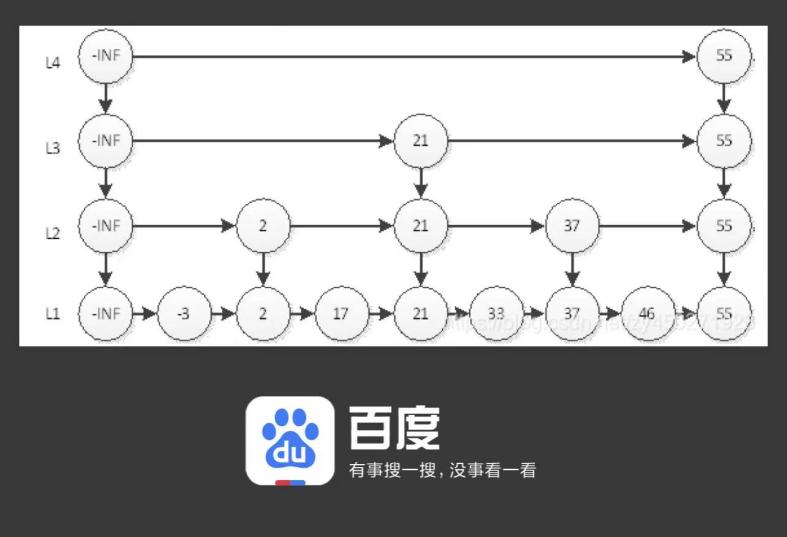
Redis(remote dictionary server)

1. Redis的**zset**底层是跳跃表—多层的有序链表，用基于概率统计的插入算法

**不用红黑树**因为：1.平衡树做范围查询更复杂，在查最小值的时候要用中序遍历查最大值，但跳跃表找到最小值后去最底层链表遍历就行 2.插入和删除平衡树要调整更多，跳跃表只要改相邻节点就行 插入/删除/查询都只要N(logn)

这几级索引的结点总和就是 n/2+n/4+n/8…+8+4+2=n-2，所以跳表的**空间复杂度为O(n)**



跳表其实就是一种可以进行二分查找的有序链表

跳表在原有的有序链表上面**增加了多级索引**，通过索引来实现快速查找。首先在最高级索引上查找最后一个小于当前查找元素的位置，然后再跳到次高级索引继续查找，直到跳到最底层为止，这时候以及十分接近要查找的元素的位置了(如果查找元素存在的话)。由于根据索引可以一次跳过多个元素，所以跳查找的查找速度也就变快了

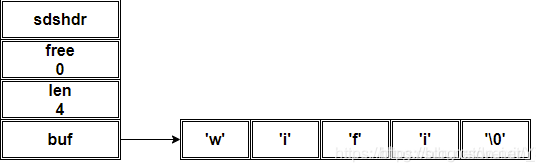
**常见数据结构**：**string、hash、set、zset、list**

所有的数据结构都是以唯一的 key 字符串作为名称，然后通过这个唯一 key 值来获取相应的 value 数据。不同类型的数据结构的差异就在于 value 的结构不一样。



**String底层：**

**简单动态字符串(SDS) ，内存块结构和3个参数**

****

**free** = 0：表示这个 SDS 没有分配任何未使用空间；

**len** = 4：表示这个 SDS 保存了一个四字节长的字符串；

**buf**：是一个 char 类型的数组，最后一个字节和C语言字符串一样，保存了空字符 ‘\0’, 不计入len

**3种内部编码：int、embstr、raw**

**int编码**：当一个key的value是整型时，Redis就将它编码为int类型(另外还有一个条件：把这个value当作字符串来看，长度不能超过20可以用 long 类型表示整数值来保存) 为了节省内存Redis默认会缓存10000个整型值，这意味着，如果有10个不同的KEY，value都是10000以内的值，事实上全部都是共享同一个对象

**embstr编码**：保存长度 < 44字节的字符串（3.2版本之前是39字节）

**raw编码**：保存长度 > 44字节的字符串（redis3.2版本之前是39字节）

**embstr和raw的区别**：都使用redisObject和sds保存数据，

embstr的使用只分配一次内存空间（因此redisObject和sds是连续的）

raw需要分配两次内存空间（分别为redisObject和sds分配空间）。

因此与raw相比，embstr的好处在于创建时少分配一次空间，删除时少释放一次空间，以及对象的所有数据连在一起，寻找方便。而embstr的坏处也很明显，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，因此redis中的embstr实现为只读。

**编码转换**：1.当 int 编码保存的值不再是整数，或大小超过了long的范围时，自动转化为raw。2.对于 embstr 编码，由于 Redis 没有对它编写修改程序(embstr 是只读的)，在对embstr对象进行修改时，都会先转化为raw再进行修改，因此，只要是修改embstr对象，修改后的对象一定是raw的，无论是否达到44字

**优点**：**1.**读写性能，速度可以达到上万次每秒

**2.**支持数据持久化，有AOF和RDB两种持久化方式。

**3.**支持事务，Redis的所有操作都是原子性的，而且还支持对几个 操作合并后的原子性执行。

**4.**数据结构丰富，string、hash、set、zset、list

**5.**支持主从复制，主机会自动把数据同步到从机，可以进行读写分离

6.高性能 高并发

**缺点：1.**数据库容量受到物理内存的限制，不能用作海量数据的高性能读写， 因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

**2.**Redis不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分 读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。

**3.**主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引 入数据不一致的问题，降低了系统的可用性。

**4.**Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复 杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间， 这对资源造成了很大的浪费。

**保证高并发**

直接操作缓存能够承受的请求是远大于直接访问数据库的，所以可以把数据库部分数据转到缓存里，这样用户一部分请求会直接到缓存不用经过数据库

**保证高性能**

**1.高效数据结构**

Redis是基于KV内存数据库，它内部构建了一个哈希表，根据指定的key访问时，只要O(1)的时间复杂度就可以找到对应的数据，而value的值又是一些拥有各种特性的数据结构，这样在数据操作的时候就能提供很好的性能

**2.基于内存存储**

用户第一次访问数据库某些数据比较慢，因为是从硬盘上读取的。之后把访问的数据存在redis缓存中，这样下次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，速度很快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据就行

**3.IO多路复用**

Redis 是基于Reactor 单线程模式来实现的，IO多路复用程序接收到用户的请求后，全部推送到一个队列里，交给文件分派器。对于后续的操作进行分发。针对建立连接请求事件，通过 Acceptor 处理，并建立对应的handler 负责后续业务处理，针对非连接事件，Reactor会调用对应的handler 完成 read->业务处理->write 处理流程，并将结果返回给客户端，整个过程都在一个线程里完成，因此 Redis 被称为是单线程的操作。**这种模式快是指它的请求处理过程快**，在单线程中监听多个Socket的请求，在任意一个Socket可读/可写时，Redis去读取客户端请求，在内存中操作对应的数据，然后再写回到Socket中

**单线程好处：**没有了访问共享资源加锁的性能损耗，开发和调试非常友好，可维护性高，减少了多个线程上下文切换带来的额外开销和一些锁竞争

单机redis如果处理大数据请求时还是会出现瓶颈，但是redis有集群高可用解决方案可以解决，主节点只负责写，从节点负责读

**4.管道通讯**

每次Redis客户端操作的时候，命令和元数据都被打包成redis协议进行传输到服务器上。如果每操作一条命令就要一次网络io，那么客户端频繁操作数据就会频繁网络操作，这个过程是很耗时而且影响性能的。所以redis在客户端中做了一些优化引入了一个管道这个概念，管道会把多条无关命令批量执行，以减少多个命令分别执行带来的网络交互时间，在一些批量操作数据的场景

**Redis保证数据不丢失**

redis会从主进程中通过fork()系统调用，创建一个子进程，将父进程的虚拟内存 与 物理内存 映射关系复制到子进程中，并将设置内存共享的，子进程只负责将内存里面数据写入到rdb进行持久化操作，如果在操作的时候主进程对内存修改了，使用写时拷贝技术，将对应的内存创建一个副本然后进行写入持久化。

**持久化机制 RDB和AOF**

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

**RDB：Redis DataBase缩写快照**

RDB是Redis默认的持久化方式。按照一定的时间将内存的数据以快照的形式保存到硬盘中，对应产生的数据文件为dump.rdb。通过配置文件中的save参数来定义快照的周期。

**优点**：1、只有一个文件 dump.rdb，方便持久化。

2、容灾性好，一个文件可以保存到安全的磁盘。

3、性能最大化，fork子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能

4.相对于数据集大时，比AOF的启动效率更高。

**缺点**：1、数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)

2、AOF（Append-only file)持久化方式：是指所有的命令行记录以 redis 命令请 求协议的格式完全持久化存储)保存为 aof 文件。

**AOF**

AOF持久化(即Append Only File持久化)，则是将Redis执行的每次写命令记录到单独的日志文件中，当重启Redis会重新将持久化的日志中文件恢复数据。

当两种方式同时开启时，数据恢复Redis会优先选择AOF恢复。

**优点**：1、数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次 命令操作就记录到 aof 文件中一次。

2、通过 append 模式写文件，即使中途服务器宕机，可以通过 redis-check-aof 工具解决数据一致性问题。

3、AOF 机制的 rewrite 模式。AOF 文件没被 rewrite 之前（文件过大时会对命令 进行合并重写），可以删除其中的某些命令（比如误操作的 flushall）)

**缺点**：1、AOF 文件比 RDB 文件大，且恢复速度慢。

2、数据集大的时候，比 rdb 启动效率低。

**RDB vs AOF**

AOF文件比RDB更新频率高，优先使用AOF还原数据。

AOF比RDB更安全也更大

RDB性能比AOF好

如果两个都配了优先加载AOF

**什么时候用RDB or AOF**

数据安全性，你应该同时使用两种持久化功能

可以承受数分钟以内的数据丢失，那么你可以只使用RDB持久化。

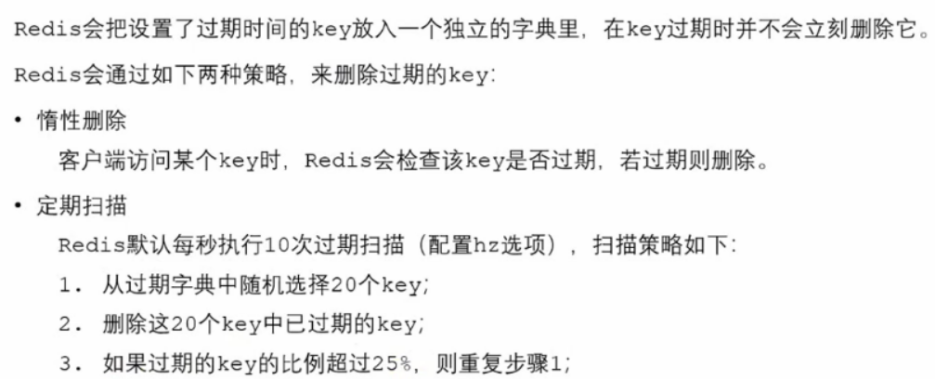
只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。

**扩容**

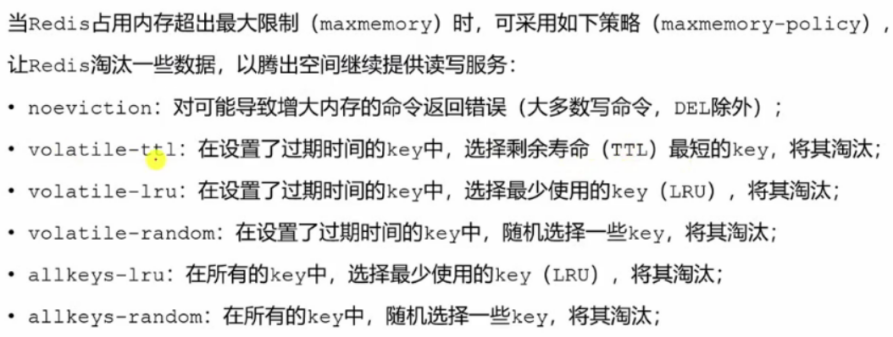
1.如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容。

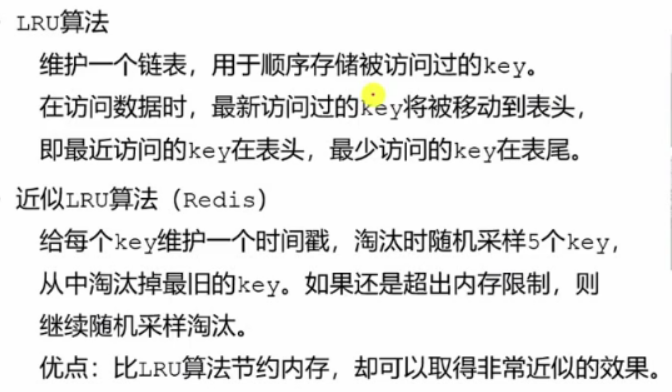
2.如果Redis被当做一个持久化存储使用，必须使用固定的keys-to-nodes映射关系，节点的数量一旦确定不能变化。否则的话(即Redis节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。

**过期策略**



**淘汰策略**





**事务**

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

它是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

Redis 事务的本质是通过MULTI、EXEC、WATCH等一组命令的集合

在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中

**相关命令**

Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的。

Redis会将一个事务中的所有命令序列化，然后按顺序执行。

1）redis 不支持回滚，“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”， 所以 Redis 的内部可以保持简单且快速。

2）如果在一个事务中的命令出现错误，那么所有的命令都不会执行；

3）如果在一个事务中出现运行错误，那么正确的命令会被执行。

WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set （CAS）行为。可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。

MULTI命令用于开启一个事务，它总是返回OK。MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 nulll 。

通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务， 并且客户端会从事务状态中退出。

UNWATCH命令可以取消watch对所有key的监控。

有**隔离性**(Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止)

但没有**原子性**(单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行)

**3种集群模式**

**哨兵(sentinel，一个组件)模式**

基于主从模式做的一定变化，它为Redis提供了高可用性。在实际生产中，可以在一定程度上能够帮我们规避掉服务器宕机，停电，硬件损坏等状况导致的灾难性后果。**哨兵模式的核心还是主从复制**。**只不过相对于主从模式在主节点宕机导致不可写的情况下，多了一个竞选机制**—**从所有的从节点竞选出新的主节点**。竞选机制的实现，是依赖于在系统中启动一个sentinel进程。

**哨兵sentinel功能**：

1.集群监控：负责监控 redis master 和 slave 进程是否正常工作。

2.消息通知：如果某个 redis 实例有故障就发送消息作为报警通知给管理员。

3.故障转移：如果 master node 挂掉了，会从所有的从节点中竞选出一个节点作为主节点。

4.提供主服务器地址：它还能够向使用者提供当前主节点的地址。这在故障转移后，使用者不用做任何修改就可以知道当前主节点地址。

**实现集群化**：部署多个哨兵，哨兵可以通过发布与订阅来自动发现Redis集群上的其它哨兵并且放到一个列表中，这个列表存储了所有已被发现的哨兵

**故障转移：**集群中的所有sentinel不会并发着去对同一个主节点进行故障转移。故障转移只会从第一个sentinel开始，当第一个故障转移失败后，才会尝试下一个。当选择一个从节点作为新的主节点后，故障转移即成功了(而不会等到所有的从节点配置了新的主节点后)。这过程中，如果重启了旧的主节点，那么就会出现无主节点的情况，这种情况下，只能重启集群。

当竞选出新的主节点后，被选为新的主节点的从节点的配置信息会被sentinel改写为旧的主节点的配置信息。完成改写后，再将新主节点的配置广播给所有的从节点。

**哨兵实现高可用：**

1.故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题。

2.即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了

**核心知识：**

哨兵至少需要 3 个实例，来保证自己的健壮性。

哨兵 + redis 主从的部署架构，是不保证数据零丢失的，只能保证 redis 集群的**高可用**

**集群cluster模式**

提供在多个Redis间节点间共享数据的程序集。

数据分片没有使用一致性hash，而是采用slot(哈希槽)的概念，一共分成16384个槽。将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行

**具体实现：**

1.通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了16384 个槽位

2.每份数据分片会存储在多个互为主从的多节点上

3.数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)

4.同一分片多个节点间的数据不保持一致性

5.读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点

6.扩容时需要把旧节点的数据迁移一部分到新节点

**底层：**

每个key通过CRC16校验后对16384取模来决定放置哪个槽。集群的每个节点负责一部分hash槽，举个例子，比如当前集群有3个节点,那么：  
节点 A 包含 0 到 5500号哈希槽。  
节点 B 包含5501 到 11000 号哈希槽。  
节点 C 包含11001 到 16384号哈希槽。

这种结构很容易添加或者删除节点。比如如果我想新添加个节点D， 我需要从节点 A、B、C中得部分槽到D上。如果我想移除节点A，需要将A中的槽移到B和C节点上，然后将没有任何槽的A节点从集群中移除即可。由于从一个节点将哈希槽移动到另一个节点并不会停止服务，所以无论添加删除或者改变某个节点的哈希槽的数量都不会造成集群不可用的状态。

**Redis 集群的主从复制模型 为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有N-1个复制品.**

具有A、B、C三个节点的集群，在没有复制模型的情况下，如果节点B失败了，那么整个集群就会以为缺少5501-11000这个范围的槽而不可用。然而如果在集群创建的时候（或者过一段时间）我们为每个节点添加一个从节点A1、B1、C1。那么整个集群便有三个master节点和三个slave节点组成，这样在节点B失败后，集群便会选举B1为新的主节点继续服务，整个集群便不会因为槽找不到而不可用了。不过当B和B1 都失败后，集群是不可用的。

**优点：**1.自动分割数据到不同的节点上。

2.整个集群的部分节点失败或者不可达的情况下能够继续处理命令。

3.无中心架构，支持动态扩容，对业务透明 4.具备Sentinel的监控和自动Failover(故障转移)能力 5.客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可 6.高性能，客户端直连redis服务，免去了proxy代理的损耗

**缺点：**1.Redis集群不支持处理多个keys的命令，因为这需要在不同的节点间移动数据，所以达不到Redis那样的性能，在高负载的情况下可能会导致不可预料的错误。

2.运维很复杂，数据迁移需要人工干预3.只能使用0号数据库4.不支持批量操作(pipeline管道操作) 5.分布式逻辑和存储模块耦合等

**分布式寻址算法**

1.hash 算法（大量缓存重建）

2.一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡）

3.redis cluster 的 hash slot 算法

**主从模式--高可用**

基于Redis的主从复制特性架构的。通常会设置一个主节点和多个从节点;默认情况下，主节点负责处理使用者的IO操作，从节点会对主节点的数据进行备份，并且也会对外提供读操作的处理。所有的读请求全部走从节点。这样也可以很轻松实现水平扩容，支持**读高并发**

**特点**：

1.当某一节点损坏时，因为其会将数据备份到其它Redis实例上，这样做在很大程度上可以恢复丢失的数据。

2.可以保证负载均衡

3.主节点和从节点是读写分离的。使用者可以从主节点和从节点上读取到数据，这一定程度上缓解了主机的压力。

4.从节点也是能够支持写入数据的，但从从节点写入的数据不会同步到主节点以及其它的从节点下。

4.一个主节点可以拥有多个从节点，但是一个slave只能对应一个master

5.slave挂了不影响其他slave的读和master的读和写，重新启动后会将数据从master同步过来5.master挂了以后，不影响slave的读，但redis不再提供写服务，master重启后redis将重新对外提供写服务

6.master挂了以后，不会在slave节点中重新选一个master

**缺点**：Redis在主从模式下，必须保证主节点不会宕机——一旦主节点宕机，其它节点不会竞争称为主节点，此时，Redis将丧失写的能力。这点在生产环境中，是致命的。

**主从复制原理**：当启动一个 slave node 的时候，它会发送一个 PSYNC 命令给 master node。如果这是slave node 初次连接到 master node，那么会触发一次 full resynchronization 全量复制。此时 master 会启动一个后台线程，开始生成一份 RDB 快照文件。同时还会将从客户端 client 新收到的所有写命令缓存在内存中。RDB 文件生成完毕后， master 会将这个 RDB 发送给 slave，slave 会先写入本地磁盘，然后再从本地磁盘加载到内存中。接着 master 会将内存中缓存的写命令发送到 slave，slave 也会同步这些数据。slave node 如果跟 master node 有网络故障，断开了连接，会自动重连，连接之后 master node 仅会复制给 slave 部分缺少的数据。

**过程**：当从库和主库建立MS关系后，会向主数据库发送SYNC命令

主库接收到SYNC命令后会开始在后台保存快照(RDB持久化过程)，并将期间接收到的写命令缓存起来，当快照完成后，主Redis会将快照文件和所有缓存的写命令发送给从Redis，从Redis接收到后，会载入快照文件并且执行收到的缓存的命令。之后，主Redis每当接收到写命令时就会将命令发送从Redis，从而保证数据的一致

**缺点**：所有的slave节点数据的复制和同步都由master节点来处理，会照成master节点压力太大，使用主从从结构来解决

**Redis一致性保证Redis 并不能保证数据的强一致性。**

这意味这在实际中集群在特定的条件下可能会丢失写操作。

**原因1**：集群是用了异步复制，写操作过程:  
客户端向主节点B写入一条命令；主节点B向客户端回复命令状态；主节点将写操作复制给他得从节点 B1、B2、B3。主节点对命令的复制工作发生在返回命令回复之后， 因为如果每次处理命令请求都需要等待复制操作完成的话， 那么主节点处理命令请求的速度将极大地降低 —— 我们必须在性能和一致性之间做出权衡。 注意：Redis 集群可能会在将来提供同步写的方法。

**原因2**：集群出现了网络分区， 并且一个客户端与至少包括一个主节点在内的少数实例被孤立。

举个例子 假设集群包含 A 、 B 、 C 、 A1 、 B1 、 C1 六个节点， 其中 A 、B 、C 为主节点， A1 、B1 、C1 为A、B、C的从节点， 还有一个客户端 Z1 假设集群中发生网络分区，那么集群可能会分为两方，大部分的一方包含节点 A 、C 、A1 、B1 和 C1 ，小部分的一方则包含节点 B 和客户端 Z1。  
Z1仍然能够向主节点B中写入，如果网络分区发生时间较短，那么集群将会继续正常运作，如果分区的时间足够让大部分的一方将B1选举为新的master，那么Z1写入B中得数据便丢失了。

注意：在网络分裂出现期间， 客户端 Z1 可以向主节点 B 发送写命令的最大时间是有限制的， 这一时间限制称为节点超时时间（node timeout）， 是 Redis 集群的一个重要的配置选项：

**Zookeeper(给分布式应用提供一致性服务的软件)**

**a.用处**：分布式应用可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能

**b.目标**就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户

**c.一致性特征**：1.顺序一致性 2.原子性3.单一视图4.可靠性5.实时性(最终一致性)

**d.怎么保证主从节点状态同步**：

核心是**原子广播机制**，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式：

**恢复模式**：当服务启动或者在leader崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数 server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 server 具有相同的系统状态。

**广播模式**：一旦 leader 已经和多数的 follower 进行了状态同步后，它就可以开始广播消息了，也就是进入广播状态。这时候当一个 server 加入 ZooKeeper 服务中，它会在恢复模式下启动，发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。ZooKeeper 服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分的 followers 支持。

**e.4种数据节点的类型**：**Znode(最小数据单位)** tree 多叉树 可以增删改查

**1.持久节点：**除非手动删除，否则节点一直存在于 Zookeeper 上**2.临时节点：**临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。**3.持久顺序节点：**基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

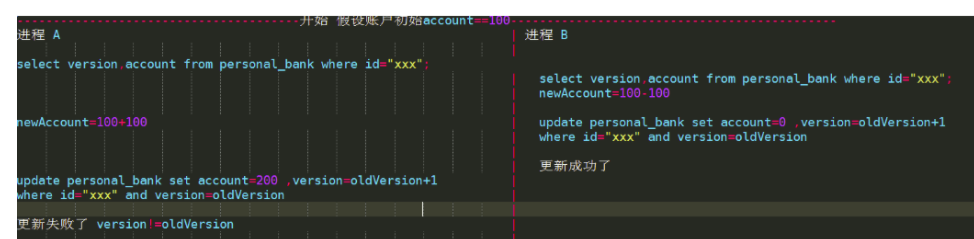
**4.临时顺序节点：**基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字

**分布式锁**

1. **基于数据库实现**

**悲观锁**：select...where...for update (排他锁)，要注意的是“where name=lock ”，name字段必须要走索引，否则会锁表

**乐观锁**：基于CAS思想，是不具有互斥性，不会产生锁等待而消耗资源，操作过程中认为不存在并发冲突，只有update version失败后才能觉察到。我们的抢购、秒杀就是用了这种实现以防止超卖



1. **基于缓存(redis等)实现**

1.获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进行判断。  
2.获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。  
3.释放锁的时候，通过UUID判断是不是该锁，是就执行delete进行锁释放。

使用redis实现分布式锁的时候用到的命令：

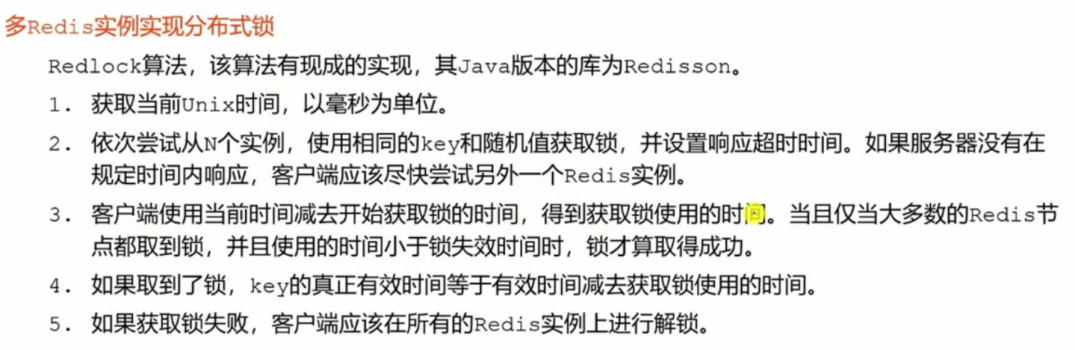
1. SETNX key val：当且仅当key不存在时，set一个key为val的字符串，返回 1；若key存在，则什么都不做，返回0。  
   2.expire key timeout：为key设置一个超时时间，单位为second，超过这个时间 锁会自动释放，避免死锁。  
   3.delete key：删除key

**RedLock特征**

安全特性：互斥访问，即永远只有一个 client 能拿到锁

避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某 资源的 client crash 了或者出现了网络分区

容错性：只要大部分 Redis 节点存活就可以正常提供服务



1. **基于zookeeper实现**
2. 创建一个目录mylock；  
   （2）线程A想获取锁就在mylock目录下创建临时顺序节点；  
   （3）获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获得锁；  
   （4）线程B获取所有节点,判断自己不是最小节点, 设置监听比自己次小的节点；  
   （5）线程A处理完，删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节点，如果是则获得锁。

**优点**：具备高可用、可重入、阻塞锁特性，可解决失效死锁问题。

**缺点**：因为需要频繁的创建和删除节点，性能上不如Redis方式。

**3种实现方式的对比：  
数据库**分布式锁实现  
缺点：

1.db操作性能较差，并且有锁表的风险  
2.非阻塞操作失败后，需要轮询，占用cpu资源;  
3.长时间不commit或者长时间轮询，可能会占用较多连接资源

**Redis(缓存)**分布式锁实现  
缺点：

1.锁删除失败 过期时间不好控制  
2.非阻塞，操作失败后，需要轮询，占用cpu资源;

**ZK**分布式锁实现  
缺点：性能不如redis实现，主要原因是写操作（获取锁释放锁）都需要在Leader上执行，然后同步到follower。

总之：ZooKeeper有较好的性能和可靠性。

从理解的难易程度角度（从低到高）数据库 > 缓存 > Zookeeper

从实现的复杂性角度（从低到高）Zookeeper >= 缓存 > 数据库

从性能角度（从高到低）缓存 > Zookeeper >= 数据库

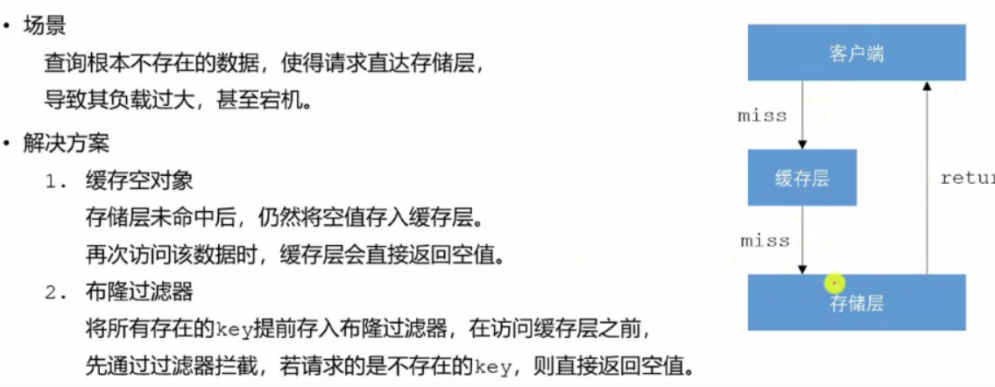
从可靠性角度（从高到低）Zookeeper > 缓存 > 数据库

**[分布式](https://so.csdn.net/so/search?q=%E5%88%86%E5%B8%83%E5%BC%8F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Echouuu/article/details/104372226/_blank)缓存(中央缓存)**

1.发起请求2.检查redis中是否有缓存，有就返回，没有就查询  
3.没有就查询数据库数据4.同步一份数据到redis5.返回结果 **优点：**1.作为独立服务，不会和其他服务抢占资源；  
2.中央缓存不存在数据不同步的问题

**13. 缓存问题**

**缓存穿透**



布隆过滤器(只有0和1的bit数组)-- 对待过滤的数值求hash散列后可以查看这个数组中对应的位置上是否为1来进行判断过滤

也就是引入了k(k>1)k(k>1)个相互独立的哈希函数，保证在给定的空间、误判率下，完成元素判重的过程。

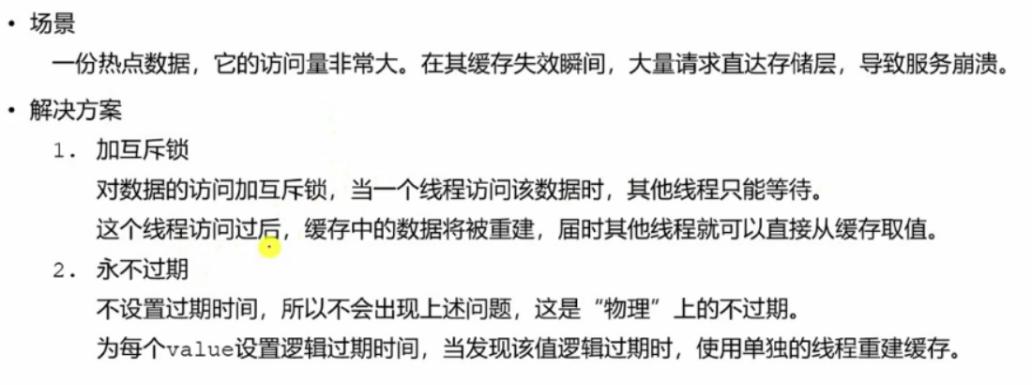
**优点**：是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

Bloom-Filter算法的核心思想就是利用多个不同的Hash函数来解决“冲突”。

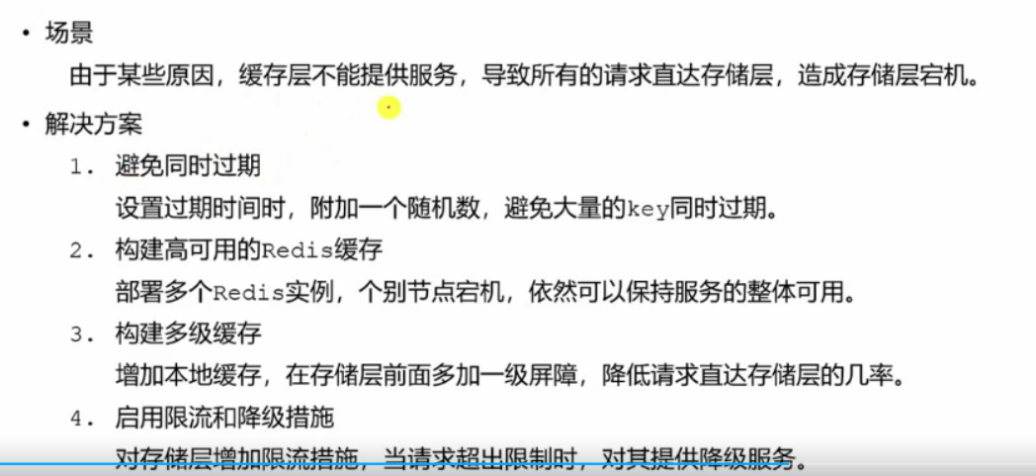
Hash存在一个冲突（碰撞）的问题，用同一个Hash得到的两个URL的值有可能相同。为了减少冲突，我们可以多引入几个Hash，如果通过其中的一个Hash值我们得出某元素不在集合中，那么该元素肯定不在集合中。只有在所有的Hash函数告诉我们该元素在集合中时，才能确定该元素存在于集合中。这便是Bloom-Filter的基本思想。

Bloom-Filter一般用于在大数据量的集合中判定某元素是否存在。

**缓存击穿**



**缓存雪崩**



**缓存预热**

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题，用户直接查询事先被预热的缓存数据

解决方案：1.直接写个缓存刷新页面，上线时手工操作一下2.数据量不大，可以在项目启动的时候自动进行加载 3.定时刷新缓存；

**缓存降级**

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

缓存降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

警告：有些服务在一段时间内成功率有波动（如在95~100%之间）可以自动降级或人工降级，并发送告警；

错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

服务降级的目的，是为了防止Redis服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis出现问题，不去数据库查询，而是直接返回默认值给用户。

**内存 vs 缓存**

内存是作为CPU和硬盘间的存储支撑，缓存是cpu的一部分，在cpu里，逻辑上在内存和CPU之间。cpu读数据很快，内存就慢很多。缓存是为了解决cpu速度和内存速度差异问题的。内存里被cpu访问最频繁的数据和指令被复制到cpu的缓存里。在缓存没命中就去内存和磁盘找(数据库在磁盘里，缓存没命中的话数据库会把这些信息更新到缓存)

**缓存使用/MySQL + Redis架构/先更再删**

**读数据**：先读取缓存，若不存在则从DB中读取，并将结果写入到缓存中；下次数据读取时便可以直接从缓存中获取数据。

**写数据**：**mysql数据更新成功后，再直接删除redis中对应项**。目的是在保证一致性的过程中以mysql中数据为准，redis中的数据始终保持和mysql的同步

**并发高的时候**：写数据异步的话，先写入redis的缓存，就直接返回；定期或特定动作将数据保存到mysql，可以做到多次更新，一次保存；

**为什么不能更新mysql后更新redis**：更新的操作可能是并发的，而写mysql和写redis是两个步骤，不是原子性的。例如有线程1和线程2同时进行写操作，执行顺序可能是如下的情况：

线程1写mysql->线程2写mysql->线程2写redis->线程1写redis

这样mysql的内容是线程2写入的，但redis的内容是线程1写入的，mysql和redis中的数据就不一致了，后续的数据读取都是错的。

而采用每次写完mysql后就清除redis的方式，就保证了写完后的读取必然会重新从mysql读取数据，然后写入redis。这样就保证了redis里的数据最终和mysql中是一致的，保证了数据的最终一致性。

**1.先删缓存 -> 再更新数据库**会导致请求数据不一致

同时一个请求1进行更新操作，一个请求2进行查询操作:

（1）请求1进行写操作，删除缓存

（2）请求2查询发现缓存不存在

（3）请求2去数据库查询得到旧值

（4）请求2将旧值写入缓存

（5）请求1将新值写入数据库

上述情况就会导致不一致的情形出现。而且，如果不采用给缓存设置过期时间策略，该数据永远都是脏数据。

**2.先更新数据库 -> 再删缓存**

（1）缓存刚好失效

（2）请求1查询数据库，得一个旧值

（3）请求2将新值写入数据库

（4）请求2删除缓存

（5）请求1将查到的旧值写入缓存

发生上述情况有一个先天性条件，就是请求2写数据库操作比请求1的读数据库操作耗时更短，才有可能使得请求2删除缓存先于请求1把查到的旧值写入缓存。但是，数据库的读操作的速度远快于写操作的（不然做读写分离干嘛，做读写分离的意义就是因为读操作比较快，耗资源少），所以请求2写数据库操作比请求1的读数据库操作耗时更短这一情形很难出现。

所以虽然先更新数据库，再删缓存依然会有问题，不过，问题出现的可能性会因为上面说的原因，变得比较低

**所以没有办法做到绝对的一致性，两种方法都有缓存一致性问题。**这是由CAP理论决定的，缓存系统适用的场景就是非强一致性的场景，所以它属于CAP中的AP

分布式是指通过网络连接的几个组件，然后通过交换信息协作这样 形成的系统

分布式的CAP定理：Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），最多只能同时三个特性中的两个，三者不可兼得

C：更新操作成功并返回客户端后，所有节点在同一时间的数据完全一致

A：服务一直可用，而且是正常响应时间

P：分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性或可用性的服务

**所以可以委曲求全达到最终一致性**：本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性

**缓存不一致问题**

**1.先删除缓存，再更新数据库的情况下，如果给缓存设置过期时间，那么这个数据永远都是脏数据 / 先更新数据库再删缓存**

解决：延时双删--也就是为了避免更新数据库的时候，其他线程从缓存里读不到数据，就在更新完数据库后，再 Sleep 一段时间，然后再次删除缓存--sleep时间大于读写缓存时间就行

EX：线程1删了缓存然后去更新数据库，线程2发现缓存被删了就去数据库读了旧值，并且它把旧值放回到了缓存，线程1sleep时间之后再去把这个缓存删了

**但是**这种同步淘汰策略**吞吐量低**，所以可以把第二次删除作为**异步**，自己启一个线程，异步删除

**最后一步删缓存失败了怎么办：重试机制**

用有监听binlog(二进制日志文件)消息的消息队列(做核对的工作),也就是借助监听binlog的消息队列来删缓存，这样中间件直接帮忙做了解耦((解除2个东西互相影响的现象) 不用单独引入一个消息队列)

（1）更新数据库

（2）数据库会将操作信息写入binlog日志当中

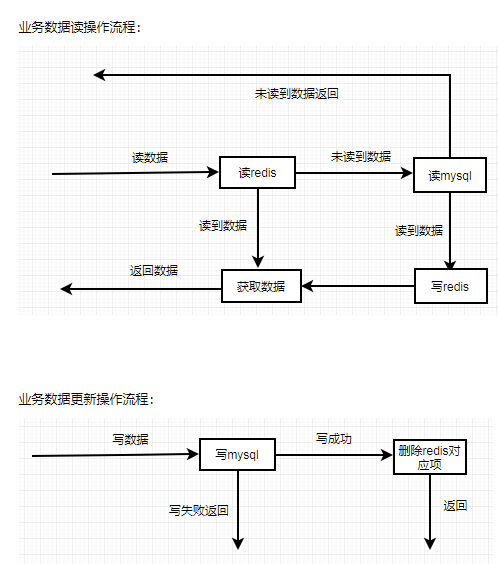
（3）订阅程序提取出所需要的数据以及key

（4）另起一段非业务代码，获得该信息

（5）尝试删除缓存操作，发现删除失败

（6）将这些信息发送至消息队列

（7）重新从消息队列中获得该数据，重试操作。



**Raft分布式协议**

**一般分布式都会遵从的协议**：有点类似美国的总统选举。

1.比如有一个建议，有Ñ个服务器，有大于2N + 1的服务器表示赞同，那么这个建议就被通过。

2.假如建议是“一致性”，只要大于2N + 1的服务器之间的数据一致，那他们就是一致。

3.假如建议是“主服务器”，只要大于2N + 1的服务器投票给要被选举的服务器，被选举的服务器也将变成主服务器

**Rehash**

redis由位桶组成的，随着缓存信息越来越大，就需要对redis进行扩容。然而，之前已经存在的键值对是redis经过[hash](https://so.csdn.net/so/search?q=hash&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/jobali/article/details/_blank)计算计算出的值来存放在对应的位桶上的，所以rehash就是对之前的键值对进行重新计算来扩容

**渐进式哈希(rehash)步骤**：

1.为 ht[1] 分配空间， 让字典同时持有 ht[0] 和 ht[1] 两个哈希表。

2.在字典中维持一个索引计数器变量 rehashidx ， 并将它的值设置为 0 ， 表示 rehash 工作正式开始。

3.在 rehash 进行期间， 每次对字典执行添加、删除、查找或者更新操作时， 程序除了执行指定的操作以外， 还会顺带将 ht[0] 哈希表在 rehashidx 索引上的所有键值对 rehash 到 ht[1] ， 当 rehash 工作完成之后， 程序将 rehashidx 属性的值增一。

4.随着字典操作的不断执行， 最终在某个时间点上， ht[0] 的所有键值对都会被 rehash 至 ht[1] ， 这时程序将 rehashidx 属性的值设为 -1 ， 表示 rehash 操作已完成。

因为在渐进式 rehash的时候， 字典会同时使用 ht[0] 和 ht[1] 两个哈希表， 所以在渐进式 rehash 进行期间， 字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表上进行： 比如说， 要在字典里面查找一个键的话， 程序会先在 ht[0] 里面进行查找， 如果没找到的话， 就会继续到 ht[1] 里面进行查找。另外， 在渐进式 rehash 执行期间， 新添加到字典的键值对一律会被保存到 ht[1] 里面， 而ht[0] 则不再进行任何添加操作： 这一措施保证了 ht[0] 包含的键值对数量会只减不增， 并随着 rehash 操作的执行而最终变成空表