Redis支持多个数据库，并且每个数据库的数据是隔离的不能共享，并且基于单机才有，如果是集群就没有数据库的概念。

Redis是一个字典结构的存储服务器，而实际上一个Redis实例提供了多个用来存储数据的字典，客户端可以指定将数据存储在哪个字典中。这与我们熟知的在一个关系数据库实例中可以创建多个数据库类似，所以可以将其中的每个字典都理解成一个独立的数据库。

每个数据库对外都是一个从0开始的递增数字命名，Redis默认支持16个数据库（可以通过配置文件支持更多，无上限），可以通过配置databases来修改这一数字。客户端与Redis建立连接后会自动选择0号数据库，不过可以随时使用SELECT命令更换数据库，如要选择1号数据库：

```
redis> SELECT 1
OK
redis [1] > GET foo
(nil)
```

然而这些以数字命名的数据库又与我们理解的数据库有所区别。首先Redis不支持自定义数据库的名字，每个数据库都以编号命名，开发者必须自己记录哪些数据库存储了哪些数据。另外Redis也不支持为每个数据库设置不同的访问密码，所以一个客户端要么可以访问全部数据库，要么连一个数据库也没有权限访问。最重要的一点是多个数据库之间并不是完全隔离的，比如FLUSHALL命令可以清空一个Redis实例中所有数据库中的数据。综上所述，这些数据库更像是一种命名空间，而不适宜存储不同应用程序的数据。比如可以使用0号数据库存储某个应用生产环境中的数据，使用1号数据库存储测试环境中的数据，但不适宜使用0号数据库存储A应用的数据而使用1号数据库B应用的数据，不同的应用应该使用不同的Redis实例存储数据。由于Redis非常轻量级，一个空Redis实例占用的内在只有1M左右，所以不用担心多个Redis实例会额外占用很多内存。

一、redis中的数据类型

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)。

1.string 字符串

string 是 redis 最基本的类型，你可以理解成与 Memcached 一模一样的类型，一个 **key** 对应一个 **value**。

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，string 类型的值最大能存储 512MB。

```
redis 127.0.0.1:6379> SET name "runoob"
OK
redis 127.0.0.1:6379> GET name
"runoob"
```

使用redis中的set和get命令。键为name，对应的值为runoob。

一个键最大能存储512MB。

2.Hash 哈希

Redis hash是一个键值（key=>value）对集合。

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。

DEL runoob 用于删除前面测试用过的 key，不然会报错：(error) WRONGTYPE Operation against a key holding the wrong kind of value。

```
redis 127.0.0.1:6379> DEL runoob
redis 127.0.0.1:6379> HMSET myhash field1 "Hello" field2 "World"
"OK"
redis 127.0.0.1:6379> HGET myhash field1
"Hello"
redis 127.0.0.1:6379> HGET myhash field2
"World"
```

实例中我们使用了 Redis HMSET, HGET 命令，HMSET 设置了两个 field=>value 对, HGET 获取对应 field 对应的 value。

每个 hash 可以存储 232 -1 键值对（40多亿）

3.List 列表

Redis 列表是简单的字符串列表，**按照插入顺序排序**。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

```
redis 127.0.0.1:6379> DEL runoob
redis 127.0.0.1:6379> lpush runoob redis
(integer) 1
redis 127.0.0.1:6379> lpush runoob mongodb
(integer) 2
redis 127.0.0.1:6379> lpush runoob rabitmq
(integer) 3
redis 127.0.0.1:6379> lrange runoob 0 10
1) "rabitmq"
2) "mongodb"
3) "redis"
redis 127.0.0.1:6379>
```

列表最多可存储$2^{32} - 1$元素 (4294967295, 每个列表可存储40多亿)。

lpush命令说明（Left push）：

将所有指定的值插入到存于 key 的列表的头部。如果 key 不存在，那么在进行 push 操作前会创建一个空列表。 如果 key 对应的值不是一个 list 的话，那么会返回一个错误。

可以使用一个命令把多个元素 push 进入列表，只需在命令末尾加上多个指定的参数。元素是从最左端的到最右端的、一个接一个被插入到 list 的头部。 所以对于这个命令例子 `LPUSH mylist a b c`，返回的列表是 c 为第一个元素， b 为第二个元素， a 为第三个元素。时间复杂度为O(1)。

lpop命令说明（Left pop）：

移除并且返回 key 对应的 list 的第一个元素。

rpush命令说明（Right push）：

向存于 key 的列表的尾部插入所有指定的值。如果 key 不存在，那么会创建一个空的列表然后再进行 push 操作。 当 key 保存的不是一个列表，那么会返回一个错误。

可以使用一个命令把多个元素打入队列，只需要在命令后面指定多个参数。元素是从左到右一个接一个从列表尾部插入。 比如命令 RPUSH mylist a b c 会返回一个列表，其第一个元素是 a ，第二个元素是 b ，第三个元素是 c。

rpop命令说明（Right pop）：

移除并返回存于 key 的 list 的最后一个元素。

brpop命令说明（block right pop）：

BRPOP 是一个阻塞的列表弹出原语。 它是 RPOP 的阻塞版本，因为这个命令会在给定list无法弹出任何元素的时候阻塞连接。 该命令会按照给出的 key 顺序查看 list，并在找到的第一个非空 list 的尾部弹出一个元素。

请在 BLPOP 文档 中查看该命令的准确语义，因为 BRPOP 和 BLPOP 基本是完全一样的，除了它们一个是从尾部弹出元素，而另一个是从头部弹出元素。

```
redis> DEL list1 list2
(integer) 0
redis> RPUSH list1 a b c
(integer) 3
redis> BRPOP list1 list2 0
1) "list1"
2) "c"
```

BLPOP 是阻塞式列表的弹出原语。 它是命令 LPOP 的阻塞版本，这是因为当给定列表内没有任何元素可供弹出的时候， 连接将被 BLPOP 命令阻塞。 当给定多个 key 参数时，按参数 key 的先后顺序依次检查各个列表，**弹出第一个非空列表的头元素（即使后面的列表不为空，也不弹出）**。

```
127.0.0.1:6379> rpush queue1 a b c d
127.0.0.1:6379> rpush queue2 a b c d
127.0.0.1:6379> blpop queue1 queue2 0
1) "queue1"
2) "a"
## 注意queue2中的头元素没有弹出！！！
```

阻塞行为：如果所有给定 key 都不存在或包含空列表，那么 BLPOP 命令将阻塞连接， 直到有另一个客户端对给定的这些 key 的任意一个执行 LPUSH 或 RPUSH 命令为止。

一旦有新的数据出现在其中一个列表里，那么这个命令会解除阻塞状态，并且返回 key 和弹出的元素值。

当 BLPOP 命令引起客户端阻塞并且设置了一个非零的超时参数 timeout 的时候， 若经过了指定的 timeout 仍没有出现一个针对某一特定 key 的 push 操作，则客户端会解除阻塞状态并且返回一个 nil 的多组合值(multi-bulk value)。

timeout 参数表示的是一个指定阻塞的最大秒数的整型值。当 timeout 为 0 是表示阻塞时间无限制。

4.Set 集合

Redis的Set是string类型的无序集合。

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)。

sadd 命令：添加一个 string 元素到 key 对应的 set 集合中，成功返回1，如果元素已经在集合中返回 0，如果 key 对应的 set 不存在则返回错误。添加方式为：sadd key member

```
redis 127.0.0.1:6379> DEL runoob
redis 127.0.0.1:6379> sadd runoob redis
(integer) 1
redis 127.0.0.1:6379> sadd runoob mongodb
(integer) 1
redis 127.0.0.1:6379> sadd runoob rabitmq
(integer) 1
redis 127.0.0.1:6379> sadd runoob rabitmq
(integer) 0
redis 127.0.0.1:6379> smembers runoob

1) "redis"
2) "rabitmq"
3) "mongodb"
```

注意：以上实例中 rabitmq 添加了两次，但根据集合内元素的唯一性，第二次插入的元素将被忽略。

集合中最大的成员数为$2^{32}-1$(4294967295, 每个集合可存储40多亿个成员)。

SUNION key [key ...]命令说明：返回给定的多个集合的并集中的所有成员。如：

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SUNION key1 key2 key3 = {a,b,c,d,e}
```

不存在的key可以认为是空的集合.

SCARD key命令说明：返回集合存储的key的基数 (集合元素的数量)。返回值为integer-reply: 集合的基数(元素的数量),如果key不存在,则返回 0.

SDIFF key [key ...]命令说明：返回一个集合与给定集合的差集的元素。如

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SDIFF key1 key2 key3 = {b,d}
```

不存在的key认为是空集。

SDIFFSTORE destination key [key ...]命令说明：该命令类似于 SDIFF, 不同之处在于该命令不返回结果集，而是将结果存放在destination集合中。如果destination已经存在, 则将其覆盖重写。

```
redis> SADD key1 "a"
(integer) 1
redis> SADD key1 "b"
(integer) 1
redis> SADD key1 "c"
(integer) 1
redis> SADD key2 "c"
(integer) 1
redis> SADD key2 "d"
(integer) 1
redis> SADD key2 "e"
(integer) 1
redis> SDIFFSTORE key key1 key2
(integer) 2
redis> SMEMBERS key
1) "b"
2) "a"
redis>
```

SINTER key [key ...]命令说明：返回指定所有的集合的成员的交集.如：

```
key1 = {a,b,c,d}
key2 = {c}
key3 = {a,c,e}
SINTER key1 key2 key3 = {c}
```

SISMEMBER key member命令说明：返回成员 member 是否是存储的集合 key的成员.如：

```
redis> SADD myset "one"
(integer) 1
redis> SISMEMBER myset "one"
(integer) 1
redis> SISMEMBER myset "two"
(integer) 0
redis>
```

SMEMBERS key命令说明：返回key集合所有的元素。该命令的作用与使用一个参数的 SINTER 命令作用相同。

5.zset 有序集合 (sorted set)

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。 不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

zadd命令：添加元素到集合，元素在集合中存在则更新对应score ，添加方法为：zadd key score member

```
redis 127.0.0.1:6379> DEL runoob
redis 127.0.0.1:6379> zadd runoob 0 redis
(integer) 1
redis 127.0.0.1:6379> zadd runoob 0 mongodb
(integer) 1
redis 127.0.0.1:6379> zadd runoob 0 rabitmq
(integer) 1
redis 127.0.0.1:6379> zadd runoob 0 rabitmq
(integer) 0
redis 127.0.0.1:6379> > ZRANGEBYSCORE runoob 0 1000
1) "mongodb"
2) "rabitmq"
3) "redis"
```

类型	简介	特性	场景
String(字符串)	二进制安全	可以包含任何数据,比如jpg图片或者序列化的对象,一个键最大能存储512M	---
Hash(字典)	键值对集合,即编程语言中的Map类型	适合存储对象,并且可以像数据库中update一个属性一样只修改某一项属性值(Memcached中需要取出整个字符串反序列化成对象修改完再序列化存回去)	存储、读取、修改用户属性
List(列表)	链表(双向链表)	增删快,提供了操作某一段元素的API	1,最新消息排行等功能(比如朋友圈的时间线) 2,消息队列
Set(集合)	哈希表实现,元素不重复	1、添加、删除,查找的复杂度都是O(1) 2、为集合提供了求交集、并集、差集等操作	1、共同好友 2、利用唯一性,统计访问网站的所有独立ip 3、好友推荐时,根据tag求交集,大于某个阈值就可以推荐
Sorted Set(有序集合)	将Set中的元素增加一个权重参数score,元素按score有序排列	数据插入集合时,已经进行天然排序	1、排行榜 2、带权重的消息队列

二、redis集群

http://www.redis.cn/topics/cluster-tutorial.html

Redis 集群是一个提供在多个**Redis**间节点间共享数据的程序集。

Redis集群并不支持处理多个keys的命令,因为这需要在不同的节点间移动数据,从而达不到像Redis那样的性能,在高负载的情况下可能会导致不可预料的错误.

Redis 集群通过分区来提供一定程度的可用性,在实际环境中当某个节点宕机或者不可达的情况下继续处理命令. Redis 集群的优势:

- 自动分割数据到不同的节点上。
- 整个集群的部分节点失败或者不可达的情况下能够继续处理命令。

1.Redis 集群的数据分片

Redis 集群没有使用一致性hash, 而是引入了 哈希槽的概念.

Redis 集群有16384（2^{14}）个哈希槽,每个key通过CRC16校验后对16384取模来决定放置哪个槽.集群的每个节点负责一部分hash槽,举个例子,比如当前集群有3个节点,那么:

- 节点 A 包含 0 到 5500号哈希槽.
- 节点 B 包含5501 到 11000 号哈希槽.
- 节点 C 包含11001 到 16384号哈希槽.

这种结构很容易添加或者删除节点. 比如如果我想新添加个节点D, 我需要从节点 A, B, C中得部分槽到D上. 如果我想移除节点A,需要将A中的槽移到B和C节点上,然后将没有任何槽的A节点从集群中移除即可. 由于从一个节点将哈希槽移动到另一个节点并不会停止服务,所以无论添加删除或者改变某个节点的哈希槽的数量都不会造成集群不可用的状态.

2.Redis 集群的主从复制模型

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有N-1个复制品.

在我们例子中具有A，B，C三个节点的集群,在没有复制模型的情况下,如果节点B失败了，那么整个集群就会以为缺少5501-11000这个范围的槽而不可用.

然而如果在集群创建的时候（或者过一段时间）我们为每个节点添加一个从节点A1，B1，C1,那么整个集群便有三个master节点和三个slave节点组成，这样在节点B失败后，集群便会选举B1为新的主节点继续服务，整个集群便不会因为槽找不到而不可用了

不过当B和B1 都失败后，集群是不可用的.

3.Redis 一致性保证

Redis 并不能保证数据的强一致性. 这意味这在实际中集群在特定的条件下可能会丢失写操作.

第一个原因是因为集群是用了异步复制. 写操作过程:

- 客户端向主节点B写入一条命令.
- 主节点B向客户端回复命令状态.
- 主节点将写操作复制给他的从节点 B1, B2 和 B3.

主节点对命令的复制工作发生在返回命令回复之后， 因为如果每次处理命令请求都需要等待复制操作完成的话， 那么主节点处理命令请求的速度将极大地降低 —— 我们必须在性能和一致性之间做出权衡。 注意：Redis 集群可能会在将来提供同步写的方法。 Redis 集群另外一种可能会丢失命令的情况是集群出现了网络分区， 并且一个客户端与至少包括一个主节点在内的少数实例被孤立。

举个例子 假设集群包含 A 、 B 、 C 、 A1 、 B1 、 C1 六个节点， 其中 A 、 B 、 C 为主节点， A1 、 B1 、 C1 为A，B，C的从节点， 还有一个客户端 Z1 假设集群中发生网络分区，那么集群可能会分为两方，大部分的一方包含节点 A 、 C 、 A1 、 B1 和 C1 ，小部分的一方则包含节点 B 和客户端 Z1 .

Z1仍然能够向主节点B中写入, 如果网络分区发生时间较短,那么集群将会继续正常运作,如果分区的时间足够让大部分的一方将B1选举为新的master，那么Z1写入B中得数据便丢失了.

注意， 在网络分裂出现期间， 客户端 Z1 可以向主节点 B 发送写命令的最大时间是有限制的， 这一时间限制称为节点超时时间（node timeout）， 是 Redis 集群的一个重要的配置选项:

4.搭建并使用Redis集群

（1）配置实例

搭建集群的第一件事情我们需要一些运行在 集群模式的Redis实例. 这意味这集群并不是由一些普通的Redis实例组成的，集群模式需要通过配置启用，开启集群模式后的Redis实例便可以使用集群特有的命令和特性了.

下面是一个最少选项的集群的配置文件:

```
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

文件中的 cluster-enabled 选项用于开实例的集群模式， 而 cluster-conf-file 选项则设定了保存节点配置文件的路径， 默认值为 nodes.conf。节点配置文件无须人为修改， 它由 Redis 集群在启动时创建， 并在有需要时自动进行更新。

要让集群正常运作至少需要三个主节点，不过在刚开始试用集群功能时， 强烈建议使用六个节点： 其中三个为主节点， 而其余三个则是各个主节点的从节点。

首先， 让我们进入一个新目录， 并创建六个以端口号为名字的子目录， 稍后我们在将每个目录中运行一个 Redis 实例： 命令如下:

```
mkdir cluster-test
cd cluster-test
mkdir 7000 7001 7002 7003 7004 7005
```

在文件夹 7000 至 7005 中， 各创建一个 redis.conf 文件， 文件的内容可以使用上面的示例配置文件， 但记得将配置中的端口号从 7000 改为与文件夹名字相同的号码。

启动配置的集群中的每个redis实例:

```
cd 7000
../redis-server ./redis.conf
```

实例打印的日志显示，因为nodes.conf文件不存在，所以每个节点都为它自身指定了一个新的 ID :

```
[6876] 15 Apr 13:59:37.159 * No cluster configuration found, I'm
58fd86059ad0035ca6789717e1f6b74fdf99d61e
```

实例会一直使用同一个 ID ， 从而在集群中保持一个独一无二（unique）的名字。

（2）搭建集群

现在我们已经有了六个正在运行中的 Redis 实例， 接下来我们需要使用这些实例来创建集群， 并为每个节点编写配置文件。

通过使用 Redis 集群命令行工具 redis-trib ， 编写节点配置文件的工作可以非常容易地完成： redis-trib 位于 Redis 源码的 src 文件夹中， 它是一个 Ruby 程序， 这个程序通过向实例发送特殊命令来完成创建新集群， 检查集群， 或者对集群进行重新分片（reshared）等工作。

```
./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
```

这个命令在这里用于创建一个新的集群, 选项–replicas 1 表示我们希望为集群中的每个主节点创建一个从节点。

之后跟着的其他参数则是这个集群实例的地址列表,3个master3个slave redis-trib 会打印出一份预想中的配置给你看， 如果你觉得没问题的话， 就可以输入 yes ， redis-trib 就会将这份配置应用到集群当中,让各个节点开始互相通讯,最后可以得到如下信息:

```
[OK] All 16384 slots covered
```

这表示集群中的 16384 个槽都有至少一个主节点在处理， 集群运作正常。

（3）使用集群

测试 Redis 集群比较简单的办法就是使用 redis-rb-cluster 或者 redis-cli ， 接下来我们将使用 redis-cli 为例来进行演示:

```
$ redis-cli -c -p 7000
redis 127.0.0.1:7000> set foo bar
-> Redirected to slot [12182] located at 127.0.0.1:7002
OK
redis 127.0.0.1:7002> set hello world
-> Redirected to slot [866] located at 127.0.0.1:7000
OK
redis 127.0.0.1:7000> get foo
-> Redirected to slot [12182] located at 127.0.0.1:7002
"bar"
redis 127.0.0.1:7000> get hello
-> Redirected to slot [866] located at 127.0.0.1:7000
"world"
```

Redis 集群现阶段的一个问题是客户端实现很少。

以下是一些我知道的实现:

- redis-rb-cluster 是我（@antirez）编写的 Ruby 实现， 用于作为其他实现的参考。 该实现是对 redis-rb 的一个简单包装， 高效地实现了与集群进行通讯所需的最

少语义（semantic）.
- redis-py-cluster 看上去是 redis-rb-cluster 的一个 Python 版本， 这个项目有一段时间没有更新了（最后一次提交是在六个月之前）， 不过可以将这个项目用作学习集群的起点。
- 流行的 Predis 曾经对早期的 Redis 集群有过一定的支持， 但我不确定它对集群的支持是否完整， 也不清楚它是否和最新版本的 Redis 集群兼容 （因为新版的 Redis 集群将槽的数量从 4k 改为 16k 了）.
- 使用最多的时java客户端, Jedis 最近添加了对集群的支持, 详细请查看项目README中*Jedis Cluster*部分.
- StackExchange.Redis 提供对 C# 的支持(并且包括大部分 .NET 下面的语言，比如： VB, F#等等)
- thunk-redis 提供对 Node.js 和 io.js的支持。
- Redis unstable 分支中的 `redis-cli` 程序实现了非常基本的集群支持， 可以使用命令 redis-cli -c 来启动。

（4）集群重新分片

现在， 让我们来试试对集群进行重新分片操作。在执行重新分片的过程中， 请让你的 example.rb 程序处于运行状态， 这样你就会看到， 重新分片并不会对正在运行的集群程序产生任何影响， 你也可以考虑将 example.rb 中的 sleep 调用删掉， 从而让重新分片操作在近乎真实的写负载下执行 重新分片操作基本上就是将某些节点上的哈希槽移动到另外一些节点上面， 和创建集群一样， 重新分片也可以使用 redis-trib 程序来执行 执行以下命令可以开始一次重新分片操作:

```
1 ./redis-trib.rb reshard 127.0.0.1:7000
```

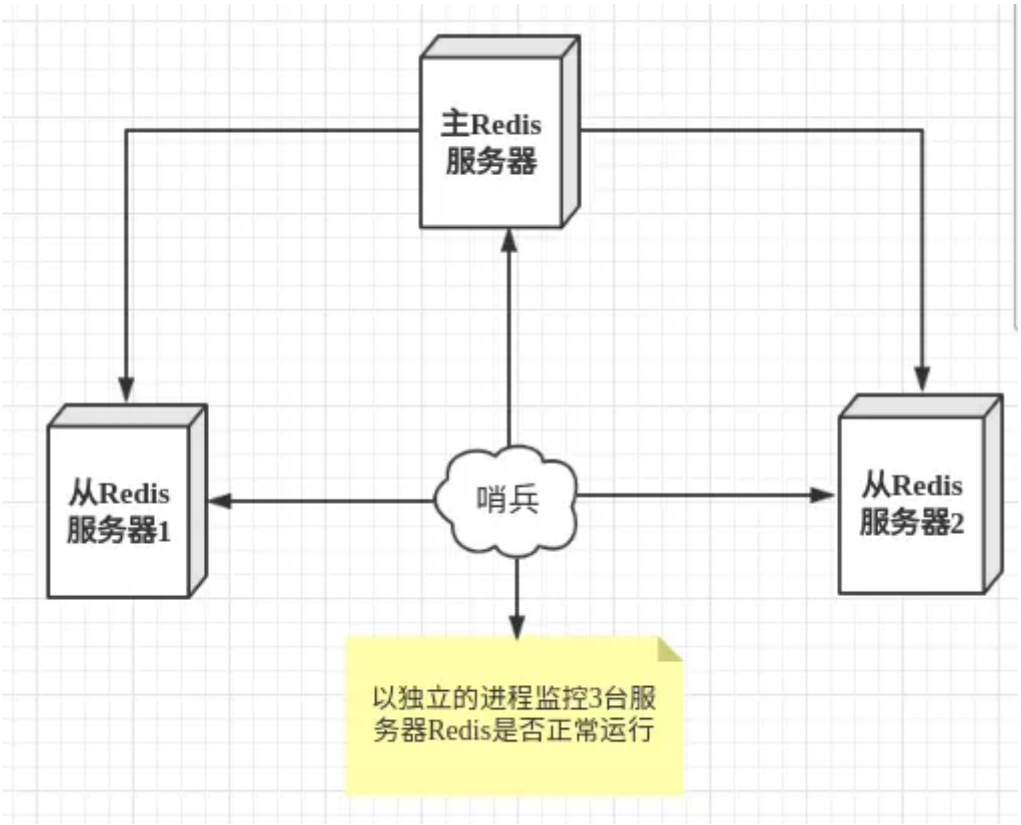
你只需要指定集群中其中一个节点的地址， redis-trib 就会自动找到集群中的其他节点。

目前 redis-trib 只能在管理员的协助下完成重新分片的工作， 要让 redis-trib 自动将哈希槽从一个节点移动到另一个节点， 目前来说还做不到。

5.哨兵模式

主从切换技术的方法是：当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这不是一种推荐的方式，更多时候，我们优先考虑哨兵模式。

哨兵模式是一种特殊的模式，首先Redis提供了哨兵的命令，哨兵是一个独立的进程，作为进程，它会独立运行。其原理是**哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。**



这里的哨兵有两个作用：

- 通过发送命令，让Redis服务器返回监控其运行状态，包括主服务器和从服务器。
- 当哨兵监测到master宕机，会自动将slave切换成master，然后通过发布订阅模式通知其他的从服务器，修改配置文件，让它们切换主机。

然而一个哨兵进程对Redis服务器进行监控，可能会出现问题，为此，我们可以使用多个哨兵进行监控。各个哨兵之间还会进行监控，这样就形成了多哨兵模式。

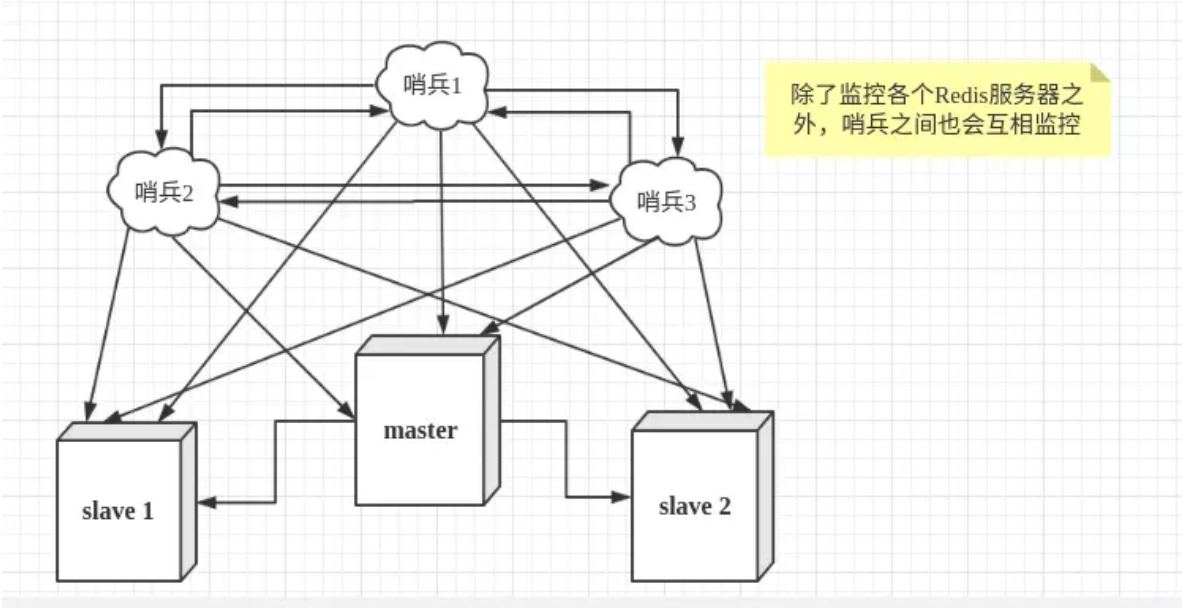
用文字描述一下**故障切换（failover）**的过程。假设主服务器宕机，哨兵1先检测到这个结果，系统并不会马上进行failover过程，仅仅是哨兵1主观的认为主服务器不可用，这个现象成为**主观下线**。当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行failover操作。切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为**客观下线**。这样对于客户端而言，一切都是透明的。

（1）配置过程

配置3个哨兵和1主2从的Redis服务器来演示这个过程。

服务类型	是否是主服务器	IP地址	端口
Redis	是	192.168.11.128	6379
Redis	否	192.168.11.129	6379
Redis	否	192.168.11.130	6379
Sentinel	-	192.168.11.128	26379

服务类型	是否是主服务器	IP地址	端口
Sentinel	-	192.168.11.129	26379
Sentinel	-	192.168.11.130	26379



配置Redis的主从服务器，修改redis.conf文件如下。

下述内容主要是配置Redis服务器，从服务器比主服务器多一个slaveof的配置和密码。

```
# 使得Redis服务器可以跨网络访问
bind 0.0.0.0
# 设置密码
requirepass "123456"
# 指定主服务器，注意：有关slaveof的配置只是配置从服务器，主服务器不需要配置
slaveof 192.168.11.128 6379
# 主服务器密码，注意：有关slaveof的配置只是配置从服务器，主服务器不需要配置
masterauth 123456
```

配置3个哨兵，每个哨兵的配置都是一样的。在Redis安装目录下有一个sentinel.conf文件，copy一份进行修改：

```
# 禁止保护模式
protected-mode no
# 配置监听的主服务器，这里sentinel monitor代表监控，mymaster代表服务器的名称，可以自定义，192.168.11.128代表监控的主服务器，6379代表端口，2代表只有两个或两个以上的哨兵认为主服务器不可用的时候，才会进行failover操作。
sentinel monitor mymaster 192.168.11.128 6379 2
# sentinel author-pass定义服务的密码，mymaster是服务名称，123456是Redis服务器密码
# sentinel auth-pass <master-name> <password>
sentinel auth-pass mymaster 123456
```

上述关闭了保护模式，便于测试。

有了上述的修改，我们可以进入Redis的安装目录的src目录，通过下面的命令启动服务器和哨兵：

```
# 启动Redis服务器进程
./redis-server ../redis.conf
# 启动哨兵进程
./redis-sentinel ../sentinel.conf
```

注意启动的顺序。首先是主机（**192.168.11.128**）的**Redis**服务进程，然后启动从机的服务进程，最后启动**3**个哨兵的服务进程。

（2）Java中使用哨兵模式

```
/**
 * 测试Redis哨兵模式
 * @author liu
 */
public class TestSentinels {
    @SuppressWarnings("resource")
    @Test
    public void testSentinel() {
        JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
        jedisPoolConfig.setMaxTotal(10);
        jedisPoolConfig.setMaxIdle(5);
        jedisPoolConfig.setMinIdle(5);
        // 哨兵信息
        Set<String> sentinels = new HashSet<>
(Arrays.asList("192.168.11.128:26379",
                "192.168.11.129:26379","192.168.11.130:26379"));
        // 创建连接池
        JedisSentinelPool pool = new JedisSentinelPool("mymaster",
sentinels,jedisPoolConfig,"123456");
        // 获取客户端
        Jedis jedis = pool.getResource();
        // 执行两个命令
        jedis.set("mykey", "myvalue");
        String value = jedis.get("mykey");
        System.out.println(value);
    }
}
```

上面是通过Jedis进行使用的，同样也可以使用Spring进行配置RedisTemplate使用。

```
        <bean id = "poolConfig"
class="redis.clients.jedis.JedisPoolConfig">
            <!-- 最大空闲数 -->
            <property name="maxIdle" value="50"></property>
            <!-- 最大连接数 -->
            <property name="maxTotal" value="100"></property>
            <!-- 最大等待时间 -->
            <property name="maxWaitMillis" value="20000">
</property>
        </bean>

```

```
10          <bean id="connectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnect
ionFactory">
11              <constructor-arg name="poolConfig" ref="poolConfig">
</constructor-arg>
12              <constructor-arg name="sentinelConfig"
ref="sentinelConfig"></constructor-arg>
13              <property name="password" value="123456"></property>
14          </bean>
15
16          <!-- JDK序列化器 -->
17          <bean id="jdkSerializationRedisSerializer"
class="org.springframework.data.redis.serializer.JdkSerializationRe
disSerializer"></bean>
18
19          <!-- String序列化器 -->
20          <bean id="stringRedisSerializer"
class="org.springframework.data.redis.serializer.StringRedisSeriali
zer"></bean>
21
22          <bean id="redisTemplate"
class="org.springframework.data.redis.core.RedisTemplate">
23              <property name="connectionFactory"
ref="connectionFactory"></property>
24              <property name="keySerializer"
ref="stringRedisSerializer"></property>
25              <property name="defaultSerializer"
ref="stringRedisSerializer"></property>
26              <property name="valueSerializer"
ref="jdkSerializationRedisSerializer"></property>
27          </bean>
28
29          <!-- 哨兵配置 -->
30          <bean id="sentinelConfig"
class="org.springframework.data.redis.connection.RedisSentinelConfi
guration">
31              <!-- 服务名称 -->
32              <property name="master">
33                  <bean
class="org.springframework.data.redis.connection.RedisNode">
34                      <property name="name" value="mymaster">
</property>
35                  </bean>
36              </property>
37              <!-- 哨兵服务IP和端口 -->
38              <property name="sentinels">
39                  <set>
40                      <bean
class="org.springframework.data.redis.connection.RedisNode">
41                          <constructor-arg name="host"
value="192.168.11.128"></constructor-arg>
42                          <constructor-arg name="port" value="26379">
</constructor-arg>
43                      </bean>
```

```
                    <bean class="org.springframework.data.redis.connection.RedisNode">
                        <constructor-arg name="host" value="192.168.11.129"></constructor-arg>
                        <constructor-arg name="port" value="26379"></constructor-arg>
                    </bean>
                    <bean class="org.springframework.data.redis.connection.RedisNode">
                        <constructor-arg name="host" value="192.168.11.130"></constructor-arg>
                        <constructor-arg name="port" value="26379"></constructor-arg>
                    </bean>
                </set>
            </property>
        </bean>
```

三、Redis实现分布式锁

在多线程环境中，锁是实现共享资源互斥访问的重要机制，以保证任何时刻只有一个线程在访问共享资源。锁的基本原理是：用一个状态值表示锁，对锁的占用和释放通过状态值来标识，因此基于redis实现的分布式锁主要依赖redis的SETNX命令和DEL命令，SETNX相当于上锁，DEL相当于释放锁，当然，在下面的具体实现中会更复杂些。之所以称为分布式锁，是因为客户端可以在redis集群环境中向集群中任一个可用Master节点请求上锁（即SETNX命令存储key到redis缓存中是随机的）。

1.分布式锁的实现要点

为了实现分布式锁，需要确保锁同时满足以下四个条件：

- 互斥性。在任意时刻，只有一个客户端能持有锁
- 不会发送死锁。即使一个客户端持有锁的期间崩溃而没有主动释放锁，也需要保证后续其他客户端能够加锁成功
- 加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给释放了。
- 容错性。只要大部分的Redis节点正常运行，客户端就可以进行加锁和解锁操作。

2.分布式锁实现

```
SET resource_name my_random_value NX PX 30000
```

这个命令仅在不存在key的时候才能被执行成功（NX选项），并且这个key有一个30秒的自动失效时间（PX属性）。这个key的值是“my_random_value”(一个随机值），这个值在所有的客户端必须是唯一的，所有同一key的获取者（竞争者）这个值都不能一样。value的值必须是随机数主要是为了更安全的释放锁，释放锁的时候使用脚本告诉Redis:只有key存在并且存储的值和我指定的值一样才能告诉我删除成功。

加锁操作的正确姿势为：

- 使用setnx命令保证互斥性， SET key value NX 效果等同于 SETNX key value
- 需要设置锁的过期时间，避免死锁

- setnx和设置过期时间需要保持原子性，避免在设置setnx成功之后在设置过期时间客户端崩溃导致死锁
- 加锁的Value 值为一个唯一标示。可以采用UUID作为唯一标示。加锁成功后需要把唯一标示返回给客户端来用来客户端进行解锁操作

解锁的正确姿势为：

- 需要拿加锁成功的唯一标示要进行解锁，从而保证加锁和解锁的是同一个客户端
- 解锁操作需要比较唯一标示是否相等，相等再执行删除操作。这2个操作可以采用Lua脚本方式使2个命令的原子性。即：

```
if redis.call('get', KEYS[1]) == ARGV[1] then
  return redis.call('del', KEYS[1])
else
  return 0
end
```

具体实现如下：

定义接口DistributedLock：

```
public interface DistributedLock {
    /**
     * 获取锁
     * @author zhi.li
     * @return 锁标识
     */
    String acquire();

    /**
     * 释放锁
     * @author zhi.li
     * @param indentifier
     * @return
     */
    boolean release(String indentifier);
}

```

定义实现RedisDistributedLock：

```
public class RedisDistributedLock implements DistributedLock {
    private static final String LOCK_SUCCESS = "OK";
    private static final Long RELEASE_SUCCESS = 1L;
    private static final String SET_IF_NOT_EXIST = "NX";

    /**
     * redis 客户端
     */
    private Jedis jedis;

    /**
     * 分布式锁的键值
     */
```

```
14      private String lockKey;
15
16      /**
17       * 锁的超时时间 10s
18       */
19      int expireTime = 10 * 1000;
20
21      /**
22       * 锁等待，防止线程饥饿
23       */
24      int acquireTimeout  = 1 * 1000;
25
26      /**
27       * 获取指定键值的锁
28       * @param jedis jedis Redis客户端
29       * @param lockKey 锁的键值
30       */
31      public RedisDistributedLock(Jedis jedis, String lockKey) {
32          this.jedis = jedis;
33          this.lockKey = lockKey;
34      }
35
36      /**
37       * 获取指定键值的锁,同时设置获取锁超时时间
38       * @param jedis jedis Redis客户端
39       * @param lockKey 锁的键值
40       * @param acquireTimeout 获取锁超时时间
41       */
42      public RedisDistributedLock(Jedis jedis,String lockKey, int
    acquireTimeout) {
43          this.jedis = jedis;
44          this.lockKey = lockKey;
45          this.acquireTimeout = acquireTimeout;
46      }
47
48      /**
49       * 获取指定键值的锁,同时设置获取锁超时时间和锁过期时间
50       * @param jedis jedis Redis客户端
51       * @param lockKey 锁的键值
52       * @param acquireTimeout 获取锁超时时间
53       * @param expireTime 锁失效时间
54       */
55      public RedisDistributedLock(Jedis jedis, String lockKey, int
    acquireTimeout, int expireTime) {
56          this.jedis = jedis;
57          this.lockKey = lockKey;
58          this.acquireTimeout = acquireTimeout;
59          this.expireTime = expireTime;
60      }
61      @Override
62      public String acquire() {
63          // 获取锁的超时时间，超过这个时间则放弃获取锁
64          try{
65              long end = System.currentTimeMillis() +
    acquireTimeout;
```

```
66              String requireToken = UUID.randomUUID().toString();
67              while (System.currentTimeMillis() < end) {
68                  SetParams params = new SetParams();
69                  // 设置过期时间（单位毫秒）
70                  params.px(expireTime);
71                  // 一定要这个nx，它表示的意思是：只在键不存在时，才对键进行设置操作。 SET key value NX 效果等同于 SETNX key value 。
72                  // 当键存在，则返回的是(nil)，当键不存在，则返回的是OK
73                  params.nx();
74                  // 根据需要再添加一些其他参数 ...
75                  String result = jedis.set(lockKey, requireToken, params);
76                  if(LOCK_SUCCESS.equals(result)){
77                      System.out.printf("当前线程%s成功获取锁!\n",Thread.currentThread().getName());
78                      return requireToken;
79                  }
80                  System.out.println("======================");
81                  try {
82                      Thread.sleep(100);
83                  } catch (InterruptedException e) {
84                      Thread.currentThread().interrupt();
85                  }
86              }
87          }catch(Exception e){
88              e.printStackTrace();
89          }
90          return null;
91      }
92
93      @Override
94      public boolean release(String identify ) {
95          if(identify == null){
96              return false;
97          }
98          try{
99              String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";
100             Object result = new Object();
101             result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(identify));
102             if (RELEASE_SUCCESS.equals(result)) {
103                 System.out.printf("当前线程%s成功释放锁!\n",Thread.currentThread().getName());
104                 return true;
105             }
106         }catch(Exception e){
107             e.printStackTrace();
108         }finally{
109             if(jedis != null){
110                 jedis.close();
111             }
112         }
113         return false;
```

```
    }
}
```

下面就以秒杀库存数量为场景，测试下上面实现的分布式锁的效果：

```
public class RedisDistributedLockTest {
    static int n = 500;
    public static void secskill() {
        System.out.println(--n);
    }

    public static void main(String[] args) {
        Runnable runnable = () -> {
            RedisDistributedLock lock = null;
            String unLockIdentify = null;
            try {
                Jedis conn = new Jedis("127.0.0.1",6379);
                lock = new RedisDistributedLock(conn, "test1");
                unLockIdentify = lock.acquire();
                System.out.println(Thread.currentThread().getName()
+ "正在运行");
                secskill();
            } finally {
                if (lock != null) {
                    lock.release(unLockIdentify);
                }
            }
        };
        ExecutorService executorService =
Executors.newFixedThreadPool(10);
        for (int i = 0; i < 10; i++) {
            executorService.submit(runnable);
        }
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        executorService.shutdown();
    }
}
```

3.redlock算法

https://github.com/redisson/redisson/wiki/8.-%E5%88%86%E5%B8%83%E5%BC%8F%E9%94%81%E5%92%8C%E5%90%8C%E6%AD%A5%E5%99%A8#84-%E7%BA%A2%E9%94%81redlock

http://redis.cn/topics/distlock.html

上述方法是单Redis实例实现分布式锁的实现思路，基于Redis单实例，假设这个单实例总是可用，这种方法已经足够安全。现在让我们扩展一下，假设Redis没有总是可用的保障。

在Redis的分布式环境中，我们假设有N个Redis master。这些节点完全互相独立，不存在主从复制或者其他集群协调机制。之前我们已经描述了在Redis单实例下怎么安全地获取和释放锁。我们确保将在每（N)个实例上使用此方法获取和释放锁。在这个样例中，我们假设有5个Redis master节点，这是一个比较合理的设置，所以我们需要在5台机器上面或者5台虚拟机上面运行这些实例，这样保证他们不会同时都宕掉。

为了取到锁，客户端应该执行以下操作:

1. 获取当前Unix时间，以毫秒为单位。
2. 依次尝试从N个实例，使用相同的key和随机值获取锁。在步骤2，当向Redis设置锁时,客户端应该设置一个网络连接和响应超时时间，这个超时时间应该小于锁的失效时间。例如你的锁自动失效时间为10秒，则超时时间应该在5-50毫秒之间。这样可以避免服务器端Redis已经挂掉的情况下，客户端还在死死地等待响应结果。如果服务器端没有在规定时间内响应，客户端应该尽快尝试另外一个Redis实例。
3. 客户端使用当前时间减去开始获取锁时间（步骤1记录的时间）就得到获取锁使用的时间。当且仅当从大多数（这里是3个节点）的Redis节点都取到锁，并且使用的时间小于锁失效时间时，锁才算获取成功。
4. 如果取到了锁，key的真正有效时间等于有效时间减去获取锁所使用的时间（步骤3计算的结果）。
5. 如果因为某些原因，获取锁失败（*没有*在至少N/2+1个Redis实例取到锁或者取锁时间已经超过了有效时间），客户端应该在所有的Redis实例上进行解锁（即便某些Redis实例根本就没有加锁成功）。

四、Redis为何是单线程

因为CPU不是Redis的瓶颈。Redis的瓶颈最有可能是机器内存或者网络带宽。（以上主要来自官方FAQ）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。

如果在多线程中操作，那就需要为这些对象加锁。所以使用多线程可以提高性能，但是每个线程的效率严重下降了，而且程序的逻辑严重复杂化。Redis的数据结构并不全是简单的Key-Value，还有list，hash等复杂的结构，这些结构有可能会进行很细粒度的操作，比如在很长的列表后面添加一个元素，在hash当中添加或者删除一个对象，这些操作还可以合成MULTI/EXEC的组。这样一个操作中可能就需要加非常多的锁，导致的结果是同步开销大大增加。Redis在权衡之后的选择是用单线程，突出自己功能的灵活性。在单线程基础上任何原子操作都可以几乎无代价地实现，多么复杂的数据结构都可以轻松运用，甚至可以使用Lua脚本这样的功能。对于多线程来说这需要高得多的代价。

五、订阅发布模式

为了订阅foo和bar，客户端发出一个订阅的频道名称:

```
SUBSCRIBE foo bar
```

其他客户端发到这些频道的消息将会被推送到所有订阅的客户端。

客户端订阅到一个或多个频道不必发出命令，尽管他能订阅和取消订阅其他频道。订阅和取消订阅的响应被封装在发送的消息中，以便客户端只需要读一个连续的消息流，其中第一个元素表示消息类型。

1.推送消息的格式

消息是一个有三个元素的多块响应 。

第一个元素是消息类型:

- subscribe: 表示我们成功订阅到响应的第二个元素提供的频道。第三个参数代表我们现在订阅的频道的数量。
- unsubscribe:表示我们成功取消订阅到响应的第二个元素提供的频道。第三个参数代表我们目前订阅的频道的数量。当最后一个参数是0的时候，我们不再订阅到任何频道。当我们在Pub/Sub以外状态，客户端可以发出任何redis命令。
- message: 这是另外一个客户端发出的发布命令的结果。第二个元素是来源频道的名称，第三个参数是实际消息的内容。

2.数据库与作用域

发布/订阅与key所在空间没有关系，它不会受任何级别的干扰，包括不同数据库编码。 发布在db 10,订阅可以在db 1。 如果你需要区分某些频道，可以通过在频道名称前面加上所在环境的名称（例如：测试环境，演示环境，线上环境等）。

示例:

```
在一个客户端A内，订阅了两个频道：
127.0.0.1:6379> subscribe channel1 channel2
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
1) "subscribe"
2) "channel2"
3) (integer) 2

然后在另外一个客户端B发布在channel1上发布消息：
127.0.0.1:6379> publish channel1 channelMessage
(integer) 1

在客户端A中收到消息：
1) "message"
2) "channel1"
3) "channelMessage"
```

现在客户端用没有任何参数的 UNSUBSCRIBE命令取消订阅所有频道:

3.模式匹配订阅

Redis 的Pub/Sub实现支持模式匹配。客户端可以订阅全风格的模式以便接收所有来自能匹配到给定模式的频道的消息。

比如:

```
PSUBSCRIBE news.*
```

将接收所有发到news.art.figurative, news.music.jazz等等的消息，所有模式都是有效的，所以支持多通配符。

```
PUNSUBSCRIBE news.*
```

将取消订阅匹配该模式的客户端，这个调用不影响其他订阅。

当作模式匹配结果的消息会以不同的格式发送:

- 消息类型是pmessage:这是另一客户端发出的PUBLISH命令的结果,匹配一个模式匹配订阅。第一个元素是原匹配的模式，第三个元素是原频道名称，最后一个元素是实际消息内容。

同样的，系统默认 SUBSCRIBE 和 UNSUBSCRIBE, PSUBSCRIBE 和 PUNSUBSCRIBE 命令在发送 psubscribe 和punsubscribe类型的消息时使用像subscribe 和 unsubscribe一样的消息格式。

六、将Redis当做LRU算法的缓存来使用

当Redis被当做缓存来使用，当你新增数据时，让它自动地回收旧数据是件很方便的事情。这个行为在开发者社区非常有名，因为它是流行的memcached系统的默认行为。

LRU是Redis唯一支持的回收方法。本页面包括一些常规话题，Redis的`maxmemory`指令用于将可用内存限制成一个固定大小，还包括了Redis使用的LRU算法，这个实际上只是近似的LRU。

1.Maxmemory配置指令

`maxmemory`配置指令用于配置Redis存储数据时指定限制的内存大小。通过redis.conf可以设置该指令，或者之后使用CONFIG SET命令来进行运行时配置。

例如为了配置内存限制为100mb，以下的指令可以放在`redis.conf`文件中。

```
maxmemory 100mb
```

设置`maxmemory`为0代表没有内存限制。对于64位的系统这是个默认值，对于32位的系统默认内存限制为3GB。

当指定的内存限制大小达到时，需要选择不同的行为，也就是**策略**。 Redis可以仅仅对命令返回错误，这将使得内存被使用得更多，或者回收一些旧的数据来使得添加数据时可以避免内存限制。

2.回收策略

当maxmemory限制达到的时候Redis会使用的行为由 Redis的maxmemory-policy配置指令来进行配置。

以下的策略是可用的:

- **noeviction**:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）
- **allkeys-lru**: 尝试回收最近最少使用的键（LRU），使得新添加的数据有空间存放。
- **volatile-lru**: 尝试回收最近最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。
- **allkeys-random**: 回收随机的键使得新添加的数据有空间存放。
- **volatile-random**: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。
- **volatile-ttl**: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

如果没有键满足回收的前提条件的话，策略**volatile-lru**, **volatile-random**以及**volatile-ttl**就和noeviction 差不多了。

选择正确的回收策略是非常重要的，这取决于你的应用的访问模式，不过你可以在运行时进行相关的策略调整，并且监控缓存命中率和没命中的次数，通过RedisINFO命令输出以便调优。

一般的经验规则:

- 使用**allkeys-lru**策略：当你希望你的请求符合一个幂定律分布，也就是说，你希望部分的子集元素将比其它其它元素被访问的更多。如果你不确定选择什么，这是个很好的选择。.
- 使用**allkeys-random**：如果你是循环访问，所有的键被连续的扫描，或者你希望请求分布正常（所有元素被访问的概率都差不多）。
- 使用**volatile-ttl**：如果你想要通过创建缓存对象时设置TTL值，来决定哪些对象应该被过期。

allkeys-lru 和 **volatile-random**策略对于当你想要单一的实例实现缓存及持久化一些键时很有用。不过一般运行两个实例是解决这个问题的更好方法。

为了键设置过期时间也是需要消耗内存的，所以使用**allkeys-lru**这种策略更加高效，因为没有必要为键取设置过期时间当内存有压力时。

七、Redis应用场景

string——适合最简单的k-v存储，类似于memcached的存储结构，短信验证码，配置信息等，就用这种类型来存储。

hash——一般key为ID或者唯一标示，value对应的就是详情了。如商品详情，个人信息详情，新闻详情等。

list——因为list是有序的，比较适合存储一些有序且数据相对固定的数据。如省市区表、字典表等。因为list是有序的，适合根据写入的时间来排序，如：最新的***，消息队列等。

set——可以简单的理解为ID-List的模式，如微博中一个人有哪些好友，set最牛的地方在于，可以对两个set提供交集、并集、差集操作。例如：查找两个人共同的好友等。

Sorted Set——是set的增强版本，增加了一个score参数，自动会根据score的值进行排序。比较适合类似于top 10等不根据插入的时间来排序的数据。

八、数据持久化

Redis支持RDB和AOF两种持久化机制，持久化功能有效地避免因进程退出造成的数据丢失问题，当下次重启时利用之前持久化的文件即可实现数据恢复。

1.RDB

RDB持久化时把当前进程数据生成快照保存到硬盘的过程，出发RDB持久化过程分为手动触发和自动触发。

手动触发本别对应save和bgsave命令。

- save命令：阻塞当前Redis服务器，直到RDB过程完成为止，对于内存比较大的实例会造成长时间阻塞，线上环境不建议使用。运行save命令对应的Redis日志如下：

  > ** DB saved on disk*

- bgsave命令：Redis进程执行fork操作创建子进程，RDB持久化过程由于子进程负责，完成后自动结束。阻塞只发生在fork阶段，一般时间很短。运行bgsave命令对应的Redis日志如下：

  > ** Background saving started by pid 3151*
  >
  > ** DB saved on disk*
  >
  > ** RDB: 0 MB of memory used by copy-on-write*
  >
  > ** Background saving terminated with success*

显然bgsave命令是针对save阻塞问题做的优化。因此Redis内部所有的设计RDB的操作都采用bgsave的方式，而save命令已经废弃。除了执行命令手动触发之外，Redis内部还存在自动触发RDB的持久化机制，例如如下场景：

1）使用save相关配置，如“save m n”。表示m秒内数据集存在n次修改时，自动触发bgsave。

2）如果从节点执行全量复制操作，主节点自动执行bgsave生成RDB文件并发送给从节点。

3）执行debug reload 命令重新加载Redis时，也会自动触发save操作。

4）默认情况下执行shutdown命令时，如果没有开启AOF持久化功能则自动执行bgsave。

bgsave是主流的触发RDB持久化方式，其流程如下如：

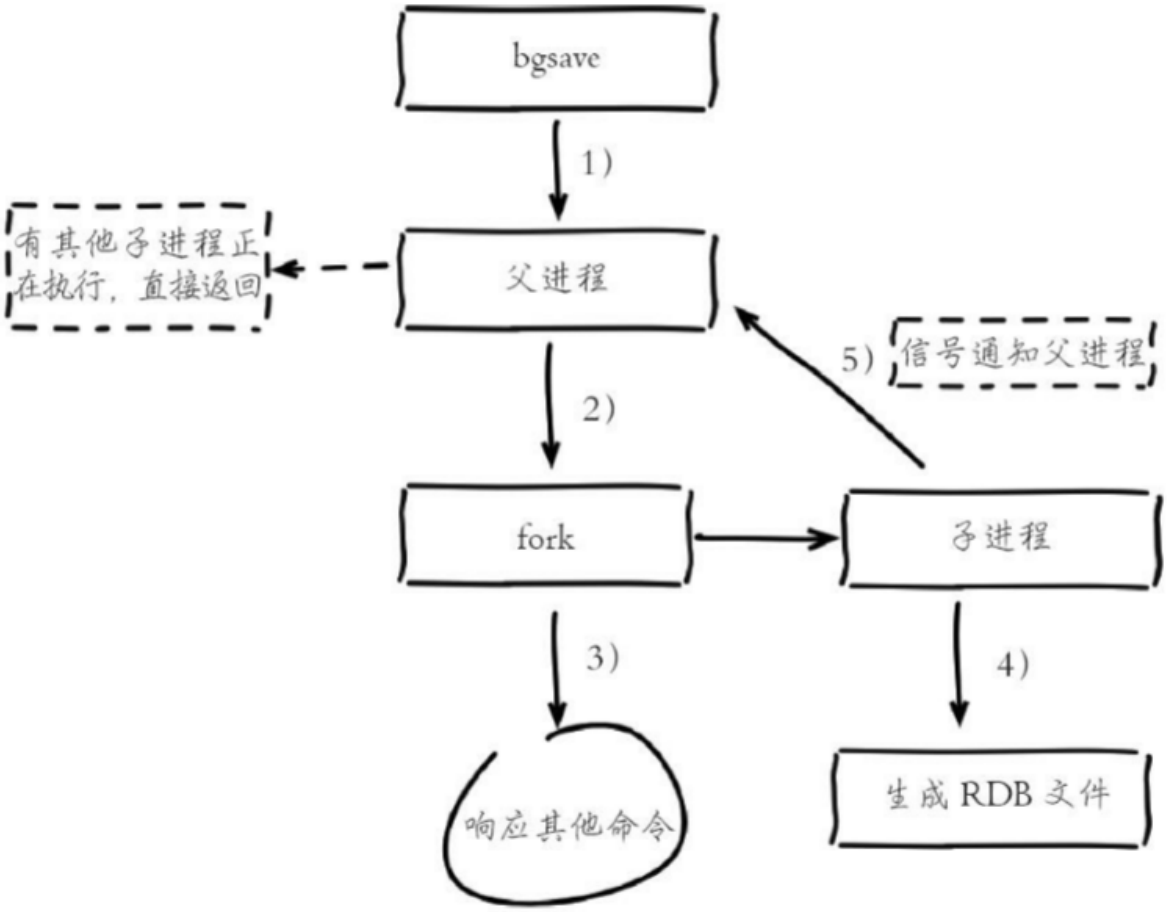


图5-1　bgsave命令的运作流程

1）执行bgsave命令，Redis父进程判断当前是否存在正在执行的子进程，如RDB/AOF子进程，如果存在bgsave命令直接返回。

2）父进程执行fork操作创建子进程，Fork操作过程中父进程会阻塞，通过info stats命令查看latest_fork_usec选项，可以获取最近一个fork操作的耗时，单位为微秒。

3）父进程fork完成后，bgsave命令返回“Background saving started”信息并不再阻塞父进程，可以继续响应其他命令。

4）子进程创建RDB文件，根据父进程内存生成临时快照文件，完成后对原有文件进行原子替换。执行lastsave命令可以获取最后一次生成RDB的时间，对应info统计的rdb_last_save_time，存储的是时间戳。

当遇到坏盘或磁盘写满情况时，可以通过config set dir {newDir}在线修改文件路径到可用磁盘路径，之后执行bgsave进行磁盘切换，同样适用于AOF持久化文件。Redis默认使用LZF算法对生成的RDB文件做压缩处理，压缩后的文件远远小于内存大小，默认开启，可以通过参数config set rdbcompression {yes|no}动态修改。虽然压缩RDB会消耗CPU，但可大幅度降低文件的体积，方便保存到硬盘或通过网络发送给从节点，

RDB的优点：

- RDB是一个紧凑压缩的二进制文件，代表Redis在某个时间点上的数据快照。非常适用于备份，全量复制等场景。比如每6小时执行bgsave备份，并把RDB文件拷贝到远程机器或文件系统中（如HDFS），用于灾难恢复。
- Redis加载RDB恢复数据源远快于AOF的方式

RDB的缺点：

- RDB方式数据没办法做到实时持久化/秒级持久化。因为bgsave每次运行都要执行fork操作创建子进程，属于重量级操作，频繁执行成本过高。
- RDB文件使用特定二进制格式保存，Redis版本演进过程中有多个格式的RDB版本，存在老版本Redis服务无法兼容新版本RDB格式的问题。

2.AOF

针对RDB不适合实时持久化的问题，Redis提供了AOF持久化方式来解决。

AOF（append only file）持久化：以独立日志的方式记录每次写命令，重启时再重新执行AOF文件中的命令达到回复数据的目的。AOF的主要作用是解决了数据持久化的实时性，目前已经是Redis持久化的主流方式。

开启AOF功能需要设置配置：appendonly yes，默认不开启。AOF文件名通过appendfilename配置设置，默认文件名是appendonly.aof。保存路径同RDB持久化方式一致，通过dir配置指定。AOF的工作流程操作：命令写入（append）、文件同步（sync）、文件重写（rewrite）、重启加载（load），具体如下图：

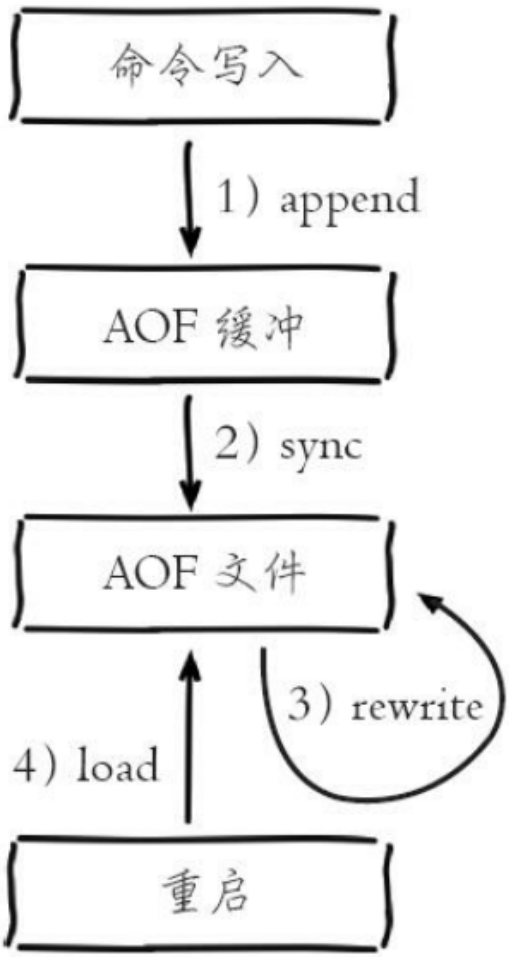


图5-2　AOF工作流程

流程如下：

1）所有的写入命令会追加到aof_buf（缓冲区）中。

2）AOF缓冲区根据对应的策略向硬盘同步操作。

3）随着AOF文件越来越大，需要定期对AOF文件进行重写，达到压缩的目的。

4）当Redis服务器重启时，可以加载AOF文件进行数据恢复。

AOF命令写入的内容直接是文本协议格式。例如set hello world这条命令，在AOF缓冲区会追如下文本：

```
*2\r\n$6\r\nSELECT\r\n$1\r\n0*3\r\n$3\r\nset\r\n$5\r\nhello\r\n$5\r\nworld\r\n
```

为什么AOF直接采用文本协议格式?

- 文本协议具有很好的兼容性
- 开启AOF后，所有的写入命令都包含追加操作，直接采用协议格式，避免了二次处理开销
- 文本协议具有可读性，方便直接修改和处理

AOF为什么把命令追加到aof_buf中?

Redis使用单线程响应命令，如果每次写入AOF文件命令都直接追加到硬盘，那么性能完全取决于当前硬盘负载。先写入缓冲区aof_buf中，还有一个好处，Redis可以提供多种缓冲区同步硬盘的策略，在性能和安全性方面做出平衡。

Redis提供了多种AOF缓冲区同步文件策略，由参数appendfsync控制，其含义如下表：

表5-1　AOF缓冲区同步文件策略

可配置值	说　明
always	命令写入 aof_buf 后调用系统 fsync 操作同步到 AOF 文件，fsync 完成后线程返回
everysec	命令写入 aof_buf 后调用系统 write 操作，write 完成后线程返回。fsync 同步文件操作由专门线程每秒调用一次
no	命令写入 aof_buf 后调用系统 write 操作，不对 AOF 文件做 fsync 同步，同步硬盘操作由操作系统负责，通常同步周期最长 30 秒

- write操作会出发延迟写（delayed write）机制。Linux在内核提供页缓冲区用来提高硬盘IO性能。write操作在写入系统缓冲区后直接返回。同步硬盘操作依赖于系统调度机制，例如：缓冲区页空间写满或达到特定时间周期。同步文件之前，如果此时系统故障宕机，缓冲区内数据将丢失。
- fsync针对单个文件操作（比如AOF文件），做强制硬盘同步，fsync将阻塞直到写入硬盘完成后返回，保证了数据持久化。

除了write、fsync，Linux还提供了sync、fdatasync操作。

可配置值的解释：

- always：配置always时，每次写入都要同步AOF文件，在一般的SATA硬盘上，Redis只能支持大约几百TPS（吞吐量）写入，显然根Redis高性能特性背道而驰，不建议配置。
- no：由于操作系统每次同步AOF文件的周期不可控，而且会加大每次同步硬盘的数据量，虽然提升了性能，但数据安全性无法保证。
- everysec：是建议的同步策略，也是默认配置，做到兼顾性能和数据安全性。理论上只有在系统宕机的情况下丢失1秒的数据。

随着命令不断写入AOF，文件会越来越大，为了解决这个问题，Redis引入AOF重写机制压缩文件体积。AOF文件重写是把Redis进程内的数据转化为写命令同步到新AOF文件的过程。

重写后的AOF文件为什么可以变小？有如下原因：

- 进程内已经超时的数据不再写入文件
- 旧的AOF文件含有无效命令，如del key1、hdel key2、srem keys、set a111、set a222等。重写使用进程内数据直接生成，这样新的AOF文件只保留最终数据的写入命令。
- 多条写命令可以合并为一个，如：lpush list a、lpush list b、lpush list c可以转化为：lpush list a b c。为了防止单条命令过大造成客户端缓冲区溢出，对于list、set、hash、zset等类型操作，以64个元素为界拆分为多条。

AOF重写降低了文件占用空间，除此之外，另一个目的是：更小的AOF文件可以更快地被Redis加载。

AOF重写

AOF重写过程可以手动触发和自动触发：

- 手动触发：直接调用bgrewriteaof命令。
- 自动触发：根据auto-aof-rewrite-min-size和auto-aof-rewrite-percentage参数确定自动触发时机。
    - auto-aof-rewrite-min-size：表示运行AOF重写时文件最小体积，默认为64MB。
    - auto-aof-rewrite-percentage：代表当前AOF文件空间（aof_current_size）和上一次重写后AOF文件空间（aof_base_size）的比值。
    - 自动触发时机=aof_current_size > auto-aof-rewrite-min-size && (aof_current_size - aof_base_size) / aof_base_size >= auto-aof-rewrite-percentage

  其中aof_current_size和aof_base_size可以在info persistence统计信息中查看。

当触发AOF重写时，内部做了哪些事，如下图：

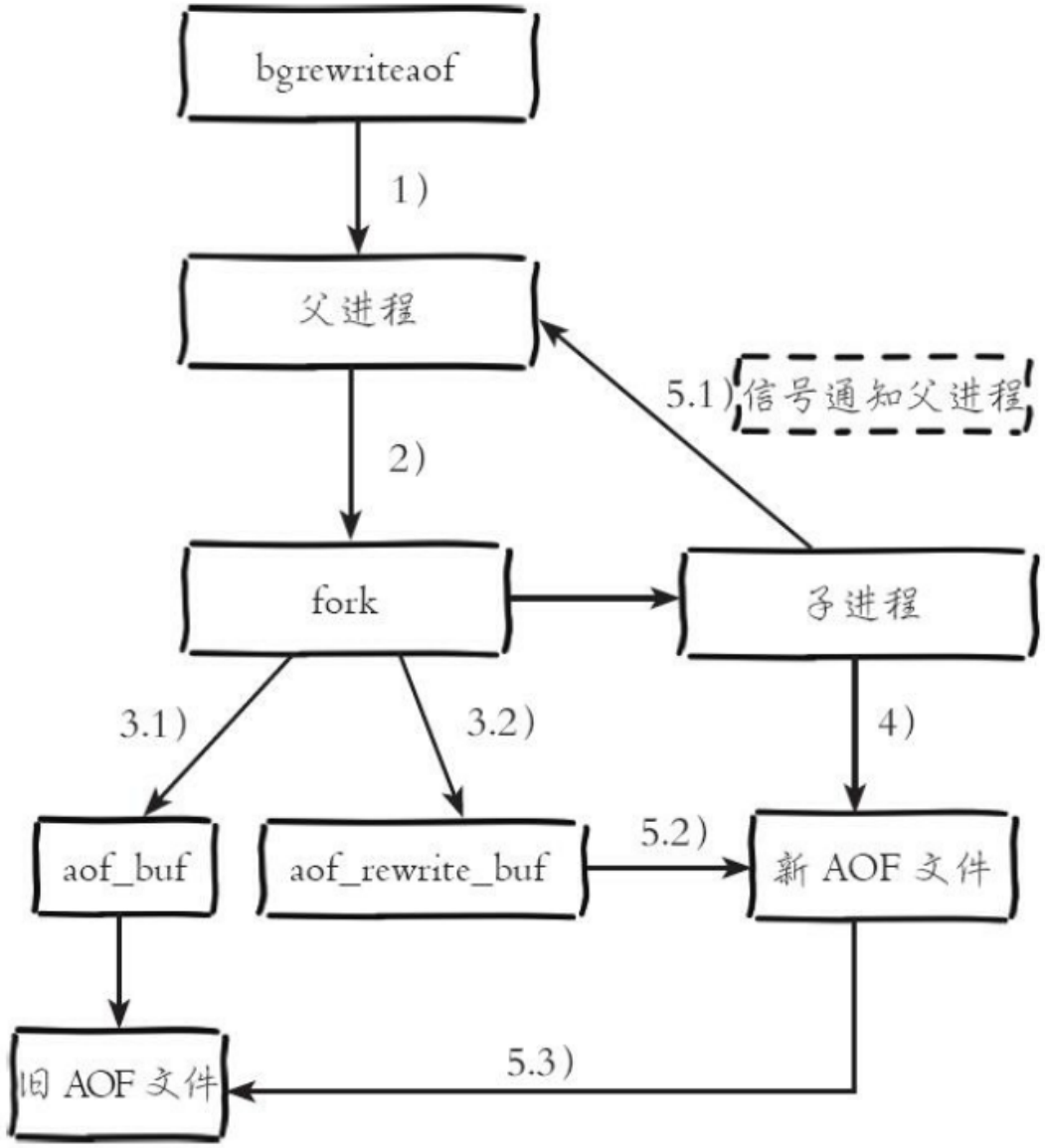


图5-3　AOF重写运作流程

1）执行AOF重写请求。

如果当前进程正在执行AOF重写，请求不执行并返回如下响应：

```
ERR Background append only file rewriting already in progress
```

如果当前进程正在执行bgsave操作，重写命令延迟到bgsave完成之后再执行，返回如下响应：

```
Background append only file rewriting scheduled
```

2）父进程执行fork创建子进程，开销等同于bgsave过程。

3.1）主进程fork操作完成后，继续响应其他命令。所有修改命令依然写入AOF缓冲区并根据appendfsync策略同步到硬盘，保证原有AOF机制正确性。

3.2）由于fork操作运用写时复制技术，子进程只能共享fork操作时的内存数据。由于父进程依然响应命令，Redis使用“AOF重写缓冲区”保存这部分新数据，防止新AOF文件生成区间丢失这部分数据。

4）子进程根据内存快照，按照命令合并规则写入到新的AOF文件。每次批量写入硬盘数据量由配置aof-rewrite-incremental-fsync控制，默认为32MB，防止单词刷盘数据过多造成硬盘阻塞。

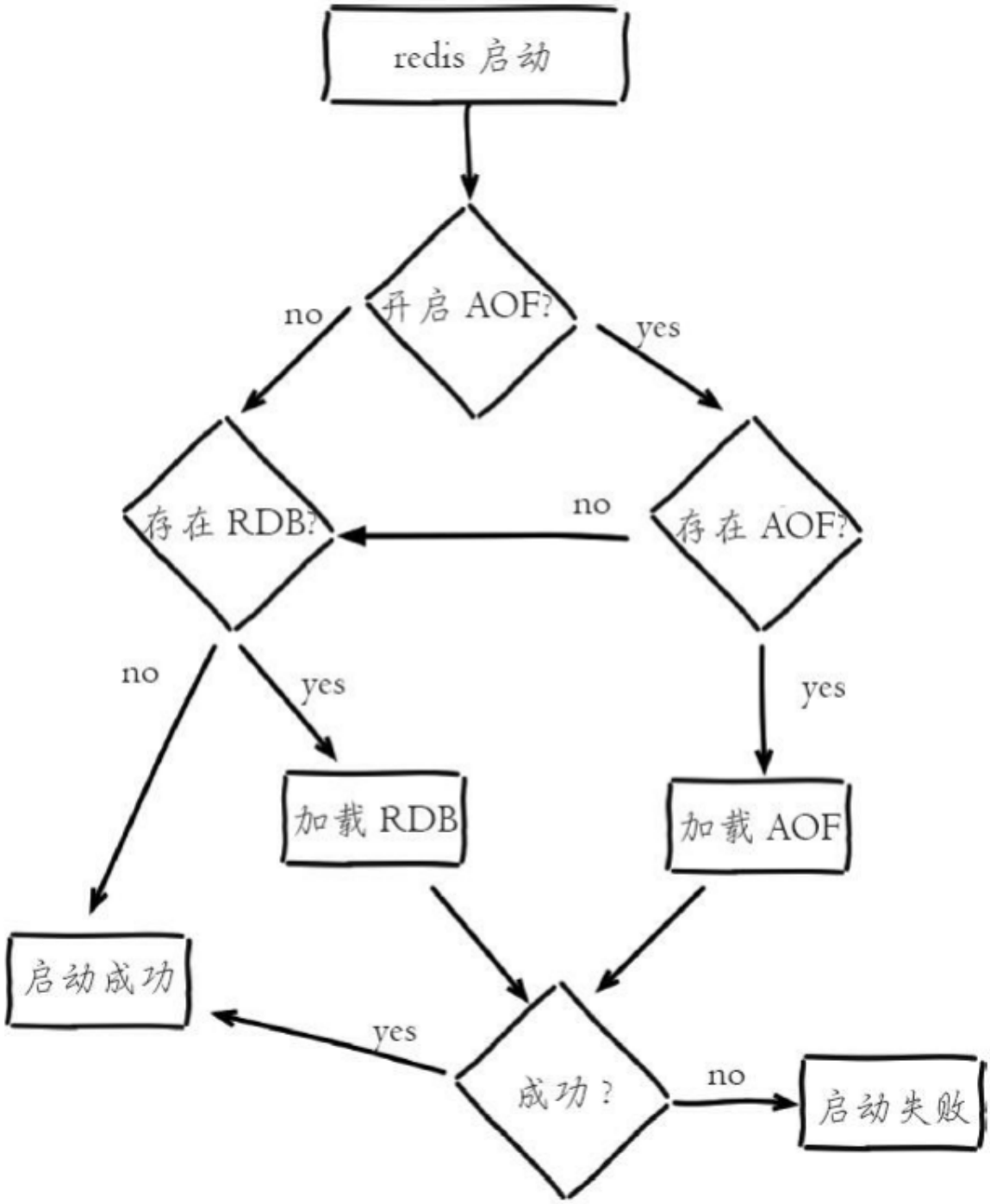
5.1）新AOF文件写入完成后，子进程发送信号给父进程，父进程更新统计信息，具体见info persistence 下的aof_*相关统计。

5.2）父进程把AOF重写缓冲区的数据写入到新的AOF文件。

5.3）使用新AOF文件替换老文件，完成AOF重写。

3.重启加载

AOF和RDB文件都可以用于服务器重启时的数据恢复，加载流程如下：



九、Redis数据淘汰策略

可通过执行info memory命令获取内存相关指标。

属性名	属性说明
used_memory	Redis分配器分配的内存总量，也就是内部存储的所有数据内存占用量
used_memory_human	以可读的格式返回used_memory
used_memory_rss	从操作系统角度显示Redis进程占用的物理内存总量
used_memory_peak	内存使用的最大值，表示used_memory的峰值
used_memory_peak_human	以可读的格式返回used_memory_peak
used_memory_lua	Lua引擎所消耗的内存大小
mem_fragmentation_ratio	used_memory_rss/used_memory比值，表示内存碎片率
mem_allocator	Redis所使用的内存分配器。默认为jemalloc

需要重点关注的指标有：used_memory_rss和used_memory以及他们的比值mem_fragmentation_ratio。

当 mem_fragmentation_ratio>1 时，说明used_memory_rss - used_memory多出的部分内存并没有与用于数据存储，而是被内存碎片所消耗，如果两者相差很大，说明碎片率严重。

当 mem_fragmentation_ratio<1时，这种情况一般出现在操作系统把Redis内存交换（swap）到硬盘所致，出现这种情况时需要格外关注，由于硬盘速度远远慢于内存，Redis性能会变得很差，甚至僵死。

1.内存消耗划分

Redis进程内消耗主要包括：自身内存+对象内存+缓冲内存+内存碎片。其中Redis空进程自身内存消耗非常少，通常used_memory_rss在3MB左右，used_memory在800KB左右，一个空的Redis进程消耗内存可以忽略不计。

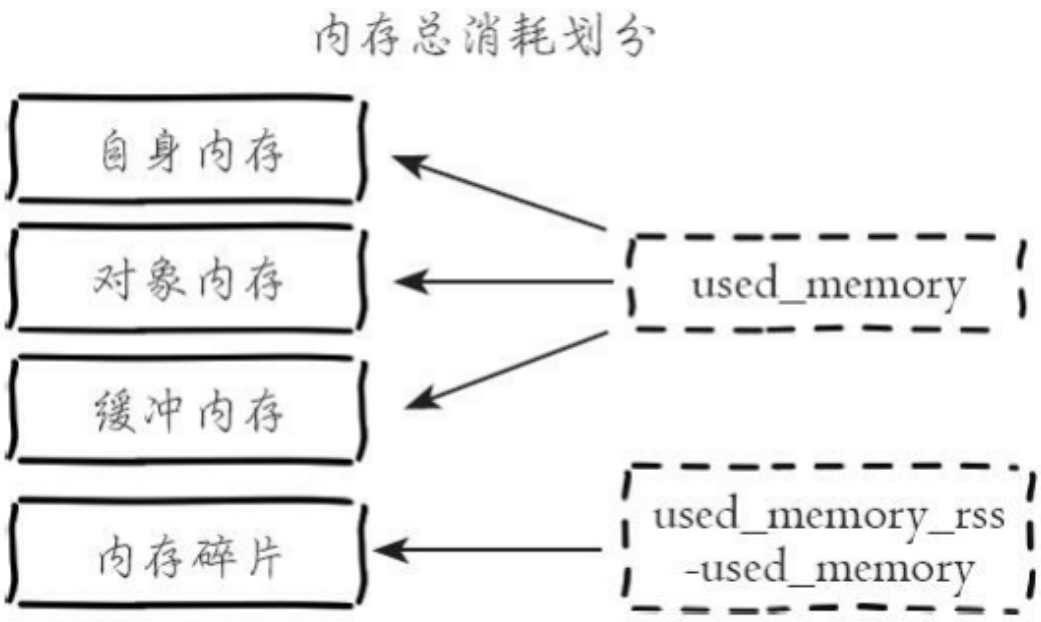


图8-1　Redis内存消耗划分

（1）对象内存

对象内存是Redis内存占用最大的一块，存储着用户所有的数据。Redis所有的数据都采用key-value数据类型，每次创建键值对时，至少创建两个类型对象：key对象和value对象。对象内存消耗可以简单理解为sizeof（keys）+sizeof（values）。键对象都是字符串，在使用Redis时很容易忽略键对内存消耗的影响，应当避免使用过长的键。value对象更复杂些，主要包括5种基本数据类型：string、list、hash、set、zset。每种value对象根据使用规模不同，占用内存不同。在使用时一定要合理预估并监控value对象占用情况，避免内存溢出。

（2）缓冲内存

缓冲内存主要包括：客户端缓冲、复制积压缓冲区、AOF缓冲区。

客户端缓冲指的是所有接入到Redis服务器TCP连接的输入输出缓冲。输入缓冲无法控制，最大空间为1G，如果超过将断开连接。输出缓冲通过参数client-output-buffer-limit控制，如下所示：

- 普通客户端：除了复制和订阅的客户端之外的所有连接，Redis默认配置是：client-output-bufferlimit normal000，Redis并没有对普通客户端的输出缓冲区做限制，一般普通客户端的内存消耗可以忽略不计，但是当有大量慢连接接入时这部分内存消耗就不能忽略了，可以设置maxclients做限制。特别是当使用大量数据输出的命令且数据无法及时推送给客户端时，如monitor命令，容易造成Redis服务器内存突然飙升。
- 从客户端：主节点会为每个从节点单独简历一条连接用于命令复制，默认配置是：client-output-buffer-limit slave256mb64mb64。当主从节点之间网络延迟较高或主节点挂在大量从节点时这部分内存消耗将占用很大一部分，建议主节点挂载的从节点不要多于2个，主节点不要部署在较差的网络环境下，如异地跨机房环境，防止复制客户端连接缓慢造成溢出。
- 订阅客户端：当使用发布订阅功能时，连接客户端使用单独的输出缓冲区，默认配置是client-output-buffer-limit pubsub32mb8mb60，当订阅服务的消息生产快于消费速度时，输出缓冲区会产生积压造成输出缓冲区空间溢出。

输入输出缓冲区在大流量的场景中容易失控，造成Redis内存的不稳定，需要重点监控。

复制积压缓冲区：Redis在2.8版本之后提供了一个可重用的固定大小缓冲区用于实现部分复制功能，根据repl-backlog-size参数控制，默认1MB。对于复制积压缓冲区整个主节点只有一个，所有的从节点共享此缓冲区，因此可以设置较大的缓冲区空间，如100MB，这部分内存投入是有价值的，可以有效避免全量复制。

AOF缓冲区：这部分空间用于在Redis重写期间保存最近的写入命令。AOF缓冲区空间消耗用户无法控制，消耗的内存取决于AOF重写时间和写入命令量，这部分空间占用通常很小。

（3）内存碎片

Redis默认的内存分配器采用jemalloc，可选的分配器还有：glibc、tcmalloc。内存分配器为了更好地管理和重复利用内存，分配内存策略一般采用固定范围的内存块进行分配。例如jemalloc在64位系统中将内存空间划分为：小、大、巨大三个范围。每个范围又划分为多个小的内存块单位，如下所示：

- 小：[8 byte], [16 byte, 32 byte, 48 byte, .. , 128 byte], [192 byte, 256 byte, ..., 512 byte], [768 byte, 1024 byte, ..., 3840 byte]
- 大：[4 KB, 8 KB, 12 KB, ... , 4072 KB]
- 巨大：[4 MB, 8 MB, 12 MB, ...]

比如保存5KB对象时jemalloc可能会采用8KB的块存储，而剩下的3KB空间变为了内存碎片不能再分配给其他对象存储。内存碎片问题虽然是所有内存服务的通病，但是jemalloc针对碎片化问题专门做了优化，一般不会存在过度碎片化的问题，正常的碎片率（mem_fragmentation_ratio）在1.03左右。但是当存储的数据长短差异较大时，以下场景容易出现高内存碎片问题：

- 频繁做更新操作，例如频繁对已存在的键执行append、setrange等更新操作。
- 大量过期键删除，键对象过期删除后，释放的空间无法得到充分利用，导致碎片率上升。

出现高内存碎片问题时常见的解决方式如下：

- 数据对齐：在条件允许的情况下尽量做数据对齐，比如数据尽量采用数字类型或者固定长度字符串等，但这要视具体的业务而定，有些场景无法做到。
- 安全重启：重启节点可以做到内存碎片重新整理，因此可以利用高可用架构，如Sentinel或Cluster，将碎片率过高的主节点转换为从节点，进行安全重启。

（4）子进程内存消耗

子进程内存消耗主要执行AOF/RDB重写时Redis创建的子进程内存消耗。Redis执行fork操作产生的子进程内存占用量对外表现为与父进程相同，理论上需要一倍的物理内存来完成重写操作。但Linux具有写时复制技术（copy-on-write），父子进程会共享相同的物理内存页，当父进程处理写请求时会对需要修改的页面复制出一份副本完成写操作，而子进程依然读取fork时整个父进程的内存快照。

2.内存管理

（1）设置内存上限

Redis使用maxmemory参数限制最大可用内存。限制内存的目的主要有：

- 用于缓存场景，当超出内存上限maxmemory时使用LRU等删除策略释放空间。
- 防止所用内存超过物理服务器内存。

需要注意，maxmemory限制的是Redis实际使用的内存量，也就是used_memory统计项对应的内存。由于内存碎片率的存在，实际消耗的内存可能比maxmemory设置的更大，实际使用时要小心这部分内存溢出。通过设置内存上线可以非常方便地实现一台服务器部署多个Redis进程的内存控制。比如一台24GB内存的服务器，为系统预留4GB内存，预留4GB空闲内存给其他进程或Redis fork进程，留给Redis 16GB内存，这样可以部署4个maxmemory=4GB的Redis进程。得益于Redis单线程架构和内存限制机制，即使没有采用虚拟化，不同的Redis进程之间也可以很好地实现CPU和内存的隔离性。

Redis的内存上限可以通过config set maxmemory进行动态修改，即修改最大可用内存。Redis默认无线使用服务器内存，为防止极端情况下导致系统内存耗尽，建议所有的Redis进程都要配置maxmemory。

（2）内存回收策略

Redis的内存回收机制主要体现在以下两个方面：

- 删除到达过期时间的键对象
- 内存使用达到maxmemory上限时触发内存溢出控制策略

删除过期键对象：Redis所有的键都可以设置过期属性，内部保存在过期字典中。由于进程内保存大量的键，维护每个键精准的过期删除机制会导致消耗大量的CPU，对于单进程的Redis来说成本过高，因此Redis采用惰性删除和定时任务删除机制实现过期键的内存回收。

- 惰性删除：惰性删除用于当客户端读取带有超时属性的键时，如果已经超过键设置的过期时间，会执行删除操作并返回空，这种策略是出于节省CPU成本考虑，不需要单独维护TTL链表来处理过期键的删除。但是单独用这种方式存在内存泄漏的问题，当过期键一直没有访问将无法得到及时删除，从而导致内存不能及时释放。正因为如此，Redis提供另一种定时任务删除机制作为惰性删除的补充。
- 定时任务删除：Redis内部维护一个定时任务，默认每秒运行10次（通过配置hz控制）。定时任务中删除过期键逻辑采用了自适应算法，根据键的过期比例、使用快慢两种速率模式回收键。流程如下：

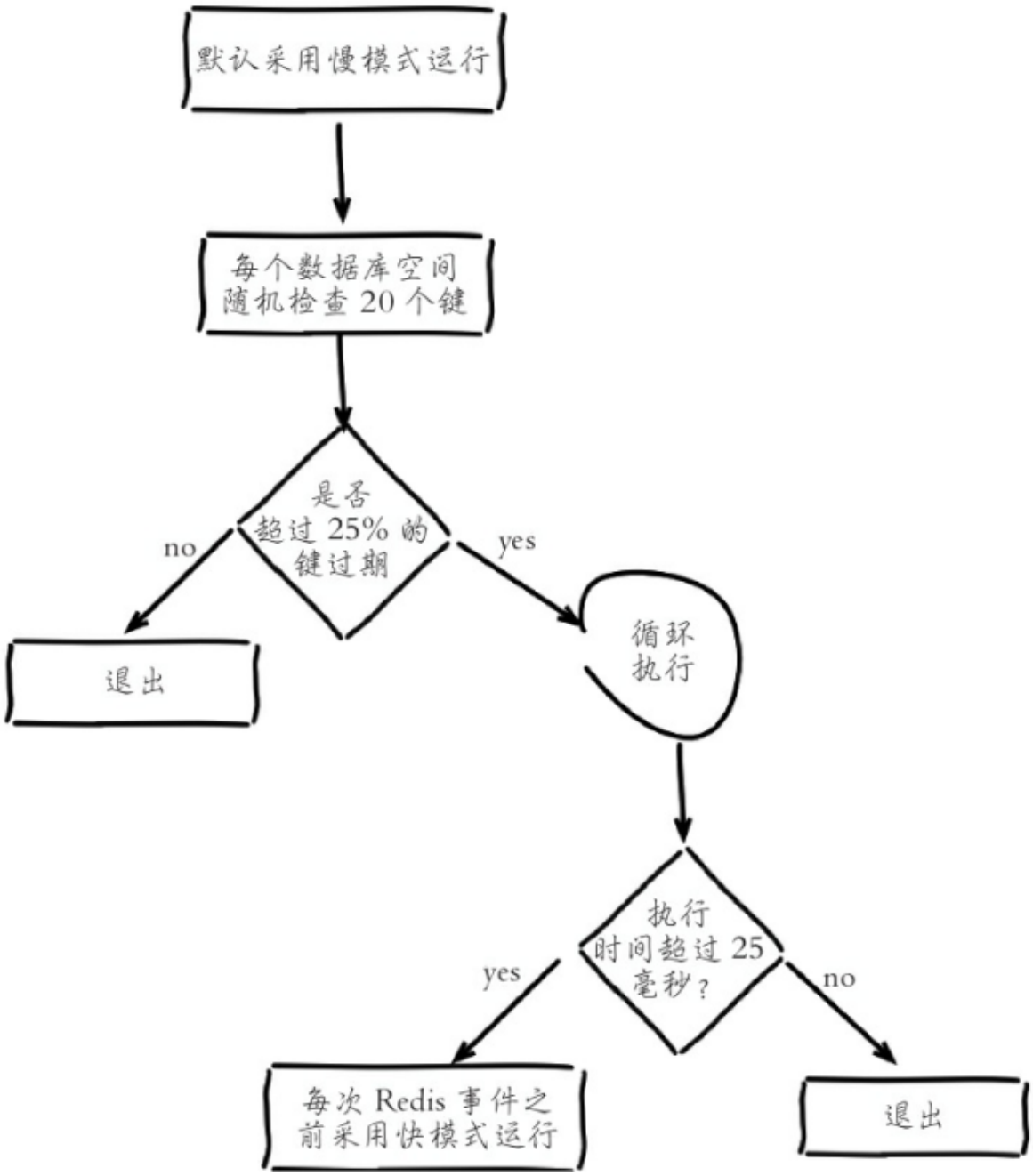


图8-4　定时任务删除过期键逻辑

流程说明：

1）定时任务在每个数据库空间随机检查20个键，当发现过期时删除对应的键。

2）如果超过检查数25%的检查过期，循序安执行回收逻辑直到不足25%或运行超时为止，慢模式下超时时间为25毫秒。

3）如果之前回收键逻辑超时，则在Redis触发内部事件之前再次以快模式运行回收过期键任务，快模式下超时时间为1毫秒且2秒内只能运行1次。

（3）内存溢出控制策略

当Redis所用内存达到maxmemory上限时会触发相应的溢出控制策略。具体策略受maxmemory-policy参数控制，Redis支持6种策略，如下所示：

1）noeviction：默认策略，不会删除任何数据，拒绝所有写入操作并返回客户端错误信息（error）OOM command not allowed when used memory，此时Redis只响应读操作。

2）volatile-lru：根据LRU算法删除设置了超时属性（expire）的键，直到腾出足够空间为止。如果没有可删除的键对象，回退到noveiction策略。

3）allkeys-lru：根据LRU算法删除键，不管数据有没有设置超时属性，直到腾出足够空间为止。

4）allkeys-random：随机删除所有键，直到腾出足够空间为止。

5）volatile-random：随机删除过期键，直到腾出足够空间为止。

6）volatile-ttl：根据键值对象的ttl属性，删除最近将要过期数据。如果没有，回退到noeviction策略。

内存溢出控制策略可以采用config set maxmemory-policy{policy}动态配置。可以根据实际需求灵活定制。如当设置volatile-lru策略时，保证具有过期属性的键可以根据LRU提出，而未设置超时的键可以永久保留。还可以采用allkeys-lru策略把Redis变为纯缓存服务器使用。当Redis因为内存溢出删除键时，可以通过执行info stats命令查看evicted_keys指标找出当前Redis服务器已踢出的键数量。

每次Redis执行命令时如果设置了maxmemory参数，都会尝试执行回收内存操作。当Redis一直工作在内存溢出（used_memory > maxmemory）的状态下且设置非noeviction策略时，会频繁地触发回收内存的操作，影响Redis服务器的性能。

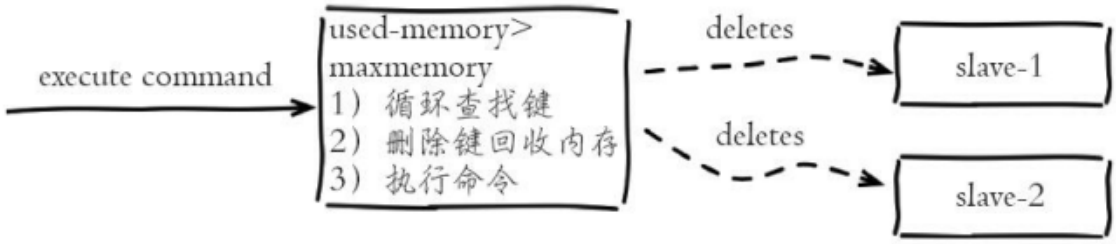


图8-5　写入数据触发内存回收操作

（4）redisObject对象

Redis存储的所有值对象在内部定义为redisObject结构体，内部结构如图：

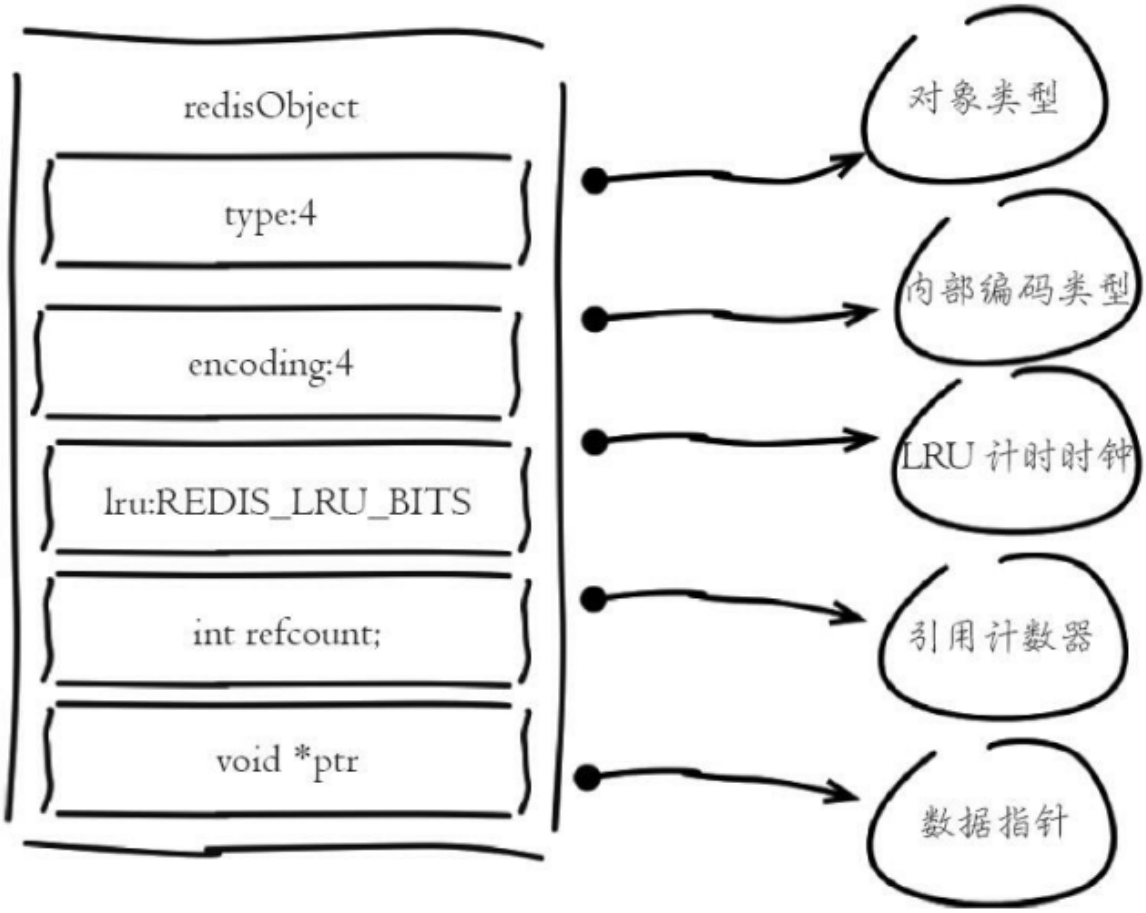


图8-6　redisObject内部结构

Redis存储的数据都使用redisObject来封装，包括string、hash、list、set、zset在内的所有数据类型。理解redisObject对内存优化非常有帮助，详细说明如下：

- type字段：表示当前对象使用的数据类型，Redis主要支持5种数据类型：string、hash、list、set、zset。可以使用type{key}命令查看对象所属类型，type返回的是值对象类型，键都是string类型。
- encoding字段：表示Redis内部编码类型，encoding在Redis内部使用，表示当前对象内部采用哪种数据结构实现。理解Redis内部编码方式对于优化内存非常重要，同一个对象采用不同的编码实现内存占用存在差异。
- lru字段：记录对象最后一次被访问的时间，当配置了maxmemory和maxmemory-policy=volatile-lru或者allkeys-lru时，用于辅助LRU算法算出键数据。可以使用object idletime{key}命令在不更新lru字段情况下查看当前键的空闲时间。
- refcount字段：记录当前对象被引用的次数，用于通过引用次数回收内存，当refcount=0时，可以安全回收当前对象空间。使用object refcount {key}获取对象引用。当对象为整数且范围在[0-9999]时，Redis可以使用共享对象的方式节省内存。
- *ptr字段：与对象的数据内容相关，如果是整数，直接存储数据；否则表示指向数据的指针。Redis在3.0之后对值对象是字符串且长度<=39字节的数据，内部编码为embstr类型，字符串sds和redisObject一起分配，从而只要一次内存操作即可。

（5）缩减键值对象

降低Redis内存使用最直接的方式就是缩减键（key）和值（value）的长度。

- key长度：在完整描述业务情况下，键值越短越好。
- value长度：值对象缩减比较复杂，常见需求是把业务对象序列化成二进制数据存放如Redis。首先应该在业务上精简业务对象，去掉不必要的属性避免存储无效数据。其次在序列化工具上，应该选择更高效的序列化工具来降低字节数据大小。以Java为

例，内置的序列化方式无论从速度还是压缩比都不尽如人意，这时可以选择更高效的序列化工具，如：protostuff、kryo等。

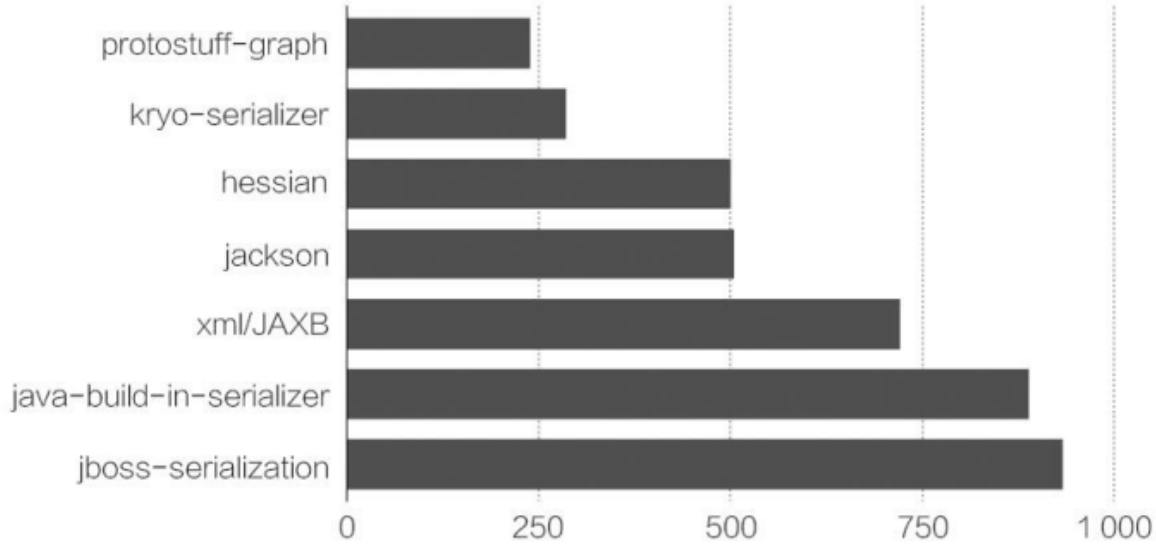


图8-7　Java常见序列化组件占用内存空间对比（单位字节）

（6）对象共享池

共享对象池是指Redis内部维护[0-9999]的整数对象池。创建大量的整数类型redisObject存在内存开销，每个redisObject内部结构至少占16字节，甚至超过了整数自身空间消耗。所以Redis内存维护了一个[0-9999]的整数对象池，用于节约内存。除了整数值对象，其他类型如list、hash、set、zset内部元素也可以使用整数对象池。因此开发中在满足需求的前提下，尽量使用整数对象节省内存。

整数对象池在Redis中通过变量REDIS_SHARED_INTEGERS定义，不能通过配置修改。可以通过object refcount命令查看对象引用数验证是否启用整数对象池技术。

开启maxmemory和LRU淘汰策略后对象池失效。LRU算法获取对象最后被访问时间，以便淘汰最长未访问数据，每个对象最后访问时间存储在redisObject对象的lru字段。对象共享意味着多个引用共享同一个redisObject，这时lru字段也会被共享，导致无法获取每个对象的最后访问时间。如果没有设置maxmemory，直到内存被用尽Redis也不会触发内存回收，所以共享对象池可以正常工作。

综上所述，共享对象池与maxmemory+LRU策略冲突，使用时需要注意。

十、物理内存和虚拟内存