**Redis中有哪些数据类型？**

**一个字符串类型的值能存储最大容量是多少？**

**Redis有哪些常用命令？**

**在项目中，你们用redis做什么？**

<http://redisdoc.com/string/set.html>

## Redis数据类型

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 string类型是Redis最基本的数据类型，一个键最大能存储512MB。

## 应用场景

**string**

**常用命令**：get、set、incr、decr、mget等。

使用场景：验证码、校验码、会话缓存（Session Cache）

**hash**

**常用命令：**hget,hset,hgetall 等。

应用场景 ：适用于对象存储，存储部分变更数据，如 购物车 等。

|  |  |  |
| --- | --- | --- |
| 用户 | 购物车 | |
| UserId:946202 | 78302 | {Info:{name:””,price:78.5},num:5,subtotal:10} |
| 83272 | {Info:{name:””,price:78.5},num:5,subtotal:10} |
| … | …. |

**list**

**常用命令：**lpush,rpush,lpop,rpop,lrange等。

使用场景： 分布式队列

**set**

**常用命令：**sadd,spop,smembers,sunion 等。

使用场景： 集合运算 共同关注、共同喜好、二度好友 （微博）【社交app】

**zset**

**常用命令：**zadd,zrange,zrem,zcard等

使用场景： 排行榜 （游戏排行榜）

**总结：哪些场景下适合使用redis？**

**1、频繁访问的、一致性要求相对低，缓存数据**

2、访问频繁的、临时数据

3、使用redis本身提供的特有功能

## 淘汰策略

redis设置配置文件的maxmemory参数，可以控制其最大可用内存大小

redis 提供 6种数据淘汰策略通过maxmemory-policy设置策略：

LRU：最近最少使用

淘汰策略：

|  |  |
| --- | --- |
| 规则名称 | 规则说明 |
| volatile-lru | 在设置了过期时间的键空间中，优先移除最近未使用的key。 |
| allkeys-lru | 在主键空间中，优先移除最近未使用的key。 |
| volatile-random | 在设置了过期时间的键空间中，随机移除某个key。 |
| allkeys-random | 在主键空间中，随机移除某个key。 |
| volatile-ttl | 在设置了过期时间的键空间中，具有更早过期时间的key优先移除。 |
| noeviction | 当内存使用达到阈值的时候，所有引起申请内存的命令会报错。 |

**Redis是单进程单线程的？**

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。

redis的读取和处理性能非常强大，一般服务器的cpu都不会是性能瓶颈。redis的性能瓶颈主要集中在内存和网络方面。

## redis持久化

**什么是Redis持久化？**

**Redis有哪几种持久化方式？**

**优缺点是什么？**

**Redis提供两种持久化机制RDB和AOF机制:**

**1）RDB(Redis DataBase)持久化方式：** 是指用数据集快照的方式(半持久化模式)记录redis数据库的所有键值对,在某个时间点将数据写入一个临时文件，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复。

 优点：

1.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)

2.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。

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。

 缺点：

1.AOF文件比RDB文件大，且恢复速度慢。

2.AOF开启后支持写的QPS会比RDB支持的写的QPS低，因为AOF一般会配置成每秒fsync操作，每秒的fsync操作还是很高的（有可能阻塞主线程，直到同步完成）

**RDB的启动优先级低于AOF的启动优先级，在宕机的情况下会优先启动AOF**

**RDB和AOF到底该如何选择？**

（1）不要仅仅使用RDB这样会丢失很多数据。

（2）也不要仅仅使用AOF，因为这一会有两个问题，第一通过AOF做备份没有RDB做备份恢复的速度快；第二RDB每次简单粗暴生成数据快照，更加健壮。

（3）综合AOF和RDB两种持久化方式，用AOF来保证数据不丢失，作为恢复数据的第一选择；用RDB来做不同程度的备份，在AOF文件都丢失或损坏不可用的时候，可以使用RDB进行快速的数据恢复。

**总结：两种持久化方式对比最大的差异点？**

1、安全性问题：

2、效率性能问题：恢复速度、请求写效率

**持久化：aof和rdb原理**

**注意：原理学习请结合上面的面试题**

**RDB快照原理**

      Redis默认会将快照文件存储在Redis当前进程的工作目录的dump.rdb文件中，可以通过配置文件中的dir和dbfilename两个参数分别指定快照文件的存储路径和文件名，默认的存储路径和文件名如下图所示：

快照执行的过程如下：

（1）Redis使用fork函数复制一份当前进程（父进程）的副本（子进程）；

（2）父进程继续处理来自客户端的请求，子进程开始将内存中的数据写入硬盘中的临时文件；

（3）当子进程写完所有的数据后，用该临时文件替换旧的RDB文件，至此，一次快照操作完成。

需要注意的是：

在执行fork的时候 操作系统（类Unix操作系统）会使用写时复制（copy-on-write）策略，即fork函数发生的一刻，父进程和子进程共享同一块内存数据，当父进程需要修改其中的某片数据（如执行写命令）时，操作系统会将该片数据复制一份以保证子进程不受影响，所以RDB文件存储的是执行fork操作那一刻的内存数据。所以RDB方式理论上是会存在丢数据的情况的(fork之后修改的的那些没有写进RDB文件)。

**AOF同步硬盘数据**

     虽然每次执行更改数据库的内容时，AOF都会记录执行的命令，但是由于操作系统本身的硬盘缓存的缘故，AOF文件的内容并没有真正地写入硬盘，在默认情况下，操作系统会每隔30s将硬盘缓存中的数据同步到硬盘，但是为了防止系统异常退出而导致丢数据的情况发生，我们还可以在Redis的配置文件中配置这个同步的频率：

1 # appendfsync always 2 appendfsync everysec 3 # appendfsync no

第一行表示每次AOF写入一个命令都会执行同步操作，这是最安全也是最慢的方式；

第二行表示每秒钟进行一次同步操作，一般来说使用这种方式已经足够；

第三行表示不主动进行同步操作，这是最不安全的方式。

选项：

**1、appendfsync no**

　　当设置appendfsync为no的时候，Redis不会主动调用fsync去将AOF日志内容同步到磁盘，所以这一切就完全依赖于操作系统的调试了。对大多数Linux操作系统，是每30秒进行一次fsync，将缓冲区中的数据写到磁盘上。

**2、appendfsync everysec**

      当设置appendfsync为everysec的时候，Redis会默认每隔一秒进行一次fsync调用，将缓冲区中的数据写到磁盘。但是当这一次的fsync调用时长超过1秒时。Redis会采取延迟fsync的策略，再等一秒钟。也就是在两秒后再进行fsync，这一次的fsync就不管会执行多长时间都会进行。这时候由于在fsync时文件描述符会被阻塞，所以当前的写操作就会阻塞。所以，结论就是：在绝大多数情况下，Redis会每隔一秒进行一次fsync。在最坏的情况下，两秒钟会进行一次fsync操作。这一操作在大多数数据库系统中被称为group commit，就是组合多次写操作的数据，一次性将日志写到磁盘。

**3、appednfsync always**

      当设置appendfsync为always时，每一次写操作都会调用一次fsync，这时数据是最安全的，当然，由于每次都会执行fsync，所以其性能也会受到影响。

**建议采用 appendfsync everysec（缺省）**

**什么是缓存穿透？如何避免？**

**缓存穿透 ：** 正常的使用缓存流程大致是，数据查询先进行缓存查询，如果key不存在或者key已经过期，再对数据库进行查询，并把查询到的对象，放进缓存。如果数据库查询对象为空，则不放进缓存。一些恶意的请求会故意查询不存在的key,请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

|  |
| --- |
| User user=Redis.get(“USER\_ID”+960302601)  If(user==null){  user =userDao.findById(960302601);  if(user!=null){  redis.put(“USER\_ID”+960302601, user);  }  }  Return user; |

如何避免？

对空查询情况做缓存，设置较短的缓存时间，对应insert的key进行缓存清理

|  |
| --- |
| String value=Redis.get(“USER\_ID”+960302601)  User user=null  If(value !=null&&value.equal(“null”)){  return  }  If(value！=null){  user=Deserialize(value);//反序列化  }    If(user==null){  user =userDao.findById(960302601);  if(user!=null){  redis.put(“USER\_ID”+960302601, user，3600s);  }else{  redis.put(“USER\_ID”+960302601, “null”,5s);  }  } |

**什么是缓存雪崩？何如避免？**

**缓存雪崩**，是指在某一个时间段，缓存集中过期失效。

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，会给后端系统带来很大压力。导致系统崩溃。

场景：淘宝双11零点抢商品

如何避免？

1、服务启动时，为其加载热门数据

2、添加二级缓存，当一级缓存实失效时，可以访问二级缓存。

3、为缓存key设置不同的过期时间，避免同一时间失效，造成一时大量访问数据库

**什么是缓存击穿？何如避免？**

**缓存击穿**，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

|  |
| --- |
| // 方式一：每次请求都要加锁  synchronized(“GOOdS\_ID”+960302601){  User user=Redis.get(“GOOdS\_ID”+960302601)  If(user==null){  user =userDao.findById(960302601);  if(user!=null){  redis.put(“GOOdS\_ID”+960302601, user);  }  }  } |
|  |

|  |
| --- |
| // 方式二  User user=Redis.get(“GOOdS\_ID”+960302601)  if(user!=null){  return user;  }      synchronized(“GOOdS\_ID”+960302601){  User user=Redis.get(“GOOdS\_ID”+960302601)  If(user==null){  user =userDao.findById(960302601);  if(user!=null){  redis.put(“GOOdS\_ID”+960302601, user);  }  }  } |

|  |
| --- |
| // 方式三  User user=Redis.get(“GOOdS\_ID”+960302601)  if(user!=null){  return user;  }      while(true){  boolean bo= lock.tryLock(100ms)//申请锁，知道timeout  if(bo){  User user=Redis.get(“GOOdS\_ID”+960302601)  If(user==null){  user =userDao.findById(960302601);  if(user!=null){  redis.put(“GOOdS\_ID”+960302601, user);  return user;  }  lock.unlock();  return user;  }else{  User user=Redis.get(“GOOdS\_ID”+960302601)    if(user!=null){  return user;  }else{  continue  }  }    }//while |

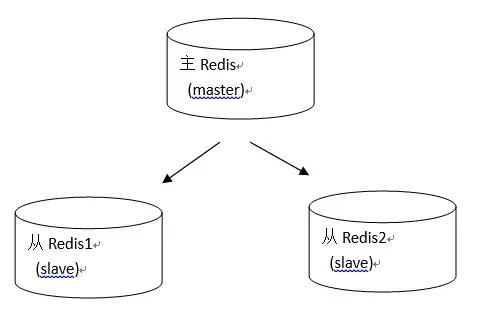
如何避免？

1：缓存失效后，可以通过添加互斥锁控制数据库查询写缓存的线程数，比如对某个key加锁控制只有一个线程可访问，让其他线程等待。

## Redis架构模式

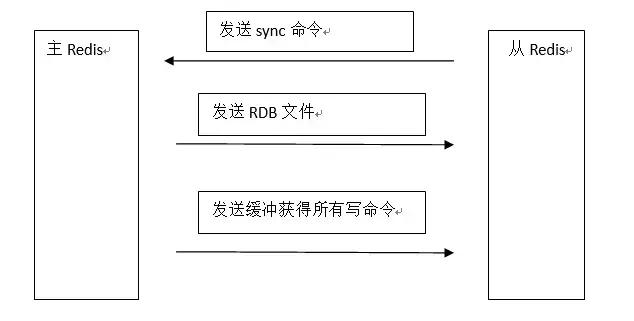
**Redis主从复制**

主从结构图



在复制的概念中，数据库分为两类，一类是主数据库（master），另一类是从数据库（slave）。主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。而从数据库一般是只读的，并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。

主从模式的配置，一般只需要再作为slave的redis节点的conf文件上加入“slaveof masterip masterport”， 或者作为slave的redis节点启动时使用如下参考命令：redis-server --port 6380 --slaveof masterIp masterPort



当一个从数据库启动后，会向主数据库发送SYNC命令。同时主数据库接收到SYNc命令后会开始在后台保存快照（即RDB持久化的过程），并将保存快照期间接收到的命令缓存起来。当快照完成后，Redis会将快照文件和所有缓存的命令发送给从数据库。从数据库收到后，会载入快照文件并执行收到的缓存的命令。以上过程称为复制初始化。复制初始化结束后，主数据库每当收到写命令时就会将命令同步给从数据库，从而保证主从数据库数据一致。

当主从数据库之间的连接断开重连后，Redis2.6以及之前的版本会重新进行复制初始化（即主数据库重新保存快照并传送给从数据库），即使从数据库可以仅有几条命令没有收到，主数据库也必须要将数据库里的所有数据重新传送给从数据库。这使得主从数据库断线重连后的数据恢复过程效率很低下，在网络环境不好的时候这一问题尤其明显。Redis 2.8版的一个重要改进就是断线重连能够支持有条件的增量数据传输，当从数据库重新连接上主数据库后，主数据库只需要将断线期间执行的命令传送给从数据库，从而大大提高Redis复制实用性。

**总结：从数据库第一次从主数据库同步数据时采用RDB全量同步，复制初始化后有新写入命令时，只会同步新写入的命令。早期版本中如果主从数据库有短连，重连时还需全量同步，现在版本中已支持断后重连增量同步。**

**乐观复制**

Redis采用了乐观复制（optimistic replication）的复制策略，容忍在一定时间内主从数据库的内容是不同的，但是两者的数据会最终同步，县体来说，Redis在主从数据库之间复制数据的过程本身是异步的，这意味着，主数据库执行完客户端请求的命令后会立即将命令在主数据库的执行结果返回给客户端，并异步地将命令同步给从数据库，而不会等待从数据库接收到该命令后再返回给客户端。这一特性保证了启用复制后主数据库的性能不会受到影响，但另一方面也会产生一个主从数据库数据不一致的时间窗口，当主数据库执行了一条写命令后，主数据库的数据已经发生的变动，然而在王数据库将该命令传送给从数据库之前，如果两个数据库之间的网络连接断开了，此时二者之间的数据就会是不一致的。从这个角度来看，主数据库是无法得知某个命令最终同步给了多少个从数据库的，不过Redis提供了两个配置选项来限制只有当数据至少同步给指定数量的从数据库时，主数据库才是可写的：

min-slaves-to-write 3

min-slaves-max-lag 10

上面的配置中，min-slaves-to-write表示只有当3个或3个以上的从数据库连接到主数据库时，主数据库才是可写的，否则会返回错误。

min-slaves-max-lag 表示允许从数据库最长失去连接的时间，如果从数据库最后与主数据库联系（即发送REPLCONFACK命令）的时间小于这个值，则认为从数据库还在保持与主数据库的连接。举个例子，按上面的配置，主数据库假设与3个从数据库相连，其中一个从数据库上一次与主数据库联系是9秒前，这时主数据库可以正常接受写入，一旦1秒过后这台从数据库依旧没有活动，则主数据库则认为目前连接的从数据库只有2个，从而拒绝写入。这一特性默认是关闭的，在分布式系统中，打开并合理配置该选项后可以降低主从架构中因为网络分区导致的数据不一致的问题。

**注意：redis主从数据库数据复制是异步的，保证了启用复制后主数据库的性能不会受到影响。同时也会产生一个主从数据库数据不一致的时间窗口。**

**增量复制**

当主从数据库连接断开后，从数据库会发送sync命令来重新进行一次完整的复制操作（全量同步），这种实现方式十分不理想，影响性能，redis2.8之后实现了主从断线重连的情况下的增量复制。

增量复制是基于如下3点实现的。

（1）从数据库会存储主数据库的运行ID（run id）。每个Redis 运行实例均会拥有一个唯一的运行ID，每当实例重启后，就会自动生成一个新的运行ID。

（2）在复制同步阶段，主数据库每将一个命令传送给从数据库时，都会同时把该命令存放到一个积压队列（backlog）中，并记录下当前积压队列中存放的命令的偏移量范围。

（3）同时，从数据库接收到主数据库传来的命令时，会记录下该命令的偏移量。

这3点是实现增量复制的基础。当主从连接准备就绪后，从数据库会发送一条SYNC命令来告诉主数据库可以开始把所有数据同步过来了。而2.8版之后，不再发送sYNC命令，取而代之的是发送PSYNC，格式为“PSYNC主数据库的运行ID断开前最新的命令偏移量”。主数据库收到PSYNC命令后，会执行以下判断来决定此次重连是否可以执行增量复制。

**是否可执行增量复制判断**

（1）首先主数据库会判断从数据库传送来的运行ID是否和自己的运行ID相同。这一步骤的意义在于确保从数据库之前确实是和自己同步的，以免从数据库拿到错误的数据。

（2）然后判断从数据库最后同步成功的命令偏移量是否在积压队列中，如果在则可以执行增量复制，并将积压队列中相应的命令发送给从数据库。

如果此次重连不满足增量复制的条件，主数据库会进行一次全部同步（即与Redis 2.6的过程相同）。

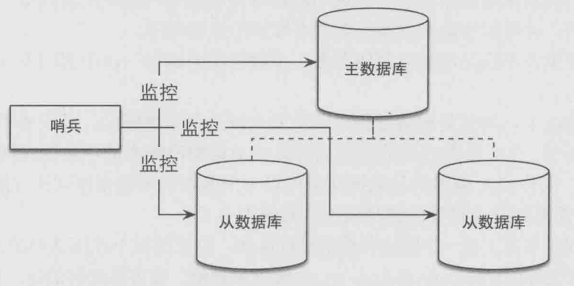
**注意：为了解决旧版本中主从数据库断线重连后需要全量数据复制的缺点，增加了增量复制功能，当在主从数据库数据差异太大时，才进行全量数据更新。**

**哨兵sentinel**

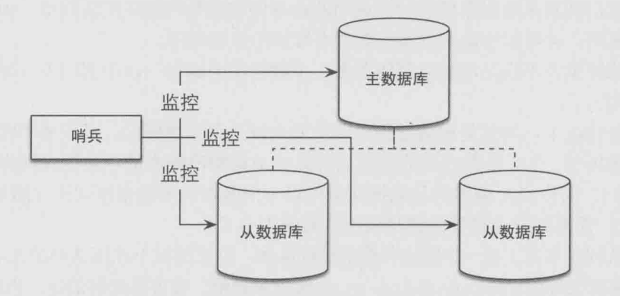
哨兵的作用就是监控Redis系统的运行状况。它的功能包括以下两个。

**（1）监控主数据库和从数据库是否正常运行。**

**（2）主数据库出现故障时自动将从数据库转换为主数据库。**



在一个一主多从的Redis系统中，可以使用多个哨兵进行监控任务以保证系统足够稳健，如图所示。注意，此时不仅哨兵会同时监控主数据库和从数据库，哨兵之间也会互相监控。



和主数据库的连接建立完成后，哨兵会定时执行下面3个操作。

（1）每10秒哨兵会向主数据库和从数据库发送INFO命令。

（2）每2秒哨兵会向主数据库和从数据库的\_sentinel:hel1o频道发送自己的息。

（3）每1秒哨兵会向主数据库、从数据库和其他哨兵节点发送PING命令。

首先，发送INFO命令使得哨兵可以获得当前数据库的相关信息（包括运行ID、复制信息等）从而实现新节点的自动发现。前面说配置哨兵监控Redis主从系统时只需要指定主数据库的信息即可，因为哨兵正是借助INFO命令来获取所有复制该主数据库的从数据库信息的。启动后，哨兵向主数据库发送INFO命令，通过解析返回结果来得知从数据库列表，而后对每个从数据库同样建立两个连接，两个连接的作用和前文介绍的与主数据库建立的两个连接完全一致。在此之后，哨兵会每10秒定时向已知的所有主从数据库发送INFO命令来获取信息更新并进行相应操作，比如对新增的从数据库建立连接并加入监控列表，对主从数据库的角色变化（由故障恢复操作引起）进行信息更新等。

当超过down-after-mi1liseconds选项指定时间后，如果被PING的数据库或节点仍然未进行回复，则哨兵认为其主观下线（subiectively down）。主观下线表示从当前的哨兵进程看来，该节点已经下线。如果该节点是主数据库，则哨兵会进一步判断是否需要对其进行故障恢复：哨兵发送SENTINELis-master-down-by-addr命令询问其他哨兵节点以了解他们是否也认为该主数据库主观下线，如果达到指定数量时，哨兵会认为其客观下线（objectively down），并选举领头的哨兵节点对主从系统发起故障恢复。这个指定数量即为前文介绍的quorum参数。例如，下面的配置：

sentinel monitor mymaster 127.0.0.163792

虽然当前哨兵节点发现了主数据库客观下线，需要故障恢复，但是故障恢复需要由领头的哨兵来完成，这样可以保证同一时间只有一个哨兵节点来执行故障恢复。选举领头哨兵的过程使用了Raft算法，具体过程如下。

（1）发现主数据库客观下线的哨兵节点（下面称作A）向每个哨兵节点发送命令，要求对方选自己成为领头哨兵。

（2）如果目标哨兵节点没有选过其他人，则会同意将A设置成领头哨兵。

（3）如果A发现有超过半数且超过quorum参数值的哨兵节点同意选自己成为领头哨兵，则A成功成为领头哨兵。

（4）当有多个哨兵节点同时参选领头哨兵，则会出现没有任何节点当选的可能。此时每个参选节点将等待一个随机时间重新发起参选请求，进行下一轮选举，直到选举成功。

**集群原理**

**集群介绍**

**即使使用哨兵，此时的Redis集群的每个数据库依然存有集群中的所有数据，从而导致集群的总数据存储量受限于可用存储内存最小的数据库节点，形成木桶效应**。由于Redis中的所有数据都是基于内存存储，这一问题就尤为突出了，尤其是当使用Redis做持久化存储服务使用时。

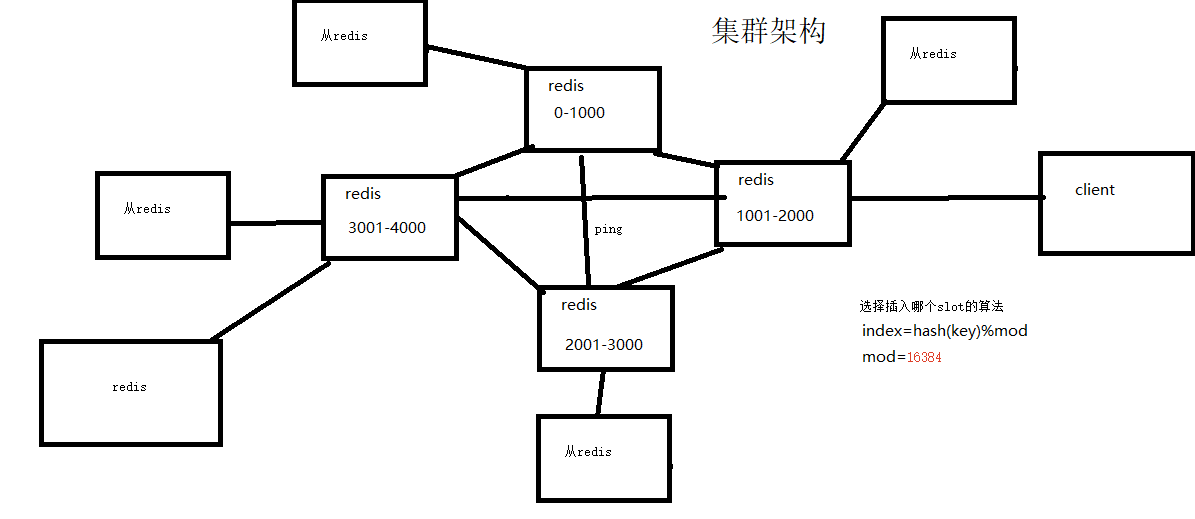
对Redis进行水平扩容，在旧版Redis中通常使用客户端分片来解决这个问题，即启动多个Redis数据库节点，由客户端决定每个键交由哪个数据库节点存储，下次客户端读取该键时直接到该节点读取。这样可以实现将整个数据分布存储在N个数据库节点中，每个节点只存放总数据量的1/N。但对于需要扩容的场景来说，在客户端分片后，如果想增加更多的节点，就需要对数据进行手工迁移，同时在迁移的过程中为了保证数据的一致性，还需要将集群暂时下线，相对比较复杂。

考虑到Redis 实例非常轻量的特点，可以采用预分片技术（presharding）来在一定程度上避免此问题，具体来说是在节点部署初期，就提前考虑日后的存储规模，建立足够多的实例（如128个节点），初期时数据很少，所以每个节点存储的数据也非常少，但由于节点轻量的特性，数据之外的内存开销并不大，这使得只需要很少的服务器即可运行这些实例。日后存储规模扩大后，所要做的不过是将某些实例迁移到其他服务器上，而不需要对所有数据进行重新分片并进行集群下线和数据迁移了。

无论如何，客户端分片终归是有非常多的缺点，比如维护成本高，增加、移除节点较繁琐等。Redis3.0版的一大特性就是支持集群（Cluster）功能。集群的特点在于拥有和单机实例同样的性能，同时在网络分区后能够提供一定的可访问性以及对主数据库故障恢复的支持。另外集群支持几乎所有的单机实例支持的命令，对于涉及多键的命令（如MGET），如果每个键都位于同一个节点中，则可以正常支持，否则会提示错误。除此之外集群还有一个限制是只能使用默认的0号数据库，如果执行SELECT切换数据库则会提示错误。

哨兵与集群是两个独立的功能，但从特性来看哨兵可以视为集群的子集，当不需要数据分片或者已经在客户端进行分片的场景下哨兵就足够使用了，但如果需要进行水平扩容，则集群是一个非常好的选择。

**注意：主从哨兵模式每个redis实例存储的是相同数据,无法进行水平扩展，即无法增大整个redis的存储容量。而集群模式下，部分节点存储部分redis分片数据，可以通过增加节点来扩容redis存储能力。**



**新增节点**

新的节点加入集群后有两种选择，要么使用CLUSTER REPLICATE命令复制每个主数据库来以从数据库的形式运行，要么向集群申请分配插槽（slot）来以主数据库的形式运行。

在一个集群中，所有的键会被分配给16384个插槽，而每个主数据库会负责处理其中的一部分插槽。

M:d4f906940d68714db787a60837f57fa496de5d12 127.0.0.1：6380

slots：0-5460（5461 slots）master M:b547d05c9d0e188993befec4ae5ccb430343fb4b 127.0.0.1：6381

slots：5461-10922（5462 slots）master M：887fe91bf218f203194403807e0aee941e985286 127.0.0.1：6382

slots：10923-16383（5461 slots）master

上面的每一行表示一个主数据库的信息，其中可以看到6380负责处理0~5460这5461个插槽，6381负责处理5461~10922这5462个插槽，6382则负责处理10923~16383这5461个插槽。虽然redis-trib.rb初始化集群时分配给每个节点的插槽都是连续的，但是实际上Redis并没有此限制，可以将任意的几个插槽分配给任意的节点负责。

**故障恢复**

在一个集群中，每个节点都会定期向其他节点发送PING命令，并通过有没有收到回复来判断目标节点是否已经下线了。具体来说，集群中的每个节点每隔1秒钟就会随机选择5个节点，然后选择其中最久没有响应的节点发送PING命令。

如果一定时间内目标节点没有响应回复，则发起PING命令的节点会认为目标节点疑似下线（PFAIL）。疑似下线可以与哨兵的主观下线类比，两者都表示某一节点从自身的角度认为目标节点是下线的状态。与哨兵的模式类似，如果要使在整个集群中的所有节点都认为某一节点已经下线，需要一定数量的节点都认为该节点疑似下线才可以，这一过程具体为：

（1）一旦节点A认为节点B是疑似下线状态，就会在集群中传播该消息，所有其他节点收到消息后都会记录下这一信息；

（2）当集群中的某一节点C收集到半数以上的节点认为B是疑似下线的状态时，就会将B标记为下线（FAIL），并且向集群中的其他节点传播该消息，从而使得B在整个集群中下线。

在集群中，当一个主数据库下线时，就会出现一部分插槽无法写入的问题。这时如果该主数据库拥有至少一个从数据库，集群就进行故障恢复操作来将其中一个从数据库转变成主数据库来保证集群的完整。选择哪个从数据库来作为主数据库的过程与在哨兵中选择领头哨兵的过程一样，都是基于Raft算法，过程如下。

（1）发现其复制的主数据库下线的从数据库（下面称作A）向每个集群中的节点发送请求，要求对方选自己成为主数据库。

（2）如果收到请求的节点没有选过其他人，则会同意将A设置成主数据库。

（3）如果A发现有超过集群中节点总数一半的节点同意选自己成为主数据库，则A则成功成为主数据库。

（4）当有多个从数据库节点同时参选主数据库，则会出现没有任何节点当选的可能。此时每个参选节点将等待一个随机时间重新发起参选请求，进行下一轮选举，直到选举成功。

当某个从数据库当选为主数据库后，会通过命请求★更多程义。转换成主数据库，并将旧的主数据库的插槽转换给自己负责。/aing qio/

如果一个至少负责一个插槽的主数据库下线且[request进行故障恢复，则整个集群默认会进入下线状态无法继续工作。为了让集群仍能正常工作，可以修改配置cluster-require-ful1-co ask

5）：

cluster-require-full-coverage no

**生产环境中的 redis 是怎么部署的？**

我们公司redis存储数据还不是很大（大概10G左右），我们采用主从加哨兵模式保证redis服务的高可用。整个集群1个主redis，1个从redis，3个哨兵（至少要3个哨兵才能保证高可用，否则无法选举leader，无法实现主从切换）。可允许一台哨兵或redis实例挂掉。每台机器配置：16G 内存+ 4 核 CPU + 1T 磁盘。

当使用哨兵模式的时候 客户端就不需要直接连接redis，而是连接哨兵的ip和端口，由sentinel来提供具体的可提供服务的Redis实现，这样当master节点挂掉以后，sentinel就会感知并将新的master节点提供给使用者。

主从容灾部署，参考博文：<https://www.cnblogs.com/kevingrace/p/9004460.html>