HZ BOOKS
华章科技

数据库
技术丛书

Redis
设计与实现

黄健宏 著

The Design and Implementation of Redis

- 系统而全面地描述了 Redis 内部运行机制。
- 图示丰富，描述清晰，并给出大量参考信息，是NoSQL数据库开发人员案头必备。
- 包括大部分Redis单机特征，以及所有多机特性。

机械工业出版社
China Machine Press

数据库技术丛书

Redis 设计与实现

黄健宏 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Redis设计与实现/黄健宏著. —北京：机械工业出版社，2014.4
（数据库技术丛书）

ISBN 978-7-111-46474-7

I. R… II. 黄… III. 数据库—基本知识 IV. TP311.13

中国版本图书馆CIP数据核字（2014）第079820号

本书全面而完整地讲解了 Redis 的内部机制与实现方式，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，展示了这些功能的核心数据结构以及关键的算法思想，图示丰富，描述清晰，并给出大量参考信息。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，更好、更高效地使用 Redis。

本书主要分为四大部分。第一部分“数据结构与对象”介绍了 Redis 中的各种对象及其数据结构，并说明这些数据结构如何影响对象的功能和性能。第二部分“单机数据库的实现”对 Redis 实现单机数据库的方法进行了介绍，包括数据库、RDB 持久化、AOF 持久化、事件等。第三部分“多机数据库的实现”对 Redis 的 Sentinel、复制、集群三个多机功能进行了介绍。第四部分“独立功能的实现”对 Redis 中各个相对独立的功能模块进行了介绍，涉及发布与订阅、事务、Lua 脚本、排序、二进制位数组、慢查询日志、监视器等。本书作者专门维护了 www.redisbook.com 网站，提供带有详细注释的 Redis 源代码，以及本书相关的更新内容。

Redis设计与实现

黄健宏 著

出版发行：机械工业出版社（北京市西城区百万庄大街22号 邮政编码：100037）

责任编辑：吴 怡　　责任校对：殷 虹

印　　刷：　　版　　次：2014年6月第1版第1次印刷

开　　本：186mm×240mm 1/16　　印　　张：25.25

书　　号：ISBN 978-7-111-46474-7　　定　　价：79.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzjsj@hzbook.com

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。

目　　录

第二部分 单机数据库的实现

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。

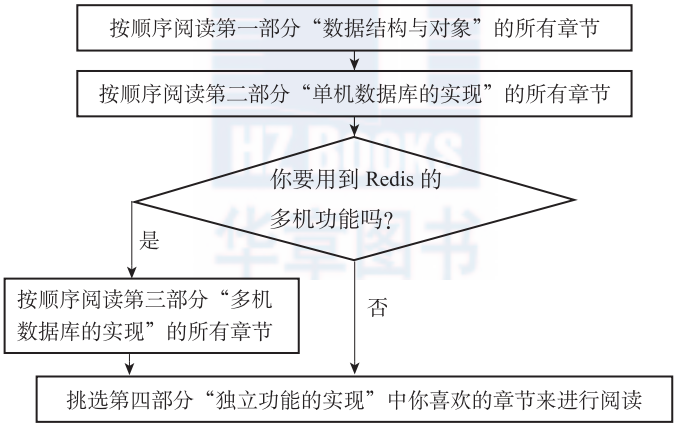

图 1-1 推荐阅读方法

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[];

};
```

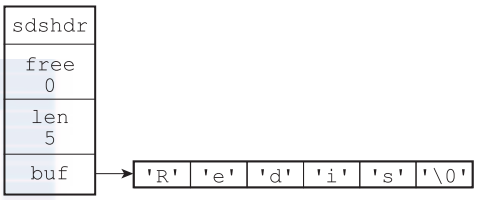

图 2-1　SDS 示例

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）。

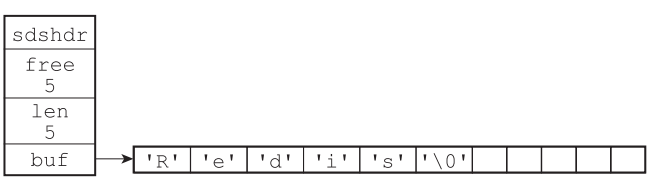

图 2-2　带有未使用空间的 SDS 示例

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。

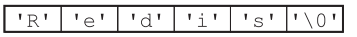

图 2-3 C 字符串

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。

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)。

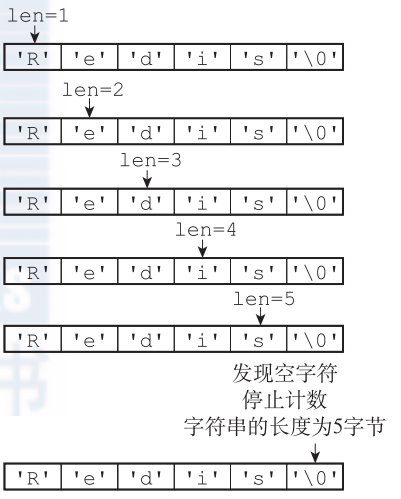

图 2-4 计算 C 字符串长度的过程

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。

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。

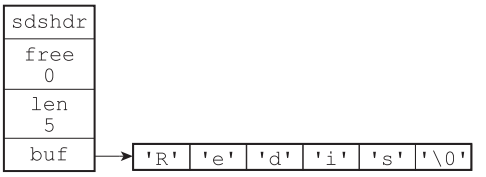

图 2-5 5 字节长的 SDS

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)。

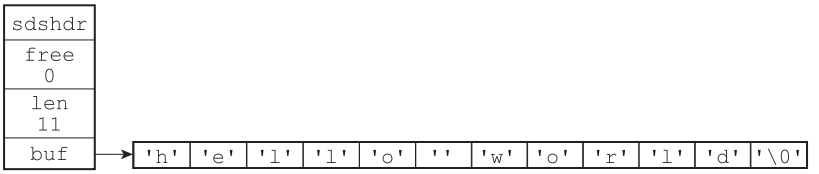

图 2-6　11 字节长的 SDS

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。

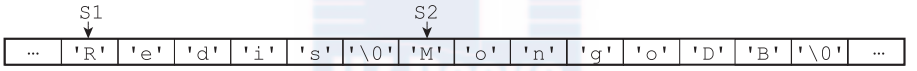

图 2-7　在内存中紧邻的两个 C 字符串

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。

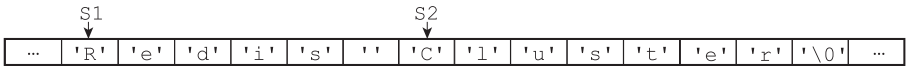

图 2-8　s1 的内容溢出到了 s2 所在的位置上

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。

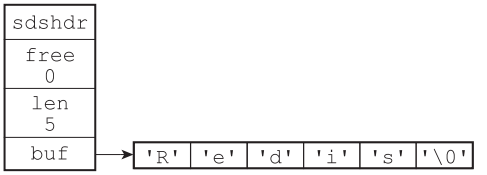

图 2-9 sdscat 执行之前的 SDS

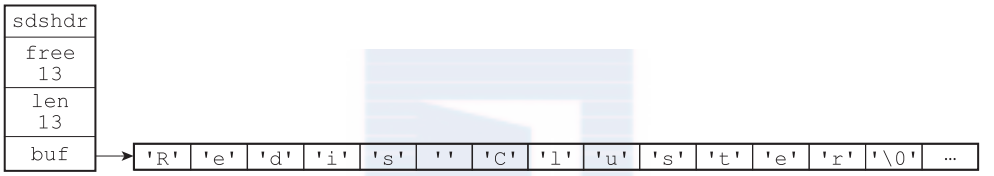

图 2-10 sdscat 执行之后的 SDS

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");
```

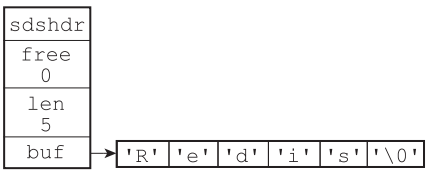

图 2-11　执行 sdscat 之前的 SDS

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。

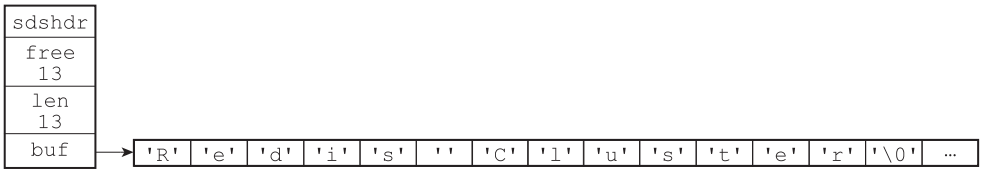

图 2-12 执行 sdscat 之后 SDS

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。

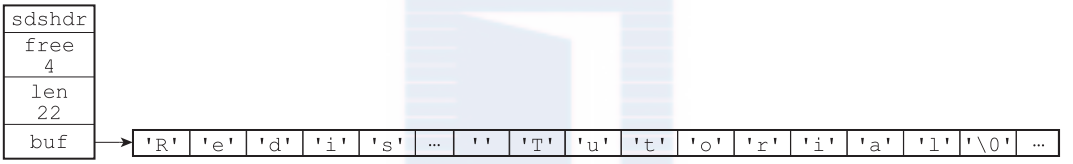

图 2-13 再次执行 sdscat 之后的 SDS

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：

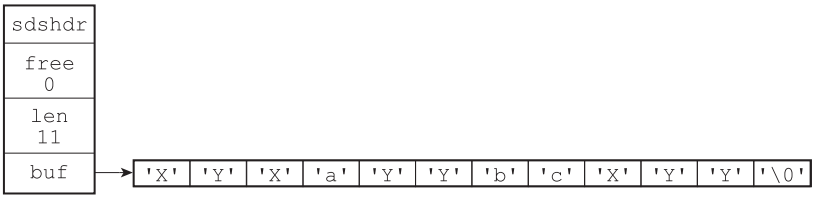

图 2-14 执行 sdstrim 之前的 SDS

```
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。

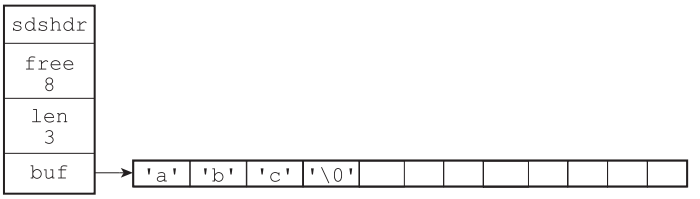

图 2-15　执行 sdstrim 之后的 SDS

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。

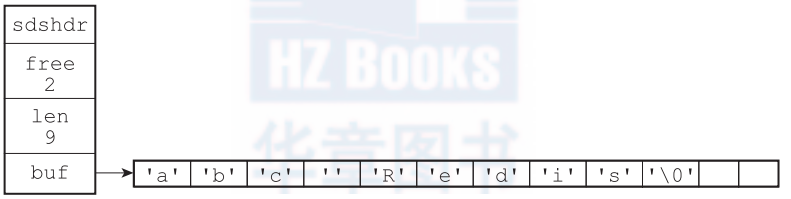

图 2-16　执行 sdscat 之后的 SDS

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"。

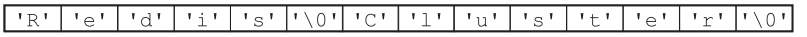

图 2-17 使用空字符来分割单词的特殊数据格式

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。

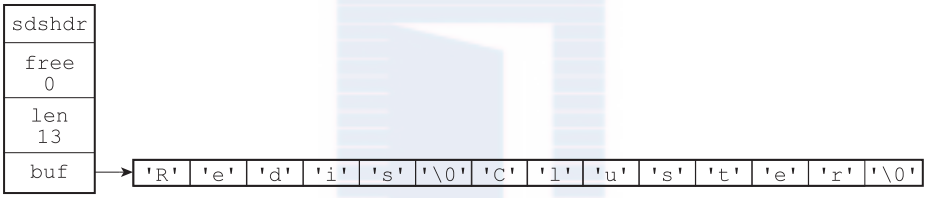

图 2-18 保存了特殊数据格式的 SDS

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。

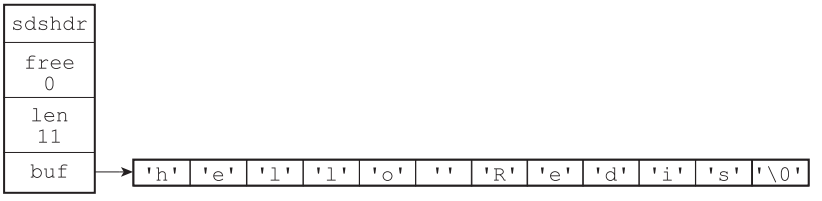

图 2-19 一个保存着文本数据的 SDS

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。

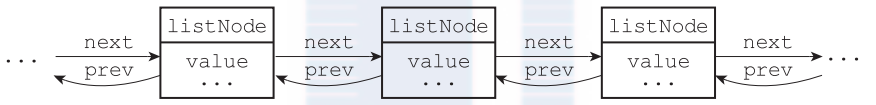

图 3-1 由多个 listNode 组成的双端链表

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。

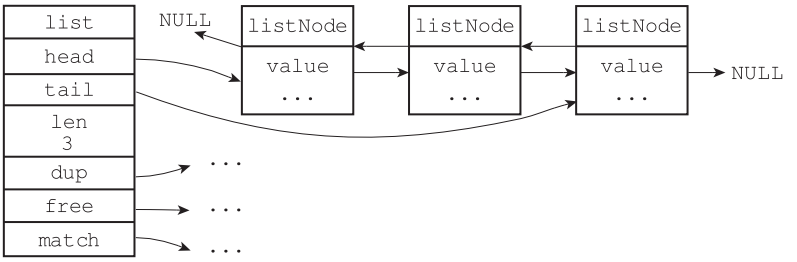

图 3-2　由 list 结构和 listNode 结构组成的链表

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。