```
Redis 是一个使用 C 语言开发的非关系型(NoSQL)内存键值数据库,读写速度非常快
                                                              键的类型只能为字符串,值支持五种数据类型:字符串String、列表List、集合Set、散列表Hash、有序集合ZSet
                                                                   将高频数据存在缓存,操作缓存就是直接操作内存,所以速度相当快。 ⊖ 高性能
                                                 一般像 MySQL 这类的数据库的 QPS 大概都在 1w 左右(4 核 8g),使用 Redis 缓存之后很容易达到 10w+ \ominus 高并发
                                                                                              可以存储字符串、整数或者浮点数
                                                                  对整个字符串或者字符串的其中一部分执行操作;对整数和浮点数执行自增或者自减操作
                                                                 <mark>|向链表:从两端压入或者弹出元素;对单个或者多个元素进行修剪;只保留一个范围内的元素</mark>
                                                                      数组+链表:添加、获取、移除单个键值对;获取所有键值对;检查某个键是否存在
                                                     添加、获取、移除单个元素;检查一个元素是否存在于集合中;计算交集、并集、差集;从集合里面随机获取元素
                                                                                    交集、并集、差集的操作(共同好友) ⊖ 应用场景
                                                                     与set相比增加了权重参数score,使得集合中的元素可以按照score进行有序排列
                                                                     添加、获取、删除元素;根据分值范围或者成员来获取元素;计算一个键的排名
                                                                                                                ZSet有序
                                                                           有序集合保存的元素数量小于128个
                                                                       有序集合保存的所有元素的长度小于64字节
                                                                              查询/插入/删除 复杂度O(logN)
                                   跳跃链表本质是一个多层链表,每一层都是一个链表,最底层拥有所有的元素,层次往上划分为一级索引 二级索引....
                                                         在查找时,从上层指针开始查找,找到对应的区间之后再到下一层去查找
                                                                                 可以对 String 进行自增自减运算,从而实现计数器功能。
                                                                     将热点数据放到内存中,设置内存的最大使用量以及淘汰策略来保证缓存的命中率。
                                                             查找表的内容不能失效,而缓存的内容可以失效,例如 DNS 记录就很适合使用 Redis 进行存储。
                                                                              List 是一个双向链表,可以通过 lpush 和 rpop 写入和读取消息   ⊖ 消息队列
                                                                                可以使用 Redis 来统一存储多台应用服务器的会话信息。
                                      当应用服务器不再存储用户的会话信息,也就不再具有状态,一个用户可以请求任意一个应用服务器,从而更容易实现高可用性以及可伸缩性
                                                                      在分布式场景下,无法使用单机环境下的锁来对多个节点上的进程进行同步。
                                                     可以使用 Redis 自带的 SETNX 命令实现分布式锁,除此之外,还可以使用官方提供的 RedLock 分布式锁实现。
                                                                                 Set 可以实现交集、并集等操作,从而实现共同好友等功能。
                                                                                    ZSet 可以实现有序性操作,从而实现排行榜等功能。
                                                                                        Memcached 仅支持字符串类型
                                                                                                          ♥ 数据类型
                                                                                        Redis 支持五种不同的数据类型
                                                                           Memcached 把数据全部存在内存之中,不支持持久化
                                       Redis 支持两种持久化策略:RDB 快照和 AOF 日志,可以将内存中的数据保持在磁盘中,重启的时候可以再次加载进行使用
                                                                                    Memcached 的数据会一直在内存
                                                                                                                 Redis和Memcached
                                                      在 Redis 中,并不是所有数据都一直存储在内存中,可以将一些很久没用的 value 交换到磁盘
                                                                                Memcached 是多线程,非阻塞 IO 复用的网络模型
                                                                        Redis 使用单线程的多路 IO 复用模型。 (Redis6.0引入了多线程)
                                                                               Memcached 过期数据的删除策略只用了惰性删除
                                                                                    Redis 同时使用了惰性删除与定期删除
                                                                                    单线程编程容易并且更容易维护
                                                                           Redis 的性能瓶颈不在 CPU ,主要在内存和网络 🗦 Redis6.0之前单线程
                                                                    多线程就会存在死锁、线程上下文切换等问题,甚至会影响性能
                                                                                              核心命令处理和响应仍然是单线程的
                                                                                     引入多线程主要是为了提高网络 IO 读写性能 ○ 多线程
                                                                                   Redis 可以为每个键设置过期时间,当键过期时,会自动删除该键。
                                  Redis 通过过期字典(可以看作是 hash 表)来保存数据过期的时间。过期字典的键指向 Redis 数据库中的某个 key(键),过期字典的值是一个 long long 类型的整数
                                                            对于散列表这种容器,只能为整个键设置过期时间(整个散列表),而不能为键里面的单个元素设置过期时间。
                                                   redis默认是每隔 100ms 就随机抽取一些设置了过期时间的key,检查其是否过期,如果过期就删除。
                                                                               只会在取出 key 的时候才对数据进行过期检查。
                                                                             可以设置内存最大使用量,当内存使用量超出时,会施行数据淘汰策略。
                                                                       从已设置过期时间的数据集中挑选最近最少使用的数据淘汰 ⊖ volatile-lru
                                                                          从已设置过期时间的数据集中挑选将要过期的数据淘汰
                                                                          从已设置过期时间的数据集中任意选择数据淘汰 ⊖ volatile-random
                                                                               从所有数据集中挑选最近最少使用的数据淘汰 ⊖ allkeys-lru
                                                                               从所有数据集中任意选择数据进行淘汰 ⊖ allkeys-random
                                                               从已设置过期时间的数据集中挑选访问频率最少的数据淘汰
                                                                     从所有数据集中挑选访问频率最少的数据淘汰
                                                                                                      Redis 服务器是一个事件驱动程序
                                                                  服务器通过套接字与客户端或者其它服务器进行通信,文件事件就是对套接字操作的抽象。
                                                                  Redis使用 I/O 多路复用程序来同时监听多个套接字,并将到达的事件传送给文件事件分派器
                                                                              文件事件分派器会根据套接字产生的事件类型调用相应的事件处理器
                                                                             多个Socket>>>I/O多路复用程序>>>>文件事件分派器>>>事件处理器
                                                                      服务器有一些操作需要在给定的时间点执行,时间事件是对这类定时操作的抽象。
                                                                                     定时事件: 是让一段程序在指定的时间之内执行一次
                                                                                     周期性事件: 是让一段程序每隔指定时间就执行一次
                                                   Redis 将所有时间事件都放在一个无序链表中,通过遍历整个链表查找出已到达的时间事件,并调用相应的事件处理器。
                                                     I/O 多路复用技术的使用让 Redis 不需要额外创建多余的线程来监听客户端的大量连接,降低了资源的消耗
                                                                                                             单线程I/O多路复用
                                                                                                  select/poll/epoll
                                                                    Redis 是内存型数据库,为了保证数据在断电后不会丢失,需要将内存中的数据持久化到硬盘上。
                                                        B时的RDB实际上就是一个时间事件,事件可执行时,则调用BGSAVE命令,fork出一个子进程来生成RDB文件。
                                                                      Redis 可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。
                                                          如果系统发生故障,将会丢失最后一次创建快照之后的数据。如果数据量很大,保存快照的时间会很长。
                                                                            可以将快照复制到其它服务器从而创建具有相同数据的服务器副本。
                                                           开启 AOF 持久化后,Redis 会将每一条更改 Redis 中的数据的命令写入到内存缓存 server.aof_buf 中
                                                                         然后再根据 appendfsync 配置来决定何时将其同步到硬盘中的 AOF 文件。
                                                               同步选项: always每个写命令都同步、everysec每秒同步一次、no让操作系统来决定何时同步
                                                                                                    Redis不支持rollback, 不满足原子性
                                                              大量请求的 key 根本不存在于缓存中,导致请求直接到了数据库上,根本没有经过缓存这一层
                                                                    不合法的参数请求直接抛出异常返回给客户端 ○ 在接口做参数校验
                                                   当访问缓存和DB都没有查询到值时,可以将key写进缓存,但是设置较短的过期时间 ⊖ 缓存无效key
                                                            布隆过滤器判断不存在的一定不存在,判断存在的,大概率存在
                当字符串存储要加入到布隆过滤器中时,该字符串首先由多个哈希函数生成不同的哈希值,然后将对应的位数组的下标设置为 1 (当位数组初始化时,所有位置均为 0)
                                                       当第二次存储相同字符串时,因为先前的对应位置已设置为 1,所以很容易知道此值是否存在
                                    某一个热点 key,在缓存过期的一瞬间,同时有大量的请求打进来,请求最终都会走到数据库,造成瞬时数据库请求量大、甚至可能打垮数据库。
                                                                  加互斥锁(redis分布式锁/JVM 锁),使得只有少数请求才会走到数据库
                                                                         设置热点数据不过期,由定时任务去异步加载数据更新缓存
                                                        y设置了相同的过期时间,导致缓存在同一时刻全部失效,造成瞬时数据库请求量大、压力骤增,引起雪崩
                                                      给缓存的过期时间时加上一个随机值时间,使得每个 key 的过期时间分布开来
                                                                                                        解决方案
                                                                                                  加互斥锁
                                                     采用Redis集群,避免单机出现问题整个缓存服务都没法使用
                                                                                     ♥ Redis服务不可用,导致请求都落到了数据库上
                                                                    限流,避免同时处理大量请求
                                                                          读请求先读缓存,如果缓存不存在,则从数据库读取,并重建缓存
                                                                                         写入缓存中的数据,都设置失效时间
                                                          保证缓存和数据库数据「实时」一致,当数据发生更新时,不仅要操作数据库,还要一并操作缓存
                              在「并发」场景下无法保证缓存和数据一致性,且存在「缓存资源浪费」和「机器性能浪费」的情况发生 ⊖ 更新数据库+更新缓存
           请求1先把cache中的A数据删除 -> 请求2从DB中读取数据->请求2将旧值写入缓存->请求1再把新值写入数据库,造成数据不一致
                            解决方案是「延迟双删」,先删除缓存,更新数据库,延迟删除缓存,但这个延迟时间很难评估
                                                                                                   缓存和数据库一致性
                                       请求A进行写操作删除缓存->请求A将数据写入数据库
    ->请求B去从库读数据,此时没有完成主从同步,查询到旧值->请求B将旧值写入缓存->数据库完成主从同步,从库变为新值
                                                                                      更新数据库+删除缓存
                                     延迟双删,延迟时间在主从同步的时间基础上增加几百ms
             为了保证两步都成功执行,需配合「消息队列」或「订阅变更日志」的方案来做,本质是通过「重试」的方式保证数据一致性
                                                                                                                缓存方案
                              「读写分离 + 主从库延迟」也会导致缓存和数据库不一致,缓解此问题的方案是「延迟双删」
                    凭借经验发送「延迟消息」到队列中,延迟删除缓存,同时也要控制主从库延迟,尽可能降低不一致发生的概率
                                                                        先更新DB, 然后直接删除cache ⊖ 写
                                   从cache中读取数据,读取到就直接返回;cache中读取不到,从DB读取数据返回;再把数据放cache中 😑 读 🗦 旁路缓存模
                                                  先提前把热点数据放入cache ⊖ 首次请求数据一定不在 cache 的问题
   更新DB的时候同样更新cache,强一致加锁,软一致缓存设置较短过期时间
                                          ○ 写操作比较频繁的话导致cache中的数据会被频繁被删除,这样会影响缓存命中率。
                                      Read/Write Through Pattern 中服务端把 cache 视为主要数据存储,从中读取数据并将数据写入其中。
                                                      cache 服务负责将此数据读取和写入 DB,从而减轻了应用程序的职责。
                                                                                                       缓存读写策略
                                                        先查 cache, cache 中不存在,直接更新 DB。
                                                                                             读/写穿透模式
                                                                                  ∀ Write Through
                                   cache 中存在,则先更新 cache,然后 cache 服务自己更新 DB (同步更新 cache 和 DB)
             由cache服务来负责cache和DB的读写
                                                          从 cache 中读取数据,读取到就直接返回。
                                                                                  Read Through
                                                  读取不到的话,先从 DB 加载,写入到 cache 后返回响应。
                                                              只更新缓存,不直接更新 DB,异步批量的方式来更新 DB
                                   DB 的写性能非常高,非常适合一些数据经常变化又对数据一致性要求没那么高的场景,比如浏览量、点赞量。
                                                                                         部署简单,成本低,高性能,不需要同步数据
                                                                                              可靠性保证不是很好,有宕机风险
                                                   多启用几台redis服务器,作为主服务器Master的从服务器Slave,Slave数据由Master复制过去,主从服务器数据一致
                                                                 如果主服务器挂了,那可以「手动」把「从服务器」升级为「主服务器」,缩短不可用时间
                    如果是第一次「同步」,从服务器没有复制过任何的主服务器,或者从服务器要复制的主服务器跟上次复制的主服务器不一样
                                                主服务器会在后台生成RDB文件,通过前面建立好的连接发给从服务器
                                          从服务器收到RDB文件后,首先把自己的数据清空,然后对RDB文件进行加载恢复
         主服务器把生成RDB文件「之后修改的命令」会用「buffer」记录,从服务器加载完RDB之后,主服务器把「buffer」记录下的命令都发给从服务器
                                                                                              使用「PSYNC」命令进行数据同题
                                      如果只是由于网络中断,只是「短时间」断连,那就会采用「部分重同步」模式进行复制
                             从服务器存储着主服务器ID (RUNID)和复制进度(offset),主服务器每次传播命令都会把offset给到从服务器
                                        从服务器断连之后,就会发「PSYNC」命令给主服务器,同样也会带着RUNID和offset
                                                  可靠性保证不好,主节点故障;没有解决主节点写的压力;数据冗余;主节点的写能力和存储能力受到单机限制 🕒 缺点
                                                                                    在主从复制基础上, 哨兵实现自动化故障恢复
                                                     哨兵节点:哨兵节点是特殊的 Redis 节点,不存储数据,监听主从服务器状态等
                                                                     数据节点: 主服务器和从服务器都是数据节点。
                                                                                         监控: 监控主服务器状态
                                                                                                                      Redis架构
                                                                                       通知: 故障发送消息给管理员
                                                                                                             哨兵Sentinel
                                   每 1 秒每个 Sentinel 对其他 Sentinel 和 Redis 节点执行 PING 操作(监控),这是一个心跳检测,是失败判定的依据
                                                    每 2 秒每个 Sentinel 通过 Master 节点的 channel 交换信息 (Publish/Subscribe)
                                              每 10 秒每个 Sentinel 会对 Master 和 Slave 执行 INFO 命令,来发现Slave节点以及确认主从关系
                   单个Sentinel不断地用ping命令看主服务器有没有下线,如果主服务器在「配置时间」内没有正常响应,那当前哨兵就「主观」认为该主服务器下线了
                               「足够多」 (还是看配置) 的哨兵认为该主服务器已经下线,那就认为「客观下线」,这时就要对主服务器执行故障转移操作。
                                        认为主观下线的单个sentinel询问其他sentinel是否也认为主观下线,达成共识的满足一定个数,认定为客观下线
                                                  主从切换还是需要时间,会丢失数据;没有解决主节点写的压力;主节点的写能力存储能力受到单机限制 ⊖ 缺点
                                                                     哨兵模式中,单个节点的写能力,存储能力受到单机的限制,动态扩容困难复杂
                                             用多个Redis实例来组成一个集群,按照一定的规则把数据「分发」到不同的Redis实例上。需要解决数据路由和数据迁移问题
                                                Redis Cluster默认一个集群有16384个哈希槽,哈希槽会被分配到Redis集群中的实例中
                                         Redis集群的实例会相互「通讯」,交互自己所负责哈希槽信息(最终每个实例都有完整的映射关系)
                                  当客户端请求时,使用CRC16算法算出Hash值并模以16384,自然就能得到哈希槽进而得到所对应的Redis实例位置
                                                                                                               分片集群
                   16384个既能让Redis实例分配到的数据相对均匀,又不会影响Redis实例之间交互槽信息产生严重的网络性能开销问题
                                                                               ○ 为什么16384个哈希槽
                   哈希槽实现相对简单高效,每次扩缩容只需要动对应Solt (槽) 的数据,一般不会动整个Redis实例 ○ 为什么使用哈希槽,而非一致性哈希算法
                                                                          Redis Cluster客户端直连Redis实例
默认分配1024个哈希槽,哈希槽与Redis实例的映射关系由Zookeeper进行存储和管理,Proxy会缓存一份至本地,Redis集群实例发生变化时,DashBoard更新Zookeeper和Proxy的映射信息
                                                           Codis客户端直连Proxy,再由Proxy进行分发到不同的Redis实例进行处理
```

Redis Cluster支持同步迁移,Codis支持同步迁移&&异步迁移 ⊖ 数据迁移

```
原子性 (Atomicity)
                           回滚可以用回滚日志Undo log实现,回滚日志记录着事务所执行的修改操作,在回滚时反向执行这些修改操作即可。回滚日志记录数据的逻辑修改
              一致性 (Consistency) ⊝ 数据库在事务执行前后都保持一致性状态。在一致性状态下,所有事务对同一个数据的读取结果都是相同的
              隔离性 (Isolation) ⊖ 一个事务所做的修改在最终提交以前,对其它事务是不可见的。
                           一旦事务提交,则其所做的修改将会永远保存到数据库中。即使系统发生崩溃,事务执行的结果也不能丢失。
              持久性 (Durability) ♀
                           系统发生崩溃可以用重做日志(Redo Log)进行恢复,从而实现持久性。重做日志记录的是数据页的物理修改
             - 无并发,事务串行,只要满足原子性,就能满足一致性;并发,事务并行,要满足原子性和隔离性,才能满足一致性,从而保证执行结果正确。持久性来应对系统崩溃。
         并发一致性问题的主要原因是破坏了事务的隔离性,通过并发控制来保证隔离性;并发控制可以通过封锁来实现,但是操作复杂,使用数据库提供的事
                     丢失修改指一个事务的更新操作被另外一个事务的更新操作替换。
                     T1 和 T2 两个事务都对一个数据进行修改,T1 先修改,T1尚未commit,T2 随后修改,T2 的修改覆盖了 T1 的修改。
                   脏读指在不同的事务下,当前事务可以读到另外事务未提交的数据。
                   T1 修改一个数据但未提交,T2 随后读取这个数据。如果 T1 撤销了这次修改,那么 T2 读取的数据是脏数据。
                       不可重复读指在一个事务内多次读取同一数据集合。在这一事务还未结束前,另一事务修改了这一数据集合,因此第一次事务的两次读取的数据可能不一致。
                      {f T2} 读取一个数据,{f T1} 对该数据做了修改。如果 {f T2} 再次读取这个数据,此时读取的结果和第一次读取的结果不同。
                   幻读本质上也属于不可重复读的情况
                   T1 读取某个范围的数据,T2 在这个范围内插入新的数据,T1 再次读取这个范围的数据,此时读取的结果和和第一次读取的结果不同
                MySQL 中提供了两种封锁粒度: 行级锁以及表级锁。
                尽量只锁定需要修改的那部分数据,锁定的数据量越少,发生锁争用的可能就越小,系统的并发程度就越高
                加锁需要消耗资源,锁的各种操作(包括获取锁、释放锁、以及检查锁状态)都会增加系统开销。封锁粒度越小,系统开销就越大。
                          读锁(共享锁/S锁) \,\,\ominus\,\,\,\,\,一个事务对数据对象 A 加了 S 锁,可以对 A 进行读取操作,但是不能进行更新操作。加锁期间其它事务能对 A 加 S 锁,但是不能加 X 锁。
                          写锁(互斥锁、X锁) \ominus 一个事务对数据对象 A 加了 X 锁,就可以对 A 进行读取和更新。加锁期间其它事务不能对 A 加任何锁。
                          使用意向锁 (Intention Locks) 可以更容易地支持多粒度封锁。
                          如果事务T想对表A加锁,就要检查是否有其他事务已对表A加锁或表A某一行加锁,如果去一行行检测是很耗时的,所以引入意向锁,可以表示表A是否被加锁。
                          意向锁在原来的 X/S 锁之上引入了 IX/IS, IX/IS 都是表锁,表示一个事务想要在表中的某个数据行上加 X 锁或 S 锁。
                          一个事务在对某行数据添加S锁前,必须先获得该表的IS锁或更强的锁;对某行数据加X锁前,必须先获得IX锁
                          IS/IX之间互相兼容,且不会和行级的X/S锁发生冲突,只和表级的X/S锁冲突。
                          一级封锁协议 ⊝ 事务修改数据前先加X锁,事务结束才释放锁。可以解决<mark>丢失修改</mark>问题。
                          二级封锁协议 ⊝ 在一级封锁协议的基础上规定,读取数据前必须加S锁,读完释放。进一步解决脏读问题
                          三级封锁协议 ⊝ 在二的基础上规定,读取数据前必须加S锁,事务结束才释放。进一步解决不可重复读问题。
                        可串行化调度:通过并发控制,使得并发执行的事务结果与某个串行执行的事务结果相同。串行执行的事务互不干扰,不会出现并发一致性问题。
                        可串行性是并发事务正确调度的准则,为了保证并发调度的正确性,使用两段锁协议实现并发调度的可串行性。
                        在对任何数据进行读、写操作前,事务首先要申请对该数据的封锁;在释放一个封锁之后,事务不再申请和获得任何其他封锁。
                       MySQL 的 InnoDB 存储引擎采用两段锁协议,会根据隔离级别在需要的时候自动加锁,并且所有的锁都是在同一时刻被释放,这被称为隐式锁定。
         MySQL隐式和显式锁定
                      InnoDB 也可以使用特定的语句进行显示锁定
             不同的隔离级别对事务之间的隔离性是不一样的(级别越高事务隔离性越好,但性能就越低),而隔离性是由MySQL的各种锁来实现的,只是它屏蔽了加锁的细节。
                                事务的修改,即使未提交,也会被其他事务看到。
                               锁的维度:读不加锁,写加排它锁
             读未提交READ UNCOMMIT
                                解决了丢失修改,可能导致脏读、不可重复读、幻读
             读提交READ COMMITTE
                             解决了丢失修改、脏读,不可重复读或幻读仍然可能发生
                              保证同一个事务中多次读取同一数据的结果相同。
              可重复读REPEATABLE READ
                              解决了丢失修改、脏读和不可重复读,幻读仍可能发生
             可串行化SERIALIZABLE 强制事务串行执行,不会出现一致性问题。 该隔离级别需要加锁实现,保证同一时间只有一个事务执行。
                   不支持数据库异常崩溃后的安全恢复
                   支持行级锁和表级锁,默认为行级锁
                   提供事务支持, 具有commit和rollback事务能力
                  使用redo log恢复数据库异常崩溃前的状态
                              原子性 ⊖ 使用 undo log(回滚日志) 来保证事务的原子性
                              隔离性 ⊖ 通过 锁机制、MVCC 等手段来保证事务的隔离性 ( 默认支持的隔离级别是 REPEATABLE-READ ) 。
                              持久性 ⊖ 使用 redo log(重做日志) 保证事务的持久性
                              一致性 ⊝ 保证了事务的持久性、原子性、隔离性之后,一致性才能得到保障
              概念 ○ 索引是一种用于快速查询和检索数据的数据结构。常见的索引结构有: B 树, B+树和 Hash。
                        索引可以大大加快 数据的检索速度(大大减少检索的数据量),索引减少了查询过程中的IO次数,将随机I/O变为顺序I/O
                        通过创建唯一性索引,可以保证数据库表中每一行数据的唯一性
                        创建索引和维护索引需要耗费许多时间。当对表中的数据进行增删改的时候,如果数据有索引,那么索引也需要动态的修改,会降低 SQL 执行效率。
                        索引需要使用物理文件存储,也会耗费一定空间。
                             Hash 索引不支持顺序和范围查询(假如我们要对表中的数据进行排序或者进行范围查询,那 Hash 索引就不行了。)
                               B 树也称 B-树,全称为 多路平衡查找树 , B+ 树是 B 树的一种变体。 B 树和 B+树中的 B 是 Balanced (平衡) 的意思。
                                   B 树的所有节点既存放键(key) 也存放 数据(data)
                                   B 树的叶子节点都是独立的
                                    B 树的检索的过程相当于对范围内的每个节点的关键字做二分查找,可能还没有到达叶子节点,检索就结束了
                                   m叉树,树高度能够大大降低,且每个节点可以存储j个记录,如果设置为页大小4k,能够利用局部性原理(磁盘是按页预读的),减少磁盘IO
                                    B+树是基于B 树和叶子节点顺序访问指针实现的,具有B树的平衡性,并通过顺序访问指针来提高区间查询性能
                                    B+树只有叶子节点存放 key 和 data, 其他内部节点 (索引节点) 只存放 key
                       B 树& B+树 [
              索引数据结构
                                    B+树的叶子节点有一条引用链指向与它相邻的叶子节点
                                    B+树的检索效率很稳定,任何查找都是从根节点到叶子节点的过程,叶子节点的顺序检索很明显
                                                    范围查找不需要像B树一样中序回溯
                                                    叶子节点存储实际行,适合大数据量磁盘存储,非叶子存储指针适合内存存储
                                                    相同内存的情况下,B+树能够存储更多的索引
                               B和B+树主要用于数据存储在磁盘上的场景,这两种数据结构树比较矮胖,每个结点存放一个磁盘大小的数据,这样一次可以把一个磁盘的数据读入内存,减少磁盘转动的耗时,提高效率。
                                   红黑树等平衡树也可以用来实现索引,但是文件和数据库系统普遍采用B+树作为索引结构,是因为B+树访问磁盘数据有更高的性能
                                   B+树是m叉树, 有着更低的树高; 红黑树是2叉树, 树高比B+树高很多
                       B+树和红黑树比纳
                                            数据库系统将索引的一个节点大小设为磁盘一页的大小,使得一次I/O就能完全载入一个节点
                                            磁盘访问寻道次数和树高成正比,同一个磁盘块访问只需要很短的磁盘旋转时间,B+树更适合磁盘数据的读取
     MySQL索引
                                   磁盘预读特性 ⊝ 相邻节点能够被预先载入
                            数据表的主键列使用的就是主键索引。
                            一张数据表有只能有一个主键,并且主键不能为 null,不能重复
                            没有指定表的主键时,InnoDB 会自动先检查表中是否有唯一索引且不允许存在null值的字段,如果有则选择该字段,否则 InnoDB 将会自动创建一个 6Byte 的自增主键。
                                  辅助索引的叶子节点存储的数据是主键,通过辅助索引,可以定位主键的位置。
                                              唯一索引的属性列不能出现重复的数据,但是允许数据为 NULL,一张表允许创建多个唯一索引。
                     辅助索引(二级索引)
                                              建立唯一索引的目的大部分时候都是为了该属性列的数据的唯一性,而不是为了查询效率。
                                        普通索引 ⊝ 普通索引的唯一作用就是为了快速查询数据,一张表允许创建多个普通索引,并允许数据重复和 NULL。
                                             · 前缀索引只适用于字符串类型的数据。前缀索引是对文本的前几个字符创建索引,相比普通索引建立的数据更小, 因为只取前几个字符。
                                       全文索引 ⊝ 全文索引主要是为了检索大文本数据中的关键字的信息,是目前搜索引擎数据库使用的一种技术。MySQL5.6的InnoDB存储引擎开始支持全文索引
                                           聚簇索引即索引结构和数据一起存放的索引,叶子节点存储的是主键和整行数据。主键索引属于聚簇索引。
                                           优点:聚集索引的查询速度非常的快,因为整个 B+树本身就是一颗多叉平衡树,叶子节点也都是有序的,定位到索引的节点,就相当于定位到了数据。
                                               依赖于有序的数据 ⊝ 因为 B+树是多路平衡树,如果索引的数据不是有序的,需要在插入时排序,如果数据是整型还好,字符串或 UUID 又长又难比较的数据,插入或查找的速度肯定比较慢。
                                                       如果对索引列的数据被修改时,那么对应的索引也将会被修改, 而且况聚集索引的叶子节点还存放着数据,修改代价肯定是较大的, 所以对于主键索引来说,主键一般都是不可被修改的。
                                             聚簇索引和非聚簇索引
                                            非聚簇索引的叶子节点并不一定存放数据的指针,因为辅助索引的叶子节点就存放的是主键,根据主键再回表查数据。
                                            优点 ○ 更新代价比聚簇索引要小 , 非聚簇索引的叶子节点是不存放数据的
                           非聚簇索引secondary index
                                                 跟聚簇索引一样,非聚簇索引也依赖于有序的数据
                                                              回表,即使用索引查询数据时,需要检索出来的数据可能包含其他列,但走的索引树叶子节点只能查到当前列值以及主键ID
                                                   可以直接在索引列中过滤掉不需要数据,减少回表的次数,需要建立联合索引
                     覆盖索引即需要查询的字段正好是索引的字段,那么直接根据该索引,就可以查到数据了,而无需回表查询。
                     多字段索引,违反最佳左前缀原则,但是MySQL 针对这一问题进行优化可以命中索引
                     当查询条件为等值或范围查询时,索引可以根据查询条件去找对应的条目。查询条件为:〈〉、NOT、in、not exists,索引定位困难,执行计划此时可能更倾向于全表扫描。
                     如果where中有or,即使其中有条件带索引也不会命中索引
                     where中索引列上做操作(计算,函数,(自动或者手动)类型装换)
                       redo log (重做日志) 是InnoDB存储引擎独有的,它让MySQL拥有了崩溃恢复能力
                       redo log是物理日志,记录内容是"在某个数据页上做了什么修改"
                undo log ⊖ undo log回滚日志记录着事务所执行的修改操作,在回滚时反向执行这些修改操作即可
     MySQL三大日志
                      bin log 归档日志是逻辑日志,记录内容是语句的原始逻辑,属于MySQL Server 层。
                      binlog记录了数据库表结构和表数据「变更」,比如update/delete/insert/truncate/create。在MySQL中,主从同步实际上就是应用了binlog来实现的
                      MySQL数据库的数据备份、主备、主主、主从都离不开binlog,需要依靠binlog来同步数据,保证数据一致性,保证了MySQL集群架构的数据一致性。
                      写入机制 ⊝ 事务执行过程中,先把日志写到binlog cache,事务提交的时候,再把binlog cache写到binlog文件中。
                        多版本并发控制MVCC是 MySQL 的 InnoDB 存储引擎实现隔离级别的一种具体方式
                       实现READ COMMITTED和REPEATABLE READ这两种隔离级别,本质上就是对比版本
                       频繁加锁会导致数据库性能低下,引入了MVCC多版本控制能够实现读写不阻塞,提高数据库性能
                       READ UNCOMIITTED总是读取最新的数据行,无需使用 MVCC。SEARIALIZABLE需要对所有读取的行都加锁,单纯使用 MVCC 无法实现。
                                   封锁可以解决一致性问题,但实际场景中读操作多于写操作,于是引入了读锁、写锁,读与读之间不互斥,读与写互斥。
                                  MVCC利用多版本思想,写操作更新最新的版本快照,读去读旧版本快照,没有互斥关系,做到读写不阻塞
                                                                     read committed读提交隔离级别生成的是语句级快照
                   基本思想
                         MVCC通过生成版本快照,用快照实现一定级别(语句级或事务级)的一致性读取
                                                                     repeatable read可重复读隔离级别生成的是事务级的快照
                         MVCC规定只能读取已提交的快照和自身未提交的快照,来解决脏读和不可重复读问题。一个事务可以读取自身未提交的快照,这不算是脏读。
                        MVCC通过【版本比对】来实现读写不阻塞,版本数据存在于undo log中,针对不同的隔离级别read view获取时机也不同
                       read commit隔离级别下,每次select都获取一个新的read view,对应于语句级快照;repeatable read隔离级别则每次事务只获取一个read view,对应于事务级快照
                               MVCC 的多版本指的是多个版本的快照,快照存储在 Undo log中,该日志通过回滚指针 ROLL_PTR 把一个数据行的所有快照连接起来。
                               事务版本号TRX_ID、操作、DEL字段(标记是否被删除)
                              DB_TRX_ID (6字节) : 表示最后一次插入或更新该行的事务 id。
                              DB_ROLL_PTR (7字节) 回滚指针,指向上一个版本数据在 undo log 里的指针。
                               在查询时, InnoDB会生成read view, 主要做事务版本快照的可见性判断
                                       trx ids (read view生成时刻尚未提交commit的事务ID (活跃的事务) 集合)
     多版本并发控制MVCC
                       read view
                                       creator_trx_id (当前的事务ID/创建该read view的事务ID)
                                       up_limit_id (活跃事务列表中的最小事务ID) ⊖ 小于这个ID的数据版本均可见
                                       low_limit_id (read view生成时刻尚未分配的下一个事务ID)
                                       对于更新操作而言, InnoDB引擎是肯定会加<mark>写锁的(</mark>数据库是不可能允许在同一时间,更新同一条记录的),这就解决了丢失修改问题
                                    read commit (读已提交) 隔离级别生成的是语句级快照
                                    事务A读取数据生成版本号,事务B修改了数据(此时加了写锁)也会生成版本号,事务A再读取时,读取的是依据最新版本号来读取的,如果事务B未commit,读取的仍然是之前版本号数据
                   并发一致性问题解决。
                                不可重复读 ⊝ repeatable read (可重复复读)隔离级别生成的是事务级快照,每次读取的都是「当前事务的版本」,即使当前数据被其他事务修改了(commit),也只会读取当前事务版本的数据。
                                                                   执行普通select语句,会以快照读的方式读取数据,已经解决了幻读的问题(因为它是读历史版本的数据)
                                幻读 ⊖ repeatable read (可重复复读)隔离级别会存在幻读的问题
                                                                   当前读,则需要配合Next-key Lock 间隙锁来解决幻读的问题
                                        如果读取的行正在执行 DELETE 或 UPDATE 操作,这时读取操作不会去等待行上锁的释放;InnoDB会去读取行的一个快照数据,对于这种读取历史数据的方式,称为快照读 (snapshot read)
                                        在 Repeatable Read 和 Read Committed 两个隔离级别下,如果是执行普通的 select 语句则会使用 快照读 (MVCC)
                                        在快照读下,即使读取的记录已被其它事务加上 X 锁,这时记录也是可以被读取的,即读取的快照数据。
                                        在 Repeatable Read 下 MVCC 实现了可重复读和防止部分幻读
                                        在 Repeatable Read 下 MVCC 防止了部分幻读,"部分"是指在 快照。
                                                                           读 情况下,只能读取到第一次查询之前所插入的数据(根据 Read View 判断数据可见性,Read View 在第一次查询时生成)
                                       如果执行的是下列语句, 就是 当前读/锁定读
                                       insert、update、delete 操作
                                       在锁定读下,读取的是数据的最新版本,这种读也被称为 当前读 (current read)
                                                        select ... lock in share mode:对记录加 S 锁,其它事务也可以加S锁,如果加 x 锁则会被阻塞
                                                        select ... for update、insert、update、delete:对记录加 X 锁,且其它事务不能加任何锁
                                       当前读每次读取的都是最新数据,如果两次查询中间有其它事务插入数据,就会产生幻读。Repeatable Read ,如果执行的是当前读,需要使用 Next-key Lock 间隙锁,来防止其它事务在间隙间插入数据,防止幻读
                          锁定一个记录上的索引,而不是记录本身。
                          如果表没有设置索引,InnoDB 会自动在主键上创建隐藏的聚簇索引,因此 Record Locks 依然可以使用。
                        锁定索引之间的间隙,但是不包含索引本身。 ⊖ SELECT c FROM t WHERE c BETWEEN 10 and 20 FOR UPDATE
                  Next-key lock ⊖ 它是 Record Locks 和 Gap Locks 的结合,不仅锁定一个记录上的索引,也锁定索引之间的间隙。它锁定一个前开后闭区间,
                          只返回必要的列:最好不要使用 SELECT * 语句
                         │ 只返回必要的行:使用 LIMIT 语句来限制返回的数据→ 利用子查询优化超多分页场景
                          缓存重复查询的数据
                        切分大的查询
                        分解大连接查询
                      是否能使用「覆盖索引」,减少「回表」所消耗的时间。在select 的时候,一定要指明对应的列,而不是select *
                      考虑是否组建「联合索引」,如果组建「联合索引」,尽量将区分度最高的放在最左边,并且需要考虑「最左匹配原则」
                      对于查询概率比较高,经常作为where条件的字段设置索引
                      对索引进行函数操作或者表达式计算会导致索引失效
MySQL调何
                      通过explain命令来查看SQL的执行计划,看看自己写的SQL是否走了索引,走了什么索引
               在开启事务后,在事务内尽可能只操作数据库,并有意识地减少锁的持有时间(比如在事务内需要插入&&修改数据,那可以先插入后修改。
                                  主库接收写请求,从库接收读请求。
              单库升级为主从架构,实现读写分离
                                  从库的数据由主库发送的binlog进而更新,实现主从数据一致(在一般场景下,主从的数据是通过异步来保证最终一致性的)
        写优化
                     将数据量打散到多库和多表
             分库分表
                    主键自增实现:雪花算法、Redis自增、MySQL自增
```

概念:事务指的是满足 ACID 特性的一组操作,可以通过 Commit 提交一个事务,也可以使用 Rollback 进行回滚。

事务的所有操作要么全部提交成功,要么全部失败回滚