Redis的单机性能优化

2017-06-30 | Redis |

在《Redis拾遗》中提到`查询本地redis的延迟通常低于1毫秒，而查询同一个数据中心的redis的延迟通常低于5毫秒`。也就是说，网络传输的损耗为实际操作用时的5倍。

使用Pipeline或Lua脚本可以在一次请求中发送多条命令，通过分摊一次请求的网络及系统延迟，从而可以极大的提高性能。

当数据量过大，单台机器的内存不足时，往往首先考虑的不应该是集群扩展，而是内存优化。《Redis in Action》的作者曾经就有过将70多GB的数据缩小至3GB的壮举。内存占用小了，单台机器可以存储的数据就多了，这样也避免了集群环境中很多不必要的损耗与复杂性。

使用Pipeline与Lua脚本提高并发

网络延时

客户端与服务器都是通过网络连接，无论是本机的Loopback网络，容器或虚拟机的bridge网络，局域网还是广域网，都会有不同程度的延迟。一般将请求从客户端发出，到达服务器，再返回客户端的时间总和叫做RTT（Round Trip Time）。这个延迟，使用ping命令就可以大概测出来，下边是我做的一个简单的测试：

网络类型	RTT
本机Loopback网络	0.05ms
Docker Bridge网络	0.2ms
局域网	2ms
国内门户	25ms
国外门户	400ms

在一个网络状况较恶劣的情景下，假设RTT为250毫秒，就算服务器本身每秒可以处理100K的请求，但是这种网络条件下，对于单个客户端，服务器每秒最多只能处理4个请求。

可以用一个公式更直观的说明这个问题：网络延时为`L（毫秒）`，Redis处理请求的时间`R（毫秒）`，Redis客户端一次请求携带的命令总数`n`，则RTT为`L + R`。忽略请求数据量大小的差异，则Redis每秒可以处理的请求数量就为`n * 1000 / (L + R * n)`。其中，`L`与`R`都是常量，在`L`远大于`R`的情况下，增大`n`能够提高并发，在`L`小于或接近`R`的情况下，增大`n`并不会有太大的差别。

操作系统

另一个需要考虑的因素是操作系统对Socket I/O的实现机制。每一次网络请求都涉及`read()`与`write()`系统调用，会发生用户空间与内核空间上下文的切换，对于很多简单Redis操作来说，这个损耗相当昂贵。而将多条命令一次处理，Redis往往会合并进行一次系统调用。这种性能的提升是线性的，最多能达到原来10倍的处理速度。上边的函数可以改写为`n * 1000 / (L + R * log(n))`（使用log函数只是方便表达Redis的批处理时间比单独处理时间总和要小，并不能用于计算），这样不论在何种网络环境下，一次处理的命令越多，效率越高。

测试与结论

下边是在我的开发机器，使用redis-benchmark对Pipeline性能的一个简单测试。

本机Loopback网络:

```
> redis-benchmark -q -n 100000 -t get,set
SET: 74962.52 requests per second
GET: 76982.29 requests per second

> redis-benchmark -q -n 100000 -t get,set -P 5
SET: 253164.55 requests per second
GET: 288184.44 requests per second

> redis-benchmark -q -n 100000 -t get,set -P 10
SET: 362318.84 requests per second
GET: 456621.00 requests per second

> redis-benchmark -q -n 100000 -t get,set -P 20
SET: 487804.88 requests per second
GET: 641025.62 requests per second

> redis-benchmark -q -n 100000 -t get,set -P 50
SET: 591716.00 requests per second
GET: 884955.75 requests per second

> redis-benchmark -q -n 100000 -t get,set -P 100
```

```
SET: 657894.75 requests per second
GET: 980392.19 requests per second
```

局域网:

```
> redis-benchmark -q -n 100000 -t get,set -h 192.168.0.141
SET: 9804.88 requests per second
GET: 9636.70 requests per second


> redis-benchmark -q -n 100000 -t get,set -h 192.168.0.141 -P 5
SET: 48661.80 requests per second
GET: 47732.70 requests per second

> redis-benchmark -q -n 100000 -t get,set -h 192.168.0.141 -P 10
SET: 94876.66 requests per second
GET: 92506.94 requests per second

> redis-benchmark -q -n 100000 -t get,set -h 192.168.0.141 -P 20
SET: 167785.23 requests per second
GET: 179211.45 requests per second

> redis-benchmark -q -n 100000 -t get,set -h 192.168.0.141 -P 50
SET: 217391.30 requests per second
GET: 261096.61 requests per second

> redis-benchmark -q -n 100000 -t get,set -h 192.168.0.141 -P 100
```

```
SET: 290697.66 requests per second
GET: 344827.59 requests per second
```

在本地这样低延迟的网络环境下，使用Pipeline，每次发送5条命令比不使用Pipeline快了3倍多，每次发送10条命令快了5倍多，而局域网的提升要更明显一些。不论何种网络环境，性能的提升基本上是线性的。

在Redis中，一次请求运行多条命令的方式有两个，Pipeline与Lua脚本。两者的区别是，Lua脚本可以以最小的延时同时读写数据，但是Pipeline的客户端需要等待服务器的读取结果才能继续下边的写入操作。如果请求里边包含大量服务端的计算，一些中间变量只用于计算不需要传回客户端，Lua脚本的优势更加明显。

内存优化

Redis的内存优化并不是免费的，一般是以牺牲性能为代价的，同时也可能会引入实现上额外的复杂性。因此，对Redis的内存进行优化，做何种程度的优化，需要权衡数据量的大小、硬件成本、网络环境、业务实现以及性能等多方面因素。每个项目面对的问题都各不相同，很难提供一个优化的共同准则。

内存分配

在谈论内存优化之前，有必要对Redis的内存分配方式进行简单的说明。

Redis的内存使用情况可以在`redis-cli`中使用命令`info`查看：

```
# Memory
used_memory:1007440                     # 使用内存
used_memory_human:983.83K
used_memory_rss:2555904                 # 占用内存
used_memory_rss_human:2.44M
used_memory_peak:3486736                # 峰值内存
used_memory_peak_human:3.33M
total_system_memory:8589934592          # 操作系统内存
total_system_memory_human:8.00G
used_memory_lua:37888                   # lua脚本占用内存
used_memory_lua_human:37.00K
maxmemory:0                             # 配置文件设定的内存上限
maxmemory_human:0B
maxmemory_policy:noeviction             # 内存超限时的释放空间策略
mem_fragmentation_ratio:2.54            # 内存碎片率（used_memory_rss / used_memory
mem_allocator:libc                      # 内存分配器
```

当从Redis中移除数据的时候，Redis并不总会将释放的内存还给操作系统。例如，对于一个有5GB数据的Redis实例，如果删除掉2GB的数据，Redis报告的使用内存（`used_memory`）是3GB，但是占用的内存（`used_memory_rss`)差不多还是5GB。这其实不是Redis故意为之，而是`malloc()`方法的实现机制，因为删除掉的数据可能与其他正常数据在同一个内存分页中，因此这些分页就无法被释放掉。所以在给Redis分配内存时，需要参考峰值内存（`used_memory_peak`）而不是实际使用的内存。例如，一个Redis实例工作时最多需要10GB的内存，但是大多数时候5GB内存就够用了，但是，仍然要为其预留10GB的内存。

这些无法还给操作系统的内存并不会浪费掉，当有新数据写入的时候，Redis会重用这部分空闲空间。如果此时观察Redis的内存使用情况，就会发现`used_memory_rss`基本保持不变，但是`used_memory`会不断增长。

如果在Redis的配置文件中没有配置最大内存限制（`maxmemory`），需要时，Redis就会不停的向操作系统申请内存，直到耗尽系统内存为止。因此主动对Redis的内存使用进行限制是十分有必要的，这样当Redis占用的内存超限后，Redis会向客户端返回错误信息，而不是尝试申请更多的内存，最终导致系统崩溃。如果Redis中保存的只是缓存一类不重要的数据，可以设置`maxmemory-policy`，让Redis根据特定的策略删除一些数据来释放空间。

Redis支持三种内存分配器：tcmalloc，jemalloc和libc(ptmalloc)。在大量小数据的使用场景下，使用jemalloc与tcmalloc可以显著的降低内存的碎片率。根据这里的评测，保存200个列表，每个列表有100万的数字，使用jemalloc的碎片率为3%，共使用12.1GB内存，而使用libc时，碎片率为33%，使用了17.7GB内存。但是保存大对象时libc分配器要稍有优势，例如保存3000个列表，每个列表里保存800个大小为2.5k的条目，jemalloc的碎片率为3%，占用8.4G，而libc为1%，占用8GB。

Redis内存对象的数据结构为：

```
typedef struct redisObject {
```

```
typedef struct redisObject {
    unsigned type:4;              /* 类型，比如hash, list等 */
    unsigned encoding:4;          /* 编码格式 */
    unsigned lru:REDIS_LRU_BITS; /* LRU数据 */
    int refcount;                 /* 引用计数器 */
    void *ptr;                    /* 指向数据的指针 */
} robj;
```

缩减键值长度

优化内存最简单直接的方式就是缩减`键`与`值`的长度。`键`当然是越短越好，必要时也可以对长键使用hash算法。`值`如果是对象，可以使用hash结构保存，或者选择更节省空间的序列化工具，比如protostuff进行序列化。如果内容较大，可以先压缩再存储，比如使用snappy。

字符串对象与随机访问

Redis的字符串值有三种编码方式：SDS（Simple Dynamic Strings），embstr与int。借用一下《Redis 设计与实现》中的图：

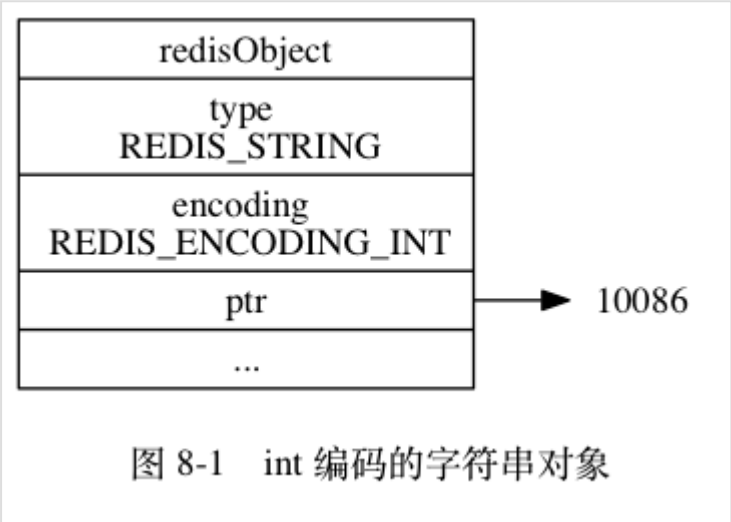

图 8-1　int 编码的字符串对象

当字符串内容是整数，并且不超过long类型的最大值，会使用int编码，最省内存。

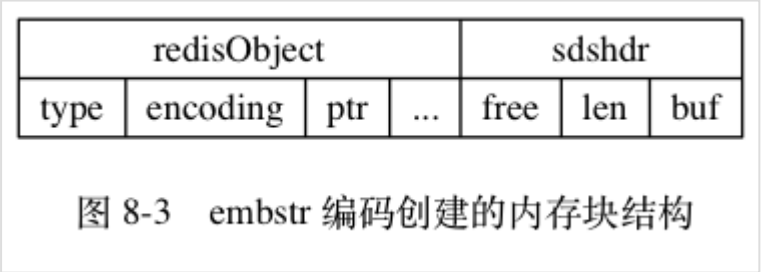

图 8-3　embstr 编码创建的内存块结构

当字符串的长度不大于39字节时，会使用embstr，内嵌SDS结构的字符串，节省一个指针的空间与一次内存IO时间。但对其内容进行修改的时候，会被转成raw编码。

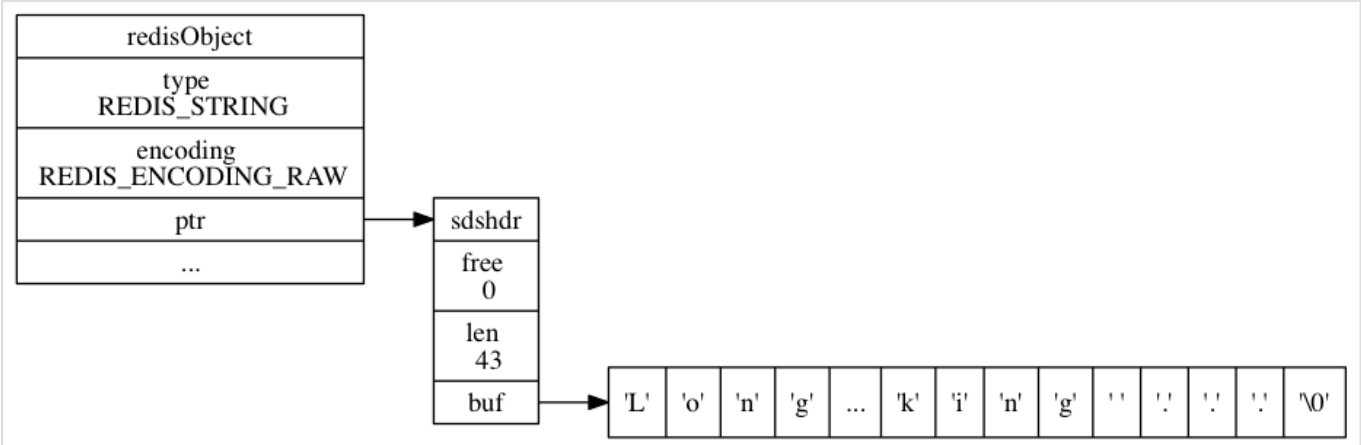

图 8-2　raw 编码的字符串对象

其他情况会使用指针指向SDS。

可以使用`STRLEN`命令查看占用空间，`OBJECT ENCODING`查看编码格式。

SDS结构的定义为：

```
struct sdshdr {
    long len;         /* buf 已占用长度 */
    long free;        /* buf 剩余可用长度 */
    char buf[];       /* 实际保存字符串数据的地方，以'\0'结尾 */
};
```

在字符串第一次被赋值的时候，`len`为字符串的长度，`free`为0，字符串刚好填满`buf`。如果对已赋值的字符串调用`APPEND`追加内容，Redis就会复制原来的值到一个新的`buf`，并预分配一倍的空间，即`free = len`，`buf`里边一半是数据，一半空闲。之后再追加新内容，就可以使用预分配的空闲空间而不需要复制。因此，对Redis的字符串进行追加操作，不论从速度上还是内存使用上都不高效。

因为Redis记录了字符串的长度信息，所以对字符串内容的读取可以做到O(1)，这包括`GETBIT`，`GETRANGE`这类的随机读取。同时，在不增加字符串长度的前提下，使用`SETBIT`与`SETRANGE`对字符串进行随机写入，也是O(1)。基于Redis的这个特性，将多个值合并到一个字符串里保存，就可以节省大量的内存，同时又基本上不会对性能有影响。

举个例子，假设一个应用更需要保存用户的地址。地址为 省 + 市 + 区。

第一种方案，分key：

```
user:1:province -> '北京市'
user:1:city -> '市辖区'
user:1:area -> '东城区'
```

第二种方案，使用hash：

```
user:1 -> {
    province -> '北京市'
    city -> '市辖区'
    area -> '东城区'
}
```

第三种方案，使用string（使用一个6位定长的代码表示省市区，并分别建立三个字典，使用lua脚本一次取出完整地址）：

```
user:1:address -> 110101

provinces -> {
    11 -> '北京市'
}

cities -> {
    1101 -> '市辖区'
}

areas -> {
    110101 -> '东城区'
}
```

在第一个方案使用Pipeline的情况下，三者读写速度差别不大，但是第三种方案内存占用最低，第二种次之（下文会做说明）。

集合短结构

与前边提到的字符串值一样，Redis在几个集合类型的编码上也做了优化，在数据量较小时，Redis会使用短结构，当数据量增大时，再替换成更高效但是内存占用更多的标准结构，这种转换对用户是透明的。下表列出了类型与其可能编码的对应关系：

类型	编码
hash	ziplist, hashtable
list	ziplist, quicklist(3.2版), linkedlist
set	intset, hashset
zset	ziplist, skiplist

hashtable，linkedlist，hashset以及skiplist都是标准的数据结构，Redis的实现也没有特别之处。这些结构的缺点是，因为要记录指针、长度等信息，会有较大的额外空间损耗，尤其数据本身很小的时候，元信息可能比数据还要大。例如，linkedlist的每个节点需要保存3个指针，一共24个字节，如果用来保存4个字节的数字，很不划算。保存100万的数字，数据其实只有4MB，但是指针却占了40多MB。使用ziplist编码后，额外的空间占用可以缩减到1MB左右。但是ziplist是连续存储的，对其进行修改往往需要复制并修改整个列表，效率要低一些。

以上提到的各种编码格式、性能及取舍，在《Redis开发与运维》一书第八章中都有图文并茂的说明，这里就不复述了。

Quicklist

2014年，来自Twitter缓存团队的Yao Yu发表了一段关于Redis在Twitter中应用的演讲。其中提到，Twitter有一个叫做Timeline的重要服务，缓存了一个用户或主题相关的最新的twits的id列表。在当时，Twitter的一个数据中心就有6000台Redis服务器，存储着40TB的Timeline数据。这是一个由大量小数据list组成的巨大缓存，使用ziplist可以成倍的节省内存空间。但是，对ziplist进行随机修改，比如从中间删除一个条目，为了保证ziplist的连续性，就需要移动大量的数据，如果在列表末尾进行追加，也会造成全ziplist复制，这是一个不能忽视的性能问题。ziplist的条目越多，性能问题就越严重的。

同时，Timeline的个体差距也是巨大的， 不活跃的用户，可能一个月也不会发几条twits，但是一个热门的话题，可能每分钟都会有大量的twits产生。当热门Timeline不断增大，机器内存不足时，就需要删除大量小的Timeline，这种行为被叫做扇出（Fan Out）。每当扇出的时候，就会有大量IO发生，Redis进程会发生短暂的阻塞。

于是Twitter对Redis的数据结构进行了扩展，使用ziplist替换普通字符串作为linkedlist的节点。这样在追加的时候，如果条目数量超出了单个ziplist的限制，可以分配一个新的ziplist，并用指针相连，从而避免内存复制。从列表中间删除数据，也仅需要移动一个ziplist中的数据即可。同时linkedlist中指针的空间损耗，由一个ziplist中的全部元素分摊，也可以节省下来大量的内存。但是，当用户删除大量的twits时，可能会造成linkedlist中有大量只含有少量元素的ziplist，留下大量的内存碎片。然而，用户删除大量twits的行为并不常见，所以这个问题并没有造成太大的困扰。

Twitter给这种扩展的数据结构起名为Hybrid List，很可惜的是Twitter当时并没有将其开源。

在2016年5月，Redis发布了3.2版，其中加入了一种新的list编码格式，叫做quicklist，它的原理与Twitter的Hybrid List如出一辙。通过配置`list-max-ziplist-size`的值，可以指定linkedlist中单个ziplist可以容纳的最大容量或者条目。为了进一步节省内存，Redis允许对ziplist中的数据进行压缩，使用的算法为LZF，压缩的同时，可以通过调整`list-compress-depth`的值，对列表两端的元素不做压缩，这样即节省了更多的空间，也保证了对列表两端操作的效率。

下边是quicklist结构的定义：

```
/* quicklistNode is a 32 byte struct describing a ziplist for a quicklist.
 * We use bit fields keep the quicklistNode at 32 bytes.
 * count: 16 bits, max 65536 (max zl bytes is 65k, so max count actually < 32k).
 * encoding: 2 bits, RAW=1, LZF=2.
 * container: 2 bits, NONE=1, ZIPLIST=2.
 * recompress: 1 bit, bool, true if node is temporarry decompressed for usage.
 * attempted_compress: 1 bit, boolean, used for verifying during testing.
 * extra: 12 bits, free for future use; pads out the remainder of 32 bits */
typedef struct quicklistNode {
    struct quicklistNode *prev;
    struct quicklistNode *next;
    unsigned char *zl;
    unsigned int sz;             /* ziplist size in bytes */
    unsigned int count : 16;     /* count of items in ziplist */
    unsigned int encoding : 2;   /* RAW==1 or LZF==2 */
    unsigned int container : 2;  /* NONE==1 or ZIPLIST==2 */
    unsigned int recompress : 1; /* was this node previous compressed? */
    unsigned int attempted_compress : 1; /* node can't compress; too small */
    unsigned int extra : 10; /* more bits to steal for future usage */
} quicklistNode;

/* quicklist is a 32 byte struct (on 64-bit systems) describing a quicklist.
 * 'count' is the number of total entries.
 * 'len' is the number of quicklist nodes.
 * 'compress' is: -1 if compression disabled, otherwise it's the number
 *                of quicklistNodes to leave uncompressed at ends of quicklist.
 * 'fill' is the user-requested (or default) fill factor. */
typedef struct quicklist {
    quicklistNode *head;
    quicklistNode *tail;
    unsigned long count;        /* total count of all entries in all ziplists */
    unsigned int len;           /* number of quicklistNodes */
    int fill : 16;              /* fill factor for individual nodes */
    unsigned int compress : 16; /* depth of end nodes not to compress;0=off */
} quicklist;
```

配置

下边是Redis配置文件中关于短结构的相关配置：

```
# 使用ziplist编码hash的最大数量
hash-max-ziplist-entries 512
# 使用ziplist编码hash的单个元素最大长度（byte）
hash-max-ziplist-value 64

# 使用ziplist编码list的最大数量
list-max-ziplist-entries 512
```

```
# 使用ziplist编码list的单个元素最大长度（byte）
list-max-ziplist-value 64
# 使用quicklist编码list的最大空间或数量，最小值为-5。
# 正值表示数量，负值表示最大空间，成倍递增，-1表示4KB，-2表示8KB，以此类推
list-max-ziplist-size -2
# quicklist的两端不被压缩的节点数量，可以保证对集合两端操作不需要操作整个集合
# 0表示不压缩，1表示quicklist两端各有1个节点不压缩，以此类推
list-compress-depth 0

# 使用intset编码set的最大数量
set-max-intset-entries 512

# 使用ziplist编码zset的最大数量
zset-max-ziplist-entries 128
# 使用ziplist编码zset的单个元素最大长度（byte）
zset-max-ziplist-value 64
```

只要将短结构的数量限制在500～2000个以内，单个元素的长度限制在128个字节以内，其性能一般都会处于合理的范围之内。《Redis in Action》中推荐的配置为：数量限制为1024个，长度限制为64字节，这样可以同时兼顾低内存占用与高性能。

数据分片

数据分片，在大多存储系统的集群中都有应用。在Redis中，除了可以用于集群环境，同样可以用于单个实例环境的内存优化。

使用hash存储普通键-值对

Redis最常被用到的就是其基本的键-值存储功能。然而Redis对一级的存储编码并没有优化，这个时候，将普通的键-值通过分片，存储到hash下边，就可以利用ziplist编码达到优化内存适用的目的。

例如，要使用Redis记录用户的最后请求时间（数值型），系统有100万的用户，用户的ID是一个自增的连续数字，使用传统的方式，可以存储为：

```
user:1 -> 1499256798295
user:2 -> 1499256810227
...
user:1000000 -> 1499256848856
```

更优化的方式是使用hash进行存储，比如每个hash存储1000个用户，这样可以保证hash使用的编码为ziplist，优化之后的结构为：

```
group:1 -> {
    user:1 -> 1499256798295
```

```
    ...
    user:1000 -> 1499256810227
}

....

group:1000 -> {
    user:999001 -> 1499256798295
    ...
    user:1000000 -> 1499256848856
}
```

实现上只需要在程序端，根据ID对用户进行分片就可以了，对既有系统进行重构，完全可以把分片隐藏在接口之下，对上层业务模块透明。

使用散列算法对普通键类型进行分片

上边的例子里，键是连续的数字，分片的逻辑比较简单。如果键是普通的字符串，可以先通过CRC32计算，将其转换成数字，再根据分片的数量取模。使用CRC32做为散列算法，而没有使用更常用的MD5与SHA的原因是，前者的的计算速度更快，并且对大多数情况已经足够好了。

下边摘录《Redis in Action》中提供的通用分片函数：

```python
def shard_key(base, key, total_elements, shard_size):
    if isinstance(key, (int, long)) or key.isdigit():
        # 如果键的类型是数值型，则假定其是连续的，使用其二进制位的高位来选择分片ID
        shard_id = int(str(key), 10) // shard_size
    else:
        # 根据预计元素总数与每个分片容纳的元素数量，计算出实际分片总数量
        shards = 2 * total_elements // shard_size
        # 计算键的散列值与分片数量之间的模数来得到分片ID
        shard_id = binascii.crc32(key) % shards
    return "%s:%s"%(base, shard_id)
```

上边代码中的`total_elements`与`shard_size`的一致性很重要，改变其中的任何一个，都需要对全部数据做重新分片。因此在设计之初，需要留出一定的余量。

参考

- 《Redis in Action》- *Josiah Carlson*
- 《Redis开发与运维》第八章：Redis的内存优化
- 《Redis 设计与实现》
- Pipelining
- Memory optimization