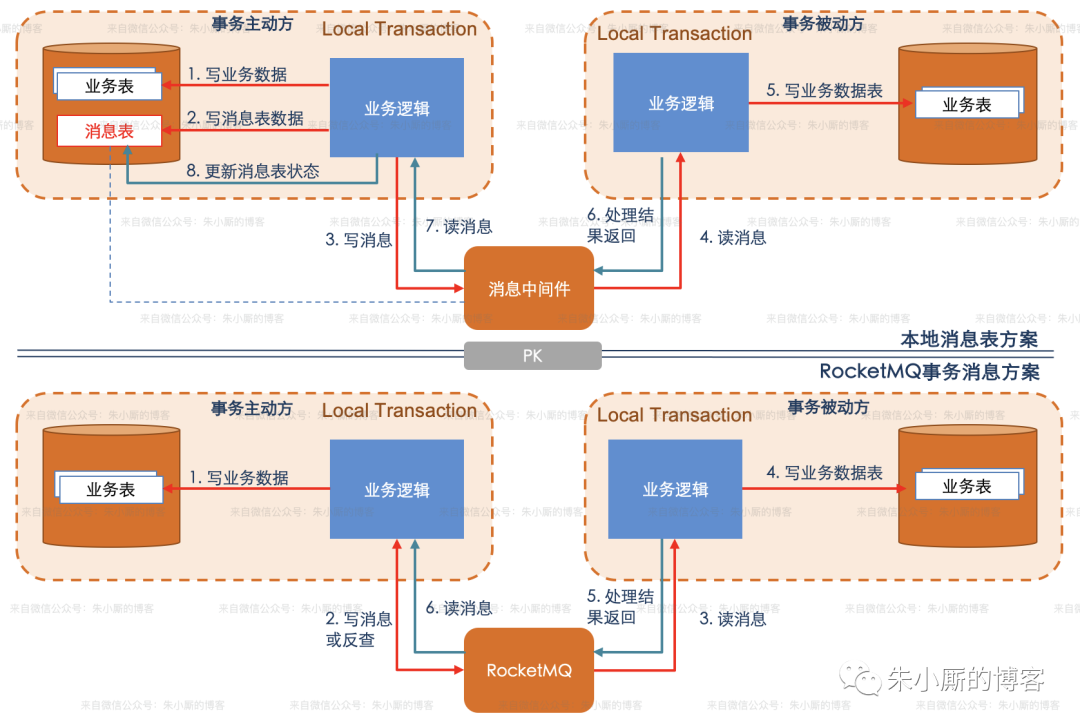
## TCC

* 主业务服务首先开启事务，主业务服务向事务管理器申请启动分布式事务主业务活动，然后针对要调用的从业务服务，主业务活动先向事务管理器注册从业务活动，然后调用从业务服务的Try接口，当所有的从业务服务的Try接口都调用成功，主业务服务提交本地事务，若调用失败，主业务服务回滚本地事务，若主业务服务提交本地事务，则TCC模型分别调用所有从业务服务的Confirm接口。若主业务服务回滚本地事务，则调用Cancel接口，所有从业务服务的Confirm或者Cancel执行成功后，事务结束。

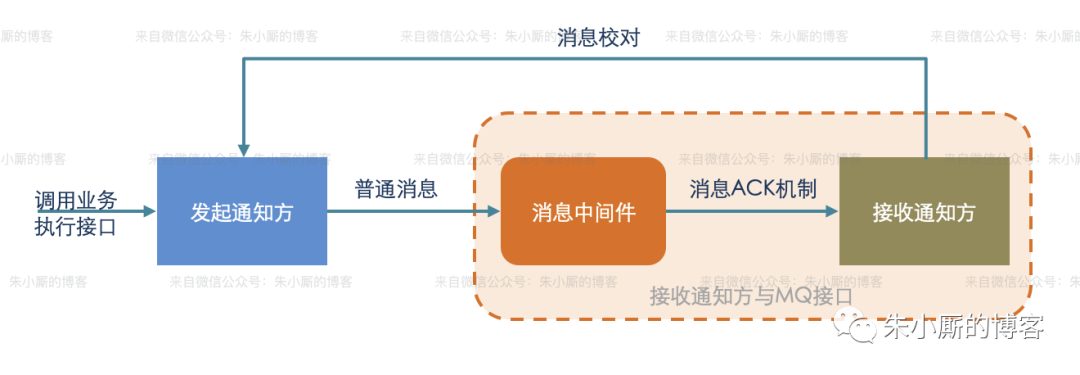
## 本地消息表

* 

## 消息事务

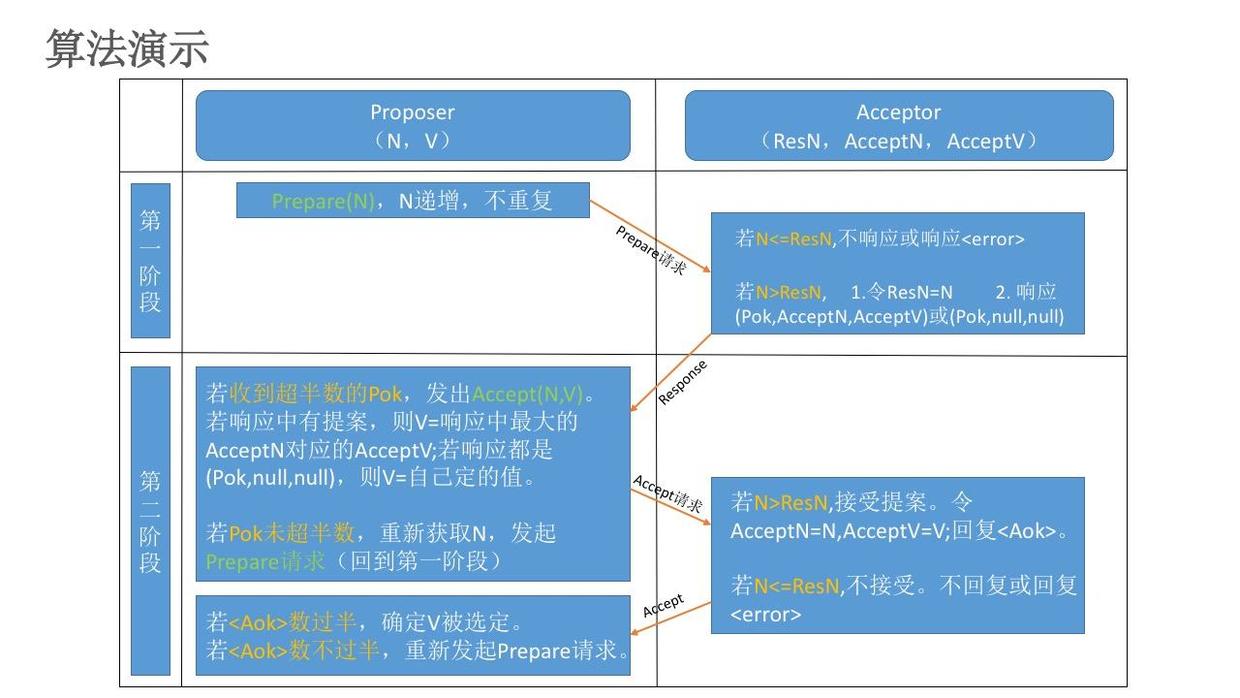
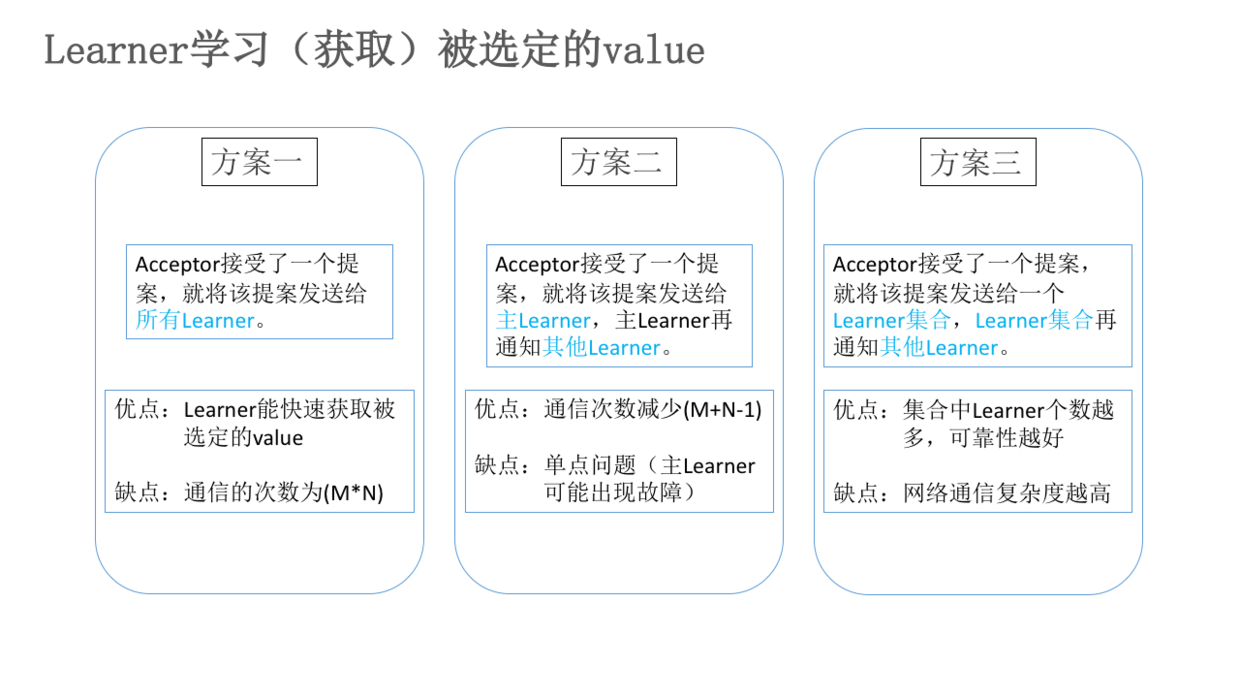
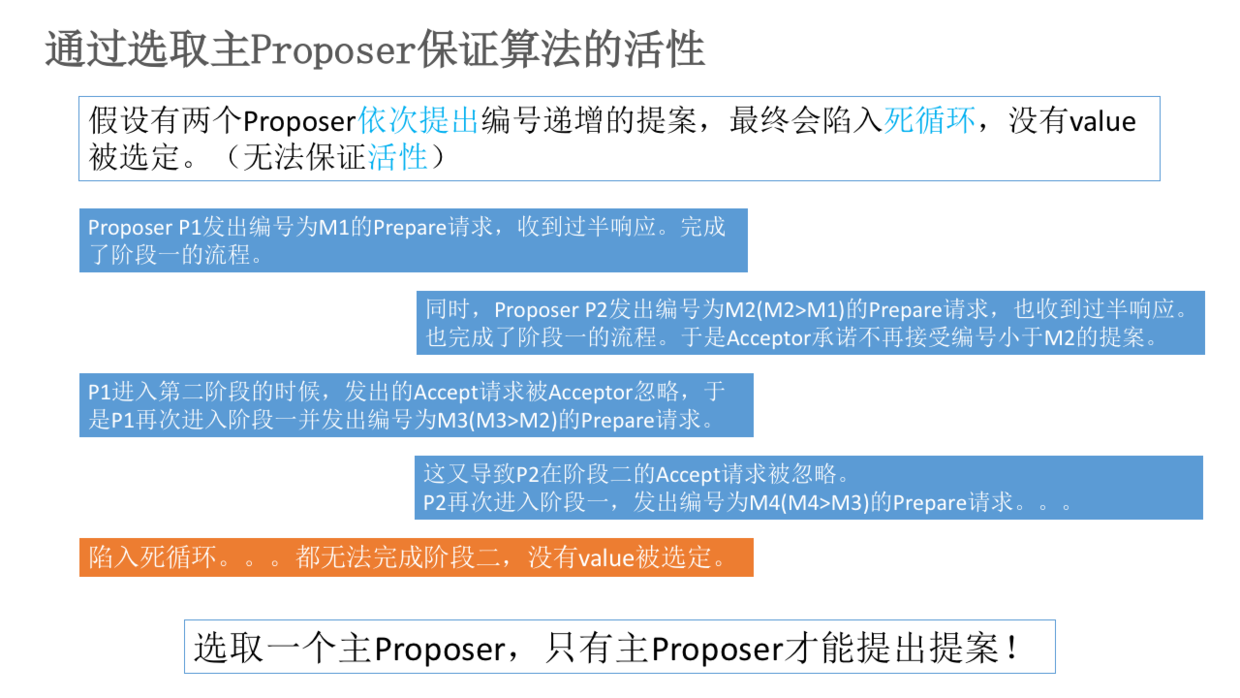
* 
* 

## 最大努力通知

* 
* 发起通知方将通知发给MQ，接收通知方监听 MQ 消息。接收通知方收到消息后，处理完业务回应ACK。接收通知方若没有回应ACK，则 MQ 会间隔 1min、5min、10min 等重复通知。接受通知方可调用消息校对接口，保证消息的一致性。

# Paxos算法

## Paxos算法描述

* 三种角色
  + Proposer：提出提案，提案信息包括提案编号N和提议的V
  + Acceptor：收到提案后可以接受提案
  + Learners：只能学习被批准的提案
* 阶段一：
  + Proposer选择一个提案编号N，然后想半数以上的Acceptor发送编号为N的Prepare请求
  + 如果一个Acceptor收到一个编号为N的Prepare请求，且N大于该Acceptor已经响应过的所有的Prepare请求的编号，那么它就会将它已经接受过的编号的最大提案作为响应返回给Proposer，同时承诺不在接受任何小于编号N的提案。
* 阶段二
  + 如果Proposer收到半数以上Acceptor对其发出的编号为N的Prepare请求的响应。那么它就会发送一个针对【N，V】提案的Accept请求给半数以上的Acceptor。V就是收到的响应中编号最大的提案的value，如果响应中不包含任何提案，那么V由Proposer自己决定。
  + 如果Acceptor收到一个针对编号N的提案的Accept请求，只要该Acceptor没有对编号大于N的Prepare请求做出响应，它就接受该提案。
* 
* Learner学习被选定的value
  + 
* 如何保证Paxos算法的活性
  + 

# 其他

1. 聊项目，画项目架构图，画一个用户从发起请求 到接收到响应 中间经过哪些服务 每个服务做什么事情 的流程图
2. 讲项目中的难点、挑战，你是如何解决的
3. redis 中有几种类型 & 各自底层怎么实现的 & 项目中哪个地方用了什么类型，怎么使用的
4. redis如何实现分布式锁，zk如何实现分布式锁，两者的区别。如果service还没执行完，分布式锁在redis中已经过期了，怎么解决这种问题
5. synchronized底层实现，加在方法上和加在同步代码块中编译后的区别、类锁、对象锁
6. 锁升级的过程
7. java运行时区域 及 各个区域的作用、对GC的了解、java内存模型 及 为什么要这么设计
8. 对索引的理解，组合索引，索引的最佳实践
9. countDownLatch用过没有，在项目中如何使用的，对aqs 的了解
10. 写生产者消费者问题，考虑高并发的情况，可以使用Java 类库，白纸写代码
11. 设计一个发号器，考虑集群和高并发的情况，要求发号器生成的id是递增趋势，通过id可以区分出来是今天生成的id还是昨天生成的id，但是生成的id中不能直接带有日期，要具有一定的混淆功能，白纸写代码
12. 一个二位数组，每个元素都可以往上下左右四个方向走，寻找最长递增路径。如下图所示，最长递增路径即红色字体路径。白纸写代码。
13. 数据库和缓存的一致性问题。先更新数据库，再更新缓存，若更新完数据库了，还没有更新缓存，此时有请求过来了，访问到了缓存中的数据，怎么办？
14. 谈下对GC的了解，何为垃圾，有哪些GC算法，有哪些垃圾回收器，cms和g1的区别，emm，还有一个直击灵魂的问题，看过cms的源码吗，笑cry
15. 有没有排查过线上oom的问题，如何排查的
16. 有没有使用过jvm自带的工具，如何使用的
17. 假设有下图所示的一个full gc 的图，纵向是内存使用情况，横向是时间，你如何排查这个full gc的问题，怎么去解决你说出来的这些问题【原创公众号：Bella的技术轮子】
18. 说说对java中集合类的理解，项目中用过哪些，哪个地方用的，如何使用的
19. 对CAS的理解，CAS带来的问题，如何解决这些问题
20. volatile底层、synchronized底层、锁升级的过程、MESI
21. ehcache支持哪些缓存
22. juc有研究没有，讲一讲
23. 聊项目，画项目架构图，画一个用户从发起请求 到接收到响应 中间经过哪些服务 每个服务做什么事情 的流程图
24. 讲项目中的难点、挑战，如何解决的，项目这一块会问的特别细
25. 如何保证RocketMQ 消息的顺序性，如何解决重复消费问题
26. 项目中如何保证接口的幂等操作
27. 讲一讲对redis 的了解，项目中如何使用的，哪个地方使用的，为什么要使用
28. 哨兵机制、redis两种备份方式的区别，项目中用的哪种，为什么
29. 讲一讲对分布式锁的了解
30. 项目中系统监控怎么做的
31. 如何理解Spring中的AOP 和 IOC，以及DI，读过Spring源码没有
32. 读过MyBatis源码没有
33. 说一个你了解最多的框架，说出你的理解
34. 如何理解分布式事务，为什么会出现这个问题，如何去解决，了解哪些分布式事务中间件
35. 聊一聊对分库分表的理解
36. hystrix功能  & 在项目中怎么使用的 & hystrix 怎么检测断路器是否要开启/关闭 & hystrix 实现原理，除hystrix之外的其他熔断限流中间件有了解没有，了解多少说多少
37. dubbo有了解没有
38. 怎么理解java 中和 mysql 中的乐观锁、悲观锁
39. 一致性hash
40. 聊项目，画项目架构图，画一个用户从发起请求 到接收到响应 中间经过哪些服务 每个服务做什么事情 的流程图，讲数据库设计
41. 处理过线上oom问题没有，如何处理的
42. 遇到过线上服务器cpu飙高的情况没有，如何处理的
43. 线上有没有遇到其他问题，如何处理的
44. 对线程池的理解，项目中哪个地方使用了，如何使用的，用的Excutor框架中的哪个实现类，为什么用这个
45. 对CAS的理解，CAS带来的问题，如何解决这些问题
46. volatile底层、synchronized底层、锁升级的过程、MESI
47. 对mysql索引的理解、对组合索引的理解、索引的最佳实践
48. 分布式锁的实现、对比redis分布式锁 & zk分布式锁
49. 唯一id如何实现的，snowflake实现原理，snowflake有哪些问题，如何避免根据订单号可以推算出今天的订单量
50. 如果线上一个功能是用栈结构实现的，使用过程中要注意哪些问题，为什么
51. 怎么理解线程安全
52. 怎么理解接口幂等，项目中如何保证的接口幂等
53. 怎么理解微服务，服务如何划分，可以从哪几个方面去划分，为什么这样划分，微服务带来了哪些好处，哪些坏处，如何看待这个问题
54. 如何理解网关，网关带来的好处和坏处，如何解决
55. hystrix功能  & 在项目中怎么使用的 & hystrix 怎么检测断路器是否要开启/关闭 & hystrix 实现原理
56. 怎么理解命令模式和观察者模式，手写一个观察者模式或者命令模式的代码，策略模式也行
57. 掌握哪些设计模式，常用哪些，项目中如何使用的，为什么用这个，不用那个，手写一个线程安全的单例模式
58. 如何设计一个秒杀系统
59. 如果我现在就是要实现每秒10w请求，不能熔断限流，如何去设计
60. 假设现在双十一零点，大量下单请求，如何对这些订单进行分库分表，为什么
61. 服务A调用服务B中一个接口，服务B调用服务C中一个接口，如何实现若服务B响应服务A成功，则服务C一定响应服务B成功，需要考虑系统性能问题
62. 有两个表，table a，table b，写sql查询出仅在table a中的数据、仅在table b中的数据、既在table a 又在table b 中的数据
63. spring 源码有了解没有
64. myBatis源码有了解没有
65. mysql事务隔离级别、mvcc
66. 一个final修饰的属性，定义的时候没有初始化，在无参构造函数中初始化，可以吗，为什么
67. 说说对java中集合类的理解，项目中用过哪些，哪个地方用的，如何使用的，为什么不用其他的集合类
68. hashMap，concurrentHashMap底层实现，
69. list删除是怎么实现的，遍历的时候可以删除吗，为什么
70. redis中有哪些数据结构，了解过其底层怎么实现的吗，和java中相似的数据结构的对比
71. redis是单线程的还是多线程的，为什么这么快
72. redis hash中某个key过大，变为String类型的大key，怎么处理，使用中如何避免出现这种问题
73. 设计模式在项目中哪个地方用到了，怎么使用的，能不能画一个你熟悉的设计模式的UML图，手写单例模式，手写静态内部类实现的单例模式
74. 讲一讲mysql索引，实际工作中，哪些场景用了b+tree索引，哪些场景用了hash索引
75. explain 可以看到哪些信息，什么信息说明什么，explain的结果列讲一下
76. Spring源码看过没有，会多少讲多少
77. MyBatis源码看过没有，会多少讲多少
78. cas，cas的缺点，如何解决
79. aqs，countDownLatch如何实现
80. 线程池如何实现，核心线程数和最大线程数设置成多少，为什么这么设置，项目中哪个地方使用了线程池，使用时需要注意什么
81. mysql事务隔离级别，幻读，脏读，项目中用什么事务隔离级别，为什么
82. volatile底层原理、synchronized实现机制，
83. 对XA、TCC的理解，了解哪些分布式事务框架，有什么缺点
84. feign 和 dubbo，了解多少说多少
85. eureka 和 zookeeper，了解多少说多少
86. hystrix 和 sentinel，了解多少说多少
87. Spring cloud alibaba，了解多少说多少
88. 对分库分表、读写分离的了解，了解多少说多少
89. 画一下java 线程几个状态 及 状态之间互相转换的图
90. 聊项目，画项目架构图，画一个用户从发起请求 到接收到响应 中间经过哪些服务 每个服务做什么事情 的流程图，讲数据库设计 具体到部分表中有哪些字段【原创公众号：Bella的技术轮子】
91. kafka 如何保证消息顺序消费、在consumer group 中新增一个consumer  会提高消费消息的速度吗、那如果我想提高消息消费的速度，我要怎么办
92. redis几种数据结构 及 底层，项目中如何使用的redis
93. 哨兵机制、选举算法
94. 一致性hash
95. redis是单线程的还是多线程的，为什么速度这么快
96. 多路复用的几种方式以及区别
97. 对线程池的理解，在项目中如何使用的，多个线程之间如何共享数据，多个进程之间如何共享数据
98. hashMap、concurrentHashMap的区别 及 底层实现、hashMap和hashTable 的区别
99. 什么是红黑树，什么是b-tree，为什么hashMap中用红黑树不用其他树
100. 对mysql 索引的理解，为什么mysql索引中用b+tree，不用b-tree 或者其他树，为什么不用hash 索引
101. 数据库和缓存的双写一致性问题
102. 用过哪些Object类的方法，如何使用的
103. java如何实现序列化的，Serialization底层如何实现的
104. countDownLatch如何实现的
105. 项目中监控报警机制如何做的，说说你的了解
106. 线上服务器cpu飙高，如何处理这个问题
107. 服务A调用服务B，用户请求服务A，发现返回较慢，如何定位这个问题
108. TIME\_WAIT是什么状态还记得吗，什么情况下网络会出现这个状态

1.内核态 和 用户态、cas 和 sout 哪个用到了内核态和用户态的切换

2.哪些典型的应用用的是udp

3.线程池有了解吗，项目中如何使用的

4.计算密集型/IO密集型 任务 分别如何设置线程池的核心线程数和最大线程数，为什么这么设置

5.假如我下午5点要和5个人一起开会，但是这5个人现在都出去了，不在公司，但是今天会回来，问，我如何开这场会，用java 并发方面的知识回答

* 先机试（50分钟时间，三选二，不联网，明确告知机试不通过没有后续）
* 一面给面试官讲一下自己机试题的思路，面试官运行看结果，然后问了几个问题（什么是B-tree，什么是B+tree之类的）
* 笔试（10道选择题+2道数据库+2道算法题，30分钟）
* 二面给面试官讲自己的机试题的思路，面试官运行看结果，然后给面试官讲笔试题，一道一道讲为什么这么写，过程中面试官可能会改题，然后问你怎么解决修改后的题，然后又问了几个题
* 三面开始正常面试，但不是看简历问，一部分是简历上的，一部分是看面试官心情
* hr面