[吃透了这些Redis知识点，面试官一定觉得你很NB](https://www.cnblogs.com/lixinjie/p/a-key-point-of-redis-in-interview.html)

是数据结构而非类型

很多文章都会说，redis支持5种常用的~~数据类型~~，这其实是存在很大的歧义。redis里存的都是二进制数据，其实就是字节数组（byte[]），这些字节数据是没有数据类型的，只有把它们按照合理的格式解码后，可以变成一个字符串，整数或对象，此时才具有数据类型。  
  
这一点必须要记住。所以任何东西只要能转化成字节数组（byte[]）的，都可以存到redis里。管你是字符串、数字、对象、图片、声音、视频、还是文件，只要变成byte数组。  
  
因此redis里的String指的并不是字符串，它其实表示的是一种最简单的数据结构，即一个key只能对应一个value。这里的key和value都是byte数组，只不过key一般是由一个字符串转换成的byte数组，value则根据实际需要而定。  
  
在特定情况下，对value也会有一些要求，比如要进行自增或自减操作，那value对应的byte数组必须要能被解码成一个数字才行，否则会报错。  
  
那么List这种数据结构，其实表示一个key可以对应多个value，且value之间是有先后顺序的，value值可以重复。  
  
Set这种数据结构，表示一个key可以对应多个value，且value之间是没有先后顺序的，value值也不可以重复。  
  
Hash这种数据结构，表示一个key可以对应多个key-value对，此时这些key-value对之间的先后顺序一般意义不大，这是一个按照名称语义来访问的数据结构，而非位置语义。  
  
Sorted Set这种数据结构，表示一个key可以对应多个value，value之间是有大小排序的，value值不可以重复。每个value都和一个浮点数相关联，该浮点数叫score。元素排序规则是：先按score排序，再按value排序。  
  
相信现在你对这5种数据结构有了更清晰的认识，那它们的对应命令对你来说就是小case了。

集群带来的问题与解决思路

集群带来的好处是显而易见的，比如容量增加、处理能力增强，还可以按需要进行动态的扩容、缩容。但同时也会引入一些新的问题，至少会有下面这两个。  
  
一是数据分配：存数据时应该放到哪个节点上，取数据时应该去哪个节点上找。二是数据移动：集群扩容，新增加节点时，该节点上的数据从何处来；集群缩容，要剔除节点时，该节点上的数据往何处去。  
  
上面这两个问题有一个共同点就是，如何去描述和存储数据与节点的映射关系。又因为数据的位置是由key决定的，所以问题就演变为如何建立起各个key和集群所有节点的关联关系。  
  
集群的节点是相对固定和少数的，虽然有增加节点和剔除节点。但集群里存储的key，则是完全随机、没有规律、不可预测、数量庞多，还非常琐碎。  
  
这就好比一所大学和它的所有学生之间的关系。如果大学和学生直接挂钩的话，一定会比较混乱。现实是它们之间又加入了好几层，首先有院系，其次有专业，再者有年级，最后还有班级。经过这四层映射之后，关系就清爽很多了。  
  
这其实是一个非常重要的结论，这个世界上没有什么问题是不能通过加入一层来解决的。如果有，那就再加入一层。计算机里也是这样的。  
  
redis在数据和节点之间又加入了一层，把这层称为槽（slot），因该槽主要和哈希有关，又叫哈希槽。  
  
最后变成了，节点上放的是槽，槽里放的是数据。槽解决的是粒度问题，相当于把粒度变大了，这样便于数据移动。哈希解决的是映射问题，使用key的哈希值来计算所在的槽，便于数据分配。  
  
可以这样来理解，你的学习桌子上堆满了书，乱的很，想找到某本书非常困难。于是你买了几个大的收纳箱，把这些书按照书名的长度放入不同的收纳箱，然后把这些收纳箱放到桌子上。  
  
这样就变成了，桌子上是收纳箱，收纳箱里是书籍。这样书籍移动很方便，搬起一个箱子就走了。寻找书籍也很方便，只要数一数书名的长度，去对应的箱子里找就行了。  
  
其实我们也没做什么，只是买了几个箱子，按照某种规则把书装入箱子。就这么简单的举动，就彻底改变了原来一盘散沙的状况。是不是有点小小的神奇呢。  
  
一个集群只能有16384个槽，编号0-16383。这些槽会分配给集群中的所有主节点，分配策略没有要求。可以指定哪些编号的槽分配给哪个主节点。集群会记录节点和槽的对应关系。  
  
接下来就需要对key求哈希值，然后对16384取余，余数是几key就落入对应的槽里。slot = CRC16(key) % 16384。  
  
以槽为单位移动数据，因为槽的数目是固定的，处理起来比较容易，这样数据移动问题就解决了。  
  
使用哈希函数计算出key的哈希值，这样就可以算出它对应的槽，然后利用集群存储的槽和节点的映射关系查询出槽所在的节点，于是数据和节点就映射起来了，这样数据分配问题就解决了。  
  
我想说的是，一般的人只会去学习各种技术，高手更在乎如何跳出技术，寻求一种解决方案或思路方向，顺着这个方向走下去，八九不离十能找到你想要的答案。

集群对命令操作的取舍

客户端只要和集群中的一个节点建立链接后，就可以获取到整个集群的所有节点信息。此外还会获取所有哈希槽和节点的对应关系信息，这些信息数据都会在客户端缓存起来，因为这些信息相当有用。  
  
客户端可以向任何节点发送请求，那么拿到一个key后到底该向哪个节点发请求呢？其实就是把集群里的那套key和节点的映射关系理论搬到客户端来就行了。  
  
所以客户端需要实现一个和集群端一样的哈希函数，先计算出key的哈希值，然后再对16384取余，这样就找到了该key对应的哈希槽，利用客户端缓存的槽和节点的对应关系信息，就可以找到该key对应的节点了。  
  
接下来发送请求就可以了。还可以把key和节点的映射关系缓存起来，下次再请求该key时，直接就拿到了它对应的节点，不用再计算一遍了。  
  
理论和现实总是有差距的，集群已经发生了变化，客户端的缓存还没来得及更新。肯定会出现拿到一个key向对应的节点发请求，其实这个key已经不在那个节点上了。此时这个节点应该怎么办？  
  
这个节点可以去key实际所在的节点上拿到数据再返回给客户端，也可以直接告诉客户端key已经不在我这里了，同时附上key现在所在的节点信息，让客户端再去请求一次，类似于HTTP的302重定向。  
  
这其实是个选择问题，也是个哲学问题。结果就是redis集群选择了后者。因此，节点只处理自己拥有的key，对于不拥有的key将返回重定向错误，即-MOVED key 127.0.0.1:6381，客户端重新向这个新节点发送请求。  
  
所以说选择是一种哲学，也是个智慧。稍后再谈这个问题。先来看看另一个情况，和这个问题有些相同点。  
  
redis有一种命令可以一次带多个key，如MGET，我把这些称为多key命令。这个多key命令的请求被发送到一个节点上，这里有一个潜在的问题，不知道大家有没有想到，就是这个命令里的多个key一定都位于那同一个节点上吗？  
  
就分为两种情况了，如果多个key不在同一个节点上，此时节点只能返回重定向错误了，但是多个key完全可能位于多个不同的节点上，此时返回的重定向错误就会非常乱，所以redis集群选择不支持此种情况。  
  
如果多个key位于同一个节点上呢，理论上是没有问题的，redis集群是否支持就和redis的版本有关系了，具体使用时自己测试一下就行了。  
  
在这个过程中我们发现了一件颇有意义的事情，就是让一组相关的key映射到同一个节点上是非常有必要的，这样可以提高效率，通过多key命令一次获取多个值。  
  
那么问题来了，如何给这些key起名字才能让他们落到同一个节点上，难不成都要先计算个哈希值，再取个余数，太麻烦了吧。当然不是这样了，redis已经帮我们想好了。  
  
可以来简单推理下，要想让两个key位于同一个节点上，它们的哈希值必须要一样。要想哈希值一样，传入哈希函数的字符串必须一样。那我们只能传进去两个一模一样的字符串了，那不就变成同一个key了，后面的会覆盖前面的数据。  
  
这里的问题是我们都是拿整个key去计算哈希值，这就导致key和参与计算哈希值的字符串耦合了，需要将它们解耦才行，就是key和参与计算哈希值的字符串有关但是又不一样。  
  
redis基于这个原理为我们提供了方案，叫做key哈希标签。先看例子，{user1000}.following，{user1000}.followers，相信你已经看出了门道，就是仅使用Key中的位于{和}间的字符串参与计算哈希值。  
  
这样可以保证哈希值相同，落到相同的节点上。但是key又是不同的，不会互相覆盖。使用哈希标签把一组相关的key关联了起来，问题就这样被轻松愉快地解决了。  
  
相信你已经发现了，要解决问题靠的是巧妙的奇思妙想，而不是非要用牛逼的技术牛逼的算法。这就是小强，小而强大。  
  
最后再来谈选择的哲学。redis的核心就是以最快的速度进行常用数据结构的key/value存取，以及围绕这些数据结构的运算。对于与核心无关的或会拖累核心的都选择弱化处理或不处理，这样做是为了保证核心的简单、快速和稳定。  
  
其实就是在广度和深度面前，redis选择了深度。所以节点不去处理自己不拥有的key，集群不去支持多key命令。这样一方面可以快速地响应客户端，另一方面可以避免在集群内部有大量的数据传输与合并。

\*\*单线程模型

redis集群的每个节点里只有一个线程负责接受和执行所有客户端发送的请求。技术上使用多路复用I/O，使用Linux的epoll函数，这样一个线程就可以管理很多socket连接。  
  
除此之外，选择单线程还有以下这些原因：  
1、redis都是对内存的操作，速度极快（10W+QPS）  
2、整体的时间主要都是消耗在了网络的传输上  
3、如果使用了多线程，则需要多线程同步，这样实现起来会变的复杂  
4、线程的加锁时间甚至都超过了对内存操作的时间  
5、多线程上下文频繁的切换需要消耗更多的CPU时间  
6、还有就是单线程天然支持原子操作，而且单线程的代码写起来更简单

\*\*事务

事务大家都知道，就是把多个操作捆绑在一起，要么都执行（成功了），要么一个也不执行（回滚了）。redis也是支持事务的，但可能和你想要的不太一样，一起来看看吧。  
  
redis的事务可以分为两步，定义事务和执行事务。使用multi命令开启一个事务，然后把要执行的所有命令都依次排上去。这就定义好了一个事务。此时使用exec命令来执行这个事务，或使用discard命令来放弃这个事务。  
  
你可能希望在你的事务开始前，你关心的key不想被别人操作，那么可以使用watch命令来监视这些key，如果开始执行前这些key被其它命令操作了则会取消事务的。也可以使用unwatch命令来取消对这些key的监视。  
  
redis事务具有以下特点：  
1、如果开始执行事务前出错，则所有命令都不执行  
2、一旦开始，则保证所有命令一次性按顺序执行完而不被打断  
3、如果执行过程中遇到错误，会继续执行下去，不会停止的  
4、对于执行过程中遇到错误，是不会进行回滚的  
看完这些，真想问一句话，你这能叫事务吗？很显然，这并不是我们通常认为的事务，因为它连原子性都保证不了。保证不了原子性是因为redis不支持回滚，不过它也给出了不支持的理由。  
  
不支持回滚的理由：  
1、redis认为，失败都是由命令使用不当造成  
2、redis这样做，是为了保持内部实现简单快速  
3、redis还认为，回滚并不能解决所有问题  
  
哈哈，这就是霸王条款，因此，好像使用redis事务的不太多

管道

客户端和集群的交互过程是串行化阻塞式的，即客户端发送了一个命令后必须等到响应回来后才能发第二个命令，这一来一回就是一个往返时间。如果你有很多的命令，都这样一个一个的来进行，会变得很慢。  
  
redis提供了一种管道技术，可以让客户端一次发送多个命令，期间不需要等待服务器端的响应，等所有的命令都发完了，再依次接收这些命令的全部响应。这就极大地节省了许多时间，提升了效率。  
  
聪明的你是不是意识到了另外一个问题，多个命令就是多个key啊，这不就是上面提到的多key操作嘛，那么问题来了，你如何保证这多个key都是同一个节点上的啊，哈哈，redis集群又放弃了对管道的支持。  
  
不过可以在客户端模拟实现，就是使用多个连接往多个节点同时发送命令，然后等待所有的节点都返回了响应，再把它们按照发送命令的顺序整理好，返回给用户代码。哎呀，好麻烦呀。

协议

简单了解下redis的协议，知道redis的数据传输格式。  
  
发送请求的协议：  
  
\*参数个数CRLF$参数1的字节数CRLF参数1的数据CRLF...$参数N的字节数CRLF参数N的数据CRLF  
  
例如，SET name lixinjie，实际发送的数据是：

\*3\r\n$3\r\nSET\r\n$4\r\nname\r\n$8\r\nlixinjie\r\n  
  
接受响应的协议：  
  
单行回复，第一个字节是+

错误消息，第一个字节是-

整型数字，第一个字节是:

批量回复，第一个字节是$

多个批量回复，第一个字节是\*  
  
例如，

+OK\r\n

-ERR Operation against\r\n

:1000\r\n

$6\r\nfoobar\r\n

\*2\r\n$3\r\nfoo\r\n$3\r\nbar\r\n  
  
可见redis的协议设计的非常简单。

<https://blog.csdn.net/Butterfly_resting/article/details/89668661>

【2】

## 单线程的redis为什么这么快

(一)纯内存操作

(二)单线程操作，避免了频繁的上下文切换

(三)采用了非阻塞I/O多路复用机制

## redis的数据类型，以及每种数据类型的使用场景

回答：一共五种

(一)String

这个其实没啥好说的，最常规的set/get操作，value可以是String也可以是数字。一般做一些复杂的计数功能的缓存。

(二)hash

这里value存放的是结构化的对象，比较方便的就是操作其中的某个字段。博主在做单点登录的时候，就是用这种数据结构存储用户信息，以cookieId作为key，设置30分钟为缓存过期时间，能很好的模拟出类似session的效果。

(三)list

使用List的数据结构，可以做简单的消息队列的功能。另外还有一个就是，可以利用lrange命令，做基于redis的分页功能，性能极佳，用户体验好。本人还用一个场景，很合适—取行情信息。就也是个生产者和消费者的场景。LIST可以很好的完成排队，先进先出的原则。

(四)set

因为set堆放的是一堆不重复值的集合。所以可以做全局去重的功能。为什么不用JVM自带的Set进行去重？因为我们的系统一般都是集群部署，使用JVM自带的Set，比较麻烦，难道为了一个做一个全局去重，再起一个公共服务，太麻烦了。

另外，就是利用交集、并集、差集等操作，可以计算共同喜好，全部的喜好，自己独有的喜好等功能。

(五)sorted set

sorted set多了一个权重参数score,集合中的元素能够按score进行排列。可以做排行榜应用，取TOP N操作。

## Redis 内部结构

dict 本质上是为了解决算法中的查找问题（Searching）是一个用于维护key和value映射关系的数据结构，与很多语言中的Map或dictionary类似。 本质上是为了解决算法中的查找问题（Searching）

sds sds就等同于char \* 它可以存储任意二进制数据，不能像C语言字符串那样以字符’\0’来标识字符串的结 束，因此它必然有个长度字段。

skiplist （跳跃表） 跳表是一种实现起来很简单，单层多指针的链表，它查找效率很高，堪比优化过的二叉平衡树，且比平衡树的实现，

quicklist

ziplist 压缩表 ziplist是一个编码后的列表，是由一系列特殊编码的连续内存块组成的顺序型数据结构，

## Redis 为什么是单线程的

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）Redis利用队列技术将并发访问变为串行访问

1）绝大部分请求是纯粹的内存操作（非常快速）

2）采用单线程,避免了不必要的上下文切换和竞争条件

3）非阻塞IO优点：

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如何解决redis的并发竞争key问题

同时有多个子系统去set一个key。这个时候要注意什么呢？ 不推荐使用redis的事务机制。

因为我们的生产环境，基本都是redis集群环境，做了数据分片操作。你一个事务中有涉及到多个key操作的时候，这多个key不一定都存储在同一个redis-server上。因此，redis的事务机制，十分鸡肋。

(1)如果对这个key操作，不要求顺序： 准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可

(2)如果对这个key操作，要求顺序： 分布式锁+时间戳。 假设这会系统B先抢到锁，将key1设置为{valueB 3:05}。接下来系统A抢到锁，发现自己的valueA的时间戳早于缓存中的时间戳，那就不做set操作了。以此类推。

(3) 利用队列，将set方法变成串行访问也可以redis遇到高并发，如果保证读写key的一致性

对redis的操作都是具有原子性的,是线程安全的操作,你不用考虑并发问题,redis内部已经帮你处理好并发的问题了。

## 讲解下Redis线程模型

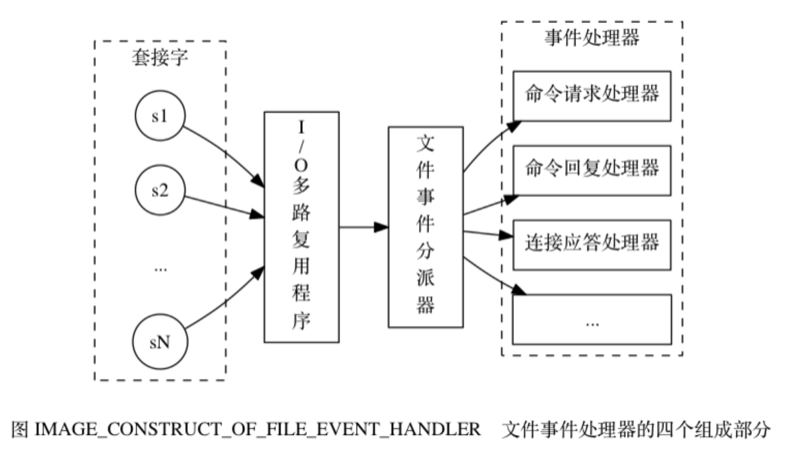
文件事件处理器包括分别是套接字、 I/O 多路复用程序、 文件事件分派器（dispatcher）、 以及事件处理器。使用 I/O 多路复用程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

I/O 多路复用程序负责监听多个套接字， 并向文件事件分派器传送那些产生了事件的套接字。

### 工作原理：

1)I/O 多路复用程序负责监听多个套接字， 并向文件事件分派器传送那些产生了事件的套接字。

尽管多个文件事件可能会并发地出现， 但 I/O 多路复用程序总是会将所有产生事件的套接字都入队到一个队列里面， 然后通过这个队列， 以有序（sequentially）、同步（synchronously）、每次一个套接字的方式向文件事件分派器传送套接字： 当上一个套接字产生的事件被处理完毕之后（该套接字为事件所关联的事件处理器执行完毕）， I/O 多路复用程序才会继续向文件事件分派器传送下一个套接字。如果一个套接字又可读又可写的话， 那么服务器将先读套接字， 后写套接字.



## 为什么Redis的操作是原子性的，怎么保证原子性的？

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。

Redis的操作之所以是原子性的，是因为Redis是单线程的。

Redis本身提供的所有API都是原子操作，Redis中的事务其实是要保证批量操作的原子性。

多个命令在并发中也是原子性的吗？

不一定， 将get和set改成单命令操作，incr 。使用Redis的事务，或者使用Redis+Lua==的方式实现.

## Redis事务

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。

注：redis的discard只是结束本次事务,正确命令造成的影响仍然存在.

1）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。

2）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 nil 。

3）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。

4）WATCH 命令可以为 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。

## Redis实现分布式锁

Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系Redis中可以使用SETNX命令实现分布式锁。

将 key 的值设为 value ，当且仅当 key 不存在。 若给定的 key 已经存在，则 SETNX 不做任何动作



解锁：使用 del key 命令就能释放锁

解决死锁：

1）通过Redis中expire()给锁设定最大持有时间，如果超过，则Redis来帮我们释放锁。

2） 使用 setnx key “当前系统时间+锁持有的时间”和getset key “当前系统时间+锁持有的时间”组合的命令就可以实现。