**1.什么是redis**

Redis是一个开源的、使用c语言编写的、可基于内存也可持久化的key-value数据库

**2.redis的数据结构**

Redis是一种高级的key：value存储系统，其中value支持五种数据类型

①strings

②lists，字符串列表

③set，字符串集合

④sorted set有序字符串集合

⑤hashes 哈希

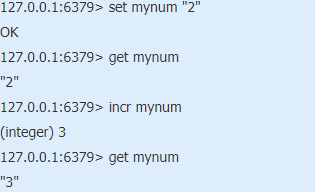
Key的注意事项：key不要太长，尽量不超过1024字节，这会消耗内存，降低查找效率；也不要太短，可读性降低；最好使用统一命名，user：10000：password

**2.1 strings**

Strings类型是一个基础数据类型

set mystr "hello world!" //设置字符串类型内容

get mystr //读取字符串类型内容



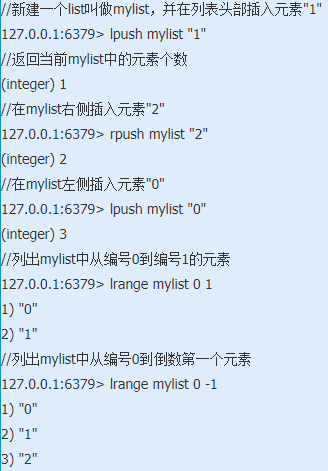
遇到数值操作时，redis会将字符串类型，转换为数值类型

Incr、decr都是原子操作，能够实现原子计数效果

**2.2 lists**

Redis中的lists在底层实现上并不是数组，而是链表，也就是说对于一个具有上百万个元素的lists来说，头部和尾部插入一个新元素，其时间复杂度是常数级别，在10个元素lists头部插入新元素与上千个元素的lists头部插入新元素的速度应该是相同的。

常用操作，LPUSH，RPUSH，LRANGE等，左插，右插，指定范围提取元素

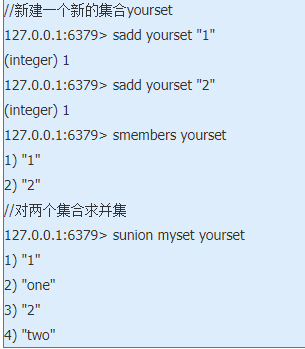
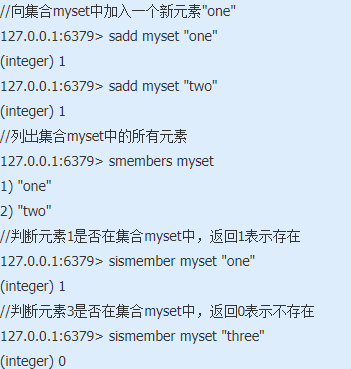


Lists的应用

利用lists实现一个消息队列，而且可以保证先后顺序，利用LRANGE能够实现分页功能。博客系统中每篇博文的评论也可以存入一个单独的lists中

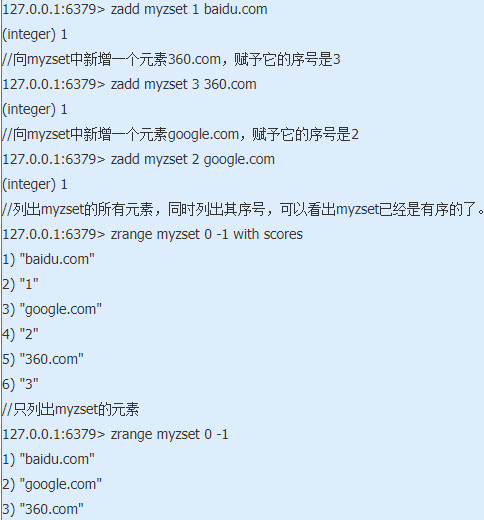
**2.3 无序集合 sets**

没有顺序的集合



**2.4 有序集合 sorted sets**

有序集合一般称为zsets，相关操作都是z开头的



**2.5 hashes**

存的是字符串与字符串值之间的映射



**3.redis的持久化**

Redis提供了两种持久化的方式，分别是RDB（Redis DataBase） 和AOF （Append Only File），RDB就是在不同时间点，将redis存储的数据生成快照并存储到磁盘介质上；AOF是将redis执行过的所有写指令记录下来，在下次redis重新启动时，把这些写指令从前到后重复执行一遍，就可以实现数据恢复了。

RDB和AOF两种方式也可以同时使用，这种情况下，如果redis重启的话，右下采用AOF来进行数据恢复，因为AOF数据恢复完整度高

**3.1 RDB持久化**

将redis某一时刻的数据持久化到磁盘中，是一种快照式的持久化方法

先将数据写入到一个临时文件中，待持久化过程都结束了，**才会用这个临时文件替换上次持久化好的文件**，正是这种特性，我们可以**随时进行备份，因为快照文件总是完整可用的**。

1、save

该命令会阻塞当前Redis服务器，执行save命令期间，**Redis不能处理其他命令**，直到RDB过程完成为止。

2.bgsave

执行bgsave，fork出子进程，然后子进程将数据存到RDB文件中，主进程继续执行请求，更新数据，最终子进程会将原来数据与更新数据都写入到RDB文件中，再用新的RDB文件替换旧的文件

如果需要进行大规模的数据恢复，且对于数据恢复的完整性不是很敏感，RDB比AOF高效

如果对完整性敏感，那么RDB方式就不适合，因为即使每5分钟都持久化一次，当redis故障的时候，仍然会有近5分钟的数据丢失

**3.2 AOF持久化（append only file）只追加不改写**

将执行过的写指令记录下来，在数据恢复时，按照从前到后的顺序将指令执行一遍。

配置redis.conf中的appendonly yes 就可以打开AOF功能，如果有写操作set等，redis命令就会被追加到aof 文件的末尾

默认的AOF持久化策略是每秒钟fsync一次（fsync是指把缓存中的写指令记录到磁盘中），因为在这种情况下，redis仍然可以保持很好的性能，即使redis故障，也只会丢失一秒钟的数据。

如果追加日志时，恰好遇到磁盘空间满，inode满，或者断电情况等导致日志写入不完整，也没有关系，redis提供了redis-check-aof工具，用来日志修复

因为采用追加方式，如果不做任何处理的话，AOF文件会变得**越来越大**，为此，redis提供了AOF文件**重写（rewrite）机制**，即当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**。举个例子或许更形象，假如我们调用了**100次INCR指令**，在AOF文件中就要**存储100条指令**，但这明显是很低效的，完全可以把这**100条指令合并成一条SET指令，这就是重写机制的原理**。

在进行AOF重写时，仍然是采用**先写临时文件，全部完成后再替换的流程**，所以断电、磁盘满等问题都不会影响AOF文件的可用性。

AOF方式的另一个好处，我们通过一个“场景再现”来说明。某同学在操作redis时，不小心执行了**FLUSHALL**，导致redis内存中的数据**全部被清空**了，这是很悲剧的事情。不过这也不是世界末日，只要redis配置了**AOF持久化方式**，且**AOF文件还没有被重写**（rewrite），我们就可以用最快的速度**暂停redis并编辑AOF文件**，将**最后一行的FLUSHALL命令删除**，然后重启redis，就可以恢复redis的所有数据到FLUSHALL之前的状态了。是不是很神奇，这就是AOF持久化方式的好处之一。但是**如果AOF文件已经被重写了，那就无法通过这种方法来恢复数据了**。

虽然优点多多，但AOF方式也同样存在缺陷，比如在同样数据规模的情况下，**AOF文件要比RDB文件的体积大。而且，AOF方式的恢复速度也要慢于RDB方式**。

如果你直接执行**BGREWRITEAOF**命令，那么redis会生成一个**全新的AOF文件**，其中便包括了可以**恢复现有数据的最少的命令集**。

如果运气比较差，AOF文件出现了**被写坏**的情况，也不必过分担忧，redis**并不会贸然加载这个有问题的AOF文件，而是报错退出**。这时可以通过以下步骤来修复出错的文件：

1.备份被写坏的AOF文件

2.运行redis-check-aof –fix进行修复

3.用diff -u来看下两个文件的差异，确认问题点

4.重启redis，加载修复后的AOF文件

**深入AOF重写**

在重写即将开始之际，**redis会创建（fork）一个“重写子进程”**，这个子进程会首先读取**现有的AOF文件**，并将其包含的指令进行**分析压缩并写入到一个临时文件**中。

与此同时，主工作进程会将新接收到的写指令一边**累积到内存缓冲区**中，一边**继续写入到原有的AOF文件**中，这样做是保证**原有的AOF文件的可用性**，避免在**重写过程中出现意外**。

当“重写子进程”完成**重写工作**后，它会给**父进程发一个信号**，父进程收到信号后就会将**内存中缓存的写指令追加到新AOF文件中**。

当追加结束后，redis就会用新AOF文件来代替旧AOF文件，之后再有新的写指令，就都会追加到新的AOF文件中了。

**子进程读取旧的aof文件，然后分析压缩到新的aof；**

**同时主进程对新的写指令操作，既放到内存缓冲区，又放到旧的aof文件中，防止重写失败，保证旧aof文件的可用性；**

**等到子进程重写结束了，给父进程信号，会将内存缓冲区的指令追加到新的aof文件中，然后覆盖旧的aof文件。**

**4.主从用法**

redis是支持主从同步的，而且也支持一主多从以及多级从结构。

主从结构，一是为了纯粹的冗余备份，二是为了提升读性能，比如很消耗性能的SORT就可以由从服务器来承担。

**redis的主从同步是异步进行的，这意味着主从同步不会影响主逻辑，也不会降低redis的处理性能。**

主从架构中，可以考虑**关闭主服务器的数据持久化**功能，只让**从服务器进行持久化**，这样可以提高**主服务器的处理性能**。

在主从架构中，**从服务器通常被设置为只读模式**，这样可以避免从服务器的数据被误修改。但是从服务器仍然可以接受CONFIG等指令，所以还是不应该将从服务器直接暴露到不安全的网络环境中。如果必须如此，那可以考虑给**重要指令进行重命名，来避免命令被外人误执行。**

**4.1.主从同步原理**

从服务器会向主服务器**发出SYNC指令**，当主服务器接到此命令后，就会调用**BGSAVE指令**fork创建一个**子进程专门进行数据持久化工作**，也就是将主服务器的数据写入**RDB文件中**。在数据持久化期间，**主服务器将执行的写指令都缓存在内存中**。

在**BGSAVE指令执行完成后**，主服务器会将**持久化好的RDB文件发送给从服务器**，从服务器接到此**文件后会将其存储到磁盘上**，**然后再将其读取到内存中**。这个动作完成后，主服务器会将**这段时间缓存的写指令再以redis协议的格式发送给从服务器**。

**从服务器发送SYNC命令，主服务器调用BGSAVE指令，阻塞只发生在fork阶段，一般时间很短，创建一个子线程进行持久化操作，子线程将主服务器的数据写入到一个RDB文件中（文件在主服务器中），子线程持久化期间，主服务器执行的写指令将用一个缓冲区记录。在BGSAVE指令执行完成后，主服务器将持久化的RDB文件发送给从服务器，从服务器接受后将文件存储到磁盘并读入内存，将自己的数据库状态更新到主服务器执行BGSAVE时的状态；动作完成后，主服务器将持久化期间缓冲区记录的写指令发送给从服务器，从服务器执行后，将自己数据状态更新到主服务器的数据库的当前的状态。**

另外，要说的一点是，即使有多个从服务器同时发来SYNC指令，主服务器也只会执行一次BGSAVE，然后把持久化好的RDB文件发给多个从服务器。

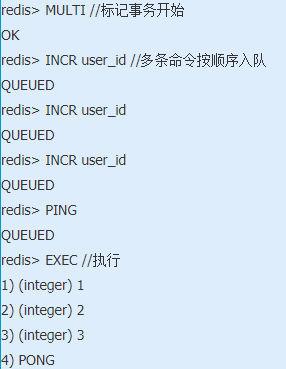
**5. redis的事务处理**

1.MULTI用来组装一个事务；标记事务的开始

2.EXEC用来执行一个事务；

3.DISCARD用来取消一个事务；

4.WATCH用来监视一些key，一旦这些key在事务执行之前被改变，则取消事务的执行。



我们看到了QUEUED的字样，这表示我们在用MULTI组装事务时，每一个命令都会进入到内存队列中缓存起来，如果出现QUEUED则表示我们这个命令成功插入了缓存队列，在将来执行EXEC时，**这些被QUEUED的命令都会被组装成一个事务来执行。**

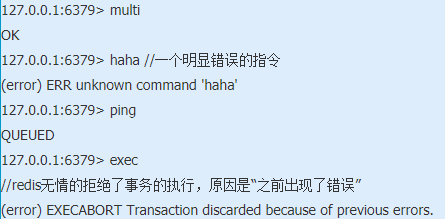
对于事务的执行来说，如果redis开启了AOF持久化的话，那么一旦事务被成功执行，事务中的命令就会通过**write命令一次性写到磁盘中去**，写到AOF文件中。如果在向磁盘中写的过程中恰好出现断电、硬件故障等问题，那么就可能出现**只有部分命令进行了AOF持久化**，这时AOF文件就会出现不完整的情况，这时，我们可以使用**redis-check-aof工具来修复这一问题**，这个工具会将**AOF文件中不完整的信息移除，确保AOF文件完整可用**。

有关事务，大家经常会遇到的是两类错误：

1.调用EXEC之前的错误

2.调用EXEC之后的错误

“调用EXEC之前的错误”，有可能是由于**语法有误导致**的，也可能时由于内存不足导致的。**只要出现某个命令无法成功写入缓冲队列的情况，redis都会进行记录**，在客户端调用EXEC时，redis会拒绝执行这一事务。我们来看一个这样的例子：

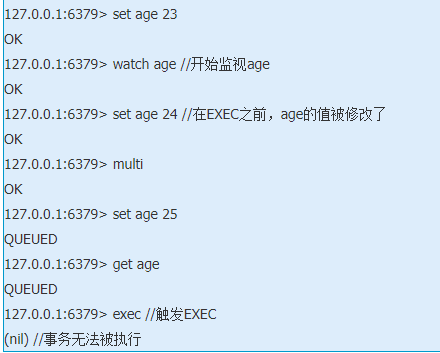


而对于“调用EXEC之后的错误”，redis则采取了完全不同的策略，即**redis不会理睬这些错误，而是继续向下执行事务中的其他命令**。这是因为，对于应用层面的错误，并不是redis自身需要考虑和处理的问题，所以一个事务中如果某一条命令执行失败，并不会影响接下来的其他命令的执行。我们也来看一个例子：

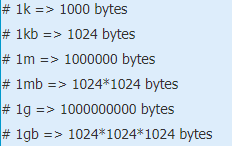


指令“WATCH”，这是一个很好用的指令，它可以帮我们实现类似于“乐观锁”的效果，即CAS（check and set）。

WATCH本身的作用是“监视key是否被改动过”，而且支持同时监视多个key，只要还没真正触发事务，WATCH都会尽职尽责的监视，一旦发现某个key被修改了，在执行EXEC时就会返回nil，表示事务无法触发。



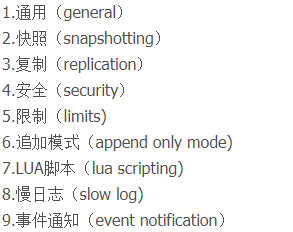
**6.Redis配置-简介**



redis配置中对单位的大小写不敏感，1GB、1Gb和1gB都是相同的。由此也说明，redis只支持bytes，不支持bit单位。

redis支持“主配置文件中引入外部配置文件”，很像C/C++中的include指令，比如：





**6.1 通用**

默认情况下，redis并不是以daemon形式来运行的。通过daemonize配置项可以控制redis的运行形式，如果改为yes，那么redis就会以daemon形式运行：



当以daemon形式运行时，redis会生成一个pid文件，默认会生成在/var/run/redis.pid。当然，你可以通过pidfile来指定pid文件生成的位置，比如：



默认情况下，redis会响应本机所有可用网卡的连接请求。当然，redis允许你通过bind配置项来指定要绑定的IP，比如：



redis的默认服务端口是6379，你可以通过port配置项来修改。如果端口设置为0的话，redis便不会监听端口了。



当一个redis-client一直没有请求发向server端，那么server端有权主动关闭这个连接，可以通过timeout来设置“空闲超时时限”，0表示永不关闭。



TCP连接保活策略，可以通过tcp-keepalive配置项来进行设置，单位为秒，假如设置为60秒，则server端会**每60秒向连接空闲的客户端发起一次ACK请求**，以检查客户端**是否已经挂掉**，对于无响应的客户端则会**关闭其连接**。所以关闭一个连接**最长需要120秒的时间**。如果设置为0，则不会进行保活检测。

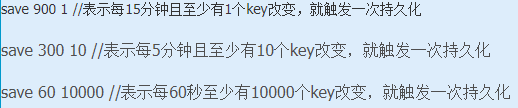


**6.2快照**

主要涉及的是**redis的RDB持久化**相关的配置，我们来一起看一看。

我们可以用如下的**指令来让数据保存到磁盘上**，即控制**RDB快照功能**：

change表示规定时间到达多少次变化，才会执行一次持久化



如果用户开启了RDB快照功能，那么在redis持久化数据到磁盘时如果出现失败，默认情况下，**redis会停止接受所有的写请求**。**这样做的好处在于可以让用户很明确的知道内存中的数据和磁盘上的数据已经存在不一致了。**如果redis不顾这种不一致，一意孤行的继续接收写请求，就可能会引起一些灾难性的后果。

**如果下一次RDB持久成功，redis会自动恢复接受写请求**

当然，如果你不在乎这种数据不一致或者有其他的手段发现和控制这种不一致的话，你完全**可以关闭这个功能**，以便在快照写入失败时，也能确保redis继续接受新的写请求。配置项如下：



**6.3 复制**

Redis提供了主从同步功能

通过slaveof配置项可以控制**某一个redis作为另一个redis的从服务器**，通过指定IP和端口来定位到主redis的位置。一般情况下，我们会建议用户为从redis设置一个不同频率的快照持久化的周期，或者为从redis配置一个不同的服务端口等等。

slaveof <masterip> <masterport>

当**从redis失去了与主redis的连接**，或者**主从同步正在进行中**时，redis该如何处理外部发来的访问请求呢？这里，从redis可以有两种选择：

第一种选择：如果slave-serve-stale-data设置为yes（默认），则从**redis仍会继续响应客户端的读写请求**。

第二种选择：如果slave-serve-stale-data设置为no，则**从redis会对客户端的请求返回“SYNC with master in progress”**，当然也有例外，当客户端发来INFO请求和SLAVEOF请求，从redis还是会进行处理。

你可以控制一个**从redis是否可以接受写请求**。将数据直接写入**从redis**，一般只适用于那些**生命周期非常短的数据**，因为**在主从同步时，这些临时数据就会被清理掉**。自从redis2.6版本之后，默认从redis为只读。

从redis会周期性的向主redis发出PING包。你可以通过repl\_ping\_slave\_period指令来控制其周期。默认是10秒。



在**主从同步**时，可能在这些情况下会有**超时发生**：

1.以从redis的角度来看，当有大规模IO传输时。

2.以从redis的角度来看，当数据传输或PING时，主redis超时

3.以主redis的角度来看，在回复从redis的PING时，从redis超时

用户可以设置上述超时的时限，不过要确保这个时限比repl-ping-slave-period的值要大，否则每次主redis都会认为从redis超时。



在主redis持续工作不正常的情况，优先级高的从redis将会升级为主redis。而编号越小，优先级越高。比如一个主redis有三个从redis，优先级编号分别为10、100、25，那么编号为10的从redis将会被首先选中升级为主redis。当优先级被设置为0时，这个从redis将永远也不会被选中。默认的优先级为100。



主redis发现有超过M个从redis的连接延时大于N秒，那么主redis就停止接受外来的写请求。这是因为从redis一般会每秒钟都向主redis发出PING，而主redis会记录每一个从redis最近一次发来PING的时间点，所以主redis能够了解每一个从redis的运行情况。



假如有大于等于3个从redis的连接延迟大于10秒，那么主redis就不再接受外部的写请求。上述两个配置中有一个被置为0，则这个特性将被关闭。默认情况下min-slaves-to-write为0，而min-slaves-max-lag为10。

**6.4 慢日志**

指一个系统进行**日志查询超过了指定的时长，就会进行慢日志的记录**。这个时长不包括IO操作，比如与客户端的交互、发送响应内容等，而仅包括**实际执行查询命令的时间**。

针对慢日志，你可以设置两个参数，一个是**执行时长**，单位是微秒，另一个是**慢日志最多存储多少条**。当一个新的命令被写入日志时，最老的一条会从命令日志队列中被移除。

单位是微秒，即1000000表示一秒。负数则会禁用慢日志功能，而0则表示强制记录每一个命令。





**7.面试redis**

**7.1 redis的特点**

Redis本质上是一个Key-Value类型的内存数据库，整个数据库统统加载在内存当中进行操作，定期通过**异步操作把数据库数据flush到硬盘上进行保存**。Redis是单进程单线程的，redis利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销。

**7.2 redis与memcached比较**



**7.3 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有哪些数据淘汰策略？？？）**

1. volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

6. no-enviction（驱逐）：禁止驱逐数据

**7.4 Redis的并发竞争问题如何解决?**

Redis的并发竞争问题，主要是发生在并发写竞争。

考虑到redis没有像db中的sql语句，update val = val + 10 where ...，无法使用这种方式进行对数据的更新。

假如有某个key = "price"， value值为10，现在想把value值进行+10操作。正常逻辑下，就是**先把数据key为price的值读回来，加上10，再把值给设置回去。**如果只有一个连接的情况下，这种方式没有问题，可以工作得很好，但如果有两个连接时，两个连接同时想对还price进行+10操作，就可能会出现问题了。

例如：两个连接同时对price进行写操作，同时加10，最终结果我们知道，应该为30才是正确。

考虑到一种情况：

T1时刻，连接1将price读出，目标设置的数据为10+10 = 20。

T2时刻，连接2也将数据读出，也是为10，目标设置为20。

T3时刻，连接1将price设置为20。

T4时刻，连接2也将price设置为20，则最终结果是一个错误值20。

如何解决?

方案一：可以使用独占锁的方式，类似操作系统的mutex机制。不过实现相对复杂，成本较高)

方案二：使用乐观锁的方式进行解决(**成本较低，非阻塞，性能较高**)

如何用乐观锁方式进行解决?

本质上是假设不会进行冲突，**使用redis的命令watch进行构造条件**。伪代码如下：

watch price

get price $price

$price = $price + 10

multi

set price $price

exec

解释一下：

watch这里表示监控该key值，**后面的事务是有条件的执行，如果从watch的exec语句执行时，watch的key对应的value值被修改了，则事务不会执行。**

同样考虑刚刚的场景，

T1时刻，连接1对price进行watch，读出price值为10，目标计算为20;

T2时刻，连接2对price进行watch，读出price值为10，目标计算为20;

T3时刻，**连接2将目标值为20写到redis中，执行事务，事务返回成功**。

T4时刻，连接1也对price进行写操作，执行事务时，由于之前已经watch了price，price在T1至T4之间已经被修改过了，所以事务执行失败。

综上，该乐观锁机制可以简单明了的解决了写冲突的问题。

又问：如果多个写操作同时过来，100个写操作同时watch，则最终只会有一个成功，99个执行失败，何解?

如果同时进行有多个请求进行写操作，例如同一时刻有100个请求过来，那么只会有一个最终成功，其余99个全部会失败，效率不高。

而且从业务层面，有些是不可接受的场景。例如：大家同时去抢一个红包，如果背后也是用乐观锁的机制去处理，那每个请求后都只有一个人成功打开红包，这对业务是不可忍受的。

在这种情况下，如果想让总体效率最大化，**可以采用排队的机制进行。**

**将所有需要对同一个key的请求进行入队操作，然后用一个消费者线程从队头依次读出请求，并对相应的key进行操作。**

这样对于**同一个key的所有请求就都是顺序访问**，正常逻辑下则不会有写失败的情况下产生。

**7.5 Redis回收进程如何工作的? Redis回收使用的是什么算法?**

六种淘汰策略，见上7.3

**7.6 Redis 大量数据插入**

Redis实例需要装载大量用户在短时间内产生的数据，数以百万计的keys需要被快速的创建。我们称之为大量数据插入(mass insertion)

如果我们需要生成一个10亿的`keyN -> ValueN’的大数据集，我们会创建一个如下的redis命令集的文件：

SET Key0 Value0

SET Key1 Value1

...

SET KeyN ValueN

一旦创建了这个文件，其余的就是让Redis尽可能快的执行

从Redis 2.6开始redis-cli支持一种新的被称之为**pipe mode**的新模式用于执行大量数据插入工作。

使用pipe mode模式的执行命令如下：

cat data.txt | redis-cli --pipe

这将产生类似如下的输出：

All data transferred. Waiting for the last reply...

Last reply received from server.

errors: 0, replies: 1000000

使用**redis-cli将有效的确保错误输出到Redis实例的标准输出里面**。

pipe mode的工作原理是什么？

redis-cli –pipe试着尽可能快的发送数据到服务器。读取数据的同时，解析它。一旦没有更多的数据输入，它就会发送一个特殊的ECHO命令，后面跟着20个随机的字符。一旦这个特殊命令发出，收到的答复就开始匹配这20个字符，当匹配时，确认数据已插入完毕

**7.7 Redis 分区的优势、不足以及分区类型**

分割数据到多个Redis实例的处理过程，因此每个实例只保存key的一个子集

**分区的优势**

通过利用多台计算机内存的和值，允许我们构造**更大的数据库**。

通过多核和多台计算机，允许我们**扩展计算能力**；通过多台计算机和网络适配器，允许**我们扩展网络带宽**。

**分区的不足**

redis的一些特性在分区方面**表现的不是很好**：

涉及**多个key的操作通常是不被支持的**。举例来说，**当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。**

**涉及多个key的redis事务不能使用**。

当使用分区时，数据处理较为复杂，比如你需要处理**多个rdb/aof文件，并且从多个实例和主机备份持久化文件**。

增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。

Redis 有两种类型分区。 假设有4个Redis实例 R0，R1，R2，R3，和类似user:1，user:2这样的表示用户的多个key，对既定的**key有多种不同方式来选择这个key存放在哪个实例中**。也就是说，有**不同的系统来映射某个key到某个Redis服务**。

**范围分区**

最简单的分区方式是按范围分区，就是映射一定范围的对象到特定的Redis实例。

**ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1**

这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的**映射表**。这个表要被管理，同时还**需要各种对象的映射表**，通常对Redis来说并非是好的方法。

**哈希分区**

**用一个hash函数将key转换为一个数字**，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。

**对这个整数取模，将其转化为0-3之间的数字**，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。

**7.8 Redis常见性能问题和解决方案:**

1. Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件
2. 如果数据比较重要，**某个Slave开启AOF备份数据，策略设置为每秒同步一次**
3. 为了主从复制的速度和连接的稳定性，**Master和Slave最好在同一个局域网内**
4. 尽量避免在**压力很大的主库上增加从库**

**7.9 redis的好处**

①速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

②支持丰富数据类型，支持string，list，set，sorted set，hash

③支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行

④丰富的特性：可用于缓存，消息，**按key设置过期时间，过期后将会自动删除**

**7.10redis的缓存失效策略和主键失效机制**

作为缓存系统都要定期清理无效数据，就需要一个主键失效和淘汰策略.

在Redis当中，有生存期的key被称为**volatile**。在创建缓存时，要为给定的key设置生存期，**当key过期的时候（生存期为0），它可能会被删除**。

1、影响生存时间的一些操作

**修改key对应的value和使用另外相同的key和value来覆盖以后，当前数据的生存时间不同。但是对key进行操作的时候生存时间不会变化。**

使用**PERSIST**命令可以在不删除 key 的情况下，**移除 key 的生存时间**，让 key 重新成为一个persistent key 。

2、如何更新生存时间

可以对一个已经带有生存时间的 key 执行**EXPIRE命令（用于指定生存时间）**，**新指定**的生存时间会取代**旧的生存时间**。TTL可以查看**key的当前生存时间**。Expire设置成功返回 1；**当 key 不存在或者不能为 key 设置生存时间时，返回 0 。**

最大缓存配置：

在 redis 中，允许用户设置最大使用内存大小，**server.maxmemory默认为0**，没有指定最大缓存，如果有新的数据添加，**超过最大内存，则会使redis崩溃**，所以一定要设置。redis 内存数据集大小上升到**一定大小的时候，就会实行数据淘汰策略**。

redis 提供 6种数据淘汰策略：

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据

注意这里的6种机制，volatile和allkeys规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的lru、ttl以及random是三种不同的淘汰策略，再加上一种no-enviction永不回收的策略。

使用策略规则：

如果数据呈现幂律分布，也就是**一部分数据访问频率高，一部分数据访问频率低**，则使用**allkeys-lru**

如果数据呈现平等分布，也就是所有的数据**访问频率都相同**，则使用**allkeys-random**

三种数据淘汰策略：

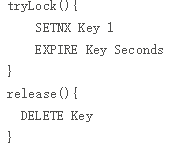
ttl和random比较容易理解，实现也会比较简单。主要是Lru最近最久未被使用的淘汰策略

**7.11使用redis如何设计分布式锁？说一下实现思路？**

**SETNX命令：key 的值设为 value ，当且仅当 key 不存在，返回1，若给定的 key 已经存在，则 SETNX 不做任何动作，返回0**

1.0版本：使用SETNX key 1和EXPIRE key seconds两条指令加锁，DELETE 进行解锁；

问题：每次提交一个redis请求，如果执行完setnx命令后，异常或者重启，锁将无法过期，引起死锁；



1.1版本：1. setnx(lockkey, 当前时间+过期超时时间) ，如果返回1，则获取锁成功；如果返回0则没有获取到锁，转向2。2. get(lockkey)获取值oldExpireTime ，并将这个value值与当前的系统时间进行比较，如果小于当前系统时间，则认为这个锁已经超时，可以允许别的请求重新获取，转向3。3. 计算newExpireTime=当前时间+过期超时时间，然后getset(lockkey, newExpireTime) 修改Value为NewExpireTime， 会返回当前lockkey的值currentExpireTime。 4. 判断currentExpireTime与oldExpireTime 是否相等，如果相等，说明当前getset设置成功，新线程获取到了锁。如果不相等，说明这个锁又被别的请求获取走了，那么当前请求可以直接返回失败，或者继续重试。

**7.12 缓存穿透，缓存击穿，缓存雪崩解决？**

缓存穿透、缓存击穿、缓存雪崩解决方案？缓存穿透：指查询一个一定不存在的数据，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到DB去查询，可能导致DB挂掉。

解决方案：1.查询返回的数据为空，仍把这个空结果进行缓存，但过期时间会比较短；2.布隆过滤器：将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对DB的查询。缓存击穿：对于设置了过期时间的key，缓存在某个时间点过期的时候，恰好这时间点对这个Key有大量的并发请求过来，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DB压垮。

解决方案：1.使用互斥锁：当缓存失效时，不立即去load db，先使用如Redis的setnx去设置一个互斥锁，当操作成功返回时再进行load db的操作并回设缓存，否则重试get缓存的方法。2.永远不过期：物理不过期，但逻辑过期（后台异步线程去刷新）。缓存雪崩：设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。与缓存击穿的区别：雪崩是很多key，击穿是某一个key缓存。

解决方案：将缓存失效时间分散开，比如可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

**7.13 缓存与数据库数据不一致**

出现数据不一致的场景：

①更新数据时，如果先更新数据库，然后删除缓存，但是缓存删除失败了，即缓存中还是旧值。

解决：先删除缓存，然后更新数据库，如果删除缓存失败，那就不要更新数据库，如果删除缓存成功，更新数据库失败，此时只是查到了旧的数据，但是能保证数据一致性

②A请求，删除缓存，更新数据库，等到下次查询，写缓存；数据库没有更新完，B请求来查询数据，发现缓存没有，就去数据库查，查到旧值，并写缓存。此时A请求更新完了数据，新值与旧值不一致。

解决：使用队列解决，对于，查询，更新请求，都放到队列中；如果有更新请求，放到队列中，更新完了再去队列里清除，如果更新过程中，遇到查询请求，先去缓存看看有没有数据，如果有，直接读缓存，如果没有去队列中查找是否有相同对象的更新请求正在处理，如果有，查询请求进队列，等待更新请求结束。

**8. redis最适合的场景**

Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存储。

Redis支持数据的备份，即master-slave模式的数据备份。

Redis支持数据的持久化，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。

1、会话缓存（Session Cache）

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于：**Redis提供持久化**。

2、全页缓存（FPC）

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。

3、队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。

4、排行榜/计数器

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。

所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE **user\_scores** 0 10 WITHSCORES（user\_scores是集合中的一个字段），使用ZREVRANK user\_scores tom;可以得到其排名的索引值，从0开始

5、发布/订阅

最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

Redis提供的所有特性中，我感觉这个是喜欢的人最少的一个，虽然它为用户提供如果此多功能。