

A detailed illustration of a man from the side, facing right. He has long, powdered hair and is wearing a white cravat and a dark coat over a red waistcoat. He is smoking a long-stemmed pipe. The style is reminiscent of 18th-century portraiture.

# Redis

## 实战

# Redis

IN ACTION

[美] Josiah L. Carlson 著  
黄健宏 译



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

# Redis 实战

Redis  
IN ACTION

〔美〕Josiah L. Carlson 著  
黄健宏 译

人民邮电出版社  
北京

## 图书在版编目 (C I P) 数据

Redis实战 / (美) 卡尔森 (Carlson, J. L.) 著 ; 黄健宏译. — 北京 : 人民邮电出版社, 2015. 11 (2015. 12重印)  
书名原文: Redis in Action  
ISBN 978-7-115-40284-4

I. ①R… II. ①卡… ②黄… III. ①数据库—基本知识 IV. ①TP311. 13

中国版本图书馆CIP数据核字(2015)第217212号

## 版权声明

Original English language edition, entitled *Redis in Action* by Josiah L. Carlson published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, CT 06830. Copyright © 2013 by Manning Publications Co.

Simplified Chinese-language edition copyright ©2015 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由 Manning Publications Co. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

---

◆ 著 [美] Josiah L. Carlson  
译 黄健宏  
责任编辑 杨海玲  
责任印制 张佳莹 焦志炜  
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号  
邮编 100164 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
北京艺辉印刷有限公司印刷  
◆ 开本: 800×1000 1/16  
印张: 19  
字数: 395 千字 2015 年 11 月第 1 版  
印数: 3501-5000 册 2015 年 12 月北京第 2 次印刷  
著作权合同登记号 图字: 01-2013-4956 号

---

定价: 69.00 元

读者服务热线: (010) 81055410 印装质量热线: (010) 81055316  
反盗版热线: (010) 81055315

# 内容提要

---

本书深入浅出地介绍了 Redis 的 5 种数据类型，并通过多个实用示例展示了 Redis 的用法。除此之外，书中还讲述了 Redis 的优化方法以及扩展方法，是一本对于学习和使用 Redis 来说不可多得的参考书籍。

本书一共由三个部分组成。第一部分对 Redis 进行了介绍，说明了 Redis 的基本使用方法、它拥有的 5 种数据结构以及操作这 5 种数据结构的命令，并讲解了如何使用 Redis 去构建文章聚合网站、cookie、购物车、网页缓存、数据库行缓存等一系列程序。第二部分对 Redis 命令进行了更详细的介绍，并展示了如何使用 Redis 去构建更为复杂的辅助工具和应用程序，并在最后展示了如何使用 Redis 去构建一个简单的社交网站。第三部分对 Redis 用户经常会遇到的一些问题进行了介绍，讲解了降低 Redis 内存占用的方法、扩展 Redis 性能的方法以及使用 Lua 语言进行脚本编程的方法。

本书既涵盖了命令用法等入门主题，也包含了复制、集群、性能扩展等深入主题，所以无论是 Redis 新手还是有一定经验的 Redis 使用者，应该都能从本书中获益。本书面向具有基本数据库概念的读者，读者无需预先了解任何 NoSQL 知识，也不必具备任何 Redis 使用经验。

# 致谢

把这本书献给我亲爱的妻子 See luan, 以及我的宝贝女儿 Mikela。

# 译者序

---

大家好，我是本书的译者黄健宏 ( huangz )。

本书是《Redis in Action》一书的中文翻译版，该书是一本广受欢迎的 Redis 著作，因为书中内容贴近实战而受到了不少赞许，是学习和深入了解 Redis 不可不读的一本书。

承蒙出版社和编辑的厚爱，我有幸担任本书的译者一职。为了不辜负出版社、编辑以及读者们的期待，我把大量心思和时间都投入到了本书的翻译工作当中，希望能够尽我所能地把最好的译作带给大家，而您正在阅读的这本书就是这一努力的成果。

尽管我已经努力地给大家呈现一个高质量的译本，但是因为本人的翻译水平和 Redis 水平都还有很多不足的方面，所以本书肯定也会有许多不尽如人意的地方，如果读者能够联系我并把您认为做得不够好的地方告诉我，我将不胜感激。

我的联系方式可以在 [huangz.me](http://huangz.me) 上面找到，欢迎读者就本书给我提供意见、建议或是问题反馈，非常感谢大家对本书的支持！

## 中文版支持网站和中文源代码

我为本书创建了支持网站 [redisinaction.com](http://redisinaction.com)，读者可以在这个网站上面看到本书的购买链接、试读章样、内容简介、作者介绍、译者介绍等信息，也可以通过网站附带的留言系统进行留言。

为了方便读者学习书中展示的程序源码，我还把这些源码中的注释从英文翻译成了中文，这些带有中文注释的源码也可以在支持网站上面下载到。

## 译者致谢

感谢杨海玲编辑在本书的翻译过程中对我的支持和信任，如果没有她的帮助，我是绝对没办法完成这本书的翻译工作的。

感谢冯春丽细致入微地检查和修正工作，她发现了许多我没有注意到的错误，改正了许多我写下的乱糟糟的句子。

感谢 fleuria 和 Juanito Fatas，他们最先阅读了本书的译文，并给了我很多反馈意见，让我获益良多。

最后，感谢我的家人和朋友，以及各个社交网站上面一直关注本书翻译进度的读者们，他们的支持和鼓励帮助我顺利地完成了这本译作。

## 译者简介

---

黄健宏 (huangz)，男，1990 年出生，目前是程序员、技术图书作者和译者。著有《Redis 设计与实现》，翻译了《Redis 命令参考》《Disque 使用教程》等技术文档。想要了解更多关于黄健宏的信息，请访问他的个人网站 [huangz.me](http://huangz.me)。

# 序

---

Redis 是我在大约 3 年前为了解决一个实际问题而创造出来的：简单来说，当时我在尝试做一件使用硬盘存储关系数据库（on-disk SQL database）无法完成的事情——在一台我能够支付得起的小虚拟机上面处理大量写入负载。

我要解决的问题在概念上并不复杂：多个网站会通过一个小型的 JavaScript 追踪器（tracker）连续不断地向我的服务器发送页面访问记录（page view），而我的服务器需要为每个网站保存一定数量的最新页面访问记录，并通过网页将这些记录实时地展示给用户观看。

在最大负载达到每秒数千条页面记录的情况下，无论我使用什么样的数据库模式（schema），无论我如何进行优化，我所使用的关系数据库都没办法在这个小虚拟机上处理如此大的负载。因为囊中羞涩，我没办法对虚拟机进行升级，并且我觉得应该有更简单的方法来处理一个由推入值组成的列表。最终，我决定自己写一个实验性质的内存数据库原型（prototype），这个数据库使用列表作为基本数据类型，并且能够对列表的两端执行常数时间复杂度的弹出（pop）和推入（push）操作。长话短说吧，这个内存数据库的想法的确奏效了，于是我用 C 语言重写了最初的数据原型，并给它加上了基于子进程实现的持久化特性，Redis 就这样诞生了。

在 Redis 诞生数年之后的今天，这个项目已经发生了显著的变化：我们现在拥有了一个更为健壮的系统，并且随着 Redis 2.6 的发布，开发的重点已经转移到实现集群以及高可用特性上面，Redis 正在进入它的成熟期。在我看来，Redis 生态系统中进步最为明显的一个地方，就是 redis.io 网站以及 Redis Google Group 这些由用户和贡献者组成的社区。数以千计的人通过 GitHub 的问题反馈系统参与到了这个项目里面，他们为 Redis 编写客户端库、提交补丁并帮助其他遇到麻烦的用户。

时至今日，Redis 仍然是一个 BSD 授权的社区项目，它没有那些需要付钱才能使用的闭源插件或者功能增强版。Redis 的参考文档非常详细和准确，在遇到问题时也很容易就可以找到 Redis 开发者或者专家来为你排忧解难。

Redis 始于实用主义——它是一个程序员因为找不到合适的工具来解决手头上的问题而发明的，这是我喜欢《Redis 实战》（*Redis in*

*Action*) 的原因：这本书是为那些想要解决问题的人而写的，它没有乏味地介绍 API，而是通过一系列引人入胜的例子深入地探究了 Redis 的各项特性以及数据类型。

值得一提的是，《Redis 实战》同样来源于 Redis 社区：本书的作者 Josiah 在出版这本书之前，已经在很多不同的方面帮助了数以百计的 Redis 用户——从模式设计到硬件延迟问题，他的建议和贡献在 Redis Group 里随处可见。

本书另一个非常好的地方在于它介绍了服务器运维方面的主题：实际上大部分人在开发应用程序的同时也需要自己部署服务器，而理解服务器运维操作、了解正在使用的硬件和服务器软件的基本限制，有助于写出最大限度地利用硬件和服务器软件的应用程序。

综上所述，《Redis 实战》将是一本把读者带入 Redis 世界、向读者指明正确方向从而避免常见陷阱的书。我认为《Redis 实战》对于 Redis 的生态系统非常有帮助，Redis 的用户应该都会喜欢这本书。

——Salvatore Sanfilippo，“Redis 之父”

# 前言

Chris Testa 是我在圣莫尼卡 Google 分部工作时认识的一个朋友，我从 2010 年 3 月开始和他一起在加利福尼亚州贝弗利山的一间小创业公司工作，Chris 是公司的领头和主管，而我则受聘于他成为了公司研究部门的架构师。

在对某个不相关的问题进行了一个下午的讨论之后，Chris 向我推荐了 Redis，他认为我这个理论计算机科学专业毕业的人应该会对这个数据库感兴趣。在使用 Redis 并按照自己的想法对 Redis 打补丁几个星期之后，我开始参与邮件列表里面的讨论，并向其他人提供建议或者补丁。

随着时间的推移，我将 Redis 广泛应用到了我们公司的各个项目里面：搜索、广告定向引擎、Twitter 分析引擎以及一些将架构中的各个不同部分连接起来的小工具，所有这些项目都要求我学习更多关于 Redis 的知识。每当有其他 Redis 使用者在邮件列表里面提问的时候，我总会情不自禁地给出我的建议（我最喜欢回答的是与职位搜索有关的问题，本书的 7.4 节对此进行了介绍），并因此成为了 Redis 邮件列表里面发言最积极的用户之一。

2011 年 9 月下旬，当时我正在巴黎度蜜月，Manning 出版社的策划编辑 Michael Stephens 给我打来了电话，但因为我的手机只能在美国使用，所以我未能接到 Michael 打来的电话。之后又由于手机固件 bug 的缘故，直到 10 月的第 2 周，我才收到 Michael 发给我的短信。

当我终于收到短信并与 Michael 联系上的时候，我才知道 Manning 出版社打算出版一本《Redis 实战》。在阅读了相关的邮件列表并且向人们咨询应该由谁来写这本书的时候，我的名字出现了。幸运的是，在我回电话的时候，Manning 出版社仍在接受关于《Redis 实战》一书的提案。

在对本书的提案进行了几个星期的讨论和数次修改之后（提案的内容主要来源于我平时在 Redis 邮件列表发表的帖子），Manning 出版社接受了我的提案，然后我开始了本书的写作工作。转眼之间，现在已经是我和 Michael 首次交谈之后的第 17 个月了，《Redis 实战》一书已经基本完成，只剩下一些收尾的工作了。我花费了一整年的所有夜晚和假日，通过编写这本书来帮助其他人理解和使用我认为最有趣的技术——它比我在 20 年前的圣诞节第一次坐在电脑前面以来所知道的大部分技术都要有趣。

虽然自己未能有足够的远见来亲自发明 Redis 是有点儿遗憾，不过至少现在我有机会为它写一本书了。

# 致谢

---

我要感谢我的编辑，Manning 的 Beth Lexleigh，感谢她对我的整个写作过程给予的帮助：你的耐心指导和悉心教诲让我获益良多。

我还要感谢我的开发编辑 Bert Bates：感谢你指出我需要为读者改变自己的写作风格，你对我写作风格的影响遍及全书，极大地改善了本书的可读性。

谢谢你 Salvatore Sanfilippo：没有你，就没有 Redis，更没有这本书，非常感谢你能为本书作序。

谢谢你 Pieter Noordhuis：除了感谢你对 Redis 的贡献之外，我还要感谢你在 RedisConf 2012 大会期间，与我开怀畅饮并听取我关于 Redis 数据结构设计的想法，尽管这些想法未能变为现实，但能够与你交流关于 Redis 内部实现的知识，我仍深感荣幸。

感谢我的技术校对团队（以名字的首字母排序）：James Phillips、Kevin Chang 和 Nicholas Lindgren，多亏了你们的帮助，本书的质量才能更上一层楼。

感谢我的朋友兼同事 Eric Van Dewoestine：感谢你不辞劳苦地为本书编写了 Java 版本的示例代码，这些代码可以在 GitHub 页面找到：<https://github.com/josiahcarlson/redis-in-action>。

感谢包括 Amit Nandi、Bennett Andrews、Bobby Abraham、Brian Forester、Brian Gyss、Brian McNamara、Daniel Sundman、David Miller、Felipe Gutierrez、Filippo Pacini、Gerard O’Sullivan、JC Pretorius、Jonathan Crawley、Joshua White、Leo Cassarani、Mark Wigmans、Richard Clayton、Scott Lyons、Thomas O’Rourke 和 Todd Fiala 在内的参与本书一审、二审、三审以及最终评审的所有审稿人，我已经尽可能地将你们的宝贵意见采纳到本书当中了。

感谢所有在 Manning 的《Redis 实战》作者在线论坛上发表反馈的读者，你们的火眼金睛让错误无处可逃。

我要特别感谢我的妻子 See Luan，她宽宏大量地允许我在一年多的时间里，将数不清的夜晚和假日都花在写作上面，而她却独自忍受着怀孕带来的辛苦与不适；直到最近，在我完成本书最终定稿的这段时间里，她又开始独自照顾我们刚出生的女儿。

最后，感谢我的家人和朋友，谢谢他们一直忍受因为写书而无暇他顾的我。

# 关于本书

---

本书将对 Redis 的使用方法进行说明。Redis 是一个内存数据库（或者说内存数据结构）服务器，最初由 Salvatore Sanfilippo 创建，现在是一个开源软件。本书不要求读者有任何使用 Redis 的经验，不过因为本书的绝大部分示例都使用了 Python 编程语言来与 Redis 进行交互，所以读者需要对 Python 有一定程度的认识才能更好地理解本书的内容。

如果读者不熟悉 Python 的话，那么可以去看看 Python 2.7.x 版本的 Python 语言教程（Python language tutorial），并在本书提到某种 Python 语法结构的时候，查找并阅读相应语法结构的文档。虽然本书展示的 Python 代码在将来可能会被翻译成 Java 代码、JavaScript 代码或者 Ruby 代码，但这些翻译代码的清晰性和简洁性可能会比不上现有的 Python 代码，并且在读者阅读本书的时候，将 Python 代码翻译成其他代码的工作可能尚未完成。

如果读者没有任何使用 Redis 的经验，那么就应该先阅读本书的第 1 章和第 2 章，然后再阅读本书的其他章节（介绍 Redis 安装方法和 Python 安装方法的附录 A 是一个例外，它可以在阅读第 1 章和第 2 章之前阅读）。第 1 章和第 2 章介绍了 Redis 是什么，它能做什么，以及读者可能会想要使用它的理由。之后的第 3 章介绍了 Redis 提供的各种结构，说明了这些结构的作用和总体概念。第 4 章介绍了 Redis 的管理操作，以及实现数据持久化的方法。

如果读者已经有使用 Redis 的经验，那么可以考虑跳过第 1 章和第 3 章——这两章介绍的入门内容都是为那些没有使用过 Redis 的读者准备的。另外，虽然第 2 章也属于入门内容，但即使是有 Redis 使用经验的读者也不应该跳过这一章，因为它展示了本书解决问题时的风格：首先展示问题，然后解决问题，之后回顾问题并改善已有的解决方案，最后，如果读者还想继续深究下去的话，本书还会指出比已有的解决方案更好的新方案。

本书在回顾一个主题的时候，通常会说明第一次讨论这个主题的章节。并非所有主题都要求读者先阅读之前的相关章节，但如果书本确实这么要求的话，那么读者最好还是照书本所说的去做，因为这有助于读者更好地了解整个主题的来龙去脉。

本书很少会给出某个特定问题的最佳解法，更多的是通过展示例子来让读者思考如何去解决某一类问题，并从直觉和非直觉两个方面为这些问题构建解答，所以如果读者在阅读某

个主题的时候，发现了比本书列出的解法更好、更快或者更简单的解法，那将是一件非常棒的事情。

本书每一章对应的源代码都包含了一个测试运行器（test runner），测试运行器提供了那一章定义的绝大部分函数或者方法的使用示例，如果读者在理解某一章的示例时遇到了困难，或者想不明白示例是怎样运作的，那么可以去看看那一章对应的源代码。除此之外，每章对应的源代码还给出了那一章大部分练习的答案。

## 内容编排

本书总共分为 3 个部分：第一部分对 Redis 进行了基本介绍，并展示了一些 Redis 的使用示例；第二部分对 Redis 的多个命令进行了详细的介绍，之后还介绍了 Redis 的管理操作以及使用 Redis 构建更复杂的应用程序的方法；最后，第三部分介绍了如何通过内存优化、水平分片以及 Lua 脚本这 3 种技术来扩展 Redis。

第 1 章对 Redis 进行了基本介绍，列举了 Redis 提供的 5 种数据结构，对比了 Redis 与其他数据库之间的相同之处和不同之处，实现了一个可以对文章进行投票的简单文章聚合网站。

第 2 章介绍了如何使用 Redis 来提升应用程序的性能以及如何使用 Redis 来实现基本的网络分析。不太了解 Redis 的读者应该会从第 2 章开始逐渐明白 Redis 在最近几年变得越来越流行的原因——因为它简单易用，而且性能强劲。

第 3 章基本上是一个命令文档，它陆续介绍了 Redis 的常用命令、基本事务命令、排序命令和过期时间命令，并给出了这些命令的使用示例。

第 4 章介绍了数据持久化、性能测试、故障恢复以及防止数据丢失等概念。这一章前几节介绍的内容都是和 Redis 管理有关的，而之后的 4.4 节和 4.5 节则深入地讨论了 Redis 事务和流水线命令的性能。Redis 新手和中级 Redis 用户都应该阅读 4.4 节和 4.5 节，因为本书在之后的章节里面会再次回顾这两节提到的问题。

第 5 章介绍了将 Redis 用作数据库，并使用它来实现日志、计数器、IP 所属地查找程序和服务配置程序的方法。

第 6 章介绍了一些对于规模日益增长的应用程序非常有用的组件，比如自动补全、加锁、任务队列、消息传递以及文件分发。

第 7 章深入研究了一系列与搜索有关的问题和解决方案，它们可能会改变读者对于数据查询和数据过滤的看法。

第 8 章详细地说明了如何构建一个类似 Twitter 的社交网站，并给出了包括流 API 在内的整个网站后端实现。

第 9 章讨论了扩展 Redis 时会用到的内存优化技术，其中包括结构分片方法以及短结构的使用方法。

第 10 章介绍了对 Redis 进行水平分片和主从复制的方法。当一台服务器不足以满足需求的

时候，这两项特性可以提供更强劲的性能以及更多的可用内存。

第 11 章介绍了如何通过 Lua 脚本编程在服务器端对 Redis 的功能进行扩展，并在某些场景下把 Lua 脚本用作提升性能的方法。

附录 A 介绍了如何在 Linux、OS X 和 Windows 这 3 种不同的平台上安装 Redis、Python 以及 Python 的 Redis 客户端。

附录 B 是一个参考手册，它列出了各种在使用 Redis 时可能会有用的资源，比如本书用到的 Python 语法结构的文档，一些 Redis 使用案例，用于完成各种任务的第三方 Redis 库，诸如此类。

## 代码约定和下载

为了与一般文本区别开来，本书在代码清单和正文中使用 `fixed-width font like this` 这样的字体来显示代码。重要的代码都带有相应的注释，有些代码还会带有编号，以便在之后的内容中对被编号的代码进行说明。

本书列出的所有代码清单的源代码都可以在 Manning 网站下载到：[www.manning.com/RedisinAction](http://www.manning.com/RedisinAction)。如果读者想要查看被翻译成其他语言的源代码，或者想要在线阅览用 Python 语言编写的源代码，那么可以访问这个 GitHub 地址：[github.com/josiahcarlson/redis-in-action](https://github.com/josiahcarlson/redis-in-action)。

## 作者在线论坛

Manning 出版社为本书创建了相应的专属论坛，读者可以通过这个论坛来发表关于本书的评论，询问技术问题，或者寻求作者或其他读者的帮助。[www.manning.com/RedisinAction](http://www.manning.com/RedisinAction) 记载了访问本书专属论坛的方法，部分功能（如发帖）可能需要在注册或者登录之后才能使用。

Manning 出版社承诺为本书提供论坛以供读者和作者使用，但并不对作者的参与度做任何保证：作者对该论坛的所有贡献都是自愿的，并且是无偿的，因此，读者应该尽可能地询问一些有挑战性的问题，从而尽量激发作者的积极性。

在本书正常销售期间，这个作者在线论坛会一直对外开放。

## 关于作者

在大学毕业之后，Josiah Carlson 博士继续在加州大学欧文分校学习理论计算机科学。在学习之余，Josiah 断断续续地做过一些助教工作，偶尔还会承接一些编程方面的工作。在 Josiah 快要研究生毕业的时候，他发现教职方面的工作机会并不多，于是他加入了 Networks in Motion 公司，开始了自己的职业生涯。在 Networks in Motion 公司任职期间，Josiah 负责开发实时 GPS 导航软件，以及交通事故通知系统。

在离开 Networks in Motion 公司之后，Josiah 加入了 Google 公司，之后他又跳槽到了 Adly 公司工作，开始学习和使用 Redis 来构建内容定向广告系统和 Twitter 分析平台。几个月之后，Josiah 加入了 Redis 邮件列表，并在那里回答了数百个关于使用和配置 Redis 的问题。在离开 Adly 公司并成为 ChowNow 公司的首席架构师兼联合创始人之后不久，Josiah 开始创作这本《Redis 实战》。

## 关于封面图画

---

本书封面插图的标题为“一介草民”(A Man of the People), 这幅插图取自 19 世纪法国再版的地区服饰风俗四卷汇编(four-volume compendium of regional dress customs), 作者是 Sylvain Maréchal。书中所有插图都是手工精心绘制并上色的。Maréchal 书中丰富多样的服饰生动地描述了 200 多年前世界上不同城镇和地区的文化差异, 人们相互隔绝, 说着不同的方言和语言, 仅仅从穿着就可以判断他们是住在城镇还是乡间, 知悉他们的工作和身份。

随着时间的流逝, 人们的着装规范已经发生了变化, 曾经丰富多彩的地区多样性也已经逐渐消失不见——现在仅仅通过穿着已经很难区分不同大洲的居民, 更别说是不同城镇和地区了。也许我们已经舍弃了对文化多样性的追求, 转为拥抱更丰富多彩的个人生活以及更多样和快节奏的技术生活去了。

同样地, 在这个难以分辨不同计算机书籍的时代, Manning 出版社希望通过 Maréchal 的作品, 将两个世纪前丰富多彩的地区生活融入本书封面, 以此来赞美计算机行业不断创新和敢为人先的精神。

# 目录

## 第一部分 入门

<b>1 第1章 初识 Redis</b>	<b>2</b>
1.1 Redis 简介	3
1.1.1 Redis 与其他数据库和 软件的对比	3
1.1.2 附加特性	4
1.1.3 使用 Redis 的理由	5
1.2 Redis 数据结构简介	6
1.2.1 Redis 中的字符串	7
1.2.2 Redis 中的列表	9
1.2.3 Redis 的集合	10
1.2.4 Redis 的散列	11
1.2.5 Redis 的有序集合	12
1.3 你好 Redis	13
1.3.1 对文章进行投票	15
1.3.2 发布并获取文章	17
1.3.3 对文章进行分组	19
1.4 寻求帮助	21
1.5 小结	21

<b>2 第2章 使用 Redis 构建 Web 应用</b>	<b>23</b>
2.1 登录和 cookie 缓存	24
2.2 使用 Redis 实现购物车	28
2.3 网页缓存	29
2.4 数据行缓存	30
2.5 网页分析	33
2.6 小结	34

## 第二部分 核心概念

<b>3 第3章 Redis 命令</b>	<b>38</b>
3.1 字符串	39
3.2 列表	42
3.3 集合	44
3.4 散列	46
3.5 有序集合	48
3.6 发布与订阅	52
3.7 其他命令	54
3.7.1 排序	54
3.7.2 基本的 Redis 事务	56
3.7.3 键的过期时间	58
3.8 小结	60

<b>4 第4章 数据安全与性能保障</b>	<b>61</b>
4.1 持久化选项	61
4.1.1 快照持久化	62
4.1.2 AOF 持久化	66
4.1.3 重写/压缩 AOF 文件	67
4.2 复制	68
4.2.1 对 Redis 的复制相关选项进 行配置	69
4.2.2 Redis 复制的启动过程	70
4.2.3 主从链	71
4.2.4 检验硬盘写入	72
4.3 处理系统故障	73
4.3.1 验证快照文件和 AOF 文件	74
4.3.2 更换故障主服务器	75

4.4 Redis 事务	76
4.4.1 定义用户信息和用户包裹	77
4.4.2 将商品放到市场上销售	78
4.4.3 购买商品	80
4.5 非事务型流水线	82
4.6 关于性能方面的注意事项	85
4.7 小结	87

## 5 第5章 使用 Redis 构建支持程序 88

5.1 使用 Redis 来记录日志	88
5.1.1 最新日志	89
5.1.2 常见日志	90
5.2 计数器和统计数据	91
5.2.1 将计数器存储到 Redis 里面	91
5.2.2 使用 Redis 存储统计数据	96
5.2.3 简化统计数据的记录与发现	98
5.3 查找 IP 所属城市以及国家	100
5.3.1 载入位置表格	100
5.3.2 查找 IP 所属城市	102
5.4 服务的发现与配置	103
5.4.1 使用 Redis 存储配置信息	103
5.4.2 为每个应用程序组件分别配置一个 Redis 服务器	104
5.4.3 自动 Redis 连接管理	106
5.5 小结	107

## 6 第6章 使用 Redis 构建应用程序组件 109

6.1 自动补全	109
6.1.1 自动补全最近联系人	110
6.1.2 通讯录自动补全	112
6.2 分布式锁	115
6.2.1 锁的重要性	116
6.2.2 简易锁	118
6.2.3 使用 Redis 构建锁	119

6.2.4 细粒度锁	122
6.2.5 带有超时限制特性的锁	124
6.3 计数信号量	126
6.3.1 构建基本的计数信号量	126
6.3.2 公平信号量	128
6.3.3 刷新信号量	131
6.3.4 消除竞争条件	132
6.4 任务队列	133
6.4.1 先进先出队列	133
6.4.2 延迟任务	136
6.5 消息拉取	139
6.5.1 单接收者消息的发送与订阅替代品	140
6.5.2 多接收者消息的发送与订阅替代品	141
6.6 使用 Redis 进行文件分发	145
6.6.1 根据地理位置聚合用户数据	146
6.6.2 发送日志文件	148
6.6.3 接收日志文件	149
6.6.4 处理日志文件	150
6.7 小结	152

## 7 第7章 基于搜索的应用程序 153

7.1 使用 Redis 进行搜索	153
7.1.1 基本搜索原理	154
7.1.2 对搜索结果进行排序	160
7.2 有序索引	162
7.2.1 使用有序集合对搜索结果进行排序	162
7.2.2 使用有序集合实现非数值排序	164
7.3 广告定向	166
7.3.1 什么是广告服务器	167
7.3.2 对广告进行索引	167
7.3.3 执行广告定向操作	170
7.3.4 从用户行为中学习	174
7.4 职位搜索	180
7.4.1 逐个查找合适的职位	180
7.4.2 以搜索方式查找合适的职位	181
7.5 小结	182

## 8 第8章 构建简单的社交网站 184

- 8.1 用户和状态 185
  - 8.1.1 用户信息 185
  - 8.1.2 状态消息 186
- 8.2 主页时间线 187
- 8.3 关注者列表和正在关注列表 188
- 8.4 状态消息的发布与删除 191
- 8.5 流 API 194
  - 8.5.1 流 API 提供的数据 195
  - 8.5.2 提供数据 196
  - 8.5.3 对流消息进行过滤 199
- 8.6 小结 205

## 第三部分 进阶内容

### 9 第9章 降低内存占用 208

- 9.1 短结构 208
  - 9.1.1 压缩列表表示 209
  - 9.1.2 集合的整数集合编码 211
  - 9.1.3 长压缩列表和大整数集合带来的性能问题 212
- 9.2 分片结构 214
  - 9.2.1 分片式散列 215
  - 9.2.2 分片集合 218
- 9.3 打包存储二进制位和字节 221
  - 9.3.1 决定被存储位置信息的格式 221
  - 9.3.2 存储打包后的数据 223
  - 9.3.3 对分片字符串进行聚合计算 224
- 9.4 小结 226

### 10 第10章 扩展 Redis 227

- 10.1 扩展读性能 227
- 10.2 扩展写性能和内存容量 230
  - 10.2.1 处理分片配置信息 232

- 10.2.2 创建分片服务器连接装饰器 233
- 10.3 扩展复杂的查询 234
  - 10.3.1 扩展搜索查询量 235
  - 10.3.2 扩展搜索索引大小 235
  - 10.3.3 对社交网站进行扩展 240
- 10.4 小结 247

## 11 第11章 Redis 的 Lua 脚本编程 248

- 11.1 在不编写 C 代码的情况下添加新功能 248
  - 11.1.1 将 Lua 脚本载入 Redis 249
  - 11.1.2 创建新的状态消息 251
- 11.2 使用 Lua 重写锁和信号量 254
  - 11.2.1 使用 Lua 实现锁的原因 254
  - 11.2.2 重写锁实现 255
  - 11.2.3 使用 Lua 实现计数信号量 257
- 11.3 移除 WATCH/MULTI/EXEC 事务 258
  - 11.3.1 回顾群组自动补全程序 259
  - 11.3.2 再次对商品买卖市场进行改进 261
- 11.4 使用 Lua 对列表进行分片 263
  - 11.4.1 分片列表的构成 263
  - 11.4.2 将元素推入分片列表 265
  - 11.4.3 从分片里面里面弹出元素 266
  - 11.4.4 对分片列表执行阻塞弹出操作 267
- 11.5 小结 270

## A 附录 A 快速安装指南 271

## B 附录 B 其他资源和参考资料 279

# 第一部分

## 入门

本书最开始的两章将对 Redis 进行介绍，并展示 Redis 的一些基本用法。读完这两章之后，读者应该能够用 Redis 对自己的项目进行一些简单的优化。

# 第1章 初识 Redis

## 本章主要内容

- Redis 与其他软件的相同之处和不同之处
- Redis 的用法
- 使用 Python 示例代码与 Redis 进行简单的互动
- 使用 Redis 解决实际问题

Redis 是一个远程内存数据库，它不仅性能强劲，而且还具有复制特性以及为解决问题而生的独一无二的数据模型。Redis 提供了 5 种不同类型的数据结构，各式各样的问题都可以很自然地映射到这些数据结构上：Redis 的数据结构致力于帮助用户解决问题，而不会像其他数据库那样，要求用户扭曲问题来适应数据库。除此之外，通过复制、持久化（persistence）和客户端分片（client-side sharding）等特性，用户可以很方便地将 Redis 扩展成一个能够包含数百 GB 数据、每秒处理上百万次请求的系统。

笔者第一次使用 Redis 是在一家公司里面，这家公司需要对一个保存了 6 万个客户联系方式的关系数据库进行搜索，搜索可以根据名字、邮件地址、所在地和电话号码来进行，每次搜索需要花费 10~15 秒的时间。在花了一周时间学习 Redis 的基础知识之后，我使用 Redis 重写了一个新的搜索引擎，然后又花费了数周时间来仔细测试这个新系统，使它达到生产级别，最终这个新的搜索系统不仅可以根据名字、邮件地址、所在地和电话号码等信息来过滤和排序客户联系方式，并且每次操作都可以在 50 毫秒之内完成，这比原来的搜索系统足足快了 200 倍。阅读本书可以让你学到很多小技巧、小窍门以及使用 Redis 解决某些常见问题的方法。

本章将介绍 Redis 的适用范围，以及在不同环境中使用 Redis 的方法（比如怎样跟不同的组件和编程语言进行通信等）；而之后的章节则会展示各式各样的问题，以及使用 Redis 来解决这些问题的方法。

现在你已经知道我是怎样开始使用 Redis 的了，也知道了这本书大概要讲些什么内容了，是时候更详细地介绍一下 Redis，并说明为什么应该使用 Redis 了。

安装 Redis 和 Python 附录 A 介绍了快速安装 Redis 和 Python 的方法。

在其他编程语言里面使用 Redis 本书只展示了使用 Python 语言编写的示例代码，使用 Ruby、Java 和 JavaScript（Node.js）编写的示例代码可以在这里找到：<https://github.com/josiahcarlson/redis-in-action>。使用 Spring 框架的读者可以通过查看 <http://www.springsource.org/spring-data/redis> 来学习如何在 Spring 框架中使用 Redis。

## 1.1 Redis 简介

前面对于 Redis 数据库的描述只说出了一部分真相。Redis 是一个速度非常快的非关系数据库（non-relational database），它可以存储键（key）与 5 种不同类型的值（value）之间的映射（mapping），可以将存储在内存的键值对数据持久化到硬盘，可以使用复制特性来扩展读性能，还可以使用客户端分片<sup>①</sup>来扩展写性能，接下来的几节将分别介绍 Redis 的这几个特性。

### 1.1.1 Redis 与其他数据库和软件的对比

如果你熟悉关系数据库，那么你肯定写过用来关联两个表的数据的 SQL 查询。而 Redis 则属于人们常说的 NoSQL 数据库或者非关系数据库：Redis 不使用表，它的数据库也不会预定义或者强制去要求用户对 Redis 存储的不同数据进行关联。

高性能键值缓存服务器 memcached 也经常被拿来与 Redis 进行比较：这两者都可用于存储键值映射，彼此的性能也相差无几，但是 Redis 能够自动以两种不同的方式将数据写入硬盘，并且 Redis 除了能存储普通的字符串键之外，还可以存储其他 4 种数据结构，而 memcached 只能存储普通的字符串键。这些不同之处使得 Redis 可以用于解决更为广泛的问题，并且既可以用作主数据库（primary database）使用，又可以作为其他存储系统的辅助数据库（auxiliary database）使用。

本书的后续章节会分别介绍将 Redis 用作主存储（primary storage）和二级存储（secondary storage）时的用法和查询模式。一般来说，许多用户只会在 Redis 的性能或者功能是必要的情况下，才会将数据存储到 Redis 里面：如果程序对性能的要求不高，又或者因为费用原因而没办法将大量数据存储到内存里面，那么用户可能会选择使用关系数据库，或者其他非关系数据库。在实际中，读者应该根据自己的需求来决定是否使用 Redis，并考虑是将 Redis 用作主存储还是辅

<sup>①</sup> 分片是一种将数据划分为多个部分的方法，对数据的划分可以基于键包含的 ID、基于键的散列值，或者基于以上两者的某种组合。通过对数据进行分片，用户可以将数据存储到多台机器里面，也可以从多台机器里面获取数据，这种方法在解决某些问题时可以获得线性级别的性能提升。

助存储，以及如何通过复制、持久化和事务等手段保证数据的完整性。

表1-1展示了部分在功能上与Redis有重叠的数据库服务器和缓存服务器，从这个表可以看出Redis与这些数据库及软件之间的区别。

表1-1 一些数据库和缓存服务器的特性与功能

名称	类型	数据存储选项	查询类型	附加功能
Redis	使用内存存储(in-memory)的非关系数据库	字符串、列表、集合、散列表、有序集合	每种数据类型都有自己的专属命令，另外还有批量操作(bulk operation)和不完全(partial)的事务支持	发布与订阅，主从复制(master/slave replication)，持久化，脚本(stored procedure)
memcached	使用内存存储的键值缓存	键值之间的映射	创建命令、读取命令、更新命令、删除命令以及其他几个命令	为提升性能而设的多线程服务器
MySQL	关系数据库	每个数据库可以包含多个表，每个表可以包含多个行；可以处理多个表的视图(view)；支持空间(spatial)和第三方扩展	SELECT、INSERT、UPDATE、DELETE、函数、存储过程	支持ACID性质(需要使用InnoDB)，主从复制和主主复制(master/master replication)
PostgreSQL	关系数据库	每个数据库可以包含多个表，每个表可以包含多个行；可以处理多个表的视图；支持空间和第三方扩展；支持可定制类型	SELECT、INSERT、UPDATE、DELETE、内置函数、自定义的存储过程	支持ACID性质，主从复制，由第三方支持的多主复制(multi-master replication)
MongoDB	使用硬盘存储(on-disk)的非关系文档存储	每个数据库可以包含多个表，每个表可以包含多个无schema(schema-less)的BSON文档	创建命令、读取命令、更新命令、删除命令、条件查询命令等	支持map-reduce操作，主从复制，分片，空间索引(spatial index)

## 1.1.2 附加特性

在使用类似Redis这样的内存数据库时，一个首先要考虑的问题就是“当服务器被关闭时，服务器存储的数据将何去何从呢？”Redis拥有两种不同形式的持久化方法，它们都可以用小而紧凑的格式将存储在内存中的数据写入硬盘：第一种持久化方法为时间点转储(point-in-time dump)，转储操作既可以在“指定时间段内有指定数量的写操作执行”这一条件被满足时执行，又可以通过调用两条转储到硬盘(dump-to-disk)命令中的任何一条来执行；第二种持久化方法将所有修改了数据库的命令都写入一个只追加 append-only 文件里面，用户可以根据数据的重要程度，将只追加写入设置为从不同步(sync)、每秒同步一次或者每写入一个命令就同步一次。我们将在第4章中更加深入地讨论这些持久化选项。

另外，尽管Redis的性能很好，但受限于Redis的内存存储设计，有时候只使用一台Redis服务器可能没有办法处理所有请求。因此，为了扩展Redis的读性能，并为Redis提供故障转移

( failover ) 支持, Redis 实现了主从复制特性: 执行复制的从服务器会连接上主服务器, 接收主服务器发送的整个数据库的初始副本 ( copy ); 之后主服务器执行的写命令, 都会被发送给所有连接着的从服务器去执行, 从而实时地更新从服务器的数据集。因为从服务器包含的数据会不断地进行更新, 所以客户端可以向任意一个从服务器发送读请求, 以此来避免对主服务器进行集中式的访问。我们将在第 4 章中更加深入地讨论 Redis 从服务器。

### 1.1.3 使用 Redis 的理由

有 memcached 使用经验的读者可能知道, 用户只能用 APPEND 命令将数据添加到已有字符串的末尾。memcached 的文档中声明, 可以用 APPEND 命令来管理元素列表。这很好! 用户可以将元素追加到一个字符串的末尾, 并将那个字符串当作列表来使用。但随后如何删除这些元素呢? memcached 采用的办法是通过黑名单 ( blacklist ) 来隐藏列表里面的元素, 从而避免对元素执行读取、更新、写入 ( 包括在一次数据库查询之后执行的 memcached 写入 ) 等操作。相反地, Redis 的 LIST 和 SET 允许用户直接添加或者删除元素。

使用 Redis 而不是 memcached 来解决问题, 不仅可以让代码变得更简短、更易懂、更易维护, 而且还可以使代码的运行速度更快 ( 因为用户不需要通过读取数据库来更新数据 )。除此之外, 在其他许多情况下, Redis 的效率和易用性也比关系数据库要好得多。

数据库的一个常见用法是存储长期的报告数据, 并将这些报告数据用作固定时间范围内的聚合数据 ( aggregates )。收集聚合数据的常见做法是: 先将各个行插入一个报告表里面, 之后再通过扫描这些行来收集聚合数据, 并根据收集到的聚合数据来更新聚合表中已有的那些行。之所以使用插入行的方式来存储, 是因为对于大部分数据库来说, 插入行操作的执行速度非常快 ( 插入行只会在硬盘文件末尾进行写入 )。不过, 对表里面的行进行更新却是一个速度相当慢的操作, 因为这种更新除了会引起一次随机读 ( random read ) 之外, 还可能会引起一次随机写 ( random write )。而在 Redis 里面, 用户可以直接使用原子的 ( atomic ) INCR 命令及其变种来计算聚合数据, 并且因为 Redis 将数据存储在内存里面<sup>①</sup>, 而且发送给 Redis 的命令请求并不需要经过典型的查询分析器 ( parser ) 或者查询优化器 ( optimizer ) 进行处理, 所以对 Redis 存储的数据执行随机写的速度总是非常迅速的。

使用 Redis 而不是关系数据库或者其他硬盘存储数据库, 可以避免写入不必要的临时数据, 也免去了对临时数据进行扫描或者删除的麻烦, 并最终改善程序的性能。虽然上面列举的都是一些简单的例子, 但它们很好地证明了“工具会极大地改变人们解决问题的方式”这一点。

---

① 客观来讲, memcached 也能用在这个简单的场景里, 但使用 Redis 存储聚合数据有以下 3 个好处: 首先, 使用 Redis 可以将彼此相关的聚合数据放在同一个结构里面, 这样访问聚合数据就会变得更容易; 其次, 使用 Redis 可以将聚合数据放到有序集合里面, 构建出一个实时的排行榜; 最后, Redis 的聚合数据可以是整数或者浮点数, 而 memcached 的聚合数据只能是整数。

除了第 6 章提到的任务队列 (task queue) 之外, 本书的大部分内容都致力于实时地解决问题。本书通过展示各种技术并提供可工作的代码来帮助读者消灭瓶颈、简化代码、收集数据、分发 (distribute) 数据、构建实用程序 (utility), 并最终帮助读者更轻松地完成构建软件的任务。只要正确地使用书中介绍的技术, 读者的软件就可以扩展至令那些所谓的“Web 扩展技术 (web-sacle technology)”相形见绌的地步。

在了解了 Redis 是什么、它能做什么以及我们为什么要使用它之后, 是时候来实际地使用一下它了。接下来的一节将对 Redis 提供的数据结构进行介绍, 说明这些数据结构的作用, 并展示操作这些数据结构的其中一部分命令。

## 1.2 Redis 数据结构简介

正如之前的表 1-1 所示, Redis 可以存储键与 5 种不同数据结构类型之间的映射, 这 5 种数据结构类型分别为 STRING (字符串)、LIST (列表)、SET (集合)、HASH (散列) 和 ZSET (有序集合)。有一部分 Redis 命令对于这 5 种结构都是通用的, 如 DEL、TYPE、RENAME 等; 但也有一部分 Redis 命令只能对特定的一种或者两种结构使用, 第 3 章将对 Redis 提供的命令进行更深入的介绍。

大部分程序员应该都不会对 Redis 的 STRING、LIST、HASH 这 3 种结构感到陌生, 因为它们和很多编程语言内建的字符串、列表和散列等结构在实现和语义 (semantics) 方面都非常相似。有些编程语言还有集合数据结构, 在实现和语义上类似于 Redis 的 SET。ZSET 在某种程度上是一种 Redis 特有的结构, 但是当你熟悉了它之后, 就会发现它也是一种非常有用的结构。表 1-2 对比了 Redis 提供的 5 种结构, 说明了这些结构存储的值, 并简单介绍了它们的语义。

表 1-2 Redis 提供的 5 种结构

结构类型	结构存储的值	结构的读写能力
STRING	可以是字符串、整数或者浮点数	对整个字符串或者字符串的其中一部分执行操作; 对整数和浮点数执行自增 (increment) 或者自减 (decrement) 操作
LIST	一个链表, 链表上的每个节点都包含了一个字符串	从链表的两端推入或者弹出元素; 根据偏移量对链表进行修剪 (trim); 读取单个或者多个元素; 根据值查找或者移除元素
SET	包含字符串的无序收集器 (unordered collection), 并且被包含的每个字符串都是独一无二、各不相同的	添加、获取、移除单个元素; 检查一个元素是否存在于集合中; 计算交集、并集、差集; 从集合里面随机获取元素
HASH	包含键值对的无序散列表	添加、获取、移除单个键值对; 获取所有键值对
ZSET (有序集合)	字符串成员 (member) 与浮点数分值 (score) 之间的有序映射, 元素的排列顺序由分值的大小决定	添加、获取、删除单个元素; 根据分值范围 (range) 或者成员来获取元素

**命令列表** 本节在介绍每个数据类型的时候，都会在一个表格里面展示一小部分处理这些数据结构的命令，之后的第 3 章会展示一个更详细（但仍不完整）的命令列表，完整的 Redis 命令列表可以在 <http://redis.io/commands> 找到。

这一节将介绍如何表示 Redis 的这 5 种结构，并且还会介绍 Redis 命令的使用方法，从而为本书的后续内容打好基础。本书展示的所有示例代码都是用 Python 写的，如果读者已经按照附录 A 里面描述的方法安装好了 Redis，那么应该也已经安装好了 Python，以及在 Python 里面使用 Redis 所需的客户端库。只要读者在电脑里面安装了 Redis、Python 和 redis-py 库，就可以在阅读本书的同时，尝试执行书中展示的示例代码了。

**请安装 Redis 和 Python** 在阅读后续内容之前，请读者先按照附录 A 中介绍的方法安装 Redis 和 Python。如果读者觉得附录 A 描述的安装方法过于复杂，那么这里有一个更简单的方法，但这个方法只能用于 Debian 系统（或者该系统的衍生系统）：从 <http://redis.io/download> 下载 Redis 的压缩包，解压压缩包，执行 `make && sudo make install`，之后再执行 `sudo python -m easy_install redis hiredis` (`hiredis` 是可选的，它是一个使用 C 语言编写的高性能 Redis 客户端）。

如果读者熟悉过程式编程语言或者面向对象编程语言，那么即使没有使用过 Python，应该也可以看懂 Python 代码。另一方面，如果读者决定使用其他编程语言来操作 Redis，那么就需要自己来将本书的 Python 代码翻译成正在使用的语言的代码。

**使用其他语言编写的示例代码** 尽管没有包含在书中，但本书展示的 Python 示例代码已经被翻译成了 Ruby 代码、Java 代码和 JavaScript 代码，这些翻译代码可以在 <https://github.com/josiahcarlson/redis-in-action> 下载到。跟 Python 编写的示例代码一样，这些翻译代码也包含相应的注释，方便读者参考。

为了让示例代码尽可能地简单，本书会尽量避免使用 Python 的高级特性，并使用函数而不是类或者其他东西来执行 Redis 操作，以此来将焦点放在使用 Redis 解决问题上面，而不必过多地关注 Python 的语法。本节将使用 redis-cli 控制台与 Redis 进行互动。首先，让我们来了解一下 Redis 中最简单的结构：STRING。

## 1.2.1 Redis 中的字符串

Redis 的字符串和其他编程语言或者其他键值存储提供的字符串非常相似。本书在使用图片表示键和值的时候，通常会将键名（key name）和值的类型放在方框的顶部，并将值放在方框的里面。图 1-1 以键为 hello、值为 world 的字符串为例，分别标记了方框的各个部分。

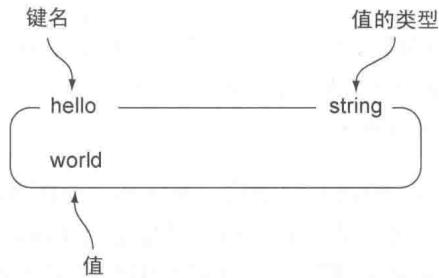


图 1-1 一个字符串示例，键为 hello，值为 world

字符串拥有一些和其他键值存储相似的命令，比如 GET（获取值）、SET（设置值）和 DEL（删除值）。如果读者已经按照附录 A 中给出的方法安装了 Redis，那么可以根据代码清单 1-1 展示的例子，尝试使用 redis-cli 执行 SET、GET 和 DEL，表 1-3 描述了这 3 个命令的基本用法。

表 1-3 字符串命令

命令	行为
GET	获取存储在给定键中的值
SET	设置存储在给定键中的值
DEL	删除存储在给定键中的值（这个命令可以用于所有类型）

### 代码清单 1-1 SET、GET 和 DEL 的使用示例

```
$ redis-cli
redis 127.0.0.1:6379> set hello world
OK
redis 127.0.0.1:6379> get hello
"world"
redis 127.0.0.1:6379> del hello
(integer) 1
redis 127.0.0.1:6379> get hello
(nil)
redis 127.0.0.1:6379>
```

启动 redis-cli  
客户端。  
将键 hello 的值  
设置为 world。  
键的值仍然是 world，跟我们刚  
才设置的一样。  
删除这个键值对。  
因为键的值已经不存在，所以尝  
试获取键的值将得到一个 nil，  
Python 客户端会将这个 nil 转  
换成 None。

SET 命令在执行成功  
时返回 OK，Python  
客户端会将这个 OK  
转换成 True。  
获取存储在键 hello 中  
的值。  
在对值进行删除的时  
候，DEL 命令将返回被  
成功删除的值的数量。

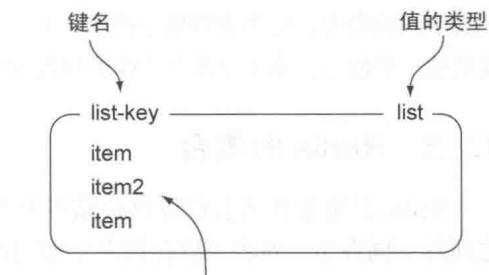
**使用 redis-cli** 为了让读者在一开始就能便捷地与 Redis 进行交互，本章将使用 redis-cli 这个交互式客户端来介绍 Redis 命令。

除了能够 GET、SET 和 DEL 字符串值之外，Redis 还提供了一些可以对字符串的其中一部分内容进行读取和写入的命令，以及一些能对字符串存储的数值执行自增或者自减操作的命令。第 3 章将对这些命令进行介绍，但是在此之前，我们还有许多基础知识需要了解，下面来看一下 Redis 的列表及其功能。

## 1.2.2 Redis 中的列表

Redis 对链表 (linked-list) 结构的支持使得它在键值存储的世界中独树一帜。一个列表结构可以有序地存储多个字符串，和表示字符串时使用的方法一样，本节使用带有标签的方框来表示列表，并将列表包含的元素放在方框里面。图 1-2 展示了一个这样的示例。

Redis 列表可执行的操作和很多编程语言里面的列表操作非常相似：LPUSH 命令和 RPUSH 命令分别用于将元素推入列表的左端 (left end) 和右端 (right end)；LPOP 命令和 RPOP 命令分别用于从列表的左端和右端弹出元素；LINDEX 命令用于获取列表在给定位置上的一个元素；LRANGE 命令用于获取列表在给定范围上的所有元素。代码清单 1-2 展示了一些列表命令的使用示例，表 1-4 简单介绍了示例中用到的各个命令。



列表包含的元素，相同元素可以重复出现

图 1-2 list-key 是一个包含 3 个元素的列表键，注意列表里面的元素是可以重复的

表 1-4 列表命令

命令	行为
RPUSH	将给定值推入列表的右端
LRANGE	获取列表在给定范围上的所有值
LINDEX	获取列表在给定位置上的单个元素
LPOP	从列表的左端弹出一个值，并返回被弹出的值

### 代码清单 1-2 RPUSH、LRANGE、LINDEX 和 LPOP 的使用示例

```
redis 127.0.0.1:6379> rpush list-key item
(integer) 1
redis 127.0.0.1:6379> rpush list-key item2
(integer) 2
redis 127.0.0.1:6379> rpush list-key item
(integer) 3
redis 127.0.0.1:6379> lrange list-key 0 -1
1) "item"
2) "item2"
3) "item"
redis 127.0.0.1:6379> lindex list-key 1
"item2"
redis 127.0.0.1:6379> lpop list-key
"item"
redis 127.0.0.1:6379> lrange list-key 0 -1
1) "item2"
2) "item"
redis 127.0.0.1:6379>
```

使用 0 为范围的起始索引，-1 为范围的结束索引，可以取出列表包含的所有元素。

在向列表推入新元素之后，该命令会返回列表当前的长度。

在向列表推入新元素之后，该命令会返回列表当前的长度。

使用 LINDEX 可以从列表里面取出单个元素。

从列表里面弹出一个元素，被弹出的元素将不再存在于列表。

即使 Redis 的列表只支持以上提到的几个命令，它也已经可以用来解决很多问题了，但 Redis 并没有就此止步——除了上面提到的命令之外，Redis 列表还拥有从列表里面移除元素的命令、将元素插入列表中间的命令、将列表修剪至指定长度（相当于从列表的其中一端或者两端移除元素）的命令，以及其他一些命令。第 3 章将介绍许多列表命令，但是在此之前，让我们先来了解一下 Redis 的集合。

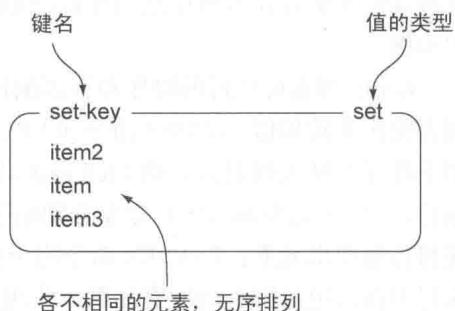
### 1.2.3 Redis 的集合

Redis 的集合和列表都可以存储多个字符串，它们之间的不同在于，列表可以存储多个相同的字符串，而集合则通过使用散列表来保证自己存储的每个字符串都是各不相同的（这些散列表只有键，但没有与键相关联的值）。本书表示集合的方法和表示列表的方法基本相同，图 1-3 展示了一个包含 3 个元素的示例集合。

因为 Redis 的集合使用无序（unordered）方式存储元素，所以用户不能像使用列表那样，将元素推入集合的某一端，或者从集合的某一端弹出元素。不过用户可以使用 SADD 命令将元素添加到集合，或者使用 SREM 命令从集合里面移除元素。另外还可以通过 SISMEMBER 命令快速地检查一个元素是否已经存在于集合中，或者使用 SMEMBERS 命令获取集合包含的所有元素（如果集合包含的元素非常多，那么 SMEMBERS 命令的执行速度可能会很慢，所以请谨慎地使用这个命令）。代码清单 1-3 展示了一些集合命令的使用示例，表 1-5 简单介绍了代码清单里面用到的各个命令。

代码清单 1-3 SADD、SMEMBERS、SISMEMBER 和 SREM 的使用示例

```
redis 127.0.0.1:6379> sadd set-key item
(integer) 1
redis 127.0.0.1:6379> sadd set-key item2
(integer) 1
redis 127.0.0.1:6379> sadd set-key item3
(integer) 1
redis 127.0.0.1:6379> sadd set-key item
(integer) 0
redis 127.0.0.1:6379> smembers set-key
1) "item"
2) "item2"
3) "item3"
redis 127.0.0.1:6379> sismember set-key item4
(integer) 0
redis 127.0.0.1:6379> sismember set-key item
(integer) 1
redis 127.0.0.1:6379> srem set-key item2
(integer) 1
redis 127.0.0.1:6379> srem set-key item2
(integer) 0
```



各不相同的元素，无序排列

图 1-3 set-key 是一个包含  
3 个元素的集合键

在尝试将一个元素添加到集合的时候，命令返回 1 表示这个元素被成功地添加到了集合里面，而返回 0 则表示这个元素已经存在于集合中。

获取集合包含的所有元素将得到一个由元素组成的序列，Python 客户端会将这个序列转换成 Python 集合。

检查一个元素是否存在于集合中，Python 客户端会返回一个布尔值来表示检查结果。

在使用命令移除集合中的元素时，命令会返回被移除元素的数量。

```
redis 127.0.0.1:6379> smembers set-key
1) "item"
2) "item3"
redis 127.0.0.1:6379>
```

表 1-5 集合命令

命令	行为
SADD	将给定元素添加到集合
SMEMBERS	返回集合包含的所有元素
SISMEMBER	检查给定元素是否存在于集合中
SRM	如果给定的元素存在于集合中，那么移除这个元素

跟字符串和列表一样，集合除了基本的添加操作和移除操作之外，还支持很多其他操作，比如 SINTER、SUNION、SDIFF 这 3 个命令就可以分别执行常见的交集计算、并集计算和差集计算。第 3 章将对集合的相关命令进行更详细的介绍，另外第 7 章还会展示如何使用集合来解决多个问题。不过别心急，因为在 Redis 提供的 5 种数据结构中，还有两种我们尚未了解，让我们先来看看 Redis 的散列。

#### 1.2.4 Redis 的散列

Redis 的散列可以存储多个键值对之间的映射。和字符串一样，散列存储的值既可以是字符串又可以是数字值，并且用户同样可以对散列存储的数字值执行自增操作或者自减操作。图 1-4 展示了一个包含两个键值对的散列。

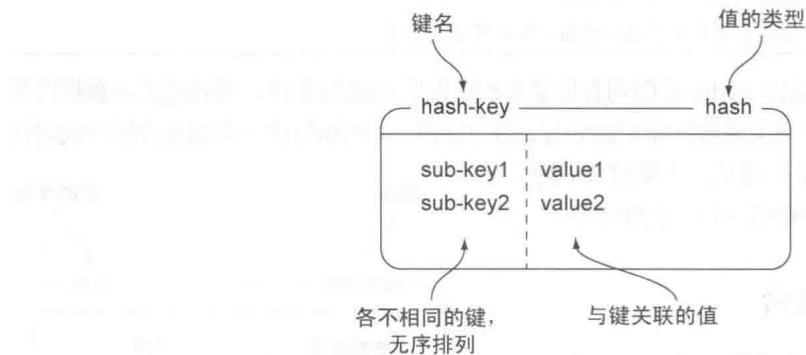


图 1-4 hash-key 是一个包含两个键值对的散列键

散列在很多方面就像是一个微缩版的 Redis，不少字符串命令都有相应的散列版本。代码清单 1-4 展示了怎样对散列执行插入元素、获取元素和移除元素等操作，表 1-6 简单介绍了代码清单里面用到的各个命令。

## 代码清单 1-4 HSET、HGET、HGETALL 和 HDEL 的使用示例

```
redis 127.0.0.1:6379> hset hash-key sub-key1 value1
(integer) 1
redis 127.0.0.1:6379> hset hash-key sub-key2 value2
(integer) 1
redis 127.0.0.1:6379> hset hash-key sub-key1 value1
(integer) 0
redis 127.0.0.1:6379> hgetall hash-key
1) "sub-key1"
2) "value1"
3) "sub-key2"
4) "value2"
redis 127.0.0.1:6379> hdel hash-key sub-key2
(integer) 1
redis 127.0.0.1:6379> hdel hash-key sub-key2
(integer) 0
redis 127.0.0.1:6379> hget hash-key sub-key1
"value1"
redis 127.0.0.1:6379> hgetall hash-key
1) "sub-key1"
2) "value1"
```

在尝试添加键值对到散列的时候，命令会返回一个值来表示给定的键是否已经存在于散列里面。

在获取散列包含的所有键值对时，Python 客户端会把整个散列转换成一个 Python 字典。

在删除键值对的时候，命令会返回一个值来表示给定的键在移除之前是否存在于散列里面。

从散列里面获取某个键的值。

表 1-6 散列命令

命令	行为
HSET	在散列里面关联起给定的键值对
HGET	获取指定散列键的值
HGETALL	获取散列包含的所有键值对
HDEL	如果给定键存在于散列里面，那么移除这个键

熟悉文档数据库的读者可以将 Redis 的散列看作是文档数据库里面的文档，而熟悉关系数据库的读者则可以将 Redis 的散列看作是关系数据库里面的行，因为散列、文档和行这三者都允许用户同时访问或者修改一个或多个域（field）。最后，让我们来了解一下 Redis 的 5 种数据结构中的最后一种：有序集合。

## 1.2.5 Redis 的有序集合

有序集合和散列一样，都用于存储键值对：有序集合的键被称为成员（member），每个成员都是各不相同的；而有序集合的值则被称为分值（score），分值必须为浮点数。有序集合是 Redis 里面唯一一个既可以根据成员访问元素（这一点和散列一样），又可以根据分值以及分值的排列顺序来访问元素的结构。图 1-5 展

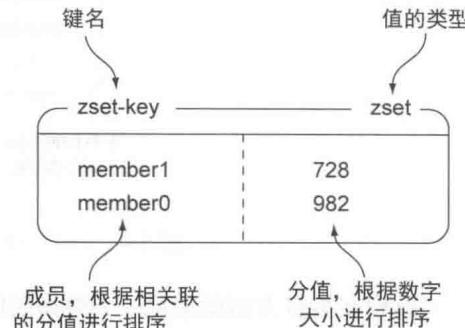


图 1-5 zset-key 是一个包含两个元素的有序集合键

示了一个包含两个元素的有序集合示例。

和 Redis 的其他结构一样，用户可以对有序集合执行添加、移除和获取等操作，代码清单 1-5 展示了这些操作的执行示例，表 1-7 简单介绍了代码清单里面用到的各个命令。

#### 代码清单 1-5 ZADD、ZRANGE、ZRANGEBYSCORE 和 ZREM 的使用示例

```
redis 127.0.0.1:6379> zadd zset-key 728 member1
(integer) 1
redis 127.0.0.1:6379> zadd zset-key 982 member0
(integer) 1
redis 127.0.0.1:6379> zadd zset-key 982 member0
(integer) 0
redis 127.0.0.1:6379> zrange zset-key 0 -1 withscores
1) "member1"
2) "728"
3) "member0"
4) "982"
redis 127.0.0.1:6379> zrangebyscore zset-key 0 800 withscores
1) "member1"
2) "728"
redis 127.0.0.1:6379> zrem zset-key member1
(integer) 1
redis 127.0.0.1:6379> zrem zset-key member1
(integer) 0
redis 127.0.0.1:6379> zrange zset-key 0 -1 withscores
1) "member0"
2) "982"
```

在尝试向有序集合添加元素的时候，命令会返回新添加元素的数量。

在获取有序集合包含的所有元素时，多个元素会按照分值大小进行排序，并且 Python 客户端会将元素的分值转换成浮点数。

用户也可以根据分值来获取有序集合中的一部分元素。

在移除有序集合元素的时候，命令会返回被移除元素的数量。

表 1-7 有序集合命令

命令	行为
ZADD	将一个带有给定分值的成员添加到有序集合里面
ZRANGE	根据元素在有序排列中所处的位置，从有序集合里面获取多个元素
ZRANGEBYSCORE	获取有序集合在给定分值范围内的所有元素
ZREM	如果给定成员存在于有序集合，那么移除这个成员

现在读者应该已经知道有序集合是什么和它能干什么了，到目前为止，我们基本了解了 Redis 提供的 5 种结构。接下来的一节将展示如何通过结合散列的数据存储能力和有序集合内建的排序能力来解决一个常见的问题。

## 1.3 你好 Redis

在对 Redis 提供的 5 种结构有了基本的了解之后，现在是时候来学习一下怎样使用这些结构来解决实际问题了。最近几年，越来越多的网站开始提供对网页链接、文章或者问题进行投票的功能，其中包括图 1-6 展示的 reddit 以及图 1-7 展示的 StackOverflow。这些网站会根据文章的发布时间和文章获得的投票数量计算出一个评分，然后按照这个评分来决定如何排序和展示文章。

本节将展示如何使用 Redis 来构建一个简单的文章投票网站的后端。

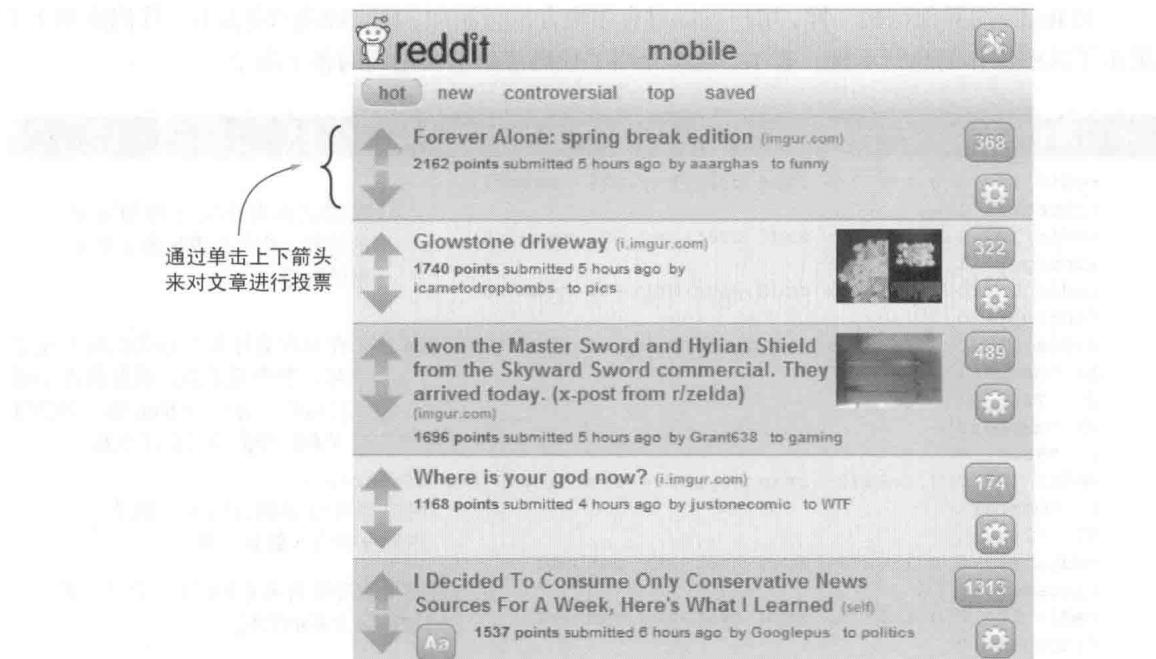


图 1-6 Reddit 是一个可以对文章进行投票的网站



图 1-7 StackOverflow 是一个可以对问题进行投票的网站

### 1.3.1 对文章进行投票

要构建一个文章投票网站，我们首先要做的就是为了这个网站设置一些数值和限制条件：如果一篇文章获得了至少 200 张支持票（up vote），那么网站就认为这篇文章是一篇有趣的文章；假如这个网站每天发布 1000 篇文章，而其中的 50 篇符合网站对有趣文章的要求，那么网站要做的就是把这 50 篇文章放到文章列表前 100 位至少一天；另外，这个网站暂时不提供投反对票（down vote）的功能。

为了产生一个能够随着时间流逝而不断减少的评分，程序需要根据文章的发布时间和当前时间来计算文章的评分，具体的计算方法为：将文章得到的支持票数量乘以一个常量，然后加上文章的发布时间，得出的结果就是文章的评分。

我们使用从 UTC 时区 1970 年 1 月 1 日到现在为止经过的秒数来计算文章的评分，这个值通常被称为 Unix 时间。之所以选择使用 Unix 时间，是因为在所有能够运行 Redis 的平台上面，使用编程语言获取这个值都是一件非常简单的事情。另外，计算评分时与支持票数量相乘的常量为 432，这个常量是通过将一天的秒数（86 400）除以文章展示一天所需的支持票数量（200）得出的：文章每获得一张支持票，程序就需要将文章的评分增加 432 分。

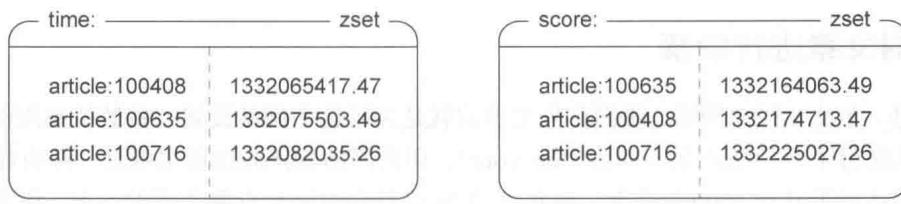
构建文章投票网站除了需要计算文章评分之外，还需要使用 Redis 结构存储网站上的各种信息。对于网站里的每篇文章，程序都使用一个散列来存储文章的标题、指向文章的网址、发布文章的用户、文章的发布时间、文章得到的投票数量等信息，图 1-8 展示了一个使用散列来存储文章信息的例子。

article:92617		hash
title	Go to statement considered harmful	
link	http://goo.gl/kZUSu	
poster	user:83271	
time	1331382699.33	
votes	528	

图 1-8 一个使用散列存储文章信息的例子

**使用冒号作为分隔符** 本书使用冒号（：）来分隔名字的不同部分：比如图 1-8 里面的键名 article:92617 就使用了冒号来分隔单词 article 和文章的 ID 号 92617，以此来构建命名空间（namespace）。使用：作为分隔符只是我的个人喜好，不过大部分 Redis 用户也都是这么做的，另外还有一些常见的分隔符，如句号（.）、斜线（/），有些人甚至还会使用管道符号（|）。无论使用哪个符号来做分隔符，都要保持分隔符的一致性。同时，请读者注意观察和学习本书使用冒号创建嵌套命名空间的方法。

我们的文章投票网站将使用两个有序集合来有序地存储文章：第一个有序集合的成员为文章 ID，分值为文章的发布时间；第二个有序集合的成员同样为文章 ID，而分值则为文章的评分。通过这两个有序集合，网站既可以根据文章发布的先后顺序来展示文章，又可以根据文章评分的高低来展示文章，图 1-9 展示了这两个有序集合的一个示例。



根据发布时间排序文章的有序集合

根据评分排序文章的有序集合

图 1-9 两个有序集合分别记录了根据发布时间排序的文章和根据评分排序的文章

为了防止用户对同一篇文章进行多次投票，网站需要为每篇文章记录一个已投票用户名单。为此，程序将为每篇文章创建一个集合，并使用这个集合来存储所有已投票用户的 ID，图 1-10 展示了一个这样的集合示例。

为了尽量节约内存，我们规定当一篇文章发布期满一周之后，用户将不能再对它进行投票，文章的评分将被固定下来，而记录文章已投票用户名单的集合也会被删除。

在实现投票功能之前，让我们来看看图 1-11：这幅图展示了当 115423 号用户给 100408 号文章投票的时候，数据结构发生的变化。

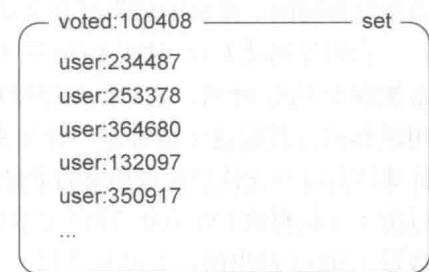
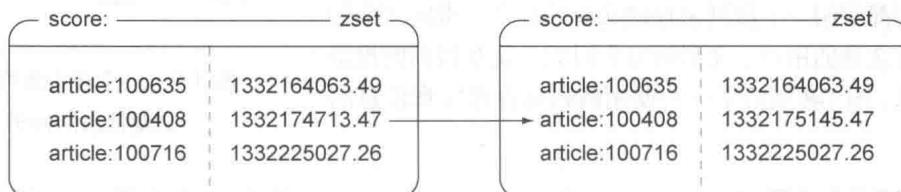
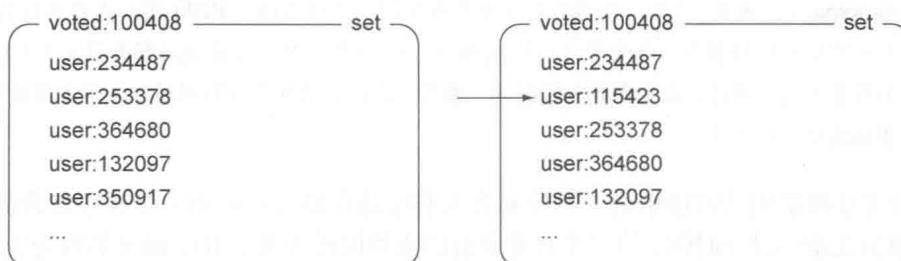


图 1-10 为 100408 号文章投过票的一部分用户



100408号文章得到了一张新的支持票，它的评分增加了



115423号用户会被追加到对100408号文章的已投票用户名单里面

图 1-11 当 115423 号用户给 100408 号文章投票的时候，数据结构发生的变化

既然我们已经知道了网站计算文章评分的方法，也知道了网站存储数据所需的数据结构，那么现在是时候实际地实现这个投票功能了！当用户尝试对一篇文章进行投票时，程序需要使用 ZSCORE 命令检查记录文章发布时间的有序集合，判断文章的发布时间是否未超过一周。如果文章仍然处于可以投票的时间范围之内，那么程序将使用 SADD 命令，尝试将用户添加到记录文章已投票用户名单的集合里面。如果添加操作执行成功的话，那么说明用户是第一次对这篇文章进行投票，程序将使用 ZINCRBY 命令为文章的评分增加 432 分（ZINCRBY 命令用于对有序集合成员的分值执行自增操作），并使用 HINCRBY 命令对散列记录的文章投票数量进行更新（HINCRBY 命令用于对散列存储的值执行自增操作），代码清单 1-6 展示了投票功能的实现代码。

#### 代码清单 1-6 article\_vote() 函数

```
ONE_WEEK_IN_SECONDS = 7 * 86400           | 准备好需要用到的常量。
VOTE_SCORE = 432                         |

def article_vote(conn, user, article):
    cutoff = time.time() - ONE_WEEK_IN_SECONDS
    if conn.zscore('time:', article) < cutoff:
        return

    article_id = article.partition(':')[1]
    if conn.sadd('voted:' + article_id, user):
        conn.zincrby('score:', article, VOTE_SCORE)
        conn.hincrby(article, 'votes', 1)

如果用户是第一次为这篇文章投票，那么增加这篇文章的投票数量和评分。
```

计算文章的投票截止时间。  
检查是否还可以对文章进行投票（虽然使用散列也可以获取文章的发布时间，但有序集合返回的文章发布时间为浮点数，可以不进行转换直接使用）。  
从 article:id 标识符 (identifier) 里面取出文章的 ID。

**Redis 事务** 从技术上来讲，要正确地实现投票功能，我们需要将代码清单 1-6 里面的 SADD、ZINCRBY 和 HINCRBY 这 3 个命令放到一个事务里面执行，不过因为本书要等到第 4 章才介绍 Redis 事务，所以我们暂时忽略这个问题。

这个投票功能还是很不错的，对吧？那么发布文章的功能要怎么实现呢？

### 1.3.2 发布并获取文章

发布一篇新文章首先需要创建一个新的文章 ID，这项工作可以通过对一个计数器（counter）执行 INCR 命令来完成。接着程序需要使用 SADD 将文章发布者的 ID 添加到记录文章已投票用户名单的集合里面，并使用 EXPIRE 命令为这个集合设置一个过期时间，让 Redis 在文章发布期满一周之后自动删除这个集合。之后，程序会使用 HMSET 命令来存储文章的相关信息，并执行两个 ZADD 命令，将文章的初始评分（initial score）和发布时间分别添加到两个相应的有序集合里面。代码清单 1-7 展示了发布新文章功能的实现代码。

## 代码清单 1-7 post\_article() 函数

```

def post_article(conn, user, title, link):
    article_id = str(conn.incr('article:'))
    ←—— 生成一个新的文章 ID。
    voted = 'voted:' + article_id
    conn.sadd(voted, user)
    conn.expire(voted, ONE_WEEK_IN_SECONDS)
    now = time.time()
    article = 'article:' + article_id
    conn.hmset(article, {
        'title': title,
        'link': link,
        'poster': user,
        'time': now,
        'votes': 1,
    })
    conn.zadd('score:', article, now + VOTE_SCORE)
    conn.zadd('time:', article, now)
    return article_id

```

将发布文章的用户添加到文章的已投票用户名单里面，然后将这个名单的过期时间设置为一周（第3章将对过期时间作更详细的介绍）。

将文章信息存储到一个散列里面。

将文章添加到根据发布时间排序的有序集合和根据评分排序的有序集合里面。

好了，我们已经陆续实现了文章投票功能和文章发布功能，接下来要考虑的就是如何取出评分最高的文章以及如何取出最新发布的文章了。为了实现这两个功能，程序需要先使用 ZREVRANGE 命令取出多个文章 ID，然后再对每个文章 ID 执行一次 HGETALL 命令来取出文章的详细信息，这个方法既可以用于取出评分最高的文章，又可以用于取出最新发布的文章。这里特别要注意的一点是，因为有序集合会根据成员的分值从小到大地排列元素，所以使用 ZREVRANGE 命令，以“分值从大到小”的排列顺序取出文章 ID 才是正确的做法，代码清单 1-8 展示了文章获取功能的实现函数。

## 代码清单 1-8 get\_articles() 函数

```

ARTICLES_PER_PAGE = 25

def get_articles(conn, page, order='score:'):
    start = (page-1) * ARTICLES_PER_PAGE
    end = start + ARTICLES_PER_PAGE - 1
    ids = conn.zrevrange(order, start, end)
    articles = []
    for id in ids:
        article_data = conn.hgetall(id)
        article_data['id'] = id
        articles.append(article_data)
    ←—— 获取多个文章 ID。
    ↓ 根据文章 ID 获取文章的
       详细信息。
    ↑ 根据文章 ID 获取文章的详
       细信息。
    return articles

```

**Python 的默认值参数和关键字参数** 代码清单 1-8 中的 get\_articles() 函数为 order 参数设置了默认值 score:。Python 语言的初学者可能会对“默认值参数”以及“根据名字（而不是位置）来传入参数”的一些细节感到陌生。如果读者在理解函数定义或者参数传递方面有困难，可以考虑去

看看《Python 语言教程》，教程里面对这两个方面进行了很好的介绍，点击以下短链接就可以直接访问教程的相关章节：<http://mng.bz/KM5x>。

虽然我们构建的网站现在已经可以展示最新发布的文章和评分最高的文章了，但它还不具备目前很多投票网站都支持的群组（group）功能：这个功能可以让用户只看见与特定话题有关的文章，比如与“可爱的动物”有关的文章、与“政治”有关的文章、与“Java 编程”有关的文章或者介绍“Redis 用法”的文章等等。接下来的一节将向我们展示为文章投票网站添加群组功能的方法。

### 1.3.3 对文章进行分组

群组功能由两个部分组成，一个部分负责记录文章属于哪个群组，另一个部分负责取出群组里面的文章。为了记录各个群组都保存了哪些文章，网站需要为每个群组创建一个集合，并将所有同属一个群组的文章 ID 都记录到那个集合里面。代码清单 1-9 展示了将文章添加到群组里面的方法，以及从群组里面移除文章的方法。

代码清单 1-9 add\_remove\_groups() 函数

```
def add_remove_groups(conn, article_id, to_add=[], to_remove=[]):
    article = 'article:' + article_id
    for group in to_add:
        conn.sadd('group:' + group, article)           ← 构建存储文章信息的键名。
    for group in to_remove:
        conn.srem('group:' + group, article)           ← 将文章添加到它所属的群组里面。
    from群组里面移除文章。
```

初看上去，可能会有读者觉得使用集合来记录群组文章并没有多大用处。到目前为止，读者只看到了集合结构检查某个元素是否存在的能力，但实际上 Redis 不仅可以对多个集合执行操作，甚至在一些情况下，还可以在集合和有序集合之间执行操作。

为了能够根据评分对群组文章进行排序和分页（paging），网站需要将同一个群组里面的所有文章都按照评分有序地存储到一个有序集合里面。Redis 的 ZINTERSTORE 命令可以接受多个集合和多个有序集合作为输入，找出所有同时存在于集合和有序集合的成员，并以几种不同的方式来合并（combine）这些成员的分值（所有集合成员的分值都会被视为是 1）。对于我们的文章投票网站来说，程序需要使用 ZINTERSTORE 命令选出相同成员中最大的那个分值来作为交集成员的分值：取决于所使用的排序选项，这些分值既可以是文章的评分，也可以是文章的发布时间。

图 1-12 展示了对一个包含少量文章的群组集合和一个包含大量文章及评分的有序集合执行 ZINTERSTORE 命令的过程，注意观察那些同时出现在集合和有序集合里面的文章是怎样被添加到结果有序集合里面的。

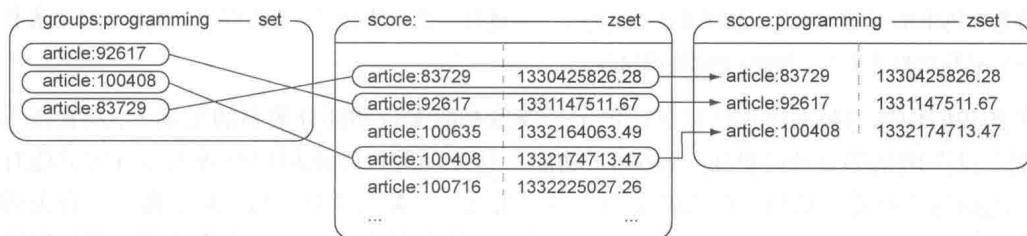


图 1-12 对集合 `groups:programming` 和有序集合 `score:` 进行交集计算得出了新的有序集合 `score:programming`，它包含了所有同时存在于集合 `groups:programming` 和有序集合 `score:` 的成员。因为集合 `groups:programming` 的所有成员的分值都被视为是 1，而有序集合 `score:` 的所有成员的分值都大于 1，并且这次交集计算挑选的分值为相同成员中的最大分值，所以有序集合 `score:programming` 的成员的分值实际上是由有序集合 `score:` 的成员的分值来决定的

通过对存储群组文章的集合和存储文章评分的有序集合执行 `ZINTERSTORE` 命令，程序可以得到按照文章评分排序的群组文章；而通过对存储群组文章的集合和存储文章发布时间的有序集合执行 `ZINTERSTORE` 命令，程序则可以得到按照文章发布时间排序的群组文章。如果群组包含的文章非常多，那么执行 `ZINTERSTORE` 命令就会比较耗时，为了尽量减少 Redis 的工作量，程序会将这个命令的计算结果缓存 60 秒。另外，我们还重用了已有的 `get_articles()` 函数来分页并获取群组文章，代码清单 1-10 展示了网站从群组里面获取一整页文章的方法。

#### 代码清单 1-10 `get_group_articles()` 函数

```
def get_group_articles(conn, group, page, order='score:'):
    key = order + group
    if not conn.exists(key):
        conn.zinterstore(key,
                          ['group:' + group, order],
                          aggregate='max')
        conn.expire(key, 60)
    return get_articles(conn, page, key)
```

根据评分或者发布时间，对群组文章进行排序。

为每个群组的每种排列顺序都创建一个键。

检查是否有已缓存的排序结果，如果没有的话就现在进行排序。

让 Redis 在 60 秒之后自动删除这个有序集合。

调用之前定义的 `get_articles()` 函数来进行分页并获取文章数据。

有些网站只允许用户将文章放在一个或者两个群组里面（其中一个是“所有文章”群组，另一个是最适合文章的群组）。在这种情况下，最好直接将文章所在的群组记录到存储文章信息的散列里面，并在 `article_vote()` 函数的末尾增加一个 `ZINCRBY` 命令调用，用于更新文章在群组中的评分。但是在这个示例里面，我们构建的文章投票网站允许一篇文章同时属于多个群组（比如一篇文章可以同时属于“编程”和“算法”两个群组），所以对于一篇同时属于多个群组的文章来说，更新文章的评分意味着程序需要对文章所属的全部群组执行自增操作。在这种情况下，如果一篇文章同时属于很多个群组，那么更新文章评分这一操作可能

会变得相当耗时，因此，我们在 `get_group_articles()` 函数里面对 `ZINTERSTORE` 命令的执行结果进行了缓存处理，以此来尽量减少 `ZINTERSTORE` 命令的执行次数。开发者在灵活性或限制条件之间的取舍将改变程序存储和更新数据的方式，这一点对于任何数据库都是适用的，Redis 也不例外。

### 练习：实现投反对票的功能

我们的示例目前只实现了投支持票的功能，但是在很多实际的网站里面，反对票也能给用户提供有用的反馈信息。因此，请读者能想办法在 `article_vote()` 函数和 `post_article()` 函数里面添加投反对票的功能。除此之外，读者还可以尝试为用户提供对调投票的功能：比如将支持票转换成反对票，或者将反对票转换成支持票。提示：如果读者在实现对调投票功能时出现了困难，可以参考一下第 3 章介绍的 `SMOVE` 命令。

好的，现在我们已经成功地构建起了一个展示最受欢迎文章的网站后端，这个网站可以获取文章、发布文章、对文章进行投票甚至还可以对文章进行分组。如果你觉得前面展示的内容不好理解，或者弄不懂这些示例，又或者没办法运行本书提供的源代码，那么请阅读下一节来了解如何获取帮助。

## 1.4 寻求帮助

当你遇到与 Redis 有关的问题时，不要害怕求助于别人，因为其他人可能也遇到过类似的问题。首先，你可以根据错误信息在搜索引擎里面进行查找，看是否有所发现。

如果搜索一无所获，又或者你遇到的问题与本书的示例代码有关，那么你可以到 Manning 出版社提供的论坛里面发问 (<http://www.manning-sandbox.com/forum.jspa?forumID=809>)，我和其他熟悉本书的人将为你提供帮助。

如果你遇到的问题与 Redis 本身有关，又或者你正在解决的问题在这本书里面没有出现过，那么你可以到 Redis 的邮件列表里面发问 (<https://groups.google.com/d/forum/redis-db/>)，同样地，我和其他熟悉 Redis 的人将为你提供帮助。

最后，如果你遇到的问题与某个函数库或者某种编程语言有关，那么比起在 Redis 邮件列表里面发帖提问，更好的方法是直接到你正在使用的那个函数库或者那种编程语言的邮件列表或论坛里面寻求帮助。

## 1.5 小结

本章对 Redis 进行了初步的介绍，说明了 Redis 与其他数据库的相同之处和不同之处，以及一些读者可能会使用 Redis 的理由。在阅读本书的后续章节之前，请记住本书的目标并不是构建

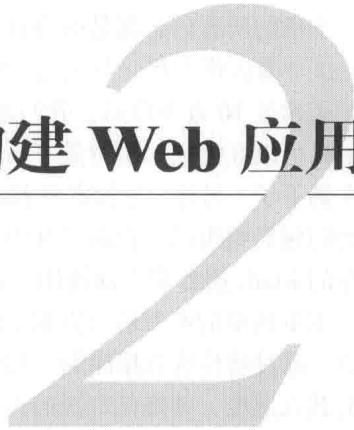
一个完整的应用或者工具，而是展示各式各样的问题，并给出使用 Redis 来解决这些问题的办法。

本章希望向读者传达这样一个概念：Redis 是一个可以用来解决问题的工具，它既拥有其他数据库不具备的数据结构，又拥有内存存储（这使得 Redis 的速度非常快）、远程（这使得 Redis 可以与多个客户端和服务器进行连接）、持久化（这使得服务器可以在重启之后仍然保持重启之前的数据）和可扩展（通过主从复制和分片）等多个特性，这使得用户可以以熟悉的方式为各种不同的问题构建解决方案。

在阅读本书的后续章节时，请读者注意自己解决问题的方式发生了什么变化：你也许会惊讶地发现，自己思考数据问题的方式已经从原来的“怎样将我的想法塞进数据库的表和行里面”，变成了“使用哪种 Redis 数据结构来解决这个问题比较好呢？”。

接下来的第 2 章将介绍使用 Redis 构建 Web 应用的方法，阅读这一章将帮助你更好地了解 Redis 的用法和用途。

# 第 2 章 使用 Redis 构建 Web 应用



## 本章主要内容

- 登录 cookie
- 购物车 cookie
- 缓存生成的网页
- 缓存数据库行
- 分析网页访问记录

前面的第 1 章对 Redis 的特性和功能做了简单的介绍，本章将紧接上一章的步伐，通过几个示例，对一些典型的 Web 应用进行介绍。尽管本章展示的问题比起实际情况要简单得多，但这里给出的网络应用实际上只需要进行少量修改就可以直接应用到真实的程序里面。本章的主要任务是作为一个实用指南，告诉你可以使用 Redis 来做些什么事情，而之后的第 3 章将对 Redis 命令进行更详细的介绍。

从高层次的角度来看，Web 应用就是通过 HTTP 协议对网页浏览器发送的请求进行响应的服务器或者服务（service）。一个 Web 服务器对请求进行响应的典型步骤如下。

- (1) 服务器对客户端发来的请求（request）进行解析。
- (2) 请求被转发给一个预定义的处理器（handler）。
- (3) 处理器可能会从数据库中取出数据。
- (4) 处理器根据取出的数据对模板（template）进行渲染（render）。
- (5) 处理器向客户端返回渲染后的内容作为对请求的响应（response）。

以上列举的 5 个步骤从高层次的角度展示了典型 Web 服务器的运作方式，这种情况下 Web 请求被认为是无状态的（stateless），也就是说，服务器本身不会记录与过往请求有关的任何信息，这使得失效（fail）的服务器可以很容易地被替换掉。有不少书籍专门介绍了如何优化响应过程的各个步骤，本书要做的事情也和它们类似，不同之处在于，本书讲解的是如何使用更快的 Redis 查询来代替传统的关系数据库查询，以及如何使用 Redis 来完成一些使用关系数据库没办法高效完成的任务。

本章的所有内容都是围绕着发现并解决 Fake Web Retailer 这个虚构的大型网上商店来展开的，这个商店每天都会有大约 500 万名不同的用户，这些用户会给网站带来 1 亿次点击，并从网站购买超过 10 万件商品。我们之所以将 Fake Web Retailer 的几个数据量设置得特别大，是考虑到如果可以在大数据量背景下顺利地解决问题，那么解决小数据量和中等数据量引发的问题就更不在话下了。另外，尽管本章展示的解决方案都是为了解决 Fake Web Retailer 这个大型网店所遇到的问题而给出的，但除了其中一个解决方案之外，其他所有解决方案都可以在一个只有几 GB 内存的 Redis 服务器上面使用，并且这些解决方案的目标都在于提高系统响应实时请求的性能。

本章列举的所有解决方案（以及它们的一些变种）都在生产环境中实际使用过。说得更具体一点，通过将传统数据库的一部分数据处理任务以及存储任务转交给 Redis 来完成，可以提升网页的载入速度，并降低资源的占用量。

我们要解决的第一个问题就是使用 Redis 来管理用户登录会话（session）。

## 2.1 登录和 cookie 缓存

每当我们登录互联网服务（比如银行账户或者电子邮件）的时候，这些服务都会使用 cookie 来记录我们的身份。cookie 由少量数据组成，网站会要求我们的浏览器存储这些数据，并在每次服务发送请求时将这些数据传回给服务。对于用来登录的 cookie，有两种常见的方法可以将登录信息存储在 cookie 里面：一种是签名（signed）cookie，另一种是令牌（token）cookie。

签名 cookie 通常会存储用户名，可能还有用户 ID、用户最后一次成功登录的时间，以及网站觉得有用的其他任何信息。除了用户的相关信息之外，签名 cookie 还包含一个签名，服务器可以使用这个签名来验证浏览器发送的信息是否未经改动（比如将 cookie 中的登录用户名改成另一个用户）。

令牌 cookie 会在 cookie 里面存储一串随机字节作为令牌，服务器可以根据令牌在数据库中查找令牌的拥有者。随着时间的推移，旧令牌会被新令牌取代。表 2-1 展示了签名 cookie 和令牌 cookie 的优点与缺点。

表 2-1 签名 cookie 和令牌 cookie 的优点与缺点

cookie 类型	优点	缺点
签名 cookie	验证 cookie 所需的一切信息都存储在 cookie 里面。cookie 可以包含额外的信息（additional information），并且对这些信息进行签名也很容易	正确地处理签名很难。很容易忘记对数据进行签名，或者忘记验证数据的签名，从而造成安全漏洞
令牌 cookie	添加信息非常容易。cookie 的体积非常小，因此移动终端和速度较慢的客户端可以更快地发送请求	需要在服务器中存储更多信息。如果使用的是关系数据库，那么载入和存储 cookie 的代价可能会很高

因为 Fake Web Retailer 没有实现签名 cookie 的需求，所以我们选择了使用令牌 cookie 来引

用关系数据库表中负责存储用户登录信息的条目（entry）。除了用户登录信息之外，Fake Web Retailer 还可以将用户的访问时长和已浏览商品的数量等信息存储到数据库里面，这样便于将来通过分析这些信息来学习如何更好地向用户推销商品。

一般来说，用户在决定购买某个或某些商品之前，通常都会先浏览多个不同的商品，而记录用户浏览过的所有商品以及用户最后一次访问页面的时间等信息，通常会导致大量的数据库写入。从长远来看，用户的这些浏览数据的确非常有用，但问题在于，即使经过优化，大多数关系数据库在每台数据库服务器上面每秒也只能插入、更新或者删除 200~2000 个数据库行。尽管批量插入、批量更新和批量删除等操作可以以更快的速度执行，但因为客户端每次浏览网页都只更新少数几个行，所以高速的批量插入在这里并不适用。

因为 Fake Web Retailer 目前一天的负载量相对比较大——平均情况下每秒大约 1200 次写入，高峰时期每秒接近 6000 次写入，所以它必须部署 10 台关系数据库服务器才能应对高峰时期的负载量。而我们要做的就是使用 Redis 重新实现登录 cookie 功能，取代目前由关系数据库实现的登录 cookie 功能。

首先，我们将使用一个散列来存储登录 cookie 令牌与已登录用户之间的映射。要检查一个用户是否已经登录，需要根据给定的令牌来查找与之对应的用户，并在用户已经登录的情况下，返回该用户的 ID。代码清单 2-1 展示了检查登录 cookie 的方法。

### 代码清单 2-1 check\_token() 函数

```
def check_token(conn, token):
    return conn.hget('login:', token)
```

尝试获取并返回令牌  
对应的用户。

对令牌进行检查并不困难，因为大部分复杂的工作都是在更新令牌时完成的：用户每次浏览页面的时候，程序都会对用户存储在登录散列里面的信息进行更新，并将用户的令牌和当前时间戳添加到记录最近登录用户的有序集合里面；如果用户正在浏览的是一个商品页面，那么程序还会将这个商品添加到记录这个用户最近浏览过的商品的有序集合里面，并在被记录商品的数量超过 25 个时，对这个有序集合进行修剪。代码清单 2-2 展示了程序更新令牌的方法。

### 代码清单 2-2 update\_token() 函数

```
def update_token(conn, token, user, item=None):
    timestamp = time.time()
    conn.hset('login:', token, user)
    conn.zadd('recent:', token, timestamp)
    if item:
        conn.zadd('viewed:' + token, item, timestamp)
        conn.zremrangebyrank('viewed:' + token, 0, -26)
```

记录用户  
浏览过的  
商品。

移除旧的记录，只保留用户  
最近浏览过的 25 个商品。

获取当前时间戳。

维持令牌与已  
登录用户之间  
的映射。

记录令牌最后  
一次出现的时间。

通过 `update_token()` 函数，我们可以记录用户最后一次浏览商品的时间以及用户最近浏览了哪些商品。在一台最近几年生产的服务器上面，使用 `update_token()` 函数每秒至少可以记录 20 000 件商品，这比 Fake Web Retailer 高峰时期所需的 6000 次写入要高 3 倍有余。不仅如此，通过后面介绍的一些方法，我们还可以进一步优化 `update_token()` 函数的运行速度。但即使是现在这个版本的 `update_token()` 函数，比起原来的关系数据库，性能也已经提升了 10~100 倍。

因为存储会话数据所需的内存会随着时间的推移而不断增加，所以我们需要定期清理旧的会话数据。为了限制会话数据的数量，我们决定只保存最新的 1000 万个会话。<sup>①</sup> 清理旧会话的程序由一个循环构成，这个循环每次执行的时候，都会检查存储最近登录令牌的有序集合的大小，如果有有序集合的大小超过了限制，那么程序就会从有序集合里面移除最多 100 个最旧的令牌，并从记录用户登录信息的散列里面，移除被删除令牌对应的用户的信息，并对存储了这些用户最近浏览商品记录的有序集合进行清理。与此相反，如果令牌的数量未超过限制，那么程序会先休眠 1 秒，之后再重新进行检查。代码清单 2-3 展示了清理旧会话程序的具体代码。

代码清单 2-3 `clean_sessions()` 函数

```
QUIT = False
LIMIT = 10000000

def clean_sessions(conn):
    while not QUIT:
        size = conn.zcard('recent:')           | 找出目前已有令牌
                                                | 的数量。
        if size <= LIMIT:                   | 令牌数量未超过限制，休眠
            time.sleep(1)                  | 并在之后重新检查。
            continue

        end_index = min(size - LIMIT, 100)    | 获取需要移除的令牌
        tokens = conn.zrange('recent:', 0, end_index-1) | ID。
                                                | 为那些将要被删除的令牌
                                                | 构建键名。

        session_keys = []
        for token in tokens:
            session_keys.append('viewed:' + token)

        conn.delete(*session_keys)
        conn.hdel('login:', *tokens)
        conn.zrem('recent:', *tokens)          | 移除最旧的那些令牌。
```

让我们通过计算来了解一下，这段简单的代码为什么能够妥善地处理每天 500 万人次的访问：假设网站每天有 500 万用户访问，并且每天的用户都和之前的不一样，那么只需要两天，令牌的数量就会达到 1000 万个的上限，并将网站的内存空间消耗殆尽。因为一天有  $24 \times 3600 = 86400$  秒，而网站平均每秒产生  $5000000 / 86400 < 58$  个新会话，如果清理函数和

<sup>①</sup> 因为 Fake Web Retailer 这个示例假设的是生产环境，所以保存会话的数量会设置得比较高，在测试或者开发这个程序的时候，读者可以按照自己的需要调低这个值。

我们之前在代码里面定义的一样，以每秒一次的频率运行的话，那么它每秒需要清理将近 60 个令牌，才能防止令牌数量过多的问题发生。但是实际上，我们定义的令牌清理函数在通过网络来运行时，每秒能够清理 10 000 多个令牌，在本地运行时，每秒能够清理 60 000 多个令牌，这比所需的清理速度快了 150~1000 倍，所以因为旧令牌过多而导致网站空间耗尽的问题不会出现。

**在哪里执行清理函数？** 本书会包含一些类似代码清单 2-3 的清理函数，它们可能会像代码清单 2-3 那样，以守护进程的方式来运行，也可能会作为定期作业（cron job）每隔一段时间运行一次，甚至在每次执行某个操作时运行一次（例如，6.3 节就在一个获取锁操作里面包含了一个清理操作）。一般来说，本书中包含 `while not QUIT`: 代码的函数都应该作为守护进程来执行，不过如果有需要的话，也可以把它们改成周期性地运行。

**Python 传递和接收可变数量参数的语法** 代码清单 2-3 用到了 3 次类似 `conn.delete (*vtokens)` 这样的语法。简单来说，这种语法可以直接将一连串的多个参数传入函数里面，而不必先对这些参数进行解包（unpack）。要了解关于这一语法的更多信息，请通过以下短链接访问《Python 语言教程》的相关章节：<http://mng.bz/8I7W>。

**Redis 的过期数据处理** 随着对 Redis 的了解逐渐加深，读者应该会慢慢发现本书展示的一些解决方案有时候并不是问题的唯一解决办法。比如对于这个登录 cookie 例子来说，我们可以直接将登录用户和令牌的信息存储到字符串键值对里面，然后使用 Redis 的 EXPIRE 命令，为这个字符串和记录用户商品浏览记录的有序集合设置过期时间，让 Redis 在一段时间之后自动删除它们，这样就不再需要再使用有序集合来记录最近出现的令牌了。但是这样一来，我们就没有办法将会话的数量限制在 1000 万之内了，并且在将来有需要的时候，我们也没办法在会话过期之后对被废弃的购物车进行分析了。

熟悉多线程编程或者并发编程的读者可能会发现代码清单 2-3 展示的清理函数实际上包含一个竞争条件（race condition）：如果清理函数正在删除某个用户的信息，而这个用户又在同一时间访问网站的话，那么竞争条件就会导致用户的信息被错误地删除。目前来看，这个竞争条件除了会使得用户需要重新登录一次之外，并不会对程序记录的数据产生明显的影响，所以我们暂时先搁置这个问题，之后的第 3 章和第 4 章会说明怎样防止类似的竞争条件发生，并进一步加快清理函数的执行速度。

通过使用 Redis 来记录用户信息，我们成功地将每天要对数据库执行的行写入操作减少了数百万次。虽然这非常的了不起，但这只是我们使用 Redis 构建 Web 应用程序的第一步，接下来的一节将向读者们展示如何使用 Redis 来处理另一种类型的 cookie。

## 2.2 使用Redis实现购物车

网景(Netscape)公司在20世纪90年代中期最先在网络中使用了cookie，这些cookie最终变成了我们在上一节讨论的登录会话cookie。cookie最初的意图在于为网络零售商(web retailer)提供一种购物车，让用户可以收集他们想要购买的商品。在cookie之前，有过几种不同的购物车解决方案，但这些方案全都不太好用。

使用cookie实现购物车——也就是将整个购物车都存储到cookie里面的做法非常常见，这种做法的一大优点是无须对数据库进行写入就可以实现购物车功能，而缺点则是程序需要重新解析和验证(validate)cookie，确保cookie的格式正确，并且包含的商品都是真正可购买的商品。cookie购物车还有一个缺点：因为浏览器每次发送请求都会连cookie一起发送，所以如果购物车cookie的体积比较大，那么请求发送和处理的速度可能会有所降低。

因为我们在前面已经使用Redis实现了会话cookie和记录用户最近浏览过的商品这两个特性，所以我们决定将购物车的信息也存储到Redis里面，并且使用与用户会话cookie相同的cookie ID来引用购物车。

购物车的定义非常简单：每个用户的购物车都是一个散列，这个散列存储了商品ID与商品订购数量之间的映射。对商品数量进行验证的工作由Web应用程序负责，我们要做的则是在商品的订购数量出现变化时，对购物车进行更新：如果用户订购某件商品的数量大于0，那么程序会将这件商品的ID以及用户订购该商品的数量添加到散列里面，如果用户购买的商品已经存在于散列里面，那么新的订购数量会覆盖已有的订购数量；相反地，如果用户订购某件商品的数量不大于0，那么程序将从散列里面移除该条目。代码清单2-4的add\_to\_cart()函数展示了程序是如何更新购物车的。

代码清单2-4 add\_to\_cart()函数

```
def add_to_cart(conn, session, item, count):
    if count <= 0:
        conn.hrem('cart:' + session, item)           ← 从购物车里面移除
    else:                                         指定的商品。
        conn.hset('cart:' + session, item, count)    ← 将指定的商品添加到购物车。
```

接着，我们需要对之前的会话清理函数进行更新，让它在清理旧会话的同时，将旧会话对应用户的购物车也一并删除，更新后的函数如代码清单2-5所示。

代码清单2-5 clean\_full\_sessions()函数

```
def clean_full_sessions(conn):
    while not QUIT:
        size = conn.zcard('recent:')
        if size <= LIMIT:
            time.sleep(1)
            continue
```

```
end_index = min(size - LIMIT, 100)
sessions = conn.zrange('recent:', 0, end_index-1)

session_keys = []
for sess in sessions:
    session_keys.append('viewed:' + sess)
    session_keys.append('cart:' + sess)

conn.delete(*session_keys)
conn.hdel('login:', *sessions)
conn.zrem('recent:', *sessions)
```

新增加的这行代码  
用于删除旧会话  
应用用户的购物车。

我们现在将会话和购物车都存储到了 Redis 里面，这种做法除了可以减少请求的体积之外，还使得我们可以根据用户浏览过的商品、用户放入购物车的商品以及用户最终购买的商品进行统计计算，并构建起很多大型网络零售商都在提供的“在查看过这件商品的用户当中，有 X% 的用户最终购买了这件商品”“购买了这件商品的用户也购买了某某其他商品”等功能，这些功能可以帮助用户查找其他相关的商品，并最终提升网站的销售业绩。

通过将会话 cookie 和购物车 cookie 存储在 Redis 里面，我们得到了进行数据分析所需的两个重要的数据来源，接下来的一节将展示如何使用缓存来减少数据库和 Web 前端的负载。

## 2.3 网页缓存

在动态生成网页的时候，通常会使用模板语言（templating language）来简化网页的生成操作。需要手写每个页面的日子已经一去不复返——现在的 Web 页面通常由包含首部、尾部、侧栏菜单、工具条、内容域的模板生成，有时候模板还用于生成 JavaScript。

尽管 Fake Web Retailer 也能够动态地生成内容，但这个网站上的多数页面实际上并不会经常发生大的变化：虽然会向分类中添加新商品、移除旧商品、有时有特价促销、有时甚至还有“热卖商品”页面，但是在一般情况下，网站只有账号设置、以往订单、购物车（结账信息）以及其他少数几个页面才包含需要每次载入都要动态生成的内容。

通过对浏览数据进行分析，Fake Web Retailer 发现自己所处理的 95% 的 Web 页面每天最多只会改变一次，这些页面的内容实际上并不需要动态地生成，而我们的工作就是想办法不再生成这些页面。减少网站在动态生成内容上面所花的时间，可以降低网站处理相同负载所需的服务器数量，并让网站的速度变得更快。（研究表明，减少用户等待页面载入的时间，可以增加用户使用网站的欲望，并改善用户对网站的印象。）

所有标准的 Python 应用框架都提供了在处理请求之前或者之后添加层（layer）的能力，这些层通常被称为中间件（middleware）或者插件（plugin）。我们将创建一个这样的层来调用 Redis 缓存函数：对于一个不能被缓存的请求，函数将直接生成并返回页面；而对于可以被缓存的请求，函数首先会尝试从缓存里面取出并返回被缓存的页面，如果缓存页面不存在，那么函数会生成页面并将其缓存在 Redis 里面 5 分钟，最后再将页面返回给函数调用者。代码清单 2-6 展示了这个缓存函数。

## 代码清单2-6 cache\_request()函数

```

对于不能被缓存的请求，直
接调用回调函数。
def cache_request(conn, request, callback):
    if not can_cache(conn, request):
        return callback(request)

    page_key = 'cache:' + hash_request(request)
    content = conn.get(page_key)

    if not content:
        content = callback(request)
        conn.setex(page_key, content, 300)

    return content
    
```

返回页面。

对于Fake Web Retailer网站上面95%的可被缓存并且频繁被载入的内容来说，代码清单2-6展示的缓存函数可以让网站在5分钟之内无需再为它们动态地生成视图页面。取决于网页的内容有多复杂，这一改动可以将包含大量数据的页面的延迟值从20~50毫秒降低至查询一次Redis所需的时间：查询本地Redis的延迟值通常低于1毫秒，而查询位于同一个数据中心的Redis的延迟值通常低于5毫秒。对于那些需要访问数据库的页面来说，这个缓存函数对于减少页面载入时间和降低数据库负载的作用会更加显著。

在这一节中，我们学习了如何使用Redis来减少载入不常改变页面所需的时间，那么对于那些经常发生变化的页面，我们是否也能够使用Redis来减少它们的载入时间呢？答案是肯定的，接下来的一节将介绍实现这一目标的具体做法。

## 2.4 数据行缓存

到目前为止，我们已经将原本由关系数据库和网页浏览器实现的登录和访客会话转移到了Redis上面实现；将原本由关系数据库实现的购物车也放到了Redis上面实现；还将所有页面缓存到了Redis里面。这一系列工作提升了网站的性能，降低了关系数据库的负载并减少了网站成本。

Fake Web Retailer的商品页面通常只会从数据库里面载入一两行数据，包括已登录用户的用户信息（这些信息可以通过AJAX动态地载入，所以不会对页面缓存造成影响）和商品本身的信息。即使是那些无法被整个缓存起来的页面——比如用户账号页面、记录用户以往购买商品的页面等等，程序也可以通过缓存页面载入时所需的数据库行来减少载入页面所需的时间。

为了展示数据行缓存的作用，我们假设Fake Web Retailer为了清空旧库存和吸引客户消费，决定开始新一轮的促销活动：这个活动每天都会推出一些特价商品供用户抢购，所有特价商品的数量都是限定的，卖完即止。在这种情况下，网站是不能对整个促销页面进行缓存的，因为这可能会导致用户看到错误的特价商品剩余数量，但是每次载入页面都从数据库里面取出特价商品的

剩余数量的话，又会给数据库带来巨大的压力，并导致我们需要花费额外的成本来扩展数据库。

为了应对促销活动带来的大量负载，我们需要对数据行进行缓存，具体的做法是：编写一个持续运行的守护进程函数，让这个函数将指定的数据行缓存到 Redis 里面，并不定期地对这些缓存进行更新。缓存函数会将数据行编码（encode）为 JSON 字典并存储在 Redis 的字符串里面，其中，数据列（column）的名字会被映射为 JSON 字典的键，而数据行的值则会被映射为 JSON 字典的值，图 2-1 展示了一个被缓存的数据行示例。

程序使用了两个有序集合来记录应该在何时对缓存进行更新：第一个有序集合为调度（schedule）有序集合，它的成员为数据行的行 ID，而分值则是一个时间戳，这个时间戳记录了应该在何时将指定的数据行缓存到 Redis 里面；第二个有序集合为延时（delay）有序集合，它的成员也是数据行的行 ID，而分值则记录了指定数据行的缓存需要每隔多少秒更新一次。

**使用 JSON 而不是其他格式** 因为 JSON 简明易懂，并且据我们所知，目前所有拥有 Redis 客户端的编程语言都带有能够高效地编码和解码 JSON 格式的函数库，所以这里的缓存函数使用了 JSON 格式来表示数据行，而没有使用 XML、Google 的 protocol buffer、Thrift、BSON、MessagePack 或者其他序列化格式。在实际应用中，读者可以根据自己的需求和喜好来选择编码数据行的格式。

**嵌套多个结构** 使用过其他非关系数据库的用户可能会期望 Redis 也拥有嵌套多个结构的能力，比如说，一个刚开始使用 Redis 的用户可能会期盼着散列能够包含有序集合值或者列表值。尽管嵌套结构这个特性在概念上并无不妥，但这个特性很快就会引起类似以下这样的问题：“对于一个位于嵌套第 5 层的散列，我们如何才能对它的值执行自增操作呢？”为了保证命令语法的简单性，Redis 并不支持嵌套结构特性。如果有需要的话，读者可以通过使用键名来模拟嵌套结构特性：比如使用键 user:123 表示存储用户信息的散列，并使用键 user:123:posts 表示存储用户最近发表文章的有序集合；又或者直接将嵌套结构存储到 JSON 或者其他序列化格式里面（第 11 章将介绍使用 Lua 脚本在服务器端直接以 JSON 格式或者 MessagePack 格式对数据进行编码的方法）。

为了让缓存函数定期地缓存数据行，程序首先需要将行 ID 和给定的延迟值添加到延迟有序集合里面，然后再将行 ID 和当前时间的时间戳添加到调度有序集合里面。实际执行缓存操作的函数需要用到数据行的延迟值，如果某个数据行的延迟值不存在，那么程序将取消对这个数据行的调度。如果我们想要移除某个数据行已有的缓存，并且让缓存函数不再缓存那个数据行，

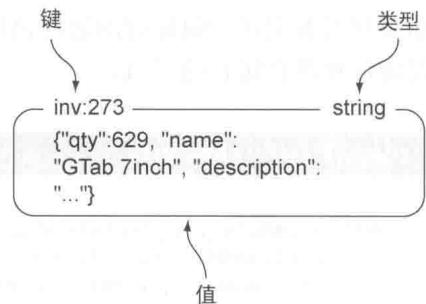


图 2-1 一个被缓存的数据行，这个数据行包含了在线售卖商品的信息

那么只需要把那个数据行的延迟值设置为小于或等于0就可以了。代码清单2-7展示了负责调度缓存和终止缓存的函数。

#### 代码清单2-7 schedule\_row\_cache()函数

```
def schedule_row_cache(conn, row_id, delay):
    conn.zadd('delay:', row_id, delay)           ← 先设置数据行的
    conn.zadd('schedule:', row_id, time.time())   ← 延迟值。
                                                ← 立即对需要缓存的数据行进行调度。
```

现在我们已经完成了调度部分，那么接下来该如何对数据行进行缓存呢？负责缓存数据行的函数会尝试读取调度有序集合的第一个元素以及该元素的分值，如果调度有序集合没有包含任何元素，或者分值存储的时间戳所指定的时间尚未来临，那么函数会先休眠50毫秒，然后再重新进行检查。当缓存函数发现一个需要立即进行更新的数据行时，缓存函数会检查这个数据行的延迟值：如果数据行的延迟值小于或者等于0，那么缓存函数会从延迟有序集合和调度有序集合里面移除这个数据行的ID，并从缓存里面删除这个数据行已有的缓存，然后再重新进行检查；对于延迟值大于0的数据行来说，缓存函数会从数据库里面取出这些行，将它们编码为JSON格式并存储到Redis里面，然后更新这些行的调度时间。执行以上工作的缓存函数如代码清单2-8所示。

#### 代码清单2-8 守护进程函数cache\_rows()

```
def cache_rows(conn):
    while not QUIT:
        next = conn.zrange('schedule:', 0, 0, withscores=True)
        now = time.time()
        if not next or next[0][1] > now:
            time.sleep(.05)           ← 尝试获取下一个需要被缓存的数据行
            continue                  ← 以及该行的调度时间戳，命令会返回一个包含零个或一个元组(tuple)的列表。
        row_id = next[0][0]          ← 暂时没有行需要被缓存，休眠50毫秒后重试。
        delay = conn.zscore('delay:', row_id)           ← 提前获取下一次调度的延迟时间。
        if delay <= 0:
            conn.zrem('delay:', row_id)                 ← 不必再缓存这个行，将它从
            conn.zrem('schedule:', row_id)               ← 缓存中移除。
            conn.delete('inv:' + row_id)
            continue
        row = Inventory.get(row_id)                     ← 读取数据行。
        conn.zadd('schedule:', row_id, now + delay)
        conn.set('inv:' + row_id, json.dumps(row.to_dict()))
```

更新调度时间  
并设置缓存值。

通过组合使用调度函数和持续运行缓存函数，我们实现了一种重复进行调度的自动缓存机制，并且可以随心所欲地控制数据行缓存的更新频率：如果数据行记录的是特价促销商品的剩余数量，并且参与促销活动的用户非常多的话，那么我们最好每隔几秒更新一次数据行缓存；另一方面，如

果数据并不经常改变，或者商品缺货是可以接受的，那么我们可以每分钟更新一次缓存。

在这一节中，我们学习了如何将数据行缓存到 Redis 里面，在接下来的一节中，我们将通过只缓存一部分页面来减少实现页面缓存所需的内存数量。

## 2.5 网页分析

网站可以从用户的访问、交互和购买行为中收集到有价值的信息。例如，如果我们只想关注那些浏览量最高的页面，那么我们可以尝试修改页面的格局、配色甚至是页面上展示的其他链接。每一个修改尝试都能改变用户对一个页面或者后续页面的体验，或好或坏，甚至还能影响用户的购买行为。

前面的 2.1 节和 2.2 节中介绍了如何记录用户浏览过的商品或者用户添加到购物车中的商品，2.3 节中则介绍了如何通过缓存 Web 页面来减少页面载入时间并提升页面的响应速度。不过遗憾的是，我们对 Fake Web Retailer 采取的缓存措施做得过了火：Fake Web Retailer 总共包含 100 000 件商品，而冒然地缓存所有商品页面将耗尽整个网站的全部内存！经过一番调研之后，我们决定只对其中 10 000 件商品的页面进行缓存。

前面的 2.1 节中曾经介绍过，每个用户都有一个相应的记录用户浏览商品历史的有序集合，尽管使用这些有序集合可以计算出用户最经常浏览的商品，但进行这种计算却需要耗费大量的时间。为了解决这个问题，我们决定在 `update_token()` 函数里面添加一行代码，如代码清单 2-2 所示。

代码清单 2-9 修改后的 `update_token()` 函数

```
def update_token(conn, token, user, item=None):
    timestamp = time.time()
    conn.hset('login:', token, user)
    conn.zadd('recent:', token, timestamp)

    if item:
        conn.zadd('viewed:' + token, item, timestamp)
        conn.zremrangebyrank('viewed:' + token, 0, -26)
        conn.zincrby('viewed:', item, -1)
```

这行代码是新  
添加的。

新添加的代码记录了所有商品的浏览次数，并根据浏览次数对商品进行了排序，被浏览得最多的商品将被放到有序集合的索引 0 位置上，并且具有整个有序集合最少的分值。随着时间的流逝，商品的浏览次数会呈现两极分化的状态，一些商品的浏览次数会越来越多，而另一些商品的浏览次数则会越来越少。除了缓存最常被浏览的商品之外，程序还需要发现那些变得越来越流行的新商品，并在合适的时候缓存它们。

为了让商品浏览次数排行榜能够保持最新，我们需要定期修剪有序集合的长度并调整已有元素的分值，从而使得新流行的商品也可以在排行榜里面占据一席之地。之前的 2.1 节中已经介绍过从有序集合里面移除元素的方法，而调整元素分值的动作则可以通过 `ZINTERSTORE` 命令来完成。`ZINTERSTORE` 命令可以组合起一个或多个有序集合，并将有序集合包含的每个分值都乘以

一个给定的数值（用户可以为每个有序集合分别指定不同的相乘数值）。每隔5分钟，代码清单2-10展示的函数就会删除所有排名在20 000名之后的商品，并将删除之后剩余的所有商品的浏览次数减半。

#### 代码清单2-10 守护进程函数rescale\_viewed()

```
def rescale_viewed(conn):
    while not QUIT:
        conn.zremrangebyrank('viewed:', 0, -20001) ←
        conn.zinterstore('viewed:', {'viewed': .5})
        time.sleep(300) ←
    
```

通过记录商品的浏览次数，并定期对记录浏览次数的有序集合进行修剪和分值调整，我们为Fake Web Retailer建立起了一个持续更新的最常浏览商品排行榜。接下来要做的就是修改之前介绍过的can\_cache()函数，让它使用新的方法来判断页面是否需要被缓存，如代码清单2-11所示。

#### 代码清单2-11 can\_cache()函数

```
def can_cache(conn, request):
    item_id = extract_item_id(request) ←
    if not item_id or is_dynamic(request): ←
        return False
    rank = conn.zrank('viewed:', item_id) ←
    return rank is not None and rank < 10000 ←

```

通过使用前面介绍的几个函数，Fake Web Retailer现在可以统计商品被浏览的次数，并以此来缓存用户最经常浏览的10 000个商品页面。如果我们想以最少的代价来存储更多页面，那么可以考虑先对页面进行压缩，然后再缓存到Redis里面；或者使用Edge Side Includes技术移除页面中的部分内容；又或者对模板进行提前优化（pre-optimize），移除所有非必要的空格字符。这些技术能够减少内存消耗并增加Redis能够缓存的页面数量，为访问量不断增长的网站带来额外的性能提升。

## 2.6 小结

本章介绍了几种用于降低Fake Web Retailer的数据库负载和Web服务器负载的方法，这些例子里面介绍的都是真实的Web应用程序当今正在使用的思路和方法。

本章希望向读者传达这样一个概念：在为应用程序创建新构件时，不要害怕回过头去重构已

有的构件，因为就像本章展示的购物车 cookie 的例子和基于登录会话 cookie 实现网页分析的例子一样，已有的构件有时候需要进行一些细微的修改才能真正满足你的需求。本书之后的章节也会继续引入新的主题，并且偶尔会回过头去审视之前介绍过的主题，对它们的功能或者性能进行改进，又或者重用之前已经介绍过的思路。

本章向读者介绍了怎样使用 Redis 来构建真实的应用程序组件，下一章将向读者介绍 Redis 提供的各种命令：通过更深入地了解 Redis 提供的各种结构以及这些结构的作用，读者将掌握到构建更复杂也更有用的组件所需的知识。不要犹豫，赶快阅读下一章吧！



## 第二部分

# 核心概念

这一部分的前面几章将深入探讨标准的 Redis 命令，其中包括数据操作命令和配置命令，而后面的几章将展示如何使用 Redis 构建更为复杂的辅助工具和应用程序，并在最后使用 Redis 来构建一个简单的社交网站。

# 第3章 Redis 命令

## 本章主要内容

- 字符串命令、列表命令和集合命令
- 散列命令和有序集合命令
- 发布命令与订阅命令
- 其他命令

本章将介绍一些没有在第1章和第2章出现过的Redis命令，学习这些命令有助于读者在已有示例的基础上构建更为复杂的程序，并学会如何更好地去解决自己遇到的问题。本章将使用客户端与Redis服务器进行简单的互动，并以此来介绍命令的用法，如果读者想要看一些更为具体的代码示例，那么可以阅读第2章。

根据结构或者概念的不同，本章将多个命令分别放到了多个不同的节里面进行介绍，并且这里展示的命令都是各种应用程序最经常会用到的。和第1章介绍各个结构时的做法类似，本章也是通过与客户端进行互动的方式来介绍各个命令，在有需要的时候，文中还会说明本书在哪些章节用到了正在介绍的命令。

在每个不同的数据类型的章节里，展示的都是该数据类型所独有的、最具代表性的命令。首先让我们来看看，除了GET和SET之外，Redis的字符串还支持哪些命令。

**查阅本章未介绍命令的文档** 本章只会介绍最常用的Redis命令或者本书后续章节会用到的命令，如果读者需要一份完整的命令文档作为参考，那么可以访问<http://redis.io/commands>。

**Redis 2.4 和 Redis 2.6** 正如附录A所说，在本书编写之际，Windows平台上面只有Redis 2.4可用，而本书却会用到只有Redis 2.6或以上版本才支持的特性。Redis 2.4 和 Redis 2.6 之间的主要区别包括(但不限于)Lua脚本(将在第11章介绍)、毫秒精度的过期操作(相关的PTTL命令、PEXPIRE命令和PEXPIREAT命令将在本章介绍)、一些二进制位操作(BITOP命令和BITCOUNT命令)，另外还有一些在Redis 2.6以前只能接受单个参数的命令，比如RPUSH、LPUSH、SADD、SRM、HDEL、ZADD和ZREM，从Redis 2.6开始都可以接受多个参数了。

## 3.1 字符串

本书在第 1 章和第 2 章曾经说过, Redis 的字符串就是一个由字节组成的序列, 它们和很多编程语言里面的字符串没有什么明显的不同, 跟 C 或者 C++ 风格的字符数组也相去不远。在 Redis 里面, 字符串可以存储以下 3 种类型的值。

- 字节串 (byte string)。
- 整数。
- 浮点数。

用户可以通过给定一个任意的数值, 对存储着整数或者浮点数的字符串执行自增 (increment) 或者自减 (decrement) 操作, 在有需要的时候, Redis 还会将整数转换成浮点数。整数的取值范围和系统的长整数 (long integer) 的取值范围相同 (在 32 位系统上, 整数就是 32 位有符号整数, 在 64 位系统上, 整数就是 64 位有符号整数), 而浮点数的取值范围和精度则与 IEEE 754 标准的双精度浮点数 (double) 相同。Redis 明确地区分字符串、整数和浮点数的做法是一种优势, 比起只能够存储字符串的做法, Redis 的做法在数据表现方面具有更大的灵活性。

本节将对 Redis 里面最简单的结构——字符串进行讨论, 介绍基本的数值自增和自减操作, 以及二进制位 (bit) 和子串 (substring) 处理命令, 读者可能会惊讶地发现, Redis 里面最简单的结构居然也有如此强大的作用。

表 3-1 展示了对 Redis 字符串执行自增和自减操作的命令。

表 3-1 Redis 中的自增命令和自减命令

命令	用例和描述
INCR	INCR key-name——将键存储的值加上 1
DECR	DECR key-name——将键存储的值减去 1
INCRBY	INCRBY key-name amount——将键存储的值加上整数 amount
DECRBY	DECRBY key-name amount——将键存储的值减去整数 amount
INCRBYFLOAT	INCRBYFLOAT key-name amount——将键存储的值加上浮点数 amount, 这个命令在 Redis 2.6 或以上的版本可用

当用户将一个值存储到 Redis 字符串里面的时候, 如果这个值可以被解释 (interpret) 为十进制整数或者浮点数, 那么 Redis 会察觉到这一点, 并允许用户对这个字符串执行各种 INCR\* 和 DECR\* 操作。如果用户对一个不存在的键或者一个保存了空串的键执行自增或者自减操作, 那么 Redis 在执行操作时会将这个键的值当作是 0 来处理。如果用户尝试对一个值无法被解释为整数或者浮点数的字符串键执行自增或者自减操作, 那么 Redis 将向用户返回一个错误。代码清单 3-1 展示了对字符串执行自增操作和自减操作的一些例子。

## 代码清单3-1 这个交互示例展示了Redis的INCR操作和DECR操作

和自增操作一样，执行自减操作的函数也可以通过可选的参数来指定减量。

在尝试获取一个键的时候，命令将以字符串格式返回被存储的整数。

```
>>> conn = redis.Redis()
>>> conn.get('key')
>>> conn.incr('key')
1
>>> conn.incr('key', 15)
16
>>> conn.decr('key', 5)
11
>>> conn.get('key')
'11'
>>> conn.set('key', '13')
True
>>> conn.incr('key')
14
```

尝试获取一个不存在的键将得到一个None值，终端不会显示这个值。

我们既可以对不存在的键执行自增操作，也可以通过可选的参数来指定自增操作的增量。

即使在设置键时输入的值为字符串，但只要这个值可以被解释为整数，我们就可以把它当作整数来处理。

在读完本书其他章节之后，读者可能会发现本书只调用了`incr()`，这是因为Python的Redis库在内部使用`INCRBY`命令来实现`incr()`方法，并且这个方法的第二个参数是可选的：如果用户没有为这个可选参数设置值，那么这个参数就会使用默认值1。在编写本书的时候，Python的Redis客户端库支持Redis 2.6的所有命令，这个库通过`incrbyfloat()`方法来实现`INCRBYFLOAT`命令，并且`incrbyfloat()`方法也有类似于`incr()`方法的可选参数特性。

除了自增操作和自减操作之外，Redis还拥有对字节串的其中一部分内容进行读取或者写入的操作（这些操作也可以用于整数或者浮点数，但这种用法并不常见），本书在第9章将展示如何使用这些操作来高效地将结构化数据打包（pack）存储到字符串键里面。表3-2展示了用来处理字符串子串和二进制位的命令。

表3-2 供Redis处理子串和二进制位的命令

命令	用例和描述
APPEND	APPEND key-name value——将值value追加到给定键key-name当前存储的值的末尾
GETRANGE	GETRANGE key-name start end——获取一个由偏移量start至偏移量end范围内所有字符组成的子串，包括start和end在内
SETRANGE	SETRANGE key-name offset value——将从start偏移量开始的子串设置为给定值
GETBIT	GETBIT key-name offset——将字节串看作是二进制位串(bit string)，并返回位串中偏移量为offset的二进制位的值
SETBIT	SETBIT key-name offset value——将字节串看作是二进制位串，并将位串中偏移量为offset的二进制位的值设置为value
BITCOUNT	BITCOUNT key-name [start end]——统计二进制位串里面值为1的二进制位的数量，如果给定了可选的start偏移量和end偏移量，那么只对偏移量指定范围内的二进制位进行统计
BITOP	BITOP operation dest-key key-name [key-name ...]——对一个或多个二进制位串执行包括并(AND)、或(OR)、异或(XOR)、非(NOT)在内的任意一种按位运算操作(bitwise operation)，并将计算得出的结果保存在dest-key键里面

**GETRANGE 和 SUBSTR** Redis 现在的 GETRANGE 命令是由以前的 SUBSTR 命令改名而来的，因此，Python 客户端至今仍然可以使用 substr() 方法来获取子串，但如果读者使用的是 2.6 或以上版本的 Redis，那么最好还是使用 getrange() 方法来获取子串。

在使用 SETRANGE 或者 SETBIT 命令对字符串进行写入的时候，如果字符串当前的长度不能满足写入的要求，那么 Redis 会自动地使用空字节（null）来将字符串扩展至所需的长度，然后才执行写入或者更新操作。在使用 GETRANGE 读取字符串的时候，超出字符串末尾的数据会被视为是空串，而在使用 GETBIT 读取二进制位串的时候，超出字符串末尾的二进制位会被视为是 0。代码清单 3-2 展示了一些字符串处理命令的使用示例。

代码清单 3-2 这个交互示例展示了 Redis 的子串操作和二进制位操作

APPEND 命令在执行之后会返回字符串当前的长度。

对字符串执行范围设置操作。

SETRANGE 命令在执行之后同样会返回字符串的当前总长度。

SETRANGE 命令既可以用于替换字符串里已有的内容，又可以用于增长字符串。

SETBIT 命令会返回二进制位被设置之前的值。

```
>>> conn.append('new-string-key', 'hello ')
6L
>>> conn.append('new-string-key', 'world!')
12L
>>> conn.substr('new-string-key', 3, 7)
'lo wo'
>>> conn.setrange('new-string-key', 0, 'H')
12
>>> conn.setrange('new-string-key', 6, 'W')
12
>>> conn.get('new-string-key')
'Hello World!'
>>> conn.setrange('new-string-key', 11, ' how are you?')
25
>>> conn.get('new-string-key')
'Hello World, how are you?'
>>> conn.setbit('another-key', 2, 1)
0
>>> conn.setbit('another-key', 7, 1)
0
>>> conn.get('another-key')
'!'
通过将第 2 个二进制位以及第 7 个二进制位的值设置为 1，键的值将变为 ‘!’，也就是编码为 33 的字符。
```

将字符串 'hello' 追加到目前并不存在的 'new-string-key' 键里。

Redis 的索引以 0 为开始，在进行范围访问时，范围的终点（endpoint）默认也包含在这个范围之内。

字符串 'lo wo' 位于字符串 'hello world!' 的中间。

前面执行的两个 SETRANGE 命令成功地将字母 h 和 w 从原来的小写改成了大写。

前面执行的 SETRANGE 命令移除了字符串末尾的感叹号，并将更多字符追加到了字符串末尾。

对超出字符串长度的二进制位进行设置时，超出的部分会被填充为空字节。

在对 Redis 存储的二进制位进行解释（interpret）时，请记住 Redis 存储的二进制位是按照偏移量从高到低排列的。

很多键值数据库只能将数据存储为普通的字符串，并且不提供任何字符串处理操作，有一些键值数据库允许用户将字节追加到字符串的前面或者后面，但是却没办法像 Redis 一样对字符串的子串进行读写。从很多方面来讲，即使 Redis 只支持字符串结构，并且只支持本节列出的字符串处理命令，Redis 也比很多别的数据库要强大得多；通过使用子串操作和二进制位操作，配合 WATCH

命令、MULTI命令和EXEC命令（本书的3.7.2节将对这3个命令进行初步的介绍，并在第4章对它们进行更深入的讲解），用户甚至可以自己动手去构建任何他们想要的数据结构。第9章将介绍如何使用字符串去存储一种简单的映射，这种映射可以在某些情况下节省大量内存。

只要花些心思，我们甚至可以将字符串当作列表来使用，但这种做法能够执行的列表操作并不多，更好的办法是直接使用下一节介绍的列表结构，Redis为这种结构提供了丰富的列表操作命令。

## 3.2 列表

在第1章曾经介绍过，Redis的列表允许用户从序列的两端推入或者弹出元素，获取列表元素，以及执行各种常见的列表操作。除此之外，列表还可以用来存储任务信息、最近浏览过的文章或者常用联系人信息。

本节将对列表这个由多个字符串值组成的有序序列结构进行介绍，并展示一些最常用的列表处理命令，阅读本节可以让读者学会如何使用这些命令来处理列表。表3-3展示了其中一部分最常用的列表命令。

表3-3 一些常用的列表命令

命令	用例和描述
RPUSH	RPUSH key-name value [value ...]——将一个或多个值推入列表的右端
LPUSH	LPUSH key-name value [value ...]——将一个或多个值推入列表的左端
RPOP	RPOP key-name——移除并返回列表最右端的元素
LPOP	LPOP key-name——移除并返回列表最左端的元素
LINDEX	LINDEX key-name offset——返回列表中偏移量为offset的元素
LRANGE	LRANGE key-name start end——返回列表从start偏移量到end偏移量范围内的所有元素，其中偏移量为start和偏移量为end的元素也会包含在被返回的元素之内
LTRIM	LTRIM key-name start end——对列表进行修剪，只保留从start偏移量到end偏移量范围内的元素，其中偏移量为start和偏移量为end的元素也会被保留

因为本书在第1章已经对列表的几个推入和弹出操作进行了简单的介绍，所以读者应该不会对上面列出的推入和弹出操作感到陌生，代码清单3-3展示了这些操作的用法。

### 代码清单3-3 这个交互示例展示了Redis列表的推入操作和弹出操作

在向列表推入元素时，推入操作执行完毕之后会返回列表当前的长度。

```
>>> conn.rpush('list-key', 'last')
1L
>>> conn.lpush('list-key', 'first')
2L
>>> conn.rpush('list-key', 'new last')
3L
>>> conn.lrange('list-key', 0, -1)
['first', 'last', 'new last']
```

可以很容易地对列表的两端执行推入操作。

从语义上来说，列表的左端为开头，右端为结尾。

```

>>> conn.lpop('list-key')
'first'
>>> conn.lpop('list-key')
'last'
>>> conn.lrange('list-key', 0, -1)
['new last']
>>> conn.rpush('list-key', 'a', 'b', 'c')    ← 可以同时推入多个
4L
元素。
>>> conn.lrange('list-key', 0, -1)
['new last', 'a', 'b', 'c']
>>> conn.ltrim('list-key', 2, -1)
True
>>> conn.lrange('list-key', 0, -1)
['b', 'c']   ← 可以从列表的左端、右端或者
左右两端删减任意数量的元素。

```

这个示例里面第一次用到了 LTRIM 命令，组合使用 LTRIM 和 LRANGE 可以构建出一个在功能上类似于 LPOP 或 RPOP，但是却能够一次返回并弹出多个元素的操作。本章稍后将会介绍原子地<sup>①</sup>执行多个命令的方法，而更高级的 Redis 事务特性则会在第 4 章介绍。

有几个列表命令可以将元素从一个列表移动到另一个列表，或者阻塞（block）执行命令的客户端直到有其他客户端给列表添加元素为止，这些命令在第 1 章都没有介绍过，表 3-4 列出了这些阻塞弹出命令和元素移动命令。

表 3-4 阻塞式的列表弹出命令以及在列表之间移动元素的命令

命令	用例和描述
BLPOP	BLPOP key-name [key-name ...] timeout——从第一个非空列表中弹出位于最左端的元素，或者在 timeout 秒之内阻塞并等待可弹出的元素出现
BRPOP	BRPOP key-name [key-name ...] timeout——从第一个非空列表中弹出位于最右端的元素，或者在 timeout 秒之内阻塞并等待可弹出的元素出现
RPOPLPUSH	RPOPLPUSH source-key dest-key——从 source-key 列表中弹出位于最右端的元素，然后将这个元素推入 dest-key 列表的最左端，并向用户返回这个元素
BRPOPLPUSH	BRPOPLPUSH source-key dest-key timeout——从 source-key 列表中弹出位于最右端的元素，然后将这个元素推入 dest-key 列表的最左端，并向用户返回这个元素；如果 source-key 为空，那么在 timeout 秒之内阻塞并等待可弹出的元素出现

在第 6 章讨论队列时，这组命令将会非常有用。代码清单 3-4 展示了几个使用 BRPOPLPUSH 移动列表元素的例子以及使用 BLPOP 从列表里面弹出多个元素的例子。

代码清单 3-4 这个交互示例展示了 Redis 列表的阻塞弹出命令以及元素移动命令

```

将一个元素从一个
列表移动到另一个
列表，并返回被
移动的元素。    ← 将一些元素添加到两
                个列表里面。
1
>>> conn.rpush('list', 'item1')
1
>>> conn.rpush('list', 'item2')
2
>>> conn.rpush('list2', 'item3')
1
>>> conn.brpoplpush('list2', 'list', 1)
'item3'

```

① 在 Redis 里面，多个命令原子地执行指的是，在这些命令正在读取或者修改数据的时候，其他客户端不能读取或者修改相同的数据。

当列表不包含任何元素时，阻塞弹出操作会在给定的时限内等待可弹出的元素出现，并在时限到达后返回 None（交互终端不会打印这个值）。

```
>>> conn.brpoplpush('list2', 'list', 1)
>>> conn.lrange('list', 0, -1)
['item3', 'item1', 'item2']
>>> conn.brpoplpush('list', 'list2', 1)
'item2'
>>> conn.blpop(['list', 'list2'], 1)
('list', 'item3')
>>> conn.blpop(['list', 'list2'], 1)
('list', 'item1')
>>> conn.blpop(['list', 'list2'], 1)
('list2', 'item2')
>>> conn.blpop(['list', 'list2'], 1)
>>>
```

弹出“list2”最右端的元素，并将被弹出的元素推入“list”的左端。

BLPOP 命令会从左到右地检查传入的列表，并对最先遇到的非空列表执行弹出操作。

对于阻塞弹出命令和弹出并推入命令，最常见的用例就是消息传递（messaging）和任务队列（task queue），本书将在第 6 章对这两个主题进行介绍。

### 练习：通过列表来降低内存占用

在 2.1 节和 2.5 节中，我们使用了有序集合来记录用户最近浏览过的商品，并把用户浏览这些商品时的时间戳设置为分值，从而使得程序可以在清理旧会话的过程中或是执行完购买操作之后，进行相应的数据分析。但由于保存时间戳需要占用相应空间，所以如果分析操作并不需要用到时间戳的话，那么就没有必要使用有序集合来保存用户最近浏览过的商品了。为此，请在保证语义不变的情况下，将 `update_token()` 函数里面使用的有序集合替换成列表。提示：如果读者在解答这个问题时遇上困难的话，可以到 6.1.1 节中找找灵感。

列表的一个主要优点在于它可以包含多个字符串值，这使得用户可以将数据集中在同一个地方。Redis 的集合也提供了与列表类似的特性，但集合只能保存各不相同的元素。接下来的一节中就让我们来看看不能保存相同元素的集合都能做些什么。

## 3.3 集合

Redis 的集合以无序的方式来存储多个各不相同的元素，用户可以快速地对集合执行添加元素操作、移除元素操作以及检查一个元素是否存在于集合里。第 1 章曾经对集合进行过简单的介绍，并在构建文章投票网站时，使用集合记录文章已投票用户名单以及群组属下的所有文章。

本节将对最常用的集合命令进行介绍，包括插入命令、移除命令、将元素从一个集合移动到另一个集合的命令，以及对多个集合执行交集运算、并集运算和差集运算的命令。阅读本节也有助于读者更好地理解本书在第 7 章介绍的搜索示例。

表 3-5 展示了其中一部分最常用的集合命令。

表 3-5 一些常用的集合命令

命令	用例和描述
SADD	SADD key-name item [item ...]——将一个或多个元素添加到集合里面，并返回被添加元素当中原本并不存在于集合里面的元素数量
SREM	SREM key-name item [item ...]——从集合里面移除一个或多个元素，并返回被移除元素的数量
SISMEMBER	SISMEMBER key-name item——检查元素 item 是否存在于集合 key-name 里
SCARD	SCARD key-name——返回集合包含的元素的数量
SMEMBERS	SMEMBERS key-name——返回集合包含的所有元素
SRANDMEMBER	SRANDMEMBER key-name [count]——从集合里面随机地返回一个或多个元素。当 count 为正数时，命令返回的随机元素不会重复；当 count 为负数时，命令返回的随机元素可能会出现重复
SPOP	SPOP key-name——随机地移除集合中的一个元素，并返回被移除的元素
SMOVE	SMOVE source-key dest-key item——如果集合 source-key 包含元素 item，那么从集合 source-key 里面移除元素 item，并将元素 item 添加到集合 dest-key 中；如果 item 被成功移除，那么命令返回 1，否则返回 0

表 3-5 里面的不少命令都已经在第 1 章介绍过了，代码清单 3-5 展示了这些命令的使用示例。

代码清单 3-5 这个交互示例展示了 Redis 中的一些常用的集合命令

SADD 命令会将那些目前并不存在于集合里面的元素添加到集合里面，并返回被添加元素的数量。

获取集合包含的所有元素。

```
>>> conn.sadd('set-key', 'a', 'b', 'c')
3
>>> conn.srem('set-key', 'c', 'd')
True
>>> conn.srem('set-key', 'c', 'd')
False
>>> conn.scard('set-key')
2
>>> conn.smembers('set-key')
set(['a', 'b'])
>>> conn.smove('set-key', 'set-key2', 'a')
True
>>> conn.smove('set-key', 'set-key2', 'c')
False
>>> conn.smembers('set-key2')
set(['a'])
```

srem 函数在元素被成功移除时返回 True，移除失败时返回 False；注意这是 Python 客户端的一个 bug，实际上 Redis 的 SREM 命令返回的是被移除元素的数量，而不是布尔值。

查看集合包含的元素数量。

可以很容易地将元素从一个集合移动到另一个集合。在执行 SMOVE 命令时，如果用户想要移动的元素不存在于第一个集合里，那么移动操作就不会执行。

通过使用上面展示的命令，我们可以将各不相同的多个元素添加到集合里面，比如第 1 章就使用集合记录了文章已投票用户名单，以及文章属于哪个群组。但集合真正厉害的地方在于组合和关联多个集合，表 3-6 展示了相关的命令。

表 3-6 用于组合和处理多个集合的 Redis 命令

命令	用例和描述
SDIFF	SDIFF key-name [key-name ...]——返回那些存在于第一个集合、但不存在于其他集合中的元素（数学上的差集运算）

续表

命令	用例和描述
SDIFFSTORE	SDIFFSTORE dest-key key-name [key-name ...]——将那些存在于第一个集合但并不存在于其他集合中的元素（数学上的差集运算）存储到 dest-key 键里面
SINTER	SINTER key-name [key-name ...]——返回那些同时存在于所有集合中的元素（数学上的交集运算）
SINTERSTORE	SINTERSTORE dest-key key-name [key-name ...]——将那些同时存在于所有集合的元素（数学上的交集运算）存储到 dest-key 键里面
SUNION	SUNION key-name [key-name ...]——返回那些至少存在于一个集合中的元素（数学上的并集计算）
SUNIONSTORE	SUNIONSTORE dest-key key-name [key-name ...]——将那些至少存在于一个集合中的元素（数学上的并集计算）存储到 dest-key 键里面

这些命令分别是并集运算、交集运算和差集运算这3个基本集合操作的“返回结果”版本和“存储结果”版本，代码清单3-6展示了这些命令的使用示例。

#### 代码清单3-6 这个交互示例展示了Redis的差集运算、交集运算以及并集运算

```
计算出从第一个
集合里面移除第
二个集合包含的
所有元素的结果。    >>> conn.sadd('skey1', 'a', 'b', 'c', 'd')
4
>>> conn.sadd('skey2', 'c', 'd', 'e', 'f')
4
>>> conn.sdiff('skey1', 'skey2')
set(['a', 'b'])
首先将一些元素添
加到两个集合里面。
计算出同时存在于两个
集合里面的所有元素。    >>> conn.sinter('skey1', 'skey2')
set(['c', 'd'])
>>> conn.sunion('skey1', 'skey2')
set(['a', 'c', 'b', 'e', 'd', 'f'])
计算出两个集合包含的所有各不相同的元素。
```

和Python的集合相比，Redis的集合除了可以被多个客户端远程地进行访问之外，其他的语义和功能基本都是相同的。

接下来的一节将对Redis的散列处理命令进行介绍，这些命令允许用户将多个相关的键值对存储在一起，以便执行获取操作和更新操作。

## 3.4 散列

第1章提到过，Redis的散列可以让用户将多个键值对存储到一个Redis键里面。从功能上来说，Redis为散列值提供了一些与字符串值相同的特性，使得散列非常适用于将一些相关的数据存储在一起。我们可以把这种数据聚集看作是关系数据库中的行，或者文档数据库中的文档。

本节将对最常用的散列命令进行介绍：其中包括添加和删除键值对的命令、获取所有键值对的命令，以及对键值对的值进行自增或者自减操作的命令。阅读这一节可以让读者学习到如何将

数据存储到散列里面，以及这样做的好处是什么。表 3-7 展示了一部分常用的散列命令。

表 3-7 用于添加和删除键值对的散列操作

命令	用例和描述
HMGET	HMGET key-name key [key ...]——从散列里面获取一个或多个键的值
HMSET	HMSET key-name key value [key value ...]——为散列里面的一个或多个键设置值
HDEL	HDEL key-name key [key ...]——删除散列里面的一个或多个键值对，返回成功找到并删除的键值对数量
HLEN	HLEN key-name——返回散列包含的键值对数量

在表 3-7 列出的命令当中，HDEL 命令已经在第 1 章中介绍过了，而 HLEN 命令以及用于一次读取或者设置多个键的 HMGET 和 HMSET 则是新出现的命令。像 HMGET 和 HMSET 这种批量处理多个键的命令既可以给用户带来方便，又可以通过减少命令的调用次数以及客户端与 Redis 之间的通信往返次数来提升 Redis 的性能。代码清单 3-7 展示了这些命令的使用方法。

#### 代码清单 3-7 这个交互示例展示了 Redis 中的一些常用的散列命令

```
使用 HMSET 命令可以一次将多个键值对添加到散列里面。
    >>> conn.hmset('hash-key', {'k1':'v1', 'k2':'v2', 'k3':'v3'})
    True
    使用 HMGET 命令可以一次获取多个键的值。
    >>> conn.hmget('hash-key', ['k2', 'k3'])
    ['v2', 'v3']
    >>> conn.hlen('hash-key')
    3
    HLEN 命令通常用于调试一个包含非常多键值对的散列。
    >>> conn.hdel('hash-key', 'k1', 'k3')
    True
    HDEL 命令在成功地移除了至少一个键值对时返回 True，因为 HDEL 命令已经可以同时删除多个键值对了，所以 Redis 没有实现 HMDEL 命令。
```

第 1 章介绍的 HGET 命令和 HSET 命令分别是 HMGET 命令和 HMSET 命令的单参数版本，这些命令的唯一区别在于单参数版本每次执行只能处理一个键值对，而多参数版本每次执行可以处理多个键值对。

表 3-8 列出了散列的其他几个批量操作命令，以及一些和字符串操作类似的散列命令。

表 3-8 展示 Redis 散列的更高级特性

命令	用例和描述
HEXISTS	HEXISTS key-name key——检查给定键是否存在于散列中
HKEYS	HKEYS key-name——获取散列包含的所有键
HVALS	HVALS key-name——获取散列包含的所有值
HGETALL	HGETALL key-name——获取散列包含的所有键值对
HINCRBY	HINCRBY key-name key increment——将键 key 存储的值加上整数 increment
HINCRBYFLOAT	HINCRBYFLOAT key-name key increment——将键 key 存储的值加上浮点数 increment

尽管有 HGETALL 存在，但 HKEYS 和 HVALUES 也是非常有用的：如果散列包含的值非常大，那么用户可以先使用 HKEYS 取出散列包含的所有键，然后再使用 HGET 一个接一个地取出键的值，从而避免因为一次获取多个大体积的值而导致服务器阻塞。

HINCRBY 和 HINCRBYFLOAT 可能会让读者回想起用于处理字符串的 INCRBY 和 INCRBYFLOAT，这两对命令拥有相同的语义，它们的不同在于 HINCRBY 和 HINCRBYFLOAT 处理的是散列，而不是字符串。代码清单 3-8 展示了这些命令的使用方法。

**代码清单 3-8** 这个交互示例展示了 Redis 散列的一些更高级的特性

```
>>> conn.hmset('hash-key2', {'short':'hello', 'long':1000*'1'})
True
>>> conn.hkeys('hash-key2')
['long', 'short']
>>> conn.hexists('hash-key2', 'num')
False
>>> conn.hincrby('hash-key2', 'num')
1L
>>> conn.hexists('hash-key2', 'num')
True
```

在考察散列的时候，我们可以只取出散列包含的键，避免传输体积较大的值。

检查给定的键是否存在  
于散列中。

和字符串一样，对散列中一个尚未存在的键执行自增操作时，Redis 会将键的值当作 0 来处理。

正如前面所说，在对散列进行处理的时候，如果键值对的值的体积非常庞大，那么用户可以先使用 HKEYS 获取散列的所有键，然后通过只获取必要的值来减少需要传输的数据量。除此之外，用户还可以像使用 SISMEMBER 检查一个元素是否存在于集合里面一样，使用 HEXISTS 检查一个键是否存在于散列里面。另外第 1 章也用到了本节刚刚回顾过的 HINCRBY 来记录文章被投票的次数。

在接下来的一节中，我们要了解的是之后的章节里面会经常用到的有序集合结构。

## 3.5 有序集合

和散列存储着键与值之间的映射类似，有序集合也存储着成员与分值之间的映射，并且提供了分值<sup>①</sup>处理命令，以及根据分值大小有序地获取（fetch）或扫描（scan）成员和分值的命令。本书曾在第 1 章使用有序集合实现过基于发表时间排序的文章列表和基于投票数量排序的文章列表，还在第 2 章使用有序集合存储过 cookie 的过期时间。

本节将对操作有序集合的命令进行介绍，其中包括向有序集合添加新元素的命令、更新已有元素的命令，以及对有序集合进行交集运算和并集运算的命令。阅读本节可以加深读者对有序集合的认识，从而帮助读者更好地理解本书在第 1 章、第 5 章、第 6 章和第 7 章展示的有序集合示例。

① 这些分值在 Redis 中以 IEEE 754 双精度浮点数的格式存储。

表 3-9 展示了一部分常用的有序集合命令。

表 3-9 一些常用的有序集合命令

命令	用例和描述
ZADD	ZADD key-name score member [score member ...]——将带有给定分值的成员添加到有序集合里面
ZREM	ZREM key-name member [member ...]——从有序集合里面移除给定的成员，并返回被移除成员的数量
ZCARD	ZCARD key-name——返回有序集合包含的成员数量
ZINCRBY	ZINCRBY key-name increment member——将 member 成员的分值加上 increment
ZCOUNT	ZCOUNT key-name min max——返回分值介于 min 和 max 之间的成员数量
ZRANK	ZRANK key-name member——返回成员 member 在有序集合中的排名
ZSCORE	ZSCORE key-name member——返回成员 member 的分值
ZRANGE	ZRANGE key-name start stop [WITHSCORES]——返回有序集合中排名介于 start 和 stop 之间的成员，如果给定了可选的 WITHSCORES 选项，那么命令会将成员的分值也一并返回

在上面列出的命令当中，有一部分命令已经在第 1 章和第 2 章使用过了，所以读者应该不会对它们感到陌生，代码清单 3-9 回顾了这些命令的用法。

#### 代码清单 3-9 这个交互示例展示了 Redis 中的一些常用的有序集合命令

在 Python 客户端执行 ZADD 命令需要先输入成员、后输入分值，这跟 Redis 标准的先输入分值、后输入成员的做法正好相反。

获取指定成员的排名（排名以 0 为开始），之后可以根据这个排名来决定 ZRANGE 的访问范围。

从有序集合里面移除成员和添加成员一样容易。

```
>>> conn.zadd('zset-key', 'a', 3, 'b', 2, 'c', 1)
```

3

```
>>> conn.zcard('zset-key')
```

3

```
>>> conn.zincrby('zset-key', 'c', 3)
```

4.0

```
>>> conn.zscore('zset-key', 'b')
```

2.0

```
>>> conn.zrank('zset-key', 'c')
```

2

```
>>> conn.zcount('zset-key', 0, 3)
```

2L

对于某些任务来说，统计给定分值范围内的元素数量非常有用。

取得有序集合的大小可以让我们在某些情况下知道是否需要对有序集合进行修剪。

跟字符串和散列一样，有序集合的成员也可以执行自增操作。

获取单个成员的分值对于实现计数器或者排行榜之类的功能非常有用。

```
>>> conn.zrem('zset-key', 'b')
```

True

```
>>> conn.zrange('zset-key', 0, -1, withscores=True)
```

[('a', 3.0), ('c', 4.0)]

在进行调试时，我们通常会使用 ZRANGE 取出有序集合包含的所有元素，但是在实际用例中，通常一次只会取出一小部分元素。

因为 ZADD、ZREM、ZINCRBY、ZSCORE 和 ZRANGE 都已经在第 1 章和第 2 章介绍过了，所以读者应该不会对它们感到陌生。ZCOUNT 命令和其他命令不太相同，它主要用于计算分值在给定范围内的成员数量。

表3-10展示了另外一些非常有用的有序集合命令。

表3-10 有序集合的范围型数据获取命令和范围型数据删除命令，以及并集命令和交集命令

命令	用例和描述
ZREVRANK	ZREVRANK key-name member——返回有序集合里成员member的排名，成员按照分值从大到小排列
ZREVRANGE	ZREVRANGE key-name start stop [WITHSCORES]——返回有序集合给定排名范围内的成员，成员按照分值从大到小排列
ZRANGEBYSCORE	ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]——返回有序集合中，分值介于min和max之间的所有成员
ZREVRANGEBYSCORE	ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]——获取有序集合中分值介于min和max之间的所有成员，并按照分值从大到小的顺序来返回它们
ZREMRANGEBYRANK	ZREMRANGEBYRANK key-name start stop——移除有序集合中排名介于start和stop之间的所有成员
ZREMRANGEBYSCORE	ZREMRANGEBYSCORE key-name min max——移除有序集合中分值介于min和max之间的所有成员
ZINTERSTORE	ZINTERSTORE dest-key key-count key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM MIN MAX]——对给定的有序集合执行类似于集合的交集运算
ZUNIONSTORE	ZUNIONSTORE dest-key key-count key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM MIN MAX]——对给定的有序集合执行类似于集合的并集运算

在表3-10展示的命令里面，有几个是之前没介绍过的新命令。除了使用逆序来处理有序集合之外，ZREV\*命令的工作方式和相对应的非逆序命令的工作方式完全一样（逆序就是指元素按照分值从大到小地排列）。代码清单3-10展示了ZINTERSTORE和ZUNIONSTORE的用法。

代码清单3-10 这个交互示例展示了ZINTERSTORE命令和ZUNIONSTORE命令的用法

首先创建两个有序集合。

```
>>> conn.zadd('zset-1', 'a', 1, 'b', 2, 'c', 3)
3
>>> conn.zadd('zset-2', 'b', 4, 'c', 1, 'd', 0)
3
>>> conn.zinterstore('zset-i', ['zset-1', 'zset-2'])
2L
>>> conn.zrange('zset-i', 0, -1, withscores=True)
[('c', 4.0), ('b', 6.0)]
```

ZINTERSTORE和ZUNIONSTORE默认使用的聚合函数为sum，这个函数会把各个有序集合的成员的分值都加起来。

```
>>> conn.zunionstore('zset-u', ['zset-1', 'zset-2'], aggregate='min')
4L
>>> conn.zrange('zset-u', 0, -1, withscores=True)
[('d', 0.0), ('a', 1.0), ('c', 1.0), ('b', 2.0)]
>>> conn.sadd('set-1', 'a', 'd')
2
>>> conn.zunionstore('zset-u2', ['zset-1', 'zset-2', 'set-1'])
4L
>>> conn.zrange('zset-u2', 0, -1, withscores=True)
[('d', 1.0), ('a', 2.0), ('c', 4.0), ('b', 6.0)]
```

用户可以在执行并集运算和交集运算的时候传入不同的聚合函数，共有 sum、min、max 三个聚合函数可选。

用户还可以把集合作为输入传给 ZINTERSTORE 和 ZUNIONSTORE，命令会将集合看作是成员分值全为 1 的有序集合来处理。

有序集合的并集运算和交集运算在刚开始接触时可能会比较难懂，所以本节将使用图片来展示交集运算和并集运算的执行过程。图 3-1 展示了对两个输入有序集合执行交集运算并得到输出有序集合的过程，这次交集运算使用的是默认的聚合函数 sum，所以输出有序集合成员的分值都是通过加法计算得出的。

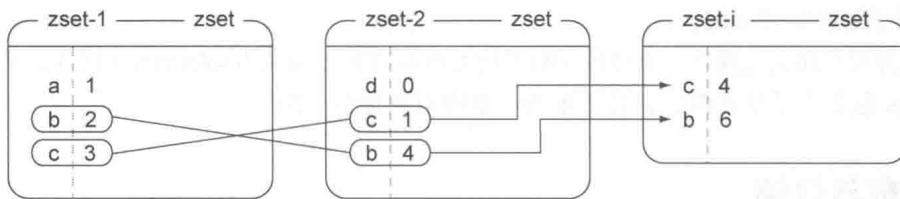


图 3-1 执行 conn.zinterstore('zset-i', ['zset-1', 'zset-2'])

将使得同时存在于 zset-1 和 zset-2 里面的元素被添加到 zset-i 里面

并集运算和交集运算不同，只要某个成员存在于至少一个输入有序集合里面，那么这个成员就会被包含在输出有序集合里面。图 3-2 展示了使用聚合函数 min 执行并集运算的过程，min 函数在多个输入有序集合都包含同一个成员的情况下，会将最小的那个分值设置为这个成员在输出有序集合的分值。

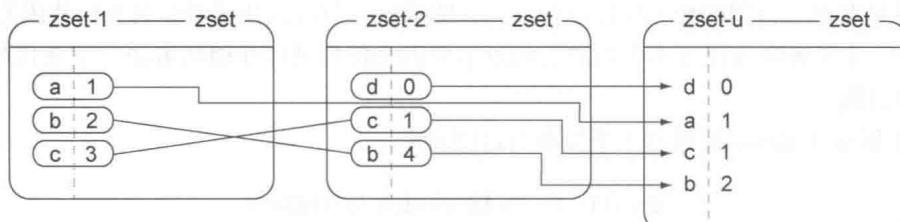


图 3-2 执行 conn.zunionstore('zset-u', ['zset-1', 'zset-2'], aggregate='min')

会将存在于 zset-1 或者 zset-2 里面的元素通过 min 函数组合到 zset-u 里面

在第 1 章中，我们就基于“集合可以作为 ZUNIONSTORE 操作和 ZINTERSTORE 操作的输

入”这个事实，在没有使用额外的有序集合来存储群组文章的评分和发布时间的情况下，实现了群组文章的添加和删除操作。图3-3展示了如何使用ZUNIONSTORE命令来将两个有序集合和一个集合组合成一个有序集合。

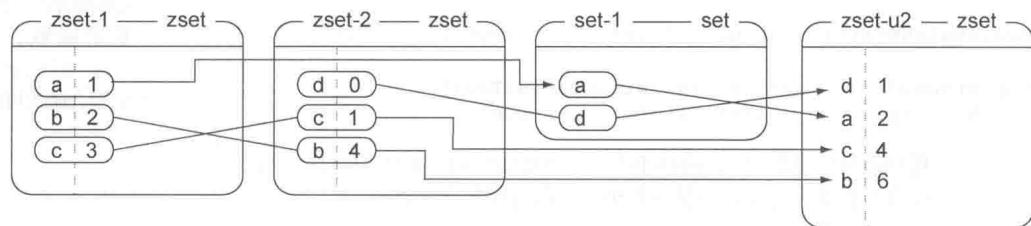


图3-3 执行`conn.zunionstore('zset-u2', ['zset-1', 'zset-2', 'set-1'])`  
将使得所有存在于zset-1、zset-2或者set-1里面的元素都被添加到zset-u2里面

第7章将使用ZINTERSTORE和ZUNIONSTORE来构建几个不同类型的搜索系统，并说明如何通过可选的WEIGHTS参数来以几种不同的方式组合有序集合的分值，从而使得集合和有序集合可以用于解决更多问题。

读者在开发应用的过程中，也许曾经听说过发布与订阅（publish/subscribe）模式，又称pub/sub模式，Redis也实现了这种模式，接下来的一节将对其进行介绍。

## 3.6 发布与订阅

如果你因为想不起来本书在前面的那个章节里面介绍过发布与订阅而困惑，那么大可不必——这是本书目前为止第一次介绍发布与订阅。一般来说，发布与订阅（又称pub/sub）的特点是订阅者（listener）负责订阅频道（channel），发送者（publisher）负责向频道发送二进制字符串消息（binary string message）。每当有消息被发送至给定频道时，频道的所有订阅者都会收到消息。我们也可以把频道看作是电台，其中订阅者可以同时收听多个电台，而发送者则可以在任何电台发送消息。

本节将对发布与订阅的相关操作进行介绍，阅读这一节可以让读者学会怎样使用发布与订阅的相关命令，并了解到为什么本书在之后的章节里面会使用其他相似的解决方案来代替Redis提供的发布与订阅。

表3-11展示了Redis提供的5个发布与订阅命令。

表3-11 Redis提供的发布与订阅命令

命令	用例和描述
SUBSCRIBE	SUBSCRIBE channel [channel ...]——订阅给定的一个或多个频道
UNSUBSCRIBE	UNSUBSCRIBE [channel [channel ...]]——退订给定的一个或多个频道，如果执行时没有给定任何频道，那么退订所有频道

续表

命令	用例和描述
PUBLISH	PUBLISH channel message——向给定频道发送消息
PSUBSCRIBE	PSUBSCRIBE pattern [pattern ...]——订阅与给定模式相匹配的所有频道
PUNSUBSCRIBE	PUNSUBSCRIBE [pattern [pattern ...]]——退订给定的模式，如果执行时没有给定任何模式，那么退订所有模式

考虑到 PUBLISH 命令和 SUBSCRIBE 命令在 Python 客户端的实现方式，一个比较简单的演示发布与订阅的方法，就是像代码清单 3-11 那样使用辅助线程（helper thread）来执行 PUBLISH 命令<sup>①</sup>。

代码清单 3-11 这个交互示例展示了如何使用 Redis 中的 PUBLISH 命令以及 SUBSCRIBE 命令

```

>>> def publisher(n):
...     time.sleep(1)
...     for i in xrange(n):
...         conn.publish('channel', i)
...         time.sleep(1)

启动发送者
线程，并让
它发送三条
消息。
    >>> def run_pubsub():
...     threading.Thread(target=publisher, args=(3,)).start()
...     pubsub = conn.pubsub()
...     pubsub.subscribe(['channel'])
...     count = 0

通过遍历函
数pubsub.
listen()的
执行结果来
监听订阅消息。
    >>> for item in pubsub.listen():
...     print item
...     count += 1
...     if count == 4:
...         pubsub.unsubscribe()
...     if count == 5:
...         break
...     <<<

    >>> run_pubsub()
{'pattern': None, 'type': 'subscribe', 'channel': 'channel', 'data': 1L}
实际运行函数并观察它
们的行为了。
{'pattern': None, 'type': 'message', 'channel': 'channel', 'data': '0'}
{'pattern': None, 'type': 'message', 'channel': 'channel', 'data': '1'}
{'pattern': None, 'type': 'message', 'channel': 'channel', 'data': '2'}
{'pattern': None, 'type': 'unsubscribe', 'channel': 'channel', 'data': '0L'}
```

在退订频道时，客户端会接收到一条反馈消息，告知被退订的是哪个频道，以及客户端目前仍在订阅的频道数量。

这些结构就是我们在遍历 pubsub.listen() 函数时得到的元素。

在刚开始执行时会先休眠，让订阅者有足够的  
时间来连接服务器并监听消息。

在发布消息之后进行短暂的休眠，让  
消息可以一条接一条地出现。

创建发布与订阅  
对象，并让它订  
阅给定的频道。

打印接收到的每条消息。

在接收到一条订阅反馈消息和三条发布者发送的消息之后，执行退订操作，停止监听新消息。

客户端在接收到退订反馈消息之后就不再接收消息。

<sup>①</sup> 代码清单中的 publisher() 函数和 run\_pubsub() 函数都包含在本章对应的源代码里面，如果读者有兴趣的话，可以自己亲自试一下。

虽然 Redis 的发布与订阅模式非常有用，但本书只在这一节和 8.5 节中使用了这个模式，这样做的原因有以下两个。

第一个原因和 Redis 系统的稳定性有关。对于旧版 Redis 来说，如果一个客户端订阅了某个或某些频道，但它读取消息的速度却不够快的话，那么不断积压的消息就会使得 Redis 输出缓冲区的体积变得越来越大，这可能会导致 Redis 的速度变慢，甚至直接崩溃。也可能导致 Redis 被操作系统强制杀死，甚至导致操作系统本身不可用。新版的 Redis 不会出现这种问题，因为它会自动断开不符合 `client-output-buffer-limit pubsub` 配置选项要求的订阅客户端（本书第 8 章将对这个选项做更详细的介绍）。

第二个原因和数据传输的可靠性有关。任何网络系统在执行操作时都可能会遇上断线情况，而断线产生的连接错误通常会使得网络连接两端中的其中一端进行重新连接。本书使用的 Python 语言的 Redis 客户端会在连接失效时自动进行重新连接，也会自动处理连接池（connection pool，具体信息将在第 4 章介绍），诸如此类。但是，如果客户端在执行订阅操作的过程中断线，那么客户端将丢失在断线期间发送的所有消息，因此依靠频道来接收消息的用户可能会对 Redis 提供的 `PUBLISH` 命令和 `SUBSCRIBE` 命令的语义感到失望。

基于以上两个原因，本书在第 6 章编写了两个不同的方法来实现可靠的消息传递操作，这两个方法除了可以处理网络断线之外，还可以防止 Redis 因为消息积压而耗费过多内存（这个方法即使对于旧版 Redis 也是有效的）。

如果你喜欢简单易用的 `PUBLISH` 命令和 `SUBSCRIBE` 命令，并且能够承担可能会丢失一小部分数据的风险，那么你也可以继续使用 Redis 提供的发布与订阅特性，而不是 8.5 节中提供的实现，只要记得先把 `client-output-buffer-limit pubsub` 选项设置好就行了。

到目前为止，本书介绍的大多数命令都是与特定数据类型相关的。接下来的一节要介绍的命令你可能也会用到，但它们既不属于 Redis 提供的 5 种数据结构，也不属于发布与订阅特性。

## 3.7 其他命令

到目前为止，本章介绍了 Redis 提供的 5 种结构以及 Redis 的发布与订阅模式。本节将要介绍的命令则可以用于处理多种类型的数据：首先要介绍的是可以同时处理字符串、集合、列表和散列的 `SORT` 命令；之后要介绍是用于实现基本事务特性的 `MULTI` 命令和 `EXEC` 命令，这两个命令可以让用户将多个命令当作一个命令来执行；最后要介绍的是几个不同的自动过期命令，它们可以自动删除无用数据。

阅读本节有助于读者更好地理解如何同时组合和操作多种数据类型。

### 3.7.1 排序

Redis 的排序操作和其他编程语言的排序操作一样，都可以根据某种比较规则对一系列元素

进行有序的排列。负责执行排序操作的 SORT 命令可以根据字符串、列表、集合、有序集合、散列这 5 种键里面存储着的数据，对列表、集合以及有序集合进行排序。如果读者之前曾经使用过关系数据库的话，那么可以将 SORT 命令看作是 SQL 语言里的 order by 子句。表 3-12 展示了 SORT 命令的定义。

表 3-12 SORT 命令的定义

命令	用例和描述
SORT	SORT source-key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC DESC] [ALPHA] [STORE dest-key]——根据给定的选项，对输入列表、集合或者有序集合进行排序，然后返回或者存储排序的结果

使用 SORT 命令提供的选项可以实现以下功能：根据降序而不是默认的升序来排序元素；将元素看作是数字来进行排序，或者将元素看作是二进制字符串来进行排序（比如排序字符串 '110' 和 '12' 的结果就跟排序数字 110 和 12 的结果不一样）；使用被排序元素之外的其他值作为权重来进行排序，甚至还可以从输入的列表、集合、有序集合以外的其他地方进行取值。

代码清单 3-12 展示了一些 SORT 命令的使用示例。其中，最开头的几行代码设置了一些初始数据，然后对这些数据进行了数值排序和字符串排序，最后的代码演示了如何通过 SORT 命令的特殊语法来将散列存储的数据作为权重进行排序，以及怎样获取并返回散列存储的数据。

代码清单 3-12 这个交互示例展示了 **SORT** 命令的一些简单的用法

```
>>> conn.rpush('sort-input', 23, 15, 110, 7)
4
>>> conn.sort('sort-input')
['7', '15', '23', '110']

>>> conn.sort('sort-input', alpha=True)
['110', '15', '23', '7']
>>> conn.hset('d-7', 'field', 5)
1L
>>> conn.hset('d-15', 'field', 1)
1L
>>> conn.hset('d-23', 'field', 9)
1L
>>> conn.hset('d-110', 'field', 3)
1L
>>> conn.sort('sort-input', by='d-->field')
['15', '110', '7', '23']
>>> conn.sort('sort-input', by='d-->field', get='d-->field')
['1', '3', '5', '9']

首先将一些元素添加到列表里面。
根据数字大小对元素进行排序。
根据字母表顺序对元素进行排序。
添加一些用于执行排序操作和获取操作的附加数据。
将散列的域 (field) 用作权重，对 sort-input 列表进行排序。
获取外部数据，并将它们用作命令的返回值，而不是返回被排序的数据。
```

SORT 命令不仅可以对列表进行排序，还可以对集合进行排序，然后返回一个列表形式的排序结果。代码清单 3-12 除了展示如何使用 alpha 关键字参数对元素进行字符串排序之外，还展

示了如何基于外部数据对元素进行排序，以及如何获取并返回外部数据。第7章将介绍如何组合使用集合操作和 SORT 命令：当集合结构计算交集、并集和差集的能力，与 SORT 命令获取散列存储的外部数据的能力相结合时，SORT 命令将变得非常强大。

尽管 SORT 是 Redis 中唯一一个可以同时处理 3 种不同类型的数据的命令，但基本的 Redis 事务同样可以让我们在一连串不间断执行的命令里面操作多种不同类型的数据。

### 3.7.2 基本的 Redis 事务

有时候为了同时处理多个结构，我们需要向 Redis 发送多个命令。尽管 Redis 有几个可以在两个键之间复制或者移动元素的命令，但却没有那种可以在两个不同类型之间移动元素的命令（虽然可以使用 ZUNIONSTORE 命令将元素从一个集合复制到一个有序集合）。为了对相同或者不同类型的多个键执行操作，Redis 有 5 个命令可以让用户在不被打断（interruption）的情况下对多个键执行操作，它们分别是 WATCH、MULTI、EXEC、UNWATCH 和 DISCARD。

这一节只介绍最基本的 Redis 事务用法，其中只会用到 MULTI 命令和 EXEC 命令。如果读者想看看使用 WATCH、MULTI、EXEC 和 UNWATCH 等多个命令的事务是什么样子的，可以阅读 4.4 节，其中解释了为什么需要在使用 MULTI 和 EXEC 的同时使用 WATCH 和 UNWATCH。

#### 什么是 Redis 的基本事务

Redis 的基本事务（basic transaction）需要用到 MULTI 命令和 EXEC 命令，这种事务可以让一个客户端在不被其他客户端打断的情况下执行多个命令。和关系数据库那种可以在执行的过程中进行回滚（rollback）的事务不同，在 Redis 里面，被 MULTI 命令和 EXEC 命令包围的所有命令会一个接一个地执行，直到所有命令都执行完毕为止。当一个事务执行完毕之后，Redis 才会处理其他客户端的命令。

要在 Redis 里面执行事务，我们首先需要执行 MULTI 命令，然后输入那些我们想要在事务里面执行的命令，最后再执行 EXEC 命令。当 Redis 从一个客户端那里接收到 MULTI 命令时，Redis 会将这个客户端之后发送的所有命令都放入到一个队列里面，直到这个客户端发送 EXEC 命令为止，然后 Redis 就会在不被打断的情况下，一个接一个地执行存储在队列里面的命令。从语义上来说，Redis 事务在 Python 客户端上面是由流水线（pipeline）实现的：对连接对象调用 `pipeline()` 方法将创建一个事务，在一切正常的情况下，客户端会自动地使用 MULTI 和 EXEC 包裹起用户输入的多个命令。此外，为了减少 Redis 与客户端之间的通信往返次数，提升执行多个命令时的性能，Python 的 Redis 客户端会存储起事务包含的多个命令，然后在事务执行时一次性地将所有命令都发送给 Redis。

跟介绍 PUBLISH 命令和 SUBSCRIBE 命令时的情况一样，要展示事务执行结果，最简单的方法就是将事务放到线程里面执行。代码清单 3-13 展示了在没有使用事务的情况下，执行并行（parallel）自增操作的结果。

## 代码清单 3-13 在并行执行命令时，缺少事务可能会引发的问题

```

等待 100 毫秒。    >>> def notrans():
...     print conn.incr('notrans:')
...     time.sleep(.1)
...     conn.incr('notrans:', -1)
...
>>> if 1:
...     for i in xrange(3):
...         threading.Thread(target=notrans).start()
...         time.sleep(.5)
...
1
2
3
    对 'notrans:' 计数器执行自增操作并打印操作的执行结果。
    对 'notrans:' 计数器执行自减操作。
    等待 500 毫秒，让操作有足够的时
间完成。
    因为没有使用事务，所以 3 个线程
执行的各个命令将互相交错，使得
计数器的值持续地增大。

```

因为没有使用事务，所以 3 个线程都可以在执行自减操作之前，对 notrans: 计数器执行自增操作。虽然代码清单里面通过休眠 100 毫秒的方式来放大了潜在的问题，但如果我们确实需要在不受其他命令干扰的情况下，对计数器执行自增操作和自减操作，那么我们就不得不解决这个潜在的问题。代码清单 3-14 展示了如何使用事务来执行相同的操作。

## 代码清单 3-14 使用事务来处理命令的并行执行问题

```

创建一个事务型
(transactional) 流
水线对象。    >>> def trans():
...     pipeline = conn.pipeline()
...     pipeline.incr('trans:')
...     time.sleep(.1)
...
...     pipeline.incr('trans:', -1)
...     print pipeline.execute()[0]
...
>>> if 1:
...     for i in xrange(3):
...         threading.Thread(target=trans).start()
...         time.sleep(.5)
...
1
1
1
    把针对 'trans:' 计数器
    的自增操作放入队列。
    等待 100 ms。
    启动 3 个线程来执行被事务包裹的
    自增、休眠和自减 3 个操作。
    等待 500 ms,
    让操作有足够
    的时间完成。
    因为每组自增、休眠和自减操作都在事务
    里面执行，所以命令之间不会互相交错，
    因此所有事务的执行结果都是 1。

```

可以看到，尽管自增操作和自减操作之间有一段延迟时间，但通过使用事务，各个线程都可以在不被其他线程打断的情况下，执行各自队列里面的命令。记住，Redis 要在接收到 EXEC 命令之后，才会执行那些位于 MULTI 和 EXEC 之间的人队命令。

使用事务既有利也有弊，本书的 4.4 节将对这个问题进行讨论。

### 练习：移除竞争条件

正如前面的代码清单 3-13 所示，MULTI 和 EXEC 事务的一个主要作用是移除竞争条件。第 1 章展示的 article\_vote() 函数包含一个竞争条件以及一个因为竞争条件而出现的 bug。函数的竞争条件可能会造成内存泄漏，而函数的 bug 则可能会导致不正确的投票结果出现。尽管 article\_vote() 函数的竞争条件和 bug 出现的机会都非常少，但为了防范于未然，你能想个办法修复它们么？提示：如果你觉得很难理解竞争条件为什么会导致内存泄漏，那么可以在分析第 1 章的 post\_article() 函数的同时，阅读一下 6.2.5 节。

### 练习：提高性能

在 Redis 里面使用流水线的另一个目的是提高性能（详细的信息会在之后的 4.4 节至 4.6 节中介绍）。在执行一连串命令时，减少 Redis 与客户端之间的通信往返次数可以大幅降低客户端等待回复所需的时间。第 1 章的 get\_articles() 函数在获取整个页面的文章时，需要在 Redis 与客户端之间进行 26 次通信往返，这种做法简直低效得令人发指，你能否想个办法将 get\_articles() 函数的往返次数从 26 次降低为 2 次呢？

在使用 Redis 存储数据的时候，有些数据仅在一段很短的时间内有用，虽然我们可以在数据的有效期过了之后手动删除无用的数据，但更好的办法是使用 Redis 提供的键过期操作来自动删除无用数据。

### 3.7.3 键的过期时间

在使用 Redis 存储数据的时候，有些数据可能在某个时间点之后就不再有用了，用户可以使用 DEL 命令显式地删除这些无用数据，也可以通过 Redis 的过期时间（expiration）特性来让一个键在给定的时限（timeout）之后自动被删除。当我们说一个键“带有生存时间（time to live）”或者一个键“会在特定时间之后过期（expire）”时，我们指的是 Redis 会在这个键的过期时间到达时自动删除该键。

虽然过期时间特性对于清理缓存数据非常有用，不过如果读者翻一下本书的其他章节，就会发现除了 6.2 节、7.1 节和 7.2 节之外，本书使用过期时间特性的情况并不多，这主要和本书使用的结构类型有关。在本书常用的命令当中，只有少数几个命令可以原子地为键设置过期时间，并且对于列表、集合、散列和有序集合这样的容器（container）来说，键过期命令只能为整个键设置过期时间，而没办法为键里面的单个元素设置过期时间（为了解决这个问题，本书在好几个地方都使用了存储时间戳的有序集合来实现针对单个元素的过期操作）。

本节将对那些可以在给定时限或者给定时间之后，自动删除过期键的 Redis 命令进行介绍。通过阅读本节，读者将学会如何使用过期操作来自动地删除过期数据并降低 Redis 的内存占用。

表 3-13 列出了 Redis 提供的用于为键设置过期时间的命令，以及查看键的过期时间的命令。

表 3-13 用于处理过期时间的 Redis 命令

命令	示例和描述
PERSIST	PERSIST key-name——移除键的过期时间
TTL	TTL key-name——查看给定键距离过期还有多少秒
EXPIRE	EXPIRE key-name seconds——让给定键在指定的秒数之后过期
EXPIREAT	EXPIREAT key-name timestamp——将给定键的过期时间设置为给定的 UNIX 时间戳
PTTL	PTTL key-name——查看给定键距离过期时间还有多少毫秒，这个命令在 Redis 2.6 或以上版本可用
PEXPIRE	PEXPIRE key-name milliseconds——让给定键在指定的毫秒数之后过期，这个命令在 Redis 2.6 或以上版本可用
PEXPIREAT	PEXPIREAT key-name timestamp-milliseconds——将一个毫秒级精度的 UNIX 时间戳设置为给定键的过期时间，这个命令在 Redis 2.6 或以上版本可用

代码清单 3-15 展示了几个对键执行过期时间操作的例子。

#### 代码清单 3-15 展示 Redis 中过期时间相关的命令的使用方法

```
>>> conn.set('key', 'value')
True
>>> conn.get('key')
'value'

>>> conn.expire('key', 2)
True
>>> time.sleep(2)
>>> conn.get('key')
>>> conn.set('key', 'value2')
True
>>> conn.expire('key', 100); conn.ttl('key')
True
100
```

设置一个简单的字符串值，作为过期时间的设置对象。

如果我们为键设置了过期时间，并在键过期后尝试获取键的值，那么就会发现键已经被删除了。

查看键距离过期还有多长时间。

#### 练习：使用 **EXPIRE** 命令代替时间戳有序集合

2.1 节、2.2 节和 2.5 节中使用了一个根据时间戳进行排序、用于清除会话 ID 的有序集合，通过这个有序集合，程序可以在清理会话的时候，对用户浏览过的商品以及用户购物车里面的商品进行分析。但是，如果我们决定不对商品进行分析的话，那么就可以使用 Redis 提供的过期时间操作来自动清理过期的会话 ID，而无须使用清理函数。那么，你能否想办法修改在第 2 章定义的 `update_token()` 函数和 `add_to_cart()` 函数，让它们使用过期时间操作来删除会话 ID，从而代替目前使用有序集合来记录并清除会话 ID 的做法呢？

## 3.8 小结

本章对 Redis 最常用的一些命令进行了介绍，其中包括各种不同数据类型的常用命令、PUBLISH 命令和 SUBSCRIBE 命令、SORT 命令、两个事务命令 MULTI 和 EXEC，以及与过期时间有关的几个命令。

本章的第一个目标是让读者知道——Redis 为每种结构都提供了大量的处理命令，本章只展示了其中最重要的 70 多个命令，其余的命令可以在 <http://redis.io/commands> 看到。

本章的第二个目标是让读者知道——本书并非为每个问题都提供了完美的答案。通过在练习里面对第 1 章和第 2 章展示的示例进行回顾（练习的答案在本书附带的源码里面），本书向读者提供了一个机会，让读者把已经不错的代码变得更好，或者变得更适合于读者自己的问题。

在本章没有介绍到的命令当中，有一大部分都是与配置相关的，接下来的一章将向读者介绍如何配置 Redis 以确保数据安全，以及如何确保 Redis 拥有良好的性能。

# 第4章 数据安全与性能保障

## 本章主要内容

- 将数据持久化至硬盘
- 将数据复制至其他机器
- 处理系统故障
- Redis 事务
- 非事务型流水线（non-transactional pipeline）
- 诊断性能问题

前面的几章介绍了各式各样的 Redis 命令以及使用这些命令来操作数据结构的方法，还列举了几个使用 Redis 来解决实际问题的例子。为了让读者做好使用 Redis 构建真实软件的准备，本章将展示维护数据安全以及应对系统故障的方法。另外，本章还会介绍一些能够在保证数据完整性的同时提升 Redis 性能的方法。

本章首先会介绍 Redis 的各个持久化选项，这些选项可以让用户将自己的数据存储到硬盘上面。接着本章将介绍如何通过 Redis 的复制特性，把不断更新的数据副本存储到附加的机器上面，从而提升系统的性能和数据的可靠性。之后本章将会说明同时使用复制和持久化的好处和坏处，并通过一些例子来告诉读者应该如何去选择适合自己的持久化选项和复制选项。最后本章将对 Redis 的事务特性和流水线特性进行介绍，并讨论如何诊断某些性能问题。

阅读这一章的重点是要弄懂更多的 Redis 运作原理，从而学会如何在首先保证数据正确的前提下，加快数据操作的执行速度。

现在，让我们来看看 Redis 是如何将数据存储到硬盘里面，使得数据在 Redis 重启之后仍然存在的。

## 4.1 持久化选项

Redis 提供了两种不同的持久化方法来将数据存储到硬盘里面。一种方法叫快照（snapshotting），

它可以将存在于某一时刻的所有数据都写入硬盘里面。另一种方法叫只追加文件（append-only file，AOF），它会在执行写命令时，将被执行的写命令复制到硬盘里面。这两种持久化方法既可以同时使用，又可以单独使用，在某些情况下甚至可以两种方法都不使用，具体选择哪种持久化方法需要根据用户的数据以及应用来决定。

将内存中的数据存储到硬盘的一个主要原因是为了在之后重用数据，或者是为了防止系统故障而将数据备份到一个远程位置。另外，存储在 Redis 里面的数据有可能是经过长时间计算得出的，或者有程序正在使用 Redis 存储的数据进行计算，所以用户会希望自己可以将这些数据存储起来以便之后使用，这样就不必再重新计算了。对于一些 Redis 应用来说，“计算”可能只是简单地将另一个数据库的数据复制到 Redis 里面（2.4 节中就介绍过这样的例子），但对于另外一些 Redis 应用来说，Redis 存储的数据可能是根据数十亿行日志进行聚合分析得出的结果。

两组不同的配置选项控制着 Redis 将数据写入硬盘里面的方式，代码清单 4-1 展示了这些配置选项以及它们的示例配置值。因为之后的 4.1.1 节和 4.1.2 节会更详细地介绍这些选项，所以目前我们只要稍微了解一下这些选项就可以了。

#### 代码清单 4-1 Redis 提供的持久化配置选项

```
save 60 1000
stop-writes-on-bgsave-error no
rdbcompression yes
dbfilename dump.rdb

appendonly no
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
dir ./
```

快照持久化选项。

AOF 持久化  
选项。

共享选项，这个选项决定了快照文件和 AOF 文件的保存位置。

代码清单 4-1 最开头的几个选项和快照持久化有关，比如：如何命名硬盘上的快照文件、多久执行一次自动快照操作、是否对快照文件进行压缩，以及在创建快照失败后是否仍然继续执行写命令。代码清单的第二组选项用于配置 AOF 子系统（subsystem）：这些选项告诉 Redis 是否使用 AOF 持久化、多久才将写入的内容同步到硬盘、在对 AOF 进行压缩（compaction）的时候能否执行同步操作，以及多久执行一次 AOF 压缩。接下来的一节将介绍如何使用快照来保持数据安全。

### 4.1.1 快照持久化

Redis 可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。在创建快照之后，用户可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器。

副本，还可以将快照留在原地以便重启服务器时使用。

根据配置，快照将被写入 `dbfilename` 选项指定的文件里面，并储存在 `dir` 选项指定的路径上面。如果在新的快照文件创建完毕之前，Redis、系统或者硬件这三者之中的任意一个崩溃了，那么 Redis 将丢失最近一次创建快照之后写入的所有数据。

举个例子，假设 Redis 目前在内存里面存储了 10GB 的数据，上一个快照是在下午 2:35 开始创建的，并且已经创建成功。下午 3:06 时，Redis 又开始创建新的快照，并且在下午 3:08 快照文件创建完毕之前，有 35 个键进行了更新。如果在下午 3:06 至下午 3:08 期间，系统发生崩溃，导致 Redis 无法完成新快照的创建工作，那么 Redis 将丢失下午 2:35 之后写入的所有数据。另一方面，如果系统恰好在新的快照文件创建完毕之后崩溃，那么 Redis 将只丢失 35 个键的更新数据。

创建快照的办法有以下几种。

- 客户端可以通过向 Redis 发送 `BGSAVE` 命令来创建一个快照。对于支持 `BGSAVE` 命令的平台来说（基本上所有平台都支持，除了 Windows 平台），Redis 会调用 `fork`<sup>①</sup> 来创建一个子进程，然后子进程负责将快照写入硬盘，而父进程则继续处理命令请求。
- 客户端还可以通过向 Redis 发送 `SAVE` 命令来创建一个快照，接到 `SAVE` 命令的 Redis 服务器在快照创建完毕之前将不再响应任何其他命令。`SAVE` 命令并不常用，我们通常只会在没有足够内存去执行 `BGSAVE` 命令的情况下，又或者即使等待持久化操作执行完毕也无所谓的情况下，才会使用这个命令。
- 如果用户设置了 `save` 配置选项，比如 `save 60 10000`，那么从 Redis 最近一次创建快照之后开始算起，当“60 秒之内有 10 000 次写入”这个条件被满足时，Redis 就会自动触发 `BGSAVE` 命令。如果用户设置了多个 `save` 配置选项，那么当任意一个 `save` 配置选项所设置的条件被满足时，Redis 就会触发一次 `BGSAVE` 命令。
- 当 Redis 通过 `SHUTDOWN` 命令接收到关闭服务器的请求时，或者接收到标准 `TERM` 信号时，会执行一个 `SAVE` 命令，阻塞所有客户端，不再执行客户端发送的任何命令，并在 `SAVE` 命令执行完毕之后关闭服务器。
- 当一个 Redis 服务器连接另一个 Redis 服务器，并向对方发送 `SYNC` 命令来开始一次复制操作的时候，如果主服务器目前没有在执行 `BGSAVE` 操作，或者主服务器并非刚刚执行完 `BGSAVE` 操作，那么主服务器就会执行 `BGSAVE` 命令。更多有关复制的信息请参考 4.2 节。

在只使用快照持久化来保存数据时，一定要记住：如果系统真的发生崩溃，用户将丢失最近一次生成快照之后更改的所有数据。因此，快照持久化只适用于那些即使丢失一部分数据也不会造成问题的应用程序，而不能接受这种数据损失的应用程序则可以考虑使用 4.1.2 节中介紹的其他持久化方法。

<sup>①</sup> 当一个进程创建子进程的时候，底层的操作系统会创建该进程的一个副本。在 Unix 和类 Unix 系统上面，创建子进程的操作会进行如下优化：在刚开始的时候，父子进程共享相同的内存，直到父进程或者子进程对内存进行了写入之后，对被写入内存的共享才会结束。

绍的 AOF 持久化。接下来将展示几个使用快照持久化的场景，读者可以从中学习到如何通过修改配置来获得自己想要的快照持久化行为。

## 1. 个人开发

在个人开发服务器上面，我主要考虑的是尽可能地降低快照持久化带来的资源消耗。基于这个原因以及对自己硬件的信任，我只设置了 `save 900 1` 这一条规则。其中 `save` 选项告知 Redis，它应该根据这个选项提供的两个值来执行 `BGSAVE` 操作。在这个规则设置下，如果服务器距离上次成功生成快照已经超过了 900 秒（也就是 15 分钟），并且在此期间执行了至少一次写入操作，那么 Redis 就会自动开始一次新的 `BGSAVE` 操作。

如果你打算在生产服务器中使用快照持久化并存储大量数据，那么你的开发服务器最好能够运行在与生产服务器相同或者相似的硬件上面，并在这两个服务器上使用相同的 `save` 选项、存储相似的数据集并处理相近的负载量。把开发环境设置得尽量贴近生产环境，有助于判断快照是否生成得过于频繁或者过于稀少（过于频繁会浪费资源，而过于稀少则带有丢失大量数据的隐患）。

## 2. 对日志进行聚合计算

在对日志文件进行聚合计算或者对页面浏览量进行分析的时候，我们唯一需要考虑的就是：如果 Redis 因为崩溃而未能成功创建新的快照，那么我们能够承受丢失多长时间以内产生的新数据。如果丢失一个小时之内产生的数据是可以被接受的，那么可以使用配置值 `save 3600 1`（`3600` 为一小时的秒数）。在决定好了持久化配置值之后，另一个需要解决的问题就是如何恢复因为故障而被中断的日志处理操作。

在进行数据恢复时，首先要做的就是弄清楚我们丢失了哪些数据。为了弄明白这一点，我们需要在处理日志的同时记录被处理日志的相关信息。代码清单 4-2 展示了一个用于处理新日志的函数，该函数有 3 个参数，它们分别是：一个 Redis 连接；一个存储日志文件的路径；待处理日志文件中各个行（line）的回调函数（callback）。这个函数可以在处理日志文件的同时，记录被处理日志文件的名字以及偏移量。

代码清单 4-2 `process_logs()` 函数会将被处理日志的信息存储到 Redis 里面

获取文件当前的  
处理进度。

```
def process_logs(conn, path, callback):  
    current_file, offset = conn.mget(  
        'progress:file', 'progress:position')  
    pipe = conn.pipeline()
```

日志处理函数接受的  
其中一个参数为回调函数，  
这个回调函数接受一个  
Redis 连接和一个日志行  
作为参数，并通过调用流  
水线对象的方法来执行  
Redis 命令。

```

通过使用闭包 (closure) 来减少重复代码。    def update_progress():
    pipe.mset({
        'progress:file': fname,
        'progress:position': offset
    })
    pipe.execute()

for fname in sorted(os.listdir(path)):
    if fname < current_file:
        continue

    inp = open(os.path.join(path, fname), 'rb')
    if fname == current_file:
        inp.seek(int(offset), 10)
    else:
        offset = 0
    current_file = None

    for lno, line in enumerate(inp):
        callback(pipe, line)
        offset += int(offset) + len(line)

        if not (lno+1) % 1000:
            update_progress()
            update_progress()

    inp.close()

```

这个语句负责执行实际的日志更新操作，并将日志文件的名字和目前的处理进度记录到 Redis 里面。

在接着处理一个因为系统崩溃而未能完成处理的日志文件时，略过已处理的内容。

枚举函数遍历一个由文件行组成的序列，并返回任意多个二元组，每个二元组包含了行号 lno 和行数据 line，其中行号从 0 开始。

更新正在处理的日志文件的名字和偏移量。

有序地遍历各个日志文件。

略过所有已处理的日志文件。

处理日志行。

更新已处理内容的偏移量。

每当处理完 1000 个日志行或者处理完整个日志文件的时候，都更新一次文件的处理进度。

通过将日志的处理进度记录到 Redis 里面，程序可以在系统崩溃之后，根据进度记录继续执行之前未完成的处理工作。而通过使用第 3 章介绍的事务流水线，程序保证日志的处理结果和处理进度总是会同时被记录到文件里面。

### 3. 大数据

当 Redis 存储的数据量只有几个 GB 的时候，使用快照来保存数据是没有问题的。Redis 会创建子进程并将数据保存到硬盘里面，生成快照所需的时间比你读这句话所需的时间还要短。但随着 Redis 占用的内存越来越多，BGSAVE 在创建子进程时耗费的时间也会越来越多。如果 Redis 的内存占用量达到数十个 GB，并且剩余的空闲内存并不多，或者 Redis 运行在虚拟机（virtual machine）上面，那么执行 BGSAVE 可能会导致系统长时间地停顿，也可能引发系统大量地使用虚拟内存（virtual memory），从而导致 Redis 的性能降低至无法使用的程度。

执行 BGSAVE 而导致的停顿时间有多长取决于 Redis 所在的系统：对于真实的硬件、VMWare 虚拟机或者 KVM 虚拟机来说，Redis 进程每占用一个 GB 的内存，创建该进程的子进程所需的时间就要增加 10~20 毫秒；而对于 Xen 虚拟机来说，根据配置的不同，Redis 进程每占用一个 GB 的内存，创建该进程的子进程所需的时间就要增加 200~300 毫秒。因此，如果我们的 Redis 进程占用了 20 GB 的内存，那么在标准硬件上运行 BGSAVE 所创建的子进程将导致 Redis 停顿 200~400 毫秒；如果我们使用的是 Xen 虚拟机（亚马逊 EC2 和其他几个云计算供应商都使用这

种虚拟机），那么相同的创建子进程操作将导致 Redis 停顿 4~6 秒。用户必须考虑自己的应用程序能否接受这种停顿。

为了防止 Redis 因为创建子进程而出现停顿，我们可以考虑关闭自动保存，转而通过手动发送 BGSAVE 或者 SAVE 来进行持久化。手动发送 BGSAVE 一样会引起停顿，唯一不同的是用户可以通过手动发送 BGSAVE 命令来控制停顿出现的时间。另一方面，虽然 SAVE 会一直阻塞 Redis 直到快照生成完毕，但是因为它不需要创建子进程，所以就不会像 BGSAVE 一样因为创建子进程而导致 Redis 停顿；并且因为没有子进程在争抢资源，所以 SAVE 创建快照的速度会比 BGSAVE 创建快照的速度要来得更快一些。

根据我的个人经验，在一台拥有 68 GB 内存的 Xen 虚拟机上面，对一个占用 50 GB 内存的 Redis 服务器执行 BGSAVE 命令的话，光是创建子进程就需要花费 15 秒以上，而生成快照则需要花费 15~20 分钟；但使用 SAVE 只需要 3~5 分钟就可以完成快照的生成工作。因为我的应用程序只需要每天生成一次快照，所以我写了一个脚本，让它在每天凌晨 3 点停止所有客户端对 Redis 的访问，调用 SAVE 命令并等待该命令执行完毕，之后备份刚刚生成的快照文件，并通知客户端继续执行操作。

如果用户能够妥善地处理快照持久化可能会带来的大量数据丢失，那么快照持久化对用户来说将是一个不错的选择，但对于很多应用程序来说，丢失 15 分钟、1 小时甚至更长时间的数据都是不可接受的，在这种情况下，我们可以使用 AOF 持久化来将存储在内存里面的数据尽快地保存到硬盘里面。

### 4.1.2 AOF 持久化

简单来说，AOF 持久化会将被执行的写命令写到 AOF 文件的末尾，以此来记录数据发生的变化。因此，Redis 只要从头到尾重新执行一次 AOF 文件包含的所有写命令，就可以恢复 AOF 文件所记录的数据集。AOF 持久化可以通过设置代码清单 4-1 所示的 appendonly yes 配置选项来打开。表 4-1 展示了 appendfsync 配置选项对 AOF 文件的同步频率的影响。

**文件同步** 在向硬盘写入文件时，至少会发生 3 件事。当调用 `file.write()` 方法（或者其他编程语言里面的类似操作）对文件进行写入时，写入的内容首先会被存储到缓冲区，然后操作系统会在将来的某个时候将缓冲区存储的内容写入硬盘，而数据只有在被写入硬盘之后，才算是真正地保存到了硬盘里面。用户可以通过调用 `file.flush()` 方法来请求操作系统尽快地将缓冲区存储的数据写入硬盘里，但具体何时执行写入操作仍然由操作系统决定。除此之外，用户还可以命令操作系统将文件同步（sync）到硬盘，同步操作会一直阻塞直到指定的文件被写入硬盘为止。当同步操作执行完毕之后，即使系统出现故障也不会对被同步的文件造成任何影响。

表 4-1 `appendfsync` 选项及同步频率

选 项	同 步 频 率
always	每个 Redis 写命令都要同步写入硬盘。这样做会严重降低 Redis 的速度
everysec	每秒执行一次同步，显式地将多个写命令同步到硬盘
no	让操作系统来决定应该何时进行同步

如果用户使用 `appendfsync always` 选项的话，那么每个 Redis 写命令都会被写入硬盘，从而将发生系统崩溃时出现的数据丢失减到最少。不过遗憾的是，因为这种同步策略需要对硬盘进行大量写入，所以 Redis 处理命令的速度会受到硬盘性能的限制：转盘式硬盘（spinning disk）在这种同步频率下每秒只能处理大约 200 个写命令，而固态硬盘（solid-state drive，SSD）每秒大概也只能处理几万个写命令。

**警告：** 固态硬盘和 `appendfsync always` 使用固态硬盘的用户请谨慎使用 `appendfsync always` 选项，因为这个选项让 Redis 每次只写入一个命令，而不是像其他 `appendfsync` 选项那样一次写入多个命令，这种不断地写入少量数据的做法有可能会引发严重的写入放大（write amplification）问题，在某些情况下甚至会将固态硬盘的寿命从原来的几年降低为几个月。

为了兼顾数据安全和写入性能，用户可以考虑使用 `appendfsync everysec` 选项，让 Redis 以每秒一次的频率对 AOF 文件进行同步。Redis 每秒同步一次 AOF 文件时的性能和不使用任何持久化特性时的性能相差无几，而通过每秒同步一次 AOF 文件，Redis 可以保证，即使出现系统崩溃，用户也最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis 还会优雅地放慢自己的速度以便适应硬盘的最大写入速度。

最后，如果用户使用 `appendfsync no` 选项，那么 Redis 将不对 AOF 文件执行任何显式的同步操作，而是由操作系统来决定应该在何时对 AOF 文件进行同步。这个选项在一般情况下不会对 Redis 的性能带来影响，但系统崩溃将导致使用这种选项的 Redis 服务器丢失不定数量的数据。另外，如果用户的硬盘处理写入操作的速度不够快的话，那么当缓冲区被等待写入硬盘的数据填满时，Redis 的写入操作将被阻塞，并导致 Redis 处理命令请求的速度变慢。因为这个原因，一般来说并不推荐使用 `appendfsync no` 选项，在这里介绍它只是为了完整列举 `appendfsync` 选项可用的 3 个值。

虽然 AOF 持久化非常灵活地提供了多种不同的选项来满足不同应用程序对数据安全的不同要求，但 AOF 持久化也有缺陷——那就是 AOF 文件的体积大小。

### 4.1.3 重写/压缩 AOF 文件

在阅读了上一节对 AOF 持久化的介绍之后，读者可能会感到疑惑：AOF 持久化既可以将丢

失数据的时间窗口降低至 1 秒（甚至不丢失任何数据），又可以在极短的时间内完成定期的持久化操作，那么我们有什么理由不使用 AOF 持久化呢？但是这个问题实际上并没有那么简单，因为 Redis 会不断地将被执行的写命令记录到 AOF 文件里面，所以随着 Redis 不断运行，AOF 文件的体积也会不断增长，在极端情况下，体积不断增大的 AOF 文件甚至可能会用完硬盘的所有可用空间。还有另一个问题就是，因为 Redis 在重启之后需要通过重新执行 AOF 文件记录的所有写命令来还原数据集，所以如果 AOF 文件的体积非常大，那么还原操作执行的时间就可能会非常长。

为了解决 AOF 文件体积不断增大的问题，用户可以向 Redis 发送 BGREWRITEAOF 命令，这个命令会通过移除 AOF 文件中的冗余命令来重写（rewrite）AOF 文件，使 AOF 文件的体积变得尽可能地小。BGREWRITEAOF 的工作原理和 BGSAVE 创建快照的工作原理非常相似：Redis 会创建一个子进程，然后由子进程负责对 AOF 文件进行重写。因为 AOF 文件重写也需要用到子进程，所以快照持久化因为创建子进程而导致的性能问题和内存占用问题，在 AOF 持久化中也同样存在。更糟糕的是，如果不加以控制的话，AOF 文件的体积可能会比快照文件的体积大好几倍，在进行 AOF 重写并删除旧 AOF 文件的时候，删除一个体积达到数十 GB 大的旧 AOF 文件可能会导致操作系统挂起（hang）数秒。

跟快照持久化可以通过设置 save 选项来自动执行 BGSAVE 一样，AOF 持久化也可以通过设置 auto-aof-rewrite-percentage 选项和 auto-aof-rewrite-min-size 选项来自动执行 BGREWRITEAOF。举个例子，假设用户对 Redis 设置了配置选项 auto-aof-rewrite-percentage 100 和 auto-aof-rewrite-min-size 64mb，并且启用了 AOF 持久化，那么当 AOF 文件的体积大于 64 MB，并且 AOF 文件的体积比上一次重写之后的体积大了至少一倍（100%）的时候，Redis 将执行 BGREWRITEAOF 命令。如果 AOF 重写执行得过于频繁的话，用户可以考虑将 auto-aof-rewrite-percentage 选项的值设置为 100 以上，这种做法可以让 Redis 在 AOF 文件的体积变得更大之后才执行重写操作，不过也会让 Redis 在启动时还原数据集所需的时间变得更长。

无论是使用 AOF 持久化还是快照持久化，将数据持久化到硬盘上都是非常有必要的，但除了进行持久化之外，用户还必须对持久化所得的文件进行备份（最好是备份到多个不同的地方），这样才能尽量避免数据丢失事故发生。如果条件允许的话，最好能将快照文件和最新重写的 AOF 文件备份到不同的服务器上面。

通过使用 AOF 持久化或者快照持久化，用户可以在系统重启或者崩溃的情况下仍然保留数据。随着负载量的上升，或者数据的完整性变得越来越重要时，用户可能需要使用复制特性。

## 4.2 复制

对于有扩展平台以适应更高负载经验的工程师和管理员来说，复制（replication）是不可或缺的。复制可以让其他服务器拥有一个不断地更新的数据副本，从而使得拥有数据副本的服务器

可以用于处理客户端发送的读请求。关系数据库通常会使用一个主服务器（master）向多个从服务器（slave）发送更新，并使用从服务器来处理所有读请求。Redis 也采用了同样的方法来实现自己的复制特性，并将其用作扩展性能的一种手段。本节将对 Redis 的复制配置选项进行讨论，并说明 Redis 在进行复制时的各个步骤。

尽管 Redis 的性能非常优秀，但它也会遇上没办法快速地处理请求的情况，特别是在对集合和有序集合进行操作的时候，涉及的元素可能会有上万个甚至上百万个，在这种情况下，执行操作所花费的时间可能需要以秒来进行计算，而不是毫秒或者微秒。但即使一个命令只需要花费 10 毫秒就能完成，单个 Redis 实例（instance）1 秒也只能处理 100 个命令。

**SUNIONSTORE 命令的性能** 作为对 Redis 性能的一个参考，在主频为 2.4 GHz 的英特尔酷睿 2 处理器上，对两个分别包含 10 000 个元素的集合执行 SUNIONSTORE 命令并产生一个包含 20 000 个元素的结果集合，需要花费 Redis 七八毫秒的时间。

在需要扩展读请求的时候，或者在需要写入临时数据的时候（第 7 章对此有详细的介绍），用户可以通过设置额外的 Redis 从服务器来保存数据集的副本。在接收到主服务器发送的数据初始副本（initial copy of the data）之后，客户端每次向主服务器进行写入时，从服务器都会实时地得到更新。在部署好主从服务器之后，客户端就可以向任意一个从服务器发送读请求了，而不必再像之前一样，总是把每个读请求都发送给主服务器（客户端通常会随机地选择使用哪个从服务器，从而将负载平均分配到各个从服务器上）。

接下来的一节将介绍配置 Redis 主从服务器的方法，并说明 Redis 在整个复制过程中所做的各项操作。

### 4.2.1 对 Redis 的复制相关选项进行配置

4.1.1 节中曾经介绍过，当从服务器连接主服务器的时候，主服务器会执行 BGSAVE 操作。因此为了正确地使用复制特性，用户需要保证主服务器已经正确地设置了代码清单 4-1 里面列出的 dir 选项和 dbfilename 选项，并且这两个选项所指示的路径和文件对于 Redis 进程来说都是可写的（writable）。

尽管有多个不同的选项可以控制从服务器自身的行为，但开启从服务器所必须的选项只有 slaveof 一个。如果用户在启动 Redis 服务器的时候，指定了一个包含 slaveof host port 选项的配置文件，那么 Redis 服务器将根据该选项给定的 IP 地址和端口号来连接主服务器。对于一个正在运行的 Redis 服务器，用户可以通过发送 SLAVEOF no one 命令来让服务器终止复制操作，不再接受主服务器的数据更新；也可以通过发送 SLAVEOF host port 命令来让服务器开始复制一个新的主服务器。

开启 Redis 的主从复制特性并不需要进行太多的配置，但了解 Redis 服务器是如何变成主服

务器或者从服务器的，对于我们来说将是非常有用和有趣的过程。

### 4.2.2 Redis 复制的启动过程

本章前面曾经说过，从服务器在连接一个主服务器的时候，主服务器会创建一个快照文件并将其发送至从服务器，但这只是主从复制执行过程的其中一步。表 4-2 完整地列出了当从服务器连接主服务器时，主从服务器执行的所有操作。

表 4-2 从服务器连接主服务器时的步骤

步 骤	主服务器操作	从服务器操作
1	(等待命令进入)	连接（或者重连接）主服务器，发送 SYNC 命令
2	开始执行 BGSAVE，并使用缓冲区记录 BGSAVE 之后执行的所有写命令	根据配置选项来决定是继续使用现有的数据（如果有的话）来处理客户端的命令请求，还是向发送请求的客户端返回错误
3	BGSAVE 执行完毕，向从服务器发送快照文件，并在发送期间继续使用缓冲区记录被执行的写命令	丢弃所有旧数据（如果有的话），开始载入主服务器发来的快照文件
4	快照文件发送完毕，开始向从服务器发送存储在缓冲区里面的写命令	完成对快照文件的解释操作，像往常一样开始接受命令请求
5	缓冲区存储的写命令发送完毕；从现在开始，每执行一个写命令，就向从服务器发送相同的写命令	执行主服务器发来的所有存储在缓冲区里面的写命令；并从现在开始，接收并执行主服务器传来的每个写命令

通过使用表 4-2 所示的办法，Redis 在复制进行期间也会尽可能地处理接收到的命令请求，但是，如果主从服务器之间的网络带宽不足，或者主服务器没有足够的内存来创建子进程和创建记录写命令的缓冲区，那么 Redis 处理命令请求的效率就会受到影响。因此，尽管这并不是必须的，但在实际中最好还是让主服务器只使用 50%~65% 的内存，留下 30%~45% 的内存用于执行 BGSAVE 命令和创建记录写命令的缓冲区。

设置从服务器的步骤非常简单，用户既可以通过配置选项 SLAVEOF host port 来将一个 Redis 服务器设置为从服务器，又可以通过向运行中的 Redis 服务器发送 SLAVEOF 命令来将其设置为从服务器。如果用户使用的是 SLAVEOF 配置选项，那么 Redis 在启动时首先会载入当前可用的任何快照文件或者 AOF 文件，然后连接主服务器并执行表 4-2 所示的复制过程。如果用户使用的是 SLAVEOF 命令，那么 Redis 会立即尝试连接主服务器，并在连接成功之后，开始表 4-2 所示的复制过程。

从服务器在进行同步时，会清空自己的所有数据 因为有些用户在第一次使用从服务器时会忘记这件事，所以这里要特别提醒一下：从服务器在与主服务器进行初始连接时，数据库中原有的所有数据都将丢失，并被替换成主服务器发来的数据。

**警告:** Redis 不支持主主复制 (master-master replication) 因为 Redis 允许用户在服务器启动之后使用 SLAVEOF 命令来设置从服务器选项 (slaving options), 所以可能会有读者误以为可以通过将两个 Redis 实例互相设置为对方的主服务器来实现多主复制 (multi-master replication) (甚至可能会在一个循环里面将多个实例互相设置为主服务器)。遗憾的是, 这种做法是行不通的: 被互相设置为主服务器的两个 Redis 实例只会持续地占用大量处理器资源并且连续不断地尝试与对方进行通信, 根据客户端连接的服务器的不同, 客户端的请求可能会得到不一致的数据, 或者完全得不到数据。

当多个从服务器尝试连接同一个主服务器的时候, 就会出现表 4-3 所示的两种情况中的其中一种。

表 4-3 当一个从服务器连接一个已有的主服务器时, 有时可以重用已有的快照文件

当有新的从服务器连接主服务器时	主服务器的操作
表 4-2 的步骤 3 尚未执行	所有从服务器都会接收到相同的快照文件和相同的缓冲区写命令
表 4-2 的步骤 3 正在执行或者已经执行完毕	当主服务器与较早进行连接的从服务器执行完复制所需的 5 个步骤之后, 主服务器会与新连接的从服务器执行一次新的步骤 1 至步骤 5

在大部分情况下, Redis 都会尽可能地减少复制所需的工作, 然而, 如果从服务器连接主服务器的时间并不凑巧, 那么主服务器就需要多做一些额外的工作。另一方面, 当多个从服务器同时连接主服务器的时候, 同步多个从服务器所占用的带宽可能会使得其他命令请求难以传递给主服务器, 与主服务器位于同一网络中的其他硬件的网速可能也会因此而降低。

### 4.2.3 主从链

有些用户发现, 创建多个从服务器可能会造成网络不可用——当复制需要通过互联网进行或者需要在不同数据中心之间进行时, 尤为如此。因为 Redis 的主服务器和从服务器并没有特别不同的地方, 所以从服务器也可以拥有自己的从服务器, 并由此形成主从链 (master/slave chaining)。

从服务器对从服务器进行复制在操作上和从服务器对主服务器进行复制的唯一区别在于, 如果从服务器 X 拥有从服务器 Y, 那么当从服务器 X 在执行表 4-2 中的步骤 4 时, 它将断开与从服务器 Y 的连接, 导致从服务器 Y 需要重新连接并重新同步 (resync)。

当读请求的重要性明显高于写请求的重要性, 并且读请求的数量远远超出一台 Redis 服务器可以处理的范围时, 用户就需要添加新的从服务器来处理读请求。随着负载不断上升, 主服务器可能会无法快速地更新所有从服务器, 或者因为重新连接和重新同步从服务器而导致系统超载。为了缓解这个问题, 用户可以创建一个由 Redis 主从节点 (master/slave node) 组成的中间层来分担主服务器的复制工作, 如图 4-1 所示。



图 4-1 一个 Redis 主从复制树 (master/slave replica tree) 示例，树的中层  
有 3 个帮助开展复制工作的服务器，底层有 9 个从服务器

尽管主从服务器之间并不一定要像图 4-1 那样组成一个树状结构，但记住并理解这种树状结构对于 Redis 复制来说是可行的 (possible) 并且是合理的 (reasonable) 将有助于读者理解之后的内容。本书在前面的 4.1.2 节中曾经介绍过，AOF 持久化的同步选项可以控制数据丢失的时间长度：通过将每个写命令同步到硬盘里面，用户几乎可以不损失任何数据（除非系统崩溃或者硬盘驱动器损坏），但这种做法会对服务器的性能造成影响；另一方面，如果用户将同步的频率设置为每秒一次，那么服务器的性能将回到正常水平，但故障可能会造成 1 秒的数据丢失。通过同时使用复制和 AOF 持久化，我们可以将数据持久化到多台机器上面。

为了将数据保存到多台机器上面，用户首先需要为主服务器设置多个从服务器，然后对每个从服务器设置 `appendonly yes` 选项和 `appendfsync everysec` 选项（如果有需要的话，也可以对主服务器进行相同的设置），这样的话，用户就可以让多台服务器以每秒一次的频率将数据同步到硬盘上了。但这还只是第一步：因为用户还必须等待主服务器发送的写命令到达从服务器，并且在执行后续操作之前，检查数据是否已经被同步到了硬盘里面。

#### 4.2.4 检验硬盘写入

为了验证主服务器是否已经将写数据发送至从服务器，用户需要在向主服务器写入真正的数据之后，再向主服务器写入一个唯一的虚构值 (unique dummy value)，然后通过检查虚构值是否存在与从服务器来判断写数据是否已经到达从服务器，这个操作很容易就可以实现。另一方面，判断数据是否已经被保存到硬盘里面则要困难得多。对于每秒同步一次 AOF 文件的 Redis 服务器来说，用户总是可以通过等待 1 秒来确保数据已经被保存到硬盘里面；但更节约时间的做法是，检查 `INFO` 命令的输出结果中 `aof_pending_bio_fsync` 属性的值是否为 0，如果是的话，那么就表示服务器已经将已知的所有数据都保存到硬盘里面了。在向主服务器写入数据之后，用户可以将主服务器和从服务器的连接作为参数，调用代码清单 4-3 所示的函数来自动进行上述的检查操作。

代码清单 4-3 `wait_for_sync()` 函数

```

def wait_for_sync(mconn, sconn):
    identifier = str(uuid.uuid4())
    mconn.zadd('sync:wait', identifier, time.time()) ← 将令牌添加至主服务器。

    while not sconn.info()['master_link_status'] != 'up': ← 如果有必要的话，等待
        time.sleep(.001)                                     从服务器完成同步。

    while not sconn.zscore('sync:wait', identifier): ← 等待从服务器接收数据更新。
        time.sleep(.001)

    deadline = time.time() + 1.01                         ← 最多只等待 1 秒。
    while time.time() < deadline:
        if sconn.info()['aof_pending_bio_fsync'] == 0: ← 检查数据更新是否已经被
            break                                         同步到了硬盘。
        time.sleep(.001)

    mconn.zrem('sync:wait', identifier)
    mconn.zremrangebyscore('sync:wait', 0, time.time()-900) ← 清理刚刚创建的新令牌以及之前可能
                                                               留下的旧令牌。

```

**INFO 命令中的其他信息** INFO 命令提供了大量的与 Redis 服务器当前状态有关的信息，比如内存占用量、客户端连接数、每个数据库包含的键的数量、上一次创建快照文件之后执行的命令数量，等等。总的来说，INFO 命令对于了解 Redis 服务器的综合状态非常有帮助，网上有很多资源都对 INFO 命令进行了详细的介绍。

为了确保操作可以正确执行，`wait_for_sync()` 函数会首先确认从服务器已经连接上主服务器，然后检查自己添加到等待同步有序集合（`sync wait ZSET`）里面的值是否已经存在于从服务器。在发现值已经存在于从服务器之后，函数会检查从服务器写入缓冲区的状态，并在 1 秒之内，等待从服务器将缓冲区中的所有数据写入硬盘里面。虽然函数最多会花费 1 秒来等待同步完成，但实际上大部分同步都会在很短的时间完成。最后，在确认数据已经被保存到硬盘之后，函数会执行一些清理操作。

通过同时使用复制和 AOF 持久化，用户可以增强 Redis 对于系统崩溃的抵抗能力。

## 4.3 处理系统故障

用户必须做好相应的准备来应对 Redis 的系统故障。本章在系统故障这个主题上花费了大量的篇幅，这是因为如果我们决定要将 Redis 用作应用程序唯一的数据存储手段，那么就必须确保 Redis 不会丢失任何数据。跟提供了 ACID<sup>①</sup> 保证的传统关系数据库不同，在使用 Redis 为后端构建应用程序

① ACID 是指原子性（atomicity）、一致性（consistency）、隔离性（isolation）和耐久性（durability），如果一个数据库想要实现可靠的数据事务，那么它就必须保证 ACID 性质。

的时候，用户需要多做一些工作才能保证数据的一致性。Redis 是一个软件，它运行在硬件之上，即使软件和硬件都设计得完美无瑕，也有可能会出现停电、发电机因为燃料耗尽而无法发电或者备用电池电量消尽等情况。这一节接下来将对 Redis 提供的一些工具进行介绍，说明如何使用这些工具来应对潜在的系统故障。下面先来看看在出现系统故障时，用户应该采取什么措施。

### 4.3.1 验证快照文件和 AOF 文件

无论是快照持久化还是 AOF 持久化，都提供了在遇到系统故障时进行数据恢复的工具。Redis 提供了两个命令行程序 `redis-check-aof` 和 `redis-check-dump`，它们可以在系统故障发生之后，检查 AOF 文件和快照文件的状态，并在有需要的情况下对文件进行修复。在不给定任何参数的情况下运行这两个程序，就可以看见它们的基本使用方法：

```
$ redis-check-aof
Usage: redis-check-aof [--fix] <file.aof>
$ redis-check-dump
Usage: redis-check-dump <dump.rdb>
$
```

如果用户在运行 `redis-check-aof` 程序时给定了`--fix` 参数，那么程序将对 AOF 文件进行修复。程序修复 AOF 文件的方法非常简单：它会扫描给定的 AOF 文件，寻找不正确或者不完整的命令，当发现第一个出错命令的时候，程序会删除出错的命令以及位于出错命令之后的所有命令，只保留那些位于出错命令之前的正确命令。在大多数情况下，被删除的都是 AOF 文件末尾的不完整的写命令。

遗憾的是，目前并没有办法可以修复出错的快照文件。尽管发现快照文件首个出现错误的地方是有可能的，但因为快照文件本身经过了压缩，而出现在快照文件中间的错误有可能会导致快照文件的剩余部分无法被读取。因此，用户最好为重要的快照文件保留多个备份，并在进行数据恢复时，通过计算快照文件的 SHA1 散列值和 SHA256 散列值来对内容进行验证。（当今的 Linux 平台和 Unix 平台都包含类似 `shasum` 和 `sha256sum` 这样的用于生成和验证散列值的命令行程序。）

**校验和（checksum）与散列值（hash）** 从 2.6 版本开始，Redis 会在快照文件中包含快照文件自身的 CRC64 校验和。CRC 校验和对于发现典型的网络传输错误和硬盘损坏非常有帮助，而 SHA 加密散列值则更擅长于发现文件中的任意错误（arbitrary error）。简单来说，用户可以翻转文件中任意数量的二进制位，然后通过翻转文件最后 64 个二进制位的一个子集（subset）来产生与原文件相同的 CRC64 校验和。而对于 SHA1 和 SHA256，目前还没有任何已知的方法可以做到这一点。

在了解了如何验证持久化文件是否完好无损，并且在有需要时对其进行修复之后，我们接下来要考虑的就是如何更换出现故障的 Redis 服务器。

### 4.3.2 更换故障主服务器

在运行一组同时使用复制和持久化的 Redis 服务器时，用户迟早都会遇上某个或某些 Redis 服务器停止运行的情况。造成故障的原因可能是硬盘驱动器出错、内存出错或者电量耗尽，但无论服务器因为何种原因出现故障，用户最终都要对发生故障的服务器进行更换。现在让我们来看看，在拥有一个主服务器和一个从服务器的情况下，更换主服务器的具体步骤。

假设 A、B 两台机器都运行着 Redis，其中机器 A 的 Redis 为主服务器，而机器 B 的 Redis 为从服务器。不巧的是，机器 A 刚刚因为某个暂时无法修复的故障而断开了网络连接，因此用户决定将同样安装了 Redis 的机器 C 用作新的主服务器。

更换服务器的计划非常简单：首先向机器 B 发送一个 SAVE 命令，让它创建一个新的快照文件，接着将这个快照文件发送给机器 C，并在机器 C 上面启动 Redis。最后，让机器 B 成为机器 C 的从服务器<sup>①</sup>。代码清单 4-4 展示了更换服务器时用到的各个命令。

代码清单 4-4 用于替换故障主节点的一连串命令

```
user@vpn-master ~:$ ssh root@machine-b.vpn
Last login: Wed Mar 28 15:21:06 2012 from ...
root@machine-b ~:$ redis-cli
redis 127.0.0.1:6379> SAVE
OK
redis 127.0.0.1:6379> QUIT
root@machine-b ~:$ scp \
> /var/local/redis/dump.rdb machine-c.vpn:/var/local/redis/
dump.rdb          100%   525MB   8.1MB/s   01:05
root@machine-b ~:$ ssh machine-c.vpn
Last login: Tue Mar 27 12:42:31 2012 from ...
root@machine-c ~:$ sudo /etc/init.d/redis-server start
Starting Redis server...
root@machine-c ~:$ exit
root@machine-b ~:$ redis-cli
redis 127.0.0.1:6379> SLAVEOF machine-c.vpn 6379
OK
redis 127.0.0.1:6379> QUIT
root@machine-b ~:$ exit
user@vpn-master ~:$
```

通过 VPN 网络连接机器 B。

启动命令行 Redis 客户端来执行几个简单的操作。

将快照文件发送至新的主服务器——机器 C。

连接新的主服务器并启动 Redis。

告知机器 B 的 Redis，让它将机器 C 用作新的主服务器。

代码清单 4-4 中列出的大部分命令，对于使用和维护 Unix 系统或者 Linux 系统的人来说应该都不会陌生。在这些命令当中，比较有趣的要数在机器 B 上运行的 SAVE 命令，以及将机器 B 设置为机器 C 的从服务器的 SLAVEOF 命令。

另一种创建新的主服务器的方法，就是将从服务器升级（turn）为主服务器，并为升级后的主服务器创建从服务器。以上列举的两种方法都可以让 Redis 回到之前的一个主服务

<sup>①</sup> 因为机器 B 原本就是一个从服务器，所以我们的客户端不能对它进行写入，并且在机器 B 执行快照操作之后，我们的客户端也不会与其他试图对机器 B 进行写入的客户端产生竞争条件。

器和一个从服务器的状态，而用户接下来要做的就是更新客户端的配置，让它们去读写正确的服务器。除此之外，如果用户需要重启 Redis 的话，那么可能还需要对服务器的持久化配置进行更新。

**Redis Sentinel** Redis Sentinel 可以监视指定的 Redis 主服务器及其属下的从服务器，并在主服务器下线时自动进行故障转移（failover）。本书将在第 10 章介绍 Redis Sentinel。

在接下来的一节中，我们将介绍一种保证数据安全所必不可少的功能，该功能可以在多个客户端同时对相同的数据进行写入时，防止数据出错。

## 4.4 Redis 事务

为了保证数据的正确性，我们必须认识到这一点：在多个客户端同时处理相同的数据时，不谨慎的操作很容易会导致数据出错。本节将介绍使用 Redis 事务来防止数据出错的方法，以及在某些情况下，使用事务来提升性能的方法。

Redis 的事务和传统关系数据库的事务并不相同。在关系数据库中，用户首先向数据库服务器发送 BEGIN，然后执行各个相互一致（consistent）的写操作和读操作，最后，用户可以选择发送 COMMIT 来确认之前所做的修改，或者发送 ROLLBACK 来放弃那些修改。

在 Redis 里面也有简单的方法可以处理一连串相互一致的读操作和写操作。正如本书在 3.7.2 节中介绍的那样，Redis 的事务以特殊命令 MULTI 为开始，之后跟着用户传入的多个命令，最后以 EXEC 为结束。但是由于这种简单的事务在 EXEC 命令被调用之前不会执行任何实际操作，所以用户将没办法根据读取到的数据来做决定。这个问题看上去似乎无足轻重，但实际上无法以一致的形式读取数据将导致某一类型的问题变得难以解决，除此之外，因为在多个事务同时处理同一个对象时通常需要用到二阶提交（two-phase commit），所以如果事务不能以一致的形式读取数据，那么二阶提交将无法实现，从而导致一些原本可以成功执行的事务沦落至执行失败的地步。比如说：“在市场里面购买一件商品”就是其中一个会因为无法以一致的形式读取数据而变得难以解决的问题，本节接下来将在实际环境中对这个问题进行介绍。

**延迟执行事务有助于提升性能** 因为 Redis 在执行事务的过程中，会延迟执行已入队的命令直到客户端发送 EXEC 命令为止。因此，包括本书使用的 Python 客户端在内的很多 Redis 客户端都会等到事务包含的所有命令都出现了之后，才一次性地将 MULTI 命令、要在事务中执行的一系列命令，以及 EXEC 命令全部发送给 Redis，然后等待直到接收到所有命令的回复为止。这种“一次性发送多个命令，然后等待所有回复出现”的做法通常被称为流水线（pipelining），它可以通过减少客户端与 Redis 服务器之间的网络通信次数来提升 Redis 在执行多个命令时的性能。

最近几个月，Fake Game 公司发现他们在 YouTwitFace（一个虚构的社交网站）上面推出的角色扮演游戏正在变得越来越受欢迎。因此，关心玩家需求的 Fake Game 公司决定在游戏里面增加一个商品买卖市场，让玩家们可以在市场里面销售和购买商品。本节接下来的内容将介绍设计和实现这个商品买卖市场的方法，并说明如何按需对这个商品买卖市场进行扩展。

#### 4.4.1 定义用户信息和用户包裹

图 4-2 展示了游戏中用于表示用户信息和用户包裹（inventory）的结构：用户信息存储在一个散列里面，散列的各个键值对分别记录了用户的姓名、用户拥有的钱数等属性。用户包裹使用一个集合来表示，它记录了包裹里面每件商品的唯一编号。

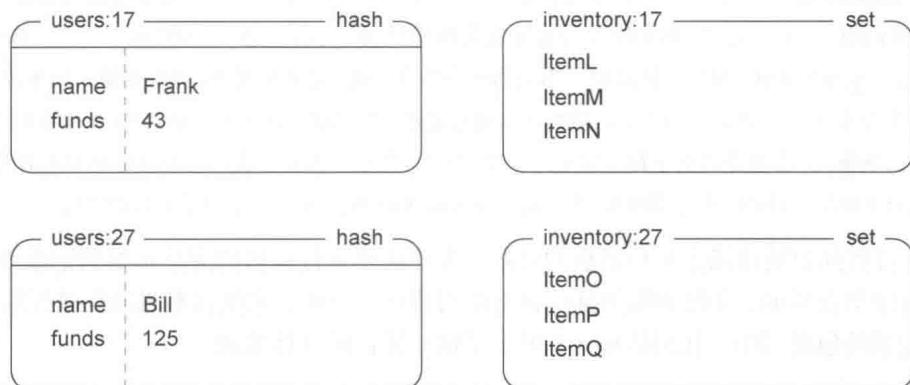


图 4-2 用户信息示例和用户包裹示例。Frank 有 43 块钱，并且他打算卖掉自己包裹里面的其中一件商品

商品买卖市场的需求非常简单：一个用户（卖家）可以将自己的商品按照给定的价格放到市场上进行销售，当另一个用户（买家）购买这个商品时，卖家就会收到钱。另外，本节实现的市场只能根据商品的价格来进行排序，稍后的第 7 章将介绍如何在市场里面实现其他排序方式。

为了将被销售商品的全部信息都存储到市场里面，我们会将商品的 ID 和卖家的 ID 拼接起来，并将拼接的结果用作成员存储到市场有序集合（market ZSET）里面，而商品的售价则用作成员的分值。通过将所有数据都包含在一起，我们极大地简化了实现商品买卖市场所需的数据结构，并且因为市场里面的所有商品都按照价格排序，所以针对商品的分页功能和查找功能都可以很容易地实现。图 4-3 展示了一个只包含数个商品的市场例子。

既然我们已经知道了实现商品买卖市场所需的数据结构，那么接下来该考虑如何实现市场的商品上架功能了。

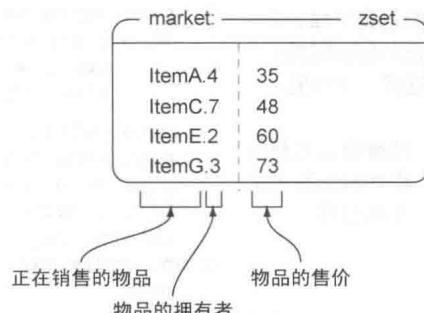


图 4-3 一个基本的商品买卖市场，其中用户 4 正在销售商品 ItemA，售价为 35 块钱

#### 4.4.2 将商品放到市场上销售

为了将商品放到市场上进行销售，程序除了要使用 MULTI 命令和 EXEC 命令之外，还需要配合使用 WATCH 命令，有时候甚至还会用到 UNWATCH 或 DISCARD 命令。在用户使用 WATCH 命令对键进行监视之后，直到用户执行 EXEC 命令的这段时间里面，如果有其他客户端抢先对任何被监视的键进行了替换、更新或删除等操作，那么当用户尝试执行 EXEC 命令的时候，事务将失败并返回一个错误（之后用户可以选择重试事务或者放弃事务）。通过使用 WATCH、MULTI/EXEC、UNWATCH/DISCARD 等命令，程序可以在执行某些重要操作的时候，通过确保自己正在使用的数据没有发生变化来避免数据出错。

**什么是 DISCARD？** UNWATCH 命令可以在 WATCH 命令执行之后、MULTI 命令执行之前对连接进行重置（reset）；同样地，DISCARD 命令也可以在 MULTI 命令执行之后、EXEC 命令执行之前对连接进行重置。这也就是说，用户在使用 WATCH 监视一个或多个键，接着使用 MULTI 开始一个新的事务，并将多个命令入队到事务队列之后，仍然可以通过发送 DISCARD 命令来取消 WATCH 命令并清空所有已入队命令。本章展示的例子都没有用到 DISCARD，主要原因在于我们已经清楚地知道自己是否想要执行 MULTI/EXEC 或者 UNWATCH，所以没有必要在这些例子里面使用 DISCARD。

在将一件商品放到市场上进行销售的时候，程序需要将被销售的商品添加到记录市场正在销售商品的有序集合里面，并且在添加操作执行的过程中，监视卖家的包裹以确保被销售的商品的确存在于卖家的包裹当中，代码清单 4-5 展示了这一操作的具体实现。

代码清单 4-5 list\_item() 函数

```
def list_item(conn, itemid, sellerid, price):
    inventory = "inventory:%s"%sellerid
    item = "%s.%s"%(itemid, sellerid)
    end = time.time() + 5
    pipe = conn.pipeline()
    if specified.item not in user.package, then stop watching package key and return None.
    if item not in user.package, then unwatch package key and return None.
    if execute successfully, then return True.
    if WatchError occurs, then pass.
    return False.

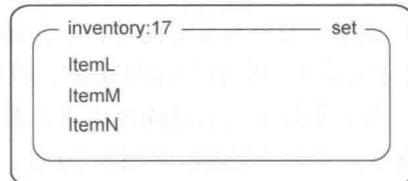
    while time.time() < end:
        try:
            pipe.watch(inventory)
            if not pipe.sismember(inventory, itemid):
                pipe.unwatch()
                return None
            pipe.multi()
            pipe.zadd("market:", item, price)
            pipe.srem(inventory, itemid)
            pipe.execute()
            return True
        except redis.exceptions.WatchError:
            pass
    return False
```

list\_item() 函数的行为就和我们之前描述的一样：它首先执行一些初始化步骤，然后对卖家的包裹进行监视，验证卖家想要销售的商品是否仍然存在于卖家的包裹当中，如果是的话，函数就

会将被销售的商品添加到买卖市场里面，并从卖家的包裹中移除该商品。正如函数中的 while 循环所示，在使用 WATCH 命令对包裹进行监视的过程中，如果包裹被更新或者修改，那么程序将接收到错误并进行重试。

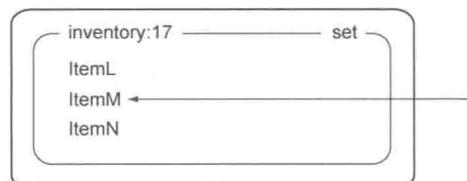
图 4-4 展示了当 Frank（用户 ID 为 17）尝试以 97 块钱的价格销售 ItemM 时，list\_item() 函数的执行过程。

```
watch('inventory:17')
```



监视包裹发生的任何变化。

```
sismember('inventory:17', 'ItemM')
```



确保被销售的物品仍然存在于  
Frank的包裹里面。

```
market: zset
```

ItemA.4	35
ItemC.7	48
ItemE.2	60
ItemG.3	73
ItemM.17	97

```
zadd('market:', 'ItemM.17', 97)
```

因为没有一个Redis命令  
可以在移除集合元素的同  
时，将被移除的元素改名  
并添加到有序集合里面，  
所以这里使用了ZADD和  
SREM两个命令来实现这  
一操作。

```
srem('inventory:17', 'ItemM')
```

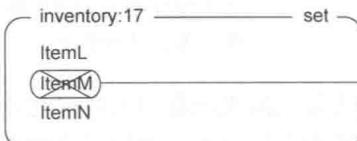


图 4-4 list\_item(conn, "ItemM", 17, 97) 的执行过程

因为程序会确保用户只能销售他们自己所拥有的商品，所以在一般情况下，用户都可以顺利地将自己想要销售的商品添加到商品买卖市场上面，但是正如之前所说，如果用户的包裹在

WATCH 执行之后直到 EXEC 执行之前的这段时间内发生了变化，那么添加操作将执行失败并重试。

在弄懂了怎样将商品放到市场上销售之后，接下来让我们来了解一下怎样从市场上购买商品。

### 4.4.3 购买商品

代码清单 4-6 中的 `purchase_item()` 函数展示了从市场里面购买一件商品的具体方法：程序首先使用 WATCH 对市场以及买家的个人信息进行监视，然后获取买家拥有的钱数以及商品的售价，并检查买家是否有足够的钱来购买该商品。如果买家没有足够的钱，那么程序会取消事务；相反地，如果买家的钱足够，那么程序首先会将买家支付的钱转移给卖家，然后将售出的商品移动至买家的包裹，并将该商品从市场中移除。当买家的个人信息或者商品买卖市场出现变化而导致 WatchError 异常出现时，程序将进行重试，其中最大重试时间为 10 秒。

代码清单 4-6 `purchase_item()` 函数

```
def purchase_item(conn, buyerid, itemid, sellerid, lprice):
    buyer = "users:%s"%buyerid
    seller = "users:%s"%sellerid
    item = "%s.%s"%(itemid, sellerid)
    inventory = "inventory:%s"%buyerid
    end = time.time() + 10
    pipe = conn.pipeline()
    while time.time() < end:
        try:
            pipe.watch("market:", buyer)           ← 对商品买卖市场以及买家的个人信息进行监视。
            price = pipe.zscore("market:", item)
            funds = int(pipe.hget(buyer, "funds"))
            if price != lprice or price > funds:
                pipe.unwatch()
                return None
            pipe.multi()
            pipe.hincrby(seller, "funds", int(price))
            pipe.hincrby(buyer, "funds", int(-price))
            pipe.sadd(inventory, itemid)
            pipe.zrem("market:", item)
            pipe.execute()
            return True
        except redis.exceptions.WatchError:
            pass
    return False                                ← 如果买家的个人信息或者商品买卖市场在交易的过程中出现了变化，那么进行重试。
```

在执行商品购买操作的时候，程序除了需要花费大量时间来准备相关数据之外，还需要对商品买卖市场以及买家的个人信息进行监视：监视商品买卖市场是为了确保买家想要购买的商品仍然有售（或者在商品已经被其他人买走时进行提示），而监视买家的个人信息则是为了验证买家是否有足够的钱来购买自己想要的商品。

当程序确认商品仍然存在并且买家有足够的钱的时候，程序会将被购买的商品移动到买家的包