Redis 应用实例

黄健宏　著

人 民 邮 电 出 版 社
北　京

内 容 提 要

本书将从内部组件、外部应用和数据结构 3 个方面为读者介绍 Redis 常见、经典的用法与实例，并且所有实例均附有完整的 Python 代码，方便读者学习和参考。全书分 3 个部分，第一部分讲内部组件，介绍的实例通常用于系统内部，如缓存、锁、计数器、迭代器、速率限制器等，这些都是很多系统中不可或缺的部分；第二部分讲外部应用，介绍的实例都是一些日常常见的、用户可以直接接触到的应用，如直播弹幕、社交关系、排行榜、分页、地理位置等；第三部分讲数据结构，介绍的实例是一些使用 Redis 实现的常见数据结构，如先进先出队列、栈、优先队列和矩阵等。本书希望通过展示常见的 Redis 应用实例来帮助读者了解使用 Redis 解决各类问题的方法，并加深读者对 Redis 各项命令及数据结构的认识，使读者真正成为能够使用 Redis 解决各类问题的 Redis 专家。

本书适合对 Redis 有基本了解且想要进一步掌握 Redis 及键值数据库具体应用的技术人群，是理想的 Redis 技术进阶读物。

前　言

近年来，随着 Redis 大热并成为内存数据库的事实标准，大量关于 Redis 的图书也随之涌现。

2023 年，在决定创作并推出全新的 Redis 图书之际，我对市面上已有的 Redis 图书进行了一番调研，发现大多数 Redis 图书关注的是命令、运维、架构、源码分析等方面的内容，而对实际应用只是一笔带过，或者在介绍命令时做锦上添花之用，很少有图书愿意详细地介绍使用 Redis 的应用实例。

然而，对 Redis 的大量使用导致网上关于 Redis 使用方法的各种问题越来越频繁地出现：如何使用 Redis 实现锁？如何使用 Redis 实现消息队列？如何使用 Redis 表示好友关系？如何使用 Redis 存储地理位置数据？随便去哪一个 Redis 社区，都会看到大量类似的问题。

考虑到这一点，我决定编写本书。书中包含 32 个精挑细选的经典 Redis 应用实例，如缓存、锁、计数器、消息队列、自动补全、社交关系、排行榜、先进先出队列等，这些实例无一不是我们日常开发中经常会遇到的，而且往往也是网上咨询最多的。

我希望通过在书中展示常见的 Redis 应用实例来帮助读者了解使用 Redis 解决各类问题的方法，并加深读者对 Redis 各项命令及数据结构的认识，使读者真正成为能够使用 Redis 解决各类问题的 Redis 专家。

内容编排

本书通过大量 Redis 应用实例来展示 Redis 的经典用法与用例，全书分为 3 个部分，共 32 章。

第一部分讲内部组件。这个部分介绍的实例通常用于系统内部，如缓存、锁、计数器、迭代器、速率限制器等，这些都是很多系统中不可或缺的部分。通过学习如何使用 Redis 构建这些组件，并使用它们代替系统原有的低效组件，读者将能够大幅地提升系统的整体性能。

第二部分讲外部应用。这个部分介绍的实例都是一些日常常见的、用户可以直接接触到的应用，如直播弹幕、社交关系、排行榜、分页、地理位置等。通过学习如何使用 Redis 构建这些应用，读者将能够进一步地了解到 Redis 各个数据结构和命令的强大之处，还能够在实例应用已有功能的基础上，按需扩展出自己想要的其他功能。

第三部分讲数据结构。这个部分介绍的实例是一些使用 Redis 实现的常见数据结构，如先进先出队列、栈、优先队列和矩阵等。在需要快速、可靠的内存存储数据结构时，这些数据结构可以作为其他程序的底层数据结构或者基本构件使用。

除少数章引用了其他章的代码或内容之外，本书的大部分章都自成一体、可以独立阅读，读者可以按需阅读自己感兴趣的任意章。

当然，如果读者只是想要学习 Redis 的多种使用方法，并无特别喜好，也可以像阅读普通教程一样，按顺序阅读本书的每一章。本书基于难度和内容详略等因素对各章的顺序做了编排和优化，力求为读者带来流畅的阅读体验。

目标读者

阅读本书需要读者对 Redis 有一定的了解，并且熟悉 Redis 各个命令的基本语法。

因为本书关注的是如何使用 Redis 命令实现各种应用，而不是详细介绍某个或某些 Redis 命令的具体语法，所以刚开始学习 Redis 或者对 Redis 命令的语法并不熟悉的读者需要在阅读本书的过程中自行查找并学习书中提到的命令。相信这种边做边学、学以致用的方式将有助于读者快速、有效地掌握 Redis 命令及其用法，从而成为熟练的 Redis 使用者。

书中所有实例程序均使用 Python 编程语言编写，程序的风格以简单易懂为第一要务，基本上没有用到 Python 的高级特性。任何学过 Python 编程语言的读者都应该能很好地理解书中的代码，而没有学过 Python 编程语言的读者可以把这些朴素的代码看作伪代码，以此来理解程序想要完成的工作。

本书适合任何想要学习 Redis 应用构建方法和使用 Redis 解决实际问题的人，也可以作为 Redis 学习者在具备一定基础知识之后的进阶应用教程。

代码风格说明

正如前文所言，本书展示的程序以简单易懂为第一要务，为了达到这个目的，本书有时候可能会故意把代码写得详细一些。

例如，为了清晰地展现判断语句的判断条件，本书将采用下面这样的具体写法：

```
if bool_value is True:
    pass
```

而不是采用下面的简单写法：

```
if bool_value:
    pass
```

又如，为了让没学过 Python 语言的人也能看懂程序的打开文件操作，本书将采用以下语句：

```
f = open(file, mode)
# do something
f.close()
```

而不是 Python 程序员更常用的 with 语句：

```
with open(file, mode) as f:
    # do something
```

基于上述原因，本书的部分代码对熟练的 Python 使用者来说可能会稍显啰唆，但这是事出有因的，希望读者可以理解。

代码注释

本书在展示 Python 示例代码的时候，将使用标准的#符号来标识 Python 代码中的注释：

```
>>> from random import random  # 导入随机数生成函数
>>>
```

另外，由于本书在展示 Redis 操作时需要用到 Redis 官方客户端 redis-cli，但该客户端并不支持注释语法，因此本书将采用自选的--符号作为注释：

```
redis> PING  -- 向服务器发送一个请求
PONG
```

因为 redis-cli 实际上并不支持这种注释语法，所以读者在把本书展示的 Redis 操作代码复制到 redis-cli 中运行时，请不要复制代码中的注释内容，以免代码在运行时出错。

关注核心原理而非细枝末节

本书聚焦实战，书中展示的各种实例无一不来源于实际的编程问题，但考虑到现实中的程序往往包含大量无关的逻辑和细节，在书中事无巨细地展示它们除模糊焦点和浪费篇幅之外，不会有其他任何好处。

举个例子，一个现实中的消息队列程序可能由数千行代码和数十个 API 组成，但如果仅在讲消息队列的第 14 章中就包含如此大量的代码和 API，那么本书的篇幅将膨胀至让人无法接受的程度。

为了解决这个问题，本书采取了算法书介绍算法时的策略：不罗列和介绍每种应用可能包含的全部 API，而是精挑细选出一组关键、核心的 API，然后用简洁精练的代码在书中实现它们，配上合理的描述和解释，力求让读者尽可能地理解这些核心 API 的实现原理。一旦读者弄懂了这些核心 API，就可以根据自己的需求移植这些应用，并在此基础上举一反三，为应用扩

展出自己想要的任何 API。

软件版本信息

本书展示的所有 Redis 代码均在 Redis 7.4 版本中测试，Python 代码均在 Python 3.12 版本中测试，使用的 redis-py 客户端版本为 5.1.0b7，这是截至本书写作完成时这几种软件的最新版本。

要运行本书展示的代码和程序，读者需要在计算机上安装以上 3 种软件，并确保它们的版本不低于上面提到的版本。具体的软件安装方法请参考它们各自的官方网站。

获取程序源码

本书展示的所有程序的源码都可以在异步社区（www.epubit.com）通过搜索本书书名找到下载链接，读者也可以通过执行以下命令克隆程序源码：

```
git clone git@github.com:huangzworks/redis-usage-collection.git
```

致谢

感谢人民邮电出版社杨海玲编辑在本书创作过程中的专业指导，感谢我的家人的悉心关怀，还要感谢关注本书的读者对本书的期待，本书是在众多人的关心和支持下才得以完成的。

黄健宏

2024 年 7 月于广东清远

目 录

第一部分 内部组件

第三部分 数据结构

第一部分

内部组件

内部组件部分介绍的实例通常用于系统内部，如缓存、锁、计数器、迭代器、速率限制器等，这些都是很多系统中不可或缺的部分。通过学习如何使用Redis构建这些组件，并用其代替系统原有的低效组件，读者将能够大幅地提升系统的整体性能。

第1章

缓存文本数据

因为 Redis 把数据存储在内存中，并且提供了方便的键值对索引方式以及多样化的数据类型，所以使用 Redis 作为缓存是 Redis 最常见的用法。

很多国内外的社交平台都会把核心的时间线/信息流和好友关系/社交关系存储在 Redis 中，这种做法不仅能够加快用户的访问速度，而且系统访问数据的方式也会变得更加简单、直接。不少追求访问速度的视频网站也会把经常访问的静态文件放到 Redis 中，或者把短时间内最火爆的视频文件存储在 Redis 中，从而尽可能地减少用户观看视频时需要等待的载入时间。

本书将介绍多种使用 Redis 缓存数据和文件的方法，其中本章将介绍如何使用字符串键缓存单项数据（如 HTML 文件的内容），还有如何使用 JSON 和哈希键缓存多项数据（如 SQL 表中的行）；第 2 章将介绍缓存图片、视频文件等二进制数据的方法；至于缓存结构更复杂数据的方法（如社交网站的时间线、好友关系等），则会在之后的章节中陆续介绍。

1.1 需求描述

使用 Redis 缓存系统中的文本数据。这些数据可能只有单独一项，也可能会由多个项组成。

1.2 解决方案：使用字符串键缓存单项数据

有些时候，需要缓存的数据可能非常单纯，只有单独一项。例如，在缓存 Web 服务器生成的模板文件时，整个模板就是一个以<HTML>...</HTML>标签包围的字符串。在这种情况下，缓存程序只需要使用单个 Redis 字符串键就足以缓存整个模板。

具体来说，可以使用 SET 命令，将指定的名字和被缓存的内容关联起来：

```
SET name content
```

如果需要，还可以在设置缓存的同时，为其设置过期时间以便让缓存实现自动更新：

```
SET name content EX ttl
```

至于获取缓存内容的工作则通过 GET 命令来完成：

```
GET name
```

1.3　实现代码：使用字符串键缓存单项数据

代码清单 1-1 展示了基于 1.2 节所述解决方案实现的缓存程序。

代码清单 1-1　基本的缓存程序 cache.py

```
class Cache:

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

    def set(self, name, content, ttl=None):
        """
        为指定名字的缓存设置内容。
        可选的 ttl 参数用于设置缓存的存活时间。
        """
        if ttl is None:
            self.client.set(name, content)
        else:
            self.client.set(name, content, ex=ttl)

    def get(self, name):
        """
        尝试获取指定名字的缓存内容，若缓存不存在则返回 None。
        """
        return self.client.get(name)
```

提示：提高过期时间精度

如果需要更精确的过期时间，那么可以把缓存程序中过期时间的精度参数从代表秒的 ex 修改为代表毫秒的 px。

作为例子，下面这段代码展示了这个缓存程序的基本用法。

```
from redis import Redis
from cache import Cache
```

```
ID = 10086
TTL = 60
REQUEST_TIMES = 5

client = Redis(decode_responses=True)
cache = Cache(client)

def get_content_from_db(id):
    # 模拟从数据库中取出数据
    return "Hello World!"

def get_post_from_template(id):
    # 模拟使用数据库数据生成模板
    content = get_content_from_db(id)
    return "<html><p>{}</p></html>".format(content)

for _ in range(REQUEST_TIMES):
    # 尝试直接从缓存中取出模板
    post = cache.get(ID)
    if post is None:
        # 缓存不存在，访问数据库并生成模板
        # 然后把它放入缓存以便之后访问
        post = get_post_from_template(ID)
        cache.set(ID, post, TTL)
        print("Fetch post from database&template.")
    else:
        # 缓存存在，无须访问数据库也无须生成模板
        print("Fetch post from cache.")
```

根据这段程序的执行结果可知，程序只会在第一次请求时访问数据库并生成模板，而后续一分钟内发生的其他请求都是通过访问 Redis 保存的缓存来完成的：

```
$ python3 cache_usage.py
Fetch post from database&template.
Fetch post from cache.
Fetch post from cache.
Fetch post from cache.
Fetch post from cache.
```

1.4 解决方案：使用 JSON/哈希键缓存多项数据

在复杂的系统中，单项数据往往只占少数，更多的是由多个项组成的复杂数据。例如，表 1-1 列出的这组用户信息，就来自 SQL 数据库 `Users` 表中的 3 行，每行由 `id`、`name`、

gender 和 age 4 个属性值组成。

表 1-1　SQL 数据库中的用户信息

id	name	gender	age
10086	Peter	male	56
10087	Jack	male	37
10088	Mary	female	24

可以通过下面两种不同的方法来缓存这类多项数据。

- 使用 JSON 等序列化手段将多项数据打包成单项数据，然后复用之前缓存单项数据的方法来缓存序列化数据。
- 使用 Redis 的哈希、列表等存储多项数据的数据结构来缓存数据。

接下来介绍这两种方法。

1.5　实现代码：使用 JSON/哈希键缓存多项数据

代码清单 1-2 展示了使用 JSON 缓存多项数据的方法。这个程序复用了代码清单 1-1 中的 Cache 类，它要做的就是在设置缓存之前把 Python 数据编码为 JSON 数据，并在获取缓存之后将 JSON 数据解码为 Python 数据。

代码清单 1-2　使用 JSON 实现的多项数据缓存程序 json_cache.py

```
import json
from cache import Cache

class JsonCache:

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

    def set(self, name, content, ttl=None):
        """
        为指定名字的缓存设置内容。
        可选的 ttl 参数用于设置缓存的存活时间。
        """
        json_data = json.dumps(content)
        self.cache.set(name, json_data, ttl)
```

```
    def get(self, name):
        """
        尝试获取指定名字的缓存内容，若缓存不存在则返回 None。
        """
        json_data = self.cache.get(name)
        if json_data is not None:
            return json.loads(json_data)
```

作为例子，下面这段代码展示了如何使用上述多项数据缓存程序来缓存前面展示的用户信息：

```
>>> from redis import Redis
>>> from json_cache import JsonCache
>>> client = Redis(decode_responses=True)
>>> cache = JsonCache(client)  # 创建缓存对象
>>> data = {"id":10086, "name": "Peter", "gender": "male", "age": 56}
>>> cache.set("User:10086", data)  # 缓存数据
>>> cache.get("User:10086")  # 获取缓存
{'id': 10086, 'name': 'Peter', 'gender': 'male', 'age': 56}
```

除了将多项数据编码为 JSON 然后将其存储在字符串键中，还可以直接将多项数据存储在 Redis 的哈希键中。为此，在设置缓存时需要用到 HSET 命令：

```
HSET name field value [field value] [...]
```

如果用户在设置缓存的同时还指定了缓存的存活时间，那么还需要使用 EXPIRE 命令为缓存设置过期时间，并使用事务或者其他类似措施保证多个命令在执行时的安全性：

```
MULTI
HSET name field value [field value] [...]
EXPIRE name ttl
EXEC
```

与此相对，当要获取被缓存的多项数据时，只需要使用 HGETALL 命令获取所有数据即可：

```
HGETALL name
```

代码清单 1-3 展示了基于上述原理实现的多项数据缓存程序。

代码清单 1-3 使用哈希键实现的多项数据缓存程序 hash_cache.py

```
class HashCache:

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

    def set(self, name, content, ttl=None):
        """
```

```
            为指定名字的缓存设置内容。
            可选的 ttl 参数用于设置缓存的存活时间。
            """
            if ttl is None:
                self.client.hset(name, mapping=content)
            else:
                tx = self.client.pipeline()
                tx.hset(name, mapping=content)
                tx.expire(name, ttl)
                tx.execute()

        def get(self, name):
            """
            尝试获取指定名字的缓存内容，若缓存不存在则返回 None。
            """
            result = self.client.hgetall(name)
            if result != {}:
                return result
```

作为例子，下面这段代码展示了如何使用上述多项数据缓存程序来缓存前面展示的用户信息：

```
>>> from redis import Redis
>>> from hash_cache import HashCache
>>> client = Redis(decode_responses=True)
>>> cache = HashCache(client)
>>> data = {"id":10086, "name": "Peter", "gender": "male", "age": 56}
>>> cache.set("User:10086", data)  # 缓存数据
>>> cache.get("User:10086")  # 获取缓存
{'id': '10086', 'name': 'Peter', 'gender': 'male', 'age': '56'}
```

可以看到，这个程序的效果跟之前使用 JSON 实现的缓存程序的效果完全一致。

提示：缩短键名以节约内存

在使用 Redis 缓存多项数据的时候，不仅需要缓存数据本身（值），还需要缓存数据的属性/字段（键）。当数据的数量巨大时，缓存属性的内存开销也会相当巨大。

为此，缓存程序可以通过适当缩短属性名来尽可能地减少内存开销。例如，把上面用户信息中的 name 属性缩短为 n 属性，age 属性缩短为 a 属性，诸如此类。

还有一种更彻底的方法，就是移除数据的所有属性，将数据本身存储为数组，然后根据各个值在数组中的索引来判断它们对应的属性。例如，可以修改缓存程序，让它把数据 {"id":10086, "name": "Peter", "gender": "male", "age": 56}简化为[10086, "Peter", "male", 56]，然后使用 JSON 数组或者 Redis 列表来存储简化后的数据。

1.6 重点回顾

- 因为 Redis 把数据存储在内存中，并且提供了方便的键值对索引方式以及多样化的数据类型，所以使用 Redis 作为缓存是 Redis 最常见的用法。
- 有些时候，需要缓存的数据可能非常单纯，只有单独一项。在这种情况下，缓存程序只需要使用单个 Redis 字符串键就足以缓存它们。
- 在复杂的系统中，单项数据往往只占少数，更多的是由多个项组成的复杂数据。这时缓存程序可以考虑使用 JSON 等序列化手段，将多项数据打包为单项数据进行缓存，或者直接使用 Redis 的哈希、列表等数据结构进行缓存。

第2章

缓存二进制数据

除了缓存文本数据，Redis 还经常被用于缓存二进制数据，如图片、视频、音频等。本章将介绍使用 Redis 缓存二进制数据的方法。

2.1 需求描述

像缓存文本数据一样，使用 Redis 缓存图片、视频、音频等二进制数据。

2.2 解决方案

跟那些只能存储文本数据或者需要额外支持才能存储二进制数据的数据库相比，Redis 的一个明显优势就是完全支持存储二进制数据：Redis 把所有输入都看作单纯的二进制序列（或者字节串），用户可以在 Redis 中存储任何类型的数据，数据存储的时候是什么样子，取出的时候就是什么样子。

得益于这种数据存储方式，用户不仅可以在 Redis 中存储文本数据，还可以存储任意二进制数据，如视频、音频、图片、压缩文件、加密文件等。不过，为了正确地存储和获取二进制数据，用户在编程语言中使用 Redis 客户端时，必须正确地设置客户端与 Redis 服务器之间的连接方式，让它们以二进制方式而不是字符串方式连接；否则，编程语言所使用的 Redis 客户端可能就会在存储或者获取二进制数据的时候把它们解释为文本数据，从而引发错误。

以 Python 客户端 redis-py 为例，第 1 章中在展示代码示例的时候，一直将 `decode_responses` 参数的值设置为 `True`，这样客户端在获取数据之后就会自动将其转换为相应的 Python 类型（如字符串）：

```
>>> from redis import Redis
>>> client = Redis(decode_responses=True)
>>> client.set("msg", "hi")
```

```
True
>>> client.get("msg")
'hi'    # 字符串值
```

在存储文本数据的时候，这种做法是正确的，但是在存储二进制数据的时候，这种做法却会带来麻烦。为此，需要在初始化 redis-py 客户端实例的时候去掉 `decode_responses` 参数，让它以二进制形式存储和获取数据：

```
>>> from redis import Redis
>>> client = Redis()
>>> client.set("msg", "hi")
True
>>> client.get("msg")
b'hi'    # 二进制值
```

可以看到，客户端这次并没有将`"msg"`键的值解码为字符串`"hi"`，而是维持了数据原本的二进制形式，这正是我们想要的结果。

2.3　实现代码

在弄懂了不同编码设置之间的区别之后，接下来就可以直接复用第 1 章中的缓存程序来缓存二进制文件了。

举个例子，如果现在想要构建一个图片缓存系统，那么需要做的就是以二进制方式读取图片文件，然后将它们缓存到 Redis 中。

代码清单 2-1 展示了一个通用的二进制文件缓存程序：它接受二进制文件的路径作为参数，接着打开并读取该文件，然后将其缓存到 Redis 中，而具体的缓存操作则是通过复用代码清单 1-1 中的 `Cache` 类来实现。

代码清单 2-1　二进制文件缓存程序 binary_cache.py

```
from cache import Cache

class BinaryCache:

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

    def set(self, name, path, ttl=None):
        """
        根据给定的名字和文件路径，缓存指定的二进制文件。
        可选的 ttl 参数用于设置缓存的存活时间。
        """
        # 以二进制方式打开文件，并读取文件
        file = open(path, "rb")
```

```
        data = file.read()
        file.close()
        # 缓存二进制文件
        self.cache.set(name, data, ttl)

    def get(self, name):
        """
        尝试获取指定名字的缓存内容，若缓存不存在则返回 None。
        """
        return self.cache.get(name)
```

作为例子，下面这段代码展示了如何使用这个程序缓存一个图片文件，再从缓存中取出该图片的数据并查看其中的前 10 个字节：

```
>>> from redis import Redis
>>> from binary_cache import BinaryCache
>>> client = Redis()
>>> cache = BinaryCache(client)
>>> cache.set("redis-logo", "./redis-logo.png")  # 缓存文件
True
>>> cache.get("redis-logo")[:10]  # 读取被缓存文件的前 10 个字节
b'\x89PNG\r\n\x1a\n\x00\x00'
```

2.4 重点回顾

- 除了缓存文本数据，Redis 还经常被用于缓存二进制数据，如图片、视频、音频等。
- Redis 把所有输入都看作单纯的二进制序列（或者字节串），用户可以在 Redis 存储任何类型的数据，它们存储的时候是什么样子，取出的时候就是什么样子。
- 为了正确地存储和获取二进制数据，用户在编程语言中使用 Redis 客户端时，必须正确地设置客户端与 Redis 服务器之间的连接方式，让它们以二进制方式而不是字符串方式连接；否则，编程语言所使用的 Redis 客户端可能就会在存储或者获取二进制数据的时候把它们解释为文本数据，从而引发错误。

第3章 锁

锁是计算机系统中经常会用到的一种重要的机制，它可以用来保证特定资源在任何时候最多只能有一个使用者。

Redis 可以通过多种方法实现锁，其中包括带有基本功能的锁、带有自动解锁功能的锁和带有密码保护功能的锁等。本章将介绍前两种锁，而带有密码保护功能的锁将在第 4 章中介绍。

3.1 需求描述

在 Redis 中构建锁，并使用它来保护特定的资源。

3.2 解决方案

每个锁程序至少需要实现以下两个方法。

- 加锁——尝试获得锁的独占权，在任何时候只能有最多一个客户端成功加锁，而除此以外的其他客户端则会失败。
- 解锁——成功加锁的客户端可以通过解锁释放对锁的独占权，使包括它自身在内的所有客户端都能够重新获得加锁的机会。

在 Redis 中实现上述两个操作最基本的方法就是使用字符串数据结构，其中加锁操作可以通过 SET 命令及其 NX 选项来实现：

```
SET key value NX
```

NX 选项的效果保证了给定键只会在没有值（也就是键不存在）的情况下被设置。通过将一个键指定为锁键，并使用客户端尝试对它执行带 NX 选项的 SET 命令，就可以根据命令返回的结果判断加锁是否成功：

- 如果命令成功设置了指定的锁键，那么代表当前客户端成功加锁；
- 如果命令未能成功设置锁键，那么说明锁已被其他客户端占用。

因为带 NX 选项的 SET 命令是原子命令，所以即使有多个客户端同时对同一个锁键执行相同的设置命令，也只会有一个客户端能够成功执行设置操作，因此上述的加锁操作实现是安全的。

另外，当客户端需要解锁的时候，只需要使用 DEL 命令将锁键删除即可：

```
DEL key
```

在锁键被删除之后，它所代表的锁也会重新回到解锁状态。

3.3　实现代码

代码清单 3-1 展示了根据 3.2 节所述解决方案实现的锁程序。

代码清单 3-1　锁程序 lock.py

```
VALUE_OF_LOCK = ""

class Lock:

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

    def acquire(self):
        """
        尝试加锁，成功时返回 True，失败时则返回 False。
        """
        return self.client.set(self.key, VALUE_OF_LOCK, nx=True) is True

    def release(self):
        """
        尝试解锁，成功时返回 True，失败时则返回 False。
        """
        return self.client.delete(self.key) == 1
```

在 acquire() 方法中，程序通过检查 SET 命令的返回值是否为 True 来判断设置是否被成功执行；而在 release() 方法中，程序则通过检查 DEL 命令返回的成功删除键数量是否为 1 来判断锁键是否已被成功删除。

作为例子，下面这段代码展示了上述锁程序的具体用法：

```
>>> from redis import Redis
>>> from lock import Lock
>>> client = Redis(decode_responses=True)
>>> locker1 = Lock(client, "Lock:10086")
```

```
>>> locker1.acquire()  # 加锁
True
>>> locker1.release()  # 解锁
True
```

在 locker1 持有锁期间，如果有其他客户端尝试加锁，那么 acquire()方法将返回 False 表示加锁失败：

```
>>> locker1.acquire()  # locker1 尝试加锁并成功
True
>>> locker2 = Lock(client, "Lock:10086")  # 模拟另一客户端
>>> locker2.acquire()  # locker2 也尝试加锁，但失败
False
```

3.4 扩展方案：带自动解锁功能的锁

3.3 节展示的基本锁实现有一个问题，就是它的解锁操作必须由持有锁的客户端手动执行：如果持有锁的客户端在完成任务之后忘记解锁，或者客户端在执行过程中非正常退出，那么锁可能永远也不会被解锁，而其他等待的客户端也永远无法解锁。

要解决这个问题，可以给锁实现加上自动解锁功能，这样，即使客户端没有手动解锁，Redis 也可以在超过指定时长之后自动删除锁键并解锁。

自动解锁功能可以通过 Redis 的键自动过期特性来实现，为了做到这一点，需要在执行带 NX 选项的 SET 命令时，通过 EX 选项或 PX 选项为锁键设置最大存活时间：

```
SET key value NX EX sec
SET key value NX PX ms
```

在此之后，如果锁键没有被解锁操作手动删除，Redis 会在指定的时限到达之后自动删除带有存活时间的锁键，从而解锁。

代码清单 3-2 展示了基于上述解决方案实现的锁程序。

代码清单 3-2 带自动解锁功能的锁程序 auto_release_lock.py

```
VALUE_OF_LOCK = ""

class AutoReleaseLock:

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

    def acquire(self, timeout, unit="sec"):
        """
        尝试获取一个能够在指定时长之后自动解锁的锁。
        timeout 参数用于设置锁的最大加锁时长。
        可选的 unit 参数则用于设置时长的单位，
        它的值可以是代表秒的 sec 或者代表毫秒的 ms，默认为 sec。
```

```
        """
        if unit == "sec":
            return self.client.set(self.key, VALUE_OF_LOCK, nx=True, ex=timeout) is True
        elif unit == "ms":
            return self.client.set(self.key, VALUE_OF_LOCK, nx=True, px=timeout) is True
        else:
            raise ValueError("Unit must be 'sec' or 'ms'!")

    def release(self):
        """
        尝试解锁，成功时返回 True，失败时则返回 False。
        """
        return self.client.delete(self.key) == 1
```

这个锁实现的 `acquire()` 方法接受 `timeout` 和 `unit` 两个参数，分别用于设置锁的最大加锁时长及其单位，而程序会根据 `unit` 参数的值决定是使用 `SET` 命令的 `EX` 选项还是 `PX` 选项来设置键的存活时间。除此之外，这个锁实现的解锁方法没有发生任何变化，跟之前一样，它只需要执行 `DEL` 命令将锁键删除即可。

作为例子，下面这段代码展示了上述锁程序的具体用法：

```
>>> from redis import Redis
>>> from auto_release_lock import AutoReleaseLock
>>> client = Redis(decode_responses=True)
>>> lock = AutoReleaseLock(client, "Lock:10086")
>>> lock.acquire(5)  # 最多加锁 5 s
True
>>> # 等待 5 s，自动解锁
>>> lock.acquire(5)  # 再次加锁成功
True
>>> lock.release()  # 在 5 s 之内手动解锁
True
```

注意

在使用带有自动解锁功能的锁实现时，锁的最大加锁时长必须超过程序在正常情况下完成任务操作所需的最大时长。

举个例子，如果客户端在成功加锁之后需要消耗 1 s 来完成指定的任务操作，那么你应该将最大加锁时长设置为 30 s 甚至更长，以便让 Redis 在加锁客户端出现真正的意外时自动解锁。

但如果你只是把最大加锁时长设置为 1 s 或者 2 s，那么当程序运行出现延误的时候，可能就会出现“客户端持有的锁已经被自动解锁，但它仍然在使用锁所保护的资源”这类情况，从而导致锁的安全性在实质上被破坏。

换句话说，锁的自动解锁功能就跟程序的异常一样，应该被用作保护措施而不是一般特性。成功加锁的客户端在程序正常运行的情况下还是应该手动解锁，而不是依靠自动解锁。

3.5 重点回顾

- 锁是计算机系统中经常会用到的一种重要的机制，它可以用来保证特定资源在任何时候最多只能有一个使用者。
- 每个锁程序至少会包含加锁和解锁两种操作，在 Redis 中，实现加锁操作最基本的方法就是使用带有 NX 选项的 SET 命令，该命令的性质保证了即使有多个客户端同时对同一个锁键执行相同的设置命令，最多也只会有一个客户端成功进行设置，而其他客户端的设置会失败，这保证了锁实现的安全性。
- 解锁可以通过删除锁对应的锁键来实现，而使用 Redis 的键自动过期特性，锁程序还可以让锁在指定的时长之后自动解锁。
- 锁的自动解锁功能就跟程序的异常一样，应该被用作保护措施而不是一般特性。成功加锁的客户端在程序正常运行的情况下还是应该手动解锁，而不是依靠自动解锁。

第 4 章

带密码保护功能的锁

第 3 章介绍的两个锁实现都假设只有持有锁的客户端会调用 `release()` 方法来解锁，但实际上其他客户端即使没有成功加锁，也可以通过指定相同的锁键并执行 `release()` 方法来解锁。

一般情况下，除非程序出现 bug 或者操作错误，否则没有持有锁的客户端是不应该解锁的。为了避免出现没有持有锁的客户端解锁这种情况，可以给锁加上密码保护功能，使锁只在给定正确密码的情况下才会被解锁。

4.1 需求描述

使用 Redis 实现一个带有密码保护功能的锁，使客户端只在输入正确密码的情况下才能解锁指定的锁。

4.2 解决方案

密码保护功能可以通过以下方式实现：

- 在执行加锁操作的时候，客户端需要提供一个密码，锁实现需要将成功加锁的客户端所提供的密码存储起来；
- 在执行解锁操作的时候，客户端同样需要提供一个密码，锁实现需要验证这个密码与成功加锁的客户端所设置的密码是否相同，如果相同则解锁，反之则不然。

在第 3 章的锁实现中，程序一直将锁键的值设置为空字符串""，但是在具有密码保护功能的锁实现中，程序将通过 `SET` 命令把客户端给定的密码设置为锁键的值。

举个例子，如果客户端提供字符串"top_secret"作为锁键 `Lock:10086` 的密码，那么加锁操作将执行以下命令：

```
SET Lock:10086 "top_secret" NX
```

与此相对，要实现解锁操作，需要执行以下操作。

（1）获取锁键的值（也就是加锁时设置的密码）。

（2）检查锁键的值是否与给定的密码相同，如果相同就执行第 3 步，否则执行第 4 步。

（3）删除锁键并返回 `True` 表示解锁成功。

（4）不对锁键做任何动作，只返回 `False` 表示解锁失败。

为了保证安全性，以上操作必须在事务中完成。

4.3　实现代码

代码清单 4-1 展示了根据 4.2 节所述解决方案实现的锁程序。

代码清单 4-1　带有密码保护功能的锁程序 identity_lock.py

```
from redis import WatchError

class IdentityLock:

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

    def acquire(self, password):
        """
        尝试获取一个带有密码保护功能的锁，
        成功时返回 True，失败时则返回 False。
        password 参数用于设置加锁/解锁密码。
        """
        return self.client.set(self.key, password, nx=True) is True

    def release(self, password):
        """
        根据给定的密码，尝试解锁。
        锁存在并且密码正确时返回 True，
        返回 False 则表示密码不正确或者锁已不存在。
        """
        tx = self.client.pipeline()
        try:
            # 监视锁键以防它发生变化
            tx.watch(self.key)
            # 获取锁键存储的密码
            lock_password = tx.get(self.key)
            # 比对密码
```

```
            if lock_password == password:
                # 情况 1：密码正确，尝试解锁
                tx.multi()
                tx.delete(self.key)
                return tx.execute()[0]==1  # 返回删除结果
            else:
                # 情况 2：密码不正确
                tx.unwatch()
        except WatchError:
            # 尝试解锁时发现键已变化
            pass
        finally:
            # 确保连接正确回归连接池，redis-py 的要求
            tx.reset()
        # 密码不正确或者尝试解锁时失败
        return False
```

正如前面所说，acquire()方法会接受一个密码作为参数并在之后将其设置为键的值。与此对应，release()方法在尝试解锁的时候，会先使用 WATCH 命令监视锁键，接着获取锁键的当前值并与给定密码进行比对，如果比对结果一致，程序就会以事务方式尝试删除锁键以解锁。

作为例子，下面这段代码展示了上述锁程序的具体用法：

```
>>> from redis import Redis
>>> from identity_lock import IdentityLock
>>> client = Redis(decode_responses=True)
>>> lock = IdentityLock(client, "Lock:10086")
>>> lock.acquire("top_secret")  # 尝试加锁并成功
True
>>> lock.release("wrong_password")  # 密码错误，解锁失败
False
>>> lock.release("top_secret")  # 密码正确，解锁成功
True
```

可以看到，release()方法只有在提供了正确密码的情况下，才会实际地执行解锁操作。

你可能会注意到，这个带密码保护功能的锁实现并未包含自动解锁功能：实际上密码保护功能和自动解锁功能是可以同时存在的，只是这样一来实现锁程序的代码就会变得相当复杂，因此本章将不会展示同时带有这两个功能的锁实现，有兴趣的读者可以自行尝试实现它。

4.4 重点回顾

- 第 3 章介绍的两个锁实现都假设只有持有锁的客户端会调用 release()方法来解

锁，但实际上其他客户端即使没有成功加锁，也可以通过指定相同的锁键并执行`release()`方法来解锁。

- 锁的密码保护功能可以通过两个步骤来实现：（1）当客户端尝试加锁时，将成功加锁客户端给定的密码保存在锁键中；（2）当客户端尝试解锁的时候，比对它给定的密码和锁键中保存的密码，只在两个密码相匹配的时候才执行实际的解锁操作。

第5章

自增数字ID

传统SQL数据库可以使用SERIAL等类型创建连续的自动递增数字ID，并将其用作检索数据或者连接数据时的标识符。但由于Redis直接使用键名作为检索数据的标识符，而且每个应用的键名通常都不相同，因此Redis并未内置自动的数字序列ID生成机制。

尽管如此，作为通用的数据标识手段，数字ID对很多使用Redis的应用来说仍是不可或缺的。通过Redis的字符串键或者哈希键可以创建自动递增的数字ID，本章将介绍具体的实现方法。

5.1 需求描述

使用Redis生成连续的自动递增数字ID。

5.2 解决方案：使用字符串键

要使用字符串键创建自动递增数字ID，需要用到Redis的INCR命令：

```
INCR key
```

这个命令可以将存储在键key中的数字值加1。如果给定键不存在，那么它会先将键的值初始化为0，再执行加1操作。通过连续执行INCR命令可以使用字符串键创建连续的自增数字ID：

```
redis> INCR UserID
(integer) 1
redis> INCR UserID
(integer) 2
redis> INCR UserID
(integer) 3
```

除生成数字ID之外，关于数字ID的另一个常见要求是保留特定数字之前的ID，以此来满足系统未来的需求，或者防止用户出现抢注“靓号”等行为。

为了做到这一点，可以使用 SET 命令为给定键设置一个初始值，这样之后对该键执行的 INCR 命令只会产生大于初始值的数字 ID。但需要注意的是，为了避免重复生成相同的 ID，设置初始值的操作必须在给定键生成任何连续 ID 之前进行，也就是在键还没有值的时候进行。

为此，需要使用带 NX 选项的 SET 命令来保证设置只会在键没有值的情况下执行：

```
redis> SET PostID 1000000 NX
OK
redis> INCR PostID
(integer) 1000001
redis> INCR PostID
(integer) 1000002
```

正如上面这段代码所示，通过合理地使用 SET 命令和 INCR 命令，初始 ID 及其之前的数字 ID 将被保留，而后续产生的自增数字 ID 都将大于初始 ID。

5.3　实现代码：使用字符串键实现自增数字 ID 生成器

代码清单 5-1 展示了基于 5.2 节所述解决方案实现的自增数字 ID 生成器。

代码清单 5-1　使用字符串键实现的自增数字 ID 生成器 id_generator.py

```
class IdGenerator:

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

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

    def reserve(self, n):
        """
        保留前 N 个 ID，使之后生成的 ID 都大于 N。
        这个方法只能在执行 produce() 之前执行，否则函数将返回 False 表示执行失败。
        返回 True 则表示保留成功。
        """
        return self.client.set(self.name, n, nx=True) is True
```

作为例子，下面这段代码展示了如何使用这个程序保留前 100 万个 ID，并在之后生成连续的数字 ID：

```
>>> from redis import Redis
>>> from id_generator import IdGenerator
```

```
>>> client = Redis(decode_responses=True)
>>> gen = IdGenerator(client, "UserID")
>>> gen.reserve(1000000)  # 保留前 100 万个 ID
True
>>> gen.produce()  # 生成后续 ID
1000001
>>> gen.produce()
1000002
>>> gen.produce()
1000003
>>> gen.reserve(9999)  # 这个方法无法在生成 ID 之后调用
False
```

5.4 解决方案：使用哈希键

代码清单 5-1 展示的自动递增数字 ID 程序同样可以使用哈希键来实现。正如通过连续执行 INCR 命令可以使用字符串键创建连续的自增数字 ID 一样，通过连续执行 HINCRBY 命令也可以使用哈希键创建连续的自增数字 ID：

```
redis> HINCRBY UserID_Coll PostID 1
(integer) 1
redis> HINCRBY UserID_Coll PostID 1
(integer) 2
```

此外，也可以通过执行 HSETNX 命令，为代表指定 ID 生成器的字段设置默认值，使之后针对该字段生成的 ID 都大于该值，从而达到保护指定数量 ID 的目的：

```
redis> HSETNX UserID_Coll CommentID 1000000
(integer) 1
redis> HINCRBY UserID_Coll CommentID 1
(integer) 1000001
redis> HINCRBY UserID_Coll CommentID 1
(integer) 1000002
```

与使用字符串键实现的自增数字 ID 生成器相比，使用哈希键实现的自增数字 ID 生成器的好处是可以将多个相关的 ID 生成器放到同一个键中进行管理。例如，上面的两段代码就将生成文章 ID 的 PostID 和生成评论 ID 的 CommentID 两个生成器都放到了聚合用户相关 ID 的 UserID_Coll 键中。

5.5 实现代码：使用哈希键实现自增数字 ID 生成器

代码清单 5-2 展示了基于 5.4 节所述解决方案实现的自增数字 ID 生成器。

代码清单 5-2 使用哈希键实现的自增数字 ID 生成器 hash_id_generator.py

```
class HashIdGenerator:
```

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

    def produce(self, name):
        """
        生成并返回下一个 ID。
        """
        return self.client.hincrby(self.key, name, 1)

    def reserve(self, name, number):
        """
        保留前 N 个 ID，使之后生成的 ID 都大于 N。
        这个方法只能在执行 produce() 之前执行，否则函数将返回 False 表示执行失败。
        返回 True 则表示保留成功。
        """
        return self.client.hsetnx(self.key, name, number) == 1
```

作为例子，下面这段代码展示了上述 ID 生成器程序的具体用法：

```
>>> from redis import Redis
>>> from hash_id_generator import HashIdGenerator
>>> client = Redis(decode_responses=True)
>>> gen = HashIdGenerator(client, "UserID_Coll")
>>> gen.reserve("PostID", 1000000)
True
>>> gen.produce("PostID")
1000001
>>> gen.produce("PostID")
1000002
```

提示：数字 ID 的最大值

在 64 位计算机上，使用字符串键或哈希键生成的数字 ID 的最大值为 $2^{63}-1$，也就是 9 223 372 036 854 775 807。

5.6　重点回顾

- 作为通用的数据标识手段，数字 ID 对很多使用 Redis 的应用来说仍是不可或缺的，因此学习如何使用 Redis 生成这类 ID 非常有必要。
- 通过连续执行 INCR 命令可以使用字符串键创建连续的自增数字 ID，而带有 NX 选项的 SET 命令可以在这种情况下用于保留指定数量的前置 ID。
- 通过连续执行 HINCRBY 命令可以使用哈希键创建连续的自增数字 ID，而保留前置 ID 的工作由 HSETNX 命令来完成。

第6章

计数器

计数器是应用最常见的功能之一，它在整个互联网中随处可见。

- 阅读应用会用计数器记录每本书、每篇文章被阅读的次数。
- 应用商店会用计数器记录每个应用被下载的次数和付费购买应用的人数。
- 视频应用、音乐应用会用计数器记录视频和音乐被播放的次数。
- 为了保护用户的财产安全，银行应用可能会在后台用计数器记录每个账户的登录失败次数，并在需要的时候锁定账户以防止密码被暴力破解。

类似的例子还有很多。

6.1 需求描述

使用 Redis 实现计数器，从而对系统或用户的某些操作进行计数。

6.2 解决方案：使用字符串键

在 Redis 中实现计数器最常见的方法是使用字符串键：计数器核心的增加计数和减少计数操作可以分别通过 INCRBY 命令和 DECRBY 命令来完成。此外还需要用到 GET 命令和带有 GET 选项的 SET 命令，前者用于获取计数器的当前值，而后者则用于重置计数器的值并获取重置前的旧值。

作为例子，以下命令序列展示了如何对计数器键 GlobalCounter 执行加法和减法操作，并在需要的时候通过 GET 和 SET 命令获取它的值或者重置它的值：

```
redis> INCRBY GlobalCounter 1     # 增加计数器的值
(integer) 1
redis> INCRBY GlobalCounter 1
(integer) 2
```

```
redis> INCRBY GlobalCounter 100
(integer) 102
redis> DECRBY GlobalCounter 50     # 减少计数器的值
(integer) 52
redis> SET GlobalCounter 0 GET     # 重置计数器并获取旧值
"52"
redis> GET GlobalCounter     # 获取计数器的当前值
"0"
```

6.3　实现代码：使用字符串键实现计数器

代码清单 6-1 展示了基于 6.2 节所述解决方案实现的计数器程序。

代码清单 6-1　使用字符串键实现的计数器程序 counter.py

```
class Counter:

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

    def increase(self, n=1):
        """
        将计数器的值加上指定的数字。
        """
        return self.client.incr(self.key, n)

    def decrease(self, n=1):
        """
        将计数器的值减去指定的数字。
        """
        return self.client.decr(self.key, n)

    def get(self):
        """
        返回计数器的当前值。
        """
        value = self.client.get(self.key)
        return 0 if value is None else int(value)

    def reset(self, n=0):
        """
        将计数器的值重置为参数 n 指定的数字，并返回计数器在重置之前的旧值。
        参数 n 是可选的，若省略则默认将计数器重置为 0。
        """
        value = self.client.set(self.key, n, get=True)
        return 0 if value is None else int(value)
```

作为例子，下面这段代码展示了上述计数器程序的具体用法：

```
>>> from redis import Redis
>>> from counter import Counter
>>> client = Redis(decode_responses=True)
>>> counter = Counter(client, "GlobalCounter")
>>> counter.increase()     # 增加计数器的值
1
>>> counter.increase()
2
>>> counter.increase(100)
102
>>> counter.decrease(50)     # 减少计数器的值
52
>>> counter.reset()     # 重置计数器并获取旧值
52
>>> counter.get()     # 获取计数器的当前值
0
```

这段代码执行的操作跟前面用 Redis 命令执行的操作完全一样。

6.4 解决方案：使用哈希键

除了使用字符串键，计数器还可以使用哈希键来实现。跟使用字符串键相比，使用哈希键实现计数器有两个优点。

- 使用哈希键实现的计数器可以将多个相关计数器放到同一个哈希键中进行管理。举个例子，如果应用需要为每个用户维护多个计数器，如访问计数器、下载计数器、付费计数器等，那么可以考虑将这些计数器都放到同一个哈希键中（如 `User:<id>:Counters` 键）。
- 对于一些使用哈希键存储的文档数据，也可以使用接下来将要介绍的技术，为它们加上计数功能。例如，一个存储文章信息的哈希键 `Post:<id>`可能会包含文章的标题、正文、作者、发布日期等信息，这时比起使用别的字符串键存储文章的浏览量，更好的做法是把浏览量也包含在相同的哈希键中，然后使用哈希键实现计数器的原理，在文章被阅读的同时更新它的浏览量。

使用哈希键实现计数器的核心是 `HINCRBY` 命令。

- 与使用字符串键实现的计数器通过执行 `INCRBY` 命令增加计数器的值类似，使用哈希键实现的计数器也将通过执行 `HINCRBY` 命令增加计数器的值。
- 因为 Redis 并没有为哈希键提供与 `HINCRBY` 对应的 `HDECRBY` 命令，所以哈希键实现的计数器将通过向 `HINCRBY` 命令传入负值的方式减少计数器的值。

此外，由于哈希键无法像带有 GET 选项的 SET 命令那样，在获取字符串键旧值的同时为键设置新值，因此哈希键实现的计数器必须通过用事务包裹 HGET 命令和 HSET 命令的方法来实现相应的 reset() 方法。

6.5　实现代码：使用哈希键实现计数器

代码清单 6-2 展示了根据 6.4 节所述解决方案实现的计数器程序。

代码清单 6-2　使用哈希键实现的计数器程序 hash_counter.py

```
class HashCounter:

    def __init__(self, client, key, name):
        """
        创建一个哈希键计数器对象。
        其中 key 参数用于指定包含多个计数器的哈希键的键名，
        而 name 参数则用于指定具体的计数器在该键中的名字。
        """
        self.client = client
        self.key = key
        self.name = name

    def increase(self, n=1):
        """
        将计数器的值加上指定的数字。
        """
        return self.client.hincrby(self.key, self.name, n)

    def decrease(self, n=1):
        """
        将计数器的值减去指定的数字。
        """
        return self.client.hincrby(self.key, self.name, 0-n)

    def get(self):
        """
        返回计数器的当前值。
        """
        value = self.client.hget(self.key, self.name)
        if value is None:
            return 0
        else:
            return int(value)

    def reset(self, n=0):
        """
```

```
        将计数器的值重置为参数 n 指定的数字，并返回计数器在重置之前的旧值。
        参数 n 是可选的，若省略则默认将计数器重置为 0。
        """
        tx = self.client.pipeline()
        tx.hget(self.key, self.name)  # 获取旧值
        tx.hset(self.key, self.name, n)  # 设置新值
        old_value, _ = tx.execute()
        if old_value is None:
            return 0
        else:
            return int(old_value)
```

作为例子，下面这段代码展示了上述计数器程序的具体用法：

```
>>> from redis import Redis
>>> from hash_counter import HashCounter
>>> client = Redis(decode_responses=True)
>>> counter = HashCounter(client, "User:10086:Counters", "login_counter")
>>> counter.increase()  # 增加计数器的值
1
>>> counter.increase()
2
>>> counter.decrease()  # 减小计数器的值
1
>>> counter.reset()  # 重置计数器并获取旧值
1
>>> counter.get()  # 获取计数器的当前值
0
```

6.6 重点回顾

- 计数器是应用最常见的功能之一，它在整个互联网中随处可见。
- 在 Redis 中实现计数器最常见的方法是使用字符串键：计数器核心的增加计数和减少计数操作可以分别通过 INCRBY 命令和 DECRBY 命令来完成。此外还需要用到 GET 命令和带有 GET 选项的 SET 命令，前者用于获取计数器的当前值，而后者则用于重置计数器的值并获取重置前的旧值。
- 除了使用字符串键，计数器还可以使用哈希键来实现。利用这一技术，程序可以将多个相关联的计数器聚合在一起，或者给存储文档数据的哈希键加入计数功能。

第 7 章

唯一计数器

第 6 章中介绍了简单计数器的实现方式，它可以在用户每次执行特定动作之后更新计数值。对于记录下载次数、浏览次数这类场景，这种计数器已经可以满足需求。但是，这种计数器对重复出现的对象或动作会多次进行计数，因此它并不适用于某些场景。

举个例子，假如现在想要统计的是访问网站的用户数量而不是网站被浏览的次数，那么第 6 章介绍的计数器将无法满足要求：因为它无法判断访问网站的是不同的用户还是重复访问网站多次的同一个用户。

这时需要的就是唯一计数器，这种计数器对每个特定的对象或动作只会计数一次。具体到统计用户数量的场景，这种计数器只会对不同的用户进行计数，而对同一个用户不会重复计数。

7.1 需求描述

使用 Redis 构建唯一计数器，这种计数器对每个特定的对象或动作只会计数一次。

7.2 解决方案：使用集合键

实现唯一计数器的一种方法，也是最直接的方法，就是使用 Redis 集合：通过将每个被计数的对象加到集合中，可以轻而易举地获取集合包含的成员数量，并在需要的时候快速地增删被计数的对象。

举个例子，如果现在想要使用 Redis 集合来统计访问网站的用户数量，那么只需要一直使用 `SADD` 命令将用户添加到某个集合当中即可。由于集合不会保留重复元素，因此只有未被计数过的用户才会被添加到集合中。之后，只需要使用 `SCARD` 命令就可以获取目前访问网站的用户数量，还可以在需要的时候使用 `SREM` 命令从集合中移除指定的用户。

作为例子，以下命令序列展示了如何使用集合键 `VisitCounter` 记录访问网站的用户：

```
redis> SADD VisitCounter "Peter"    -- 将用户添加至集合
(integer) 1
redis> SADD VisitCounter "Jack" "Tom"
(integer) 2
redis> SADD VisitCounter "Tom"    -- 重复对象不会被计数
(integer) 0
```

然后就可以通过 SCARD 命令获取当前访客的数量，或者从集合中移除特定的用户了：

```
redis> SCARD VisitCounter    -- 当前访客数量
(integer) 3
redis> SREM VisitCounter "Peter"    -- 从访客集合中移除指定用户
(integer) 1
redis> SCARD VisitCounter    -- 再次获取当前访客数量
(integer) 2
```

7.3　实现代码：使用集合键实现唯一计数器

代码清单 7-1 展示了基于 7.2 节所述解决方案实现的唯一计数器程序。

代码清单 7-1　使用集合键实现的唯一计数器程序 unique_counter.py

```
class UniqueCounter:

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

    def include(self, item):
        """
        尝试对给定元素进行计数。
        如果该元素之前没有被计数过，那么返回 True，否则返回 False。
        """
        return self.client.sadd(self.key, item) == 1

    def exclude(self, item):
        """
        尝试将被计数的元素移出计数器。
        移除成功返回 True，因元素尚未被计数而导致移除失败则返回 False。
        """
        return self.client.srem(self.key, item) == 1

    def count(self):
        """
        返回计数器当前已计数的元素数量。
        如果计数器为空，那么返回 0。
        """
        return self.client.scard(self.key)
```

作为例子，下面这段代码展示了上述唯一计数器程序的具体用法：

```
>>> from redis import Redis
>>> from unique_counter import UniqueCounter
>>> client = Redis(decode_responses=True)
>>> counter = UniqueCounter(client, "VisitCounter")
>>> counter.include("Peter")  # 对元素进行计数
True
>>> counter.include("Jack")
True
>>> counter.include("Tom")
True
>>> counter.include("Tom")  # 重复元素不会被计数
False
>>> counter.count()  # 查看当前计数结果
3
```

7.4 解决方案：使用 HyperLogLog 键

使用集合键实现的唯一计数器虽然能够满足需求，但它并非完美无缺：使用集合键实现的计数器需要将被计数的所有元素都添加到集合中，当需要计数的元素数量非常多，或者需要进行大量计数的时候，这种计数器将消耗大量内存。

为了解决这个问题，可以修改唯一计数器的程序，使用 HyperLogLog 而不是用集合作为底层结构。HyperLogLog 和集合的相同与不同之处如下。

- HyperLogLog 和集合一样，都可以对元素进行计数。
- HyperLogLog 和集合的不同之处在于，它返回的计数结果并不是准确的集合基数，而是一个与基数八九不离十的估算基数。
- HyperLogLog 的好处是它的内存占用量不会随着被计数元素的增多而增多，无论对多少元素进行计数，HyperLogLog 的内存开销都是固定的，并且是非常少的。

如果应用并不追求完全正确的计数结果，并且不需要准确知道某个元素是否已经被计数，那么完全可以使用 HyperLogLog 代替集合来实现唯一计数器。举个例子，如果你只是想要知道网站大概的访问人数，并且只关心这个计数结果，而不是想要知道某个具体的用户是否一定访问过网站，就可以使用 HyperLogLog 实现的计数器。

举个例子，使用以下命令序列可以将给定的用户添加到用 HyperLogLog 键实现的计数器中，再获取当前的计数结果：

```
redis> PFADD HllVisitCounter "Peter" "Jack" "Tom"  -- 进行计数
(integer) 1
```

```
redis> PFADD HllVisitCounter "Peter"  -- 已计数的对象通常不会重复计数
(integer) 0
redis> PFCOUNT HllVisitCounter  -- 获取当前计数结果
(integer) 3
```

7.5　实现代码：使用 HyperLogLog 键实现唯一计数器

代码清单 7-2 展示了基于 7.4 节所述解决方案实现的唯一计数器程序。

代码清单 7-2　使用 HyperLogLog 键实现的唯一计数器程序 hll_unique_counter.py

```
class HllUniqueCounter:

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

    def include(self, item):
        """
        尝试对给定元素进行计数。
        如果该元素之前没有被计数过，那么返回 True，否则返回 False。
        """
        return self.client.pfadd(self.key, item) == 1

    def exclude(self, item):
        """
        尝试将被计数的元素移出计数器。
        移除成功返回 True，因元素尚未被计数而导致移除失败则返回 False。
        """
        raise NotImplementedError

    def count(self):
        """
        返回计数器当前已计数的元素数量。
        如果计数器为空，那么返回 0。
        """
        return self.client.pfcount(self.key)
```

因为 HyperLogLog 无法撤销对给定元素的计数，所以这个计数器也没有实现相应的 `exclude()` 方法。

作为例子，下面这段代码展示了上述计数器程序的具体用法：

```
>>> from redis import Redis
>>> from hll_unique_counter import HllUniqueCounter
>>> client = Redis(decode_responses=True)
>>> counter = HllUniqueCounter(client, "HllVisitCounter")
>>> counter.include("Peter")    # 计数元素
```

```
True
>>> counter.include("Jack")
True
>>> counter.include("Tom")
True
>>> counter.include("Tom")     # 已计数的元素没有被计数
False
>>> counter.count()     # 获取当前计数结果
3
```

可以看到，这个新的计数器使用起来就跟之前使用集合键实现的计数器一样，并且返回的结果也完全一致。如果继续向这个新的计数器输入更多元素，那么它可能会多计数或少计数其中一些元素，但无论如何，这个计数器的计数结果仍然会处于合理范围之内。

7.6 重点回顾

- 唯一计数器与简单计数器不一样，它对每个特定对象或动作只会计数一次而不是多次。
- 实现唯一计数器的一种方法，也是最直接的方法，就是使用 Redis 集合：通过将每个被计数的对象加到集合中，可以轻而易举地获取集合包含的成员数量，并在需要的时候快速地增删被计数的对象。
- 使用集合键实现的唯一计数器虽然能够满足需求，但它并非完美无缺：使用集合键实现的计数器需要将被计数的所有元素都添加到集合中，当需要计数的元素数量非常多，或者需要进行大量计数的时候，这种计数器将消耗大量内存。
- 一种更高效地实现唯一计数器的方法就是使用 HyperLogLog 键：这种数据结构既可以对元素进行计数，又只需要少量内存。虽然 HyperLogLog 记录的计数并不是完全精确的，但这对很多不需要精确计数结果的应用来说并不是问题。