**###############################**REDIS**############################**

### NoSql数据库

NoSql=Not only SQL 即“不仅仅是sql” 泛指非关系型数据库

**Redis**和Memcached类似，也是NoSQL的一种，是一个基于内存的高性能key-value（k-v）数据库。

**适用场景：**

**对数据高并发的读写**

**海量数据的读写**

**对数据高可扩展性**

#### 1.1. Redis和Memcache的不同

redis支持多数据类型，支持持久化，单线程+多路IO复用memcache单一类型不支持持久化 是多线程+锁

reids是一个开源的key-value存储系统，它支持存储的value类型更多，包括：String、list、set、zset(sort set – 有序集合)和Hash类型

### 常用的五大数据类型

#### 2.1、Redis键（key）

keys \* 查看当前库所有key

exists key判断某个key是否存在

type key查看你的key是什么类型

del key 删除指定的key数据

unlink key 根据value选择非阻塞删除

仅将keys从keyspace元数据中删除，真正的删除会在后续异步操作

expire key 10 10秒钟：给定的Key设置过期时间

ttl key查看还有多少秒过期，-1表示永不过期，-2表示已经过期

select命令切换数据库

dbseize查看当前数据库的key的数量

#### 2.2、常用命令

set <key><value>添加键值对

get <key>查询对应键值

append<key><value>将给定的<value>追加到原值末尾

strlen <key>获取值的长度

setnx <key><value>只有在key不存在时 设置的值才有效

incr <key> 将key中存储的数字值增1

只能对数字操作，如果为空，新增值为1

decr <key>

将key中存储的数字值减一

只能对数字操作，如果为空，新增值为-1

incrby / decrby <key> 将Key中存储的数字按照步长增减

原子性：incr key

所谓**原子**操作是指不会被线程调度机制打断的操作；

这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch （切换到另一个线程）。

（1）在单线程中， 能够在单条指令中完成的操作都可以认为是"原子操作"，因为中断只能发生于指令之间。

（2）在多线程中，不能被其它进程（线程）打断的操作就叫原子操作。

Redis单命令的原子性主要得益于Redis的单线程。

**案例：**

java中的i++是否是原子操作？**不是**

i=0;两个线程分别对i进行++100次,值是多少？

mset <key1><value1><key2><value2>

同时设置一个或多个key-value

mget <key1><key2><key3>

同时获取一个或多个value

msetnx <key1><value1><key2><value2>

同时设置一个或多个key-value对，当且仅当所有给定的值都不存在

**原子性，有一个失败则都失败**

getrange <key><起始位置><结束位置>

获得值的范围，类似java中的substring,前包，后包

setrange <key><起始位置><value>

用value覆写所存储的字符串值

serex <key><过期时间><value>

设置键值的同时，设置过期时间，单位秒

getset <key><value>

以新换旧，设置了新值同时获得旧值

**Redis字符串（String）:** String类型是二进制安全的。意味着Redis的string可以包含任何数据，一个Redis中字符串value最多可以是512M.

**Redis列表(List):** Redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部(左边)或者尾部(右边)。它的底层实际是**双向链表，**对两端的操作性能很高

通过索引下标的操作中间的节点性能可能会比较差。

List的数据结构为快速链表quickList。

首先在列表元素比较少的情况下会使用一块连续的内存存储，这个结构是ziplist,也即是压缩列表

当数据量比较多的时候才会改成quicklist

**Redis集合(set)**: Redis的set是string类型的无序集合。它的底层其实是一个value为null的hash表，所以添加，删除，查找的复杂度都是O(1)

**Redis哈希（hash）**：redis hash是一个键值对集合。

redis hash是一个string类型的field和value的映射表，hash特别时候用于存储对象。

类似java里面的Map(String, Object)

**Redis有序集合Zset(sorted set)**: Redis有序集合zset与 普通集合set非常相似，是一个没有重复元素的字符串集合。

不同之处是有序集合的每个成员都关联了一个评分，这个评分被用来按照最低分到最高分的方式排序集合中的成员。集合的成员是唯一的，但是评分可以是重复的

#########################JAVA连接Redis######################

1）首先检查虚拟机的防火墙是否关闭

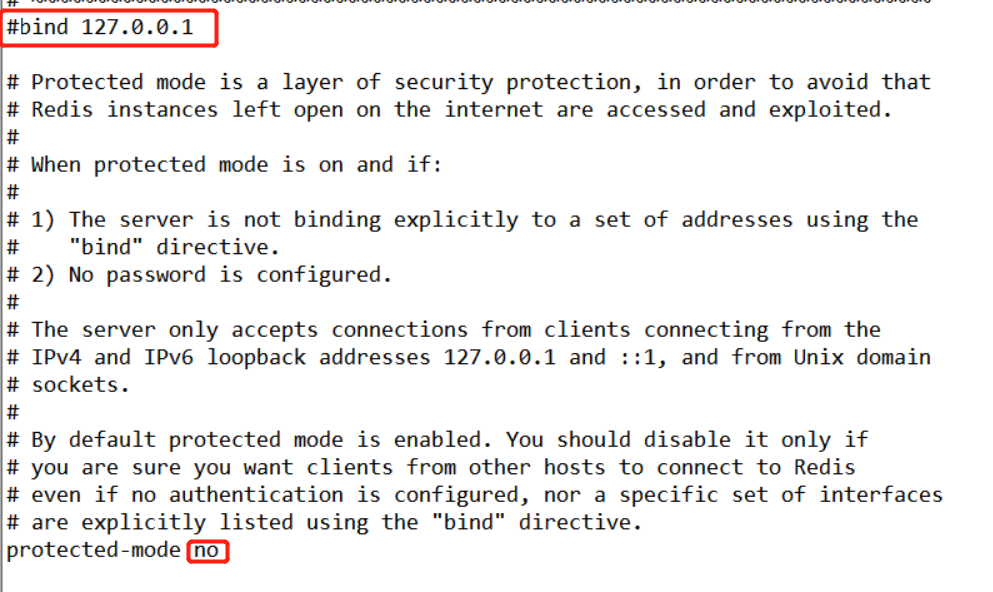
systemctl status firewalld

然后关闭防火墙

2）systemctl stop firewalld

3）修改配置文件redis.conf

注释掉bind，protected-mode为no



开启redis

1. 开启服务器 redis-server /etc/redis.conf
2. 客户端连接：redis-cli

4) 重启redis

redis-cli -p 6379 shutdown  
redis-server /etc/redis.conf

**1、Redis的事务定义：**

redis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，按顺序的

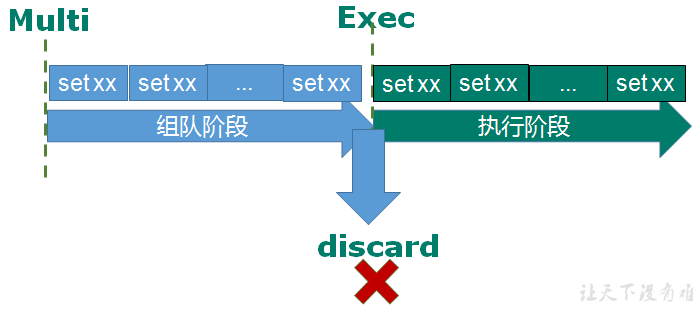
执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

redis事务的主要作用就是串联多个命令防止别的命令插队。

* 1. **Multi、Exec、discard**

从输入Multi命令开始，输入的命令都会依次进入命令队列中，但不会执行，直到输入Exec后，Redis会将之前的命令队列中的命令依次执行。

组队的过程中可以通过discard来放弃组队。



**案例：**

|  |
| --- |
| C:\Users\Administrator\AppData\Local\Temp\ksohtml17908\wps63.jpg  组队成功，提交成功 |
| C:\Users\Administrator\AppData\Local\Temp\ksohtml17908\wps64.jpg  组队阶段报错，提交失败 |

组队中某个命令出现了报告错误，执行时整个的所有队列都会被取消

如果执行阶段某个命令报出了错误，则只有报错的命令不会执行，其他的命令都会执行不会回滚

**事务冲突问题的解决：**

**悲观锁(Pessimistic Lock)**, 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。**传统的关系型数据库里边就用到了很多这种锁机制**，比如**行锁**，**表锁**等，**读锁**，**写锁**等，都是在做操作之前先上锁

**乐观锁(Optimistic Lock),** 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。**乐观锁适用于多读的应用类型，这样可以提高吞吐量**。Redis就是利用这种check-and-set机制实现事务的。

共享锁（S锁）：共享 (S) 用于不更改或不更新数据的操作（只读操作），如 SELECT 语句。

如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。获准共享锁的事务只能读数据，不能修改数据。

排他锁（X锁）：用于数据修改操作，例如 INSERT、UPDATE 或 DELETE。确保不会同时同一资源进行多重更新。

如果事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。

我们在操作数据库的时候，可能会由于并发问题而引起的数据的不一致性（数据冲突）

乐观锁  
乐观锁不是数据库自带的，需要我们自己去实现。乐观锁是指操作数据库时(更新操作)，想法很乐观，认为这次的操作不会导致冲突，在操作数据时，并不进行任何其他的特殊处理（也就是不加锁），而在进行更新后，再去判断是否有冲突了。

通常实现是这样的：在表中的数据进行操作时(更新)，先给数据表加一个版本(version)字段，每操作一次，将那条记录的版本号加1。也就是先查询出那条记录，获取出version字段,如果要对那条记录进行操作(更新),则先判断此刻version的值是否与刚刚查询出来时的version的值相等，如果相等，则说明这段期间，没有其他程序对其进行操作，则可以执行更新，将version字段的值加1；如果更新时发现此刻的version值与刚刚获取出来的version的值不相等，则说明这段期间已经有其他程序对其进行操作了，则不进行更新操作。

悲观锁  
与乐观锁相对应的就是悲观锁了。悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。

说到这里，由悲观锁涉及到的另外两个锁概念就出来了，它们就是共享锁与排它锁。共享锁和排它锁是悲观锁的不同的实现，它俩都属于悲观锁的范畴。

共享锁  
共享锁指的就是对于多个不同的事务，对同一个资源共享同一个锁。相当于对于同一把门，它拥有多个钥匙一样。就像这样，你家有一个大门，大门的钥匙有好几把，你有一把，你女朋友有一把，你们都可能通过这把钥匙进入你们家，进去啪啪啪啥的，一下理解了哈，没错，这个就是所谓的共享锁。  
刚刚说了，对于悲观锁，一般数据库已经实现了，共享锁也属于悲观锁的一种，那么共享锁在mysql中是通过什么命令来调用呢。通过查询资料，了解到通过在执行语句后面加上lock in share mode就代表对某些资源加上共享锁了。

事务三特性：

单独的隔离操作

没有隔离级别的概念

不保证原子性

* RDB：**RDB** 持久化机制，是对 **Redis** 中的数据执行**周期性**的持久化。
* AOF：**AOF** 机制对每条写入命令作为日志，以 **append-only** 的模式写入一个日志文件中，因为这个模式是只追加的方式，所以没有任何磁盘寻址的开销，所以很快，有点像Mysql中的**binlog**。

**redis持久化之RDB(Redis DataBase)**

在指定的时间间隔内将内存中的数据集快照写入磁盘

RDB的缺点是最后一次持久化后的数据可能丢失

**针对RDB方式的持久化，手动触发可以使用：**

　　1）：save：会阻塞当前Redis服务器，直到持久化完成，线上应该禁止使用。

2）：bgsave：该触发方式会fork一个子进程，由子进程负责持久化过程，因此阻塞只会发生在fork子进程的时候。

备份：将\*.rdb的文件拷贝到别的地方（因为当重新启动redis服务的时候，会覆盖掉备份的\*rdb文件）

rdb恢复

关闭redis

先把备份的文件拷贝到redis备份文件所在的目录下cp dump2.rdb. dump.rdb

启动redis,备份数据会直接加载

RDB的优缺点

优点：

适合大规模数据恢复

节省空间

对数据完整性和一致性要求不高

恢复速度快

节省磁盘空间(相较于AOF)，因为它存储数据的rdb文件是压缩过的。

恢复速度快。

缺点：

redis在fork时使用了写时拷贝技术，但是如果数据庞大时比较消耗性能。

备份周期在一定时间间隔做一次备份，在这个周期中如果以外Down掉，则会丢失最后一次快照后的所有修改。

**redis持久化之AOF(Append Only File)**

以日日志的形式记录每个写操作（增量保存），将Redis执行过的所有写指令记录下来(读操作不记录)，只许追加文件但不可以改写文件

**AOF默认不开启，可以在配置文件中修改appendonly yes**

**AOF和RDB同时开启，系统默认取AOF等待数据（数据不会存在丢失）**

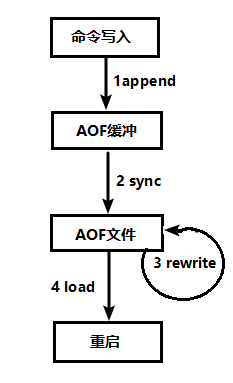
* + 1. **AOF持久化流程**

（1）客户端的请求写命令会被append追加到AOF缓冲区内；

（2）AOF缓冲区根据AOF持久化策略[always,everysec,no]将操作sync同步到磁盘的AOF文件中；

（3）AOF文件大小超过重写策略或手动重写时，会对AOF文件rewrite重写，压缩AOF文件容量；

（4）Redis服务重启时，会重新load加载AOF文件中的写操作达到数据恢复的目的；



当AOF文件的体积大于64Mb，并且AOF文件的体积比上一次重写之久的体积大了至少一倍（100%）时，Redis将执行 bgrewriteaof 命令进行重写

**aof的恢复和rdb的恢复基本一样**

* 异常恢复

修改默认的appendonly no 改为yes

如遇到AOF文件损坏，通过/usr/local/redis-check-aof - -fix appendonly.aof进行恢复

恢复：重启redis,然后重新加载

* 同步频率设置

appendfsync always

始终同步，每次redis的写入都会立刻记入日志，性能较差但数据完整性好

appendfsync everysec

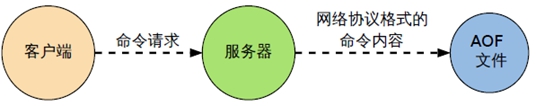
每秒同步，每秒计入日志一次，如果宕机，本秒的数可能丢失

appendfsync no

redis不主动机进行同步，把同步时机交给操作系统

AOF采用文件追加方式，文件会越来越大为避免出现此种情况，新增了重写机制, 当AOF文件的大小超过所设定的阈值时，Redis就会启动AOF文件的内容压缩， 只保留可以恢复数据的最小指令集.可以使用命令bgrewriteaof

* + 1. **优势**



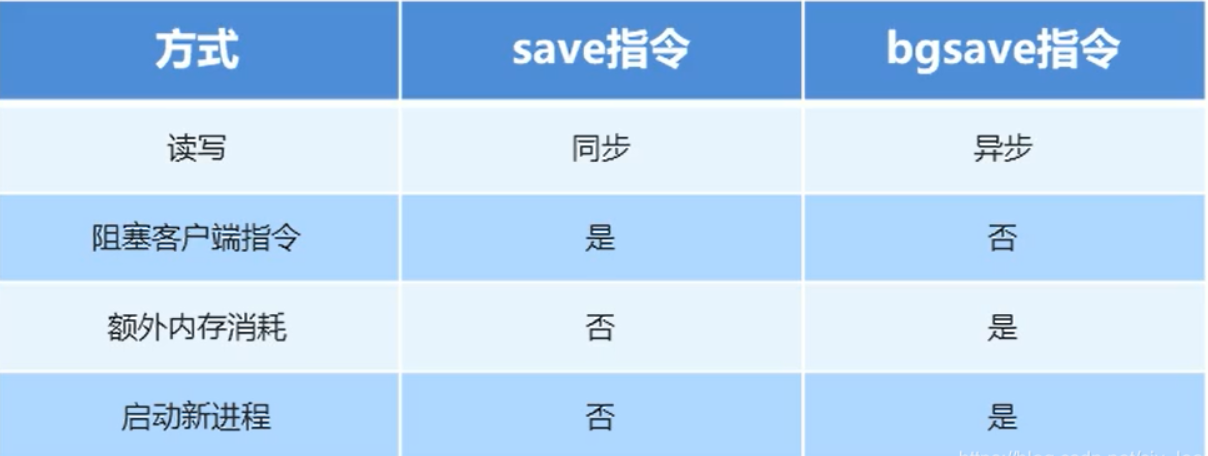
* 备份机制更稳健，丢失数据概率更低。
* 可读的日志文本，通过操作AOF稳健，可以处理误操作。
  + 1. **劣势**
* 比起RDB占用更多的磁盘空间。
* 恢复备份速度要慢。
* 每次读写都同步的话，有一定的性能压力。

存在个别Bug，造成恢复不能。

一般来说， 如果想达到足以媲美 PostgreSQL 的数据安全性， 你应该同时使用两种持久化功能。

如果你非常关心你的数据， 但仍然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。

有很多用户都只使用 AOF 持久化， 但并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快。



**一仆二主**

redis\_主从复制

主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制,master以写为主，slave以读为主

读写分离，性能扩展

容灾快速恢复

启动三个redis服务器

查看当前主机运行状态：info replication

* 配从库不配主库

slaveof <ip> <port>

称为某个实例的从服务器

在从机上执行slaveof 主机ip 端口号

1. 当从服务器挂掉时（shutdown），再次开启从服务器，此时这个服务器变为主服务器

需要再次配从库才能作为从服务器，此时从服务器里面的内容是主服务器开启后的内容。

1. 当主服务器挂掉时，从服务器还是从服务器，从服务器的主服务器也不变，只是从服务器知道主服务器已经挂掉了（shutdown）,当主服务器再次开启后，从服务器还包含之前的那些从服务器

**主从复制原理：**

#### 过程原理

1. 当从库和主库建立MS关系后，会向主数据库发送SYNC命令
2. 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
3. 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
4. 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
5. 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
6. 和Mysql主从复制的原因一样，Redis虽然读取写入的速度都特别快，但是也会产生读压力特别大的情况。为了分担读压力，Redis支持主从复制，Redis的主从结构可以采用一主多从或者级联结构，Redis主从复制可以根据是否是全量分为全量同步和增量同步。下图为级联结构。
7. **全量同步**  
   Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤如下：   
   -  从服务器连接主服务器，发送SYNC命令；   
   -  主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；   
   -  主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；   
   -  从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；   
   -  主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；   
   -  从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；
8. 完成上面几个步骤后就完成了从服务器数据初始化的所有操作，从服务器此时可以接收来自用户的读请求。
9. **增量同步**  
   Redis增量复制是指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。   
   增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。  
      
   **Redis主从同步策略**  
   主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。

当从连接上主服务器后，从服务器向主服务发送进行数据同步的消息，当主服务器接到从服务器发送过来的同步消息，把主服务器数据进行持久化到rdb文件，把rdb文件发送从服务器，从服务器拿到rdb文件进行读取

每次主服务器进行写操作之后，主动向从服务器进行数据同步

**薪火相传：**

上一个slave可以是下一个slave的master,slave同样可以接收其他slaves的连接和同步请求，那么该slave作为了链条中下一个master可以有效减轻masetr的写压力，去中心化降低风险

风险是一旦slave宕机了，后面的slave都无法备份

主机挂了，从机还是从机，无法写数据。

**反客为主：**

当一个master宕机了，后面的slave可以立刻升为master，其后面的slave不用做任何修改 用slaveof no one 将从机变为主机

**哨兵模式：**

反客为主的主动版，能够后台监控主机是否障碍，如果故障了根据投票数自动将从库转换为主库

创建sentinel.conf配置文件

填写内容：sentinel monitor mymaster 127.0.0.1 6379 1

其中 mymaster为监控对象起的服务器的名称，1至少有多少个哨兵同意迁移的数量

启动哨兵：redis-sentinel sentinel.conf

当主机挂掉时，会选择一个从机作为主服务器 根据配置文件内的replica-priority 值越小优先级越高作为主机

当之前挂掉的主服务器开后后，则会变成此时主服务器的从机

复制延时：由于所有的写操作都是先在master上操作，然后同步更新到slave上，所以从master同步到slave机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，slave机器数量的增加也会使这个问题更加严重

**Redis集群为什么至少需要三个master节点，并且推荐节点数为奇数？**

因为新master的选举需要大于半数的集群master节点同意才能选举成功，如果只有两个master节点，当其中一个挂了，是达不到选举新master的条件的。

**redis三种集群方式：主从复制，哨兵模式，集群cluster**

**集群：**

容量不够，redis如何进行扩容？

并发写操作，redis如何分摊

另外，主从模式，薪火相传，主机宕机，导致ip地址发生变化，应用程序中配置需要修改对应的主机地址、端口信息。

Redis 集群实现了对Redis的水平扩容，即启动N个redis节点，将整个数据库分布存储在这N个节点中，每个节点存储总数据的1/N。

Redis集群通过分区来提供一定程度的可用性：即使集群中有一部分节点失效或者无法进行通讯，集群也可以继续处理命令请求。

之前通过代理主机来解决，但是redis3.0中提供了解决方案。就是无中心化集群配置。

以集群方式登录：redis-cli -c -p 6379

设置数据会自动切换到相应的写主机

通过cluster nodes 命令查看集群信息

一个集群至少使用三个主节点

分配原则尽量保证每个主数据库运行在不同的ip地址，每个从库和主库不在一个ip地址上。

一个redis集群包含16384个插槽，数据库中的每一个健都属于这16384个插槽的其中一个

集群使用CRC16(key)%16384来计算健Key属于哪个槽

查询集群中的值：

cluster keyslot k1:返回k1对应插槽的槽值

cluster countkeysinslot 4832:返回对应槽的健的个数

cluster getkeysinslot 4847 10:返回该插槽10个键

只能找到自己的插槽进行查看，无法查看其他节点的插槽

**故障恢复**：

如果主节点下线，从节点自动升为主节点，主节点恢复后变成从机

如果某一段插槽的主从节点都宕机了，而cluster-require-full-converage为yes，那么，整个集群都挂掉

如果某一段插槽主从都挂掉了，而cluster-require-full-converage为no，那么只有该插槽的数据不能使用和存储

**哨兵模式的优缺点**

**优点：**

* 哨兵模式是基于主从模式的，所有主从的优点，哨兵模式都具有。
* 主从可以自动切换，系统更健壮，可用性更高。

**缺点：**

* 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。

**3.Redis-Cluster集群**

redis的哨兵模式基本已经可以实现高可用，读写分离 ，但是在这种模式下每台redis服务器都存储相同的数据，很浪费内存，所以在redis3.0上加入了cluster模式，实现的redis的分布式存储，也就是说每台redis节点上存储不同的内容。

 Redis-Cluster采用无中心结构,它的特点如下：

* 所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽。
* 节点的fail是通过集群中超过半数的节点检测失效时才生效。
* 客户端与redis节点直连,不需要中间代理层.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

**Redis6应用问题解决**

**缓存穿透**

问题描述：key对应的数据在数据源并不存在、每次针对此key的请求从缓存获取不到，请求都会压到数据源、从而可能压垮数据源。

解决方案：

1. 对空值缓存：如果一个查询返回的数据为空（不管数据是否存在）仍然把 这个空结果进行缓存，设置空结果的过期时间会很短，最长超过五分钟
2. 设置可访问的白名单(白名单)：使用bitmaps类型定义一个可以访问的名单，名单id作为bitmaps的偏移量，每次访问和bitmaps里面的id进行比较，如果访问的id不在bitmaps里面，

进行拦截，不允许访问。

1. 采用布隆过滤器：是1970年布隆提出的。它实际上是一个很长的二进制向量(位图)和一系列随机映射函数(哈希函数)。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一点的误识别率和删除困难

将所有肯存在的数据哈希到一个足够大的bitmaps中，一个一定不存在的数据会被这个bitmaps拦截掉，从而避免了对底层存储系统的查询压力。

1. 进行实时监控：当发现redis的命中率开始急速降低，需要排查范文对象和访问数据，和运维人员配合，可以设置黑名单限制服务

缓存击穿

问题描述：key对应的数据存在，但在redis中过期，此时若有大量并发请求过来这些请求发现缓存过期都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可以瞬间把后端DB压垮

解决方案：

1. 预先设置热门数据：在redis高访问之前，把一些热门数据提前存入到redis里面，加大这些热门数据Key的时长
2. 实时调整：现场监控哪些数据热门，实时调整key的过期时长
3. 使用锁：

**缓存雪崩**

问题描述：key对应的数据存在，但在redis中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案：



所谓原子操作是指不会被线程调度机制打断的操作

(说人话 ： 两个线程之间的操作互不影响,每个操作都是独立的线程)

* 单线程中，能够在单条指令中完成的操作都可以认为是"原子操作"，因为终端只能发生于指令之间
* 在多线程中，不能被其他进程（线程）打断的操作就叫原子操作

Redis 单命令的原子性主要得益于Redis的单线程

对于Redis而言，命令的原子性指的是：一个操作的不可以再分，操作要么执行，要么不执行

++1就不是原子操作 可以再分

Redis的操作之所以是原子性的，是因为Redis是单线程的。

### 什么是原子性操作

在多进程（线程）访问共享资源时，能够确保所有其他的进程（线程）都不在同一时间内访问相同的资源，（要么完全执行，要么完全不执行）

### 简单描述

有个变量X=0，要进行+1操作，步骤如下：

* 读取x；
* 取得+1的计算结果
* 写入x

现在有人进行另一个操作 +2；步骤如下;

* 读取x；
* 取得+2计算结果
* 写入x

如果操作不是原子性的，指令就会错乱，得到的结果可能是1,2,3情况均可能出现；  
如果操作是原子性的，就可以保证在进行加1操作的时候其他操作无法插入进来，在完成操作后他才能开始进行他的操作，从而保证数据是可靠的。

++1就不是原子操作 可以再分

### redis的原子性

Redis所有单个命令的执行都是原子性的，这与它的单线程机制有关；  
Redis命令的原子性使得我们不用考虑并发问题，可以方便的利用原子性自增操作 实现简单计数器功能;

#########################面试题##############################

## 过期键的删除策略

### Redis的过期键的删除策略

我们都知道，Redis是key-value数据库，我们可以设置Redis中缓存的key的过期时间。Redis的过期策略就是指当Redis中缓存的key过期了，Redis如何处理。

过期策略通常有以下三种：

* 定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。
* 惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
* 定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。  
  (expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。)

Redis中同时使用了惰性过期和定期过期两种过期策略。

### Redis key的过期时间和永久有效分别怎么设置？

EXPIRE和PERSIST命令。

### 我们知道通过expire来设置key 的过期时间，那么对过期的数据怎么处理呢?

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

1. 定时去清理过期的缓存；
2. 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

## 内存相关

### MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据

redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

### Redis的内存淘汰策略有哪些

Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

全局的键空间选择性移除

* noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。
* allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。（这个是最常用的）
* allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。

设置过期时间的键空间选择性移除

* volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。
* volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。
* volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

**总结**

Redis的内存淘汰策略的选取并不会影响过期的key的处理。内存淘汰策略用于处理内存不足时的需要申请额外空间的数据；过期策略用于处理过期的缓存数

### Redis如何做内存优化？

可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key，而是应该把这个用户的所有信息存储到一张散列表里面

一、背景

线上你写代码的时候，想当然的认为写进 redis 的数据就一定会存在，后面导致系统各种 bug，谁来负责？

常见的有两个问题：

往 redis 写入的数据怎么没了？

可能有同学会遇到，在生产环境的 redis 经常会丢掉一些数据，写进去了，过一会儿可能就没了。我的天，同学，你问这个问题就说明 redis 你就没用对啊。redis 是缓存，你给当存储了是吧？

啥叫缓存？用内存当缓存。内存是无限的吗，内存是很宝贵而且是有限的，磁盘是廉价而且是大量的。可能一台机器就几十个 G 的内存，但是可以有几个 T 的硬盘空间。redis 主要是基于内存来进行高性能、高并发的读写操作的。

那既然内存是有限的，比如 redis 就只能用 10G，你要是往里面写了 20G 的数据，会咋办？当然会干掉 10G 的数据，然后就保留 10G 的数据了。那干掉哪些数据？保留哪些数据？当然是干掉不常用的数据，保留常用的数据了。

数据明明过期了，怎么还占用着内存？

这是由 redis 的过期策略来决定。

二、分析

redis 过期策略

redis是单线程，收割时间也会占用线程处理时间，如果收割过于频繁，会导致读写出现卡顿。

1、主库过期策略

## [定期删除策略](https://www.zhihu.com/search?q=%E5%AE%9A%E6%9C%9F%E5%88%A0%E9%99%A4%E7%AD%96%E7%95%A5&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A91212102%7D)

Redis 会将每个设置了过期时间的 key 放入到一个独立的字典中，默认每 100ms 进行一次过期扫描：

1. 随机抽取 20 个 key
2. 删除这 20 个key中过期的key
3. 如果过期的 key 比例超过 1/4，就重复步骤 1，继续删除。

如果在同一时间出现大面积 key 过期，Redis 循环多次扫描过期词典，直到过期的 key 比例小于 1/4。这会导致卡顿，而且在高并发的情况下，可能会导致[缓存雪崩](https://www.zhihu.com/search?q=%E7%BC%93%E5%AD%98%E9%9B%AA%E5%B4%A9&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A91212102%7D)。

1.1.1、从过期key字典中，随机找20个key。

1.1.2、删除20个key中过期的key

1.1.3、如果2中过期的key超过1/4，则重复第一步

1.1.4、每次处理的时间都不会25ms

如果有大量的key在同一时间段内过期，就会造成数据库的集中访问，就是缓存雪崩！

1.2、惰性策略

客户端访问的时候，会对这个key的过期时间进行检查，如果过期了就立即删除。惰性策略是对定时策略的补充，因为定时策略不会删除所有过期的key

2、从库过期策略

redis不会扫描从库，删除主库数据的时候，在aof文件里生成一条del指令，在主从同步的时候，从库会执行这条指令，删除过期key。

所以集群分布式锁算法的漏洞就是这样产生的。

三、常见的几种缓存失效策略

FIFO ，first in first out ，最先进入缓存的数据在缓存空间不够情况下(超出最大元素限制时)会被首先清理出去

LFU ， Less Frequently Used ，一直以来最少被使用的元素会被被清理掉。这就要求缓存的元素有一个hit 属性，在缓存空间不够得情况下,hit 值最小的将会被清出缓存。

LRU ，Least Recently Used ，最近最少使用的，缓存的元素有一个时间戳，当缓存容量满了，而又需要腾出地方来缓存新的元素的时候，那么现有缓存元素中时间戳离当前时间最远的元素将被清出缓存。

四、缓存更新策略

1. 先删缓存，再更新数据库
2. 先更新数据库，再删缓存
3. 先更新数据库，再更新缓存
4. read/write through
5. 写回。在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库

更新缓存的设计模式有四种：Cache aside, Read through, Write through, Write behind caching

1、Cache aside

读取：

失效：应用程序先从cache取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。

命中：应用程序从cache中取数据，取到后返回。

更新：先把数据存到数据库中，成功后，再让缓存失效。

Read/Write Through Pattern

我们可以看到，在上面的Cache Aside套路中，我们的应用代码需要维护两个数据存储，一个是缓存（Cache），一个是数据库（Repository）。所以，应用程序比较啰嗦。而Read/Write Through套路是把更新数据库（Repository）的操作由缓存自己代理了，所以，对于应用层来说，就简单很多了。可以理解为，应用认为后端就是一个单一的存储，而存储自己维护自己的Cache。

2、Read Through

Read Through 套路就是在查询操作中更新缓存，也就是说，当缓存失效的时候（过期或LRU换出），Cache Aside是由调用方负责把数据加载入缓存，而Read Through则用缓存服务自己来加载，从而对应用方是透明的。

3、Write Through

Write Through 套路和Read Through相仿，不过是在更新数据时发生。当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回。如果命中了缓存，则更新缓存，然后再由Cache自己更新数据库（这是一个同步操作）

4、Write Behind Caching Pattern

Write Back套路，一句说就是，在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。这个设计的好处就是让数据的I/O操作飞快无比（因为直接操作内存嘛 ），因为异步，write backg还可以合并对同一个数据的多次操作，所以性能的提高是相当可观的。

但是，其带来的问题是，数据不是强一致性的。

redis 过期策略是：定期删除+惰性删除。

所谓定期删除，指的是 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。

假设 redis 里放了 10w 个 key，都设置了过期时间，你每隔几百毫秒，就检查 10w 个 key，那 redis 基本上就死了，cpu 负载会很高的，消耗在你的检查过期 key 上了。注意，这里可不是每隔 100ms 就遍历所有的设置过期时间的 key，那样就是一场性能上的灾难。实际上 redis 是每隔 100ms 随机抽取一些 key 来检查和删除的。

但是问题是，定期删除可能会导致很多过期 key 到了时间并没有被删除掉，那咋整呢？所以就是惰性删除了。这就是说，在你获取某个 key 的时候，redis 会检查一下 ，这个 key 如果设置了过期时间那么是否过期了？如果过期了此时就会删除，不会给你返回任何东西。

获取 key 的时候，如果此时 key 已经过期，就删除，不会返回任何东西。

但是实际上这还是有问题的，如果定期删除漏掉了很多过期 key，然后你也没及时去查，也就没走惰性删除，此时会怎么样？如果大量过期 key 堆积在内存里，导致 redis 内存块耗尽了，咋整？

答案是：走内存淘汰机制。

内存淘汰机制

redis 内存淘汰机制有以下几个：

noeviction: 不删除策略, 达到最大内存限制时, 如果需要更多内存, 直接返回错误信息。 大多数写命令都会导致占用更多的内存(有极少数会例外, 如 DEL )。

allkeys-lru：所有key通用; 优先删除最近最少使用(less recently used ,LRU) 的 key。

allkeys-random： 所有key通用; 随机删除一部分 key。

volatile-lru：只限于设置了 expire 的部分; 优先删除最近最少使用(less recently used ,LRU) 的 key。

volatile-random：只限于设置了 expire 的部分; 随机删除一部分 key。

volatile-ttl：只限于设置了 expire 的部分; 优先删除剩余时间(time to live,TTL) 短的key。

缓存一般是为了应对高并发场景、缓解数据库读写压力，而将数据存储在读写更快的某种存储介质中（如内存），以加快读取数据的速度。缓存一般分为本地缓存（如[java堆内存缓存](https://www.zhihu.com/search?q=java%E5%A0%86%E5%86%85%E5%AD%98%E7%BC%93%E5%AD%98&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A86396877%7D)）、分布式缓存（如redis)等。

既然是缓存，就意味着缓存中暂存的数据只是个副本，也就意味着需要保证副本和主数据之间的数据一致性，这就是接下来要分析的缓存的更新。

常见的缓存更新策略有：

1. 先删缓存，再更新数据库
2. 先更新数据库，再删缓存
3. 先更新数据库，再更新缓存
4. read/write through
5. 写回。在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库

## 一、先删缓存再更新数据库

很明显这个逻辑是有问题的，假设有两个并发操作，一个操作更新、另一个操作查询，更新操作删除缓存后还没来得及更新数据库，此时另一个用户发起了查询操作，它因没有[命中缓存](https://www.zhihu.com/search?q=%E5%91%BD%E4%B8%AD%E7%BC%93%E5%AD%98&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A86396877%7D)进而从数据库读，此时第一个操作还没到更新数据库的阶段，读取到的是老数据，接着写到缓存中，导致缓存中数据变成脏数据，并且会一直脏下去直到缓存过期或发起新的更新操作。

## 二、先更新数据库，再删缓存

**这是目前业界最常用的方案**。虽然它同样不够完美，但问题发生的概率很小，它的读流程和写流程见下图（图片来源于网络，侵删）：

写操作先更新数据库，更新成功后使缓存失效。读操作先读缓存，缓存中读到了则直接返回，缓存中读不到再读数据库，之后再将数据库数据加载到缓存中。

但它同样也有问题，如下图，查询操作未命中缓存，接着读数据库老数据之后、写缓存之前，此时另一个用户发起了更新操作更新了数据库并清了缓存，接着查询操作将数据库中老数据更新到缓存。这就导致缓存中数据变成脏数据，并且会一直脏下去直到缓存过期或发起新的更新操作。

为什么这种思路存在这么明显的问题，却还具有那么广泛的应用呢？因为这个case实际上出现的概率非常低，产生这个case需要具备如下4个条件

1. 读操作读缓存失效
2. 有个并发的写操作
3. 写操作比读操作更快
4. 读操作早于写操作进入数据库，晚于写操作更新缓存

而实际上数据库的写操作会比读操作慢得多，而且还要锁表，而读操作必需在写操作前进入数据库操作，而又要晚于写操作更新缓存，所有的这些条件都具备的概率基本并不大。并且即使出现这个问题还有一个缓存过期时间来自动兜底。

## 三、先更新数据库，再更新缓存

相对来讲，理论上这种方式比先更新数据库再删缓存有着更高的读性能，因为它事先准备好数据。但由于要更新数据库和缓存两块数据，所以它的写性能就比较低，而且关键在于它也会出现脏数据，如下图，两个并发更新操作，分别出现一前一后写数据库、一后一前写缓存，则最终缓存的数据是二者中前一次写入的数据，不是最新的。

## 四、read/write through 缓存代理

Read/Write Through套路是把更新数据库（Repository）的操作由缓存自己代理了，所以，对于应用层来说，就简单很多了。**可以理解为，应用认为后端就是一个单一的存储，而存储自己维护自己的Cache。**数据库由缓存代理，缓存未命中时由缓存加载数据库数据然后应用从缓存读，写数据时更新完缓存后同步写数据库。应用只感知缓存而不感知数据库。

## 五、写回

这种方式英文名叫Write Behind 又叫 Write Back。一些了解Linux操作系统内核的同学对[write back](https://www.zhihu.com/search?q=write+back&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A86396877%7D)应该非常熟悉，这不就是Linux文件系统的Page Cache的算法吗？是的，就是那个东西。这种模式是指在更新数据的时候，只更新缓存，不更新数据库，而我们的缓存会异步地批量更新数据库。

这种方式的问题在于数据不是强一致性的，而且可能会丢失（我们知道Unix/Linux非正常关机会导致数据丢失，就是因为这个）。另外，Write Back实现逻辑比较复杂，因为他需要track有哪数据是被更新了的，需要刷到持久层上。操作系统的write back会在仅当这个cache需要失效的时候，才会被真正持久起来，比如，内存不够了，或是进程退出了等情况，这又叫[lazy write](https://www.zhihu.com/search?q=lazy+write&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A86396877%7D)。

## 六、总结

本文归纳了常见的缓存更新的五种思路，其中先更新数据库再删缓存的思路是目前使用得最多的。先删缓存再更新数据库因为出问题概率太大并没有什么用。第三到第五种思路在特定的应用场景下也有很多用途，比如先更新数据库再更新缓存可以解决高并发下缓存未命中导致瞬时大量请求穿透到数据库的问题。每一种方案也有其各自的优点和不足，**总而言之，没有完美的方案，只有契合场景的更适合的方案。**