## 介绍一下 Redis

### Redis

Redis是一个开源的，**基于内存的数据结构存储**，可用作于数据库、**缓存**、消息中间件。Redis是**基于内存**，常用作于**缓存**的一种技术，并且Redis存储的方式是以key-value的形式。

**第一是数据库的数据，第二是页面的一些静态数据。**

### 数据存储？？？

Redis 集群中内置了 **16384个哈希槽**，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 **crc16 算法**算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 **0-16383** 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点。

### Redis单线程为什么快？

* 1）Redis是**纯内存**数据库，一般都是简单的**存取操作**，线程占用的时间很多，时间的花费主要集中在IO上，所以**读取速度快。**
* 2）Redis使用的是**非阻塞IO，IO多路复用**，使用了**单线程来轮询描述符，**将数据库的开、关、读、写都转**换成了事件**，减少了线程切换时上下文的切换和竞争。
* 3）**单线程保证了每个操作的原子性。**
* 4）Redis全程使用**hash结构，读取速度快**，还有一些特殊的数据结构，对数据存储进行了优化，如压缩表，对短数据进行压缩存储，再如，**跳表，使用有序的数据结构加快读取的速度。**

**不支持回滚，redis事务串行执行，执行失败也不回滚。**

MySQL 为了能进行回滚是花了不少的代价，Redis 应用的场景更多是对抗高并发具备高性能

### Redis的数据结构

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)**。

* 字符串类型的用法就是这么简单，因为是二进制安全的，所以你完全可以把一个图片文件的内容作为字符串来存储。**遇到数值操作时**，redis会将字符串类型转换成数值。（一个键最大能存储512MB）

字符串类型有当前值的字符长度决定使用哪种编码方式

int: 8个字节长度

embstr:大于8个字节小于39个字节

raw：大于39个字节

set college szu

1. 与MySQL做缓存  
2. 做Session共享  
3. 计数，如视频的播放次数  
4. 利用setnx做分布式锁  
5. 限速，利用expire过时机制，避免非法的频繁访问

* Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。存储、读取、修改用户属性

ziplist（压缩列表）  
在节省内存方面比较优秀

hashtable  
读写效率比较高，但是当hashtable作为hash的编码内部实现的时候，ziplist的读写效率会下降

hset person name bingo

* redis中的**lists在底层实现上并不是数组，而是链表（数据较少时使用ziplist，当数据量较大时改为使用linkedlist 双端链表）。**list的常用操作包括LPUSH、RPUSH、LRANGE、LPOP等。我们可以用LPUSH在list的左侧插入一个新元素，用RPUSH在list的右侧插入一个新元素，用LRANGE命令从list中指定一个范围来提取元素。1.实现一个**消息队列**，而且可以确保先后顺序2. LRANGE还可以很方便的实现分页的功能。

lpush mylist 1

lpush mylist 2

lpush mylist 3 4 5

rpop mylist

* **Set（集合）** 是通过**哈希表**实现的，所以添加，删除，查找的复杂度都是O(1)。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**。

#-------操作一个set-------

# 添加元素

**sadd mySet 1**

# 查看全部元素

smembers mySet

# 判断是否包含某个值

sismember mySet 3

# 删除某个/些元素

srem mySet 1

srem mySet 2 4

# 随机删除一个元素

spop mySet

#-------操作多个set-------

# 将一个set的元素移动到另外一个set

smove yourSet mySet 2

# 求两set的交集

**sinter** yourSet mySet

# 求两set的并集

**sunion** yourSet mySet

# 求在yourSet中而不在mySet中的元素

**sdiff** yourSet mySet

* Sorted Sets（Zset主要是**使用跳跃表**，每个节点具有一个value值作为排序依据）是将 **Set 中的元素增加了一个权重参数 score**，使得集合中的元素能够按 score 进行有序排列，比如一个存储全班同学成绩的 Sorted Sets，其集合 value 可以是同学的学号，而 score 就可以是其考试得分，这样在数据插入集合的时候，就已经进行了天然的排序。

sorted set 是排序的 set，去重但可以排序，写进去的时候给一个分数，自动根据分数排序。

zadd board 85 zhangsan

zadd board 72 lisi

zadd board 96 wangwu

zadd board 63 zhaoliu

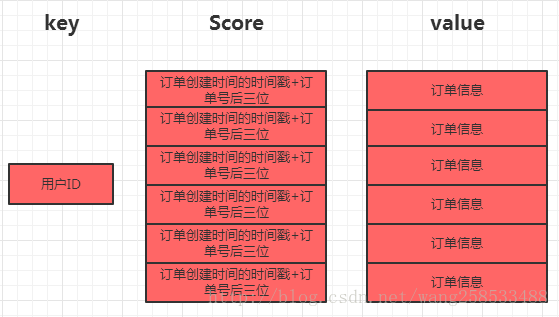
# 获取排名前三的用户（默认是升序，所以需要 rev 改为降序）

zrevrange board 0 3

# 获取某用户的排名

zrank board zhaoliu

***整个订单***的存储结构如下：



* 使用Redis的zset数据结构存储每个用户的订单，按照下单时间倒序排列，用户唯一标识作为key，用户的订单集合作为value，使用订单创建时间的时间戳+订单号后三位作为分数
* 为什么不直接使用下单时间的时间戳作为分数？因为下单时间只精确到秒，同一秒可能出现多个订单情况，这样就会出现相同的分数，而加上订单号后三位就能基本上避免这种情景。
* 只放用户的前N条订单即可，因为很少有用户会查看很久以前的订单，这样做会节省很多空间。如果有用户需要查看前N条之后的订单，再从数据库中查询即可，当然这种概率就比较小了。

简单来说就是**Redis对key-value封装成对象**，key是一个对象，value也是一个对象。每个对象都有type(类型)、encoding(编码)、ptr(指向底层数据结构的指针)来表示。

### 布隆过滤器

很长的[二进制](https://baike.baidu.com/item/%E4%BA%8C%E8%BF%9B%E5%88%B6/361457" \t "_blank)向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。本质上布隆过滤器是一种数据结构，比较巧妙的概率型数据结构。

它的优点是空间效率和查询时间都比一般的算法要好的多，缺点是有一定的误**识别率和删除困难。**

布隆过滤器是一个 bit 向量或者说 bit 数组。针对值 “baidu” 和三个不同的哈希函数分别生成了哈希值 1、4、7。随着增加的值越来越多，被置为 1 的 bit 位也会越来越多，这样某个值 “taobao” 即使没有被存储过，但是万一哈希函数返回的三个 bit 位都被其他值置位了 1 ，那么程序还是会判断 “taobao” 这个值存在。

如何选择**哈希函数个数**和布隆过滤器长度。

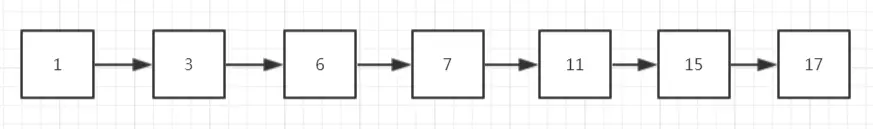
**误判率：bit数目和add的元素数目应该线性同步增加。**

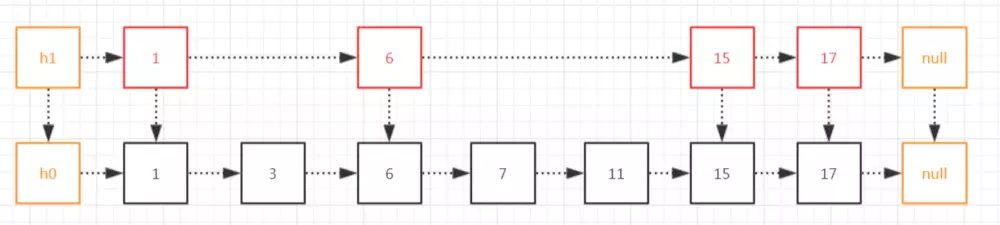
### 跳表

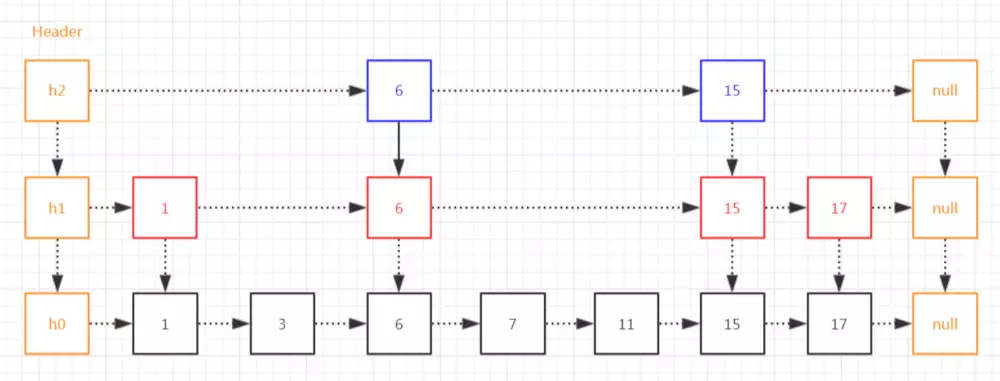
是一个**随机化的数据结构**，实质就是一种可以进行**二分**查找的**有序链表**。

**跳表在原有的有序链表**上面增加了**多级索引**，通过索引来实现快速查找。

我们要查找3、7、17这几个元素：







为什么Redis选择**使用跳表而不是红黑树来实现有序集合？**

首先，我们来分析下Redis的有序集合支持的操作：

1）插入元素

2）删除元素

3）查找元素

4）有序输出所有元素

5）查找区间内所有元素

其中，前4项红黑树都可以完成，且时间复杂度与跳表一致。

但是，最后一项，红黑树的效率就没有跳表高了。

在跳表中，要**查找区间的元素，**我们只要定位到两个区间端点在最低层级的位置，然后**按顺序遍历元素**就可以了，非常高效。

### 消息队列(MQ)

**任务队列是逻辑模型，消息队列是通信模型，RPC是包含了通信模型的编程模型（或者框架）**

消息队列（MQ）是一种能实现生产者到消费者单向通信的通信模型，一般来说是指实现这个模型的中间件，比如RabbitMQ。它可以是一个产品，或者是操作系统提供的一种服务之类。

RPC一般来说是具体指某一种RPC**编程框架或协议**。对于RPC来说，调用方不太关心底层的通信机制，只关心它能实现**远程调用**这一点；框架则不太关心上面承载的究竟是怎样的应用，只负责将调用过程发送到执行端，并将结果回传。

Redis提供了两种方式来做消息队列，一种是生产消费模式，另一种是发布订阅模式。

**生产消费模式**

**PUSH/POP**

生产消费模式会让**一个或多个客户端监听消息队列**，一旦消息到达，消费者马上消费，谁先抢到算谁的。如果队列中没有消息，消费者会继续监听。

Redis数据结构的列表List提供了push和pup命令，遵循着先入先出FIFO的原则。使用push/pop方式的优点在于消息可以持久化，**缺点是一条消息只能被一个消费者接收**，消费者完全靠手速来获取，是一种比较简陋的消息队列。

Redis的队列list是有序的且可以重复的，作为消息队列使用时可使用rpush/lpush操作入队，使用lpop/rpop操作出队。**当发布消息是执行lpush命令，将消息从列表左侧加入队列。消息接收方执行rpop命令从列表右侧弹出消息。**

如果使用休眠的方式让消费者线程间隔一段时间再消费，但这样做也有两个问题：

* 如果生产者速度大于消费者消费的速度，消息队列长度会一直增大，时间久了会占用大量内存空间。
* 如果休眠时间过长，就无法处理一些时效性的消息。如果休眠时间过短也会在连接上造成比较大的开销。

**LPUSH/BRPOP**

Redis提供了阻塞命令brpop/blpop。使用**brpop会阻塞队列**，而且每次只会弹出一个消息，如果没有消息则会阻塞。

Redis列表List支持带阻塞的命令，生产者从列表左侧lpush加入消息到队列，消费者使用brpop命令从列表右侧弹出消息并设置超时时间，如果列表中没有消息则一直阻塞直到超时。这样做的目的在于减小Redis的压力。

这种方案相对于发布订阅模式的好处是数据可靠性提高了，只有在Redis宕机且数据没有持久化的情况下会丢失数据。但相对于专业的消息队列中间件，发布订阅模式的状态过于简单（没有状态），而且没有ACK机制，消息取出后消费失败依赖于客户端记录日志或重新push到队列中。

**发布订阅模式**

发布订阅模式是一个或多个客户端订阅消息频道，只要发布者发布消息，所有订阅者都能收到消息，订阅者都是平等的。

PUB/SUB

**Redis自带pub/sub机制即发布订阅模式**，此模式中生产者producer和消费者consumer之间的关系是一对多的，也就是一条消息会被多个消费者所消费，当只有一个消费者时可视为一对一的消息队列。

发布订阅模式命令

* psubscribe 订阅一个或多个符合给定模式的频道
* publish 将消息发布到指定的频道
* pubsub查看订阅与发布系统状态
* pubsub channels pattern 列出当前的活跃频道
* pubsub numsub channel-1 channel-n 获取给定频道的订阅者数量
* pubsub numpat 获取订阅模式的数量
* punsubscribe 指示客户端退订所有给定模式
* subscribe 订阅给定的一个或多个频道的消息
* unsubscribe 指示客户端退订给定的频道

#### select，poll，epoll

(1)select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以**select具有O(n)的无差别轮询复杂度**，同时处理的流越多，无差别轮询时间就越长。

(2)poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到**内核空间**，然后查询每个fd对应的设备状态， **但是它没有最大连接数的限制**，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

**epoll可以理解为event poll**，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是**事件驱动（每个事件关联上fd）**的，此时我们对这些流的操作都是有意义的。**（复杂度降低到了O(1)）**

select，poll，epoll都是**IO多路复用的机制**。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的**，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

## Redis服务器中的数据库

### Redis数据库

数据库与数据库之间的数据是**隔离**的。

Redis的数据库就是使用字典(哈希表)来作为底层实现的，对**数据库的增删改查都是构建在字典(哈希表)的操作之上的**。

#### 生命周期

因为我们的内存是**有限**的。所以我们**会干掉不常用的数据，保留常用的数据**。这就需要我们**设置一下键的过期(生存)时间了**。

* 设置键的**生存**时间可以通过EXPIRE或者PEXPIRE命令。
* 设置键的**过期**时间可以通过EXPIREAT或者PEXPIREAT命令。

其实EXPIRE、PEXPIRE、EXPIREAT这三个命令都是通过PEXPIREAT命令来实现的。PEXPIREAT message 1391234400000

既然有**设置过期(生存)时间的命令**，那肯定也有移除过期时间，查看剩余生存时间的命令了：

* PERSIST(移除过期时间)
* TTL(Time To Live)返回剩余生存时间，**以秒为单位**；当 key 不存在时，返回 -2 。 当 key 存在但没有设置剩余生存时间时，返回 -1 。 否则，以秒为单位，返回 key 的剩余生存时间。
* PTTL以毫秒为单位返回键的剩余生存时间

#### 数据淘汰策略

删除策略的知识，**删除策略可分为三种**

* 定时删除(对内存友好，对CPU不友好)
  + 到时间点上就把所有过期的键删除了。
* 惰性删除(对CPU极度友好，对内存极度不友好)
  + 每次从键空间取键的时候，**判断一下该键是否过期了**，如果过期了就删除。
* 定期删除(折中)
  + **每隔**一段时间去删除过期键，**限制**删除的执行时长和频率。

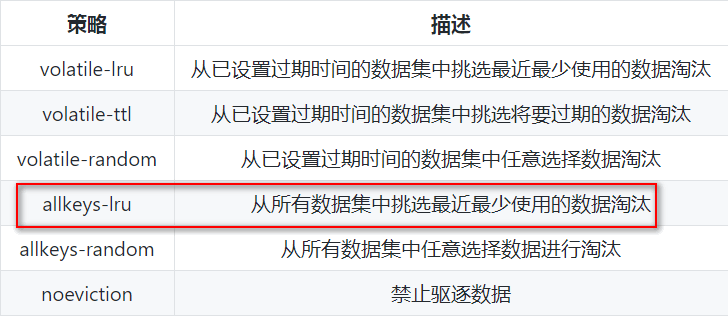
Redis采用的是**惰性删除+定期删除**两种策略，所以说，在Redis里边如果过期键到了过期的时间了，未必被立马删除的！

不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用**内存淘汰机制**。

#### 内存淘汰机制

我们可以**设置内存最大使用量，**当内存使用量超出时，会施行**数据淘汰策略**。

Redis的内存淘汰机制有以下几种：



1）noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。**应该没人用吧。**  
2）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。**推荐使用，目前项目在用这种。**  
3）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。**应该也没人用吧，你不删最少使用Key,去随机删。**  
4）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。**这种情况一般是把redis既当缓存，又做持久化存储的时候才用。不推荐**  
5）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。**依然不推荐**  
6）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。**不推荐**

一般场景：

使用 Redis 缓存数据时，为了提高缓存命中率，需要保证缓存数据都是**热点数据**。可以将内存最大使用量设置为热点数据占用的内存量，然后启用**allkeys-lru淘汰策略，**将最近最少使用的数据淘汰

### Redis持久化（保证可靠性）

**Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。**

Redis提供了两种不同的持久化方法来讲数据存储到硬盘里边：

* RDB(基于快照)，将**某一时刻的所有数据**保存到一个RDB文件中。
* AOF(append-only-file)，记录Redis服务器执行**写命令**，当服务器重启的时候会重新执行这些命令来恢复原始的数据，AOF 命令以 Redis 协议追加保存每次写的操作到文件末尾。

#### RDB(快照持久化)

有两个命令可以生成RDB文件：

* SAVE会**阻塞**Redis服务器进程，服务器不能接收任何请求，直到RDB文件创建完毕为止。
* BGSAVE创建出一个**子进程**，由子进程来负责创建RDB文件，服务器进程可以继续接收请求。

Redis服务器在启动的时候，如果发现有RDB文件，就会**自动**载入RDB文件(不需要人工干预)

* 服务器在载入RDB文件期间，会处于阻塞状态，直到载入工作完成。

除了手动调用SAVE或者BGSAVE命令生成RDB文件之外，我们可以使用**配置的方式来定期执行：**

在默认的配置下，如果以下的条件**被触发**，就会执行BGSAVE命令

save 900 1 #在900秒(15分钟)之后，至少有1个key发生变化，

save 300 10 #在300秒(5分钟)之后，至少有10个key发生变化

save 60 10000 #在60秒(1分钟)之后，至少有10000个key发生变化

总结：通过手动调用SAVE或者BGSAVE命令或者配置条件触发，将数据库**某一时刻**的数据快照，生成RDB文件实现持久化。

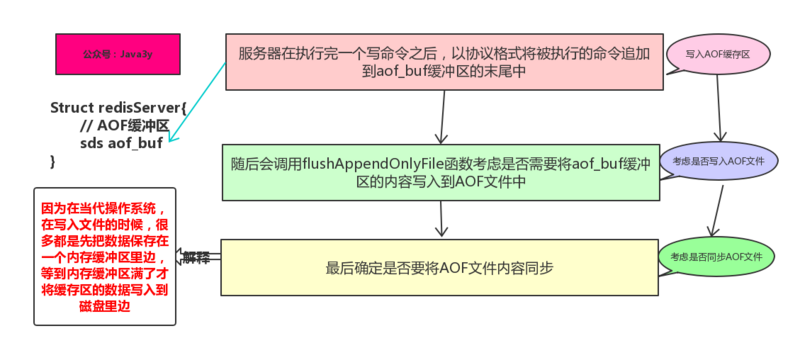
copy on write在另外一个进程做持久化，**这个时候Redis线程还是会接受写操作，**这部分**不会存在持久化进程**，所以不太可靠，AOF的话接受写的同时会同步消息到子进程；

#### AOF(文件追加)

* AOF是通过保存Redis服务器所执行的**写命令**来记录数据库的数据的。

AOF持久化功能的实现可以分为3个步骤：

* 命令追加：命令写入**aof\_buf缓冲区**
* 文件写入：调用flushAppendOnlyFile函数，考虑是否要将aof\_buf缓冲区写入AOF文件中
* 文件同步：考虑是否将内存缓冲区的数据真正写入到硬盘



flushAppendOnlyFile函数的行为**由服务器配置的appendfsyn选项**来决定的：

appendfsync always # 每次有数据修改发生时都会写入AOF文件。

appendfsync everysec # 每秒钟同步一次，该策略为AOF的默认策略。

appendfsync no # 从不同步。高效但是数据不会被持久化。

AOF重写由**Redis自行触发(参数配置)，**也可以用BGREWRITEAOF命令**手动触发**重写操作。

* 要值得说明的是：**AOF重写不需要对现有的AOF文件进行任何的读取、分析。AOF重写是通过读取服务器当前数据库的数据来实现的！**

Redis将AOF重写程序放到子进程里执行(BGREWRITEAOF命令)，像**BGSAVE命令一样fork出一个子进程来完成重写AOF的操作**，从而不会影响到主进程。

AOF后台重写是**不会阻塞**主进程接收请求的，新的写命令请求可能会导致**当前数据库和重写后的AOF文件的数据不一致**！

为了解决数据不一致的问题，Redis服务器设置了一个**AOF重写缓冲区**，这个缓存区会在服务器**创建出子进程之后使用**。

AOF 重写和 RDB 创建快照一样，都巧妙地利用了**写时复制机制**：

* Redis 执行 fork() ，现在同时拥有父进程和子进程。
* 子进程开始将新 AOF 文件的内容写入到临时文件。
* 对于所有新执行的**写入命令**，父进程一边将它们累积到**一个内存缓存中**，**一边将这些改动追加到现有 AOF 文件的末尾**，这样即使在重写的中途发生停机，现有的 AOF 文件也还是安全的。
* 当子进程完成重写工作时，它给父进程发送一个信号，父进程在接收到信号之后，将内存缓存中的所有数据追加到新 AOF 文件的末尾。
* 现在 Redis 原子地用新文件替换旧文件，之后所有命令都会直接追加到新 AOF 文件的末尾。

AOF存在什么问题吗：因为是以追加的方式实现的，所以日志存在膨胀的问题，但是有AOF重写的过程，所以这个其实问题不大，可能性能上差一点；

#### RDB和AOF

**RDB持久化对过期键的策略：**

* 执行SAVE或者BGSAVE命令创建出的RDB文件，程序会对数据库中的过期键检查，**已过期的键不会保存在RDB文件中**。
* 载入RDB文件时，程序同样会对RDB文件中的键进行检查，**过期的键会被忽略**。

AOF持久化对过期键的策略：

* 如果数据库的键已过期，但还没被惰性/定期删除，AOF文件不会因为这个过期键产生任何影响(**也就说会保留**)，当过期的键**被删除**了以后，会追加一条DEL命令来显示记录该键被删除了
* 重写AOF文件时，程序会对RDB文件中的键进行检查，**过期的键会被忽略**。

**RDB和AOF用哪个？**

RDB和AOF并不互斥，它俩可以**同时使用**。

* RDB的优点：载入时**恢复数据快**、文件体积小。
* RDB的缺点：会一定程度上**丢失数据**(因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。)
* AOF的优点：丢失数据少(默认配置只丢失一秒的数据)。
* **AOF的缺点：恢复数据相对较慢，文件体积大**
* RDB 方式可以保存过去一段时间内的数据，并且保存结果是一个单一的文件，可以将文件备份到其他服务器，并且在回复大量数据的时候，**RDB 方式的速度会比 AOF 方式的回复速度要快。**
* AOF 方式默认每秒钟备份 1 次，频率很高，它的操作方式是以追加的方式记录日志而不是数据，并且它的重写过程是按顺序进行追加，所以它的文件内容非常容易读懂。

如果Redis服务器**同时开启**了RDB和AOF持久化，服务器会**优先使用AOF文件**来还原数据(因为**AOF更新频率**比RDB更新频率要高，还原的数据更完善)

**可能涉及到RDB和AOF的配置：**

redis持久化，两种方式

1、rdb快照方式

2、aof日志方式

----------rdb快照------------

save 900 1

save 300 10

save 60 10000

**stop**-writes-**on**-bgsave-**error** yes

rdbcompression yes

rdbchecksum yes

dbfilename dump.rdb

dir /**var**/rdb/

-----------Aof的配置-----------

appendonly **no** # 是否打开 aof日志功能

appendfsync **always** #每一个命令都立即同步到aof，安全速度慢

appendfsync everysec

appendfsync **no** 写入工作交给操作系统，由操作系统判断缓冲区大小，统一写入到aof 同步频率低，速度快

## Redis事件

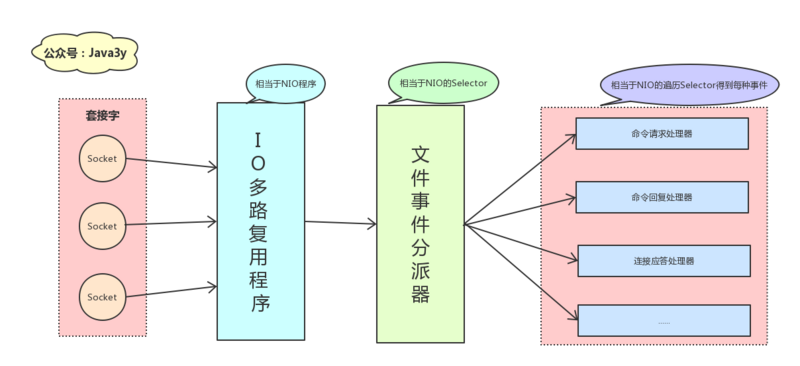
Redis服务器是一个**事件驱动程序**，主要处理以下两类事件：

* 文件事件：文件事件其实就是**对Socket操作的抽象**，Redis服务器与Redis客户端的**通信会产生文件事件**，服务器通过监听并处理这些事件来完成一系列的网络操作
* 时间事件：时间事件其实就是对**定时操作的抽象**，前面我们已经讲了RDB、AOF、定时删除键这些操作都可以由服务端去定时或者周期去完成，底层就是通过**触发时间事件**来实现的！

### 文件事件

Redis开发了自己的网络事件处理器，这个处理器被称为**文件事件处理器**。

文件事件处理器由四部分组成：

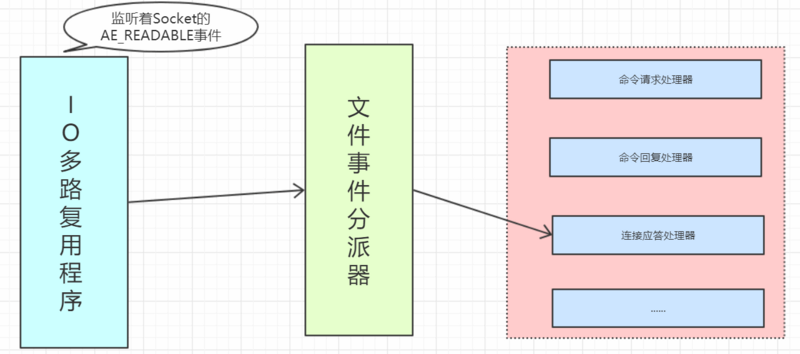


文件事件处理器使用I/O多路复用程序来**同时监听多个Socket**。当被监听的Socket准备好执行连接应答(accept)、读取(read)等等操作时，与操作相对应的文件事件就会产生，根据文件事件来为Socket关联对应的事件处理器，从而实现功能。

要值得注意的是：Redis中的I/O多路复用程序会将所有**产生事件的Socket放到一个队列**里边，然后通过这个队列以有序、同步、每次一个Socket的方式向文件事件分派器传送套接字。也就是说：**当上一个Socket处理完毕后，I/O多路复用程序才会向文件事件分派器传送下一个Socket。**

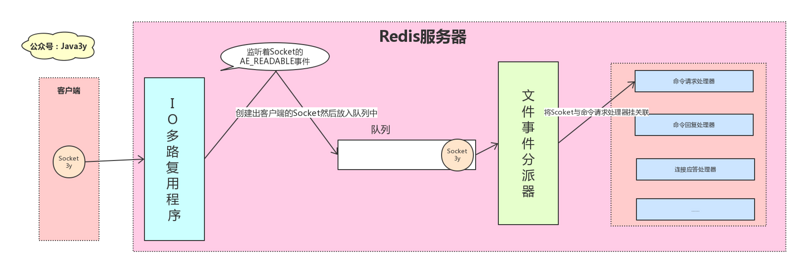
首先，IO多路复用程序**首先**会监听着Socket的AE\_READABLE事件，该事件对应着连接应答处理器

* 可以理解简单成**SocketServet.accpet()**



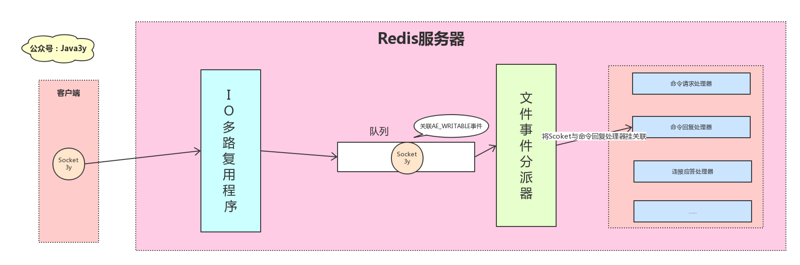
此时，一个名字叫做xz的Socket要连接服务器啦。服务器会用**连接应答处理器**处理。创建出客户端的Socket，并将客户端的Socket与命令请求处理器进行关联，使得客户端可以向服务器发送命令请求。

* 相当于Socket s = ss.accept();，创建出客户端的Socket，然后将该Socket关联**命令请求处理器**
* 此时客户端就可以向主服务器发送命令请求了



假设现在客户端发送一个命令请求set Java3y "关注、点赞、评论" ，客户端Socket将产生AE\_READABLE事件，**引发命令请求处理器执行**。处理器读取客户端的命令内容，然后传给对应的程序去执行。

客户端发送完命令请求后，**服务端总得给客户端回应的**。此时服务端会将客户端的Scoket的AE\_WRITABLE事件与命令回复处理器关联。



最后客户端尝试读取命令回复时，客户端Socket**产生**AE\_WRITABLE事件，触发命令回复处理器执行。当把所有的回复数据写入到Socket之后，服务器就会**解除**客户端Socket的AE\_WRITABLE事件与命令回复处理器的关联。

### 时间事件

持续运行的Redis服务器会**定期**对自身的资源和状态进行检查和调整，这些定期的操作由**serverCron函数**负责执行，它的主要工作包括：

* 更新服务器的统计信息(时间、内存占用、数据库占用)
* 清理数据库的过期键值对
* AOF、RDB持久化
* 如果是主从服务器，对从服务器进行定期同步
* 如果是集群模式，对进群进行定期同步和连接
* ...

Redis服务器将时间事件放在一个**链表**中，当时间事件执行器运行时，会遍历整个链表。时间事件包括：

* **周期性事件**(Redis一般只执行serverCron时间事件，serverCron时间事件是周期性的)
* 定时事件

### 时间事件和文件事件

* 文件事件和时间事件之间是**合作**关系，服务器会轮流**处理**这两种事件，并且处理事件的过程中不会发生抢占。
* 时间事件的实际处理事件通常会比设定的到达时间**晚**一些

## 主从架构

### 集群

* Redis集群**至少需要3个节点**，因为投票容错机制要求超过半数节点认为某个节点挂了该节点才是挂了，所以2个节点无法构成集群。
* 要保证集群的**高可用**，需要每个节点都有从节点，也就是**备份节点**，所以Redis集群至少需要6台服务器。因为我没有那么多服务器，也启动不了那么多虚拟机，所在这里搭建的是伪分布式集群，即一台服务器虚拟运行6个redis实例，修改端口号为（7001-7006），当然实际生产环境的Redis集群搭建和这里是一样的。
* 在usr/local目录下新建**redis-cluster目录**，用于存放集群节点。删除redis01目录下的**快照文件dump.rdb，**并且修改该目录下的**redis.cnf**文件，具体修改两处地方：一是端口号修改为7001，二是开启集群创建模式，打开注释即可。创建一个批量启动redis节点的脚本文件，**命令为start-all.sh；**创建好启动脚本文件之后，需要**修改该脚本的权限**，使之能够执行
* ruby工具，./redis-trib.rb create --replicas 1 192.168.131.102:7001 192.168.131.102:7002 192.168.131.102:7003 192.168.131.102:7004 192.168.131.102:7005 192.168.131.102:7006

### 主从架构的特点

Redis的主从架构特点：

* **主**服务器负责接收**写**请求
* **从**服务器负责接收**读**请求，比如很消耗性能的SORT就可以由从服务器来承担。
* 从服务器的数据由主服务器**复制**过去。主从服务器的数据是**一致**的

主从架构的**好处**：

* 读写分离(主服务器负责写，从服务器负责读)
* 高可用(某一台从服务器挂了，其他从服务器还能继续接收请求，不影响服务)
* 处理更多的并发量(每台从服务器**都可以接收读请求**，读QPS就上去了)
* 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。

### 复制功能

在Redis中，用户可以通过**执行SALVEOF命令或者设置salveof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(salve)

#### 复制功能的具体实现

复制功能分为两个操作：

* 同步(sync)
  + 将从服务器的数据库状态**更新至**主服务器的数据库状态
* 命令传播(command propagate)
  + 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**不一致**，执行主服务器相同的命令，让主从服务器的数据库状态**重新回到一致状态**。

**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**来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。

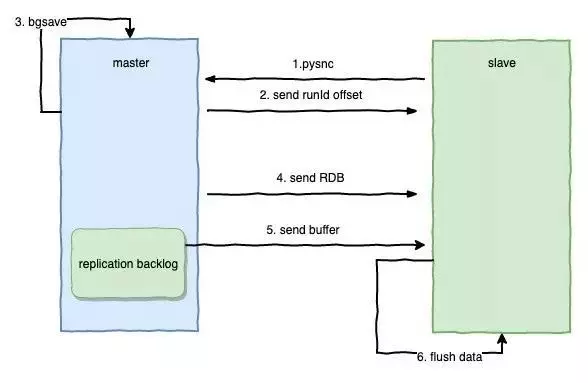
在BGSAVE指令执行完成后，主服务器会将持久化好的RDB文件发送给从服务器，从服务器接到此文件后会将其存储到磁盘上，然后再将其读取到内存中。这个动作完成后，主服务器会**将这段时间缓存的写指令再以redis协议的格式发送给从服务器。**

从服务器对主服务器的**同步又可以分为两种情况**：

* 初次同步：从服务器**没有复制过任何**的主服务器，或者从服务器要复制的主服务器跟上次复制的主服务器**不一样**。
* 断线后同步：处于**命令传播阶段**的主从服务器因为**网络原因**中断了复制，从服务器通过**自动重连**重新连接主服务器，并继续复制主服务器

在Redis2.8以前，断线后复制这部分其实缺少的只是**部分的数据**，但是要让主从服务器**重新执行SYNC命令**，这样的做法是非常低效的。(因为执行SYNC命令是把**所有的数据**再次同步，而不是只同步丢失的数据)

### PSYNC命令



Redis从2.8版本开始，使用PSYNC命令来**替代**SYNC命令执行复制时同步的操作。

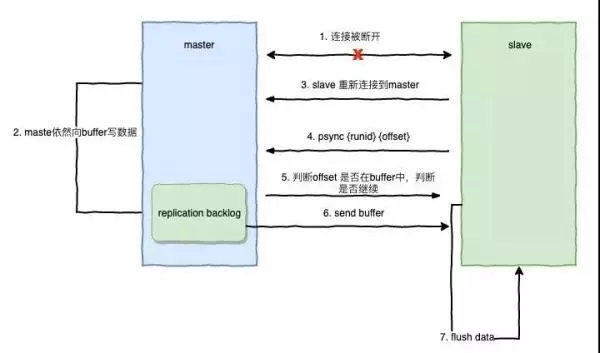
PSYNC命令具有**完整**重同步和**部分**重同步两种模式(其实就跟上面所说的初次复制和断线后复制差不多个意思)。

#### 完整重同步

下面先来看看**完整**重同步是怎么实现的：

* 从服务器向主服务器发送PSYNC命令
* 收到PSYNC命令的主服务器执行BGSAVE命令，在后台**生成一个RDB文件**。并用一个**缓冲区**来记录从现在开始执行的所有**写命令**。
* 当主服务器的BGSAVE命令执行完后，将生成的RDB文件发送给从服务器，**从服务器接收和载入RBD文件**。将自己的数据库状态更新至与主服务器执行BGSAVE命令时的状态。
* 主服务器将所有缓冲区的**写命令发送给从服务器**，从服务器执行这些写命令，达到数据最终一致性。

#### 部分重同步



接下来我们来看看**部分**重同步，部分重同步可以让我们断线后重连**只需要同步缺失的数据**(而不是Redis2.8之前的同步全部数据)，这是符合逻辑的！

部分重同步功能由以下部分组成：

* 主从服务器的**复制偏移量**
* 主服务器的**复制积压缓冲区**
* 服务器运行的ID(**run ID**)

首先我们来解释一下上面的名词：

复制偏移量：执行复制的双方都会**分别维护**一个复制偏移量

* 主服务器每次传播N个字节，就将自己的复制偏移量加上N
* 从服务器每次收到主服务器的N个字节，就将自己的复制偏移量加上N

通过**对比主从复制的偏移量**，就很容易知道主从服务器的数据是否处于一致性的状态！

那断线重连以后，从服务器向主服务器发送PSYNC命令，报告现在的偏移量，那么主服务器该对从服务器执行完整重同步还是部分重同步呢？？这就交由**复制积压缓冲区**来决定。

当主服务器进行命令传播时，不仅仅会将写命令发送给所有的从服务器，还会将写命令**入队到复制积压缓冲区**里面(这个大小可以调的)。如果复制积压缓冲区**存在**丢失的偏移量的数据，那就执行部分重同步，否则执行完整重同步。

服务器运行的ID(**run ID**)实际上就是用来比对ID是否相同。如果不相同，则说明从服务器断线之前复制的主服务器和当前连接的主服务器是两台服务器，这就会进行完整重同步。

### 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。这时主服务器只要将自己的写命令发送给从服务器，而从服务器接收并执行主服务器发送过来的写命令，就可以保证主从服务器一直保持数据一致了！

在命令传播阶段，从服务器默认会以每秒一次的频率，向服务器发送命令REPLCONF ACK <replication\_offset> 其中replication\_offset是从服务器当前的复制偏移量

发送这个命令主要有三个作用：

* **检测主从服务器的网络状态**
* 辅助实现min-slaves选项
* 检测命令丢失

## 哨兵(Sentinal)机制

### 哨兵

哨兵(Sentinal)机制主要用于实现Redis的**高可用性**，主要的功能如下：

* Sentinel**不停地监控**Redis主从服务器是否正常工作
* 如果某个Redis实例有故障，那么哨兵负责**发送消息通知**管理员
* 如果主服务器挂掉了，会**自动**将从服务器提升为主服务器(包括配置都会修改)。

挑选某一个从服务器作为主服务器也是有**策略**的，大概如下：

* （1）跟master断开连接的时长
* （2）slave优先级
* （3）复制offset
* （4）run id
* ***Sentinel可以作为配置中心***，能够提供当前主服务器的信息。

Sentinel本质上只是一个**运行在特殊模式下的Redis服务器**。

在启动的时候会将普通Redis服务器的代码替换成**Sentinel专用代码**。

初始化Sentinel的状态，并根据给定的配置文件**初始化**Sentinel监视的**主服务器列表**。

Sentinel会创建两个**连向主服务器的网络连接**：

* 命令连接(发送和接收命令)
* 订阅连接(订阅主服务器的\_sentinel\_:hello频道)

Sentinel通过主服务器发送**INFO命令**来获得主服务器属下所有从服务器的地址信息，并为这些从服务器创建相应的实例结构。

发现有**新的从服务器出现时**，除了创建对应的从服务器实例结构，Sentinel还会创建命令连接和订阅连接。

在Sentinel运行的过程中，通过命令连接会以每**两秒一次的频率**向**监视的主从服务器**的\_sentinel\_:hello频道发送命令(主要发送Sentinel本身的信息，监听主从服务器的信息)，并通过订阅连接接收\_sentinel\_:hello频道的信息。

* 这样一来一回，我们就可以**更新每个Sentinel实例结构的信息**。

### 判断主服务器是否下线了

判断主服务器是否下线有两种情况：

* 主观下线
  + Sentinel会以每秒一次的频率向与它创建命令连接的实例(包括主从服务器和其他的Sentinel)**发送PING命令**，通过PING命令返回的信息判断实例是否在线
  + 如果一个**主服务器**在down-after-milliseconds毫秒内连续向Sentinel发送**无效回复**，那么当前Sentinel就会**主观认为**该主服务器已经下线了。
* 客观下线
  + 当Sentinel将一个主服务器判断为主观下线以后，为了确认该主服务器是否真的下线，它会向同样监视该主服务器的Sentinel**询问**，看它们是否也认为该主服务器是否下线。
  + 如果**足够多**的Sentinel认为该主服务器是下线的，那么就判定该主服务为客观下线，并对主服务器执行故障转移操作。

### 丢失数据

目前为止的主从+哨兵架构可以说Redis是**高可用**的，Redis还是会**丢失数据**的

丢失数据有两种情况：

* 异步复制导致的数据丢失
  + **有部分数据还没复制到从服务器，主服务器就宕机了**，此时这些部分数据就丢失了
* 脑裂导致的数据丢失
  + 有时候主服务器脱离了正常网络，跟其他从服务器不能连接。此时哨兵可能就会**认为主服务器下线了**(然后开启选举，将某个从服务器切换成了主服务器)，但是实际上主服务器还运行着。这个时候，集群里就会有两个服务器(也就是所谓的脑裂)。
  + 虽然某个从服务器被切换成了主服务器，但是可能客户端**还没来得及切换到新的主服务器**，客户端还继续写向旧主服务器写数据。旧的服务器重新连接时，会作为从服务器复制新的主服务器(这意味着旧数据丢失)。

可以通过以下两个配置**尽量**减少数据丢失的可能：

min-slaves-to-write 1

min-slaves-max-lag 10

## 缓存问题

### 缓存雪崩

如果缓存数据**设置的过期时间是相同**的，并且Redis恰好将这部分数据全部删光了。这就会导致在这段时间内，这些缓存**同时失效**，全部请求到数据库中。

**这就是缓存雪崩**：

* Redis挂掉了，请求全部走数据库。
* 对缓存数据设置相同的过期时间，导致某段时间内缓存失效，请求全部走数据库。

缓存雪崩如果发生了，很可能就把我们的数据库**搞垮**，导致整个服务瘫痪！

**如何解决缓存雪崩？**

对于“对缓存数据设置相同的过期时间，导致某段时间内缓存失效，请求全部走数据库。”这种情况，非常好解决：

* 解决方法：在缓存的时候给过期时间加上一个**随机值**，这样就会大幅度的**减少缓存在同一时间过期**。

对于“Redis挂掉了，请求全部走数据库”这种情况，我们可以有以下的思路：

* 事发前：实现Redis的**高可用**(主从架构+Sentinel 或者Redis Cluster)，尽量**避免Redis挂掉**这种情况发生。
* 事发中：万一Redis真的挂了，我们可以设置**本地缓存(ehcache)+限流(hystrix)**，尽量避免我们的数据库被干掉(起码能保证我们的服务还是能正常工作的)
* 事发后：redis持久化，重启后自动从磁盘上加载数据，**快速恢复缓存数据**。

### 缓存穿透

缓存穿透是指查询一个一定**不存在的数据**。由于缓存不命中，并且出于容错考虑，如果从**数据库查不到数据则不写入缓存**，这将导致这个不存在的数据**每次请求都要到数据库去查询**，失去了缓存的意义。

**如何解决缓存穿透？**

解决缓存穿透也有两种方案：

* 由于请求的参数是不合法的(每次都请求不存在的参数)，于是我们可以使用布隆过滤器(BloomFilter)或者压缩filter**提前拦截**，不合法就不让这个请求到数据库层！
* 当我们从数据库找不到的时候，我们也将这个**空对象设置到缓存里边去**。下次再请求的时候，就可以从缓存里边获取了。
  + 这种情况我们一般会将空对象设置一个**较短的过期时间**。

### redis和数据库双写一致性问题

**分析**:一致性问题是分布式常见问题，还可以再分为最终一致性和强一致性。数据库和缓存双写，就必然会存在不一致的问题。

**突然数据库修改，数据库的数据更新到redis是会有时间差的，这样的时间差就会导致数据不一致。**

只能说**降低不一致发生的概率**，无法完全避免。

1.如果删除了缓存Redis，**还没有来得及写库MySQL**，另一个线程就来读取，发现缓存为空，则去数据库中读取数据写入缓存，此时缓存中为**脏数据**。

2.如果先写了库，在删除缓存前，**写库的线程宕机了**，没有删除掉缓存，则也会出现数据不一致情况。

因为写和读是并发的，**没法保证顺序,**就会出现缓存和数据库的数据不一致的问题。

**数据库和缓存之间一般不需要强一致性。**

一般缓存是这样的：

* 读走redis，如果redis不存在，从数据库里读，读完了更新redis。增删改先更新数据库，再更新redis。 好处是简单，但是如果更新数据库成功了，更新redis失败了，会查询到不一致的数据。
* **每次更新了相关的数据，都要把该缓存清理掉（**因为考虑到更新数据库后更新缓存可能会因为多线程下导致写入脏数据（比如线程A先更新数据库成功，接下来要取更新缓存，接着线程B更新数据库，但B又更新了缓存，接着B的时间片用完了，线程A更新了缓存**）**
* 为了避免极端条件下造成的缓存与数据库之间的数据不一致，**缓存需要设置一个失效时间。**时间到了，缓存自动被清理，达到缓存和数据库数据的“最终一致性”
* **增量消费mysql的binlog，自动同步redis。**binlog的读取等也是有延迟的，适用于对失效要求不严格的场景。
* 如果对数据实时性要求不高，就做个从oracle字典表同步到缓存的后台作业，**每隔5秒或10秒运行一次。还有一个思路，**就是在oracle在做个字典表的**触发器**，如果有数据更新的话，在触发器中调用java程序，同步到缓存中。