***Redis的基本存储类型 事务 和使用场景***

Redis最为常用的数据类型主要有以下：

String

Hash

List

Set

Sorted set

pub/sub

Transactions

***Redis读取数据哪些情况会出现超时***

(1). 网络原因：比如是否存在跨机房、网络割接等等。

(2). 慢查询，因为redis是单线程，如果有慢查询的话，会阻塞住之后的操作。

(3). value值过大？比如value几十兆，当然这种情况比较少，其实也可以看做是慢查询的一种

(4). aof重写/rdb fork发生？瞬间会堵一下Redis服务器。

(5). 其他..................

***Redis集群（主从复制怎么触发，数据如何分配）？***

在Redis中，用户可以通过执行SLAVEOF命令或者设置slaveof选项，让一个服务器去复制（replicate）另一个服务器，我们称呼被复制的服务器为主服务器（master），而对主服务器进行复制的服务器则被称为从服务器（slave），如图所示。

假设现在有两个Redis服务器，地址分别为127.0.0.1:6379和127.0.0.1:12345，如果我们向服务器127.0.0.1:12345发送以下命令：

127.0.0.1:12345> SLAVEOF 127.0.0.1 6379

OK

那么服务器127.0.0.1:12345将成为127.0.0.1:6379的从服务器，而服务器127.0.0.1:6379则会成为127.0.0.1:12345的主服务器。

（记得去http://redisdoc.com/topic/replication.html上将一些操作进行补充）

本文是按照《Redis设计与实现》一书所整理的，感觉原书讲的非常棒，所以下面的这部分的知识将按照原书的逻辑进行介绍：

先介绍旧版复制功能在处理断线后重新连接的从服务器时，会遇上怎样的低效情况。新版复制功能是如何通过部分重同步来解决旧版复制功能的低效问题的，并说明部分重同步的实现原理。

**旧版复制功能的实现**

Redis的复制功能分为同步（sync）和命令传播（command propagate）两个操作：

* 同步操作用于将从服务器的数据库状态更新至主服务器当前所处的数据库状态；
* 命令传播操作则用于在主服务器的数据库状态被修改，导致主从服务器的数据库状态出现不一致时，让主从服务器的数据库重新回到一致状态。

**同步**

当客户端向从服务器发送SLAVEOF命令，要求从服务器复制主服务器时，从服务器首先需要执行同步操作，也即是，将从服务器的数据库状态更新至主服务器当前所处的数据库状态。

从服务器对主服务器的同步操作需要通过向主服务器发送SYNC命令来完成，以下是SYNC命令的执行步骤：

1. 从服务器向主服务器发送SYNC命令；
2. 收到SYNC命令的主服务器执行BGSAVE命令，在后台生成一个RDB文件，并使用一个缓冲区记录从现在开始执行的所有写命令；
3. 当主服务器的BGSAVE命令执行完毕时，主服务器会将BGSAVE命令生成的RDB文件发送给从服务器，从服务器接收并载入这个RDB文件，将自己的数据库状态更新至主服务器**执行BGSAVE命令时的数据库状态**。
4. 主服务器将记录在缓冲区里面的所有写命令发送给从服务器，**从服务器执行这些写命令**，将自己的数据库状态更新至主服务器数据库当前所处的状态。

**命令传播**

在执行完同步操作之后，主从服务器之间数据库状态已经相同了。但这个状态并非一成不变，如果主服务器执行了写操作，那么主服务器的数据库状态就会修改，并导致主从服务器状态不再一致。

所以为了让主从服务器再次回到一致状态，主服务器需要对从服务器执行命令传播操作：主服务器会将自己执行的写命令，也即是造成主从服务器不一致的那条写命令，发送给从服务器执行，当从服务器执行了相同的写命令之后，主从服务器将再次回到一致状态。

**旧版复制功能的缺陷**

在Redis中，从服务器对主服务器的复制可以分为以下两种情况：

* 初次复制：从服务器以前没有复制过任何主服务器，或者从服务器当前要复制的主服务器和上一次复制的主服务器不同；
* 断线后重复制：处于命令传播阶段的主从服务器因为网络原因而中断了复制，但从服务器通过自动重连接重新连上了主服务器，并继续复制主服务器。

对于初次复制来说，旧版复制功能能够很好地完成任务，但对于断线后重复制来说，旧版复制功能虽然也能让主从服务器重新回到一致状态，但效率却非常低。

我们给出一个例子进行说明：

从服务器终于重新连接上主服务器，因为这时主从服务器的状态已经不再一致，所以从服务器将向主服务器发送SYNC命令，而主服务器会将包含键k1至键k10089的RDB文件发送给从服务器，从服务器通过接收和载入这个RDB文件来将自己的数据库更新至主服务器数据库当前所处的状态。

上面给出的例子可能有一点理想化，因为在主从服务器断线期间，主服务器执行的写命令可能会有成百上千个之多，而不仅仅是两三个写命令。但总的来说，主从服务器断开的时间越短，主服务器在断线期间执行的写命令就越少，而执行少量写命令所产生的数据量通常比整个数据库的数据量要少得多，在这种情况下，为了让从服务器补足一小部分缺失的数据，却要让主从服务器重新执行一次SYNC命令，这种做法无疑是非常低效的。

**SYNC命令是一个非常耗费资源的操作**

SYNC命令是非常消耗资源的，因为每次执行SYNC命令，主从服务器需要执行一下操作：

1. 主服务器需要执行BGSAVE命令来生成RDB文件，这个生成操作会耗费主服务器大量的CPU、内存和磁盘I/O资源；
2. 主服务器需要将自己生成的RDB文件发送给从服务器，这个发送操作会耗费主从服务器大量的网络资源（带宽和流量），并对主服务器响应命令请求的时间产生影响；
3. 接收到RDB文件的从服务器需要载入主服务器发来的RDB文件，并且在载入期间，从**服务器会因为阻塞而没办法处理命令请求**。

SYNC是一个如此消耗资源的命令，所以Redis最好在真需要的时候才需要执行SYNC命令。

**新版复制功能的实现**

为了解决旧版复制功能在处理断线重复制情况时的低效问题，Redis从2.8版本开始，使用**PSYNC**命令代替SYNC命令来执行复制时的同步操作。

PSYNC命令具有完整重同步（full resynchronization）和部分重同步（partial resynchronization）两种模式：

* 其中完整重同步用于处理初次复制情况：完整重同步的执行步骤和SYNC命令的执行步骤基本一样，它们都是通过让主服务器创建并发送RDB文件，以及向从服务器发送保存在缓冲区里面的写命令来进行同步；
* 而部分重同步则用于处理断线后重复制情况：当从服务器在断线后重新连接主服务器时，**如果条件允许，主服务器可以将主从服务器连接断开期间执行的写命令发送给从服务器，从服务器只要接收并执行这些写命令，就可以将数据库更新至主服务器当前所处的状态。**

其实看到这里的时候心里还是有一个疑问的：如果上面的例子是T3时候从服务器掉线，然后在T10093的时候才连接上或者更长的时间呢！！！你这样一条指令一条指令地传输过去还不如直接来一个SYNC命令快一些。所以在我看来使用PSYNC进行操作时，什么时候部分重同步，什么时候全部重同步是一个策略问题。当然Redis会解决这个问题，所以大家继续看0\_0

**部分重同步的实现**

部分重同步功能由以下三个部分构成：

* 主服务器的**复制偏移量**（replication offset）和从服务器的复制偏移量；
* **主服务器的复制积压缓冲区（replication backlog）**；
* **服务器的运行ID**（run ID）。

**复制偏移量**

执行复制的双方——主服务器和从服务器会分别维护一个复制偏移量：

* 主服务器每次向从服务器传播N个字节的数据时，就将自己的复制偏移量的值加上N；
* 从服务器每次收到主服务器传播来的N个字节的数据时，就将自己的复制偏移量的值加上N；

（我靠！！难道从服务器没有反馈吗？丢包了怎么办？难道是用TCP？大家继续看，我只是想穿插一些我的思路）

通过对比主从服务器的复制偏移量，程序可以很容易地知道主从服务器是否处于一致状态：

* 如果主从服务器处于一致状态，那么主从服务器两者的偏移量总是相同的；
* 相反，如果主从服务器两者的偏移量并不相同，那么说明主从服务器并未处于一致状态。

假设从服务器A在断线之后就立即重新连接主服务器，并且成功，那么接下来，**从服务器将向主服务器发送PSYNC命令，报告从服务器A当前的复制偏移量为10086，**那么这时，主服务器应该对从服务器执行完整重同步还是部分重同步呢？如果执行部分重同步的话，主服务器又如何补偿从服务器A在断线期间丢失的那部分数据呢？以上问题的答案都和复制积压缓冲区有关。

**复制积压缓冲区**

复制积压缓冲区是由主服务器维护的一个固定长度（fixed-size）先进先出（FIFO）队列，默认大小为1MB。

和普通先进先出队列随着元素的增加和减少而动态调整长度不同，固定长度先进先出队列的长度是固定的，当入队元素的数量大于队列长度时，最先入队的元素会被弹出，而新元素会被放入队列。

当主服务器进行命令传播时，它不仅会将写命令发送给所有从服务器，还会将写命令入队到复制积压缓冲区里面。

因此，主服务器的复制积压缓冲区里面会保存着一部分最近传播的写命令，并且复制积压缓冲区会为队列中的每个字节记录相应的复制偏移量，就像下表所示的那样。

当从服务器重新连上主服务器时，从服务器会通过PSYNC命令将自己的复制偏移量offset发送给主服务器，主服务器会根据这个复制偏移量来决定对从服务器执行何种同步操作：

* 如果offset偏移量之后的数据（也即是偏移量offset+1开始的数据）仍然存在于复制积压缓冲区里面，那么主服务器将对从服务器执行部分重同步操作；
* 相反，如果offset偏移量之后的数据已经不存在于复制积压缓冲区，那么主服务器将对从服务器执行完整重同步操作。

**根据需要调整复制积压缓冲区的大小**

Redis为复制积压缓冲区设置的默认大小为1MB，如果主服务器需要执行大量写命令，又或者主从服务器断线后重连接所需的时间比较长，那么这个大小也许并不合适。如果复制积压缓冲区的大小设置得不恰当，那么PSYNC命令的复制重同步模式就不能正常发挥作用，因此，正确估算和设置复制积压缓冲区的大小非常重要。

复制积压缓冲区的最小大小可以根据公式second\*write\_size\_per\_second来估算：

* 其中second为从服务器断线后重新连接上主服务器所需的平均时间（以秒计算）；
* 而write\_size\_per\_second则是主服务器平均每秒产生的写命令数据量（协议格式的写命令的长度总和）；

例如，如果主服务器平均每秒产生1 MB的写数据，而从服务器断线之后平均要5秒才能重新连接上主服务器，那么复制积压缓冲区的大小就不能低于5MB。

为了安全起见，可以将复**制积压缓冲区的大小设为2\*second\*write\_size\_per\_second**，这样可以保证绝大部分断线情况都能用部分重同步来处理。

至于复制积压缓冲区大小的修改方法，可以参考配置文件中关于**repl-backlog-size**选项的说明。

**服务器运行ID**

除了复制偏移量和复制积压缓冲区之外，实现部分重同步还需要用到服务器运行ID（run ID）：

* 每个Redis服务器，**不论主服务器还是从服务，都会有自己的运行ID**；
* 运行ID在服务器启动时自动生成，由40个随机的十六进制字符组成，例如53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3；

当从服务器对主服务器进行初次复制时，主服务器会将自己的运行ID传送给从服务器，而从服务器则会将这个运行ID保存起来（注意哦，是**从服务器保存了主服务器的ID**）。

当从服务器断线并重新连上一个主服务器时，从服务器将向当前连接的主服务器发送之前保存的运行ID：

* 如果从服务器保存的运行ID和当前连接的主服务器的运行ID相同，那么说明从服务器断线之前复制的就是当前连接的这个主服务器，主服务器可以继续尝试执行部分重同步操作；
* 相反地，如果从服务器保存的运行ID和当前连接的主服务器的运行ID并不相同，那么说明从服务器断线之前复制的主服务器并不是当前连接的这个主服务器，主服务器将对从服务器执行完整重同步操作。

**PSYNC命令的实现**

PSYNC命令的调用方法有两种：

* 如果从服务器以前没有复制过任何主服务器，或者之前执行过SLAVEOF no one命令，那么从服务器在开始一次新的复制时将向主服务器发送PSYNC ? -1命令，主动请求主服务器进行完整重同步（因为这时不可能执行部分重同步）；
* 相反地，如果从服务器已经复制过某个主服务器，那么从服务器在开始一次新的复制时将向主服务器发送PSYNC <runid> <offset>命令：其中runid是上一次复制的主服务器的运行ID，而offset则是从服务器当前的复制偏移量，接收到这个命令的主服务器会通过这两个参数来判断应该对从服务器执行哪种同步操作。

根据情况，接收到PSYNC命令的主服务器会向从服务器返回以下三种回复的其中一种：

* 如果主服务器返回+FULLRESYNC <runid> <offset>回复，那么表示主服务器将与从服务器执行完整重同步操作：其中runid是这个主服务器的运行ID，从服务器会将这个ID保存起来，在下一次发送PSYNC命令时使用；而offset则是主服务器当前的复制偏移量，从服务器会将这个值作为自己的初始化偏移量；
* 如果主服务器返回+CONTINUE回复，那么表示主服务器将与从服务器执行部分重同步操作，从服务器只要等着主服务器将自己缺少的那部分数据发送过来就可以了；
* 如果主服务器返回-ERR回复，那么表示主服务器的版本低于Redis 2.8，它识别不了PSYNC命令，从服务器将向主服务器发送SYNC命令，并与主服务器执行完整同步操作。

**复制的实现**

**步骤1：设置主服务器的地址和端口**

当客户端向从服务器发送以下命令时：

127.0.0.1:12345> SLAVEOF 127.0.0.1 6379

OK

从服务器首先要做的就是将客户端给定的主服务器IP地址127.0.0.1以及端口6379保存到服务器状态的masterhost属性和masterport属性里面：

struct redisServer {

    // ...

    // 主服务器的地址

    char \*masterhost;

    // 主服务器的端口

    int masterport;

    // ...

};

SLAVEOF命令是一个异步命令，在完成masterhost属性和masterport属性的设置工作之后，**从服务器将向发送SLAVEOF命令的客户端返回OK，表示复制指令已经被接收，而实际的复制工作将在OK返回之后才真正开始执行。**

**步骤2：建立套接字连接**

在SLAVEOF命令执行之后，从服务器将根据命令所设置的IP地址和端口，创建连向主服务器的套接字连接，如图15-14所示。

如果从服务器创建的套接字能成功连接（connect）到主服务器，那么从服务器将为这个套接字关联一个专门用于处理复制工作的文件事件处理器，这个处理器将负责执行后续的复制工作，比如接收RDB文件，以及接收主服务器传播来的写命令，诸如此类。

而主服务器在接受（accept）从服务器的套接字连接之后，将为该套接字创建相应的客户端状态，并将从服务器看作是一个连接到主服务器的客户端来对待，这时从服务器将同时具有服务器（server）和客户端（client）两个身份：从服务器可以向主服务器发送命令请求，而主服务器则会向从服务器返回命令回复。

**步骤3：发送PING命令**

从服务器成为主服务器的客户端之后，做的第一件事就是向主服务器发送一个PING命令。

这个PING命令主要是为了：

* 通过发送PING命令检查套接字的读写状态；
* 通过PING命令可以检查主服务器能否正常处理命令。

从服务器在发送PING命令之后可能遇到以下三种情况：

* 主服务器向从服务器返回了一个命令回复，但从服务器却不能在规定的时限内读取命令回复的内容(timeout)，说明网络连接状态不佳，从服务器将断开并重新创建连向主服务器的套接字；
* 如果主服务器返回一个错误，那么表示主服务器暂时没有办法处理从服务器的命令请求，，从服务器也将断开并重新创建连向主服务器的套接字；
* 如果从服务器读取到"PONG"回复，那么表示主从服务器之间的网络连接状态正常，那就继续执行下面的复制步骤。

**步骤4：身份验证**

从服务器在收到主服务器返回的"PONG"回复之后，下一步要做的就是决定是否进行身份验证：

* 如果从服务器**设置了masterauth选项**，那么进行身份验证。否则不进行身份认证；

在需要进行身份验证的情况下，从服务器将向主服务器发送一条**AUTH命令，命令的参数为从服务器masterauth选项的值**。

从服务器在身份验证阶段可能遇到的情况有以下几种：

* 主服务器没有设置requirepass选项，从服务器没有设置masterauth,那么就继续后面的复制工作；
* 如果从服务器的通过AUTH命令发送的密码和主服务器requirepass选项所设置的密码相同，那么也继续后面的工作，否则返回错误invaild password;
* 如果主服务器设置了requireoass选项，但从服务器没有设置masterauth选项，那么服务器将返回NOAUTH错误。反过来如果主服务器没有设置requirepass选项，但是从服务器却设置了materauth选项，那么主服务器返回no password is set错误；

所有错误到只有一个结果：中止目前的复制工作，并从创建套接字开始重新执行复制，直到身份验证通过，或者从服务器放弃执行复制为止。

**步骤5：发送端口信息**

身份验证步骤之后，从服务器将执行命令**REPLCONF listening-port <port-number>**，向主服务器发送从服务器的监听端口号。

主服务器在接收到这个命令之后，会将端口号记录在从服务器所对应的客户端状态的slave\_listening\_port属性中：

typedef struct redisClient {

    // ...

    // 从服务器的监听端口号

    int **slave\_listening\_port**;

    // ...

｝redisClient;

slave\_listening\_port属性目前唯一的作用就是在主服务器执行INFO replication命令时打印出从服务器的端口号。

**步骤6：同步**

在这一步，从服务器将向主服务器发送PSYNC命令，执行同步操作，并将自己的数据库更新至主服务器数据库当前所处的状态。

需要注意的是在执行同步操作前，只有从服务器是主服务器的客户端。但是执行从不操作之后，主服务器也会称为从服务器的客户端：

* 如果PSYNC命令执行的是完整同步操作，那么主服务器只有成为了从服务器的客户端才能将保存在**缓冲区中的写命令**发送给从服务器执行；
* 如果PSYNC命令执行的是部分同步操作，那么主服务器只有成为了从服务器的客户端才能将保存在**复制积压缓冲区中的写命令**发送给从服务器执行；

**步骤7：命令传播**

当完成了同步之后，主从服务器就会进入命令传播阶段，这时主服务器只要一直将自己执行的写命令发送给从服务器，而从服务器只要一直接收并执行主服务器发来的写命令，就可以保证主从服务器一直保持一致了。

**心跳检测**

在命令传播阶段，从服务器默认会以**每秒一次的频率**，向主服务器发送命令：**REPLCONF ACK <replication\_offset>**

其中replication\_offset是从服务器当前的复制偏移量。

发送REPLCONF ACK命令对于主从服务器有三个作用：

* 检测主从服务器的网络连接状态；
* 辅助实现min-slaves选项；
* 检测命令丢失。

**检测主从服务器的网络连接状态**

如果主服务器超过一秒钟没有收到从服务器发来的REPLCONF ACK命令，那么主服务器就知道主从服务器之间的连接出现问题了。

通过向主服务器发送**INFO replication命令**，在列出的从服务器列表的lag一栏中，我们可以看到相应从服务器最后一次向主服务器发送REPLCONF ACK命令距离现在过了多少秒：

127.0.0.1:6379> INFO replication

# Replication

role:master

connected\_slaves:2

slave0:ip=127.0.0.1,port=12345,state=online,offset=211,**lag=0**

#刚刚发送过 REPLCONF ACK命令

slave1:ip=127.0.0.1,port=56789,state=online,offset=197,**lag=15**

#15秒之前发送过REPLCONF ACK命令

*master\_repl\_offset:211*

*repl\_backlog\_active:1*

*repl\_backlog\_size:1048576*

*repl\_backlog\_first\_byte\_offset:2*

*repl\_backlog\_histlen:210*

在一般情况下，**lag的值应该在0秒或者1秒之间跳动，**如果超过1秒的话，那么说明主从服务器之间的连接出现了故障。

**辅助实现min-slaves配置选项**

Redis的**min-slaves-to-write**和**min-slaves-max-lag**两个选项可以**防止主服务器在不安全的情况下执行写命令。**

举个例子，如果我们向主服务器提供以下设置：

min-slaves-to-write 3

min-slaves-max-lag 10

那么在**从服务器的数量少于3个，或者三个从服务器的延迟（lag）值都大于或等于10秒时**，主服务器将拒绝执行写命令，这里的延迟值就是上面提到的INFO replication命令的lag值。

**检测命令丢失**

我们从命令：REPLCONF ACK <replication\_offset>就可以知道，每发送一次这个命令从服务器都会向主服务器报告一次自己的复制偏移量。那此时尽管主服务器发送给从服务器的SET key value丢失了。也无所谓，主服务器马上就知道了。

***Redis持久化 Snapshot和AOF说明***

redis是一个支持持久化的内存数据库，也就是说redis需要经常将内存中的数据同步到磁盘来保证持久化。redis支持两种持久化方式，一种是 Snapshot（RDB）<二进制文件> 也是默认方式，另一种是Append only file（AOF）的方式。

我们应该明确持久化的数据有什么用，答案是用于重启后的数据恢复。Redis是一个内存数据库，无论是RDB还是AOF，都只是其保证数据恢复的措施。所以Redis在利用RDB和AOF进行恢复的时候，都会读取RDB或AOF文件，重新加载到内存中。

Save：save 时只管保存，其他不管，全部阻塞。

Bgsave：redis 会在后台进行快照操作，快照操作的同时还可以响应客户端的请求，可以通过 lastsave 命令获取最后一次成功执行快照的时间。

执行 fluhall 命令，也会产生 dump.rdb 文件，但里面是空的。

RDB就是Snapshot快照存储，是默认的持久化方式。即按照一定的策略周期性的将数据保存到磁盘。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。Redis支持将当前数据的快照存成一个数据文件的持久化机制。而一个持续写入的数据库如何生成快照呢。Redis借助了fork命令的copy on write机制。在生成快照时，将当前进程fork出一个子进程，然后在子进程中循环所有的数据，将数据写成为RDB文件。

Client 也可以使用save或者bgsave命令通知redis做一次快照持久化。save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。另一点需要注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不 是增量的只同步脏数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘io操作，可能会严重影响性能。

Redis的RDB文件不会坏掉，因为其写操作是在一个新进程中进行的。当生成一个新的RDB文件时，Redis生成的子进程会先将数据写到一个临时文件中，然后通过原子性rename系统调用将临时文件重命名为RDB文件。这样在任何时候出现故障，Redis的RDB文件都总是可用的。并且Redis的RDB文件也是Redis主从同步内部实现中的一环:

***第一次Slave向Master同步的实现是：***

Slave向Master发出同步请求，Master先dump出rdb文件，然后将rdb文件全量传输给slave，然后Master把缓存的命令转发给Slave，初次同步完成。

***第二次以及以后的同步实现是：***

Master将变量的快照直接实时依次发送给各个Slave。但不管什么原因导致Slave和Master断开重连都会重复以上两个步骤的过程。

Redis的主从复制是建立在内存快照的持久化基础上的，只要有Slave就一定会有内存快照发生。

快照持久化过程

1.redis调用fork,现在有了子进程和父进程。

2.父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的写时复制机制（copy on write)父子进程会共享相同的物理页面，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程的地址空间内的数据是fork时刻整个数据库的一个快照。

3.当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出。

不足：就是一旦数据库出现问题，那么我们的RDB文件中保存的数据并不是全新的，从上次RDB文件生成到Redis停机这段时间的数据全部丢掉了（因为刷写机制还没有发出）。RDB就是Snapshot快照存储，是默认的持久化方式。

上面提出了RDB持久化的不足，要是不允许数据丢失，则需要用AOF来持久化。

AOF(Append Only File)<二进制文件>比RDB方式有更好的持久化性。由于在使用AOF持久化方式时，Redis会将每一个收到的写命令都通过Write函数追加到文件最后，类似于MySQL的binlog。当Redis重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。

AOF的完全持久化方式同时也带来了另一个问题，持久化文件会变得越来越大。(比如我们调用INCR test命令100次，文件中就必须保存全部的100条命令，但其实99条都是多余的。因为要恢复数据库的状态其实文件中保存一条SET test 100就够了)。为了合并重写AOF的持久化文件，Redis提供了bgrewriteaof命令。收到此命令后Redis将使用与快照类似的方式将内存中的数据以命令的方式保存到临时文件中，最后替换原来的文件，以此来实现控制AOF文件的合并重写。由于是模拟快照的过程，因此在重写AOF文件时并没有读取旧的AOF文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的AOF文件。

AOF持久化过程：

1. redis调用fork ，现在有父子两个进程

2. 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令

3.父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。

4.当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。

5.现在父进程可以使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。

从上面看出，RDB和AOF操作都是顺序IO操作，性能都很高。而同时在通过RDB文件或者AOF日志进行数据库恢复的时候，也是顺序的读取数据加载到内存中。所以也不会造成磁盘的随机读。

到底选择什么呢？下面是来自官方的建议：

通常，如果你要想提供很高的数据保障性，那么建议你同时使用两种持久化方式。如果你可以接受灾难带来的几分钟的数据丢失，那么你可以仅使用RDB。

很多用户仅使用了AOF，但是我们建议，既然RDB可以时不时的给数据做个完整的快照，并且提供更快的重启，所以最好还是也使用RDB。

在数据恢复方面：

RDB的启动时间会更短，原因有两个：

一是RDB文件中每一条数据只有一条记录，不会像AOF日志那样可能有一条数据的多次操作记录。所以每条数据只需要写一次就行了。

另一个原因是RDB文件的存储格式和Redis数据在内存中的编码格式是一致的，不需要再进行数据编码工作，所以在CPU消耗上要远小于AOF日志的加载。

注意：

上面说了RDB快照的持久化，需要注意：在进行快照的时候（save），fork出来进行dump操作的子进程会占用与父进程一样的内存，真正的copy-on-write，对性能的影响和内存的耗用都是比较大的。比如机器8G内存，Redis已经使用了6G内存，这时save的话会再生成6G，变成12G，大于系统的8G。这时候会发生交换；要是虚拟内存不够则会崩溃，导致数据丢失。所以在用redis的时候一定对系统内存做好容量规划。

目前，通常的设计思路是利用Replication机制来弥补aof、snapshot性能上的不足，达到了数据可持久化。即Master上Snapshot和AOF都不做，来保证Master的读写性能，而Slave上则同时开启Snapshot和AOF来进行持久化，保证数据的安全性。

AOF策略

Appendfsync 参数：

Always 同步持久化 每次发生数据变更会被立即记录到磁盘，性能较差但数据完整性较好。

Everysec： 出厂默认推荐，异步操作，每秒记录，日过一秒宕机，有数据丢失

No：从不 fsync ：将数据交给操作系统来处理。更快，也更不安全的选择。

Rewrite

概念：AOF 采用文件追加方式，文件会越来越来大为避免出现此种情况，新增了重写机制，aof 文件的大小超过所设定的阈值时，redis 就会自动 aof 文件的内容压缩，值保留可以恢复数据的最小指令集，可以使用命令 bgrewirteaof。

重写原理：aof 文件持续增长而大时，会 fork 出一条新进程来将文件重写(也就是先写临时文件最后再 rename)，遍历新进程的内存中的数据，每条记录有一条 set 语句，重写 aof 文件的操作，并没有读取旧的的 aof 文件，而是将整个内存的数据库内容用命令的方式重写了一个新的 aof 文件，这点和快照有点类似。

触发机制：redis 会记录上次重写的 aof 的大小，默认的配置当 aof 文件大小上次 rewrite 后大小的一倍且文件大于 64M 触发(3G)

**Redis集群方案：高可用**

<https://blog.csdn.net/yinwenjie/article/details/53672232>

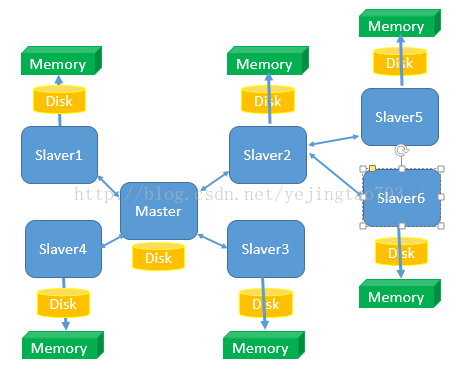
<https://blog.csdn.net/sunhuiliang85/article/details/78361211?locationNum=9&fps=1>

# 秒懂Redis集群设计原理

*Redis*集群设计包括*2*部分：哈希*Slot*和节点主从，本篇博文通过*3*张图来搞明白*Redis*的集群设计。

节点主从：

主从设计不算什么新鲜玩意，在数据库中我们也经常用主从来做读写分离，直接上图：



图上能看得到的信息：

*1，* 只有*1*个*Master*，可以有*N*个*slaver*，而且*Slaver*也可以有自己的*Slaver*，由于这种主从的关系决定他们是在配置阶段就要指定他们的上下级关系，而不是*Zookeeper*那种平行关系是自主推优出来的。

*2，* 读写分离，*Master*只负责写和同步数据给*Slaver*，*Slaver*承担了被读的任务，所以*Slaver*的扩容只能提高读效率不能提高写效率。

*3，* *Slaver*先将*Master*那边获取到的信息压入磁盘，再*load*进内存，*client*端是从内存中读取信息的，所以*Redis*是内存数据库。

当一个新的*Slaver*加入到这个集群时，会主动找*Master*来拜码头，*Master*发现新的小弟后将全量数据发送给新的*Slaver*，数据量越大性能消耗也就越大，所以尽量避免在运行时做*Slaver*的扩容。

简单总结下主从模式的设计：

优点：读写分离，通过增加*Slaver*可以提高并发读的能力。

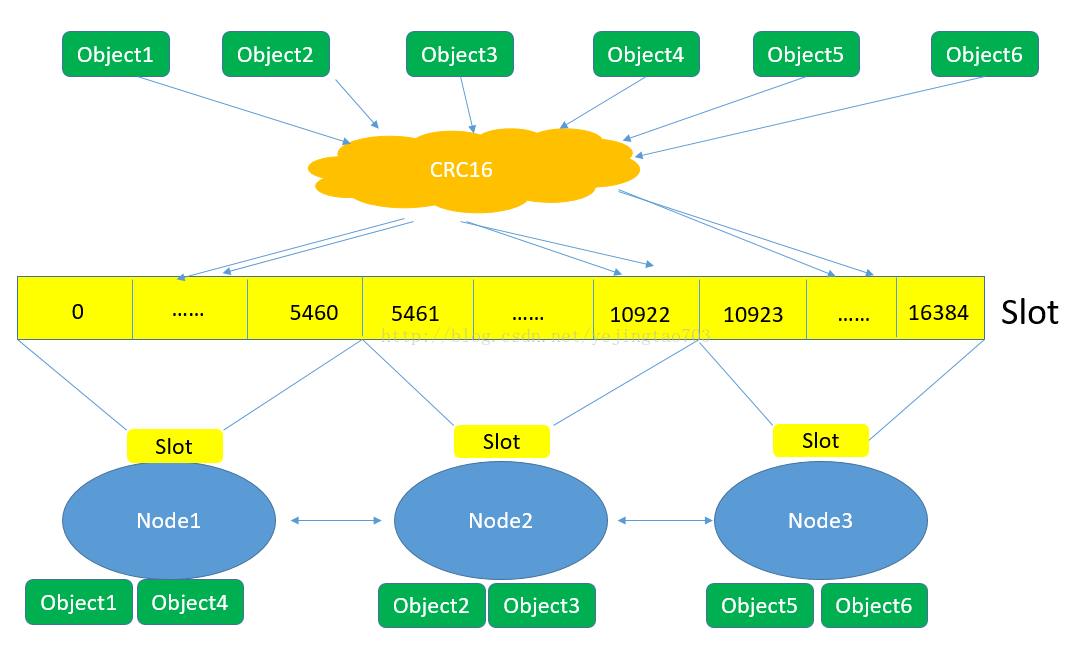
缺点：*Master*写能力是瓶颈。

虽然理论上对*Slaver*没有限制但是维护*Slaver*开销总将会变成瓶颈。

*Master*的*Disk*大小也将会成为整个*Redis*集群存储容量的瓶颈。

哈希*Slot*：

这个艺名看起来很文艺，但也不是什么新技术，他的真名就叫分表分库，再上一个图：



图上能看到的信息：

*1，* 对象保存到*Redis*之前先经过*CRC16*哈希到一个指定的*Node*上，例如*Object4*最终*Hash*到了*Node1*上。

*2，* 每个*Node*被平均分配了一个*Slot*段，对应着*0-16384*，*Slot*不能重复也不能缺失，否则会导致对象重复存储或无法存储。

*3，* *Node*之间也互相监听，一旦有*Node*退出或者加入，会按照*Slot*为单位做数据的迁移。例如*Node1*如果掉线了，*0-5640*这些*Slot*将会平均分摊到*Node2*和*Node3*上*,*由于*Node2*和*Node3*本身维护的*Slot*还会在自己身上不会被重新分配，所以迁移过程中不会影响到*5641-16384Slot*段的使用。

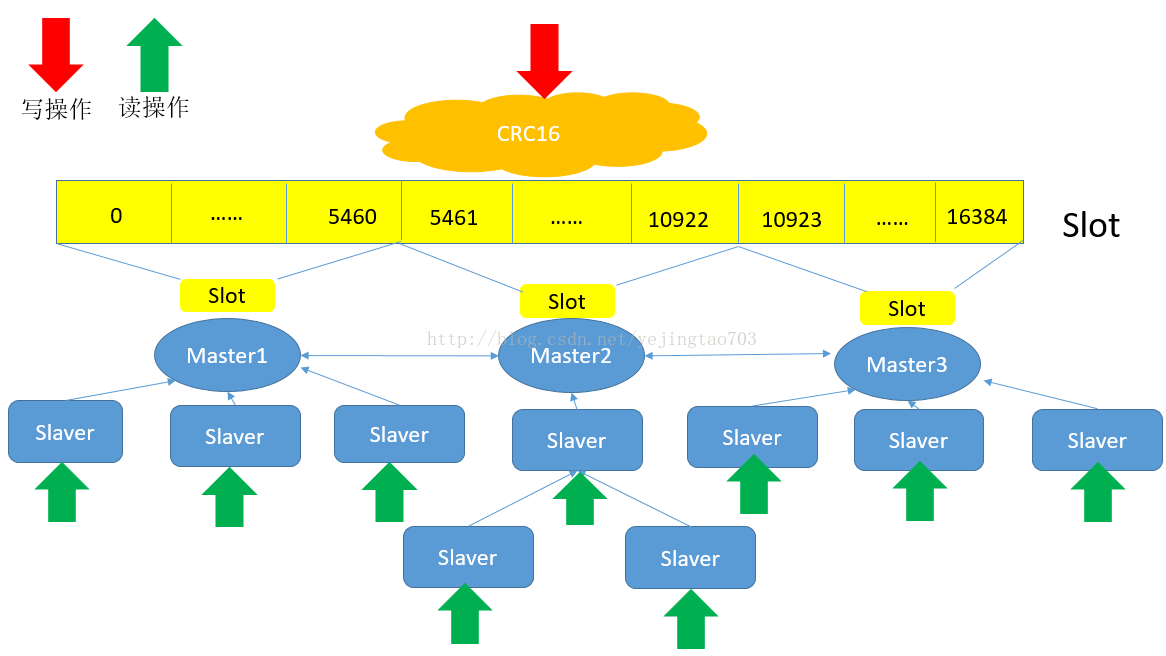
简单总结下哈希*Slot*的优缺点：

缺点：每个*Node*承担着互相监听、高并发数据写入、高并发数据读出，工作任务繁重

优点：将*Redis*的写操作分摊到了多个节点上，提高写的并发能力，扩容简单。

双剑合并：

看到这里大家也就发现了，主从和哈希的设计优缺点正好是相互弥补的，将图一每一套主从对应到图二中的每一个*Node*，就是*Redis*集群的终极形态，先*Hash*分逻辑节点，然后每个逻辑节点内部是主从，如图：



想扩展并发读就添加*Slaver*，想扩展并发写就添加*Master*，想扩容也就是添加*Master*，任何一个*Slaver*或者几个*Master*挂了都不会是灾难性的故障。