前　言

时间回到2011年4月，当时我正在编写一个用户关系模块，这个模块需要实现一个“共同关注”功能，用于计算出两个用户关注了哪些相同的用户。

举个例子，假设huangz关注了peter、tom、jack三个用户，而john关注了peter、tom、bob、david四个用户，那么当huangz访问john的页面时，共同关注功能就会计算并打印出类似“你跟john都关注了peter和tom”这样的信息。

从集合计算的角度来看，共同关注功能本质上就是计算两个用户关注集合的交集，因为交集这个概念是如此的常见，所以我很自然地认为共同关注这个功能可以很容易地实现，但现实却给了我当头一棒：我所使用的关系数据库并不直接支持交集计算操作，要计算两个集合的交集，除了需要对两个数据表执行合并（join）操作之外，还需要对合并的结果执行去重复（distinct）操作，最终导致交集操作的实现变得异常复杂。

是否存在直接支持集合操作的数据库呢？带着这个疑问，我在搜索引擎上面进行查找，并最终发现了Redis。在我看来，Redis正是我想要找的那种数据库——它内置了集合数据类型，并支持对集合执行交集、并集、差集等集合计算操作，其中的交集计算操作可以直接用于实现我想要的共同关注功能。

得益于Redis本身的简单性，以及Redis手册的详尽和完善，我很快学会了怎样使用Redis的集合数据类型，并用它重新实现了整个用户关系模块：重写之后的关系模块不仅代码量更少，速度更快，更重要的是，之前需要使用一段甚至一大段SQL查询才能实现的功能，现在只需要调用一两个Redis命令就能够实现了，整个模块的可读性得到了极大的提高。

自此之后，我开始在越来越多的项目里面使用Redis，与此同时，我对Redis的内部实现也越来越感兴趣，一些问题开始频繁地出现在我的脑海中，比如：

- Redis的五种数据类型分别是由什么数据结构实现的？
- Redis的字符串数据类型既可以存储字符串（比如"hello world"），又可以存储整数和浮点数（比如10086和3.14），甚至是二进制位（使用*SETBIT*等命令），Redis在内部是怎样存储这些值的？
- Redis的一部分命令只能对特定数据类型执行（比如*APPEND*只能对字符串执行，*HSET*

只能对哈希表执行），而另一部分命令却可以对所有数据类型执行（比如*DEL*、*TYPE*和*EXPIRE*），不同的命令在执行时是如何进行类型检查的？Redis在内部是否实现了一个类型系统？

- Redis的数据库是怎样存储各种不同数据类型的键值对的？数据库里面的过期键又是怎样实现自动删除的？
- 除了数据库之外，Redis还拥有发布与订阅、脚本、事务等特性，这些特性又是如何实现的？
- Redis使用什么模型或者模式来处理客户端的命令请求？一条命令请求从发送到返回需要经过什么步骤？

为了找到这些问题的答案，我再次在搜索引擎上面进行查找，可惜的是这次搜索并没有多少收获：Redis还是一个非常年轻的软件，对它的最好介绍就是官方网站上面的文档，但是这些文档主要关注的是怎样使用Redis，而不是介绍Redis的内部实现。另外，网上虽然有一些博客文章对Redis的内部实现进行了介绍，但这些文章要么不齐全（只介绍了Redis中的少数几个特性），要么就写得过于简单（只是一些概述性的文章），要么关注的就是旧版本（比如2.0、2.2或者2.4，而当时的最新版已经是2.6了）。

综合来看，详细而且完整地介绍Redis内部实现的资料，无论是外文还是中文都不存在。意识到这一点之后，我决定自己动手注释Redis的源代码，从中寻找问题的答案，并通过写博客的方式与其他Redis用户分享我的发现。在积累了七八篇Redis源代码注释文章之后，我想如果能将这些博文汇集成书的话，那一定会非常有趣，并且我自己也会从中学到很多知识。于是我在2012年年末开始创作《Redis设计与实现》，并最终于2013年3月8日在互联网发布了本书的第一版。

尽管《Redis设计与实现》第一版顺利发布了，但在我的心目中，这个第一版还是有很多不完善的地方：

- 比如说，因为第一版是我边注释Redis源代码边写的，如果有足够时间让我先完整地注释一遍Redis的源代码，然后再进行写作的话，那么书本在内容方面应该会更为全面。
- 又比如说，第一版只介绍了Redis的内部机制和单机特性，但并没有介绍Redis多机特性，而我认为只有将关于多机特性的介绍也包含进来，这本《Redis设计与实现》才算是真正的完成了。

就在我考虑应该何时编写新版来修复这些缺陷的时候，机械工业出版社的吴怡编辑来信询问我是否有兴趣正式地出版《Redis设计与实现》，能够正式地出版自己写的书一直是我梦寐以求的事情，我找不到任何拒绝这一邀请的理由，就这样，在《Redis设计与实现》第一版发布几天之后，新版《Redis设计与实现》的写作也马不停蹄地开始了。

从2013年3月到2014年1月这11个月间，我重新注释了Redis在unstable分支的源代码（也即是现在的Redis 3.0源代码），重写了《Redis设计与实现》第一版已有的所有章节，并向书中添加了关于二进制位操作（bitop）、排序、复制、Sentinel和集群等主题的新章节，最终完成了这本新版的《Redis 设计与实现》。本书不仅介绍了Redis的内部机制（比如数据库实现、类型系统、事件模型），而且还介绍了大部分Redis单机特性（比如事务、持久化、Lua脚本、排序、二进制位操

作），以及所有Redis多机特性（如复制、Sentinel和集群）。

虽然作者创作本书的初衷只是为了满足自己的好奇心，但了解Redis内部实现的好处并不仅仅在于满足好奇心：通过了解Redis的内部实现，理解每一个特性和命令背后的运作机制，可以帮助我们更高效地使用Redis，避开那些可能会引起性能问题的陷阱。我衷心希望这本新版《Redis设计与实现》能够帮助读者更好地了解Redis，并成为更优秀的Redis使用者。

本书的第一版获得了很多热心读者的反馈，这本新版的很多改进也来源于读者们的意见和建议，因此我将继续在www.RedisBook.com设置disqus论坛（可以不注册直接发贴），欢迎读者随时就这本新版《Redis设计与实现》发表提问、意见、建议、批评、勘误，等等，我会努力地采纳大家的意见，争取在将来写出更好的《Redis设计与实现》，以此来回报大家对本书的支持。

黄健宏（huangz）

2014年3月于清远

致　　谢

我要感谢hoterran 和iammutex 这两位良师益友，他们对我的帮助和支持贯穿整本书从概念萌芽到正式出版的整个阶段，也感谢他们抽出宝贵的时间为本书审稿。

我要感谢吴怡编辑鼓励我创作并出版这本新版《Redis 设计与实现》，以及她在写作过程中对我的悉心指导。

我要感谢TimYang 在百忙之中抽空为本书审稿，并耐心地给出了详细的意见。

我要感谢Redis 之父Salvatore Sanfilippo ，如果不是他创造了Redis 的话，这本书也不会出现了。

我要感谢所有阅读了《Redis 设计与实现》第一版的读者，他们的意见和建议帮助我更好地完成这本新版《Redis 设计与实现》。

最后，我要感谢我的家人和朋友，他们的关怀和鼓励使得本书得以顺利完成。

第 1 章
引　言

本书对 Redis 的大多数单机功能以及所有多机功能的实现原理进行了介绍，力图展示这些功能的核心数据结构以及关键的算法思想。

通过阅读本书，读者可以快速、有效地了解 Redis 的内部构造以及运作机制，这些知识可以帮助读者更好地、也更高效地使用 Redis。

为了让本书的内容保持简单并且容易读懂，本书会尽量以高层次的角度来对 Redis 的实现原理进行描述，如果读者只是对 Redis 的实现原理感兴趣，但并不想研究 Redis 的源代码，那么阅读本书就足够了。

另一方面，如果读者打算深入了解 Redis 实现原理的底层细节，本书在 RedisBook.com 提供了一份带有详细注释的 Redis 源代码，读者可以先阅读本书对某一功能的介绍，然后再阅读该功能对应的实现代码，这有助于读者更快地读懂实现代码，也有助于读者更深入地了解该功能的实现原理。

1.1　Redis 版本说明

本书是基于 Redis 2.9——也即是 Redis 3.0 的开发版来编写的，因为 Redis 3.0 的更新主要与 Redis 的多机功能有关，而 Redis 3.0 的单机功能则与 Redis 2.6、Redis 2.8 的单机功能基本相同，所以本书的内容对于使用 Redis 2.6 至 Redis 3.0 的读者来说应该都是有用的。

另外，因为 Redis 通常都是渐进地增加新功能，并且很少会大幅地修改已有的功能，所以本书的大部分内容对于 Redis 3.0 之后的几个版本来说，应该也是有用的。

1.2　章节编排

本书由“数据结构与对象”、“单机数据库的实现”、“多机数据库的实现”、“独立功能的实现”四个部分组成。

第一部分“数据结构与对象”

Redis 数据库里面的每个键值对（key-value pair）都是由对象（object）组成的，其中：

- 数据库键总是一个字符串对象（string object）；
- 而数据库键的值则可以是字符串对象、列表对象（list object）、哈希对象（hash object）、集合对象（set object）、有序集合对象（sorted set object）这五种对象中的其中一种。

比如说，执行以下命令将在数据库中创建一个键为字符串对象，值也为字符串对象的键值对：

```
redis> SET msg "hello world"
OK
```

而执行以下命令将在数据库中创建一个键为字符串对象，值为列表对象的键值对：

```
redis> RPUSH numbers 1 3 5 7 9
(integer) 5
```

本书的第一部分将对以上提到的五种不同类型的对象进行介绍，剖析这些对象所使用的底层数据结构，并说明这些数据结构是如何深刻地影响对象的功能和性能的。

第二部分“单机数据库的实现”

本书的第二部分对 Redis 实现单机数据库的方法进行了介绍。

第 9 章“数据库”对 Redis 数据库的实现原理进行了介绍，说明了服务器保存键值对的方法，服务器保存键值对过期时间的方法，以及服务器自动删除过期键值对的方法等等。

第 10 章“RDB 持久化”和第 11 章“AOF 持久化”分别介绍了 Redis 两种不同的持久化方式的实现原理，说明了服务器根据数据库来生成持久化文件的方法，服务器根据持久化文件来还原数据库的方法，以及 *BGSAVE* 命令和 *BGREWRITEAOF* 命令的实现原理等等。

第 12 章“事件”对 Redis 的文件事件和时间事件进行了介绍：

- 文件事件主要用于应答（accept）客户端的连接请求，接收客户端发送的命令请求，以及向客户端返回命令回复；
- 而时间事件则主要用于执行 redis.c/serverCron 函数，这个函数通过执行常规的维护和管理操作来保持 Redis 服务器的正常运作，一些重要的定时操作也是由这个函数负责触发的。

第 13 章“客户端”对 Redis 服务器维护和管理客户端状态的方法进行了介绍，列举了客户端状态包含的各个属性，说明了客户端的输入缓冲区和输出缓冲区的实现方法，以及 Redis 服务器创建和销毁客户端状态的条件等等。

第 14 章“服务器”对单机 Redis 服务器的运作机制进行了介绍，详细地说明了服务器处理命令请求的步骤，解释了 serverCron 函数所做的工作，并讲解了 Redis 服务器的初始化过程。

第三部分“多机数据库的实现”

本书的第三部分对 Redis 的 Sentinel、复制（replication）、集群（cluster）三个多机功能进行了介绍。

第 15 章“复制”对 Redis 的主从复制功能（master-slave replication）的实现原理进行了介绍，说明了当用户指定一个服务器（从服务器）去复制另一个服务器（主服务器）时，主从服务器之间执行了什么操作，进行了什么数据交互，诸如此类。

第 16 章“Sentinel”对 Redis Sentinel 的实现原理进行了介绍，说明了 Sentinel 监视服务器的方法，Sentinel 判断服务器是否下线的方法，以及 Sentinel 对下线服务器进行故障转移的方法等等。

第 17 章“集群”对 Redis 集群的实现原理进行了介绍，说明了节点（node）的构建方法，节点处理命令请求的方法，转发（redirection）错误的实现方法，以及各个节点之间进行通信的方法等等。

第四部分“独立功能的实现”

本书的第四部分对 Redis 中各个相对独立的功能模块进行了介绍。

第 18 章“发布与订阅”对 *PUBLISH*、*SUBSCRIBE*、*PUBSUB* 等命令的实现原理进行了介绍，解释了 Redis 的发布与订阅功能是如何实现的。

第 19 章“事务”对 *MULTI*、*EXEC*、*WATCH* 等命令的实现原理进行了介绍，解释了 Redis 的事务是如何实现的，并说明了 Redis 的事务对 ACID 性质的支持程度。

第 20 章“Lua 脚本”对 *EVAL*、*EVALSHA*、*SCRIPT LOAD* 等命令的实现原理进行了介绍，解释了 Redis 服务器是如何执行和管理用户传入的 Lua 脚本的；这一章还对 Redis 服务器构建 Lua 环境的过程，以及主从服务器之间复制 Lua 脚本的方法进行了介绍。

第 21 章“排序”对 *SORT* 命令以及 *SORT* 命令所有可用选项（比如 *DESC*、*ALPHA*、*GET* 等等）的实现原理进行了介绍，并说明了当 SORT 命令带有多个选项时，不同选项执行的先后顺序。

第 22 章“二进制位数组”对 Redis 保存二进制位数组的方法进行了介绍，并说明了 *GETBIT*、*SETBIT*、*BITCOUNT*、*BITOP* 这几个二进制位数组操作命令的实现原理。

第 23 章“慢查询日志”对 Redis 创建和保存慢查询日志（slow log）的方法进行了介绍，并说明了 *SLOWLOG GET*、*SLOWLOG LEN*、*SLOWLOG RESET* 等慢查询日志操作命令的实现原理。

第 24 章“监视器”介绍了将客户端变为监视器（monitor）的方法，以及服务器在处理命令请求时，向监视器发送命令信息的方法。

1.3 推荐的阅读方法

因为 Redis 的单机功能是多机功能的子集，所以无论读者使用的是单机模式的 Redis，还是多机模式的 Redis，都应该阅读本书的第一部分和第二部分，这两个部分包含的知识是所有 Redis 使用者都必然会用到的。

如果读者要使用 Redis 的多机功能，那么在阅读本书的第一部分和第二部分之后，应该接着阅读本书的第三部分。如果读者只使用 Redis 的单机功能，那么可以跳过第三部分，直接阅读第四部分。

本书的前三个部分都是以自底向上（bottom-up）的方式来写的，也就是说，排在后面的章节会假设读者已经读过了排在前面的章节。如果一个概念在前面的章节已经介绍过，那么后面的章节就不会再重复介绍这个概念，所以读者最好按顺序阅读这三部分的各个章节。

本书的第四部分包含的各章是完全独立的，读者可以按自己的兴趣来挑选要读的章节。在本书的第四部分中，除了第 20 章的其中一节涉及多机功能的内容之外，其他章节都没有涉及多机功能的内容，所以第四部分的大部分章节都可以在只阅读了本书第一部分和第二部分的情况下阅读。

图 1-1 对上面描述的阅读方法进行了总结。

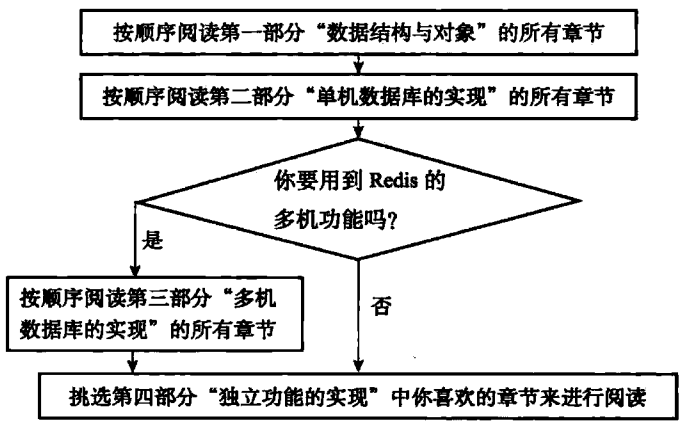

图 1-1 推荐阅读方法

1.4 行文规则

名字引用规则

在第一次引用 Redis 源代码文件 `file` 中的名字 `name` 时，本书使用 `file/name` 格式，比如 `redis.c/main` 表示 `redis.c` 文件中的 `main` 函数，而 `redis.h/redisDb` 则表示 `redis.h` 文件中的 `redisDb` 结构，诸如此类。

另外，在第一次引用标准库头文件 file 中的名字 name 时，本书使用 <file>/name 格式，比如 <unistd.h>/write 表示 unistd.h 头文件的 write 函数，而 <stdio.h>/printf 则表示 stdio.h 头文件的 printf 函数，诸如此类。

在第一次引用某个名字之后，本书就会去掉名字前缀的文件名，直接使用名字本身。举个例子，当第一次引用 redis.h 文件的 redisDb 结构的时候，会使用 redis.h/redisDb 格式，而之后再次引用 redisDb 结构时，只使用名字 redisDb。

结构引用规则

本书使用 struct.property 格式来引用 struct 结构的 property 属性，比如 redisDb.id 表示 redisDb 结构的 id 属性，而 redisDb.expires 则表示 redisDb 结构的 expires 属性，诸如此类。

算法规则

除非有额外说明，否则本书列出的算法复杂度一律为最坏情形下的算法复杂度。

代码规则

本书使用 C 语言和 Python 语言来展示代码：

- 在描述数据结构以及比较简短的代码时，本书通常会直接粘贴 Redis 的源代码，也即 C 语言代码。
- 而当需要使用代码来描述比较长或者比较复杂的程序时，本书通常会使用 Python 语言来表示伪代码。

本书展示的 Python 伪代码中通常会包含 server 和 client 两个全局变量，其中 server 表示服务器状态（redis.h/redisServer 结构的实例），而 client 则表示正在执行操作的客户端状态（redis.h/redisClient 结构的实例）。

1.5 配套网站

本书配套网站 redisbook.com 记录了本书的最新消息，并且提供了附带详细注释的 Redis 源代码可供下载，读者也可以通过这个网站查看和反馈本书的勘误，或者发表与本书有关的问题、意见以及建议。

第一部分

数据结构与对象

第 2 章
简单动态字符串

Redis 没有直接使用 C 语言传统的字符串表示（以空字符结尾的字符数组，以下简称 C 字符串），而是自己构建了一种名为简单动态字符串（simple dynamic string，SDS）的抽象类型，并将 SDS 用作 Redis 的默认字符串表示。

在 Redis 里面，C 字符串只会作为字符串字面量（string literal）用在一些无须对字符串值进行修改的地方，比如打印日志：

```
redisLog(REDIS_WARNING,"Redis is now ready to exit, bye bye...");
```

当 Redis 需要的不仅仅是一个字符串字面量，而是一个可以被修改的字符串值时，Redis 就会使用 SDS 来表示字符串值，比如在 Redis 的数据库里面，包含字符串值的键值对在底层都是由 SDS 实现的。

举个例子，如果客户端执行命令：

```
redis> SET msg "hello world"
OK
```

那么 Redis 将在数据库中创建一个新的键值对，其中：

- 键值对的键是一个字符串对象，对象的底层实现是一个保存着字符串 "msg" 的 SDS。
- 键值对的值也是一个字符串对象，对象的底层实现是一个保存着字符串 "hello world" 的 SDS。

又比如，如果客户端执行命令：

```
redis> RPUSH fruits "apple" "banana" "cherry"
(integer) 3
```

那么 Redis 将在数据库中创建一个新的键值对，其中：

- 键值对的键是一个字符串对象，对象的底层实现是一个保存了字符串 "fruits" 的 SDS。
- 键值对的值是一个列表对象，列表对象包含了三个字符串对象，这三个字符串对象分别由三个 SDS 实现：第一个 SDS 保存着字符串 "apple"，第二个 SDS 保存着字符串 "banana"，第三个 SDS 保存着字符串 "cherry"。

除了用来保存数据库中的字符串值之外，SDS 还被用作缓冲区（buffer）：AOF 模块中

的 AOF 缓冲区，以及客户端状态中的输入缓冲区，都是由 SDS 实现的，在之后介绍 AOF 持久化和客户端状态的时候，我们会看到 SDS 在这两个模块中的应用。

本章接下来将对 SDS 的实现进行介绍，说明 SDS 和 C 字符串的不同之处，解释为什么 Redis 要使用 SDS 而不是 C 字符串，并在本章的最后列出 SDS 的操作 API。

2.1 SDS 的定义

每个 sds.h/sdshdr 结构表示一个 SDS 值：

```
struct sdshdr {

    // 记录 buf 数组中已使用字节的数量
    // 等于 SDS 所保存字符串的长度
    int len;

    // 记录 buf 数组中未使用字节的数量
    int free;

    // 字节数组，用于保存字符串
    char buf[];

};
```

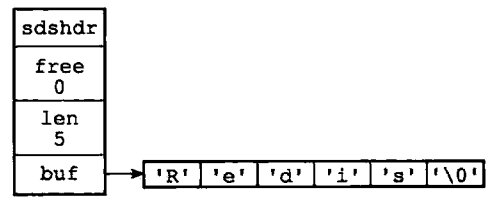

图 2-1 SDS 示例

图 2-1 展示了一个 SDS 示例：

- free 属性的值为 0，表示这个 SDS 没有分配任何未使用空间。
- len 属性的值为 5，表示这个 SDS 保存了一个五字节长的字符串。
- buf 属性是一个 char 类型的数组，数组的前五个字节分别保存了 'R'、'e'、'd'、'i'、's' 五个字符，而最后一个字节则保存了空字符 '\0'。

SDS 遵循 C 字符串以空字符结尾的惯例，保存空字符的 1 字节空间不计算在 SDS 的 len 属性里面，并且为空字符分配额外的 1 字节空间，以及添加空字符到字符串末尾等操作，都是由 SDS 函数自动完成的，所以这个空字符对于 SDS 的使用者来说是完全透明的。遵循空字符结尾这一惯例的好处是，SDS 可以直接重用一部分 C 字符串函数库里面的函数。

举个例子，如果我们有一个指向图 2-1 所示 SDS 的指针 s，那么我们可以直接使用 <stdio.h>/printf 函数，通过执行以下语句：

```
printf("%s", s->buf);
```

来打印出 SDS 保存的字符串值 "Redis"，而无须为 SDS 编写专门的打印函数。

图 2-2 展示了另一个 SDS 示例。这个 SDS 和之前展示的 SDS 一样，都保存了字符串值 "Redis"。这个 SDS 和之前展示的 SDS 的区别在于，这个 SDS 为 buf 数组分配了五字节未使用空间，所以它的 free 属性的值为 5（图中使用五个空格来表示五字节的未使用空间）。

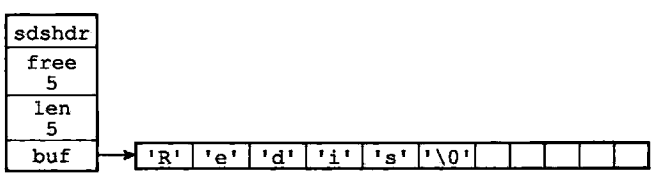

图 2-2 带有未使用空间的 SDS 示例

接下来的一节将详细地说明未使用空间在 SDS 中的作用。

2.2 SDS 与 C 字符串的区别

根据传统，C 语言使用长度为 N+1 的字符数组来表示长度为 N 的字符串，并且字符数组的最后一个元素总是空字符 '\0'。

例如，图 2-3 就展示了一个值为 "Redis" 的 C 字符串。

C 语言使用的这种简单的字符串表示方式，并不能满足 Redis 对字符串在安全性、效率以及功能方面的要求，本节接下来的内容将详细对比 C 字符串和 SDS 之间的区别，并说明 SDS 比 C 字符串更适用于 Redis 的原因。

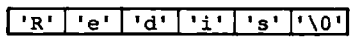

图 2-3 C 字符串

2.2.1 常数复杂度获取字符串长度

因为 C 字符串并不记录自身的长度信息，所以为了获取一个 C 字符串的长度，程序必须遍历整个字符串，对遇到的每个字符进行计数，直到遇到代表字符串结尾的空字符为止，这个操作的复杂度为 *O*(*N*)。

举个例子，图 2-4 展示了程序计算一个 C 字符串长度的过程。

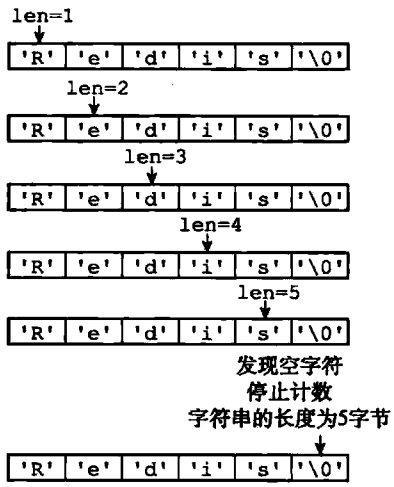

图 2-4 计算 C 字符串长度的过程

和 C 字符串不同，因为 SDS 在 len 属性中记录了 SDS 本身的长度，所以获取一个 SDS 长度的复杂度仅为 *O*(1)。

举个例子，对于图 2-5 所示的 SDS 来说，程序只要访问 SDS 的 len 属性，就可以立即知道 SDS 的长度为 5 字节。

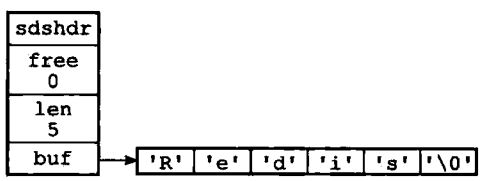

图 2-5 5 字节长的 SDS

又例如，对于图 2-6 展示的 SDS 来说，程序只要访问 SDS 的 len 属性，就可以立即知道 SDS 的长度为 11 字节。

设置和更新 SDS 长度的工作是由 SDS 的 API 在执行时自动完成的，使用 SDS 无须进行任何手动修改长度的工作。

通过使用 SDS 而不是 C 字符串，Redis 将获取字符串长度所需的复杂度从 *O*(*N*) 降低到了 *O*(1)，这确保了获取字符串长度的工作不会成为 Redis 的性能瓶颈。例如，因为字符串键在底层使用 SDS 来实现，所以即使我们对一个非常长的字符串键反复执行 *STRLEN* 命令，也

不会对系统性能造成任何影响，因为 *STRLEN* 命令的复杂度仅为 *O*(1)。

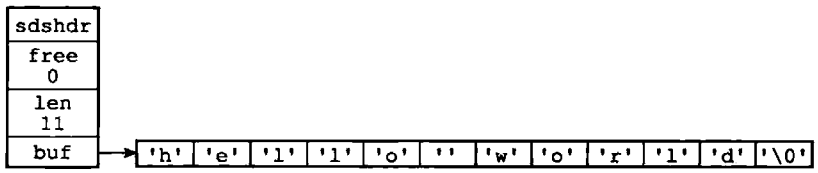

图 2-6 11 字节长的 SDS

2.2.2 杜绝缓冲区溢出

除了获取字符串长度的复杂度高之外，C 字符串不记录自身长度带来的另一个问题是容易造成缓冲区溢出（buffer overflow）。举个例子，<string.h>/strcat 函数可以将 src 字符串中的内容拼接到 dest 字符串的末尾：

```
char *strcat(char *dest, const char *src);
```

因为 C 字符串不记录自身的长度，所以 strcat 假定用户在执行这个函数时，已经为 dest 分配了足够多的内存，可以容纳 src 字符串中的所有内容，而一旦这个假定不成立时，就会产生缓冲区溢出。

举个例子，假设程序里有两个在内存中紧邻着的 C 字符串 s1 和 s2，其中 s1 保存了字符串 "Redis"，而 s2 则保存了字符串 "MongoDB"，如图 2-7 所示。

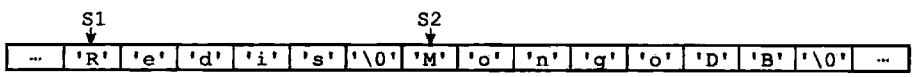

图 2-7 在内存中紧邻的两个 C 字符串

如果一个程序员决定通过执行：

```
strcat(s1, " Cluster");
```

将 s1 的内容修改为 "Redis Cluster"，但粗心的他却忘了在执行 strcat 之前为 s1 分配足够的空间，那么在 strcat 函数执行之后，s1 的数据将溢出到 s2 所在的空间中，导致 s2 保存的内容被意外地修改，如图 2-8 所示。

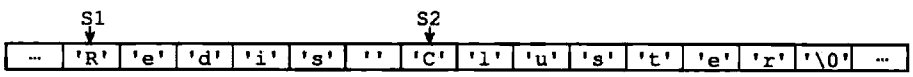

图 2-8 s1 的内容溢出到了 s2 所在的位置上

与 C 字符串不同，SDS 的空间分配策略完全杜绝了发生缓冲区溢出的可能性：当 SDS API 需要对 SDS 进行修改时，API 会先检查 SDS 的空间是否满足修改所需的要求，如果不满足的话，API 会自动将 SDS 的空间扩展至执行修改所需的大小，然后才执行实际的修改操作，所以使用 SDS 既不需要手动修改 SDS 的空间大小，也不会出现前面所说的缓冲区溢出问题。

举个例子，SDS 的 API 里面也有一个用于执行拼接操作的 sdscat 函数，它可以将一

个 C 字符串拼接到给定 SDS 所保存的字符串的后面，但是在执行拼接操作之前，sdscat 会先检查给定 SDS 的空间是否足够，如果不够的话，sdscat 就会先扩展 SDS 的空间，然后才执行拼接操作。

例如，如果我们执行：

```
sdscat(s, " Cluster");
```

其中 SDS 值 s 如图 2-9 所示，那么 sdscat 将在执行拼接操作之前检查 s 的长度是否足够，在发现 s 目前的空间不足以拼接 " Cluster" 之后，sdscat 就会先扩展 s 的空间，然后才执行拼接 " Cluster" 的操作，拼接操作完成之后的 SDS 如图 2-10 所示。

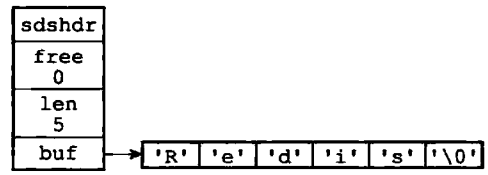

图 2-9 sdscat 执行之前的 SDS

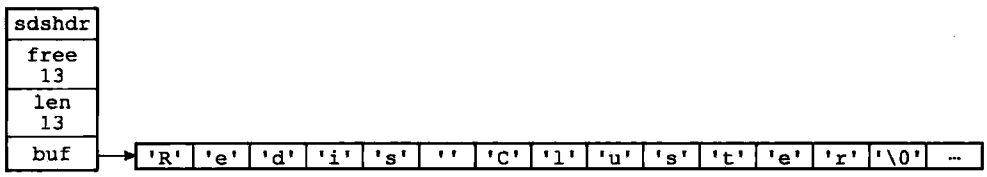

图 2-10 sdscat 执行之后的 SDS

注意，图 2-10 所示的 SDS，sdscat 不仅对这个 SDS 进行了拼接操作，它还为 SDS 分配了 13 字节的未使用空间，并且拼接之后的字符串也正好是 13 字节长，这种现象既不是 bug 也不是巧合，它和 SDS 的空间分配策略有关，接下来的小节将对这一策略进行说明。

2.2.3 减少修改字符串时带来的内存重分配次数

正如前两个小节所说，因为 C 字符串并不记录自身的长度，所以对于一个包含了 N 个字符的 C 字符串来说，这个 C 字符串的底层实现总是一个 N+1 个字符长的数组（额外的一个字符空间用于保存空字符）。因为 C 字符串的长度和底层数组的长度之间存在着这种关联性，所以每次增长或者缩短一个 C 字符串，程序都总要对保存这个 C 字符串的数组进行一次内存重分配操作：

- 如果程序执行的是增长字符串的操作，比如拼接操作（append），那么在执行这个操作之前，程序需要先通过内存重分配来扩展底层数组的空间大小——如果忘了这一步就会产生缓冲区溢出。
- 如果程序执行的是缩短字符串的操作，比如截断操作（trim），那么在执行这个操作之后，程序需要通过内存重分配来释放字符串不再使用的那部分空间——如果忘了这一步就会产生内存泄漏。

举个例子，如果我们持有一个值为 "Redis" 的 C 字符串 s，那么为了将 s 的值改为 "Redis Cluster"，在执行：

```
strcat(s, " Cluster");
```

之前，我们需要先使用内存重分配操作，扩展 s 的空间。

之后，如果我们又打算将 s 的值从 "Redis Cluster" 改为 "Redis Cluster Tutorial"，那么在执行：

```
strcat(s, " Tutorial");
```

之前，我们需要再次使用内存重分配扩展 s 的空间，诸如此类。

因为内存重分配涉及复杂的算法，并且可能需要执行系统调用，所以它通常是一个比较耗时的操作：

- 在一般程序中，如果修改字符串长度的情况不太常出现，那么每次修改都执行一次内存重分配是可以接受的。
- 但是 Redis 作为数据库，经常被用于速度要求严苛、数据被频繁修改的场合，如果每次修改字符串的长度都需要执行一次内存重分配的话，那么光是执行内存重分配的时间就会占去修改字符串所用时间的一大部分，如果这种修改频繁地发生的话，可能还会对性能造成影响。

为了避免 C 字符串的这种缺陷，SDS 通过未使用空间解除了字符串长度和底层数组长度之间的关联：在 SDS 中，buf 数组的长度不一定就是字符数量加一，数组里面可以包含未使用的字节，而这些字节的数量就由 SDS 的 free 属性记录。

通过未使用空间，SDS 实现了空间预分配和惰性空间释放两种优化策略。

1. 空间预分配

空间预分配用于优化 SDS 的字符串增长操作：当 SDS 的 API 对一个 SDS 进行修改，并且需要对 SDS 进行空间扩展的时候，程序不仅会为 SDS 分配修改所必须要的空间，还会为 SDS 分配额外的未使用空间。

其中，额外分配的未使用空间数量由以下公式决定：

- 如果对 SDS 进行修改之后，SDS 的长度（也即是 len 属性的值）将小于 1MB，那么程序分配和 len 属性同样大小的未使用空间，这时 SDS len 属性的值将和 free 属性的值相同。举个例子，如果进行修改之后，SDS 的 len 将变成 13 字节，那么程序也会分配 13 字节的未使用空间，SDS 的 buf 数组的实际长度将变成 13+13+1=27 字节（额外的一字节用于保存空字符）。
- 如果对 SDS 进行修改之后，SDS 的长度将大于等于 1MB，那么程序会分配 1MB 的未使用空间。举个例子，如果进行修改之后，SDS 的 len 将变成 30MB，那么程序会分配 1MB 的未使用空间，SDS 的 buf 数组的实际长度将为 30 MB + 1MB + 1byte。

通过空间预分配策略，Redis 可以减少连续执行字符串增长操作所需的内存重分配次数。

举个例子，对于图 2-11 所示的 SDS 值 s 来说，如果我们执行：

```
sdscat(s, " Cluster");
```

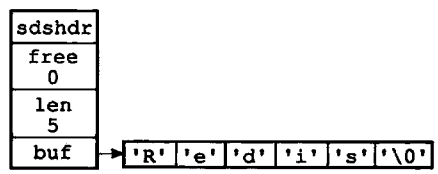

图 2-11 执行 sdscat 之前的 SDS

那么 sdscat 将执行一次内存重分配操作，将 SDS 的长度修改为 13 字节，并将 SDS 的未使用空间同样修改为 13 字节，如图 2-12 所示。

图 2-12 执行 sdscat 之后 SDS

如果这时，我们再次对 s 执行：

```
sdscat(s, " Tutorial");
```

那么这次 sdscat 将不需要执行内存重分配，因为未使用空间里面的 13 字节足以保存 9 字节的 " Tutorial"，执行 sdscat 之后的 SDS 如图 2-13 所示。

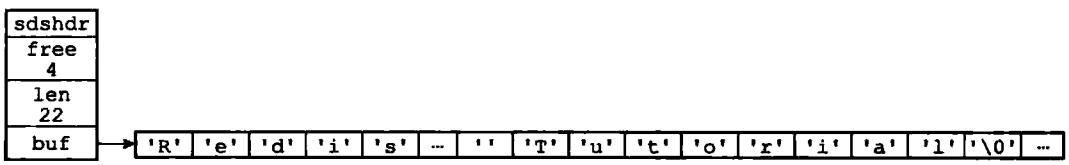

图 2-13 再次执行 sdscat 之后的 SDS

在扩展 SDS 空间之前，SDS API 会先检查未使用空间是否足够，如果足够的话，API 就会直接使用未使用空间，而无须执行内存重分配。

通过这种预分配策略，SDS 将连续增长 N 次字符串所需的内存重分配次数从必定 N 次降低为最多 N 次。

2. 惰性空间释放

惰性空间释放用于优化 SDS 的字符串缩短操作：当 SDS 的 API 需要缩短 SDS 保存的字符串时，程序并不立即使用内存重分配来回收缩短后多出来的字节，而是使用 free 属性将这些字节的数量记录起来，并等待将来使用。

举个例子，sdstrim 函数接受一个 SDS 和一个 C 字符串作为参数，移除 SDS 中所有在 C 字符串中出现过的字符。

比如对于图 2-14 所示的 SDS 值 s 来说，执行：

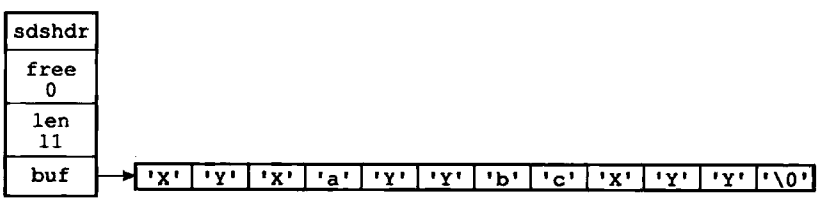

图 2-14 执行 sdstrim 之前的 SDS

```
sdstrim(s, "XY"); // 移除 SDS 字符串中的所有 'X' 和 'Y'
```

会将 SDS 修改成图 2-15 所示的样子。

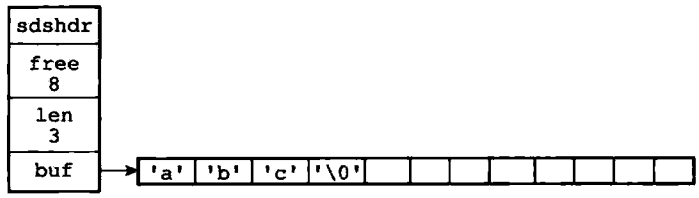

图 2-15　执行 sdstrim 之后的 SDS

注意执行 sdstrim 之后的 SDS 并没有释放多出来的 8 字节空间，而是将这 8 字节空间作为未使用空间保留在了 SDS 里面，如果将来要对 SDS 进行增长操作的话，这些未使用空间就可能会派上用场。

举个例子，如果现在对 s 执行：

```
sdscat(s, " Redis");
```

那么完成这次 sdscat 操作将不需要执行内存重分配：因为 SDS 里面预留的 8 字节空间已经足以拼接 6 个字节长的 " Redis"，如图 2-16 所示。

通过惰性空间释放策略，SDS 避免了缩短字符串时所需的内存重分配操作，并为将来可能有的增长操作提供了优化。

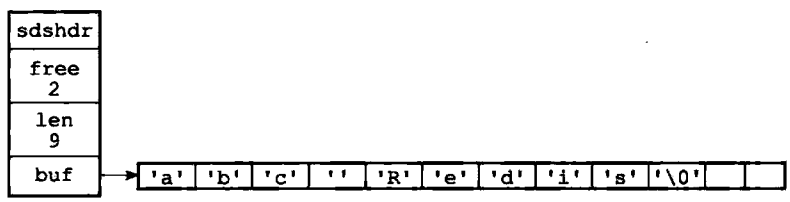

图 2-16　执行 sdscat 之后的 SDS

与此同时，SDS 也提供了相应的 API，让我们可以在有需要时，真正地释放 SDS 的未使用空间，所以不用担心惰性空间释放策略会造成内存浪费。

2.2.4　二进制安全

C 字符串中的字符必须符合某种编码（比如 ASCII），并且除了字符串的末尾之外，字符串里面不能包含空字符，否则最先被程序读入的空字符将被误认为是字符串结尾，这些限制使得 C 字符串只能保存文本数据，而不能保存像图片、音频、视频、压缩文件这样的二进制数据。

举个例子，如果有一种使用空字符来分割多个单词的特殊数据格式，如图 2-17 所示，那么这种格式就不能使用 C 字符串来保存，因为 C 字符串所用的函数只会识别出其中的 "Redis"，而忽略之后的 "Cluster"。

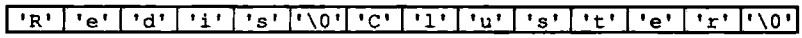

图 2-17 使用空字符来分割单词的特殊数据格式

虽然数据库一般用于保存文本数据，但使用数据库来保存二进制数据的场景也不少见，因此，为了确保 Redis 可以适用于各种不同的使用场景，SDS 的 API 都是二进制安全的（binary-safe），所有 SDS API 都会以处理二进制的方式来处理 SDS 存放在 buf 数组里的数据，程序不会对其中的数据做任何限制、过滤、或者假设，数据在写入时是什么样的，它被读取时就是什么样。

这也是我们将 SDS 的 `buf` 属性称为字节数组的原因——Redis 不是用这个数组来保存字符，而是用它来保存一系列二进制数据。

例如，使用 SDS 来保存之前提到的特殊数据格式就没有任何问题，因为 SDS 使用 `len` 属性的值而不是空字符来判断字符串是否结束，如图 2-18 所示。

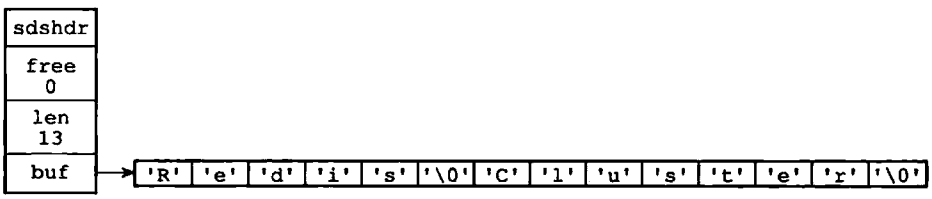

图 2-18 保存了特殊数据格式的 SDS

通过使用二进制安全的 SDS，而不是 C 字符串，使得 Redis 不仅可以保存文本数据，还可以保存任意格式的二进制数据。

2.2.5 兼容部分 C 字符串函数

虽然 SDS 的 API 都是二进制安全的，但它们一样遵循 C 字符串以空字符结尾的惯例：这些 API 总会将 SDS 保存的数据的末尾设置为空字符，并且总会在为 buf 数组分配空间时多分配一个字节来容纳这个空字符，这是为了让那些保存文本数据的 SDS 可以重用一部分 `<string.h>` 库定义的函数。

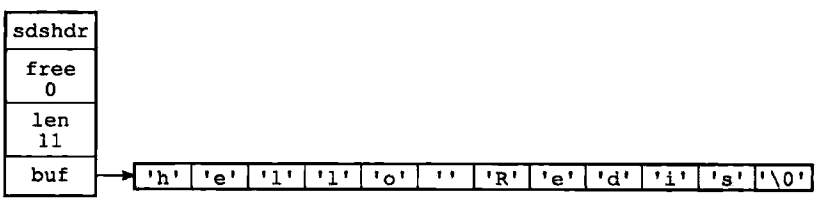

图 2-19 一个保存着文本数据的 SDS

举个例子，如图 2-19 所示，如果我们有一个保存文本数据的 SDS 值 `sds`，那么我们就可以重用 `<string.h>/strcasecmp` 函数，使用它来对比 SDS 保存的字符串和另一个 C 字符串：

```
strcasecmp(sds->buf, "hello world");
```

这样 Redis 就不用自己专门去写一个函数来对比 SDS 值和 C 字符串值了。

与此类似，我们还可以将一个保存文本数据的 SDS 作为 strcat 函数的第二个参数，将 SDS 保存的字符串追加到一个 C 字符串的后面：

```
strcat(c_string, sds->buf);
```

这样 Redis 就不用专门编写一个将 SDS 字符串追加到 C 字符串之后的函数了。

通过遵循 C 字符串以空字符结尾的惯例，SDS 可以在有需要时重用 <string.h> 函数库，从而避免了不必要的代码重复。

2.2.6　总结

表 2-1 对 C 字符串和 SDS 之间的区别进行了总结。

表 2-1　C 字符串和 SDS 之间的区别

C 字符串	SDS
获取字符串长度的复杂度为 $O(N)$	获取字符串长度的复杂度为 $O(1)$
API 是不安全的，可能会造成缓冲区溢出	API 是安全的，不会造成缓冲区溢出
修改字符串长度 N 次必然需要执行 N 次内存重分配	修改字符串长度 N 次最多需要执行 N 次内存重分配
只能保存文本数据	可以保存文本或者二进制数据
可以使用所有 <string.h> 库中的函数	可以使用一部分 <string.h> 库中的函数

2.3　SDS API

表 2-2 列出了 SDS 的主要操作 API。

表 2-2　SDS 的主要操作 API

函　数	作　用	时间复杂度
sdsnew	创建一个包含给定 C 字符串的 SDS	$O(N)$，N 为给定 C 字符串的长度
sdsempty	创建一个不包含任何内容的空 SDS	$O(1)$
sdsfree	释放给定的 SDS	$O(N)$，N 为被释放 SDS 的长度
sdslen	返回 SDS 的已使用空间字节数	这个值可以通过读取 SDS 的 len 属性来直接获得，复杂度为 $O(1)$
sdsavail	返回 SDS 的未使用空间字节数	这个值可以通过读取 SDS 的 free 属性来直接获得，复杂度为 $O(1)$
sdsdup	创建一个给定 SDS 的副本（copy）	$O(N)$，N 为给定 SDS 的长度
sdsclear	清空 SDS 保存的字符串内容	因为惰性空间释放策略，复杂度为 $O(1)$
sdscat	将给定 C 字符串拼接到 SDS 字符串的末尾	$O(N)$，N 为被拼接 C 字符串的长度
sdscatsds	将给定 SDS 字符串拼接到另一个 SDS 字符串的末尾	$O(N)$，N 为被拼接 SDS 字符串的长度

（续）

函 数	作 用	时间复杂度
sdscpy	将给定的C字符串复制到SDS里面，覆盖SDS原有的字符串	$O(N)$，N为被复制C字符串的长度
sdsgrowzero	用空字符将SDS扩展至给定长度	$O(N)$，N为扩展新增的字节数
sdsrange	保留SDS给定区间内的数据，不在区间内的数据会被覆盖或清除	$O(N)$，N为被保留数据的字节数
sdstrim	接受一个SDS和一个C字符串作为参数，从SDS中移除所有在C字符串中出现过的字符	$O(N^2)$，N为给定C字符串的长度
sdscmp	对比两个SDS字符串是否相同	$O(N)$，N为两个SDS中较短的那个SDS的长度

2.4 重点回顾

- Redis只会使用C字符串作为字面量，在大多数情况下，Redis使用SDS（Simple Dynamic String，简单动态字符串）作为字符串表示。
- 比起C字符串，SDS具有以下优点：

1）常数复杂度获取字符串长度。
2）杜绝缓冲区溢出。
3）减少修改字符串长度时所需的内存重分配次数。
4）二进制安全。
5）兼容部分C字符串函数。

2.5 参考资料

- 《C语言接口与实现：创建可重用软件的技术》一书的第15章和第16章介绍了一个和SDS类似的通用字符串实现。
- 维基百科的Binary Safe词条（http://en.wikipedia.org/wiki/Binary-safe）和http://computer.yourdictionary.com/binary-safe给出了二进制安全的定义。
- 维基百科的Null-terminated string词条给出了空字符结尾字符串的定义，说明了这种表示的来源，以及C语言使用这种字符串表示的历史原因：http://en.wikipedia.org/wiki/Null-terminated_string
- 《C标准库》一书的第14章给出了<string.h>标准库所有API的介绍，以及这些API的基础实现。
- GNU C库的主页上提供了GNU C标准库的下载包，其中的/string文件夹包含了所有<string.h> API的完整实现：http://www.gnu.org/software/libc

第3章
链　表

链表提供了高效的节点重排能力，以及顺序性的节点访问方式，并且可以通过增删节点来灵活地调整链表的长度。

作为一种常用数据结构，链表内置在很多高级的编程语言里面，因为 Redis 使用的 C 语言并没有内置这种数据结构，所以 Redis 构建了自己的链表实现。

链表在 Redis 中的应用非常广泛，比如列表键的底层实现之一就是链表。当一个列表键包含了数量比较多的元素，又或者列表中包含的元素都是比较长的字符串时，Redis 就会使用链表作为列表键的底层实现。

举个例子，以下展示的 integers 列表键包含了从 1 到 1024 共一千零二十四个整数：

```
redis> LLEN integers
(integer) 1024

redis> LRANGE integers 0 10
1)"1"
2)"2"
3)"3"
4)"4"
5)"5"
6)"6"
7)"7"
8)"8"
9)"9"
10)"10"
11)"11"
```

integers 列表键的底层实现就是一个链表，链表中的每个节点都保存了一个整数值。

除了链表键之外，发布与订阅、慢查询、监视器等功能也用到了链表，Redis 服务器本身还使用链表来保存多个客户端的状态信息，以及使用链表来构建客户端输出缓冲区（output buffer），本书后续的章节将陆续对这些链表应用进行介绍。

本章接下来的内容将对 Redis 的链表实现进行介绍，并列出相应的链表和链表节点 API。

因为已经有很多优秀的算法书籍对链表的基本定义和相关算法进行了详细的讲解，所以本章不会介绍这些内容，如果不具备关于链表的基本知识的话，可以参考《算法：C 语言实

现（第1～4部分）》一书的3.3至3.5节，或者《数据结构与算法分析：C语言描述》一书的3.2节，又或者《算法导论（第三版）》一书的10.2节。

3.1 链表和链表节点的实现

每个链表节点使用一个adlist.h/listNode结构来表示：

```
typedef struct listNode {

    // 前置节点
    struct listNode *prev;

    // 后置节点
    struct listNode *next;

    // 节点的值
    void *value;

}listNode;
```

多个listNode可以通过prev和next指针组成双端链表，如图3-1所示。

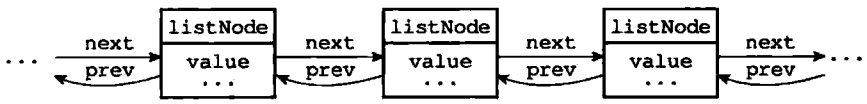

图3-1 由多个listNode组成的双端链表

虽然仅仅使用多个listNode结构就可以组成链表，但使用adlist.h/list来持有链表的话，操作起来会更方便：

```
typedef struct list {

    // 表头节点
    listNode *head;

    // 表尾节点
    listNode *tail;

    // 链表所包含的节点数量
    unsigned long len;

    // 节点值复制函数
    void *(*dup)(void *ptr);

    // 节点值释放函数
    void (*free)(void *ptr);

    // 节点值对比函数
    int (*match)(void *ptr,void *key);

} list;
```

list结构为链表提供了表头指针head、表尾指针tail，以及链表长度计数器len，而dup、free和match成员则是用于实现多态链表所需的类型特定函数：

- dup函数用于复制链表节点所保存的值；
- free函数用于释放链表节点所保存的值；

- match 函数则用于对比链表节点所保存的值和另一个输入值是否相等。

图 3-2 是由一个 list 结构和三个 listNode 结构组成的链表。

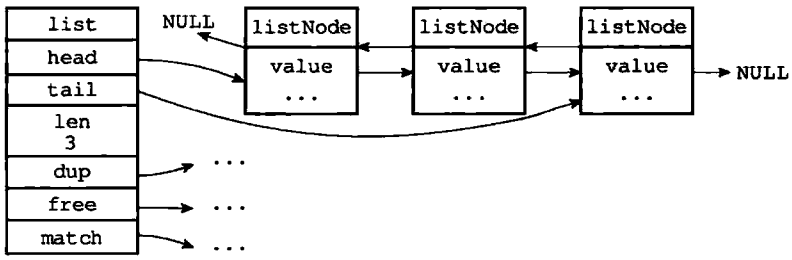

图 3-2 由 list 结构和 listNode 结构组成的链表

Redis 的链表实现的特性可以总结如下：

- 双端：链表节点带有 prev 和 next 指针，获取某个节点的前置节点和后置节点的复杂度都是 $O(1)$。
- 无环：表头节点的 prev 指针和表尾节点的 next 指针都指向 NULL，对链表的访问以 NULL 为终点。
- 带表头指针和表尾指针：通过 list 结构的 head 指针和 tail 指针，程序获取链表的表头节点和表尾节点的复杂度为 $O(1)$。
- 带链表长度计数器：程序使用 list 结构的 len 属性来对 list 持有的链表节点进行计数，程序获取链表中节点数量的复杂度为 $O(1)$。
- 多态：链表节点使用 void* 指针来保存节点值，并且可以通过 list 结构的 dup、free、match 三个属性为节点值设置类型特定函数，所以链表可以用于保存各种不同类型的值。

3.2 链表和链表节点的 API

表 3-1 列出了所有用于操作链表和链表节点的 API。

表 3-1 链表和链表节点 API

函数	作用	时间复杂度
listSetDupMethod	将给定的函数设置为链表的节点值复制函数	复制函数可以通过链表的 dup 属性直接获得，$O(1)$
listGetDupMethod	返回链表当前正在使用的节点值复制函数	$O(1)$
listSetFreeMethod	将给定的函数设置为链表的节点值释放函数	释放函数可以通过链表的 free 属性直接获得，$O(1)$
listGetFree	返回链表当前正在使用的节点值释放函数	$O(1)$
listSetMatchMethod	将给定的函数设置为链表的节点值对比函数	对比函数可以通过链表的 match 属性直接获得，$O(1)$
listGetMatchMethod	返回链表当前正在使用的节点值对比函数	$O(1)$

（续）

函数	作用	时间复杂度
listLength	返回链表的长度（包含了多少个节点）	链表长度可以通过链表的 len 属性直接获得，$O(1)$
listFirst	返回链表的表头节点	表头节点可以通过链表的 head 属性直接获得，$O(1)$
listLast	返回链表的表尾节点	表尾节点可以通过链表的 tail 属性直接获得，$O(1)$
listPrevNode	返回给定节点的前置节点	前置节点可以通过节点的 prev 属性直接获得，$O(1)$
listNextNode	返回给定节点的后置节点	后置节点可以通过节点的 next 属性直接获得，$O(1)$
listNodeValue	返回给定节点目前正在保存的值	节点值可以通过节点的 value 属性直接获得，$O(1)$
listCreate	创建一个不包含任何节点的新链表	$O(1)$
listAddNodeHead	将一个包含给定值的新节点添加到给定链表的表头	$O(1)$
listAddNodeTail	将一个包含给定值的新节点添加到给定链表的表尾	$O(1)$
listInsertNode	将一个包含给定值的新节点添加到给定节点的之前或者之后	$O(1)$
listSearchKey	查找并返回链表中包含给定值的节点	$O(N)$，N 为链表长度
listIndex	返回链表在给定索引上的节点	$O(N)$，N 为链表长度
listDelNode	从链表中删除给定节点	$O(N)$，N 为链表长度
listRotate	将链表的表尾节点弹出，然后将被弹出的节点插入到链表的表头，成为新的表头节点	$O(1)$
listDup	复制一个给定链表的副本	$O(N)$，N 为链表长度
listRelease	释放给定链表，以及链表中的所有节点	$O(N)$，N 为链表长度

3.3 重点回顾

- 链表被广泛用于实现 Redis 的各种功能，比如列表键、发布与订阅、慢查询、监视器等。
- 每个链表节点由一个 listNode 结构来表示，每个节点都有一个指向前置节点和后置节点的指针，所以 Redis 的链表实现是双端链表。
- 每个链表使用一个 list 结构来表示，这个结构带有表头节点指针、表尾节点指针，以及链表长度等信息。
- 因为链表表头节点的前置节点和表尾节点的后置节点都指向 NULL，所以 Redis 的链表实现是无环链表。
- 通过为链表设置不同的类型特定函数，Redis 的链表可以用于保存各种不同类型的值。

第4章
字　典

字典，又称为符号表（symbol table）、关联数组（associative array）或映射（map），是一种用于保存键值对（key-value pair）的抽象数据结构。

在字典中，一个键（key）可以和一个值（value）进行关联（或者说将键映射为值），这些关联的键和值就称为键值对。

字典中的每个键都是独一无二的，程序可以在字典中根据键查找与之关联的值，或者通过键来更新值，又或者根据键来删除整个键值对，等等。

字典经常作为一种数据结构内置在很多高级编程语言里面，但 Redis 所使用的 C 语言并没有内置这种数据结构，因此 Redis 构建了自己的字典实现。

字典在 Redis 中的应用相当广泛，比如 Redis 的数据库就是使用字典来作为底层实现的，对数据库的增、删、查、改操作也是构建在对字典的操作之上的。

举个例子，当我们执行命令：

```
redis> SET msg "hello world"
OK
```

在数据库中创建一个键为 "msg"，值为 "hello world" 的键值对时，这个键值对就是保存在代表数据库的字典里面的。

除了用来表示数据库之外，字典还是哈希键的底层实现之一，当一个哈希键包含的键值对比较多，又或者键值对中的元素都是比较长的字符串时，Redis 就会使用字典作为哈希键的底层实现。

举个例子，website 是一个包含 10086 个键值对的哈希键，这个哈希键的键都是一些数据库的名字，而键的值就是数据库的主页网址：

```
redis> HLEN website
(integer) 10086

redis> HGETALL website
1)"Redis"
```

```
2)"Redis.io"
3)"MariaDB"
4)"MariaDB.org"
5)"MongoDB"
6)"MongoDB.org"
# ...
```

website 键的底层实现就是一个字典，字典中包含了 10086 个键值对，例如：

- 键值对的键为 "Redis"，值为 "Redis.io"。
- 键值对的键为 "MariaDB"，值为 "MariaDB.org"；
- 键值对的键为 "MongoDB"，值为 "MongoDB.org"；

除了用来实现数据库和哈希键之外，Redis 的不少功能也用到了字典，在后续的章节中会不断地看到字典在 Redis 中的各种不同应用。

本章接下来的内容将对 Redis 的字典实现进行详细介绍，并列出字典的操作 API。本章不会对字典的基本定义和基础算法进行介绍，如果有需要的话，可以参考以下这些资料：

- 维基百科的 Associative Array 词条（http://en.wikipedia.org/wiki/Associative_array ）和 Hash Table 词条（http://en.wikipedia.org/wiki/Hash_table）。
- 《算法：C 语言实现（第 1～4 部分）》一书的第 14 章。
- 《算法导论（第三版）》一书的第 11 章。

4.1 字典的实现

Redis 的字典使用哈希表作为底层实现，一个哈希表里面可以有多个哈希表节点，而每个哈希表节点就保存了字典中的一个键值对。

接下来的三个小节将分别介绍 Redis 的哈希表、哈希表节点以及字典的实现。

4.1.1 哈希表

Redis 字典所使用的哈希表由 dict.h/dictht 结构定义：

```
typedef struct dictht {

    // 哈希表数组
    dictEntry **table;

    // 哈希表大小
    unsigned long size;

    // 哈希表大小掩码，用于计算索引值
    // 总是等于 size-1
    unsigned long sizemask;

    // 该哈希表已有节点的数量
    unsigned long used;

} dictht;
```

table 属性是一个数组，数组中的每个元素都是一个指向 dict.h/dictEntry 结构

的指针，每个 dictEntry 结构保存着一个键值对。size 属性记录了哈希表的大小，也即是 table 数组的大小，而 used 属性则记录了哈希表目前已有节点（键值对）的数量。sizemask 属性的值总是等于 size-1，这个属性和哈希值一起决定一个键应该被放到 table 数组的哪个索引上面。

图 4-1 展示了一个大小为 4 的空哈希表（没有包含任何键值对）。

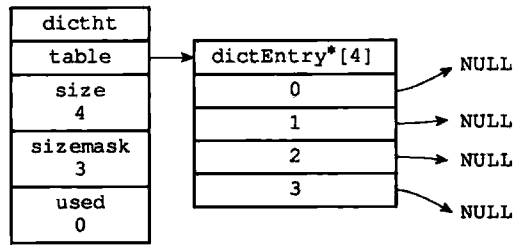

图 4-1 一个空的哈希表

4.1.2 哈希表节点

哈希表节点使用 dictEntry 结构表示，每个 dictEntry 结构都保存着一个键值对：

```
typedef struct dictEntry {

    // 键
    void *key;

    // 值
    union{
        void *val;
        uint64_tu64;
        int64_ts64;
    } v;

    // 指向下个哈希表节点，形成链表
    struct dictEntry *next;
} dictEntry;
```

key 属性保存着键值对中的键，而 v 属性则保存着键值对中的值，其中键值对的值可以是一个指针，或者是一个 uint64_t 整数，又或者是一个 int64_t 整数。

next 属性是指向另一个哈希表节点的指针，这个指针可以将多个哈希值相同的键值对连接在一次，以此来解决键冲突（collision）的问题。

举个例子，图 4-2 就展示了如何通过 next 指针，将两个索引值相同的键 k1 和 k0 连接在一起。

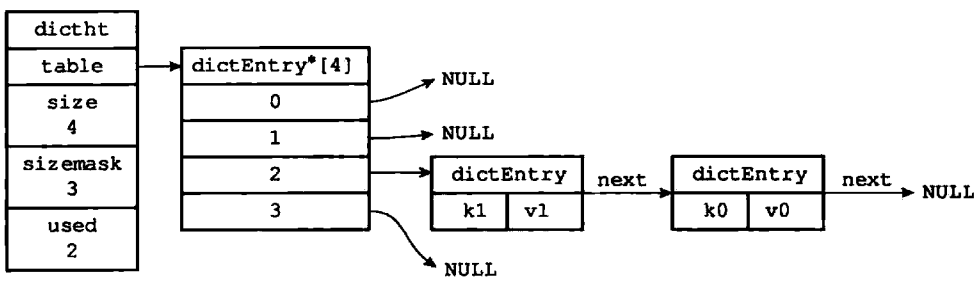

图 4-2 连接在一起的键 K1 和键 K0

4.1.3 字典

Redis 中的字典由 dict.h/dict 结构表示：

```
typedef struct dict {

    // 类型特定函数
    dictType *type;

    // 私有数据
    void *privdata;

    // 哈希表
    dictht ht[2];

    // rehash 索引
    // 当 rehash 不在进行时，值为 -1
    in trehashidx; /* rehashing not in progress if rehashidx == -1 */

} dict;
```

type 属性和 privdata 属性是针对不同类型的键值对，为创建多态字典而设置的：

- type 属性是一个指向 dictType 结构的指针，每个 dictType 结构保存了一簇用于操作特定类型键值对的函数，Redis 会为用途不同的字典设置不同的类型特定函数。
- 而 privdata 属性则保存了需要传给那些类型特定函数的可选参数。

```
typedef struct dictType {

    // 计算哈希值的函数
    unsigned int (*hashFunction)(const void *key);

    // 复制键的函数
    void *(*keyDup)(void *privdata, const void *key);

    // 复制值的函数
    void *(*valDup)(void *privdata, const void *obj);

    // 对比键的函数
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);

    // 销毁键的函数
    void (*keyDestructor)(void *privdata, void *key);

    // 销毁值的函数
    void (*valDestructor)(void *privdata, void *obj);

} dictType;
```

ht 属性是一个包含两个项的数组，数组中的每个项都是一个 dictht 哈希表，一般情况下，字典只使用 ht[0] 哈希表，ht[1] 哈希表只会在对 ht[0] 哈希表进行 rehash 时使用。

除了 ht[1] 之外，另一个和 rehash 有关的属性就是 rehashidx，它记录了 rehash 目前的进度，如果目前没有在进行 rehash，那么它的值为 -1。

图 4-3 展示了一个普通状态下（没有进行 rehash）的字典。

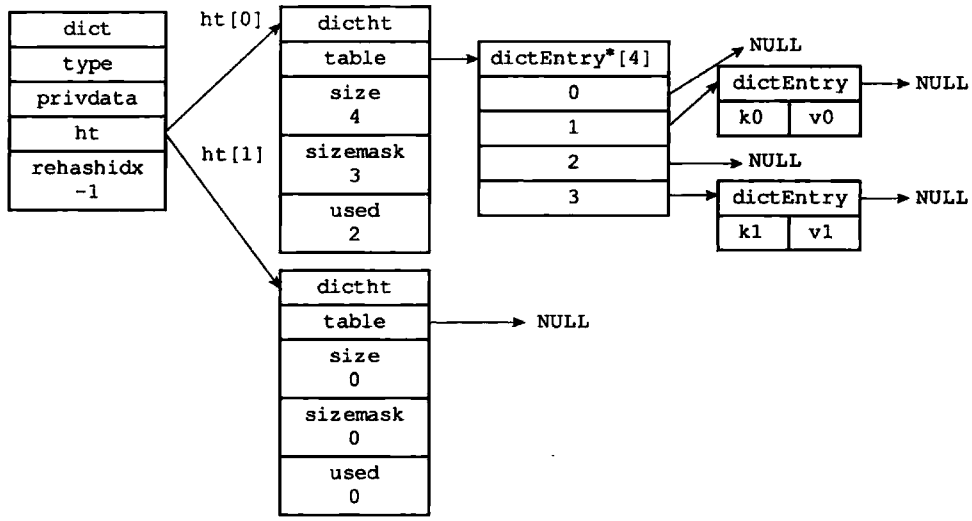

图 4-3 普通状态下的字典

4.2 哈希算法

当要将一个新的键值对添加到字典里面时，程序需要先根据键值对的键计算出哈希值和索引值，然后再根据索引值，将包含新键值对的哈希表节点放到哈希表数组的指定索引上面。

Redis 计算哈希值和索引值的方法如下：

```
# 使用字典设置的哈希函数，计算键 key 的哈希值
hash = dict->type->hashFunction(key);

# 使用哈希表的 sizemask 属性和哈希值，计算出索引值
# 根据情况不同，ht[x] 可以是 ht[0] 或者 ht[1]
index = hash & dict->ht[x].sizemask;
```

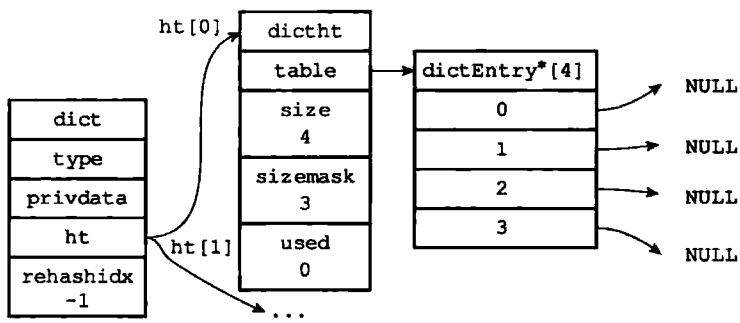

图 4-4 空字典

举个例子，对于图 4-4 所示的字典来说，如果我们要将一个键值对 k0 和 v0 添加到字典里面，那么程序会先使用语句：

```
hash = dict->type->hashFunction(k0);
```

计算键 k0 的哈希值。

假设计算得出的哈希值为 8，那么程序会继续使用语句：

```
index = hash&dict->ht[0].sizemask = 8 & 3 = 0;
```

计算出键 k0 的索引值 0，这表示包含键值对 k0 和 v0 的节点应该被放置到哈希表数组的索引 0 位置上，如图 4-5 所示。

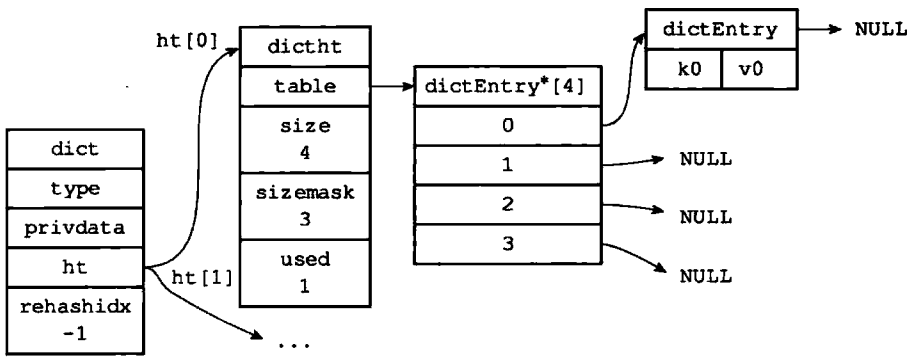

图 4-5 添加键值对 k0 和 v0 之后的字典

当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis 使用 MurmurHash2 算法来计算键的哈希值。

MurmurHash 算法最初由 Austin Appleby 于 2008 年发明，这种算法的优点在于，即使输入的键是有规律的，算法仍能给出一个很好的随机分布性，并且算法的计算速度也非常快。

MurmurHash 算法目前的最新版本为 MurmurHash3，而 Redis 使用的是 MurmurHash2，关于 MurmurHash 算法的更多信息可以参考该算法的主页：http://code.google.com/p/smhasher/。

4.3 解决键冲突

当有两个或以上数量的键被分配到了哈希表数组的同一个索引上面时，我们称这些键发生了冲突（collision）。

Redis 的哈希表使用链地址法（separate chaining）来解决键冲突，每个哈希表节点都有一个 next 指针，多个哈希表节点可以用 next 指针构成一个单向链表，被分配到同一个索引上的多个节点可以用这个单向链表连接起来，这就解决了键冲突的问题。

举个例子，假设程序要将键值对 k2 和 v2 添加到图 4-6 所示的哈希表里面，并且计算得出 k2 的索引值为 2，那么键 k1 和 k2 将产生冲突，而解决冲突的办法就是使用 next 指针将键 k2 和 k1 所在的节点连接起来，如图 4-7 所示。

因为 dictEntry 节点组成的链表没有指向链表表尾的指针，所以为了速度考虑，程序总是将新节点添加到链表的表头位置（复杂度为 $O(1)$），排在其他已有节点的前面。

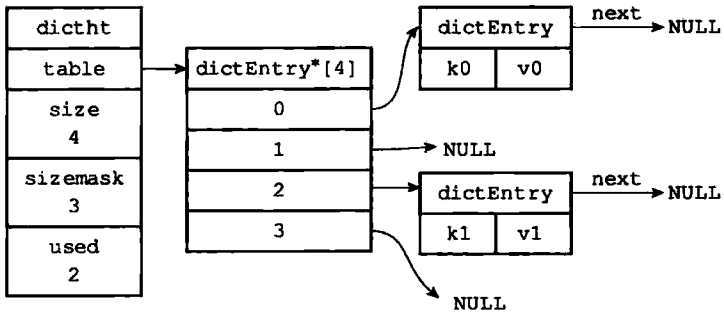

图 4-6 一个包含两个键值对的哈希表

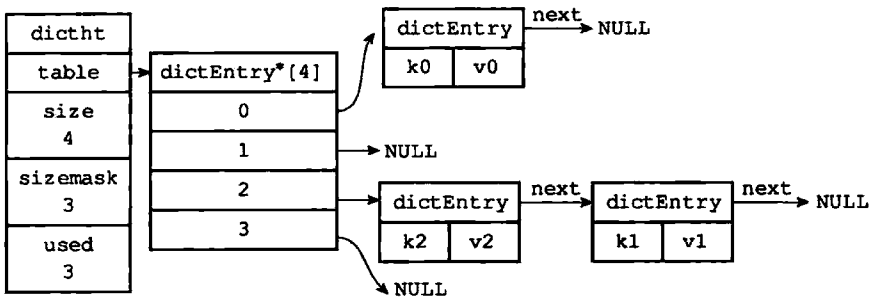

图 4-7 使用链表解决 k2 和 k1 的冲突

4.4 rehash

随着操作的不断执行，哈希表保存的键值对会逐渐地增多或者减少，为了让哈希表的负载因子（load factor）维持在一个合理的范围之内，当哈希表保存的键值对数量太多或者太少时，程序需要对哈希表的大小进行相应的扩展或者收缩。

扩展和收缩哈希表的工作可以通过执行 rehash（重新散列）操作来完成，Redis 对字典的哈希表执行 rehash 的步骤如下：

1）为字典的 ht[1] 哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及 ht[0] 当前包含的键值对数量（也即是 ht[0].used 属性的值）：

- 如果执行的是扩展操作，那么 ht[1] 的大小为第一个大于等于 ht[0].used*2 的 2^n（2 的 n 次方幂）；
- 如果执行的是收缩操作，那么 ht[1] 的大小为第一个大于等于 ht[0].used 的 2^n。

2）将保存在 ht[0] 中的所有键值对 rehash 到 ht[1] 上面：rehash 指的是重新计算键的哈希值和索引值，然后将键值对放置到 ht[1] 哈希表的指定位置上。

3）当 ht[0] 包含的所有键值对都迁移到了 ht[1] 之后（ht[0] 变为空表），释放 ht[0]，将 ht[1] 设置为 ht[0]，并在 ht[1] 新创建一个空白哈希表，为下一次 rehash 做准备。

举个例子，假设程序要对图 4-8 所示字典的 ht[0] 进行扩展操作，那么程序将执行以下步骤：

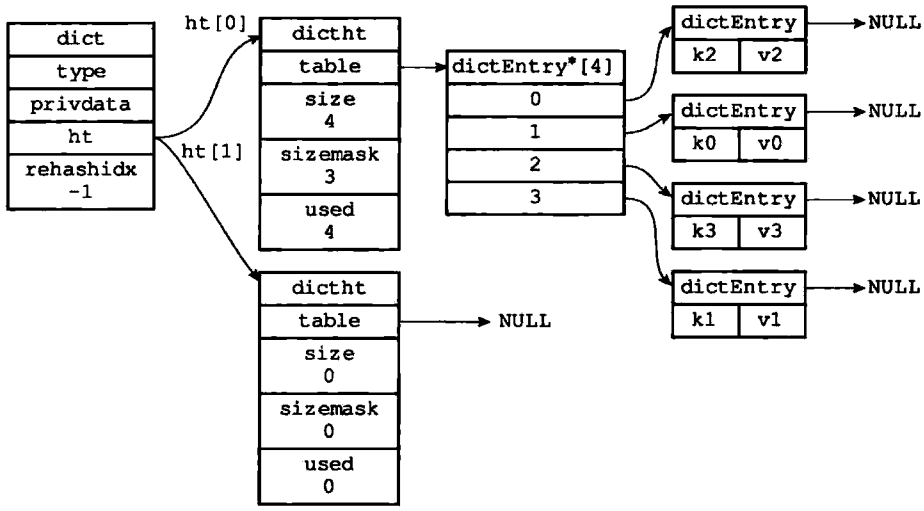

图 4-8 执行 rehash 之前的字典

1）ht[0].used 当前的值为 4，4 * 2=8，而 8（2^3）恰好是第一个大于等于 4 的 2 的 n 次方，所以程序会将 ht[1] 哈希表的大小设置为 8。图 4-9 展示了 ht[1] 在分配空间之后，字典的样子。

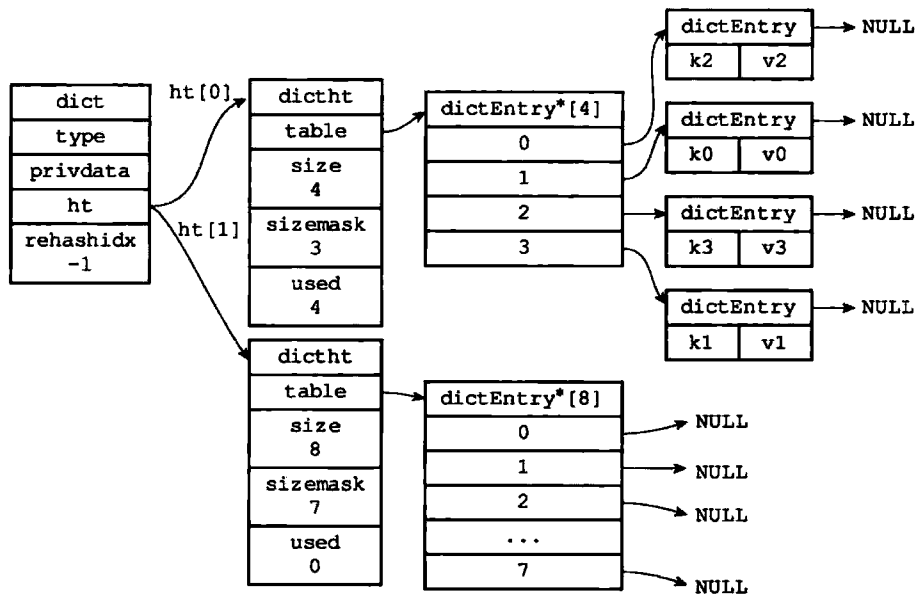

图 4-9 为字典的 ht[1] 哈希表分配空间

2）将 ht[0] 包含的四个键值对都 rehash 到 ht[1]，如图 4-10 所示。

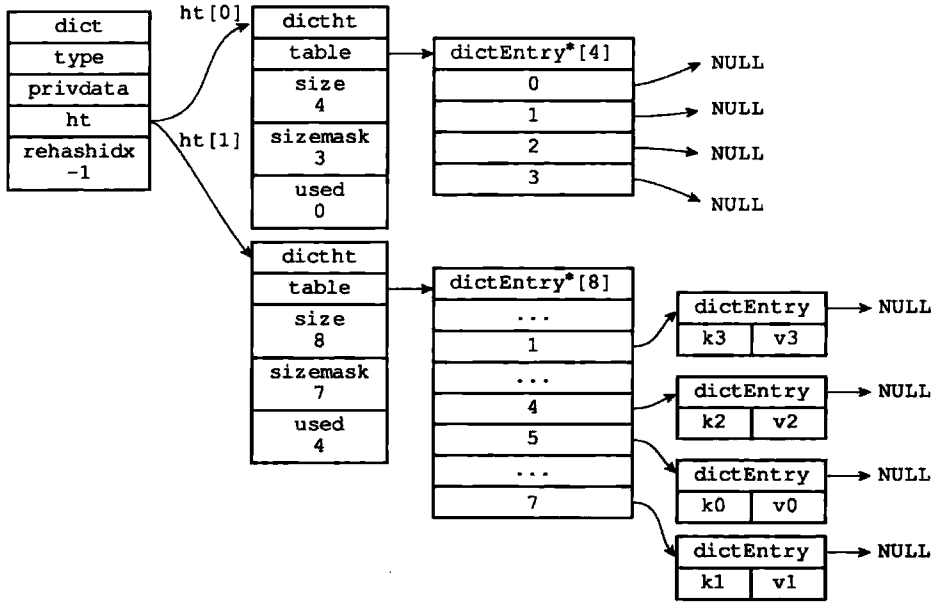

图 4-10 ht[0] 的所有键值对都已经被迁移到 ht[1]

3）释放 ht[0]，并将 ht[1] 设置为 ht[0]，然后为 ht[1] 分配一个空白哈希表，如图 4-11 所示。至此，对哈希表的扩展操作执行完毕，程序成功将哈希表的大小从原来的 4 改为了现在的 8。

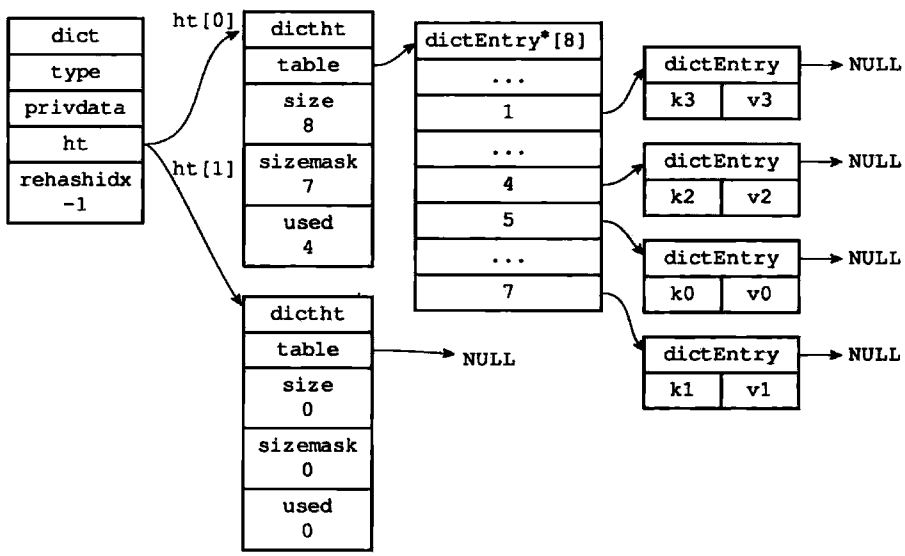

图 4-11 完成 rehash 之后的字典

哈希表的扩展与收缩

当以下条件中的任意一个被满足时，程序会自动开始对哈希表执行扩展操作：

1）服务器目前没有在执行 *BGSAVE* 命令或者 *BGREWRITEAOF* 命令，并且哈希表的负载因子大于等于 1。

2）服务器目前正在执行 *BGSAVE* 命令或者 *BGREWRITEAOF* 命令，并且哈希表的负载因子大于等于 5。

其中哈希表的负载因子可以通过公式：

```
# 负载因子 = 哈希表已保存节点数量 / 哈希表大小
load_factor = ht[0].used / ht[0].size
```

计算得出。

例如，对于一个大小为 4，包含 4 个键值对的哈希表来说，这个哈希表的负载因子为：

```
load_factor = 4 / 4 = 1
```

又例如，对于一个大小为 512，包含 256 个键值对的哈希表来说，这个哈希表的负载因子为：

```
load_factor = 256 / 512 = 0.5
```

根据 *BGSAVE* 命令或 *BGREWRITEAOF* 命令是否正在执行，服务器执行扩展操作所需的负载因子并不相同，这是因为在执行 *BGSAVE* 命令或 *BGREWRITEAOF* 命令的过程中，Redis需要创建当前服务器进程的子进程，而大多数操作系统都采用写时复制（copy-on-write）技术来优化子进程的使用效率，所以在子进程存在期间，服务器会提高执行扩展操作所需的负载因子，从而尽可能地避免在子进程存在期间进行哈希表扩展操作，这可以避免不必要的内存写入操作，最大限度地节约内存。

另一方面，当哈希表的负载因子小于 0.1 时，程序自动开始对哈希表执行收缩操作。

4.5 渐进式 rehash

上一节说过，扩展或收缩哈希表需要将 ht[0] 里面的所有键值对 rehash 到 ht[1] 里面，但是，这个 rehash 动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成的。

这样做的原因在于，如果 ht[0] 里只保存着四个键值对，那么服务器可以在瞬间就将这些键值对全部 rehash 到 ht[1]；但是，如果哈希表里保存的键值对数量不是四个，而是四百万、四千万甚至四亿个键值对，那么要一次性将这些键值对全部 rehash 到 ht[1] 的话，庞大的计算量可能会导致服务器在一段时间内停止服务。

因此，为了避免 rehash 对服务器性能造成影响，服务器不是一次性将 ht[0] 里面的所有键值对全部 rehash 到 ht[1]，而是分多次、渐进式地将 ht[0] 里面的键值对慢慢地 rehash 到 ht[1]。

以下是哈希表渐进式 rehash 的详细步骤：

1）为 ht[1] 分配空间，让字典同时持有 ht[0] 和 ht[1] 两个哈希表。

2）在字典中维持一个索引计数器变量 rehashidx，并将它的值设置为 0，表示 rehash 工作正式开始。

3）在 rehash 进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将 ht[0] 哈希表在 rehashidx 索引上的所有键值对 rehash 到 ht[1]，当 rehash 工作完成之后，程序将 rehashidx 属性的值增一。

4）随着字典操作的不断执行，最终在某个时间点上，ht[0] 的所有键值对都会被 rehash 至 ht[1]，这时程序将 rehashidx 属性的值设为 -1，表示 rehash 操作已完成。

渐进式 rehash 的好处在于它采取分而治之的方式，将 rehash 键值对所需的计算工作均摊到对字典的每个添加、删除、查找和更新操作上，从而避免了集中式 rehash 而带来的庞大计算量。

图 4-12 至图 4-17 展示了一次完整的渐进式 rehash 过程，注意观察在整个 rehash 过程中，字典的 rehashidx 属性是如何变化的。

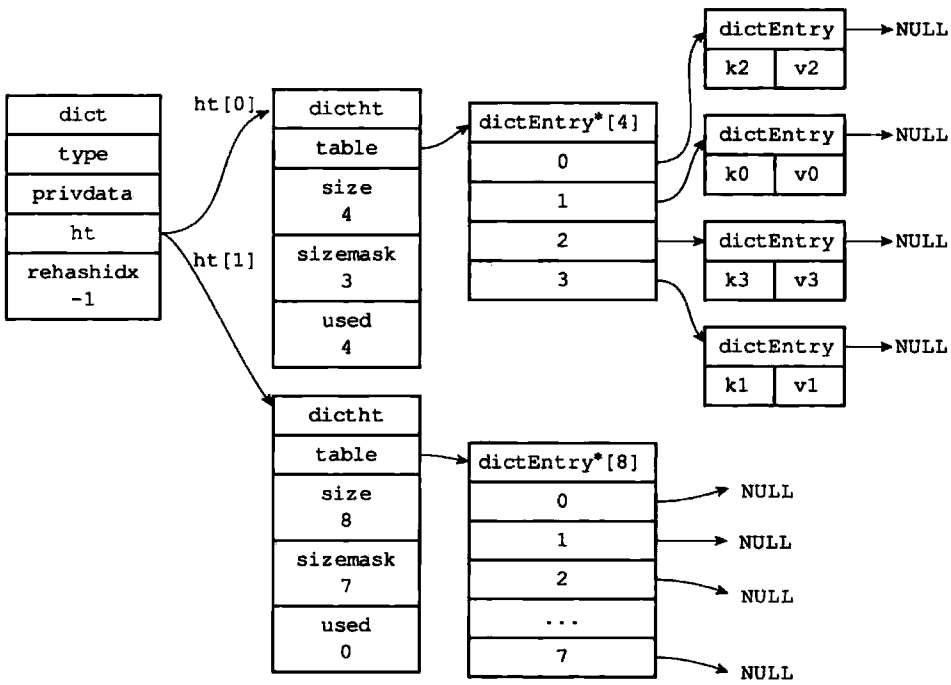

图 4-12 准备开始 rehash

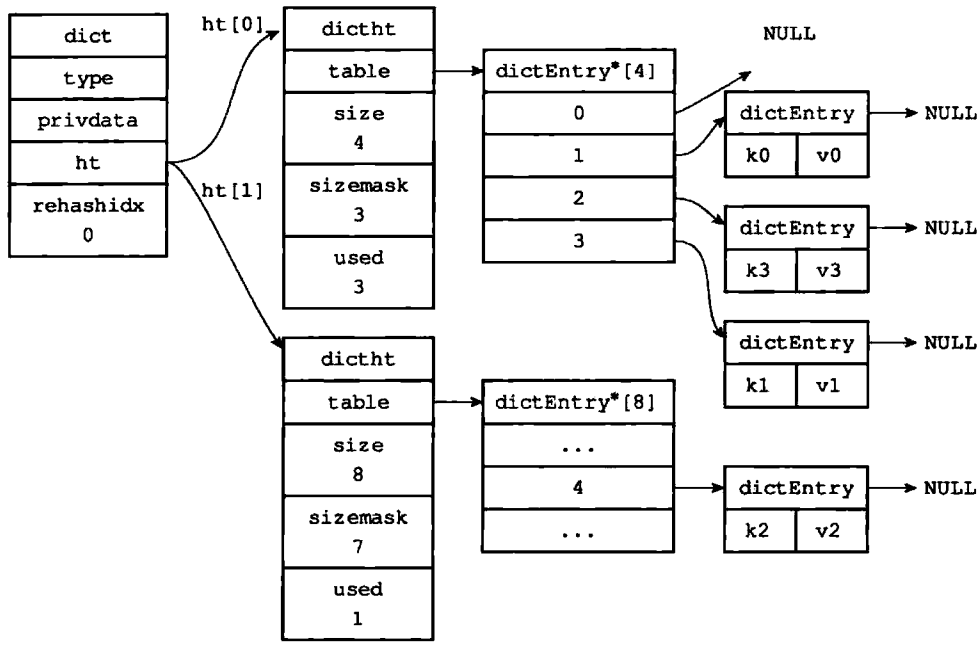

图 4-13 rehash 索引 0 上的键值对

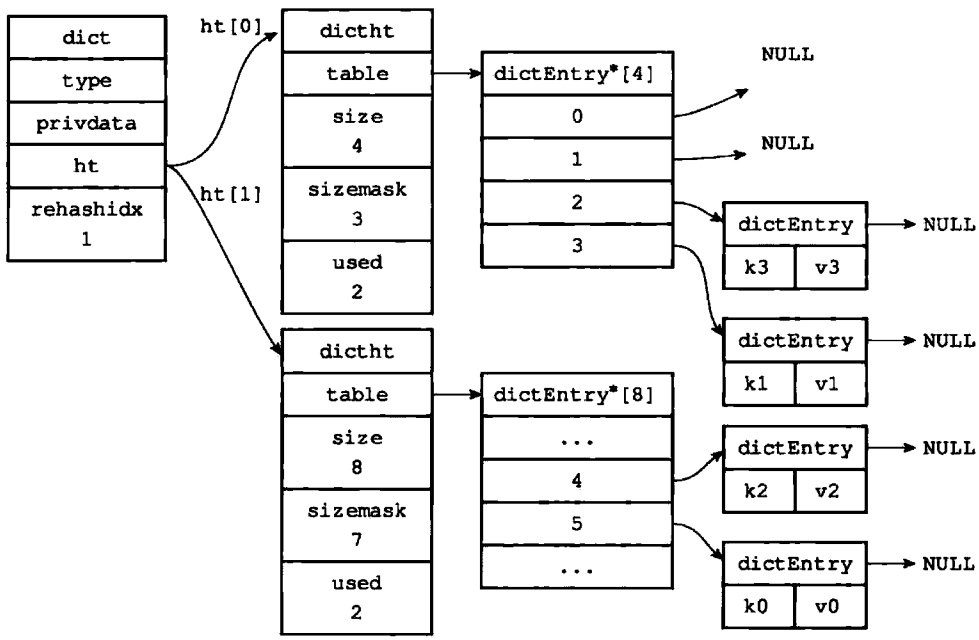

图 4-14 rehash 索引 1 上的键值对

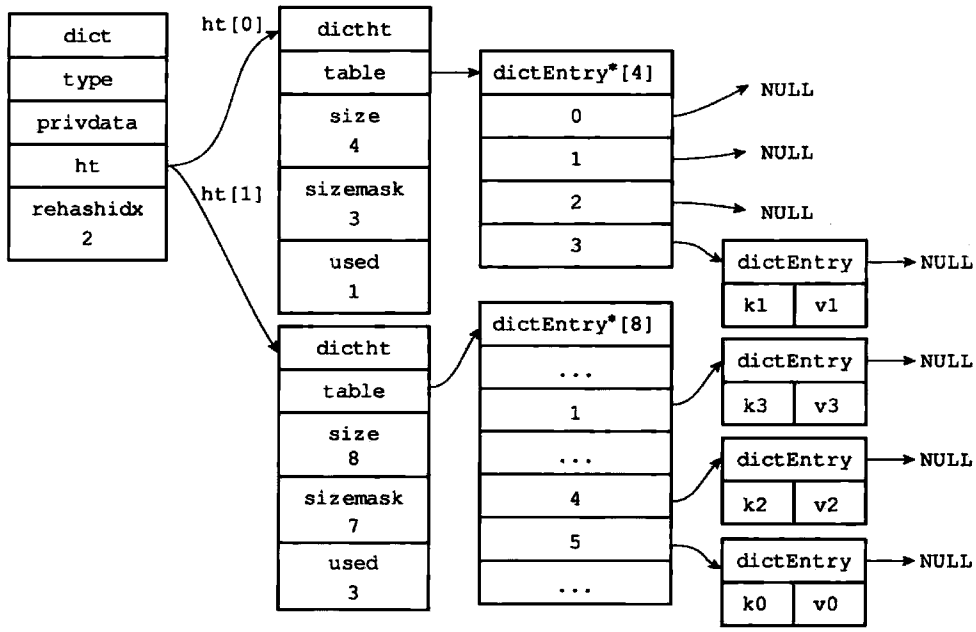

图 4-15 rehash 索引 2 上的键值对

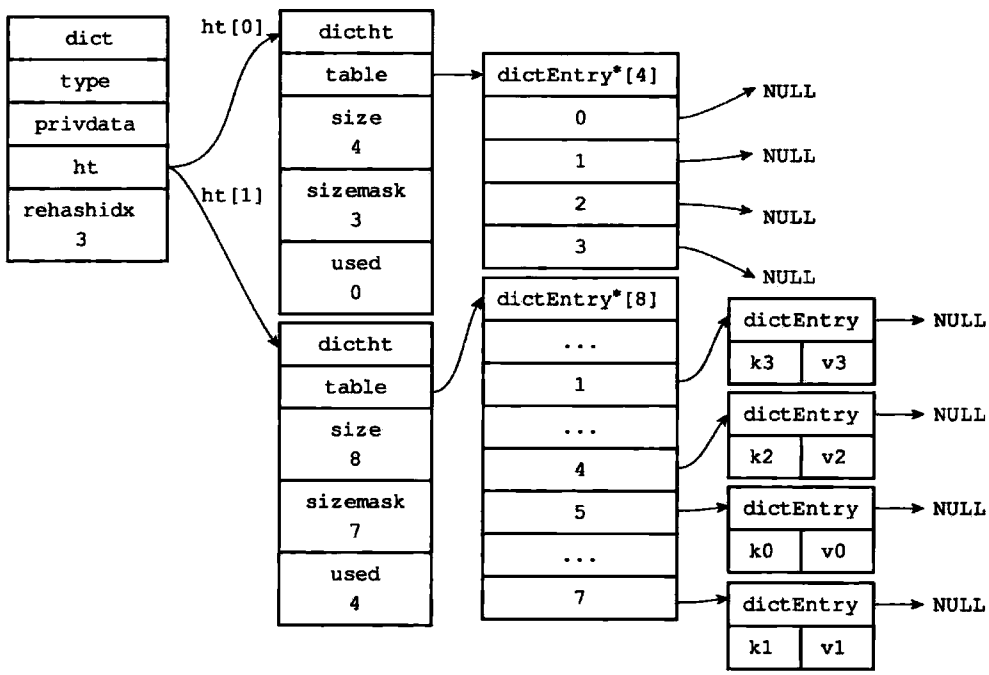

图 4-16 rehash 索引 3 上的键值对

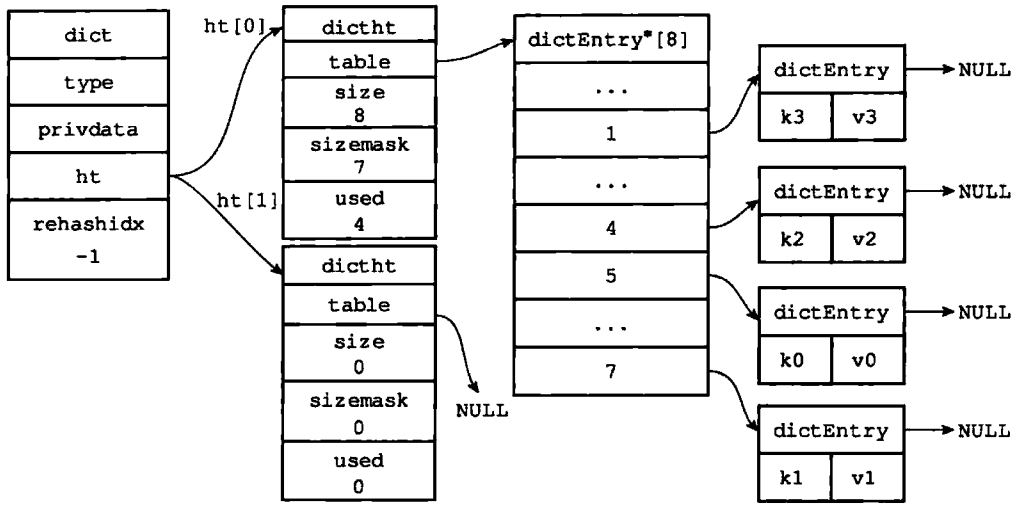

图 4-17 rehash 执行完毕

渐进式 rehash 执行期间的哈希表操作

因为在进行渐进式 rehash 的过程中，字典会同时使用 ht[0] 和 ht[1] 两个哈希表，所以在渐进式 rehash 进行期间，字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表上进行。例如，要在字典里面查找一个键的话，程序会先在 ht[0] 里面进行查找，如果没找到的话，就会继续到 ht[1] 里面进行查找，诸如此类。

另外，在渐进式 rehash 执行期间，新添加到字典的键值对一律会被保存到 ht[1] 里面，而 ht[0] 则不再进行任何添加操作，这一措施保证了 ht[0] 包含的键值对数量会只减不增，并随着 rehash 操作的执行而最终变成空表。

4.6 字典 API

表 4-1 列出了字典的主要操作 API。

表 4-1 字典的主要操作 API

函 数	作 用	时间复杂度
dictCreate	创建一个新的字典	$O(1)$
dictAdd	将给定的键值对添加到字典里面	$O(1)$
dictReplace	将给定的键值对添加到字典里面，如果键已经存在于字典，那么用新值取代原有的值	$O(1)$
dictFetchValue	返回给定键的值	$O(1)$
dictGetRandomKey	从字典中随机返回一个键值对	$O(1)$

（续）

函 数	作 用	时间复杂度
dictDelete	从字典中删除给定键所对应的键值对	$O(1)$
dictRelease	释放给定字典，以及字典中包含的所有键值对	$O(N)$，N为字典包含的键值对数量

4.7 重点回顾

- 字典被广泛用于实现Redis的各种功能，其中包括数据库和哈希键。
- Redis中的字典使用哈希表作为底层实现，每个字典带有两个哈希表，一个平时使用，另一个仅在进行rehash时使用。
- 当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用MurmurHash2算法来计算键的哈希值。
- 哈希表使用链地址法来解决键冲突，被分配到同一个索引上的多个键值对会连接成一个单向链表。
- 在对哈希表进行扩展或者收缩操作时，程序需要将现有哈希表包含的所有键值对rehash到新哈希表里面，并且这个rehash过程并不是一次性地完成的，而是渐进式地完成的。

第5章
跳 跃 表

跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。

跳跃表支持平均 $O(\log N)$、最坏 $O(N)$ 复杂度的节点查找，还可以通过顺序性操作来批量处理节点。

在大部分情况下，跳跃表的效率可以和平衡树相媲美，并且因为跳跃表的实现比平衡树要来得更为简单，所以有不少程序都使用跳跃表来代替平衡树。

Redis 使用跳跃表作为有序集合键的底层实现之一，如果一个有序集合包含的元素数量比较多，又或者有序集合中元素的成员（member）是比较长的字符串时，Redis 就会使用跳跃表来作为有序集合键的底层实现。

举个例子，`fruit-price` 是一个有序集合键，这个有序集合以水果名为成员，水果价钱为分值，保存了 130 款水果的价钱：

```
redis> ZRANGE fruit-price 0 2 WITHSCORES
1)"banana"
2)"5"
3)"cherry"
4)"6.5"
5)"apple"
6)"8"

redis> ZCARD fruit-price
(integer)130
```

`fruit-price` 有序集合的所有数据都保存在一个跳跃表里面，其中每个跳跃表节点（node）都保存了一款水果的价钱信息，所有水果按价钱的高低从低到高在跳跃表里面排序：

- 跳跃表的第一个元素的成员为 `"banana"`，它的分值为 `5`；
- 跳跃表的第二个元素的成员为 `"cherry"`，它的分值为 `6.5`；
- 跳跃表的第三个元素的成员为 `"apple"`，它的分值为 `8`；

和链表、字典等数据结构被广泛地应用在 Redis 内部不同，Redis 只在两个地方用到了跳跃表，一个是实现有序集合键，另一个是在集群节点中用作内部数据结构，除此之外，跳

跃表在 Redis 里面没有其他用途。本章将对 Redis 中的跳跃表实现进行介绍，并列出跳跃表的操作 API。本章不会对跳跃表的基本定义和基础算法进行介绍，如果有需要的话，可以参考 WilliamPugh 关于跳跃表的论文《Skip Lists: A Probabilistic Alternative to Balanced Trees》，或者《算法：C 语言实现（第 1～4 部分）》一书的 13.5 节。

5.1 跳跃表的实现

Redis 的跳跃表由 redis.h/zskiplistNode 和 redis.h/zskiplist 两个结构定义，其中 zskiplistNode 结构用于表示跳跃表节点，而 zskiplist 结构则用于保存跳跃表节点的相关信息，比如节点的数量，以及指向表头节点和表尾节点的指针等等。

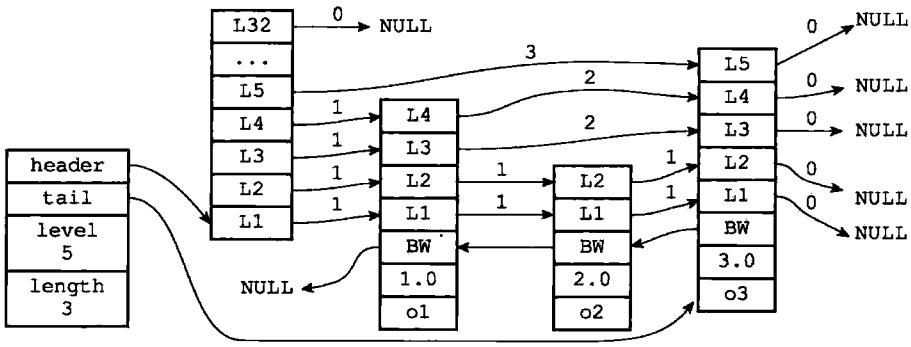

图 5-1 一个跳跃表

图 5-1 展示了一个跳跃表示例，位于图片最左边的是 zskiplist 结构，该结构包含以下属性：

- header：指向跳跃表的表头节点。
- tail：指向跳跃表的表尾节点。
- level：记录目前跳跃表内，层数最大的那个节点的层数（表头节点的层数不计算在内）。
- length：记录跳跃表的长度，也即是，跳跃表目前包含节点的数量（表头节点不计算在内）。

位于 zskiplist 结构右方的是四个 zskiplistNode 结构，该结构包含以下属性：

- 层（level）：节点中用 L1、L2、L3 等字样标记节点的各个层，L1 代表第一层，L2 代表第二层，以此类推。每个层都带有两个属性：前进指针和跨度。前进指针用于访问位于表尾方向的其他节点，而跨度则记录了前进指针所指向节点和当前节点的距离。在上面的图片中，连线上带有数字的箭头就代表前进指针，而那个数字就是跨度。当程序从表头向表尾进行遍历时，访问会沿着层的前进指针进行。
- 后退（backward）指针：节点中用 BW 字样标记节点的后退指针，它指向位于当前节点的前一个节点。后退指针在程序从表尾向表头遍历时使用。

- 分值（score）：各个节点中的 1.0、2.0 和 3.0 是节点所保存的分值。在跳跃表中，节点按各自所保存的分值从小到大排列。
- 成员对象（obj）：各个节点中的 o1、o2 和 o3 是节点所保存的成员对象。

注意表头节点和其他节点的构造是一样的：表头节点也有后退指针、分值和成员对象，不过表头节点的这些属性都不会被用到，所以图中省略了这些部分，只显示了表头节点的各个层。

本节接下来的内容将对 zskiplistNode 和 zskiplist 两个结构进行更详细的介绍。

5.1.1 跳跃表节点

跳跃表节点的实现由 redis.h/zskiplistNode 结构定义：

```
typedef struct zskiplistNode {

    // 层
    struct zskiplistLevel {

        // 前进指针
        struct zskiplistNode *forward;

        // 跨度
        unsigned int span;

    } level[];

    // 后退指针
    struct zskiplistNode *backward;

    // 分值
    double score;

    // 成员对象
    robj *obj;

} zskiplistNode;
```

1. 层

跳跃表节点的 level 数组可以包含多个元素，每个元素都包含一个指向其他节点的指针，程序可以通过这些层来加快访问其他节点的速度，一般来说，层的数量越多，访问其他节点的速度就越快。

每次创建一个新跳跃表节点的时候，程序都根据幂次定律（power law，越大的数出现的概率越小）随机生成一个介于 1 和 32 之间的值作为 level 数组的大小，这个大小就是层的“高度”。

图 5-2 分别展示了三个高度为 1 层、3 层和 5 层的节点，因为 C 语言的数组索引总是从 0 开始的，所以节点的第一层是 level[0]，而第二层是 level[1]，以此类推。

2. 前进指针

每个层都有一个指向表尾方向的前进指针（level[i].forward 属性），用于从表头向表尾方向访问节点。图 5-3 用虚线表示出了程序从表头向表尾方向，遍历跳跃表中所有节点的路径：

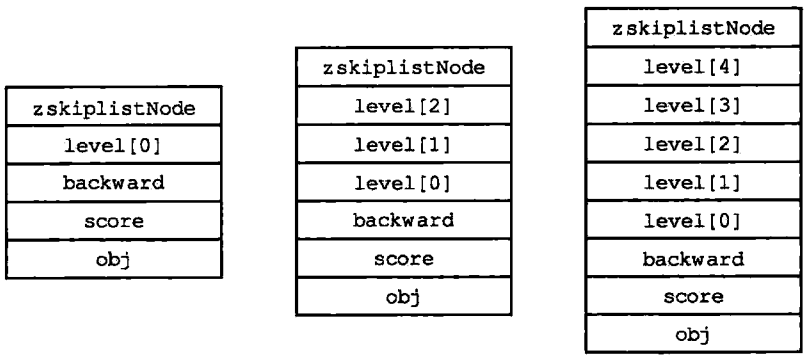

图 5-2 带有不同层高的节点

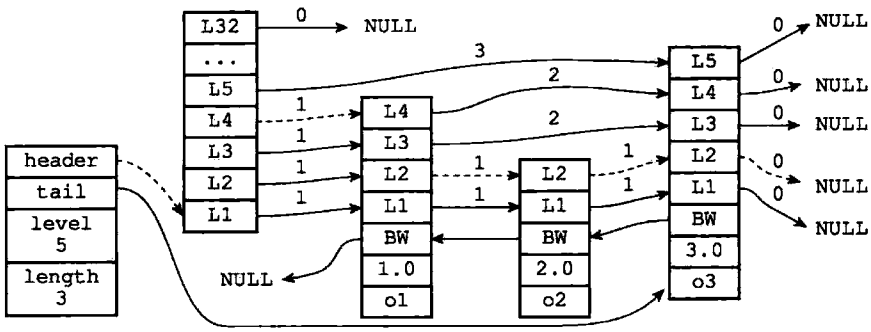

图 5-3 遍历整个跳跃表

1）迭代程序首先访问跳跃表的第一个节点（表头），然后从第四层的前进指针移动到表中的第二个节点。

2）在第二个节点时，程序沿着第二层的前进指针移动到表中的第三个节点。

3）在第三个节点时，程序同样沿着第二层的前进指针移动到表中的第四个节点。

4）当程序再次沿着第四个节点的前进指针移动时，它碰到一个 NULL，程序知道这时已经到达了跳跃表的表尾，于是结束这次遍历。

3. 跨度

层的跨度（level[i].span 属性）用于记录两个节点之间的距离：

- 两个节点之间的跨度越大，它们相距得就越远。
- 指向 NULL 的所有前进指针的跨度都为 0，因为它们没有连向任何节点。

初看上去，很容易以为跨度和遍历操作有关，但实际上并不是这样，遍历操作只使用前进指针就可以完成了，跨度实际上是用来计算排位（rank）的：在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点在跳跃表中的排位。

举个例子，图 5-4 用虚线标记了在跳跃表中查找分值为 3.0、成员对象为 o3 的节点时，沿途经历的层：查找的过程只经过了一个层，并且层的跨度为 3，所以目标节点在跳跃表中的排位为 3。

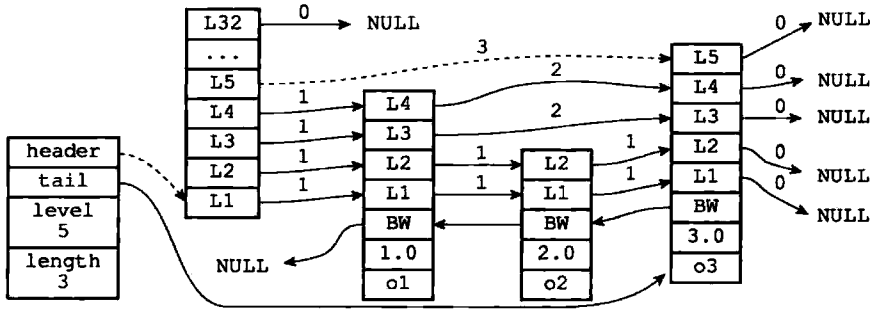

图 5-4 计算节点的排位

再举个例子，图 5-5 用虚线标记了在跳跃表中查找分值为 2.0、成员对象为 o2 的节点时，沿途经历的层：在查找节点的过程中，程序经过了两个跨度为 1 的节点，因此可以计算出，目标节点在跳跃表中的排位为 2。

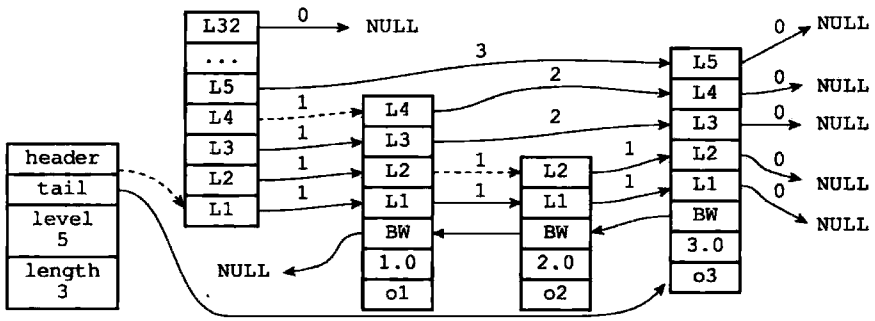

图 5-5 另一个计算节点排位的例子

4. 后退指针

节点的后退指针（backward 属性）用于从表尾向表头方向访问节点：跟可以一次跳过多个节点的前进指针不同，因为每个节点只有一个后退指针，所以每次只能后退至前一个节点。

图 5-6 用虚线展示了如果从表尾向表头遍历跳跃表中的所有节点：程序首先通过跳跃表的 tail 指针访问表尾节点，然后通过后退指针访问倒数第二个节点，之后再沿着后退指针访问倒数第三个节点，再之后遇到指向 NULL 的后退指针，于是访问结束。

5. 分值和成员

节点的分值（score 属性）是一个 double 类型的浮点数，跳跃表中的所有节点都按分值从小到大来排序。

节点的成员对象（obj 属性）是一个指针，它指向一个字符串对象，而字符串对象则保存着一个 SDS 值。

在同一个跳跃表中，各个节点保存的成员对象必须是唯一的，但是多个节点保存的分值却可以是相同的：分值相同的节点将按照成员对象在字典序中的大小来进行排序，成员对象

较小的节点会排在前面（靠近表头的方向），而成员对象较大的节点则会排在后面（靠近表尾的方向）。

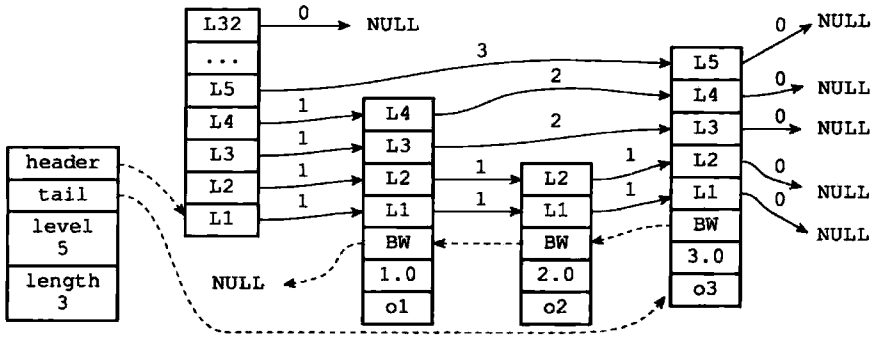

图 5-6 从表尾向表头方向遍历跳跃表

举个例子，在图 5-7 所示的跳跃表中，三个跳跃表节点都保存了相同的分值 10086.0，但保存成员对象 o1 的节点却排在保存成员对象 o2 和 o3 的节点之前，而保存成员对象 o2 的节点又排在保存成员对象 o3 的节点之前，由此可见，o1、o2、o3 三个成员对象在字典中的排序为 o1<=o2<=o3。

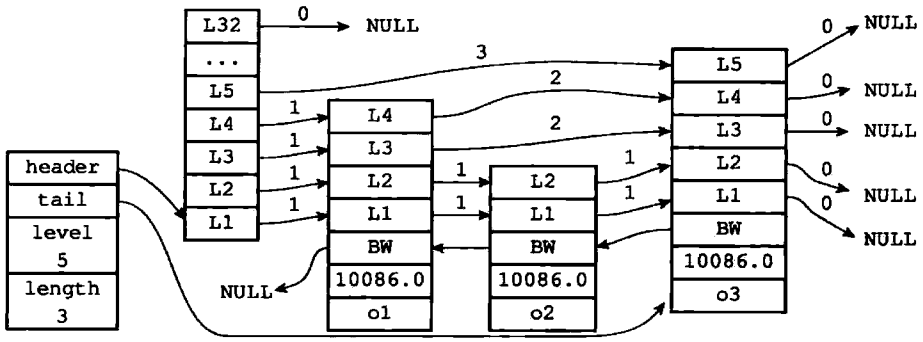

图 5-7 三个带有相同分值的跳跃表节点

5.1.2 跳跃表

仅靠多个跳跃表节点就可以组成一个跳跃表，如图 5-8 所示。

但通过使用一个 zskiplist 结构来持有这些节点，程序可以更方便地对整个跳跃表进行处理，比如快速访问跳跃表的表头节点和表尾节点，或者快速地获取跳跃表节点的数量（也即是跳跃表的长度）等信息，如图 5-9 所示。

zskiplist 结构的定义如下：

```
typedef struct zskiplist {

    // 表头节点和表尾节点
```

```
    structz skiplistNode *header, *tail;

    // 表中节点的数量
    unsigned long length;

    // 表中层数最大的节点的层数
    int level;

} zskiplist;
```

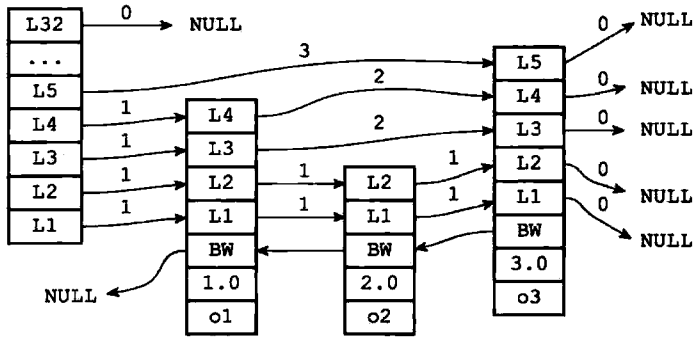

图 5-8 由多个跳跃节点组成的跳跃表

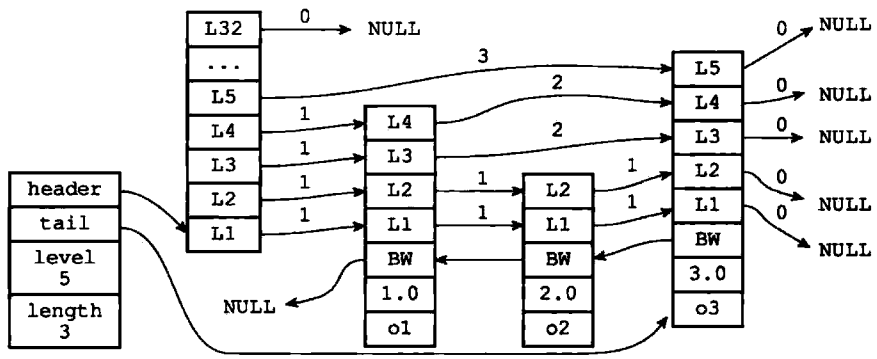

图 5-9 带有 zskiplist 结构的跳跃表

header 和 tail 指针分别指向跳跃表的表头和表尾节点，通过这两个指针，程序定位表头节点和表尾节点的复杂度为 $O(1)$。

通过使用 length 属性来记录节点的数量，程序可以在 $O(1)$ 复杂度内返回跳跃表的长度。

level 属性则用于在 $O(1)$ 复杂度内获取跳跃表中层高最大的那个节点的层数量，注意表头节点的层高并不计算在内。

5.2 跳跃表 API

表 5-1 列出了跳跃表的所有操作 API。

表 5-1 跳跃表 API

函 数	作 用	时间复杂度
zslCreate	创建一个新的跳跃表	$O(1)$
zslFree	释放给定跳跃表，以及表中包含的所有节点	$O(N)$，N 为跳跃表的长度
zslInsert	将包含给定成员和分值的新节点添加到跳跃表中	平均 $O(\log N)$，最坏 $O(N)$，N 为跳跃表长度
zslDelete	删除跳跃表中包含给定成员和分值的节点	平均 $O(\log N)$，最坏 $O(N)$，N 为跳跃表长度
zslGetRank	返回包含给定成员和分值的节点在跳跃表中的排位	平均 $O(\log N)$，最坏 $O(N)$，N 为跳跃表长度
zslGetElementByRank	返回跳跃表在给定排位上的节点	平均 $O(\log N)$，最坏 $O(N)$，N 为跳跃表长度
zslIsInRange	给定一个分值范围（range），比如 0 到 15，20 到 28，诸如此类，如果跳跃表中有至少一个节点的分值在这个范围之内，那么返回 1，否则返回 0	通过跳跃表的表头节点和表尾节点，这个检测可以用 $O(1)$ 复杂度完成
zslFirstInRange	给定一个分值范围，返回跳跃表中第一个符合这个范围的节点	平均 $O(\log N)$，最坏 $O(N)$。N 为跳跃表长度
zslLastInRange	给定一个分值范围，返回跳跃表中最后一个符合这个范围的节点	平均 $O(\log N)$，最坏 $O(N)$。N 为跳跃表长度
zslDeleteRangeByScore	给定一个分值范围，删除跳跃表中所有在这个范围之内的节点	$O(N)$，N 为被删除节点数量
zslDeleteRangeByRank	给定一个排位范围，删除跳跃表中所有在这个范围之内的节点	$O(N)$，N 为被删除节点数量

5.3 重点回顾

- 跳跃表是有序集合的底层实现之一。
- Redis 的跳跃表实现由 zskiplist 和 zskiplistNode 两个结构组成，其中 zskiplist 用于保存跳跃表信息（比如表头节点、表尾节点、长度），而 zskiplistNode 则用于表示跳跃表节点。
- 每个跳跃表节点的层高都是 1 至 32 之间的随机数。
- 在同一个跳跃表中，多个节点可以包含相同的分值，但每个节点的成员对象必须是唯一的。
- 跳跃表中的节点按照分值大小进行排序，当分值相同时，节点按照成员对象的大小进行排序。

第6章
整数集合

整数集合（intset）是集合键的底层实现之一，当一个集合只包含整数值元素，并且这个集合的元素数量不多时，Redis就会使用整数集合作为集合键的底层实现。

举个例子，如果我们创建一个只包含五个元素的集合键，并且集合中的所有元素都是整数值，那么这个集合键的底层实现就会是整数集合：

```
redis> SADD numbers 1 3 5 7 9
(integer) 5

redis> OBJECT ENCODING numbers
"intset"
```

在这一章，我们将对整数集合及其相关操作的实现原理进行介绍。

6.1 整数集合的实现

整数集合（intset）是Redis用于保存整数值的集合抽象数据结构，它可以保存类型为int16_t、int32_t或者int64_t的整数值，并且保证集合中不会出现重复元素。

每个intset.h/intset结构表示一个整数集合：

```
typedef struct intset {

    // 编码方式
    uint32_t encoding;

    // 集合包含的元素数量
    uint32_t length;

    // 保存元素的数组
    int8_t contents[];

} intset;
```

contents数组是整数集合的底层实现：整数集合的每个元素都是contents数组的一个数组项（item），各个项在数组中按值的大小从小到大有序地排列，并且数组中不包含任何重复项。

length属性记录了整数集合包含的元素数量，也即是contents数组的长度。

虽然intset结构将contents属性声明为int8_t类型的数组，但实际上contents数组并不保存任何int8_t类型的值，contents数组的真正类型取决于encoding属性的值：

- 如果encoding属性的值为INTSET_ENC_INT16，那么contents就是一个int16_t类型的数组，数组里的每个项都是一个int16_t类型的整数值（最小值为-32 768，最大值为32 767）。
- 如果encoding属性的值为INTSET_ENC_INT32，那么contents就是一个int32_t类型的数组，数组里的每个项都是一个int32_t类型的整数值（最小值为-2 147 483 648，最大值为2 147 483 647）。
- 如果encoding属性的值为INTSET_ENC_INT64，那么contents就是一个int64_t类型的数组，数组里的每个项都是一个int64_t类型的整数值（最小值为-9 223 372 036 854 775 808，最大值为9 223 372 036 854 775 807）。

图6-1展示了一个整数集合示例：

- encoding属性的值为INTSET_ENC_INT16，表示整数集合的底层实现为int16_t类型的数组，而集合保存的都是int16_t类型的整数值。
- length属性的值为5，表示整数集合包含五个元素。
- contents数组按从小到大的顺序保存着集合中的五个元素。

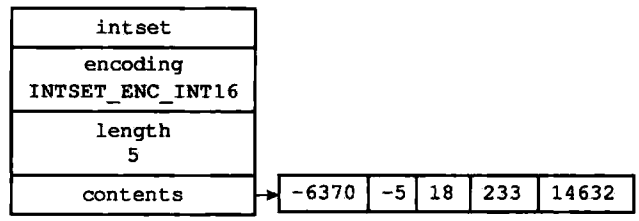

图6-1 一个包含五个int16_t类型整数值的整数集合

- 因为每个集合元素都是int16_t类型的整数值，所以contents数组的大小等于sizeof(int16_t)* 5 = 16 * 5 = 80位。

图6-2展示了另一个整数集合示例：

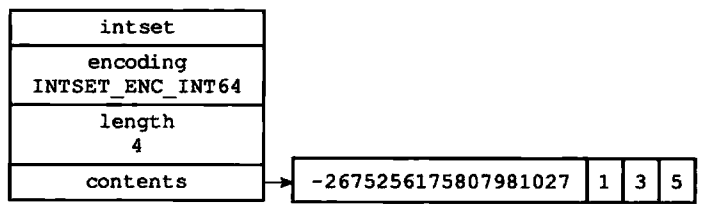

图6-2 一个包含四个int16_t类型整数值的整数集合

- encoding属性的值为INTSET_ENC_INT64，表示整数集合的底层实现为int64_t类型的数组，而数组中保存的都是int64_t类型的整数值。
- length属性的值为4，表示整数集合包含四个元素。
- contents数组按从小到大的顺序保存着集合中的四个元素。
- 因为每个集合元素都是int64_t类型的整数值，所以contents数组的大小为sizeof(int64_t)* 4 = 64 * 4 = 256位。

虽然contents数组保存的四个整数值中，只有-2 675 256 175 807 981 027是真正需要用int64_t类型来保存的，而其他的1、3、5三个值都可以用int16_t类型来保存，不过根据整数集合的升级规则，当向一个底层为int16_t数组的整数集合添加一个int64_t类型的整数值时，整数集合已有的所有元素都会被转换成int64_t类型，所以contents数组保存的四个整数值都是int64_t类型的，不仅仅是-2 675 256 175 807 981 027。

接下来的一节将对整数集合的升级操作进行详细介绍。

6.2 升级

每当我们要将一个新元素添加到整数集合里面，并且新元素的类型比整数集合现有所有元素的类型都要长时，整数集合需要先进行升级（upgrade），然后才能将新元素添加到整数集合里面。

升级整数集合并添加新元素共分为三步进行：

1）根据新元素的类型，扩展整数集合底层数组的空间大小，并为新元素分配空间。

2）将底层数组现有的所有元素都转换成与新元素相同的类型，并将类型转换后的元素放置到正确的位上，而且在放置元素的过程中，需要继续维持底层数组的有序性质不变。

3）将新元素添加到底层数组里面。

举个例子，假设现在有一个INTSET_ENC_INT16编码的整数集合，集合中包含三个int16_t类型的元素，如图6-3所示。

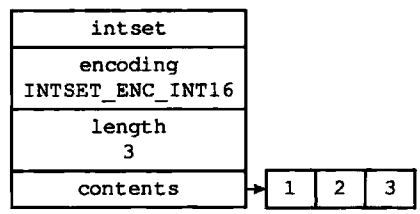

图6-3 一个包含三个int16_t类型的元素的整数集合

因为每个元素都占用16位空间，所以整数集合底层数组的大小为3*16=48位，图6-4展示了整数集合的三个元素在这48位里的位置。

位	0至15位	16至31位	32至47位
元素	1	2	3

图6-4 contents数组的各个元素，以及它们所在的位

现在，假设我们要将类型为int32_t的整数值65 535添加到整数集合里面，因为65 535的类型int32_t比整数集合当前所有元素的类型都要长，所以在将65 535添加到整数集合之前，程序需要先对整数集合进行升级。

升级首先要做的是，根据新类型的长度，以及集合元素的数量（包括要添加的新元素在内），对底层数组进行空间重分配。

整数集合目前有三个元素，再加上新元素65 535，整数集合需要分配四个元素的空间，因为每个int32_t整数值需要占用32位空间，所以在空间重分配之后，底层数组的大小将是32 * 4 = 128位，如图6-5所示。虽然程序对底层数组进行了空间重分配，但数组原有的三个元素1、2、3仍然是int16_t类型，这些元素还保存在数组的前48位里面，

所以程序接下来要做的就是将这三个元素转换成 int32_t 类型，并将转换后的元素放置到正确的位上面，而且在放置元素的过程中，需要维持底层数组的有序性质不变。

位	0至15位	16至31位	32至47位	48至127位
元素	1	2	3	（新分配空间）

图 6-5 进行空间重分配之后的数组

首先，因为元素 3 在 1、2、3、65535 四个元素中排名第三，所以它将被移动到 contents 数组的索引 2 位置上，也即是数组 64 位至 95 位的空间内，如图 6-6 所示。

位	0至15位	16至31位	32至47位	48至63位	64位至95位	96位至127位
元素	1	2	3	（新分配空间）	3	（新分配空间）

从int16_t类型转换为int32_t类型

图 6-6 对元素 3 进行类型转换，并保存在适当的位上

接着，因为元素 2 在 1、2、3、65535 四个元素中排名第二，所以它将被移动到 contents 数组的索引 1 位置上，也即是数组的 32 位至 63 位的空间内，如图 6-7 所示。

位	0至15位	16至31位	32至63位	64位至95位	96位至127位
元素	1	2	2	3	（新分配空间）

从int16_t类型转换为int32_t类型

图 6-7 对元素 2 进行类型转换，并保存在适当的位上

之后，因为元素 1 在 1、2、3、65535 四个元素中排名第一，所以它将被移动到 contents 数组的索引 0 位置上，即数组的 0 位至 31 位的空间内，如图 6-8 所示。

位	0至31位	32至63位	64位至95位	96位至127位
元素	1	2	3	（新分配空间）

从 int16_t 类型转换为 int32_t 类型

图 6-8 对元素 1 进行类型转换，并保存在适当的位上

然后，因为元素 65535 在 1、2、3、65535 四个元素中排名第四，所以它将被添加到 contents 数组的索引 3 位置上，也即是数组的 96 位至 127 位的空间内，如图 6-9 所示。

位	0至31位	32至63位	64位至95位	96位至127位
元素	1	2	3	65535

添加新元素

图 6-9 添加 65535 到数组

最后，程序将整数集合 encoding 属性的值从 INTSET_ENC_INT16 改为 INTSET_ENC_INT32，并将 length 属性的值从 3 改为 4，设置完成之后的整数集合如图 6-10 所示。

因为每次向整数集合添加新元素都可能会引起升级，而每次升级都需要对底层数组中已有的所有元素进行类型转换，所以向整数集合添加新元素的时间复杂度为 $O(N)$。

其他类型的升级操作，比如从 INTSET_ENC_INT16 编码升级为 INTSET_ENC_INT64 编码，或者从 INTSET_ENC_INT32 编码升级为 INTSET_ENC_INT64 编码，升级的过程都和上面展示的升级过程类似。

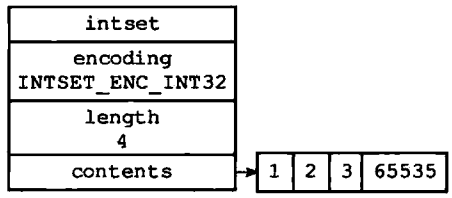

图 6-10 完成添加操作之后的整数集合

升级之后新元素的摆放位置

因为引发升级的新元素的长度总是比整数集合现有所有元素的长度都大，所以这个新元素的值要么就大于所有现有元素，要么就小于所有现有元素：

- 在新元素小于所有现有元素的情况下，新元素会被放置在底层数组的最开头（索引 0）；
- 在新元素大于所有现有元素的情况下，新元素会被放置在底层数组的最末尾（索引 length-1）。

6.3 升级的好处

整数集合的升级策略有两个好处，一个是提升整数集合的灵活性，另一个是尽可能地节约内存。

6.3.1 提升灵活性

因为 C 语言是静态类型语言，为了避免类型错误，我们通常不会将两种不同类型的值放在同一个数据结构里面。

例如，我们一般只使用 int16_t 类型的数组来保存 int16_t 类型的值，只使用 int32_t 类型的数组来保存 int32_t 类型的值，诸如此类。

但是，因为整数集合可以通过自动升级底层数组来适应新元素，所以我们可以随意地将 int16_t、int32_t 或者 int64_t 类型的整数添加到集合中，而不必担心出现类型错误，这种做法非常灵活。

6.3.2 节约内存

当然，要让一个数组可以同时保存 int16_t、int32_t、int64_t 三种类型的值，最简单的做法就是直接使用 int64_t 类型的数组作为整数集合的底层实现。不过这样一来，即使添加到整数集合里面的都是 int16_t 类型或者 int32_t 类型的值，数组都需要使用 int64_t 类型的空间去保存它们，从而出现浪费内存的情况。

而整数集合现在的做法既可以让集合能同时保存三种不同类型的值，又可以确保升级操作只会在有需要的时候进行，这可以尽量节省内存。

例如，如果我们一直只向整数集合添加 int16_t 类型的值，那么整数集合的底层实现就会一直是 int16_t 类型的数组，只有在我们要将 int32_t 类型或者 int64_t 类型的值添加到集合时，程序才会对数组进行升级。

6.4 降级

整数集合不支持降级操作，一旦对数组进行了升级，编码就会一直保持升级后的状态。

举个例子，对于图 6-11 所示的整数集合来说，即使我们将集合里唯一一个真正需要使用 int64_t 类型来保存的元素 4 294 967 295 删除了，整数集合的编码仍然会维持 INTSET_ENC_INT64，底层数组也仍然会是 int64_t 类型的，如图 6-12 所示。

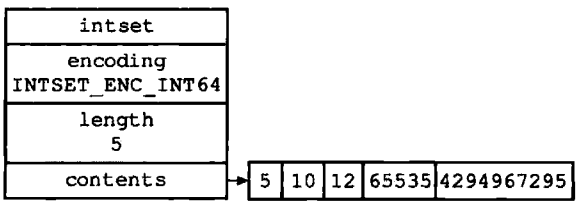

图 6-11 数组编码为 INTSET_ENC_INT64 的整数集合

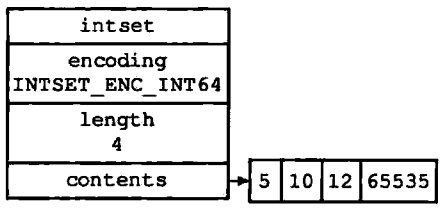

图 6-12 删除 4 294 967 295 的整数集合

6.5 整数集合 API

表 6-1 列出了整数集合的操作 API。

表 6-1 整数集合 API

函 数	作 用	时间复杂度
intsetNew	创建一个新的压缩列表	$O(1)$
intsetAdd	将给定元素添加到整数集合里面	$O(N)$
intsetRemove	从整数集合中移除给定元素	$O(N)$
intsetFind	检查给定值是否存在于集合	因为底层数组有序，查找可以通过二分查找法来进行，所以复杂度为 $O(\log N)$
intsetRandom	从整数集合中随机返回一个元素	$O(1)$
intsetGet	取出底层数组在给定索引上的元素	$O(1)$
intsetLen	返回整数集合包含的元素个数	$O(1)$
intsetBlobLen	返回整数集合占用的内存字节数	$O(1)$

6.6 重点回顾

- 整数集合是集合键的底层实现之一。
- 整数集合的底层实现为数组，这个数组以有序、无重复的方式保存集合元素，在有需要时，程序会根据新添加元素的类型，改变这个数组的类型。
- 升级操作为整数集合带来了操作上的灵活性，并且尽可能地节约了内存。
- 整数集合只支持升级操作，不支持降级操作。

第 7 章

压 缩 列 表

压缩列表（ziplist）是列表键和哈希键的底层实现之一。当一个列表键只包含少量列表项，并且每个列表项要么就是小整数值，要么就是长度比较短的字符串，那么 Redis 就会使用压缩列表来做列表键的底层实现。

例如，执行以下命令将创建一个压缩列表实现的列表键：

```
redis> RPUSH lst 1 3 5 10086 "hello" "world"
(integer)6

redis> OBJECT ENCODING lst
"ziplist"
```

列表键里面包含的都是 1、3、5、10086 这样的小整数值，以及 "hello"、"world" 这样的短字符串。

另外，当一个哈希键只包含少量键值对，比且每个键值对的键和值要么就是小整数值，要么就是长度比较短的字符串，那么 Redis 就会使用压缩列表来做哈希键的底层实现。

举个例子，执行以下命令将创建一个压缩列表实现的哈希键：

```
redis> HMSET profile "name" "Jack" "age" 28 "job" "Programmer"
OK

redis> OBJECT ENCODING profile
"ziplist"
```

哈希键里面包含的所有键和值都是小整数值或者短字符串。本章将对压缩列表的定义以及相关操作进行详细的介绍。

7.1 压缩列表的构成

压缩列表是 Redis 为了节约内存而开发的，是由一系列特殊编码的连续内存块组成的顺序型（sequential）数据结构。一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值。

图 7-1 展示了压缩列表的各个组成部分，表 7-1 则记录了各个组成部分的类型、长度以

及用途。

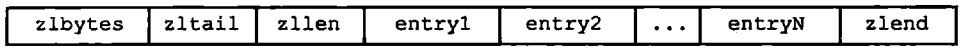

图 7-1 压缩列表的各个组成部分

表 7-1 压缩列表各个组成部分的详细说明

属性	类型	长度	用途
zlbytes	uint32_t	4字节	记录整个压缩列表占用的内存字节数：在对压缩列表进行内存重分配，或者计算 zlend 的位置时使用
zltail	uint32_t	4字节	记录压缩列表表尾节点距离压缩列表的起始地址有多少字节：通过这个偏移量，程序无须遍历整个压缩列表就可以确定表尾节点的地址
zllen	uint16_t	2字节	记录了压缩列表包含的节点数量：当这个属性的值小于 UINT16_MAX（65535）时，这个属性的值就是压缩列表包含节点的数量；当这个值等于 UINT16_MAX 时，节点的真实数量需要遍历整个压缩列表才能计算得出
entryX	列表节点	不定	压缩列表包含的各个节点，节点的长度由节点保存的内容决定
zlend	uint8_t	1字节	特殊值 0xFF（十进制 255），用于标记压缩列表的末端

图 7-2 展示了一个压缩列表示例：

- 列表 zlbytes 属性的值为 0x50（十进制 80），表示压缩列表的总长为 80 字节。
- 列表 zltail 属性的值为 0x3c（十进制 60），这表示如果我们有一个指向压缩列表起始地址的指针 p，那么只要用指针 p 加上偏移量 60，就可以计算出表尾节点 entry3 的地址。
- 列表 zllen 属性的值为 0x3（十进制 3），表示压缩列表包含三个节点。

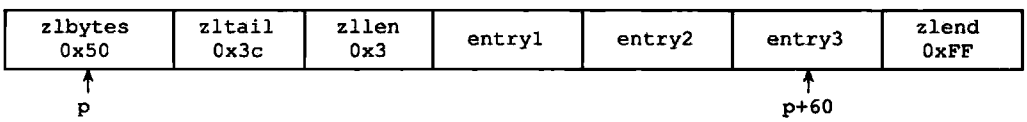

图 7-2 包含三个节点的压缩列表

图 7-3 展示了另一个压缩列表示例：

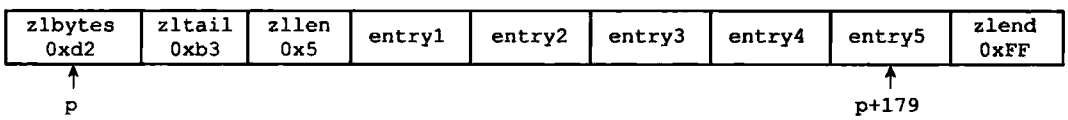

图 7-3 包含五个节点的压缩列表

- 列表 zlbytes 属性的值为 0xd2（十进制 210），表示压缩列表的总长为 210 字节。
- 列表 zltail 属性的值为 0xb3（十进制 179），这表示如果我们有一个指向压缩列表起始地址的指针 p，那么只要用指针 p 加上偏移量 179，就可以计算出表尾节点 entry5 的地址。
- 列表 zllen 属性的值为 0x5（十进制 5），表示压缩列表包含五个节点。

7.2 压缩列表节点的构成

每个压缩列表节点可以保存一个字节数组或者一个整数值，其中，字节数组可以是以下三种长度的其中一种：

- 长度小于等于 63（2^6-1）字节的字节数组；
- 长度小于等于 16 383（2^{14}-1）字节的字节数组；
- 长度小于等于 4 294 967 295（2^{32}-1）字节的字节数组；

而整数值则可以是以下六种长度的其中一种：

- 4 位长，介于 0 至 12 之间的无符号整数；
- 1 字节长的有符号整数；
- 3 字节长的有符号整数；
- int16_t 类型整数；
- int32_t 类型整数；
- int64_t 类型整数。

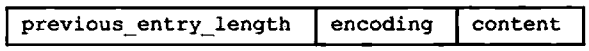

图 7-4 压缩列表节点的各个组成部分

每个压缩列表节点都由 previous_entry_length、encoding、content 三个部分组成，如图 7-4 所示。

接下来的内容将分别介绍这三个组成部分。

7.2.1 previous_entry_length

节点的 previous_entry_length 属性以字节为单位，记录了压缩列表中前一个节点的长度。previous_entry_length 属性的长度可以是 1 字节或者 5 字节：

- 如果前一节点的长度小于 254 字节，那么 previous_entry_length 属性的长度为 1 字节：前一节点的长度就保存在这一个字节里面。
- 如果前一节点的长度大于等于 254 字节，那么 previous_entry_length 属性的长度为 5 字节：其中属性的第一字节会被设置为 0xFE（十进制值 254），而之后的四个字节则用于保存前一节点的长度。

图 7-5 展示了一个包含一字节长 previous_entry_length 属性的压缩列表节点，属性的值为 0x05，表示前一节点的长度为 5 字节。

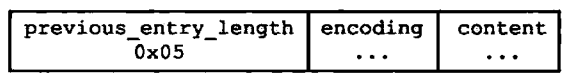

图 7-5 当前节点的前一节点的长度为 5 字节

图 7-6 展示了一个包含五字节长 previous_entry_length 属性的压缩节点，属性的值为 0xFE00002766，其中值的最高位字节 0xFE 表示这是一个五字节长的 previous_entry_length 属性，而之后的四字节 0x00002766（十进制值 10086）才是前一节点的实际长度。

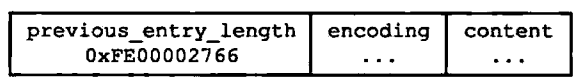

图 7-6 当前节点的前一节点的长度为 10086 字节

因为节点的previous_entry_length属性记录了前一个节点的长度，所以程序可以通过指针运算，根据当前节点的起始地址来计算出前一个节点的起始地址。

举个例子，如果我们有一个指向当前节点起始地址的指针c，那么我们只要用指针c减去当前节点previous_entry_length属性的值，就可以得出一个指向前一个节点起始地址的指针p，如图7-7所示。

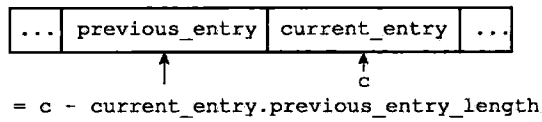

图7-7 通过指针运算计算出前一个节点的地址

压缩列表的从表尾向表头遍历操作就是使用这一原理实现的，只要我们拥有了一个指向某个节点起始地址的指针，那么通过这个指针以及这个节点的previous_entry_length属性，程序就可以一直向前一个节点回溯，最终到达压缩列表的表头节点。

图7-8展示了一个从表尾节点向表头节点进行遍历的完整过程：

- 首先，我们拥有指向压缩列表表尾节点entry4起始地址的指针p1（指向表尾节点的指针可以通过指向压缩列表起始地址的指针加上zltail属性的值得出）；
- 通过用p1减去entry4节点previous_entry_length属性的值，我们得到一个指向entry4前一节点entry3起始地址的指针p2；
- 通过用p2减去entry3节点previous_entry_length属性的值，我们得到一个指向entry3前一节点entry2起始地址的指针p3；
- 通过用p3减去entry2节点previous_entry_length属性的值，我们得到一个指向entry2前一节点entry1起始地址的指针p4，entry1为压缩列表的表头节点；
- 最终，我们从表尾节点向表头节点遍历了整个列表。

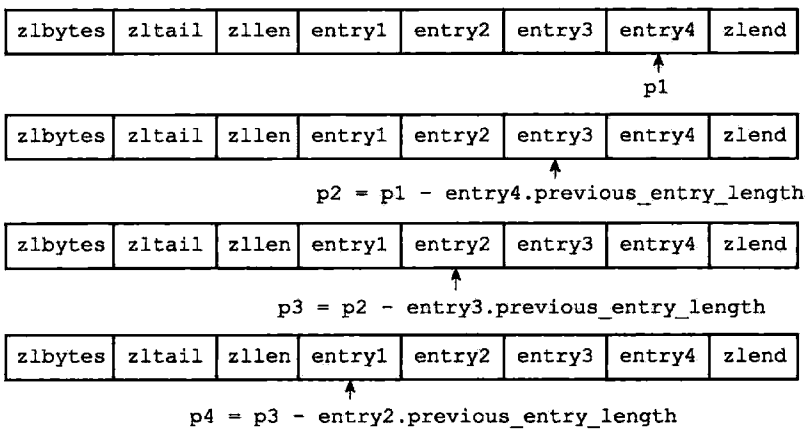

图7-8 一个从表尾向表头遍历的例子

7.2.2 encoding

节点的encoding属性记录了节点的content属性所保存数据的类型以及长度：

- 一字节、两字节或者五字节长，值的最高位为 00、01 或者 10 的是字节数组编码：这种编码表示节点的 content 属性保存着字节数组，数组的长度由编码除去最高两位之后的其他位记录；
- 一字节长，值的最高位以 11 开头的是整数编码：这种编码表示节点的 content 属性保存着整数值，整数值的类型和长度由编码除去最高两位之后的其他位记录；

表 7-2 记录了所有可用的字节数组编码，而表 7-3 则记录了所有可用的整数编码。表格中的下划线“_”表示留空，而 b、x 等变量则代表实际的二进制数据，为了方便阅读，多个字节之间用空格隔开。

表 7-2 字节数组编码

编 码	编码长度	content 属性保存的值
00bbbbbb	1 字节	长度小于等于 63 字节的字节数组
01bbbbbb xxxxxxxx	2 字节	长度小于等于 16 383 字节的字节数组
10______ aaaaaaaa bbbbbbbb cccccccc dddddddd	5 字节	长度小于等于 4 294 967 295 的字节数组

表 7-3 整数编码

编码	编码长度	content 属性保存的值
11000000	1 字节	int16_t 类型的整数
11010000	1 字节	int32_t 类型的整数
11100000	1 字节	int64_t 类型的整数
11110000	1 字节	24 位有符号整数
11111110	1 字节	8 位有符号整数
1111xxxx	1 字节	使用这一编码的节点没有相应的 content 属性，因为编码本身的 xxxx 四个位已经保存了一个介于 0 和 12 之间的值，所以它无须 content 属性

7.2.3 content

节点的 content 属性负责保存节点的值，节点值可以是一个字节数组或者整数，值的类型和长度由节点的 encoding 属性决定。

图 7-9 展示了一个保存字节数组的节点示例：

- 编码的最高两位 00 表示节点保存的是一个字节数组；
- 编码的后六位 001011 记录了字节数组的长度 11；
- content 属性保存着节点的值 "hello world"。

previous_entry_length ...	encoding 00001011	content "hello world"

图 7-9 保存着节数组 "hello world" 的节点

图 7-10 展示了一个保存整数值的节点示例：

- 编码 11000000 表示节点保

previous_entry_length ...	encoding 11000000	content 10086

图 7-10 保存着整数值 10086 的节点

存的是一个 int16_t 类型的整数值；

- content 属性保存着节点的值 10086。

7.3 连锁更新

前面说过，每个节点的 previous_entry_length 属性都记录了前一个节点的长度：

- 如果前一节点的长度小于 254 字节，那么 previous_entry_length 属性需要用 1 字节长的空间来保存这个长度值。
- 如果前一节点的长度大于等于 254 字节，那么 previous_entry_length 属性需要用 5 字节长的空间来保存这个长度值。

现在，考虑这样一种情况：在一个压缩列表中，有多个连续的、长度介于 250 字节到 253 字节之间的节点 e1 至 eN，如图 7-11 所示。

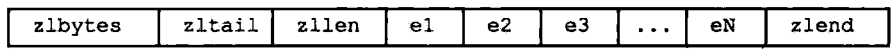

图 7-11 包含节点 e1 至 eN 的压缩列表

因为 e1 至 eN 的所有节点的长度都小于 254 字节，所以记录这些节点的长度只需要 1 字节长的 previous_entry_length 属性，换句话说，e1 至 eN 的所有节点的 previous_entry_length 属性都是 1 字节长的。

这时，如果我们将一个长度大于等于 254 字节的新节点 new 设置为压缩列表的表头节点，那么 new 将成为 e1 的前置节点，如图 7-12 所示。

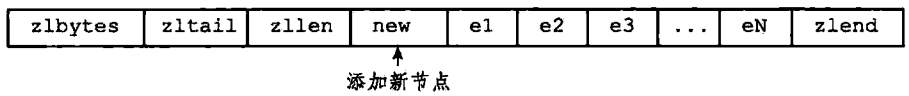

图 7-12 添加新节点到压缩列表

因为 e1 的 previous_entry_length 属性仅长 1 字节，它没办法保存新节点 new 的长度，所以程序将对压缩列表执行空间重分配操作，并将 e1 节点的 previous_entry_length 属性从原来的 1 字节长扩展为 5 字节长。

现在，麻烦的事情来了，e1 原本的长度介于 250 字节至 253 字节之间，在为 previous_entry_length 属性新增四个字节的空间之后，e1 的长度就变成了介于 254 字节至 257 字节之间，而这种长度使用 1 字节长的 previous_entry_length 属性是没办法保存的。

因此，为了让 e2 的 previous_entry_length 属性可以记录下 e1 的长度，程序需要再次对压缩列表执行空间重分配操作，并将 e2 节点的 previous_entry_length 属性从原来的 1 字节长扩展为 5 字节长。

正如扩展 e1 引发了对 e2 的扩展一样，扩展 e2 也会引发对 e3 的扩展，而扩展 e3 又会引发对 e4 的扩展……为了让每个节点的 previous_entry_length 属性都符合压缩列

表对节点的要求，程序需要不断地对压缩列表执行空间重分配操作，直到 eN 为止。

Redis 将这种在特殊情况下产生的连续多次空间扩展操作称之为“连锁更新”（cascade update），图 7-13 展示了这一过程。

除了添加新节点可能会引发连锁更新之外，删除节点也可能会引发连锁更新。

考虑图 7-14 所示的压缩列表，如果 e1 至 eN 都是大小介于 250 字节至 253 字节的节点，big 节点的长度大于等于 254 字节（需要 5 字节的 previous_entry_length 来保存），而 small 节点的长度小于 254 字节（只需要 1 字节的 previous_entry_length 来保存），那么当我们将 small 节点从压缩列表中删除之后，为了让 e1 的 previous_entry_length 属性可以记录 big 节点的长度，程序将扩展 e1 的空间，并由此引发之后的连锁更新。

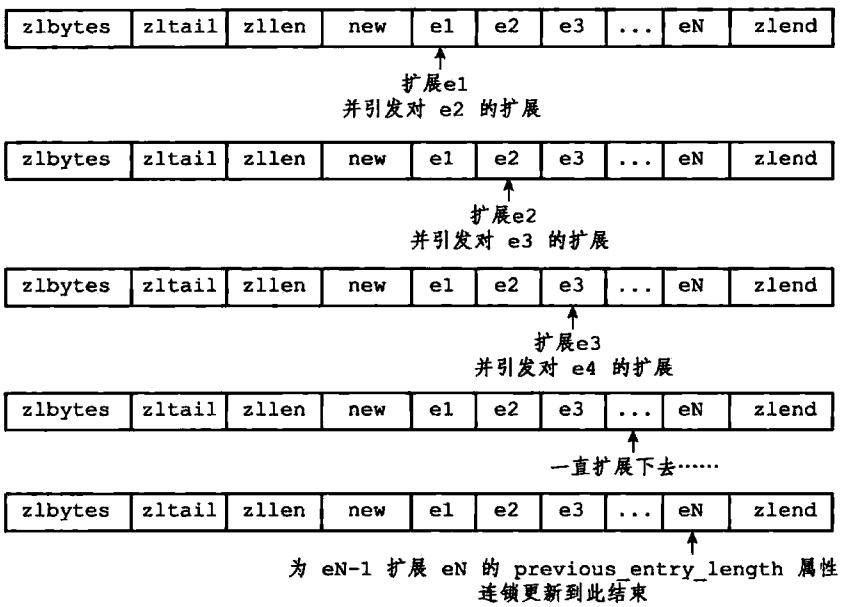

图 7-13 连锁更新过程

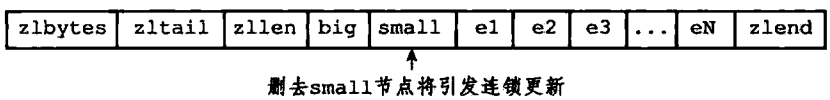

图 7-14 另一种引起连锁更新的情况

因为连锁更新在最坏情况下需要对压缩列表执行 N 次空间重分配操作，而每次空间重分配的最坏复杂度为 $O(N)$，所以连锁更新的最坏复杂度为 $O(N^2)$。

要注意的是，尽管连锁更新的复杂度较高，但它真正造成性能问题的几率是很低的：

- 首先，压缩列表里要恰好有多个连续的、长度介于 250 字节至 253 字节之间的节点，连锁更新才有可能被引发，在实际中，这种情况并不多见；
- 其次，即使出现连锁更新，但只要被更新的节点数量不多，就不会对性能造成任何

影响：比如说，对三五个节点进行连锁更新是绝对不会影响性能的；

因为以上原因，ziplistPush 等命令的平均复杂度仅为 $O(N)$，在实际中，我们可以放心地使用这些函数，而不必担心连锁更新会影响压缩列表的性能。

7.4 压缩列表 API

表 7-4 列出了所有用于操作压缩列表的 API。

表 7-4 压缩列表 API

函数	作用	算法复杂度
ziplistNew	创建一个新的压缩列表	$O(1)$
ziplistPush	创建一个包含给定值的新节点，并将这个新节点添加到压缩列表的表头或者表尾	平均 $O(N)$，最坏 $O(N^2)$
ziplistInsert	将包含给定值的新节点插入到给定节点之后	平均 $O(N)$，最坏 $O(N^2)$
ziplistIndex	返回压缩列表给定索引上的节点	$O(N)$
ziplistFind	在压缩列表中查找并返回包含了给定值的节点	因为节点的值可能是一个字节数组，所以检查节点值和给定值是否相同的复杂度为 $O(N)$，而查找整个列表的复杂度则为 $O(N^2)$
ziplistNext	返回给定节点的下一个节点	$O(1)$
ziplistPrev	返回给定节点的前一个节点	$O(1)$
ziplistGet	获取给定节点所保存的值	$O(1)$
ziplistDelete	从压缩列表中删除给定的节点	平均 $O(N)$，最坏 $O(N^2)$
ziplistDeleteRange	删除压缩列表在给定索引上的连续多个节点	平均 $O(N)$，最坏 $O(N^2)$
ziplistBlobLen	返回压缩列表目前占用的内存字节数	$O(1)$
ziplistLen	返回压缩列表目前包含的节点数量	节点数量小于 65 535 时为 $O(1)$，大于 65 535 时为 $O(N)$

因为 ziplistPush、ziplistInsert、ziplistDelete 和 ziplistDeleteRange 四个函数都有可能会引发连锁更新，所以它们的最坏复杂度都是 $O(N^2)$。

7.5 重点回顾

- 压缩列表是一种为节约内存而开发的顺序型数据结构。
- 压缩列表被用作列表键和哈希键的底层实现之一。
- 压缩列表可以包含多个节点，每个节点可以保存一个字节数组或者整数值。
- 添加新节点到压缩列表，或者从压缩列表中删除节点，可能会引发连锁更新操作，但这种操作出现的几率并不高。

第 8 章
对　　象

在前面的数个章节里，我们陆续介绍了 Redis 用到的所有主要数据结构，比如简单动态字符串（SDS）、双端链表、字典、压缩列表、整数集合等等。

Redis 并没有直接使用这些数据结构来实现键值对数据库，而是基于这些数据结构创建了一个对象系统，这个系统包含字符串对象、列表对象、哈希对象、集合对象和有序集合对象这五种类型的对象，每种对象都用到了至少一种我们前面所介绍的数据结构。

通过这五种不同类型的对象，Redis 可以在执行命令之前，根据对象的类型来判断一个对象是否可以执行给定的命令。使用对象的另一个好处是，我们可以针对不同的使用场景，为对象设置多种不同的数据结构实现，从而优化对象在不同场景下的使用效率。

除此之外，Redis 的对象系统还实现了基于引用计数技术的内存回收机制，当程序不再使用某个对象的时候，这个对象所占用的内存就会被自动释放；另外，Redis 还通过引用计数技术实现了对象共享机制，这一机制可以在适当的条件下，通过让多个数据库键共享同一个对象来节约内存。

最后，Redis 的对象带有访问时间记录信息，该信息可以用于计算数据库键的空转时长，在服务器启用了 `maxmemory` 功能的情况下，空转时长较大的那些键可能会优先被服务器删除。

本章接下来将逐一介绍以上提到的 Redis 对象系统的各个特性。

8.1　对象的类型与编码

Redis 使用对象来表示数据库中的键和值，每次当我们在 Redis 的数据库中新创建一个键值对时，我们至少会创建两个对象，一个对象用作键值对的键（键对象），另一个对象用作键值对的值（值对象）。

举个例子，以下 *SET* 命令在数据库中创建了一个新的键值对，其中键值对的键是一个包含了字符串值 `"msg"` 的对象，而键值对的值则是一个包含了字符串值 `"hello world"` 的对象：

```
redis> SET msg "hello world"
OK
```

Redis 中的每个对象都由一个 redisObject 结构表示，该结构中和保存数据有关的三个属性分别是 type 属性、encoding 属性和 ptr 属性：

```
typedef struct redisObject {

    // 类型
    unsigned type:4;

    // 编码
    unsigned encoding:4;

    // 指向底层实现数据结构的指针
    void *ptr;

    // ...

} robj;
```

8.1.1 类型

对象的 type 属性记录了对象的类型，这个属性的值可以是表 8-1 列出的常量的其中一个。

表 8-1 对象的类型

类型常量	对象的名称
REDIS_STRING	字符串对象
REDIS_LIST	列表对象
REDIS_HASH	哈希对象
REDIS_SET	集合对象
REDIS_ZSET	有序集合对象

对于 Redis 数据库保存的键值对来说，键总是一个字符串对象，而值则可以是字符串对象、列表对象、哈希对象、集合对象或者有序集合对象的其中一种，因此：

- 当我们称呼一个数据库键为“字符串键”时，我们指的是“这个数据库键所对应的值为字符串对象”；
- 当我们称呼一个键为“列表键”时，我们指的是“这个数据库键所对应的值为列表对象”。

TYPE 命令的实现方式也与此类似，当我们对一个数据库键执行 *TYPE* 命令时，命令返回的结果为数据库键对应的值对象的类型，而不是键对象的类型：

```
# 键为字符串对象，值为字符串对象

redis> SET msg "hello world"
OK

redis> TYPE msg
string

# 键为字符串对象，值为列表对象

redis> RPUSH numbers 1 3 5
(integer) 6

redis> TYPE numbers
```

```
list

# 键为字符串对象，值为哈希对象

redis> HMSET profile name Tom age 25 career Programmer
OK

redis> TYPE profile
hash

# 键为字符串对象，值为集合对象

redis> SADD fruits apple banana cherry
(integer) 3

redis> TYPE fruits
set

# 键为字符串对象，值为有序集合对象

redis> ZADD price 8.5 apple 5.0
    banana 6.0 cherry
(integer) 3

redis> TYPE price
zset
```

表 8-2 列出了 *TYPE* 命令在面对不同类型的值对象时所产生的输出。

表 8-2 不同类型值对象的 *TYPE* 命令输出

对象	对象 type 属性的值	*TYPE* 命令的输出
字符串对象	REDIS_STRING	"string"
列表对象	REDIS_LIST	"list"
哈希对象	REDIS_HASH	"hash"
集合对象	REDIS_SET	"set"
有序集合对象	REDIS_ZSET	"zset"

8.1.2 编码和底层实现

对象的 ptr 指针指向对象的底层实现数据结构，而这些数据结构由对象的 encoding 属性决定。

encoding 属性记录了对象所使用的编码，也即是说这个对象使用了什么数据结构作为对象的底层实现，这个属性的值可以是表 8-3 列出的常量的其中一个。

表 8-3 对象的编码

编码常量	编码所对应的底层数据结构
REDIS_ENCODING_INT	long 类型的整数
REDIS_ENCODING_EMBSTR	embstr 编码的简单动态字符串
REDIS_ENCODING_RAW	简单动态字符串
REDIS_ENCODING_HT	字典
REDIS_ENCODING_LINKEDLIST	双端链表
REDIS_ENCODING_ZIPLIST	压缩列表
REDIS_ENCODING_INTSET	整数集合
REDIS_ENCODING_SKIPLIST	跳跃表和字典

每种类型的对象都至少使用了两种不同的编码，表 8-4 列出了每种类型的对象可以使用的编码。

表 8-4 不同类型和编码的对象

类 型	编 码	对 象
REDIS_STRING	REDIS_ENCODING_INT	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用 embstr 编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳跃表和字典实现的有序集合对象

使用 *OBJECT ENCODING* 命令可以查看一个数据库键的值对象的编码：

```
redis> SET msg "hello wrold"
OK

redis> OBJECT ENCODING msg
"embstr"

redis> SET story "long long long long long long ago ..."
OK

redis> OBJECT ENCODING story
"raw"

redis> SADD numbers 1 3 5
(integer) 3

redis> OBJECT ENCODING numbers
"intset"

redis> SADD numbers "seven"
(integer) 1

redis> OBJECT ENCODING numbers
"hashtable"
```

表 8-5 列出了不同编码的对象所对应的 *OBJECT ENCODING* 命令输出。

表 8-5 *OBJECT ENCODING* 对不同编码的输出

对象所使用的底层数据结构	编码常量	*OBJECT ENCODING* 命令输出
整数	REDIS_ENCODING_INT	"int"
embstr 编码的简单动态字符串（SDS）	REDIS_ENCODING_EMBSTR	"embstr"
简单动态字符串	REDIS_ENCODING_RAW	"raw"
字典	REDIS_ENCODING_HT	"hashtable"

（续）

对象所使用的底层数据结构	编码常量	*OBJECT ENCODING* 命令输出
双端链表	REDIS_ENCODING_LINKEDLIST	"linkedlist"
压缩列表	REDIS_ENCODING_ZIPLIST	"ziplist"
整数集合	REDIS_ENCODING_INTSET	"intset"
跳跃表和字典	REDIS_ENCODING_SKIPLIST	"skiplist"

通过 encoding 属性来设定对象所使用的编码，而不是为特定类型的对象关联一种固定的编码，极大地提升了 Redis 的灵活性和效率，因为 Redis 可以根据不同的使用场景来为一个对象设置不同的编码，从而优化对象在某一场景下的效率。

举个例子，在列表对象包含的元素比较少时，Redis 使用压缩列表作为列表对象的底层实现：

- 因为压缩列表比双端链表更节约内存，并且在元素数量较少时，在内存中以连续块方式保存的压缩列表比起双端链表可以更快被载入到缓存中；
- 随着列表对象包含的元素越来越多，使用压缩列表来保存元素的优势逐渐消失时，对象就会将底层实现从压缩列表转向功能更强、也更适合保存大量元素的双端链表上面；

其他类型的对象也会通过使用多种不同的编码来进行类似的优化。

在接下来的内容中，我们将分别介绍 Redis 中的五种不同类型的对象，说明这些对象底层所使用的编码方式，列出对象从一种编码转换成另一种编码所需的条件，以及同一个命令在多种不同编码上的实现方法。

8.2 字符串对象

字符串对象的编码可以是 int、raw 或者 embstr。

如果一个字符串对象保存的是整数值，并且这个整数值可以用 long 类型来表示，那么字符串对象会将整数值保存在字符串对象结构的 ptr 属性里面（将 void* 转换成 long），并将字符串对象的编码设置为 int。

举个例子，如果我们执行以下 SET 命令，那么服务器将创建一个如图 8-1 所示的 int 编码的字符串对象作为 number 键的值：

```
redis> SET number 10086
OK

redis> OBJECT ENCODING number
"int"
```

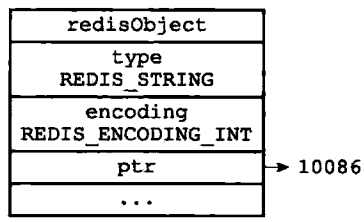

图 8-1 int 编码的字符串对象

如果字符串对象保存的是一个字符串值，并且这个字符串值的长度大于 32 字节，那么字符串对象将使用一个简单动态字符串（SDS）来保存这个字符串值，并将对象的编码设置为 raw。

举个例子，如果我们执行以下命令，那么服务器将创建一个如图 8-2 所示的 raw 编码

的字符串对象作为 story 键的值：

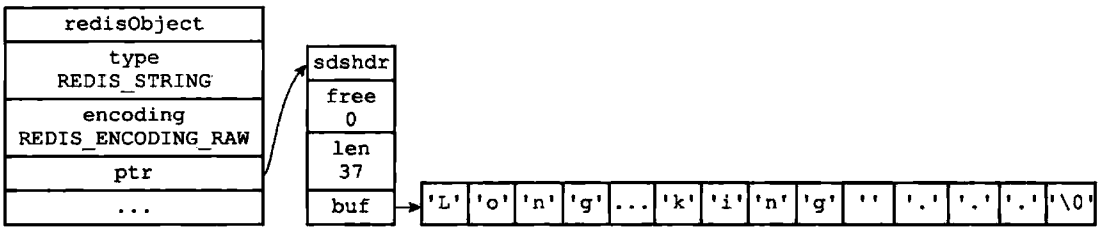

图 8-2 raw 编码的字符串对象

```
redis> SET story "Long, long ago there lived a king ..."
OK

redis> STRLEN story
(integer) 37

redis> OBJECT ENCODING story
"raw"
```

如果字符串对象保存的是一个字符串值，并且这个字符串值的长度小于等于 32 字节，那么字符串对象将使用 embstr 编码的方式来保存这个字符串值。

embstr 编码是专门用于保存短字符串的一种优化编码方式，这种编码和 raw 编码一样，都使用 redisObject 结构和 sdshdr 结构来表示字符串对象，但 raw 编码会调用两次内存分配函数来分别创建 redisObject 结构和 sdshdr 结构，而 embstr 编码则通过调用一次内存分配函数来分配一块连续的空间，空间中依次包含 redisObject 和 sdshdr 两个结构，如图 8-3 所示。

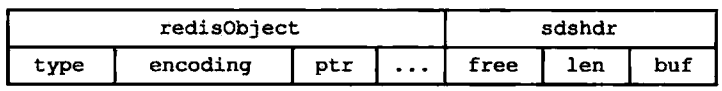

图 8-3 embstr 编码创建的内存块结构

embstr 编码的字符串对象在执行命令时，产生的效果和 raw 编码的字符串对象执行命令时产生的效果是相同的，但使用 embstr 编码的字符串对象来保存短字符串值有以下好处：

- embstr 编码将创建字符串对象所需的内存分配次数从 raw 编码的两次降低为一次。
- 释放 embstr 编码的字符串对象只需要调用一次内存释放函数，而释放 raw 编码的字符串对象需要调用两次内存释放函数。
- 因为 embstr 编码的字符串对象的所有数据都保存在一块连续的内存里面，所以这种编码的字符串对象比起 raw 编码的字符串对象能够更好地利用缓存带来的优势。

作为例子，以下命令创建了一个 embstr 编码的字符串对象作为 msg 键的值，值对象的样子如图 8-4 所示：

```
redis> SET msg "hello"
OK

redis> OBJECT ENCODING msg
"embstr"
```

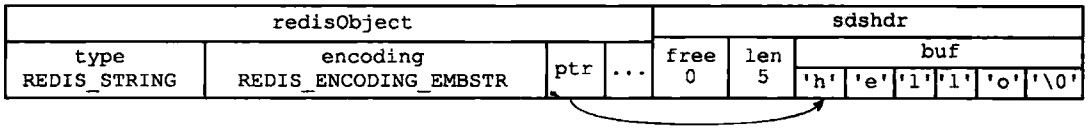

图 8-4 embstr 编码的字符串对象

最后要说的是，可以用 long double 类型表示的浮点数在 Redis 中也是作为字符串值来保存的。如果我们要保存一个浮点数到字符串对象里面，那么程序会先将这个浮点数转换成字符串值，然后再保存转换所得的字符串值。

举个例子，执行以下代码将创建一个包含 3.14 的字符串表示 "3.14" 的字符串对象：

```
redis> SET pi 3.14
OK

redis> OBJECT ENCODING pi
"embstr"
```

在有需要的时候，程序会将保存在字符串对象里面的字符串值转换回浮点数值，执行某些操作，然后再将执行操作所得的浮点数值转换回字符串值，并继续保存在字符串对象里面。

举个例子，如果我们执行以下代码：

```
redis> INCRBYFLOAT pi 2.0
"5.14"

redis> OBJECT ENCODING pi
"embstr"
```

那么程序首先会取出字符串对象里面保存的字符串值 "3.14"，将它转换回浮点数值 3.14，然后把 3.14 和 2.0 相加得出的值 5.14 转换成字符串 "5.14"，并将这个 "5.14" 保存到字符串对象里面。表 8-6 总结并列出了字符串对象保存各种不同类型的值所使用的编码方式。

表 8-6 字符串对象保存各类型值的编码方式

值	编码
可以用 long 类型保存的整数	int
可以用 long double 类型保存的浮点数	embstr 或者 raw
字符串值，或者因为长度太大而没办法用 long 类型表示的整数，又或者因为长度太大而没办法用 long double 类型表示的浮点数	embstr 或者 raw

8.2.1 编码的转换

int 编码的字符串对象和 embstr 编码的字符串对象在条件满足的情况下，会被转换为 raw 编码的字符串对象。

对于 int 编码的字符串对象来说，如果我们向对象执行了一些命令，使得这个对象保

存的不再是整数值，而是一个字符串值，那么字符串对象的编码将从 int 变为 raw。

在下面的示例中，我们通过 *APPEND* 命令，向一个保存整数值的字符串对象追加了一个字符串值，因为追加操作只能对字符串值执行，所以程序会先将之前保存的整数值 10086 转换为字符串值 "10086"，然后再执行追加操作，操作的执行结果就是一个 raw 编码的、保存了字符串值的字符串对象：

```
redis> SET number 10086
OK

redis> OBJECT ENCODING number
"int"

redis> APPEND number " is a good number!"
(integer) 23

redis> GET number
"10086 is a good number!"

redis> OBJECT ENCODING number
"raw"
```

另外，因为 Redis 没有为 embstr 编码的字符串对象编写任何相应的修改程序（只有 int 编码的字符串对象和 raw 编码的字符串对象有这些程序），所以 embstr 编码的字符串对象实际上是只读的。当我们对 embstr 编码的字符串对象执行任何修改命令时，程序会先将对象的编码从 embstr 转换成 raw，然后再执行修改命令。因为这个原因，embstr 编码的字符串对象在执行修改命令之后，总会变成一个 raw 编码的字符串对象。

以下代码展示了一个 embstr 编码的字符串对象在执行 *APPEND* 命令之后，对象的编码从 embstr 变为 raw 的例子：

```
redis> SET msg "hello world"
OK

redis> OBJECT ENCODING msg
"embstr"

redis> APPEND msg " again!"
(integer) 18

redis> OBJECT ENCODING msg
"raw"
```

8.2.2 字符串命令的实现

因为字符串键的值为字符串对象，所以用于字符串键的所有命令都是针对字符串对象来构建的，表 8-7 列举了其中一部分字符串命令，以及这些命令在不同编码的字符串对象下的实现方法。

表 8-7 字符串命令的实现

命令	int 编码的实现方法	embstr 编码的实现方法	raw 编码的实现方法
SET	使用 int 编码保存值	使用 embstr 编码保存值	使用 raw 编码保存值
GET	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，然后向客户端返回这个字符串值	直接向客户端返回字符串值	直接向客户端返回字符串值
APPEND	将对象转换成 raw 编码，然后按 raw 编码的方式执行此操作	将对象转换成 raw 编码，然后按 raw 编码的方式执行此操作	调用 sdscatlen 函数，将给定字符串追加到现有字符串的末尾
INCRBYFLOAT	取出整数值并将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来	取出字符串值并尝试将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来。如果字符串值不能被转换成浮点数，那么向客户端返回一个错误	取出字符串值并尝试将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来。如果字符串值不能被转换成浮点数，那么向客户端返回一个错误
INCRBY	对整数值进行加法计算，得出的计算结果会作为整数被保存起来	embstr 编码不能执行此命令，向客户端返回一个错误	raw 编码不能执行此命令，向客户端返回一个错误
DECRBY	对整数值进行减法计算，得出的计算结果会作为整数被保存起来	embstr 编码不能执行此命令，向客户端返回一个错误	raw 编码不能执行此命令，向客户端返回一个错误
STRLEN	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，计算并返回这个字符串值的长度	调用 sdslen 函数，返回字符串的长度	调用 sdslen 函数，返回字符串的长度
SETRANGE	将对象转换成 raw 编码，然后按 raw 编码的方式执行此命令	将对象转换成 raw 编码，然后按 raw 编码的方式执行此命令	将字符串特定索引上的值设置为给定的字符
GETRANGE	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，然后取出并返回字符串指定索引上的字符	直接取出并返回字符串指定索引上的字符	直接取出并返回字符串指定索引上的字符

8.3 列表对象

列表对象的编码可以是 ziplist 或者 linkedlist。

ziplist 编码的列表对象使用压缩列表作为底层实现，每个压缩列表节点（entry）保存了一个列表元素。举个例子，如果我们执行以下 *RPUSH* 命令，那么服务器将创建一个列表对象作为 numbers 键的值：

```
redis> RPUSH numbers 1 "three" 5
(integer) 3
```

如果 numbers 键的值对象使用的是 ziplist 编码，这个这个值对象将会是图 8-5 所

展示的样子。

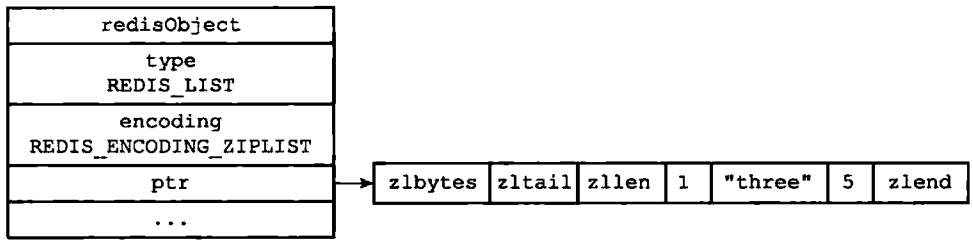

图 8-5 ziplist 编码的 numbers 列表对象

另一方面，linkedlist 编码的列表对象使用双端链表作为底层实现，每个双端链表节点（node）都保存了一个字符串对象，而每个字符串对象都保存了一个列表元素。

举个例子，如果前面所说的 numbers 键创建的列表对象使用的不是 ziplist 编码，而是 linkedlist 编码，那么 numbers 键的值对象将是图 8-6 所示的样子。

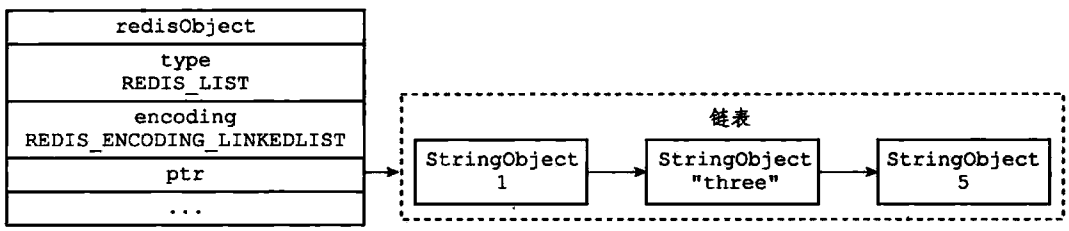

图 8-6 linkedlist 编码的 numbers 列表对象

注意，linkedlist 编码的列表对象在底层的双端链表结构中包含了多个字符串对象，这种嵌套字符串对象的行为在稍后介绍的哈希对象、集合对象和有序集合对象中都会出现，字符串对象是 Redis 五种类型的对象中唯一一种会被其他四种类型对象嵌套的对象。

注意

为了简化字符串对象的表示，我们在图 8-6 使用了一个带有 StringObject 字样的格子来表示一个字符串对象，而 StringObject 字样下面的是字符串对象所保存的值。比如说，图 8-7 代表的就是一个包含了字符串值 "three" 的字符串对象，它是图 8-8 的简化表示。

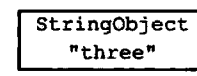

图 8-7 简化的字符串对象表示

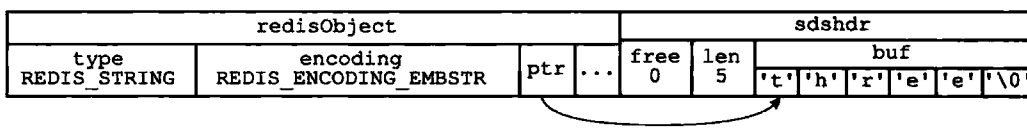

图 8-8 完整的字符串对象表示

本书接下来的内容将继续沿用这一简化表示。

8.3.1 编码转换

当列表对象可以同时满足以下两个条件时，列表对象使用 ziplist 编码：

- 列表对象保存的所有字符串元素的长度都小于 64 字节；
- 列表对象保存的元素数量小于 512 个；不能满足这两个条件的列表对象需要使用 linkedlist 编码。

注意

以上两个条件的上限值是可以修改的，具体请看配置文件中关于 list-max-ziplist-value 选项和 list-max-ziplist-entries 选项的说明。

对于使用 ziplist 编码的列表对象来说，当使用 ziplist 编码所需的两个条件的任意一个不能被满足时，对象的编码转换操作就会被执行，原本保存在压缩列表里的所有列表元素都会被转移并保存到双端链表里面，对象的编码也会从 ziplist 变为 linkedlist。

以下代码展示了列表对象因为保存了长度太大的元素而进行编码转换的情况：

```
# 所有元素的长度都小于 64 字节
redis> RPUSH blah "hello" "world" "again"
(integer)3

redis> OBJECT ENCODING blah
"ziplist"

# 将一个 65 字节长的元素推入列表对象中
redis> RPUSH blah "wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww"
(integer) 4

# 编码已改变
redis> OBJECT ENCODING blah
"linkedlist"
```

除此之外，以下代码展示了列表对象因为保存的元素数量过多而进行编码转换的情况：

```
# 列表对象包含 512 个元素
redis> EVAL "for i=1, 512 do redis.call('RPUSH', KEYS[1],i)end" 1 "integers"
(nil)

redis> LLEN integers
(integer) 512

redis> OBJECT ENCODING integers
"ziplist"

# 再向列表对象推入一个新元素，使得对象保存的元素数量达到 513 个
redis> RPUSH integers 513
(integer) 513

# 编码已改变
redis> OBJECT ENCODING integers
"linkedlist"
```

8.3.2 列表命令的实现

因为列表键的值为列表对象，所以用于列表键的所有命令都是针对列表对象来构建的，表 8-8 列出了其中一部分列表键命令，以及这些命令在不同编码的列表对象下的实现方法。

表 8-8 列表命令的实现

命令	ziplist 编码的实现方法	linkedlist 编码的实现方法
LPUSH	调用 ziplistPush 函数，将新元素推入到压缩列表的表头	调用 listAddNodeHead 函数，将新元素推入到双端链表的表头
RPUSH	调用 ziplistPush 函数，将新元素推入到压缩列表的表尾	调用 listAddNodeTail 函数，将新元素推入到双端链表的表尾
LPOP	调用 ziplistIndex 函数定位压缩列表的表头节点，在向用户返回节点所保存的元素之后，调用 ziplistDelete 函数删除表头节点	调用 listFirst 函数定位双端链表的表头节点，在向用户返回节点所保存的元素之后，调用 listDelNode 函数删除表头节点
RPOP	调用 ziplistIndex 函数定位压缩列表的表尾节点，在向用户返回节点所保存的元素之后，调用 ziplistDelete 函数删除表尾节点	调用 listLast 函数定位双端链表的表尾节点，在向用户返回节点所保存的元素之后，调用 listDelNode 函数删除表尾节点
LINDEX	调用 ziplistIndex 函数定位压缩列表中的指定节点，然后返回节点所保存的元素	调用 listIndex 函数定位双端链表中的指定节点，然后返回节点所保存的元素
LLEN	调用 ziplistLen 函数返回压缩列表的长度	调用 listLength 函数返回双端链表的长度
LINSERT	插入新节点到压缩列表的表头或者表尾时，使用 ziplistPush 函数；插入新节点到压缩列表的其他位置时，使用 ziplistInsert 函数	调用 listInsertNode 函数，将新节点插入到双端链表的指定位置
LREM	遍历压缩列表节点，并调用 ziplistDelete 函数删除包含了给定元素的节点	遍历双端链表节点，并调用 listDelNode 函数删除包含了给定元素的节点
LTRIM	调用 ziplistDeleteRange 函数，删除压缩列表中所有不在指定索引范围内的节点	遍历双端链表节点，并调用 listDelNode 函数删除链表中所有不在指定索引范围内的节点
LSET	调用 ziplistDelete 函数，先删除压缩列表指定索引上的现有节点，然后调用 ziplistInsert 函数，将一个包含给定元素的新节点插入到相同索引上面	调用 listIndex 函数，定位到双端链表指定索引上的节点，然后通过赋值操作更新节点的值

8.4 哈希对象

哈希对象的编码可以是 ziplist 或者 hashtable。

ziplist 编码的哈希对象使用压缩列表作为底层实现，每当有新的键值对要加入到哈希对象时，程序会先将保存了键的压缩列表节点推入到压缩列表表尾，然后再将保存了值的压缩列表节点推入到压缩列表表尾，因此：

- 保存了同一键值对的两个节点总是紧挨在一起，保存键的节点在前，保存值的节点在后；

- 先添加到哈希对象中的键值对会被放在压缩列表的表头方向，而后来添加到哈希对象中的键值对会被放在压缩列表的表尾方向。

举个例子，如果我们执行以下*HSET*命令，那么服务器将创建一个列表对象作为profile键的值：

```
redis> HSET profile name "Tom"
(integer) 1

redis> HSET profile age 25
(integer) 1

redis> HSET profile career "Programmer"
(integer) 1
```

如果profile键的值对象使用的是ziplist编码，那么这个值对象将会是图 8-9 所示的样子，其中对象所使用的压缩列表如图 8-10 所示。

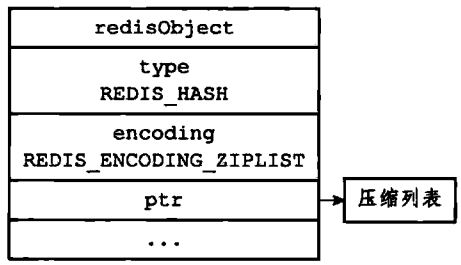

图 8-9 ziplist 编码的 profile 哈希对象

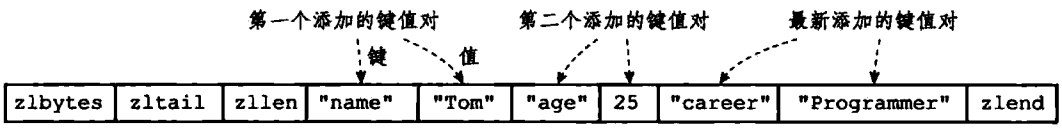

图 8-10 profile 哈希对象的压缩列表底层实现

另一方面，hashtable 编码的哈希对象使用字典作为底层实现，哈希对象中的每个键值对都使用一个字典键值对来保存：

- 字典的每个键都是一个字符串对象，对象中保存了键值对的键；
- 字典的每个值都是一个字符串对象，对象中保存了键值对的值。

举个例子，如果前面profile键创建的不是ziplist编码的哈希对象，而是hashtable编码的哈希对象，那么这个哈希对象应该会是图 8-11 所示的样子。

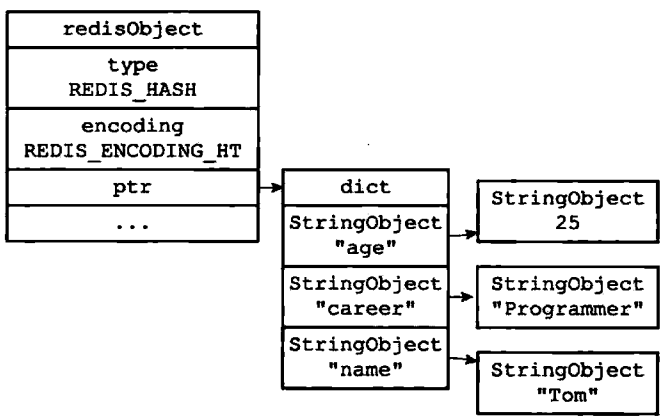

图 8-11 hashtable 编码的 profile 哈希对象

8.4.1 编码转换

当哈希对象可以同时满足以下两个条件时，哈希对象使用 ziplist 编码：

- 哈希对象保存的所有键值对的键和值的字符串长度都小于 64 字节；
- 哈希对象保存的键值对数量小于 512 个；不能满足这两个条件的哈希对象需要使用 hashtable 编码。

注意

这两个条件的上限值是可以修改的，具体请看配置文件中关于 hash-max-ziplist-value 选项和 hash-max-ziplist-entries 选项的说明。

对于使用 ziplist 编码的列表对象来说，当使用 ziplist 编码所需的两个条件的任意一个不能被满足时，对象的编码转换操作就会被执行，原本保存在压缩列表里的所有键值对都会被转移并保存到字典里面，对象的编码也会从 ziplist 变为 hashtable。

以下代码展示了哈希对象因为键值对的键长度太大而引起编码转换的情况：

```
# 哈希对象只包含一个键和值都不超过 64 个字节的键值对
redis> HSET book name "Mastering C++ in 21 days"
(integer) 1

redis> OBJECT ENCODING book
"ziplist"

# 向哈希对象添加一个新的键值对，键的长度为 66 字节
redis> HSET book long_long_long_long_long_long_long_long_long_long_
    description "content"
(integer) 1

# 编码已改变
redis> OBJECT ENCODING book
"hashtable"
```

除了键的长度太大会引起编码转换之外，值的长度太大也会引起编码转换，以下代码展示了这种情况的一个示例：

```
# 哈希对象只包含一个键和值都不超过 64 个字节的键值对
redis> HSET blah greeting "hello world"
(integer) 1

redis> OBJECT ENCODING blah
"ziplist"

# 向哈希对象添加一个新的键值对，值的长度为 68 字节
redis> HSET blah story "many string ... many string ... many string ... many
    string ... many"
(integer) 1

# 编码已改变
redis> OBJECT ENCODING blah
"hashtable"
```

最后，以下代码展示了哈希对象因为包含的键值对数量过多而引起编码转换的情况：

```
# 创建一个包含 512 个键值对的哈希对象
redis> EVAL "for i=1, 512 do redis.call('HSET', KEYS[1], i, i)end" 1 "numbers"
(nil)

redis> HLEN numbers
(integer) 512

redis> OBJECT ENCODING numbers
"ziplist"

# 再向哈希对象添加一个新的键值对，使得键值对的数量变成 513 个
redis> HMSET numbers "key" "value"
OK

redis> HLEN numbers
(integer) 513

# 编码改变
redis> OBJECT ENCODING numbers
"hashtable"
```

8.4.2 哈希命令的实现

因为哈希键的值为哈希对象，所以用于哈希键的所有命令都是针对哈希对象来构建的，表 8-9 列出了其中一部分哈希键命令，以及这些命令在不同编码的哈希对象下的实现方法。

表 8-9 哈希命令的实现

命令	ziplist 编码实现方法	hashtable 编码的实现方法
HSET	首先调用 ziplistPush 函数，将键推入到压缩列表的表尾，然后再次调用 ziplistPush 函数，将值推入到压缩列表的表尾	调用 dictAdd 函数，将新节点添加到字典里面
HGET	首先调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，然后调用 ziplistNext 函数，将指针移动到键节点旁边的值节点，最后返回值节点	调用 dictFind 函数，在字典中查找给定键，然后调用 dictGetVal 函数，返回该键所对应的值
HEXISTS	调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，如果找到的话说明键值对存在，没找到的话就说明键值对不存在	调用 dictFind 函数，在字典中查找给定键，如果找到的话说明键值对存在，没找到的话就说明键值对不存在
HDEL	调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，然后将相应的键节点、以及键节点旁边的值节点都删除掉	调用 dictDelete 函数，将指定键所对应的键值对从字典中删除掉
HLEN	调用 ziplistLen 函数，取得压缩列表包含节点的总数量，将这个数量除以 2，得出的结果就是压缩列表保存的键值对的数量	调用 dictSize 函数，返回字典包含的键值对数量，这个数量就是哈希对象包含的键值对数量
HGETALL	遍历整个压缩列表，用 ziplistGet 函数返回所有键和值（都是节点）	遍历整个字典，用 dictGetKey 函数返回字典的键，用 dictGetVal 函数返回字典的值

8.5 集合对象

集合对象的编码可以是 intset 或者 hashtable。

intset 编码的集合对象使用整数集合作为底层实现，集合对象包含的所有元素都被保存在整数集合里面。

举个例子，以下代码将创建一个如图 8-12 所示的 intset 编码集合对象：

```
redis> SADD numbers 1 3 5
(integer) 3
```

另一方面，hashtable 编码的集合对象使用字典作为底层实现，字典的每个键都是一个字符串对象，每个字符串对象包含了一个集合元素，而字典的值则全部被设置为 NULL。

举个例子，以下代码将创建一个如图 8-13 所示的 hashtable 编码集合对象：

```
redis> SAD Dfruits "apple" "banana" "cherry"
(integer)3
```

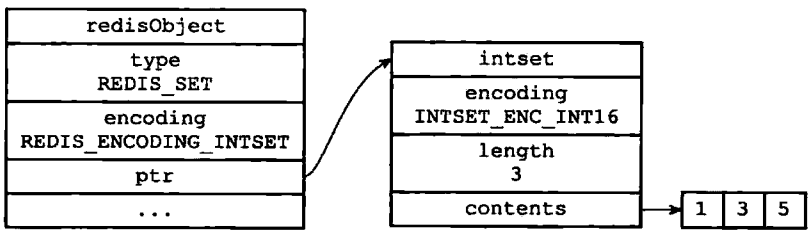

图 8-12 intset 编码的 numbers 集合对象

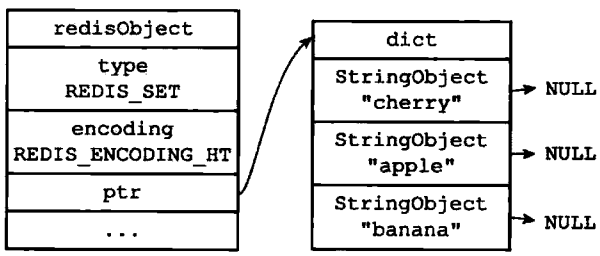

图 8-13 hashtable 编码的 fruits 集合对象

8.5.1 编码的转换

当集合对象可以同时满足以下两个条件时，对象使用 intset 编码：

- 集合对象保存的所有元素都是整数值；
- 集合对象保存的元素数量不超过 512 个。

不能满足这两个条件的集合对象需要使用 hashtable 编码。

注意

第二个条件的上限值是可以修改的，具体请看配置文件中关于 set-max-intset-entries 选项的说明。

对于使用 intset 编码的集合对象来说，当使用 intset 编码所需的两个条件的任意一个不能被满足时，就会执行对象的编码转换操作，原本保存在整数集合中的所有元素都会被转移并保存到字典里面，并且对象的编码也会从 intset 变为 hashtable。

举个例子，以下代码创建了一个只包含整数元素的集合对象，该对象的编码为 intset：

```
redis> SADD numbers 1 3 5
(integer) 3

redis> OBJECT ENCODING numbers
"intset"
```

不过，只要我们向这个只包含整数元素的集合对象添加一个字符串元素，集合对象的编码转移操作就会被执行：

```
redis> SADD numbers "seven"
(integer) 1

redis> OBJECT ENCODING numbers
"hashtable"
```

除此之外，如果我们创建一个包含 512 个整数元素的集合对象，那么对象的编码应该会是 intset：

```
redis> EVAL "for i=1, 512 do redis.call('SADD', KEYS[1], i) end" 1 integers
(nil)

redis> SCARD integers
(integer) 512

redis> OBJECT ENCODING integers
"intset"
```

但是，只要我们再向集合添加一个新的整数元素，使得这个集合的元素数量变成 513，那么对象的编码转换操作就会被执行：

```
redis> SADD integers 10086
(integer) 1

redis> SCARD integers
(integer) 513

redis> OBJECT ENCODING integers
"hashtable"
```

8.5.2 集合命令的实现

因为集合键的值为集合对象，所以用于集合键的所有命令都是针对集合对象来构建的，表 8-10 列出了其中一部分集合键命令，以及这些命令在不同编码的集合对象下的实现方法。

表 8-10 集合命令的实现方法

命令	intset 编码的实现方法	hashtable 编码的实现方法
SADD	调用 intsetAdd 函数，将所有新元素添加到整数集合里面	调用 dictAdd，以新元素为键，NULL 为值，将键值对添加到字典里面

（续）

命令	intset 编码的实现方法	hashtable 编码的实现方法
SCARD	调用 intsetLen 函数，返回整数集合所包含的元素数量，这个数量就是集合对象所包含的元素数量	调用 dictSize 函数，返回字典所包含的键值对数量，这个数量就是集合对象所包含的元素数量
SISMEMBER	调用 intsetFind 函数，在整数集合中查找给定的元素，如果找到了说明元素存在于集合，没找到则说明元素不存在于集合	调用 dictFind 函数，在字典的键中查找给定的元素，如果找到了说明元素存在于集合，没找到则说明元素不存在于集合
SMEMBERS	遍历整个整数集合，使用 intsetGet 函数返回集合元素	遍历整个字典，使用 dictGetKey 函数返回字典的键作为集合元素
SRANDMEMBER	调用 intsetRandom 函数，从整数集合中随机返回一个元素	调用 dictGetRandomKey 函数，从字典中随机返回一个字典键
SPOP	调用 intsetRandom 函数，从整数集合中随机取出一个元素，在将这个随机元素返回给客户端之后，调用 intsetRemove 函数，将随机元素从整数集合中删除掉	调用 dictGetRandomKey 函数，从字典中随机取出一个字典键，在将这个随机字典键的值返回给客户端之后，调用 dictDelete 函数，从字典中删除随机字典键所对应的键值对
SREM	调用 intsetRemove 函数，从整数集合中删除所有给定的元素	调用 dictDelete 函数，从字典中删除所有键为给定元素的键值对

8.6 有序集合对象

有序集合的编码可以是 ziplist 或者 skiplist。

ziplist 编码的压缩列表对象使用压缩列表作为底层实现，每个集合元素使用两个紧挨在一起的压缩列表节点来保存，第一个节点保存元素的成员（member），而第二个元素则保存元素的分值（score）。

压缩列表内的集合元素按分值从小到大进行排序，分值较小的元素被放置在靠近表头的方向，而分值较大的元素则被放置在靠近表尾的方向。

举个例子，如果我们执行以下 *ZADD* 命令，那么服务器将创建一个有序集合对象作为 price 键的值：

```
redis> ZADD price 8.5 apple 5.0 banana 6.0 cherry
(integer) 3
```

如果 price 键的值对象使用的是 ziplist 编码，那么这个值对象将会是图 8-14 所示的样子，而对象所使用的压缩列表则会是 8-15 所示的样子。

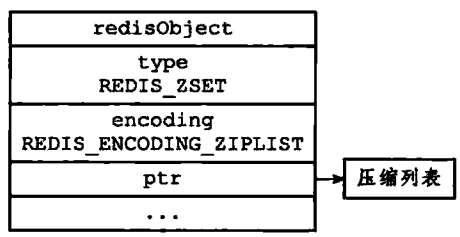

图 8-14 ziplist 编码的有序集合对象

skiplist 编码的有序集合对象使用 zset 结构作为底层实现，一个 zset 结构同时包含一个字典和一个跳跃表：

```
typedef struct zset {

    zskiplist *zsl;
```

```
    dict *dict;
} zset;
```

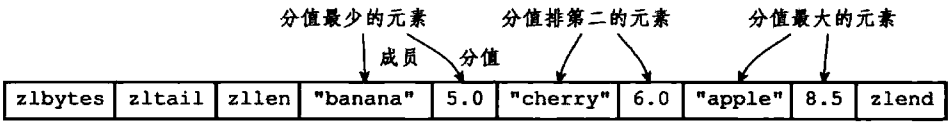

图 8-15 有序集合元素在压缩列表中按分值从小到大排列

zset 结构中的 zsl 跳跃表按分值从小到大保存了所有集合元素，每个跳跃表节点都保存了一个集合元素：跳跃表节点的 object 属性保存了元素的成员，而跳跃表节点的 score 属性则保存了元素的分值。通过这个跳跃表，程序可以对有序集合进行范围型操作，比如 *ZRANK*、*ZRANGE* 等命令就是基于跳跃表 API 来实现的。

除此之外，zset 结构中的 dict 字典为有序集合创建了一个从成员到分值的映射，字典中的每个键值对都保存了一个集合元素：字典的键保存了元素的成员，而字典的值则保存了元素的分值。通过这个字典，程序可以用 $O(1)$ 复杂度查找给定成员的分值，*ZSCORE* 命令就是根据这一特性实现的，而很多其他有序集合命令都在实现的内部用到了这一特性。

有序集合每个元素的成员都是一个字符串对象，而每个元素的分值都是一个 double 类型的浮点数。值得一提的是，虽然 zset 结构同时使用跳跃表和字典来保存有序集合元素，但这两种数据结构都会通过指针来共享相同元素的成员和分值，所以同时使用跳跃表和字典来保存集合元素不会产生任何重复成员或者分值，也不会因此而浪费额外的内存。

为什么有序集合需要同时使用跳跃表和字典来实现？

在理论上，有序集合可以单独使用字典或者跳跃表的其中一种数据结构来实现，但无论单独使用字典还是跳跃表，在性能上对比起同时使用字典和跳跃表都会有所降低。举个例子，如果我们只使用字典来实现有序集合，那么虽然以 $O(1)$ 复杂度查找成员的分值这一特性会被保留，但是，因为字典以无序的方式来保存集合元素，所以每次在执行范围型操作——比如 *ZRANK*、*ZRANGE* 等命令时，程序都需要对字典保存的所有元素进行排序，完成这种排序需要至少 $O(N\log N)$ 时间复杂度，以及额外的 $O(N)$ 内存空间（因为要创建一个数组来保存排序后的元素）。

另一方面，如果我们只使用跳跃表来实现有序集合，那么跳跃表执行范围型操作的所有优点都会被保留，但因为没有了字典，所以根据成员查找分值这一操作的复杂度将从 $O(1)$ 上升为 $O(\log N)$。因为以上原因，为了让有序集合的查找和范围型操作都尽可能快地执行，Redis 选择了同时使用字典和跳跃表两种数据结构来实现有序集合。

举个例子，如果前面 price 键创建的不是 ziplist 编码的有序集合对象，而是 skiplist 编码的有序集合对象，那么这个有序集合对象将会是图 8-16 所示的样子，而对象所使用的 zset 结构将会是图 8-17 所示的样子。

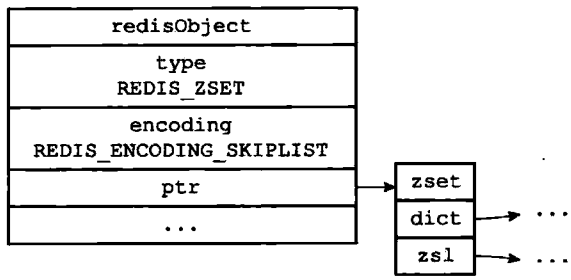

图 8-16 skiplist 编码的有序集合对象

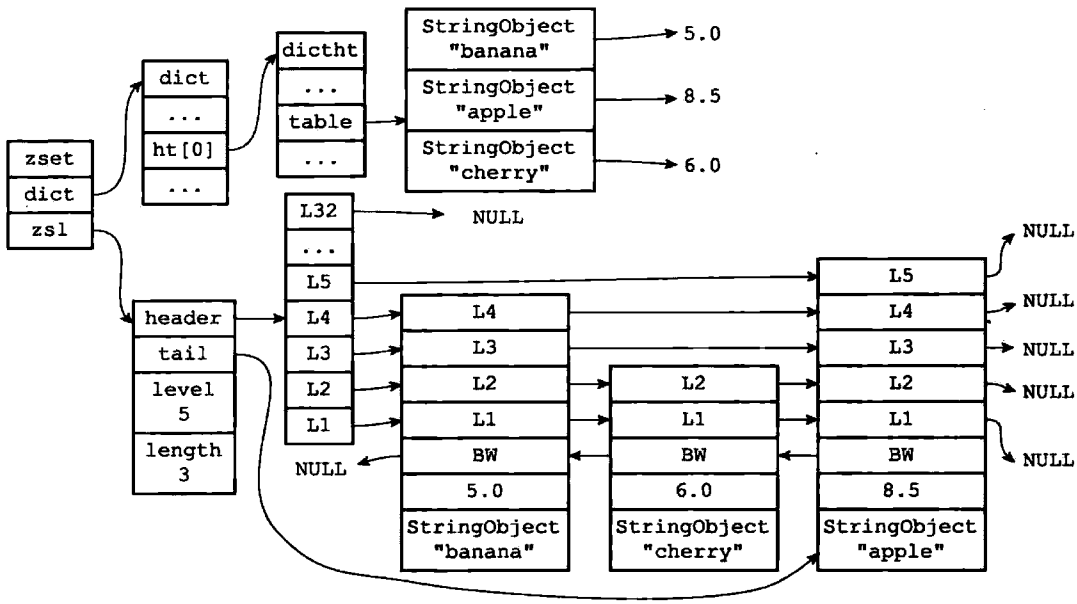

图 8-17 有序集合元素同时被保存在字典和跳跃表中

注意

为了展示方便，图 8-17 在字典和跳跃表中重复展示了各个元素的成员和分值，但在实际中，字典和跳跃表会共享元素的成员和分值，所以并不会造成任何数据重复，也不会因此而浪费任何内存。

8.6.1 编码的转换

当有序集合对象可以同时满足以下两个条件时，对象使用 ziplist 编码：

- 有序集合保存的元素数量小于 128 个；
- 有序集合保存的所有元素成员的长度都小于 64 字节；

不能满足以上两个条件的有序集合对象将使用 skiplist 编码。

注意

以上两个条件的上限值是可以修改的，具体请看配置文件中关于 zset-max-ziplist-entries 选项和 zset-max-ziplist-value 选项的说明。

对于使用 ziplist 编码的有序集合对象来说，当使用 ziplist 编码所需的两个条件中的任意一个不能被满足时，就会执行对象的编码转换操作，原本保存在压缩列表里的所有集合元素都会被转移并保存到 zset 结构里面，对象的编码也会从 ziplist 变为 skiplist。

以下代码展示了有序集合对象因为包含了过多元素而引发编码转换的情况：

```
# 对象包含了 128 个元素
redis> EVAL "for i=1, 128 do redis.call('ZADD', KEYS[1], i, i) end" 1 numbers
(nil)

redis> ZCARD numbers
(integer) 128

redis> OBJECT ENCODING numbers
"ziplist"

# 再添加一个新元素
redis> ZADD numbers 3.14 pi
(integer) 1

# 对象包含的元素数量变为 129 个
redis> ZCARD numbers
(integer) 129

# 编码已改变
redis> OBJECT ENCODING numbers
"skiplist"
```

以下代码则展示了有序集合对象因为元素的成员过长而引发编码转换的情况：

```
# 向有序集合添加一个成员只有三字节长的元素
redis> ZADD blah 1.0 www
(integer) 1

redis> OBJECT ENCODING blah
"ziplist"

# 向有序集合添加一个成员为 66 字节长的元素
redis> ZADD blah 2.0 ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
(integer) 1

# 编码已改变
redis> OBJECT ENCODING blah
"skiplist"
```

8.6.2 有序集合命令的实现

因为有序集合键的值为哈希对象，所以用于有序集合键的所有命令都是针对哈希对象来构建的，表 8-11 列出了其中一部分有序集合键命令，以及这些命令在不同编码的哈希对象

下的实现方法。

表 8-11 有序集合命令的实现方法

命令	ziplist 编码的实现方法	zset 编码的实现方法
ZADD	调用 ziplistInsert 函数，将成员和分值作为两个节点分别插入到压缩列表	先调用 zslInsert 函数，将新元素添加到跳跃表，然后调用 dictAdd 函数，将新元素关联到字典
ZCARD	调用 ziplistLen 函数，获得压缩列表包含节点的数量，将这个数量除以 2 得出集合元素的数量	访问跳跃表数据结构的 length 属性，直接返回集合元素的数量
ZCOUNT	遍历压缩列表，统计分值在给定范围内的节点的数量	遍历跳跃表，统计分值在给定范围内的节点的数量
ZRANGE	从表头向表尾遍历压缩列表，返回给定索引范围内的所有元素	从表头向表尾遍历跳跃表，返回给定索引范围内的所有元素
ZREVRANGE	从表尾向表头遍历压缩列表，返回给定索引范围内的所有元素	从表尾向表头遍历跳跃表，返回给定索引范围内的所有元素
ZRANK	从表头向表尾遍历压缩列表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名	从表头向表尾遍历跳跃表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名
ZREVRANK	从表尾向表头遍历压缩列表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名	从表尾向表头遍历跳跃表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名
ZREM	遍历压缩列表，删除所有包含给定成员的节点，以及被删除成员节点旁边的分值节点	遍历跳跃表，删除所有包含了给定成员的跳跃表节点。并在字典中解除被删除元素的成员和分值的关联
ZSCORE	遍历压缩列表，查找包含了给定成员的节点，然后取出成员节点旁边的分值节点保存的元素分值	直接从字典中取出给定成员的分值

8.7 类型检查与命令多态

Redis 中用于操作键的命令基本上可以分为两种类型。

其中一种命令可以对任何类型的键执行，比如说 *DEL* 命令、*EXPIRE* 命令、*RENAME* 命令、*TYPE* 命令、*OBJECT* 命令等。

举个例子，以下代码就展示了使用 *DEL* 命令来删除三种不同类型的键：

```
# 字符串键
redis> SET msg "hello"
OK

# 列表键
```

```
redis> RPUSH numbers 1 2 3
(integer) 3

# 集合键
redis> SADD fruits apple banana cherry
(integer) 3

redis> DEL msg
(integer) 1

redis> DEL numbers
(integer) 1

redis> DEL fruits
(integer) 1
```

而另一种命令只能对特定类型的键执行，比如说：

- *SET*、*GET*、*APPEND*、*STRLEN* 等命令只能对字符串键执行；
- *HDEL*、*HSET*、*HGET*、*HLEN* 等命令只能对哈希键执行；
- *RPUSH*、*LPOP*、*LINSERT*、*LLEN* 等命令只能对列表键执行；
- *SADD*、*SPOP*、*SINTER*、*SCARD* 等命令只能对集合键执行；
- *ZADD*、*ZCARD*、*ZRANK*、*ZSCORE* 等命令只能对有序集合键执行；

举个例子，我们可以用 *SET* 命令创建一个字符串键，然后用 *GET* 命令和 *APPEND* 命令操作这个键，但如果我们试图对这个字符串键执行只有列表键才能执行的 *LLEN* 命令，那么 Redis 将向我们返回一个类型错误：

```
redis> SET msg "hello world"
OK

redis> GET msg
"hello world"

redis> APPEND msg " again!"
(integer) 18

redis> GET msg
"hello world again!"

redis> LLEN msg
(error) WRONGTYPE Operation against a key holding the wrong kind of value
```

8.7.1 类型检查的实现

从上面发生类型错误的代码示例可以看出，为了确保只有指定类型的键可以执行某些特定的命令，在执行一个类型特定的命令之前，Redis 会先检查输入键的类型是否正确，然后再决定是否执行给定的命令。

类型特定命令所进行的类型检查是通过 `redisObject` 结构的 `type` 属性来实现的：

- 在执行一个类型特定命令之前，服务器会先检查输入数据库键的值对象是否为执行

命令所需的类型，如果是的话，服务器就对键执行指定的命令；

- 否则，服务器将拒绝执行命令，并向客户端返回一个类型错误。

举个例子，对于 *LLEN* 命令来说：

- 在执行 *LLEN* 命令之前，服务器会先检查输入数据库键的值对象是否为列表类型，也即是，检查值对象 redisObject 结构 type 属性的值是否为 REDIS_LIST，如果是的话，服务器就对键执行 *LLEN* 命令；
- 否则的话，服务器就拒绝执行命令并向客户端返回一个类型错误；图 8-18 展示了这一类型检查过程。

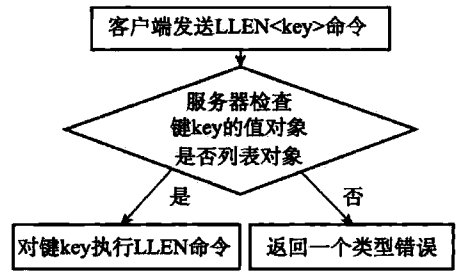

图 8-18 LLEN 命令执行时的类型检查过程

其他类型特定命令的类型检查过程也和这里展示的 *LLEN* 命令的类型检查过程类似。

8.7.2 多态命令的实现

Redis 除了会根据值对象的类型来判断键是否能够执行指定命令之外，还会根据值对象的编码方式，选择正确的命令实现代码来执行命令。

举个例子，在前面介绍列表对象的编码时我们说过，列表对象有 ziplist 和 linkedlist 两种编码可用，其中前者使用压缩列表 API 来实现列表命令，而后者则使用双端链表 API 来实现列表命令。

现在，考虑这样一个情况，如果我们对一个键执行 *LLEN* 命令，那么服务器除了要确保执行命令的是列表键之外，还需要根据键的值对象所使用的编码来选择正确的 *LLEN* 命令实现：

- 如果列表对象的编码为 ziplist，那么说明列表对象的实现为压缩列表，程序将使用 ziplistLen 函数来返回列表的长度；
- 如果列表对象的编码为 linkedlist，那么说明列表对象的实现为双端链表，程序将使用 listLength 函数来返回双端链表的长度；

借用面向对象方面的术语来说，我们可以认为 *LLEN* 命令是多态（polymorphism）的，只要执行 *LLEN* 命令的是列表键，那么无论值对象使用的是 ziplist 编码还是 linkedlist 编码，命令都可以正常执行。

图 8-19 展示了 *LLEN* 命令从类型检查到根据编码选择实现函数的整个执行过程，其他类型特定命令的执行过程也是类似的。

实际上，我们可以将 *DEL*、*EXPIRE*、*TYPE* 等命令也称为多态命令，因为无论输入的键是什么类型，这些命令都可以正确地执行。

DEL、*EXPIRE* 等命令和 *LLEN* 等命令的区别在于，前者是基于类型的多态——一个命令可以同时用于处理多种不同类型的键，而后者是基于编码的多态——一个命令可以同时用于处理多种不同编码。

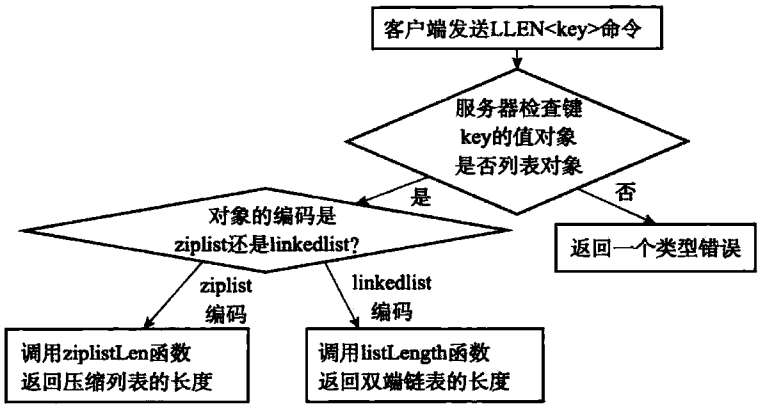

图 8-19 LLEN 命令的执行过程

8.8 内存回收

因为 C 语言并不具备自动内存回收功能，所以 Redis 在自己的对象系统中构建了一个引用计数（reference counting）技术实现的内存回收机制，通过这一机制，程序可以通过跟踪对象的引用计数信息，在适当的时候自动释放对象并进行内存回收。

每个对象的引用计数信息由 `redisObject` 结构的 `refcount` 属性记录：

```
typedef struct redisObject {

    // ...

    // 引用计数
    int refcount;

    // ...

} robj;
```

对象的引用计数信息会随着对象的使用状态而不断变化：

- 在创建一个新对象时，引用计数的值会被初始化为 1；
- 当对象被一个新程序使用时，它的引用计数值会被增一；
- 当对象不再被一个程序使用时，它的引用计数值会被减一；
- 当对象的引用计数值变为 0 时，对象所占用的内存会被释放。

表 8-12 列出了修改对象引用计数的 API，这些 API 分别用于增加、减少、重置对象的引用计数。

表 8-12 修改对象引用计数的 API

函数	作用
`incrRefCount`	将对象的引用计数值增一
`decrRefCount`	将对象的引用计数值减一，当对象的引用计数值等于 0 时，释放对象
`resetRefCount`	将对象的引用计数值设置为 0，但并不释放对象，这个函数通常在需要重新设置对象的引用计数值时使用

对象的整个生命周期可以划分为创建对象、操作对象、释放对象三个阶段。作为例子，以下代码展示了一个字符串对象从创建到释放的整个过程：

```
// 创建一个字符串对象 s，对象的引用计数为 1
robj *s = createStringObject(...)

// 对象 s 执行各种操作 ...

// 将对象 s 的引用计数减一，使得对象的引用计数变为 0
// 导致对象 s 被释放
decrRefCount(s)
```

其他不同类型的对象也会经历类似的过程。

8.9 对象共享

除了用于实现引用计数内存回收机制之外，对象的引用计数属性还带有对象共享的作用。举个例子，假设键 A 创建了一个包含整数值 100 的字符串对象作为值对象，如图 8-20 所示。

如果这时键 B 也要创建一个同样保存了整数值 100 的字符串对象作为值对象，那么服务器有以下两种做法：

1）为键 B 新创建一个包含整数值 100 的字符串对象；

2）让键 A 和键 B 共享同一个字符串对象；

以上两种方法很明显是第二种方法更节约内存。

在 Redis 中，让多个键共享同一个值对象需要执行以下两个步骤：

1）将数据库键的值指针指向一个现有的值对象；

2）将被共享的值对象的引用计数增一。

举个例子，图 8-21 就展示了包含整数值 100 的字符串对象同时被键 A 和键 B 共享之后的样子，可以看到，除了对象的引用计数从之前的 1 变成了 2 之外，其他属性都没有变化。共享对象机制对于节约内存非常有帮助，数据库中保存的相同值对象越多，对象共享机制就能节约越多的内存。

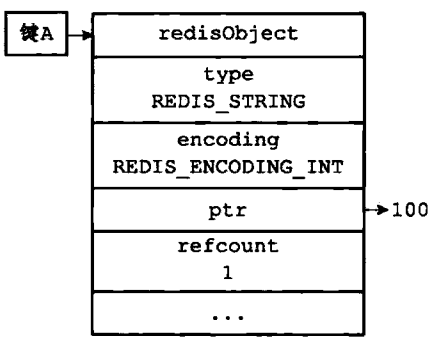

图 8-20 未被共享的字符串对象

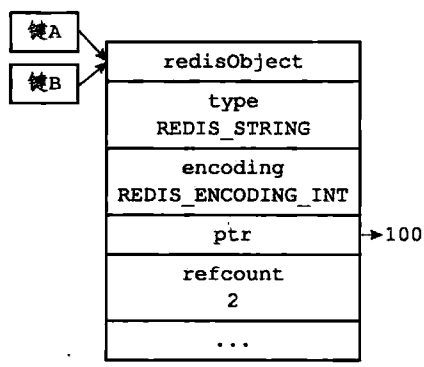

图 8-21 被共享的字符串对象

例如，假设数据库中保存了整数值 100 的键不只有键 A 和键 B 两个，而是有一百个，那么服务器只需要用一个字符串对象的内存就可以保存原本需要使用一百个字符串对象的内存才能保存的数据。

目前来说，Redis 会在初始化服务器时，创建一万个字符串对象，这些对象包含了从 0 到 9999 的所有整数值，当服务器需要用到值为 0 到 9999 的字符串对象时，服务器就会使用这些共享对象，而不是新创建对象。

注意

创建共享字符串对象的数量可以通过修改 redis.h/REDIS_SHARED_INTEGERS 常量来修改。

举个例子，如果我们创建一个值为 100 的键 A，并使用 *OBJECT REFCOUNT* 命令查看键 A 的值对象的引用计数，我们会发现值对象的引用计数为 2：

```
redis> SET A 100
OK

redis> OBJECT REFCOUNT A
(integer) 2
```

引用这个值对象的两个程序分别是持有这个值对象的服务器程序，以及共享这个值对象的键 A，如图 8-22 所示。

如果这时我们再创建一个值为 100 的键 B，那么键 B 也会指向包含整数值 100 的共享对象，使得共享对象的引用计数值变为 3：

```
redis> SET B 100
OK

redis> OBJECT REFCOUNT A
(integer) 3

redis> OBJECT REFCOUNT B
(integer) 3
```

图 8-23 展示了共享值对象的三个程序。

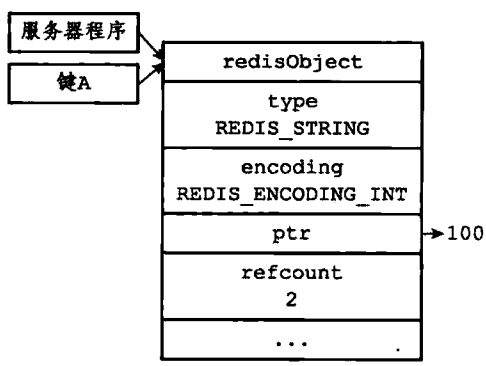

图 8-22 引用数为 2 的共享对象

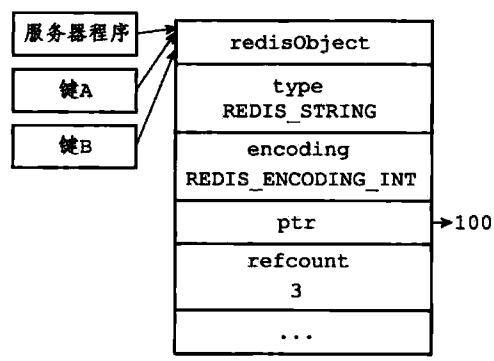

图 8-23 引用数为 3 的共享对象

另外，这些共享对象不单单只有字符串键可以使用，那些在数据结构中嵌套了字符串对象的对象（linkedlist 编码的列表对象、hashtable 编码的哈希对象、hashtable 编码的集合对象，以及 zset 编码的有序集合对象）都可以使用这些共享对象。

为什么 Redis 不共享包含字符串的对象？

当服务器考虑将一个共享对象设置为键的值对象时，程序需要先检查给定的共享对象和键想创建的目标对象是否完全相同，只有在共享对象和目标对象完全相同的情况下，程序才会将共享对象用作键的值对象，而一个共享对象保存的值越复杂，验证共享对象和目标对象是否相同所需的复杂度就会越高，消耗的 CPU 时间也会越多：

- 如果共享对象是保存整数值的字符串对象，那么验证操作的复杂度为 $O(1)$；
- 如果共享对象是保存字符串值的字符串对象，那么验证操作的复杂度为 $O(N)$；
- 如果共享对象是包含了多个值（或者对象的）对象，比如列表对象或者哈希对象，那么验证操作的复杂度将会是 $O(N^2)$。

因此，尽管共享更复杂的对象可以节约更多的内存，但受到 CPU 时间的限制，Redis 只对包含整数值的字符串对象进行共享。

8.10 对象的空转时长

除了前面介绍过的 type、encoding、ptr 和 refcount 四个属性之外，redisObject 结构包含的最后一个属性为 lru 属性，该属性记录了对象最后一次被命令程序访问的时间：

```
typedef struct redisObject {

    // ...

    unsigned lru:22;

    // ...

} robj;
```

OBJECT IDLETIME 命令可以打印出给定键的空转时长，这一空转时长就是通过将当前时间减去键的值对象的 lru 时间计算得出的：

```
redis> SET msg "hello world"
OK

# 等待一小段时间
redis> OBJECT IDLETIME msg
(integer) 20

# 等待一阵子
redis> OBJECT IDLETIME msg
(integer) 180

# 访问 msg 键的值
redis> GET msg
```

```
"hello world"

# 键处于活跃状态，空转时长为 0
redis> OBJECT IDLETIME msg
(integer) 0
```

注意

OBJECT IDLETIME 命令的实现是特殊的，这个命令在访问键的值对象时，不会修改值对象的 lru 属性。

除了可以被 *OBJECT IDLETIME* 命令打印出来之外，键的空转时长还有另外一项作用：如果服务器打开了 maxmemory 选项，并且服务器用于回收内存的算法为 volatile-lru 或者 allkeys-lru，那么当服务器占用的内存数超过了 maxmemory 选项所设置的上限值时，空转时长较高的那部分键会优先被服务器释放，从而回收内存。

配置文件的 maxmemory 选项和 maxmemory-policy 选项的说明介绍了关于这方面的更多信息。

8.11 重点回顾

- Redis 数据库中的每个键值对的键和值都是一个对象。
- Redis 共有字符串、列表、哈希、集合、有序集合五种类型的对象，每种类型的对象至少都有两种或以上的编码方式，不同的编码可以在不同的使用场景上优化对象的使用效率。
- 服务器在执行某些命令之前，会先检查给定键的类型能否执行指定的命令，而检查一个键的类型就是检查键的值对象的类型。
- Redis 的对象系统带有引用计数实现的内存回收机制，当一个对象不再被使用时，该对象所占用的内存就会被自动释放。
- Redis 会共享值为 0 到 9999 的字符串对象。
- 对象会记录自己的最后一次被访问的时间，这个时间可以用于计算对象的空转时间。

第二部分

单机数据库的实现

第9章

数 据 库

本章将对Redis服务器的数据库实现进行详细介绍，说明服务器保存数据库的方法，客户端切换数据库的方法，数据库保存键值对的方法，以及针对数据库的添加、删除、查看、更新操作的实现方法等。除此之外，本章还会说明服务器保存键的过期时间的方法，以及服务器自动删除过期键的方法。最后，本章还会说明Redis 2.8新引入的数据库通知功能的实现方法。

9.1 服务器中的数据库

Redis服务器将所有数据库都保存在服务器状态redis.h/redisServer结构的db数组中，db数组的每个项都是一个redis.h/redisDb结构，每个redisDb结构代表一个数据库：

```
struct redisServer {

    // ...

    // 一个数组，保存着服务器中的所有数据库
    redisDb *db;

    // ...

};
```

在初始化服务器时，程序会根据服务器状态的dbnum属性来决定应该创建多少个数据库：

```
struct redisServer {

    // ...

    // 服务器的数据库数量
    int dbnum;

    // ...

};
```

dbnum 属性的值由服务器配置的 database 选项决定，默认情况下，该选项的值为 16，所以 Redis 服务器默认会创建 16 个数据库，如图 9-1 所示。

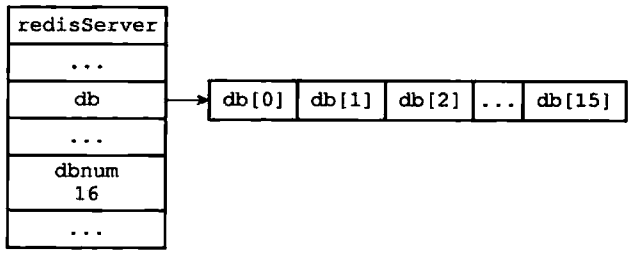

图 9-1 服务器数据库示例

9.2 切换数据库

每个 Redis 客户端都有自己的目标数据库，每当客户端执行数据库写命令或者数据库读命令的时候，目标数据库就会成为这些命令的操作对象。

默认情况下，Redis 客户端的目标数据库为 0 号数据库，但客户端可以通过执行 *SELECT* 命令来切换目标数据库。

以下代码示例演示了客户端在 0 号数据库设置并读取键 msg，之后切换到 2 号数据库并执行类似操作的过程：

```
redis> SET msg "hello world"
OK

redis> GET msg
"hello world"

redis> SELECT 2
OK

redis[2]> GET msg
(nil)

redis[2]> SET msg"another world"
OK

redis[2]> GET msg
"another world"
```

在服务器内部，客户端状态 redisClient 结构的 db 属性记录了客户端当前的目标数据库，这个属性是一个指向 redisDb 结构的指针：

```
typedef struct redisClient {

// ...
// 记录客户端当前正在使用的数据库

redisDb *db;

// ...
```

```
} redisClient;
```

redisClient.db 指针指向 redisServer.db 数组的其中一个元素，而被指向的元素就是客户端的目标数据库。

比如说，如果某个客户端的目标数据库为 1 号数据库，那么这个客户端所对应的客户端状态和服务器状态之间的关系如图 9-2 所示。

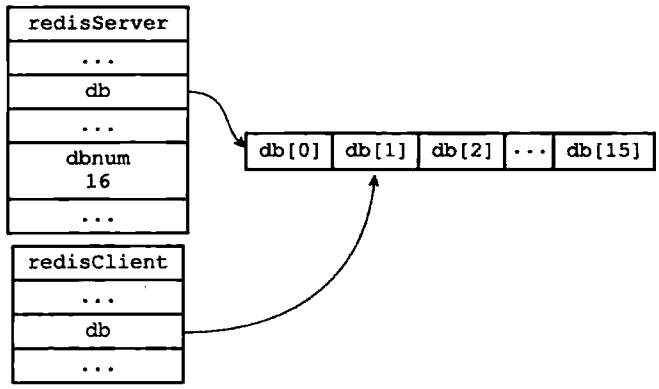

图 9-2 客户端的目标数据库为 1 号数据库

如果这时客户端执行命令 SELECT 2，将目标数据库改为 2 号数据库，那么客户端状态和服务器状态之间的关系将更新成图 9-3。

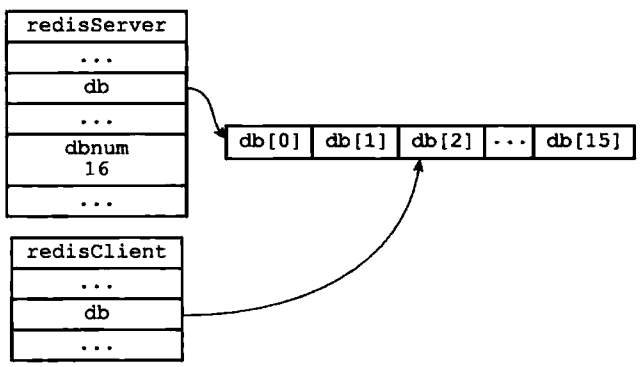

图 9-3 客户端的目标数据库为 2 号数据库

通过修改 redisClient.db 指针，让它指向服务器中的不同数据库，从而实现切换目标数据库的功能——这就是 *SELECT* 命令的实现原理。

谨慎处理多数据库程序

到目前为止，Redis 仍然没有可以返回客户端目标数据库的命令。虽然 redis-cli 客户端会在输入符旁边提示当前所使用的目标数据库：

```
redis> SELECT 1
OK

redis[1]> SELECT 2
OK
redis[2]>
```

但如果你在其他语言的客户端中执行 Redis 命令，并且该客户端没有像 redis-cli 那样一直显示目标数据库的号码，那么在数次切换数据库之后，你很可能会忘记自己当前正在使用的是哪个数据库。当出现这种情况时，为了避免对数据库进行误操作，在执行 Redis 命令特别是像 *FLUSHDB* 这样的危险命令之前，最好先执行一个 *SELECT* 命令，显式地切换到指定的数据库，然后才执行别的命令。

9.3 数据库键空间

Redis 是一个键值对（key-value pair）数据库服务器，服务器中的每个数据库都由一个 redis.h/redisDb 结构表示，其中，redisDb 结构的 dict 字典保存了数据库中的所有键值对，我们将这个字典称为键空间（key space）：

```
typedef struct redisDb {

    // ...

    // 数据库键空间，保存着数据库中的所有键值对
    dict *dict;

    // ...

} redisDb;
```

键空间和用户所见的数据库是直接对应的：

- 键空间的键也就是数据库的键，每个键都是一个字符串对象。
- 键空间的值也就是数据库的值，每个值可以是字符串对象、列表对象、哈希表对象、集合对象和有序集合对象中的任意一种 Redis 对象。

举个例子，如果我们在空白的数据库中执行以下命令：

```
redis> SET message "hello world"
OK

redis> RPUSH alphabet "a" "b" "c"
(integer)3

redis> HSET book name "Redis in Action"
(integer) 1

redis> HSET book author "Josiah L. Carlson"
(integer) 1

redis> HSET book publisher "Manning"
(integer) 1
```

那么在这些命令执行之后，数据库的键空间将会是图 9-4 所展示的样子：

- alphabet 是一个列表键，键的名字是一个包含字符串 "alphabet" 的字符串对象，键的值则是一个包含三个元素的列表对象。
- book 是一个哈希表键，键的名字是一个包含字符串 "book" 的字符串对象，键的值则是一个包含三个键值对的哈希表对象。
- message 是一个字符串键，键的名字是一个包含字符串 "message" 的字符串对象，键的值则是一个包含字符串 "hello world" 的字符串对象。

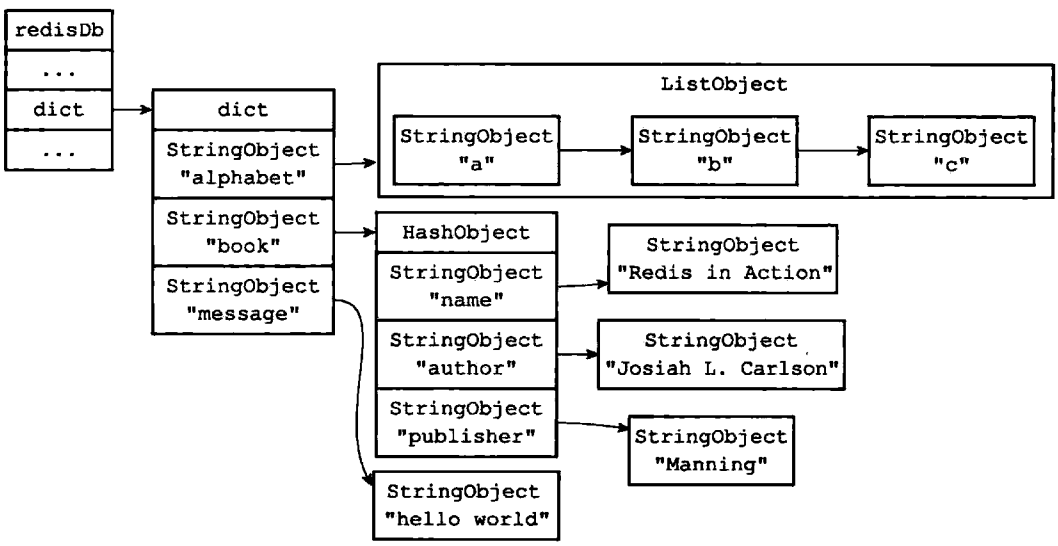

图 9-4 数据库键空间例子

因为数据库的键空间是一个字典，所以所有针对数据库的操作，比如添加一个键值对到数据库，或者从数据库中删除一个键值对，又或者在数据库中获取某个键值对等，实际上都是通过对键空间字典进行操作来实现的，以下几个小节将分别介绍数据库的添加、删除、更新、取值等操作的实现原理。

9.3.1 添加新键

添加一个新键值对到数据库，实际上就是将一个新键值对添加到键空间字典里面，其中键为字符串对象，而值则为任意一种类型的 Redis 对象。

举个例子，如果键空间当前的状态如图 9-4 所示，那么在执行以下命令之后：

```
redis> SET date "2013.12.1"
OK
```

键空间将添加一个新的键值对，这个新键值对的键是一个包含字符串 "date" 的字符串对象，而键值对的值则是一个包含字符串 "2013.12.1" 的字符串对象，如图 9-5 所示。

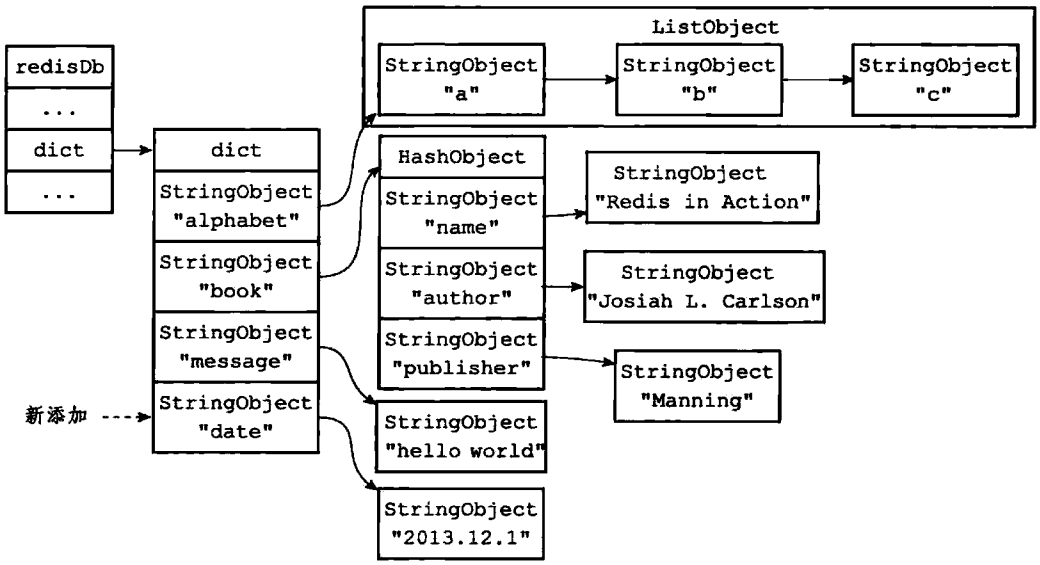

图 9-5 添加 date 键之后的键空间

9.3.2 删除键

删除数据库中的一个键，实际上就是在键空间里面删除键所对应的键值对对象。

举个例子，如果键空间当前的状态如图 9-4 所示，那么在执行以下命令之后：

```
redis> DEL book
(integer) 1
```

键 book 以及它的值将从键空间中被删除，如图 9-6 所示。

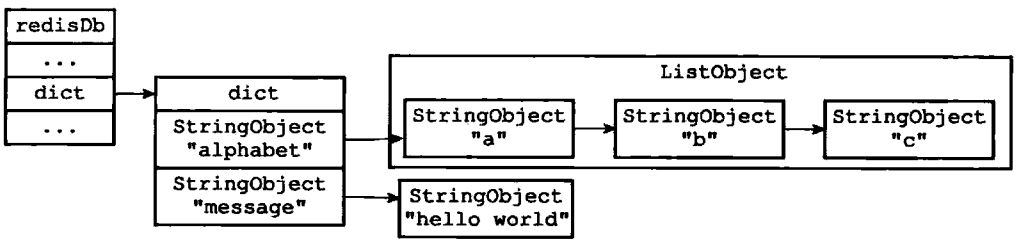

图 9-6 删除 book 键之后的键空间

9.3.3 更新键

对一个数据库键进行更新，实际上就是对键空间里面键所对应的值对象进行更新，根据值对象的类型不同，更新的具体方法也会有所不同。

举个例子，如果键空间当前的状态如图 9-4 所示，那么在执行以下命令之后：

```
redis> SET message "blah blah"
OK
```

键 message 的值对象将从之前包含 "hello world" 字符串更新为包含 "blah blah" 字符串，如图 9-7 所示。

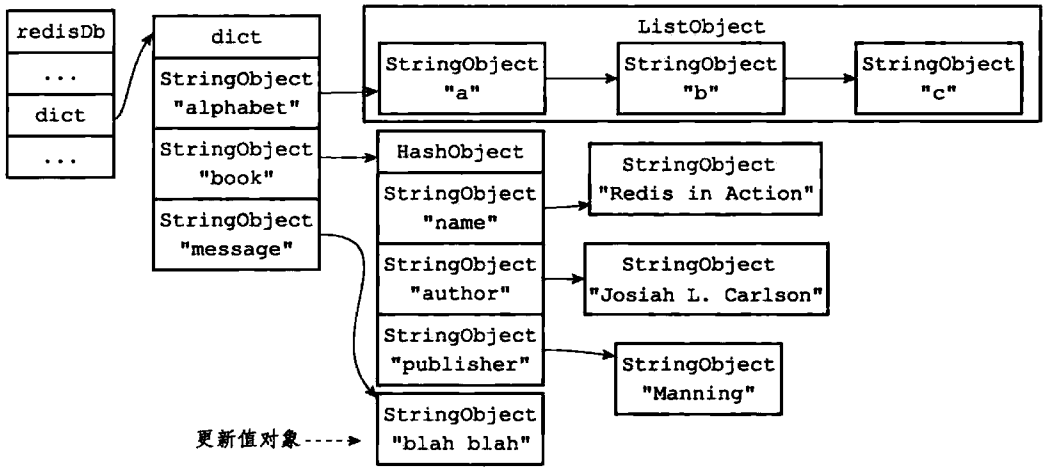

图 9-7 使用 SET 命令更新 message 键

再举个例子，如果我们继续执行以下命令：

```
redis> HSET book page 320
(integer) 1
```

那么键空间中 book 键的值对象（一个哈希对象）将被更新，新的键值对 page 和 320 会被添加到值对象里面，如图 9-8 所示。

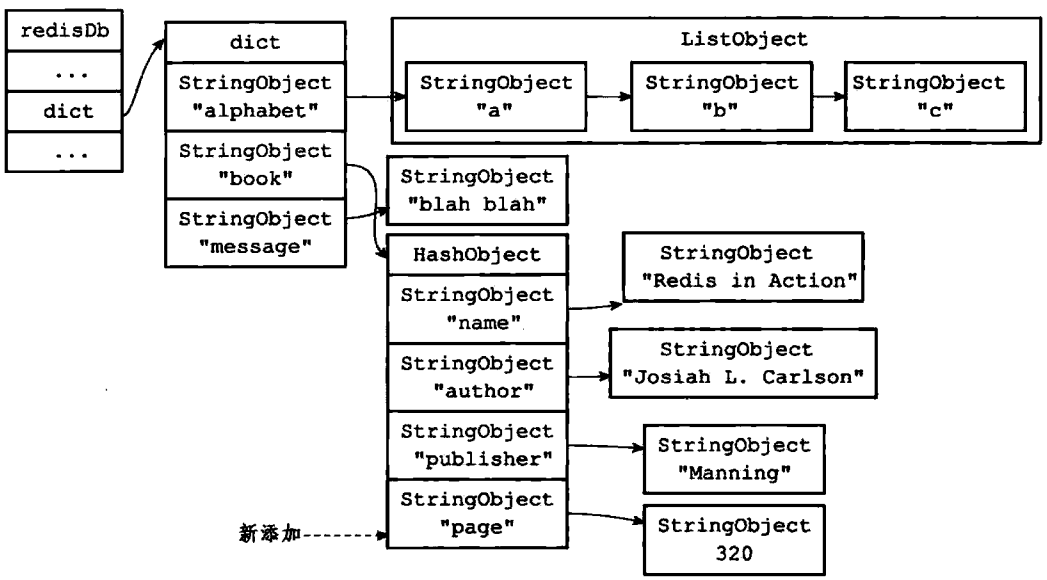

图 9-8 使用 HSET 更新 book 键