关于此书

最新版本

此书的最新有效资源在：
http://github.com/karlseguin/the-little-redis-book

中文版是英文版的一个分支，最新的中文版本在：
https://github.com/JasonLai256/the-little-redis-book

简介

最近几年来，关于持久化和数据查询的相关技术，其需求已经增长到了让人惊讶的程度。可以断言，关系型数据库再也不是放之四海皆准。换一句话说，围绕数据的解决方案不可能再只有唯一一种。

对于我来说，在众多新出现的解决方案和工具里，最让人兴奋的，无疑是Redis。为什么？首先是因为其让人不可思议的容易学习，只需要简短的几个小时学习时间，就能对Redis有个大概的认识。还有，Redis在处理一组特定的问题集的同时能保持相当的通用性。更准确地说就是，Redis不会尝试去解决关于数据的所有事情。在你足够了解Redis后，事情就会变得越来越清晰，什么是可行的，什么是不应该由Redis来处理的。作为一名开发人员，如此的经验当是相当的美妙。

当你能仅使用Redis去构建一个完整系统时，我想大多数人将会发现，Redis能使得他们的许多数据方案变得更为通用，不论是一个传统的关系型数据库，一个面向文档的系统，或是其它更多的东西。这是一种用来实现某些特定特性的解决方法。就类似于一个索引引擎，你不会在 Lucene 上构建整个程序，但当你需要足够好的搜索，为什么不使用它呢？这对你和你的用户都有好处。当然，关于Redis和索引引擎之间相似性的讨论到此为止。

本书的目的是向读者传授掌握Redis所需要的基本知识。我们将会注重于学习Redis的5种数据结构，并研究各种数据建模方法。我们还会接触到一些主要的管理细节和调试技巧。

入门

每个人的学习方式都不一样，有的人喜欢亲自实践学习，有的喜欢观看教学视频，还有的喜欢通过阅读来学习。对于Redis，没有什么比亲自实践学习来得效果更好的了。Redis的安装非常简单。而且通过随之安装的一个简单的命令解析程序，就能处理我们想做的一切事情。让我们先花几分钟的时间把Redis安装到我们的机器上。

Windows平台

Redis并没有官方支持Windows平台，但还是可供选择。你不会想在这里配置实际的生产环境，不过在我过往的开发经历里并没有感到有什么限制。

首先进入https://github.com/dmajkic/redis/downloads，然后下载最新的版本（应该会在列表的最上方）。

获取zip文件，然后根据你的系统架构，打开`64bit`或`32bit`文件夹。

*nix和MacOSX平台

对于*nix和MacOSX平台的用户，从源文件来安装是你的最佳选择。通过最新的版本号来选择，有效地址于http://redis.io/download。在编写此书的时候，最新的版本是2.4.6，我们可以运行下面的命令来安装该版本：

```
wget http://redis.googlecode.com/files/redis-2.4.6.tar.gz
tar xzf redis-2.4.6.tar.gz
cd redis-2.4.6
make
```

（当然，Redis同样可以通过套件管理程序来安装。例如，使用Homebrew的MaxOSX用户可以只键入`brew install redis`即可。）

如果你是通过源文件来安装，二进制可执行文件会被放置在`src`目录里。通过运行`cd src`可跳转到`src`目录。

运行和连接**Redis**

如果一切都工作正常，那Redis的二进制文件应该已经可以曼妙地跳跃于你的指尖之下。Redis只有少量的可执行文件，我们将着重于Redis的服务器和命令行界面（一个类DOS的客户端）。首先，让我们来运行服务器。在Windows平台，双击`redis-server`，在*nix/MacOSX平台则运行`./redis-server`.

如果你仔细看了启动信息，你会看到一个警告，指没能找到`redis.conf`文件。Redis将会采用内置的默认设置，这对于我们将要做的已经足够了。

然后，通过双击`redis-cli`（Windows平台）或者运行`./redis-cli`（`*nix/MacOSX`平台），启动Redis的控制台。控制台将会通过默认的端口（6379）来连接本地运行的服务器。

可以在命令行界面键入`info`命令来查看一切是不是都运行正常。你会很乐意看到这么一大组关键字-值（key-value）对的显示，这为我们查看服务器的状态提供了大量有效信息。

如果在上面的启动步骤里遇到什么问题，我建议你到Redis的官方支持组里获取帮助。

驱动**Redis**

很快你就会发现，Redis的API就如一组定义明确的函数那般容易理解。Redis具有让人难以置信的简单性，其操作过程也同样如此。这意味着，无论你是使用命令行程序，或是使用你喜欢的语言来驱动，整体的感觉都不会相差多少。因此，相对于命令行程序，如果你更愿意通过一种编程语言去驱动Redis，你不会感觉到有任何适应的问题。如果真想如此，可以到Redis的客户端推荐页面下载适合的Redis载体。

第**1**章 **-** 基础知识

是什么使Redis显得这么特别？Redis具体能解决什么类型的问题？要实际应用Redis，开发者必须储备什么知识？在我们能回答这么一些问题之前，我们需要明白Redis到底是什么。

Redis通常被人们认为是一种持久化的存储器关键字-值型存储（in-memory persistent key-value store）。我认为这种对Redis的描述并不太准确。Redis的确是将所有的数据存放于存储器（更多是是按位存储），而且也确实通过将数据写入磁盘来实现持久化，但是**Redis**的实际意义比单纯的关键字**-**值型存储要来得深远。纠正脑海里的这种误解观点非常关键，否则你对于Redis之道以及其应用的洞察力就会变得越发狭义。

事实是，Redis引入了5种不同的数据结构，只有一个是典型的关键字-值型结构。理解Redis的关键就在于搞清楚这5种数据结构，其工作的原理都是如何，有什么关联方法以及你能怎样应用这些数据结构去构建模型。首先，让我们来弄明白这些数据结构的实际意义。

应用上面提及的数据结构概念到我们熟悉的关系型数据库里，我们可以认为其引入了一个单独的数据结构——表格。表格既复杂又灵活，基于表格的存储和管理，没有多少东西是你不能进行建模的。然而，这种通用性并不是没有缺点。具体来说就是，事情并不是总能达到假设中的简单或者快速。相对于这种普遍适用（one-size-fits-all）的结构体系，我们可以使用更为专门化的结构体系。当然，因此可能有些事情我们会完成不了(至少，达不到很好的程度）。但话说回来，这样做就能确定我们可以获得想象中的简单性和速度吗？

针对特定类型的问题使用特定的数据结构？我们不就是这样进行编程的吗？你不会使用一个散列表去存储每份数据，也不会使用一个标量变量去存储。对我来说，这正是Redis的做法。如果你需要处理标量、列表、散列或者集合，为什么不直接就用标量、列表、散列和集合去存储他们？为什么不是直接调用`exists(key)`去检测一个已存在的值，而是要调用其他比O(1)（常量时间查找，不会因为待处理元素的增长而变慢）慢的操作？

数据库（**Databases**）

与你熟悉的关系型数据库一致，Redis有着相同的数据库基本概念，即一个数据库包含一组数据。典型的数据库应用案例是，将一个程序的所有数据组织起来，使之与另一个程序的数据保持独立。

在Redis里，数据库简单的使用一个数字编号来进行辨认，默认数据库的数字编号是`0`。如果你想切换到一个不同的数据库，你可以使用`select`命令来实现。在命令行界面里键入`select 1`，Redis应该会回复一条`OK`的信息，然后命令行界面里的提示符会变成类似`redis 127.0.0.1:6379[1]>`这样。如果你想切换回默认数据库，只要在命令行界面键入`select 0`即可。

命令、关键字和值（**Commands, Keys and Values**）

Redis不仅仅是一种简单的关键字-值型存储，从其核心概念来看，Redis的5种数据结构中的每一个都至少有一个关键字和一个值。在转入其它关于Redis的有用信息之前，我们必须理解关键字和值的概念。

关键字（Keys）是用来标识数据块。我们将会经常跟关键字打交道，不过在现在，明白关键字就是类似于`users:leto`这样的表述就足够了。一般都能很好地理解到，这样关键字包含的信息是一个名为`leto`的用户。这个关键字里的冒号没有任何特殊含义，对于Redis而言，使用分隔符来组织关键字是很常见的方法。

值（Values）是关联于关键字的实际值，可以是任何东西。有时候你会存储字符串，有时候是整数，还有时候你会存储序列化对象（使用JSON、XML或其他格式）。在大多数情况下，Redis会把值看做是一个字节序列，而不会关注它们实质上是什么。要注意，不同的Redis载体处理序列化会有所不同（一些会让你自己决定）。因此，在这本书里，我们将仅讨论字符串、整数和JSON。

现在让我们活动一下手指吧。在命令行界面键入下面的命令：

```
set users:leto "{name: leto, planet: dune, likes: [spice]}"
```

这就是Redis命令的基本构成。首先我们要有一个确定的命令，在上面的语句里就是`set`。然后就是相应的参数，`set`命令接受两个参数，包括要设置的关键字，以及相应要设置的值。很多的情况是，命令接受一个关键字（当这种情况出现，其经常是第一个参数）。你能想到如何去获取这个值吗？我想你会说（当然一时拿不准也没什么）：

```
get users:leto
```

关键字和值的是Redis的基本概念，而`get`和`set`命令是对此最简单的使用。你可以创建更多的用户，去尝试不同类型的关键字以及不同的值，看看一些不同的组合。

查询（**Querying**）

随着学习的持续深入，两件事情将变得清晰起来。对于Redis而言，`key`就是一切，而值是没有任何意义。更通俗来看就是，Redis不允许你通过值来进行查询。回到上面的例子，我们就不能查询生活在`dune`行星上的用户。

对许多人来说，这会引起一些担忧。在我们生活的世界里，数据查询是如此的灵活和强大，而Redis的方式看起来是这么的原始和不高效。不要让这些扰乱你太久。要记住，Redis不是一种普遍使用（one-size-fits-all）的解决方案，确实存在这么一些事情是不应该由Redis来解决的（因为其查询的限制）。事实上，在考虑了这些情况后，你会找到新的方法去构建你的数据。

很快，我们就能看到更多实际的用例。很重要的一点是，我们要明白关于Redis的这些基本事实。这能帮助我们弄清楚: 为什么`value`可以是任何东西，因为Redis从来不需要去读取或理解它们。而且，这也可以帮助我们理清思路，然后去思考如何在这个新世界里建立模型。

存储器和持久化（**Memory and Persistence**）

我们之前提及过，Redis是一种持久化的存储器内存储（in-memory persistent store）。对于持久化，默认情况下，Redis会根据已变更的关键字数量来进行判断，然后在磁盘里创建数据库的快照（snapshot）。你可以对此进行设置，如果X个关键字已变更，那么每隔Y秒存储数据库一次。默认情况下，如果1000个或更多的关键字已变更，Redis会每隔60秒存储数据库；而如果9个或更少的关键字已变更，Redis会每隔15分钟存储数据库。

除了创建磁盘快照外，Redis可以在附加模式下运行。任何时候，如果有一个关键字变更，一个单一附加（append-only）的文件会在磁盘里进行更新。在一些情况里，虽然硬件或软件可能发生错误，但用那60秒有效数据存储去换取更好性能是可以接受的。而在另一些情况里，这种损失就难以让人接受，Redis为你提供了选择。在第5章里，我们将会看到第三种选择，其将持久化任务减荷到一个从属数据库里。

至于存储器，Redis会将所有数据都保留在存储器中。显而易见，运行Redis具有不低的成本：因为RAM仍然是最昂贵的服务器硬件部件。

我很清楚有一些开发者对即使是一点点的数据空间都是那么的敏感。一本《威廉·莎士比亚全集》需要近5.5MB的存储空间。对于缩放的需求，其它的解决方案趋向于IO-bound或者CPU-bound。这些限制（RAM或者IO）将会需要你去理解更多机器实际依赖的数据类型，以及应该如何去进行存储和查询。除非你是存储大容量的多媒体文件到Redis中，否则存储器内存储应该不会是一个问题。如果这对于一个程序是个问题，你就很可能不会用IO-bound的解决方案。

Redis有虚拟存储器的支持。然而，这个功能已经被认为是失败的了（通过Redis的开发者），而且它的使用已经被废弃了。

（从另一个角度来看，一本5.5MB的《威廉·莎士比亚全集》可以通过压缩减小到近2MB。当然，Redis不会自动对值进行压缩，但是因为其将所有值都看作是字节，没有什么限制让你不能对数据进行压缩/解压，通过牺牲处理时间来换取存储空间。）

整体来看（**Putting It Together**）

我们已经接触了好几个高层次的主题。在继续深入Redis之前，我想做的最后一件事情是将这些主题整合起来。这些主题包括，查询的限制，数据结构以及Redis在存储器内存储数据的方法。

当你将这3个主题整合起来，你最终会得出一个绝妙的结论：速度。一些人可能会想，当然Redis会很快速，要知道所有的东西都在存储器里。但这仅仅是其中的一部分，让Redis闪耀的真正原因是其不同于其它解决方案的特殊数据结构。

能有多快速？这依赖于很多东西，包括你正在使用着哪个命令，数据的类型等等。但Redis的性能测试是趋向于数万或数十万次操作每秒。你可以通过运行`redis-benchmark`（就在`redis-server`和`redis-cli`的同一个文件夹里）来进行测试。

我曾经试过将一组使用传统模型的代码转向使用Redis。在传统模型里，运行一个我写的载入测试，需要超过5分钟的时间来完成。而在Redis里，只需要150毫秒就完成了。你不会总能得到这么好的收获，但希望这能让你对我们所谈的东西有更清晰的理解。

理解Redis的这个特性很重要，因为这将影响到你如何去与Redis进行交互。拥有SQL背景的程序员通常会致力于让数据库的数据往返次数减至最小。这对于任何系统都是个好建议，包括Redis。然而，考虑到我们是在处理比较简单的数据结构，有时候我们还是需要与Redis服务器频繁交互，以达到我们的目的。刚开始的时候，可能会对这种数据访问模式感到不太自然。实际上，相对于我们通过Redis获得的高性能而言，这仅仅是微不足道的损失。

小结

虽然我们只接触和摆弄了Redis的冰山一角，但我们讨论的主题已然覆盖了很大范围内的东西。如果觉得有些事情还是不太清楚（例如查询），不用为此而担心，在下一章我们将会继续深入探讨，希望你的问题都能得到解答。

这一章的要点包括：

- 关键字（Keys）是用于标识一段数据的一个字符串
- 值（Values）是一段任意的字节序列，Redis不会关注它们实质上是什么
- Redis展示了（也实现了）5种专门的数据结构
- 上面的几点使得Redis快速而且容易使用，但要知道Redis并不适用于所有的应用场景

第2章 - 数据结构

现在开始将探究Redis的5种数据结构，我们会解释每种数据结构都是什么，包含了什么有效的方法（Method），以及你能用这些数据结构处理哪些类型的特性和数据。

目前为止，我们所知道的Redis构成仅包括命令、关键字和值，还没有接触到关于数据结构的具体概念。当我们使用`set`命令时，Redis是怎么知道我们是在使用哪个数据结构？其解决方法是，每个命令都相对应于一种特定的数据结构。例如，当你使用`set`命令，你就是将值存储到一个字符串数据结构里。而当你使用`hset`命令，你就是将值存储到一个散列数据结构里。考虑到Redis的关键字集很小，这样的机制具有相当的可管理性。

Redis的网站里有着非常优秀的参考文档，没有任何理由去重造轮子。但为了搞清楚这些数据结构的作用，我们将会覆盖那些必须知道的重要命令。

没有什么事情比高兴的玩和试验有趣的东西来得更重要的了。在任何时候，你都能通过键入`flushdb`命令将你数据库里的所有值清除掉，因此，不要再那么害羞了，去尝试做些疯狂的事情吧！

字符串（Strings)

在Redis里，字符串是最基本的数据结构。当你在思索着关键字-值对时，你就是在思索着字符串数据结构。不要被名字给搞混了，如之前说过的，你的值可以是任何东西。我更喜欢将他们称作“标量”（Scalars），但也许只有我才这样想。

我们已经看到了一个常见的字符串使用案例，即通过关键字存储对象的实例。有时候，你会频繁地用到这类操作：

```
set users:leto "{name: leto, planet: dune, likes: [spice]}"
```

除了这些外，Redis还有一些常用的操作。例如，`strlen <key>`能用来获取一个关键字对应值的长度；`getrange <key> <start> <end>`将返回指定范围内的关键字对应值；`append <key> <value>`会将value附加到已存在的关键字对应值中（如果该关键字并不存在，则会创建一个新的关键字-值对）。不要犹豫，去试试看这些命令吧。下面是我得到的：

```
> strlen users:leto
(integer) 42

> getrange users:leto 27 40
"likes: [spice]"

> append users:leto " OVER 9000!!"
(integer) 54
```

现在你可能会想，这很好，但似乎没有什么意义。你不能有效地提取出一段范围内的JSON文件，或者为其附加一些值。你是对的，这里的经验是，一些命令，尤其是关于字符串数据结构的，只有在给定了明确的数据类型后，才会有实际意义。

之前我们知道了，Redis不会去关注你的值是什么东西。通常情况下，这没有错。然而，一些字符串命令是专门为一些类型或值的结构而设计的。作为一个有些含糊的用例，我们可以看到，对于一些自定义的空间效率很高的（space-efficient）串行化对象，`append`和`getrange`命令将会很有用。对于一个更为具体的用例，我们可以再看一下`incr`、`incrby`、`decr`和`decrby`命令。这些命令会增长或者缩减一个字符串数据结构的值：

```
> incr stats:page:about
(integer) 1
> incr stats:page:about
(integer) 2

> incrby ratings:video:12333 5
(integer) 5
> incrby ratings:video:12333 3
(integer) 8
```

由此你可以想象到，Redis的字符串数据结构能很好地用于分析用途。你还可以去尝试增长`users:leto`（一个不是整数的值），然后看看会发生什么（应该会得到一个错误）。

更为进阶的用例是`setbit`和`getbit`命令。“今天我们有多少个独立用户访问”是个在Web应用里常见的问题，有一篇精彩的博文，在里面可以看到Spool是如何使用这两个命令有效地解决此问题。对于1.28亿个用户，一部笔记本电脑在不到50毫秒的时间里就给出了答复，而且只用了16MB的存储空间。

最重要的事情不是在于你是否明白位图（Bitmaps)的工作原理，或者Spool是如何去使用这些命令，而是应该要清楚Redis的字符串数据结构比你当初所想的要有用许多。然而，最常见的应用案例还是上面我们给出的：存储对象（简单或复杂）和计数。同时，由于通过关键字来获取一个值是如此之快，字符串数据结构很常被用来缓存数据。

散列（Hashes）

我们已经知道把Redis称为一种关键字-值型存储是不太准确的，散列数据结构是一个很好的例证。你会看到，在很多方面里，散列数据结构很像字符串数据结构。两者显著的区别在于，散列数据结构提供了一个额外的间接层：一个域（Field）。因此，散列数据结构中的`set`和`get`是：

```
hset users:goku powerlevel 9000
hget users:goku powerlevel
```

相关的操作还包括在同一时间设置多个域、同一时间获取多个域、获取所有的域和值、列出所有的域或者删除指定的一个域：

```
hmset users:goku race saiyan age 737
hmget users:goku race powerlevel
hgetall users:goku
hkeys users:goku
hdel users:goku age
```

如你所见，散列数据结构比普通的字符串数据结构具有更多的可操作性。我们可以使用一个散列数据结构去获得更精确的描述，是存储一个用户，而不是一个序列化对象。从而得到的好处是能够提取、更新和删除具体的数据片段，而不必去获取或写入整个值。

对于散列数据结构，可以从一个经过明确定义的对象的角度来考虑，例如一个用户，关键之处在于要理解他们是如何工作的。从性能上的原因来看，这是正确的，更具粒度化的控制可能会相当有用。在下一章我们将会看到，如何用散列数据结构去组织你的数据，使查询变得更为实效。在我看来，这是散列真正耀眼的地方。

列表（**Lists**）

对于一个给定的关键字，列表数据结构让你可以存储和处理一组值。你可以添加一个值到列表里、获取列表的第一个值或最后一个值以及用给定的索引来处理值。列表数据结构维护了值的顺序，提供了基于索引的高效操作。为了跟踪在网站里注册的最新用户，我们可以维护一个`newusers`的列表：

```
lpush newusers goku
ltrim newusers 0 50
```

（译注：`ltrim`命令的具体构成是`LTRIM Key start stop`。要理解`ltrim`命令，首先要明白**Key**所存储的值是一个列表，理论上列表可以存放任意个值。对于指定的列表，根据所提供的两个范围参数**start**和**stop**，`ltrim`命令会将指定范围外的值都删除掉，只留下范围内的值。）

首先，我们将一个新用户推入到列表的前端，然后对列表进行调整，使得该列表只包含50个最近被推入的用户。这是一种常见的模式。`ltrim`是一个具有O(N)时间复杂度的操作，N是被删除的值的数量。从上面的例子来看，我们总是在插入了一个用户后再进行列表调整，实际上，其将具有O(1)的时间复杂度（因为N将永远等于1）的常数性能。

这是我们第一次看到一个关键字的对应值索引另一个值。如果我们想要获取最近的10个用户的详细资料，我们可以运行下面的组合操作：

```
keys = redis.lrange('newusers', 0, 10)
redis.mget(*keys.map {|u| "users:#{u}"})
```

我们之前谈论过关于多次往返数据的模式，上面的两行Ruby代码为我们进行了很好的演示。

当然，对于存储和索引关键字的功能，并不是只有列表数据结构这种方式。值可以是任意的东西，你可以使用列表数据结构去存储日志，也可以用来跟踪用户浏览网站时的路径。如果你过往曾构建过游戏，你可能会使用列表数据结构去跟踪用户的排队活动。

集合（**Sets**）

集合数据结构常常被用来存储只能唯一存在的值，并提供了许多的基于集合的操作，例如并集。集合数据结构没有对值进行排序，但是其提供了高效的基于值的操作。使用集合数据结构的典型用例是朋友名单的实现：

```
sadd friends:leto ghanima paul chani jessica
sadd friends:duncan paul jessica alia
```

不管一个用户有多少个朋友，我们都能高效地（O(1)时间复杂度）识别出用户X是不是用户Y的朋友：

```
sismember friends:leto jessica
sismember friends:leto vladimir
```

而且，我们可以查看两个或更多的人是不是有共同的朋友：

```
sinter friends:leto friends:duncan
```

甚至可以在一个新的关键字里存储结果：

```
sinterstore friends:leto_duncan friends:leto friends:duncan
```

有时候需要对值的属性进行标记和跟踪处理，但不能通过简单的复制操作完成，集合数据结构是解决此类问题的最好方法之一。当然，对于那些需要运用集合操作的地方（例如交集和并集），集合数据结构就是最好的选择。

分类集合（**Sorted Sets**）

最后也是最强大的数据结构是分类集合数据结构。如果说散列数据结构类似于字符串数据结构，主要区分是域（field）的概念；那么分类集合数据结构就类似于集合数据结构，主要区分是标记（score）的概念。标记提供了排序（sorting）和秩划分（ranking）的功能。如果我们想要一个秩分类的朋友名单，可以这样做：

```
zadd friends:duncan 70 ghanima 95 paul 95 chani 75 jessica 1 vladimir
```

对于`duncan`的朋友，要怎样计算出标记（score）为90或更高的人数？

```
zcount friends:duncan 90 100
```

如何获取`chani`在名单里的秩（rank）？

```
zrevrank friends:duncan chani
```

（译注：`zrank`命令的具体构成是`ZRANK Key menber`，要知道**Key**存储的**Sorted Set**默认是根据**Score**对各个**menber**进行升序的排列，该命令就是用来获取**menber**在该排列里的次序，这就是所谓的秩。）

我们使用了`zrevrank`命令而不是`zrank`命令，这是因为Redis的默认排序是从低到高，但是在这个例子里我们的秩划分是从高到低。对于分类集合数据结构，最常见的应用案例是用来实现排行榜系统。事实上，对于一些基于整数排序，且能以标记（score）来进行有效操作的东西，使用分类集合数据结构来处理应该都是不错的选择。

小结

对于Redis的5种数据结构，我们进行了高层次的概述。一件有趣的事情是，相对于最初构建时的想法，你经常能用Redis创造出一些更具实效的事情。对于字符串数据结构和分类集合数据结构的使用，很有可能存在一些构建方法是还没有人想到的。当你理解了那些常用的应用案例后，你将发现Redis对于许多类型的问题，都是很理想的选择。还有，不要因为Redis展示了5种数据结构和相应的各种方法，就认为你必须要把所有的东西都用上。只使用一些命令

去构建一个特性是很常见的。

第3章 - 使用数据结构

在上一章里，我们谈论了Redis的5种数据结构，对于一些可能的用途也给出了用例。现在是时候来看看一些更高级，但依然很常见的主题和设计模式。

大O表示法（Big O Notation）

在本书中，我们之前就已经看到过大O表示法，包括O(1)和O(N)的表示。大O表示法的惯常用途是，描述一些用于处理一定数量元素的行为的综合表现。在Redis里，对于一个要处理一定数量元素的命令，大O表示法让我们能了解该命令的大概运行速度。

在Redis的文档里，每一个命令的时间复杂度都用大O表示法进行了描述，还能知道各命令的具体性能会受什么因素影响。让我们来看看一些用例。

常数时间复杂度O(1)被认为是最快速的，无论我们是在处理5个元素还是5百万个元素，最终都能得到相同的性能。对于`sismember`命令，其作用是告诉我们一个值是否属于一个集合，时间复杂度为O(1)。`sismember`命令很强大，很大部分的原因是其高效的性能特征。许多Redis命令都具有O(1)的时间复杂度。

对数时间复杂度O(log(N))被认为是第二快速的，其通过使需扫描的区间不断皱缩来快速完成处理。使用这种“分而治之”的方式，大量的元素能在几个迭代过程里被快速分解完整。`zadd`命令的时间复杂度就是O(log(N))，其中N是在分类集合中的元素数量。

再下来就是线性时间复杂度O(N)，在一个表格的非索引列里进行查找就需要O(N)次操作。`ltrim`命令具有O(N)的时间复杂度，但是，在`ltrim`命令里，N不是列表所拥有的元素数量，而是被删除的元素数量。从一个具有百万元素的列表里用`ltrim`命令删除1个元素，要比从一个具有一千个元素的列表里用`ltrim`命令删除10个元素来的快速（实际上，两者很可能会是一样快，因为两个时间都非常的小）。

根据给定的最小和最大的值的标记，`zremrangebyscore`命令会在一个分类集合里进行删除元素操作，其时间复杂度是O(log(N)+M)。这看起来似乎有点儿杂乱，通过阅读文档可以知道，这里的N指的是在分类集合里的总元素数量，而M则是被删除的元素数量。可以看出，对于性能而言，被删除的元素数量很可能会比分类集合里的总元素数量更为重要。

（译注：`zremrangebyscore`命令的具体构成是`ZREMRANGEBYSCORE Key max mix`。）

对于`sort`命令，其时间复杂度为O(N+M*log(M))，我们将会在下一章谈论更多的相关细节。从`sort`命令的性能特征来看，可以说这是Redis里最复杂的一个命令。

还存在其他的时间复杂度描述，包括O(N^2)和O(C^N)。随着N的增大，其性能将急速下降。在Redis里，没有任何一个命令具有这些类型的时间复杂度。

值得指出的一点是，在Redis里，当我们发现一些操作具有O(N)的时间复杂度时，我们可能可以找到更为好的方法去处理。

（译注：对于**Big O Notation**，相信大家都非常的熟悉，虽然原文仅仅是对该表示法进行简单的介绍，但限于个人的算法知识和文笔水平实在有限，此小节的翻译让我头痛颇久，最终成果也确实难以让人满意，望见谅。）

仿多关键字查询（Pseudo Multi Key Queries）

时常，你会想通过不同的关键字去查询相同的值。例如，你会想通过电子邮件（当用户开始登录时）去获取用户的具体信息，或者通过用户id（在用户登录后）去获取。有一种很不实效的解决方法，其将用户对象分别放置到两个字符串值里去：

```
set users:leto@dune.gov "{id: 9001, email: 'leto@dune.gov', ...}"
set users:9001 "{id: 9001, email: 'leto@dune.gov', ...}"
```

这种方法很糟糕，如此不但会产生两倍数量的内存，而且这将会成为数据管理的恶梦。

如果Redis允许你将一个关键字链接到另一个的话，可能情况会好很多，可惜Redis并没有提供这样的功能（而且很可能永远都不会提供）。Redis发展到现在，其开发的首要目的是要保持代码和API的整洁简单，关键字链接功能的内部实现并不符合这个前提（对于关键字，我们还有很多相关方法没有谈论到）。其实，Redis已经提供了解决的方法：散列。

使用散列数据结构，我们可以摆脱重复的缠绕：

```
set users:9001 "{id: 9001, email: leto@dune.gov, ...}"
hset users:lookup:email leto@dune.gov 9001
```

我们所做的是，使用域来作为一个二级索引，然后去引用单个用户对象。要通过id来获取用户信息，我们可以使用一个普通的`get`命令：

```
get users:9001
```

而如果想通过电子邮箱来获取用户信息，我们可以使用`hget`命令再配合使用`get`命令（Ruby代码）：

```
id = redis.hget('users:lookup:email', 'leto@dune.gov')
user = redis.get("users:#{id}")
```

你很可能将会经常使用这类用法。在我看来，这就是散列真正耀眼的地方。在你了解这类用法之前，这可能不是一个明显的用例。

引用和索引（References and Indexes）

我们已经看过几个关于值引用的用例，包括介绍列表数据结构时的用例，以及在上面使用散列数据结构来使查询更灵活一些。进行归纳后会发现，对于那些值与值间的索引和引用，我们都必须手动的去管理。诚实来讲，这确实会让人有点沮丧，尤其是当你想到那些引用相关的操作，如管理、更新和删除等，都必须手动的进行时。在Redis里，这个问题还没有很好的解决方法。

我们已经看到，集合数据结构很常被用来实现这类索引：

```
sadd friends:leto ghanima paul chani jessica
```

这个集合里的每一个成员都是一个Redis字符串数据结构的引用，而每一个引用的值则包含着用户对象的具体信息。那么如果`chani`改变了她的名字，或者删除了她的帐号，应该如何处理？从整个朋友圈的关系结构来看可能会更好理解，我们知道，`chani`也有她的朋友：

```
sadd friends_of:chani leto paul
```

如果你有什么待处理情况像上面那样，那在维护成本之外，还会有对于额外索引值的处理和存储空间的成本。这可能会令你感到有点退缩。在下一小节里，我们将会谈论减少使用额外数据交互的性能成本的一些方法（在第1章我们粗略地讨论了下）。

如果你确实在担忧着这些情况，其实，关系型数据库也有同样的开销。索引需要一定的存储空间，必须通过扫描或查找，然后才能找到相应的记录。其开销也是存在的，当然他们对此做了很多的优化工作，使之变得更为有效。

再次说明，需要在Redis里手动地管理引用确实是颇为棘手。但是，对于你关心的那些问题，包括性能或存储空间等，应该在经过测试后，才会有真正的理解。我想你会发现这不会是一个大问题。

数据交互和流水线（Round Trips and Pipelining）

我们已经提到过，与服务器频繁交互是Redis的一种常见模式。这类情况可能很常出现，为了使我们能获益更多，值得仔细去看看我们能利用哪些特性。

许多命令能接受一个或更多的参数，也有一种关联命令（sister-command）可以接受多个参数。例如早前我们看到过`mget`命令，接受多个关键字，然后返回值：

```
keys = redis.lrange('newusers', 0, 10)
redis.mget(*keys.map {|u| "users:#{u}"})
```

或者是`sadd`命令，能添加一个或多个成员到集合里：

```
sadd friends:vladimir piter
sadd friends:paul jessica leto "leto II" chani
```

Redis还支持流水线功能。通常情况下，当一个客户端发送请求到Redis后，在发送下一个请求之前必须等待Redis的答复。使用流水线功能，你可以发送多个请求，而不需要等待Redis响应。这不但减少了网络开销，还能获得性能上的显著提高。

值得一提的是，Redis会使用存储器去排列命令，因此批量执行命令是一个好主意。至于具体要多大的批量，将取决于你要使用什么命令（更明确来说，该参数有多大）。另一方面来看，如果你要执行的命令需要差不多50个字符的关键字，你大概可以对此进行数千或数万的批量操作。

对于不同的Redis载体，在流水线里运行命令的方式会有所差异。在Ruby里，你传递一个代码块到`pipelined`方法：

```
redis.pipelined do
  9001.times do
        redis.incr('powerlevel')
  end
end
```

正如你可能猜想到的，流水线功能可以实际地加速一连串命令的处理。

事务（Transactions）

每一个Redis命令都具有原子性，包括那些一次处理多项事情的命令。此外，对于使用多个命令，Redis支持事务功能。

你可能不知道，但Redis实际上是单线程运行的，这就是为什么每一个Redis命令都能够保证具有原子性。当一个命令在执行时，没有其他命令会运行（我们会在往后的章节里简略谈论一下Scaling）。在你考虑到一些命令去做多项事情时，这会特别的有用。例如：

`incr`命令实际上就是一个`get`命令然后紧随一个`set`命令。

`getset`命令设置一个新的值然后返回原始值。

`setnx`命令首先测试关键字是否存在，只有当关键字不存在时才设置值

虽然这些都很有用，但在实际开发时，往往会需要运行具有原子性的一组命令。若要这样做，首先要执行`multi`命令，紧随其后的是所有你想要执行的命令（作为事务的一部分），最后执行`exec`命令去实际执行命令，或者使用`discard`命令放弃执行命令。Redis的事务功能保证了什么？

- 事务中的命令将会按顺序地被执行
- 事务中的命令将会如单个原子操作般被执行（没有其它的客户端命令会在中途被执行）
- 事务中的命令要么全部被执行，要么不会执行

你可以（也应该）在命令行界面对事务功能进行一下测试。还有一点要注意到，没有什么理由不能结合流水线功能和事务功能。

```
multi
hincrby groups:1percent balance -9000000000
hincrby groups:99percent balance 9000000000
exec
```

最后，Redis能让你指定一个关键字（或多个关键字），当关键字有改变时，可以查看或者有条件地应用一个事务。这是用于当你需要获取值，且待运行的命令基于那些值时，所有都在一个事务里。对于上面展示的代码，我们不能去实现自己的`incr`命令，因为一旦`exec`命令被调用，他们会全部被执行在一块。我们不能这么做：

```
redis.multi()
current = redis.get('powerlevel')
redis.set('powerlevel', current + 1)
redis.exec()
```

（译注：虽然**Redis**是单线程运行的，但是我们可以同时运行多个**Redis**客户端进程，常见的并发问题还是会出现。像上面的代码，在`get`运行之后，`set`运行之前，`powerlevel`的值可能会被另一个**Redis**客户端给改变，从而造成错误。）

这些不是Redis的事务功能的工作。但是，如果我们增加一个`watch`到`powerlevel`，我们可以这样做：

```
redis.watch('powerlevel')
current = redis.get('powerlevel')
redis.multi()
redis.set('powerlevel', current + 1)
redis.exec()
```

在我们调用`watch`后，如果另一个客户端改变了`powerlevel`的值，我们的事务将会运行失败。如果没有客户端改变`powerlevel`的值，那么事务会继续工作。我们可以在一个循环里运行这些代码，直到其能正常工作。

关键字反模式（**Keys Anti-Pattern**）

在下一章中，我们将会谈论那些没有确切关联到数据结构的命令，其中的一些是管理或调试工具。然而有一个命令我想特别地在这里进行谈论：`keys`命令。这个命令需要一个模式，然后查找所有匹配的关键字。这个命令看起来很适合一些任务，但这不应该用在实际的产品代码里。为什么？因为这个命令通过线性扫描所有的关键字来进行匹配。或者，简单地说，这个命令太慢了。

人们会如此去使用这个命令？一般会用来构建一个本地的Bug追踪服务。每一个帐号都有一个`id`，你可能会通过一个看起来像`bug:account_id:bug_id`的关键字，把每一个Bug存储到一个字符串数据结构值中去。如果你在任何时候需要查询一个帐号的Bug（显示它们，或者当用户删除了帐号时删除掉这些Bugs），你可能会尝试去使用`keys`命令：

```
keys bug:1233:*
```

更好的解决方法应该使用一个散列数据结构，就像我们可以使用散列数据结构来提供一种方法去展示二级索引，因此我们可以使用域来组织数据：

```
hset bugs:1233 1 "{id:1, account: 1233, subject: '...'}"
hset bugs:1233 2 "{id:2, account: 1233, subject: '...'}"
```

从一个帐号里获取所有的Bug标识，可以简单地调用`hkeys bugs:1233`。去删除一个指定的Bug，可以调用`hdel bugs:1233 2`。如果要删除了一个帐号，可以通过`del bugs:1233`把关键字删除掉。

小结

结合这一章以及前一章，希望能让你得到一些洞察力，了解如何使用Redis去支持（Power）实际项目。还有其他的模式可以让你去构建各种类型的东西，但真正的关键是要理解基本的数据结构。你将能领悟到，这些数据结构是如何能够实现你最初视角之外的东西。

第4章 超越数据结构

5种数据结构组成了Redis的基础，其他没有关联特定数据结构的命令也有很多。我们已经看过一些这样的命令：`info`，`select`，`flushdb`，`multi`，`exec`，`discard`，`watch`和`keys`。这一章将看看其他的一些重要命令。

使用期限（**Expiration**）

Redis允许你标记一个关键字的使用期限。你可以给予一个Unix时间戳形式（自1970年1月1日起）的绝对时间，或者一个基于秒的存活时间。这是一个基于关键字的命令，因此其不在乎关键字表示的是哪种类型的数据结构。

```
expire pages:about 30
expireat pages:about 1356933600
```

第一个命令将会在30秒后删除掉关键字（包括其关联的值）。第二个命令则会在2012年12月31日上午12点删除掉关键字。

这让Redis能成为一个理想的缓冲引擎。通过`ttl`命令，你可以知道一个关键字还能够存活多久。而通过`persist`命令，你可以把一个关键字的使用期限删除掉。

```
ttl pages:about
persist pages:about
```

最后，有个特殊的字符串命令，`setex`命令让你可以在一个单独的原子命令里设置一个字符串值，同时里指定一个生存期（这比任何事情都要方便）。

```
setex pages:about 30 '<h1>about us</h1>....'
```

发布和订阅（Publication and Subscriptions）

Redis的列表数据结构有`blpop`和`brpop`命令，能从列表里返回且删除第一个（或最后一个）元素，或者被堵塞，直到有一个元素可供操作。这可以用来实现一个简单的队列。

（译注：对于`blpop`和`brpop`命令，如果列表里没有关键字可供操作，连接将被堵塞，直到有另外的**Redis**客户端使用`lpush`或`rpush`命令推入关键字为止。）

此外，Redis对于消息发布和频道订阅有着一流的支持。你可以打开第二个`redis-cli`窗口，去尝试一下这些功能。在第一个窗口里订阅一个频道（我们会称它为`warnings`）：

```
subscribe warnings
```

其将会答复你订阅的信息。现在，在另一个窗口，发布一条消息到`warnings`频道：

```
publish warnings "it's over 9000!"
```

如果你回到第一个窗口，你应该已经接收到`warnings`频道发来的消息。

你可以订阅多个频道（`subscribe channel1 channel2 ...`），订阅一组基于模式的频道（`psubscribe warnings:*`），以及使用`unsubscribe`和`punsubscribe`命令停止监听一个或多个频道，或一个频道模式。

最后，可以注意到`publish`命令的返回值是1，这指出了接收到消息的客户端数量。

监控和延迟日志（Monitor and Slow Log）

`monitor`命令可以让你查看Redis正在做什么。这是一个优秀的调试工具，能让你了解你的程序如何与Redis进行交互。在两个`redis-cli`窗口中选一个（如果其中一个还处于订阅状态，你可以使用`unsubscribe`命令退订，或者直接关掉窗口再重新打开一个新窗口）键入`monitor`命令。在另一个窗口，执行任何其他类型的命令（例如`get`或`set`命令）。在第一个窗口里，你应该可以看到这些命令，包括他们的参数。

在实际生产环境里，你应该谨慎运行`monitor`命令，这真的仅仅就是一个很有用的调试和开发工具。除此之外，没有更多要说的了。

随同`monitor`命令一起，Redis拥有一个`slowlog`命令，这是一个优秀的性能剖析工具。其会记录执行时间超过一定数量微秒的命令。在下一章节，我们会简略地涉及如何配置Redis，现在你可以按下面的输入配置Redis去记录所有的命令：

```
config set slowlog-log-slower-than 0
```

然后，执行一些命令。最后，你可以检索到所有日志，或者检索最近的那些日志：

```
slowlog get
slowlog get 10
```

通过键入`slowlog len`，你可以获取延迟日志里的日志数量。

对于每个被你键入的命令，你应该查看4个参数：

- 一个自动递增的id
- 一个Unix时间戳，表示命令开始运行的时间
- 一个微妙级的时间，显示命令运行的总时间
- 该命令以及所带参数

延迟日志保存在存储器中，因此在生产环境中运行（即使有一个低阀值）也应该不是一个问题。默认情况下，它将会追踪最近的1024个日志。

排序（Sort）

`sort`命令是Redis最强大的命令之一。它让你可以在一个列表、集合或者分类集合里对值进行排序（分类集合是通过标记来进行排序，而不是集合里的成员）。下面是一个`sort`命令的简单用例：

```
rpush users:leto:guesses 5 9 10 2 4 10 19 2
sort users:leto:guesses
```

这将返回进行升序排序后的值。这里有一个更高级的例子：

```
sadd friends:ghanima leto paul chani jessica alia duncan
sort friends:ghanima limit 0 3 desc alpha
```

上面的命令向我们展示了，如何对已排序的记录进行分页（通过`limit`），如何返回降序排序的结果（通过`desc`），以及如何用字典序排序代替数值序排序（通过`alpha`）。

`sort`命令的真正力量是其基于引用对象来进行排序的能力。早先的时候，我们说明了列表、集合和分类集合很常被用于引用其他的Redis对象，`sort`命令能够解引用这些关系，而且通过潜在值来进行排序。例如，假设我们有一个Bug追踪器能让用户看到各类已存在问题。我们可能使用一个集合数据结构去追踪正在被监视的问题：

```
sadd watch:leto 12339 1382 338 9338
```

你可能会有强烈的感觉，想要通过id来排序这些问题（默认的排序就是这样的），但是，我们更可能是通过问题的严重性来对这些问题进行排序。为此，我们要告诉Redis将使用什么模式来进行排序。首先，为了可以看到一个有意义的结果，让我们添加多一点数据：

```
set severity:12339 3
set severity:1382 2
set severity:338 5
set severity:9338 4
```

要通过问题的严重性来降序排序这些Bug，你可以这样做：

```
sort watch:leto by severity:* desc
```

Redis将会用存储在列表（集合或分类集合）中的值去替代模式中的`*`（通过`by`）。这会创建出关键字名字，Redis将通过查询其实际值来排序。

在Redis里，虽然你可以有成千上万个关键字，类似上面展示的关系还是会引起一些混乱。幸好，`sort`命令也可以工作在散列数据结构及其相关域里。相对于拥有大量的高层次关键字，你可以利用散列：

```
hset bug:12339 severity 3
hset bug:12339 priority 1
hset bug:12339 details "{id: 12339, ....}"

hset bug:1382 severity 2
hset bug:1382 priority 2
hset bug:1382 details "{id: 1382, ....}"

hset bug:338 severity 5
hset bug:338 priority 3
hset bug:338 details "{id: 338, ....}"

hset bug:9338 severity 4
hset bug:9338 priority 2
hset bug:9338 details "{id: 9338, ....}"
```

所有的事情不仅变得更为容易管理，而且我们能通过`severity`或`priority`来进行排序，还可以告诉`sort`命令具体要检索出哪一个域的数据：

```
sort watch:leto by bug:*->priority get bug:*->details
```

相同的值替代出现了，但Redis还能识别`->`符号，用它来查看散列中指定的域。里面还包括了`get`参数，这里也会进行值替代和域查看，从而检索出Bug的细节（details域的数据）。

对于太大的集合，`sort`命令的执行可能会变得很慢。好消息是，`sort`命令的输出可以被存储起来：

```
sort watch:leto by bug:*->priority get bug:*->details store watch_by_priority:leto
```

使用我们已经看过的`expiration`命令，再结合`sort`命令的`store`能力，这是一个美妙的组合。

小结

这一章主要关注那些非特定数据结构关联的命令。和其他事情一样，它们的使用依情况而定。构建一个程序或特性时，可能不会用到使用期限、发布和订阅或者排序等功能。但知道这些功能的存在是很好的。而且，我们也只接触到了一些命令。还有更多的命令，当你消化理解完这本书后，非常值得去浏览一下完整的命令列表。

第5章 - 管理

在最后一章里，我们将集中谈论Redis运行中的一些管理方面内容。这是一个不完整的Redis管理指南，我们将会回答一些基本的问题，初接触Redis的新用户可能会很感兴趣。

配置（Configuration）

当你第一次运行Redis的服务器，它会向你显示一个警告，指`redis.conf`文件没有被找到。这个文件可以被用来配置Redis的各个方面。一个充分定义（well-documented）的`redis.conf`文件对各个版本的Redis都有效。范例文件包含了默认的配置选项，因此，对于想要了解设置在干什么，或默认设置是什么，都会很有用。你可以在https://github.com/antirez/redis/raw/2.4.6/redis.conf找到这个文件。

这个配置文件针对的是**Redis 2.4.6**，你应该用你的版本号替代上面**URL**里的**"2.4.6"**。运行`info`命令，其显示的第一个值就是**Redis**的版本号。

因为这个文件已经是充分定义（well-documented），我们就不去再进行设置了。

除了通过`redis.conf`文件来配置Redis，`config set`命令可以用来对个别值进行设置。实际上，在将`slowlog-log-slower-than`设置为0时，我们就已经使用过这个命令了。

还有一个`config get`命令能显示一个设置值。这个命令支持模式匹配，因此如果我们想要显示关联于日志（logging）的所有设置，我们可以这样做：

```
config get *log*
```

验证（**Authentication**）

通过设置`requirepass`（使用`config set`命令或`redis.conf`文件），可以让Redis需要一个密码验证。
当`requirepass`被设置了一个值（就是待用的密码），客户端将需要执行一个`auth password`命令。

一旦一个客户端通过了验证，就可以在任意数据库里执行任何一条命令，包括`flushall`命令，这将会清除掉每一个数据库里的所有关键字。通过配置，你可以重命名一些重要命令为混乱的字符串，从而获得一些安全性。

```
rename-command CONFIG 5ec4db169f9d4dddacbfb0c26ea7e5ef
rename-command FLUSHALL 1041285018a942a4922cbf76623b741e
```

或者，你可以将新名字设置为一个空字符串，从而禁用掉一个命令。

大小限制（**Size Limitations**）

当你开始使用Redis，你可能会想知道，我能使用多少个关键字？还可能想知道，一个散列数据结构能有多少个域（尤其是当你用它来组织数据时），或者是，一个列表数据结构或集合数据结构能有多少个元素？对于每一个实例，实际限制都能达到亿万级别（hundreds of millions）。

复制（**Replication**）

Redis支持复制功能，这意味着当你向一个Redis实例（Master）进行写入时，一个或多个其他实例（Slaves）能通过Master实例来保持更新。可以在配置文件里设置`slaveof`，或使用`slaveof`命令来配置一个Slave实例。对于那些没有进行这些设置的Redis实例，就可能一个Master实例。

为了更好保护你的数据，复制功能拷贝数据到不同的服务器。复制功能还能用于改善性能，因为读取请求可以被发送到Slave实例。他们可能会返回一些稍微滞后的数据，但对于大多数程序来说，这是一个值得做的折衷。

遗憾的是，Redis的复制功能还没有提供自动故障恢复。如果Master实例崩溃了，一个Slave实例需要手动的进行升级。如果你想使用Redis去达到某种高可用性，对于使用心跳监控（heartbeat monitoring）和脚本自动开关（scripts to automate the switch）的传统高可用性工具来说，现在还是一个棘手的难题。

备份文件（**Backups**）

备份Redis非常简单，你可以将Redis的快照（snapshot）拷贝到任何地方，包括S3、FTP等。默认情况下，Redis会把快照存储为一个名为`dump.rdb`的文件。在任何时候，你都可以对这个文件执行`scp`、`ftp`或`cp`等常用命令。

有一种常见情况，在Master实例上会停用快照以及单一附加文件（aof），然后让一个Slave实例去处理备份事宜。这可以帮助减少Master实例的载荷。在不损害整体系统响应性的情况下，你还可以在Slave实例上设置更多主动存储的参数。

缩放和**Redis**集群（**Scaling and Redis Cluster**）

复制功能（Replication）是一个成长中的网站可以利用的第一个工具。有一些命令会比另外一些来的昂贵（例如`sort`命令），将这些运行载荷转移到一个Slave实例里，可以保持整体系统对于查询的快速响应。

此外，通过分发你的关键字到多个Redis实例里，可以达到真正的缩放Redis（记住，Redis是单线程的，这些可以运行在同一个逻辑框里）。随着时间的推移，你将需要特别注意这些事情（尽管许多的Redis载体都提供了consistent-hashing算法）。对于数据水平分布（horizontal distribution）的考虑不在这本书所讨论的范围内。这些东西你也很可能不需要去担心，但是，无论你使用哪一种解决方案，有一些事情你还是必须意识到。

好消息是，这些工作都可在Redis集群下进行。不仅提供水平缩放（包括均衡），为了高可用性，还提供了自动故障恢复。

高可用性和缩放是可以达到的，只要你愿意为此付出时间和精力，Redis集群也使事情变得简单多了。

小结

在过去的一段时间里，已经有许多的计划和网站使用了Redis，毫无疑问，Redis已经可以应用于实际生产中了。然而，一些工具还是不够成熟，尤其是一些安全性和可用性相关的工具。对于Redis集群，我们希望很快就能看到其实现，这应该能为一些现有的管理挑战提供处理帮忙。

总结

在许多方面，Redis体现了一种简易的数据处理方式，其剥离掉了大部分的复杂性和抽象，并可有效的在不同系统里运行。不少情况下，选择Redis不是最佳的选择。在另一些情况里，Redis就像是为你的数据提供了特别定制的解决方案。

最终，回到我最开始所说的：Redis很容易学习。现在有许多的新技术，很难弄清楚哪些才真正值得我们花时间去学习。如果你从实际好处来考虑，Redis提供了他的简单性。我坚信，对于你和你的团队，学习Redis是最好的技术投资之一。