《Redis使用手册》（试读版）

黄健宏/著

关于本文档

本文档为《Redis使用手册》一书的试读版本，由 RedisGuide.com 免费提供。

因为本文档为书本原稿的未编辑未排版版本，所以本文档和正式出版的图书在内容和编排上将会稍有不同，具体的细节以正式出版的图书为准。

《Redis使用手册》预计将于 2019 年 9 月下旬正式开始发售， 届时你将能够在 RedisGuide.com 购买到本书的完整版本。

版权声明

本文档及《Redis使用手册》一书中的全部内容版权归机械工业出版社所有，受著作权法律保护，任何人未经允许不得将本文档及其内容用于任何商业用途，违者必究。

目录

以下是《Redis使用手册》一书的完整目录，其中标题后面带 * 的大章都在本文档提供了试读。

前言*
1. 引言*

第一部分：数据结构与应用

2. 字符串（String）*
3. 散列（Hash）*
4. 列表（List）*
5. 集合（Set）*
6. 有序集合（Sorted Set）
7. HyperLogLog
8. 位图（bitmap）
9. 地理坐标（GEO）
10. 流（Stream）

第二部分：附加功能

11. 数据库*
12. 自动过期*
13. 流水线与事务*
14. Lua 脚本
15. 持久化
16. 发布与订阅
17. 模块

第三部分：多机功能

18. 复制*
19. Sentinel
20. 集群

附录

附录 A ： Redis 安装方法*
附录 B ： redis-py 安装方法*

前言

时光荏苒， 距离我的第一本书《Redis设计与实现》出版已经过去了整整五年。 在这五年间， Redis 从一个不为人熟知、只有少量应用的崭新数据库， 逐渐变成了内存数据库领域的事实标准。

五年之前， 当人们提到 Redis 的时候， 语气通常都充满了怀疑：“Redis 我还是第一次听说，它好用吗？”“Redis 比起 Memcached 有什么优势？”“用 Redis 储存数据安全吗，不会丢数据吧？！”。 然而时至今日， 经过大量的实践应用， Redis 简洁高效、安全稳定的印象已经深入人心。 无论是国内还是国外， 从财富五百强到小型初创公司都在使用 Redis ， 很多云服务提供商还以 Redis 为基础构建了相应的缓存服务、消息队列服务以及内存存储服务 —— 当你使用这些服务时， 你实际上就是在使用 Redis 。

除了变得越来越受欢迎之外， Redis 在过去数年的另一个变化就是更新速度越来越快， 功能也变得越来越多、越来越强大： 比如说， Redis 的数据结构数量已经从过去的五种增加到了九种， RDB-AOF 混合持久化模式的引入使得用户不必再陷入“鱼和熊掌不可兼得”的难题中， 而集群功能和模块机制的引入则让 Redis 在性能和功能上拥有了近乎无限的扩展能力。

综上所述， 我们可以说现在的 Redis 跟五年前比起来已经完全不一样了， 而如何向读者讲述新版 Redis 方方面面的变化， 则是每一本 Redis 书都必须回答的问题。 本书以服务 Redis 初学者和使用者为目标， 介绍了 Redis 日常使用中最常用到的部分， 并以“命令描述+代码示例”的模式详细列举了各个 Redis 命令的用法和用例。 我相信无论是刚开始学习 Redis 的读者， 还是每天都要使用 Redis 的读者， 在阅读本书的时候都会有所收获。

虽然本书在写作的过程中已经思虑再三并且数易其稿， 但百密一疏， 书中难免还是会有错误或者遗漏的地方。 如果读者朋友在阅读的过程中发现任何错误或是有任何疑问或建议， 都可以通过邮箱 huangz1990@gmail.com 或者 huangz.me 中列出的联系方式来联系我。 由于技术研究和写作工作较为繁重， 本人可能无法每封邮件都予以回复， 但只要有来信我就一定会阅读， 决不食言。

最后， 感谢吴怡编辑在写作过程中给我的帮助和指导， 感谢赵亮宇编辑为本书出版所做的努力， 还有感谢我的家人和朋友， 如果没有他们的关怀和支持， 本书是不可能顺利完成。

黄健宏
2019 年 8 月于清远

1. 引言

欢迎来到本书的第一章。 在这一章， 我们首先会了解到一些关于 Redis 的基本信息， 比如它提供了什么功能、它能做什么、它的优点是什么、有哪些公司使用它等等。

之后我们会快速地了解本书各个章节的具体编排， 并完成一些学习 Redis 的前期准备工作， 比如安装 Redis 服务器等等。 在一切准备就绪之后， 我们就会开始学习如何执行 Redis 命令， 以及如何通过配置选项对 Redis 服务器进行配置。

在本章的最后， 我们还会看到获取本书示例代码的方法， 并知悉本书使用的 Redis 版本以及本书配套的读者服务网站。

1.1 Redis 简介

Redis 是一个主要由 Salvatore Sanfilippo （antirez）开发的开源的内存数据结构存储器， 它经常被用作数据库、缓存以及消息代理等用途。

Redis 因为它丰富的数据结构、极快的速度、齐全的功能而为人所知， 它是目前内存数据库方面的事实标准， 在互联网上有非常广泛的应用， 包括微博、Twitter、GitHub、Stack Overflow、知乎等国内外公司都大量地使用了 Redis 。

Redis 之所以广受开发者欢迎， 跟它自身拥有强大的功能以及简洁的设计不无关系。

虽然 Redis 拥有各式各样的特点和优点， 但其中最重要的不外乎以下这些：

- **丰富多样的数据结构**

  Redis 为用户提供了字符串、散列、列表、集合、有序集合、HyperLogLog、位图、流、地理坐标等一系列丰富的数据结构， 每种数据结构都适用于解决特定的问题。 在有需要的时候， 用户还可以通过事务、Lua 脚本、模块等特性， 扩展已有数据结构的功能， 甚至从零实现自己专属的数据结构。 通过这些数据结构和特性， Redis 可以保证， 用户总是可以使用最适合的工具去解决手头上的问题。
- **完备的功能**

  在上述这些数据结构的基础上， Redis 提供了很多非常有用和实用的附加功能， 比如自动过期、流水线、事务、数据持久化等， 这些功能能够帮助用户将 Redis 应用在更多不同的场景中， 或者给予用户以方便。 更重要的是， Redis 不仅可以单机使用， 还可以多机使用： 通过 Redis 自带的复制、Sentinel 和集群功能， 用户可以将自己的数据库扩展至任意大小。 无论你运营的是一个小型的个人网站， 还是一个为上千万消费者服务的热门站点， 你都可以在 Redis 找到你想要的功能， 并将其部署到你的服务器里面。

- **风驰电掣般的执行速度**

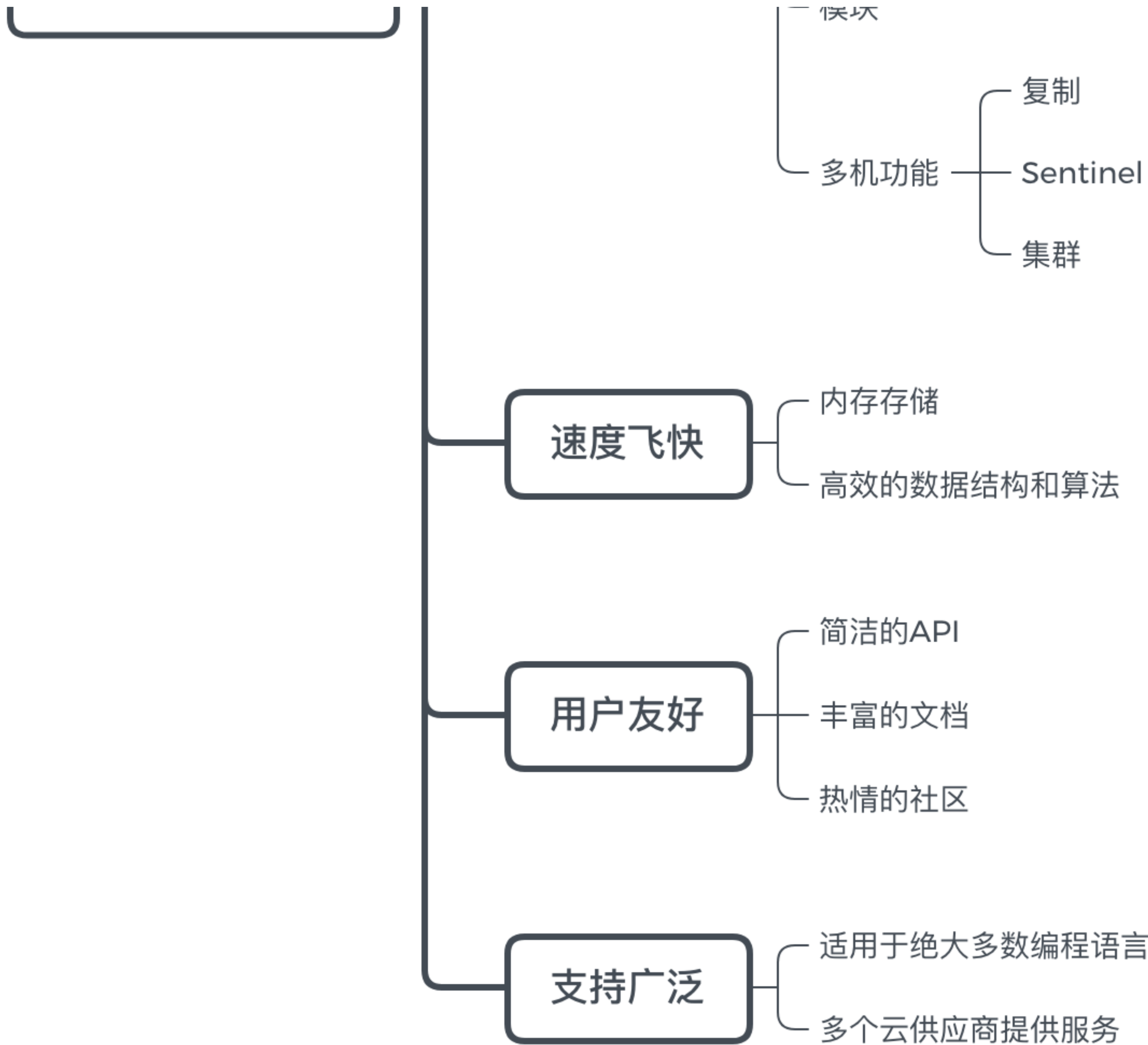
  Redis 是一款内存数据库， 它将所有数据都储存在内存里面。 因为计算机访问内存的速度要远远高于访问硬盘的速度， 因此与基于硬盘设计的传统数据库相比， Redis 在数据的存取速度方面具有天然的优势。 虽然说“背靠大树好乘凉”， 但 Redis 并没有因为自己拥有天然的速度优势就放弃了自己在效率方面的追求。 与此相反， Redis 的开发者在实现各项数据结构和特性的时候都经过了大量考量， 在底层选用了很多非常高效的数据结构和算法， 以此来确保每个操作都可以在尽可能短的时间内完成， 并且尽可能地节省内存。
- **对用户友好的API、文档以及社区**

  “虽然 Redis 提供了很多很棒的数据结构和特性， 但如果它们使用起来非常困难的话， 那么这一切就没有意义。”——如果你对此有所担心的话， 那么现在可以打消你的疑虑了！ Redis API 遵循的是 UNIX “一次只做一件事，并把它做好”的设计哲学， Redis 的 API 虽然丰富， 但它们大部分都非常简短， 并且只需接受几个参数就可以完成用户指定的操作。 更棒的是， Redis 在官方网站（redis.io）上为每个 API 以及相关特性都提供了详尽的文档， 并且客户端本身也可以在线查询这些文档。 当你遇到文档无法解决的问题时， 还可以在 Redis 项目的 GitHub 页面（github.com/antirez/redis）、Google Group（groups.google.com/forum/#!forum/redis-db）甚至作者的 Twitter（twitter.com/antirez） 上提问。
- **广泛的支持**

  正如之前所说， Redis 已经得到了互联网公司的广泛使用， 许多开发者为不同的编程语言开发了相应的客户端（redis.io/clients）， 绝大多数编程语言的使用者都可以轻而易举地找到他们所需的客户端， 然后直接开始使用 Redis 。 此外，包括亚马逊、谷歌、RedisLabs、阿里云和腾讯云在内的多个云服务提供商都提供了基于 Redis 或兼容 Redis 的服务， 如果你不打算自己搭建 Redis 服务器， 那么上述的这些提供商可能是一个不错的选择。

图 1-1 Redis 特色一览

- Redis特色
  - 结构丰富
    - 集合
    - 有序集合
    - HyperLogLog
    - 位图
    - 地理坐标
    - 流
  - 功能完备
    - 数据库管理
    - 自动过期
    - 流水线
    - 事务
    - Lua脚本
    - 持久化
    - 发布与订阅



多机功能
复制
Sentinel
集群
速度飞快
内存存储
高效的数据结构和算法
用户友好
简洁的API
丰富的文档
热情的社区
支持广泛
适用于绝大多数编程语言
多个云供应商提供服务

1.2 内容编排

本书由数据结构与应用、附加功能和多机功能三个部分共二十章组成。

在数据结构与应用部分， 书本介绍了 Redis 核心的九种数据结构， 列举了操作这些数据结构的众多命令及其详细信息， 并在其中穿插介绍了多个使用 Redis 命令构建应用程序的示例。 通过这些程序示例， 读者可以进一步加深对命令的认识， 并学会如何在实际中应用这些命令， 从而达到学而致用的目的。

附加功能部分介绍了 Redis 在数据结构的基础上， 为用户提供的额外功能。 其中包括管理数据结构的数据库管理功能和自动过期功能， 将数据结构持久化至硬盘从而避免数据丢失的持久化功能， 提高多条命令执行效率的流水线功能， 保证命令安全性的事务和 Lua 脚本功能， 还有扩展服务器特性的模块功能等等。 这些功能在为用户提供方便的同时， 也进一步扩大了 Redis 的适用范围， 读者可以通过阅读这一部分来学会如何将 Redis 应用在更多场景中。

多机功能部分介绍了 Redis 的三项多机功能， 它们分别是复制、Sentinel 和集群。 其中复制用于创建多个 Redis 服务器的副本， 并藉此提升整个 Redis 系统的读性能以及容灾能力。 至于 Sentinel 则在复制的基础上， 为 Redis 系统提供了自动的故障转移功能， 从而使得整个系统可以更健壮地运行。 最后， 通过使用 Redis 集群， 用户可以在线扩展 Redis 系统的读写能力。 读者可以通过阅读这一部分来获得扩展 Redis 读写性能的相关知识， 并根据自己的情况为 Redis 系统选择合适的扩展方式。

1.3 目标读者

本书面向所有 Redis 初学者和 Redis 使用者， 是学习和日常使用 Redis 必不可少的一本书。

一方面， 对于 Redis 初学者来说， 本书的章节经过妥善的编排， 按照从简单到复杂的顺序详细罗列了 Redis 的各项特性， 因此 Redis 初学者只需要沿着书本一直阅读下去就可以循序渐进地学习到具体的 Redis 知识， 而穿插其中的应用示例则让读者有机会亲自实践书中介绍的命令知识， 真正做到学以致用。

另一方面， 对于 Redis 使用者来说， 本书包含了大量 Redis 新版特性的介绍， 读者可以通过本书了解到最新的 Redis 知识。 除此之外， 对于日常的命令文档查找和应用示例查找， 本书在目录处也做了优化， 读者可以通过书本的目录和附录快速定位命令和示例， 非常便于日常查阅。

1.4 预备工作

本书包含大量 Redis 命令操作实例和 Python 代码应用示例， 执行和测试这些示例需要用到 Redis 服务器及其附带的 redis-cli 客户端、Python 编程环境和 redis-py 客户端， 如果你尚未安装这些软件， 那么请查阅本书的附录 A 和附录 B 并按照指引进行安装。

在正确安装 Redis 服务器之后， 你应该可以通过执行以下命令来启动 Redis 服务器：

```
$ redis-server
28393:C 02 Jul 2019 23:49:25.952 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
28393:C 02 Jul 2019 23:49:25.952 # Redis version=999.999.999, bits=64, commit=0cabe0cf, modified=1, pi
28393:C 02 Jul 2019 23:49:25.952 # Warning: no config file specified, using the default config. In ord
28393:M 02 Jul 2019 23:49:25.953 * Increased maximum number of open files to 10032 (it was originally
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 999.999.999 (0cabe0cf/1) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 28393
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

28393:M 02 Jul 2019 23:49:25.954 # Server initialized
28393:M 02 Jul 2019 23:49:25.954 * Ready to accept connection
```

并通过以下命令启动 redis-cli 客户端：

```
$ redis-cli
127.0.0.1:6379>
```

以及通过以下命令启动 Python 解释器并载入 redis-py 库：

```
$ python3
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] on darwin
```

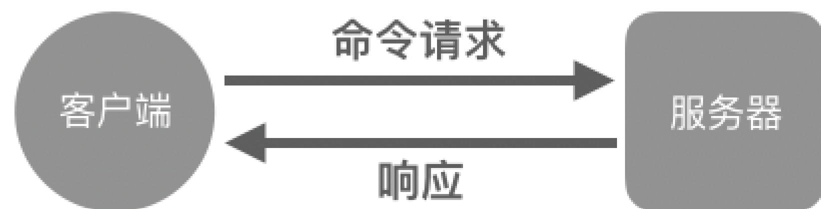
```
Type "help", "copyright", "credits" or "license" for more information.
>>> from redis import Redis
>>>
```

在上述准备工作圆满完成之后， 我们就可以开始学习 Redis 命令的基本知识了。

1.5 执行命令

Redis 服务器通过接收客户端发送的命令请求来执行指定的命令， 并在命令执行完毕之后通过响应将命令的执行结果返回给客户端， 至于结果的内容则被称为命令回复。

图 1-2 命令请求与响应



Redis 为每种数据结构和功能特性都提供了一簇相应的命令， 学习如何使用这些命令是学习 Redis 的重中之重。 幸运的是， 大部分 Redis 命令都非常简单， 只需要接受少量几个参数就可以完成非常强大的操作。

Redis 的所有命令都由一个命令名后跟任意多个参数以及可选项组成：

```
COMMAND [arg1 arg2 arg3 ...] [[OPTION1 value1] [OPTION2 value2] [...]]
```

在本书中， 命令和可选项的名字通常以大写字母形式出现， 而命令参数和可选项的值则以小写字母形式出现。 比如上例中的 `COMMAND` 就是命令的名字， 而 `OPTION1` 和 `OPTION2` 则是可选项的名字。 至于 `arg1` 、 `arg2` 和 `arg3` 则是命令的参数， 而 `value1` 和 `value2` 则是可选项的值。

命令描述中的方括号 `[]` 仅用于包围命令中可选的参数和选项， 在执行命令的时候并不需要给出这些方括号。 最后， 命令描述中的 `...` 用于表示命令接受任意数量的参数或可选项。

好的， 关于 Redis 命令格式的描述已经足够多了， 现在让我们来看一个实际的例子。 Redis 的 `PING` 命令接受一条可选的消息作为参数， 这个命令通常用于测试客户端和服务器之间的连接是否正常：

```
PING [message]
```

如果用户以无参数形式执行这个命令， 那么服务器在连接正常的情况下， 将向客户端返回 `"PONG"` 作为回复：

```
127.0.0.1:6379> PING
PONG
```

但是， 如果用户给定了可选的消息， 那么服务器将原封不动地向客户端返回该消息：

```
127.0.0.1:6379> PING "hello world"
"hello world"
```

另一方面， 如果服务器与客户端的连接不正常， 那么客户端将返回一个错误：

```
-- 客户端未能连接服务器，返回一个连接错误
127.0.0.1:6379> PING
Could not connect to Redis at 127.0.0.1:6379: Connection refused
```

我们为这个命令调用添加了一条注释， 用于说明客户端遇到的问题。 在本书中， redis-cli 客户端的命令执行示例都使用 `--` 作为注释前缀， 这些注释仅用于对被执行的命令做进一步的说明， 它们并不是被执行命令的一部分。

图 1-3 在 redis-cli 中执行 Redis 命令

1.6 配置服务器

在阅读本书的过程中， 有时候我们还需要使用配置选项对 Redis 服务器进行配置， 这一点可以通过两种方法来完成。

第一种方法是在启动 Redis 服务器的时候给定配置选项作为参数， 格式为：

```
$ redis-server --OPTION1 [value1 value2 ...] --OPTION2 [value1 value2 ...] [...]
```

比如说， Redis 服务器默认使用 `6379` 作为端口号， 但如果你想要使用 `10086` 而不是 `6379` 作为端口号的话， 那么可以在启动 Redis 服务器时通过给定 `port` 可选项来指定你想要的端口号：

```
$ redis-server --port 10086
```

向 Redis 服务器提供配置选项的第二种方法则是在启动 Redis 服务器的时候为其提供配置文件， 并将想要修改的配置选项写在配置文件里面：

```
$ redis-server /path/to/your/file
```

比如说， 为了将 Redis 服务器的端口号改为 12345 ， 我们可以在当前文件夹中创建配置文件 `myredis.conf` ，并在文件中包含以下内容：

```
port 10086
```

然后在启动 Redis 服务器时向其提供该配置文件：

```
$ redis-server myredis.conf
```

1.7 示例代码

正如上面提到的那样， 本书提供了大量 Python 代码示例， 这些示例的源码可以通过访问以下页面获取：github.com/huangz1990/RedisGuide-code 。

本书在展示代码示例的同时， 会在示例标题的旁边给出源代码的具体访问路径。 比如说， 对于代码清单 1-1 中展示的连接检查脚本 `check_connection.py` 来说， 该文件就位于 `/introduction` 文件夹中：

代码清单 1-1 检查连接的脚本： `/introduction/check_connection.py`

```python
from redis import Redis

client = Redis()

# ping() 方法在连接正常时将返回 True
if client.ping() is True:
    print("connecting")
else:
    print("disconnected")
```

1.8 版本说明

本书基于 Redis 5.0 版本撰写， 它是书本创作期间 Redis 的最新版本。

为了方便使用旧版 Redis 的读者， 本书在介绍每个命令和特性的时候都指出了它们具体可用的版本， 读者通过查阅这一信息就可以知道特定的命令和特性在自己的版本中是否可用。

另一方面， 得益于 Redis 极好的向后兼容性， 即使读者将来使用的是 Redis 6.0 、7.0 甚至更新的版本， 本书的绝大部分知识对于读者来说仍将是有效的。

1.9 读者服务器网站

本书配套了读者服务网站 RedisGuide.com ， 上面列举了书本的介绍信息、购买链接、目录、试读章节、示例代码和勘误等内容， 有兴趣的读者朋友可以上去浏览一下。

1.10 启程！

一切准备就绪， 是时候开始我们的 Redis 旅程了。 在接下来的一章， 我们将开始学习 Redis 最基本的数据结构 —— 字符串。

2. 字符串（String）

字符串键是 Redis 最基本的键值对类型， 这种类型的键值对会在数据库里面把单独的一个键和单独的一个值关联起来， 被关联的键和值既可以是普通的文字数据， 也可以是图片、视频、音频、压缩文件等更为复杂的二进制数据。

作为例子， 图 2-1 展示了数据库视角下的四个字符串键， 其中：

- 与键 `"message"` 相关联的值是 `"hello world"` ；
- 与键 `"number"` 相关联的值是 `"10086"` ；
- 与键 `"homepage"` 相关联的值是 `"redis.io"` ；
- 与键 `"redis-logo.jpg"` 相关联的值是二进制数据 `"\xff\xd8\xff\xe0\x00\x10JFIF\x00..."` 。

图 2-1 数据库中的字符串键示例

数据库

“message”	→	“hello world”
“number”	→	“10086”
“homepage”	→	“redis.io”
“redis-logo.jpg”	→	“\xff\xd8\xff\xe0\x00\x10JFIF\x00…”

Redis 为字符串键提供了一系列操作命令， 通过使用这些命令， 用户可以：

- 为字符串键设置值。
- 获取字符串键的值。
- 在获取旧值的同时为字符串键设置新值。
- 同时为多个字符串键设置值，或者同时获取多个字符串键的值。
- 取得字符串值的长度。
- 获取字符串值指定索引范围上的内容，或者对字符串值指定索引范围上的内容进行修改。
- 将一些内容追加到字符串值的末尾。
- 对字符串键储存的整数值或者浮点数值执行加法操作或减法操作。

本章接下来将对以上提到的这些字符串键命令进行介绍， 并演示如何使用这些命令去解决各种实际的问题。

2.1 SET：为字符串键设置值

创建字符串键最常用的方法就是使用 `SET` 命令， 这个命令可以为一个字符串键设置相应的值。 在最基本的情况下， 用户只需要向 `SET` 命令提供一个键和一个值就可以了：

```
SET key value
```

跟之前提到过的一样， 这里的键和值既可以是文字也可以是二进制数据。

SET 命令在成功创建字符串键之后将返回 OK 作为结果。 比如说， 通过执行以下命令， 我们可以创建出一个字符串键， 它的键为 "number" ， 值为 "10086" ：

```
redis> SET number "10086"
OK
```

又比如说， 通过执行以下命令， 我们可以创建出一个键为 "book" ， 值为 "The Design and Implementation of Redis" 的字符串键：

```
redis> SET book "The Design and Implementation of Redis"
OK
```

图 2-2 和图 2-3 分别展示了数据库在以上两条 SET 命令执行之前以及执行之后的状态。

图 2-2 执行 SET 命令之前，数据库的状态

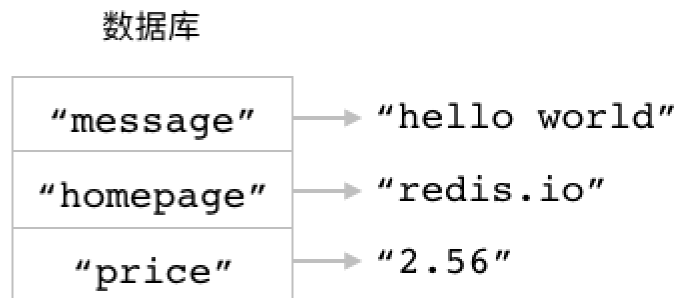


图 2-3 执行 SET 命令之后，数据库的状态

数据库

键	值
“message”	“hello world”
“homepage”	“redis.io”
“price”	“2.56”
“number”	“10086”
“book”	“The Design and Implementation of Redis”

Note: 数据库键的存放方式

为了方便阅读， 本书总会将数据库中新出现的键放置到已有键的下方。 比如在上面展示的数据库图 2-3 里面， 我们就将新添加的 `"number"` 键和 `"book"` 键放置到了已有键的下方。

在实际中， Redis 数据库是以无序的方式存放数据库键的， 一个新加入的键可能会出现在数据库的任何位置上， 因此我们在使用 Redis 的过程中不应该对键在数据库中的摆放位置做任何假设， 以免造成错误。

2.1.1 改变覆盖规则

在默认情况下， 对一个已经设置了值的字符串键执行 SET 命令将导致键的旧值被新值覆盖。

举个例子， 如果我们连续执行以下两条 SET 命令， 那么第一条 SET 命令设置的值将被第二条 SET 命令设置的值所覆盖：

```
redis> SET song_title "Get Wild"
OK

redis> SET song_title "Running to Horizon"
OK
```

在第二条 SET 命令执行完毕之后， song_title 键的值将从原来的 "Get Wild" 变为 "Running to Horizon" 。

从 Redis 2.6.12 版本开始， 用户可以通过向 SET 命令提供可选的 NX 选项或者 XX 选项来指示 SET 命令是否要覆盖一个已经存在的值：

```
SET key value [NX|XX]
```

如果用户在执行 SET 命令时给定了 NX 选项， 那么 SET 命令只会在键没有值的情况下执行设置操作， 并返回 OK 表示设置成功； 如果键已经存在， 那么 SET 命令将放弃执行设置操作， 并返回空值 nil 表示设置失败。

以下代码展示了带有 NX 选项的 SET 命令的行为：

```
redis> SET password "123456" NX
OK    -- 对尚未有值的 password 键进行设置，成功

redis> SET password "999999" NX
(nil)    -- password 键已经有了值，设置失败
```

因为第二条 SET 命令没有改变 password 键的值， 所以 password 键的值仍然是刚开始时设置的 "123456" 。

另一方面， 如果用户在执行 SET 命令时给定了 XX 选项， 那么 SET 命令只会在键已经有值的情况下执行设置操作， 并返回 OK 表示设置成功； 如果给定的键并没有值， 那么 SET 命令将放弃执行设置操作， 并返回空值表示设置失败。

举个例子， 如果我们对一个没有值的键 mongodb-homepage 执行以下 SET 命令， 那么命令将因为 XX 选项的作用而放弃执行设置操作：

```
redis> SET mongodb-homepage "mongodb.com" XX
(nil)
```

相反地， 如果我们对一个已经有值的键执行带有 XX 选项的 SET 命令， 那么命令将使用新值去覆盖已有的旧值：

```
redis> SET mysql-homepage "mysql.org"
OK    -- 为键 mysql-homepage 设置一个值

redis> SET mysql-homepage "mysql.com" XX
OK    -- 对键的值进行更新
```

在第二条 SET 命令执行之后， mysql-homepage 键的值将从原来的 "mysql.org" 更新为 "mysql.com" 。

2.1.2 其他信息

属性	值
复杂度	O(1)
版本要求	不带任何可选项的 SET 命令从 Redis 1.0.0 版本开始可用； 带有 NX 、 XX 等可选项的 SET 命令从 Redis 2.6.12 版本开始可用。

2.2 GET：获取字符串键的值

用户可以通过使用 GET 命令， 从数据库里面获取指定字符串键的值：

```
GET key
```

GET 命令接受一个字符串键作为参数， 然后返回与该键相关联的值。

比如对于图 2-4 所示的数据库来说， 我们可以通过执行以下 GET 命令来取得各个字符串键相关联的值：

```
redis> GET message
"hello world"

redis> GET number
"10086"

redis> GET homepage
"redis.io"
```

图 2-4 使用 GET 命令获取数据库键的值

数据库

"message"	→ "hello world"
"homepage"	→ "redis.io"
"price"	→ "2.56"
"number"	→ "10086"
"book"	→ "The Design and Implementation of Redis"

另一方面， 如果用户给定的字符串键在数据库中并没有与之相关联的值， 那么 GET 命令将返回一个空值：

```
redis> GET date
(nil)
```

上面这个 GET 命令的执行结果表示数据库中并不存在 date 键， 它也没有与之相关联的值。

因为 Redis 的数据库要求所有键必须拥有与之相关联的值， 所以如果一个键有值， 那么我们就说这个键存在于数据库； 相反地， 如果一个键没有值， 那么我们就说这个键不存在于数据库。 比如对于上面展示的几个键来说， date 键就不存在于数据库， 而 message 键、 number 键和 homepage 键则存在于数据库。

2.2.1 其他信息

属性	值
复杂度	O(1)
版本要求	GET 命令从 Redis 1.0.0 开始可用。

2.3 GETSET：获取旧值并设置新值

GETSET 命令就像 GET 命令和 SET 命令的组合版本， 它首先获取字符串键目前已有的值， 接着为键设置新值， 最后把之前获取到的旧值返回给用户：

```
GETSET key new_value
```

以下代码展示了如何使用 GETSET 命令去获取 number 键的旧值并为它设置新值：

```
redis> GET number     -- number 键现在的值为 "10086"
"10086"

redis> GETSET number "12345"
"10086"     -- 返回旧值

redis> GET number     -- number 键的值已被更新为 "12345"
"12345"
```

另一方面， 如果被设置的键并不存在于数据库， 那么 GETSET 命令将返回空值作为键的旧值：

```
redis> GET counter
(nil)     -- 键不存在

redis> GETSET counter 50
(nil)     -- 返回空值作为旧值

redis> GET counter
"50"
```

2.3.1 其他信息

属性	值
复杂度	O(1)
版本要求	GETSET 命令从 Redis 1.0.0 开始可用。

2.4 示例：缓存

对数据进行缓存是 Redis 最常见的用法之一： 因为 Redis 把数据储存在内存而不是硬盘上面， 并且访问内存数据的速度比访问硬盘数据的速度要快得多， 所以用户可以通过把需要快速访问的数据储存在 Redis 里面来提升应用程序访问这些数据时的速度。

代码清单 2-1 展示了一个使用 Redis 实现的缓存程序代码， 这个程序使用 `SET` 命令来将需要被缓存的数据储存到指定的字符串键里面， 并使用 `GET` 命令来从指定的字符串键里面获取被缓存的数据。

代码清单 2-1 使用字符串键实现的缓存程序： `/string/cache.py`

```
class Cache:

    def __init__(self, client):
        self.client = client

    def set(self, key, value):
        """
        把需要被缓存的数据储存到键 key 里面,
        如果键 key 已经有值, 那么使用新值去覆盖旧值。
        """
        self.client.set(key, value)

    def get(self, key):
        """
        获取储存在键 key 里面的缓存数据,
        如果数据不存在, 那么返回 None 。
        """
        return self.client.get(key)

    def update(self, key, new_value):
        """
        对键 key 储存的缓存数据进行更新,
        并返回键 key 在被更新之前储存的缓存数据。
        如果键 key 之前并没有储存数据,
        那么返回 None 。
        """
        return self.client.getset(key, new_value)
```

除了用于设置缓存的 `set()` 方法以及用于获取缓存的 `get()` 方法之外， 缓存程序还提供了由 `GETSET` 命令实现的 `update()` 方法： 这个方法可以让用户在对缓存进行设置的同时， 获得之前被缓存的旧值。 用户可以根据自己的需要决定是使用 `set()` 方法还是 `update()` 方法对缓存进行设置。

以下代码展示了怎样使用这个程序来缓存一个 HTML 页面， 并在有需要时获取它：

```
>>> from redis import Redis
>>> from cache import Cache
>>> client = Redis(decode_responses=True)  # 使用文本编码方式打开客户端
>>> cache = Cache(client)
>>> cache.set("greeting-page", "<html><p>hello world</p></html>")
>>> cache.get("greeting-page")
'<html><p>hello world</p></html>'
>>> cache.update("greeting-page", "<html><p>good morning</p></html>")
'<html><p>hello world</p></html>'
>>> cache.get("greeting-page")
'<html><p>good morning</p></html>'
```

因为 Redis 的字符串键不仅可以储存文本数据， 还可以储存二进制数据， 所以这个缓存程序不仅可以用来缓存网页等文本数据， 还可以用来缓存图片和视频等二进制数据。 比如说， 如果你正在运营一个图片网站， 那么你同样可以使用这个缓存程序来缓存网站上的热门图片， 从而提高用户访问这些热门图片的速度。

作为例子， 以下代码展示了将 Redis 的 Logo 图片缓存到键 `redis-logo.jpg` 里面的方法：

```
>>> from redis import Redis
>>> from cache import Cache
>>> client = Redis()                       # 使用二进制编码方式打开客户端
>>> cache = Cache(client)
>>> image = open("redis-logo.jpg", "rb")   # 以二进制只读方式打开图片文件
>>> data = image.read()                    # 读取文件内容
>>> image.close()                          # 关闭文件
>>> cache.set("redis-logo.jpg", data)      # 将内存缓存到键 redis-logo.jpg 里面
>>> cache.get("redis-logo.jpg")[:20]       # 读取二进制数据的前 20 个字节
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00H\x00H\x00\x00'
```

Note: 在测试以上两段代码的时候， 请务必以正确的编码方式打开客户端（第一段代码采用文本方式，第二段代码采用二进制方式）， 否则测试代码将会出现编码错误。

2.5 示例：锁

锁是一种同步机制， 它可以保证一项资源在任何时候只能被一个进程使用， 如果有其他进程想要使用相同的资源， 那么它们就必须等待， 直到正在使用资源的进程放弃使用权为止。

一个锁实现通常会有获取（acquire）和释放（release）这两种操作：

- 获取操作用于取得资源的独占使用权。 在任何时候， 最多只能有一个进程取得锁， 我们把成功取得锁的这个进程称之为锁的持有者。 在锁已经被持有的情况下， 所有尝试再次获取锁的操作都会失败。
- 释放操作用于放弃资源的独占使用权， 一般由锁的持有者调用。 在锁被释放之后， 其他进程就可以再次尝试获取这个锁了。

代码清单 2-2 展示了一个使用字符串键实现的锁程序， 这个程序会根据给定的字符串键是否有值来判断锁是否已经被获取， 而针对锁的获取操作和释放操作则是分别通过设置字符串键和删除字符串键来完成的。

代码清单 2-2 使用字符串键实现的锁程序： `/string/lock.py`

```
VALUE_OF_LOCK = "locking"

class Lock:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def acquire(self):
        """
        尝试获取锁。
        成功时返回 True ，失败时返回 False 。
        """
        result = self.client.set(self.key, VALUE_OF_LOCK, nx=True)
        return result is True

    def release(self):
        """
        尝试释放锁。
        成功时返回 True ，失败时返回 False 。
        """
        return self.client.delete(self.key) == 1
```

获取操作 `acquire()` 方法是通过执行带有 `NX` 选项的 `SET` 命令来实现的：

```
result = self.client.set(self.key, VALUE_OF_LOCK, nx=True)
```

`NX` 选项的效果确保了代表锁的字符串键只会在没有值的情况下被设置：

- 如果给定的字符串键没有值， 那么说明锁尚未被获取， `SET` 命令将执行设置操作， 并将 `result` 变量的值设置为 `True` ；
- 与此相反， 如果给定的字符串键已经有值了， 那么说明锁已经被获取， `SET` 命令将放弃执行设置操作， 并将 `result` 变量的值将为 `None` ；

`acquire()` 方法最后会通过检查 `result` 变量的值是否为 `True` 来判断自己是否成功取得了锁。

释放操作 `release()` 方法使用了本书之前没有介绍过的 `DEL` 命令， 这个命令接受一个或多个数据库键作为参数， 尝试删除这些键以及与之相关联的值， 并返回被成功删除的键数量作为结果：

```
DEL key [key ...]
```

因为 Redis 的 `DEL` 命令和 Python 的 `del` 关键字重名， 所以在 redis-py 客户端中， 执行 `DEL` 命令实际上是通过调用 `delete()` 方法来完成的：

```
self.client.delete(self.key) == 1
```

`release()` 方法通过检查 `delete()` 方法的返回值是否为 `1` 来判断删除操作是否执行成功： 如果用户尝试对一个尚未被获取的锁执行 `release()` 方法， 那么方法将返回 `false` ， 表示没有锁被释放。

在使用 `DEL` 命令删除代表锁的字符串键之后， 字符串键将重新回到没有值的状态， 这时用户就可以再次调用 `acquire()` 方法去获取锁了。

以下代码演示了这个锁的使用方法：

```
>>> from redis import Redis
>>> from lock import Lock
>>> client = Redis(decode_responses=True)
>>> lock = Lock(client, 'test-lock')
>>> lock.acquire()   # 成功获取锁
True
>>> lock.acquire()   # 锁已被获取，无法再次获取
False
>>> lock.release()   # 释放锁
True
>>> lock.acquire()   # 锁释放之后可以再次被获取
True
```

虽然代码清单 2-2 中展示的锁实现了基本的获取和释放功能， 但它并不完美：

1. 因为这个锁的释放操作无法验证进程的身份， 所以无论执行释放操作的进程是否就是锁的持有者， 锁都会被释放。 如果锁被持有者以外的其他进程释放了的话， 那么系统中可能就会同时出现多个锁， 导致锁的唯一性被破坏。
2. 这个锁的获取操作不能设置最大加锁时间， 它无法让锁在超过给定的时限之后自动释放。 因此， 如果持有锁的进程因为故障或者编程错误而没有在退出之前主动释放锁， 那么锁就会一直处于已被获取的状态， 导致其他进程永远无法取得锁。

本书后续将继续改进这个锁实现， 使得它可以解决这两个问题。

2.6 MSET：一次为多个字符串键设置值

除了 `SET` 命令和 `GETSET` 命令之外， Redis 还提供了 `MSET` 命令用于对字符串键进行设置。 跟 `SET` 命令和 `GETSET` 命令只能设置单个字符串键的做法不同， `MSET` 命令可以一次为多个字符串键设置值：

```
MSET key value [key value ...]
```

作为例子， 以下代码展示了如何使用一条 `MSET` 命令去设置 `message` 、 `number` 和 `homepage` 三个键：

```
redis> MSET message "hello world" number "10086" homepage "redis.io"
OK

redis> GET message
"hello world"

redis> GET number
"10086"

redis> GET homepage
"redis.io"
```

跟 `SET` 命令一样， `MSET` 命令也会在执行设置操作之后返回 `OK` 表示设置成功。 此外， 如果给定的字符串键已经有相关联的值， 那么 `MSET` 命令也会直接使用新值去覆盖已有的旧值。

比如以下代码就展示了怎样使用 `MSET` 命令去覆盖上一个 `MSET` 命令为 `message` 键和 `number` 键设置的值：

```
redis> MSET message "good morning!" number "12345"
OK

redis> GET message
"good morning!"

redis> GET number
"12345"
```

MSET 命令除了可以让用户更为方便地执行多个设置操作之外， 还能够有效地提高程序的效率： 执行多条 SET 命令需要客户端和服务器之间进行多次网络通信， 并因此耗费大量的时间； 通过使用一条 MSET 命令去代替多条 SET 命令， 可以将原本所需的多次网络通信降低为只需一次网络通信， 从而有效地减少程序执行多个设置操作时所需的时间。

2.6.1 其他信息

属性	值
复杂度	O(N)， 其中 N 为用户给定的字符串键数量。
版本要求	MSET 命令从 Redis 1.0.1 开始可用。

2.7 MGET：一次获取多个字符串键的值

MGET 命令就是一个多键版本的 GET 命令， 它接受一个或多个字符串键作为参数， 并返回这些字符串键的值：

```
MGET key [key ...]
```

MGET 命令返回一个列表作为结果， 这个列表按照用户执行命令时给定键的顺序排列各个键的值： 比如说， 列表的第一个元素就是第一个给定键的值， 而列表的第二个元素则是第二个给定键的值， 以此类推。

作为例子， 以下代码展示了如何使用一条 MGET 命令去获取 message 、 number 和 homepage 三个键的值：

```
redis> MGET message number homepage
1) "hello world"     -- message 键的值
2) "10086"           -- number 键的值
3) "redis.io"        -- homepage 键的值
```

跟 GET 命令一样， MGET 命令在碰到不存在的键时也会返回空值：

```
redis> MGET not-exists-key
1) (nil)
```

跟 MSET 命令类似， MGET 命令也可以将执行多个获取操作所需的网络通信次数从原来的 N 次降低至只需一次，从而有效地提高程序的运行效率。

2.7.1 其他信息

属性	值
复杂度	O(N)，其中 N 为用户给定的字符串键数量。
版本要求	MGET 命令从 Redis 1.0.0 开始可用。

2.8 MSETNX：只在键不存在的情况下，一次为多个字符串键设置值

MSETNX 命令跟 MSET 命令一样， 都可以对多个字符串键进行设置：

```
MSETNX key value [key value ...]
```

MSETNX 跟 MSET 的主要区别在于 MSETNX 只会在所有给定键都不存在的情况下对键进行设置， 而不会像 MSET 那样直接覆盖键已有的值： 如果在给定键当中， 有哪怕一个键已经有值了， 那么 MSETNX 命令也会放弃对所有给定键的设置操作。 MSETNX 命令在成功执行设置操作时返回 1 ， 在放弃执行设置操作时则返回 0 。

在以下的这段代码中， 因为键 k4 已经存在， 所以 MSETNX 将放弃对键 k1 、 k2 、 k3 和 k4 进行设置操作：

```
redis> MGET k1 k2 k3 k4
1) (nil)            -- 键 k1 、 k2 和 k3 都不存在
2) (nil)
3) (nil)
4) "hello world"    -- 键 k4 已存在

redis> MSETNX k1 "one" k2 "two" k3 "three" k4 "four"
(integer) 0    -- 因为键 k4 已存在，所以 MSETNX 未能执行设置操作
```

```
redis> MGET k1 k2 k3 k4     -- 各个键的值没有变化
1) (nil)
2) (nil)
3) (nil)
4) "hello world"
```

但是如果我们只对不存在的键 `k1` 、 `k2` 和 `k3` 进行设置， 那么 `MSETNX` 可以正常地完成设置操作：

```
redis> MSETNX k1 "one" k2 "two" k3 "three"
(integer) 1    -- 所有给定键都不存在，成功执行设置操作

redis> MGET k1 k2 k3 k4
1) "one"            -- 刚刚使用 MSETNX 设置的三个值
2) "two"
3) "three"
4) "hello world"    -- 之前已经存在的键 k4 的值没有改变
```

2.8.1 其他信息

属性	值
复杂度	O(N)，其中 N 为用户给定的字符串键数量。
版本要求	`MSETNX` 命令从 Redis 1.0.1 开始可用。

2.9 示例：储存文章信息

在构建应用程序的时候， 我们经常会需要批量地设置和获取多项信息。 以博客程序为例子：

- 当用户想要注册成为博客的作者时， 程序就需要把这位作者的名字、账号、密码、注册时间等多项信息储存起来， 并在用户登录的时候取出这些信息。
- 又比如说， 当博客的作者想要撰写一篇新文章的时候， 程序就需要把文章的标题、内容、作者、发表时间等多项信息储存起来， 并在用户阅读文章的时候取出这些信息。

通过使用 `MSET` 命令、 `MSETNX` 命令以及 `MGET` 命令， 我们可以实现上面提到的这些批量设置操作和批量获取操作。 比如代码清单 2-3 就展示了一个文章储存程序， 这个程序使用 `MSET` 命令和 `MSETNX` 命令将文章的标题、内容、作者、发表时间等多项信息储存到不同的字符串键里面， 并通过 `MGET` 命令从这些键里面获取文章的各项信息。

代码清单 2-3 文章储存程序：/string/article.py

```
from time import time  # time() 函数用于获取当前 Unix 时间戳

class Article:

    def __init__(self, client, article_id):
        self.client = client
        self.id = str(article_id)
        self.title_key = "article::" + self.id + "::title"
        self.content_key = "article::" + self.id + "::content"
        self.author_key = "article::" + self.id + "::author"
        self.create_at_key = "article::" + self.id + "::create_at"

    def create(self, title, content, author):
        """
        创建一篇新的文章，创建成功时返回 True ，
        因为文章已存在而导致创建失败时返回 False 。
        """
        article_data = {
            self.title_key: title,
            self.content_key: content,
            self.author_key: author,
            self.create_at_key: time()
        }
        return self.client.msetnx(article_data)

    def get(self):
        """
        返回 ID 对应的文章信息。
        """
        result = self.client.mget(self.title_key,
                                  self.content_key,
                                  self.author_key,
                                  self.create_at_key)
        return {"id": self.id, "title": result[0], "content": result[1],
                "author": result[2], "create_at": result[3]}

    def update(self, title=None, content=None, author=None):
        """
        对文章的各项信息进行更新，
        更新成功时返回 True ，失败时返回 False 。
        """
        article_data = {}
        if title is not None:
```

```
            article_data[self.title_key] = title
        if content is not None:
            article_data[self.content_key] = content
        if author is not None:
            article_data[self.author_key] = author
        return self.client.mset(article_data)
```

这个文章储存程序比较长， 让我们来逐个分析它的各项功能。 首先， `Article` 类的初始化方法 `__init__()` 接受一个 Redis 客户端和一个文章 ID 作为参数， 并将文章 ID 从数字转换为字符串：

```
self.id = str(article_id)
```

接着程序会使用这个字符串格式的文章 ID ， 构建出用于储存文章各项信息的字符串键的键名：

```
self.title_key = "article::" + self.id + "::title"
self.content_key = "article::" + self.id + "::content"
self.author_key = "article::" + self.id + "::author"
self.create_at_key = "article::" + self.id + "::create_at"
```

在这些键当中， 第一个键将用于储存文章的标题， 第二个键将用于储存文章的内容， 第三个键将用于储存文章的作者， 而第四个键则会用于储存文章的创建时间。

当用户想要根据给定的文章 ID 创建具体的文章时， 他就需要调用 `create()` 方法， 并传入文章的标题、内容以及作者作为参数。 `create()` 方法会把以上这些信息以及当前的 UNIX 时间戳放入到一个 Python 字典里面：

```
article_data = {
    self.title_key: title,
    self.content_key: content,
    self.author_key: author,
    self.create_at_key: time()
}
```

`article_data` 字典的键储存了代表文章各项信息的字符串键的键名， 而与这些键相关联的则是这些字符串键将要被设置的值。 接下来， 程序会调用 `MSETNX` 命令， 对字典中给定的字符串键进行设置：

```
self.client.msetnx(article_data)
```

因为 `create()` 方法的设置操作是通过 `MSETNX` 命令来进行的， 所以这一操作只会在所有给定字符串键都不存在的情况下进行：

- 如果给定的字符串键已经有值了， 那么说明与给定 ID 相对应的文章已经存在。 在这种情况下， `MSETNX` 命令将放弃执行设置操作， 并且 `create()` 方法也会向调用者返回 `False` 表示文章创建失败。
- 与此相反， 如果给定的字符串键尚未有值， 那么 `create()` 方法将根据用户给定的信息创建文章， 并在成功之后返回 `True` 。

在成功创建文章之后， 用户就可以使用 `get()` 方法去获取文章的各项信息了。 `get()` 方法会调用 `MGET` 命令， 从各个字符串键里面取出文章的标题、内容、作者等信息， 并把这些信息储存到 `result` 列表中：

```
result = self.client.mget(self.title_key,
                          self.content_key,
                          self.author_key,
                          self.create_at_key)
```

为了让用户可以更方便地访问文章的各项信息， `get()` 方法会将储存在 `result` 列表里面的文章信息放入到一个字典里面， 然后再返回给用户：

```
return {"id": self.id, "title": result[0], "content": result[1],
        "author": result[2], "create_at": result[3]}
```

这样做的好处有两点：

1. 它隐藏了 `get()` 方法由 `MGET` 命令实现这一底层细节。 如果程序直接向用户返回 `result` 列表， 那么用户就必须知道列表中的各个元素代表文章的哪一项信息， 然后通过列表索引来访问文章的各项信息。 这种做法非常不方便， 而且也非常容易出错。
2. 返回一个字典可以让用户以 `dict[key]` 这样的方式去访问文章的各个属性， 比如使用 `article["title"]` 去访问文章的标题， 使用 `article["content"]` 去访问文章的内容， 诸如此类， 这使得针对文章数据的各项操作可以更方便地进行。

另外要注意的一点是， 虽然用户可以通过访问 `Article` 类的 `id` 属性来获得文章的 ID ， 但是为了方便起见， `get()` 方法在返回文章信息的时候也会将文章的 ID 包含在字典里面一并返回。

对文章信息进行更新的 `update()` 方法是整个程序最复杂的部分。 首先， 为了让用户可以自由选择需要更新的信息项， 这个函数在定义时使用了 Python 的具名参数特性：

```python
def update(self, title=None, content=None, author=None):
```

通过具名参数， 用户可以根据自己想要更新的文章信息项来决定传入哪个参数， 而不需要更新的信息项则会被赋予默认值 `None` ：

- 比如说， 如果用户只想要更新文章的标题， 那么只需要调用 `update(title=new_title)` 即可；
- 又比如说， 如果用户想要同时更新文章的内容和作者， 那么只需要调用 `update(content=new_content, author=new_author)` 即可；

诸如此类。

在定义了具名参数之后， `update()` 方法会检查各个参数的值， 并将那些不为 `None` 的参数以及与之相对应的字符串键键名放入到 `article_data` 字典里面：

```python
article_data = {}
if title is not None:
    article_data[self.title_key] = title
if content is not None:
    article_data[self.content_key] = content
if author is not None:
    article_data[self.author_key] = author
```

`article_data` 字典中的键就是需要更新的字符串键的键名， 而与之相关联的则是这些字符串键的新值。

在一切准备就绪之后， `update()` 方法会根据 `article_data` 字典中设置好的键值对， 调用 `MSET` 命令对文章进行更新：

```python
self.client.mset(article_data)
```

以下代码展示了这个文章储存程序的使用方法：

```python
>>> from redis import Redis
>>> from article import Article
>>> client = Redis(decode_responses=True)
>>> article = Article(client, 10086)                    # 指定文章 ID
>>> article.create('message', 'hello world', 'peter')  # 创建文章
True
>>> article.get()                                       # 获取文章
{'id': '10086', 'title': 'message', 'content': 'hello world',
```

```
 'author': 'peter', 'create_at': '1551199163.4296808'}
>>> article.update(author="john")                    # 更新文章的作者
True
>>> article.get()                                     # 再次获取文章
{'id': '10086', 'title': 'message', 'content': 'hello world',
 'author': 'john', 'create_at': '1551199163.4296808'}
```

表 1–1 展示了上面这段代码创建出的键，以及这些键的值。

表 1–1 文章数据储存示例

被储存的内容	数据库中的键	键的值
文章的标题	`article::10086::title`	`'message'`
文章的内容	`article::10086::content`	`'hello world'`
文章的作者	`article::10086::author`	`'john'`
文章的创建时间戳	`article::10086::create_at`	`'1461145575.631885'`

Note: 键的命名格式

`Article` 程序使用了多个字符串键去储存文章信息， 并且每个字符串键的名字都是以 `article::<id>::<attribute>` 格式命名的， 这是一种 Redis 使用惯例： Redis 用户通常会为逻辑上相关联的键设置相同的前缀， 并通过分隔符来区分键名的各个部分， 以此来构建一种键的命名格式。

比如对于 `article::10086::title` 、 `article::10086::author` 这些键来说， `article` 前缀表明这些键都储存着与文章信息相关的数据， 而分隔符 `::` 则区分开了键名里面的前缀、ID 以及具体的属性。 除了 `::` 符号之外， 常用的键名分隔符还包括 `.` 符号， 比如 `article.10086.title` ； 或者 `->` 符号， 比如 `article->10086->title` ； 又或者 `|` 符号， 比如 `article|10086|title` ； 诸如此类。

分隔符的选择通常只是一个个人喜好的问题， 而键名的具体格式也可以根据需要进行构造： 比如说， 如果你不喜欢 `article::<id>::<attribute>` 格式， 那么也可以考虑使用 `article::<attribute>::<id>` 格式， 诸如此类。 唯一需要注意的是， 一个程序应该只使用一种键名分隔符， 并且持续地使用同一种键名格式， 以免造成混乱。

通过使用相同的格式去命名逻辑上相关联的键， 我们可以让程序产生的数据结构变得更容易被理解， 并且在有需要的时候， 还可以根据特定的键名格式， 在数据库里面以模式匹配的方式查找指定的键。

2.10 STRLEN：获取字符串值的字节长度

通过对字符串键执行 STRLEN 命令， 用户可以取得字符串键储存的值的字节长度：

```
STRLEN key
```

以下代码展示了如何使用 STRLEN 去获取不同字符串值的字节长度：

```
redis> GET number
"10086"

redis> STRLEN number    -- number 键的值长 5 字节
(integer) 5

redis> GET message
"hello world"

redis> STRLEN message   -- message 键的值长 11 字节
(integer) 11

redis> GET book
"The Design and Implementation of Redis"

redis> STRLEN book      -- book 键的值长 38 字节
(integer) 38
```

对于不存在的键， STRLEN 命令将返回 0 ：

```
redis> STRLEN not-exists-key
(integer) 0
```

2.10.1 其他信息

属性	值
复杂度	O(1)
版本要求	STRLEN 命令从 Redis 2.2.0 开始可用。

2.11 字符串值的索引

因为每个字符串都是由一系列连续的字节组成的， 所以字符串中的每个字节实际上都拥有与之相对应的索引。 Redis 为字符串键提供了一系列索引操作命令， 这些命令允许用户通过正数索引或者负数索引， 对字符串值的某个字节或者某个部分进行处理， 其中：

- 字符串值的正数索引以 0 为开始， 从字符串的开头向结尾不断递增；
- 字符串值的负数索引以 -1 为开始， 从字符串的结尾向开头不断递减。

图 2-5 就展示了值为 `"hello world"` 的字符串， 以及它的各个字节相对应的正数索引和负数索引。

图 2-5 字符串的索引示例

正数索引	0	1	2	3	4	5	6	7	8	9	10
负数索引	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'

本章接下来将对 `GETRANGE` 和 `SETRANGE` 这两个字符串键的索引操作命令进行介绍。

2.12 GETRANGE：获取字符串值指定索引范围上的内容

通过使用 `GETRANGE` 命令， 用户可以获取字符串值从 `start` 索引开始， 直到 `end` 索引为止的所有内容：

```
GETRANGE key start end
```

`GETRANGE` 命令接受的是闭区间索引范围， 也即是说， 位于 `start` 索引和 `end` 索引上的值也会被包含在命令返回的内容当中。

举个例子， 以下代码展示了如何使用 GETRANGE 命令去获取 message 键的值的不同部分：

```
redis> GETRANGE message 0 4      -- 获取字符串值索引 0 至索引 4 上的内容
"hello"

redis> GETRANGE message 6 10     -- 获取字符串值索引 6 至索引 10 上的内容
"world"

redis> GETRANGE message 3 7      -- 获取字符串值的中间部分
"lo wo"

redis> GETRANGE message -11 -7   -- 使用负数索引获取指定内容
"hello"
```

图 2-6 展示了上面的这四个命令是如何根据索引去获取值的内容的。

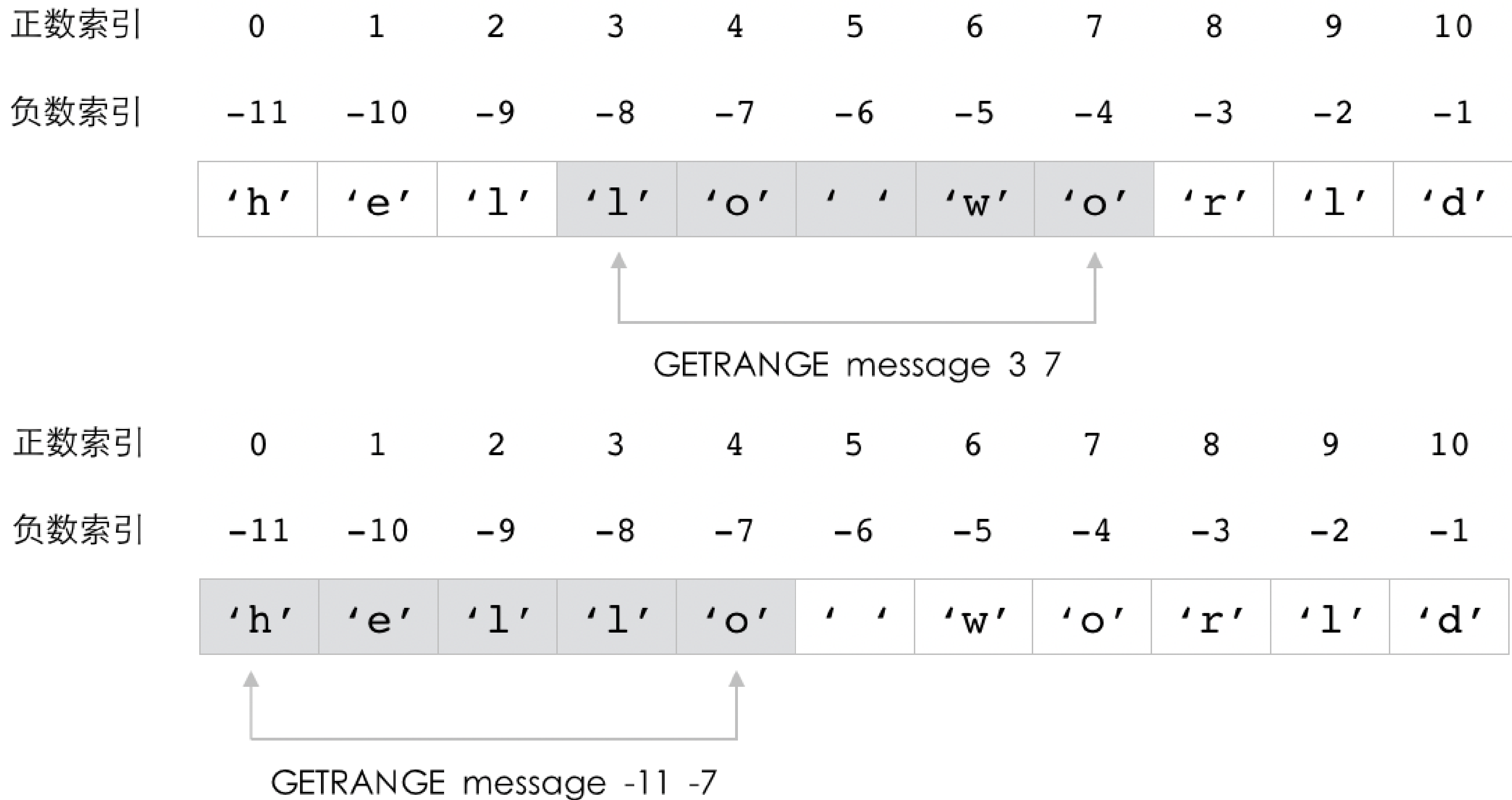
图 2-6 GETRANGE 命令执行示例

正数索引	0	1	2	3	4	5	6	7	8	9	10
负数索引	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

GETRANGE message 0 4

正数索引	0	1	2	3	4	5	6	7	8	9	10
负数索引	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

GETRANGE message 6 10



2.12.1 其他信息

属性	值
复杂度	O(N)，其中 N 为被返回内容的长度。
版本要求	GETRANGE 命令从 Redis 2.4.0 开始可用。

2.13 SETRANGE：对字符串值的指定索引范围进行设置

通过使用 SETRANGE 命令， 用户可以将字符串键的值从索引 index 开始的部分替换为指定的新内容， 被替换内容的长度取决于新内容的长度：

```
SETRANGE key index substitute
```

SETRANGE 命令在执行完设置操作之后， 会返回字符串值当前的长度作为结果。

比如说， 我们可以通过执行以下命令， 将 message 键的值从原来的 "hello world" 修改为 "hello Redis" ：

```
redis> GET message
"hello world"

redis> SETRANGE message 6 "Redis"
(integer) 11    -- 字符串值当前的长度为 11 字节

redis> GET message
"hello Redis"
```

这个例子中的 SETRANGE 命令会将 message 键的值从索引 6 开始的内容替换为 "Redis" ， 图 2-7 展示了这个命令的执行过程。

图 2-7 SETRANGE 命令修改 message 键的过程

正数索引	0	1	2	3	4	5	6	7	8	9	10
	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'

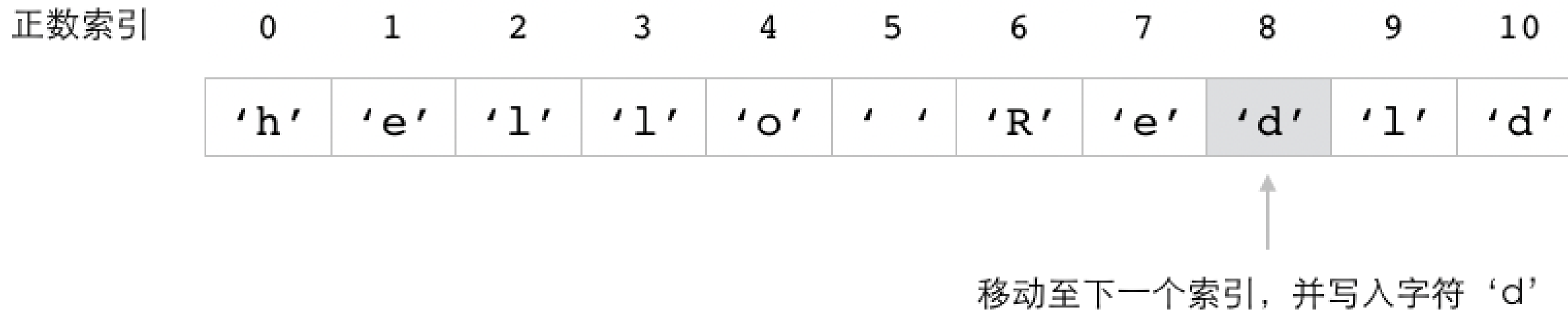
定位至索引 6

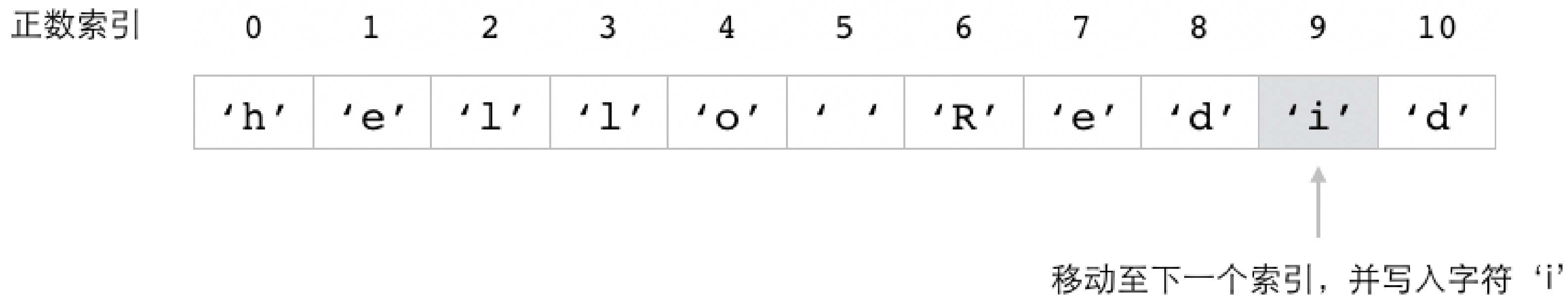
正数索引	0	1	2	3	4	5	6	7	8	9	10
	'h'	'e'	'l'	'l'	'o'	' '	'R'	'o'	'r'	'l'	'd'

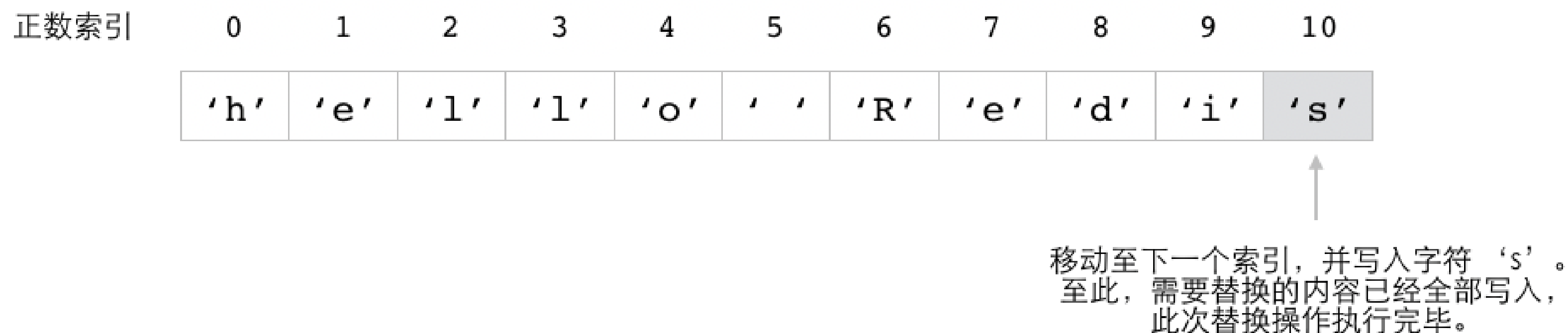
写入第一个字符‘R’

正数索引	0	1	2	3	4	5	6	7	8	9	10
	'h'	'e'	'l'	'l'	'o'	' '	'R'	'e'	'r'	'l'	'd'

移动至下一个索引，并写入字符‘e’







2.13.1 自动扩展被修改的字符串

当用户给定的新内容比被替换的内容更长时， SETRANGE 命令就会自动扩展被修改的字符串值， 从而确保新内容可以顺利写入。

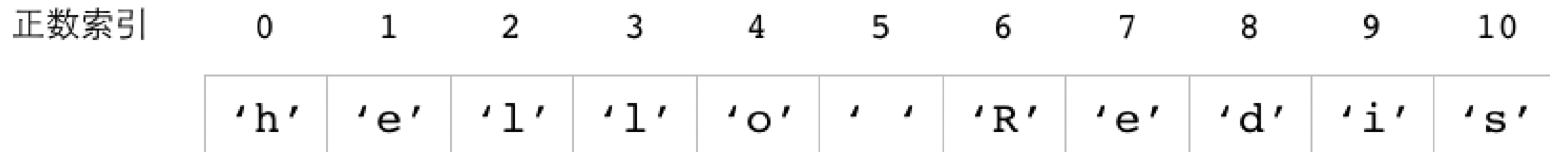
比如说， 以下代码就展示了如何通过 SETRANGE 命令， 将 message 键的值从原来的 11 字节长修改为 41 字节长：

```
redis> GET message
"hello Redis"

redis> SETRANGE message 5 ", this is a message send from peter."
(integer) 41

redis> GET message
"hello, this is a message send from peter."
```

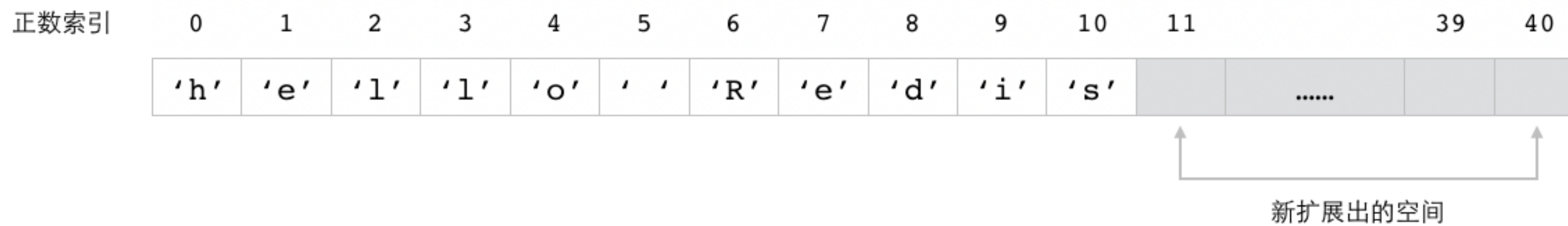
图 2-8 展示了这个 SETRANGE 命令扩展字符串并进行写入的过程。

图 2-8 SETRANGE 命令的执行过程示例

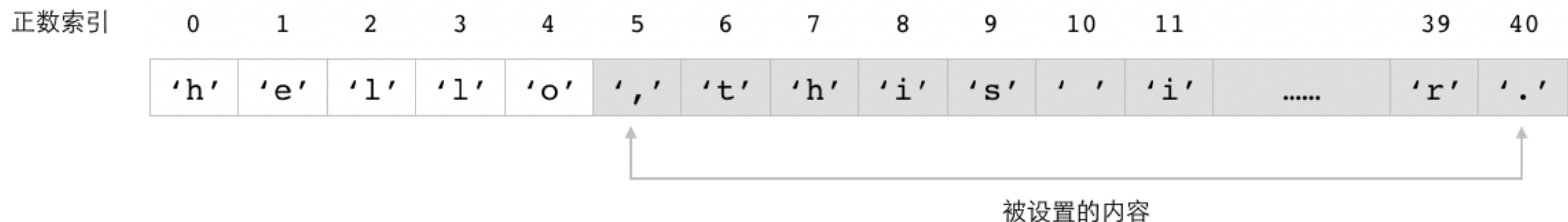
SETRANGE 命令执行之前的字符串值



将字符串值扩展至 41 字节长



对字符串值进行设置



2.13.2 在值里面填充空字节

SETRANGE 命令除了会根据用户给定的新内容自动扩展字符串值之外， 还会根据用户给定的 index 索引扩展字符串： 当用户给定的 index 索引超出字符串值的长度时， 字符串值末尾直到索引 index-1 之间的部分将使用空字节进行填充， 换句话说， 这些字节的所有二进制位都会被设置为 0 。

举个例子， 对于字符串键 greeting 来说：

```
redis> GET greeting
"hello"
```

当我们执行以下命令时：

```
redis> SETRANGE greeting 10 "world"
(integer) 15
```

SETRANGE 命令会先将字符串值扩展为 15 个字节长， 然后将 "hello" 末尾直到索引 9 之间的所有字节都填充为空字节， 最后再将索引 10 到索引 14 的内容设置为 "world" 。 图 2-9 展示了这个扩展、填充、最后设置的过程。

图 2-9 SETRANGE greeting 10 "world" 的执行过程

执行 SETRANGE 之前的字符串值

正数索引	0	1	2	3	4
	'h'	'e'	'l'	'l'	'o'

将字符串值从 5 个字节扩展至 15 个字节

正数索引	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	'h'	'e'	'l'	'l'	'o'										

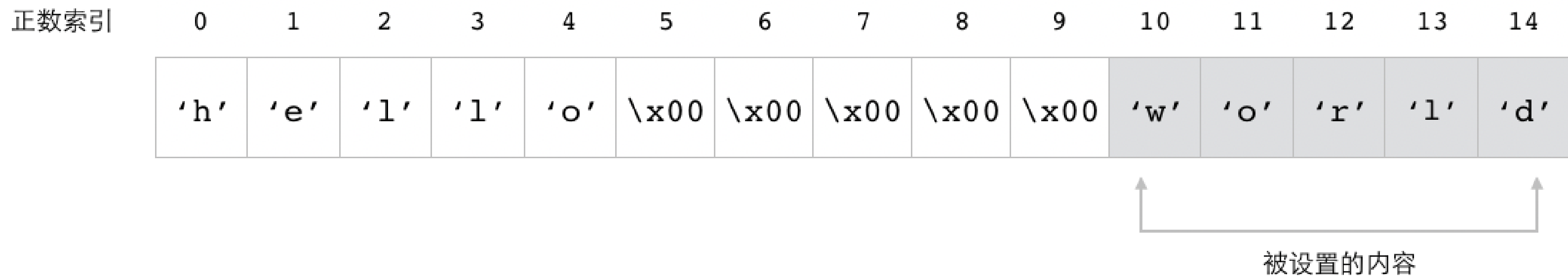
新扩展出的空间

使用空字节填充索引 5 至索引 9

正数索引	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	'h'	'e'	'l'	'l'	'o'	\x00	\x00	\x00	\x00	\x00					

被空字节填充的空间

将索引 10 至索引 14 的内容设置为“world”



通过执行 GET 命令， 我们可以取得 greeting 键在执行 SETRANGE 命令之后的值：

```
redis> GET greeting
"hello\x00\x00\x00\x00\x00world"
```

可以看到， greeting 键的值现在包含了多个 \x00 符号， 而每个 \x00 符号就代表一个空字节。

2.13.3 其他信息

属性	值
复杂度	O(N)，其中 N 为被修改内容的长度。
版本要求	SETRANGE 命令从 Redis 2.2.0 开始可用。

2.14 示例：给文章储存程序加上文章长度计数功能和文章预览功能

在前面的内容中， 我们使用 MSET 、 MGET 等命令构建了一个储存文章信息的程序， 在学习了 STRLEN 命令和 GETRANGE 命令之后， 我们可以给这个文章储存程序加上两个新功能， 其中一个是文章长度计数功能， 而另一个则是文章预览功能：

- 文章长度计数功能用于显示文章内容的长度， 读者可以通过这个长度值来了解一篇文章大概有多长， 从而决定是否阅读一篇文章。
- 文章预览功能则用于显示文章开头的一部分内容， 这些内容可以帮助读者快速地了解文章本身， 并吸引读者进一步阅读整篇文章。

代码清单 2-4 展示了这两个功能的具体实现代码， 其中文章长度计数功能是通过对文章内容执行 STRLEN 命令来实现的， 而文章预览功能则是通过对文章内容执行 GETRANGE 命令来实现的。

代码清单 2-4 带有长度计数功能和预览功能的文章储存程序：/string/article.py

```
from time import time  # time() 函数用于获取当前 Unix 时间戳

class Article:

    # 省略之前展示过的 __init__()、create() 、update() 等方法……

    def get_content_len(self):
        """
        返回文章内容的字节长度。
        """
        return self.client.strlen(self.content_key)

    def get_content_preview(self, preview_len):
        """
        返回指定长度的文章预览内容。
        """
        start_index = 0
        end_index = preview_len-1
        return self.client.getrange(self.content_key, start_index, end_index)
```

---

get_content_len() 方法的实现非常简单直接， 没有什么需要说明的。 与此相比， get_content_preview() 方法显得更复杂一些， 让我们来对它进行一些分析。

首先， get_content_preview() 方法会接受一个 preview_len 参数， 用于记录调用者指定的预览长度。 接着程序会根据这个预览长度， 计算出预览内容的起始索引和结束索引：

```
start_index = 0
end_index = preview_len-1
```

因为预览功能要做的就是返回文章内容的前 preview_len 个字节， 所以上面的这两条赋值语句要做的就是计算并记录文章前 preview_len 个字节所在的索引范围， 其中 start_index 的值总是 0 ， 而 end_index 的值则为 preview_len 减一。 举个例子， 假如用户输入的预览长度为 150 ， 那么 start_index 将被赋值为 0 ， 而 end_index 将被赋值为 149 。

最后， 程序会调用 GETRANGE 命令， 根据上面计算出的两个索引， 从储存着文章内容的字符串键里面取出指定的预览内容：

```
self.client.getrange(self.content_key, start_index, end_index)
```

以下代码展示了如何使用文章长度计数功能以及文章预览功能：

```
>>> from redis import Redis
>>> from article import Article
>>> client = Redis(decode_responses=True)
>>> article = Article(client, 12345)
>>> title = "Improving map data on GitHub"
>>> content = "You've been able to view and diff geospatial data on GitHub for a while, but now, in ad
>>> author = "benbalter"
>>> article.create(title, content, author)  # 将一篇比较长的文章储存起来
True
>>> article.get_content_len()              # 文章总长 273 字节
273
>>> article.get_content_preview(100)       # 获取文章前 100 字节的内容
"You've been able to view and diff geospatial data on GitHub for a while, but now, in addition to bei"
```

2.15 APPEND：追加新内容到值的末尾

通过调用 APPEND 命令， 用户可以将给定的内容追加到字符串键已有值的末尾：

```
APPEND key suffix
```

APPEND 命令在执行追加操作之后， 会返回字符串值当前的长度作为返回值。

举个例子， 对于以下这个名为 description 的键来说：

```
redis> GET description
"Redis"
```

我们可以通过执行以下命令， 将字符串 " is a database" 追加到 description 键已有值的末尾：

```
redis> APPEND description " is a database"
(integer) 19    -- 追加操作执行完毕之后，值的长度
```

以下是 description 键在执行完追加操作之后的值：

```
redis> GET description
"Redis is a database"
```

在此之后， 我们可以继续执行以下 APPEND 命令， 将字符串 " with many different data structure." 追加到 description 键已有值的末尾：

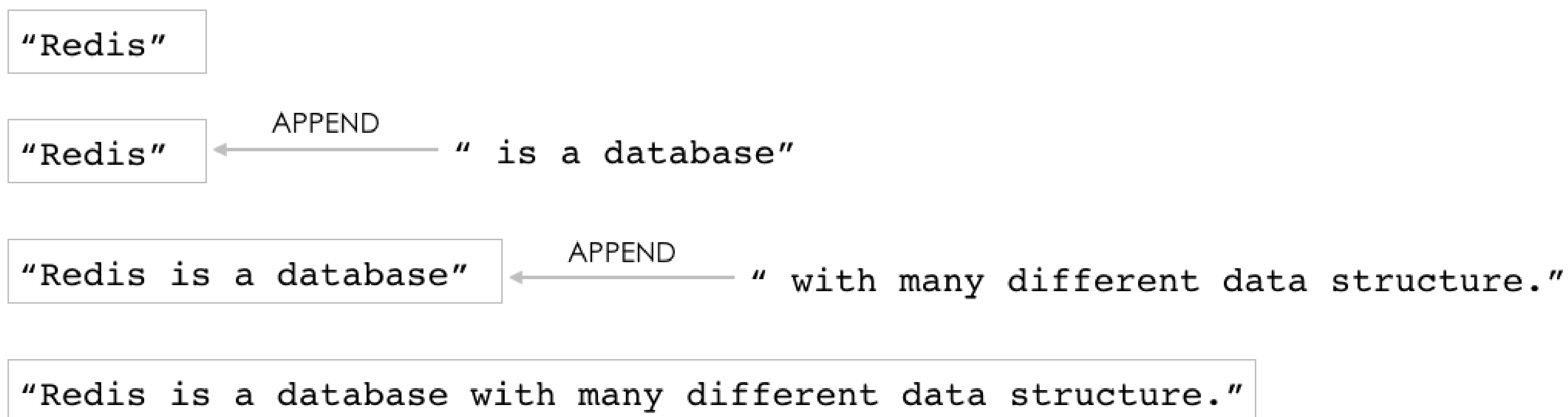
```
redis> APPEND description " with many different data structure."
(integer) 55
```

现在， description 键的值又变成了以下这个样子：

```
redis> GET description
"Redis is a database with many different data structure."
```

图 2-10 展示了 description 键的值是如何随着 APPEND 命令的执行而变化的。

图 2-10 description 键的值随着 APPEND 命令的执行而变化



2.15.1 处理不存在的键

如果用户给定的键并不存在， 那么 APPEND 命令会先将键的值初始化为空字符串 "" ， 然后再执行追加操作， 最终效果跟使用 SET 命令为键设置值的情况类似：

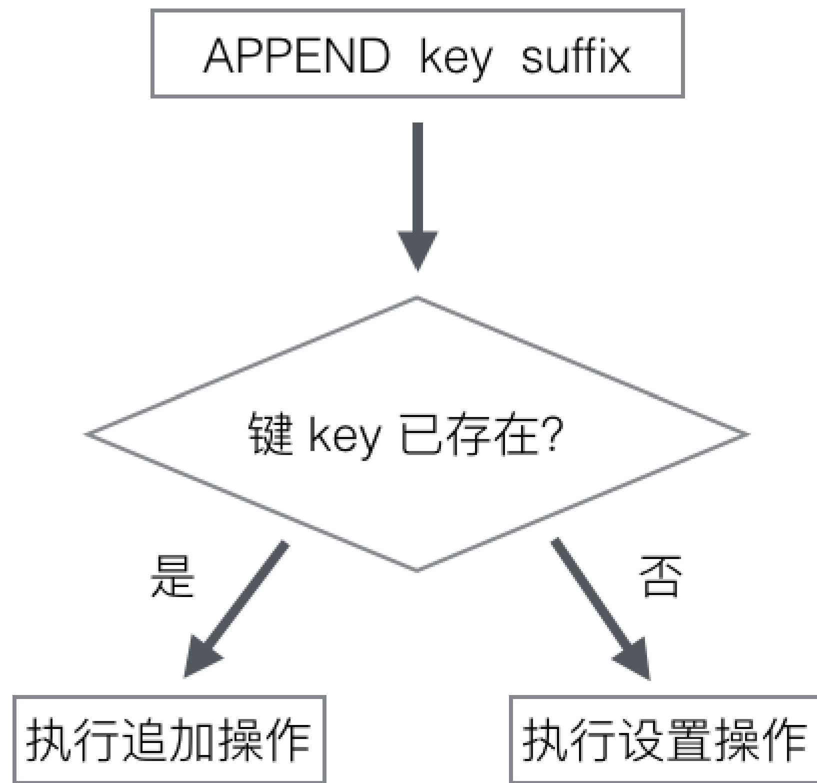
```
redis> GET append_msg   -- 键不存在
(nil)

redis> APPEND append_msg "hello"  -- 效果相当于执行 SET append_msg "hello"
(integer) 5

redis> GET append_msg
"hello"
```

当键有了值之后， APPEND 又会像平时一样， 将用户给定的值追加到已有值的末尾：

```
redis> APPEND append_msg ", how are you?"
(integer) 19

redis> GET append_msg
"hello, how are you?"
```

图 2-11 展示了 APPEND 命令是如何根据键是否存在来判断应该执行哪种操作的。

图 2-11 APPEND 的判断过程



2.15.2 其他信息

属性	值
复杂度	O(N)，其中 N 为新追加内容的长度。
版本要求	APPEND 命令从 Redis 2.0.0 开始可用。

2.16 示例：储存日志

很多程序在运行的时候都会生成一些日志， 这些日志记录了程序的运行状态以及执行过的重要操作。

比如说， 以下展示的就是 Redis 服务器运行时输出的一些日志， 这些日志记录了 Redis 开始运行的时间， 载入数据库所耗费的时长， 接收客户端连接所使用的端口号， 以及进行数据持久化操作的时间点等信息：

```
6066:M 06 Jul 17:40:49.611 # Server started, Redis version 3.1.999
6066:M 06 Jul 17:40:49.627 * DB loaded from disk: 0.016 seconds
6066:M 06 Jul 17:40:49.627 * The server is now ready to accept connections on port 6379
6066:M 06 Jul 18:29:20.009 * DB saved on disk
```

为了记录程序运行的状态， 又或者为了对日志进行分析， 我们有时候会需要把程序生成的日志储存起来。

比如说， 我们可以通过使用 `SET` 命令， 将日志的生成时间用作键、日志的内容用作值， 把上面展示的日志储存到多个字符串键里面：

```
redis> SET "06 Jul 17:40:49.611" "# Server started, Redis version 3.1.999"
OK

redis> SET "06 Jul 17:40:49.627" "* DB loaded from disk: 0.016 seconds"
OK

redis> SET "06 Jul 17:40:49.627" "* The server is now ready to accept connections on port 6379"
OK

redis> SET "06 Jul 18:29:20.009" "* DB saved on disk"
OK
```

遗憾的是， 这种日志储存方式并不理想， 它的主要问题有两个：

1. 这种方法需要在数据库里面创建非常多的键。 因为 Redis 每创建一个键就需要消耗一定的额外资源（overhead）来对键进行维护， 所以键的数量越多， 消耗的额外资源就会越多。
2. 这种方法将全部日志分散地储存在不同的键里面， 当程序想要对特定的日志进行分析的时候， 它就需要花费额外的时间和资源去查找指定的日志， 这给分析操作带来了额外的麻烦和资源消耗。

代码清单 2-5 展示了另一种更为方便和高效的日志储存方式， 这个程序会把同一天之内产生的所有日志都储存在同一个字符串键里面， 从而使得用户可以非常高效地取得指定日期内产生的所有日志。

代码清单 2-5 使用字符串键实现高效的日志储存程序： `/string/log.py`

```
LOG_SEPARATOR = "\n"
```

```python
class Log:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def add(self, new_log):
        """
        将给定的日志储存起来。
        """
        new_log += LOG_SEPARATOR
        self.client.append(self.key, new_log)

    def get_all(self):
        """
        以列表形式返回所有日志。
        """
        all_logs = self.client.get(self.key)
        if all_logs is not None:
            log_list = all_logs.split(LOG_SEPARATOR)
            log_list.remove("")
            return log_list
        else:
            return []
```

日志储存程序的 `add()` 方法负责将新日志储存起来。 这个方法首先会将分隔符追加到新日志的末尾：

```python
new_log += LOG_SEPARATOR
```

然后调用 APPEND 命令， 将新日志追加到已有日志的末尾：

```python
self.client.append(self.key, new_log)
```

举个例子， 如果用户输入的日志是：

```python
"this is log1"
```

那么 `add()` 方法首先会把分隔符 `"\n"` 追加到这行日志的末尾， 使之变成：

```python
"this is log1\n"
```

然后调用以下命令， 将新日志追到已有日志的末尾：

```
APPEND key "this is log1\n"
```

负责获取所有日志的 get_all() 方法比较复杂， 因为它不仅需要从字符串键里面取出包含了所有日志的字符串值， 还需要从这个字符串值里面分割出每一条日志。 首先， 这个方法使用 GET 命令从字符串键里面取出包含了所有日志的字符串值：

```
all_logs = self.client.get(self.key)
```

接着， 程序会检查 all_logs 这个值是否为空， 如果为空则表示没有日志被储存， 程序直接返回空列表 [] 作为 get_all() 方法的执行结果； 另一方面， 如果值不为空， 那么程序将调用 Python 的 split() 方法对字符串值进行分割， 并将分割结果储存到 log_list 列表里面：

```
log_list = all_logs.split(LOG_SEPARATOR)
```

因为 split() 方法会在结果中包含一个空字符串， 而我们并不需要这个空字符串， 所以程序还会调用 remove() 方法， 将空字符串从分割结果中移除， 使得 log_list 列表里面只保留被分割的日志：

```
log_list.remove("")
```

在此之后， 程序只需要将包含了多条日志的 log_list 列表返回给调用者就可以了：

```
return log_list
```

举个例子， 假设我们使用 add() 方法， 在一个字符串键里面储存了 "this is log1" 、 this is log2" 、 this is log3" 这三条日志， 那么 get_all() 方法在使用 GET 命令获取字符串键的值时， 将得到以下结果：

```
"this is log1\nthis is log2\nthis is log3"
```

在使用 split(LOG_SEPARATOR) 方法对这个结果进行分割之后， 程序将得到一个包含四个元素的列表， 其中列表最后的元素为空字符串：

```
["this is log1", "this is log2", "this is log3", ""]
```

在调用 `remove("")` 方法移除列表中的空字符串之后， 列表里面就只会包含被储存的日志：

```
["this is log1", "this is log2", "this is log3"]
```

这时 `get_all()` 方法只需要把这个列表返回给调用者就可以了。

以下代码展示了这个日志储存程序的使用方法：

```
>>> from redis import Redis
>>> from log import Log
>>> client = Redis(decode_responses=True)
>>> # 按日期归类日志
>>> log = Log(client, "06 Jul")
>>> # 储存日志
>>> log.add("17:40:49.611 # Server started, Redis version 3.1.999")
>>> log.add("17:40:49.627 * DB loaded from disk: 0.016 seconds")
>>> log.add("17:40:49.627 * The server is now ready to accept connections on port 6379")
>>> log.add("18:29:20.009 * DB saved on disk")
>>> # 以列表形式返回所有日志
>>> log.get_all()
['17:40:49.611 # Server started, Redis version 3.1.999', '17:40:49.627 * DB loaded from disk: 0.016 se
>>> # 单独打印每条日志
>>> for i in log.get_all():
...   print(i)
...
17:40:49.611 # Server started, Redis version 3.1.999
17:40:49.627 * DB loaded from disk: 0.016 seconds
17:40:49.627 * The server is now ready to accept connections on port 6379
18:29:20.009 * DB saved on disk
```

2.17 使用字符串键储存数字值

每当用户将一个值储存到字符串键里面的时候， Redis 都会对这个值进行检测， 如果这个值能够被解释为以下两种类型的其中一种， 那么 Redis 就会把这个值当做数字来处理：

- 一种是能够使用 C 语言的 `long long int` 类型储存的整数， 在大多数系统中， 这种类型储存的都是 64 位长度的有符号整数， 取值范围介于 `-9223372036854775808` 和 `9223372036854775807` 之间；
- 第二种是能够使用 C 语言的 `long double` 类型储存的浮点数， 在大多数系统中， 这种类型储存的都是 128 位长度的有符号浮点数， 取值范围介于 `3.36210314311209350626e-4932` 和 `1.18973149535723176502e+4932L` 之

间。

作为例子， 表 1-2 展示了一些不同类型的值， 并说明了 Redis 对它们的解释方式。

表 1-2 一些能够被 Redis 解释为数字的例子

值	Redis 解释这个值的方式
`10086`	解释为整数。
`+894`	解释为整数。
`-123`	解释为整数。
`3.14`	解释为浮点数。
`+2.56`	解释为浮点数。
`-5.12`	解释为浮点数。
`12345678901234567890`	这个值虽然是整数， 但是因为它的大小超出了 `long long int` 类型能够容纳的范围， 所以只能被解释为字符串。
`3.14e5`	因为 Redis 不能解释使用科学记数法表示的浮点数， 所以这个值只能被解释为字符串。
`"one"`	解释为字符串。
`"123abc"`	解释为字符串。

为了能够更方便地处理那些使用字符串键储存的数字值， Redis 提供了一系列加法操作命令以及减法操作命令， 用户可以通过这些命令直接对字符串键储存的数字值执行加法操作或减法操作， 本章接下来将对这些命令进行介绍。

2.18 INCRBY、DECRBY：对整数值执行加法操作和减法操作

当字符串键储存的值能够被 Redis 解释为整数时， 用户就可以通过 `INCRBY` 命令和 `DECRBY` 命令， 对被储存的整数值执行加法操作或是减法操作。

`INCRBY` 命令用于为整数值加上指定的整数增量， 并返回键在执行加法操作之后的值：

```
INCRBY key increment
```

以下代码展示了如何使用 INCRBY 命令去增加一个字符串键的值：

```
redis> SET number 100
OK

redis> GET number
"100"

redis> INCRBY number 300      -- 将键的值加上 300
(integer) 400

redis> INCRBY number 256      -- 将键的值加上 256
(integer) 656

redis> INCRBY number 1000     -- 将键的值加上 1000
(integer) 1656

redis> GET number
"1656"
```

与 INCRBY 命令的作用正好相反， DECRBY 命令用于为整数值减去指定的整数减量， 并返回键在执行减法操作之后的值：

```
DECRBY key increment
```

以下代码展示了如何使用 DECRBY 命令去减少一个字符串键的值：

```
redis> SET number 10086
OK

redis> GET number
"10086"

redis> DECRBY number 300      -- 将键的值减去 300
(integer) 9786

redis> DECRBY number 786      -- 将键的值减去 786
(integer) 9000

redis> DECRBY number 5500     -- 将键的值减去 5500
(integer) 3500
```

```
redis> GET number
"3500"
```

2.18.1 类型限制

当字符串键的值不能被 Redis 解释为整数时， 对键执行 `INCRBY` 命令或是 `DECRBY` 命令将返回一个错误：

```
redis> SET pi 3.14
OK

redis> INCRBY pi 100    -- 不能对浮点数值执行
(error) ERR value is not an integer or out of range

redis> SET message "hello world"
OK

redis> INCRBY message    -- 不能对字符串值执行
(error) ERR wrong number of arguments for 'incrby' command

redis> SET big-number 123456789123456789123456789
OK

redis> INCRBY big-number 100    -- 不能对超过 64 位长度的整数执行
(error) ERR value is not an integer or out of range
```

另外需要注意的一点是， `INCRBY` 和 `DECRBY` 的增量和减量也必须能够被 Redis 解释为整数， 使用其他类型的值作为增量或减量将返回一个错误：

```
redis> INCRBY number 3.14    -- 不能使用浮点数作为增量
(error) ERR value is not an integer or out of range

redis> INCRBY number "hello world"    -- 不能使用字符串值作为增量
(error) ERR value is not an integer or out of range
```

2.18.2 处理不存在的键

当 `INCRBY` 命令或 `DECRBY` 命令遇到不存在的键时， 命令会先将键的值初始化为 `0` ， 然后再执行相应的加法操作或减法操作。

以下代码展示了 INCRBY 命令是如何处理不存在的键 x 的：

```
redis> GET x     -- 键 x 不存在
(nil)

redis> INCRBY x 123    -- 先将键 x 的值初始化为 0 ，然后再执行加上 123 的操作
(integer) 123

redis> GET x
"123"
```

而以下代码则展示了 DECRBY 命令是如何处理不存在的键 y 的：

```
redis> GET y     -- 键 y 不存在
(nil)

redis> DECRBY y 256    -- 先将键 y 的值初始化为 0 ，然后再执行减去 256 的操作
(integer) -256

redis> GET y
"-256"
```

2.18.3 其他信息

属性	值
复杂度	O(1)
版本要求	INCRBY 命令和 DECRBY 命令从 Redis 1.0.0 开始可用。

2.19 INCR、DECR：对整数值执行加一操作和减一操作

因为对整数值执行加一操作或是减一操作的场景经常会出现， 所以为了能够更方便地执行这两个操作， Redis 分别提供了用于执行加一操作的 INCR 命令以及用于执行减一操作的 DECR 命令。

INCR 命令的作用就是将字符串键储存的整数值加上一， 它的效果相当于执行 INCRBY key 1 ：

```
INCR key
```

而 DECR 命令的作用就是将字符串键储存的整数值减去一， 它的效果相当于执行 DECRBY key 1 ：

```
DECR key
```

以下代码展示了 INCR 命令和 DECR 命令的作用：

```
redis> SET counter 100
OK

redis> INCR counter      -- 对整数值执行加一操作
(integer) 101

redis> INCR counter
(integer) 102

redis> INCR counter
(integer) 103

redis> DECR counter      -- 对整数值执行减一操作
(integer) 102

redis> DECR counter
(integer) 101

redis> DECR counter
(integer) 100
```

除了增量和减量被固定为一之外， INCR 命令和 DECR 命令的其他方面与 INCRBY 命令以及 DECRBY 命令完全相同。

2.19.1 其他信息

属性	值
复杂度	O(1)
版本要求	INCR 命令和 DECR 命令从 Redis 1.0.0 开始可用。

2.20 INCRBYFLOAT：对数字值执行浮点数加法操作

除了用于执行整数加法操作的 INCR 命令以及 INCRBY 命令之外， Redis 还提供了用于执行浮点数加法操作的 INCRBYFLOAT 命令：

```
INCRBYFLOAT key increment
```

INCRBYFLOAT 命令可以把一个浮点数增量加到字符串键储存的数字值上面， 并返回键在执行加法操作之后的数字值作为命令的返回值。

以下代码展示了如何使用 INCRBYFLOAT 命令去增加一个浮点数的值：

```
redis> SET decimal 3.14    -- 一个储存着浮点数值的键
OK

redis> GET decimal
"3.14"

redis> INCRBYFLOAT decimal 2.55    -- 将键 decimal 的值加上 2.55
"5.69"

redis> GET decimal
"5.69"
```

2.20.1 处理不存在的键

INCRBYFLOAT 命令在遇到不存在的键时， 会先将键的值初始化为 0 ， 然后再执行相应的加法操作。

在以下展示的代码里面， INCRBYFLOAT 命令就是先把 x-point 键的值初始化为 0 ， 然后再执行加法操作的：

```
redis> GET x-point    -- 不存在的键
(nil)

redis> INCRBYFLOAT x-point 12.7829
"12.7829"

redis> GET x-point
"12.7829"
```

2.20.2 使用 INCRBYFLOAT 执行浮点数减法操作

Redis 为 INCR 命令提供了相应的减法版本 DECR 命令， 也为 INCRBY 命令提供了相应的减法版本 DECRBY 命令， 但是并没有为 INCRBYFLOAT 命令提供相应的减法版本， 因此用户只能通过给 INCRBYFLOAT 命令传入负数增量来执行浮点数减法操作。

以下代码展示了如何使用 INCRBYFLOAT 命令执行浮点数减法计算：

```
redis> SET pi 3.14
OK

redis> GET pi
"3.14"

redis> INCRBYFLOAT pi -1.1    -- 值减去 1.1
"2.04"

redis> INCRBYFLOAT pi -0.7    -- 值减去 0.7
"1.34"

redis> INCRBYFLOAT pi -1.3    -- 值减去 1.3
"0.04"
```

2.20.3 INCRBYFLOAT 与整数值

INCRBYFLOAT 命令对于类型限制的要求比 INCRBY 命令和 INCR 命令要宽松得多， INCRBYFLOAT 命令不仅可以用于处理浮点数值， 它还可以用于处理整数值：

1. INCRBYFLOAT 命令既可用于浮点数值， 也可以用于整数值。
2. INCRBYFLOAT 命令的增量既可以是浮点数， 又可以是整数。
3. 当 INCRBYFLOAT 命令的执行结果可以表示为整数时， 命令的执行结果将以整数形式储存。

以下代码展示了如何使用 INCRBYFLOAT 去处理一个储存着整数值的键：

```
redis> SET pi 1      -- 创建一个整数值
OK

redis> GET pi
"1"

redis> INCRBYFLOAT pi 2.14
"3.14"
```

以下代码展示了如何使用整数值作为 INCRBYFLOAT 命令的增量：

```
redis> SET pi 3.14
OK

redis> GET pi
"3.14"

redis> INCRBYFLOAT pi 20    -- 增量为整数值
"23.14"
```

而以下代码则展示了 INCRBYFLOAT 命令是如何把计算结果储存为整数的：

```
redis> SET pi 3.14
OK

redis> GET pi
"3.14"

redis> INCRBYFLOAT pi 0.86    -- 计算结果被储存为整数
"4"
```

2.20.4 小数位长度限制

虽然 Redis 并不限制字符串键储存的浮点数的小数位长度， 但是在使用 INCRBYFLOAT 命令处理浮点数的时候， 命令最多只会保留计算结果小数点后的 17 位数字， 超过这个范围的小数将被截断：

```
redis> GET i
"0.01234567890123456789"    -- 这个数字的小数部分有 20 位长

redis> INCRBYFLOAT i 0
"0.01234567890123457"    -- 执行加法操作之后，小数部分只保留了 17 位
```

2.20.5 其他信息

属性	值
复杂度	O(1)
版本要求	INCRBYFLOAT 命令从 Redis 2.6.0 开始可用。

2.21 示例：ID 生成器

在构建应用程序的时候， 我们经常会用到各式各样的 ID （identifier，标识符）。 比如说， 储存用户信息的程序在每次出现一个新用户的时候就需要创建一个新的用户 ID ， 而博客程序在作者每次发表一篇新文章的时候也需要创建一个新的文章 ID ， 诸如此类。

ID 通常会以数字形式出现， 并且通过递增的方式来创建出新的 ID 。 比如说， 如果当前最新的 ID 值为 `10086` ，那么下一个 ID 就应该是 `10087` ， 而再下一个 ID 则是 `10088` ， 以此类推。

代码清单 2-6 展示了一个使用字符串键实现的 ID 生成器， 这个生成器通过执行 `INCR` 命令来产生新的 ID ， 并且它还可以通过执行 `SET` 命令来保留指定数字之前的 ID ， 从而避免用户为了得到某个指定的 ID 而生成大量无效 ID 。

代码清单 2-6 使用字符串键实现的 ID 生成器： `/string/id_generator.py`

```python
class IdGenerator:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def produce(self):
        """
        生成并返回下一个 ID 。
        """
        return self.client.incr(self.key)

    def reserve(self, n):
        """
        保留前 n 个 ID ， 使得之后执行的 produce() 方法产生的 ID 都大于 n 。
        为了避免 produce() 方法产生重复 ID ，
        这个方法只能在 produce() 方法和 reserve() 方法都没有执行过的情况下使用。
        这个方法在 ID 被成功保留时返回 True ，
        在 produce() 方法或 reserve() 方法已经执行过而导致保留失败时返回 False 。
        """
        result = self.client.set(self.key, n, nx=True)
        return result is True
```

在这个 ID 生成器程序中， `produce()` 方法要做的就是调用 `INCR` 命令， 对字符串键储存的整数值执行加一操作，并将执行加法操作之后得到的新值用作 ID 。

另一方面， 用于保留指定 ID 的 `reserve()` 方法是通过执行 `SET` 命令为键设置值来实现的： 当用户把一个字符串键的值设置为 `N` 之后， 对这个键执行 `INCR` 命令总是会返回比 `N` 更大的值， 因此在效果上就相当于把所有小于等于 `N` 的 ID 都保留下来了。

需要注意的是， 这种保留 ID 的方法只能在字符串键还没有值的情况下使用， 如果用户已经使用过 `produce()` 方法来生成 ID ， 又或者已经执行过 `reserve()` 方法来保留 ID ， 那么再使用 `SET` 命令去设置 ID 值可能就会导致 `produce()` 方法产生出一些已经用过的 ID ， 并因此引发 ID 冲突。

为此， `reserve()` 方法在设置字符串键时使用了带有 `NX` 选项的 `SET` 命令， 从而确保了对键的设置操作只会在键不存在的情况下执行：

```
self.client.set(self.key, n, nx=True)
```

以下代码展示了这个 ID 生成器的使用方法：

```
>>> from redis import Redis
>>> from id_generator import IdGenerator
>>> client = Redis(decode_responses=True)
>>> id_generator = IdGenerator(client, "user::id")
>>> id_generator.reserve(1000000)  # 保留前一百万个 ID
True
>>> id_generator.produce()         # 生成 ID ，这些 ID 的值都大于一百万
1000001
>>> id_generator.produce()
1000002
>>> id_generator.produce()
1000003
>>> id_generator.reserve(1000)     # 键已经有值，无法再次执行 reserve() 方法
False
```

2.22 示例：计数器

除了 ID 生成器之外， 计数器也是构建应用程序时必不可少的组件之一： 网站的访客数量、用户执行某个操作的次数、某首歌或者某个视频的播放量、论坛帖子的回复数量等等， 记录这些信息都需要用到计数器。 实际上， 计

数器在互联网中几乎无处不在， 因此如何简单高效地实现计数器一直都是构建应用程序时经常会遇到的一个问题。

代码清单 2-7 展示了一个计数器实现， 这个程序把计数器的值储存在一个字符串键里面， 并通过 INCRBY 命令和 DECRBY 命令， 对计数器的值执行加法操作和减法操作； 在有需要的时候， 用户还可以通过调用 GETSET 方法来清零计数器并取得清零之前的旧值。

代码清单 2-7 使用字符串键实现的计数器： /string/counter.py

```python
class Counter:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def increase(self, n=1):
        """
        将计数器的值加上 n ， 然后返回计数器当前的值。
        如果用户没有显式地指定 n ， 那么将计数器的值加上一。
        """
        return self.client.incr(self.key, n)

    def decrease(self, n=1):
        """
        将计数器的值减去 n ， 然后返回计数器当前的值。
        如果用户没有显式地指定 n ， 那么将计数器的值减去一。
        """
        return self.client.decr(self.key, n)

    def get(self):
        """
        返回计数器当前的值。
        """
        # 尝试获取计数器当前的值
        value = self.client.get(self.key)
        # 如果计数器并不存在，那么返回 0 作为计数器的默认值
        if value is None:
            return 0
        else:
            # 因为 redis-py 的 get() 方法返回的是字符串值
            # 所以这里需要使用 int() 函数，将字符串格式的数字转换为真正的数字类型
            # 比如将 "10" 转换为 10
            return int(value)
```

```python
    def reset(self):
        """
        清零计数器，并返回计数器在被清零之前的值。
        """
        old_value = self.client.getset(self.key, 0)
        # 如果计数器之前并不存在，那么返回 0 作为它的旧值
        if old_value is None:
            return 0
        else:
            # 跟 redis-py 的 get() 方法一样， getset() 方法返回的也是字符串值
            # 所以程序在将计数器的旧值返回给调用者之前，需要先将它转换成真正的数字
            return int(old_value)
```

在这个程序中， `increase()` 方法和 `decrease()` 方法在定义时都使用了 Python 的参数默认值特性：

```python
def increase(self, n=1):
```

```python
def decrease(self, n=1):
```

以上定义表明， 如果用户直接以无参数的方式调用 `increase()` 或者 `decrease()` ， 那么参数 `n` 的值将会被设置为 `1` 。

在设置了参数 `n` 之后， `increase()` 方法和 `decrease()` 方法会分别调用 `INCRBY` 命令和 `DECRBY` 命令， 根据参数 `n` 的值， 对给定的键执行加法或减法操作：

```python
# increase() 方法
return self.client.incr(self.key, n)
```

```python
# decrease() 方法
return self.client.decr(self.key, n)
```

注意， `increase()` 方法在内部调用的是 `incr()` 方法而不是 `incrby()` 方法， 并且 `decrease()` 方法在内部调用的也是 `decr()` 方法而不是 `decrby()` 方法， 这是因为在 redis-py 客户端中， `INCR` 命令和 `INCRBY` 命令都是由 `incr()` 方法负责执行的：

- 如果用户在调用 `incr()` 方法时没有给定增量， 那么 `incr()` 方法就默认用户指定的增量为 `1` ， 并执行 `INCR` 命令；

- 另一方面， 如果用户在调用 `incr()` 方法时给定了增量， 那么 `incr()` 方法就会执行 `INCRBY` 命令， 并根据给定的增量执行加法操作；

`decr()` 方法的情况也与此类似， 只是被调用的命令变成了 `DECR` 命令和 `DECRBY` 命令。

以下代码展示了这个计数器的使用方法：

```
>>> from redis import Redis
>>> from counter import Counter
>>> client = Redis(decode_responses=True)
>>> counter = Counter(client, "counter::page_view")
>>> counter.increase()      # 将计数器的值加上 1
1
>>> counter.increase()      # 将计数器的值加上 1
2
>>> counter.increase(10)    # 将计数器的值加上 10
12
>>> counter.decrease()      # 将计数器的值减去 1
11
>>> counter.decrease(5)     # 将计数器的值减去 5
6
>>> counter.reset()         # 重置计数器， 并返回旧值
6
>>> counter.get()           # 返回计数器当前的值
0
```

2.23 示例： 限速器

为了保障系统的安全性和性能， 并保证系统的重要资源不被滥用， 应用程序常常会对用户的某些行为进行限制， 比如说：

- 为了防止网站内容被网络爬虫抓取， 网站管理者通常会限制每个 IP 地址在固定时间段内能够访问的页面数量 —— 比如一分钟之内最多只能访问 30 个页面 —— 超过这一限制的用户将被要求进行身份验证， 确认本人并非网络爬虫， 又或者等到限制解除了之后再进行访问。
- 为了防止用户的账号遭到暴力破解， 网上银行通常会对访客的密码试错次数进行限制， 如果一个访客在尝试登录某个账号的过程中， 连续好几次输入了错误的密码， 那么这个账号将被冻结， 只能等到第二天再尝试登录， 有的银行还会向账号持有者的手机发送通知来汇报这一情况。

实现这些限制机制的其中一种方法是使用限速器， 它可以限制用户在指定时间段之内能够执行某项操作的次数。

代码清单 2-8 展示了一个使用字符串键实现的限速器， 这个限速器程序会把操作的最大可执行次数储存在一个字符串键里面， 然后在用户每次尝试执行被限制的操作之前， 使用 DECR 命令将操作的可执行次数减去一， 最后通过检查可执行次数的值来判断是否执行该操作。

代码清单 2-8 倒计时式的限速器： /string/limiter.py

```
class Limiter:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def set_max_execute_times(self, max_execute_times):
        """
        设置操作的最大可执行次数。
        """
        self.client.set(self.key, max_execute_times)

    def still_valid_to_execute(self):
        """
        检查是否可以继续执行被限制的操作。
        是的话返回 True ， 否则返回 False 。
        """
        num = self.client.decr(self.key)
        return (num >= 0)

    def remaining_execute_times(self):
        """
        返回操作的剩余可执行次数。
        """
        num = int(self.client.get(self.key))
        if num < 0:
            return 0
        else:
            return num
```

这个限速器的关键在于 set_max_execute_times() 方法和 still_valid_to_execute() 方法： 前者用于将最大可执行次数储存在一个字符串键里面， 而后者则会在每次被调用时对可执行次数执行减一操作， 并检查目前剩余的可执行次数是否已经变为负数： 如果为负数则表示可执行次数已经耗尽， 不为负数则表示操作可以继续执行。

以下代码展示了这个限制器的使用方法：

```
>>> from redis import Redis
>>> from limiter import Limiter
>>> client = Redis(decode_responses=True)
>>> limiter = Limiter(client, 'wrong_password_limiter')  # 密码错误限制器
>>> limiter.set_max_execute_times(3)  # 最多只能输入错三次密码
>>> limiter.still_valid_to_execute()  # 前三次操作能够顺利执行
True
>>> limiter.still_valid_to_execute()
True
>>> limiter.still_valid_to_execute()
True
>>> limiter.still_valid_to_execute()  # 从第四次开始，操作将被拒绝执行
False
>>> limiter.still_valid_to_execute()
False
```

而以下伪代码则展示了如何使用这个限速器去限制密码的错误次数：

```
# 试错次数未超过限制
while limiter.still_valid_to_execute():
    # 获取访客输入的账号和密码
    account, password = get_user_input_account_and_password()
    # 验证账号和密码是否匹配
    if password_match(account, password):
        ui_print("密码验证成功")
    else:
        ui_print("密码验证失败，请重新输入")
# 试错次数已超过限制
else:
    # 锁定账号
    lock_account(account)
    ui_print("连续尝试登录失败，账号已被锁定，请明天再来尝试登录。")
```

2.24 重点回顾

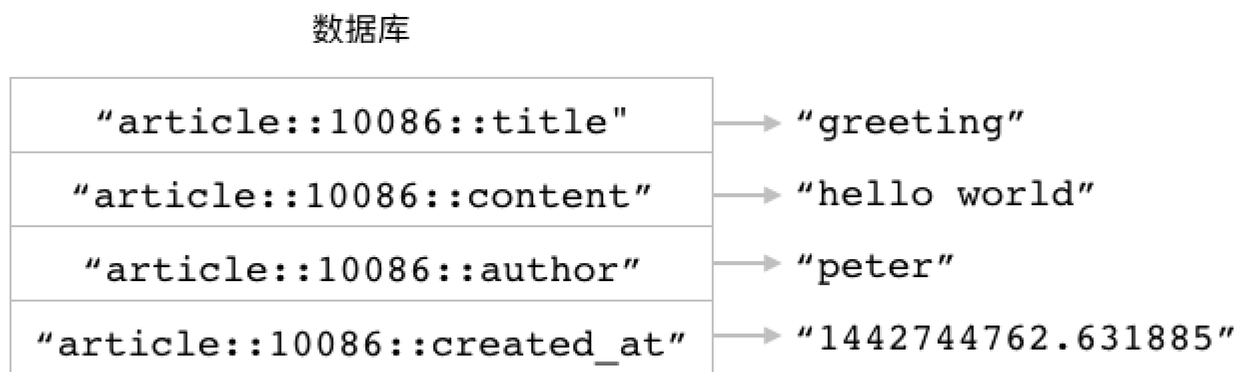
- Redis 的字符串键可以把单独的一个键和单独的一个值在数据库里面关联起来， 并且这个键和值既可以储存文字数据， 又可以储存二进制数据。
- SET 命令在默认情况下会直接覆盖字符串键已有的值， 如果我们只想在键不存在的情况下为它设置值， 那么可以使用带有 NX 选项的 SET 命令； 相反地， 如果我们只想在键已经存在的情况下为它设置新值， 那么可以使用带有 XX 选项的 SET 命令。
- 使用 MSET 、 MSETNX 以及 MGET 命令可以有效地减少程序的网络通信次数， 从而提升程序的执行效率。

- Redis 用户可以通过制定命名格式来提升 Redis 数据的可读性并避免键名冲突。
- 字符串值的正数索引以 `0` 为开始， 从字符串的开头向结尾不断递增； 字符串值的负数索引以 `-1` 为开始， 从字符串的结尾向开头不断递减。
- `GETRANGE key start end` 命令接受的是闭区间索引范围， 位于 `start` 索引和 `end` 索引上的值也会被包含在命令返回的内容当中。
- `SETRANGE` 命令在有需要时会自动对字符串值进行扩展， 并使用空字节填充新扩展空间中没有内容的部分。
- `APPEND` 命令在键不存在时执行设置操作， 在键存在时执行追加操作。
- Redis 会把能够被表示为 `long long int` 类型的整数以及能够被表示为 `long double` 类型的浮点数当做数字来处理。

3. 散列（Hash）

在前面的《字符串》一章中， 我们曾经看到过如何使用多个字符串键去储存相关联的一组数据。 比如在字符串键实现的文章储存程序中， 程序就会为每篇文章创建四个字符串键， 并把文章的标题、内容、作者和创建时间分别储存到这四个字符串键里面， 图 3-1 就展示了一个使用字符串键储存文章数据的例子。

图 3-1 使用多个字符串键储存文章



使用多个字符串键储存相关联数据虽然在技术上是可行的， 但是在实际应用中却并不是最有效的方法， 这种储存方法至少存在以下三个问题：

- 首先， 程序每储存一组相关联的数据， 就必须在数据库里面同时创建多个字符串键， 这样的数据越多， 数据库包含的键数量也会越多。 数量庞大的键会对数据库某些操作的执行速度产生影响， 并且维护这些键也会产生大量的资源消耗。
- 其次， 为了在数据库里面标识出相关联的字符串键， 程序需要为它们加上相同的前缀， 但键名实际上也是一种数据， 储存键名也需要耗费内存空间， 因此重复出现的键名前缀实际上导致很多内存空间被白白浪费了。 此外， 带前缀的键名还降低了键名的可读性， 让人无法一眼看清键的真正用途， 比如键名 `article::10086::author` 就远不如键名 `author` 简洁， 而键名 `article::10086::title` 也远不如键名 `title` 来得简洁。
- 最后， 虽然程序在逻辑上会把带有相同前缀的字符串键看作是相关联的一组数据， 但是在 Redis 看来， 它们只不过是储存在同一个数据库中的不同字符串键而已。 因此当程序需要处理一组相关联的数据时， 它就必须对所有有关的字符串键都执行相同的操作。 比如说， 如果程序想要删除 ID 为 `10086` 的文章， 那么它就必须把 `article::10086::title` 、 `article::10086::content` 等四个字符串键都删掉才行， 这给文章的删除操作带来了额外的麻烦， 并且还可能会因为漏删或者错删了某个键而发生错误。

为了解决以上问题， 我们需要一种能够真正地把相关联的数据打包起来储存的数据结构， 而这种数据结构就是本章要介绍的散列键。

3.1 散列简介

Redis 的散列键会将一个键和一个散列在数据库里面关联起来， 用户可以在散列里面为任意多个字段（field）设置值。 跟字符串键一样， 散列的字段和值既可以是文本数据， 也可以是二进制数据。

通过使用散列键， 用户可以把相关联的多项数据储存到同一个散列里面， 以便对这些数据进行管理， 又或者针对它们执行批量操作。 比如图 3-2 就展示了一个使用散列储存文章数据的例子， 在这个例子中， 散列的键为 `article::10086` ， 而这个键对应的散列则包含了四个字段， 其中：

- `"title"` 字段储存着文章的标题 `"greeting"` ；
- `"content"` 字段储存着文章的内容 `"hello world"` ；
- `"author"` 字段储存着文章的作者名字 `"peter"` ；
- `"create_at"` 字段储存着文章的创建时间 `"1442744762.631885"` 。

图 3-2 使用散列储存文章数据

与之前使用字符串键储存文章数据的做法相比， 使用散列储存文章数据只需要在数据库里面创建一个键， 并且因为散列的字段名不需要添加任何前缀， 所以它们可以直接反映字段值储存的是什么数据。

Redis 为散列键提供了一系列操作命令， 通过使用这些命令， 用户可以：

- 为散列的字段设置值， 又或者只在字段不存在的情况下为它设置值。
- 从散列里面获取给定字段的值。
- 对储存着数字值的字段执行加法操作或者减法操作。
- 检查给定字段是否存在于散列当中。
- 从散列里面删除指定字段。
- 查看散列包含的字段数量。
- 一次为散列的多个字段设置值， 又或者一次从散列里面获取多个字段的值。
- 获取散列包含的所有字段、所有值又或者所有字段和值。

本章接下来将对以上提到的散列操作进行介绍， 说明如何使用这些操作去构建各种有用的应用程序， 并在最后详细地说明散列键与字符串键之间的区别。

3.2 HSET：为字段设置值

用户可以通过执行 `HSET` 命令， 为散列中的指定字段设置值：

```
HSET hash field value
```

根据给定的字段是否已经存在于散列里面， `HSET` 命令的行为也会有所不同：

- 如果给定字段并不存在于散列当中， 那么这次设置就是一次创建操作， 命令将在散列里面关联起给定的字段和值， 然后返回 `1` 。
- 如果给定的字段原本已经存在于散列里面， 那么这次设置就是一次更新操作， 命令将使用用户给定的新值去覆盖字段原有的旧值， 然后返回 `0` 。

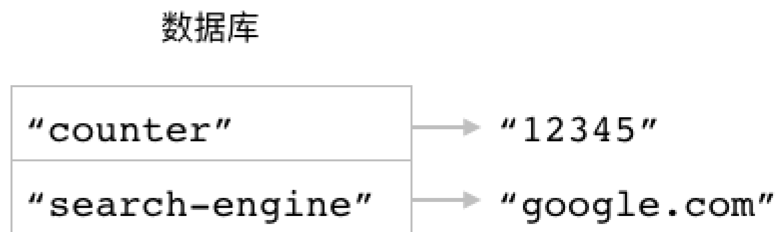
举个例子， 通过执行以下 `HSET` 命令， 我们可以创建出一个包含了四个字段的散列， 这四个字段分别储存了文章的标题、内容、作者以及创建日期：

```
redis> HSET article::10086 title "greeting"
(integer) 1

redis> HSET article::10086 content "hello world"
(integer) 1

redis> HSET article::10086 author "peter"
(integer) 1

redis> HSET article::10086 created_at "1442744762.631885"
(integer) 1
```

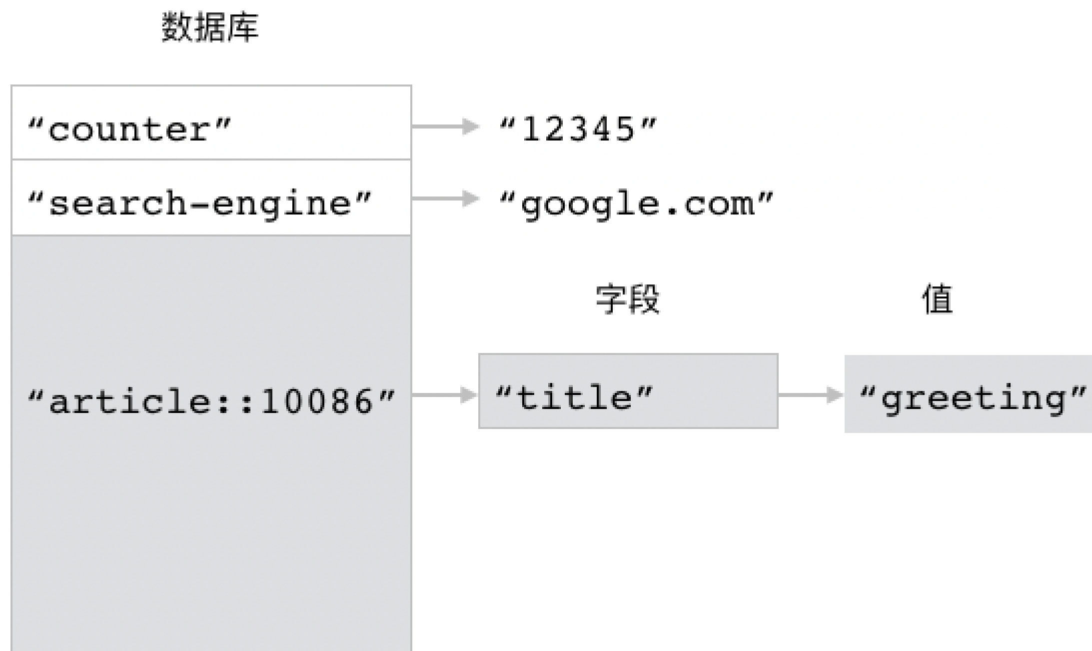
图 3-3 展示了以上这些 `HSET` 命令对散列 `article::10086` 进行设置的整个过程。

图 3-3 `HSET` 命令对 `article::10086` 进行设置的整个过程

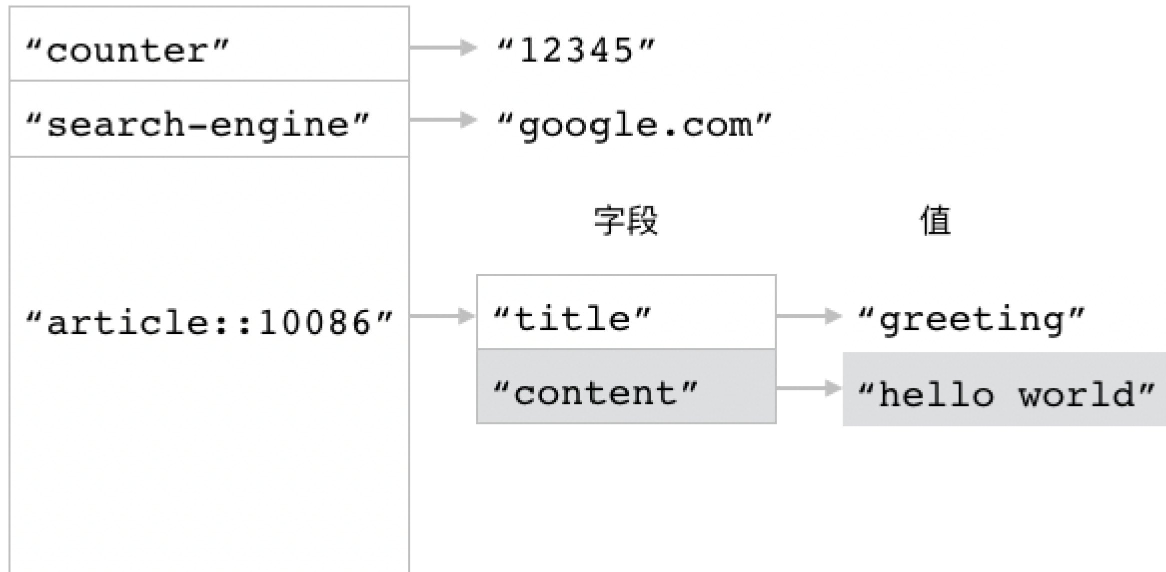
`HSET` 命令执行之前的数据库， `article::10086` 散列并不存在



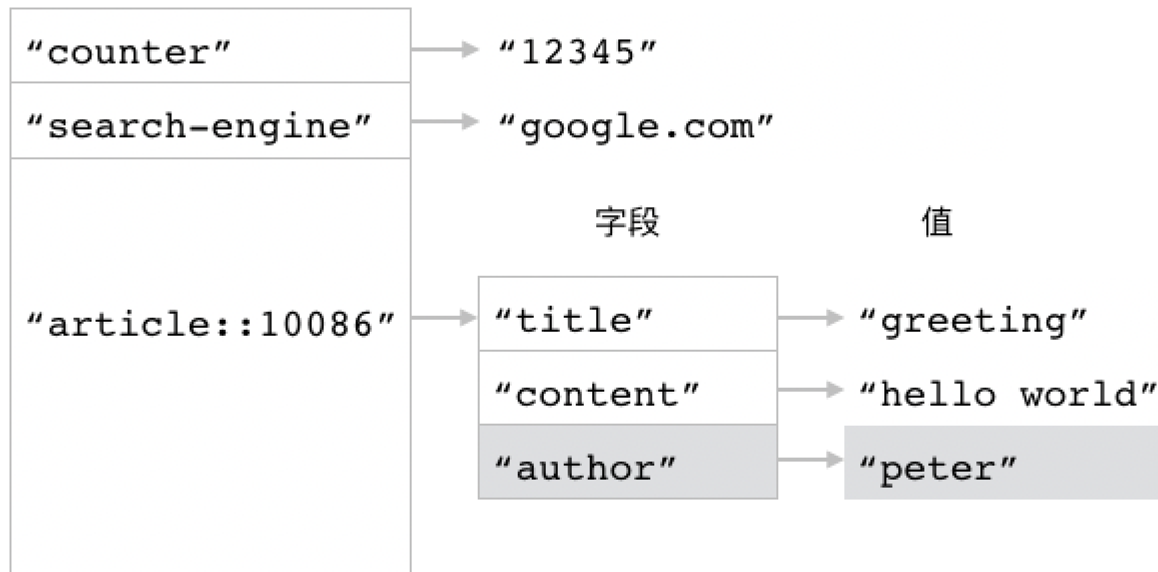
执行 `HSET article::10086 title "greeting"` 命令之后



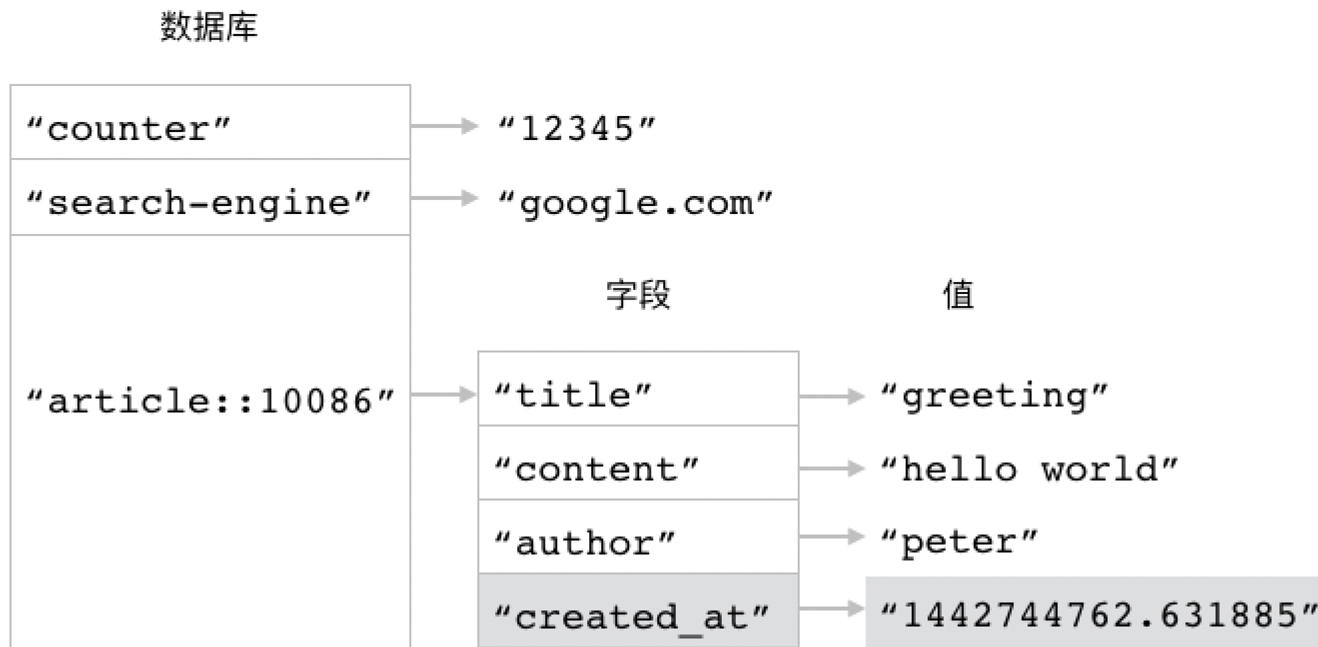
执行 `HSET article::10086 content "hello world"` 命令之后



执行 `HSET article::10086 author "peter"` 命令之后



执行 `HSET article::10086 created_at "1442744762.631885"` 命令之后



Note: 散列包含的字段就跟数据库包含的键一样， 在实际中都是以无序方式进行排列的， 不过本书为了展示方便， 一般都会把新字段添加到散列的末尾， 排在所有已有字段的后面。

3.2.1 使用新值覆盖旧值

正如之前所说， 如果用户在调用 HSET 命令时， 给定的字段已经存在于散列当中， 那么 HSET 命令将使用用户给定的新值去覆盖字段已有的旧值， 并返回 0 表示这是一次更新操作。

比如说， 以下代码就展示了如何使用 HSET 命令去更新 article::10086 散列的 title 字段以及 content 字段：

```
redis> HSET article::10086 title "Redis Tutorial"
(integer) 0

redis> HSET article::10086 content "Redis is a data structure store, ..."
(integer) 0
```

图 3-4 展示了被更新之后的 article::10086 散列。

图 3-4 被更新之后的 article::10086 散列

3.2.2 其他信息

属性	值
复杂度	O(1)
版本要求	HSET 命令从 Redis 2.0.0 版本开始可用。

3.3 HSETNX：只在字段不存在的情况下为它设置值

HSETNX 命令的作用和 HSET 命令的作用非常相似， 它们之间的区别在于， HSETNX 命令只会在指定字段不存在的情况下执行设置操作：

```
HSETNX hash field value
```

HSETNX 命令在字段不存在并且成功为它设置值时返回 1 ， 在字段已经存在并导致设置操作未能成功执行时返回 0 。

图 3-5 HSETNX 命令执行之前的 article::10086 散列

举个例子， 对于图 3-5 所示的 article::10086 散列来说， 执行以下 HSETNX 命令将不会对散列产生任何影响， 因为 HSETNX 命令想要设置的 title 字段已经存在：

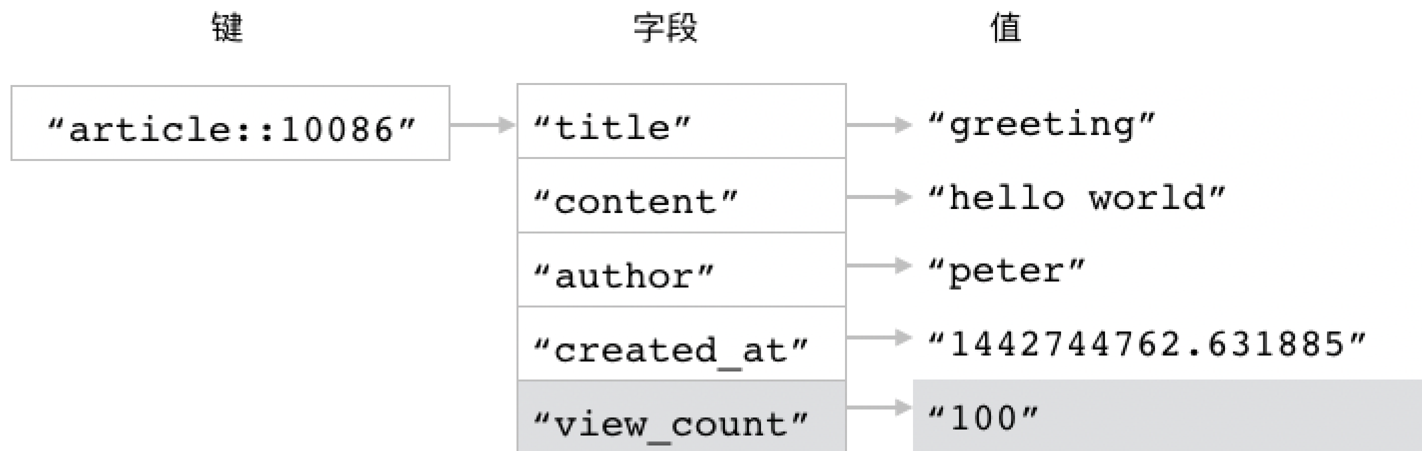
```
redis> HSETNX article::10086 title "Redis Performance Test"
(integer) 0     -- 设置失败
```

相反地， 如果我们使用 HSETNX 命令去对尚未存在的 view_count 字段进行设置， 那么这个命令将会顺利执行， 并将 view_count 字段的值设置为 100 ：

```
redis> HSETNX article::10086 view_count 100
(integer) 1     -- 设置成功
```

图 3-6 展示了 HSETNX 命令成功执行之后的 article::10086 散列。

图 3-6 HSETNX 命令执行之后的 article::10086 散列



3.3.1 其他信息

属性	值
复杂度	O(1)
版本要求	HSETNX 命令从 Redis 2.0.0 版本开始可用。

3.4 HGET：获取字段的值

HGET 命令可以根据用户给定的字段，从散列里面获取该字段的值：

```
HGET hash field
```

图 3-7 两个散列

比如对于图 3-7 所示的两个散列键来说， 执行以下命令可以从 `article::10086` 散列里面获取 `author` 字段的值：

```
redis> HGET article::10086 author
"peter"
```

而执行以下命令则可以从 `article::10086` 散列里面获取 `created_at` 字段的值：

```
redis> HGET article::10086 created_at
"1442744762.631885"
```

又比如说， 如果我们想要从 `account::54321` 散列里面获取 `email` 字段的值， 那么可以执行以下命令：

```
redis> HGET account::54321 email
"peter1984@spam_mail.com"
```

3.4.1 处理不存在的字段或者不存在的散列

如果用户给定的字段并不存在于散列当中， 那么 HGET 命令将返回一个空值。

举个例子， 在以下代码中， 我们尝试从 account::54321 散列里面获取 location 字段的值， 但由于 location 字段并不存在于 account::54321 散列当中， 所以 HGET 命令将返回一个空值：

```
redis> HGET account::54321 location
(nil)
```

尝试从一个不存在的散列里面获取一个不存在的字段值， 得到的结果也是一样的：

```
redis> HGET not-exists-hash not-exists-field
(nil)
```

3.4.2 其他信息

属性	值
复杂度	O(1)
版本要求	HGET 命令从 Redis 2.0.0 版本开始可用。

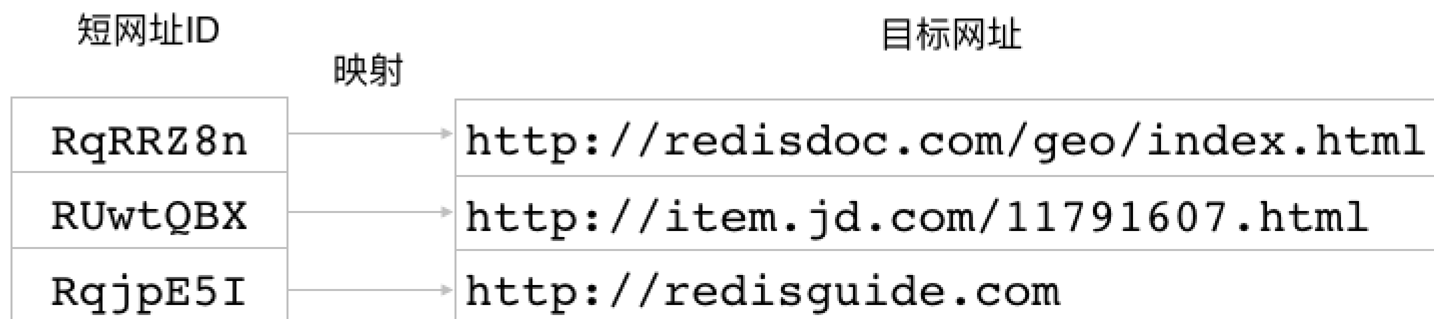
3.5 示例：实现短网址生成程序

为了给用户提供更多发言空间， 并记录用户在网站上的链接点击行为， 大部分社交网站都会将用户输入的网址转换为相应的短网址。 比如说， 如果我们在新浪微博发言时输入网址 http://redisdoc.com/geo/index.html ， 那么微博将把这个网址转换为相应的短网址 http://t.cn/RqRRZ8n ， 当用户访问这个短网址时， 微博在后台就会对这次点击进行一些数据统计， 然后再引导用户的浏览器跳转到 http://redisdoc.com/geo/index.html 上面。

创建短网址本质上就是要创建出短网址 ID 与目标网址之间的映射， 并在用户访问短网址时， 根据短网址的 ID 从映射记录中找出与之相对应的目标网址。 比如在前面的例子中， 微博的短网址程序就将短网址 http://t.cn/RqRRZ8n 中的 ID 值 RqRRZ8n 映射到了 http://redisdoc.com/geo/index.html 这个网址上面： 当用户访问短网址 http://t.cn/RqRRZ8n 时， 程序就会根据这个短网址的 ID 值 RqRRZ8n ， 找出与之对应的目标网址 http://redisdoc.com/geo/index.html ， 并将用户引导至目标网址上面去。

作为示例， 图 3-8 展示了几个微博短网址 ID 与目标网址之间的映射关系。

图 3-8 微博短网址映射关系示例



因为 Redis 的散列正好就非常适合用来储存短网址 ID 与目标网址之间的映射， 所以我们可以基于 Redis 的散列实现一个短网址程序， 代码清单 3-1 展示了一个这样的例子。

代码清单 3-1 使用散列实现的短网址程序： `/hash/shorty_url.py`

```python
from base36 import base10_to_base36

ID_COUNTER = "ShortyUrl::id_counter"
URL_HASH = "ShortyUrl::url_hash"

class ShortyUrl:

    def __init__(self, client):
        self.client = client

    def shorten(self, target_url):
        """
        为目标网址创建并储存相应的短网址 ID 。
        """
        # 为目标网址创建新的数字 ID
        new_id = self.client.incr(ID_COUNTER)
        # 通过将 10 进制数字转换为 36 进制数字来创建短网址 ID
        # 比如说，10 进制数字 10086 将被转换为 36 进制数字 7S6
        short_id = base10_to_base36(new_id)
        # 把短网址 ID 用作字段，目标网址用作值，
        # 将它们之间的映射关系储存到散列里面
        self.client.hset(URL_HASH, short_id, target_url)
        return short_id
```

```python
    def restore(self, short_id):
        """
        根据给定的短网址 ID ，返回与之对应的目标网址。
        """
        return self.client.hget(URL_HASH, short_id)
```

代码清单 3-2 将 10 进制数字转换成 36 进制数字的程序：`/hash/base36.py`

```python
def base10_to_base36(number):
    alphabets = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    result = ""

    while number != 0 :
        number, i = divmod(number, 36)
        result = (alphabets[i] + result)

    return result or alphabets[0]
```

`ShortyUrl` 类的 `shorten()` 方法负责为输入的网址生成短网址 ID ， 它的工作包括以下四个步骤：

1. 为每个给定的网址创建一个 10 进制数字 ID 。
2. 将 10 进制数字 ID 转换为 36 进制， 并将这个 36 进制数字用作给定网址的短网址 ID ， 这种方法在数字 ID 长度较大时可以有效地缩短数字 ID 的长度。 代码清单 3-2 展示了将数字从 10 进制转换成 36 进制的 `base10_to_base36` 函数的具体实现。
3. 将短网址 ID 和目标网址之间的映射关系储存到散列里面。
4. 向调用者返回刚刚生成的短网址 ID 。

另一方面， `restore()` 方法要做的事情和 `shorten()` 方法正好相反： 它会从储存着映射关系的散列里面取出与给定短网址 ID 相对应的目标网址， 然后将其返回给调用者。

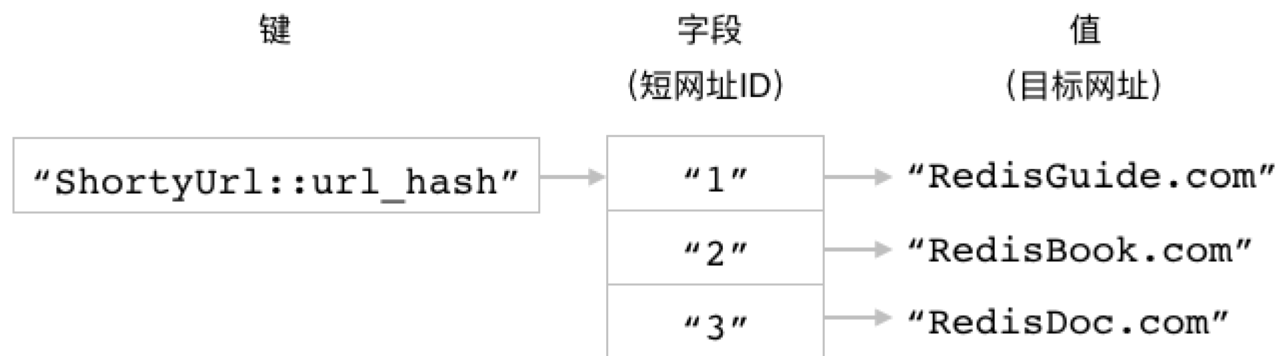
以下代码简单地展示了使用 `ShortyUrl` 程序创建短网址 ID 的方法， 以及根据短网址 ID 获取目标网址的方法：

```python
>>> from redis import Redis
>>> from shorty_url import ShortyUrl
>>> client = Redis(decode_responses=True)
>>> shorty_url = ShortyUrl(client)
>>> shorty_url.shorten("RedisGuide.com")  # 创建短网址 ID
'1'
```

```
>>> shorty_url.shorten("RedisBook.com")
'2'
>>> shorty_url.shorten("RedisDoc.com")
'3'
>>> shorty_url.restore("1")  # 根据短网址 ID 查找目标网址
'RedisGuide.com'
>>> shorty_url.restore("2")
'RedisBook.com'
```

图 3-9 展示了上面这段代码在数据库中创建的散列结构。

图 3-9 短网址程序在数据库中创建的散列结构



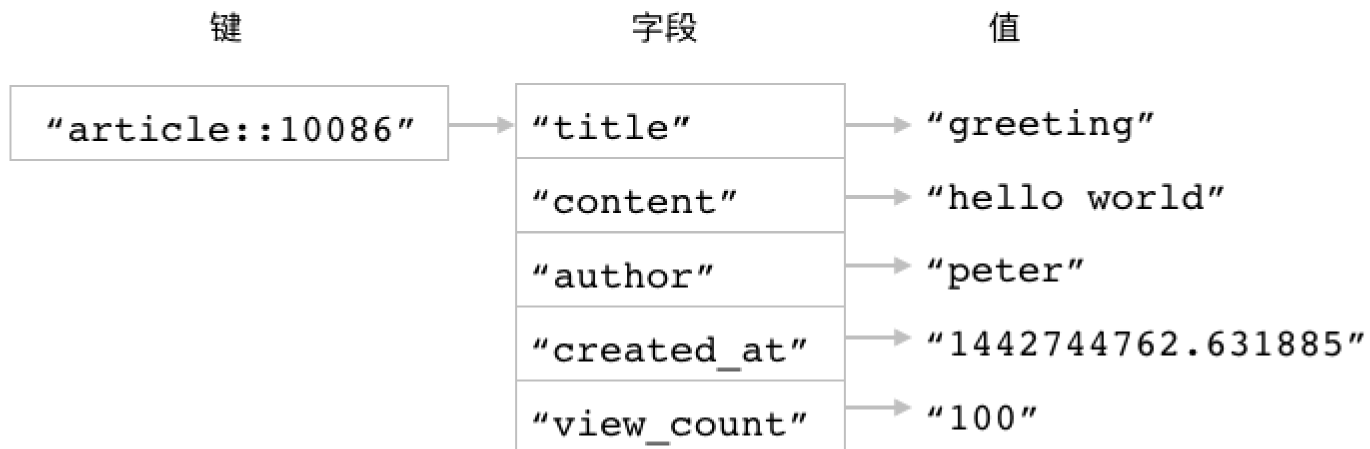
3.6 HINCRBY：对字段储存的整数值执行加法或减法操作

跟字符串键的 INCRBY 命令一样， 如果散列的字段里面储存着能够被 Redis 解释为整数的数字， 那么用户就可以使用 HINCRBY 命令为该字段的值加上指定的整数增量：

```
HINCRBY hash field increment
```

HINCRBY 命令在成功执行加法操作之后将返回字段当前的值作为命令的结果。

图 3-10 储存着文章数据的散列



比如说， 对于图 3-10 所示的 `article::10086` 散列， 我们可以通过执行以下命令， 为 `view_count` 字段的值加上 `1` ：

```
redis> HINCRBY article::10086 view_count 1
(integer) 101
```

也可以通过执行以下命令， 为 `view_count` 字段的值加上 `30` ：

```
redis> HINCRBY article::10086 view_count 30
(integer) 131
```

3.6.1 执行减法操作

因为 Redis 只为散列提供了用于执行加法操作的 `HINCRBY` 命令， 但是却并没有为散列提供相应的用于执行减法操作的命令， 所以如果用户需要对字段储存的整数值执行减法操作的话， 那么他就需要将一个负数增量传给 `HINCRBY` 命令， 从而达到对值执行减法计算的目的。

以下代码展示了如何使用 `HINCRBY` 命令去对 `view_count` 字段储存的整数值执行减法计算：

```
redis> HGET article::10086 view_count           -- 文章现在的浏览次数为 131 次
"131"
```

```
redis> HINCRBY article::10086 view_count -10    -- 将文章的浏览次数减少 10 次
"121"

redis> HINCRBY article::10086 view_count -21    -- 将文章的浏览次数减少 21 次
"100"

redis> HGET article::10086 view_count           -- 文章现在的浏览次数只有 100 次了
"100"
```

3.6.2 处理异常情况

HINCRBY 命令只能对储存着整数值的字段执行， 并且用户给定的增量也必须为整数， 尝试对非整数值字段执行 HINCRBY 命令， 又或者向 HINCRBY 命令提供非整数增量， 都会导致 HINCRBY 命令拒绝执行并报告错误。

以下是一些导致 HINCRBY 命令报错的例子：

```
redis> HINCRBY article::10086 view_count "fifty"    -- 增量必须能够被解释为整数
(error) ERR value is not an integer or out of range

redis> HINCRBY article::10086 view_count 3.14       -- 增量不能是浮点数
(error) ERR value is not an integer or out of range

redis> HINCRBY article::10086 content 100           -- 尝试向储存字符串值的字段执行 HINCRBY
(error) ERR hash value is not an integer
```

3.6.3 其他信息

属性	值
复杂度	O(1)
版本要求	HINCRBY 命令从 Redis 2.0.0 版本开始可用。

3.7 HINCRBYFLOAT：对字段储存的数字值执行浮点数加法或减法操作

HINCRBYFLOAT 命令的作用和 HINCRBY 命令的作用类似， 它们之间的主要区别在于 HINCRBYFLOAT 命令不仅可以使用整数作为增量， 还可以使用浮点数作为增量：

```
HINCRBYFLOAT hash field increment
```

HINCRBYFLOAT 命令在成功执行加法操作之后， 将返回给定字段的当前值作为结果。

举个例子， 通过执行以下 HINCRBYFLOAT 命令， 我们可以将 geo::peter 散列 longitude 字段的值从原来的 100.0099647 修改为 113.2099647 ：

```
redis> HGET geo::peter longitude
"100.0099647"

redis> HINCRBYFLOAT geo::peter longitude 13.2  -- 将字段的值加上 13.2
"113.2099647"
```

3.7.1 增量和字段值的类型限制

正如之前所说， HINCRBYFLOAT 命令不仅可以使用浮点数作为增量， 还可以使用整数作为增量：

```
redis> HGET number float
"3.14"

redis> HINCRBYFLOAT number float 10086  -- 整数增量
"10089.139999999999968"
```

此外， 不仅储存浮点数的字段可以执行 HINCRBYFLOAT 命令， 储存整数的字段也一样可以执行 HINCRBYFLOAT 命令：

```
redis> HGET number int  -- 储存整数的字段
"100"

redis> HINCRBYFLOAT number int 2.56
"102.56"
```

最后， 如果加法计算的结果能够被表示为整数， 那么 HINCRBYFLOAT 命令将使用整数作为计算结果：

```
redis> HGET number sum
"1.5"

redis> HINCRBYFLOAT number sum 3.5
"5"  -- 结果被表示为整数 5
```

3.7.2 执行减法操作

跟 HINCRBY 命令的情况一样， Redis 也没有为 HINCRBYFLOAT 命令提供对应的减法操作命令， 因此如果我们想要对字段储存的数字值执行浮点数减法操作， 那么只能通过向 HINCRBYFLOAT 命令传入负值浮点数来实现：

```
redis> HGET geo::peter longitude
"113.2099647"

redis> HINCRBYFLOAT geo::peter longitude -50  -- 将字段的值减去 50
"63.2099647"
```

3.7.3 其他信息

属性	值
复杂度	O(1)
版本要求	HINCRBYFLOAT 命令从 Redis 2.0.0 版本开始可用。

3.8 示例：使用散列键重新实现计数器

前面的《字符串》一章曾经展示过怎样使用 INCRBY 命令和 DECRBY 命令去构建一个计数器程序， 在学习了 HINCRBY 命令之后， 我们同样可以通过类似的原理来构建一个使用散列实现的计数器程序， 就像代码清单 3-3 展示的那样。

代码清单 3-3 使用散列实现的计数器：/hash/counter.py

```
class Counter:

    def __init__(self, client, hash_key, counter_name):
        self.client = client
        self.hash_key = hash_key
        self.counter_name = counter_name

    def increase(self, n=1):
        """
        将计数器的值加上 n ，然后返回计数器当前的值。
```

```
        如果用户没有显式地指定 n ，那么将计数器的值加上一。
        """
        return self.client.hincrby(self.hash_key, self.counter_name, n)

    def decrease(self, n=1):
        """
        将计数器的值减去 n ，然后返回计数器当前的值。
        如果用户没有显式地指定 n ，那么将计数器的值减去一。
        """
        return self.client.hincrby(self.hash_key, self.counter_name, -n)

    def get(self):
        """
        返回计数器的当前值。
        """
        value = self.client.hget(self.hash_key, self.counter_name)
        # 如果计数器并不存在，那么返回 0 作为默认值。
        if value is None:
            return 0
        else:
            return int(value)

    def reset(self):
        """
        将计数器的值重置为 0 。
        """
        self.client.hset(self.hash_key, self.counter_name, 0)
```

这个计数器实现充分地发挥了散列的特长：

- 它允许用户将多个相关联的计数器储存到同一个散列键里面实行集中管理， 而不必像字符串计数器那样，为每个计数器单独设置一个字符串键。
- 与此同时， 通过对散列中的不同字段执行 HINCRBY 命令， 程序可以对指定的计数器执行加法操作和减法操作， 而不会影响到储存在同一散列中的其他计数器。

作为例子， 以下代码展示了怎样将三个页面的浏览次数计数器储存到同一个散列里面：

```
>>> from redis import Redis
>>> from counter import Counter
>>> client = Redis(decode_responses=True)
>>> # 创建一个计数器，用于记录页面 /user/peter 被访问的次数
>>> user_peter_counter = Counter(client, "page_view_counters", "/user/peter")
>>> user_peter_counter.increase()
```

```
1L
>>> user_peter_counter.increase()
2L
>>> # 创建一个计数器，用于记录页面 /product/256 被访问的次数
>>> product_256_counter = Counter(client, "page_view_counters", "/product/256")
>>> product_256_counter.increase(100)
100L
>>> # 创建一个计数器，用于记录页面 /product/512 被访问的次数
>>> product_512_counter = Counter(client, "page_view_counters", "/product/512")
>>> product_512_counter.increase(300)
300L
```

因为 `user_peter_counter` 、 `product_256_counter` 和 `product_512_counter` 这三个计数器都是用来记录页面浏览次数的， 所以这些计数器都被放到了 `page_view_counters` 这个散列里面； 与此类似， 如果我们要创建一些用途完全不一样的计数器， 那么只需要把新的计数器放到其他散列里面就可以了。

比如说， 以下代码就展示了怎样将文件 `dragon_rises.mp3` 和文件 `redisbook.pdf` 的下载次数计数器放到 `download_counters` 散列里面：

```
>>> dragon_rises_counter = Counter(client, "download_counters", "dragon_rises.mp3")
>>> dragon_rises_counter.increase(10086)
10086L
>>> redisbook_counter = Counter(client, "download_counters", "redisbook.pdf")
>>> redisbook_counter.increase(65535)
65535L
```

图 3-11 展示了 `page_view_counters` 和 `download_counters` 这两个散列以及它们包含的各个计数器的样子。

图 3-11 散列计数器数据结构示意图

通过使用不同的散列储存不同类型的计数器， 程序能够让代码生成的数据结构变得更容易理解， 并且在针对某种类型的计数器执行批量操作时也会变得更加方便。 比如说， 当我们不再需要下载计数器的时候， 只要把 `download_counters` 散列删掉就可以移除所有下载计数器了。

3.9 HSTRLEN：获取字段值的字节长度

用户可以通过使用 HSTRLEN 命令， 获取给定字段值的字节长度：

```
HSTRLEN hash field
```

图 3-12 使用散列储存文章数据

比如对于图 3-12 所示的 `article::10086` 散列来说， 我们可以通过执行以下 `HSTRLEN` 命令， 取得 `title` 、 `content` 、 `author` 等字段值的字节长度：

```
redis> HSTRLEN article::10086 title
(integer) 8    -- title 字段的值 "greeting" 长 8 个字节

redis> HSTRLEN article::10086 content
(integer) 11    -- content 字段的值 "hello world" 长 11 个字节

redis> HSTRLEN article::10086 author
(integer) 5    -- author 字段的值 "peter" 长 6 个字节
```

如果给定的字段或散列并不存在， 那么 `HSTRLEN` 命令将返回 `0` 作为结果：

```
redis> HSTRLEN article::10086 last_updated_at  -- 字段不存在
(integer) 0

redis> HSTRLEN not-exists-hash not-exists-key  -- 散列不存在
(integer) 0
```

3.9.1 其他信息

属性	值
复杂度	O(1)
版本要求	`HSTRLEN` 命令从 Redis 3.2.0 版本开始可用。

3.10 HEXISTS：检查字段是否存在

HEXISTS 命令可以检查用户给定的字段是否存在于散列当中：

```
HEXISTS hash field
```

如果散列包含了给定的字段， 那么命令返回 1 ； 否则的话， 命令返回 0 。

比如说， 以下代码就展示了如何使用 HEXISTS 命令去检查 article::10086 散列是否包含某些字段：

```
redis> HEXISTS article::10086 author
(integer) 1     -- 包含该字段

redis> HEXISTS article::10086 content
(integer) 1

redis> HEXISTS article::10086 last_updated_at
(integer) 0     -- 不包含该字段
```

从 HEXISTS 命令的执行结果可以看出， article::10086 散列包含了 author 字段和 content 字段， 但是却并没有包含 last_updated_at 字段。

如果用户给定的散列并不存在， 那么 HEXISTS 命令对于这个散列所有字段的检查结果都是不存在：

```
redis> HEXISTS not-exists-hash not-exists-field
(integer) 0

redis> HEXISTS not-exists-hash another-not-exists-field
(integer) 0
```

3.10.1 其他信息

属性	值
复杂度	O(1)
版本要求	HEXISTS 命令从 Redis 2.0.0 版本开始可用。

3.11 HDEL：删除字段

HDEL 命令用于删除散列中的指定字段及其相关联的值：

```
HDEL hash field
```

当给定字段存在于散列当中并且被成功删除时， 命令返回 1 ； 如果给定字段并不存在于散列当中， 又或者给定的散列并不存在， 那么命令将返回 0 表示删除失败。

举个例子， 对于图 3-13 所示的 article::10086 散列， 我们可以使用以下命令去删除散列的 author 字段和 created_at 字段， 以及与这些字段相关联的值：

```
redis> HDEL article::10086 author
(integer) 1

redis> HDEL article::10086 created_at
(integer) 1
```

图 3-14 展示了以上两个 HDEL 命令执行之后， article::10086 散列的样子。

图 3-13 article::10086 散列

图 3-14 删除了两个字段之后的 article::10086 散列

3.11.1 其他信息

属性	值
复杂度	O(1)
版本要求	HDEL 命令从 Redis 2.0.0 版本开始可用。

3.12 HLEN：获取散列包含的字段数量

用户可以通过使用 HLEN 命令获取给定散列包含的字段数量：

```
HLEN hash
```

图 3-15 两个散列键

比如对于图 3-15 中展示的 `article::10086` 散列和 `account::54321` 散列来说， 我们可以通过执行以下命令来获取 `article::10086` 散列包含的字段数量：

```
redis> HLEN article::10086
(integer) 4      -- 这个散列包含 4 个字段
```

又或者通过执行以下命令来获取 `account::54321` 散列包含的字段数量：

```
redis> HLEN account::54321
(integer) 2      -- 这个散列包含 2 个字段
```

另一方面， 如果用户给定的散列并不存在， 那么 `HLEN` 命令将返回 `0` 作为结果：

```
redis> HLEN not-exists-hash
(integer) 0
```

3.12.1 其他信息

属性	
复杂度	O(1)
版本要求	`HLEN` 命令从 Redis 2.0.0 版本开始可用。

3.13 示例：实现用户登录会话

为了方便用户， 网站一般都会为已登录的用户生成一个加密令牌， 然后把这个令牌分别储存在服务器端和客户端， 之后每当用户再次访问该网站的时候， 网站就可以通过验证客户端提交的令牌来确认用户的身份， 从而使得用户不必重复地执行登录操作。

另一方面， 为了防止用户因为长时间不输入密码而导致遗忘密码， 并且为了保证令牌的安全性， 网站一般都会为令牌设置一个过期期限（比如一个月）， 当期限到达之后， 用户的会话就会过时， 而网站则会要求用户重新登录。

上面描述的这种使用令牌来避免重复登录的机制一般被称为登录会话（login session）， 通过使用 Redis 的散列， 我们可以构建出代码清单 3-4 所示的登录会话程序。

代码清单 3-4 使用散列实现的登录会话程序： `/hash/login_session.py`

```python
import random
from time import time  # 获取浮点数格式的 unix 时间戳
from hashlib import sha256

# 会话的默认过期时间
DEFAULT_TIMEOUT = 3600*24*30    # 一个月

# 储存会话令牌以及会话过期时间戳的散列
SESSION_TOKEN_HASH = "session::token"
SESSION_EXPIRE_TS_HASH = "session::expire_timestamp"

# 会话状态
SESSION_NOT_LOGIN = "SESSION_NOT_LOGIN"
SESSION_EXPIRED = "SESSION_EXPIRED"
SESSION_TOKEN_CORRECT = "SESSION_TOKEN_CORRECT"
SESSION_TOKEN_INCORRECT = "SESSION_TOKEN_INCORRECT"

def generate_token():
    """
    生成一个随机的会话令牌。
```

```python
    """
    random_string = str(random.getrandbits(256)).encode('utf-8')
    return sha256(random_string).hexdigest()


class LoginSession:

    def __init__(self, client, user_id):
        self.client = client
        self.user_id = user_id

    def create(self, timeout=DEFAULT_TIMEOUT):
        """
        创建新的登录会话并返回会话令牌，
        可选的 timeout 参数用于指定会话的过期时间（以秒为单位）。
        """
        # 生成会话令牌
        user_token = generate_token()
        # 计算会话到期时间戳
        expire_timestamp = time()+timeout
        # 以用户 ID 为字段，将令牌和到期时间戳分别储存到两个散列里面
        self.client.hset(SESSION_TOKEN_HASH, self.user_id, user_token)
        self.client.hset(SESSION_EXPIRE_TS_HASH, self.user_id, expire_timestamp)
        # 将会话令牌返回给用户
        return user_token

    def validate(self, input_token):
        """
        根据给定的令牌验证用户身份。
        这个方法有四个可能的返回值，分别对应四种不同情况：
        1. SESSION_NOT_LOGIN —— 用户尚未登录
        2. SESSION_EXPIRED —— 会话已过期
        3. SESSION_TOKEN_CORRECT —— 用户已登录，并且给定令牌与用户令牌相匹配
        4. SESSION_TOKEN_INCORRECT —— 用户已登录，但给定令牌与用户令牌不匹配
        """
        # 尝试从两个散列里面取出用户的会话令牌以及会话的过期时间戳
        user_token = self.client.hget(SESSION_TOKEN_HASH, self.user_id)
        expire_timestamp = self.client.hget(SESSION_EXPIRE_TS_HASH, self.user_id)

        # 如果会话令牌或者过期时间戳不存在，那么说明用户尚未登录
        if (user_token is None) or (expire_timestamp is None):
            return SESSION_NOT_LOGIN

        # 将当前时间戳与会话的过期时间戳进行对比，检查会话是否已过期
        # 因为 HGET 命令返回的过期时间戳是字符串格式的
        # 所以在进行对比之前要先将它转换成原来的浮点数格式
        if time() > float(expire_timestamp):
            return SESSION_EXPIRED
```

```python
        # 用户令牌存在并且未过期，那么检查它与给定令牌是否一致
        if input_token == user_token:
            return SESSION_TOKEN_CORRECT
        else:
            return SESSION_TOKEN_INCORRECT

    def destroy(self):
        """
        销毁会话。
        """
        # 从两个散列里面分别删除用户的会话令牌以及会话的过期时间戳
        self.client.hdel(SESSION_TOKEN_HASH, self.user_id)
        self.client.hdel(SESSION_EXPIRE_TS_HASH, self.user_id)
```

`LoginSession` 的 `create()` 方法首先会计算出随机的会话令牌以及会话的过期时间戳， 然后使用用户 ID 作为字段， 将令牌和过期时间戳分别储存到两个散列里面。

在此之后， 每当客户端向服务器发送请求并提交令牌的时候， 程序就会使用 `validate()` 方法验证被提交令牌的正确性： `validate()` 方法会根据用户的 ID ， 从两个散列里面分别取出用户的会话令牌以及会话的过期时间戳， 然后通过一系列检查判断令牌是否正确以及会话是否过期。

最后， `destroy()` 方法可以在用户手动登出（logout）时调用， 它可以删除用户的会话令牌以及会话的过期时间戳， 让用户重新回到未登录状态。

在拥有 `LoginSession` 程序之后， 我们可以通过执行以下代码， 为用户 peter 创建出相应的会话令牌：

```python
>>> from redis import Redis
>>> from login_session import LoginSession
>>>
>>> client = Redis(decode_responses=True)
>>> session = LoginSession(client, "peter")
>>>
>>> token = session.create()
>>> token
'3b000071e59fcdcaa46b900bb5c484f653de67055fde622f34c255a65bd9a561'
```

并通过以下代码， 验证给定令牌的正确性：

```python
>>> session.validate("wrong_token")
'SESSION_TOKEN_INCORRECT'
```

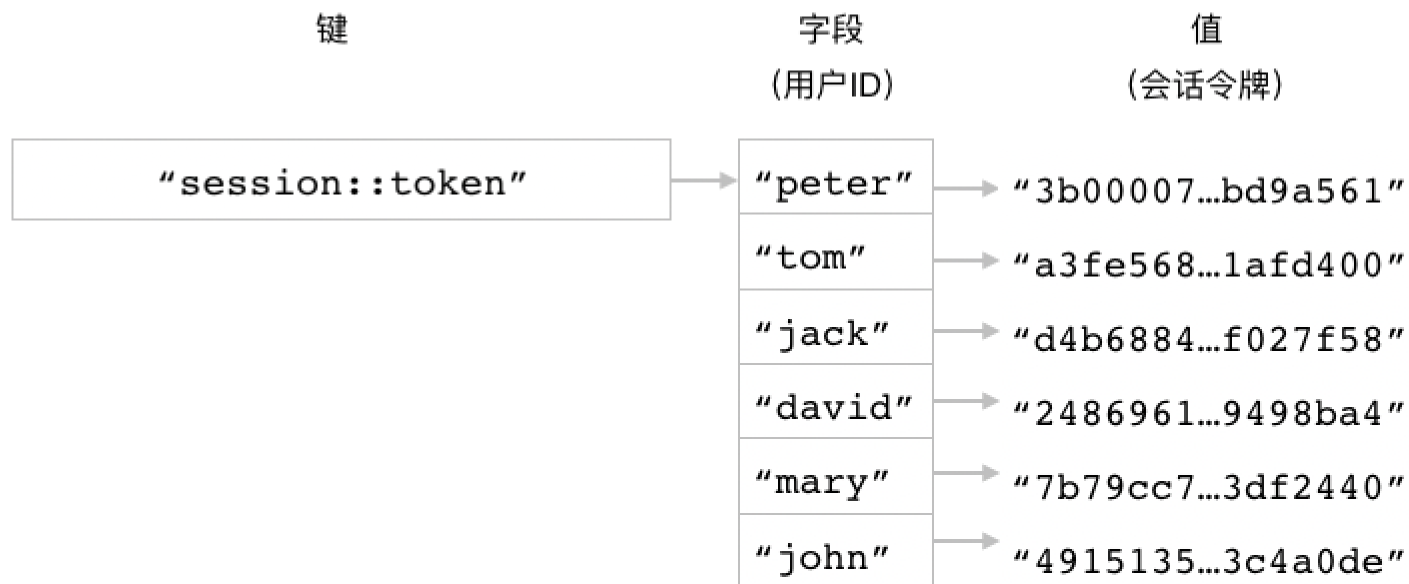
```
>>>
>>> session.validate(token)
'SESSION_TOKEN_CORRECT'
```

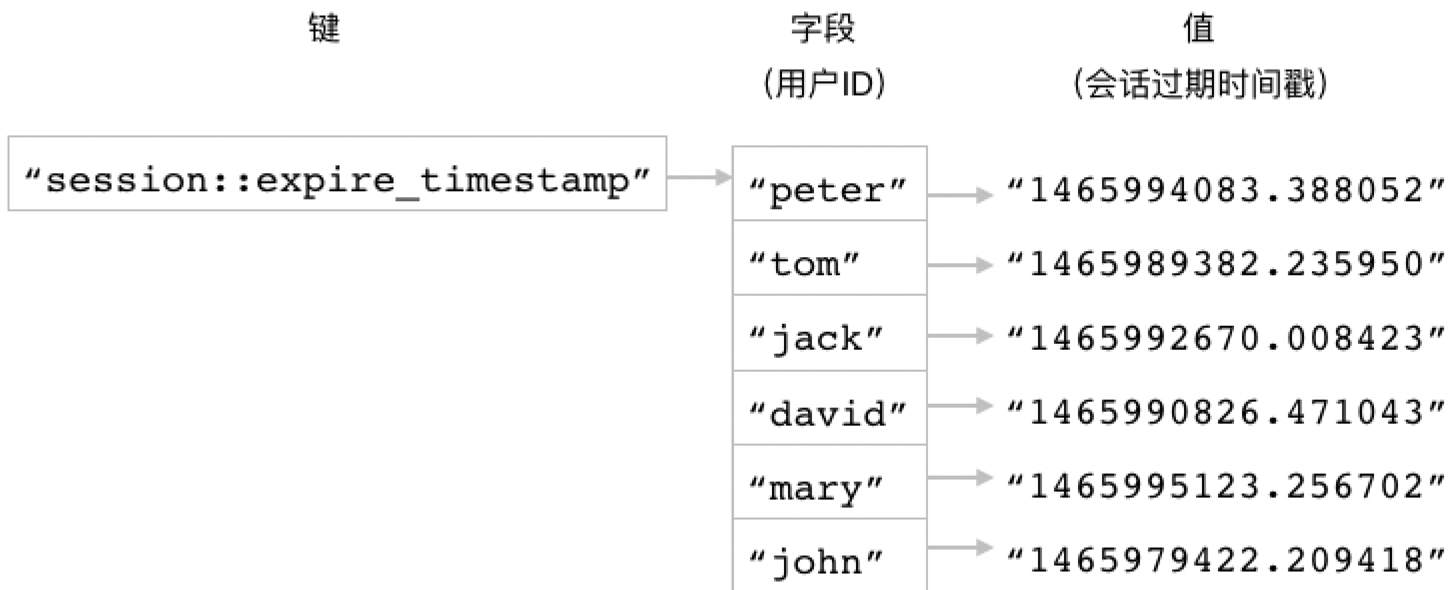
然后在会话使用完毕之后， 通过执行以下代码来销毁会话：

```
>>> session.destroy()
>>>
>>> session.validate(token)
'SESSION_NOT_LOGIN'
```

图 3-16 展示了使用 `LoginSession` 程序在数据库里面创建多个会话时的样子。

图 3-16 登录会话程序数据结构示意图


键
字段
（用户ID）
值
（会话令牌）
“session::token”
“peter”
“tom”
“jack”
“david”
“mary”
“john”
“3b00007…bd9a561”
“a3fe568…1afd400”
“d4b6884…f027f58”
“2486961…9498ba4”
“7b79cc7…3df2440”
“4915135…3c4a0de”


键
字段
（用户ID）
值
（会话过期时间戳）
“session::expire_timestamp”
“peter”
“tom”
“jack”
“david”
“mary”
“john”
“1465994083.388052”
“1465989382.235950”
“1465992670.008423”
“1465990826.471043”
“1465995123.256702”
“1465979422.209418”

3.14 HMSET：一次为多个字段设置值

用户可以通过 HMSET 命令， 一次为散列中的多个字段设置值：

```
HMSET hash field value [field value ...]
```

HMSET 命令在设置成功时返回 OK 。

图 3-17 储存文章数据的散列

比如说， 为了构建图 3-17 所示的散列， 我们可能会执行以下四个 HSET 命令：

```
redis> HSET article::10086 title "greeting"
(integer) 1

redis> HSET article::10086 content "hello world"
(integer) 1

redis> HSET article::10086 author "peter"
(integer) 1

redis> HSET article::10086 created_at "1442744762.631885"
(integer) 1
```

但是接下来的这一条 HMSET 命令可以更方便地完成相同的工作：

```
redis> HMSET article::10086 title "greeting" content "hello world" author "peter" created_at "14427447
OK
```

此外， 因为客户端在执行这条 `HMSET` 命令时只需要与 Redis 服务器进行一次通信， 而上面的四条 `HSET` 命令则需要客户端与 Redis 服务器进行四次通信， 所以前者的执行速度要比后者快得多。

3.14.1 使用新值覆盖旧值

如果用户给定的字段已经存在于散列当中， 那么 `HMSET` 命令将使用用户给定的新值去覆盖字段已有的旧值。

比如对于 `title` 和 `content` 这两个已经存在于 `article::10086` 散列的字段来说：

```
redis> HGET article::10086 title
"greeting"

redis> HGET article::10086 content
"hello world"
```

如果我们执行以下命令：

```
redis> HMSET article::10086 title "Redis Tutorial" content "Redis is a data structure store, ..."
OK
```

那么 `title` 字段和 `content` 字段已有的旧值将被新值覆盖：

```
redis> HGET article::10086 title
"Redis Tutorial"

redis> HGET article::10086 content
"Redis is a data structure store, ..."
```

3.14.2 其他信息

属性	值
复杂度	O(N)，其中 N 为被设置的字段数量。
版本要求	`HMSET` 命令从 Redis 2.0.0 版本开始可用。

3.15 HMGET：一次获取多个字段的值

通过使用 HMGET 命令，用户可以一次从散列里面获取多个字段的值：

```
HMGET hash field [field ...]
```

HMGET 命令将按照用户给定字段的顺序依次返回与之对应的值。

图 3-18 储存文章数据的散列

比如对于图 3-18 所示的 article::10086 散列来说，我们可以使用以下命令来获取它的 author 字段和 created_at 字段的值：

```
redis> HMGET article::10086 author created_at
1) "peter"                -- author 字段的值
2) "1442744762.631885"    -- created_at 字段的值
```

又或者使用以下命令来获取它的 title 字段和 content 字段的值：

```
redis> HMGET article::10086 title content
1) "greeting"        -- title 字段的值
2) "hello world"     -- content 字段的值
```

跟 HGET 命令一样， 如果用户向 HMGET 命令提供的字段或者散列不存在， 那么 HMGET 命令将返回空值作为结果：

```
redis> HMGET article::10086 title content last_updated_at
1) "greeting"
2) "hello world"
3) (nil)    -- last_updated_at 字段不存在于 article::10086 散列

redis> HMGET not-exists-hash field1 field2 field3  -- 散列不存在
1) (nil)
2) (nil)
3) (nil)
```

3.15.1 其他信息

属性	值
复杂度	O(N)，其中 N 为用户给定的字段数量。
版本要求	HMGET 命令从 Redis 2.0.0 版本开始可用。

3.16 HKEYS、HVALS、HGETALL：获取所有字段、所有值或者所有字段和值

Redis 为散列提供了 HKEYS 、 HVALS 和 HGETALL 这三个命令， 它们可以分别用于获取散列包含的所有字段、所有值以及所有字段和值：

```
HKEYS hash

HVALS hash

HGETALL hash
```

图 3-19 储存文章数据的散列

举个例子， 对于图 3-19 所示的 `article::10086` 散列来说， 我们可以使用 `HKEYS` 命令去获取它包含的所有字段：

```
redis> HKEYS article::10086
1) "title"
2) "content"
3) "author"
4) "created_at"
```

也可以使用 `HVALS` 命令去获取它包含的所有值：

```
redis> HVALS article::10086
1) "greeting"
2) "hello world"
3) "peter"
4) "1442744762.631885"
```

还可以使用 `HGETALL` 命令去获取它包含的所有字段和值：

```
redis> HGETALL article::10086
1) "title"      -- 字段
2) "greeting"   -- 字段的值
3) "content"
4) "hello world"
5) "author"
6) "peter"
7) "created_at"
8) "1442744762.631885"
```

在 HGETALL 命令返回的结果列表当中， 每两个连续的元素就代表了散列中的一对字段和值， 其中单数位置上的元素为字段， 而复数位置上的元素则为字段的值。

另一方面， 如果用户给定的散列并不存在， 那么 HKEYS 、 HVALS 和 HGETALL 都将返回一个空列表：

```
redis> HKEYS not-exists-hash
(empty list or set)

redis> HVALS not-exists-hash
(empty list or set)

redis> HGETALL not-exists-hash
(empty list or set)
```

3.16.1 字段在散列中的排列顺序

Redis 散列包含的字段在底层是以无序方式储存的， 根据字段插入的顺序不同， 包含相同字段的散列在执行 HKEYS 命令、 HVALS 命令和 HGETALL 命令时可能会得到不同的结果， 因此用户在使用这三个命令的时候， 不应该对它们返回的元素的排列顺序做任何假设。 如果有需要的话， 用户可以对这些命令返回的元素进行排序， 使得它们从无序变为有序。

举个例子， 如果我们以不同的设置顺序创建两个完全相同的散列 hash1 和 hash2 ：

```
redis> HMSET hash1 field1 value1 field2 value2 field3 value3
OK

redis> HMSET hash2 field3 value3 field2 value2 field1 value1
OK
```

那么 HKEYS 命令将以不同的顺序返回这两个散列的字段：

```
redis> HKEYS hash1
1) "field1"
2) "field2"
3) "field3"

redis> HKEYS hash2
1) "field3"
2) "field2"
3) "field1"
```

而 HVALS 命令则会以不同的顺序返回这两个散列的字段值：

```
redis> HVALS hash1
1) "value1"
2) "value2"
3) "value3"

redis> HVALS hash2
1) "value3"
2) "value2"
3) "value1"
```

至于 HGETALL 命令则会以不同的顺序返回这两个散列的字段和值：

```
redis> HGETALL hash1
1) "field1"
2) "value1"
3) "field2"
4) "value2"
5) "field3"
6) "value3"

redis> HGETALL hash2
1) "field3"
2) "value3"
3) "field2"
4) "value2"
5) "field1"
6) "value1"
```

3.16.2 其他信息

属性	值
复杂度	HKEYS 命令、 HVALS 命令和 HGETALL 命令的复杂度都为 O(N) ， 其中 N 为散列包含的字段数量。
版本要求	HKEYS 命令、 HVALS 命令和 HGETALL 命令都从 Redis 2.0.0 版本开始可用。

3.17 示例：储存图数据

在构建地图应用、设计电路图、进行任务调度、分析网络流量等多种任务中， 都需要对图（graph）数据结构实施建模， 并储存相关的图数据。 对于不少数据库来说， 想要高效直观地储存图数据并不是一件容易的事情， 但是 Redis 却能够以多种不同的方式表示图数据结构， 其中一种方法就是使用散列。

图 3-20 简单的带权重有向图

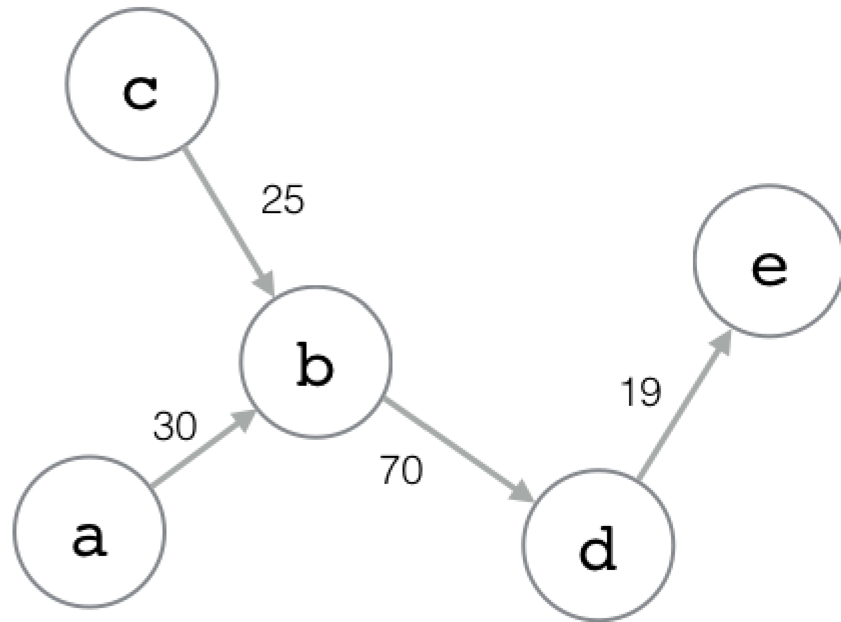
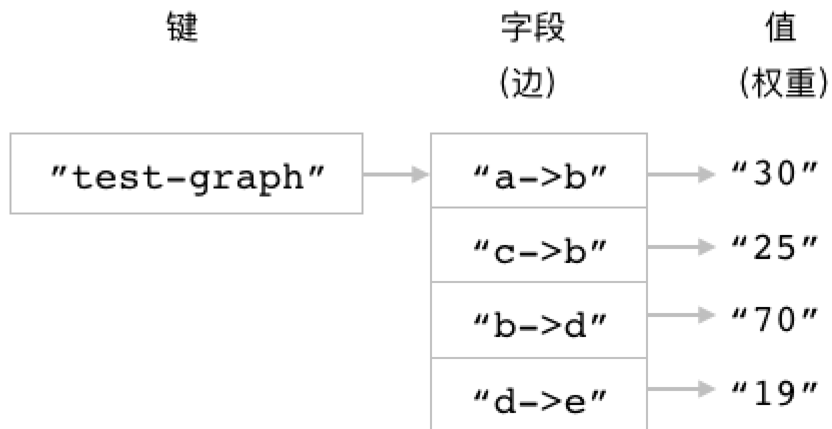


图 3-21 图对应的散列键



比如说， 假设我们想要储存图 3-20 所示的带权重有向图， 那么可以创建一个图 3-21 所示的散列键， 这个散列键会以 `start_vertex->end_vertex` 的形式， 将各个顶点之间的边储存到散列的字段里面， 并将字段的值设置成边的权重。 通过这种方法， 我们可以将图的所有相关数据全部储存到散列里面， 代码清单 3-5 展示了使用这种方法实现的图数据储存程序。

代码清单 3-5 使用散列实现的图数据储存程序： `/hash/graph.py`

```
def make_edge_name_from_vertexs(start, end):
    """
    使用边的起点和终点组建边的名字。
    例子： 对于 start 为 "a" 、 end 为 "b" 的输入， 这个函数将返回 "a->b" 。
    """
    return str(start) + "->" + str(end)

def decompose_vertexs_from_edge_name(name):
    """
    从边的名字中分解出边的起点和终点。
    例子： 对于输入 "a->b" ， 这个函数将返回结果 ["a", "b"] 。
    """
    return name.split("->")


class Graph:

    def __init__(self, client, key):
        self.client = client
```

```python
        self.key = key

    def add_edge(self, start, end, weight):
        """
        添加一条从顶点 start 连接至顶点 end 的边，并将边的权重设置为 weight 。
        """
        edge = make_edge_name_from_vertexs(start, end)
        self.client.hset(self.key, edge, weight)

    def remove_edge(self, start, end):
        """
        移除从顶点 start 连接至顶点 end 的一条边。
        这个方法在成功删除边时返回 True ，
        因为边不存在而导致删除失败时返回 False 。
        """
        edge = make_edge_name_from_vertexs(start, end)
        return self.client.hdel(self.key, edge)

    def get_edge_weight(self, start, end):
        """
        获取从顶点 start 连接至顶点 end 的边的权重，
        如果给定的边不存在，那么返回 None 。
        """
        edge = make_edge_name_from_vertexs(start, end)
        return self.client.hget(self.key, edge)

    def has_edge(self, start, end):
        """
        检查顶点 start 和顶点 end 之间是否有边，
        是的话返回 True ，否则返回 False 。
        """
        edge = make_edge_name_from_vertexs(start, end)
        return self.client.hexists(self.key, edge)

    def add_multi_edges(self, *tuples):
        """
        一次向图中添加多条边。
        这个方法接受任意多个格式为 (start, end, weight) 的三元组作为参数。
        """
        # redis-py 客户端的 hmset() 方法接受一个字典作为参数
        # 格式为 {field1: value1, field2: value2, ...}
        # 为了一次对图中的多条边进行设置
        # 我们要将待设置的各条边以及它们的权重储存在以下字典
        nodes_and_weights = {}

        # 遍历输入的每个三元组，从中取出边的起点、终点和权重
        for start, end, weight in tuples:
            # 根据边的起点和终点，创建出边的名字
```

```python
            edge = make_edge_name_from_vertexs(start, end)
            # 使用边的名字作为字段，边的权重作为值，把边及其权重储存到字典里面
            nodes_and_weights[edge] = weight

        # 根据字典中储存的字段和值，对散列进行设置
        self.client.hmset(self.key, nodes_and_weights)

    def get_multi_edge_weights(self, *tuples):
        """
        一次获取多条边的权重。
        这个方法接受任意多个格式为 (start, end) 的二元组作为参数，
        然后返回一个列表作为结果，列表中依次储存着每条输入边的权重。
        """
        # hmget() 方法接受一个格式为 [field1, field2, ...] 的列表作为参数
        # 为了一次获取图中多条边的权重
        # 我们需要把所有想要获取权重的边的名字依次放入到以下列表里面
        edge_list = []

        # 遍历输入的每个二元组，从中获取边的起点和终点
        for start, end in tuples:
            # 根据边的起点和终点，创建出边的名字
            edge = make_edge_name_from_vertexs(start, end)
            # 把边的名字放入到列表中
            edge_list.append(edge)

        # 根据列表中储存的每条边的名字，从散列里面获取它们的权重
        return self.client.hmget(self.key, edge_list)

    def get_all_edges(self):
        """
        以集合形式返回整个图包含的所有边，
        集合包含的每个元素都是一个 (start, end) 格式的二元组。
        """
        # hkeys() 方法将返回一个列表，列表中包含多条边的名字
        # 例如 ["a->b", "b->c", "c->d"]
        edges = self.client.hkeys(self.key)

        # 创建一个集合，用于储存二元组格式的边
        result = set()
        # 遍历每条边的名字
        for edge in edges:
            # 根据边的名字，分解出边的起点和终点
            start, end = decompose_vertexs_from_edge_name(edge)
            # 使用起点和终点组成一个二元组，然后把它放入到结果集合里面
            result.add((start, end))

        return result
```

```python
    def get_all_edges_with_weight(self):
        """
        以集合形式返回整个图包含的所有边，以及这些边的权重。
        集合包含的每个元素都是一个 (start, end, weight) 格式的三元组。
        """
        # hgetall() 方法将返回一个包含边和权重的字典作为结果
        # 格式为 {edge1: weight1, edge2: weight2, ...}
        edges_and_weights = self.client.hgetall(self.key)

        # 创建一个集合，用于储存三元组格式的边和权重
        result = set()
        # 遍历字典中的每个元素，获取边以及它的权重
        for edge, weight in edges_and_weights.items():
            # 根据边的名字，分解出边的起点和终点
            start, end = decompose_vertexs_from_edge_name(edge)
            # 使用起点、终点和权重构建一个三元组，然后把它添加到结果集合里面
            result.add((start, end, weight))

        return result
```

这个图数据储存程序的核心概念就是把边（edge）的起点和终点组合成一个字段名， 并把边的权重（weight）用作字段的值， 然后使用 `HSET` 命令或者 `HMSET` 命令把它们储存到散列里面。 比如说， 如果用户输入的边起点为 `"a"` ， 终点为 `"b"` ， 权重为 `"30"` ， 那么程序将执行命令 `HSET hash "a->b" 30` ， 把 `"a"` 至 `"b"` 的这条边及其权重 `30` 储存到散列里面。

在此之后， 程序就可以使用 `HDEL` 命令去删除图的某条边， 使用 `HGET` 命令或者 `HMGET` 命令去获取边的权重， 使用 `HEXISTS` 命令去检查边是否存在， 又或者使用 `HKEYS` 命令和 `HGETALL` 命令去获取图的所有边以及权重。

比如说， 我们可以通过执行以下代码， 构建出前面展示过的带权重有向图 3-20 ：

```python
>>> from redis import Redis
>>> from graph import Graph
>>>
>>> client = Redis(decode_responses=True)
>>> graph = Graph(client, "test-graph")
>>>
>>> graph.add_edge("a", "b", 30)  # 添加边
>>> graph.add_edge("c", "b", 25)
>>> graph.add_multi_edges(("b", "d", 70), ("d", "e", 19))  # 添加多条边
```

然后通过执行程序提供的方法， 获取边的权重， 又或者检查给定的边是否存在：

```
>>> graph.get_edge_weight("a", "b")  # 获取边 a->b 的权重
'30'
>>> graph.has_edge("a", "b")         # 边 a->b 存在
True
>>> graph.has_edge("b", "a")         # 边 b->a 不存在
False
```

最后， 我们还可以获取图的所有边以及它们的权重：

```
>>> graph.get_all_edges()  # 获取所有边
{('b', 'd'), ('d', 'e'), ('a', 'b'), ('c', 'b')}
>>>
>>> graph.get_all_edges_with_weight()  # 获取所有边以及它们的权重
{('c', 'b', '25'), ('a', 'b', '30'), ('d', 'e', '19'), ('b', 'd', '70')}
```

这里展示的图数据储存程序提供了针对边和权重的功能， 因为它能够非常方便地向图中添加边和移除边， 并且还可以快速地检查某条边是否存在， 所以它非常适合用来储存节点较多但边较少的稀疏图（sparse graph） 。 在后续的章节中， 我们还会继续看到更多使用 Redis 储存图数据的例子。

3.18 示例：使用散列键重新实现文章储存程序

在稍早之前， 我们用散列重写了《字符串》一章介绍过的计数器程序， 但是除了计数器程序之外， 还有另一个程序也非常适合使用散列来重写， 那就是文章数据储存程序： 比起用多个字符串键来储存文章的各项数据， 更好的做法是把每篇文章的所有数据都储存到同一个散列里面， 代码清单 3-6 展示了这一想法的具体实现。

代码清单 3-6 使用散列实现的文章数据储存程序： `/hash/article.py`

```
from time import time

class Article:

    def __init__(self, client, article_id):
        self.client = client
        self.article_id = str(article_id)
        self.article_hash = "article::" + self.article_id

    def is_exists(self):
        """
        检查给定 ID 对应的文章是否存在。
```

```python
        """
        # 如果文章散列里面已经设置了标题，那么我们认为这篇文章存在
        return self.client.hexists(self.article_hash, "title")

    def create(self, title, content, author):
        """
        创建一篇新文章，创建成功时返回 True ，
        因为文章已经存在而导致创建失败时返回 False 。
        """
        # 文章已存在，放弃执行创建操作
        if self.is_exists():
            return False

        # 把所有文章数据都放到字典里面
        article_data = {
            "title": title,
            "content": content,
            "author": author,
            "create_at": time()
        }
        # redis-py 的 hmset() 方法接受一个字典作为参数，
        # 并根据字典内的键和值对散列的字段和值进行设置。
        return self.client.hmset(self.article_hash, article_data)

    def get(self):
        """
        返回文章的各项信息。
        """
        # hgetall() 方法会返回一个包含标题、内容、作者和创建日期的字典
        article_data = self.client.hgetall(self.article_hash)
        # 把文章 ID 也放到字典里面，以便用户操作
        article_data["id"] = self.article_id
        return article_data

    def update(self, title=None, content=None, author=None):
        """
        对文章的各项信息进行更新，
        更新成功时返回 True ，失败时返回 False 。
        """
        # 如果文章并不存在，那么放弃执行更新操作
        if not self.is_exists():
            return False

        article_data = {}
        if title is not None:
            article_data["title"] = title
        if content is not None:
            article_data["content"] = content
```

```
        if author is not None:
            article_data["author"] = author
        return self.client.hmset(self.article_hash, article_data)
```

新的文章储存程序除了会用到散列之外， 还有两个需要注意的地方：

1. 虽然 Redis 为字符串提供了 `MSET` 命令和 `MSETNX` 命令， 但是却并没有为散列提供 `HMSET` 命令对应的 `HMSETNX` 命令， 所以这个程序在创建一篇新文章之前， 需要先通过 `is_exists()` 方法检查文章是否存在， 然后再考虑是否使用 `HMSET` 命令去进行设置。
2. 在使用字符串键储存文章数据的时候， 为了避免数据库中出现键名冲突， 程序必须为每篇文章的每个属性都设置一个独一无二的键， 比如使用 `article::10086::title` 键去储存 ID 为 10086 的文章的标题， 使用 `article::12345::title` 键去储存 ID 为 12345 的文章的标题， 诸如此类。 相反地， 因为新的文章储存程序可以直接将一篇文章的所有相关信息都储存到同一个散列里面， 所以它可以直接在散列里面使用 `title` 作为标题的字段， 而不必担心出现命名冲突。

以下代码简单地展示了这个文章储存程序的使用方法：

```
>>> from redis import Redis
>>> from article import Article
>>>
>>> client = Redis(decode_responses=True)
>>> article = Article(client, 10086)
>>>
>>> # 创建文章
>>> article.create("greeting", "hello world", "peter")
>>>
>>> # 获取文章内容
>>> article.get()
{'content': 'hello world', 'id': '10086', 'created_at': '1442744762.631885', 'title': 'greeting', 'aut
>>>
>>> # 检查文章是否存在
>>> article.is_exists()
True
>>> # 更新文章内容
>>> article.update(content="good morning!")
>>> article.get()
{'content': 'good morning!', 'id': '10086',  'created_at': '1442744762.631885', 'title': 'greeting', '
```

图 3-22 以图形方式展示了这段代码创建的散列键。

图 3-22 储存在散列里面的文章数据

3.19 散列与字符串

本章从开头到现在， 陆续介绍了 HSET 、 HSETNX 、 HGET 、 HINCRBY 和 HINCRBYFLOAT 等多个散列命令， 如果读者对上一章介绍过的字符串命令还有印象的话， 那么应该会记得， 字符串也有类似的 SET 、 SETNX 、 GET 、 INCRBY 和 INCRBYFLOAT 命令。 这种相似并不是巧合， 正如表 3-1 所示， 散列的确拥有很多与字符串命令功能相似的命令。

表 3-1 字符串命令与类似的散列命令

字符串	散列
SET —— 为一个字符串键设置值。	HSET —— 为散列的给定字段设置值。
SETNX —— 仅在字符串键不存在的情况下为它设置值。	HSETNX —— 仅在散列不包含指定字段的情况下，为它设置值。
GET —— 获取字符串键的值。	HGET —— 从散列里面获取给定字段的值。
STRLEN —— 获取字符串值的字节长度。	HSTRLEN —— 获取给定字段值的字节长度。
INCRBY —— 对字符串键储存的数字值执行整数加法操作。	HINCRBY —— 对字段储存的数字值执行整数加法操作。

字符串	散列
INCRBYFLOAT —— 对字符串键储存的数字值执行浮点数加法操作。	HINCRBYFLOAT —— 对字段储存的数字值执行浮点数加法操作。
MSET —— 一次为多个字符串键设置值。	HMSET —— 一次为散列的多个字段设置值。
MGET —— 一次获取多个字符串键的值。	HMGET —— 一次获取散列中多个字段的值。
EXISTS —— 检查给定的键是否存在于数据库当中，这个命令可以用于包括字符串键在内的所有数据库键，本书稍后将在《数据库》一章对这个命令进行详细的介绍。	HEXISTS —— 检查给定字段是否存在于散列当中。
DEL —— 从数据库里面删除指定的键，这个命令可以用于包括字符串键在内的所有数据库键，本书稍后将在《数据库》一章对这个命令进行详细的介绍。	HDEL —— 从散列中删除给定字段，以及它的值。

对于表中列出的字符串命令和散列命令来说， 它们之间的最大区别就是前者处理的是字符串键， 而后者处理的则是散列键， 除此之外， 这些命令要做的事情几乎都是相同的。

Redis 之所以会选择同时提供字符串键和散列键这两种数据结构， 原因在于它们虽然在操作上非常相似， 但是各自却又拥有不同的优点， 这使得它们在某些场合无法被对方替代， 本节接下来将分别介绍这两种数据结构各自的优点。

3.19.1 散列键的优点

散列的最大优势， 就是它只需要在数据库里面创建一个键， 就可以把任意多的字段和值储存到散列里面。 相反地， 因为每个字符串键只能储存一个键值对， 所以如果用户要使用字符串键去储存多个数据项的话， 那么就只能在数据库里面创建多个字符串键。

图 3-23 展示了使用字符串键和散列键储存相同数量的数据项时， 数据库中创建的字符串键和散列键。

图 3-23 使用字符串键和散列键去储存相同数量的数据项

从图中可以看到， 为了储存相同的四个数据项， 程序需要用到四个字符串键， 又或者一个散列键。 按此计算， 如果我们需要储存一百万篇文章， 那么在使用散列键的情况下， 程序只需要在数据库里面创建一百万个散列键就可以了； 但是如果使用字符串键的话， 那么程序就需要在数据库里面创建四百万个字符串键。

数据库键数量增多带来的问题主要和资源有关：

1. 为了对数据库以及数据库键的使用情况进行统计， Redis 会为每个数据库键储存一些额外的信息， 并因此带来一些额外的内存消耗。 对于单个数据库键来说， 这些额外的内存消耗几乎可以忽略不计， 但是， 当数据库键的数量达到上百万、上千万甚至更多的时候， 这些额外的内存消耗就会变得比较可观。
2. 当散列包含的字段数量比较少的时候， Redis 就会使用特殊的内存优化结构去储存散列中的字段和值： 与字符串键相比， 这种内存优化结构储存相同数据所需的内存要少得多。 使用内存优化结构的散列越多， 内存优化结构带来的效果也就越明显。 在一定条件下， 对于相同的数据， 使用散列键进行储存比使用字符串键进行储存要节约一半以上的内存， 有时候甚至会更多。

3. 除了需要耗费更多内存之外， 更多的数据库键也需要占用更多的 CPU 。 每当 Redis 需要对数据库中的键进行处理时， 数据库包含的键越多， 进行处理所需的 CPU 资源就会越多， 处理所耗费的时间也会越长， 典型的情况包括：

    - 统计数据库和数据库键的使用情况；
    - 对数据库执行持久化操作， 又或者根据持久化文件还原数据库；
    - 通过模式匹配在数据库里面查找某个键， 或者执行类似的查找操作；

    这些操作的执行时间都会受到数据库键数量的影响。

最后， 除了资源方面的优势之外， 散列键还可以有效地组织起相关的多项数据， 让程序产生出更容易理解的数据， 使得针对数据的批量操作变得更为方便。 比如在上面展示的图 3-23 中， 使用散列键储存文章数据的做法就比使用字符串键储存文章数据的做法要来得更为清晰、易懂。

3.19.2 字符串键的优点

虽然使用散列键可以有效地节约资源并更好地组织数据， 但是字符串键也有自己的优点：

1. 虽然散列键命令和字符串命令在部分功能上有重合的地方， 但是字符串键命令提供的操作比散列键命令要更为丰富。 比如说， 字符串能够使用 SETRANGE 命令和 GETRANGE 命令设置或者读取字符串值的其中一部分， 又或者使用 APPEND 命令将新内容追加到字符串值的末尾， 而散列键并不支持这些操作。
2. 本书稍后将在《自动过期》一章对 Redis 的键过期功能进行介绍， 这一功能可以在指定时间到达时， 自动删除指定的键。 因为键过期功能针对的是整个键， 用户无法为散列中的不同字段设置不同的过期时间， 所以当一个散列键过期的时候， 它包含的所有字段和值都将被删除。 与此相反， 如果用户使用字符串键储存信息项的话， 就不会遇到这样的问题： 用户可以为每个字符串键分别设置不同的过期时间， 让它们根据实际的需要自动被删除掉。

3.19.3 字符串键和散列键的选择

表 3-2 从资源占用、支持的操作以及过期时间三个方面对比了字符串键和散列键的优缺点。

表 3-2 对比字符串键和散列键

比较的范畴	结果

比较的范畴	结果
资源占用	字符串键在数量较多的情况下，将占用大量的内存和 CPU 时间。 与此相反，将多个数据项储存到同一个散列里面可以有效地减少内存和 CPU 消耗。
支持的操作	散列键支持的所有命令，几乎都有相应的字符串键版本，但字符串键支持的 SETRANGE 、 GETRANGE 等操作散列键并不具备。
过期时间	字符串键可以为每个键单独设置过期时间，独立删除某个数据项；而散列一旦到期，它包含的所有字段和值都会被删除。

既然字符串键和散列键各有优点， 那么我们在构建应用程序的时候， 什么时候应该使用字符串键， 而什么时候又应该使用散列键呢？ 对于这个问题， 以下总结了一些选择的条件和方法：

1. 如果程序需要为每个数据项单独设置过期时间， 那么使用字符串键。
2. 如果程序需要对数据项执行诸如 SETRANGE 、 GETRANGE 或者 APPEND 等操作， 那么优先考虑使用字符串键。 当然， 用户也可以选择把数据储存在散列里面， 然后将类似 SETRANGE 、 GETRANGE 这样的操作交给客户端执行。
3. 如果程序需要储存的数据项比较多， 并且你希望尽可能地减少储存数据所需的内存， 那么就应该优先考虑使用散列键。
4. 如果多个数据项在逻辑上属于同一组或者同一类， 那么应该优先考虑使用散列键。

3.20 重点回顾

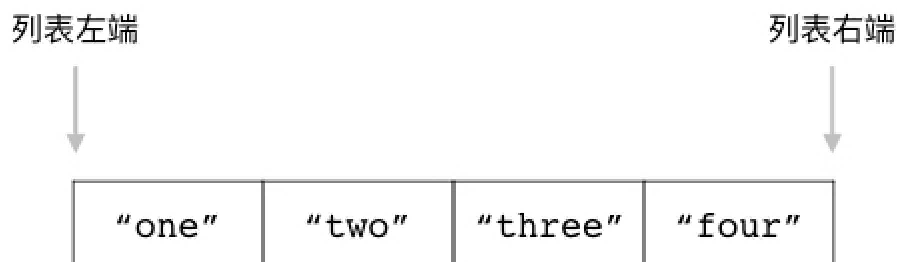
- 散列键会将一个键和一个散列在数据库里面关联起来， 用户可以在散列里面为任意多个字段设置值。 跟字符串键一样， 散列的字段和值既可以是文本数据， 也可以是二进制数据。
- 用户可以通过散列键把相关联的多项数据储存到同一个散列里面， 以便对其进行管理， 又或者针对它们执行批量操作。
- 因为 Redis 并没有为散列提供相应的减法操作命令， 所以如果用户想对字段储存的数字值执行减法操作的话， 那么就需要将负数增量传递给 HINCRBY 命令或 HINCRBYFLOAT 命令。
- Redis 散列包含的字段在底层是以无序方式储存的， 根据字段插入的顺序不同， 包含相同字段的散列在执行 HKEYS 、 HVALS 和 HGETALL 等命令时可能会得到不同的结果， 因此用户在使用这三个命令的时候， 不应该对命令返回元素的排列顺序作任何假设。
- 字符串键和散列键虽然在操作方式上非常相似， 但是因为它们都拥有各自独有的优点和缺点， 所以在一些情况下， 这两种数据结构是没有办法完全代替对方的。 因此用户在构建应用程序的时候， 应该根据自己的实际需要来选择使用相应的数据结构。

4. 列表（List）

Redis 的列表是一种线性的有序结构， 它可以按照元素被推入到列表中的顺序来储存元素， 这些元素既可以是文字数据， 又可以是二进制数据， 并且列表中的元素可以出现重复。

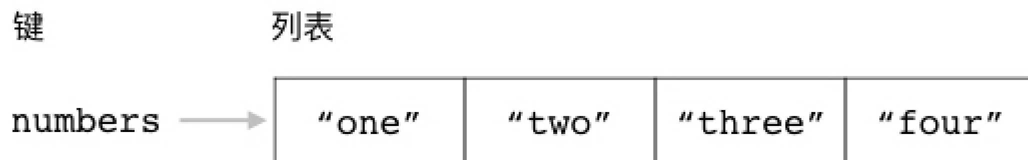
作为例子， 图 4-1 展示了一个包含多个字符串的列表， 这个列表按照从左到右的方式， 依次储存了 `"one"` 、 `"two"` 、 `"three"` 、 `"four"` 四个元素。

图 4-1 横向表示的列表



为了展示方便， 本书给出的列表图片一般都会像图 4-1 那样只展示列表本身而忽略列表的键名， 但是在有需要的时候， 本书也会像图 4-2 那样， 将列表及其键名一并给出。

图 4-2 完整的列表键视图



Redis 为列表提供了丰富的操作命令， 通过这些命令， 用户可以：

- 将新元素推入到列表的左端或者右端；
- 移除位于列表最左端或者最右端的元素；
- 移除列表最右端的元素， 然后把被移除的元素推入到另一个列表的左端；
- 获取列表包含的元素数量；
- 获取列表在指定索引上的单个元素， 又或者获取列表在指定索引范围内的多个元素；
- 为列表的指定索引设置新元素， 或者把新元素添加到某个指定元素的前面或者后面；
- 对列表进行修剪， 只保留指定索引范围内的元素；
- 从列表里面移除指定元素；
- 执行能够阻塞客户端的推入和移除操作；

本章接下来将对以上提到的各个列表操作命令进行介绍， 并说明如何使用这些命令去构建各种实用的程序。

4.1 LPUSH：将元素推入到列表左端

用户可以通过 `LPUSH` 命令， 将一个或多个元素推入到给定列表的左端：

```
LPUSH list item [item item ...]
```

在推入操作执行完毕之后， `LPUSH` 命令会返回列表当前包含的元素数量作为返回值。

比如以下代码就展示了如何通过 `LPUSH` 命令， 将 `"buy some milk"` 、 `"watch tv"` 、 `"finish homework"` 等元素依次推入到 `todo` 列表的左端：

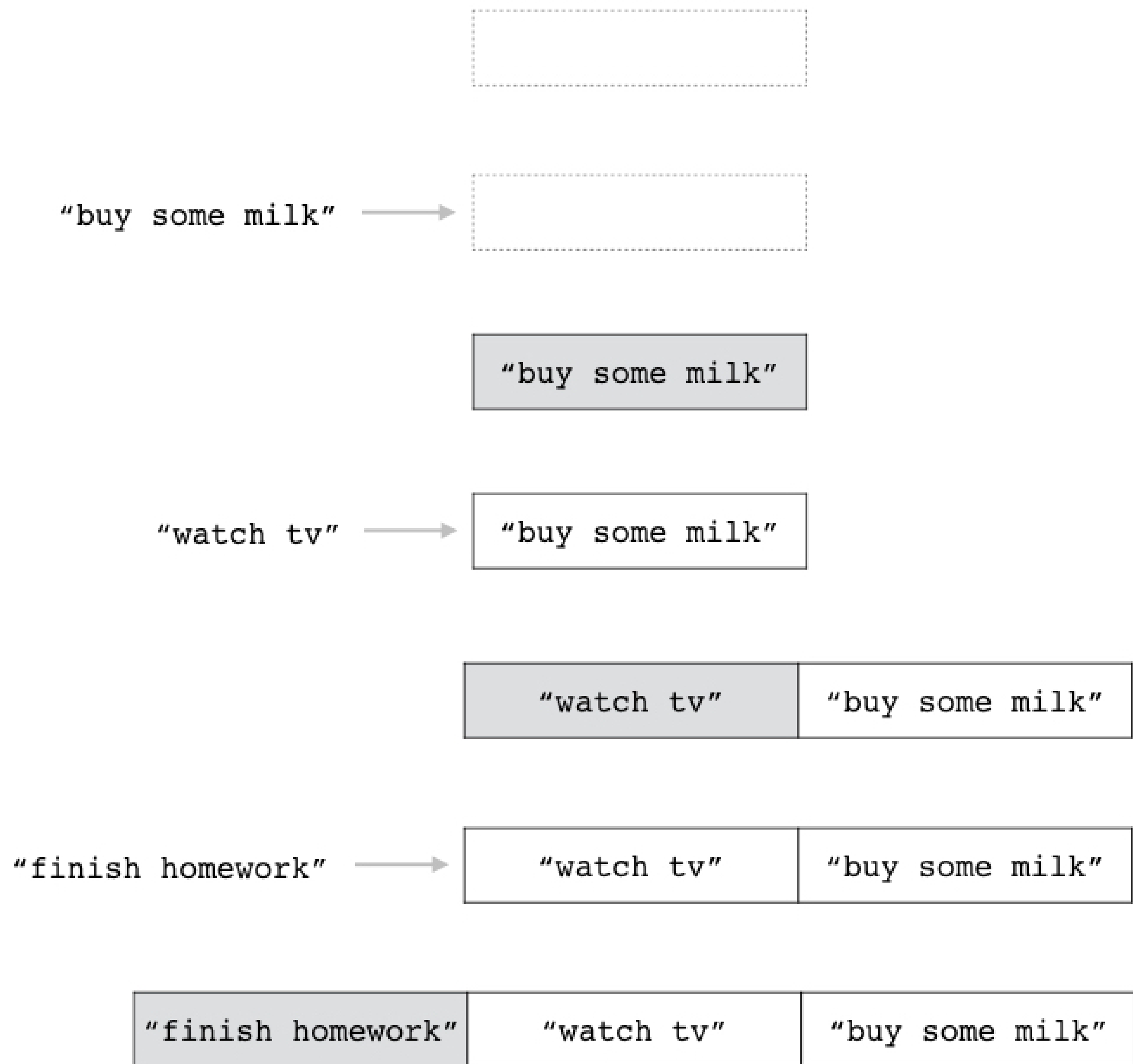
```
redis> LPUSH todo "buy some milk"
(integer) 1     -- 列表现在包含一个元素

redis> LPUSH todo "watch tv"
(integer) 2     -- 列表现在包含两个元素

redis> LPUSH todo "finish homework"
(integer) 3     -- 列表现在包含三个元素
```

图 4-3 展示了以上三个 `LPUSH` 命令的执行过程：

1. 首先，在操作执行之前， `todo` 列表为空，也即是不存在于数据库中；
2. 执行第一个 `LPUSH` 命令，将元素 `"buy some milk"` 推入到列表左端；

3. 执行完第一个 LPUSH 命令的列表现在包含一个元素；
4. 执行第二个 LPUSH 命令，将元素 "watch tv" 推入到列表左端；
5. 执行完第二个 LPUSH 命令的列表现在包含两个元素；
6. 执行第三个 LPUSH 命令，将元素 "finish homework" 推入到列表左端；
7. 执行完第三个 LPUSH 命令的 todo 列表现在包含三个元素。

图 4-3 LPUSH 命令执行过程



4.1.1 一次推入多个元素

LPUSH 命令允许用户一次将多个元素推入到列表左端： 如果用户在执行 LPUSH 命令时给定了多个元素， 那么 LPUSH 命令将按照元素给定的顺序， 从左到右依次地将所有给定元素推入到列表左端。

举个例子， 如果用户执行以下命令：

```
redis> LPUSH another-todo "buy some milk" "watch tv" "finish homework"
(integer) 3
```

那么 LPUSH 命令将按照图 4-4 所示的顺序， 将三个给定元素依次推入到 another-todo 列表的左端。

图 4-4 一次推入多个元素

首先推入这个元素

LPUSH another-todo “buy some milk” “watch tv” “finish homework”

接着推入这个元素

LPUSH another-todo “buy some milk” “watch tv” “finish homework”

最后推入这个元素

LPUSH another-todo “buy some milk” “watch tv” “finish homework”

最终， 这条 LPUSH 命令将产生图 4-5 所示的列表， 它和上一小节使用三条 LPUSH 命令构建出的列表完全相同。

图 4-5 another-todo 列表及其包含的元素

4.1.2 其他信息

属性	值
复杂度	O(N)，其中 N 为被推入到列表的元素数量。
版本要求	LPUSH 命令从 Redis 1.0.0 版本开始可用， 但是只有 Redis 2.4.0 或以上版本的 LPUSH 命令可以一次推入多个元素， Redis 2.4.0 以下版本的 LPUSH 命令每次只能推入一个元素。

4.2 RPUSH：将元素推入到列表右端

RPUSH 命令和 LPUSH 命令类似， 这两个命令执行的都是元素推入操作， 它们之间唯一的区别就在于 LPUSH 命令会将元素推入列表左端， 而 RPUSH 命令则会将元素推入列表右端：

```
RPUSH list item [item item ...]
```

在推入操作执行完毕之后， RPUSH 命令会返回列表当前包含的元素数量作为返回值。

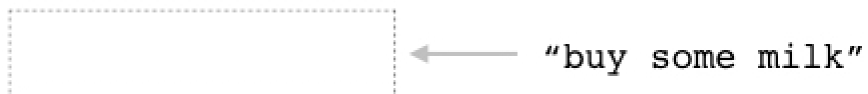
举个例子， 以下代码展示了如何通过 RPUSH 命令， 将 "buy some milk" 、 "watch tv" 、 "finish homework" 等元素依次推入到 todo 列表的右端：

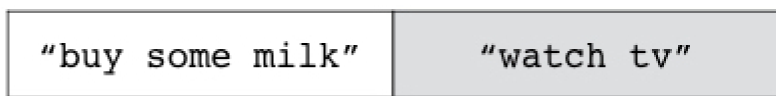
```
redis> RPUSH todo "buy some milk"
(integer) 1     -- 列表现在包含一个元素

redis> RPUSH todo "watch tv"
(integer) 2     -- 列表现在包含两个元素

redis> RPUSH todo "finish homework"
(integer) 3     -- 列表现在包含三个元素
```

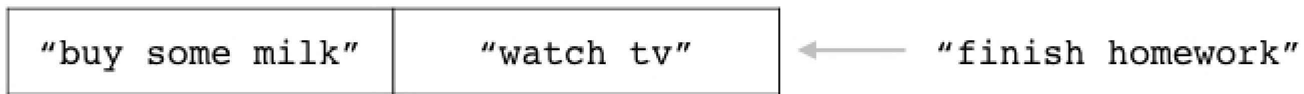
图 4-6 展示了以上三个 `RPUSH` 命令的执行过程：

1. 首先，在操作执行之前， `todo` 列表为空，也即是不存在于数据库中；
2. 执行第一个 `RPUSH` 命令， 将元素 `"buy some milk"` 推入到列表右端；
3. 执行完第一个 `RPUSH` 命令的列表现在包含一个元素；
4. 执行第二个 `RPUSH` 命令， 将元素 `"watch tv"` 推入到列表右端；
5. 执行完第二个 `RPUSH` 命令的列表现在包含两个元素；
6. 执行第三个 `RPUSH` 命令， 将元素 `"finish homework"` 推入到列表右端；
7. 执行完第三个 `RPUSH` 命令的 `todo` 列表现在包含三个元素。

图 4-6 `RPUSH` 命令执行过程







4.2.1 一次推入多个元素

跟 LPUSH 命令一样， RPUSH 命令也允许用户一次推入多个元素： 如果用户在执行 RPUSH 命令时给定了多个元素， 那么 RPUSH 命令将按照元素给定的顺序， 从左到右依次地将所有给定元素推入到列表右端。

举个例子， 如果用户执行以下命令：

```
redis> RPUSH another-todo "buy some milk" "watch tv" "finish homework"
(integer) 3
```

那么 RPUSH 命令将按照图 4-7 展示的顺序， 将三个给定元素依次推入到 another-todo 列表的右端。

图 4-7 一次推入多个元素

最终， 这条 RPUSH 命令将产生图 4-8 所示的列表， 这个列表和上一小节使用三条 RPUSH 命令构建出的列表完全相同。

图 4-8 another-todo 列表及其包含的元素

4.2.2 其他信息

属性	值
复杂度	O(N)，其中 N 为被推入到列表的元素数量。
版本要求	RPUSH 命令从 Redis 1.0.0 版本开始可用， 但是只有 Redis 2.4.0 或以上版本的 RPUSH 命令可以一次推入多个元素， Redis 2.4.0 以下版本的 RPUSH 命令每次只能推入一个元素。

4.3 LPUSHX、RPUSHX：只对已存在的列表执行推入操作

当用户调用 LPUSH 命令或是 RPUSH 命令尝试将元素推入到列表的时候， 如果给定的列表并不存在， 那么命令将自动创建一个空列表， 并将元素推入到刚刚创建的列表里面。

比如对于空列表 list1 和 list2 来说， 执行以下命令将创建图 4-9 所示的两个列表：

```
redis> LPUSH list1 "item1"
(integer) 1

redis> RPUSH list2 "item1"
(integer) 1
```

图 4-9 两个只包含单个元素的列表

列表 list1 列表 list2

"item1" "item1"

除了 LPUSH 命令和 RPUSH 命令之外， Redis 还提供了 LPUSHX 命令和 RPUSHX 命令：

```
LPUSHX list item

RPUSHX list item
```

这两个命令对待空列表的方式跟 LPUSH 命令和 RPUSH 命令正好相反：

- LPUSHX 命令只会在列表已经存在的情况下， 将元素推入列表左端；
- 而 RPUSHX 命令只会在列表已经存在的情况下， 将元素推入列表右端；

如果给定列表并不存在， 那么 LPUSHX 命令和 RPUSHX 命令将放弃执行推入操作。

LPUSHX 命令和 RPUSHX 命令在成功执行推入操作之后， 将返回列表当前的长度作为返回值； 如果推入操作未能成功执行， 那么命令将返回 0 作为结果。

举个例子， 如果我们对不存在的列表 list3 执行以下 LPUSHX 命令和 RPUSHX 命令， 那么这两个推入操作都将被拒绝：

```
redis> LPUSHX list3 "item-x"
(integer) 0     -- 没有推入任何元素

redis> RPUSHX list3 "item-y"
(integer) 0     -- 没有推入任何元素
```

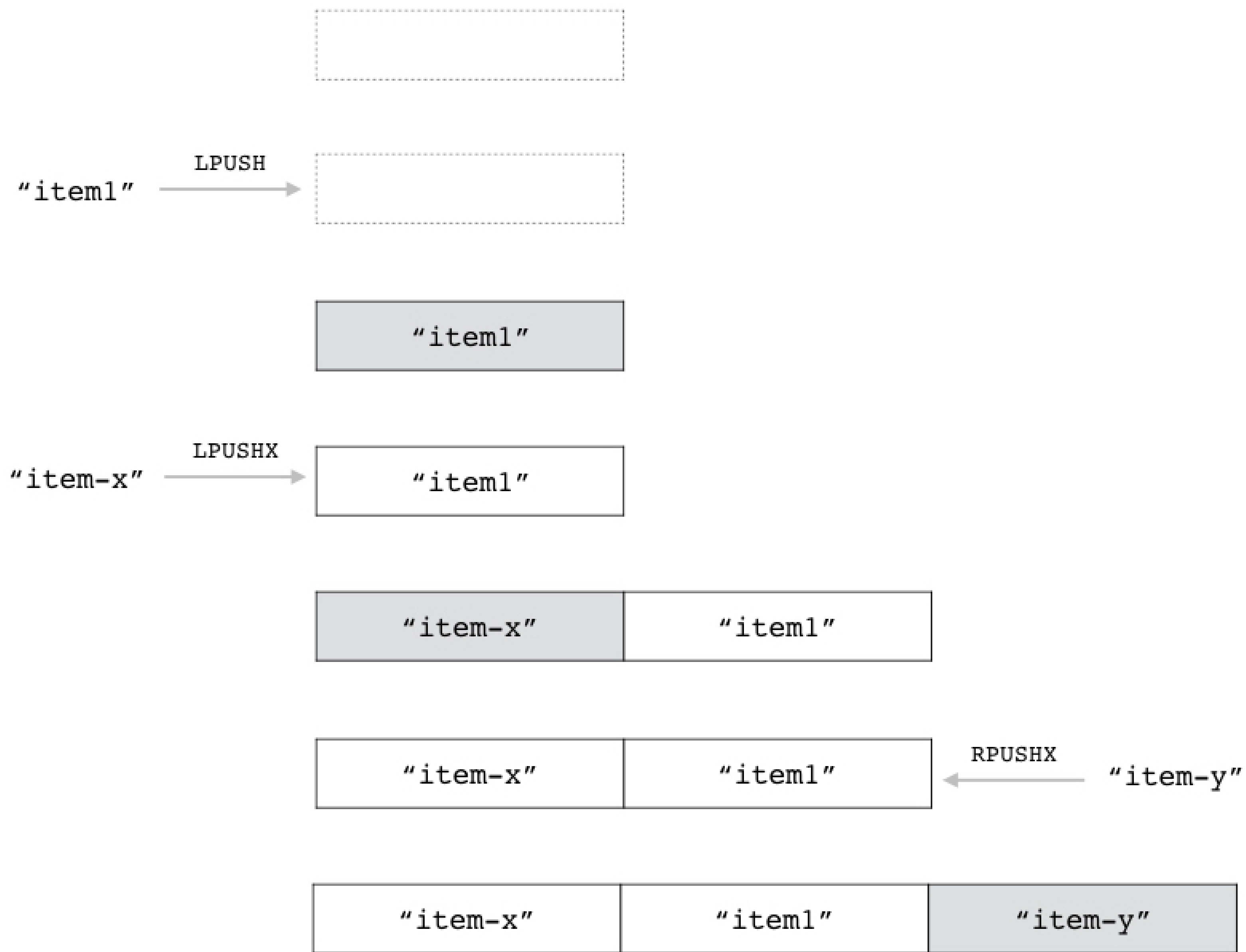
另一方面， 如果我们先使用 LPUSH 命令， 将一个元素推入到 list3 列表里面， 使得 list3 变成非空列表， 那么 LPUSHX 命令和 RPUSHX 命令就可以成功地执行推入操作：

```
redis> LPUSH list3 "item1"
(integer) 1    -- 推入一个元素，使列表变为非空

redis> LPUSHX list3 "item-x"
(integer) 2    -- 执行推入操作之后，列表包含两个元素

redis> RPUSHX list3 "item-y"
(integer) 3    -- 执行推入操作之后，列表包含三个元素
```

图 4–10 展示了列表 `list3` 的整个变化过程：

1. 在最初的 `LPUSHX` 命令和 `RPUSHX` 命令执行之后， `list3` 仍然是一个空列表。
2. 执行 `LPUSH` 命令， 将元素 `"item1"` 推入到列表里面， 使之变为非空。
3. 执行 `LPUSHX` 命令， 将元素 `"item-x"` 推入列表， 使得列表包含两个元素。
4. 执行 `RPUSHX` 命令， 将元素 `"item-y"` 推入列表， 使得列表包含三个元素。

图 4–10 `LPUSHX` 命令和 `RPUSHX` 命令的执行过程



4.3.1 每次只能推入单个元素

跟 LPUSH 命令和 RPUSH 命令不一样， LPUSHX 命令和 RPUSHX 命令每次只能推入一个元素， 尝试向 LPUSHX 命令或 RPUSHX 命令给定多个元素将引发错误：

```
redis> LPUSHX list "item1" "item2" "item3"
(error) ERR wrong number of arguments for 'lpushx' command

redis> RPUSHX list "item1" "item2" "item3"
(error) ERR wrong number of arguments for 'rpushx' command
```

4.3.2 其他信息

属性	值
复杂度	O(1)
版本要求	LPUSHX 命令和 RPUSHX 命令从 Redis 2.2.0 版本开始可用。

4.4 LPOP：弹出列表最左端的元素

用户可以通过 LPOP 命令移除位于列表最左端的元素， 并将被移除的元素返回给用户：

```
LPOP list
```

比如说， 以下代码就展示了如何使用 LPOP 命令去弹出 todo 列表的最左端元素：

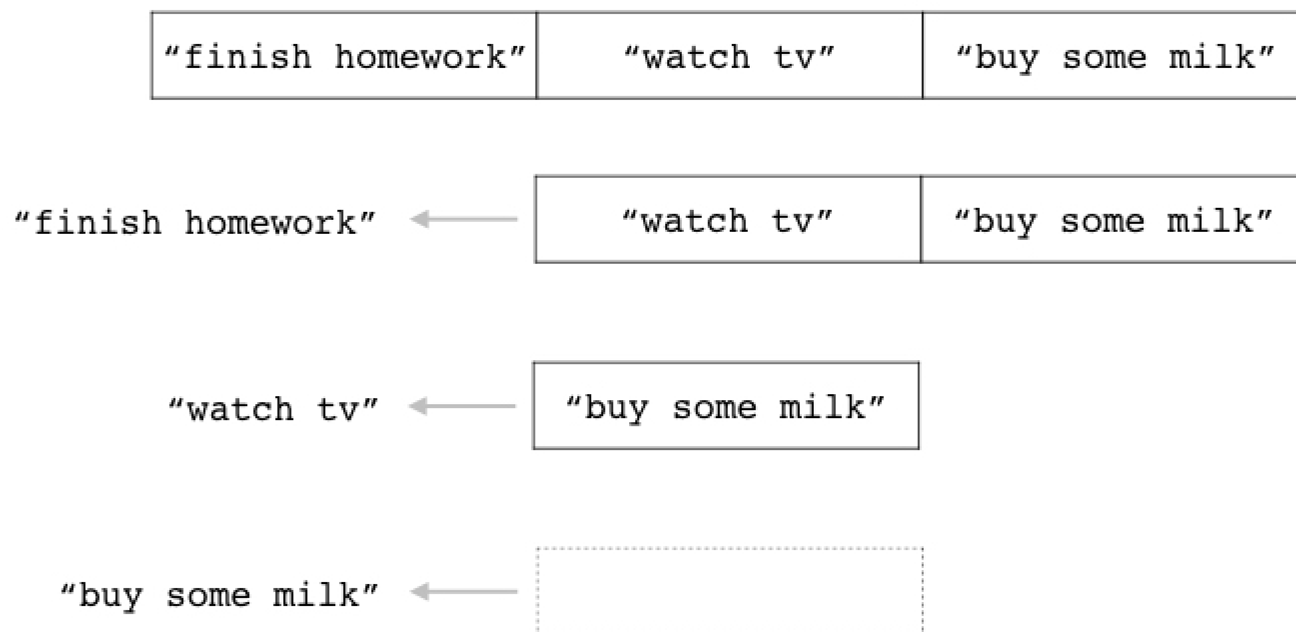
```
redis> LPOP todo
"finish homework"

redis> LPOP todo
"watch tv"

redis> LPOP todo
"buy some milk"
```

图 4-11 展示了 todo 列表在 LPOP 命令执行时的整个变化过程：

1. 在 LPOP 命令执行之前， todo 列表包含三个元素；

2. 执行第一个 LPOP 命令， 从列表中弹出 "finish homework" 元素；
3. 执行第二个 LPOP 命令， 从列表中弹出 "watch tv" 元素；
4. 执行第三个 LPOP 命令， 从列表中弹出 "buy some milk" 元素， 并使得 todo 列表变为空。

图 4-11 LPOP 命令的执行过程



另一方面， 如果用户给定的列表并不存在， 那么 LPOP 命令将返回一个空值， 表示列表为空， 没有元素可供弹出：

```
redis> LPOP empty-list
(nil)
```

4.4.1 其他信息

属性	值

属性	值
复杂度	O(1)
版本要求	LPOP 命令从 Redis 1.0.0 版本开始可用。

4.5 RPOP：弹出列表最右端的元素

用户可以通过 RPOP 命令移除位于列表最右端的元素， 并将被移除的元素返回给用户：

```
RPOP list
```

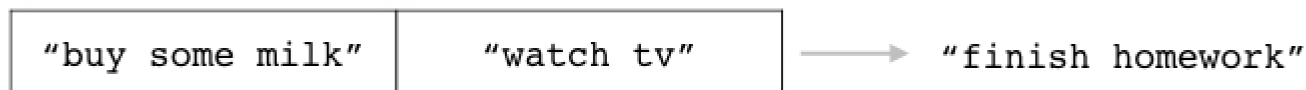
比如说， 以下代码就展示了如何使用 RPOP 命令去弹出 todo 列表最右端的元素：

```
redis> RPOP todo
"finish homework"

redis> RPOP todo
"watch tv"

redis> RPOP todo
"buy some milk"
```

图 4–12 展示了 todo 列表在 RPOP 命令执行时的整个变化过程：

1. 在 RPOP 命令执行之前， todo 列表包含三个元素；
2. 执行第一个 RPOP 命令， 从列表中弹出 "finish homework" 元素；
3. 执行第二个 RPOP 命令， 从列表中弹出 "watch tv" 元素；
4. 执行第三个 RPOP 命令， 从列表中弹出 "buy some milk" 元素， 并使得 todo 列表变为空。

图 4–12 RPOP 命令的执行过程



跟 LPOP 命令一样， 如果用户给定的列表并不存在， 那么 RPOP 命令将返回一个空值， 表示列表为空， 没有元素可供弹出：

```
redis> RPOP empty-list
(nil)
```

4.5.1 其他信息

属性	值
复杂度	O(1)
版本要求	RPOP 命令从 Redis 1.0.0 版本开始可用。

4.6 RPOPLPUSH：将右端弹出的元素推入到左端

RPOPLPUSH 命令的行为和它的名字一样， 它首先使用 RPOP 命令将源列表最右端的元素弹出， 然后使用 LPUSH 命令将被弹出的元素推入到目标列表左端， 使之成为目标列表的最左端元素：

```
RPOPLPUSH source target
```

RPOPLPUSH 命令会返回被弹出的元素作为结果。

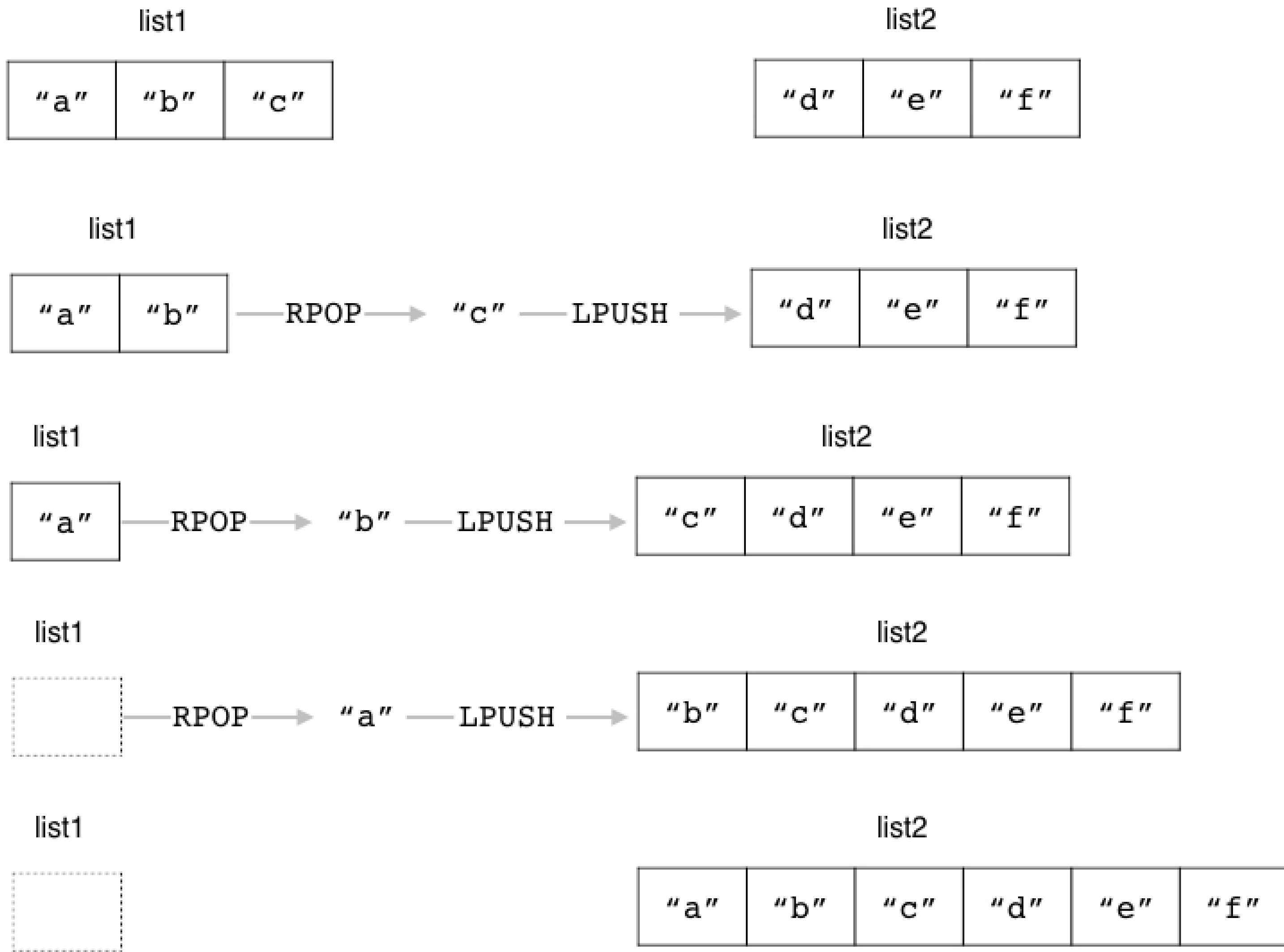
作为例子， 以下代码展示了如何使用 RPOPLPUSH 命令， 将列表 list1 的最右端元素弹出， 然后将其推入至列表 list2 的左端：

```
redis> RPUSH list1 "a" "b" "c"    -- 创建两个示例列表 list1 和 list2
(integer) 3

redis> RPUSH list2 "d" "e" "f"
(integer) 3

redis> RPOPLPUSH list1 list2
"c"

redis> RPOPLPUSH list1 list2
"b"

redis> RPOPLPUSH list1 list2
"a"
```

图 4-13 展示了列表 list1 和 list2 在执行以上 RPOPLPUSH 命令时的变化过程：

1. 在 RPOPLPUSH 命令执行之前， list1 和 list2 都包含三个元素；
2. 执行第一个 RPOPLPUSH 命令， 弹出 list1 的最右端元素 "c" ， 并将其推入至 list2 的左端；
3. 执行第二个 RPOPLPUSH 命令， 弹出 list1 的最右端元素 "b" ， 并将其推入至 list2 的左端；
4. 执行第三个 RPOPLPUSH 命令， 弹出 list1 的最右端元素 "a" ， 并将其推入至 list2 的左端；
5. 在以上三个 RPOPLPUSH 命令执行完毕之后， list1 将变为空列表， 而 list2 则会包含六个元素。

图 4-13 RPOPLPUSH 命令的执行过程



4.6.1 源列表和目标列表相同

RPOPLPUSH 命令允许用户将源列表和目标列表设置为同一个列表： 在这种情况下， RPOPLPUSH 命令的效果就相当于将列表最右端的元素变成列表最左端的元素。

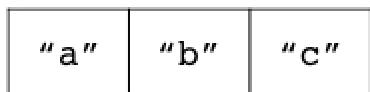
比如以下代码就展示了如何通过 RPOPLPUSH 命令， 将 rotate-list 列表的最右端元素变成列表的最左端元素：

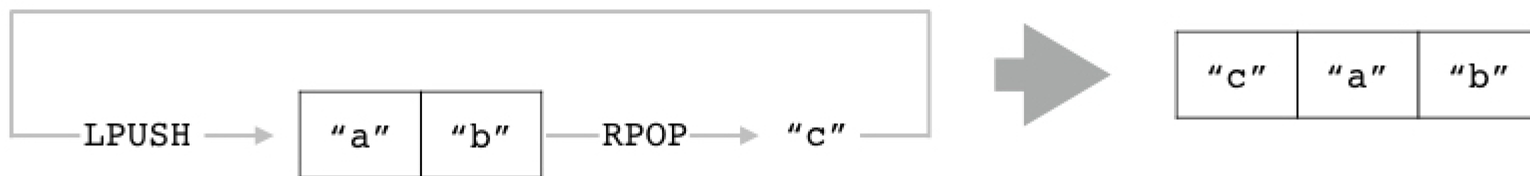
```
redis> RPUSH rotate-list "a" "b" "c"     -- 创建一个示例列表
(integer) 3

redis> RPOPLPUSH rotate-list rotate-list
"c"

redis> RPOPLPUSH rotate-list rotate-list
"b"

redis> RPOPLPUSH rotate-list rotate-list
"a"
```

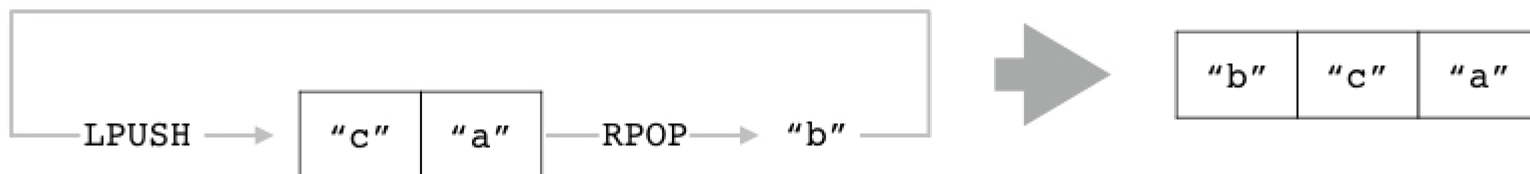
图 4-14 展示了以上三个 `RPOPLPUSH` 命令在执行时， `rotate-list` 列表的整个变化过程：

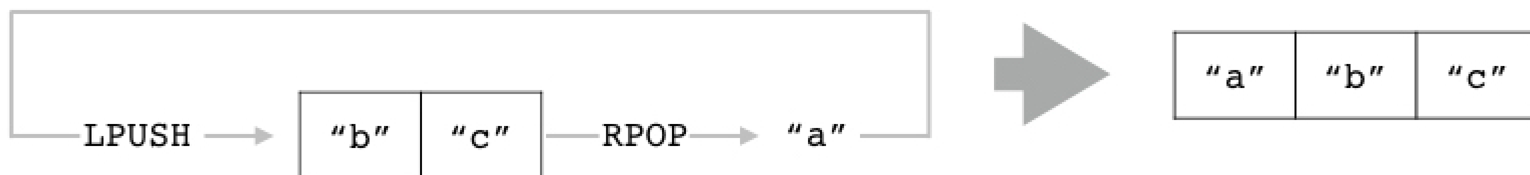
1. 在 `RPOPLPUSH` 命令执行之前，列表包含 `"a"` 、 `"b"` 、 `"c"` 三个元素。
2. 执行第一个 `RPOPLPUSH` 命令，将最右端元素 `"c"` 变为最左端元素。
3. 执行第二个 `RPOPLPUSH` 命令，将最右端元素 `"b"` 变为最左端元素。
4. 执行第三个 `RPOPLPUSH` 命令，将最右端元素 `"a"` 变为最左端元素。
5. 在以上三个 `RPOPLPUSH` 命令执行完毕之后，列表又重新回到了原样。

图 4-14 使用 `RPOPLPUSH` 对列表元素进行轮换









正如上面展示的例子所示， 通过对同一个列表重复执行 RPOPLPUSH 命令， 我们可以创建出一个对元素进行轮换的列表； 并且当我们对一个包含了 N 个元素的列表重复执行 N 次 RPOPLPUSH 命令之后， 列表元素的排列顺序将变回原来的样子。

4.6.2 处理空列表

如果用户传给 RPOPLPUSH 命令的源列表并不存在， 那么 RPOPLPUSH 命令将放弃执行弹出和推入操作， 只返回一个空值表示命令执行失败：

```
redis> RPOPLPUSH list-x list-y
(nil)
```

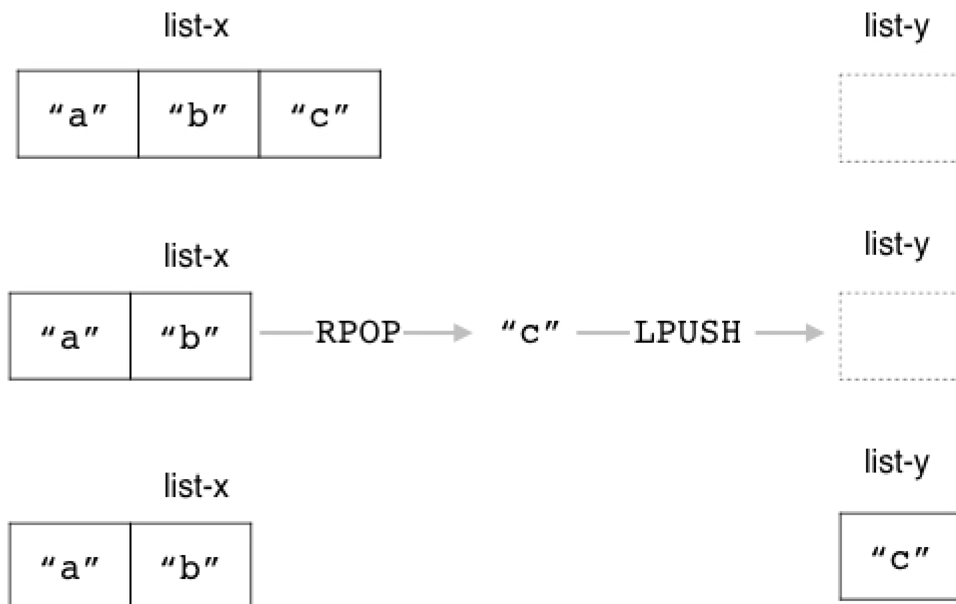
另一方面， 如果源列表非空， 但是目标列表为空， 那么 RPOPLPUSH 命令将正常执行弹出操作和推入操作：

```
redis> RPUSH list-x "a" "b" "c"     -- 将 list-x 变为非空列表
(integer) 3

redis> RPOPLPUSH list-x list-y
"c"
```

图 4-15 展示了这条 RPOPLPUSH 命令执行之前和执行之后， list-x 和 list-y 的变化：

1. 在执行 RPOPLPUSH 命令之前， list-x 包含三个元素， 而 list-y 为空。
2. 执行 RPOPLPUSH 命令， 将 list-x 的最右端元素 "c" 弹出， 并将其推入至 list-y 的左端。
3. 在 RPOPLPUSH 命令执行完毕之后， list-x 将包含两个元素， 而 list-y 则包含一个元素。

图 4-15 RPOPLPUSH 命令处理目标列表为空的例子



4.6.3 其他信息

属性	值
复杂度	O(1)
版本要求	RPOPLPUSH 命令从 Redis 1.2.0 版本开始可用。

4.7 示例：先进先出队列

先进先出队列（first in first out queue）是一种非常常见的数据结构， 一般都会包含入队（enqueue）和出队（dequeue）这两个操作， 其中入队操作会将一个元素放入到队列中， 而出队操作则会从队列中移除最先被入队的元素。

先进先出队列的应用非常广泛， 它见诸于各式各样的应用程序当中。 举个例子， 很多电商网站都会在节日时推出一些秒杀活动， 这些活动会放出数量有限的商品供用户抢购， 秒杀系统的一个特点就是在短时间内会有大量用户同时进行相同的购买操作， 如果使用事务或者锁去实现秒杀程序， 那么就会因为锁和事务的重试特性而导致性能低下， 并且由于重试的存在， 成功购买商品的用户可能并不是最早执行购买操作的用户， 因此这种秒杀系统实际上是不公平的。

解决上述问题的其中一个方法就是把用户的购买操作都放入到先进先出队列里面， 然后以队列方式处理用户的购买操作， 这样程序就可以在不使用锁或者事务的情况下实现秒杀系统， 并且得益于先进先出队列的特性， 这种秒杀系统可以按照用户执行购买操作的顺序来判断哪些用户可以成功执行购买操作， 因此它是公平的。

代码清单 4-1 展示了一个使用 Redis 列表实现先进先出队列的方法。

代码清单 4-1 使用列表实现的先进先出队列：/list/fifo_queue.py

```
class FIFOqueue:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def enqueue(self, item):
        """
        将给定元素放入队列，然后返回队列当前包含的元素数量作为结果。
```

```python
        """
        return self.client.rpush(self.key, item)

    def dequeue(self):
        """
        移除并返回队列目前入队时间最长的元素。
        """
        return self.client.lpop(self.key)
```

作为例子， 我们可以通过执行以下代码， 载入并创建一个先进先出队列：

```python
>>> from redis import Redis
>>> from fifo_queue import FIFOqueue
>>> client = Redis(decode_responses=True)
>>> q = FIFOqueue(client, "buy-request")
```

然后通过执行以下代码， 将三个用户的购买请求依次放入到队列里面：

```python
>>> q.enqueue("peter-buy-milk")
1
>>> q.enqueue("john-buy-rice")
2
>>> q.enqueue("david-buy-keyboard")
3
```

最后， 按照先进先出顺序， 依次从队列中弹出相应的购买请求：

```python
>>> q.dequeue()
'peter-buy-milk'
>>> q.dequeue()
'john-buy-rice'
>>> q.dequeue()
'david-buy-keyboard'
```

可以看到， 队列弹出元素的顺序跟元素入队时的顺序是完全相同的： 最先是 `"peter-buy-milk"` 元素， 接着是 `"john-buy-rice"` 元素， 最后是 `"david-buy-keyboard"` 元素。

4.8 LLEN：获取列表的长度

用户可以通过执行 `LLEN` 命令来获取列表的长度， 也即是列表包含的元素数量：

```
LLEN list
```

图 4-16 几个不同长度的列表

todo 列表

“finish homework”	“watch tv”	“buy some milk”

alphabets 列表

“a”	“b”	“c”	“d”	“e”	“f”	“g”	“h”

msg-queue 列表

“job::3324”	“job::5121”	“job::5497”	“job::6072”

比如对于图 4-16 所示的几个列表来说， 对它们执行 `LLEN` 命令将获得以下结果：

```
redis> LLEN todo
(integer) 3

redis> LLEN alphabets
(integer) 8

redis> LLEN msg-queue
(integer) 4
```

另一方面， 对于不存在的列表， `LLEN` 命令将返回 `0` 作为结果：

```
redis> LLEN not-exists-list
(integer) 0
```

4.8.1 其他信息

属性	值
复杂度	O(1)
版本要求	`LLEN` 命令从 Redis 1.0.0 版本开始可用。

4.9 LINDEX：获取指定索引上的元素

Redis 列表包含的每个元素都有与之相对应的正数索引和负数索引：

- 正数索引从列表的左端开始计算， 依次向右端递增： 最左端元素的索引为 `0` ， 左端排行第二的元素索引为 `1` ， 左端排行第三的元素索引为 `2` ， 以此类推。 最大的正数索引为列表长度减一， 也即是 `N-1` 。
- 负数索引从列表的右端开始计算， 依次向左端递减： 最右端元素的索引为 `-1` ， 右端排行第二的元素索引为 `-2` ， 右端排行第三的元素索引为 `-3` ， 以此类推。 最大的负数索引为列表长度的负数， 也即是 `-N` 。

作为例子， 图 4-17 展示了一个包含多个元素的列表， 并给出了列表元素对应的正数索引和负数索引。

图 4-17 列表的索引

正数索引	0	1	2	3	4	5	6	7
负数索引	-8	-7	-6	-5	-4	-3	-2	-1
	"a"	"b"	"c"	"d"	"e"	"f"	"g"	"h"

alphabets 列表

为了让用户可以方便地取得索引对应的元素， Redis 提供了 `LINDEX` 命令：

```
LINDEX list index
```

这个命令接受一个列表和一个索引作为参数， 然后返回列表在给定索引上的元素； 其中给定索引既可以是正数， 也可以是负数。

比如说， 对于前面展示的图 4-17 ， 我们可以通过执行以下命令， 取得 `alphabets` 列表在指定索引上的元素：

```
redis> LINDEX alphabets 0
"a"

redis> LINDEX alphabets 3
"d"

redis> LINDEX alphabets 6
"g"

redis> LINDEX alphabets -3
"f"

redis> LINDEX alphabets -7
"b"
```

4.9.1 处理超出范围的索引

对于一个长度为 `N` 的非空列表来说：

- 它的正数索引必然大于等于 `0` ， 并且少于等于 `N-1` ；
- 而它的负数索引则必然小于等于 `-1` ， 并且大于等于 `-N` ；

如果用户给定的索引超出了这一范围， 那么 `LINDEX` 命令将返回空值， 以此来表示给定索引上并不存在任何元素：

```
redis> LINDEX alphabets 100
(nil)
```

```
redis> LINDEX alphabets -100
(nil)
```

4.9.2 其他信息

属性	值
复杂度	O(N)， 其中 N 为给定列表的长度。
版本要求	`LINDEX` 命令从 Redis 1.0.0 版本开始可用。

4.10 LRANGE：获取指定索引范围上的元素

用户除了可以使用 `LINDEX` 命令获取给定索引上的单个元素之外， 还可以使用 `LRANGE` 命令获取给定索引范围上的多个元素：

```
LRANGE list start end
```

`LRANGE` 命令接受一个列表、一个开始索引和一个结束索引作为参数， 然后依次返回列表从开始索引到结束索引范围内的所有元素， 其中开始索引和结束索引对应的元素也会被包含在命令返回的元素当中。

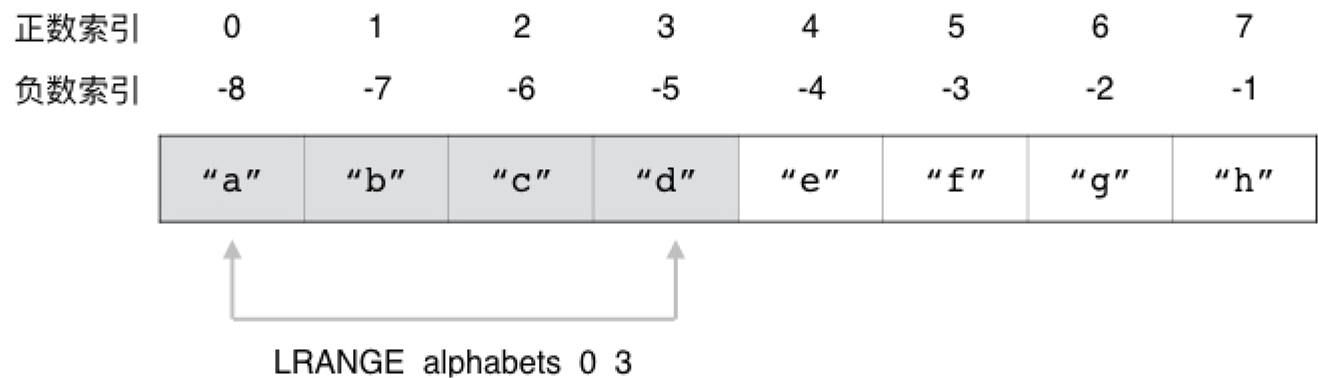
作为例子， 以下代码展示了如何使用 `LRANGE` 命令去获取 `alphabets` 列表在不同索引范围内的元素：

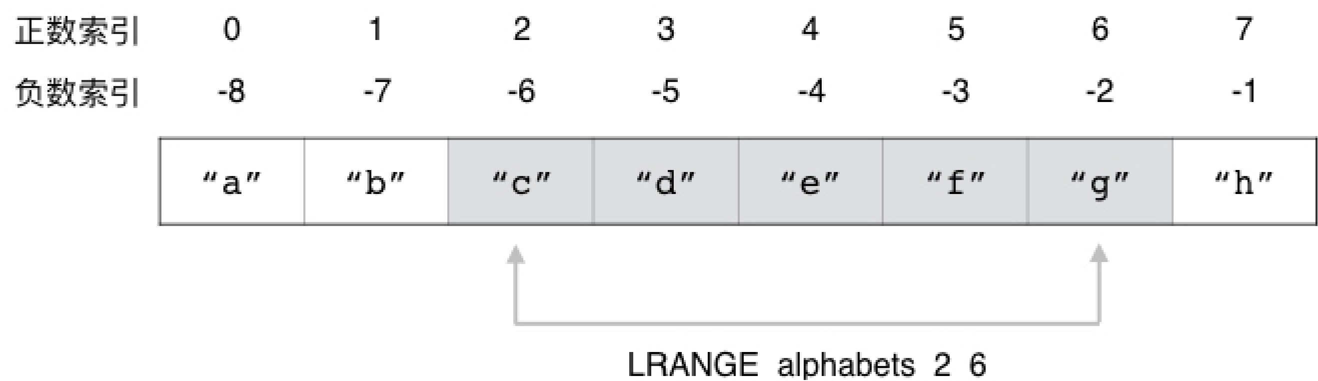
```
redis> LRANGE alphabets 0 3    -- 获取列表索引 0 至索引 3 上的所有元素
1) "a"    -- 位于索引 0 上的元素
2) "b"    -- 位于索引 1 上的元素
3) "c"    -- 位于索引 2 上的元素
4) "d"    -- 位于索引 3 上的元素

redis> LRANGE alphabets 2 6
1) "c"
2) "d"
3) "e"
4) "f"
5) "g"

redis> LRANGE alphabets -5 -1
1) "d"
2) "e"
```

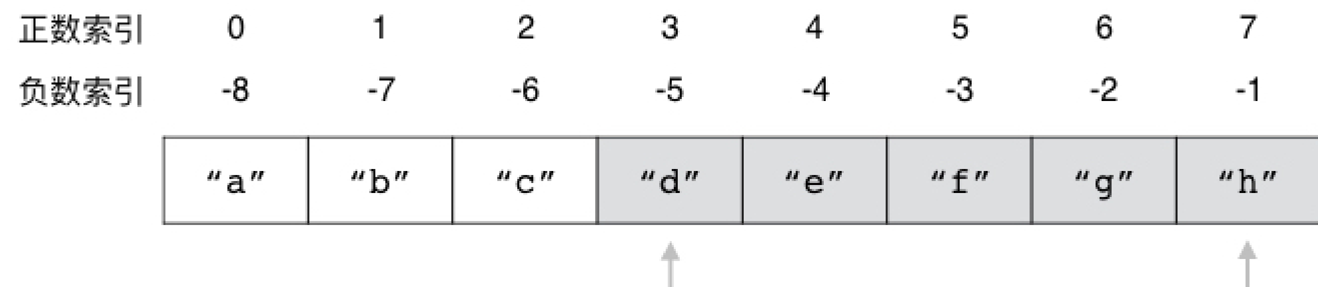
```
3) "f"
4) "g"
5) "h"

redis> LRANGE alphabets -7 -4
1) "b"
2) "c"
3) "d"
4) "e"
```

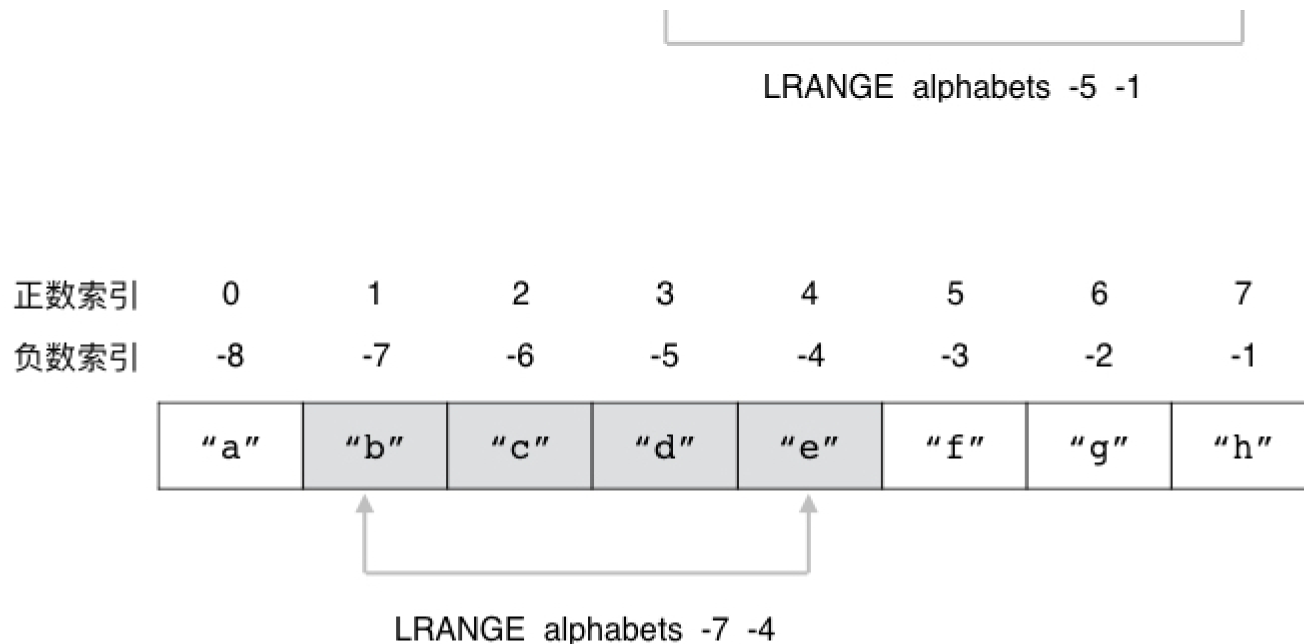
图 4-18 展示了这些 LRANGE 命令是如何根据给定的索引范围去获取列表元素的。

图 4-18 LRANGE 命令获取范围内元素的过程


正数索引 0 1 2 3 4 5 6 7
负数索引 -8 -7 -6 -5 -4 -3 -2 -1
"a" "b" "c" "d" "e" "f" "g" "h"
LRANGE alphabets 0 3


正数索引 0 1 2 3 4 5 6 7
负数索引 -8 -7 -6 -5 -4 -3 -2 -1
"a" "b" "c" "d" "e" "f" "g" "h"
LRANGE alphabets 2 6


正数索引 0 1 2 3 4 5 6 7
负数索引 -8 -7 -6 -5 -4 -3 -2 -1
"a" "b" "c" "d" "e" "f" "g" "h"



4.10.1 获取列表包含的所有元素

一个快捷地获取列表包含的所有元素的方法， 就是使用 `0` 作为起始索引、 `-1` 作为结束索引去调用 `LRANGE` 命令， 这种方法非常适用于查看长度较短的列表：

```
redis> LRANGE alphabets 0 -1
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
7) "g"
8) "h"
```

4.10.2 处理超出范围的索引

跟 `LINDEX` 一样， `LRANGE` 命令也需要处理超出范围的索引：

- 如果用户给定的起始索引和结束索引都超出了范围， 那么 LRANGE 命令将返回空列表作为结果；
- 如果用户给定的其中一个索引超出了范围， 那么 LRANGE 命令将对超出范围的索引进行修正， 然后再执行实际的范围获取操作； 其中超出范围的起始索引会被修正为 0 ， 而超出范围的结束索引则会被修正为 -1 。

以下代码展示了 LRANGE 命令在遇到两个超出范围的索引时， 返回空列表的例子：

```
redis> LRANGE alphabets 50 100
(empty list or set)

redis> LRANGE alphabets -100 -50
(empty list or set)
```

另一方面， 以下代码展示了 LRANGE 命令在遇到只有一个超出范围的索引时， 对索引进行修正并返回元素的例子：

```
redis> LRANGE alphabets -100 5
1) "a"  -- 位于索引 0 上的元素
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"  -- 位于索引 5 上的元素

redis> LRANGE alphabets 5 100
1) "f"  -- 位于索引 5 上的元素
2) "g"
3) "h"  -- 位于索引 -1 上的元素
```

在执行 LRANGE alphabets -100 5 调用时， LRANGE 命令会把超出范围的起始索引 -100 修正为 0 ， 然后执行 LRANGE alphabets 0 5 调用； 而在执行 LRANGE alphabets 5 100 调用时， LRANGE 命令会把超出范围的结束索引 100 修正为 -1 ， 然后执行 LRANGE alphabets 5 -1 调用。

4.10.3 其他信息

属性	值
复杂度	O(N)，其中 N 为给定列表的长度。
版本要求	LRANGE 命令从 Redis 1.0.0 开始可用。

4.11 示例：分页

对于互联网上每一个具有一定规模的网站来说， 分页程序都是必不可少的： 新闻站点、博客、论坛、搜索引擎等等， 都会使用分页程序将数量众多的信息分割为多个页面， 使得用户可以以页为单位浏览网站提供的信息， 并以此来控制网站每次取出的信息数量。 图 4-19 就展示了一个使用分页程序对用户发表的论坛主题进行分割的例子。

图 4-19 论坛中的分页示例

代码清单 4-2 展示了一个使用列表实现分页程序的方法， 这个程序可以将给定的元素有序地放入到一个列表里面， 然后使用 LRANGE 命令从列表中取出指定数量的元素， 从而实现分页这一概念。

代码清单 4-2 使用列表实现的分页程序： `/list/paging.py`

```
class Paging:
```

```
    def __init__(self, client, key):
        self.client = client
        self.key = key

    def add(self, item):
        """
        将给定元素添加到分页列表中。
        """
        self.client.lpush(self.key, item)

    def get_page(self, page_number, item_per_page):
        """
        从指定页数中取出指定数量的元素。
        """
        # 根据给定的 page_number （页数）和 item_per_page （每页包含的元素数量）
        # 计算出指定分页元素在列表中所处的索引范围
        # 例子：如果 page_number = 1 ， item_per_page = 10
        # 那么程序计算得出的起始索引就是 0 ，而结束索引则是 9
        start_index = (page_number - 1) * item_per_page
        end_index = page_number * item_per_page - 1
        # 根据索引范围从列表中获取分页元素
        return self.client.lrange(self.key, start_index, end_index)

    def size(self):
        """
        返回列表目前包含的分页元素数量。
        """
        return self.client.llen(self.key)
```

作为例子， 我们可以通过执行以下代码， 载入并创建出一个针对用户帖子的分页对象：

```
>>> from redis import Redis
>>> from paging import Paging
>>> client = Redis(decode_responses=True)
>>> topics = Paging(client, "user-topics")
```

并使用数字 `1` 至 `19` 作为用户帖子的 ID ， 将它们添加到分页列表里面：

```
>>> for i in range(1, 20):
...   topics.add(i)
...
```

然后我们就可以使用分页程序， 对这些帖子进行分页了：

```
>>> topics.get_page(1, 5)    # 以每页 5 个帖子的方式，取出第 1 页的帖子
['19', '18', '17', '16', '15']
>>> topics.get_page(2, 5)    # 以每页 5 个帖子的方式，取出第 2 页的帖子
['14', '13', '12', '11', '10']
>>> topics.get_page(1, 10)  # 以每页 10 个帖子的方式，取出第 1 页的帖子
['19', '18', '17', '16', '15', '14', '13', '12', '11', '10']
```

最后， 我们可以通过执行以下代码， 取得分页列表目前包含的元素数量：

```
>>> topics.size()
19
```

4.12 LSET：为指定索引设置新元素

用户可以通过 LSET 命令， 为列表的指定索引设置新元素：

```
LSET list index new_element
```

LSET 命令在设置成功时将返回 OK 。

比如对于以下这个 todo 列表来说：

```
redis> LRANGE todo 0 -1
1) "buy some milk"
2) "watch tv"
3) "finish homework"
```

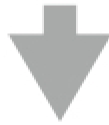
我们可以通过执行以下 LSET 命令， 将 todo 列表索引 1 上的元素设置为 "have lunch" ：

```
redis> LSET todo 1 "have lunch"
OK

redis> LRANGE todo 0 -1
1) "buy some milk"
2) "have lunch"   -- 新元素
3) "finish homework"
```

图 4-20 展示了这个 LSET 命令的执行过程。

图 4-20 LSET 命令的执行过程

定位索引 1



索引	0	1	2
	“buy some milk”	“watch tv”	“finish homework”

索引	0	1	2
	“buy some milk”	“watch tv”	“finish homework”

将其替换为新元素
“have lunch”

索引	0	1	2
	“buy some milk”	“have lunch”	“finish homework”

4.12.1 处理超出范围的索引

因为 LSET 命令只能对列表已存在的索引进行设置， 所以如果用户给定的索引超出了列表的有效索引范围， 那么 LSET 命令将返回一个错误：

```
redis> LSET todo 100 "go to sleep"
(error) ERR index out of range
```

4.12.2 其他信息

属性	值
复杂度	O(N)，其中 N 为给定列表的长度。
版本要求	LSET 命令从 Redis 1.0.0 版本开始可用。

4.13 LINSERT：将元素插入到列表

通过使用 LINSERT 命令， 用户可以将一个新元素插入到列表某个指定元素的前面或者后面：

```
LINSERT list BEFORE|AFTER target_element new_element
```

LINSERT 命令第二个参数的值可以是 BEFORE 或者 AFTER ， 它们分别用于指示命令将新元素插入到目标元素的前面或者后面； 命令在完成插入操作之后会返回列表当前的长度作为返回值。

比如说， 对于以下这个 lst 列表：

```
redis> LRANGE lst 0 -1
1) "a"
2) "b"
3) "c"
```

我们可以通过执行以下 LINSERT 命令， 将元素 "10086" 插入到元素 "b" 的前面：

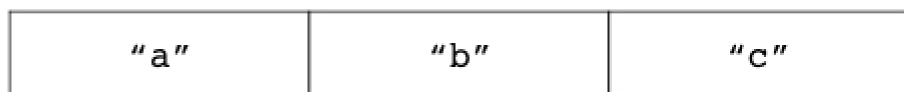
```
redis> LINSERT lst BEFORE "b" "10086"
(integer) 4

redis> LRANGE lst 0 -1
1) "a"
2) "10086"
3) "b"
4) "c"
```

还可以通过执行以下 LINSERT 命令， 将元素 "12345" 插入到元素 "c" 的后面：

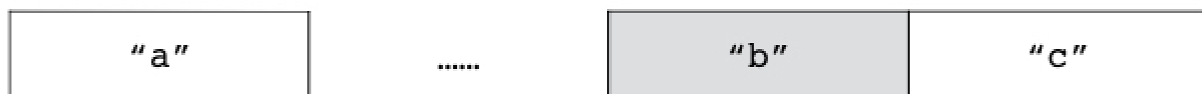
```
redis> LINSERT lst AFTER "c" "12345"
(integer) 5

redis> LRANGE lst 0 -1
1) "a"
2) "10086"
3) "b"
4) "c"
5) "12345"
```

图 4-21 展示了上述两个 LINSERT 命令的执行过程。

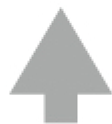
图 4-21 LINSERT 命令的执行过程



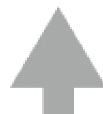
找到元素 “b”



将新元素 “10086” 插入到元素 “b” 的前面



找到元素 “c”



将新元素 “12345” 插入到元素 “c” 的后面

“a”	“10086”	“b”	“c”	“12345”

4.13.1 处理不存在的元素

为了执行插入操作， LINSERT 命令要求用户给定的目标元素必须已经存在于列表当中； 相反地， 如果用户给定的目标元素并不存在， 那么 LINSERT 命令将返回 -1 表示插入失败：

```
redis> LINSERT lst BEFORE "not-exists-element" "new element"
(integer) -1
```

在插入操作执行失败的情况下， 列表包含的元素将不会发生任何变化。

4.13.2 其他信息

属性	值
复杂度	O(N) ，其中 N 为给定列表的长度。
版本要求	LINSERT 命令从 Redis 2.2.0 版本开始可用。

4.14 LTRIM：修剪列表

LTRIM 命令接受一个列表和一个索引范围作为参数， 它会移除列表中位于给定索引范围之外的所有元素， 只保留给定范围之内的元素：

```
LTRIM list start end
```

LTRIM 命令在移除操作执行完毕之后将返回 OK 作为结果。

比如对于以下这个 alphabets 列表来说：

```
redis> RPUSH alphabets "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
(integer) 11
```

执行以下命令可以让列表只保留索引 0 到索引 6 范围内的七个元素：

```
redis> LTRIM alphabets 0 6
OK

redis> LRANGE alphabets 0 -1
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
7) "g"
```

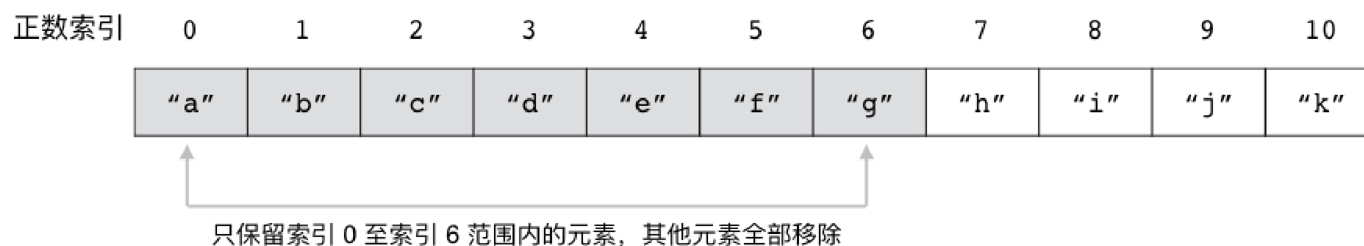
在此之后， 我们可以继续执行以下命令， 让列表只保留索引 3 到索引 5 范围内的三个元素：

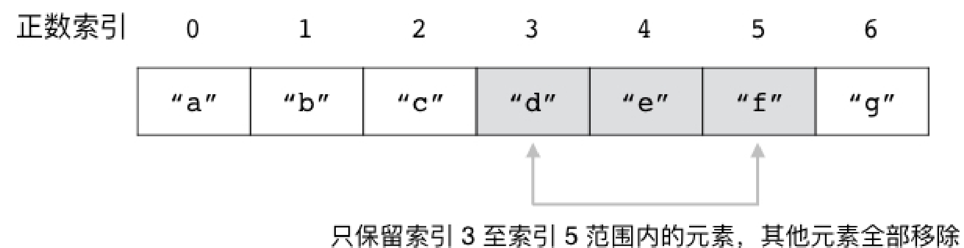
```
redis> LTRIM alphabets 3 5
OK

redis> LRANGE alphabets 0 -1
1) "d"
2) "e"
3) "f"
```

图 4-22 展示了以上两个 LTRIM 命令对 alphabets 列表进行修剪的整个过程。

图 4-22 LTRIM 命令的执行过程

正数索引	0	1	2	3	4	5	6	7	8	9	10
	“a”	“b”	“c”	“d”	“e”	“f”	“g”	“h”	“i”	“j”	“k”





正数索引	0	1	2
	“d”	“e”	“f”

4.14.1 处理负数索引

跟 LRANGE 命令一样， LTRIM 命令不仅可以接受正数索引， 还可以接受负数索引。

以下代码展示了如何通过给定负数索引， 让 LTRIM 命令只保留列表的最后五个元素：

```
redis> RPUSH numbers 0 1 2 3 4 5 6 7 8 9
(integer) 10

redis> LTRIM numbers -5 -1
OK

redis> LRANGE numbers 0 -1
1) "5"
2) "6"
3) "7"
4) "8"
5) "9"
```

4.14.2 其他信息

属性	值
复杂度	O(N)，其中 N 为给定列表的长度。
版本要求	LTRIM 命令从 Redis 1.0.0 版本开始可用。

4.15 LREM：从列表中移除指定元素

用户可以通过 LREM 命令移除列表中的指定元素：

```
LREM list count element
```

count 参数的值决定了 LREM 命令移除元素的方式：

- 如果 count 参数的值等于 0 ， 那么 LREM 命令将移除列表中包含的所有指定元素；
- 如果 count 参数的值大于 0 ， 那么 LREM 命令将从列表的左端开始向右进行检查， 并移除最先发现的 count 个指定元素；
- 如果 count 参数的值小于 0 ， 那么 LREM 命令将从列表的右端开始向左进行检查， 并移除最先发现的 abs(count) 个指定元素 （abs(count) 即是 count 的绝对值）；

LREM 命令在执行完毕之后将返回被移除的元素数量作为返回值。

举个例子， 对于以下三个包含相同元素的列表来说：

```
redis> RPUSH sample1 "a" "b" "b" "a" "c" "c" "a"
(integer) 7

redis> RPUSH sample2 "a" "b" "b" "a" "c" "c" "a"
(integer) 7

redis> RPUSH sample3 "a" "b" "b" "a" "c" "c" "a"
(integer) 7
```

执行以下命令将移除 sample1 列表包含的所有 "a" 元素：

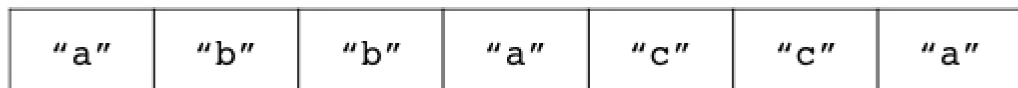
```
redis> LREM sample1 0 "a"
(integer) 3   -- 移除了三个 "a" 元素

redis> LRANGE sample1 0 -1
1) "b"   -- 列表里面已经不再包含 "a" 元素
2) "b"
3) "c"
4) "c"
```

而执行以下命令将移除 sample2 列表最靠近列表左端的两个 "a" 元素：

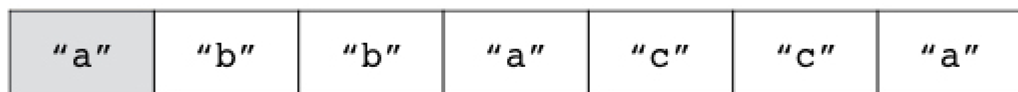
```
redis> LREM sample2 2 "a"
(integer) 2   -- 移除了两个 "a" 元素

redis> LRANGE sample2 0 -1
1) "b"
2) "b"
3) "c"
4) "c"
5) "a"
```

因为上面的 LREM 命令只要求移除最先发现的两个 "a" 元素， 所以位于列表最右端的 "a" 元素并没有被移除， 图 4-23 展示了这个 LREM 命令的执行过程。

图 4-23 从 sample2 列表中移除最靠近列表左端的两个 "a" 元素



从列表左端开始查找并移除“a”元素

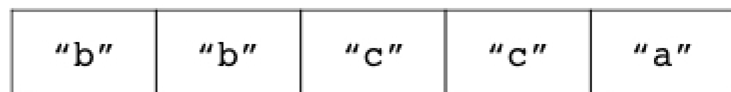


发现“a”元素，移除它

不是“a”元素，继续检查下一个元素

不是“a”元素，继续检查下一个元素

发现“a”元素，移除它



本次操作已经移除了两个“a”元素，

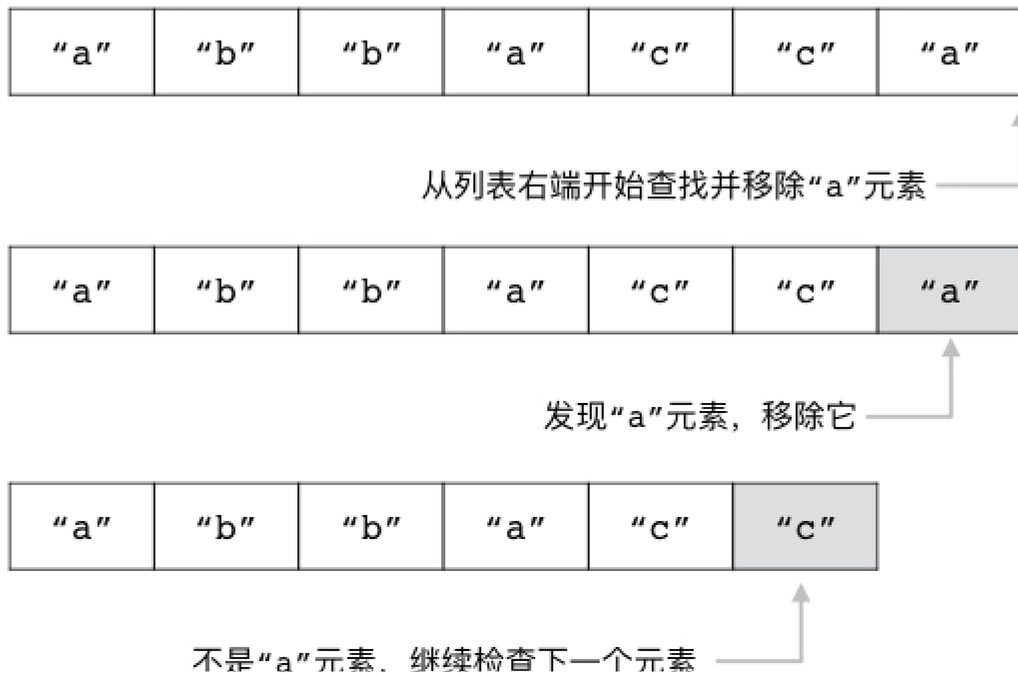
LREM 命令的要求已满足，移除操作到此结束。

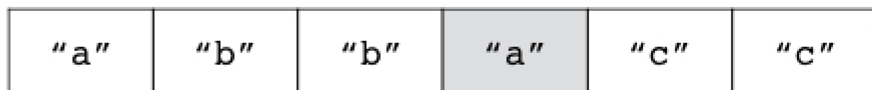
最后， 执行以下命令将移除 sample3 列表最靠近列表右端的两个 "a" 元素：

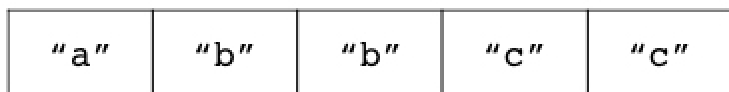
```
redis> LREM sample3 -2 "a"
(integer) 2  -- 移除了两个 "a" 元素

redis> LRANGE sample3 0 -1
1) "a"
2) "b"
3) "b"
4) "c"
5) "c"
```

因为上面的 LREM 调用只要求移除最先发现的两个 "a" 元素， 所以位于列表最左端的 "a" 元素并没有被移除， 图 4-24 展示了这个 LREM 调用的执行过程。

图 4-24 从 sample3 列表里面移除最靠近列表右端的两个 "a" 元素







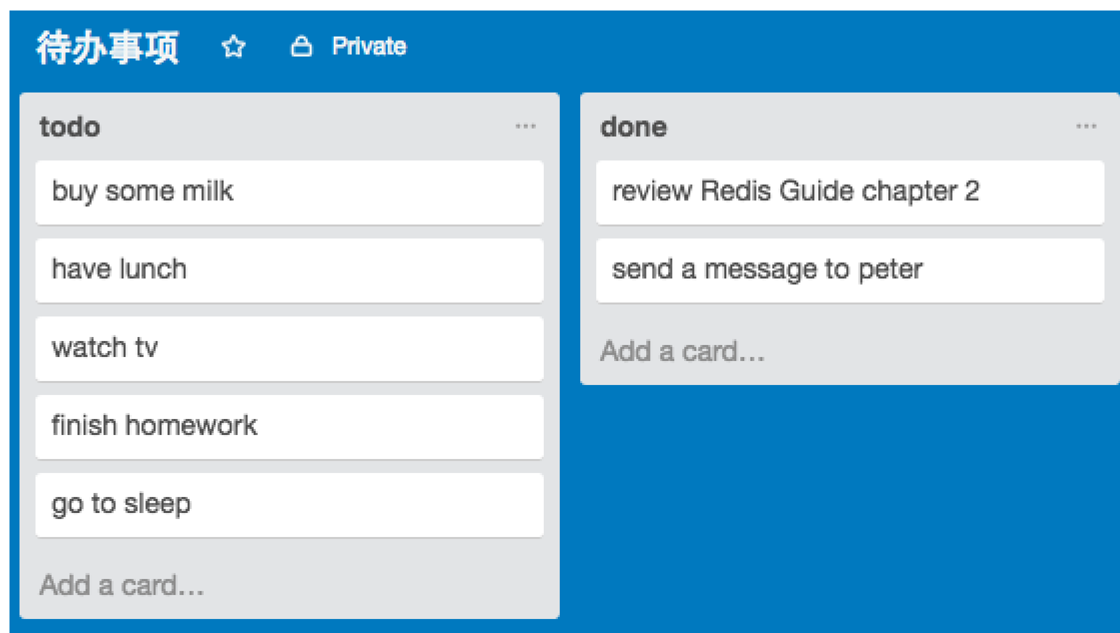
4.15.1 其他信息

属性	值
复杂度	O(N)，其中 N 为给定列表的长度。
版本要求	`LREM` 命令从 Redis 1.0.0 版本开始可用。

4.16 示例：待办事项列表

现在很多人都会使用待办事项软件（也就是通常说的TODO软件）去管理他们的待办事项，这些软件通常会提供一些列表，用户可以将他们要做的事情记录在待办事项列表里面，并将已经完成的事项放入到已完成事项列表里面。比如图 4–25 就展示了一个使用待办事项软件记录日常生活事项的例子。

图 4-25 使用待办事项软件记录日常生活事项



代码清单 4-3 展示了一个使用列表实现的待办事项程序， 这个程序的核心概念是使用两个列表来分别记录待办事项和已完成事项：

- 当用户添加一个新的待办事项时， 程序就把这个事项放入到待办事项列表中。
- 当用户完成待办事项列表中的某个事项时， 程序就把这个事项从待办事项列表中移除， 并将它放入到已完成事项列表中。

代码清单 4-3 代码事项程序： /list/todo_list.py

```
def make_todo_list_key(user_id):
    """
    储存待办事项的列表。
    """
    return user_id + "::todo_list"

def make_done_list_key(user_id):
    """
```

```python
    储存已完成事项的列表。
    """
    return user_id + "::done_list"


class TodoList:

    def __init__(self, client, user_id):
        self.client = client
        self.user_id = user_id
        self.todo_list = make_todo_list_key(self.user_id)
        self.done_list = make_done_list_key(self.user_id)

    def add(self, event):
        """
        将指定事项添加到待办事项列表中。
        """
        self.client.lpush(self.todo_list, event)

    def remove(self, event):
        """
        从待办事项列表中移除指定的事项。
        """
        self.client.lrem(self.todo_list, 0, event)

    def done(self, event):
        """
        将待办事项列表中的指定事项移动到已完成事项列表，
        以此来表示该事项已完成。
        """
        # 从待办事项列表中移除指定事项
        self.remove(event)
        # 并将它添加到已完成事项列表中
        self.client.lpush(self.done_list, event)

    def show_todo_list(self):
        """
        列出所有待办事项。
        """
        return self.client.lrange(self.todo_list, 0, -1)

    def show_done_list(self):
        """
        列出所有已完成事项。
        """
        return self.client.lrange(self.done_list, 0, -1)
```

done() 方法是 TodoList 程序的核心， 它首先会使用 LREM 命令从代办事项列表中移除指定的事项， 然后再将该事项添加到已完成事项列表中， 使得该事项可以在代办事项列表中消失， 转而出现在已完成列表中。

作为例子， 我们可以通过执行以下代码， 创建出一个 TODO 列表对象：

```
>>> from redis import Redis
>>> from todo_list import TodoList
>>> client = Redis(decode_responses=True)
>>> todo = TodoList(client, "peter's todo")
```

然后通过执行以下代码， 往 TODO 列表里面加入待完成事项：

```
>>> todo.add("go to sleep")
>>> todo.add("finish homework")
>>> todo.add("watch tv")
>>> todo.add("have lunch")
>>> todo.add("buy some milk")
>>> todo.show_todo_list()
['buy some milk', 'have lunch', 'watch tv', 'finish homework', 'go to sleep']
```

当我们完成某件事情之后， 我们可以把它从待完成事项列表移动到已完成事项列表：

```
>>> todo.done("buy some milk")
>>> todo.show_todo_list()
['have lunch', 'watch tv', 'finish homework', 'go to sleep']
>>> todo.show_done_list()
['buy some milk']
```

最后， 如果我们不再需要去做某件事情， 那么可以把它从待完成列表里面移除：

```
>>> todo.remove("watch tv")
>>> todo.show_todo_list()
['have lunch', 'finish homework', 'go to sleep']
```

4.17 BLPOP：阻塞式左端弹出操作

BLPOP 命令是带有阻塞功能的左端弹出操作， 它接受任意多个列表以及一个秒级精度的超时时限作为参数：

```
BLPOP list [list ...] timeout
```

BLPOP 命令会按照从左到右的顺序依次检查用户给定的列表， 并对最先遇到的非空列表执行左端元素弹出操作。如果 BLPOP 命令在检查了用户给定的所有列表之后都没有发现可以执行弹出操作的非空列表， 那么它将阻塞执行该命令的客户端并开始等待， 直到某个给定列表变为非空， 又或者等待时间超出给定时限为止。

当 BLPOP 命令成功对某个非空列表执行了弹出操作之后， 它将向用户返回一个包含两个元素的数组： 数组的第一个元素记录了执行弹出操作的列表， 也即是被弹出元素的来源列表； 而数组的第二个元素则是被弹出元素本身。

比如在以下这个 BLPOP 命令执行示例里面， 被弹出的元素 "a" 就来源于列表 alphabets ：

```
redis> BLPOP alphabets 5   -- 尝试弹出 alphabets 列表的最左端元素，最多阻塞 5 秒
1) "alphabets"             -- 被弹出元素的来源列表
2) "a"                     -- 被弹出元素
```

如果用户使用的是 redis-cli 客户端， 并且在执行 BLPOP 命令的过程中曾经被阻塞过， 那么客户端还会将被阻塞的时长也打印出来：

```
redis> BLPOP message-queue 5
1) "message-queue"
2) "hello world!"
(1.60s)    -- 客户端执行这个命令时被阻塞了 1.6 秒
```

注意， 这里展示的阻塞时长只是 redis-cli 客户端为了方便用户而添加的额外信息， BLPOP 命令返回的结果本身并不包含这一信息。

4.17.1 解除阻塞状态

正如前面所说， 当 BLPOP 命令发现用户给定的所有列表都为空时， 它就会让执行命令的客户端进入阻塞状态。如果在客户端被阻塞的过程中， 有另一个客户端向导致阻塞的列表推入了新的元素， 那么该列表就会变为非空， 而被阻塞的客户端也会随着 BLPOP 命令成功弹出列表元素而重新回到非阻塞状态。

作为例子， 表 4-1 展示了一个客户端从被阻塞到解除阻塞的整个过程。

表 4-1 客户端 A 从被阻塞到解除阻塞的整个过程

时间	客户端 A	客户端 B

时间	客户端 A	客户端 B
T1	执行 BLPOP lst 10 ， 因为 lst 为空而导致客户端被阻塞。	
T2		执行 RPUSH lst "hello" 命令， 将 "hello" 元素推入到列表 lst 中。
T3	服务器检测到导致这个客户端阻塞的 lst 列表已经非空， 于是从列表中弹出 "hello" 元素并将其返回给客户端。	
T4	接收到 "hello" 元素的客户端重新回到非阻塞状态。	

如果在同一时间， 有多个客户端因为同一个列表而被阻塞， 那么当导致阻塞的列表变为非空时， 服务器将按照“先阻塞先服务”的规则， 依次为被阻塞的各个客户端弹出列表元素。

比如表 4-2 就展示了一个服务器按照先阻塞先服务规则处理被阻塞客户端的例子： 在这个例子中， A、B、C 三个客户端先后执行了 BLPOP lst 10 命令， 并且都因为 lst 列表为空而被阻塞， 如果在这些客户端被阻塞期间， 客户端 D 执行了 RPUSH lst "hello" "world" "again" 命令， 那么服务器首先会处理客户端 A 的 BLPOP 命令， 并将被弹出的 "hello" 元素返回给它； 接着处理客户端 B 的 BLPOP 命令， 并将被弹出的 "world" 元素返回给它； 最后处理客户端 C 的 BLPOP 命令， 并将被弹出的 "again" 元素返回给它。

表 4-2 先阻塞先服务器处理示例

时间	客户端 A	客户端 B	客户端 C	客户端 D
T1	执行 BLPOP lst 10			
T2		执行 BLPOP lst 10		
T3			执行 BLPOP lst 10	
T4				执行 RPUSH lst "hello" "world" "again"
T5	从 lst 列表弹出 "hello" 元素并解除阻塞状态。			

时间	客户端 A	客户端 B	客户端 C	客户端 D
T6		从 lst 列表弹出 "world" 元素并解除阻塞状态。		
T7			从 lst 列表弹出 "again" 元素并解除阻塞状态。	

最后， 如果被推入到列表的元素数量少于被阻塞的客户端数量， 那么先被阻塞的客户端将会先解除阻塞， 而未能解除阻塞的客户端则需要继续等待下次推入操作。

比如说， 如果有五个客户端因为列表为空而被阻塞， 但是推入到列表的元素只有三个， 那么最先被阻塞的三个客户端将会解除阻塞状态， 而剩下的两个客户端则会继续阻塞。

4.17.2 处理空列表

如果用户向 BLPOP 命令传入的所有列表都是空列表， 并且这些列表在给定的时限之内一直没有变成非空列表， 那么 BLPOP 命令将在给定时限到达之后向客户端返回一个空值， 表示没有任何元素被弹出：

```
redis> BLPOP empty-list 5
(nil)
(5.04s)
```

4.17.3 列表名的作用

BLPOP 命令之所以会返回被弹出元素的来源列表， 是为了让用户在传入多个列表的情况下， 知道被弹出的元素来源于哪个列表。

比如在以下这个示例里面， 通过 BLPOP 命令的回复， 我们可以知道被弹出的元素来自于列表 queue2 ， 而不是 queue1 或者 queue3 ：

```
redis> BLPOP queue1 queue2 queue3 5
1) "queue2"
```

```
2) "hello world!"
```

4.17.4 阻塞效果的范围

BLPOP 命令的阻塞效果只对执行这个命令的客户端有效， 其他客户端以及 Redis 服务器本身并不会因为这个命令而被阻塞。

4.17.5 其他信息

属性	值
复杂度	O(N)， 其中 N 为用户给定的列表数量。
版本要求	BLPOP 命令从 Redis 2.0.0 版本开始可用。

4.18 BRPOP：阻塞式右端弹出操作

BRPOP 命令是带有阻塞功能的右端弹出操作， 除了弹出的方向不同之外， 它的其他方面都和 BLPOP 命令一样：

```
BRPOP list [list ...] timeout
```

作为例子， 以下代码展示了如何使用 BRPOP 命令去尝试弹出给定列表的最右端元素：

```
redis> BRPOP queue1 queue2 queue3 10
1) "queue2"    -- 被弹出元素的来源列表
2) "bye bye"   -- 被弹出元素
```

4.18.1 其他信息

属性	值
复杂度	O(N)， 其中 N 为用户给定的列表数量。
版本要求	BRPOP 命令从 Redis 2.0.0 版本开始可用。

4.19 BRPOPLPUSH：阻塞式弹出并推入操作

BRPOPLPUSH 命令是 RPOPLPUSH 命令的阻塞版本， 它接受一个源列表、一个目标列表以及一个秒级精度的超时时限作为参数：

```
BRPOPLPUSH source target timeout
```

根据源列表是否为空， BRPOPLPUSH 命令会产生以下两种行为：

- 如果源列表非空， 那么 BRPOPLPUSH 命令的行为就和 RPOPLPUSH 命令的行为一样： BRPOPLPUSH 命令会弹出位于源列表最右端的元素， 并将该元素推入到目标列表的左端， 最后向客户端返回被推入的元素。
- 如果源列表为空， 那么 BRPOPLPUSH 命令将阻塞执行该命令的客户端， 然后在给定的时限内等待可弹出的元素出现， 又或者等待时间超过给定时限为止。

举个例子， 假设现在有 list3 、 list4 两个列表如下：

```
client-1> LRANGE list3 0 -1
1) "hello"

client-1> LRANGE list4 0 -1
1) "a"
2) "b"
3) "c"
```

如果我们以这两个列表作为输入执行 BRPOPLPUSH 命令， 由于源列表 list3 非空， 所以 BRPOPLPUSH 命令将不阻塞直接执行， 就像 RPOPLPUSH 命令一样：

```
client-1> BRPOPLPUSH list3 list4 10
"hello"

client-1> LRANGE list3 0 -1
(empty list or set)

client-1> LRANGE list4 0 -1
1) "hello"
2) "a"
3) "b"
4) "c"
```

现在， 由于 list3 为空， 如果我们再次执行相同的 BRPOPLPUSH 命令， 那么客户端 client-1 将被阻塞， 直到我们从另一个客户端 client-2 向 list3 推入新元素为止：

```
client-1> BRPOPLPUSH list3 list4 10
"world"
(1.42s)  -- 被阻塞了 1.42 秒

client-1> LRANGE list3 0 -1
(empty list or set)

client-1> LRANGE list4 0 -1
1) "world"
2) "hello"
3) "a"
4) "b"
5) "c"
```

```
client-2> RPUSH list3 "world"
(integer) 1
```

表 4-3 展示了客户端从被阻塞到解除阻塞的整个过程。

表 4-3 阻塞 BRPOPLPUSH 命令的执行过程

时间	客户端 client-1	客户端 client-2
T1	尝试执行 BRPOPLPUSH list3 list4 10 并被阻塞。	
T2		执行 RPUSH list3 "world" ，向列表 list3 推入新元素。
T3	服务器执行 BRPOPLPUSH 命令， 并将元素 "world" 返回给客户端。	

4.19.1 处理源列表为空的情况

如果源列表在用户给定的时限内一直没有元素可供弹出， 那么 BRPOPLPUSH 命令将向客户端返回一个空值， 以此来表示此次操作没有弹出和推入任何元素：

```
redis> BRPOPLPUSH empty-list another-list 5
(nil)
(5.05s)   -- 客户端被阻塞了 5.05 秒
```

跟 BLPOP 命令和 BRPOP 命令一样， redis-cli 客户端也会显示 BRPOPLPUSH 命令的阻塞时长。

4.19.2 其他信息

属性	值
复杂度	O(1)
版本要求	BRPOPLPUSH 命令从 Redis 2.2.0 版本开始可用。

4.20 示例：带有阻塞功能的消息队列

在构建应用程序的时候， 我们有时候会遇到一些非常耗时的操作， 比如发送邮件、将一条新微博同步给上百万个用户、对硬盘进行大量读写、执行庞大的计算等等。 因为这些操作是如此耗时， 所以如果我们直接在响应用户请求的过程中执行它们的话， 那么用户就需要等待非常长时间。

比如说， 为了验证用户身份的有效性， 有些网站在注册新用户的时候， 会向用户给定的邮件地址发送一封激活邮件， 用户只有在点击了验证邮件里面的激活链接之后， 新注册的帐号才能够正常使用。

下面这段伪代码展示了一个带有邮件验证功能的帐号注册函数， 这个函数不仅会为用户输入的用户名和密码创建新帐号， 还会向用户给定的邮件地址发送一封激活：

```python
def register(username, password, email):
    # 创建新帐号
    create_new_account(username, password)
    # 发送激活邮件
    send_validate_email(email)
    # 向用户返回注册结果
    ui_print("帐号注册成功，请访问你的邮箱并激活帐号。")
```

因为邮件发送操作需要进行复杂的网络信息交换， 所以它并不是一个快速的操作， 如果我们直接在 send_valid_email() 函数里面执行邮件发送操作的话， 那么用户可能就需要等待一段较长的时间才能看到 ui_print() 函数打印出的反馈信息。

为了解决这个问题， 在执行 `send_validate_email()` 函数的时候， 我们可以不立即执行邮件发送操作， 而是将邮件发送任务放入到一个队列里面， 然后由后台的线程负责实际执行。 这样的话， 程序只需要执行一个入队操作， 然后就可以直接向用户反馈注册结果了， 这比实际地发送邮件之后再向用户反馈结果要快得多。

代码清单 4-4 展示了一个使用 Redis 实现的消息队列， 它使用 `RPUSH` 命令将消息推入队列， 并使用 `BLPOP` 命令从队列里面取出待处理的消息。

代码清单 4-4 使用列表实现的消息队列： `/list/message_queue.py`

```
class MessageQueue:

    def __init__(self, client, queue_name):
        self.client = client
        self.queue_name = queue_name

    def add_message(self, message):
        """
        将一条消息放入到队列里面。
        """
        self.client.rpush(self.queue_name, message)

    def get_message(self, timeout=0):
        """
        从队列里面获取一条消息，
        如果暂时没有消息可用， 那么就在 timeout 参数指定的时限内阻塞并等待可用消息出现。

        timeout 参数的默认值为 0 ， 表示一直等待直到消息出现为止。
        """
        # blpop 的结果可以是 None ， 也可以是一个包含两个元素的元组
        # 元组的第一个元素是弹出元素的来源队列， 而第二个元素则是被弹出的元素
        result = self.client.blpop(self.queue_name, timeout)
        if result is not None:
            source_queue, poped_item = result
            return poped_item

    def len(self):
        """
        返回队列目前包含的消息数量。
        """
        return self.client.llen(self.queue_name)
```

为了使用这个消息队列， 我们通常需要用到两个客户端：

- 一个客户端作为消息的发送者（sender）， 它需要将待处理的消息推入到队列里面；
- 而另一个客户端则作为消息的接收者（receiver）和消费者（consumer）， 它负责从队列里面取出消息，并根据消息内容进行相应的处理工作。

下面的这段代码展示了一个简单的消息接收者， 在没有消息的时候， 这个程序将阻塞在 `mq.get_message()` 调用上面； 当有消息（邮件地址）出现时， 程序就会打印出该消息并发送邮件：

```
>>> from redis import Redis
>>> from message_queue import MessageQueue
>>> client = Redis(decode_responses=True)
>>> mq = MessageQueue(client, 'validate user email queue')
>>> while True:
...   email_address = mq.get_message()  # 阻塞直到消息出现
...   send_email(email_address)          # 打印出邮件地址并发送邮件
...
peter@exampl.com
jack@spam.com
tom@blahblah.com
```

而以下代码则展示了消息发送者是如何将消息推入到队列里面的：

```
>>> from redis import Redis
>>> from message_queue import MessageQueue
>>> client = Redis(decode_responses=True)
>>> mq = MessageQueue(client, 'validate user email queue')
>>> mq.add_message("peter@exampl.com")
>>> mq.add_message("jack@spam.com")
>>> mq.add_message("tom@blahblah.com")
```

4.20.1 阻塞弹出操作的应用

本节展示的消息队列之所以使用 `BLPOP` 命令而不是 `LPOP` 命令来实现出队操作， 是因为阻塞弹出操作可以让消息接收者在队列为空的时候自动阻塞， 而不必手动进行休眠， 从而使得消息处理程序的编写变得更为简单直接， 并且还可以有效地节约系统资源。

作为对比， 以下代码展示了在使用 `LPOP` 命令实现出队操作的情况下， 如何实现类似上面展示的消息处理程序：

```
while True:
    # 尝试获取消息，如果没有消息，那么返回 None
    email_address = mq.get_message()
```

```
    if email_address is not None:
        # 有消息，发送邮件
        send_email(email_address)
    else:
        # 没有消息可用，休眠一百毫秒之后再试
        sleep(0.1)
```

因为缺少自动的阻塞操作， 所以这个程序在没有取得消息的情况下， 只能以一百毫秒一次的频率去尝试获取消息， 如果队列为空的时间比较长， 那么这个程序就会发送很多多余的 `LPOP` 命令， 并因此浪费很多 CPU 资源和网络资源。

4.20.2 使用消息队列实现实时提醒

消息队列除了可以在应用程序的内部中使用， 还可以用于实现面向用户的实时提醒系统。

比如说， 如果我们在构建一个社交网站的话， 那么可以使用 JavaScript 脚本， 让客户端以异步的方式调用 `MessageQueue` 类的 `get_message()` 方法， 然后程序就可以在用户被关注的时候、收到了新回复的时候又或者收到新私信的时候， 通过调用 `add_message()` 方法来向用户发送提醒信息。

4.21 重点回顾

- Redis 的列表是一种线性的有序结构， 它可以按照元素推入到列表中的顺序来储存元素， 并且列表中的元素可以出现重复。
- 用户可以使用 `LPUSH` 、 `RPUSH` 、 `RPOP` 、 `LPOP` 等多个命令， 从列表的两端推入或者弹出元素， 也可以通过 `LINSERT` 命令， 将新元素插入到列表已有元素的前面或后面。
- 用户可以使用 `LREM` 命令从列表中移除指定的元素， 又或者直接使用 `LTRIM` 命令对列表进行修剪。
- 当用户传给 `LRANGE` 命令的索引范围超出了列表的有效索引范围时， `LRANGE` 命令将对传入的索引范围进行修正， 并根据修正后的索引范围来获取列表元素。
- `BLPOP` 、 `BRPOP` 和 `BRPOPLPUSH` 是阻塞版本的弹出和推入命令， 如果用户给定的所有列表都为空， 那么执行命令的客户端将被阻塞， 直到给定的阻塞时限到达又或者某个给定列表非空为止。

5. 集合（Set）

Redis 的集合键允许用户将任意多个各不相同的元素储存到集合里面， 这些元素既可以是文本数据， 也可以是二进制数据。 虽然上一章介绍的列表键也允许我们储存多个元素， 但是跟列表相比， 集合有以下两个明显的区别：

1. 列表可以储存重复元素， 而集合只会储存非重复元素， 尝试将一个已存在的元素添加到集合将被忽略。
2. 列表以有序方式储存元素， 而集合则以无序方式储存元素。

这两个区别带来的差异主要跟命令的复杂度有关：

- 在执行像 `LINSERT` 和 `LREM` 这样的列表命令时， 即使命令只针对单个列表元素， 程序有时候也不得不遍历整个列表以确定指定的元素是否存在， 因此这些命令的复杂度都为 O(N) 。
- 另一方面， 对于集合来说， 因为所有针对单个元素的集合命令都不需要遍历整个集合， 所以它们的复杂度都为 O(1) 。

因此当我们需要储存多个元素时， 就可以考虑这些元素是否可以以无序的方式储存， 并且是否不会出现重复， 如果是的话， 那么就可以使用集合来储存这些元素， 从而有效地利用集合操作的效率优势。

作为例子， 图 5-1 展示了一个名为 `databases` 的集合， 这个集合里面包含了 `"Redis"` 、 `"MongoDB"` 、 `"MySQL"` 等八个元素。

图 5-1 集合示例

database 集合

"Redis"	"MongoDB"	"CouchDB"
"MySQL"	"PostgreSQL"	"Oracle"
"Neo4j"	"MS SQL"	

Redis 为集合键提供了一系列操作命令， 通过使用这些命令， 用户可以：

- 将新元素添加到集合里面，或者从集合里面移除已有的元素。
- 将指定的元素从一个集合移动到另一个集合。
- 获取集合包含的所有元素。
- 获取集合包含的元素数量。

- 检查给定元素是否存在于集合。
- 从集合里面随机地获取指定数量的元素。
- 对多个集合执行交集、并集、差集计算。

本章接下来将对 Redis 集合键的各个命令进行介绍， 并说明如何使用这些命令去解决各种实际存在的问题。

5.1 SADD：将元素添加到集合

通过使用 SADD 命令， 用户可以将一个或多个元素添加到集合里面：

```
SADD set element [element ...]
```

这个命令会返回成功添加的新元素数量作为返回值。

以下代码展示了如何使用 SADD 命令去构建一个 databases 集合：

```
redis> SADD databases "Redis"
(integer) 1     -- 集合新添加了一个元素

redis> SADD databases "MongoDB" "CouchDB"
(integer) 2     -- 集合新添加了两个元素

redis> SADD databases "MySQL" "PostgreSQL" "MS SQL" "Oracle"
(integer) 4     -- 集合新添加了四个元素
```

图 5-2 展示了以上三个 SADD 命令构建出 databases 集合的整个过程。

图 5-2 使用 SADD 命令构建集合的整个过程

执行 SADD databases "Redis"

database 集合

“Redis”

执行 `SADD databases "MongoDB" "CouchDB"`

database 集合

"Redis"	"MongoDB"	"CouchDB"

执行 `SADD databases "MySQL" "PostgreSQL" "MS SQL" "Oracle"`

database 集合

"Redis"	"MongoDB"	"CouchDB"
"MySQL"	"PostgreSQL"	"MS SQL"
"Oracle"		

5.1.1 忽略已存在元素

因为集合不储存相同的元素， 所以用户在使用 `SADD` 命令向集合里面添加元素的时候， `SADD` 命令会自动忽略已存在的元素， 只将不存在于集合的新元素添加到集合里面。

在以下展示的代码中， 我们分别尝试向 `databases` 集合添加元素 `"Redis"` 、 `"MySQL"` 以及 `"PostgreSQL"` ， 但是因为这些元素都已经存在于 `databases` 集合， 所以 `SADD` 命令将忽略这些元素：

```
redis> SADD databases "Redis"
(integer) 0     -- 成功添加的新元素数量为 0 ，表示没有任何新元素被添加到集合当中

redis> SADD databases "MySQL" "PostgreSQL"
(integer) 0     -- 同样，这次也没有任何元素被添加到集合里面
```

而在以下代码中， `SADD` 命令会将新元素 `"Neo4j"` 添加到集合里面， 并忽略 `"Redis"` 和 `"MySQL"` 这两个已存在的元素：

```
redis> SADD databases "Redis" "MySQL" "Neo4j"
(integer) 1
```

5.1.2 其他信息

属性	值
复杂度	O(N)，其中 N 为用户给定的元素数量。
版本要求	SADD 命令从 Redis 1.0.0 版本开始可用， 但是只有 Redis 2.4 或以上版本的 SADD 命令可以一次添加多个元素， Redis 2.4 以下版本的 SADD 命令每次只能添加一个元素。

5.2 SREM：从集合中移除元素

通过使用 SREM 命令， 用户可以从集合里面移除一个或多个已存在的元素：

```
SREM set element [element ...]
```

这个命令会返回被移除的元素数量作为返回值。

以下代码展示了如何使用 SREM 命令去移除 databases 集合中的 "Neo4j" 等元素：

```
redis> SREM databases Neo4j
(integer) 1     -- 有一个元素被移除

redis> SREM databases "MS SQL" "Oracle" "CouchDB"
(integer) 3     -- 有三个元素被移除
```

图 5-3 展示了 databases 集合在执行 SREM 命令过程中的变化。

图 5-3 databases 集合的整个变化过程

执行 SREM 命令之前

“Redis”	“MongoDB”	“CouchDB”
“MySQL”	“PostgreSQL”	“Oracle”
“Neo4j”	“MS SQL”	

执行 `SREM databases "Neo4j"` 之后

“Redis”	“MongoDB”	“CouchDB”
“MySQL”	“PostgreSQL”	“Oracle”
	“MS SQL”	

执行 `SREM databases "MS SQL" "Oracle" "CouchDB"` 之后

“Redis”	“MongoDB”	
“MySQL”	“PostgreSQL”	

5.2.1 忽略不存在的元素

如果用户给定的元素并不存在于集合当中， 那么 `SREM` 命令将忽略不存在的元素， 只移除那些确实存在的元素。

在以下代码中， 因为元素 `"Memcached"` 并不存在于 `databases` 集合， 所以 `SREM` 命令没有从集合里面移除任何元素：

```
redis> SREM databases "Memcached"
(integer) 0     -- 没有元素被移除
```

5.2.2 其他信息

属性	值
复杂度	O(N)，其中 N 为用户给定的元素数量。
版本要求	SREM 命令从 Redis 1.0.0 版本开始可用， 但是只有 Redis 2.4 或以上版本的 SREM 命令可以一次删除多个元素， Redis 2.4 以下版本的 SREM 命令每次只能删除一个元素。

5.3 SMOVE：将元素从一个集合移动到另一个集合

SMOVE 命令允许用户将指定的元素从源集合移动到目标集合：

```
SMOVE source target element
```

SMOVE 命令在移动操作成功执行时返回 1 ； 如果指定的元素并不存在于源集合， 那么 SMOVE 命令将返回 0 ， 表示移动操作执行失败。

以下代码展示了如何通过 SMOVE 命令， 将存在于 databases 集合的 "Redis" 元素以及 "MongoDB" 元素移动到 nosql 集合里面：

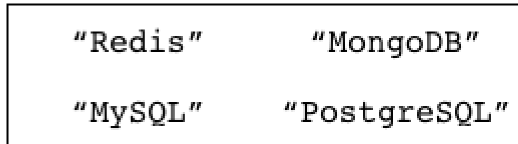
```
redis> SMOVE databases nosql "Redis"
(integer) 1     -- 移动成功

redis> SMOVE databases nosql "MongoDB"
(integer) 1     -- 移动成功
```

图 5-4 展示了这两个 SMOVE 命令的执行过程。

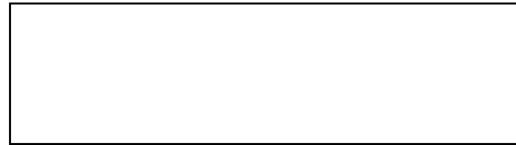
图 5-4 SMOVE 命令的执行过程：

执行 SMOVE 命令之前的 databases 集合和 nosql 集合

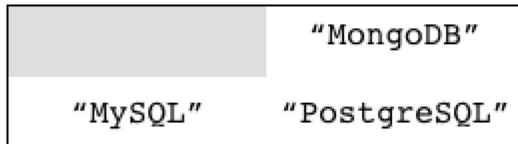
database 集合

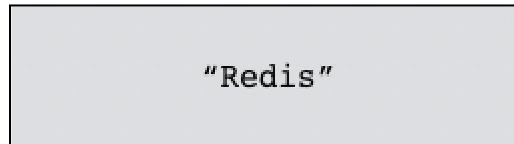


nosql 集合
(不存在的空集合)

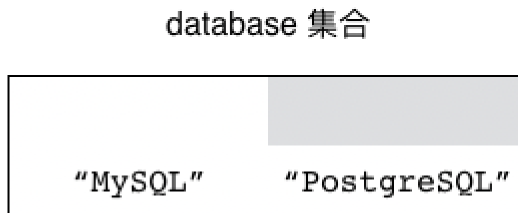


执行 `SMOVE databases nosql "Redis"` 之后

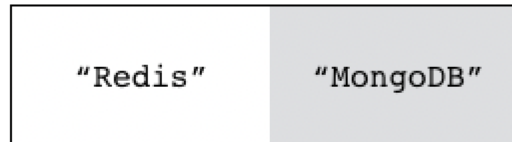




执行 `SMOVE databases nosql "MongoDB"` 之后



nosql 集合



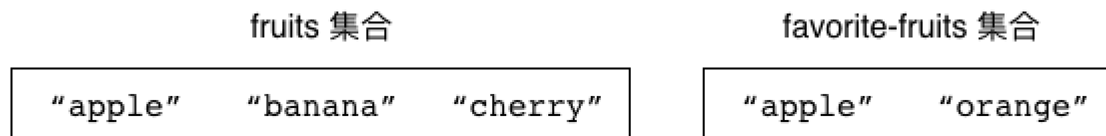
5.3.1 忽略不存在的元素

如果用户想要移动的元素并不存在于源集合， 那么 `SMOVE` 将放弃执行移动操作， 并返回 `0` 表示移动操作执行失败。

举个例子， 对于图 5-5 所示的 `fruits` 集合和 `favorite-fruits` 集合来说， 尝试把不存在于 `fruits` 集合的 `"dragon fruit"` 元素移动到 `favorite-fruits` 集合将会导致 `SMOVE` 命令执行失败：

```
redis> SMOVE fruits favorite-fruits "dragon fruit"
(integer) 0     -- 没有元素被移动
```

图 5-5 fruits 集合和 favorite-fruits 集合



5.3.2 覆盖已存在的元素

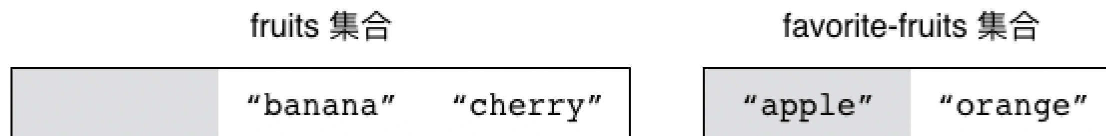
即使用户想要移动的元素已经存在于目标集合， SMOVE 命令仍然会将指定的元素从源集合移动到目标集合， 并覆盖目标集合里面的相同元素。 从结果来看， 这种移动不会改变目标集合包含的元素， 只会导致被移动的元素从源集合中消失。

以图 5-5 中展示的 fruits 集合和 favorite-fruits 集合为例， 如果我们执行以下代码：

```
redis> SMOVE fruits favorite-fruits "apple"
(integer) 1
```

那么 fruits 集合中的 "apple" 元素将被移动到 favorite-fruits 集合里面， 覆盖掉 favorite-fruits 集合原有的 "apple" 元素。 从结果来看， "apple" 元素将从 fruits 集合中消失， 而 favorite-fruits 集合包含的元素则不会发生变化。 图 5-6 展示了上面的 SMOVE 命令执行之后， fruits 集合和 favorite-fruits 集合的样子。

图 5-6 执行 SMOVE 命令之后的 fruits 集合和 favorite-fruits 集合



5.3.3 其他信息

属性	值
复杂度	O(1)

属性	值
版本要求	SMOVE 命令从 Redis 1.0.0 版本开始可用。

5.4 SMEMBERS：获取集合包含的所有元素

通过使用 SMEMBERS 命令，用户可以取得集合包含的所有元素：

```
SMEMBERS set
```

以下代码展示了如何使用 SMEMBERS 命令去获取 fruits 集合、favorite-numbers 集合以及 databases 集合的所有元素：

```
redis> SMEMBERS fruits
1) "banana"
2) "cherry"
3) "apple"

redis> SMEMBERS favorite-numbers
1) "12345"
2) "999"
3) "3.14"
4) "1024"
5) "10086"

redis> SMEMBERS databases
1) "Redis"
2) "PostgreSQL"
3) "MongoDB"
4) "MySQL"
```

5.4.1 元素的无序排列

因为 Redis 集合以无序的方式储存元素，并且 SMEMBERS 命令在获取集合元素时也不会对元素进行任何排序动作，所以根据元素添加顺序的不同，两个包含相同元素的集合在执行 SMEMBERS 命令时的结果也可能会有所不同。

比如在以下代码中， 我们就以相反的顺序向 `fruits-a` 和 `fruits-b` 两个集合添加了相同的三个元素， 但是这两个集合在执行 SMEMBERS 命令时的结果并不相同：

```
redis> SADD fruits-a "apple" "banana" "cherry"
(integer) 3

redis> SMEMBERS fruits-a
1) "cherry"
2) "banana"
3) "apple"

redis> SADD fruits-b "cherry" "banana" "apple"
(integer) 3

redis> SMEMBERS fruits-b
1) "cherry"
2) "apple"
3) "banana"
```

因此我们在使用 SMEMBERS 命令以及集合的时候， 不应该对集合元素的排列顺序做任何假设。 如果有需要的话， 我们可以在客户端里面对 SMEMBERS 命令返回的元素进行排序， 又或者直接使用 Redis 提供的有序结构（比如列表和有序集合）。

5.4.2 其他信息

属性	值
复杂度	O(N)，其中 N 为集合包含的元素数量。
版本要求	SMEMBERS 命令从 Redis 1.0.0 版本开始可用。

5.5 SCARD：获取集合包含的元素数量

通过使用 SCARD 命令， 用户可以取得给定集合的大小， 也即是集合包含的元素数量：

```
SCARD set
```

以下代码展示了如何使用 SCARD 命令去获取 `databases` 集合、 `fruits` 集合以及 `favorite-numbers` 集合的大小：

```
redis> SCARD databases
(integer) 4     -- 这个集合包含四个元素

redis> SCARD fruits
(integer) 3     -- 这个集合包含三个元素

redis> SCARD favorite-numbers
(integer) 5     -- 这个集合包含五个元素
```

5.5.1 其他信息

属性	值
复杂度	O(1)
版本要求	SCARD 命令从 Redis 1.0.0 版本开始可用。

5.6 SISMEMBER：检查给定元素是否存在于集合

通过使用 SISMEMBER 命令， 用户可以检查给定的元素是否存在于集合当中：

```
SISMEMBER set element
```

SISMEMBER 命令返回 1 表示给定的元素存在于集合当中， 而返回 0 则表示给定元素不存在于集合当中。

举个例子， 对于以下这个 databases 集合来说：

```
redis> SMEMBERS databases
1) "Redis"
2) "MySQL"
3) "MongoDB"
4) "PostgreSQL"
```

使用 SISMEMBER 命令去检测已经存在于集合中的 "Redis" 元素、 "MongoDB" 元素以及 "MySQL" 元素都将得到肯定的回答：

```
redis> SISMEMBER databases "Redis"
(integer) 1
```

```
redis> SISMEMBER databases "MongoDB"
(integer) 1

redis> SISMEMBER databases "MySQL"
(integer) 1
```

而使用 `SISMEMBER` 命令去检测不存在于集合当中的 `"Oracle"` 元素、 `"Neo4j"` 元素以及 `"Memcached"` 元素则会得到否定的回答：

```
redis> SISMEMBER databases "Oracle"
(integer) 0

redis> SISMEMBER databases "Neo4j"
(integer) 0

redis> SISMEMBER databases "Memcached"
(integer) 0
```

5.6.1 其他信息

属性	值
复杂度	O(1)
版本要求	`SISMEMBER` 命令从 Redis 1.0.0 版本开始可用。

5.7 示例：唯一计数器

本书前面在对字符串键以及散列键进行介绍的时候， 曾经展示过如何使用这两种键去实现计数器程序。 我们当时实现的计数器的作用都非常单纯： 每当某个动作被执行时， 程序就可以调用计数器的加法操作或者减法操作， 对动作的执行次数进行记录。

以上这种简单的计数行为在大部分时候都是有用的， 但是在某些情况下， 我们需要一种要求更为严格的计数器， 这种计数器只会对特定的动作或者对象进行一次计数而不是多次计数。

举个例子， 一个网站的受欢迎程度通常可以用浏览量和用户数量这两个指标进行描述：

- 浏览量记录的是网站页面被用户访问的总次数， 网站的每个用户都可以重复地对同一个页面进行多次访问， 而这些访问会被浏览量计数器一个不漏地被记录起来。

- 至于用户数量记录的则是访问网站的 IP 地址数量， 即使同一个 IP 地址多次访问相同的页面， 用户数量计数器也只会对这个 IP 地址进行一次计数。

对于网站的浏览量， 我们可以继续使用字符串键或者散列键实现的计数器进行计数； 但如果我们想要记录网站的用户数量， 那么就需要构建一个新的计数器， 这个计数器对于每个特定的 IP 地址只会进行一次计数， 我们把这种对每个对象只进行一次计数的计数器称之为唯一计数器（unique counter） 。

代码清单 5-1 展示了一个使用集合实现的唯一计数器， 这个计数器通过把被计数的对象添加到集合来保证每个对象只会被计数一次， 然后通过获取集合的大小来判断计数器目前总共对多少个对象进行了计数。

代码清单 5-1 使用集合实现唯一计数器： `/set/unique_counter.py`

```
class UniqueCounter:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def count_in(self, item):
        """
        尝试将给定元素计入到计数器当中：
        如果给定元素之前没有被计数过，那么方法返回 True 表示此次计数有效；
        如果给定元素之前已经被计数过，那么方法返回 False 表示此次计数无效。
        """
        return self.client.sadd(self.key, item) == 1

    def get_result(self):
        """
        返回计数器的值。
        """
        return self.client.scard(self.key)
```

以下代码展示了如何使用唯一计数器去计算网站的用户数量：

```
>>> from redis import Redis
>>> from unique_counter import UniqueCounter
>>> client = Redis(decode_responses=True)
>>> counter = UniqueCounter(client, 'ip counter')
>>> counter.count_in('8.8.8.8')  # 将一些 IP 地址添加到计数器当中
True
>>> counter.count_in('9.9.9.9')
```

```
True
>>> counter.count_in('10.10.10.10')
True
>>> counter.get_result()          # 获取计数结果
3
>>> counter.count_in('8.8.8.8')  # 添加一个已存在的 IP 地址
False
>>> counter.get_result()          # 计数结果没有发生变化
3
```

5.8 示例：打标签

为了对网站上的内容进行分类标识， 很多网站都提供了打标签（tagging）功能：

- 比如论坛可能会允许用户为帖子添加标签， 这些标签既可以对帖子进行归类， 又可以让其他用户快速地了解到帖子要讲述的内容；
- 又比如说， 一个图书分类网站可能会允许用户为自己收藏的每一本书添加标签， 使得用户可以快速地找到被添加了某个标签的所有图书， 并且网站还可以根据用户的这些标签进行数据分析， 从而帮助用户找到他们可能会感兴趣的图书；
- 除此之外， 购物网站也可以为自己的商品加上标签， 比如“新上架”、“热销中”、“原装进口”等等， 方便顾客了解每件商品的不同特点和属性；

类似的例子还有很多很多。

代码清单 5-2 展示了一个使用集合实现的打标签程序， 通过这个程序， 我们可以为不同的对象添加任意多个标签： 同一个对象的所有标签都会被放到同一个集合里面， 集合里的每一个元素就是一个标签。

代码清单 5-2 使用集合实现的打标签程序： /set/tagging.py

```
def make_tag_key(item):
    return item + "::tags"

class Tagging:

    def __init__(self, client, item):
        self.client = client
        self.key = make_tag_key(item)

    def add(self, *tags):
```

```
        """
        为对象添加一个或多个标签。
        """
        self.client.sadd(self.key, *tags)

    def remove(self, *tags):
        """
        移除对象的一个或多个标签。
        """
        self.client.srem(self.key, *tags)

    def is_included(self, tag):
        """
        检查对象是否带有给定的标签，
        是的话返回 True ，不是的话返回 False 。
        """
        return self.client.sismember(self.key, tag)

    def get_all_tags(self):
        """
        返回对象带有的所有标签。
        """
        return self.client.smembers(self.key)

    def count(self):
        """
        返回对象带有的标签数量。
        """
        return self.client.scard(self.key)
```

以下代码展示了如何使用这个打标签程序去为《The C Programming Language》这本书添加标签：

```
>>> from redis import Redis
>>> from tagging import Tagging
>>> client = Redis(decode_responses=True)
>>> book_tags = Tagging(client, "The C Programming Language")
>>> book_tags.add('c')              # 添加标签
>>> book_tags.add('programming')
>>> book_tags.add('programming language')
>>> book_tags.get_all_tags()        # 查看所有标签
set(['c', 'programming', 'programming language'])
>>> book_tags.count()               # 查看标签的数量
3
```

作为例子， 图 5-7 展示了一些使用打标签程序创建出的集合数据结构。

图 5-7 使用打标签程序创建出的集合

The C Programming Language::tags 集合

"c" "programming" "programming language"

Redis in Action::tags 集合

"redis" "database" "nosql" "manning" "in action series"

Ansi Common Lisp::tags 集合

"lisp" "common lisp" "paul graham"

5.9 示例：点赞

为了让用户表达自己对某一项内容的喜欢和赞赏之情， 很多网站都提供了点赞（like）功能： 通过这一功能， 用户可以给自己喜欢的内容进行点赞， 也可以查看给相同内容进行了点赞的其他用户， 还可以查看给相同内容进行点赞的用户数量， 诸如此类。

除了点赞之外， 很多网站还有诸如“+1”、“顶”、“喜欢”等功能， 这些功能的名字虽然各有不同， 但它们在本质上和点赞功能是一样的。

代码清单 5-3 展示了一个使用集合实现的点赞程序， 这个程序使用集合来储存对内容进行了点赞的用户， 从而确保每个用户只能对同一内容点赞一次， 并通过使用不同的集合命令来实现查看点赞数量、查看所有点赞用户以及取消点赞等功能。

代码清单 5-3 使用集合实现的点赞程序： `/set/like.py`

```
class Like:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def cast(self, user):
        """
        用户尝试进行点赞。
        如果此次点赞执行成功，那么返回 True ；
        如果用户之前已经点过赞，那么返回 False 表示此次点赞无效。
        """
        return self.client.sadd(self.key, user) == 1

    def undo(self, user):
        """
        取消用户的点赞。
        """
        self.client.srem(self.key, user)

    def is_liked(self, user):
        """
        检查用户是否已经点过赞。
        是的话返回 True ，否则的话返回 False 。
        """
        return self.client.sismember(self.key, user)

    def get_all_liked_users(self):
        """
        返回所有已经点过赞的用户。
        """
        return self.client.smembers(self.key)

    def count(self):
        """
        返回已点赞用户的人数。
        """
        return self.client.scard(self.key)
```

以下代码展示了如何使用点赞程序去记录一篇帖子的点赞信息：

```
>>> from redis import Redis
>>> from like import Like
>>> client = Redis(decode_responses=True)
>>> like_topic = Like(client, 'topic::10086::like')
>>> like_topic.cast('peter')          # 用户对帖子进行点赞
True
>>> like_topic.cast('john')
True
>>> like_topic.cast('mary')
True
>>> like_topic.get_all_liked_users()  # 获取所有为帖子点过赞的用户
set(['john', 'peter', 'mary'])
>>> like_topic.count()                # 获取为帖子点过赞的用户数量
3
>>> like_topic.is_liked('peter')      # peter 为帖子点过赞了
True
>>> like_topic.is_liked('dan')        # dan 还没有为帖子点过赞
False
```

5.10 示例：投票

问答网站、文章推荐网站、论坛这类注重内容质量的网站上通常都会提供投票功能， 用户可以通过投票来支持一项内容或者反对一项内容：

- 一项内容获得的支持票数越多， 它就会被网站安排到越显眼的位置， 使得网站的用户可以更快速地浏览到高质量的内容。
- 与此相反， 一项内容获得的反对票数越多， 它就会被网站安排到越不显眼的位置， 甚至被当作广告或者无用内容而被隐藏起来， 使得用户可以忽略这些低质量的内容。

根据网站性质的不同， 不同的网站可能会为投票功能设置不同的称呼， 比如有些网站可能会把“支持”和“反对”叫做“推荐”和“不推荐”， 而有些网站可能会使用“喜欢”和“不喜欢”来表示“支持”和“反对”， 诸如此类， 但这些网站的投票功能在本质上都是一样的。

作为示例， 图 5-8 展示了 StackOverflow 问答网站的一个截图， 这个网站允许用户对问题及其答案进行投票， 从而帮助用户发现高质量的问题和答案。

图 5-8 StackOverflow 网站的投票示例，图中所示的问题获得了 10 个推荐

Questions Tags Users Badg

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free.

Angular + Ionic loading all content via XHR

10

We have an Angular + Ionic app that we are planning on running through Cordova, but having an issue with performance that we are trying to track down.

What we are seeing in Chrome Dev tools Network tab when running either locally or on the built app, is the following:

- Duplicate loading of CSS
- XHR requests to get every single template file our Angular UI router links to, without having visited the routes yet

---

代码清单 5-4 展示了一个使用集合实现的投票程序： 对于每一项需要投票的内容， 这个程序都会使用两个集合来分别储存投支持票的用户以及投反对票的用户， 然后通过对这两个集合执行命令来实现投票、取消投票、统计投票数量、获取已投票用户名单等功能。

代码清单 5-4 使用集合实现的投票程序，用户可以选择支持或者反对一项内容： `/set/vote.py`

```
def vote_up_key(vote_target):
    return vote_target + "::vote_up"

def vote_down_key(vote_target):
    return vote_target + "::vote_down"
```

```python
class Vote:

    def __init__(self, client, vote_target):
        self.client = client
        self.vote_up_set = vote_up_key(vote_target)
        self.vote_down_set = vote_down_key(vote_target)

    def is_voted(self, user):
        """
        检查用户是否已经投过票（可以是赞成票也可以是反对票），
        是的话返回 True ，否则返回 False 。
        """
        return self.client.sismember(self.vote_up_set, user) or \
               self.client.sismember(self.vote_down_set, user)

    def vote_up(self, user):
        """
        让用户投赞成票，并在投票成功时返回 True ；
        如果用户已经投过票，那么返回 False 表示此次投票无效。
        """
        if self.is_voted(user):
            return False

        self.client.sadd(self.vote_up_set, user)
        return True

    def vote_down(self, user):
        """
        让用户投反对票，并在投票成功时返回 True ；
        如果用户已经投过票，那么返回 False 表示此次投票无效。
        """
        if self.is_voted(user):
            return False

        self.client.sadd(self.vote_down_set, user)
        return True

    def undo(self, user):
        """
        取消用户的投票。
        """
        self.client.srem(self.vote_up_set, user)
        self.client.srem(self.vote_down_set, user)

    def vote_up_count(self):
        """
        返回投支持票的用户数量。
        """
```

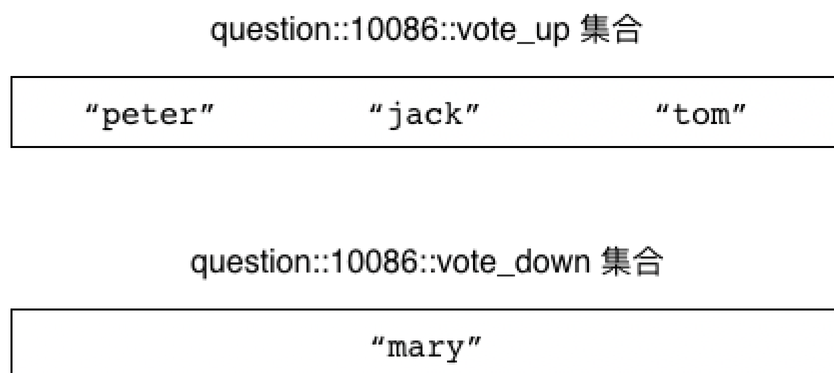
```python
        return self.client.scard(self.vote_up_set)

    def get_all_vote_up_users(self):
        """
        返回所有投支持票的用户。
        """
        return self.client.smembers(self.vote_up_set)

    def vote_down_count(self):
        """
        返回投反对票的用户数量。
        """
        return self.client.scard(self.vote_down_set)

    def get_all_vote_down_users(self):
        """
        返回所有投反对票的用户。
        """
        return self.client.smembers(self.vote_down_set)
```

以下代码展示了如何使用这个投票程序去记录一个问题的投票信息：

```python
>>> from redis import Redis
>>> from vote import Vote
>>> client = Redis(decode_responses=True)
>>> question_vote = Vote(client, 'question::10086')  # 记录问题的投票信息
>>> question_vote.vote_up('peter')    # 投支持票
True
>>> question_vote.vote_up('jack')
True
>>> question_vote.vote_up('tom')
True
>>> question_vote.vote_down('mary')  # 投反对票
True
>>> question_vote.vote_up_count()    # 统计支持票数量
3
>>> question_vote.vote_down_count()  # 统计反对票数量
1
>>> question_vote.get_all_vote_up_users()    # 获取所有投支持票的用户
{'jack', 'peter', 'tom'}
>>> question_vote.get_all_vote_down_users()  # 获取所有投反对票的用户
{'mary'}
```

图 5-9 展示了这段代码创建出的两个集合， 以及这两个集合包含的元素。

图 5-9 投票程序创建出的两个集合



5.11 示例：社交关系

微博、twitter 以及类似的社交网站都允许用户通过加关注或者加好友的方式， 构建一种社交关系： 这些网站上的每个用户都可以关注其他用户， 也可以被其他用户关注。 通过正在关注名单（following list）， 用户可以查看自己正在关注的用户及其人数； 而通过关注者名单（follower llist）， 用户可以查看有哪些人正在关注自己， 以及有多少人正在关注自己。

代码清单 5-5 展示了一个使用集合来记录社交关系的方法：

- 程序为每个用户维持两个集合， 一个集合储存用户的正在关注名单， 而另一个集合则储存用户的关注者名单。
- 当一个用户（关注者）关注另一个用户（被关注者）的时候， 程序会将被关注者添加到关注者的正在关注名单里面， 并将关注者添加到被关注者的关注者名单里面。
- 当关注者取消对被关注者的关注时， 程序会将被关注者从关注者的正在关注名单中移除， 并将关注者从被关注者的关注者名单中移除。

代码清单 5-5 使用集合实现社交关系： /set/relationship.py

```
def following_key(user):
    return user + "::following"
```

```python
def follower_key(user):
    return user + "::follower"

class Relationship:

    def __init__(self, client, user):
        self.client = client
        self.user = user

    def follow(self, target):
        """
        关注目标用户。
        """
        # 把 target 添加到当前用户的正在关注集合里面
        user_following_set = following_key(self.user)
        self.client.sadd(user_following_set, target)
        # 把当前用户添加到 target 的关注者集合里面
        target_follower_set = follower_key(target)
        self.client.sadd(target_follower_set, self.user)

    def unfollow(self, target):
        """
        取消对目标用户的关注。
        """
        # 从当前用户的正在关注集合中移除 target
        user_following_set = following_key(self.user)
        self.client.srem(user_following_set, target)
        # 从 target 的关注者集合中移除当前用户
        target_follower_set = follower_key(target)
        self.client.srem(target_follower_set, self.user)

    def is_following(self, target):
        """
        检查当前用户是否正在关注目标用户，
        是的话返回 True ，否则返回 False 。
        """
        # 如果 target 存在于当前用户的正在关注集合中
        # 那么说明当前用户正在关注 target
        user_following_set = following_key(self.user)
        return self.client.sismember(user_following_set, target)

    def get_all_following(self):
        """
        返回当前用户正在关注的所有人。
        """
        user_following_set = following_key(self.user)
        return self.client.smembers(user_following_set)
```

```python
    def get_all_follower(self):
        """
        返回当前用户的所有关注者。
        """
        user_follower_set = follower_key(self.user)
        return self.client.smembers(user_follower_set)

    def count_following(self):
        """
        返回当前用户正在关注的人数。
        """
        user_following_set = following_key(self.user)
        return self.client.scard(user_following_set)

    def count_follower(self):
        """
        返回当前用户的关注者人数。
        """
        user_follower_set = follower_key(self.user)
        return self.client.scard(user_follower_set)
```

以下代码展示了社交关系程序的基本使用方法：

```python
>>> from redis import Redis
>>> from relationship import Relationship
>>> client = Redis(decode_responses=True)
>>> peter = Relationship(client, 'peter')  # 这个对象记录的是 peter 的社交关系
>>> peter.follow('jack')  # 关注一些人
>>> peter.follow('tom')
>>> peter.follow('mary')
>>> peter.get_all_following()  # 获取目前正在关注的所有人
set(['mary', 'jack', 'tom'])
>>> peter.count_following()    # 统计目前正在关注的人数
3
>>> jack = Relationship(client, 'jack')     # 这个对象记录的是 jack 的社交关系
>>> jack.get_all_follower()    # peter 前面关注了 jack ，所以他是 jack 的关注者
set(['peter'])
>>> jack.count_follower()      # jack 目前只有一个关注者
1
```

图 5-10 展示了以上代码创建的各个集合。

图 5-10 社交关系集合示例

5.12 SRANDMEMBER：随机地获取集合中的元素

通过使用 SRANDMEMBER 命令， 用户可以从集合里面随机地获取指定数量的元素。 SRANDMEMBER 命令接受一个可选的 count 参数， 用于指定用户想要获取的元素数量， 如果用户没有给定这个参数， 那么 SRANDMEMBER 命令默认只获取一个元素：

```
SRANDMEMBER set [count]
```

需要注意的一点是， 被 SRANDMEMBER 命令返回的元素仍然会存在于集合当中， 它们不会被移除。

举个例子， 对于包含以下元素的 databases 集合来说：

```
redis> SMEMBERS databases
1) "Neo4j"
2) "Redis"
3) "PostgreSQL"
4) "CouchDB"
5) "Oracle"
6) "MS SQL"
7) "MongoDB"
8) "MySQL"
```

我们可以使用 SRANDMEMBER 命令去随机地获取集合包含的元素：

```
redis> SRANDMEMBER databases
"MySQL"

redis> SRANDMEMBER databases
"PostgreSQL"

redis> SRANDMEMBER databases
"Neo4j"

redis> SRANDMEMBER databases
"CouchDB"
```

再次提醒， SRANDMEMBER 命令不会移除被返回的集合元素， 这一点可以通过查看 databases 集合包含的元素来确认：

```
redis> SMEMBERS databases  -- 集合包含的元素和执行 SRANDMEMBER 之前完全一样
1) "Neo4j"
2) "Redis"
3) "PostgreSQL"
4) "CouchDB"
5) "Oracle"
6) "MS SQL"
7) "MongoDB"
8) "MySQL"
```

5.12.1 返回指定数量的元素

通过可选的 count 参数， 用户可以指定 SRANDMEMBER 命令返回的元素数量， 其中 count 参数的值既可以是正数也可以是负数。

如果 count 参数的值为正数， 那么 SRANDMEMBER 命令将返回 count 个不重复的元素：

```
redis> SRANDMEMBER databases 2  -- 随机地返回两个不重复的元素
1) "MySQL"
2) "Oracle"

redis> SRANDMEMBER databases 3  -- 随机地返回三个不重复的元素
1) "PostgreSQL"
2) "Oracle"
3) "MS SQL"
```

当 count 参数的值大于集合包含的元素数量时， SRANDMEMBER 命令将返回集合包含的所有元素：

```
redis> SRANDMEMBER databases 10
1) "Neo4j"            -- 因为 databases 集合的元素数量少于 10 个
2) "Redis"            -- 所以命令会返回集合包含的全部 8 个元素
3) "PostgreSQL"
4) "CouchDB"
5) "Oracle"
6) "MongoDB"
7) "MS SQL"
8) "MySQL"
```

另一方面， 如果 count 参数的值为负数， 那么 SRANDMEMBER 命令将随机地返回 abs(count) 个元素 （abs(count) 也即是 count 的绝对值）， 并且在这些元素当中允许出现重复的元素：

```
redis> SRANDMEMBER databases -3  -- 随机地返回三个可能会重复的元素
1) "Neo4j"
2) "CouchDB"
3) "MongoDB"

redis> SRANDMEMBER databases -5  -- 随机地返回五个可能会重复的元素
1) "Neo4j"
2) "MySQL"  -- 出现了两个 "MySQL" 元素
3) "MySQL"
4) "CouchDB"
5) "Oracle"
```

因为 count 参数为负数的 SRANDMEMBER 命令允许返回重复元素， 所以即使 abs(count) 的值大于集合包含的元素数量， SRANDMEMBER 命令也会按照要求返回 abs(count) 个元素：

```
redis> SRANDMEMBER databases -10  -- 随机地返回十个可能会相同的元素
1) "Redis"
2) "MySQL"
3) "CouchDB"
4) "PostgreSQL"
5) "Neo4j"
6) "MS SQL"
7) "MS SQL"
8) "MySQL"
9) "Neo4j"
10) "Redis"
```

5.12.2 其他信息

属性	值
复杂度	O(N)，其中 N 为被返回的元素数量。
版本要求	不带 count 参数的 SRANDMEMBER 命令从 Redis 1.0.0 版本开始可用； 带有 count 参数的 SRANDMEMBER 命令从 Redis 2.6.0 版本开始可用。

5.13 SPOP：随机地从集合里面移除指定数量的元素

通过使用 SPOP 命令， 用户可以从集合里面随机地移除指定数量的元素。 SPOP 命令接受一个可选的 count 参数，用于指定需要被移除的元素数量； 如果用户没有给定这个参数， 那么 SPOP 命令默认只移除一个元素：

```
SPOP key [count]
```

SPOP 命令会返回被移除的元素作为命令的返回值。

举个例子， 对于包含以下元素的 databases 集合来说：

```
redis> SMEMBERS databases
1) "MS SQL"
2) "MongoDB"
3) "Redis"
4) "Neo4j"
5) "PostgreSQL"
6) "MySQL"
```

```
7) "Oracle"
8) "CouchDB"
```

我们可以使用 `SPOP` 命令随机地移除 `databases` 集合中的元素：

```
redis> SPOP databases         -- 随机地移除一个元素
"CouchDB"                     -- 被移除的是 "CouchDB" 元素

redis> SPOP databases         -- 随机地移除一个元素
"Redis"                       -- 被移除的是 "Redis" 元素

redis> SPOP databases 3       -- 随机地移除三个元素
1) "Neo4j"                    -- 被移除的元素是 "Neo4j" 、 "PostgreSQL" 和 "MySQL"
2) "PostgreSQL"
3) "MySQL"
```

图 5-11 展示了 `databases` 集合在执行各个 `SPOP` 命令时的变化过程。

图 5-11 `databases` 集合在执行 `SPOP` 命令时的变化过程

执行 `SPOP` 命令之前的 `databases` 集合

database 集合

“Redis”	“MongoDB”	“CouchDB”
“MySQL”	“PostgreSQL”	“Oracle”
“Neo4j”	“MS SQL”	

执行 `SPOP databases` ，导致元素 `"CouchDB"` 被移除

database 集合

"Redis" "MongoDB"
"MySQL" "PostgreSQL" "Oracle"
"Neo4j" "MS SQL"

执行 `SPOP databases` ，导致元素 `"Redis"` 被移除

database 集合

"MongoDB"
"MySQL" "PostgreSQL" "Oracle"
"Neo4j" "MS SQL"

执行 `SPOP databases 3` ，导致元素 `"Neo4j"` 、 `"PostgreSQL"` 和 `"MySQL"` 被移除

database 集合

"MongoDB"
"Oracle"
"MS SQL"

5.13.1 SPOP 与 SRANDMEMBER 的区别

SPOP 命令和 SRANDMEMBER 命令的主要区别在于： SPOP 命令会移除被随机选中的元素， 而 SRANDMEMBER 命令则不会移除被随机选中的元素。

通过查看 databases 集合目前包含的元素， 我们可以证实之前被 SPOP 命令选中的元素已经不再存在于集合当中：

```
redis> SMEMBERS databases
1) "MS SQL"
2) "MongoDB"
3) "Oracle"
```

SPOP 命令和 SRANDMEMBER 命令之间的另一个不同在于， SPOP 命令只接受正数 count 值， 尝试向 SPOP 命令提供负数 count 值将引发错误， 因为负数 count 值对于 SPOP 命令是没有意义的：

```
redis> SPOP databases -3
(error) ERR index out of range
```

5.13.2 其他信息

属性	值
复杂度	O(N)，其中 N 为被移除的元素数量。
版本要求	不带 count 参数的 SPOP 命令从 Redis 1.0.0 版本开始可用； 带有 count 参数的 SPOP 命令从 Redis 3.2.0 版本开始可用。

5.14 示例：抽奖

为了推销商品并反馈消费者， 商家经常会举办一些抽奖活动， 每个符合条件的消费者都可以参加这种抽奖， 而商家则需要从所有参加抽奖的消费者里面选出指定数量的获奖者， 并给他们赠送物品、金钱或者购物优惠。

代码清单 5-6 展示了一个使用集合实现的抽奖程序， 这个程序会把所有参与抽奖活动的玩家都添加到一个集合里面， 然后通过 SRANDMEMBER 命令随机地选出获奖者。

代码清单 5-6 使用集合实现的抽奖程序：/set/lottery.py

```
class Lottery:

    def __init__(self, client, key):
        self.client = client
        self.key = key
```

```python
    def add_player(self, user):
        """
        将用户添加到抽奖名单当中。
        """
        self.client.sadd(self.key, user)

    def get_all_players(self):
        """
        返回参加抽奖活动的所有用户。
        """
        return self.client.smembers(self.key)

    def player_count(self):
        """
        返回参加抽奖活动的用户人数。
        """
        return self.client.scard(self.key)

    def draw(self, number):
        """
        抽取指定数量的获奖者。
        """
        return self.client.srandmember(self.key, number)
```

考虑到保留完整的抽奖者名单可能会有用， 所以这个抽奖程序使用了随机地获取元素的 SRANDMEMBER 命令而不是随机地移除元素的 SPOP 命令； 在不需要保留完整的抽奖者名单的情况下， 我们也可以使用 SPOP 命令去实现抽奖程序。

以下代码简单地展示了这个抽奖程序的使用方法：

```python
>>> from redis import Redis
>>> from lottery import Lottery
>>> client = Redis(decode_responses=True)
>>> lottery = Lottery(client, 'birthday party lottery')  # 这是一次生日派对抽奖活动
>>> lottery.add_player('peter')  # 添加抽奖者
>>> lottery.add_player('jack')
>>> lottery.add_player('tom')
>>> lottery.add_player('mary')
>>> lottery.add_player('dan')
>>> lottery.player_count()       # 查看抽奖者数量
5
>>> lottery.draw(1)  # 抽取一名获奖者
['dan']              # dan 中奖了！
```

5.15 SINTER、SINTERSTORE：对集合执行交集计算

SINTER 命令可以计算出用户给定的所有集合的交集， 然后返回这个交集包含的所有元素：

```
SINTER set [set ...]
```

比如对于以下这两个集合来说：

```
redis> SMEMBERS s1
1) "a"
2) "b"
3) "c"
4) "d"

redis> SMEMBERS s2
1) "c"
2) "d"
3) "e"
4) "f"
```

我们可以通过执行以下命令， 计算出这两个集合的交集：

```
redis> SINTER s1 s2
1) "c"
2) "d"
```

从结果可以看出， s1 和 s2 的交集包含了 "c" 和 "d" 这两个元素。

5.15.1 SINTERSTORE 命令

除了 SINTER 命令之外， Redis 还提供了 SINTERSTORE 命令， 这个命令可以把给定集合的交集计算结果储存到指定的键里面：

```
SINTERSTORE destination_key set [set ...]
```

如果给定的键已经存在， 那么 SINTERSTORE 命令在执行储存操作之前会先删除已有的键。 SINTERSTORE 命令在执行完毕之后会返回被储存的交集元素数量作为返回值。

比如说， 通过执行以下命令， 我们可以把 s1 和 s2 的交集计算结果储存到集合 s1-inter-s2 里面：

```
redis> SINTERSTORE s1-inter-s2 s1 s2
(integer) 2  -- 交集包含两个元素

redis> SMEMBERS s1-inter-s2
1) "c"
2) "d"
```

5.15.2 其他信息

属性	值
复杂度	SINTER 命令和 SINTERSTORE 命令的复杂度都是 O(N*M)， 其中 N 为给定集合的数量， 而 M 则是所有给定集合当中， 包含元素最少的那个集合的大小。
版本要求	SINTER 命令和 SINTERSTORE 命令从 Redis 1.0.0 版本开始可用。

5.16 SUNION、SUNIONSTORE：对集合执行并集计算

SUNION 命令可以计算出用户给定的所有集合的并集， 然后返回这个并集包含的所有元素：

```
SUNION set [set ...]
```

比如对于以下这两个集合来说：

```
redis> SMEMBERS s1
1) "a"
2) "b"
3) "c"
4) "d"

redis> SMEMBERS s2
1) "c"
2) "d"
3) "e"
4) "f"
```

我们可以通过执行以下命令， 计算出这两个集合的并集：

```
redis> SUNION s1 s2
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
```

从结果可以看出， s1 和 s2 的并集共包含六个元素。

5.16.1 SUNIONSTORE 命令

跟 SINTERSTORE 命令类似， Redis 也为 SUNION 提供了相应的 SUNIONSTORE 命令， 这个命令可以把给定集合的并集计算结果储存到指定的键里面， 并在键已经存在的情况下， 自动覆盖已有的键：

```
SUNIONSTORE destination_key set [set ...]
```

SUNIONSTORE 命令在执行完毕之后， 将返回并集元素的数量作为返回值。

比如说， 通过执行以下命令， 我们可以把 s1 和 s2 的并集计算结果储存到集合 s1-union-s2 里面：

```
redis> SUNIONSTORE s1-union-s2 s1 s2
(integer) 6   -- 并集共包含六个元素

redis> SMEMBERS s1-union-s2
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
```

5.16.2 其他信息

属性	值
复杂度	SUNION 命令和 SUNIONSTORE 命令的复杂度都是 O(N) ， 其中 N 为所有给定集合包含的元素数量总和。
版本要求	SUNION 命令和 SUNIONSTORE 命令从 Redis 1.0.0 版本开始可用。

5.17 SDIFF、SDIFFSTORE：对集合执行差集计算

SDIFF 命令可以计算出给定集合之间的差集， 并返回差集包含的所有元素：

```
SDIFF set [set ...]
```

SDIFF 命令会按照用户给定集合的顺序， 从左到右依次地对给定的集合执行差集计算。

举个例子， 对于以下这三个集合来说：

```
redis> SMEMBERS s1
1) "a"
2) "b"
3) "c"
4) "d"

redis> SMEMBERS s2
1) "c"
2) "d"
3) "e"
4) "f"

redis> SMEMBERS s3
1) "b"
2) "f"
3) "g"
```

如果我们执行以下命令：

```
redis> SDIFF s1 s2 s3
1) "a"
```

那么 SDIFF 命令首先会对集合 s1 和集合 s2 执行差集计算， 得到一个包含元素 "a" 和 "b" 的临时集合， 然后再使用这个临时集合与集合 s3 执行差集计算。 换句话说， 这个 SDIFF 命令首先会计算出 s1-s2 的结果， 然后再计算 (s1-s2)-s3 的结果。

5.17.1 SDIFFSTORE 命令

跟 SINTERSTORE 命令和 SUNIONSTORE 命令一样， Redis 也为 SDIFF 命令提供了相应的 SDIFFSTORE 命令， 这个命令可以把给定集合之间的差集计算结果储存到指定的键里面， 并在键已经存在的情况下， 自动覆盖已有的键：

```
SDIFFSTORE destination_key set [set ...]
```

SDIFFSTORE 命令会返回被储存的差集元素数量作为返回值。

作为例子， 以下代码展示了怎样将集合 s1 、 s2 、 s3 的差集计算结果储存到集合 diff-result 里面：

```
redis> SDIFFSTORE diff-result s1 s2 s3
(integer) 1   -- 计算出的差集只包含一个元素

redis> SMEMBERS diff-result
1) "a"
```

5.17.2 其他信息

属性	值
复杂度	SDIFF 命令和 SDIFFSTORE 命令的复杂度都是 O(N) ， 其中 N 为所有给定集合包含的元素数量总和。
版本要求	SDIFF 命令和 SDIFFSTORE 命令从 Redis 1.0.0 版本开始可用。

Note: 执行集合计算的注意事项

因为对集合执行交集、并集、差集等集合计算需要耗费大量的资源， 所以在有可能的情况下， 用户都应该尽量使用 SINTERSTORE 等命令来储存并重用计算结果， 而不要每次都重复进行计算。

此外， 当集合计算涉及的元素数量非常巨大时， Redis 服务器在进行计算时可能会被阻塞。 这时， 我们可以考虑使用 Redis 的复制功能， 通过从服务器来执行集合计算任务， 从而确保主服务器可以继续处理其他客户端发送的命令请求。

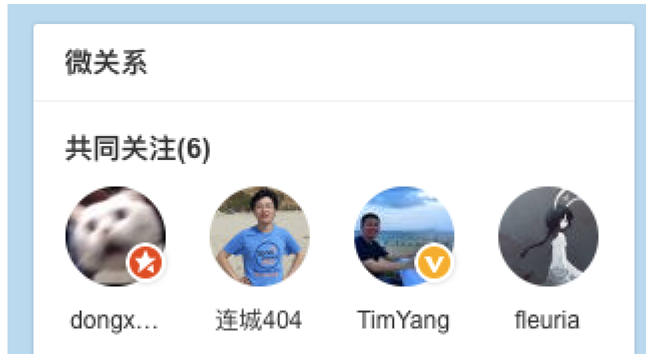
本书将在之后的《复制》一章中对 Redis 的复制功能进行介绍。

5.18 示例：共同关注与推荐关注

在前面的内容中， 我们学习了如何使用集合去储存社交网站的好友关系， 但是除了基本的关注和被关注之外， 社交网站通常还会提供一些额外的功能， 帮助用户去发现一些自己可能会感兴趣的人。

比如说， 当我们在微博上访问某个用户的个人页面时， 页面上就会展示出我们和这个用户都在关注的人， 就像图 5-12 所示那样。

图 5-12 微博上的共同关注示例



除了共同关注之外， 一些社交网站还会通过算法和数据分析， 为用户推荐一些他可能会感兴趣的人， 比如图 5-13 就展示了 twitter 是如何向用户推荐他可能会感兴趣的关注对象的。

图 5-13 twitter 的推荐关注功能示例

在接下来的两个小节中， 我们将分别学习如何使用集合去实现以上展示的共同关注功能和推荐关注功能。

5.18.1 共同关注

要实现共同关注功能， 程序需要做的就是计算出两个用户的正在关注集合之间的交集， 这一点可以通过前面介绍的 SINTER 命令和 SINTERSTORE 命令来完成， 代码清单 5-7 展示了使用这一原理实现的共同关注程序。

代码清单 5-7 共同关注功能的实现： /set/common_following.py

```
def following_key(user):
    return user + "::following"

class CommonFollowing:

    def __init__(self, client):
        self.client = client

    def calculate(self, user, target):
        """
        计算并返回当前用户和目标用户共同关注的人。
        """
        user_following_set = following_key(user)
        target_following_set = following_key(target)
        return self.client.sinter(user_following_set, target_following_set)
```

```python
    def calculate_and_store(self, user, target, store_key):
        """
        计算出当前用户和目标用户共同关注的人，
        并把结果储存到 store_key 指定的键里面，
        最后返回共同关注的人数作为返回值。
        """
        user_following_set = following_key(user)
        target_following_set = following_key(target)
        return self.client.sinterstore(store_key, user_following_set, target_following_set)
```

以下代码展示了共同关注程序的具体用法：

```python
>>> from redis import Redis
>>> from relationship import Relationship
>>> from common_following import CommonFollowing
>>> client = Redis(decode_responses=True)
>>> peter = Relationship(client, "peter")
>>> jack = Relationship(client, "jack")
>>> peter.follow("tom")   # peter 关注一些用户
>>> peter.follow("david")
>>> peter.follow("mary")
>>> jack.follow("tom")    # jack 关注一些用户
>>> jack.follow("david")
>>> jack.follow("lily")
>>> common_following = CommonFollowing(client)
>>> common_following.calculate("peter", "jack")  # 计算 peter 和 jack 的共同关注用户
set(['tom', 'david'])                             # 他们都关注了 tom 和 david
```

5.18.2 推荐关注

代码清单 5-8 展示了一个推荐关注程序的实现代码， 这个程序会从用户的正在关注集合中随机地选出指定数量的人作为种子用户， 然后对这些种子用户的正在关注集合执行并集计算， 最后再从这个并集里面随机地选出一些人作为推荐关注的对象。

代码清单 5-8 推荐关注功能的实现： `/set/recommend_follow.py`

```python
def following_key(user):
    return user + "::following"
```

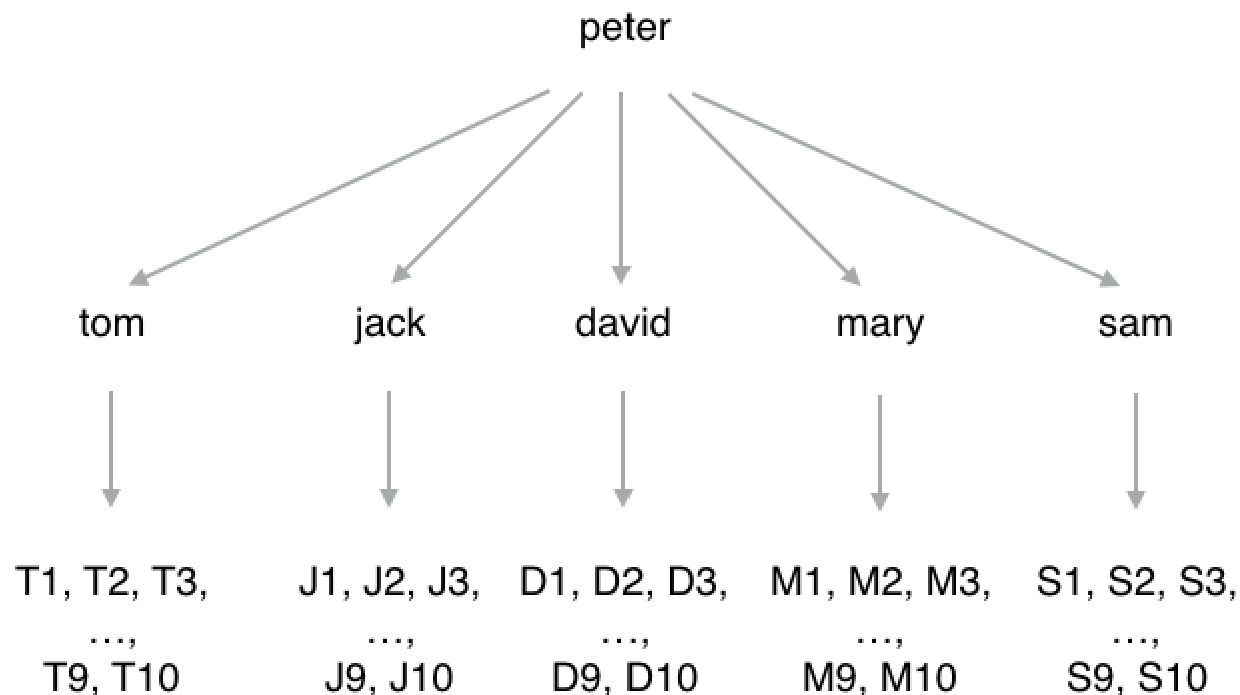
```python
def recommend_follow_key(user):
    return user + "::recommend_follow"

class RecommendFollow:

    def __init__(self, client, user):
        self.client = client
        self.user = user

    def calculate(self, seed_size):
        """
        计算并储存用户的推荐关注数据。
        """
        # 1)从用户关注的人中随机选一些人作为种子用户
        user_following_set = following_key(self.user)
        following_targets = self.client.srandmember(user_following_set, seed_size)
        # 2)收集种子用户的正在关注集合键名
        target_sets = set()
        for target in following_targets:
            target_sets.add(following_key(target))
        # 3)对所有种子用户的正在关注集合执行并集计算，并储存结果
        return self.client.sunionstore(recommend_follow_key(self.user), *target_sets)

    def fetch_result(self, number):
        """
        从已有的推荐关注数据中随机地获取指定数量的推荐关注用户。
        """
        return self.client.srandmember(recommend_follow_key(self.user), number)

    def delete_result(self):
        """
        删除已计算出的推荐关注数据。
        """
        self.client.delete(recommend_follow_key(self.user))
```

以下代码展示了这个推荐关注程序的使用方法：

```python
>>> from redis import Redis
>>> from recommend_follow import RecommendFollow
>>> client = Redis(decode_responses=True)
>>> recommend_follow = RecommendFollow(client, "peter")
>>> recommend_follow.calculate(3)        # 随机选择 3 个正在关注的人作为种子用户
30
>>> recommend_follow.fetch_result(10)    # 获取 10 个推荐关注对象
['D6', 'M0', 'S4', 'M1', 'S8', 'M3', 'S3', 'M7', 'M4', 'D7']
```

在执行这段代码之前， 用户 peter 关注了 tom 、 david 、 jack 、 mary 和 sam 这五个用户， 而这五个用户又分别关注了如图 5-14 所示的一些用户， 从结果来看， 推荐程序随机选中了 david 、 sam 和 mary 作为种子用户，然后又从这三个用户的正在关注集合的并集中， 随机地选出了 10 个人作为 peter 的推荐关注对象。

图 5-14 peter 的正在关注关系图



需要注意的是， 这里展示的推荐关注程序使用的是非常简单的推荐算法， 它假设用户会对自己正在关注的人的关注对象感兴趣， 但实际的情况可能并非如此。 为了获得更为精准的推荐效果， 实际的社交网站通常会使用更为复杂的推荐算法， 有兴趣的读者可以自行查找这方面的资料。

5.19 示例：使用反向索引构建商品筛选器

在光顾网店或者购物网站的时候， 我们经常会看见图 5-15 这样的商品筛选器， 对于不同的条件， 这些筛选器会给出不同的选项， 用户可以通过点击不同的选项来快速找到自己想要的商品。

图 5-15 笔记本电脑商品筛选器

电脑、办公 > 电脑整机 > 笔记本 >

笔记本 商品筛选 共 14736个商品

品牌： Lenovo ThinkPad 戴尔 ASUS 惠普 Authorized Reseller 授权经销商 acer SAMSUNG ALIENWARE dere Hasee 神舟 FUJITSU 富士通

价格： 0-2399 2400-3499 3500-3899 3900-5199 5200-6099 6100-7699 7700以上 确定

适用人群： 游戏达人 校园学生 时尚超薄 商务办公 高清影音 其他

尺寸： 13.3英寸 14.0英寸 15.6英寸 17.3英寸 12.5英寸 11.6英寸 10.1英寸及以下 其他

处理器： Intel i7 Intel i5 Intel i3 Intel 其他 Intel 其它 AMD FX AMD A10 AMD A8 AMD A6 AMD其他

更多选项（系统、显卡、内存容量 等）

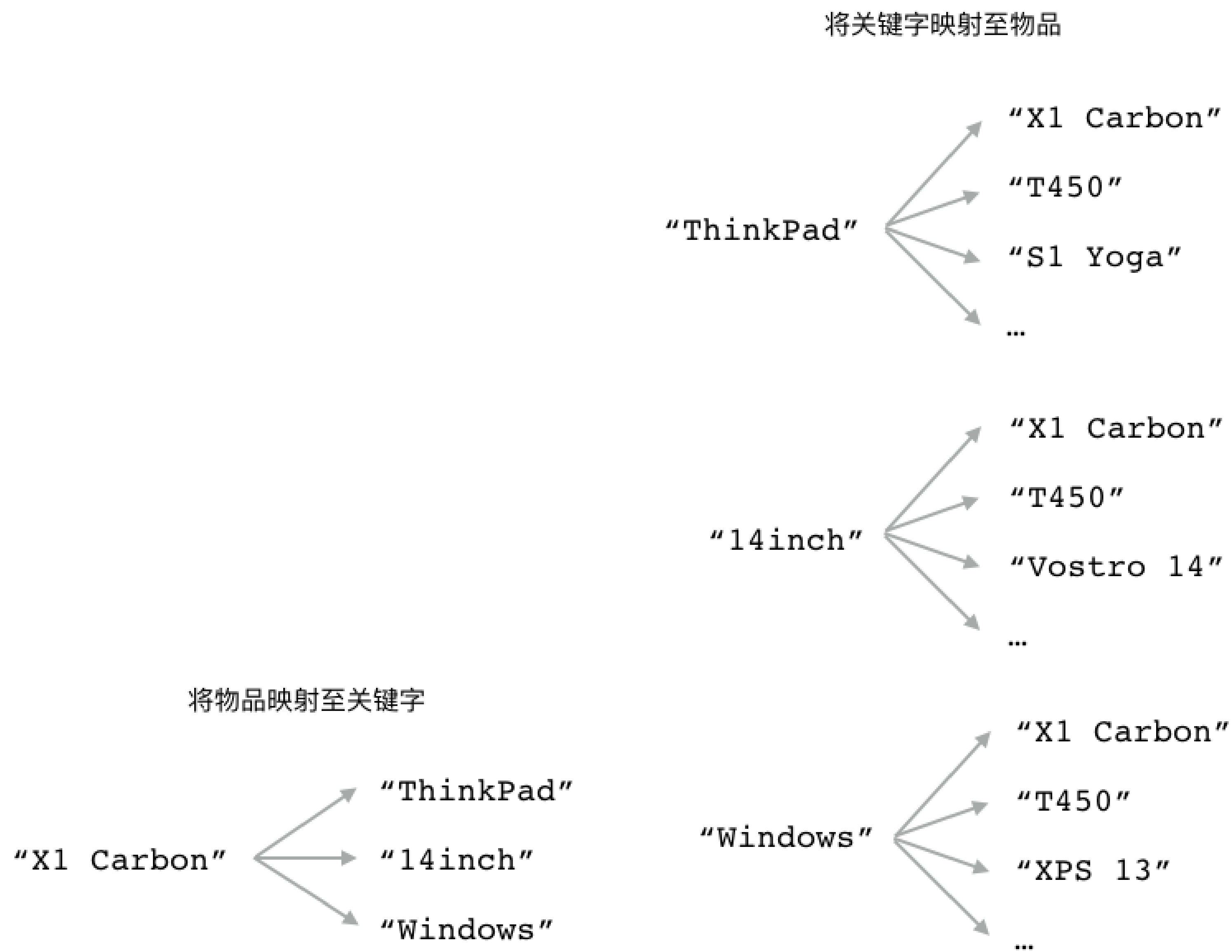
比如对于图 5-15 展示的笔记本电脑筛选器来说， 如果我们点击图中“品牌”一栏的“ThinkPad”图标， 那么筛选器将只在页面里展示 ThinkPad 品牌的笔记本电脑。 如果我们继续点击“尺寸”一栏中的“13.3英寸”选项， 那么筛选器将只在页面里展示 ThinkPad 品牌的 13.3 英寸笔记本电脑， 诸如此类。

实现商品筛选器的其中一种方法是使用反向索引， 这种数据结构可以为每个物品添加多个关键字， 然后根据关键字去反向地获取相应的物品。 举个例子， 对于 `"X1 Carbon"` 这台笔记本电脑来说， 我们可以为它添加 `"ThinkPad"` 、 `"14inch"` 、 `"Windows"` 等关键字， 然后通过这些关键字来反向获取 `"X1 Carbon"` 这台电脑。

实现反向索引的关键是要在物品和关键字之间构建起双向的映射关系， 比如对于刚刚提到的 `"X1 Carbon"` 电脑来说， 反向索引程序需要构建出图 5-16 所示的两种映射关系：

- 第一种映射关系将 `"X1 Carbon"` 映射至它带有的各个关键字；
- 而第二种映射关系则将 `"ThinkPad"` 、 `"14inch"` 、 `"Windows"` 等多个关键字映射至 `"X1 Carbon"` 。

图 5-16 X1 Carbon 电脑及其关键字的映射关系



代码清单 5-9 展示了一个使用集合实现的反向索引程序， 对于用户给定的每一件物品， 这个程序都会使用一个集合去储存物品带有的多个关键字； 与此同时， 对于这件物品的每一个关键字， 程序都会使用一个集合去储存关

键字与物品之间的映射。 因为构建反向索引所需的这两种映射都是一对多映射， 所以使用集合来储存这两种映射关系的做法是可行的。

代码清单 5-9 反向索引程序： `/set/inverted_index.py`

```python
def make_item_key(item):
    return "InvertedIndex::" + item + "::keywords"

def make_keyword_key(keyword):
    return "InvertedIndex::" + keyword + "::items"

class InvertedIndex:

    def __init__(self, client):
        self.client = client

    def add_index(self, item, *keywords):
        """
        为物品添加关键字。
        """
        # 将给定关键字添加到物品集合中
        item_key = make_item_key(item)
        result = self.client.sadd(item_key, *keywords)
        # 遍历每个关键字集合，把给定物品添加到这些集合当中
        for keyword in keywords:
            keyword_key = make_keyword_key(keyword)
            self.client.sadd(keyword_key, item)
        # 返回新添加关键字的数量作为结果
        return result

    def remove_index(self, item, *keywords):
        """
        移除物品的关键字。
        """
        # 将给定关键字从物品集合中移除
        item_key = make_item_key(item)
        result = self.client.srem(item_key, *keywords)
        # 遍历每个关键字集合，把给定物品从这些集合中移除
        for keyword in keywords:
            keyword_key = make_keyword_key(keyword)
            self.client.srem(keyword_key, item)
        # 返回被移除关键字的数量作为结果
        return result

    def get_keywords(self, item):
```

```
        """
        获取物品的所有关键字。
        """
        return self.client.smembers(make_item_key(item))

    def get_items(self, *keywords):
        """
        根据给定的关键字获取物品。
        """
        # 根据给定的关键字，计算出与之对应的集合键名
        keyword_key_list = map(make_keyword_key, keywords)
        # 然后对这些储存着各式物品的关键字集合执行并集计算
        # 从而查找出带有给定关键字的物品
        return self.client.sinter(*keyword_key_list)
```

为了测试这个反向索引程序， 我们在以下代码中， 把一些笔记本电脑产品的名称及其关键字添加到了反向索引里面：

```
>>> from redis import Redis
>>> from inverted_index import InvertedIndex
>>> client = Redis(decode_responses=True)
>>> laptops = InvertedIndex(client)
>>> laptops.add_index("MacBook Pro", "Apple", "MacOS", "13inch")  # 为电脑及其关键字建立索引
3
>>> laptops.add_index("MacBook Air", "Apple", "MacOS", "13inch")
3
>>> laptops.add_index("X1 Carbon", "ThinkPad", "Windows", "13inch")
3
>>> laptops.add_index("T450", "ThinkPad", "Windows", "14inch")
3
>>> laptops.add_index("XPS", "DELL", "Windows", "13inch")
3
```

在此之后， 我们可以通过以下语句来找出 `"T450"` 电脑带有的所有关键字：

```
>>> laptops.get_keywords("T450")
set(['Windows', '14inch', 'ThinkPad'])
```

也可以使用以下语句来找出所有屏幕大小为 13 英寸的笔记本电脑：

```
>>> laptops.get_items("13inch")
set(['MacBook Pro', 'X1 Carbon', 'MacBook Air', 'XPS'])
```

还可以使用以下语句来找出所有屏幕大小为 13 英寸并且使用 Windows 系统的笔记本电脑：

```
>>> laptops.get_items("13inch", "Windows")
set(['XPS', 'X1 Carbon'])
```

或者使用以下语句来找出所有屏幕大小为 13 英寸并且使用 Windows 系统的 ThinkPad 品牌笔记本电脑：

```
>>> laptops.get_items("13inch", "Windows", "ThinkPad")
set(['X1 Carbon'])
```

图 5–17 展示了以上代码在数据库中为物品创建的各个集合， 而图 5–18 则展示了以上代码在数据库中为关键字创建的各个集合。

图 5–17 反向索引程序为物品创建的集合

InvertedIndex::MacBook Pro::keywords 集合

“Apple” “MacOS” “13inch”

InvertedIndex::MacBook Air::keywords 集合

“Apple” “MacOS” “13inch”

InvertedIndex::X1 Carbon::keywords 集合

“ThinkPad” “Windows” “13inch”

InvertedIndex::T450::keywords 集合

“ThinkPad” “Windows” “14inch”

InvertedIndex::XPS::keywords 集合

“DELL” “Windows” “13inch”

图 5-18 反向索引程序为关键字创建的集合

InvertedIndex::DELL::items 集合

“XPS”

InvertedIndex::ThinkPad::items 集合

“X1 Carbon”	“T450”

InvertedIndex::Apple::items 集合

“MacBook Pro”	“MacBook Air”

InvertedIndex::MacOS::items 集合

“MacBook Pro”	“MacBook Air”

InvertedIndex:: Windows::items 集合

“XPS”	“X1 Carbon”	“T450”

InvertedIndex:: 13inch::items 集合

“MacBook Pro”	“X1 Carbon”	“MacBook Air”	“XPS”

InvertedIndex:: 14inch::items 集合

“T450”

5.20 重点回顾

- 集合允许用户储存任意多个各不相同的元素。

- 所有针对单个元素的集合操作，复杂度都为 O(1) 。
- 在使用 SADD 命令向集合中添加元素时， 已存在于集合中的元素会自动被忽略。
- 因为集合以无序的方式储存元素， 所以两个包含相同元素的集合在使用 SMEMBERS 命令时可能会得到不同的结果。
- SRANDMEMBER 命令不会移除被随机选中的元素， 而 SPOP 命令的做法则与此相反。
- 因为集合计算需要使用大量的计算资源， 所以我们应该尽量储存并重用集合计算的结果， 在有需要的情况下， 还可以把集合计算放到从服务器中进行。

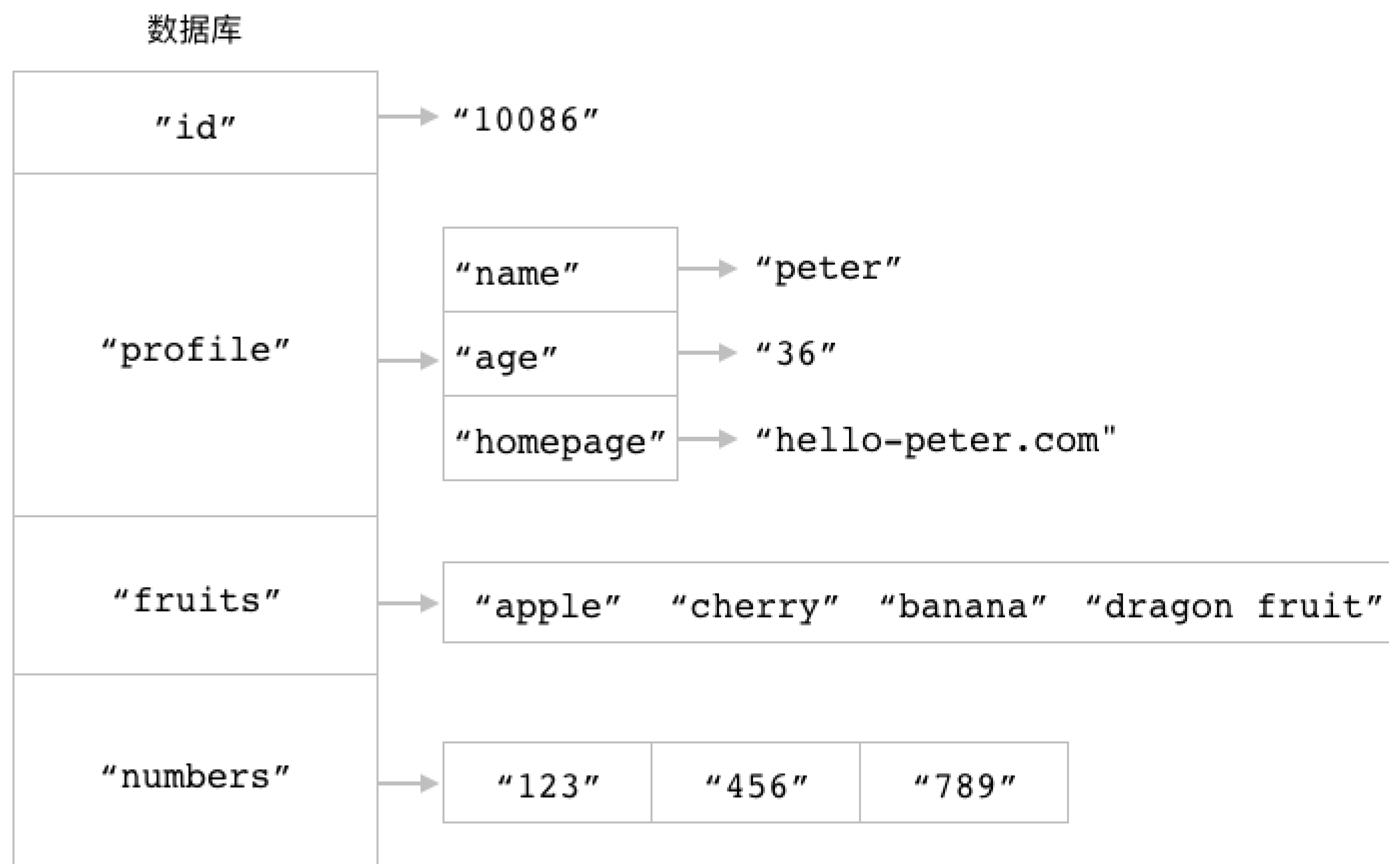
11. 数据库

在前面的章节中， 我们学习了如何使用不同的 Redis 命令去创建各种不同类型的键， 比如使用 SET 命令去创建字符串键， 使用 HSET 命令去创建散列键， 又或者使用 RPUSH 和 LPUSH 命令去创建列表键， 诸如此类。

但无论字符串键也好， 散列键又或者列表键也好， 它们都会被储存到一个名为数据库的容器里面。 因为 Redis 是一个键值对数据库服务器， 所以它的数据库跟我们之前介绍过的散列键一样， 都可以根据键的名字对数据库中的键值对进行索引： 比如说， 通过使用 Redis 提供的命令， 我们可以从数据库中移除指定的键， 又或者将指定的键从一个数据库移动到另一个数据库， 诸如此类。

作为例子， 图 11-1 展示了一个包含四个键的数据库， 其中 `id` 为字符串键， `profile` 为散列键， `fruits` 为集合键， 而 `numbers` 则为列表键。

图 11-1 一个数据库示例



Redis 为数据库提供了非常丰富的操作命令， 通过这些命令， 用户可以：

- 指定自己想要使用的数据库。
- 一次性获取数据库包含的所有键， 迭代地获取数据库包含的所有键， 又或者随机地获取数据库中的某个键。
- 根据给定键的值进行排序。
- 检查给定的一个或多个键，看它们是否存在于数据库当中。
- 查看给定键的类型。
- 对给定键进行改名。
- 移除指定的键， 又或者将它从一个数据库移动到另一个数据库。
- 清空数据库包含的所有键。

- 交换给定的两个数据库。

本章接下来的内容将对以上提到的各个命令进行介绍， 并说明如何使用这些命令去实现诸如数据库迭代器和数据库取样器这样的实用程序。

11.1 SELECT：切换至指定的数据库

一个 Redis 服务器可以包含多个数据库。 在默认情况下， Redis 服务器在启动时将会创建 16 个数据库： 这些数据库都使用号码进行标识， 其中第一个数据库为 0 号数据库， 第二个数据库为 1 号数据库， 而第三个数据库则为 2 号数据库， 以此类推。

Redis 虽然不允许在同一个数据库里面使用两个同名的键， 但是由于不同数据库拥有不同的命名空间， 因此在不同数据库里面使用同名的键是完全没有问题的， 而用户也可以通过使用不同数据库来储存不同的数据， 以此来达到重用键名并且减少键冲突的目的。

比如说， 如果我们将用户的个人信息和会话信息都存放在同一个数据库里面， 那么为了区分这两种信息， 程序就需要使用 `user::<id>::profile` 格式的键来储存用户信息， 并使用 `user::<id>::session` 格式的键来储存用户会话； 但如果我们将这两种信息分别储存在 0 号数据库和 1 号数据库里面， 那么程序就可以在 0 号数据库中使用 `user::<id>` 格式的键来储存用户信息， 并在 1 号数据库中继续使用 `user::<id>` 格式的键来储存用户会话。

当用户使用客户端与 Redis 服务器进行连接时， 客户端一般默认都会使用 0 号数据库， 但是通过使用 `SELECT` 命令， 用户可以从当前正在使用的数据库切换到自己想要使用的数据库：

```
SELECT db
```

`SELECT` 命令在切换成功之后将返回 `OK` 。

举个例子， 当我们以默认方式启动 `redis-cli` 客户端时， `redis-cli` 将连接至服务器的 0 号数据库：

```
$ redis-cli
redis>
```

这时， 如果我们想要从 0 号数据库切换至 3 号数据库， 那么只需要执行以下命令即可：

```
redis> SELECT 3
OK
```

```
redis[3]>
```

客户端提示符末尾的 `[3]` 表示客户端现在正在使用 3 号数据库。 （`redis-cli` 在使用默认的 0 号数据库时不会打印出数据库号码。）

在此之后， 我们就可以通过执行命令， 对 3 号数据库进行设置了：

```
redis[3]> SET msg "hello world"     -- 在 3 号数据库创建一个 msg 键
OK

redis[3]> SET counter 10086         -- 在 3 号数据库创建一个 counter 键
OK
```

11.1.1 其他信息

属性	值
复杂度	O(1)
版本要求	`SELECT` 命令从 Redis 1.0.0 版本开始可用。

11.2 KEYS：获取所有与给定匹配符相匹配的键

`KEYS` 命令接受一个全局匹配符作为参数， 然后返回数据库里面所有与这个匹配符相匹配的键作为结果：

```
KEYS pattern
```

举个例子， 如果我们想要获取数据库包含的所有键， 那么可以执行以下命令：

```
redis> KEYS *
1) "fruits"
2) "user::12312::profile"
3) "user::ip"
4) "user::id"
5) "cache::/user/peter"
6) "todo-list"
```

又或者说， 如果我们想要获取所有以 `user::` 为前缀的键， 那么可以执行以下命令：

```
redis> KEYS user::*
1) "user::12312::profile"
2) "user::ip"
3) "user::id"
```

最后， 如果数据库里面没有任何键与给定的匹配符相匹配， 那么 KEYS 命令将返回一个空值：

```
redis> KEYS article::*
(empty list or set)    -- 数据库里面没有任何以 article:: 为前缀的键
```

11.2.1全局匹配符

KEYS 命令允许使用多种不同的全局匹配符作为 pattern 参数的值， 表 11-1 展示了一些常见的全局匹配符， 并举例说明了这些匹配符的作用。

表 11-1 Glob 匹配符的作用及其示例

匹配符	作用	例子
*	匹配零个或任意多个任意字符	user::* 可以匹配任何以 user:: 为前缀的字符串，比如 user::ip 、 user::12312::profile 等等，以及 user:: 本身； *z 可以匹配任何以字母 z 结尾的字符串，比如 antirez 、 matz 和 huangz ，以及字母 z 本身； *::* 可以匹配任何使用了 :: 作为间隔符的字符串，比如 user::ip 和 cache::/user/peter ，但不能匹配 todo-list 。
?	匹配任意的单个字符	user::i? 可以匹配任何以 user::i 为前缀，后跟单个字符的字符串，比如 user::ip 、 user::id 等，但不能匹配 user::ime 。
[]	匹配给定字符串中的单个字符	user::i[abc] 可以匹配 user::ia 、 user::ib 和 user::ic ，但不能匹配 user::id 或者 user::ime ，诸如此类。
[?-?]	匹配给定范围中的单个字符	user::i[a-d] 可以匹配 user::ia 、 user::ib 、 user::ic 和 user::id ，但不能匹配除此以外的其他字符串， 比如 user::ip 或者 user::ime 。

关于全局匹配符的更多信息可以参考维基百科 https://en.wikipedia.org/wiki/Glob_(programming) 或者 glob 程序的手册页面： http://man7.org/linux/man-pages/man7/glob.7.html 。

11.2.2 其他信息

属性	值
复杂度	O(N)，其中 N 为数据库包含的键数量。
版本要求	KEYS 命令从 Redis 1.0.0 版本开始可用。

11.3 SCAN：以渐进方式迭代数据库中的键

因为 KEYS 命令需要检查数据库包含的所有键， 并一次性将符合条件的所有键全部返回给客户端， 所以当数据库包含的键数量比较大时， 使用 KEYS 命令可能会导致服务器被阻塞。

为了解决这个问题， Redis 从 2.8.0 版本开始提供 SCAN 命令， 该命令是一个迭代器， 它每次被调用的时候都会从数据库里面获取一部分键， 用户可以通过重复调用 SCAN 命令来迭代数据库包含的所有键：

```
SCAN cursor
```

SCAN 命令的 cursor 参数用于指定迭代时使用的游标， 游标记录了迭代进行的轨迹和进度。 在开始一次新的迭代时， 用户需要将游标设置为 0 ：

```
SCAN 0
```

SCAN 命令的执行结果由两个元素组成：

- 第一个元素是进行下一次迭代所需的游标， 如果这个游标为 0 ， 那么说明客户端已经对数据库完成了一次完整的迭代。
- 第二个元素是一个列表， 这个列表包含了本次迭代取得的数据库键； 如果 SCAN 命令在某次迭代中没有获取到任何键， 那么这个元素将是一个空列表。

关于 SCAN 命令返回的键列表， 有两个需要注意的地方：

1. SCAN 命令可能会返回重复的键， 用户如果不想在结果里面包含重复的键， 那么就需要自己在客户端里面进行检测和过滤。
2. SCAN 命令返回的键数量是不确定的， 有时候甚至会不返回任何键， 但只要命令返回的游标不为 0 ， 迭代就没有结束。

11.3.1 一次简单的迭代示例

在对 SCAN 命令有了基本的了解之后， 让我们来试试使用 SCAN 命令去完整地迭代一个数据库。

为了开始一次新的迭代， 我们将以 0 作为游标， 调用 SCAN 命令：

```
redis> SCAN 0
1) "25"                -- 进行下次迭代的游标
2)  1) "key::16"       -- 本次迭代获取到的键
    2) "key::2"
    3) "key::6"
    4) "key::8"
    5) "key::13"
    6) "key::22"
    7) "key::10"
    8) "key::24"
    9) "key::23"
    10) "key::21"
    11) "key::5"
```

这个 SCAN 调用告知我们下次迭代应该使用 25 作为游标， 并返回了十一个键的键名。

为了继续对数据库进行迭代， 我们使用 25 作为游标， 再次调用 SCAN 命令：

```
redis> SCAN 25
1) "31"
2)  1) "key::20"
    2) "key::18"
    3) "key::19"
    4) "key::7"
    5) "key::1"
    6) "key::9"
    7) "key::12"
    8) "key::11"
    9) "key::17"
    10) "key::15"
    11) "key::14"
    12) "key::3"
```

这次的 SCAN 调用返回了十二个键， 并告知我们下次迭代应该使用 31 作为游标。

跟之前的情况类似， 这次我们使用 31 作为游标， 再次调用 SCAN 命令：

```
redis> SCAN 31
1) "0"
2) 1) "key::0"
   2) "key::4"
```

这次的 SCAN 调用只返回了两个键， 并且它返回的下次迭代游标为 0 —— 这说明本次迭代已经结束， 整个数据库已经被迭代完毕。

11.3.2 SCAN 命令的迭代保证

针对数据库的一次完整迭代（full iteration）以用户给定游标 0 调用 SCAN 命令为开始， 直到 SCAN 命令返回游标 0 为结束。 SCAN 命令为完整迭代提供以下保证：

1. 从迭代开始到迭代结束的整个过程中， 一直存在于数据库里面的键总会被返回。
2. 如果一个键在迭代的过程中被添加到了数据库里面， 那么这个键是否会被返回是不确定的。
3. 如果一个键在迭代的过程中被移除了， 那么 SCAN 命令在它被移除之后将不再返回这个键； 但是这个键在被移除之前仍然有可能被 SCAN 命令返回。
4. 无论数据库如何变化， 迭代总是有始有终的， 不会出现循环迭代或者其他无法终止迭代的情况。

11.3.3 游标的使用

在很多数据库里面， 使用游标都要显式地进行申请， 并在迭代完成之后释放游标， 否则的话就会造成内存泄露。

与此相反， SCAN 命令的游标不需要申请， 也不需要释放， 它们不占用任何资源， 每个客户端都可以使用自己的游标独立地对数据库进行迭代。

此外， 用户可以随时在迭代的途中停止进行迭代， 又或者随时开始一次新的迭代， 这不会浪费任何资源， 也不会引发任何问题。

11.3.4 迭代与给定匹配符相匹配的键

在默认情况下， SCAN 命令会向客户端返回数据库包含的所有键， 它就像 KEYS * 命令调用的一个迭代版本。 但是通过使用可选的 MATCH 选项， 我们同样可以让 SCAN 命令只返回与给定全局匹配符相匹配的键：

```
SCAN cursor [MATCH pattern]
```

带有 MATCH 选项的 SCAN 命令就像是 KEYS pattern 命令调用的迭代版本。

举个例子， 假设我们想要获取数据库里面所有以 user:: 开头的键， 但是因为这些键的数量比较多， 直接使用 KEYS user::* 有可能会造成服务器阻塞， 所以我们可以使用 SCAN 命令来代替 KEYS 命令， 对符合 user::* 匹配的键进行迭代：

```
redis> SCAN 0 MATCH user::*
1) "208"
2) 1) "user::1"
   2) "user::65"
   3) "user::99"
   4) "user::51"

redis> SCAN 208 MATCH user::*
1) "232"
2) 1) "user::13"
   2) "user::28"
   3) "user::83"
   4) "user::14"
   5) "user::61"

-- 省略后续的其他迭代……
```

11.3.5 指定返回键的期望数量

在一般情况下， SCAN 命令返回的键数量是不确定的， 但是我们可以通过使用可选的 COUNT 选项， 向 SCAN 命令提供一个期望值， 以此来说明我们希望得到多少个键：

```
SCAN cursor [COUNT number]
```

这里特别需要注意的是， COUNT 选项向命令提供的只是期望的键数量， 但并不是精确的键数量。 比如说， 执行 SCAN cursor COUNT 10 并不是说 SCAN 命令最多只能返回 10 个键， 又或者一定要返回 10 个键：

- COUNT 选项只是提供了一个期望值， 告诉 SCAN 命令我们希望返回多少个键， 但每次迭代返回的键数量仍然是不确定的。
- 不过在通常情况下， 设置一个较大的 COUNT 值将有助于获得更多键， 这一点是可以肯定的。

以下代码展示了几个使用 COUNT 选项的例子：

```
redis> SCAN 0 COUNT 5
1) "160"
2) 1) "key::43"
   2) "key::s"
   3) "user::1"
   4) "key::83"
   5) "key::u"

redis> SCAN 0 MATCH user::* COUNT 10
1) "208"
2) 1) "user::1"
   2) "user::65"
   3) "user::99"
   4) "user::51"

redis> SCAN 0 MATCH key::* COUNT 100
1) "214"
2)  1) "key::43"
    2) "key::s"
    3) "key::83"
    -- 其他键……
    50) "key::28"
    51) "key::34"
```

在用户没有显式地使用 COUNT 选项的情况下， SCAN 命令将使用 10 作为 COUNT 选项的默认值， 换句话说， 以下两条命令的作用是相同的：

```
SCAN cursor

SCAN cursor COUNT 10
```

11.3.6 数据结构迭代命令

跟获取数据库键的 KEYS 命令一样， Redis 的各个数据结构也存在着一些可能会导致服务器阻塞的命令：

- 散列的 HKEYS 命令、 HVALS 命令和 HGETALL 命令在处理包含键值对较多的散列时， 可能会导致服务器阻塞。
- 集合的 SMEMBERS 命令在处理包含元素较多的集合时， 可能会导致服务器阻塞。

- 有序集合的一些范围型获取命令， 比如 ZRANGE ， 也有阻塞服务器的可能。 比如说， 为了获取有序集合包含的所有元素， 用户可能会执行命令调用 ZRANGE key 0 -1 ， 这时如果有序集合包含的成员数量较多的话， 这个 ZRANGE 命令可能就会导致服务器阻塞。

为了解决以上这些问题， Redis 为散列、集合和有序集合也提供了与 SCAN 命令类似的游标迭代命令， 它们分别是 HSCAN 命令、 SSCAN 命令和 ZSCAN 命令， 以下三个小节将分别介绍这三个命令的用法。

1. 散列迭代命令

HSCAN 命令可以以渐进的方式迭代给定散列包含的键值对：

```
HSCAN hash cursor [MATCH pattern] [COUNT number]
```

除了需要指定被迭代的散列之外， HSCAN 命令的其他参数跟 SCAN 命令的参数保持一致， 并且作用也一样。

作为例子， 以下代码展示了如何使用 HSCAN 命令去迭代 user::10086::profile 散列：

```
redis> HSCAN user::10086::profile 0
1) "0"             -- 下次迭代的游标
2)  1) "name"      -- 键
    2) "peter"     -- 值
    3) "age"
    4) "32"
    5) "gender"
    6) "male"
    7) "blog"
    8) "peter123.whatpress.com"
    9) "email"
    10) "peter123@example.com"
```

当散列包含较多键值对的时候， 我们应该尽量使用 HSCAN 去代替 HKEYS 、 HVALS 和 HGETALL ， 以免造成服务器阻塞。

2. 渐进式集合迭代命令

SSCAN 命令可以以渐进的方式迭代给定集合包含的元素：

```
SSCAN set cursor [MATCH pattern] [COUNT number]
```

除了需要指定被迭代的集合之外， SSCAN 命令的其他参数跟 SCAN 命令的参数保持一致， 并且作用也一样。

举个例子， 假设我们想要对 fruits 集合进行迭代的话， 那么可以执行以下命令：

```
redis> SSCAN fruits 0
1) "0"              -- 下次迭代的游标
2) 1) "apple"       -- 集合元素
   2) "watermelon"
   3) "mango"
   4) "cherry"
   5) "banana"
   6) "dragon fruit"
```

当集合包含较多元素的时候， 我们应该尽量使用 SSCAN 去代替 SMEMBERS ， 以免造成服务器阻塞。

3. 渐进式有序集合迭代命令

ZSCAN 命令可以以渐进的方式迭代给定有序集合包含的成员和分值：

```
ZSCAN sorted_set cursor [MATCH pattern] [COUNT number]
```

除了需要指定被迭代的有序集合之外， ZSCAN 命令的其他参数跟 SCAN 命令的参数保持一致， 并且作用也一样。

比如说， 通过执行以下命令， 我们可以对 fruits-price 有序集合进行迭代：

```
redis> ZSCAN fruits-price 0
1) "0"                    -- 下次迭代的游标
2)  1) "watermelon"       -- 成员
    2) "3.5"              -- 分值
    3) "banana"
    4) "4.5"
    5) "mango"
    6) "5"
    7) "dragon fruit"
    8) "6"
    9) "cherry"
    10) "7"
```

```
11) "apple"
12) "8.5"
```

当有序集合包含较多成员的时候， 我们应该尽量使用 ZSCAN 去代替 ZRANGE 以及其他可能会返回大量成员的范围型获取命令， 以免造成服务器阻塞。

4. 迭代命令的共通性质

HSCAN 、 SSCAN 、 ZSCAN 这三个命令除了与 SCAN 命令拥有相同的游标参数以及可选项之外， 还与 SCAN 命令拥有相同的迭代性质：

- SCAN 命令对于完整迭代所做的保证， 其他三个迭代命令也能够提供。 比如说， 使用 HSCAN 命令对散列进行一次完整迭代， 在迭代过程中一直存在的键值对总会被返回， 诸如此类。
- 跟 SCAN 命令一样， 其他三个迭代命令的游标也不耗费任何资源。 用户可以在这三个命令中随意地使用游标， 比如随时开始一次新的迭代， 又或者随时放弃正在进行的迭代， 这不会浪费任何资源， 也不会引发任何问题。
- 跟 SCAN 命令一样， 其他三个迭代命令虽然也可以使用 COUNT 选项设置返回元素数量的期望值， 但命令具体返回的元素数量仍然是不确定的。

11.3.7 其他信息

属性	值
复杂度	SCAN 命令、 HSCAN 命令、 SSCAN 命令和 ZSCAN 命令单次执行的复杂度为 O(1) ， 而使用这些命令进行一次完整迭代的复杂度则为 O(N) ，其中 N 为被迭代的元素数量。
版本要求	SCAN 命令、 HSCAN 命令、 SSCAN 命令和 ZSCAN 命令从 Redis 2.8.0 版本开始可用。

11.4 示例：构建数据库迭代器

SCAN 命令虽然可以以迭代的形式访问数据库， 但它使用起来并不是特别方便， 比如说：

- SCAN 命令每次迭代都会返回一个游标， 而用户需要手动地将这个游标用作下次迭代时的输入参数， 如果用户不小心丢失或者弄错了这个游标的话， 那么就可能会给迭代带来错误或者麻烦。
- SCAN 命令每次都会返回一个包含两个元素的结果， 其中第一个元素为游标， 而第二个元素才是当前被迭代的键， 如果迭代器能够直接返回被迭代的键， 那么它使用起来就会更加方便。

为了解决以上这两个问题， 我们可以在 SCAN 命令的基础上进行一些修改， 实现出代码清单 11-1 所示的迭代器：这个迭代器不仅会自动记录每次迭代的游标以防丢失， 它还可以直接返回被迭代的数据库键以供用户使用。

代码清单 11-1 数据库迭代器： /database/db_iterator.py

```
class DbIterator:

    def __init__(self, client, match=None, count=None):
        """
        创建一个新的迭代器。
        可选的 match 参数用于指定迭代的匹配模式，
        而可选的 count 参数则用于指定我们期待每次迭代能够返回的键数量。
        """
        self.client = client
        self.match = match
        self.count = count
        # 当前迭代游标
        self.current_cursor = 0
        # 记录迭代是否已经完成的状态变量
        self.iteration_is_over = False

    def next(self):
        """
        以列表形式返回当前被迭代到的数据库键，
        返回 None 则表示本次迭代已经完成。
        """
        if self.iteration_is_over:
            return None
        # 获取下次迭代的游标以及当前被迭代的数据库键
        next_cursor, keys = self.client.scan(self.current_cursor, self.match, self.count)
        # 如果下次迭代的游标为 0 ，那么表示迭代已完成
        if next_cursor == 0:
            self.iteration_is_over = True
        # 更新游标
        self.current_cursor = next_cursor
        # 返回当前被迭代的数据库键
        return keys
```

作为例子， 以下代码展示了如何使用这个迭代器去迭代一个数据库：

```
>>> from redis import Redis
>>> from db_iterator import DbIterator
```

```
>>> client = Redis(decode_responses=True)
>>> for i in range(50):             # 向数据库插入 50 个键
...     key = "key{0}".format(i)
...     value = i
...     client.set(key, value)
...
True
True
...
True
>>> iterator = DbIterator(client)
>>> iterator.next()  # 开始迭代
['key46', 'key1', 'key27', 'key39', 'key15', 'key0', 'key43', 'key12', 'key49', 'key41', 'key10']
>>> iterator.next()
['key23', 'key7', 'key9', 'key20', 'key18', 'key3', 'key5', 'key34', 'key32', 'key40']
>>> iterator.next()
['key4', 'key33', 'key30', 'key45', 'key38', 'key31', 'key6', 'key16', 'key25', 'key14', 'key13']
>>> iterator.next()
['key29', 'key2', 'key42', 'key11', 'key48', 'key28', 'key8', 'key44', 'key21', 'key26']
>>> iterator.next()
['key22', 'key47', 'key36', 'key17', 'key19', 'key24', 'key35', 'key37']
>>> iterator.next()  # 迭代结束
>>>
```

Note: redis-py 提供的迭代器

实际上， redis-py 客户端也为 `SCAN` 命令实现了一个迭代器 —— 用户只需要调用 redis-py 的 `scan_iter()` 方法， 就会得到一个 Python 迭代器， 然后就可以通过这个迭代器对数据库中的键进行迭代：

```
scan_iter(self, match=None, count=None) unbound redis.client.Redis method
    Make an iterator using the SCAN command so that the client doesn't
    need to remember the cursor position.

    ``match`` allows for filtering the keys by pattern

    ``count`` allows for hint the minimum number of returns
```

redis-py 提供的迭代器跟 `DbIterator` 一样， 都可以让用户免去手动输入游标的麻烦， 但它们之间也有不少区别：

1. redis-py 的迭代器每次迭代只返回一个元素。

2. 因为 redis-py 的迭代器是通过 Python 的迭代器特性实现的， 所以用户可以直接以 `for key in redis.scan_iter()` 的形式进行迭代。 （`DbIterator` 实际上也可以实现这样的特性， 但是由于 Python 迭代器的相关知识并不在本书的介绍范围之内， 所以我们这个自制的迭代器才没有配备这一特性。）
3. redis-py 的迭代器也拥有 `next()` 方法， 但这个方法每次被调用时只会返回单个元素， 并且它在所有元素都被迭代完毕时将抛出一个 `StopIteration` 异常。

以下是一个 redis-py 迭代器的使用示例：

```
>>> from redis import Redis
>>> client = Redis(decode_responses=True)
>>> client.mset({"k1":"v1", "k2":"v2", "k3":"v3"})
True
>>> for key in client.scan_iter():
...   print(key)
...
k1
k3
k2
```

因为 redis-py 为 `scan_iter()` 提供了直接支持， 它比需要额外引入的 `DbIterator` 更为方便一些， 所以本书之后展示的所有迭代程序都将使用 `scan_iter()` 而不是 `DbIterator` 。 不过由于这两个迭代器的底层实现是相仿的， 所以使用哪个其实差别都并不大。

11.5 RANDOMKEY：随机返回一个键

`RANDOMKEY` 命令可以从数据库里面随机地返回一个键：

```
RANDOMKEY
```

`RANDOMKEY` 命令不会移除被返回的键， 它们会继续留在数据库里面。

以下代码展示了如何通过 `RANDOMKEY` 命令， 从数据库里面随机地返回一些键：

```
redis> RANDOMKEY
"user::123::profile"

redis> RANDOMKEY
```

```
"cache::/user/peter"

redis> RANDOMKEY
"favorite-animal"

redis> RANDOMKEY
"fruit-price"
```

另一方面， 当数据库为空时， RANDOMKEY 命令将返回一个空值：

```
redis> RANDOMKEY
(nil)
```

11.5.1 其他信息

属性	值
复杂度	O(1)
版本要求	RANDOMKEY 命令从 Redis 1.0.0 版本开始可用。

11.6 SORT：对键的值进行排序

用户可以通过执行 SORT 命令， 对列表元素、集合元素或者有序集合成员进行排序。 为了让用户能够以不同的方式进行排序， Redis 为 SORT 命令提供了非常多的可选项， 如果我们以不给定任何可选项的方式直接调用 SORT 命令， 那么命令将对指定键储存的元素执行数字值排序：

```
SORT key
```

在默认情况下， SORT 命令将按照从小到大的顺序， 依次返回排序后的各个值。

比如说， 以下例子就展示了如何对 lucky-numbers 集合储存的六个数字值进行排序：

```
redis> SMEMBERS lucky-numbers    -- 以乱序形式储存的集合元素
1) "1024"
2) "123456"
3) "10086"
4) "3.14"
5) "888"
```

```
6) "256"

redis> SORT lucky-numbers         -- 排序后的集合元素
1) "3.14"
2) "256"
3) "888"
4) "1024"
5) "10086"
6) "123456"
```

而以下例子则展示了如何对 `message-queue` 列表中的数字值进行排序：

```
redis> LRANGE message-queue 0 -1     -- 根据插入顺序进行排列的列表元素
1) "1024"
2) "256"
3) "128"
4) "512"
5) "64"

redis> SORT message-queue            -- 排序后的列表元素
1) "64"
2) "128"
3) "256"
4) "512"
5) "1024"
```

11.6.1 指定排序方式

在默认情况下， SORT 命令执行的是升序排序操作： 较小的值将被放到结果的较前位置， 而较大的值则会被放到结果的较后位置。

通过使用可选的 ASC 选项或者 DESC 选项， 用户可以指定 SORT 命令的排序方式， 其中 ASC 表示执行升序排序操作， 而 DESC 则表示执行降序排序操作：

```
SORT key [ASC|DESC]
```

降序排序操作的做法跟升序排序操作的做法正好相反， 它会把较大的值放到结果的较前位置， 而较小的值则会被放到结果的较后位置。

举个例子， 如果我们想要对 `lucky-numbers` 集合的元素实施降序排序， 那么只需要执行以下代码就可以了：

```
redis> SORT lucky-numbers DESC
1) "123456"
2) "10086"
3) "1024"
4) "888"
5) "256"
6) "3.14"
```

因为 SORT 命令在默认情况下进行的就是升序排序， 所以 SORT key 命令和 SORT key ASC 命令产生的效果是完全相同的， 因此我们在一般情况下并不会用到 ASC 选项 —— 除非你有特别的理由， 需要告诉别人你正在进行的是升序排序。

11.6.2 对字符串值进行排序

SORT 命令在默认情况下进行的是数字值排序， 如果我们尝试直接使用 SORT 命令去对字符串元素进行排序， 那么命令将产生一个错误：

```
redis> SMEMBERS fruits
1) "cherry"
2) "banana"
3) "apple"
4) "mango"
5) "dragon fruit"
6) "watermelon"

redis> SORT fruits
(error) ERR One or more scores can't be converted into double
```

为了让 SORT 命令能够对字符串值进行排序， 我们必须让 SORT 命令执行字符串排序操作而不是数字值排序操作， 这一点可以通过使用 ALPHA 选项来实现：

```
SORT key [ALPHA]
```

作为例子， 我们可以使用带 ALPHA 选项的 SORT 命令去对 fruits 集合进行排序：

```
redis> SORT fruits ALPHA
1) "apple"
2) "banana"
3) "cherry"
```

```
4) "dragon fruit"
5) "mango"
6) "watermelon"
```

又或者使用以下命令， 对 `test-record` 有序集合的成员进行排序：

```
redis> ZRANGE test-record 0 -1 WITHSCORES     -- 在默认情况下，有序集合成员将根据分值进行排序
1) "ben"
2) "70"
3) "aimee"
4) "86"
5) "david"
6) "99"
7) "cario"
8) "100"

redis> SORT test-record ALPHA                 -- 但使用 SORT 命令可以对成员本身进行排序
1) "aimee"
2) "ben"
3) "cario"
4) "david"
```

11.6.3 只获取部分排序结果

在默认情况下， `SORT` 命令将返回所有被排序的元素， 但如果我们只需要其中一部分排序结果的话， 那么可以使用可选的 `LIMIT` 选项：

```
SORT key [LIMIT offset count]
```

其中 `offset` 参数用于指定返回结果之前需要跳过的元素数量， 而 `count` 参数则用于指定需要获取的元素数量。

举个例子， 如果我们想要知道 `fruits` 集合在排序之后的第 3 个元素是什么， 那么只需要执行以下调用就可以了：

```
redis> SORT fruits ALPHA LIMIT 2 1
1) "cherry"
```

注意， 因为 `offset` 参数的值是从 `0` 开始计算的， 所以这个命令在获取第三个被排序元素时使用了 `2` 而不是 `3` 来作为偏移量。

11.6.4 获取外部键的值作为结果

在默认情况下， SORT 命令将返回被排序的元素作为结果， 但如果用户有需要的话， 也可以使用 GET 选项去获取其他别的值作为排序结果：

```
SORT key [[GET pattern] [GET pattern] ...]
```

一个 SORT 命令可以使用任意多个 GET pattern 选项， 其中 pattern 参数的值可以是：

1. 包含 * 符号的字符串；
2. 包含 * 符号和 -> 符号的字符串；
3. 一个单独的 # 符号；

接下来的三个小节将分别介绍这三种值的用法和用途。

1. 获取字符串键的值

当 pattern 参数的值是一个包含 * 符号的字符串时， SORT 命令将把被排序的元素与 * 符号实行替换， 构建出一个键名， 然后使用 GET 命令去获取该键的值。

表 11-2 储存水果价格的各个字符串键，以及它们的值

字符串键	值
"apple-price"	8.5
"banana-price"	4.5
"cherry-price"	7
"dragon fruit-price"	6
"mango-price"	5
"watermelon-price"	3.5

举个例子， 假设数据库里面储存着表 11-2 所示的一些字符串键， 那么我们可以通过执行以下命令， 对 fruits 集合的各个元素进行排序， 然后根据排序后的元素去获取各种水果的价格：

```
redis> SORT fruits ALPHA GET *-price
1) "8.5"
2) "4.5"
3) "7"
4) "6"
5) "5"
6) "3.5"
```

这个 SORT 命令的执行过程可以分为以下三个步骤：

1. 对 fruits 集合的各个元素进行排序， 得出一个由 "apple" 、 "banana" 、 "cherry" 、 "dragon fruit" 、 "mango" 、 "watermelon" 组成的有序元素排列。
2. 将排序后的各个元素与 *-price 模式进行匹配和替换， 得出键名 "apple-price" 、 "banana-price" 、 "cherry-price" 、 "dragon fruit-price" 、 "mango-price" 和 "watermelon-price" 。
3. 使用 GET 命令去获取以上各个键的值， 并将这些值依次放入到结果列表里面， 最后把结果列表返回给客户端。

图 11-2 以图形方式展示了整个 SORT 命令的执行过程。

图 11-2 SORT fruits ALPHA GET *-price 命令的执行过程

2. 获取散列中的键值

当 pattern 参数的值是一个包含 * 符号和 -> 符号的字符串时， SORT 命令将使用 -> 左边的字符串为散列名， -> 右边的字符串为字段名， 调用 HGET 命令， 从散列里面获取指定字段的值。 此外， 用户传入的散列名还需要包含 * 符号， 这个 * 符号将被替换成被排序的元素。

表 11-3 储存着水果信息的散列

散列名	散列中 inventory 字段的值
apple-info	“1000”
banana-info	“300”
cherry-info	“50”
dragon fruit-info	“500”
mango-info	“250”
watermelon-info	“324”

举个例子， 假设数据库里面储存着表 11-3 所示的 apple-info 、 banana-info 等散列， 而这些散列的 inventory 键则储存着相应水果的存货量， 那么我们可以通过执行以下命令， 对 fruits 集合的各个元素进行排序， 然后根据排序后的元素去获取各种水果的存货量：

```
redis> SORT fruits ALPHA GET *-info->inventory
1) "1000"
2) "300"
3) "50"
4) "500"
5) "250"
6) "324"
```

这个 SORT 命令的执行过程可以分为以下三个步骤：

1. 对 fruits 集合的各个元素进行排序， 得出一个由 "apple" 、 "banana" 、 "cherry" 、 "dragon fruit" 、 "mango" 、 "watermelon" 组成的有序元素排列。
2. 将排序后的各个元素与 *-info 模式进行匹配和替换， 得出散列名 "apple-info" 、 "banana-info" 、 "cherry-info" 、 "dragon fruit-info" 、 "mango-info" 和 "watermelon-info" 。

3. 使用 HGET 命令， 从以上各个散列中取出 inventory 字段的值， 并将这些值依次放入到结果列表里面， 最后把结果列表返回给客户端。

图 11-3 以图形方式展示了整个 SORT 命令的执行过程。

图 11-3 SORT fruits ALPHA GET *-info->inventory 命令的执行过程

3. 获取被排序元素本身

当 pattern 参数的值是一个 # 符号时， SORT 命令将返回被排序的元素本身。

因为 SORT key 命令和 SORT key GET # 命令返回的是完全相同的结果， 所以单独使用 GET # 并没有任何实际作用：

```
redis> SORT fruits ALPHA
1) "apple"
2) "banana"
3) "cherry"
4) "dragon fruit"
5) "mango"
6) "watermelon"

redis> SORT fruits ALPHA GET #    -- 与上一个命令的结果完全相同
1) "apple"
```

```
2) "banana"
3) "cherry"
4) "dragon fruit"
5) "mango"
6) "watermelon"
```

因此， 我们一般只会在同时使用多个 GET 选项时， 才使用 GET # 获取被排序的元素。 比如说， 以下代码就展示了如何在对水果进行排序的同时， 获取水果的价格和库存量：

```
redis> SORT fruits ALPHA GET # GET *-price GET *-info->inventory
1) "apple"     -- 水果
2) "8.5"       -- 价格
3) "1000"      -- 库存量
4) "banana"
5) "4.5"
6) "300"
7) "cherry"
8) "7"
9) "50"
10) "dragon fruit"
11) "6"
12) "500"
13) "mango"
14) "5"
15) "250"
16) "watermelon"
17) "3.5"
18) "324"
```

11.6.5 使用外部键的值作为排序权重

在默认情况下， SORT 命令将使用被排序元素本身作为排序权重， 但在有需要时， 用户可以通过可选的 BY 选项， 指定其他键的值作为排序的权重：

```
SORT key [BY pattern]
```

pattern 参数的值既可以是包含 * 符号的字符串， 也可以是包含 * 符号和 -> 符号的字符串， 这两种值的作用和效果跟它们在 GET 选项时的作用和效果一样： 前者用于获取字符串键的值， 而后者则用于从散列里面获取指定字段的值。

举个例子， 通过执行以下命令， 我们可以使用储存在字符串键里面的水果价格作为权重， 对水果进行排序：

```
redis> SORT fruits BY *-price
1) "watermelon"
2) "banana"
3) "mango"
4) "dragon fruit"
5) "cherry"
6) "apple"
```

因为上面这个排序结果只展示了水果的名字， 却没有展示水果的价格， 所以这个排序结果并没有清楚地展示水果的名字和价格之间的关系。 相反地， 如果我们在使用 `BY` 选项的同时， 使用两个 `GET` 选项去获取水果的名字以及价格， 那么就能够直观地看出水果是按照价格进行排序的了：

```
redis> SORT fruits BY *-price GET # GET *-price
1) "watermelon"   -- 水果的名字
2) "3.5"          -- 水果的价格
3) "banana"
4) "4.5"
5) "mango"
6) "5"
7) "dragon fruit"
8) "6"
9) "cherry"
10) "7"
11) "apple"
12) "8.5"
```

同样地， 我们还可以通过执行以下命令， 使用散列中记录的库存量作为权重， 对水果进行排序并获取它们的库存量：

```
redis> SORT fruits BY *-info->inventory GET # GET *-info->inventory
1) "cherry"   -- 水果的名字
2) "50"       -- 水果的库存量
3) "mango"
4) "250"
5) "banana"
6) "300"
7) "watermelon"
8) "324"
9) "dragon fruit"
10) "500"
```

```
11) "apple"
12) "1000"
```

11.6.6 保存排序结果

在默认情况下， SORT 命令会直接将排序结果返回给客户端， 但如果用户有需要的话， 也可以通过可选的 STORE 选项， 以列表形式将排序结果储存到指定的键里面：

```
SORT key [STORE destination]
```

如果用户给定的 destination 键已经存在， 那么 SORT 命令会先移除该键， 然后再储存排序结果。 带有 STORE 选项的 SORT 命令在成功执行之后将返回被储存的元素数量作为结果。

作为例子， 以下代码展示了如何将排序 fruits 集合所得的结果储存到 sorted-fruits 列表里面：

```
redis> SORT fruits ALPHA STORE sorted-fruits
(integer) 6     -- 有六个已排序元素被储存了

redis> LRANGE sorted-fruits 0 -1     -- 查看排序结果
1) "apple"
2) "banana"
3) "cherry"
4) "dragon fruit"
5) "mango"
6) "watermelon"
```

11.6.7 其他信息

属性	值
平均复杂度	O(N*log(N)+M) ， 其中 N 为被排序元素的数量， 而 M 则为命令返回的元素数量。
版本要求	SORT 命令从 Redis 1.0.0 版本开始可用。

11.7 EXISTS：检查给定键是否存在

用户可以通过使用 EXISTS 命令， 检查给定的一个或多个键是否存在于当前正在使用的数据库里面：

```
EXISTS key [key ...]
```

`EXISTS` 命令将返回存在的给定键数量作为返回值。

通过将多个键传递给 `EXISTS` 命令， 我们可以判断出， 在给定的键里面， 有多少个键是实际存在的。 举个例子， 通过执行以下命令， 我们可以知道 `k1` 、 `k2` 和 `k3` 这三个给定键当中， 只有两个键是存在的：

```
redis> EXISTS k1 k2 k3
(integer) 2
```

另一方面， 如果我们只想要确认某个键是否存在， 那么只需要将那个键传递给 `EXISTS` 命令即可： 命令返回 `0` 表示该键不存在， 而返回 `1` 则表示该键存在。

比如说， 通过执行以下命令， 我们可以知道键 `k3` 并不存在于数据库：

```
redis> EXISTS k3
(integer) 0
```

11.7.1 只能接受单个键的 `EXISTS` 命令

`EXISTS` 命令从 Redis 3.0.3 版本开始接受多个键作为输入， 在此前的版本中， `EXISTS` 命令只能接受单个键作为输入：

```
EXISTS key
```

旧版的 `EXISTS` 命令在键存在时返回 `1` ， 不存在时返回 `0` 。

11.7.2 其他信息

属性	值
复杂度	Redis 3.0.3 版本以前， 只能接受单个键作为输入的 `EXISTS` 命令的复杂度为 O(1) ； Redis 3.0.3 及以上版本， 能够接受多个键作为输入的 `EXISTS` 命令的复杂度为 O(N) ， 其中 N 为用户给定的键数量。

属性	值
版本要求	`EXISTS` 命令从 Redis 1.0.0 版本开始可用， 但只有 Redis 3.0.3 及以上版本才能接受多个键作为输入， 此前的版本只能接受单个键作为输入。

11.8 DBSIZE：获取数据库包含的键值对数量

用户可以通过执行 `DBSIZE` 命令来获知当前使用的数据库包含了多少个键值对：

```
DBSIZE
```

比如在以下这个例子中， 我们就通过执行 `DBSIZE` 获知数据库目前包含了六个键值对：

```
redis> DBSIZE
(integer) 6
```

11.8.1 其他信息

属性	值
复杂度	O(1)
版本要求	`DBSIZE` 命令从 Redis 1.0.0 版本开始可用。

11.9 TYPE：查看键的类型

`TYPE` 命令允许我们查看给定键的类型：

```
TYPE key
```

举个例子， 如果我们对一个字符串键执行 `TYPE` 命令， 那么命令将告知我们， 这个键是一个字符串键：

```
redis> GET msg
"hello world"

redis> TYPE msg
string
```

又比如说， 如果我们对一个集合键执行 TYPE 命令， 那么命令将告知我们， 这个键是一个集合键：

```
redis> SMEMBERS fruits
1) "banana"
2) "cherry"
3) "apple"

redis> TYPE fruits
set
```

表 11-4 列出了 TYPE 命令在面对不同类型的键时返回的各项结果。

表 11-4 TYPE 命令在面对不同类型的键时返回的各项结果

键类型	TYPE 命令的返回值
字符串键	string
散列键	hash
列表键	list
集合键	set
有序集合键	zset
HyperLogLog	string
位图	string
地理位置	zset
流	stream

在这个表格里面， TYPE 命令对于字符串键、散列键、列表键、集合键和流键的返回结果都非常直观， 不过它对于之后几种类型的键的返回结果则需要做进一步解释：

- 因为所有有序集合命令 —— 比如 ZADD 、 ZREM 、 ZSCORE 等等 —— 都是以 z 为前缀命名的， 所以有序集合也被称为 zset 。 因此 TYPE 命令在接收到有序集合键作为输入时， 将返回 zset 作为结果。
- 因为 HyperLogLog 和位图这两种键在底层都是通过字符串键来实现的， 所以 TYPE 命令对于这两种键将返回 string 作为结果。

- 跟 HyperLogLog 和位图的情况类似， 因为地理位置键使用了有序集合键作为底层实现， 所以 `TYPE` 命令对于地理位置键将返回 `zset` 作为结果。

11.9.1 其他信息

属性	值
复杂度	O(1)
版本要求	`TYPE` 命令从 Redis 1.0.0 版本开始可用。

11.10 示例：数据库取样程序

在使用 Redis 的过程中， 我们可能会想要知道 Redis 数据库中各种键的类型分布状况： 比如说， 我们可能会想要知道数据库里面有多少个字符串键、有多少个列表键、有多少个散列键， 以及这些键在数据库键的总数量中占多少个百分比。

代码清单 11-2 展示了一个能够计算出以上信息的数据库取样程序。 `DbSampler` 程序会对数据库进行迭代， 使用 `TYPE` 命令获取被迭代键的类型并对不同类型的键实施计数， 最终在迭代完整个数据库之后， 打印出相应的取样结果。

代码清单 11-2 数据库取样程序： `/database/db_sampler.py`

```
def type_sample_result(type_name, type_counter, db_size):
    result = "{0}: {1} keys, {2}% of the total."
    return result.format(type_name, type_counter, type_counter*100.0/db_size)

class DbSampler:

    def __init__(self, client):
        self.client = client

    def sample(self):
        # 键类型计数器
        type_counter = {
            "string": 0,
            "list": 0,
            "hash": 0,
            "set": 0,
```

```
            "zset": 0,
            "stream": 0,
        }

        # 遍历整个数据库
        for key in self.client.scan_iter():
            # 获取键的类型
            type = self.client.type(key)
            # 对相应的类型计数器执行加一操作
            type_counter[type] += 1

        # 获取数据库大小
        db_size = self.client.dbsize()

        # 打印结果
        print("Sampled {0} keys.".format(db_size))
        print(type_sample_result("String", type_counter["string"], db_size))
        print(type_sample_result("List", type_counter["list"], db_size))
        print(type_sample_result("Hash", type_counter["hash"], db_size))
        print(type_sample_result("Set", type_counter["set"], db_size))
        print(type_sample_result("SortedSet", type_counter["zset"], db_size))
        print(type_sample_result("Stream", type_counter["stream"], db_size))
```

以下代码展示了这个数据库取样程序的使用方法：

```
>>> from redis import Redis
>>> from create_random_type_keys import create_random_type_keys
>>> from db_sampler import DbSampler
>>> client = Redis(decode_responses=True)
>>> create_random_type_keys(client, 1000)  # 创建 1000 个类型随机的键
>>> sampler = DbSampler(client)
>>> sampler.sample()
Sampled 1000 keys.
String: 179 keys, 17.9% of the total.
List: 155 keys, 15.5% of the total.
Hash: 172 keys, 17.2% of the total.
Set: 165 keys, 16.5% of the total.
SortedSet: 161 keys, 16.1% of the total.
Stream: 168 keys, 16.8% of the total.
```

可以看到， 取样程序遍历了数据库中的一千个键， 然后打印出了不同类型键的具体数量以及它们在整个数据库中所占的百分比。

为了演示方便，上面的代码使用了 `create_random_type_keys()` 函数来创建出指定数量的类型随机键，代码清单 11-3 展示了这个函数的具体定义。

代码清单 11-3 随机键生成程序：`/database/create_random_type_keys.py`

```
import random

def create_random_type_keys(client, number):
    """
    在数据库中创建指定数量的类型随机键。
    """
    for i in range(number):
        # 构建键名
        key = "key:{0}".format(i)
        # 从六个键创建函数中随机选择一个
        create_key_func = random.choice([
            create_string,
            create_hash,
            create_list,
            create_set,
            create_zset,
            create_stream
        ])
        # 实际地创建键
        create_key_func(client, key)

def create_string(client, key):
    client.set(key, "")

def create_hash(client, key):
    client.hset(key, "", "")

def create_list(client, key):
    client.rpush(key, "")

def create_set(client, key):
    client.sadd(key, "")

def create_zset(client, key):
    client.zadd(key, {"":0})

def create_stream(client, key):
    client.xadd(key, {"":""})
```

11.11 RENAME、RENAMENX：修改键名

Redis 提供了 `RENAME` 命令， 用户可以使用这个命令对键的名字进行修改：

```
RENAME origin new
```

`RENAME` 命令在执行成功时将返回 `OK` 作为结果。

作为例子， 以下代码展示了如何将键 `msg` 改名为键 `message` ：

```
redis> GET msg
"hello world"

redis> RENAME msg message
OK

redis> GET msg
(nil)  -- 原来的键在改名之后已经不复存在

redis> GET message
"hello world"  -- 访问改名之后的新键
```

11.11.1 覆盖已存在的键

如果用户指定的新键名已经被占用， 那么 `RENAME` 命令会先移除占用了新键名的那个键， 然后再执行改名操作。

在以下这个例子中， 键 `k1` 和键 `k2` 都存在， 如果我们使用 `RENAME` 命令将键 `k1` 改名为键 `k2` ， 那么原来的键 `k2` 将被移除：

```
redis> SET k1 v1
OK

redis> SET k2 v2
OK

redis> RENAME k1 k2
OK
```

```
redis> GET k2
"v1"
```

11.11.2 只在新键名尚未被占用的情况下进行改名

除了 `RENAME` 命令之外， Redis 还提供了 `RENAMENX` 命令。 `RENAMENX` 命令和 `RENAME` 命令一样， 都可以对键进行改名， 但 `RENAMENX` 命令只会在新键名尚未被占用的情况下进行改名， 如果用户指定的新键名已经被占用， 那么 `RENAMENX` 将放弃执行改名操作：

```
RENAMENX origin new
```

`RENAMENX` 命令在改名成功时返回 `1` ， 失败时返回 `0` 。

比如在以下例子中， 因为键 `k2` 已经存在， 所以尝试将键 `k1` 改名为 `k2` 将以失败告终：

```
redis> SET k1 v1
OK

redis> SET k2 v2
OK

redis> RENAMENX k1 k2
(integer) 0   -- 改名失败
```

与此相反， 因为键 `k3` 尚未存在， 所以将键 `k1` 改名为键 `k3` 的操作可以成功执行：

```
redis> GET k3
(nil)

redis> RENAMENX k1 k3
(integer) 1   -- 改名成功

redis> GET k3
"v1"

redis> GET k1
(nil)   -- 改名之后的 k1 已经不再存在
```

11.11.3 其他信息

属性	值
复杂度	O(1)
版本要求	RENAME 命令和 RENAMENX 命令都从 Redis 1.0.0 版本开始可用。

11.12 MOVE：将给定的键移动到另一个数据库

用户可以使用 MOVE 命令， 将一个键从当前数据库移动至目标数据库：

```
MOVE key db
```

当 MOVE 命令成功将给定键从当前数据库移动至目标数据库时， 命令返回 1 ； 如果给定键并不存在于当前数据库， 又或者目标数据库里面存在与给定键同名的键， 那么 MOVE 命令将不做动作， 只返回 0 表示移动失败。

作为例子， 以下代码展示了怎样将 0 号数据库中的 msg 键移动到 3 号数据库：

```
redis> GET msg            -- 位于 0 号数据库中的 msg 键
"This is a message from db 0."

redis> MOVE msg 3         -- 将 msg 键移动到 3 号数据库
(integer) 1

redis> SELECT 3           -- 切换至 3 号数据库
OK

redis[3]> GET msg         -- 获取被移动的 msg 键
"This is a message from db 0."
```

11.12.1 不覆盖同名键

当目标数据库存在与给定键同名的键时， MOVE 命令将放弃执行移动操作。

举个例子， 如果我们在 0 号数据库和 5 号数据库里面分别设置两个 lucky_number 键：

```
redis> SET lucky_number 123456     -- 在 0 号数据库设置 lucky_number 键
OK
```

```
redis> SELECT 5                     -- 切换至 5 号数据库
OK

redis[5]> SET lucky_number 777      -- 在 5 号数据库设置 lucky_number 键
OK
```

然后尝试将 5 号数据库的 lucky_number 键移动到 0 号数据库， 那么这次移动操作将不会成功：

```
redis[5]> MOVE lucky_number 0
(integer) 0
```

11.12.2 其他信息

属性	值
复杂度	O(1)
版本要求	MOVE 命令从 Redis 1.0.0 版本开始可用。

11.13 DEL：移除指定的键

DEL 命令允许用户从当前正在使用的数据库里面移除指定的一个或多个键， 以及与这些键相关联的值：

```
DEL key [key ...]
```

DEL 命令将返回成功移除的键数量作为返回值。

举个例子， 假设我们想要移除数据库中的键 k1 和键 k2 ， 那么只需要执行以下命令即可：

```
redis> DEL k1 k2
(integer) 2     -- 有两个键被移除了
```

另一方面， 如果用户给定的键并不存在， 那么 DEL 命令将不做动作：

```
redis> DEL k3
(integer) 0     -- 本次操作没有移除任何键
```

11.13.1 其他信息

属性	值
复杂度	O(N)，其中 N 为被移除键的数量。
版本要求	DEL 命令从 Redis 1.0.0 版本开始可用。

11.14 UNLINK：以异步方式移除指定的键

在前面一节， 我们介绍了如何使用 DEL 命令去移除指定的键， 但这个命令实际上隐含着一个性能问题： 因为 DEL 命令会以同步方式执行移除操作， 所以如果待移除的键非常庞大又或者数量众多， 那么服务器在执行移除操作的过程中就有可能会被阻塞。 比如说， 移除一个包含上百万个元素的集合， 移除一个包含数十万个键值对的散列， 又或者一次移除成千上万个键， 都有可能会引起服务器阻塞。

为了解决这个问题， Redis 从 4.0 版本开始新添加了一个 UNLINK 命令：

```
UNLINK key [key ...]
```

UNLINK 命令跟 DEL 命令一样， 都可以用于移除指定的键， 但它跟 DEL 命令的区别在于， 当用户调用 UNLINK 命令去移除一个数据库键的时候， UNLINK 只会在数据库里面移除对该键的引用（reference）， 而对键的实际移除操作则会交给后台线程执行， 因此 UNLINK 命令将不会造成服务器阻塞。

跟 DEL 命令一样， UNLINK 命令也会返回被移除键的数量作为结果。 此外， 由于兼容方面的原因， Redis 将在提供异步移除操作 UNLINK 命令的同时， 继续提供同步移除操作 DEL 命令。

以下是一个使用 UNLINK 命令的例子：

```
redis> MGET k1 k2 k3
1) "v1"
2) "v2"
3) "v3"

redis> UNLINK k1 k2 k3
(integer) 3

redis> MGET k1 k2 k3
1) (nil)
```

```
2) (nil)
3) (nil)
```

11.14.1 其他信息

属性	值
复杂度	O(N)，其中 N 为被移除键的数量。
版本要求	UNLINK 命令从 Redis 4.0 版本开始可用。

11.15 FLUSHDB：清空当前数据库

通过使用 FLUSHDB 命令， 用户可以清空自己当前正在使用的数据库：

```
redis> FLUSHDB
OK
```

FLUSHDB 命令会遍历用户正在使用的数据库， 移除其中包含的所有键值对， 然后返回 OK 表示数据库已被清空。

11.15.1 async 选项

跟 DEL 命令一样， FLUSHDB 命令也是一个同步移除命令， 并且因为 FLUSHDB 移除的是整个数据库而不是单个键，所以它常常会引发比 DEL 命令更为严重的服务器阻塞现象。

为了解决这个问题， Redis 4.0 给 FLUSHDB 命令新添加了一个 async 选项：

```
redis> FLUSHDB async
OK
```

如果用户在调用 FLUSHDB 命令时使用了 async 选项， 那么实际的数据库清空操作将放在后台线程里面以异步方式进行， 这样一来 FLUSHDB 命令就不会再阻塞服务器了。

11.15.2 其他信息

属性	值

属性	值
复杂度	O(N) ，其中 N 为被清空数据库包含的键值对数量。
版本要求	不带任何选项的 `FLUSHDB` 命令从 Redis 1.0.0 版本开始可用， 带有 `async` 选项的 `FLUSHDB` 命令从 Redis 4.0 版本开始可用。

11.16 FLUSHALL：清空所有数据库

通过使用 `FLUSHALL` 命令， 用户可以清空 Redis 服务器包含的所有数据库：

```
redis> FLUSHALL
OK
```

`FLUSHALL` 命令会遍历服务器包含的所有数据库， 并移除其中包含的所有键值对， 然后返回 `OK` 表示所有数据库均已被清空。

11.16.1 `async` 选项

跟 `FLUSHDB` 命令一样， 以同步方式执行的 `FLUSHALL` 命令也可能会导致服务器阻塞， 因此 Redis 4.0 也给 `FLUSHALL` 命令添加了同样的 `async` 选项：

```
redis> FLUSHALL async
OK
```

通过指定 `async` 选项， `FLUSHALL` 命令将以异步方式在后台线程里面执行所有实际的数据库清空操作， 因此它将不会再阻塞服务器。

11.16.2 其他信息

属性	值
复杂度	O(N)，其中 N 为被清空的所有数据库包含的键值对总数量。
版本要求	不带任何选项的 `FLUSHALL` 命令从 Redis 1.0.0 版本开始可用， 带有 `async` 选项的 `FLUSHALL` 命令从 Redis 4.0 版本开始可用。

11.17 SWAPDB：互换数据库

`SWAPDB` 命令接受两个数据库号码作为输入，然后对指定的两个数据库实行互换，最后返回 `OK` 作为结果：

```
SWAPDB x y
```

在 `SWAPDB` 命令执行完毕之后，原本储存在数据库 `x` 中的键值对将出现在数据库 `y` 中，而原本储存在数据库 `y` 中的键值对将出现在数据库 `x` 中。

举个例子，对于以下这个包含键 `k1` 、 `k2` 和 `k3` 的 0 号数据库：

```
db0> KEYS *
1) "k3"
2) "k2"
3) "k1"
```

以及以下这个包含键 `k4` 、 `k5` 和 `k6` 的 1 号数据库来说：

```
db1> KEYS *
1) "k5"
2) "k4"
3) "k6"
```

如果我们执行以下命令，对 0 号数据库和 1 号数据库实行互换：

```
db0> SWAPDB 0 1
OK
```

那么在此之后，原本储存在 0 号数据库中的键 `k1` 、 `k2` 和 `k3` 将出现在 1 号数据库中：

```
db1> KEYS *
1) "k3"
2) "k2"
3) "k1"
```

而原本储存在 1 号数据库中的键 `k4` 、 `k5` 和 `k6` 将出现在 0 号数据库中：

```
db0> KEYS *
1) "k5"
2) "k4"
3) "k6"
```

Note: 因为互换数据库这一操作可以通过调整指向数据库的指针来实现， 这个过程不需要移动数据库中的任何键值对， 所以 SWAPDB 命令的复杂度是 O(1) 而不是 O(N)， 并且执行这个命令也不会导致服务器阻塞。

11.17.1 其他信息

属性	值
复杂度	O(1)
版本要求	SWAPDB 命令从 Redis 4.0 版本开始可用。

11.18 示例：使用 SWAPDB 命令实行在线替换数据库

正如上一节所说， SWAPDB 命令可以以非阻塞方式互换给定的两个数据库。 因为这个命令的执行速度是如此之快，并且完全不会阻塞服务器， 所以用户实际上可以使用这个命令来实行在线的数据库替换操作。

举个例子， 假设我们拥有一个 Redis 服务器， 它的 0 号数据库储存了用户的邮件地址以及经过加密的用户密码， 这些数据可以用于登录用户账号。 不幸的是， 因为一次漏洞事故， 这个服务器遭到了黑客入侵， 并且经过确认， 这个服务器储存的所有用户密码均已泄露。 为了保障用户的信息安全， 我们决定立即重置所有用户密码，具体的做法是： 遍历所有用户的个人档案， 为每个用户生成一个新的随机密码， 并使用这个新密码替换已经泄露的旧密码。

代码清单 11-4 展示了一个用于重置用户密码的脚本， 它的 `reset_user_password()` 函数会迭代 `origin` 数据库中的所有用户数据， 为他们生成新密码， 并将更新后的用户信息储存到 `new` 数据库里面。 在此之后， 函数会使用 SWAPDB 命令互换新旧两个数据库， 并以异步方式移除旧数据库。

代码清单 11-4 用于重置用户密码的脚本代码： `/database/reset_user_password.py`

```
import random

from redis import Redis
```

```python
from hashlib import sha256

def generate_new_password():
    random_string = str(random.getrandbits(256)).encode('utf-8')
    return sha256(random_string).hexdigest()

def reset_user_password(origin, new):
    # 两个客户端，分别连接两个数据库
    origin_db = Redis(db=origin)
    new_db = Redis(db=new)

    for key in origin_db.scan_iter(match="user::*"):
        # 从源数据库获取现有用户信息
        user_data = origin_db.hgetall(key)
        # 重置用户密码
        user_data["password"] = generate_new_password()
        # 将新的用户信息储存到新数据库里面
        new_db.hmset(key, user_data)

    # 互换新旧数据库
    origin_db.swapdb(origin, new)

    # 以异步方式移除旧数据库
    # （new_db 变量现在已经指向旧数据库）
    new_db.flushdb(asynchronous=True)
```

作为例子， 表 11-5 和表 11-6 分别展示了设置新密码之前和之后的用户数据。 注意， 为了凸显新旧密码之间的区别， 重置之前的用户密码是未经加密的。

表 11-5 重置之前的用户数据

散列键名	email 字段（邮箱地址）	password 字段（未加密密码）
“user::54209”	“peter@spam.mail”	“petergogo128”
“user::73914”	“jack@spam.mail”	“happyjack256”
“user::98321”	“tom@spam.mail”	“tomrocktheworld512”
“user::39281”	“mary@spam.mail”	“maryisthebest1024”

表 11-6 重置之后的用户数据

散列键名	email 字段（邮箱地址）	password 字段（已加密密码）
“user::54209”	“peter@spam.mail”	“669a533168e4da2fce34...4d2af”
“user::73914”	“jack@spam.mail”	“e0caf7fc1245fa13fb34...a18eb”
“user::98321”	“tom@spam.mail”	“1b9f3944bec47bed3527...388c1”
“user::39281”	“mary@spam.mail”	“b7f6e3cd4ca27ac67851...75ccf”

11.19 重点回顾

- 所有 Redis 键， 无论它们是什么类型， 都会被储存到数据库里面。
- 一个 Redis 服务器可以同时拥有多个数据库， 每个数据库都拥有一个独立的命名空间。 这也即是说， 同名的键可以出现在不同数据库里面。
- 在默认情况下， Redis 服务器在启动时将创建 16 个数据库， 并使用数字 0 至 15 对其进行标识。
- 因为 KEYS 命令在数据库包含大量键的时候可能会阻塞服务器， 所以我们应该使用 SCAN 命令来代替 KEYS 命令。
- 通过使用 SORT 命令， 我们可以以多种不同的方式， 对储存在列表、集合以及有序集合里面的元素进行排序。
- 因为 DEL 命令在移除体积较大或者数量众多的键时可能会导致服务器阻塞， 所以我们应该使用异步移除命令 UNLINK 来代替 DEL 命令。
- 用户在执行 FLUSHDB 命令和 FLUSHALL 命令时可以带上 async 选项， 让这两个命令以异步方式执行， 从而避免服务器阻塞。
- SWAPDB 命令可以在完全不阻塞服务器的情况下， 对两个给定的数据库实行互换， 因此这个命令可以用于实现在线的数据库替换操作。

12. 自动过期

在构建应用时， 我们常常会碰到一些在特定时间之后就不再有用的数据， 比如说：

- 随着内容的不断更新， 一个网页的缓存可能在 5 分钟之后就没有阅读价值了， 为了让用户能够及时地获取到最新的信息， 程序必须定期地移除旧缓存并设置新缓存。

- 为了保障用户的信息安全， 应用通常会在用户登录一周或者一个月之后移除用户的会话信息， 然后通过强制要求用户重新登录来创建新的会话。
- 程序在进行聚合计算的时候， 常常会创建出大量临时数据， 这些数据在计算完毕之后通常就不再有用， 而且储存这些数据还会花费大量内存空间和硬盘空间。

在遇到上述情况时， 我们虽然可以自行编写程序来处理这些不再有用的数据， 但如果数据库本身能够提供自动移除无用数据的功能， 那么就会给我们带来非常大的方便。

为了解决这个问题， Redis 提供了自动的键过期功能（key expiring） 。 通过这个功能， 用户可以让特定的键在指定的时间之后自动被移除， 从而避免了需要在指定时间内手动执行删除操作的麻烦。

本章将对 Redis 的键过期功能进行介绍， 说明与该功能有关的各个命令的使用方法， 并展示如何使用这一功能去构建一些非常实用的程序。

12.1 EXPIRE、PEXPIRE：设置生存时间

用户可以通过执行 `EXPIRE` 命令或者 `PEXPIRE` 命令， 为键设置一个生存时间（TTL，time to live）： 键的生存时间在设置之后就会随着时间的流逝而不断地减少， 当一个键的生存时间被消耗殆尽时， Redis 就会移除这个键。

Redis 提供了 `EXPIRE` 命令用于设置秒级精度的生存时间， 它可以让键在指定的秒数之后自动被移除：

```
EXPIRE key seconds
```

而 `PEXPIRE` 命令则用于设置毫秒级精度的生存时间， 它可以让键在指定的毫秒数之后自动被移除：

```
PEXPIRE key milliseconds
```

`EXPIRE` 命令和 `PEXPIRE` 命令在生存时间设置成功时返回 `1` ； 如果用户给定的键并不存在， 那么命令返回 `0` 表示设置失败。

以下是一个使用 `EXPIRE` 命令的例子：

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

redis> EXPIRE msg 5
(integer) 1
```

```
redis> GET msg     -- 在 5 秒钟之内访问，键存在
"hello world"

redis> GET msg     -- 在 5 秒钟之后访问，键不再存在
(nil)
```

上面的代码通过执行 `EXPIRE` 命令为 `msg` 键设置了 5 秒钟的生存时间：

- 如果我们在 5 秒钟之内访问 `msg` 键， 那么 Redis 将返回 `msg` 键的值 `"hello world"` ；
- 但如果我们在 5 秒钟之后访问 `msg` 键， 那么 Redis 将返回一个空值， 因为 `msg` 键已经自动被移除了。

表 12-1 展示了 `msg` 键从设置生存时间到被移除的整个过程。

表 12-1 `msg` 键从设置生存时间到被移除的整个过程

时间（以秒为单位）	动作
0000	执行 `EXPIRE msg 5` ，将 `msg` 键的生存时间设置为 5 秒钟。
0001	`msg` 键的生存时间变为 4 秒钟。
0002	`msg` 键的生存时间变为 3 秒钟。
0003	`msg` 键的生存时间变为 2 秒钟。
0004	`msg` 键的生存时间变为 1 秒钟。
0005	`msg` 键因为过期被移除。

而以下则是一个使用 `PEXPIRE` 命令的例子：

```
redis> SET number 10086
OK

redis> PEXPIRE number 6500
(integer) 1

redis> GET number     -- 在 6500 毫秒（也即是 6.5 秒）之内访问，键存在
"10086"

redis> GET number     -- 在 6500 毫秒之后访问，键不再存在
(nil)
```

表 12-2 展示了 number 键从设置生存时间到被移除的整个过程。

表 12-2 number 键从设置生存时间到被移除的整个过程

时间（以毫秒为单位）	动作
0000	执行 PEXPIRE number 6500 ，将 number 键的生存时间设置为 6500 毫秒。
0001	number 键的生存时间变为 6499 毫秒。
0002	number 键的生存时间变为 6498 毫秒。
0003	number 键的生存时间变为 6497 毫秒。
……	……
6497	number 键的生存时间变为 3 毫秒。
6498	number 键的生存时间变为 2 毫秒。
6499	number 键的生存时间变为 1 毫秒。
6500	number 键因为过期而被移除。

12.1.1 更新键的生存时间

当用户对一个已经带有生存时间的键执行 EXPIRE 命令或是 PEXPIRE 命令时， 键原有的生存时间将会被移除， 并设置上新的生存时间。

举个例子， 如果我们执行以下命令， 将 msg 键的生存时间设置为 10 秒钟：

```
redis> EXPIRE msg 10
(integer) 1
```

然后在 10 秒钟之内执行以下命令：

```
redis> EXPIRE msg 50
(integer) 1
```

那么 msg 键的生存时间将被更新为 50 秒钟， 并重新开始倒数， 表 12-3 展示了这个更新过程。

表 12-3 msg 键生存时间的更新过程

时间（以秒为单位）	动作
0000	执行 EXPIRE msg 10 命令， 将 msg 键的生存时间设置为 10 秒钟。
0001	msg 键的生存时间变为 9 秒钟。
0002	msg 键的生存时间变为 8 秒钟。
0003	msg 键的生存时间变为 7 秒钟。
0004	执行 EXPIRE msg 50 命令， 将 msg 键的生存时间更新为 50 秒钟。
0005	msg 键的生存时间变为 49 秒钟。
0006	msg 键的生存时间变为 48 秒钟。
0007	msg 键的生存时间变为 47 秒钟。
……	……

12.1.2 其他信息

属性	值
复杂度	EXPIRE 命令和 PEXPIRE 命令的复杂度都为 O(1)
版本要求	EXPIRE 命令从 Redis 1.0.0 版本开始可用， PEXPIRE 命令从 Redis 2.6.0 版本开始可用。

12.2 示例：带有自动移除特性的缓存程序

用户在使用缓存程序的时候， 必须要考虑缓存的时效性： 对于内容不断变换的应用来说， 一份缓存存在的时间越长， 它与实际内容之间的差异往往也就越大， 因此为了让缓存能够及时地反映真实的内容， 程序必须定期对缓存进行更新。

本书前面在《字符串》一章曾经展示过怎样使用字符串键构建缓存程序， 但那个缓存程序有一个明显的缺陷， 那就是， 它无法自动移除过时的缓存。 如果我们真的要在实际中使用那个程序的话， 那么就必须再编写一个辅助程序来定期地删除旧缓存才行， 这样一来使用缓存将会变得非常麻烦。

幸运的是， 通过使用 Redis 的键过期功能， 我们可以为缓存程序加上自动移除特性， 并通过这个特性自动移除过期的、无效的缓存。

代码清单 12-1 展示了一个能够为缓存设置最大有效时间的缓存程序， 这个程序跟《字符串》一章展示的缓存程序的绝大部分代码都是相同的， 新程序的主要区别在于， 它除了会把指定的内容缓存起来之外， 还会使用 `EXPIRE` 命令为缓存设置生存时间， 从而使得缓存可以在指定时间到达之后自动被移除。

代码清单 12-1 带有自动移除特性的缓存程序： `/expire/unsafe_volatile_cache.py`

```
class VolatileCache:

    def __init__(self, client):
        self.client = client

    def set(self, key, value, timeout):
        """
        把数据缓存到键 key 里面，并为其设置过期时间。
        如果键 key 已经有值，那么使用新值去覆盖旧值。
        """
        self.client.set(key, value)
        self.client.expire(key, timeout)

    def get(self, key):
        """
        获取键 key 储存的缓存数据。
        如果键不存在，又或者缓存已经过期，那么返回 None 。
        """
        return self.client.get(key)
```

以下代码简单地展示了这个缓存程序的使用方法：

```
>>> from redis import Redis
>>> from unsafe_volatile_cache import VolatileCache
>>> client = Redis(decode_responses=True)
>>> cache = VolatileCache(client)
>>> cache.set("homepage", "<html><p>hello world</p></html>", 10)  # 设置缓存
>>> cache.get("homepage")  # 这个缓存在 10 秒钟之内有效
'<html><p>hello world</p></html>'
>>> cache.get("homepage")  # 10 秒钟过后，缓存自动被移除
>>>
```

12.3 SET 命令的 EX 选项和 PX 选项

在使用键过期功能时， 组合使用 SET 命令和 EXPIRE / PEXIRE 命令的做法非常常见， 比如上面展示的带有自动移除特性的缓存程序就是这样做的。

因为 SET 命令和 EXPIRE / PEXPIRE 命令组合使用的情况是如此的常见， 所以为了方便用户使用这两组命令， Redis 从 2.6.12 版本开始为 SET 命令提供 EX 选项和 PX 选项， 用户可以通过使用这两个选项的其中一个来达到同时执行 SET 命令和 EXPIRE / PEXPIRE 命令的效果：

```
SET key value [EX seconds] [PX milliseconds]
```

这也就是说， 如果我们之前执行的是 SET 命令和 EXPIRE 命令：

```
SET key value
EXPIRE key seconds
```

那么现在只需要执行一条带有 EX 选项的 SET 命令就可以了：

```
SET key value EX seconds
```

与此类似， 如果我们之前执行的是 SET 命令和 PEXPIRE 命令：

```
SET key value
PEXPIRE key milliseconds
```

那么现在只需要执行一条带有 PX 选项的 SET 命令就可以了：

```
SET key value PX milliseconds
```

12.3.1 组合命令的安全问题

使用带有 EX 选项或 PX 选项的 SET 命令除了可以减少命令的调用数量并提升程序的执行速度之外， 更重要的是保证了操作的原子性， 使得“为键设置值”和“为键设置生存时间”这两个操作可以一起执行。

比如说， 前面在实现带有自动移除特性的缓存程序时， 我们首先使用了 SET 命令设置缓存， 然后又使用了 EXPIRE 命令为缓存设置生存时间， 这相当于让程序依次地向 Redis 服务器发送以下两条命令：

```
SET key value

EXPIRE key timeout
```

因为这两条命令是完全独立的， 所以服务器在执行它们的时候， 就可能会出现 SET 命令被执行了， 但是 EXPIRE 命令却没有被执行的情况。 比如说， 如果 Redis 服务器在成功执行 SET 命令之后因为故障下线， 导致 EXPIRE 命令没有被执行， 那么 SET 命令设置的缓存就会一直存在， 而不会因为过期而自动被移除。

与此相反， 使用带有 EX 选项或 PX 选项的 SET 命令就没有这个问题： 当服务器成功执行了一条带有 EX 选项或 PX 选项的 SET 命令时， 键的值和生存时间都会同时被设置好， 因此程序就不会出现只设置了值但是却没有设置生存时间的情况。

基于上述原因， 我们把前面展示的缓存程序实现称之为“不安全”（unsafe）实现。 为了修复这个问题， 我们可以使用带有 EX 选项的 SET 命令来重写缓存程序， 重写之后的程序正如代码清单 12-2 所示。

代码清单 12-2 重写之后的缓存程序： /expire/volatile_cache.py

```
class VolatileCache:

    def __init__(self, client):
        self.client = client

    def set(self, key, value, timeout):
        """
        把数据缓存到键 key 里面，并为其设置过期时间。
        如果键 key 已经有值，那么使用新值去覆盖旧值。
        """
        self.client.set(key, value, ex=timeout)

    def get(self, key):
        """
        获取键 key 储存的缓存数据。
        如果键不存在，又或者缓存已经过期，那么返回 None 。
        """
        return self.client.get(key)
```

重写之后的缓存程序实现是“安全的”： 设置缓存和设置生存时间这两个操作要么就一起成功， 要么就一起失败， “设置缓存成功了，但是设置生存时间却失败了”这样的情况将不会出现。 后续的章节也会介绍如何通过 Redis 的

事务功能来保证执行多条命令时的安全性。

12.3.2 其他信息

属性	值
复杂度	O(1)
版本要求	带有 EX 选项和 PX 选项的 SET 命令从 Redis 2.6.12 版本开始可用。

12.4 示例：带有自动释放特性的锁

在前面的《字符串》一章， 我们曾经实现过一个锁程序， 它的其中一个缺陷就是无法自行释放： 如果锁的持有者因为故障下线， 那么锁将一直处于持有状态， 导致其他进程永远也无法获得锁。

为了解决这个问题， 我们可以在获取锁的同时， 通过 Redis 的自动过期特性为锁设置一个最大加锁时限， 这样的话， 即使锁的持有者由于故障下线， 锁也会在时限到达之后自动释放。

代码清单 12-3 展示了使用上述原理实现的锁程序。

代码清单 12-3 带有自动释放特性的锁：/expire/timing_lock.py

```
VALUE_OF_LOCK = "locking"

class TimingLock:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def acquire(self, timeout):
        """
        尝试获取一个带有秒级最大使用时限的锁，
        成功时返回 True ，失败时返回 False 。
        """
        result = self.client.set(self.key, VALUE_OF_LOCK, ex=timeout, nx=True)
        return result is not None

    def release(self):
```

```
        """
        尝试释放锁。
        成功时返回 True ，失败时返回 False 。
        """
        return self.client.delete(self.key) == 1
```

以下代码演示了这个锁的自动释放特性：

```
>>> from redis import Redis
>>> from timing_lock import TimingLock
>>> client = Redis()
>>> lock = TimingLock(client, "test-lock")
>>> lock.acquire(5)  # 获取一个在 5 秒钟之后自动释放的锁
True
>>> lock.acquire(5)  # 在 5 秒钟之内尝试再次获取锁，但是由于锁未被释放而失败
False
>>> lock.acquire(5)  # 在 5 秒钟之后尝试再次获取锁
True                 # 因为之前获取的锁已经自动被释放，所以这次将成功取得新的锁
```

12.5 EXPIREAT、PEXPIREAT：设置过期时间

Redis 用户不仅可以通过设置生存时间来让键在指定的秒数或毫秒数之后自动被移除， 还可以通过设置过期时间（expire time）， 让 Redis 在指定 UNIX 时间来临之后自动移除给定的键。

设置过期时间这一操作可以通过 `EXPIREAT` 命令或者 `PEXPIREAT` 命令来完成。 其中， `EXPIREAT` 命令接受一个键和一个秒级精度的 UNIX 时间戳为参数， 当系统的当前 UNIX 时间超过命令指定的 UNIX 时间时， 给定的键就会被移除：

```
EXPIREAT key seconds_timestamp
```

与此类似， `PEXPIREAT` 命令接受一个键和一个毫秒级精度的 UNIX 时间戳为参数， 当系统的当前 UNIX 时间超过命令指定的 UNIX 时间时， 给定的键就会被移除：

```
PEXPIREAT key milliseconds_timestamp
```

12.5.1 `EXPIREAT` 使用示例

如果我们想要让 msg 键在 UNIX 时间 1450005000 秒之后不再存在， 那么可以执行以下命令：

```
redis> EXPIREAT msg 1450005000
(integer) 1
```

在执行这个 EXPIREAT 命令之后， 如果我们在 UNIX 时间 1450005000 秒或之前访问 msg 键， 那么 Redis 将返回 msg 键的值：

```
redis> GET msg
"hello world"
```

另一方面， 如果我们在 UNIX 时间 1450005000 秒之后访问 msg 键， 那么 Redis 将返回一个空值， 因为这时 msg 键已经因为过期而自动被移除了：

```
redis> GET msg
(nil)
```

表 12-4 展示了 msg 键从设置过期时间到被移除的整个过程。

表 12-4 msg 键从设置过期时间到被移除的整个过程

UNIX 时间（以秒为单位）	动作
1450004000	执行 EXPIREAT msg 1450005000 命令， 将 msg 键的过期时间设置为 1450005000 秒。
1450004001	msg 键未过期，不做动作。
1450004002	msg 键未过期，不做动作。
1450004003	msg 键未过期，不做动作。
……	……
1450004999	msg 键未过期，不做动作。
1450005000	msg 键未过期，不做动作。
1450005001	系统当前的 UNIX 时间已经超过 1450005000 秒，移除 msg 键。

12.5.2 PEXPIREAT 命令使用示例

以下是一个使用 PEXPIREAT 命令设置过期时间的例子， 这个命令可以将 number 键的过期时间设置为 UNIX 时间 1450005000000 毫秒：

```
redis> PEXPIREAT number 1450005000000
(integer) 1
```

在 UNIX 时间 1450005000000 毫秒或之前访问 number 键可以得到它的值：

```
redis> GET number
"10086"
```

而在 UNIX 时间 1450005000000 毫秒之后访问 number 键则只会得到一个空值， 因为这时 number 键已经因为过期而自动被移除了：

```
redis> GET number
(nil)
```

表 12-5 展示了 number 键从设置过期时间到被移除的整个过程。

表 12-5 number 键从设置过期时间到被移除的整个过程

UNIX 时间（以毫秒为单位）	动作
1450003000000	执行 PEXPIREAT number 1450005000000 命令， 将 number 键的过期时间设置为 1450005000000 毫秒。
1450003000001	number 键未过期，不做动作。
1450003000002	number 键未过期，不做动作。
1450003000003	number 键未过期，不做动作。
……	……
1450004999999	number 键未过期，不做动作。
1450005000000	number 键未过期，不做动作。

UNIX 时间（以毫秒为单位）	动作
1450005000001	系统当前的 UNIX 时间已经超过 1450005000000 毫秒，移除 number 键。

12.5.3 更新键的过期时间

跟 EXPIRE / PEXPIRE 命令会更新键的生存时间一样， EXPIREAT / PEXPIREAT 命令也会更新键的过期时间： 如果用户在执行 EXPIREAT 命令或 PEXPIREAT 命令的时候， 给定键已经带有过期时间， 那么命令首先会移除键已有的过期时间， 然后再为其设置新的过期时间。

比如在以下调用中， 第二条 EXPIREAT 命令就将 msg 键的过期时间从原来的 1500000000 修改成了 1600000000 ：

```
redis> EXPIREAT msg 1500000000
(integer) 1

redis> EXPIREAT msg 1600000000
(integer) 1
```

12.5.4 自动过期特性的不足之处

无论是本节介绍的 EXPIREAT / PEXPIREAT ， 还是前面介绍的 EXPIRE / PEXIRE ， 它们都只能对整个键进行设置，而无法对键中的某个元素进行设置： 比如说， 用户只能对整个集合或者整个散列设置生存时间/过期时间， 但是却无法为集合中的某个元素或者散列中的某个字段单独设置生存时间/过期时间， 这也是目前 Redis 的自动过期功能不足的一个地方。

12.5.5 其他信息

属性	值
复杂度	EXPIREAT 命令和 PEXPIREAT 命令的复杂度都为 O(1) 。
版本要求	EXPIREAT 命令从 Redis 1.2.0 版本开始可用， PEXPIREAT 命令从 Redis 2.6.0 版本开始可用。

12.6 TTL、PTTL：获取键的剩余生存时间

在为键设置了生存时间或者过期时间之后， 用户可以使用 TTL 命令或者 PTTL 命令查看键的剩余生存时间， 也即是， 键还有多久才会因为过期而被移除。

其中， TTL 命令将以秒为单位返回键的剩余生存时间：

```
TTL key
```

而 PTTL 命令则会以毫秒为单位返回键的剩余生存时间：

```
PTTL key
```

作为例子， 以下代码展示了如何使用 TTL 命令和 PTTL 命令去获取 msg 键的剩余生存时间：

```
redis> TTL msg
(integer) 297       -- msg 键距离被移除还有 297 秒

redis> PTTL msg
(integer) 295561    -- msg 键距离被移除还有 295561 毫秒
```

12.6.1 没有剩余生存时间的键和不存在的键

如果给定的键存在， 但是并没有设置生存时间或者过期时间， 那么 TTL 命令和 PTTL 命令将返回 -1 ：

```
redis> SET song_title "Rise up, Rhythmetal"
OK

redis> TTL song_title
(integer) -1

redis> PTTL song_title
(integer) -1
```

另一方面， 如果给定的键并不存在， 那么 TTL 命令和 PTTL 命令将返回 -2 ：

```
redis> TTL not_exists_key
(integer) -2
```

```
redis> PTTL not_exists_key
(integer) -2
```

12.6.2 TTL 命令的精度问题

在使用 TTL 命令时， 我们有时候会碰到命令返回 0 的情况：

```
redis> TTL msg
(integer) 0
```

出现这种情况的原因在于 TTL 命令只能返回秒级精度的生存时间， 所以当给定键的剩余生存时间不足一秒钟时， TTL 命令只能返回 0 作为结果。 这时， 如果我们使用精度更高的 PTTL 命令去检查这些键， 那么就会看到它们实际的剩余生存时间， 表 12-6 非常详细地描述了这一情景。

表 12-6 PTTL 命令在 TTL 命令返回 0 时仍然可以检测到键的剩余生存时间

键的剩余生存时间（以毫秒为单位）	TTL 命令的返回值	PTTL 命令的返回值
1001	1	1001
1000	1	1000
999	0	999
998	0	998
997	0	997
……	……	……
2	0	2
1	0	1
0	0	0
-2（键已被移除）	-2	-2

12.6.3 其他信息

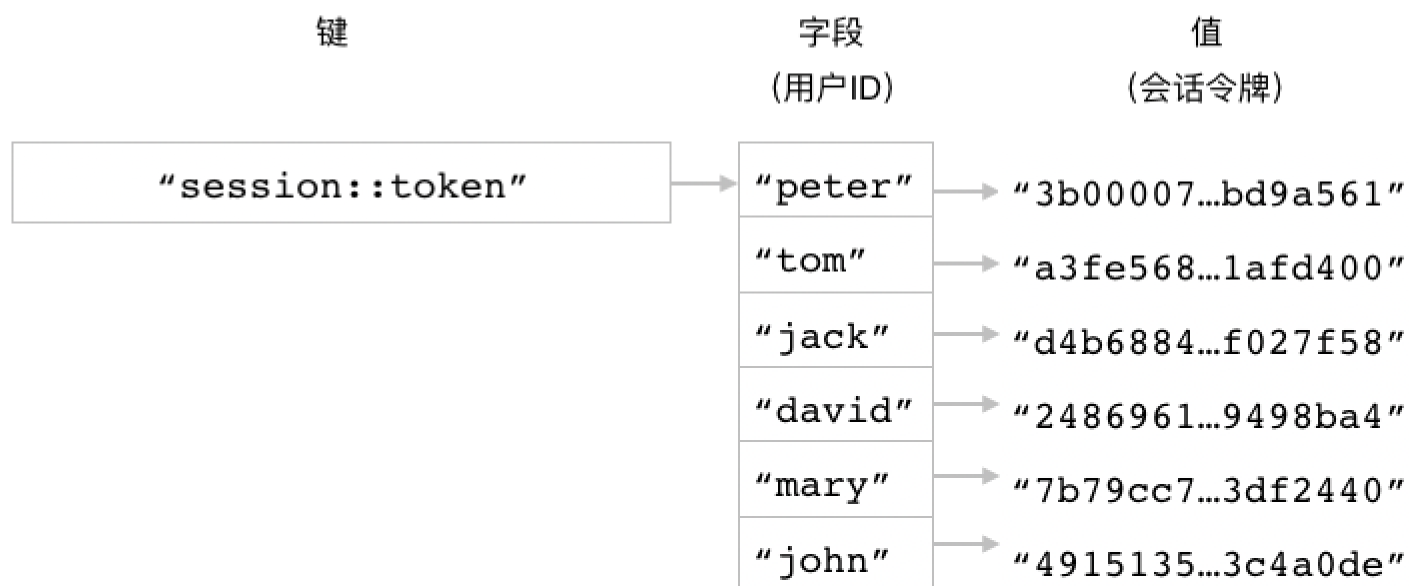
属性	值
复杂度	TTL 命令和 PTTL 命令的复杂度都为 O(1) 。

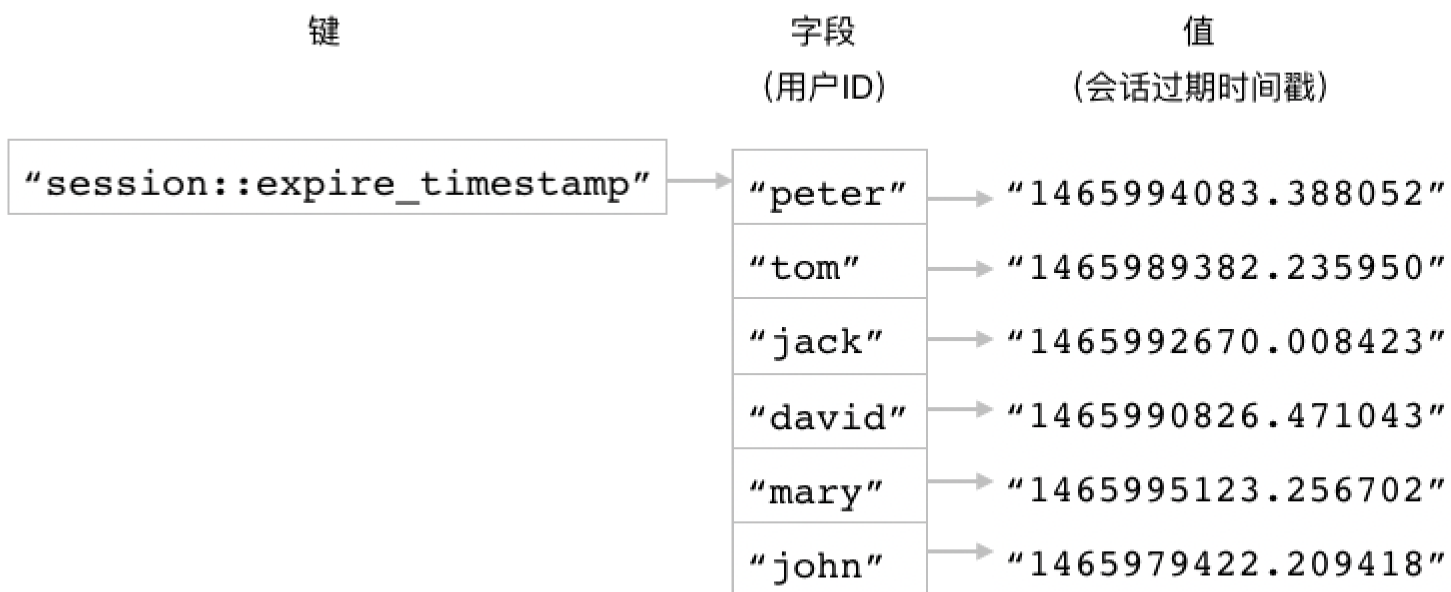
属性	值
版本要求	TTL 命令从 Redis 1.0.0 版本开始可用， PTTL 命令从 Redis 2.6.0 版本开始可用。

12.7 示例：自动过期的登录会话

在前面的《散列》一章， 我们了解到了如何使用散列去构建一个会话程序。 正如 12-1 所示， 当时的会话程序会使用两个散列分别储存会话的令牌以及过期时间戳。 这种做法虽然可行， 但是储存过期时间戳需要消耗额外的内存， 并且判断会话是否过期也需要用到额外的代码。

图 12-1 会话程序创建的散列数据结构


键
字段
（用户ID）
值
（会话令牌）
“session::token”
“peter”
“tom”
“jack”
“david”
“mary”
“john”
“3b00007…bd9a561”
“a3fe568…1afd400”
“d4b6884…f027f58”
“2486961…9498ba4”
“7b79cc7…3df2440”
“4915135…3c4a0de”


键
字段
（用户ID）
值
（会话过期时间戳）
“session::expire_timestamp”
“peter”
“tom”
“jack”
“david”
“mary”
“john”
“1465994083.388052”
“1465989382.235950”
“1465992670.008423”
“1465990826.471043”
“1465995123.256702”
“1465979422.209418”

在学习了 Redis 的自动过期特性之后， 我们可以对会话程序进行修改， 通过给会话令牌设置过期时间来让它在指定的时间之后自动被移除。 这样一来， 程序只需要检查会话令牌是否存在， 就能够知道是否应该让用户重新登录了。

代码清单 12-4 展示了修改之后的会话程序。 因为 Redis 的自动过期特性只能对整个键使用， 所以这个程序使用了字符串而不是散列来储存会话令牌， 但总的来说， 这个程序的逻辑跟之前的会话程序的逻辑基本相同。 不过由于新程序无需手动检查会话是否过期， 所以它的逻辑简洁了不少。

代码清单 12-4 带有自动过期特性的会话程序： `/expire/login_session.py`

```
import random
from hashlib import sha256

# 会话的默认过期时间
DEFAULT_TIMEOUT = 3600*24*30    # 一个月

# 会话状态
SESSION_NOT_LOGIN_OR_EXPIRED = "SESSION_NOT_LOGIN_OR_EXPIRED"
SESSION_TOKEN_CORRECT = "SESSION_TOKEN_CORRECT"
SESSION_TOKEN_INCORRECT = "SESSION_TOKEN_INCORRECT"

def generate_token():
    """
    生成一个随机的会话令牌。
    """
    random_string = str(random.getrandbits(256)).encode('utf-8')
    return sha256(random_string).hexdigest()


class LoginSession:

    def __init__(self, client, user_id):
        self.client = client
        self.user_id = user_id
        self.key = "user::{0}::token".format(user_id)

    def create(self, timeout=DEFAULT_TIMEOUT):
        """
        创建新的登录会话并返回会话令牌，
        可选的 timeout 参数用于指定会话的过期时间（以秒为单位）。
        """
        # 生成会话令牌
        token = generate_token()
```

```
        # 储存令牌，并为其设置过期时间
        self.client.set(self.key, token, ex=timeout)
        # 返回令牌
        return token

    def validate(self, input_token):
        """
        根据给定的令牌验证用户身份。
        这个方法有三个可能的返回值，分别对应三种不同情况：
        1. SESSION_NOT_LOGIN_OR_EXPIRED —— 用户尚未登录或者令牌已过期
        2. SESSION_TOKEN_CORRECT —— 用户已登录，并且给定令牌与用户令牌相匹配
        3. SESSION_TOKEN_INCORRECT —— 用户已登录，但给定令牌与用户令牌不匹配
        """
        # 获取用户令牌
        user_token = self.client.get(self.key)
        # 令牌不存在
        if user_token is None:
            return SESSION_NOT_LOGIN_OR_EXPIRED
        # 令牌存在并且未过期，那么检查它与给定令牌是否一致
        if input_token == user_token:
            return SESSION_TOKEN_CORRECT
        else:
            return SESSION_TOKEN_INCORRECT

    def destroy(self):
        """
        销毁会话。
        """
        self.client.delete(self.key)
```

以下代码展示了这个会话程序的基本使用方法：

```
>>> from redis import Redis
>>> from login_session import LoginSession
>>> client = Redis(decode_responses=True)
>>> uid = "peter"
>>> session = LoginSession(client, uid)  # 创建会话
>>> token = session.create()             # 创建令牌
>>> token
'89e77eb856a3383bb8718286802d32f6d40e135c08dedcccd143a5e8ba335d44'
>>> session.validate("wrong token")      # 验证令牌
'SESSION_TOKEN_INCORRECT'
>>> session.validate(token)
'SESSION_TOKEN_CORRECT'
>>> session.destroy()                    # 销毁令牌
```

```
>>> session.validate(token)              # 令牌已不存在
'SESSION_NOT_LOGIN_OR_EXPIRED'
```

为了演示这个会话程序的自动过期特性， 我们可以创建一个有效期非常短的令牌， 并在指定的时间后再次尝试验证该令牌：

```
>>> token = session.create(timeout=3)  # 创建有效期为三秒钟的令牌
>>> session.validate(token)            # 三秒内访问
'SESSION_TOKEN_CORRECT'
>>> session.validate(token)            # 超过三秒之后，令牌已被自动销毁
'SESSION_NOT_LOGIN_OR_EXPIRED'
```

12.8 示例：自动淘汰冷门数据

本章开头在介绍 EXPIRE 命令和 PEXPIRE 命令的时候曾经提到过， 当用户对一个已经带有生存时间的键执行 EXPIRE 命令或是 PEXPIRE 命令时， 键原有的生存时间将被新的生存时间取代。 值得一提的是， 这个特性可以用于淘汰冷门数据并保留热门数据。

举个例子， 前面的《有序集合》一章曾经介绍过如何使用有序集合来实现自动补全功能， 但是如果我们仔细地分析这个自动补全程序， 就会发现它有一个潜在的问题： 为了实现自动补全功能， 程序需要创建大量自动补全结果， 而补全结果的数量越多、体积越大， 需要耗费的内存也会越多。

为了尽可能地节约内存， 一个高效的自动补全程序应该只储存热门关键字的自动补全结果， 并移除那些无人访问的冷门关键字的自动补全结果。 要做到这一点， 其中一种方法就是使用《有序集合》里面介绍过的排行榜程序， 为用户输入的关键字构建一个排行榜， 然后定期地删除排名靠后关键字的自动补全结果。

排行榜的方法虽然可行， 但是却需要使用程序定期删除自动补全结果， 使用起来相当麻烦。 一个更方便也更优雅的方法， 就是使用 EXPIRE 命令和 PEXPIRE 命令的更新特性去实现自动的冷门数据淘汰机制： 为此， 我们可以修改自动补全程序， 让它在每次处理用户输入的时候， 为相应关键字的自动补全结果设置生存时间。 这样一来， 对于用户经常输入的那些关键字， 它们的自动补全结果的生存时间将会不断得到更新， 从而产生出一种“续期”效果， 使得热门关键字的自动补全结果可以不断地存在下去， 而冷门关键字的自动补全结果则会由于生存时间得不到更新而自动被移除。

经过上述修改， 自动补全程序就可以在无需手动删除冷门数据的情况下， 通过自动的数据淘汰机制达到节约内存的目的， 代码清单 12-5 展示了修改后的自动补全程序。

代码清单 12-5 能够自动淘汰冷门数据的自动补全程序：/expire/auto_complete.py

```
class AutoComplete:

    def __init__(self, client):
        self.client = client

    def feed(self, content, weight=1, timeout=None):
        """
        根据用户输入的内容构建自动补全结果，
        其中 content 参数为内容本身，而可选的 weight 参数则用于指定内容的权重值，
        至于可选的 timeout 参数则用于指定自动补全结果的保存时长（单位为秒）。
        """
        for i in range(1, len(content)):
            key = "auto_complete::" + content[:i]
            self.client.zincrby(key, weight, content)
            if timeout is not None:
                self.client.expire(key, timeout)  # 设置/更新键的生存时间

    def hint(self, prefix, count):
        """
        根据给定的前缀 prefix ，获取 count 个自动补全结果。
        """
        key = "auto_complete::" + prefix
        return self.client.zrevrange(key, 0, count-1)
```

在以下代码中，我们同时向自动补全程序输入了 "Redis" 和 "Coffee" 这两个关键字，并分别为它们的自动补全结果设置了 10 秒钟的生存时间：

```
>>> from redis import Redis
>>> from auto_complete import AutoComplete
>>> client = Redis(decode_responses=True)
>>> ac = AutoComplete(client)
>>> ac.feed("Redis", timeout=10); ac.feed("Coffee", timeout=10)  # 同时执行两个调用
```

然后在 10 秒钟之内，我们再次输入 "Redis" 关键字，并同样为它的自动补全结果设置 10 秒钟的生存时间：

```
>>> ac.feed("Redis", timeout=10)
```

现在，在距离最初的 feed() 调用执行十多秒钟之后，如果我们执行 hint() 方法，并尝试获取 "Re" 前缀和 "Co" 前缀的自动补全结果，那么就会发现，只有 "Redis" 关键字的自动补全结果还保留着，而 "Coffee" 关键字

的自动补全结果已经因为过期而被移除了：

```
>>> ac.hint("Re", 10)
['Redis']

>>> ac.hint("Co", 10)
[]
```

表 12-7 完整地展示了在执行以上代码时， "Redis" 关键字的自动补全结果是如何进行续期的， 而 "Coffee" 关键字的自动补全结果又是如何被移除的。 在这个表格中， "Redis" 关键字代表的就是热门数据， 而 "Coffee" 关键字代表的就是冷门数据： 一直有用户访问的热门数据将持续地存在下去， 而无人问津的冷门数据则会因为过期而被移除。

表 12-7 冷门数据淘汰示例

时间（以秒为单位）	"Redis" 关键字的自动补全结果	"Coffee" 关键字的自动补全结果
0000	执行 `ac.feed("Redis", timeout=10)` ，将自动补全结果的生存时间设置为 10 秒钟。	执行 `ac.feed("Coffee", timeout=10)` ， 将自动补全结果的生存时间设置为 10 秒钟。
0001	自动补全结果的生存时间变为 9 秒钟。	自动补全结果的生存时间变为 9 秒钟。
0002	自动补全结果的生存时间变为 8 秒钟。	自动补全结果的生存时间变为 8 秒钟。
……	……	……
0007	执行 `ac.feed("Redis", timeout=10)` ，将自动补全结果的生存时间更新为 10 秒钟。	自动补全结果的生存时间变为 3 秒钟。
0008	自动补全结果的生存时间变为 9 秒钟。	自动补全结果的生存时间变为 2 秒钟。
0009	自动补全结果的生存时间变为 8 秒钟。	自动补全结果的生存时间变为 1 秒钟。
0010	自动补全结果的生存时间变为 7 秒钟。	"Coffee" 关键字的自动补全结果因为过期而被移除。

时间（以秒为单位）	"Redis" 关键字的自动补全结果	"Coffee" 关键字的自动补全结果
0011	自动补全结果的生存时间变为 6 秒钟。	自动补全结果已不存在。
0012	自动补全结果的生存时间变为 5 秒钟。	自动补全结果已不存在。
0013	执行 `ac.hint("Re", 10)` ， 返回结果 `['Redis']` 。	执行 `ac.hint("Co", 10)` ， 返回空列表 `[]` 为结果。

除了自动补全程序之外， 我们还可以把这一机制应用到其他需要淘汰冷门数据的程序上面。 为了做到这一点， 我们必须理解上面所说的“不断更新键的生存时间，使得它一直存在”这个原理。

12.9 重点回顾

- `EXPIRE` 命令和 `PEXPIRE` 命令可以为键设置生存时间， 当键的生存时间随着时间的流逝而消耗殆尽时， 键就会被移除。
- 对已经带有生存时间的键执行 `EXPIRE` 命令或是 `PEXPIRE` 命令， 将导致键已有的生存时间被新的生存时间替代。
- 为了方便用户， Redis 给 `SET` 命令增加了 `EX` 和 `PX` 两个选项， 它们可以让用户在执行 `SET` 命令的同时，执行 `EXPIRE` 命令或是 `PEXPIRE` 命令。
- `EXPIREAT` 命令和 `PEXPIREAT` 命令可以为键设置 UNIX 时间戳格式的过期时间， 当系统时间超过这个过期时间时， 键就会被移除。
- Redis 的自动过期特性只能应用于整个键， 它无法对键中的某个元素单独执行过期操作。
- `TTL` 命令和 `PTTL` 命令可以分别以秒级和毫秒级这两种精度来获取键的剩余生存时间。
- 通过重复对键执行 `EXPIRE` 命令或是 `PEXPIRE` 命令， 程序可以构建出一种自动淘汰冷数据并保留热数据的机制。

13. 流水线与事务

在前面的内容中， 我们学习了如何使用不同的命令去操作 Redis 提供的各种数据结构， 学习了如何使用数据库命令去对数据库中的各个键进行操作， 以及如何使用自动过期特性的相关命令去为键设置过期时间或者生存时间。

在执行这些命令的时候， 我们总是单独地执行每个命令， 也即是说， 先将一个命令发送到服务器， 等服务器执行完这个命令并将结果返回给客户端之后， 再执行下一个命令， 以此类推， 直到所有命令都执行完毕为止。

这种执行命令的方式虽然可行， 但在性能方面却不是最优的， 并且在执行时可能还会出现一些非常隐蔽的错误。为了解决这些问题， 本章将会介绍 Redis 的流水线特性以及事务特性， 前者可以有效地提升 Redis 程序的性能， 而后者则可以避免单独执行命令时可能会出现的一些错误。

13.1 流水线

在一般情况下， 用户每执行一个 Redis 命令， Redis 客户端和 Redis 服务器就需要执行以下步骤：

1. 客户端向服务器发送命令请求。
2. 服务器接收命令请求， 并执行用户指定的命令调用， 然后产生相应的命令执行结果。
3. 服务器向客户端返回命令的执行结果。
4. 客户端接收命令的执行结果， 并向用户进行展示。

跟大多数网络程序一样， 执行 Redis 命令所消耗的大部分时间都用在了发送命令请求和接收命令结果上面：Redis 服务器处理一个命令请求通常只需要很短的时间， 但客户端将命令请求发送给服务器以及服务器向客户端返回命令结果的过程却需要花费不少时间。 通常情况下， 程序需要执行的 Redis 命令越多， 它需要进行的网络通讯操作也会越多， 程序的执行速度也会因此而变慢。

为了解决这个问题， 我们可以使用 Redis 提供的流水线特性： 这个特性允许客户端把任意多条 Redis 命令请求打包在一起， 然后一次性地将它们全部发送给服务器， 而服务器则会在流水线包含的所有命令请求都处理完毕之后， 一次性地将它们的执行结果全部返回给客户端。

通过使用流水线特性， 我们可以将执行多个命令所需的网络通讯次数从原来的 N 次降低为一次， 这可以大幅度地减少程序在网络通讯方面耗费的时间， 使得程序的执行效率得到显著的提升。

作为例子， 图 13-1 展示了在没有使用流水线的情况下， 执行三个 Redis 命令产生的网络通讯示意图， 而 13-2 则展示了在使用流水线的情况下， 执行相同 Redis 命令产生的网络通讯示意图。 可以看到， 在使用了流水线之后， 程序进行网络通讯的次数从原来的三次降低成了一次。

图 13-1 在不使用流水线的情况下执行三个 Redis 命令产生的网络通讯操作

图 13-2 在使用流水线的情况下执行三个 Redis 命令产生的网络通讯操作

虽然 Redis 服务器提供了流水线特性， 但这个特性还需要客户端支持才能使用。 幸运的是， 包括 redis-py 在内的绝大部分 Redis 客户端都提供了对流水线特性的支持， 因此 Redis 用户在绝大部分情况下都能够享受到流水线特性带来的好处。

为了在 redis-py 客户端中使用流水线特性， 我们需要用到 `pipeline()` 方法， 调用这个方法会返回一个流水线对象， 用户只需要像平时执行 Redis 命令那样， 使用流水线对象调用相应的命令方法， 就可以把想要执行的 Redis 命令放入到流水线里面。

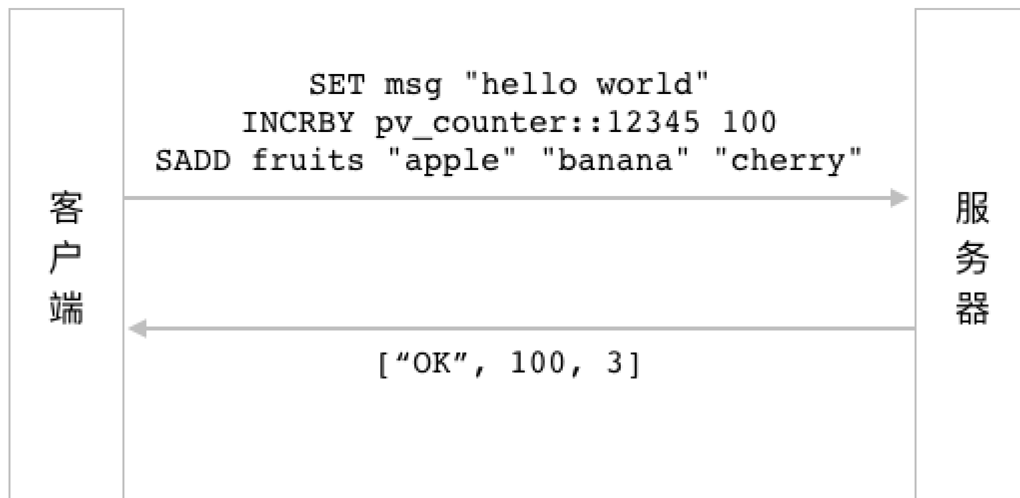
作为例子， 以下代码展示了如何以流水线方式执行 `SET` 、 `INCRBY` 和 `SADD` 命令：

```
>>> from  redis import Redis
>>> client = Redis(decode_responses=True)
>>> pipe = client.pipeline(transaction=False)
>>> pipe.set("msg", "hello world")
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> pipe.incrby("pv_counter::12345", 100)
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> pipe.sadd("fruits", "apple", "banana", "cherry")
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> pipe.execute()
[True, 100, 3]
```

这段代码首先使用 `pipeline()` 方法创建了一个流水线对象， 并将这个对象储存到了 `pipe` 变量里面 （`pipeline()` 方法中的 `transaction=False` 参数表示不在流水线中使用事务，这个参数的具体意义将在本章后续内容中说明）。在此之后， 程序通过流水线对象分别调用了 `set()` 方法、 `incrby()` 方法和 `sadd()` 方法， 将这三个方法对应的命令调用放入到了流水线队列里面。 最后， 程序调用流水线对象的 `execute()` 方法， 将队列中的三个命令调用打包发送给服务器， 而服务器则会在执行完这些命令之后， 把各个命令的执行结果依次放入到一个列表里面， 然后将这个列表返回给客户端。

图 13-3 展示了以上代码在执行期间， redis-py 客户端与 Redis 服务器之间的网络通讯情况。

图 13-3 使用流水线发送 Redis 命令



Note: 流水线使用注意事项

虽然 Redis 服务器并不会限制客户端在流水线里面包含的命令数量， 但是却会为客户端的输入缓冲区设置默认值为 1GB 的体积上限： 当客户端发送的数据量超过这一限制时， Redis 服务器将强制关闭该客户端。 因此用户在使用流水线特性时， 最好不要一下子把大量命令或者一些体积非常庞大的命令放到同一个流水线里面执行， 以免触碰到 Redis 的这一限制。

除此之外， 很多客户端本身也带有隐含的缓冲区大小限制， 如果你在使用流水线特性的过程中， 发现某些流水线命令没有被执行， 又或者流水线返回的结果不完整， 那么很可能就是你的程序触碰到了客户端内置的缓冲区大小限制。 在遇到这种情况时， 请缩减流水线命令的数量及其体积， 然后再进行尝试。

13.2 示例：使用流水线优化随机键创建程序

在《数据库》一章的《示例：数据库取样程序》一节中， 我们曾经展示过代码清单 13-1 所示的程序， 它可以根据用户给定的数量创建多个类型随机的数据库键。

代码清单 13-1 原版的随机键创建程序： `/database/create_random_type_keys.py`

```
import random
```

```python
def create_random_type_keys(client, number):
    """
    在数据库中创建指定数量的类型随机键。
    """
    for i in range(number):
        # 构建键名
        key = "key:{0}".format(i)
        # 从六个键创建函数中随机选择一个
        create_key_func = random.choice([
            create_string,
            create_hash,
            create_list,
            create_set,
            create_zset,
            create_stream
        ])
        # 实际地创建键
        create_key_func(client, key)

def create_string(client, key):
    client.set(key, "")

def create_hash(client, key):
    client.hset(key, "", "")

def create_list(client, key):
    client.rpush(key, "")

def create_set(client, key):
    client.sadd(key, "")

def create_zset(client, key):
    client.zadd(key, {"":0})

def create_stream(client, key):
    client.xadd(key, {"":""})
```

通过分析代码可知， 这个程序每创建一个键， redis-py 客户端就需要与 Redis 服务器进行一次网络通讯： 考虑到这个程序执行的都是一些非常简单的命令， 每次网络通讯只执行一个命令的做法无疑是非常低效的。 为了解决这个问题， 我们可以使用流水线把程序生成的所有命令都包裹起来， 这样的话， 创建多个随机键所需的网络通讯次数就会从原来的 N 次降低为 1 次。 代码清单 13-2 展示了修改之后的流水线版本随机键创建程序。

代码清单 13-2 流水线版本的随机键创建程序： `/pipeline-and-transaction/create_random_type_keys.py`

```python
import random

def create_random_type_keys(client, number):
    """
    在数据库中创建指定数量的类型随机键。
    """
    # 创建流水线对象
    pipe = client.pipeline(transaction=False)
    for i in range(number):
        # 构建键名
        key = "key:{0}".format(i)
        # 从六个键创建函数中随机选择一个
        create_key_func = random.choice([
            create_string,
            create_hash,
            create_list,
            create_set,
            create_zset,
            create_stream
        ])
        # 把待执行的 Redis 命令放入流水线队列中
        create_key_func(pipe, key)
    # 执行流水线包裹的所有命令
    pipe.execute()

def create_string(client, key):
    client.set(key, "")

def create_hash(client, key):
    client.hset(key, "", "")

def create_list(client, key):
    client.rpush(key, "")

def create_set(client, key):
    client.sadd(key, "")

def create_zset(client, key):
    client.zadd(key, {"":0})

def create_stream(client, key):
    client.xadd(key, {"":""})
```

即使只在本地网络里面进行测试， 新版的随机键创建程序也有 5 倍的性能提升。 当客户端与服务器处于不同的网络之中， 特别是它们之间的连接速度较慢时， 流水线版本的性能提升还会更大。

13.3 事务

虽然 Redis 的 LPUSH 命令和 RPUSH 命令允许用户一次向列表推入多个元素， 但是列表的弹出命令 LPOP 和 RPOP 每次却只能弹出一个元素：

```
redis> RPUSH lst 1 2 3 4 5 6    -- 一次推入五个元素
(integer) 6

redis> LPOP lst    -- 弹出一个元素
"1"

redis> LPOP lst
"2"

redis> LPOP lst
"3"
```

因为 Redis 并没有提供能够一次弹出多个列表元素的命令， 所以为了方便地执行这一任务， 用户可能会写出代码清单 13-3 所示的代码。

代码清单 13-3 不安全的 mlpop() 实现： /pipeline-and-transaction/unsafe_mlpop.py

```python
def mlpop(client, list_key, number):
    # 用于储存被弹出元素的结果列表
    items = []
    for i in range(number):
        # 执行 LPOP 命令，弹出一个元素
        poped_item = client.lpop(list_key)
        # 将被弹出的元素追加到结果列表末尾
        items.append(poped_item)
    # 返回结果列表
    return items
```

mlpop() 函数通过将多条 LPOP 命令发送至服务器来达到弹出多个元素的目的。 遗憾的是， 这个函数并不能保证它发送的所有 LPOP 命令都会被服务器执行： 如果服务器在执行多个 LPOP 命令的过程中下线了， 那么 mlpop() 发送的这些 LPOP 命令将只有一部分会被执行。

举个例子， 如果我们执行调用 `mlpop(client, "lst", 3)` ， 尝试从 `"lst"` 列表中弹出三个元素， 那么 `mlpop()` 将向服务器连续发送三个 `LPOP` 命令， 但如果服务器在顺利执行前两个 `LPOP` 命令之后因为故障下线了， 那么 `"lst"` 列表将只有两个元素会被弹出。

需要注意的是， 即使我们使用上一节介绍的流水线特性， 把多条 `LPOP` 命令打包在一起发送， 也不能保证所有命令都会被服务器执行： 这是因为流水线只能保证多条命令会一起被发送至服务器， 但它并不保证这些命令都会被服务器执行。

为了实现一个正确且安全的 `mlpop()` 函数， 我们需要一种能够让服务器将多个命令打包起来一并执行的技术， 而这正是本节将要介绍的事务特性：

- 事务可以将多个命令打包成一个命令来执行， 当事务成功执行时， 事务中包含的所有命令都会被执行；
- 相反地， 如果事务没有成功执行， 那么它包含的所有命令都不会被执行。

通过使用事务， 用户可以保证自己想要执行的多个命令要么全部都被执行， 要么就一个都不执行。 以 `mlpop()` 函数为例， 通过使用事务， 我们可以保证被调用的多个 `LPOP` 命令要么全部都执行， 要么就一个都不执行， 从而杜绝了只有其中一部分 `LPOP` 命令被执行的情况出现。

本节接下来的内容将会介绍 Redis 事务特性的使用方法以及相关事项， 至于事务版本 `mlpop()` 函数的具体实现则会留到下一节再行介绍。

13.3.1 MULTI：开启事务

用户可以通过执行 `MULTI` 命令来开启一个新的事务， 这个命令在成功执行之后将返回 `OK` ：

```
MULTI
```

在一般情况下， 除了少数阻塞命令之外， 用户键入到客户端里面的数据操作命令总是会立即执行：

```
redis> SET title "Hand in Hand"
OK

redis> SADD fruits "apple" "banana" "cherry"
(integer) 3

redis> RPUSH numbers 123 456 789
(integer) 3
```

但是当一个客户端执行 MULTI 命令之后， 它就进入了事务模式， 这时用户键入的所有数据操作命令都不会立即执行， 而是会按顺序被放入到一个事务队列里面， 等待事务执行时再统一执行。

比如说， 以下代码就展示了在 MULTI 命令执行之后， 将一个 SET 命令、一个 SADD 命令和一个 RPUSH 命令放入到事务队列里面的例子：

```
redis> MULTI
OK

redis> SET title "Hand in Hand"
QUEUED

redis> SADD fruits "apple" "banana" "cherry"
QUEUED

redis> RPUSH numbers 123 456 789
QUEUED
```

正如代码所示， 服务器在把客户端发送的命令放入到事务队列之后， 会向客户端返回一个 QUEUED 作为结果。

其他信息

属性	值
复杂度	O(1)
版本要求	MULTI 命令从 Redis 1.2.0 版本开始可用。

13.3.2 EXEC：执行事务

在使用 MULTI 命令开启事务并将任意多个命令放入到事务队列之后， 用户就可以通过执行 EXEC 命令来执行事务了：

```
EXEC
```

当事务成功执行时， EXEC 命令将返回一个列表作为结果， 这个列表会按照命令的入队顺序依次包含各个命令的执行结果。

作为例子， 以下代码展示了一个事务从开始到执行的整个过程：

```
redis> MULTI                       -- 1) 开启事务
OK

redis> SET title "Hand in Hand"    -- 2) 命令入队
QUEUED

redis> SADD fruits "apple" "banana" "cherry"
QUEUED

redis> RPUSH numbers 123 456 789
QUEUED

redis> EXEC             -- 3) 执行事务
1) OK                   -- SET 命令的执行结果
2) (integer) 3          -- SADD 命令的执行结果
3) (integer) 3          -- RPUSH 命令的执行结果
```

其他信息

属性	值
复杂度	事务包含的所有命令的复杂度之和。
版本要求	EXEC 命令从 Redis 1.2.0 版本开始可用。

13.3.3 DISCARD：放弃事务

如果用户在开启事务之后， 不想要执行事务而是想要放弃事务， 那么只需要执行以下命令即可：

```
DISCARD
```

DISCARD 命令会清空事务队列中已有的所有命令， 并让客户端退出事务模式， 最后返回 OK 表示事务已被取消。

以下代码展示了一个使用 DISCARD 命令放弃事务的例子：

```
redis> MULTI
OK

redis> SET page_counter 10086
QUEUED
```

```
redis> SET download_counter 12345
QUEUED

redis> DISCARD
OK
```

其他信息

属性	值
复杂度	O(N)，其中 N 为事务队列包含的命令数量。
版本要求	DISCARD 命令从 Redis 2.0.0 版本开始可用。

13.3.4 事务的安全性

在对数据库的事务特性进行介绍时， 人们一般都会通过数据库对 ACID 性质的支持程度去判断数据库的事务是否安全。

具体来说， Redis 的事务总是具有 ACID 性质中的 A、C、I 性质：

- 原子性（Atomic）： 如果事务成功执行， 那么事务中包含的所有命令都会被执行； 相反， 如果事务执行失败， 那么事务中包含的所有命令都不会被执行。
- 一致性（Consistent）： Redis 服务器会对事务及其包含的命令进行检查， 确保无论事务是否执行成功，事务本身都不会对数据库造成破坏。
- 隔离性（Isolate）： 每个 Redis 客户端都拥有自己独立的事务队列， 并且每个 Redis 事务都是独立执行的， 不同事务之间不会互相干扰。

除此之外， 当 Redis 服务器运行在特定的持久化模式之下时， Redis 的事务也具有 ACID 性质中的 D 性质：

- 耐久性（Durable）： 当事务执行完毕时， 它的结果将被储存在硬盘里面， 即使服务器在此之后停机， 事务对数据库所做的修改也不会丢失。

稍后的《持久化》一章将对事务的耐久性做补充说明。

13.3.5 事务对服务器的影响

因为事务在执行时会独占服务器， 所以用户应该避免在事务里面执行过多命令， 更不要将一些需要大量计算的命令放入到事务里面， 以免造成服务器阻塞。

13.3.6 流水线与事务

正如前面所言， 流水线与事务虽然在概念上有些相似， 但是在作用上却并不相同： 流水线的作用是将多个命令打包然后一并发送至服务器， 而事务的作用则是将多个命令打包然后让服务器一并执行它们。

因为 Redis 的事务在 `EXEC` 命令执行之前并不会产生实际效果， 所以很多 Redis 客户端都会使用流水线去包裹事务命令， 并将入队的命令缓存在本地， 等到用户键入 `EXEC` 命令之后， 再将所有事务命令通过流水线一并发送至服务器， 这样客户端在执行事务时就可以达到“打包发送，打包执行”的最优效果。

本书使用的 redis-py 客户端就是这样处理事务命令的客户端之一， 当我们使用 `pipeline()` 方法开启一个事务时， redis-py 默认将使用流水线包裹事务队列中的所有命令。

举个例子， 对于以下代码来说：

```
>>> from redis import Redis
>>> client = Redis(decode_responses=True)
>>> transaction = client.pipeline()             # 开启事务
>>> transaction.set("title", "Hand in Hand")    # 将命令放入事务队列
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> transaction.sadd("fruits", "apple", "banana", "cherry")
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> transaction.rpush("numbers", "123", "456", "789")
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> transaction.execute()                       # 执行事务
[True, 3, 3L]
```

在执行 `transaction.execute()` 调用时， redis-py 将通过流水线向服务器发送以下命令：

```
MULTI
SET title "Hand in Hand"
SADD fruits "apple" "banana" "cherry"
RPUSH numbers "123" "456" "789"
EXEC
```

这样的话， 无论事务包含了多少个命令， redis-py 也只需要与服务器进行一次网络通讯。

另一方面， 如果用户只需要用到流水线特性而不是事务特性， 那么可以在调用 `pipeline()` 方法时通过 `transaction=False` 参数显式地关闭事务特性， 就像这样：

```
>>> pipe = client.pipeline(transaction=False)    # 开启流水线
>>> pipe.set("download_counter", 10086)          # 将命令放入流水线队列
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> pipe.get("download_counter")
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> pipe.hset("user::123::profile", "name", "peter")
Pipeline<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> pipe.execute()                               # 将流水线队列中的命令打包发送至服务器
[True, '10086', 1L]
```

在执行 `pipe.execute()` 调用时， redis-py 将通过流水线向服务器发送以下命令：

```
SET download_counter 10086
GET download_counter
HSET user::123::profile "name" "peter"
```

因为这三个命令并没有被事务包裹， 所以客户端只保证它们会一并被发送至服务器， 至于这些命令在何时会以何种方式执行则由服务器本身决定。

13.4 示例： 实现 MLPOP 函数

在了解了事务的使用方法之后， 现在是时候用它来重新实现一个安全且正确的 `mlpop` 函数了： 为此， 我们需要使用事务包裹被执行的所有 `LPOP` 命令， 就像代码清单 13-4 所示的那样。

代码清单 13-4 事务版本的 `mlpop()` 函数： `/pipeline-and-transaction/mlpop.py`

```
def mlpop(client, list_key, number):
    # 开启事务
    transaction = client.pipeline()
    # 将多个 LPOP 命令放入事务队列
    for i in range(number):
        transaction.lpop(list_key)
    # 执行事务
    return transaction.execute()
```

新版的 `mlpop()` 函数通过事务确保自己发送的多个 `LPOP` 命令要么全部都执行， 要么就全部都不执行， 以此来避免只有一部分 `LPOP` 命令被执行了的情况出现。

举个例子， 如果我们执行函数调用：

```
mlpop(client, "lst", 3)
```

那么 `mlpop()` 函数将向服务器发送以下命令序列：

```
MULTI
LPOP "lst"
LPOP "lst"
LPOP "lst"
EXEC
```

如果这个事务能够成功执行， 那么它包含的三个 `LPOP` 命令也将成功执行； 相反， 如果这个事务执行失败， 那么它包含的三个 `LPOP` 命令也不会被执行。

以下是新版 `mlpop()` 函数的实际运行示例：

```
>>> from redis import Redis
>>> from mlpop import mlpop
>>> client = Redis(decode_responses=True)
>>> client.rpush("lst", "123", "456", "789")     # 向列表右端推入三个元素
3L
>>> mlpop(client, "lst", 3)                       # 从列表左端弹出三个元素
['123', '456', '789']
```

13.5 带有乐观锁的事务

本书在前面的《字符串》一章实现了具有基本获取和释放功能的锁程序， 并在《自动过期》一章为该程序加上了自动释放功能， 但是这两个锁程序都有一个问题， 那就是， 它们的释放操作都是不安全的：

- 无论某个客户端是否是锁的持有者， 只要它调用 `release()` 方法， 锁就会被释放。
- 在锁被占用期间， 如果某个不是持有者的客户端错误地调用了 `release()` 方法， 那么锁将在持有者不知情的情况下释放， 并导致系统中同时存在多个锁。

为了解决这个问题， 我们需要修改锁实现， 给它加上身份验证功能：

- 客户端在尝试获取锁的时候， 除了需要输入锁的最大使用时限之外， 还需要输入一个代表身份的标识符， 当客户端成功取得锁时， 程序将把这个标识符储存在代表锁的字符串键里面。
- 当客户端调用 `release()` 方法时， 它需要将自己的标识符传给 `release()` 方法， 而 `release()` 方法则需要验证客户端传入的标识符与锁键储存的标识符是否相同， 以此来判断调用 `release()` 方法的客户端是否就是锁的持有者， 从而决定是否释放锁。

根据以上描述， 我们可能会写出代码清单 13-5 所示的代码。

代码清单 13-5 不安全的锁实现： `/pipeline-and-transaction/unsafe_identity_lock.py`

```python
class IdentityLock:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def acquire(self, identity, timeout):
        """
        尝试获取一个带有身份标识符和最大使用时限的锁，
        成功时返回 True ，失败时返回 False 。
        """
        result = self.client.set(self.key, identity, ex=timeout, nx=True)
        return result is not None

    def release(self, input_identity):
        """
        根据给定的标识符，尝试释放锁。
        返回 True 表示释放成功；
        返回 False 则表示给定的标识符与锁持有者的标识符并不相同，释放请求被拒绝。
        """
        # 获取锁键储存的标识符
        lock_identity = self.client.get(self.key)
        if lock_identity is None:
            # 如果锁键的标识符为空，那么说明锁已经被释放
            return True
        elif input_identity == lock_identity:
            # 如果给定的标识符与锁键的标识符相同，那么释放这个锁
            self.client.delete(self.key)
            return True
        else:
            # 如果给定的标识符与锁键的标识符并不相同
```

```
        # 那么说明当前客户端不是锁的持有者
        # 拒绝本次释放请求
        return False
```

这个锁实现在绝大部分情况下都能够正常运行， 但它的 `release()` 方法包含了一个非常隐蔽的错误： 在程序使用 `GET` 命令获取锁键的值以后， 直到程序调用 `DEL` 命令删除锁键的这段时间里面， 锁键的值有可能已经发生了变化， 因此程序执行的 `DEL` 命令有可能会导致当前持有者的锁被错误地释放。

举个例子， 表 13-1 就展示了一个锁被错误释放的例子： 客户端 A 是锁原来的持有者， 它调用 `release()` 方法尝试释放自己的锁， 但是当客户端 A 执行完 `GET` 命令并确认自己就是锁的持有者之后， 锁键却因为过期而自动被移除了， 紧接着客户端 B 又通过执行 `acquire()` 方法成功取得了锁， 然而客户端 A 并未察觉这一变化， 它以为自己还是锁的持有者， 并调用 `DEL` 命令把属于客户端 B 的锁给释放了。

表 13-1 一个错误地释放锁的例子

时间	客户端 A	客户端 B	服务器
0000	调用 `release()` 方法		
0001	执行 `GET` 命令，获取锁键的值		
0002	检查锁键的值，确认自己就是持有者		
0003			移除过期的锁键
0004		执行 `acquire()` 方法并取得锁	
0005	执行 `DEL` 命令，删除锁键	(在不知情的状况下失去了锁)	

为了正确地实现 `release()` 方法， 我们需要一种机制， 它可以保证如果锁键的值在 `GET` 命令执行之后发生了变化， 那么 `DEL` 命令将不会被执行。 在 Redis 里面， 这种机制被称为乐观锁。

本节接下来的内容将对 Redis 的乐观锁机制进行介绍， 并在之后给出一个使用乐观锁实现的、正确的、具有身份验证功能的锁。

13.5.1 WATCH：对键进行监视

客户端可以通过执行 WATCH 命令， 要求服务器对一个或多个数据库键实施监视， 如果在客户端尝试执行事务之前， 这些键的值发生了变化， 那么服务器将拒绝执行客户端发送的事务， 并向它返回一个空值：

```
WATCH key [key ...]
```

与此相反， 如果所有被监视的键都没有发生任何变化， 那么服务器将会如常地执行客户端发送的事务。

通过同时使用 WATCH 命令和 Redis 事务， 我们可以构建出一种针对被监视键的乐观锁机制， 确保事务只会在被监视键没有发生任何变化的情况下执行， 从而保证事务对被监视键的所有修改都是安全、正确和有效的。

以下代码展示了一个因为乐观锁机制而导致事务执行失败的例子：

```
redis> WATCH user_id_counter
OK

redis> GET user_id_counter                          -- 获取当前最新的用户 ID
"256"

redis> MULTI
OK

redis> SET user::256::email "peter@spamer.com"  -- 尝试使用这个 ID 来储存用户信息
QUEUED

redis> SET user::256::password "topsecret"
QUEUED

redis> INCR user_id_counter                         -- 创建新的用户 ID
QUEUED

redis> EXEC
(nil)                                               -- user_id_counter 键已被修改，事务被拒绝执行
```

表 13-2 展示了这个事务执行失败的具体原因： 因为客户端 A 监视了 user_id_counter 键， 而客户端 B 却在客户端 A 执行事务之前对该键进行了修改， 所以服务器最终拒绝了客户端 A 的事务执行请求。

表 13-2 事务被拒绝执行的完整过程

时间	客户端 A	客户端 B

时间	客户端 A	客户端 B
0000	`WATCH user_id_counter`	
0001	`GET user_id_counter`	
0002	`MULTI`	
0003	`SET user::256::email "peter@spamer.com"`	
0004	`SET user::256::password "topsecret"`	
0005		`SET user_id_counter 10000`
0006	`INCR user_id_counter`	
0007	`EXEC`	

其他信息

属性	值
时间复杂度	O(N)，其中 N 为被监视键的数量。
版本要求	`WATCH` 命令从 Redis 2.2.0 版本开始可用。

13.5.2 UNWATCH：取消对键的监视

客户端可以通过执行 `UNWATCH` 命令， 取消对所有键的监视：

```
UNWATCH
```

服务器在接收到客户端发送的 `UNWATCH` 命令之后， 将不会再对之前 `WATCH` 命令指定的键实施监视， 这些键也不会再对客户端发送的事务造成任何影响。

以下代码展示了一个 `UNWATCH` 命令的执行示例：

```
redis> WATCH "lock_key" "user_id_counter" "msg"
OK

redis> UNWATCH    -- 取消对以上三个键的监视
OK
```

除了显式地执行 UNWATCH 命令之外， 使用 EXEC 命令执行事务和使用 DISCARD 取消事务， 同样会导致客户端撤销对所有键的监视， 这是因为这两个命令在执行之后都会隐式地调用 UNWATCH 命令。

其他信息

属性	值
复杂度	O(N)，其中 N 为被取消监视的键数量。
版本要求	UNWATCH 命令从 Redis 2.2.0 版本开始可用。

13.6 示例：带有身份验证功能的锁

在了解了乐观锁机制的使用方法之后， 现在是时候使用它来实现一个正确的带身份验证功能的锁了。

之前展示的锁实现的问题在于， 在 GET 命令执行之后直到 DEL 命令执行之前的这段时间里， 锁键的值有可能会发生变化， 并出现误删锁键的情况。 为了解决这个问题， 我们需要使用乐观锁去保证 DEL 命令只会在锁键的值没有发生任何变化的情况下执行， 代码清单 13-6 展示了修改之后的锁实现。

代码清单 13-6 带有身份验证功能的锁实现： /pipeline-and-transaction/identity_lock.py

```
from redis import WatchError

class IdentityLock:

    def __init__(self, client, key):
        self.client = client
        self.key = key

    def acquire(self, identity, timeout):
        """
        尝试获取一个带有身份标识符和最大使用时限的锁,
        成功时返回 True ，失败时返回 False 。
        """
        result = self.client.set(self.key, identity, ex=timeout, nx=True)
        return result is not None

    def release(self, input_identity):
        """
```

```
        根据给定的标识符，尝试释放锁。
        返回 True 表示释放成功；
        返回 False 则表示给定的标识符与锁持有者的标识符并不相同，释放请求被拒绝。
        """
        # 开启流水线
        pipe = self.client.pipeline()
        try:
            # 监视锁键
            pipe.watch(self.key)
            # 获取锁键储存的标识符
            lock_identity = pipe.get(self.key)
            if lock_identity is None:
                # 如果锁键的标识符为空，那么说明锁已经被释放
                return True
            elif input_identity == lock_identity:
                # 如果给定的标识符与锁键储存的标识符相同，那么释放这个锁
                # 为了确保 DEL 命令在执行时的安全性，我们需要使用事务去包裹它
                pipe.multi()
                pipe.delete(self.key)
                pipe.execute()
                return True
            else:
                # 如果给定的标识符与锁键储存的标识符并不相同
                # 那么说明当前客户端不是锁的持有者
                # 拒绝本次释放请求
                return False
        except WatchError:
            # 抛出异常说明在 DEL 命令执行之前，已经有其他客户端修改了锁键
            return False
        finally:
            # 取消对键的监视
            pipe.unwatch()
            # 因为 redis-py 在执行 WATCH 命令期间，会将流水线与单个连接进行绑定
            # 所以在执行完 WATCH 命令之后，必须调用 reset() 方法将连接归还给连接池
            pipe.reset()
```

注意， 因为乐观锁的效果只会在同时使用 WATCH 命令以及事务的情况下产生， 所以程序除了需要使用 WATCH 命令对锁键实施监视之外， 还需要将 DEL 命令包裹在事务里面， 这样才能确保 DEL 命令只会在锁键的值没有发生任何变化的情况下执行。

以下代码展示了这个锁实现的使用方法：

```
>>> from redis import Redis
>>> from identity_lock import IdentityLock
```

```
>>> client = Redis(decode_responses=True)
>>> lock = IdentityLock(client, "test-lock")
>>> lock.acquire("peter", 3600)  # 使用 "peter" 作为标识符，获取一个使用时限为 3600 秒的锁
True
>>> lock.release("tom")          # 尝试使用错误的标识符去释放锁，失败
False
>>> lock.release("peter")        # 使用正确的标识符去释放锁，成功
True
```

13.7 示例：带有身份验证功能的计数信号量

本书前面介绍了如何使用锁去获得一项资源的独占使用权， 并给出了几个不同的锁实现。 但是除了独占一项资源之外， 有时候我们也会想要让多个用户共享一项资源， 只要共享者的数量不超过我们限制的数量即可。

举个例子， 假设我们的系统有一项需要大量计算的操作， 如果很多用户同时执行这项操作的话， 那么系统的计算资源将会被耗尽。 为了保证系统的正常运作， 我们可以使用计数信号量来限制在同一时间内能够执行该操作的最大用户数量。

计数信号量（counter semaphore）跟锁非常相似， 它们都可以限制资源的使用权， 但是跟锁只允许单个客户端使用资源的做法不同， 计数信号量允许多个客户端同时使用资源， 只要这些客户端的数量不超过指定的限制即可。

代码清单 13-7 展示了一个带有身份验证功能的计数信号量实现：

- 这个程序会把所有成功取得信号量的客户端的标识符储存在格式为 `semaphore::<name>::holders` 的集合键里面， 至于信号量的最大可获取数量则储存在格式为 `semaphore::<name>::max_size` 的字符串键里面。
- 在使用计数信号量之前， 用户需要先通过 `set_max_size()` 方法设置计数信号量的最大可获取数量。
- `get_max_size()` 方法和 `get_current_size()` 方法可以分别获取计数信号量的最大可获取数量以及当前已获取数量。
- 获取信号量的 `acquire()` 方法是程序的核心： 在获取信号量之前， 程序会先使用两个 GET 命令分别获取信号量的当前已获取数量以及最大可获取数量， 如果信号量的当前已获取数量并未超过最大可获取数量， 那么程序将执行 SADD 命令， 将客户端给定的标识符添加到 `holders` 集合里面。
- 由于 GET 命令执行之后直到 SADD 命令执行之前的这段时间里， 可能会有其他客户端抢先取得了信号量， 并导致可用信号量数量发生变化。 因此程序需要使用 WATCH 命令监视 `holders` 键， 并使用事务包裹 SADD 命令， 以此通过乐观锁机制确保信号量获取操作的安全性。

- 因为 `max_size` 键的值也会影响信号量获取操作的执行结果， 并且这个键的值在 `SADD` 命令执行之前也可能会被其他客户端修改， 所以程序在监视 `holders` 键的同时， 也需要监视 `max_size` 键。
- 当客户端想要释放自己持有的信号量时， 它只需要把自己的标识符传给 `release()` 方法即可： `release()` 方法将调用 `SREM` 命令， 从 `holders` 集合中查找并移除客户端给定的标识符。

代码清单 13-7 计数信号量实现： `/pipeline-and-transaction/semaphore.py`

```
from redis import WatchError

class Semaphore:

    def __init__(self, client, name):
        self.client = client
        self.name = name
        # 用于储存信号量持有者标识符的集合
        self.holder_key = "semaphore::{0}::holders".format(name)
        # 用于记录信号量最大可获取数量的字符串
        self.size_key = "semaphore::{0}::max_size".format(name)

    def set_max_size(self, size):
        """
        设置信号量的最大可获取数量。
        """
        self.client.set(self.size_key, size)

    def get_max_size(self):
        """
        返回信号量的最大可获取数量。
        """
        result = self.client.get(self.size_key)
        if result is None:
            return 0
        else:
            return int(result)

    def get_current_size(self):
        """
        返回目前已被获取的信号量数量。
        """
        return self.client.scard(self.holder_key)

    def acquire(self, identity):
        """
```

```
        尝试获取一个信号量，成功时返回 True ，失败时返回 False 。
        传入的 identity 参数将被用于标识客户端的身份。

        如果调用该方法时信号量的最大可获取数量尚未被设置，那么引发一个 TypeError 。
        """
        # 开启流水线
        pipe = self.client.pipeline()
        try:
            # 监视与信号量有关的两个键
            pipe.watch(self.size_key, self.holder_key)

            # 取得当前已被获取的信号量数量，以及最大可获取的信号量数量
            current_size = pipe.scard(self.holder_key)
            max_size_in_str = pipe.get(self.size_key)
            if max_size_in_str is None:
                raise TypeError("Semaphore max size not set")
            else:
                max_size = int(max_size_in_str)

            if current_size < max_size:
                # 如果还有剩余的信号量可用
                # 那么将给定的标识符放入到持有者集合中
                pipe.multi()
                pipe.sadd(self.holder_key, identity)
                pipe.execute()
                return True
            else:
                # 没有信号量可用，获取失败
                return False
        except WatchError:
            # 获取过程中有其他客户端修改了 size_key 或者 holder_key ，获取失败
            return False
        finally:
            # 取消监视
            pipe.unwatch()
            # 将连接归还给连接池
            pipe.reset()

    def release(self, identity):
        """
        根据给定的标识符，尝试释放当前客户端持有的信号量。
        返回 True 表示释放成功，返回 False 表示由于标识符不匹配而导致释放失败。
        """
        # 尝试从持有者集合中移除给定的标识符
        result = self.client.srem(self.holder_key, identity)
        # 移除成功则说明信号量释放成功
        return result == 1
```

以下代码简单地展示了这个计数信号量的使用方法：

```
>>> from redis import Redis
>>> from semaphore import Semaphore
>>> client = Redis(decode_responses=True)
>>> semaphore = Semaphore(client, "test-semaphore")  # 创建计数信号量
>>> semaphore.set_max_size(3)   # 设置信号量的最大可获取数量
>>> semaphore.acquire("peter")  # 获取信号量
True
>>> semaphore.acquire("jack")
True
>>> semaphore.acquire("tom")
True
>>> semaphore.acquire("mary")     # 可用的三个信号量都已被获取，无法取得更多信号量
False
>>> semaphore.release("jack")     # 释放一个信号量
True
>>> semaphore.get_current_size()  # 目前有两个信号量已被获取
2
>>> semaphore.get_max_size()      # 信号量的最大可获取数量为三个
3
```

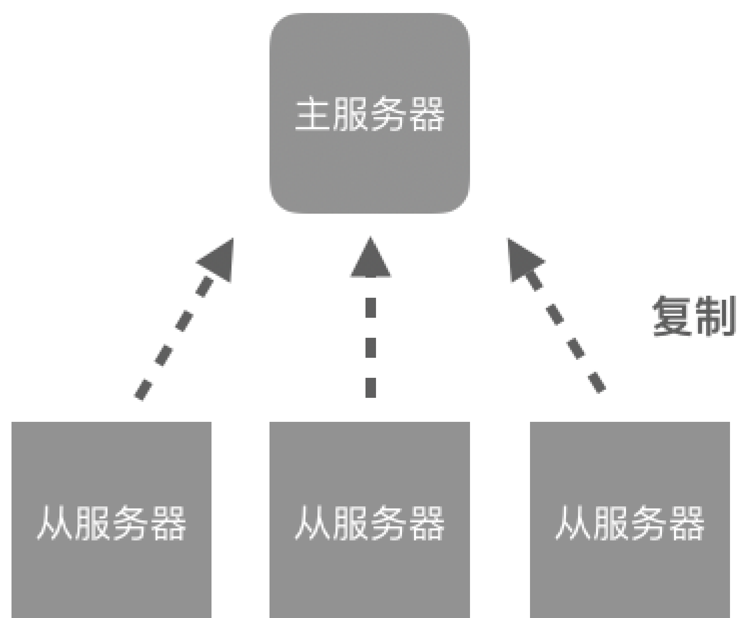
13.8 重点回顾

- 在通常情况下， 程序需要执行的 Redis 命令越多， 它需要进行的网络通讯次数也会越多， 程序的执行速度也会变得越慢。 通过使用 Redis 的流水线特性， 程序可以一次把多个命令发送给 Redis 服务器， 这可以将执行多个命令所需的网络通讯次数从原来的 N 次降低为 1 次， 从而使得程序的执行效率得到显著的提升。
- 通过使用 Redis 的事务特性， 用户可能将多个命令打包成一个命令执行： 当事务成功执行时， 事务中包含的所有命令都会被执行； 相反地， 如果事务执行失败， 那么它包含的所有命令都不会被执行。
- Redis 事务总是具有 ACID 性质中的原子性、一致性和隔离性， 至于是否具有耐久性则取决于 Redis 使用的持久化模式。
- 流水线与事务虽然在概念上有相似之处， 但它们并不相等： 流水线的作用是打包发送多条命令， 而事务的作用则是打包执行多条命令。
- 为了优化事务的执行效率， 很多 Redis 客户端都会把待执行的事务命令缓存在本地， 然后在用户执行 `EXEC` 命令时， 通过流水线一次把所有事务命令发送至 Redis 服务器。
- 通过同时使用 `WATCH` 命令和事务， 用户可以构建起一种乐观锁机制， 这种机制可以确保事务只会在指定键没有发生任何变化的情况下执行。

18. 复制

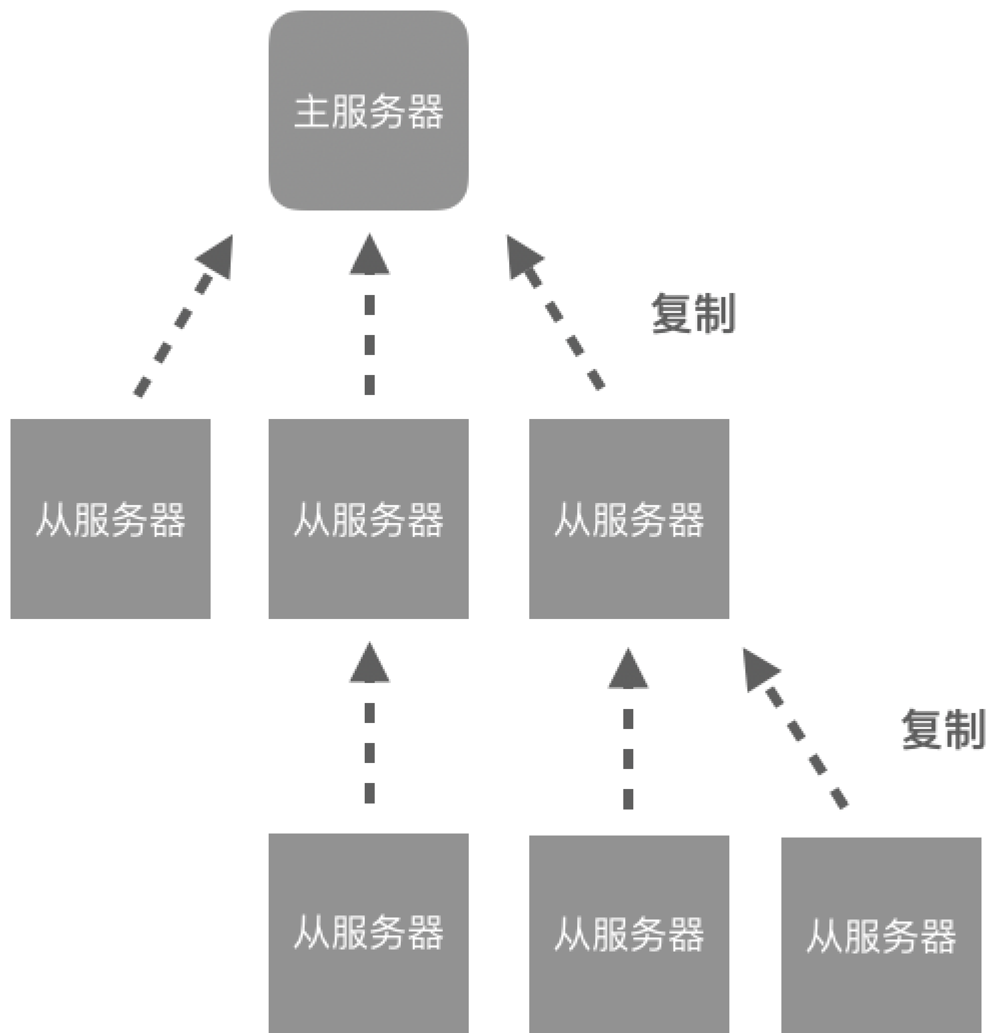
Redis 的复制功能是 Redis 提供的多机功能中最基础的一个， 这个功能是通过主从复制（master–slave replication）模式实现的， 它允许用户为储存着目标数据库的服务器创建出多个拥有相同数据库副本的服务器， 其中储存目标数据库的服务器被称为主服务器（master server）， 而储存数据库副本的服务器则被称为从服务器（slave server，或者简称 replica）， 如图 18–1 所示。

图 18–1 主服务器和从服务器



对于 Redis 来说， 一个主服务器可以拥有任意多个从服务器， 而从服务器本身也可以用作其他服务器的主服务器， 并以此构建出一个树状的服务器结构， 如图 18–2 所示。 需要注意的是， 虽然一个主服务器可以拥有多个从服务器， 但一个从服务器只能拥有一个主服务器。 换句话说， Redis 提供的是单主复制功能， 而不是多主复制功能。

图 18-2 树状服务器结构



在默认情况下， 处于复制模式的主服务器既可以执行写操作也可以执行读操作， 而从服务器则只能执行读操作， 图 18-3 和 18-4 分别展示了 Redis 服务器在无复制和有复制两种状态下的客户端访问模式。

图 18-3 没有启用复制功能的 Redis 服务器可以执行读写操作

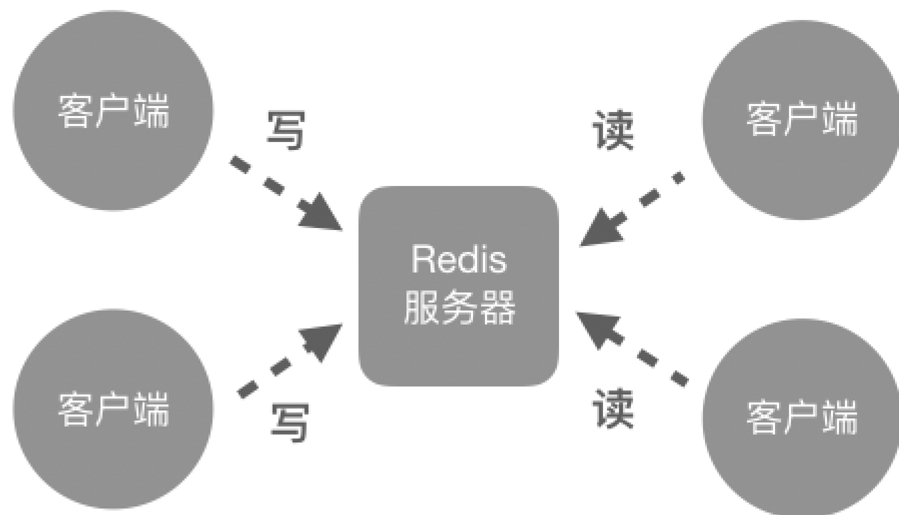
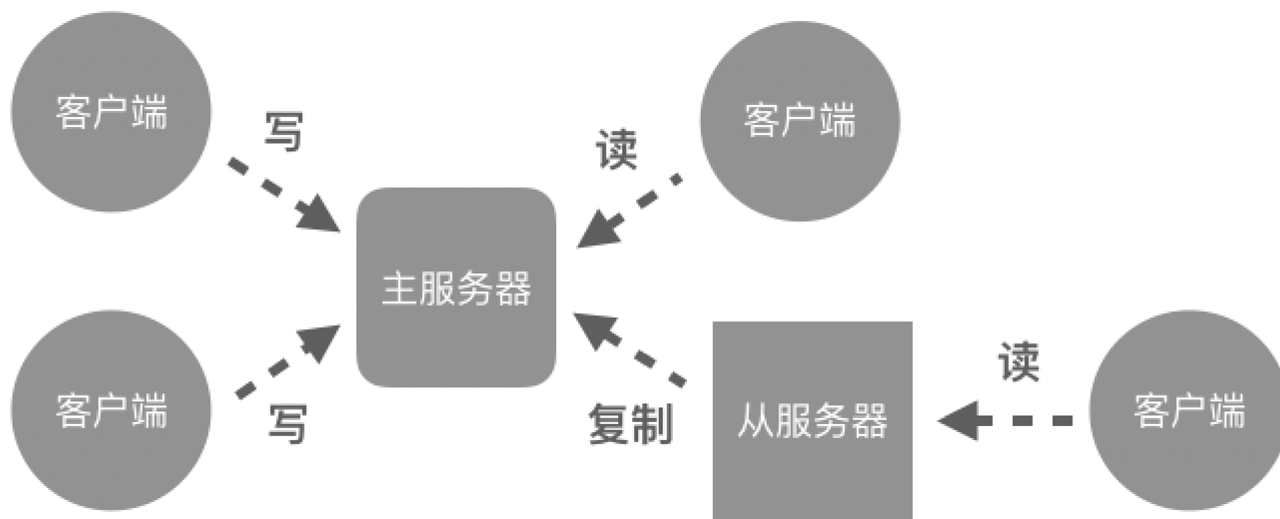


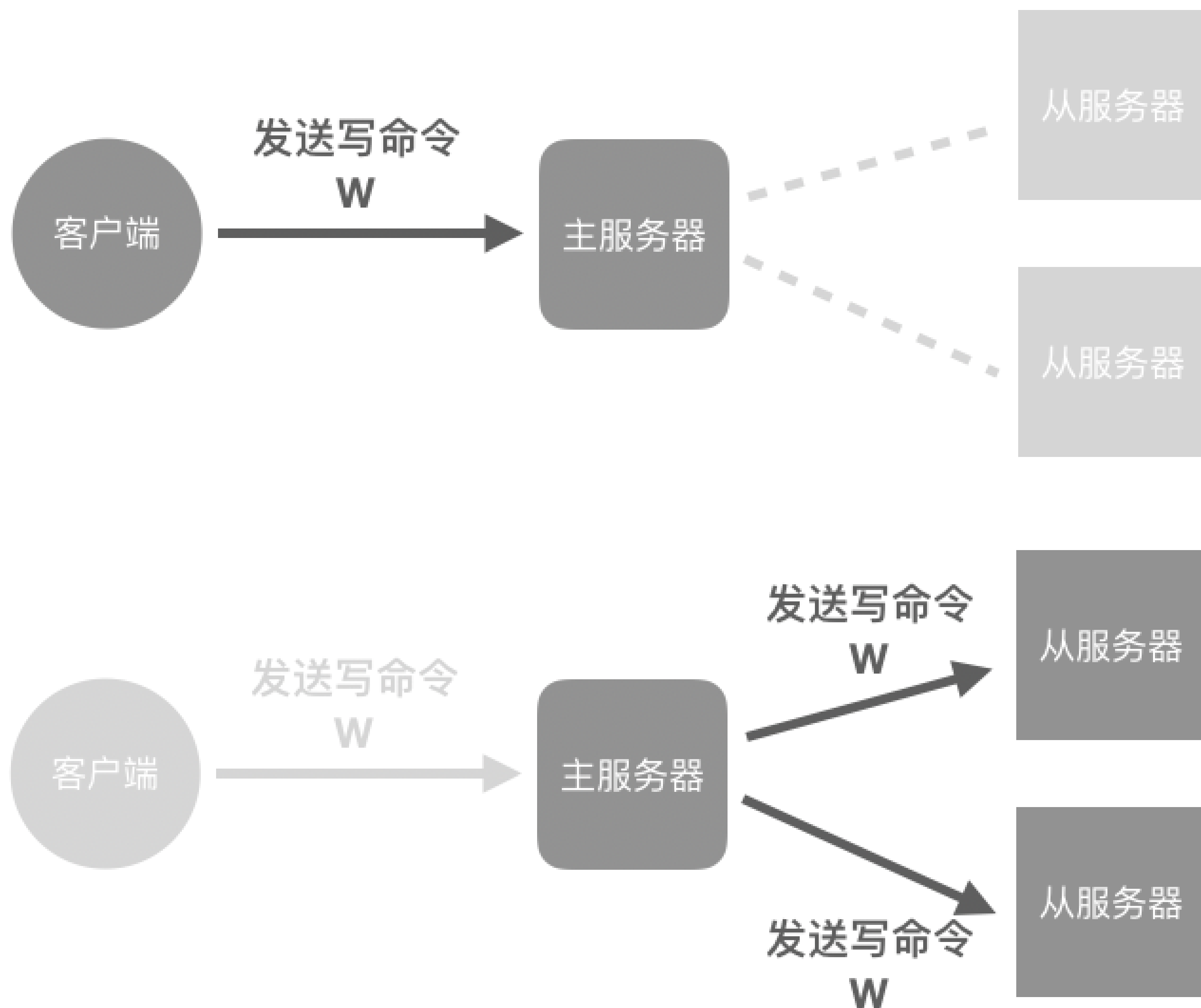
图 18-4 启用了复制功能的主服务器可以执行读写操作，但从服务器默认只能执行读操作



对于开启了复制功能的主从服务器， 主服务器在每次执行写操作之后， 都会与所有从服务器进行数据同步， 以此来将写操作产生的改动反映到各个从服务器之上。 举个例子， 在主服务器执行了客户端发来的写命令 W 之后，

主服务器会将相同的写命令 W 发送至所有从服务器执行， 以此来保持主从服务器之间的数据一致性， 如图 18–5 所示。

图 18-5 主服务器将执行过的写命令发送给从服务器执行



Redis 的复制功能可以从性能、安全性和可用性三个方面提升整个 Redis 系统：

- 首先， 在性能方面， Redis 的复制功能可以给系统的读性能带来线性级别的提升。 从理论上来说， 用户每增加一倍数量的从服务器， 整个系统的读性能就会提升一倍。
- 其次， 通过增加从服务器的数量， 用户可以降低系统在遭遇灾难故障时丢失数据的可能性。 具体来说， 如果用户只有一台服务器储存着目标数据库， 那么当这个服务器遭遇灾难故障时， 目标数据库很有可能会随着服务器故障而丢失。 但如果用户为 Redis 服务器（也即是主服务器）设置了从服务器， 那么即使主服务器遭遇灾难故障， 用户也可以通过从服务器访问数据库。 从服务器的数量越多， 因为主服务器遭遇灾难故障而出现数据库丢失的可能性就越低。
- 最后， 通过同时使用 Redis 的复制功能和 Sentinel 功能， 用户可以为整个 Redis 系统提供高可用特性。 具有这一特性的 Redis 系统在主服务器停机时， 将会自动挑选一个从服务器作为新的主服务器， 以此来继续为客户提供服务， 避免造成整个系统停机。

在本章接下来的内容中， 我们将会学到：

- 如何为主服务器创建从服务器， 从而开启 Redis 的主从复制功能；
- 如何查看服务器在复制中担当的角色以及相关数据；
- Redis 复制功能的实现原理；
- 如何在主服务器不创建 RDB 文件的情况下实现数据同步；
- 如何通过复制功能提升不同类型 Redis 命令的执行效率；

在本章的最后， 我们还会看到 Redis 服务器通过复制传播 Lua 脚本的方法， 至于 Sentinel 相关的内容将在下一章再行介绍。

18.1 REPLICAOF：将服务器设置为从服务器

Note: 复制命令的命名变化

在很长的一段时间里， Redis 一直使用 SLAVEOF 作为复制命令， 但是从 5.0.0 版本开始， Redis 正式将 SLAVEOF 命令改名成了 REPLICAOF 命令并逐渐废弃原来的 SLAVEOF 命令。 因此， 如果你使用的是 Redis 5.0.0 之前的版本， 那么请使用 SLAVEOF 命令代替本章中的 REPLICAOF 命令， 并使用 slaveof 配置选项代替本章中的 replicaof 配置选项。 与此相反， 如果你使用的是 Redis 5.0.0 或之后的版本， 那么就应该使用 REPLICAOF 命令而不是 SLAVEOF 命令， 因为后者可能会在未来的某个时候被正式废弃。

用户可以通过执行 REPLICAOF 命令， 将接收这个命令的 Redis 服务器设置为另一个 Redis 服务器的从服务器：

```
REPLICAOF host port
```

命令的 host 参数用于指定主服务器的地址， 而 port 参数则用于指定主服务器的端口号。 因为 Redis 的复制操作是以异步方式进行的， 所以收到 REPLICAOF 命令的服务器在记录主服务器的地址和端口之后就会向客户端返回 OK ， 至于实际的复制操作则会在后台开始执行。

现在， 假设我们的客户端正连接着服务器 127.0.0.1:12345 ， 如果我们想让这个服务器成为 127.0.0.1:6379 的从服务器， 那么只需要执行以下命令即可：

```
127.0.0.1:12345> REPLICAOF 127.0.0.1 6379
OK
```

在接收到 REPLICAOF 命令之后， 主从服务器将执行数据同步操作： 从服务器原有的数据将被清空， 取而代之的是主服务器传送过来的数据副本。 数据同步完成之后， 主从服务器将拥有相同的数据。

在将 127.0.0.1:12345 设置为 127.0.0.1:6379 的从服务器之后， 如果我们在主服务器 127.0.0.1:6379 执行以下命令， 创建出一个 msg 键：

```
127.0.0.1:6379> SET msg "hello world"
OK

127.0.0.1:6379> GET msg
"hello world"
```

那么这个 msg 键在从服务器 127.0.0.1:12345 上应该也能够访问到：

```
127.0.0.1:12345> GET msg
"hello world"
```

18.1.1 通过配置选项设置从服务器

用户除了可以使用 REPLICAOF 命令将运行中的 Redis 服务器设置为从服务器之外， 还可以通过设置 replicaof 配置选项， 在启动 Redis 服务器的同时将它设置为从服务器：

```
replicaof <host> <port>
```

比如说， 通过执行以下命令， 我们可以在启动服务器 127.0.0.1:10086 的同时， 将它设置为 127.0.0.1:6379 的从服务器：

```
$ redis-server --port 10086 --replicaof 127.0.0.1 6379
```

18.1.2 取消复制

在使用 REPLICAOF 命令或者 replicaof 配置选项将一个服务器设置为从服务器之后， 我们可以通过执行以下命令， 让从服务器停止进行复制， 重新变回主服务器：

```
REPLICAOF no one
```

服务器在停止复制之后不会清空数据库， 而是会继续保留复制产生的所有数据。

比如说， 对于之前设置的从服务器 127.0.0.1:12345 ， 我们可以通过执行以下命令， 让它停止进行复制， 重新变回主服务器：

```
127.0.0.1:12345> REPLICAOF no one
OK
```

命令返回 OK 表示复制已经停止。 因为服务器在停止复制之后仍然会保留复制时产生的数据， 所以我们可以继续访问之前设置的 msg 键：

```
127.0.0.1:12345> GET msg
"hello world"
```

18.1.3 其他信息

属性	值
复杂度	REPLICAOF 命令本身的复杂度为 O(1) ， 但它引起的异步复制操作的复杂度为 O(N) ， 其中 N 为主服务器包含的键值对总数量。 REPLICAOF no one 命令的复杂度为 O(1) 。
版本要求	REPLICAOF 命令从 Redis 5.0.0 版本开始可用。 SLAVEOF 命令从 1.0.0 版本开始可用， 但从 5.0.0 版本开始逐渐废弃。

18.2 ROLE：查看服务器的角色

用户可以通过执行 `ROLE` 命令来查看服务器当前担任的角色：

```
ROLE
```

`ROLE` 命令在主服务器或者从服务器上执行将产生不同的结果， 以下两个小节将分别介绍这两种情况。

18.2.1 主服务器执行 `ROLE` 命令

如果执行 `ROLE` 命令的是主服务器， 那么命令将返回一个由三个元素组成的数组作为结果：

- 数组的第一个元素是字符串 `"master"` ， 它表示这个服务器的角色为主服务器。
- 数组的第二个元素是这个主服务器的复制偏移量（replication offset）， 它是一个整数， 记录了主服务器目前向复制数据流发送的数据数量。
- 数组的第三个元素是一个数组， 它记录了这个主服务器属下的所有从服务器。 这个数组的每个元素都由三个子元素组成， 第一个子元素为从服务器的 IP 地址， 第二个子元素为从服务器的端口号， 而第三个子元素则为从服务器的复制偏移量。 从服务器的复制偏移量记录了从服务器通过复制数据流接收到的复制数据数量， 当从服务器的复制偏移量跟主服务器的复制偏移量保持一致时， 它们的数据就是一致的。

以下是一个主服务器执行 `ROLE` 命令的例子：

```
127.0.0.1:6379> ROLE
1) "master"                -- 这是一个主服务器
2) (integer) 155           -- 它的复制偏移量为 155
3) 1) 1) "127.0.0.1"       -- 第一个从服务器的 IP 地址为 127.0.0.1
      2) "12345"           -- 这个从服务器的端口号为 12345
      3) "155"             -- 它的复制偏移量为 155
   2) 1) "127.0.0.1"       -- 第二个从服务器的 IP 地址为 127.0.0.1
      2) "10086"           -- 端口号为 10086
      3) "155"             -- 复制偏移量为 155
```

18.2.2 从服务器执行 `ROLE` 命令

如果执行 `ROLE` 命令的是从服务器， 那么命令将返回一个由五个元素组成的数组作为结果：

- 数组的第一个元素是字符串 "slave" ， 它表示这个服务器的角色是从服务器。
- 数组的第二个元素和第三个元素记录了这个从服务器正在复制的主服务器的 IP 地址和端口号。
- 数组的第四个元素是主服务器与从服务器当前的连接状态， 这个状态的值及其表示的意思如下：
    - "none" ： 主从服务器尚未建立连接；
    - "connect" ： 主从服务器正在握手；
    - "connecting" ： 主从服务器成功建立了连接；
    - "sync" ： 主从服务器正在进行数据同步；
    - "connected" ： 主从服务器已经进入在线更新状态；
    - "unknown" ： 主从服务器连接状态未知。
- 数组的第五个元素是从服务器当前的复制偏移量。

以下是一个从服务器执行 ROLE 命令的例子：

```
127.0.0.1:12345> ROLE
1) "slave"              -- 这是一个从服务器
2) "127.0.0.1"          -- 主服务器的 IP 地址
3) (integer) 6379       -- 主服务器的端口号
4) "connected"          -- 主从服务器已经进入在线更新状态
5) (integer) 1765       -- 这个从服务器的复制偏移量为 1765
```

18.2.3 其他信息

属性	值
复杂度	O(1)
版本要求	ROLE 命令从 Redis 2.8.12 版本开始可用。

18.3 数据同步

当用户将一个服务器设置为从服务器， 让它去复制另一个服务器的时候， 主从服务器需要通过数据同步机制来让两个服务器的数据库状态保持一致。

这一节将对 Redis 主从服务器的数据同步机制进行介绍， 理解同步机制的运作原理是阅读本章后续内容的基础。

18.3.1 完整同步

当一个 Redis 服务器接收到 `REPLICAOF` 命令， 开始对另一个服务器进行复制的时候， 主从服务器会执行以下操作：

1. 主服务器执行 `BGSAVE` 命令， 生成一个 RDB 文件， 并使用缓冲区储存起在 `BGSAVE` 命令之后执行的所有写命令。
2. 在 RDB 文件创建完毕之后， 主服务器会通过套接字， 将 RDB 文件传送给从服务器。
3. 从服务器在接收完主服务器传送过来的 RDB 文件之后， 就会载入这个 RDB 文件， 从而获得主服务器在执行 `BGSAVE` 命令时的所有数据。
4. 当从服务器完成 RDB 文件载入操作， 并开始上线接受命令请求时， 主服务器就会把之前储存在缓存区里面的所有写命令发送给从服务器执行。

因为主服务器储存的写命令都是在执行 `BGSAVE` 命令之后执行的， 所以当从服务器载入完 RDB 文件， 并执行完主服务器储存在缓冲区里面的所有写命令之后， 主从服务器两者包含的数据库数据将完全相同。

这个通过创建、传送并载入 RDB 文件来达成数据一致的步骤， 我们称之为完整同步操作。 每个从服务器在刚开始进行复制的时候， 都需要与主服务器进行一次完整同步。

> **Note:** 在进行数据同步时重用 RDB 文件
>
> 为了提高数据同步操作的执行效率， 如果主服务器在接收到 `REPLICAOF` 命令之前已经完成了一次 RDB 创建操作， 并且它的数据库在创建 RDB 文件之后没有发生过任何变化， 那么主服务器将直接向从服务器发送已有的 RDB 文件， 以此来避免无谓的 RDB 文件生成操作。
>
> 此外， 如果在主服务器创建 RDB 文件期间， 有多个从服务器向主服务器发送数据同步请求， 那么主服务器将把发送请求的从服务器全部放入到队列里面， 等到 RDB 文件创建完毕之后， 再把它发送给队列里面的所有从服务器， 以此来复用 RDB 文件并避免多余的 RDB 文件创建操作。

18.3.2 在线更新

主从服务器在执行完完整同步操作之后， 它们的数据就达到了一致状态， 但这种一致并不是永久的： 每当主服务器执行了新的写命令之后， 它的数据库就会被改变， 这时主从服务器的数据一致性就会被破坏。

为了让主从服务器的数据一致性可以保持下去， 让它们一直拥有相同的数据， Redis 会对从服务器进行在线更新：

- 每当主服务器执行完一个写命令之后， 它就会将相同的写命令又或者具有相同效果的写命令发送给从服务器执行。
- 因为完整同步之后的主从服务器在执行最新出现的写命令之前， 两者的数据库是完全相同的， 而导致两者数据库出现不一致的正是最新被执行的写命令。 因此从服务器只要接收并执行主服务器发来的写命令， 就可以让自己的数据库重新与主服务器数据库保持一致。

只要从服务器一直与主服务器保持连接， 在线更新操作就会不断进行， 使得从服务器的数据库可以一直被更新， 并与主服务器的数据库保持一致。

Note: 异步更新引起的数据不一致

需要注意的是， 因为在线更新是异步进行的， 所以在主服务器执行完写命令之后， 直到从服务器也执行完相同写命令的这段时间里， 主从服务器的数据库将出现短暂的不一致， 因此要求强一致性的程序可能需要直接读取主服务器而不是读取从服务器。

此外， 因为主服务器可能在执行完写命令并向从服务器发送相同写命令的过程中由于故障而下线， 所以从服务器在主服务器下线之后可能会丢失主服务器已经执行的一部分写命令， 导致从服务器的数据库与下线之前的主服务器数据库处于不一致状态。

因为在线更新的异步本质， Redis 的复制功能是无法杜绝不一致的。 不过本章之后会介绍一种方法， 它可以尽量减少不一致出现的可能性。

18.3.3 部分同步

当故障下线的从服务器重新上线时， 主从服务器的数据通常已经不再一致， 因此它们必须重新进行同步， 让两者的数据库再次回到一致状态。

在 Redis 2.8 版本以前， 重同步操作是通过直接进行完整同步来实现的， 但是， 这种重同步方法在从服务器只是短暂下线的情况下是非常浪费资源的： 主从服务器的数据库在连接断开之前一直都是相同的， 造成数据不一致的原因可能仅仅是因为主服务器比从服务器多执行了几个写命令， 而为了补上这小部分写命令所产生的数据， 却要大费周章地重新进行一次完整同步， 这毫无疑问是非常低效的。

为了解决这个问题， Redis 从 2.8 版本开始使用新的重同步功能去代替原来的重同步功能：

- 当一个 Redis 服务器成为另一个服务器的主服务器时， 它会把每个被执行的写命令都记录到一个特定长度的先进先出队列里面。
- 当断线的从服务器尝试重新连接主服务器的时候， 主服务器将检查从服务器断线期间， 被执行的那些写命令是否仍然保存在队列里面。 如果是的话， 那么主服务器就会直接把从服务器缺失的那些写命令发送给从服务器执行， 从服务器通过执行这些写命令就可以重新与主服务器保持一致， 这样就避免了重新进行完整同步的麻烦。
- 另一方面， 如果从服务器缺失的那些写命令已经不存在于队列当中， 那么主从服务器将进行一次完整同步。

因为新的重同步功能需要使用先进先出队列来记录主服务器执行过的写命令， 所以这个队列的体积越大， 它能够记录的写命令就越多， 从服务器断线之后能够快速地重新回到一致状态的机会也就越大。 Redis 为这个队列设置的默认大小为 1 MB ， 用户也可以根据自己的需要， 通过配置选项 `repl-backlog-size` 来修改这个队列的大小。

18.4 无需硬盘的复制

正如之前所说， 主服务器在进行完整同步的时候， 需要在本地创建 RDB 文件， 然后通过套接字将这个 RDB 文件传送给从服务器。

但是， 如果主服务器所在宿主机器的硬盘负载非常大又或者性能不佳， 创建 RDB 文件引起的大量硬盘写入将对主服务器的性能造成影响， 并导致复制进程变慢。

为了解决这个问题， Redis 从 2.8.18 版本开始引入无需硬盘的复制特性（diskless replication）： 启用了这个特性的主服务器在接收到 `REPLICAOF` 命令时将不会再在本地创建 RDB 文件， 而是会派生出一个子进程， 然后由子进程通过套接字直接将 RDB 文件写入至从服务器。 这样主服务器就可以在不创建 RDB 文件的情况下， 完成与从服务器的数据同步。

要使用无需硬盘的复制特性， 我们只需要将 `repl-diskless-sync` 配置选项的值设置为 `yes` 就可以了：

```
repl-diskless-sync <yes|no>
```

比如以下代码就展示了如何在启动 Redis 服务器的同时， 启用服务器的无需硬盘复制特性：

```
$ redis-server --repl-diskless-sync yes
```

最后要注意的是， 无需硬盘的复制特性只是避免了在主服务器上创建 RDB 文件， 但仍然需要在从服务器上创建 RDB 文件。 Redis 目前还无法在完全不使用硬盘的情况下完成完整数据同步， 但不排除将来会出现这样的功能。

18.5 降低数据不一致的出现几率

本章前面在介绍复制原理时曾经提到过， 因为复制的在线更新操作以异步方式进行， 所以当主从服务器之间的连接不稳定， 又或者从服务器未能收到主服务器发送的更新命令时， 主从服务器就会出现数据不一致的情况。

为了尽可能地降低数据不一致的出现几率， Redis 从 2.8 版本开始引入了两个以 `min-replicas` 开头的配置选项：

```
min-replicas-max-lag <seconds>

min-replicas-to-write <numbers>
```

用户设置了这两个配置选项之后， 主服务器只会在从服务器的数量大于等于 `min-replicas-to-write` 选项的值， 并且这些从服务器与主服务器最后一次成功通讯的间隔不超过 `min-replicas-max-lag` 选项的值时才会执行写命令。

举个例子， 假设我们想要让主服务器只在拥有至少 3 个从服务器， 并且这些从服务器与主服务器最后一次成功通讯的间隔不超过 10 秒钟的情况下才执行写命令， 那么可以使用配置选项：

```
min-replicas-max-lag 10

min-replicas-to-write 3
```

通过使用这两个配置选项， 我们可以让主服务器只在主从服务器连接良好的情况下执行写命令。 因为在线更新的异步性质， `min-replicas-max-lag` 和 `min-replicas-to-write` 并没有办法完全地杜绝数据不一致出现， 但它们可以有效地减少因为主从服务器连接不稳定而导致的数据不一致， 并降低因为没有从服务器可用而导致数据丢失的可能性。

18.6 可写的从服务器

从 Redis 2.6 版本开始， Redis 的从服务器在默认状态下只允许执行读命令。 如果用户尝试对一个只读从服务器执行写命令， 那么从服务器将返回以下错误信息：

```
127.0.0.1:12345> REPLICAOF 127.0.0.1 6379
OK

127.0.0.1:12345> SET msg "hello world"
(error) READONLY You can't write against a read only replica.
```

Redis 之所以将从服务器默认设置为只读服务器， 是为了确保从服务器只能通过与主服务器进行数据同步来得到更新， 从而保证主从服务器之间的数据一致性。

但在某些情况下， 我们可能想要将一些不太重要或者临时性的数据储存在从服务器里面， 又或者不得不在从服务器里面执行一些带有写性质的命令 （比如 ZINTERSTORE 命令，它只能将计算结果储存在数据库里面，不能直接返回计算结果） 。 这时我们可以通过将 replica-read-only 配置选项的值设置为 no 来打开从服务器的写功能：

```
replica-read-only <yes|no>
```

比如说， 如果我们在启动服务器 127.0.0.1:12345 的时候， 将 replica-read-only 配置选项的值设置为 no ：

```
$ redis-server --port 12345 --replica-read-only no
```

那么即使它变成了一个从服务器， 它也能够正常地执行客户端发送的写命令：

```
127.0.0.1:12345> REPLICAOF 127.0.0.1 6379
OK

127.0.0.1:12345> SET msg "hello world again!"
OK

127.0.0.1:12345> GET msg
"hello world again!"
```

Note: 使用可写从服务器的注意事项

在使用可写的从服务器时， 用户需要注意以下几个方面：

- 在主从服务器都可写的情况下， 程序必须将写命令发送到正确的服务器上面， 不能把需要在主服务器执行的写命令发送给从服务器执行， 也不能把需要在从服务器执行的写命令发送给主服务器执行， 否则就会出现数据错误。

- 从服务器执行写命令得到的数据， 可能会被主服务器发送的写命令覆盖。 比如说， 如果从服务器在执行了客户端发送的 `SET msg "hello from client"` 命令之后， 又接收到了主服务器发送的 `SET msg "hello from master"` 命令， 那么客户端写入的 `msg` 键将被主服务器写入的 `msg` 键覆盖。 因为这个原因， 客户端在从服务器上面执行写命令时， 应该尽量避免与主服务器发生键冲突， 换句话说， 用户应该让客户端和主服务器分别对从服务器数据库中不同的键进行写入， 而不要让客户端和主服务器都去写相同的键。
- 当从服务器与主服务器进行完整同步时， 从服务器数据库包含的所有数据都将被清空， 其中包括客户端写入的数据。
- 为了减少内存占用， 降低键冲突发生的可能性， 并确保主从服务器的数据同步操作可以顺利进行， 客户端写入到从服务器的数据应该在使用完毕之后尽快删除。 一个比较简单的方法是在客户端向从服务器写入数据的同时， 为数据设置一个比较短的过期时间， 使得这些数据可以在使用完毕之后自动被删除。

18.7 示例：使用从服务器处理复杂计算操作

Redis 的数据相关命令基本上可以划分为两个种类：

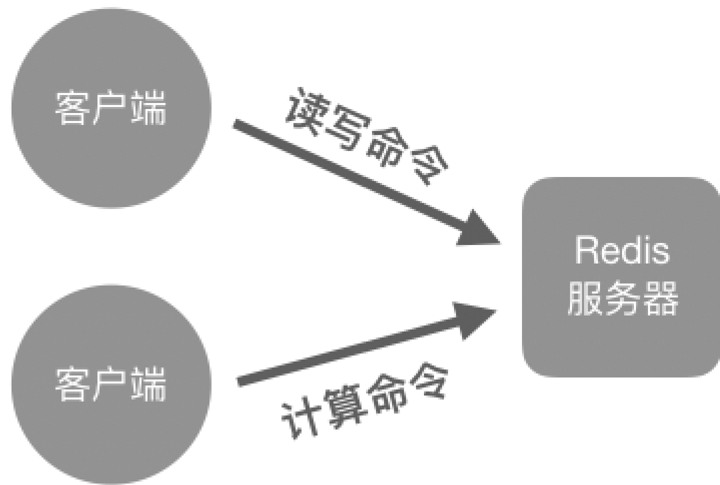
- 第一种是简单的读写操作， 比如 `GET` 命令、 `SET` 命令、 `ZADD` 命令等等， 这种命令只需要对数据库进行简单的读写， 它们的执行速度一般都非常快。
- 第二种是比较复杂的计算操作， 比如 `SUNION` 命令、 `ZINTERSTORE` 命令、 `BITOP` 命令等等， 这种命令需要对数据库中的元素进行聚合计算， 并且随着元素数量的增加， 计算耗费的时间也会增加， 因此这种命令的执行速度一般都比较慢。

从效率的角度考虑， 如果我们在同一个 Redis 服务器里面同时处理以上两种命令， 那么执行第二种命令产生的阻塞时间将导致第一种命令执行时的延迟值显著地增加。

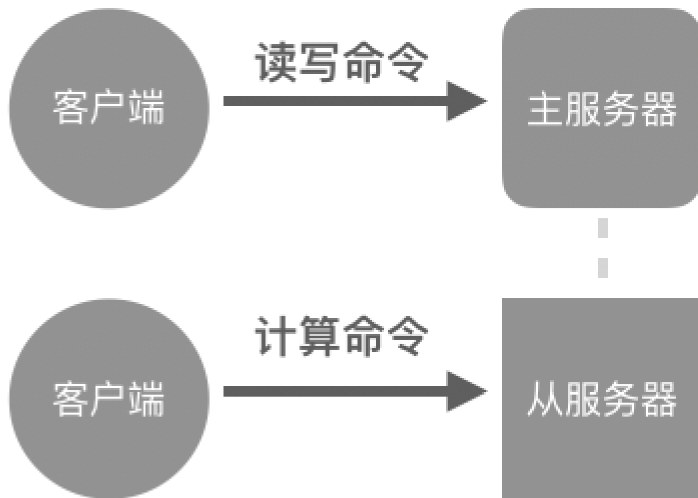
为此， 我们可以通过复制功能创建出当前服务器的从服务器， 然后让主服务器只处理第一种命令， 而第二种命令则交给从服务器处理， 如图 18-6 所示。

图 18-6 使用从服务器执行计算命令

之前：使用服务器处理简单的读写命令和计算命令



之后：使用主服务器处理简单的读写命令，使用从服务器处理计算命令



为了能够在从服务器执行诸如 `ZINTERSTORE` 这样的写命令， 我们需要把从服务器的可写特性打开， 并且在将计算结果储存到从服务器之后， 为它设置一个比较短的过期时间， 使得结果可以自动过期。 又或者在客户端获得结果之后， 由客户端将结果写入到主服务器进行保存。

如果只使用一个从服务器处理第二种命令的速度还不够快， 我们可以继续增加从服务器， 直到从服务器处理第二种命令的速度和延迟值达到我们的要求为止。

18.8 脚本复制

在了解了 Redis 服务器传播普通 Redis 命令的方法之后， 我们接下来要了解的将是 Redis 传播 Lua 脚本的具体方法。

Redis 服务器拥有两种不同的脚本复制模式， 第一种是从 Redis 2.6 版本开始支持的脚本传播模式（whole script replication）， 而另一种则是从 Redis 3.2 版本开始支持的命令传播模式（script effect replication）， 本节接下来将分别介绍这两种模式。

18.8.1 脚本传播模式

处于脚本传播模式的主服务器会将被执行的脚本及其参数（也即是 `EVAL` 命令本身）复制到 AOF 文件以及从服务器里面。 因为带有副作用的函数在不同服务器上运行时可能会产生不同的结果， 从而导致主从服务器不一致， 所以在这一模式下执行的脚本必须是纯函数： 换句话说， 对于相同的数据集， 相同的脚本以及参数必须产生相同的效果。

为了保证脚本的纯函数性质， Redis 对处于脚本传播模式的 Lua 脚本设置了以下限制：

- 脚本不能访问 Lua 的时间模块、内部状态又或者除给定参数之外的其他外部信息。
- 在 Redis 的命令当中， 存在着一部分带有随机性质的命令， 这些命令对于相同的数据集以及相同的参数可能会返回不同的结果。 如果脚本在执行这类带有随机性质的命令之后， 尝试继续执行写命令， 那么 Redis 将拒接执行该命令并返回一个错误。 带有随机性质的 Redis 命令分别为： `SPOP` 、 `SRANDMEMBER` 、 `SCAN` 、 `SSCAN` 、 `ZSCAN` 、 `HSCAN` 、 `RANDOMKEY` 、 `LASTSAVE` 、 `PUBSUB` 、 `TIME` 。
- 当用户在脚本中调用 `SINTER` 、 `SUNION` 、 `SDIFF` 、 `SMEMBERS` 、 `HKEYS` 、 `HVALS` 、 `KEYS` 这七个会以随机顺序返回结果元素的命令时， 为了消除其随机性质， Lua 环境在返回这些命令的结果之前会先对结果中包含的元素进行排序， 以此来确保命令返回的元素总是有序的。
- Redis 会确保每个被执行的脚本都拥有相同的随机数生成器种子， 这意味着如果用户不主动修改这一种子， 那么所有脚本在默认情况下产生的伪随机数列都将是相同的。

脚本传播模式是 Redis 复制脚本时默认使用的模式。 如果用户在执行脚本之前没有修改过相关的配置选项， 那么 Redis 将使用脚本传播模式来复制脚本。

作为例子， 如果我们在启用了脚本传播模式的主服务器执行以下命令：

```
EVAL "redis.call('SET', KEYS[1], 'hello world');redis.call('SET', KEYS[2], 10086);redis.call('SADD', K
```

那么主服务器将向从服务器发送完全相同的 EVAL 命令：

```
EVAL "redis.call('SET', KEYS[1], 'hello world');redis.call('SET', KEYS[2], 10086);redis.call('SADD', K
```

18.8.2 命令传播模式

处于命令传播模式的主服务器会将执行脚本产生的所有写命令用事务包裹起来， 然后将事务复制到 AOF 文件以及从服务器里面。 因为命令传播模式复制的是写命令而不是脚本本身， 所以即使脚本本身包含副作用， 主服务器给所有从服务器复制的写命令仍然是相同的， 因此处于命令传播模式的主服务器能够执行带有副作用的非纯函数脚本。

除了脚本可以不是纯函数之外， 与脚本传播模式相比， 命令传播模式对 Lua 环境还有以下放松：

- 用户可以在执行 RANDOMKEY 、 SRANDMEMBER 等带有随机性质的命令之后继续执行写命令。
- 脚本的伪随机数生成器在每次调用之前， 都会随机地设置种子。 换句话说， 被执行的每个脚本在默认情况下产生的伪随机数列都是不一样的。

除了以上两点之外， 命令传播模式跟脚本传播模式的 Lua 环境限制是一样的， 比如说， 即使在命令传播模式下， 脚本还是无法访问 Lua 的时间模块以及内部状态。

为了开启命令传播模式， 用户在使用脚本执行任何写操作之前， 需要先在脚本里面调用以下函数：

```
redis.replicate_commands()
```

redis.replicate_commands() 只对调用该函数的脚本有效： 在使用命令传播模式执行完当前脚本之后， 服务器将自动切换回默认的脚本传播模式。

作为例子， 如果我们在主服务器执行以下命令：

```
EVAL "redis.replicate_commands();redis.call('SET', KEYS[1], 'hello world');redis.call('SET', KEYS[2],
```

那么主服务器将向从服务器复制以下命令：

```
MULTI
SET "msg" "hello world"
SET "number" "10086"
SADD "fruits" "apple" "banana" "cherry"
EXEC
```

18.8.3 选择性命令传播

为了进一步提升命令传播模式的威力， Redis 允许用户在脚本里面选择性地打开或者关闭命令传播功能， 这一点可以通过在脚本里面调用 `redis.set_repl()` 函数并向它传入以下四个值来完成：

- `redis.REPL_ALL` —— 默认值，将写命令传播至 AOF 文件以及所有从服务器。
- `redis.REPL_AOF` —— 只将写命令传播至 AOF 文件。
- `redis.REPL_SLAVE` —— 只将写命令传播至所有从服务器。
- `redis.REPL_NONE` —— 不传播写命令。

跟 `redis.replicate_commands()` 函数一样， `redis.set_repl()` 函数也只对执行该函数的脚本有效。 用户可以通过这一功能来定制被传播的命令序列， 以此来确保只有真正需要的命令会被传播至 AOF 文件以及从服务器。

代码清单 18-1 储存并集计算结果的脚本

```
-- 打开目录传播模式
-- 以便在执行 SRANDMEMBER 之后继续执行 DEL
redis.replicate_commands()

-- 集合键
local set_a = KEYS[1]
local set_b = KEYS[2]
local result_key = KEYS[3]

-- 随机元素的数量
local count = tonumber(ARGV[1])

-- 计算并集，随机选出指定数量的并集元素，然后删除并集
redis.call('SUNIONSTORE', result_key, set_a, set_b)
local elements = redis.call('SRANDMEMBER', result_key, count)
```

```
redis.call('DEL', result_key)

-- 返回随机选出的并集元素
return elements
```

举个例子， 代码清单 18-1 所示的脚本会将给定的两个集合的并集计算结果储存到一个集合里面， 接着使用 SRANDMEMBER 命令从结果集合里面随机选出指定数量的元素， 然后删除结果集合并向调用者返回被选中的随机元素。

如果我们使用以下方式执行这个脚本：

```
redis-cli --eval union_random.lua set_a set_b union_random , 3
```

那么主服务器将向从服务器复制以下写命令：

```
MULTI
SUNIONSTORE "union_random" "set_a" "set_b"
DEL "union_random"
EXEC
```

但仔细地思考一下就会发现， SUNIONSTORE 命令创建的 union_random 实际上只是一个临时集合， 脚本在取出并集元素之后就会使用 DEL 命令将其删除， 因此主服务器即使不将 SUNIONSTORE 命令和 DEL 命令复制给从服务器， 主从服务器包含的数据也是相同的。

代码清单 18-2 展示了根据上述想法对脚本进行修改之后得出的新脚本， 新旧两个脚本在执行时将得到相同的结果， 但主服务器在执行新脚本时将不会向从服务器复制任何命令。

代码清单 18-2 带有选择性命令传播特性的脚本

```
-- 打开目录传播模式
-- 以便在执行 SRANDMEMBER 之后继续执行 DEL
redis.replicate_commands()

-- 因为这个脚本即使不向从服务器传播 SUNIONSTORE 命令和 DEL 命令
-- 也不会导致主从服务器数据不一致，所以我们可以把命令传播功能关掉
redis.set_repl(redis.REPL_NONE)

-- 集合键
```

```
local set_a = KEYS[1]
local set_b = KEYS[2]
local result_key = KEYS[3]

-- 随机元素的数量
local count = tonumber(ARGV[1])

-- 计算并集，随机选出指定数量的并集元素，然后删除并集
redis.call('SUNIONSTORE', result_key, set_a, set_b)
local elements = redis.call('SRANDMEMBER', result_key, count)
redis.call('DEL', result_key)

-- 返回随机选出的并集元素
return elements
```

需要注意的是， 虽然选择性复制功能非常强大， 但用户如果没有正确地使用这个功能的话， 那么就可能会导致主从服务器的数据出现不一致， 因此用户在使用这个功能的时候必须慎之又慎。

18.8.4 模式的选择

既然存在着两种不同的脚本复制模式， 那么如何选择正确的模式来复制脚本就显得至关重要了。 一般来说， 用户可以根据以下情况来判断应该使用哪种复制模式：

- 如果脚本的体积不大， 执行的计算也不多， 但是却会产生大量命令调用， 那么使用脚本传播模式可以有效地节约网络资源。
- 相反地， 如果一个脚本的体积非常大， 执行的计算非常多， 但是只会产生少量命令调用， 那么使用命令传播模式可以通过重用已有的计算结果来节约计算资源以及网络资源。

举个例子， 假设我们正在开发一个游戏系统， 该系统的其中一项功能就是在节日给符合条件的一批用户增加指定数量的金币。 为此， 我们可能会写出包含以下代码的脚本， 并在执行这个脚本的时候， 通过 KEYS 变量将数量庞大的用户余额键名传递给脚本：

```
local user_balance_keys = KEYS
local increment = ARGV[1]

-- 遍历所有给定的用户余额键，对它们执行 INCRBY 操作
for i = 1, #user_balance_keys do
    redis.call('INCRBY', user_balance_keys[i], increment)
end
```

很明显， 这个脚本将产生相当于传入键数量的 INCRBY 命令： 在用户数量极其庞大的情况下， 使用命令传播模式对这个脚本进行复制将耗费大量网络资源， 但使用脚本传播模式来复制这个脚本则会是一件非常容易的事。

现在， 考虑另一种情况， 假设我们正在开发一个数据聚合脚本， 它包含了一个需要进行大量聚合计算以及大量数据库读写操作的 `aggregate_work()` 函数：

```
local result_key = KEYS[1]

local aggregate_work = function()
    -- ... 省略大量代码
end

redis.call('SET', result_key, aggregate_work())
```

因为执行 `aggregate_work()` 函数需要耗费大量计算资源， 所以如果我们直接复制整个脚本的话， 那么相同的操作就要在每个从服务器上面都执行一遍， 这对于宝贵的计算资源来说无疑是一种巨大的浪费； 相反地， 如果我们使用命令传播模式来复制这个脚本， 那么主服务器在执行完这个脚本之后， 就可以通过 SET 命令直接将函数的计算结果复制给各个从服务器。

18.9 重点回顾

- Redis 的复制功能允许用户为一个服务器创建出多个副本， 其中被复制的服务器为主服务器， 而复制产生的副本则是从服务器。
- Redis 提供的复制功能是通过主从复制模式实现的， 一个主服务器可以有多个从服务器， 但每个从服务器只能有一个主服务器。
- Redis 的复制功能可以从性能、安全性和可用性三个方面提升整个 Redis 系统。
- Redis 主从服务器的数据同步涉及三个操作：1）完整同步；2）在线更新；3）部分同步。
- Redis 的复制功能提供了脚本传播模式和命令传播模式两种脚本复制模式可用， 前者传播的是 Lua 脚本本身， 而后者传播的则是 Redis 命令。 这两种传播模式各有特色， 用户应该根据自己的需求选择合适的模式。

附录 A： Redis 安装方法

本附录将介绍在不同操作系统上安装 Redis 服务器及其内置客户端的具体方法。

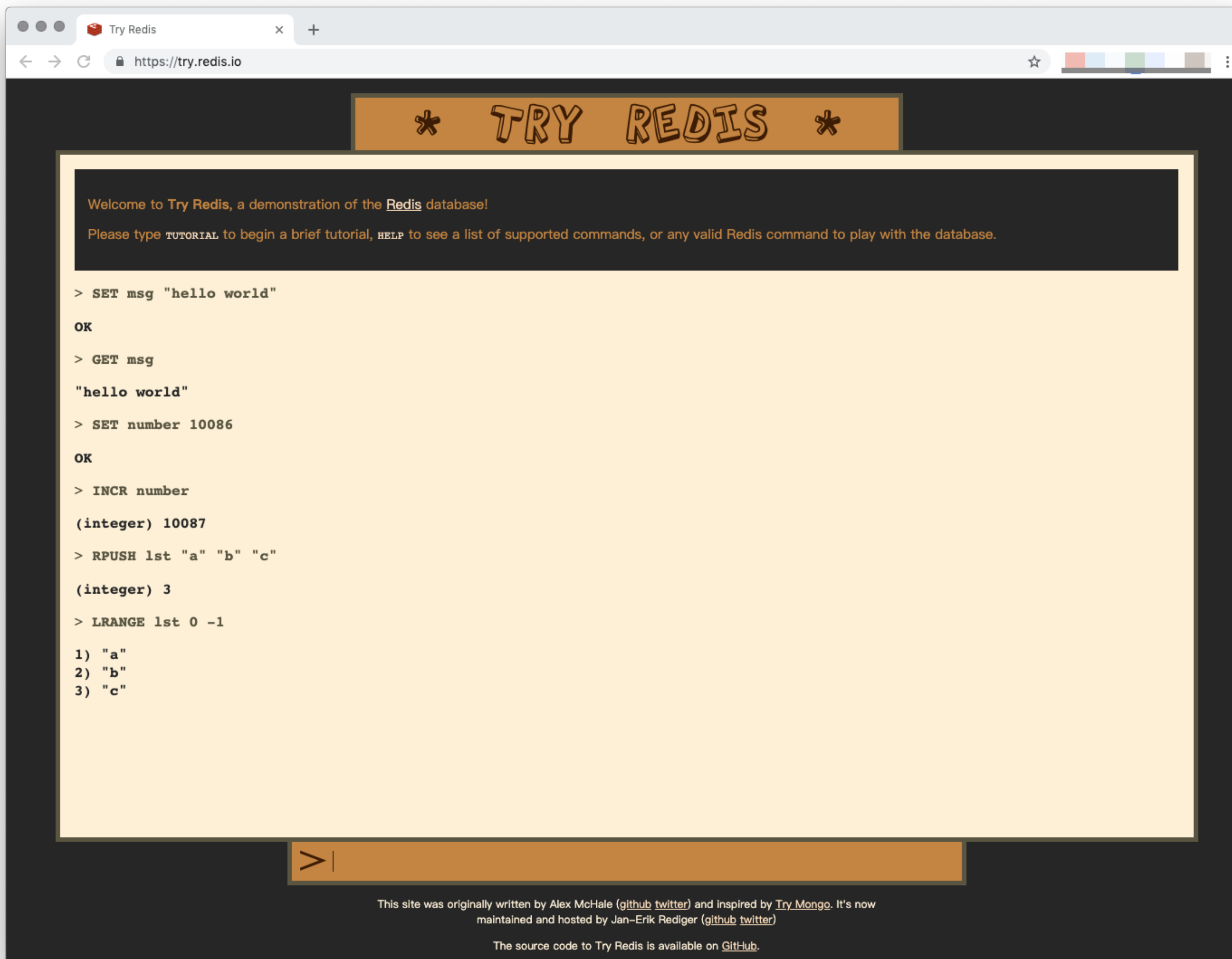
为了保证获得最新版本的 Redis 服务器，我们将通过编译方式安装 Redis 。

A1. 免安装试运行

试用 Redis 最简单的方法就是访问 Try Redis 网站： https://try.redis.io/ 。

Try Redis 可以在线执行大部分 Redis 数据操作命令，并提供了一个简单的 Redis 教程可供阅读。

当你想要快速测试某个 Redis 数据操作命令，但是身边又没有安装了 Redis 的机器可供使用的话，那么不妨尝试一下 Try Redis 。



A2. 在 macOS 上安装

为了在 macOS 上编译并安装 Redis ， 我们需要先安装 make 、GCC 和 Git 等一系列开发工具， 这一点可以通过执行以下命令完成：

```
$ xcode-select --install
```

在安装完开发工具之后， 我们需要通过执行以下命令， 获取最新版本的 Redis 项目源码：

```
$ git clone https://github.com/antirez/redis.git
```

在克隆完项目之后， 我们需要进入项目目录并编译源码：

```
$ cd redis
$ make
```

为了保证编译完成的 Redis 程序运作正常， 我们可以继续执行 Redis 附带的测试程序：

```
$ make test
```

在测试顺利结束之后， 我们就可以进入源码目录， 并通过执行以下命令启动 Redis 服务器：

```
$ cd src/
$ ./redis-server
```

又或者通过执行以下命令启动 Redis 客户端：

```
$ ./redis-cli
```

A3. 在 Linux 上安装

在 Ubuntu 等基于 Debian 的 Linux 系统上， 我们可以通过执行以下命令安装编译 Redis 所需的工具：

```
$ sudo apt install gcc make git
```

接着克隆项目：

```
$ git clone https://github.com/antirez/redis.git
```

然后编译并测试：

```
$ cd redis
$ make
$ make test
```

最后启动服务器：

```
$ cd /src
$ ./redis-server
```

还有客户端：

```
$ ./redis-cli
```

A4. 在 Windows 上安装

因为 Redis 官方并不支持 Windows 系统， 所以我们只能够通过虚拟机或者 Docker 等手段在 Windows 系统上安装 Redis 。

在决定使用 Docker 的情况下， 我们需要先按照以下文档的指示， 在 Windows 系统中下载并安装 Docker ： https://docs.docker.com/docker-for-windows/install/ 。

在 Docker 安装完毕之后， 我们就可以使用 Windows 自带的 CMD 命令行或是 PowerShell ， 通过执行以下命令拉取最新的 Redis 镜像：

```
$ docker pull redis:latest
```

在此之后， 我们可以通过执行以下命令启动 Redis 实例：

```
$ docker run --name docker_redis -d redis
f9442c418a0ae546ab76699e350bed68f56f53f68fe8c0562432e81e03e95809
```

最后， 再执行以下命令， 我们就可以启动 `redis-cli` 客户端并向 Redis 实例发送命令请求了：

```
$ docker exec -it f9442c418a0a redis-cli
127.0.0.1:6379> PING
PONG
```

关于 Redis Docker 镜像的更多信息请参考该镜像的文档： https://hub.docker.com/_/redis/ 。

附录 B ： redis-py 安装方法

本书的绝大部分代码示例都使用 Python 语言编写， 并且使用了 redis-py 客户端来连接服务器并发送命令请求。

如果你正在使用的电脑尚未安装 Python ， 那么请访问 Python 的官方网站并按照文档中介绍的方法下载并安装相应的编程环境： https://www.python.org/downloads/ 。

因为本书的程序都是使用 Python 3 编写的， 所以在下载 Python 安装程序的时候， 请确保你下载的是 Python 3 而不是 Python 2 的安装程序。

在安装 Python 之后， 你应该可以通过输入以下命令来运行该语言的解释器：

```
$ python3
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在安装完 Python 语言环境之后， 我们就可以通过执行以下命令， 使用其附带的 pip 程序来安装 redis-py 客户端了：

```
$ pip install redis
```

在成功安装 redis-py 之后， 我们就能够在 Python 解释器里面载入这个库了：

```
>>> from redis import Redis                # 载入库
>>> client = Redis()                       # 创建客户端实例
>>> client.set("msg", "hello world")       # 执行 SET 命令
True
```

```
>>> client.get("msg")                       # 执行 GET 命令
b'hello world'                              # 未解码的值
```

注意， 正如这里展示的 `GET` 命令执行结果所示， redis-py 默认将返回编码后的值作为结果。 如果我们想让 redis-py 在操作字符串数据的时候自动对其实施解码， 那么只需要在创建客户端实例的时候将 `decode_responses` 可选项的值设置为 `True` 即可， 就像这样：

```
>>> client = Redis(decode_responses=True)
>>> client.get("msg")
'hello world'           # 字符串已解码
```

最后， 如果你有兴趣的话， 还可以通过执行以下命令查看 redis-py 目前支持的 Redis 命令：

```
>>> for i in dir(client):
...     print(i)
...
RESPONSE_CALLBACKS
__class__
# ...
_zaggregate
append
bgrewriteaof
bgsave
bitcount
bitfield
bitop
bitpos
# ...
zscan
zscan_iter
zscore
zunionstore
```

试读已结束

本文档为《Redis使用手册》一书的试读版本， 你已浏览完全部试读内容。

如果你想要浏览本书的完整内容， 那么可以考虑在 RedisGuide.com 付费购买完整版。

感谢你的阅读和支持！