Redis3.0中文版

极客学院出版

前言

Redis 3.0.0 正式版终于到来了！最重要的新特性是集群（Redis Cluster），提供 Redis 功能子集（比如不支持多数据库）的分布式、容错的实现（最多支持 1000 结点）。

本教程是 Redis 3.0 官方文档的翻译版，内容上突出新特性，旨在帮助读者快速上手并掌握 Redis 3.0 的知识点。

适用人群

本教程为中级教程，适用于基于 Redis 数据库的应用开发者。

学习前提

学习本教程前，我们假定你已经对 Redis 有一定的了解。

鸣谢：http://powersoft.iteye.com/blog/2126831

目录

1 Redis 介绍

Redis 是一款开源的，基于 BSD 许可的，高级键值 (key-value) 缓存 (cache) 和存储 (store) 系统。由于 Redis 的键包括 string，hash，list，set，sorted set，bitmap 和 hyperloglog，所以常常被称为数据结构服务器。你可以在这些类型上面运行原子操作，例如，追加字符串，增加哈希中的值，加入一个元素到列表，计算集合的交集、并集和差集，或者是从有序集合中获取最高排名的元素。

为了满足高性能，Redis 采用内存 (in-memory) 数据集 (dataset)。根据你的使用场景，你可以通过每隔一段时间转储数据集到磁盘，或者追加每条命令到日志来持久化。持久化也可以被禁用，如果你只是需要一个功能丰富，网络化的内存缓存。

Redis 还支持主从异步复制，非常快的非阻塞初次同步、网络断开时自动重连局部重同步。 其他特性包括：

- 事务
- 订阅/发布
- Lua 脚本
- 带 TTL 的键
- LRU 回收健
- 自动故障转移 (failover)

你可以通过多种语言来使用 Redis。

Redis 是由 ANSI C 语言编写的，在无需额外依赖下，运行于大多数 POSIX 系统，如 Linux、*BSD、OS X。Redis 是在 Linux 和 OS X 两款操作系统下开发和充分测试的，我们推荐 Linux 为部署环境。Redis 也可以运行在 Solaris 派生系统上，如 SmartOS，但是支持有待加强。没有官方支持的 Windows 构建版本，但是微软开发和维护了一个 64 位 Windows 的版本。

2

数据类型初探

字符串 (Strings)

字符串是 Redis 最基本的数据类型。Redis 字符串是二进制安全的，也就是说，一个 Redis 字符串可以包含任意类型的数据，例如一张 JPEG 图像，或者一个序列化的 Ruby 对象。

一个字符串最大为 512M 字节。

你可以使用 Redis 的字符串类型做很多有意思的事情，例如，你可以：

- 使用 INCR 命令族 (INCR，DECR，INCRBY)，将字符串作为原子计数器。
- 使用 APPEND 命令追加字符串。
- 使用 GETRANGE 和 SETRANGE 命令，使字符串作为随机访问向量 (vectors)。
- 编码大量数据到很小的空间，或者使用 GETBIT 和 SETBIT 命令，创建一个基于 Redis 的布隆 (Bloom) 过滤器。

后续我们会详细介绍可用的字符串命令，也会详细介绍 Redis 数据类型的更多高级信息。

列表 (Lists)

Redis 列表仅仅是按照插入顺序排序的字符串列表。可以添加一个元素到 Redis 列表的头部 (左边) 或者尾部 (右边)。

LPUSH 命令用于插入一个元素到列表的头部，RPUSH 命令用于插入一个元素到列表的尾部。当这两个命令操作在一个不存在的键时，将会创建一个新的列表。同样，如果一个操作会清空列表，那么该键将会从键空间 (key space) 移除。这些是非常方便的语义，因为列表命令如果使用不存在的键作为参数，就会表现得像命令运行在一个空列表上一样。

一些列表操作的例子结果：

```
LPUSH mylist a   # now the list is “a”
LPUSH mylist b   # now the list is “b”,”a”
RPUSH mylist c   # now the list is “b”,”a”,”c” (RPUSH was used this time)
```

列表的最大长度是 223−1 个元素 (4294967295，超过 40 亿个元素)。

从时间复杂度的角度看，Redis 列表主要的特性是支持以常量时间在列表的头和尾附近插入和删除元素，即使列表中已经插入了上百万的数据。访问列表两端的元素非常的快速，但是访问一个非常大的列表的中间却非常的慢，因为这是一个 O(N)时间复杂度的操作。

你可以使用 Redis 的列表类型做很多有意思的事情，例如，你可以：

- 为社交网络时间轴 (timeline) 建模，使用 LPUSH 命令往用户时间轴插入元素，使用 LRANGE 命令获得最近事项。
- 使用 LPUSH 和 LTRIM 命令创建一个不会超出给定数量元素的列表，只存储最近的 N 个元素。
- 列表可以用作消息传递原语，例如，众所周知的用于创建后台任务的 Ruby 库 Resque。
- 你可以用列表做更多的事情，这种数据类型支持很多的命令，包括阻塞命令，如 BLPOP。

后续我们会详细介绍可用的列表命令，也会详细介绍 Redis 数据类型的更多高级信息。

集合 (Sets)

Redis 集合是没有顺序的字符串集合 (collection)。可以在 O(1) 的时间复杂度添加、删除和测试元素存在与否 (不管集合中有多少元素都是常量时间)。

Redis 集合具有你需要的不允许重复成员的性质。多次加入同一个元素到集合也只会有一个拷贝在其中。实际上，这意味着加入一个元素到集合中并不需要检查元素是否已存在。

Redis 集合非常有意思的是，支持很多服务器端的命令，可以在很短的时间内和已经存在的集合一起计算并集，交集和差集。

你可以使用 Redis 的集合类型做很多有意思的事情，例如，你可以：

- 你可以使用 Redis 集合追踪唯一性的事情。你想知道访问某篇博客文章的所有唯一 IP 吗？只要 每次页面访问时使用 SADD 命令就可以了。你可以放心，重复的 IP 是不会被插入进来的。
- Redis 集合可以表示关系。你可以通过使用集合来表示每个标签，来创建一个标签系统。然后你可以把所有拥有此标签的对象的 ID 通过 SADD 命令，加入到表示这个标签的集合中。你想获得同时拥有三个不同标签的对象的全部 ID 吗？用 SINTER 就可以了。
- 你可以使用 SPOP 或 SRANDMEMBER 命令来从集合中随机抽取元素。

后续我们会详细介绍可用的集合命令，也会详细介绍 Redis 数据类型的更多高级信息。

哈希 / 散列 (Hashes)

Redis 哈希是字符串字段 (field) 与字符串值之间的映射，所以是表示对象的理想数据类型 (例如：一个用户对象有多个字段，像用户名，姓氏，年龄等等)：

```
@cli
HMSET user:1000 username antirez password P1pp0 age 34
HGETALL user:1000
HSET user:1000 password 12345
HGETALL user:1000
```

拥有少量字段 (少量指的是大约 100) 的哈希会以占用很少存储空间的方式存储，所以你可以在一个很小的 Redis 实例里存储数百万的对象。

由于哈希主要用来表示对象，对象能存储很多元素，所以你可以用哈希来做很多其他的事情。

每个哈希可以存储多达 223-1 个字段值对 (field-value pair)(多于 40 亿个)。

后续我们会详细介绍可用的哈希命令，也会详细介绍 Redis 数据类型的更多高级信息。

有序集合 (Sorted sets)

Redis 有序集合和 Redis 集合类似，是非重复字符串集合 (collection)。不同的是，每一个有序集合的成员都有一个关联的分数 (score)，用于按照分数高低排序。尽管成员是唯一的，但是分数是可以重复的。

对有序集合我们可以通过很快速的方式添加，删除和更新元素 (在和元素数量的对数成正比的时间内)。由于元素是有序的而无需事后排序，你可以通过分数或者排名 (位置) 很快地来获取一个范围内的元素。访问有序集合的中间元素也是很快的，所以你可以使用有序集合作为一个无重复元素，快速访问你想要的一切的聪明列表：有序的元素，快速的存在性测试，快速的访问中间元素！

总之，有序集合可以在很好的性能下，做很多别的数据库无法模拟的事情。

使用有序集合你可以：

例如多人在线游戏排行榜，每次提交一个新的分数，你就使用 ZADD 命令更新。你可以很容易地使用 ZRANGE 命令获取前几名用户，你也可以用 ZRANK 命令，通过给定用户名返回其排行。同时使用 ZRANK 和 ZRANGE 命令可以展示与给定用户相似的用户及其分数。以上这些操作都非常的快。

有序集合常用来索引存储在 Redis 内的数据。例如，假设你有很多表示用户的哈希，你可以使用有序集合，用年龄作为元素的分数，用用户 ID 作为元素值，于是你就可以使用 ZRANGEBYSCORE 命令很快且轻而易举地检索出给定年龄区间的所有用户了。

有序集合或许是最高级的 Redis 数据类型，后续我们会详细介绍可用的有序集合命令，也会详细介绍 Redis 数据类型的更多高级信息。

位图 (Bitmaps) 和超重对数 (HyperLogLogs)

Redis 还支持位图和超重对数这两种基于字符串基本类型，但有自己语义的数据类型。

后续我们会详细介绍这些数据类型的更多高级信息。

3

从入门到精通（上）

Redis 不是一个无格式 (plain) 的键值存储，而是一个支持各种不同类型值的数据结构服务器。这就是说，传统键值存储是关联字符串值到字符串键，但是 Redis 的值不仅仅局限于简单字符串，还可以持有更复杂的数据结构。下面列的是 Redis 支持的所有数据结构，后面将逐一介绍：

- 二进制安全 (binary-safe) 的字符串。
- 列表：按照插入顺序排序的字符串元素 (element) 的集合 (collection)。通常是链表。
- 集合：唯一的，无序的字符串元素集合。
- 有序集合：和集合类似，但是每个字符串元素关联了一个称为分数 (score) 的浮点数。元素总是按照分数排序，所以可以检索一个范围的元素 (例如，给我前 10，或者后 10 个元素)。
- 哈希：由字段 (field) 及其关联的值组成的映射。字段和值都是字符串类型。这非常类似于 Ruby 或 Python 中的哈希 / 散列。
- 位数组 (位图)：使用特殊的命令，把字符串当做位数组来处理：你可以设置或者清除单个位值，统计全部置位为 1 的位个数，寻找第一个复位或者置位的位，等等。
- 超重对数 (HyperLogLog)：这是一个用于估算集合的基数 (cardinality，也称势，译者注) 的概率性数据结构。不要害怕，它比看起来要简单，稍后为你揭晓。

理解这些数据结构是如何工作的，以及从命令参考手册中选择什么命令来解决实际问题，并不总是一件繁琐的事情，本文档就是学习 Redis 数据结构及其最常用模式的速成班。

后面的所有例子我们都是使用 redis-cli 工具，这是一个简单而又方便的命令行工具，用于发送命令给 Redis 服务器。

Redis 键 (Keys)

Redis 键是二进制安全的，这意味着你可以使用任何二进制序列作为键，从像”foo” 这样的字符串到一个 JPEG 文件的内容。空字符串也是合法的键。

关于键的其他一些规则：

- 不要使用太长的键，例如，不要使用一个 1024 字节的键，不仅是因为内存占用，而且在数据集中查找键时需要多次耗时的键比较。即使手头需要匹配一个很大值的存在性，对其进行哈希 (例如使用 SHA1) 是个不错的主意，尤其是从内存和带宽的角度。
- 不要使用太短的键。用”u1000flw” 取代”user:1000:followers” 作为键并没有什么实际意义，后者更具有可读性，相对于键对象本身以及值对象来说，增加的空间微乎其微。然而不可否认，短的键会消耗少的内存，你的任务就是要找到平衡点。
- 坚持一种模式 (schema)。例如，”object-type:id” 就不错，就像”user:1000”。点或者横线常用来连接多单词字段，如”comment:1234:reply.to”，或者”comment:1234:reply-to”。
- 键的最大大小是 512MB。

Redis 字符串 (Strings)

Redis 字符串是可以关联给 redis 键的最简单值类型。字符串是 Memcached 的唯一数据类型，所以新手使用起来也是很自然的。

由于 Redis 的键也是字符串，当我们使用字符串作为值的时候，我们是将一个字符串映射给另一个字符串。字符串数据类型适用于很多场景，例如，缓存 HTML 片段或者页面。

让我们用 redis-cli 来玩玩字符串类型 (接下来的例子都是使用 redis-cli)。

```
> set mykey somevalue
OK
> get mykey
"somevalue"
```

你可以看到，我们使用 SET 和 GET 命令设置和检索字符串值。注意，如果键已经存在，SET 会替换掉该键已经存在的值，哪怕这个键关联的是一个非字符串类型的值。SET 执行的是赋值操作。

值可以是任何类型的字符串 (包括二进制数据)，例如，你可以存储一个 JPEG 图像。值不能大于 512MB。

SET 命令还有一些以额外的参数形式提供有意思的选项。例如，如果我要求如果键存在 (或刚好相反) 则执行失败，也就是说健存在才成功：

```
> set mykey newval nx
(nil)
> set mykey newval xx
OK
```

尽管字符串是 Redis 最基本的值类型，你仍可以执行很多有趣的操作。例如，原子性增长：

```
> set counter 100
OK
> incr counter
(integer) 101
> incr counter
(integer) 102
> incrby counter 50
(integer) 152
```

INCR 命令将字符串值解析为整数，并增加一，最后赋值后作为新值。还有一些类似的命令 INCRBY，DECR 和 DECRBY。它们以略微不同的方式执行，但其内部都是一样的命令。

为什么说 INCR 命令是原子的？因为即使多个 客户端对同一个键发送 INCR 命令也不会造成竞争条件 (race condition)。例如，一定不会发生客户端 1 和客户端 2 同时读到”10”，都增加到 11，然后设置新值为 11。最后的结果将会一直是 12，读 – 增加 – 写操作在执行时，其他客户端此时不会执行相关命令。

有许多操作字符串的命令。例如，GETSET 命令给键设置一个新值，同时返回旧值。你可以使用这个命令，例如，如果你有一个系统，每当收到一个访问请求就使用 INRC 来增加一个键。你想每隔一个小时收集一次这个信息，而不想漏掉任何一个增长。你可以使用 GETSET，将新值赋值为 0，人后读取其旧值。

在一个命令中一次设置或者检索多个键有利于减少延迟。为此有了 MSET 和 MGET 命令：

```
> mset a 10 b 20 c 30
OK
> mget a b c
1) "10"
2) "20"
3) "30"
```

当使用 MSET 时，Redis 返回一个值数组。

改变和查询键空间 (key space)

有一些命令并不定义在特定的类型上，但是对键空间的交互很有用，因此他们能作用在任意键上。

例如，EXISTS 命令返回 1 或者 0，来表示键在数据库中是否存在。另外，DEL 命令删除键及其关联的值，无论值是什么。

```
> set mykey hello
OK
> exists mykey
(integer) 1
> del mykey
(integer) 1
> exists mykey
(integer) 0
```

从上面的例子中我们还可以看到，DEL 命令本身也会返回 1 或者 0，无论键是(存在)否(不存在)删除。

有许多键空间相关的命令，但是上面两个命令与 TYPE 命令关系紧密，TYPE 命令返回某个键的值的类型。

```
> set mykey x
OK
> type mykey
string
> del mykey
(integer) 1
> type mykey
none
```

Redis 过期 (expires)：有限生存时间的键

在我们继续更复杂的数据结构之前，我们先抛出一个与类型无关的特性， 称为 Redis 过期 。你可以给键设置超时，也就是一个有限的生存时间。当生存时间到了，键就会自动被销毁，就像用户调用 DEL 命令一样。

快速过一下 Redis 过期的信息：

- 过期时间可以设置为秒或者毫秒精度。
- 过期时间分辨率总是 1 毫秒。
- 过期信息被复制和持久化到磁盘，当 Redis 停止时时间仍然在计算 (也就是说 Redis 保存了过期时间)。

设置过期非常简单：

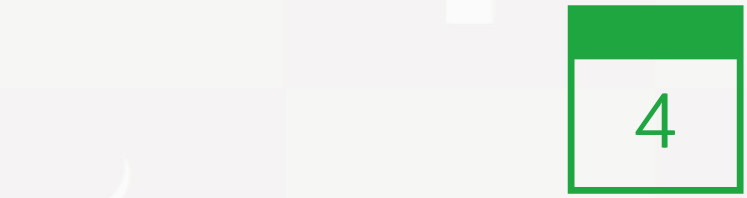
```
> set key some-value
OK
> expire key 5
(integer) 1
> get key (immediately)
"some-value"
> get key (after some time)
(nil)
```

键在两次 GET 调用期间消失了，因为第二次调用推迟了超过 5 秒。在上面的例子中，我们使用 EXPIRE 命令设置过期 (也可以为一个已经设置过期时间的键设置不同的过期时间，就像 PERSIST 命令可以删除过期时间使键永远存在)。当然我们也可以使用其他 Redis 命令来创建带过期时间的键。例如使用 SET 选项：

```
> set key 100 ex 10
OK
> ttl key
(integer) 9
```

上面的例子中设置 10 秒过期的键，值为字符串 100。然后使用 TTL 命令检查键的生存剩余时间。

为了使用毫秒来设置和检查过期，请查看 PEXPIRE 和 PTTL 命令，以及 SET 命令的全部选项。



4

从入门到精通（中）

Redis 列表(Lists)

为了解释列表类型，最好先开始来点理论，因为列表这个术语在信息技术领域常常使用不当。例如，”Python Lists”，并不是字面意思(链表)，实际是表示数组 (和 Ruby 中的 Array 是同一种类型)。

通常列表表示有序元素的序列：10，20，1，2，3 是一个列表。但是数组实现的列表和链表实现的列表，他们的属性非常不同。

Redis 的列表是使用链表实现的。这意味着，及时你的列表中有上百万个元素，增加一个元素到列表的头部或者尾部的操作都是在常量时间完成。使用 LPUSH 命令增加一个新元素到拥有 10 个元素的列表的头部的速度，与增加到拥有 1000 万个元素的列表的头部是一样的。

缺点又是什么呢？使用索引下标来访问一个数组实现的列表非常快(常量时间)，但是访问链表实现列表就没那么快了(与元素索引下标成正比的大量工作)。

Redis 采用链表来实现列表是因为，对于数据库系统来说，快速插入一个元素到一个很长的列表非常重要。另外一个即将描述的优势是，Redis 列表能在常数时间内获得常数长度。

如果需要快速访问一个拥有大量元素的集合的中间数据，可以用另一个称为有序集合的数据结构。稍后将会介绍有序集合。

Redis 列表起步

LPUSH 命令从左边 (头部) 添加一个元素到列表，RPUSH 命令从右边(尾部)添加一个元素的列表。LRANGE 命令从列表中提取一个范围内的元素。

```
> rpush mylist A
(integer) 1
> rpush mylist B
(integer) 2
> lpush mylist first
(integer) 3
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
```

注意 LRANGE 命令使用两个索引下标，分别是返回的范围的开始和结束元素。两个索引坐标可以是负数，表示从后往前数，所以 - 1 表示最后一个元素，-2 表示倒数第二个元素，等等。

如你所见，RPUSH 添加元素到列表的右边，LPUSH 添加元素到列表的左边。

两个命令都是可变参数命令，也就是说，你可以在一个命令调用中自由的添加多个元素到列表中：

```
> rpush mylist 1 2 3 4 5 "foo bar"
(integer) 9
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
4) "1"
5) "2"
6) "3"
7) "4"
8) "5"
9) "foo bar"
```

定义在 Redis 列表上的一个重要操作是弹出元素。弹出元素指的是从列表中检索元素，并同时将其从列表中清楚的操作。你可以从左边或者右边弹出元素，类似于你可以从列表的两端添加元素。

```
> rpush mylist a b c
(integer) 3
> rpop mylist
"c"
```

```
> rpop mylist
"b"
> rpop mylist
"a"
```

我们添加了三个元素并且又弹出了三个元素，所以这一串命令执行完以后列表是空的，没有元素可以弹出了。如果我们试图再弹出一个元素，就会得到如下结果：

```
> rpop mylist
(nil)
```

Redis 返回一个 NULL 值来表明列表中没有元素了。

列表的通用场景(Common use cases)

列表可以完成很多任务，两个有代表性的场景如下：

- 记住社交网络中用户最近提交的更新。
- 使用生产者消费者模式来进程间通信，生产者添加项(item)到列表，消费者(通常是 worker)消费项并执行任务。Redis 有专门的列表命令更加可靠和高效的解决这种问题。

例如，两种流行的 Ruby 库 resque 和 sidekiq，都是使用 Redis 列表作为钩子，来实现后台作业 (background jobs)。

流行的 Twitter 社交网络，使用 Redis 列表来存储用户最新的微博 (tweets)。

为了一步一步的描述通用场景，假设你想加速展现照片共享社交网络主页的最近发布的图片列表。

每次用户提交一张新的照片，我们使用 LPUSH 将其 ID 添加到列表。

当用户访问主页时，我们使用 LRANGE 0 9 获取最新的 10 张照片。

上限列表(Capped)

很多时候我们只是想用列表存储最近的项，随便这些项是什么：社交网络更新，日志或者任何其他东西。

Redis 允许使用列表作为一个上限集合，使用 LTRIM 命令仅仅只记住最新的 N 项，丢弃掉所有老的项。

LTRIM 命令类似于 LRANGE，但是不同于展示指定范围的元素，而是将其作为列表新值存储。所有范围外的元素都将被删除。

举个例子你就更清楚了：

```
> rpush mylist 1 2 3 4 5
(integer) 5
> ltrim mylist 0 2
OK
> lrange mylist 0 -1
1) "1"
2) "2"
3) "3"
```

上面 LTRIM 命令告诉 Redis 仅仅保存第 0 到 2 个元素，其他的都被抛弃。这可以让你实现一个简单而又有用的模式，一个添加操作和一个修剪操作一起，实现新增一个元素抛弃超出元素。

```
LPUSH mylist <some element>
LTRIM mylist 0 999
```

上面的组合增加一个元素到列表中，同时只持有最新的 1000 个元素。使用 LRANGE 命令你可以访问前几个元素而不用记录非常老的数据。

注意：尽管 LRANGE 是一个O(N)时间复杂度的命令，访问列表头尾附近的小范围是常量时间的操作。

列表的阻塞操作 (blocking)

列表有一个特别的特性使得其适合实现队列，通常作为进程间通信系统的积木：阻塞操作。

假设你想往一个进程的列表中添加项，用另一个进程来处理这些项。这就是通常的生产者消费者模式，可以使用以下简单方式实现：

- 生产者调用 LPUSH 添加项到列表中。
- 消费者调用 RPOP 从列表提取 / 处理项。

然而有时候列表是空的，没有需要处理的，RPOP 就返回 NULL。所以消费者被强制等待一段时间并重试 RPOP 命令。这称为轮询(polling)，由于其具有一些缺点，所以不合适在这种情况下：

1. 强制 Redis 和客户端处理无用的命令 (当列表为空时的所有请求都没有执行实际的工作，只会返回 NULL)。
2. 由于工作者受到一个 NULL 后会等待一段时间，这会延迟对项的处理。

于是 Redis 实现了 BRPOP 和 BLPOP 两个命令，它们是当列表为空时 RPOP 和 LPOP 的会阻塞版本：仅当一个新元素被添加到列表时，或者到达了用户的指定超时时间，才返回给调用者。 这个是我们在工作者中调用 BRPOP 的例子：

```
> brpop tasks 5
1) "tasks"
2) "do_something"
```

上面的意思是” 等待 tasks 列表中的元素，如果 5 秒后还没有可用元素就返回”。

注意，你可以使用 0 作为超时让其一直等待元素，你也可以指定多个列表而不仅仅只是一个，同时等待多个列表，当第一个列表收到元素后就能得到通知。

关于 BRPOP 的一些注意事项。

1. 客户端按顺序服务：第一个被阻塞等待列表的客户端，将第一个收到其他客户端添加的元素，等等。
2. 与 RPOP 的返回值不同：返回的是一个数组，其中包括键的名字，因为 BRPOP 和 BLPOP 可以阻塞等待多个列表的元素。
3. 如果超时时间到达，返回 NULL。

还有更多你需要知道的关于列表和阻塞选项，建议你阅读下面的页面：

- 使用 RPOLPUSH 构建更安全的队列和旋转队列。

- BRPOPLPUSH 命令是其阻塞变种命令。

自动创建和删除键

到目前为止的例子中，我们还没有在添加元素前创建一个空的列表，也没有删除一个没有元素的空列表。要注意，当列表为空时 Redis 将删除该键，当向一个不存在的列表键(如使用 LPUSH)添加一个元素时，将创建一个空的列表。

这并不只是针对列表，适用于所有 Redis 多元素组成的数据类型，因此适用于集合，有序集合和哈希。

基本上我们可以概括为三条规则：

1. 当我们向聚合(aggregate)数据类型添加一个元素，如果目标键不存在，添加元素前将创建一个空的聚合数据类型。
2. 当我们从聚合数据类型删除一个元素，如果值为空，则键也会被销毁。
3. 调用一个像 LLEN 的只读命令(返回列表的长度)，或者一个写命令从空键删除元素，总是产生和操作一个持有空聚合类型值的键一样的结果。

规则 1 的例子：

```
> del mylist
(integer) 1
> lpush mylist 1 2 3
(integer) 3
```

然而，我们不能执行一个错误键类型的操作：

```
> set foo bar
OK
> lpush foo 1 2 3
(error) WRONGTYPE Operation against a key holding the wrong kind of value
> type foo
string
```

规则 2 的例子：

```
> lpush mylist 1 2 3
(integer) 3
> exists mylist
(integer) 1
> lpop mylist
"3"
> lpop mylist
```

```
"2"
> lpop mylist
"1"
> exists mylist
(integer) 0
```

当所有元素弹出后，键就不存在了。

规则 3 的例子：

```
> del mylist
(integer) 0
> llen mylist
(integer) 0
> lpop mylist
(nil)
```

Redis 哈希/散列 (Hashes)

Redis 哈希看起来正如你期待的那样：

```
> hmset user:1000 username antirez birthyear 1977 verified 1
OK
> hget user:1000 username
"antirez"
> hget user:1000 birthyear
"1977"
> hgetall user:1000
1) "username"
2) "antirez"
3) "birthyear"
4) "1977"
5) "verified"
6) "1"
```

哈希就是字段值对(fields-values pairs)的集合。由于哈希容易表示对象，事实上哈希中的字段的数量并没有限制，所以你可以在你的应用程序以不同的方式来使用哈希。

HMSET 命令为哈希设置多个字段，HGET 检索一个单独的字段。HMGET 类似于 HGET，但是返回值的数组：

```
> hmget user:1000 username birthyear no-such-field
1) "antirez"
2) "1977"
3) (nil)
```

也有一些命令可以针对单个字段执行操作，例如 HINCRBY：

```
> hincrby user:1000 birthyear 10
(integer) 1987
> hincrby user:1000 birthyear 10
(integer) 1997
```

你可以从命令页找到全部哈希命令列表。

值得注意的是，小的哈希 (少量元素，不太大的值) 在内存中以一种特殊的方式编码以高效利用内存。

Redis 集合 (Sets)

Redis 集合是无序的字符串集合 (collections)。SADD 命令添加元素到集合。还可以对集合执行很多其他的操作，例如，测试元素是否存在，对多个集合执行交集、并集和差集，等等。

```
> sadd myset 1 2 3
(integer) 3
> smembers myset
1. 3
2. 1
3. 2
```

我们向集合总添加了 3 个元素，然后告诉 Redis 返回所有元素。如你所见，他们没有排序，Redis 在每次调用时按随意顺序返回元素，因为没有与用户有任何元素排序协议。

我们有测试成员关系的命令。一个指定的元素存在吗?

```
> sismember myset 3
(integer) 1
> sismember myset 30
(integer) 0
```

“3” 是集合中的成员，”30” 则不是。

集合适用于表达对象间关系。例如，我们可以很容易的实现标签。对这个问题的最简单建模，就是有一个为每个需要标记的对象的集合。集合中保存着与对象相关的标记的 ID。

假设，我们想标记新闻。如果我们的 ID 为 1000 的新闻，被标签 1，2，5 和 77 标记，我们可以有一个这篇新闻被关联标记 ID 的集合：

```
> sadd news:1000:tags 1 2 5 77
(integer) 4
```

然而有时候我们也想要一些反向的关系：被某个标签标记的所有文章：

```
> sadd tag:1:news 1000
(integer) 1
> sadd tag:2:news 1000
(integer) 1
> sadd tag:5:news 1000
(integer) 1
> sadd tag:77:news 1000
(integer) 1
```

获取指定对象的标签很简单：

```
> smembers news:1000:tags
1. 5
2. 1
3. 77
4. 2
```

注意：在这个例子中，我们假设你有另外一个数据结构，例如，一个 Redis 哈希，存储标签 ID 到标签名的映射。

还有一些使用正确的 Redis 命令就很容实现的操作。例如，我们想获取所有被标签 1，2，10 和 27 同时标记的对象列表。我们可以使用 SINTER 命令实现这个，也就是对不同的集合执行交集。我们只需要：

```
> sinter tag:1:news tag:2:news tag:10:news tag:27:news
... results here ...
```

并不仅仅是交集操作，你也可以执行并集，差集，随机抽取元素操作等等。

抽取一个元素的命令是 SPOP，就方便为很多问题建模。例如，为了实现一个基于 web 的扑克游戏，你可以将你的一副牌表示为集合。假设我们使用一个字符前缀表示(C)lubs 梅花，(D)iamonds 方块，(H)earts 红心，(S)pades 黑桃。

```
>  sadd deck C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 CJ CQ CK
   D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 DJ DQ DK H1 H2 H3
   H4 H5 H6 H7 H8 H9 H10 HJ HQ HK S1 S2 S3 S4 S5 S6
   S7 S8 S9 S10 SJ SQ SK
   (integer) 52
```

现在我们为每位选手提供 5 张牌。SPOP 命令删除一个随机元素，返回给客户端，是这个场景下的最佳操作。

然而，如果我们直接对这副牌调用，下一局我们需要再填充一副牌，这个可能不太理想。所以我们一开始要复制一下 deck 键的集合到 game:1:deck 键。

这是通过使用 SUNIONSTORE 命令完成的，这个命令通常对多个集合执行交集，然后把结果存储在另一个集合中。而对单个集合求交集就是其自身，于是我可以这样拷贝我的这副牌：

```
> sunionstore game:1:deck deck
(integer) 52
```

现在我们准备好为第一个选手提供 5 张牌：

```
> spop game:1:deck
"C6"
> spop game:1:deck
```

```
"CQ"
> spop game:1:deck
"D1"
> spop game:1:deck
"CJ"
> spop game:1:deck
"SJ"
```

只有一对 jack，不太理想……

现在是时候介绍提供集合中元素数量的命令。这个在集合理论中称为集合的基数(cardinality，也称集合的势)，所以相应的 Redis 命令称为 SCARD。

```
> scard game:1:deck
(integer) 47
```

数学计算式为：52 − 5 = 47。

当你只需要获得随机元素而不需要从集合中删除，SRANDMEMBER 命令则适合你完成任务。它具有返回重复的和非重复的元素的能力。

5

从入门到精通（下）

Redis 有序集合 (Sorted sets)

有序集合类似于集合和哈希的混合体的一种数据类型。像集合一样，有序集合由唯一的，不重复的字符串元素组成，在某种意义上，有序集合也就是集合。

集合中的每个元素是无序的，但有序集合中的每个元素都关联了一个浮点值，称为分数(score，这就是为什么该类型也类似于哈希，因为每一个元素都映射到一个值)。

此外，有序集合中的元素是按序存储的（不是请求时才排序的，顺序是依赖于表示有序集合的数据结构）。他们按照如下规则排序：

- 如果 A 和 B 是拥有不同分数的元素，A.score > B.score，则 A > B。
- 如果 A 和 B 是有相同的分数的元素，如果按字典顺序 A 大于 B，则 A > B。A 和 B 不能相同，因为排序集合只能有唯一元素。

让我们开始一个简单的例子，添加一些黑客的名字作为有序集合的元素，以他们的出生年份为分数。

```
> zadd hackers 1940 "Alan Kay"
(integer) 1
> zadd hackers 1957 "Sophie Wilson"
(integer 1)
> zadd hackers 1953 "Richard Stallman"
(integer) 1
> zadd hackers 1949 "Anita Borg"
(integer) 1
> zadd hackers 1965 "Yukihiro Matsumoto"
(integer) 1
> zadd hackers 1914 "Hedy Lamarr"
(integer) 1
> zadd hackers 1916 "Claude Shannon"
(integer) 1
> zadd hackers 1969 "Linus Torvalds"
(integer) 1
> zadd hackers 1912 "Alan Turing"
(integer) 1
```

如你所见，ZADD 命令类似于 SADD，但是多一个参数(位于添加的元素之前)，即分数。ZADD 命令也是可变参数的，所以你可以自由的指定多个分数值对(score-value pairs)，尽管上面的例子中并没有使用。

使用排序集合可以很容易返回按照出生年份排序的黑客列表，因为他们已经是排序好的。

实现注意事项：有序集合是通过双端(dual-ported)数据结构实现的，包括跳跃表(skiplist，后续文章会详细介绍，译者注)和哈希表(hashtable)，所以我们每次添加元素时 Redis 执行 O(log(N)) 的操作。这还好，但是当我们请求有序元素时，Redis 根本不需要做什么工作，因为已经是全部有序了：

```
> zrange hackers 0 -1
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
6) "Richard Stallman"
7) "Sophie Wilson"
8) "Yukihiro Matsumoto"
9) "Linus Torvalds"
```

注意：0 和 - 1 表示从索引为 0 的元素到最后一个元素(-1 像 LRANGE 命令中一样工作)。

如果我想按照相反的顺序排序，从最年轻到最年长？使用 ZREVRANGE 代替 ZRANGE：

```
> zrevrange hackers 0 -1
1) "Linus Torvalds"
2) "Yukihiro Matsumoto"
3) "Sophie Wilson"
4) "Richard Stallman"
5) "Anita Borg"
6) "Alan Kay"
7) "Claude Shannon"
8) "Hedy Lamarr"
9) "Alan Turing"
```

也可以同时返回分数，使用 WITHSCORES 参数：

```
> zrange hackers 0 -1 withscores
1) "Alan Turing"
2) "1912"
3) "Hedy Lamarr"
4) "1914"
5) "Claude Shannon"
6) "1916"
7) "Alan Kay"
8) "1940"
9) "Anita Borg"
10) "1949"
11) "Richard Stallman"
12) "1953"
```

```
13) "Sophie Wilson"
14) "1957"
15) "Yukihiro Matsumoto"
16) "1965"
17) "Linus Torvalds"
18) "1969"
```

范围操作 (ranges)

有序集合远比这些要强大。他们可以在范围上操作。让我们获取 1950 年前出生的所有人。我们使用 ZRANGEBYSCORE 命令来办到：

```
> zrangebyscore hackers -inf 1950
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
```

我们要求 Redis 返回分数在负无穷到 1950 之间的所有元素(包括两个极端)。

也可以删除某个范围的元素。让我们从有序集合中删除出生于 1940 年到 1960 年之间的黑客：

```
> zremrangebyscore hackers 1940 1960
(integer) 4
```

ZREMRANGEBYSCORE 也许不是最合适的命令名，但是非常有用，返回删除的元素数目。

另一个非常有用的操作是用来获取有序集合中元素排行的操作。也就是可以询问集合中元素的排序位置。

```
> zrank hackers "Anita Borg"
(integer) 4
```

ZREVRANK 命令用来按照降序排序返回元素的排行。

字典分数 (Lexicographical scores)

最近的 Redis2.8 版本引入了一个新的特性，假定集合中的元素都具有相同的分数，允许按字典顺序获取范围(元素按照 C 语言中的 memcmp 函数进行比较，因此可以保证没有整理，每个 Redis 实例会有相同的输出)。

操作字典顺序范围的主要命令是 ZRANGEBYLEX，ZREVRANGEBYLEX，ZREMRANGEBYLEX 和 ZLEXCOUNT。例如，我们再次添加我们的著名黑客清单。但是这次为每个元素使用 0 分数：

```
> zadd hackers 0 "Alan Kay" 0 "Sophie Wilson" 0 "Richard Stallman" 0
  "Anita Borg" 0 "Yukihiro Matsumoto" 0 "Hedy Lamarr" 0 "Claude Shannon"
  0 "Linus Torvalds" 0 "Alan Turing"
```

根据有序集合的排序规则，他们已经按照字典顺序排好了：

```
> zrange hackers 0 -1
1) "Alan Kay"
2) "Alan Turing"
3) "Anita Borg"
4) "Claude Shannon"
5) "Hedy Lamarr"
6) "Linus Torvalds"
7) "Richard Stallman"
8) "Sophie Wilson"
9) "Yukihiro Matsumoto"
```

使用 ZRANGEBYLEX 我们可以查询字典顺序范围：

```
> zrangebylex hackers [B [P
1) "Claude Shannon"
2) "Hedy Lamarr"
3) "Linus Torvalds"
```

范围可以是包容性的或者排除性的(取决于第一个字符，即开闭区间，译者注)，+ 和 - 分别表示正无穷和负无穷。查看该命令的文档获取更详细信息(该文档后续即奉献，译者注)。

这个特性非常重要，因为这允许有序集合作为通用索引。例如，如果你想用一个 128 位无符号整数来索引元素，你需要做的就是使用相同的分数(例如 0)添加元素到有序集合中，元素加上由 128 位大端(big endian)数字组成的 8 字节前缀。由于数字是大端编码，字典顺序排序(原始 raw 字节顺序)其实就是数字顺序，你可以在 128 位空间查询范围，获取元素后抛弃前缀。如果你想在一个更正式的例子中了解这个特性，可以看看 Redis 自动完成范例(后续献上，译者注)。

更新分数：排行榜 (leader boards)

这一部分是开始新的主题前最后一个关于有序集合的内容。有序集合的分数可以随时更新。对一个存在于有序集合中的元素再次调用 ZADD，将会在 O(log(N))时间复杂度更新他的分数 (和位置)，所以有序集合适合于经常更新的场合。

由于这个特性，通常的一个使用场景就是排行榜。最典型的应用就是 facebook 游戏，你可以组合使用按分数高低存储用户，以及获取排名的操作，来展示前 N 名的用户以及用户在排行榜上的排行(你是第 4932 名最佳分数)。

位图 (Bitmaps)

位图不是一个真实的数据类型，而是定义在字符串类型上的面向位的操作的集合。由于字符串类型是二进制安全的二进制大对象(blobs)，并且最大长度是 512MB，适合于设置 232 个不同的位。

位操作分为两组：常量时间单个位的操作，像设置一个位为 1 或者 0，或者获取该位的值。对一组位的操作，例如计算指定范围位的置位数量。

位图的最大优势是有时是一种非常显著的节省空间来存储信息的方式。例如，在一个系统中，不同用户由递增的用户 ID 来表示，可以使用 512MB 的内存来表示 400 万用户的单个位信息(例如他们是否需要接收信件)。

设置和检索位使用 SETBIT 和 GETBIT 命令：

```
> setbit key 10 1
(integer) 1
> getbit key 10
(integer) 1
> getbit key 11
(integer) 0
```

SETBIT 命令把第一个参数作为位数，第二个参数作为要给位设置的值，0 或者 。如果位的位置超过了当前字符串的长度，这个命令或自动扩充这个字符串。

GETBIT 命令只是返货指定下标处的位的值。超出范围的位(指定的位超出了该键下字符串的长度)被认为是 0。

有 3 个操作一组位的命令：

1. BITOP 命令对不同字符串执行逐位操。提供的操作包括与，或，异或和非。
2. BITCOUNT 命令执行计数操作，返回被设置为 1 的位的数量。
3. BITPOS 命令找到第一个值为指定值(0 或者 1)的位。

BITOPS 和 BITCOUNT 命令都可以操作字符串的字节范围，而不仅仅是运行于整个字符串长度。下面是 BITCOUNT 调用的一个简单例子：

```
> setbit key 0 1
(integer) 0
> setbit key 100 1
(integer) 0
> bitcount key
(integer) 2
```

位图的通用场景：

- 各种实时分析
- 需要高性能和高效率的空间利用来存储与对象 ID 关联的布尔信息。

例如，假设你想知道你的网站的用户的最长日访问曲线。你从 0 开始计算天数，也就是你的网站可访问的那天，并且每当用户访问你的网站的时候，就用 SETBIT 命令设置一个位。你可以使用当前 unix 时间减去初始位移，然后除以 3600*24，作为位的下标。

这种方式下每个用户都有一个记录每天访问信息的一个小字符串。使用 BITCOUNT 命令以及几次 BITPOS 调用，可以很容易获得指定用户访问网站的天数，或者只是获取并在客户端分析位图，也很容易计算出最长曲线。

位图可以很容易的拆分为多个键，例如为了数据集分片，因为通常要避免使用很大的键。为了将位图拆分为不同的键，而不是将所有的位设置到一个键，一个简单的策略就是每个键只存储 M 位，并且使用位数除以 M 作为键名，在键中使用位数模 M 来定位第 N 位。

超重对数 (HyperLogLogs)

超重对数是用于计算唯一事物数量的概率性数据结构(学术上指的是估算集合的基数)。通常计算唯一项数量需要使用和你想计算的项成正比的大量内存，因为你需要记住你已经看到的元素，以避免被多次计算。然而，有一组用内存换精度的算法：你会得到一个估算的测量，伴随一个标准错误，在 Redis 的实现中误差低于 1%，但是这些算法的魔力在于，你不再需要使用和你要计算的量成比例的大量内存，你只需要使用常量内存！最坏情况下 12K 字节，或者当你的超重对数(后续称它们为 HLL)只发现了少量元素时更是省内存。

Redis 中的超重对数，虽然技术上是一个不同的数据结构，但被编码为 Redis 字符串，所以你可以调用 GET 来序列化超重对数，使用 SET 反序列化回服务器。

从概念上讲，超重对数的 API 像是使用集合来做同样的事情。你会 SADD 元素到集合，使用 SCARD 来检查集合中元素数量，这些元素都是唯一的，因为 SCARD 捕获重复添加已经添加的元素。

你并没有真正添加项到超重对数中，因为这种数据结构只是包含了状态而没有包含真正的元素，其 API 也是一样：

- 每次你看到一个新元素，你就使用 PFADD 命令添加。
- 每次你想检索到目前为止当前近似的已添加进去的唯一元素数，你就使用 PFCOUNT 命令。

```
redis> PFADD hll a b c d
(integer) 1
redis> PFCOUNT hll
(integer) 4
```

这种数据结构的使用场景的一个例子是，计算每天搜索的唯一请求数。

Redis 也可以执行超重对数的并集，更多信息请继续关注相关命令(请关注此公众号，逐一揭晓)。

其他值得注意的特性 (notable features)

还有一些重要的 Redis API 没有在此文中探索，但是非常值得你关注：

你可以增量迭代键空间或者一个很大的集合。

你可以在服务端运行 Lua 脚本以赢得延迟和带宽。

Redis 还是也是一个订阅发布服务器。

了解更多 (Learn more)

这个教程并不完整，只涵盖了基本的 API。阅读命令参考去发现更多。

欢迎阅读此教程，和 Redis 一起狂舞！

6

使用 Redis 实现 Twitter（上）

本文讲述使用 PHP 以及 Redis 来设计和实现一个简单的微博。编程社区传统上认为，在开发 web 应用程序时，作为特殊目的的键值存储数据库不能用于替换关系型数据库。本文将向你展示 Redis 在键值层之上的数据结构是实现各种应用程序的有效数据模型。

在继续之前，你可以花点时间体验一下在线演示(http://retwis.redis.io，译者注)，看看我们究竟要做什么。长话短说：这是个练手，但是已经足够复杂到让你学习如何创建一个更复杂的程序的基础。

注意：这篇文章的原始版本写于 2009 年 Redis 发布时。当时还不清楚 Redis 的数据模型适合整个程序。5 年以后的今天，已经有许多应用程序使用 Redis 作为他们的主要存储，所以今天这篇文章的目的就是作为新学者的教程。你讲学习如何使用 Redis 设计一个简单的数据层，如何应用不同的数据结构。

我们的微博系统，叫做 Retwis，结构简单，具有很高的性能，只需少许努力能够分布于任意数量的 web 和 Redis 服务器。你可以在这里找到源代。

我使用 PHP 来做这个例子，是因为每个人都能看懂。使用 Ruby，Python，Erlang 等等语言也能得到同样(或更好)的结果。也有一些其他的实现(但是不是所有的实现都使用和当前版本教程同样的数据层，所以请使用 PHP 官方实现会更好)。

- Retwis-RB 是由 Daniel Lucraft 使用 Ruby 和 Sinatra 实现的版本！当然包含了全部的源代码，以及文章底部一个指向 Git 仓库的链接。本文剩下的部分定位为 PHP，但是 Ruby 程序员也可以查看 Redis-RB 的代码，因为他们从概念上非常相似。
- Retwis-J 是由 Costin Leau 使用 Spring Data Framework 和 Java 实现的版本。代码可以在 GitHub 上找到，springsource.org 上有更全面的文档介绍。

此处省略一万字。。。

(原文此处是对 Redis 数据类型的介绍，可以参考本系列文章的第 2 篇和第 3 篇，译者注)

前提条件(Prerequisites)

如果你还没有下载 Retwis 源码请先下载。包含一些 PHP 文件和 Predis 的一份拷贝(例子中我们使用的客户端库)。

另外你想要做的一件事是一个运行的 Redis 服务器。下载源码，使用 make 构建，使用./redis-server 运行，你就可以开始了。只是玩玩或者运行我们的 Retwis 的话，不需要配置。

数据设计(Layout)

当使用关系型数据库时，必须先设计数据库模式，这样我们先需要知道表，索引等数据库确定的东西。Redis 没有表，那我们需要设计什么呢？我们需要确定需要什么键来表示我们的对象，以及这些键需要存储什么值。

让我们从用户开始。我们需要用户名、用户 id，密码，用户粉丝(following)，关注列表等等来表示用户。第一个问题是，我们如果标识一个用户？像在关系型数据库，一个好的解决方案是用不同的号码来标识不同的用户，所以我们可以关联一个唯一 ID 给每个用户。对这个用户的引用通过其 ID。产生唯一 ID 非常简单，使用我们的原子 INCR 操作。当我们创建一个新用户我们就可以(假设用户名为 antirez)：

```
INCR next_user_id => 1000
HMSET user:1000 username antirez password p1pp0
```

注意：在真实程序中你应该使用哈希的密码，为了简化我们直接存储密码明文。

我们使用 next_user_id 键为每一位新用户提供唯一 ID。然后我们使用唯一 ID 来命名存储用户数据的哈希结构的键。记住，这是使用键值存储的通用设计模式！除了字段已经被定义了以外，我们还需要更多东西来完整定义一个用户。例如，有时通过用户名获得用户 ID，于是我们每次添加一个用户，我们也需要操作用户的键，使用用户名作为字段，用 ID 作为值的哈希。

```
HSET users antirez 1000
```

这一开始看起来有点奇怪，但是记住，我们只能采取直接访问数据的方式，而没有第二层索引。没法告诉 Redis 根据一个指定值返回其键。这也是我们的优势。强制我们使用按照主键来访问一切的新的范式来组织数据，此处主键是关系型数据库中的术语。

粉丝(followers)，关注(following)，和帖子(updates)

我们的系统还有一个核心需求。一个用户可能有很多关注他的用户，我们称他们为其粉丝。一个用户也可能会关注其他用户，我们称他们为其关注者。我们有一个为此量身打造的数据结构，就是集合。独一无二的集合元素，常量时间测试存在性，是两个非常有趣的特性。然而，记录一个用户开始关注另一个用户的时间怎么办？在我们加强版的微博系统里面。我们使用有序集合而不是一个简单的集合，用粉丝或者粉儿的用户 ID 作为元素，用用户关系创建时的 unix 时间作为分数。

让我们来定义我们的键：

```
followers:1000 => Sorted Set of uids of all the followers users
following:1000 => Sorted Set of uids of all the following users
```

我们添加一个粉丝：

```
ZADD followers:1000 1401267618 1234 => Add user 1234 with time 1401267618
```

另外一件重要的事情我们需要一个用户首页的位置来展示用户的更新。我们需要按照时间顺序来访问这些数据，从最近的到最老的，为此最好的数据结构就是列表。基本上每一个更新都会被 LPUSH 到用户更新键，多亏了 LRANGE，我们能实现分页等等。注意，我们可以互换地使用更新(updates)和帖子(posts)这两个词，因为某种意义上说，更新其实就是小型帖子。

```
posts:1000 => a List of post ids - every new post is LPUSHed here.
```

这个列表基本上就是用户的时间轴。我们会加入他自己帖子 ID，以及其关注者创建的帖子。基本上我们实现了一个写分列。

身份验证(Authentication)

好了，我们或多或少已经有了关于用户的一切，除了身份验证。我们会用一种简单而又健壮的方式处理身份验证：我们不想使用 PHP 的会话机制，我们的系统要为轻松地分布式部署于很多 web 服务器上而准备，所以我们会保存全部状态到 Redis 数据库中。所有我们要做的就是要设置一个猜不出来的字符串作为认证用户的 cookie，以及一个持有该字符串的客户端的用户 ID 的一个键。

我们需要两件事情来使得这个可以工作得健壮。第一，当前认证秘钥(不可猜测的字符串)是用户对象的一部分，所以当创建用户时，我们需要在哈希中设置一个认证字段：

```
HSET user:1000 auth fea5e81ac8ca77622bed1c2132a021f9
```

另外，我们需要映射认证秘钥到用户 ID，所以我们也需要一个认证键，使用哈希来映射秘钥和用户 ID。

```
HSET auths fea5e81ac8ca77622bed1c2132a021f9 1000
```

为了认证一个用户，我们只需要简单几步(请查看 Retwis 项目中的 login.php 源代码)：

- 从登陆表单获取用户名和密码。
- 检查用户名是否存在于 users 哈希中。
- 如果存在，我们获取其 ID(例如 1000)。
- 检查 user:1000 的密码是否匹配，否则返回错误消息。
- 认证完毕，设置 "fea5e81ac8ca77622bed1c2132a021f9"(user:1000 的 auth 字段)作为认证 cookie。

这是真实的代码：

```
include("retwis.php");

 # Form sanity checks

if (!gt("username") || !gt("password"))
    goback("You need to enter both username and password to login.");

 # The form is ok, check if the username is available

$username = gt("username");
$password = gt("password");
$r = redisLink();
$userid = $r->hget("users",$username);
if (!$userid)
```

```
    goback("Wrong username or password");
$realpassword = $r->hget("user:$userid","password");
if ($realpassword != $password)
    goback("Wrong useranme or password");

 # Username / password OK, set the cookie and redirect to index.php

$authsecret = $r->hget("user:$userid","auth");
setcookie("auth",$authsecret,time()+3600*24*365);
header("Location: index.php");
```

这些发生在每次用户登录时，但是我们还需要一个 isLoggedIn 函数来检查用户是否已经通过身份认证。以下是 isLoggedIn 函数的逻辑步骤：

- 从用户获取 auth cookie。如果没有 cookie 则用户没有登录。我们称这个 cookie 值为 `<authcookie>`。
- 检查 `<authcookie>` 是否存在于 auths 哈希字段中，以及其值(即用户 ID，本例中是 1000)。
- 为了系统更加健壮，验证 user:1000 的 auth 字段是否匹配。
- 用户验证完成，我们从$User 全局变量中加载一些信息。

代码也许比上面的描述更简单：

```
function isLoggedIn() {
    global $User, $_COOKIE;

    if (isset($User)) return true;

    if (isset($_COOKIE['auth'])) {
        $r = redisLink();
        $authcookie = $_COOKIE['auth'];
        if ($userid = $r->hget("auths",$authcookie)) {
            if ($r->hget("user:$userid","auth") != $authcookie) return false;
            loadUserInfo($userid);
            return true;
        }
    }
    return false;
}

function loadUserInfo($userid) {
    global $User;

    $r = redisLink();
    $User['id'] = $userid;
    $User['username'] = $r->hget("user:$userid","username");
```

```
    return true;
}
```

7

使用 Redis 实现 Twitter（下）

把 loadUserInfo 作为一个单独的函数有点大题小做了，但是在复杂的程序中这是一个很好的方法。认证中唯一被遗漏的事情就是登出了。我们怎么来做登出呢？很简单，我们改变 user:1000 的 auth 字段中的随机串，从 auths 哈希中删除旧的认证秘钥，然后添加一个新的。

重要：登出的步骤解释了为什么我们不是仅仅在 auths 哈希中查看认证秘钥以后认证用户，而是双重检查 user:1000 的 auth 字段。真正的认证字符串是后者，auths 哈希只不过是一个会挥发(volatile)的认证字段，或者，如果程序中 bug 或者脚本被中断，我们会发现 auths 键中有多个对应同一个用户 ID 的入口。登出代码如下(logout.php)：

```
include("retwis.php");

if (!isLoggedIn()) {
    header("Location: index.php");
    exit;
}

$r = redisLink();
$newauthsecret = getrand();
$userid = $User['id'];
$oldauthsecret = $r->hget("user:$userid","auth");

$r->hset("user:$userid","auth",$newauthsecret);
$r->hset("auths",$newauthsecret,$userid);
$r->hdel("auths",$oldauthsecret);

header("Location: index.php");
```

这就是我们所描述的，你需要去理解的。

帖子(Updates)

更新(updates)，也就是我们知道的帖子(posts)，更加简单。为了创建一个新的帖子我们这么干：

```
INCR next_post_id => 10343
HMSET post:10343 user_id $owner_id time $time body "I'm having fun with Retwis"
```

如你所见，每篇帖子由有 3 个字段的哈希组成。帖子拥有者的用户 ID，帖子撞见时间，最后是帖子的正文，真正的状态消息。

创建一个帖子后，我们获取其帖子 ID，LPUSH 其 ID 到帖子作者的每个粉丝用户的时间轴中，当然还有作者自己的帖子列表中(每个人事实上关注了他自己)。post.php 文件展示了这一切是怎么执行的：

```
include("retwis.php");

if (!isLoggedIn() || !gt("status")) {
    header("Location:index.php");
    exit;
}

$r = redisLink();
$postid = $r->incr("next_post_id");
$status = str_replace("\n"," ",gt("status"));
$r->hmset("post:$postid","user_id",$User['id'],"time",time(),"body",$status);
$followers = $r->zrange("followers:".$User['id'],0,-1);
$followers[] = $User['id']; /* Add the post to our own posts too */

foreach($followers as $fid) {
    $r->lpush("posts:$fid",$postid);
}
 # Push the post on the timeline, and trim the timeline to the

 # newest 1000 elements.

$r->lpush("timeline",$postid);
$r->ltrim("timeline",0,1000);

header("Location: index.php");
```

函数的核心是这个 foreach 循环。我们使用 ZRANGE 获取当前用户的所有粉丝，然后通过遍历 LPUSH 帖子到每一位粉丝的时间轴列表中。

注意，我们也为所有的帖子维护了一个全局的时间轴，这样我们就可以在 Retwis 首页轻易的展示每个人的帖子。这只需要执行 LPUSH 到时间轴列表。回到现实，我们难道没有开始觉得在 SQL 中使用 ORDER BY 来排序按照时间顺序添加的东西有一点点奇怪吗？至少我只这么认为的。

上面的代码有个有意思的地方值得注意：我们对全局时间轴执行完 LPUSH 操作之后使用了一个新命令 LTRIM。这是为了裁剪列表到 1000 个元素。全局时间轴事实上只会用在首页展示少量帖子，没有必要获取全部历史帖子。

基本上 LTRIM+LPUSH 是 Redis 中创建上限 (capped) 集合的一种方式。

帖子分页(Paginating)

我们如何使用 LRANGE 来获取一个范围的帖子，展现这些帖子在屏幕上，现在已经相当清楚了。代码很简单：

```
function showPost($id) {
    $r = redisLink();
    $post = $r->hgetall("post:$id");
    if (emptyempty($post)) return false;

    $userid = $post['user_id'];
    $username = $r->hget("user:$userid","username");
    $elapsed = strElapsed($post['time']);
    $userlink = "<a class=\"username\"href=\"profile.php?u=".urlencode($username)."\">".utf8entities($username)."</a>";

    echo('<div class="post">'.$userlink.' '.utf8entities($post['body'])."<br>");
    echo('<i>posted '.$elapsed.'ago via web</i></div>');
    return true;
}

function showUserPosts($userid,$start,$count) {
    $r = redisLink();
    $key = ($userid == -1) ? "timeline" : "posts:$userid";
    $posts = $r->lrange($key,$start,$start+$count);
    $c = 0;
    foreach($posts as $p) {
        if (showPost($p)) $c++;
        if ($c == $count) break;
    }
    return count($posts) == $count+1;
}
```

showPost 只是转换和打印一篇 HTML 帖子，showUserPosts 获取一个范围的帖子然后传递给 showPost。

注意：如果帖子列表很大的话，LRANGE 比较低效，我们想访问列表的中间元素，因为 Redis 列表的背后实现是链表。如果系统设计为为几百万的项分页，那最好求助于有序集合。

关注用户(Following users)

我们还没有讨论如何创建关注 / 粉丝关系，尽管这并不困难。如果 ID 为 1000 的用户(antirez) 想关注用户 ID 为 5000 的用户(pippo)，我们需要同时创建关注和被关注关系。我们只需要调用 ZADD：

```
ZADD following:1000 5000
ZADD followers:5000 1000
```

仔细关注一下同一个模式。理论上，在关系型数据库中，关注者列表和粉丝列表会在同一张表中，使用像 following_id 和 follower_id 这样的列。你可以使用 SQL 查询来抽取每个用户的关注者和粉丝。在键值数据库中则有一些不同，因为我们需要设置 1000 关注 5000，同时 5000 被 1000 关注的双重关系。这是要付出的代价，但是另一方面，访问数据很简单并相当的快。将这些作为独立的集合可以让我们做一些有意思的事情。例如，使用 ZINTERSTORE 我们可以获得两个不同用户的粉丝的交集，于是我们可以给我们的 Twitter 系统增加一个特性，当你访问某个人的主页时，可以很快的告诉你” 你和 Alice 有 34 个共同粉丝” 这样类似的事情。

你可以在 follow.php 中找到设置和删除关注/粉丝关系的代码。

水平伸缩(horizontally scalable)

亲爱的读者，如果你意识到了这一点你就已经是一个英雄了。谢谢你。在讨论水平伸缩之前有必要查看一下单台服务器的性能。Retwis 相当的快，没有任何的缓存。在一台很慢的过载的服务器上，apache 的 benchmark 使用 100 个并发客户端发出 10000 个请求，测量出平均 uv 为 5 毫秒。这意味着单台 Linux 服务器每天可以服务数以百万计的用户，这个像猴子屁股一样的慢，想象一下如果用更新的硬件会是什么结果。

然而，你不可能永远使用单台服务器，如何伸缩一个键值存储?

Retwis 不执行任何多键操作，所以伸缩很简单：你可以使用客户端分片，或者类似于 Twemproxy 的分片代理，或者是即将横空出世的 Redis 集群。

想更多的了解这个主题请阅读我们的分片文档。这里我们想强调的是，在键值存储系统中，如果你小心设计，数据集是可以拆分到相互独立的小的键上去。相比较使用语义上更复杂的数据库系统，分布这些键到多个节点更简单直接和可预见。

8

使用 Redis 作为 LRU 缓存

当 Redis 作为缓存使用时，当你添加新的数据时，有时候很方便使 Redis 自动回收老的数据。这种行为在开发者社区中众所周知，因为这是流行的 memcached 系统的默认行为。

LRU 实际上是被唯一支持的数据移除方法。本文内容将包含 Redis 的 maxmemory 指令，用于限制内存使用到一个固定的容量，也包含深入探讨 Redis 使用的 LRU 算法，一个近似准确的 LRU。

maxmemory 配置指令(configuration directive)

maxmemory 配置指令是用来配置 Redis 为数据集使用指定的内存容量大小。可以使用 redis.conf 文件来设置配置指令，或者之后在运行时使用 CONFIG SET 命令。

例如，为了配置内存限制为 100MB，可以在 redis.conf 文件中使用以下指令

```
maxmemory 100mb
```

设置 maxmemory 为 0，表示没有内存限制。这是 64 位系统的默认行为，32 位的系统则使用 3G 大小作为隐式的内存限制。

当指定的内存容量到达时，需要选择不同的行为，即策略。Redis 可以只为命令返回错误，这样将占用更多的内存，或者每次添加新数据时，回收掉一些旧的数据以避免内存限制。

回收策略(Eviction policies)

当 maxmemory 限制到达的时候，Redis 将采取的准确行为是由 maxmemory-policy 配置指令配置的。

以下策略可用：

- noeviction：当到达内存限制时返回错误。当客户端尝试执行命令时会导致更多内存占用(大多数写命令，除了 DEL 和一些例外)。
- allkeys-lru：回收最近最少使用(LRU)的键，为新数据腾出空间。
- volatile-lru：回收最近最少使用(LRU)的键，但是只回收有设置过期的键，为新数据腾出空间。
- allkeys-random：回收随机的键，为新数据腾出空间。
- volatile-random：回收随机的键，但是只回收有设置过期的键，为新数据腾出空间。
- volatile-ttl：回收有设置过期的键，尝试先回收离 TTL 最短时间的键，为新数据腾出空间。

当没有满足前提条件的话，volatile-lru，volatile-random 和 volatile-ttl 策略就表现得和 noeviction 一样了。

选择正确的回收策略是很重要的，取决于你的应用程序的访问模式，但是，你可以在程序运行时重新配置策略，使用 INFO 输出来监控缓存命中和错过的次数，以调优你的设置。

一般经验规则：

- M 如果你期待你的用户请求呈现幂律分布(power-law distribution)，也就是，你期待一部分子集元素被访问得远比其他元素多，可以使用 allkeys-lru 策略。在你不确定时这是一个好的选择。
- 如果你是循环周期的访问，所有的键被连续扫描，或者你期待请求正常分布(每个元素以相同的概率被访问)，可以使用 allkeys-random 策略。
- 如果你想能给 Redis 提供建议，通过使用你创建缓存对象的时候设置的 TTL 值，确定哪些对象应该被过期，你可以使用 volatile-ttl 策略。

当你想使用单个实例来实现缓存和持久化一些键，allkeys-lru 和 volatile-random 策略会很有用。但是，通常最好是运行两个 Redis 实例来解决这个问题。

另外值得注意的是，为键设置过期时间需要消耗内存，所以使用像 allkeys-lru 这样的策略会更高效，因为在内存压力下没有必要为键的回收设置过期时间。

回收过程 (Eviction process)

理解回收的过程是这么运作的非常的重要：

- 一个客户端运行一个新命令，添加了新数据。
- Redis 检查内存使用情况，如果大于 maxmemory 限制，根据策略来回收键。
- 一个新的命令被执行，如此等等。

我们通过检查，然后回收键以返回到限制以下，来连续不断的穿越内存限制的边界。

如果一个命令导致大量的内存被占用 (像一个很大的集合交集保存到一个新的键)，一会功夫内存限制就会被这个明显的内存量所超越。

近似的 LRU 算法(Approximated LRU algorithm)

Redis 的 LRU 算法不是一个精确的实现。这意味着 Redis 不能选择最佳候选键来回收，也就是最久钱被访问的那些键。相反，会尝试运营一个近似的 LRU 算法，通过采样一小部分键，然后在采样键中回收最适合(拥有最久访问时间)的那个。

然而，从 Redis3.0(当前还是 beta 版本)开始，算法被改进为持有回收候选键的一个池子。这改善了算法的性能，使得更接近于真实的 LRU 算法的行为

Redis 的 LRU 算法有一点很重要，你可以调整算法的精度，通过改变每次回收时检查的采样数量。这个参数可以通过如下配置指令:

```
maxmemory-samples 5
```

Redis 没有使用真实的 LRU 实现的原因，是因为这会消耗更多的内存。然而，近似值对使用 Redis 的应用来说基本上也是等价的。下面的图形对比，为 Redis 使用的 LRU 近似值和真实 LRU 之间的比较。

用于测试生成上面图像的 Redis 服务被填充了指定数量的键。键被从头访问到尾，所以第一个键是 LRU 算法的最佳候选回收键。然后，再新添加 50% 的键，强制一般的旧键被回收。

你可以从图中看到三种不同的原点，形成三个不同的带。

- 浅灰色带是被回收的对象
- 灰色带是没有被回收的对象
- 绿色带是被添加的对象

在理论的 LRU 实现中，我们期待看到的是，在旧键中第一半会过期。而 Redis 的 LRU 算法则只是概率性的过期这些旧键。

你可以看到，同样采用 5 个采样，Redis 3.0 表现得比 Redis 2.8 要好，Redis 2.8 中最近被访问的对象之间的对象仍然被保留。在 Redis 3.0 中使用 10 为采样大小，近似值已经非常接近理论性能。

注意，LRU 只是一个预言指定键在未来如何被访问的模式。另外，如果你的数据访问模式非常接近幂律，大多数的访问都将集中在一个集合中，LRU 近似算法将能处理得很好。

在模拟实验的过程中，我们发现使用幂律访问模式，真实的 LRU 算法和 Redis 的近似算法之间的差异非常小，或者根本就没有。

然而，你可以提高采样大小到 10，这会消耗额外的 CPU，来更加近似于真实的 LRU 算法，看看这会不会使你的缓存错失率有差异。

使用 CONFIG SET maxmemory-samples 命令在生产环境上试验各种不同的采样大小值是很简单的。

9

分片

分片(partitioning)就是将你的数据拆分到多个 Redis 实例的过程，这样每个实例将只包含所有键的子集。本文第一部分将向你介绍分片的概念，第二部分将向你展示 Redis 分片的可选方案。

分片为何有用(Why useful)

Redis 的分片承担着两个主要目标：

- 允许使用很多电脑的内存总和来支持更大的数据库。没有分片，你就被局限于单机能支持的内存容量。
- 允许伸缩计算能力到多核或多服务器，伸缩网络带宽到多服务器或多网络适配器。

分片基础(Basics)

有很多不同的分片标准(criteria)。假想我们有 4 个 Redis 实例 R0，R1，R2，R3，还有很多表示用户的键，像 user:1，user:2，… 等等，我们能找到不同的方式来选择一个指定的键存储在哪个实例中。换句话说，有许多不同的办法来映射一个键到一个指定的 Redis 服务器。

最简单的执行分片的方式之一是范围分片(range partitioning)，通过映射对象的范围到指定的 Redis 实例来完成分片。例如，我可以假设用户从 ID 0 到 ID 10000 进入实例 R0，用户从 ID 10001 到 ID 20000 进入实例 R1，等等。

这套办法行得通，并且事实上在实践中被采用，然而，这有一个缺点，就是需要一个映射范围到实例的表格。这张表需要管理，不同类型的对象都需要一个表，所以范围分片在 Redis 中常常并不可取，因为这要比替他分片可选方案低效得多。

一种范围分片的替代方案是哈希分片(hash partitioning)。这种模式适用于任何键，不需要键像 object_name: 这样的饿形式，就像这样简单：

- 使用一个哈希函数(例如，crc32 哈希函数) 将键名转换为一个数字。例如，如果键是 foobar，crc32(foobar)将会输出类似于 93024922 的东西。
- 对这个数据进行取模运算，以将其转换为一个 0 到 3 之间的数字，这样这个数字就可以映射到我的 4 台 Redis 实例之一。93024922 模 4 等于 2，所以我知道我的键 foobar 应当存储到 R2 实例。注意：取模操作返回除法操作的余数，在许多编程语言总实现为%操作符。

有许多其他的方式可以分片，从这两个例子中你就可以知道了。一种哈希分片的高级形式称为一致性哈希(consistent hashing)，被一些 Redis 客户端和代理实现。

分片的不同实现(Different implementations)

分片可由软件栈中的不同部分来承担。

- 客户端分片(Client side partitioning)意味着，客户端直接选择正确的节点来写入和读取指定键。许多 Redis 客户端实现了客户端分片。
- 代理协助分片(Proxy assisted partitioning)意味着，我们的客户端发送请求到一个可以理解 Redis 协议的代理上，而不是直接发送请求到 Redis 实例上。代理会根据配置好的分片模式，来保证转发我们的请求到正确的 Redis 实例，并返回响应给客户端。Redis 和 Memcached 的代理 Twemproxy 实现了代理协助的分片。
- 查询路由(Query routing)意味着，你可以发送你的查询到一个随机实例，这个实例会保证转发你的查询到正确的节点。Redis 集群在客户端的帮助下，实现了查询路由的一种混合形式 (请求不是直接从 Redis 实例转发到另一个，而是客户端收到重定向到正确的节点)。

分片的缺点(Disadvantages)

Redis 的一些特性与分片在一起时玩转的不是很好：

- 涉及多个键的操作通常不支持。例如，你不能对映射在两个不同 Redis 实例上的键执行交集(事实上有办法做到，但不是直接这么干)。
- 涉及多个键的事务不能使用。
- 分片的粒度(granularity)是键，所以不能使用一个很大的键来分片数据集，例如一个很大的有序集合。
- 当使用了分片，数据处理变得更复杂，例如，你需要处理多个 RDB/AOF 文件，备份数据时你需要聚合多个实例和主机的持久化文件。
- 添加和删除容量也很复杂。例如，Redis 集群具有运行时动态添加和删除节点的能力来支持透明地再均衡数据，但是其他方式，像客户端分片和代理都不支持这个特性。但是，有一种称为预分片(Presharding)的技术在这一点上能帮上忙。

数据存储还是缓存(Store or cache)

尽管无论是将 Redis 作为数据存储还是缓存，Redis 的分片概念上都是一样的，但是作为数据存储时有一个重要的局限。当 Redis 作为数据存储时，一个给定的键总是映射到相同的 Redis 实例。当 Redis 作为缓存时，如果一个节点不可用而使用另一个节点，这并不是一个什么大问题，按照我们的愿望来改变键和实例的映射来改进系统的可用性(就是系统回复我们查询的能力)。

一致性哈希实现常常能够在指定键的首选节点不可用时切换到其他节点。类似的，如果你添加一个新节点，部分数据就会开始被存储到这个新节点上。

这里的主要概念如下:

- 如果 Redis 用作缓存，使用一致性哈希来来实现伸缩扩展(scaling up and down)是很容易的。
- 如果 Redis 用作存储，使用固定的键到节点的映射，所以节点的数量必须固定不能改变。否则，当增删节点时，就需要一个支持再平衡节点间键的系统，当前只有 Redis 集群可以做到这一点，但是 Redis 集群现在还处在 beta 阶段，尚未考虑再生产环境中使用。

预分片(Presharding)

我们已经知道分片存在的一个问题，除非我们使用 Redis 作为缓存，增加和删除节点是一件很棘手的事情，使用固定的键和实例映射要简单得多。

然而，数据存储的需求可能一直在变化。今天我可以接受 10 个 Redis 节点(实例)，但是明天我可能就需要 50 个节点。

因为 Redis 只有相当少的内存占用(footprint)而且轻量级(一个空闲的实例只是用 1MB 内存)，一个简单的解决办法是一开始就开启很多的实例。即使你一开始只有一台服务器，你也可以在第一天就决定生活在分布式的世界里，使用分片来运行多个 Redis 实例在一台服务器上。

你一开始就可以选择很多数量的实例。例如，32 或者 64 个实例能满足大多数的用户，并且为未来的增长提供足够的空间。

这样，当你的数据存储需要增长，你需要更多的 Redis 服务器，你要做的就是简单地将实例从一台服务器移动到另外一台。当你新添加了第一台服务器，你就需要把一半的 Redis 实例从第一台服务器搬到第二台，如此等等。

使用 Redis 复制，你就可以在很小或者根本不需要停机时间内完成移动数据：

- 在你的新服务器上启动一个空实例。
- 移动数据，配置新实例为源实例的从服务。
- 停止你的客户端。
- 更新被移动实例的服务器 IP 地址配置。
- 向新服务器上的从节点发送 SLAVEOF NO ONE 命令。
- 以新的更新配置启动你的客户端。
- 最后关闭掉旧服务器上不再使用的实例。

Redis 分片的实现(Implementations)

到目前为止，我们从理论上讨论了 Redis 分片，但是实践情况如何呢？你应该使用什么系统呢？

Redis 集群(Redis Cluster)

Redis 集群是自动分片和高可用的首选方式。当前还不能完全用于生产环境，但是已经进入了 beta 阶段，所以我们推荐你开始小试牛刀。你可以从集群教程(请持续关注本公众账号后续文章，译者注)中获取更多 Redis 集群的相关信息。

一旦 Redis 集群可用，以及支持 Redis 集群的客户端可用，Redis 集群将会成为 Redis 分片的事实标准。

Redis 集群是查询路由和客户端分片的混合模式。

Twemproxy

Twemproxy 是 Twitter 开发的一个支持 Memcached ASCII 和 Redis 协议的代理。它是单线程的，由 C 语言编写，运行非常的快。他是基于 Apache 2.0 许可的开源项目。

Twemproxy 支持自动在多个 Redis 实例间分片，如果节点不可用时，还有可选的节点排除支持(这会改变键和实例的映射，所以你应该只在将 Redis 作为缓存是才使用这个特性)。

这并不是单点故障(single point of failure)，因为你可以启动多个代理，并且让你的客户端连接到第一个接受连接的代理。

从根本上说，Twemproxy 是介于客户端和 Redis 实例之间的中间层，这就可以在最下的额外复杂性下可靠地处理我们的分片。这是当前我们建议的处理 Redis 分片的方式。你可以阅读更多关于 Twemproxy 的信息(作者的这篇博客文章 http://antirez.com/news/44，译者注)。

支持一致性哈希的客户端

Twemproxy 之外的可选方案，是使用实现了客户端分片的客户端，通过一致性哈希或者别的类似算法。有多个支持一致性哈希的 Redis 客户端，例如 Redis-rb 和 Predis。

请查看完整的 Redis 客户端列表，看看是不是有支持你的编程语言的，并实现了一致性哈希的成熟客户端。

10

复制

Redis 的复制 (replication) 是一种使用和配置起来非常简单的主从(master-slave)复制，允许 Redis 从服务器成为主服务器的精确副本。以下是关于 Redis 复制的一些重要方面：

- Redis 采用异步复制。从 Redis 2.8 开始，从服务器会周期性地报告从复制流中处理的数据量。 一个主服务器可以拥有多个从服务器。
- 从服务器可以接受其他从服务器的连接。除了连接多个从服务器到同一个主服务器，从服务器也可以连接到其他的从服务器，形成图状结构。
- Redis 的复制在主服务器上是非阻塞的。这意味着，当一个或多个从服务器执行初始化同步(initial synchronization)时，主服务器能继续处理请求。
- Redis 的复制在从服务器上也是非阻塞的。当从服务器正在执行初始化同步时，假如你在
- redis.conf 中进行了相应配置，也能够继续使用旧版本的数据集处理请求。另外，你还可以配置当复制流宕(dowm)掉的时候，从服务器返回给客户端一个错误。然而，初始化同步结束后，旧的数据集需要被删除，新的数据集需要被载入。在这个简短的窗口期内，从服务器会阻塞到来的连接。
- 复制可以用来支持可伸缩性，用多个从服务器处理只读查询(例如，繁重的 SORT 操作可以分配到从服务器上)，也可以仅仅作为数据冗余。
- 可以使用复制来避免主服务器将全部数据集写到磁盘的开销：只需要配置你的主服务器的 redis.conf 来防止保存(所有的” 保存” 指令)，然后连接一个不断复制的从服务器。但是，这种设置下要确保主服务器不会自动重启(阅读下一节获取更多信息)。

主服务器关闭持久化时的安全性(Safety of replication)

当使用了 Redis 的复制时，强烈建议在主服务器上开启持久化，或者，当不可能开启持久化时，例如由于关注延迟，实例应该被配置为避免自动重启。

为了更好的理解为什么关闭了持久化的主服务器被配置为自动重启是很危险的，查看下面的失败模型，数据从主服务器以及其所有从服务器上被清除：

- 我们设置节点 A 作为主服务器，关闭了持久化，节点 B 和节点 C 从节点 A 复制。
- A 崩溃了，但是它拥有某个自动重启系统，重启了这个进程。但是，由于持久化是被关闭的，这个节点以空的数据集重启。
- 节点 B 和节点 C 从空的 A 复制，于是它们完全销毁了他们的数据拷贝。

当 Redis Sentinel 被用于高可用时，主服务器关闭了持久化，并开启了进程重启也是很危险的。例如，主务器非常快速的重启，以至于 Sentinel 没有检测到失败，于是上面描述的失败模型就发生了。

任何时刻数据安全都是很重要的，要禁止主服务器配置为关闭持久化并自动重启。

Redis 复制如何工作(How works)

当你建立一个从服务器，连接时就会发送一个 SYNC 命令。不管是第一次连接上还是重连接上。

然后主服务器开始在后台保存，并且开始缓冲所有新收到的会修改数据集的命令。当后台保存完成以后，主服务器传输数据库文件给从服务器，从服务器将其保存到磁盘上，然后加载到内存中。然后主服务器开始发送缓冲的命令给从服务器。这是通过命令流完成的，和 Redis 的协议是一样的格式。

你可以用 telnet 试试。连上一台正在工作的 Redis 的端口，然后发送 SYNC 命令。你会看到大量的传输，还有主服务器收到的每条命令被重新发送给了 telnet 会话。

当主从链路由于某些原因断开时，从服务器可以自动重连。如果主服务器收到多个并发的从服务器的同步请求，只会执行一个后台保存来服务所有从服务器。

当主服务器和从服务器断开后重连上，总是执行一次完整重同步(full resynchronization)。然而，从 Redis 2.8 以后，可以选择执行部分重同步(partial resynchronization)。

部分重同步(partial resynchronization)

从 Redis 2.8 开始，在复制链接断开后，主服务器和从服务器通常可以继续复制过程，而不需要一次完整的重同步。

这是通过在主服务器上创建一个复制流的内存缓冲区(in-memory backlog)实现的。主服务器和所有从服务器都记录一个复制偏移量(offset)和一个主服务器运行 ID(run id)，当链接断掉时，从服务器会重连接，并且请求主服务器继续复制。假设主服务器的运行 ID 还是一样的，并且指定的偏移量在复制缓冲区中可用，复制会从中断的点继续。如果这两个条件之一不满足，将会执行完整重同步(2.8 版之前的正常行为)。

新的部分重同步特性使用的是内部 PSYNC 命令，老的实现采用的是 SYNC 命令。注意，Redis 2.8 的从服务器可以检测主服务器是否不支持 PSYNC，然后使用 SYNC 代替。

无盘复制(Diskless replication)

通常，一次完整的重同步需要在磁盘上创建一个 RDB 文件，然后从磁盘重新加载同一个 RDB 来服务从服务器。

由于低速的磁盘，这对主服务器来说是很大压力的操作。Redis 2.8.18 版本是第一个对无盘复制提供试验性支持的版本。在这种设置下，子进程直接通过线路(wire)发送 RDB 文件给从服务器，而不需要使用磁盘作为中间存储。

配置(Configuration)

配置复制简直小菜一碟：只需要添加下面一行到从服务器配置文件：

```
slaveof 192.168.1.1 6379
```

当然，你得把 192.168.1.1 6379 替换成你自己的主服务器 IP 地址(或主机名)和端口。或者，你可以调用 SLAVEOF 命令和主服务器主机，开始与从服务器的一次同步。

有很多参数可以用来调整执行部分重同步主服务器的上的内存复制缓冲区。可以看看 Redis 发布版本中自带的样例文件 redis.conf 以获取更多的信息。

只读从服务器(Read-only slave)

从 Redis 2.6 开始，从服务器支持默认开启的只读模式。这个行为由 redis.conf 文件中的 slave-read-only 选项控制，可以在运行时使用 CONFIG SET 来开启和关闭。

只读从服务器会拒绝所有写命令，所以写入数据到从服务器只会引起错误。这并不意味着，这个特性打算暴露从服务器实例到互联网，或者到网络中不信任的客户端，因为诸如 DEBUG 和 CONFIG 这样的管理命令等仍可用。但是，可以通过在 redis.conf 中使用 rename-command 指令来禁止命令，从而改进只读实例的安全性。

你可能很好奇，为什么需要能够反转只读设置，使得从服务器实例能够成为写操作的目标。尽管这些写入的数据会在从服务器和主服务器重同步时，或者从服务器重启时被丢弃，还是有一些存储一些短暂的数据到可写的从服务器的合理场景。例如，客户端可以存储一些主服务器的可达性信息来调整故障转移(failover)策略。

认证主服务器(Authenticate to a master)

如果你的主服务器通过 requirepass 而有一个密码，很容易配置从服务器在所有同步操作中使用这个密码。

要做到这个，在一个运行的实例上，使用 redis-cli 并键入：

```
config set masterauth <password>
```

要永久设置这个，添加这个倒你的配置文件中：

```
masterauth <password>
```

N 个副本才能写(Allow writes only with N attached replicas)

从 Redis 2.8 开始，可以设置 Redis 主服务器在当前至少拥有 N 个从服务器的连接的情况下，才能接受写请求。

然而，由于 Redis 使用异步复制，不能保证从服务器真正收到了一个给定的写请求，于是总是有一个数据丢失的窗口期。

下面是这个特性是如何运作的：

- Redis 从服务器每秒种 ping 主服务器，上报处理完的复制流的数据量。
- Redis 主服务器记录上一次从每一个从服务收到 ping 的时间。
- 用户可以配置最小从服务器数量，每台从服务器拥有一个不大于最大秒数的滞后(lag)。

如果有至少 N 个小于 M 秒滞后的从服务器，写请求才会被接受。

你可能会认为这个像 CAP 理论中较宽松版本的”C”，不能保证指定写的一致性，但是至少数据丢失的时间窗口被限制在一个指定的秒数内。

如果条件不满足，主服务器会返回一个错误，并且不会接受写请求。

这个特性有两个配置参数：

```
min-slaves-to-write  <number of slaves>
min-slaves-max-lag  <number of seconds>
```

请查看随 Redis 源码发布版本自带的 redis.conf 文件获取更多信息。

11
持久化

本文提供对 Redis 持久化(persistence)的技术性描述，适合所有的 Redis 用户来阅读。想获得对 Redis 持久化和持久性保证有更全面的了解，也可以读一下作者的博客文章(地址为 http://antirez.com/post/redis-persistence-demystified.html，译者注)。

Redis 持久化(Persistence)

Redis 提供了不同持久化范围的选项：

- RDB 持久化以指定的时间间隔执行数据集的即时点(point-in-time)快照。
- AOF 持久化在服务端记录每次收到的写操作，在服务器启动时会重放，以重建原始数据集。命令使用和 Redis 协议一样的格式以追加的方式来记录。当文件太大时 Redis 会在后台重写日志。
- 如果你愿意，你可以完全禁止持久化，如果你只是希望你的数据在服务器运行期间才存在的话。
- 可以在同一个实例上同时支持 AOF 和 RDB。注意，在这种情况下，当 Redis 重启时，AOF 文件会被用于重建原始数据集，因为它被保证是最完整的数据。

理解 RDB 和 AOF 持久化之间的各自优劣 (trade-offs) 是一件非常重要的事情。让我们先从 RDB 开始：

RDB 优点(RDB advantages)

- RDB 是一种表示某个即时点的 Redis 数据的紧凑文件。RDB 文件适合用于备份。例如，你可能想要每小时归档最近 24 小时的 RDB 文件，每天保存近 30 天的 RDB 快照。这允许你很容易的恢复不同版本的数据集以容灾。
- RDB 非常适合于灾难恢复，作为一个紧凑的单一文件，可以被传输到远程的数据中心，或者是 Amazon S3(可能得加密)。
- RDB 最大化了 Redis 的性能，因为 Redis 父进程持久化时唯一需要做的是启动(fork)一个子进程，由子进程完成所有剩余工作。父进程实例不需要执行像磁盘 IO 这样的操作。
- RDB 在重启保存了大数据集的实例时比 AOF 要快。

RDB 缺点(RDB disadvantages)

当你需要在 Redis 停止工作(例如停电)时最小化数据丢失，RDB 可能不太好。你可以配置不同的保存点(save point)来保存 RDB 文件(例如，至少 5 分钟和对数据集 100 次写之后，但是你可以有多个保存点)。然而，你通常每隔 5 分钟或更久创建一个 RDB 快照，所以一旦 Redis 因为任何原因没有正确关闭而停止工作，你就得做好最近几分钟数据丢失的准备了。

RDB 需要经常调用 fork()子进程来持久化到磁盘。如果数据集很大的话，fork()比较耗时，结果就是，当数据集非常大并且 CPU 性能不够强大的话，Redis 会停止服务客户端几毫秒甚至一秒。AOF 也需要 fork()，但是你可以调整多久频率重写日志而不会有损(trade-off)持久性(durability)。

AOF 优点(AOF advantages)

- 使用 AOF Redis 会更具有可持久性(durable)：你可以有很多不同的 fsync 策略：没有 fsync，每秒 fsync，每次请求时 fsync。使用默认的每秒 fsync 策略，写性能也仍然很不错(fsync 是由后台线程完成的，主线程继续努力地执行写请求)，即便你也就仅仅只损失一秒钟的写数据。
- AOF 日志是一个追加文件，所以不需要定位，在断电时也没有损坏问题。即使由于某种原因文件末尾是一个写到一半的命令(磁盘满或者其他原因),redis-check-aof 工具也可以很轻易的修复。
- 当 AOF 文件变得很大时，Redis 会自动在后台进行重写。重写是绝对安全的，因为 Redis 继续往旧的文件中追加，使用创建当前数据集所需的最小操作集合来创建一个全新的文件，一旦第二个文件创建完毕，Redis 就会切换这两个文件，并开始往新文件追加。
- AOF 文件里面包含一个接一个的操作，以易于理解和解析的格式存储。你也可以轻易的导出一个 AOF 文件。例如，即使你不小心错误地使用 FLUSHALL 命令清空一切，如果此时并没有执行重写，你仍然可以保存你的数据集，你只要停止服务器，删除最后一条命令，然后重启 Redis 就可以。

AOF 缺点(AOF disadvantages)

- 对同样的数据集，AOF 文件通常要大于等价的 RDB 文件。
- AOF 可能比 RDB 慢，这取决于准确的 fsync 策略。通常 fsync 设置为每秒一次的话性能仍然很高，如果关闭 fsync，即使在很高的负载下也和 RDB 一样的快。不过，即使在很大的写负载情况下，RDB 还是能提供能好的最大延迟保证。
- 在过去，我们经历了一些针对特殊命令(例如，像 BRPOPLPUSH 这样的阻塞命令)的罕见 bug，导致在数据加载时无法恢复到保存时的样子。这些 bug 很罕见，我们也在测试套件中进行了测试，自动随机创造复杂的数据集，然后加载它们以检查一切是否正常，但是，这类 bug 几乎不可能出现在 RDB 持久化中。为了说得更清楚一点：Redis AOF 是通过递增地更新一个已经存在的状态，像 MySQL 或者 MongoDB 一样，而 RDB 快照是一次又一次地从头开始创造一切，概念上更健壮。但是，1)要注意 Redis 每次重写 AOF 时都是以当前数据集中的真实数据从头开始，相对于一直追加的 AOF 文件(或者一次重写读取老的 AOF 文件而不是读内存中的数据)对 bug 的免疫力更强。2)我们还没有收到一份用户在真实世界中检测到崩溃的报告。

我们该选谁(what)

通常来说，你应该同时使用这两种持久化方法，以达到和 PostgreSQL 提供的一样的数据安全程度。

如果你很关注你的数据，但是仍然可以接受灾难时有几分钟的数据丢失，你可以只单独使用 RDB。

有很多用户单独使用 AOF，但是我们并不鼓励这样，因为时常进行 RDB 快照非常方便于数据库备份，启动速度也较之快，还避免了 AOF 引擎的 bug。

注意：基于这些原因，将来我们可能会统一 AOF 和 RDB 为一种单一的持久化模型(长远计划)。

下面的部分将介绍两种持久化模型等多的细节。

快照(Snapshotting)

默认情况下，Redis 保存数据集快照到磁盘，名为 dump.rdb 的二进制文件。你可以设置让 Redis 在 N 秒内至少有 M 次数据集改动时保存数据集，或者你也可以手动调用 SAVE 或者 BGSAVE 命令。

例如，这个配置会让 Redis 在每个 60 秒内至少有 1000 次键改动时自动转储数据集到磁盘：

```
save 60 1000
```

这种策略被称为快照。

如何工作(How works)

每当 Redis 需要转储数据集到磁盘时，会发生：

- Redis 调用 fork()。于是我们有了父子两个进程。
- 子进程开始将数据集写入一个临时 RDB 文件。
- 当子进程完成了新 RDB 文件，替换掉旧文件。

这个方法可以让 Redis 获益于写时复制(copy-on-write)机制。

只追加文件(Append-only file)

快照并不是非常具有可持久性(durable)。如果你运行 Redis 的电脑停机了，电源线断了，或者你不小心 kill -9 掉你的实例，最近写入 Redis 的数据将会丢失。尽管这个对一些应用程序来说不是什么大事，但是也有一些需要完全可持久性(durability)的场景，在这些场景下可能就不合适了。

只追加文件是一个替代方案，是 Redis 的完全可持久性策略。在 1.1 版本中就可用了。

你可以在你的配置文件中开启 AOF：

```
appendonly yes
```

从现在开始，每次 Redis 收到修改数据集的命令，将会被追加到 AOF 中。当你重启 Redis 的时候，就会重放(re-play)AOF 文件来重建状态。

日志重写(Log rewriting)

你可以猜得到，写操作不断执行的时候 AOF 文件会越来越大。例如，如果你增加一个计数器 100 次，你的数据集里只会有一个键存储这最终值，但是却有 100 条记录在 AOF 中。其中 99 条记录在重建当前状态时是不需要的。

于是 Redis 支持一个有趣的特性：在后台重建 AOF 而不影响服务客户端。每当你发送 BGREWRITEAOF 时，Redis 将会写入一个新的 AOF 文件，包含重建当前内存中数据集所需的最短命令序列。如果你使用的是 Redis 2.2 的 AOF，你需要不时的运行 BGREWRITEAOF 命令。Redis 2.4 可以自动触发日志重写(查看 Redis 2.4 中的示例配置文件以获得更多信息)。

AOF 持久性如何(How durable)

你可以配置多久 Redis 会 fsync 数据到磁盘一次。有三个选项：

- 每次一个新命令追加到 AOF 文件中时执行 fsync。非常非常慢，但是非常安全。
- 每秒执行 fsync。够快(2.4 版本中差不多和快照一样快)，但是当灾难来临时会丢失 1 秒的数据。
- 从不执行 fsync，直接将你的数据交到操作系统手里。更快，但是更不安全。

建议的(也是默认的)策略是每秒执行一次 fsync。既快，也相当安全。一直执行的策略在实践中非常慢(尽管在 Redis 2.0 中有所改进)，因为没法让 fsync 这个操作本身更快。

AOF 损坏了怎么办(corrupted)

有可能在写 AOF 文件时服务器崩溃(crash)，文件损坏后 Redis 就无法装载了。如果这个发生的话，你可以使用下面的步骤来解决这个问题：

- 创建 AOF 的一个拷贝用于备份。
- 使用 Redis 自带的 redis-check-aof 工具来修复原文件：
- $ redis-check-aof --fix
- 使用 diff -u 来检查两个文件有什么不同。用修复好的文件来重启服务器。

如何工作(How works)

日志重写采用了和快照一样的写时复制机制。下面是过程:

- Redis 调用 fork()。于是我们有了父子两个进程。
- 子进程开始向一个临时文件中写 AOF。
- 父进程在一个内存缓冲区中积累新的变更(同时将新的变更写入旧的 AOF 文件，所以即使重写失败我们也安全)。
- 当子进程完成重写文件，父进程收到一个信号，追加内存缓冲区到子进程创建的文件末尾。
- 搞定！现在 Redis 自动重命名旧文件为新的，然后开始追加新数据到新文件。

如何从 RDB 切换到 AOF(How switch)

在 Redis 2.2 及以上版本中非常简单，也不需要重启。

- 备份你最新的 dump.rdb 文件。
- 把备份文件放到一个安全的地方。
- 发送以下两个命令：
- redis-cli config set appendonly yes
- redis-cli config set save ""
- 确保你的数据库含有其包含的相同的键的数量。
- 确保写被正确的追加到 AOF 文件。

第一个 CONFIG 命令开启 AOF。Redis 会阻塞以生成初始转储文件，然后打开文件准备写，开始追加写操作。

第二个 CONFIG 命令用于关闭快照持久化。这一步是可选的，如果你想同时开启这两种持久化方法。

重要：记得编辑你的 redis.conf 文件来开启 AOF，否则当你重启服务器时，你的配置修改将会丢失，服务器又会使用旧的配置。

此处省略一万字。。。。。。原文此处介绍 2.0 老版本怎么操作。

AOF 和 RDB 的相互作用(Interactions)

Redis 2.4 及以后的版本中，不允许在 RDB 快照操作运行过程中触发 AOF 重写，也不允许在 AOF 重写运行过程中运行 BGSAVE。这防止了两个 Redis 后台进程同时对磁盘进行繁重的 IO 操作。

当在快照运行的过程中，用户使用 BGREWRITEAOF 显式请求日志重写操作的话，服务器会答复一个 OK 状态码，告诉用户这个操作已经被安排调度，等到快照完成时开始重写。

Redis 在同时开启 AOF 和 RDB 的情况下重启，会使用 AOF 文件来重建原始数据集，因为通常 AOF 文件是保存数据最完整的。

备份数据(Backing up)

开始这一部分之前，请务必牢记：一定要备份你的数据库。磁盘损坏，云中实例丢失，等等：没有备份意味着数据丢失的巨大风险。

Redis 对数据备份非常友好，因为你可以在数据库运行时拷贝 RDB 文件：RDB 文件一旦生成就不会被修改，文件生成到一个临时文件中，当新的快照完成后，将自动使用 rename(2) 原子性的修改文件名为目标文件。

这意味着，在服务器运行时拷贝 RDB 文件是完全安全的。以下是我们的建议：

- 创建一个定时任务(cron job)，每隔一个小时创建一个 RDB 快照到一个目录，每天的快照放在另外一个目录。
- 每次定时脚本运行时，务必使用 find 命令来删除旧的快照：例如，你可以保存最近 48 小时内的每小时快照，一到两个月的内的每天快照。注意命名快照时加上日期时间信息。
- 至少每天一次将你的 RDB 快照传输到你的数据中心之外，或者至少传输到运行你的 Redis 实例的物理机之外。

灾难恢复(Disaster recovery)

在 Redis 中灾难恢复基本上就是指备份，以及将这些备份传输到外部的多个数据中心。这样即使一些灾难性的事件影响到运行 Redis 和生成快照的主数据中心，数据也是安全的。

由于许多 Redis 用户都是启动阶段的屌丝，没有太多钱花，我们会介绍一些最有意思的灾难恢复技术，而不用太多的花销。

- Amazon S3 和一些类似的服务是帮助你灾难恢复系统的一个好办法。只需要将你的每日或每小时的 RDB 快照以加密的方式传输到 S3。你可以使用 gpg -c 来加密你的数据(以对称加密模式)。确保将你的密码保存在不同的安全地方(例如给一份到你的组织中的最重要的人)。推荐使用多个存储服务来改进数据安全。
- 使用 SCP(SSH 的组成部分)来传输你的快照到远程服务器。这是一种相当简单和安全的方式：在远离你的位置搞一个小的 VPS，安装 ssh，生成一个无口令的 ssh 客户端 key，并将其添加到你的 VPS 上的 authorized_keys 文件中。你就可以自动的传输备份文件了。为了达到好的效果，最好是至少从不同的提供商那搞两个 VPS。

要知道这种系统如果没有正确的处理会很容易失败。至少一定要确保传输完成后验证文件的大小 (要匹配你拷贝的文件)，如果你使用 VPS 的话，可以使用 SHA1 摘要。

你还需要一个某种独立的告警系统，在某些原因导致的传输备份过程不正常时告警。

集中插入

有时候 Redis 实例需要在短时间内加载大量的已存在数据，或者用户产生的数据，这样，上百万的键将在很短的时间内被创建。

这被称为集中插入(mass insertion)，这篇文档的目的，就是提供如何最快地向 Redis 中插入数据的一些相关信息。

使用协议，伙计

使用标准的 Redis 客户端来完成集中插入并不是一个好主意，理由是：一条一条的发送命令很慢，因为你需要为每个命令付出往返时间的花费。可以使用管道(pipelining)，但对于许多记录的集中插入而言，你在读取响应的同时还需要写新命令，以确保插入尽可能快。

只有少部分的客户端支持非阻塞 I/O，也并不是所有的客户端都能高效地解析响应以最大化吞吐量。基于上述这些原因，首选的集中导入数据到 Redis 中的方式，是生成按照 Redis 协议的原始(raw)格式的文本文件，以调用需要的命令来插入需要的数据。

例如，如果我需要生成一个巨大的数据集，拥有数十亿形式为”keyN->ValueN” 的键，我将创建一个按照 Redis 协议格式，包含如下命令的文件：

```
SET Key0 Value0
SET Key1 Value1
...
SET KeyN ValueN
```

当这个文件被创建后，剩下的工作就是将其尽可能快的导入到 Redis 中。过去的办法是使用 netcat 来完成，命令如下：

```
(cat data.txt; sleep 10) | nc localhost 6379 > /dev/null
```

然而，这种集中导入的方式并不是十分可靠，因为 netcat 并不知道所有的数据什么时候被传输完，并且不能检查错误。在 github 上一个不稳定的 Redis 分支上，redis-cli 工具支持一种称为管道模式(pipe mode)的模式，设计用来执行集中插入。

使用管道模式运行命令如下：

```
cat data.txt | redis-cli --pipe
```

输出类似如下的内容：

```
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 1000000
```

redis-cli 工具也能够确保仅仅将来自 Redis 实例的错误重定向到标准输出。

生成 Redis 协议(Generating Redis Protocol)

Redis 协议非常容易生成和解析，可以参考其文档(请关注后续翻译文档，译者注)。但是，为了集中插入的目标而生成协议，你不必了解协议的每一个细节，仅仅需要知道每个命令通过如下方式来表示：

```
*<args><cr><lf>
$<len><cr><lf>
<arg0><cr><lf>
<arg1><cr><lf>
...
<argN><cr><lf>
```

`<cr>` 表示 "\r"(或 ASCII 字符 13)，`<lf>` 表示 "\n"(或者 ASCII 字符 10)。

例如，命令 SET key value 通过以下协议来表示：

```
*3<cr><lf>
$3<cr><lf>
SET<cr><lf>
$3<cr><lf>
key<cr><lf>
$5<cr><lf>
value<cr><lf>
```

或者表示为一个字符串：

```
"*3\r\n$3\r\nSET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n"
```

为集中插入而生成的文件，就是由一条一条按照上面的方式表示的命令组成的。

下面的 Ruby 函数生成合法的协议。

```
def gen_redis_proto(*cmd)
proto = ""
    proto << "*"+cmd.length.to_s+"\r\n"
    cmd.each{|arg|
        proto << "$"+arg.to_s.bytesize.to_s+"\r\n"
        proto << arg.to_s+"\r\n"
    }
    proto
end

puts gen_redis_proto("SET","mykey","Hello World!").inspect
```

使用上面的函数，可以很容易地生成上面例子中的键值对。程序如下：

```
(0...1000).each{|n|
STDOUT.write(gen_redis_proto("SET","Key#{n}","Value#{n}"))
}
```

我们现在可以直接以 redis-cli 的管道模式来运行这个程序，来执行我们的第一次集中导入会话。

```
$ ruby proto.rb | redis-cli --pipe
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 1000
```

管道模式如何工作(How works)

redis-cli 管道模式的魔力，就是和 netcat 一样的快，并且能理解服务器同时返回的最后一条响应。

按照以下方式获得：

- redis-cli -pipe 尝试尽可能快的发送数据到服务器。
- 与此同时读取可用数据，并尝试解析。
- 当标准输入没有数据可读时，发送一个带有 20 字节随机字符的特殊 ECHO 命令：我们确保这是最后发送的命令，我们也确保可以匹配响应的检查，如果我们收到了相同的 20 字节的批量回复(bulk reply)。
- 一旦这个特殊的最后命令被发送，收到响应的代码开始使用这 20 个字节来匹配响应。当匹配响应到达后成功退出。

使用这个技巧，我们不需要为了知道发送了多少命令而解析发送给服务端的协议，仅仅只需要知道响应就可以。

但是，在解析响应的时候，我们对所有已解析响应进行了计数，于是最后我们可以告诉用户，通过集中插入会话传输给服务器的命令的数。

13

高可用（上）

Redis Sentinel 是 Redis 的官方高可用解决方案，是设计用来帮助管理 Redis 实例的系统。用于完成下面 4 个任务：

- 监控(Monitoring)。Sentinel 不断检查你的主从实例是否运转正常。
- 通知(Notification)。Sentinel 可以通过 API 来通知系统管理员，或者其他计算机程序，被监控的 Redis 实例出了问题。
- 自动故障转移(Automatic failover)。如果一台主服务器运行不正常，Sentinel 会开始一个故障转移过程，将从服务器提升为主服务器，配置其他的从服务器使用新的主服务器，使用 Redis 服务器的应用程序在连接时会收到新的服务器地址通知。
- 配置提供者(Configuration provider)。Sentinel 充当客户端服务发现的权威来源：客户端连接到 Sentinel 来询问某个服务的当前 Redis 主服务器的地址。当故障转移发生时，Sentinel 会报告新地址。

分布式特性(Distributed nature)

Redis Sentinel 是一个分布式系统，这意味着，你通常想要在你的基础设施中运行多个 Sentinel 进程，这些进程使用 gossip 协议来判断一台主服务器是否下线(down)，使用 agreement 协议来获得授权以执行故障转移，并更新相关配置。

分布式系统具有特定的安全(safety)和活性(liveness)的问题，为了更好地使用 Redis Sentinel，你应该去理解 Sentinel 是如何作为一个分布式系统运转的，至少在较高的层面上。这会让 Sentinel 变得更复杂，但是比单进程系统更好，例如：

- Sentinel 集群能对主服务器故障转移，即使部分 Sentinel 失败。
- 单个 Sentinel 工作不正常，或者连接不正常，在没有别的 Sentinel 授权的情况下不能故障转移主服务器。
- 客户端可以随机连接到任何一个 Sentinel 来获取主服务器的配置信息。

获取 Sentinel(Obtaining Sentinel)

当前版本的 Sentinel 被称为 Sentinel 2。使用了更强大和简单的算法来重写最初的 Sentinel 实现 (本文后面会解释)。

稳定版本的 Redis Sentinel 被打包在 Redis 2.8 中，这是最新的 Redis 版本。

新的开发在不稳定的分支中进行，新的特性一旦稳定了就会合并回 2.8 分支。

重要：即使你使用的是 Redis 2.6，你也应该使用 Redis 2.8 自带的 Sentinel。Redis 2.6 自带的 Sentinel，也就是 Sentinel 1，已经不赞成使用，并且有很多的 bug。总之，你应该尽快把你的 Redis 和 Sentinel 实例都迁移到 Redis 2.8，以获得更好的全面体验。

运行 Sentinel(Running Sentinel)

如果你使用 redis-sentinel 可执行文件 (或者如果你有一个叫这个名字的到 redis-server 的符号链接)，你可以使用下面的命令行来运行 Sentinel：

```
redis-sentinel  /path/to/sentinel.conf
```

另外，你可以直接使用 redis-server 可执行文件并作为 Sentinel 模式启动：

```
redis-server  /path/to/sentinel.conf  --sentinel
```

两种方式是一样的。

但是，运行 Sentinel 强制使用配置文件，这个文件被系统用来保存当前状态，在重启时能重新加载。如果没有指定配置文件，或者配置文件的路径不可写，Sentinel 将拒绝启动。

Sentinel 运行时默认监听 TCP 端口 26379，所以为了让 Sentinel 正常运行，你的服务器必须开放 26379 端口，以接受从其他 Sentinel 实例 IP 地址的连接。否则，Sentinel 间就没法通信，没法协调，也不会执行故障转移。

配置 Sentinel(Configuring Sentinel)

Redis 的源码发行版中包含一个叫做 sentinel.conf 的自说明示例配置文件，可以用来配置 Sentinel，一个典型的最小配置文件看起来就像下面这样：

```
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 60000
sentinel failover-timeout mymaster 180000
sentinel parallel-syncs mymaster 1

sentinel monitor resque 192.168.1.3 6380 4
sentinel down-after-milliseconds resque 10000
sentinel failover-timeout resque 180000
sentinel parallel-syncs resque 5
```

你只需要指定你要监控的主服务器，并给每一个主服务器(可以拥有任意多个从服务器)一个不同的名字。没有必要指定从服务器，因为它们会被自动发现。Sentinel 会根据从服务器的额外信息来自动更新配置(为了在重启时还能保留配置)。每次故障转移时将一台从服务器提升为主服务器时都会重写配置文件。

上面的示例配置监控了两个 Redis 实例集合，每个由一个主服务器和未知数量的从服务器组成。其中一个实例集合叫做 mymaster，另一个叫做 resque。

为了说得再清楚一点，我们一行一行地来看看这些配置选项是什么意思：

第一行告诉 Redis 监控一个叫做 mymaster 的主服务器，地址为 127.0.0.1，端口为 6379，判断这台主服务器失效需要 2 个 Sentinel 同意(如果同意数没有达到，自动故障转移则不会开始)。

但是要注意，无论你指定多少个同意来检测实例是否正常工作，Sentinel 需要系统中已知的大多数 Sentinel 的投票才能开始故障转移，并且在故障转移之后获取一个新的配置纪元(configuration Epoch) 赋予新的配置。

在例子中，仲裁人数 (quorum) 被设置为 2，所以使用 2 个 Sentinel 同意某台主服务器不可到达或者在一个错误的情况中，来触发故障转移(但是，你后面还会看到，触发一个故障转移还不足以开始一次成功的故障转移，还需要授权)。

其他的选项基本上都是这样的形式：

```
sentinel <option_name> <master_name> <option_value>
```

它们的作用如下：

down-after-milliseconds 表示要使 Sentinel 开始认为实例已下线(down)，实例不可到达(没有响应我们的 PING，或者响应一个错误) 的毫秒数。这个时间过后，Sentinel 将标记实例为主观下线(subjectively down，也称 SDOWN)，这还不足以开启自动故障转移。但是，如果足够的实例认为具备主观下线条件，实例就会被标记为客观下线(objectively down)。需要同意的 Sentinel 数量依赖于为这台主服务器的配置。

parallel-syncs 设置在一次故障转移之后，被配置为同时使用新主服务器的从服务器数量。这个数字越小，完成故障转移过程需要的时间就越多，如果从服务器配置为服务旧数据，你可能不太希望所有的从服务器同时从新的主服务器重同步，尽管复制过程通常不会阻塞从服务器，但是在重同步过程中仍然会有一段停下来的时间来加载来自于主服务器的大量数据。设置这个选项的值为 1 可以确保每次只有一个从服务器不可用。

其他的选项将在本文的剩余篇幅里介绍，Redis 发行版本中自带的示例 sentinel.conf 文件中也有详细的文档。

所有的配置参数可以在运行时用 SENTINEL SET 命令修改。请看下文中运行时重新配置 Sentinel 这一部分获取更多的信息。

仲裁人数(Quorum)

本文前面的部分展示了每一个被 Sentinel 监控的主服务器都关联了一个仲裁人数的配置。它指定了同意主服务器不可达或者错误条件需要的 Sentinel 进程数，以触发一次故障转移。

但是，故障转移被触发后，为了让故障转移真正执行，必须至少大多数的 Sentinel 授权某个 Sentinel 才能错误转移。

让我们解释的再清楚一些：

- 仲裁人数：检测错误条件以标记主服务器为 ODOWN 所需要的 Sentinel 进程数。
- 故障转移由 ODOWN 状态触发。
- 一旦故障转移被触发，故障转移的 Sentinel 需要向大多数 Sentinel 请求授权(或者大于大多数，如果仲裁人数设置为大于大多数的话)。

差别看起来很微妙，但是实际上理解和使用起来都相当简单。例如，如果你有 5 个 Sentinel 实例，然后设置仲裁人数为 2，只要有 2 个 Sentinel 认为主服务器不可达就会触发一次故障转移，这两个 Sentinel 仅当得到至少 3 个 Sentinel 的授权时才能故障转移。

如果设置仲裁人数为 5，所有的 Sentinel 都必须同意主服务器的错误条件，故障转移需要所有 Sentinel 的授权。

配置纪元 (Configuration epochs)

Sentinel 需要大多数的授权来开启故障转移是有几个重要原因的：

当一个 Sentinel 得到授权了，就会为故障转移的主服务器获得一个唯一的配置纪元。这是在故障转移完成后用于标记新的配置的一个版本数字。因为大多数同意将一个指定的版本赋予一个指定的 Sentinel，所以其它的 Sentinel 不能使用它。这意味着，每一次故障转移的配置都使用一个唯一的版本来标记。我们会看到为什么这个是如此的重要。

另外，Sentinel 有一个规则：如果一个 Sentinel 为了指定的主服务器故障转移而投票给另一个 Sentinel，将会等待一段时间后试图再次故障转移这台主服务器。这个延时(delay)是 failover-timeout，你可以在 sentinel.conf 中配置。这意味着，Sentinel 不会同时故障转移同一台主服务器，第一个请求被授权的将会尝试，如果失败了，过一会后另一个将会尝试，等等。

Redis Sentinel 保证活性(liveness)属性，如果大多数 Sentinel 能够对话，如果主服务器下线，最后只会有一个被授权来故障转移。

Redis Sentinel 也保证安全(safety)属性，每个 Sentinel 将会使用不同的配置纪元来故障转移同一台主服务器。

配置传播(Configuration propagation)

一旦一个 Sentinel 能够成功故障转移一台主服务器，会开始广播新的配置，从而使其他 Sentinel 更新关于这台主服务器的信息。

为了认定故障转移是成功的，需要 Sentinel 能发送 SLAVEOF NO ONE 给选定的从服务器，并将其切换为主服务器，稍后可以在主服务器的 INFO 输出中观察到。

这时，即使从服务器的重新配置还在进行中，故障转移被认为是成功的，所有的 Sentinel 被要求开始报告新的配置。

新配置传播的方式，就是为什么我们需要每次 Sentinel 故障转移时被授权一个不同的版本号(配置纪元)的原因。

每一个 Sentinel 使用 Redis 的发布订阅(Pub/Sub)消息不断地广播主服务器的配置版本，在主服务器上以及所有从服务器上。与此同时，所有的 Sentinel 等待其它 Sentinel 通知的配置消息。

配置信息在__sentinel__:hello 频道中广播。

因为每一个配置有一个不同的版本号，所以更大的版本号总是胜过更小的版本号。

例如，一开始所有的 Sentinel 认为主服务器 mymaster 的配置为 192.168.1.50:6379。这个配置拥有版本 1。一段时间以后，一个 Sentinel 被授权以版本 2 来故障转移。如果故障转移成功，会广播一个新的配置，比如说 192.168.1.50:9000，作为版本 2。所有其他实例会看到这个配置，并相应地更新它们的配置，因为新的配置拥有一个更大的版本号。

这意味着，Sentinel 保证第二个活性属性：一个可以相互通信的 Sentinel 集合会统一到一个拥有更高版本号的相同配置上。

基本上，如果网络是分割的，每个分区会统一到一个更高版本的本地配置。在没有分割的特殊情况下，只有一个分区，每个 Sentinel 将会配置一致。

SDOWN 和 ODOWN 更多细节

正如本文已经简要提到的，Redis Sentinel 有两个不同的下线概念，一个被称为主观下线条件(SDOWN)，一个本地 Sentinel 实例的下线条件。另一个称为客观下线条件(ODOWN)，当足够的 Sentinel(至少为主服务器 quorum 参数配置的数量) 具有 SDOWN 条件时就满足 ODOWN，并且使用 SENTINEL is-master-down-by-addr 命令从其它 Sentinel 获得反馈。

从 Sentinel 的角度来看，如果我们没有在配置的 is-master-down-after-milliseconds 参数的指定时间内收到一个 PING 请求的合法响应，就达到了 SDOWN 的条件。

PING 的可接受响应可以是以下其中之一：

- 回复 + PONG。
- 回复 - LOADING 错误。
- 回复 - MASTERDOWN 错误。

其它回复 (或者没有回复) 都被认为是不合法的。

注意，SDOWN 需要在配置的整个时间区间内没有收到可以接受的回复，例如，如果间隔配置为 30000 毫秒(30 秒)，我们每隔 29 秒收到一个可以接受的 ping 回复，实例被认为是正常工作的。

从 SDOWN 切换到 ODOWN 没有使用强一致性算法，而仅仅是 gossip 的形式：如果一个指定的 Sentinel 在指定的时间范围内从足够多的 Sentinel 那里获得关于主服务器不工作的报告，SDOWN 就被提升为 ODOWN。如果这种报告不再收到，(ODOWN)标记就会被清除。

正如已经解释过的，真正开始故障转移需要更严格的授权，但是，如果没有达到 ODOWN 状态，是不会触发故障转移的。

ODOWN 条件只适用于主服务器。对于其他的实例，Sentinel 不需要任何同意，所以从服务器和其它 Sentinel 永远都不会达到 ODOWN 状态。

自动发现(Auto discovery)

Sentinel 之间保持着连接来互相检查彼此的可用性，互相交换信息，你不需要在每个你运行的 Sentinel 实例中配置其他 Sentinel 的地址，因为 Sentinel 使用 Redis 主服务器的发布订阅能力来发现监控同一台主服务器的其他 Sentinel。

这是通过向名为__sentinel__:hello 频道发送问候消息 (Hello Messages) 实现的。

同样，你不需要配置连接在主服务器上的从服务器列表，因为 Sentinel 会通过询问 Redis 自动发现这个列表。

- 每个 Sentinel 每隔 2 秒向每个被监控的主服务器和从服务器的发布订阅频道__sentinel__:hello 发送一条消息，报告自己的存在状态：IP 地址，端口号和 runid。
- 每个 Sentinel 订阅了每个主服务器和从服务器的发布订阅频道__sentinel__:hello，寻找未知的 Sentinel。当检测到新的 Sentinel，就将其添加到这台主服务器的 Sentinel 列表中。
- 问候消息也包括主服务器当前的完整配置。如果另一个 Sentinel 拥有一个比接收到的更老的主服务器配置，会立刻更新为新的配置。
- 在添加一个新的 Sentinel 到主服务器前，Sentinel 总是检查是否已经有一个相同的 runid 或者相同地址(IP 地址和端口对)的 Sentinel。如果是的话，所有匹配的 Sentinel 将会被删除，新的被添加。

14

高可用（下）

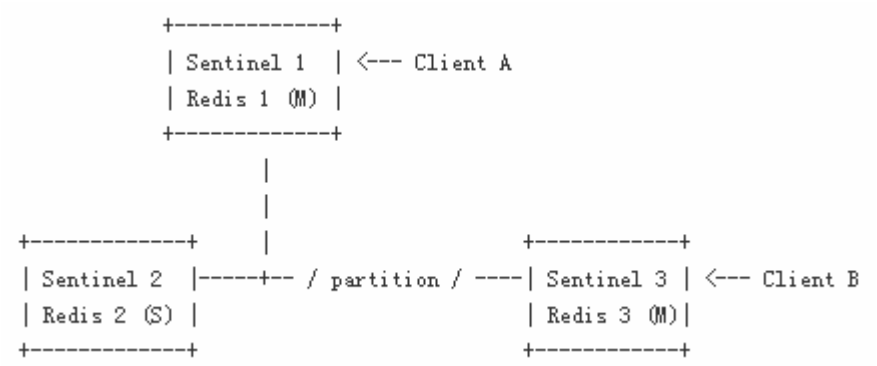
分割下的一致性(Consistency under partitions)

Redis Sentinel 的配置是最终一致性的，所以每个分区会被统一到一个可用的更高版本的配置。但是，在使用 Sentinel 的真实世界系统中有三个不同的角色：

- Redis 实例。
- Sentinel 实例。
- 客户端。

为了定义系统的行为，我们得考虑这三个角色。

下面是一个有三个节点的简单网络，每一个节点运行一个 Redis 实例和一个 Sentinel 实例：



在这个系统中，初始状态是 Redis 3 是主服务器，Redis 1 和 Redis 2 是从服务器。分割发生了，隔断了老的主服务器。Sentinel 1 和 2 开始故障转移，提升 Sentinel 1 作为新的主服务器。

Sentinel 的属性保证，Sentinel 1 和 2 现在拥有主服务器的最新配置。但是，Sentinel 3 仍是旧的配置，因为它存在于一个不同的分割中。

当网络分割恢复正常了，Sentinel 3 将会更新其配置，但是，如果有客户端与老的主服务器被分割在一起，在分割期间会发生什么事情呢?

客户端会继续向 Redis 3 写，即老的主服务器。当分割又聚合在一起，Redis 3 将会变成 Redis 1 的从服务器，分割期间所有写入的数据会丢失。

你可能想或者不想这种场景发生，取决于你的配置：

- 如果你将 Redis 用作缓存，客户端 B 可以继续往老的主服务器写，即使这些数据会丢失。
- 如果你将 Redis 用作存储，这样就不好了，你需要来配置系统以部分地阻止问题的发生。

因为 Redis 是异步复制，这种场景下没有完全阻止数据丢失的办法，但是你可以使用下面的 Redis 配置选项，来限制 Redis 3 和 Redis 1 之间的分歧：

```
min-slaves-to-write 1
min-slaves-max-lag 10
```

有了上面的配置(请查看 Redis 分发版本中自带的 redis.conf 文件中的注释获取更多的信息)，扮演主服务器的 Redis 实例如果不能写入到至少一个从服务器，将会停止接受写请求。由于复制是异步的，不能写入的意思就是从服务器也是断开的，或者在指定的 max-lag 秒数没有发送异步回应 (acknowledges)。

使用这个配置，上面例子中的 Redis 3 在 10 秒钟之后变得不可用。当分割恢复了，Sentinel 3 的配置将会统一为新的，客户端 B 可以获取合法的配置并且继续。

Sentinel 的持久化状态 (Sentinel persistent state)

Sentinel 的状态被持久化在 Sentinel 的配置文件中。例如，每次创建(领导者 leader Sentinel)或者收到新的配置，主服务器会将配置连同配置纪元持久化到磁盘中。这意味着，停止和重启 Sentinel 进程是安全的。

Sentinel 重配置实例(Sentinel reconfiguration of instances)

即使没有故障转移在进行中，Sentinel 也会一直尝试在被监控的实例上设置当前配置。尤其是：

- 声称要成为主服务器的从服务器(根据当前配置)，会被配置为从服务器来复制当前主服务器。
- 连接到错误主服务器的从服务器，会被重新配置来复制正确的主服务器。
- 为了 Sentinel 重新配置从服务器，错误的配置必须要观察一段时间，一段大于用于广播新配置所使用的时间。

这防止了持有旧配置(例如，因为刚刚从分割中恢复)的 Sentinel 会尝试在收到变更之前改变从服务器的配置。

也要注意，一直尝试使用当前配置使得故障转移对分割具有更强的抵抗力的语义是什么：

- 被故障转移的主服务器当再次可用时被重新配置成从服务器。
- 被分割的从服务器在一旦可到达时被重新配置。

从服务器的选举和优先级(Slave selection and priority)

当 Sentinel 实例准备执行故障转移，也就是当主服务器处于 ODOWN 状态，并且 Sentinel 从大多数已知 Sentinel 实例收到了故障转移授权，需要选择一个合适的从服务器。

从服务器的选择过程评估从服务器的以下信息：

1. 从主服务器断开的时间。
2. 从服务器的优先级。
3. 已处理的复制偏移量。
4. 运行 ID。

一个从服务器被发现从主服务器断开超过十倍于配置的主服务器超时(down-after-milliseconds 选项)，加上从正在执行故障转移的 Sentinel 的角度来看主服务器也不可用的时间，将会被认为不适合用于故障转移并跳过。

更严谨地说，一个从服务器的 INFO 输出表明已从主服务器断开超过：

```
(down-after-milliseconds * 10) + milliseconds_since_master_is_in_SDOWN_state
```

就被认为不可靠并且被抛弃。

从服务器选择只考虑通过了上面测试的从服务器，并且基于上面的标准排序，使用下面的顺序。

1. 从服务器按照 Redis 实例的 redis.conf 文件中配置的 slave-priority 排序。更低的优先级更偏爱。
2. 如果优先级相同，将检查已处理的复制偏移量，从主服务器收到更多数据的从服务器将被选择。
3. 如果多个从服务器有相同的优先家，并且从主服务器处理完相同的数据，将执行进一步的检查，选择按照字典顺序具有更小运行 ID 的从服务器。拥有较小的运行 ID 对从服务器并不是一个真正的优势，但是有助于从服务器选举过程更具有确定性，而不是随机选择一个。

如果对机器有强烈的偏好的话，Redis 主服务器(故障转移以后成为从服务器)和从服务器都需要配置 slave-priority。否则，所有的实例都可以使用默认的运行 ID 来运行(这是建议的设置，因为按照复制偏移量来选择从服务器要有趣得多)。

Redis 实例可以配置一个特殊的 slave-priority 值 0，这样就一定不会被 Sentinel 选举为新的主服务器。但是，按照这样配置的从服务器仍然会被 Sentinel 重新配置，从而在故障转移后复制新的主服务器，唯一的区别是永远不会变成主服务器。

Sentinel 和 Redis 身份验证(authentication)

当主服务器被配置为需要客户端传递密码时，作为安全措施，从服务器也需要知道这个密码来验证主服务器，并且创建用于异步复制协议的主从连接。

使用下面的配置指令完成：

主服务器中的 requirepass 用来设置密码验证，以确保实例不会处理没有验证过的客户端的请求。 从服务器中的 masterauth 用于从服务器验证主服务器，以正确的从其复制数据。

当使用 Sentinel 就没有单一的主服务器，因为故障转移以后从服务器可以扮演主服务器的角色，老的主服务器被重新配置以扮演从服务器，所以你要做的就是在你所有的主服务器和从服务器实例中设置以上指令。

这通常是一种逻辑上健全的设置，因为你不想只是保护主服务器中的数据，从服务器中也应拥有同样可访问的数据。

但是，在一些不常见的情况下，你需要从服务器无需验证就能访问，你仍可以通过设置从服务器的优先级为 0(这将不允许从服务器被提升为主服务器)，只为从服务器配置 masterauth 指令，不配置 requirepass 指令这样来做到，这样数据就可以让未经验证的客户端读取。

Sentinel API

Sentinel 运行默认使用 TCP 端口 26379(注意，6379 是正常的 Redis 端口)。Sentinel 接受使用 Redis 协议的命令，所以你可以使用 redis-cli 或者任何其他未修改的 Redis 客户端与 Sentinel 对话。

有两种方式与 Sentinel 对话：可以直接查询它来检查被监控的 Redis 实例的状态，看看它知道的其他 Sentinel，等等。

另外一种方式是使用发布订阅，每当某个事件发生时，例如故障转移，或者一个实例进入到了一个错误条件，等等，接收从 Sentinel 推过来的通知。

Sentinel 命令

下面是可接受的命令清单：

- PING：这个命令仅仅返回 PONG。
- SENTINEL masters：展示被监控的主服务器列表及其状态。
- SENTINEL master： 展示指定主服务器的状态和信息。
- SENTINEL slaves： 展示指定主服务器的从服务器列表及其状态。
- SENTINEL get-master-addr-by-name： 根据名字返回主服务器的 IP 地址和端口号。如果这台主服务器正在故障转移过程中或者成功结束了，返回被提升的从服务器的 IP 地址和端口。
- SENTINEL reset <pattern>：这个命令根据匹配的名字重置所有主服务器。pattern 参数是通配符风格(glob-style)。重置进程清除主服务器的任何先前状态(包括进行中的故障转移)，移除每一个主服务器上被发现和关联的从服务器和 Sentinel。
- SENTINEL failover 当主服务器不可达时强制故障转移，无需要求其他的
- Sentinel 同意(但是会发布一个新的配置版本，这样其他 Sentinel 就会更新它们的配置)。

运行时重配置 Sentinel(Reconfiguring Sentinel)

从 Redis 2.8.4 开始，Sentinel 提供了用于添加，删除和改变指定主服务器配置的 API。注意，如果你有多个 Sentinel 实例，你得将改变应用到所有的 Redis Sentinel 实例才能运转正常。也就是说，改变一个 Sentinel 的配置不会自动传播到网络中的其它 Sentinel。

下面是 SENTINEL 的子命令清单，用于更新 Sentinel 实例的配置。

- SENTINEL MONITOR <name> <ip> <port> <quorum>：这个命令告诉 Sentinel 开始监控一个指定名字，IP 地址，端口和仲裁人数的新主服务器。这等同于 sentinel.conf 配置文件中的 sentinel monitor 配置指令，不同之处在于此处不能使用主机名作为 IP 地址，你需要提供一个 IPv4 或者 Ipv6 地址。
- SENTINEL REMOVE <name>：用于删除指定主服务器：主服务器不再被监控，完全从 Sentinel 内部状态中移除，所以不会被 SENTINEL masters 列出，等等。
- SENTINEL SET <name> <option> <value>：命令 SET 非常类似于 Redis 的 CONFIG SET 命令，用于改变指定主服务器的配置参数。可以指定多个选项 – 值对(或者根本啥都没有)。所有可以通过 sentinel.conf 配置的配置参数都可以通过 SET 命令配置。

下面是 SENTINEL SET 命令的一个例子，用于修改一个名为 objects-cache 的主服务器的 down-after-milliseconds 配置：

```
SENTINEL SET objects-cache-master down-after-milliseconds 1000
```

启动以后，SENTINEL SET 能用于设置所有在启动配置文件中可设置的配置参数。此外，还可以仅仅只改变主服务器的仲裁人数配置，而不需要使用 SENTINEL REMOVE 和 SENTINEL MONITOR 来删除和重新添加主服务器，而只需要：

```
SENTINEL SET objects-cache-master quorum 5
```

注意，没有与之等价的 GET 命令，因为 SENTINEL MASTER 以一种易于解析的格式(作为一个字段 – 值对数组)提供了所有的配置参数。

添加和删除 Sentinel(Adding or removing Sentinels)

因为 Sentinel 实现的自动发现机制，添加一个新的 Sentinel 到你的部署中是一个很简单的过程。所有你需要干的就是启动一个配置用于监控当前活跃主服务器的 Sentinel。在 10 秒钟之内，Sentinel 就会获得其他 Sentinel 的列表以及连接到主服务器的从服务器集合。

如果你想一次添加多个新的 Sentinel，建议一个一个的添加，等待所有其他的 Sentinel 知道了第一个再添加另一个。这在当添加新 Sentinel 的过程中发生错误时，仍然保证在分割的一侧能达到大多数时很有用。

在没有网络分割时，这可以通过添加每个新的 Sentinel 时带 30 秒的延迟来轻易实现。

在最后，可以使用命令 SENTINEL MASTER mastername 来检查是否所有的 Sentinel 就监控主服务器的 Sentinel 数量达成一致。

删除一个 Sentinel 要稍微复杂一些：Sentinel 永远不会忘记已经发现的 Sentinel，即使他们在很长一段时间内都不可达，因为我们不想动态改变用于授权故障转移所需要的大多数以及创建新的配置版本。所以在没有网络分割情况下，需要执行下面的步骤来删除 Sentinel：

1. 停止你想删除的 Sentinel 的进程。
2. 发送 SENTINEL RESET * 命令到所有其他的 Sentinel 实例(如果你想重置单个主服务器可以使用精确的主服务器名来代替 *)。一个一个的来，前后等待至少 30 秒。
3. 通过检查每个 SENTINEL MASTER mastername 的输出，来检查所有的 Sentinel 就当前活跃的 Sentinel 数量达成一致。

删除旧的主服务器或不可达从服务器(unreachable)

Sentinel 不会忘记主服务器的从服务器，即使在很长时间内都不可达。这很有用，因为这样 Sentinel 能够在网络分割或者错误事件恢复后正确地重新配置一个返回的从服务器。

此外，故障转移之后，被故障转移的主服务器事实上被添加为新主服务器的从服务器，这样一旦恢复重新可用，就会被重新配置来复制新的主服务器。

但是，有时候你想从 Sentinel 监控的从服务器列表中永久删除一个从服务器(可能是旧的主服务器)。

要做到这个，你需要发送 SENTINEL RESET masternamе 命令到所有的 Sentinel：在接下来的 10 秒内，他们会刷新从服务器列表，只添加当前主服务器 INFO 输出中的正确复制的清单。

发布和订阅消息(Pub/Sub Messages)

客户端可以将 Sentinel 作为一个 Redis 兼容的发布订阅服务器(但是你不能使用 PUBLISH)来使用，来订阅或者发布到频道，获取指定事件通知。

频道名称与事件名称是一样的。例如，名为 + sdown 的频道会收到所有关于实例进入 SDOWN 条件的通知。

简单使用 PSUBSCRIBE * 订阅来获得所有的消息。

下面是频道的清单，以及使用这个 API 你会收到的消息格式。第一个单词是频道/事件名称，剩下的是数据的格式。

注意：指定 instance details 的地方表示提供了下面用于表示目标实例的参数：

```
<instance-type> <name> <ip> <port> @ <master-name> <master-ip> <master-port>
```

标识主服务器的部分 (从 @参数到结束) 是可选的，只在实例不是主服务器本身时指定。

- +reset-master： 主服务器被重置。
- +slave： 一个新的从服务器被发现和关联。
- +failover-state-reconf-slaves： 故障转移状态切换为 reconf-slaves 状态。
- +failover-detected： 另一个 Sentinel 启动了故障转移，或者任何其它外部实体被发现(关联的从服务器变为主服务器)。
- +slave-reconf-sent： 领导者 Sentinel 发送了 SLAVEOF 命令到这个实例，重新配置为新的从服务器。
- +slave-reconf-inprog： 从服务器正在重新配置为新的主服务器的从服务器，但是同步过程尚未完成。
- +slave-reconf-done： 从服务器完成了与新主服务器的同步。
- -dup-sentinel： 由于重复，指定主服务器的一个或多个 Sentinel 被移除。
- +sentinel： 这个主服务器的新的 Sentinel 被发现和关联。
- +sdown： 指定的实例处于主观下线状态。
- -sdown： 指定的实例不再处于主观下线状态。
- +odown： 指定的实例处于客观下线状态。
- -odown： 指定的实例不再处于客观下线状态。
- +new-epoch： 当前纪元被更新。
- +try-failover： 新的故障转移进行中，等待被大多数选中。

- +elected-leader： 赢得指定纪元的选举，可以进行故障转移。
- +failover-state-select-slave： 新的故障转移状态是 select-slave：正在寻找合适的从服务器来提升。
- no-good-slave： 没有合适的从服务器来提升。一段时间后会重试，或者干脆放弃故障转移。
- selected-slave： 找到合适的从服务器来提升。
- failover-state-send-slaveof-noone： 正在重新配置将提升的从服务器为主服务器，等待完成后切换。
- failover-end-for-timeout： 故障转移由于超时而终止， 无论如何从服务器最终被配置为复制新的主服务器。
- failover-end： 故障转移顺利完成。所有从服务器被重配置为复制新主服务器。
- switch-master <oldip> <oldport> <newip> <newport>：配置变更后主服务器的 IP 和地址都是指定的。这是大多数外部用户感兴趣的消息。
- +tilt：进入 tilt 模式。
- -tilt：退出 tilt 模式。

TILT 模式

Redis Sentinel 严重依赖于计算机时间：例如，为了了解一个实例是否可用，Sentinel 会记住最近成功回复 PING 命令的时间，与当前时间对比来了解这有多久。

但是，如果计算机时间以不可预知的方式改变了，或者计算机非常繁忙，或者某些原因进程阻塞了，Sentinel 可能会开始表现得不可预知。

TILT 模式是一个特别的保护模式，当发现一些会降低系统可靠性的奇怪问题时，Sentinel 就会进入这种模式。Sentinel 的定时中断通常每秒钟执行 10 次，所以我们期待两次定时中断调用之间相隔 100 毫秒左右。

Sentinel 做的就是记录上一次定时中断调用的时间，与当前调用进行比较：如果时间差是负数或者出乎意料的大(2 秒或更多)，就进入了 TILT 模式(或者如果已经进入了，退出 TILT 模式将被推迟)。

当处于 TILT 模式时，Sentinel 会继续监控一切，但是：

- 停止一切动作。
- 开始回复负数给 SENTINEL is-master-down-by-addr 请求，因为检测失败的能力不再可信了。

如果一切表现正常了 30 秒，将退出 TILT 模式。

处理 – BUSY 状态

(警告：还未实现)

当脚本运行超过配置的脚本限制时间时返回 – BUSY 错误。当这种情况发生时，在触发故障转移之前 Redis Sentinel 会尝试发送 SCRIPT KILL 命令，这只有在脚本是只读的情况下才能成功。

Sentinel 客户端实现

Sentinel 需要显式的客户端支持，除非系统被配置为执行一个脚本，来实现透明重定向所有请求到新的主服务器实例(虚拟 IP 或其它类似系统)。客户端库实现的主题在 Sentinel 客户端指引手册中讨论(请期待本系列后续文档，译者注)。

15

高可用客户端指引

本文档是一篇草案，其包含的指引将来可能会随着Sentinel项目的进展而改变。

支持Redis Sentinel的Redis客户端指引

Redis Sentinel是Redis实例的监控解决方案，处理Redis主服务器的自动故障转移和服务发现(谁是一组实例中的当前主服务器)。由于Sentinel具有在故障转移期间重新配置实例，以及提供配置给连接Redis主服务器或者从服务器的客户端的双重责任，客户端需要有对Redis Sentinel的显式支持。

这篇文档针对Redis客户端开发人员，他们想在其客户端实现中支持Sentinel，以达到如下目标：

- 通过Sentinel实现客户端的自动配置。
- 改进Sentinel自动故障转移的安全性。# 高可用客户端指引

本文档是一篇草案，其包含的指引将来可能会随着 Sentinel 项目的进展而改变。

支持 Redis Sentinel 的 Redis 客户端指引

Redis Sentinel 是 Redis 实例的监控解决方案，处理 Redis 主服务器的自动故障转移和服务发现(谁是一组实例中的当前主服务器)。由于 Sentinel 具有在故障转移期间重新配置实例，以及提供配置给连接 Redis 主服务器或者从服务器的客户端的双重责任，客户端需要有对 Redis Sentinel 的显式支持。

这篇文档针对 Redis 客户端开发人员，他们想在其客户端实现中支持 Sentinel，以达到如下目标：

- 通过 Sentinel 实现客户端的自动配置。
- 改进 Sentinel 自动故障转移的安全性。

要想获得 Redis Sentinel 如何工作的细节，请查看相关文档(请查看本系列相关文章，译者注)，本文只包含 Redis 客户端开发人员需要的信息，期待读者已经比较熟悉 Redis Sentinel 的工作方式。

通过 Sentinel 实现 Redis 服务发现(Redis service discovery)

Redis Sentinel 通过像”stats”或”cache”这样的名字来识别每个主服务器。每个名字实际上标识了一组实例，由一个主服务器和若干个从服务器组成。

网络中用于特定目的的 Redis 主服务器的地址，在一些像自动故障转移，手工触发故障转移(例如，为了提升一个 Redis 实例)，或者其他原因引起的这样的事件后可能会改变。

通常，Redis 客户端中有一些硬编码的配置来指定 IP 地址和端口作为网络中 Redis 主服务器的地址。但是，如果主服务器的地址改变了，就需要手工介入到每个客户端了。

支持 Sentinel 的 Redis 客户端可以从使用 Sentinel 的主服务器的名称自动发现 Redis 的地址。所以支持 Sentinel 的客户端应该可以从输入中获得，而不是硬编码的 IP 地址和端口：

- 指向已知的 Sentinel 实例的 ip:port 对列表。
- 服务的名称，像”timelines”或者”cache”。

下面是客户端为了从 Sentinel 列表和服务名称获得主服务器地址而需要遵循的步骤。

第 1 步：连接第一个 Sentinel(connecting to the first Sentinel)

客户端应该迭代 Sentinel 地址列表。应该尝试使用较短的超时(大约几百毫秒)来连接到每一个地址的 Sentinel。遇到错误或者超时就尝试下一个 Sentinel 地址。

如果所有的 Sentinel 地址都没有尝试成功，就返回一个错误给客户端。

第一个回应客户端请求的 Sentinel 被置于列表的开头，这样在下次重连时，我们会首先尝试在上一次连接尝试是可达的 Sentinel，以最小化延迟。

第 2 步：请求主服务器地址(ask for master address)

一旦与 Sentinel 的连接建立起来，客户端应该重新尝试在 Sentinel 上执行下面的命令：

```
SENTINEL get-master-addr-by-name master-name
```

这里的 master-name 应该被替换为用户指定的真实服务名称。

调用的结果可能是下面两种回复之一：

- ip:port 对。
- 一个 null 回复。这表示 Sentinel 不知道这个主服务器。

如果收到了 ip:port 对，这个地址应该用来连接到 Redis 主服务器。否则，如果收到了一个 null 回复，客户端应该尝试列表中的下一个 Sentinel。

第 3 步：在目标实例中调用 ROLE 命令(call the ROLE command in the target instance)

一旦客户端发现了主服务器实例的地址，就应该尝试与主服务器的连接，然后调用 ROLE 命令来验证实例的角色真的是一个主服务器。

如果 ROLE 命令不可用(Redis 2.8.12 引进的)，客户端可以使用 INFO 复制命令来解析角色：输出中的某一个字段。

如果实例不是期待中的主服务器，客户端应该等待一小段时间(几百毫秒)然后再尝试从第 1 步开始。

处理重连(Handling reconnections)

一旦服务名称被解析为主服务器地址，并且与 Redis 主服务器实例的连接已经建立，每次需要重新连接时，客户端应该重新从第 1 步开始使用 Sentinel 来解析地址。例如，下面的情况下需要重新联系 Sentinel：

- 如果客户端在超时或者 socket 错误后重连。
- 如果客户端因为被显式关闭或者被用户重连而重连。

在上面的情况下或者任何客户端丢失了与 Redis 服务器连接的情况下，客户端应该再次解析主服务器地址。

Sentinel 故障转移断开(Sentinel failover disconnection)

从 Redis 2.8.12 开始，当 Redis Sentinel 改变了实例的配置，例如，提升从服务器为主服务器，故障转移后降级主服务器来复制新的主服务器，或者只是改变一个旧的(stale)从服务器的主服务器地址，会发送一个 CLIENT KILL 类型的命令给实例，来确保所有的客户端都与重新配置过的实例断开。这会强制客户端再次解析主服务器地址。

如果客户端要联系一个还未更新信息的 Sentinel，通过 ROLE 命令验证 Redis 实例角色会失败，允许客户端发现联系上的 Sentinel 提供了旧的(stale)信息，然后会重试。

注意：一个旧的主服务器返回在线的同时，客户端联系一个旧的 Sentinel 实例是有可能的，所以客户端可能连接了一个旧的主服务器，然而 ROLE 的输出也是匹配的。但是，当主服务器恢复回来以后，Sentinel 将会尝试将其降级为从服务器，触发一次新的断开。这个逻辑也适用于连接到一个旧的从服务器，其会被重新配置来复制一个不同的主服务器。

连接从服务器(Connecting to slaves)

有时候客户端有兴趣连接到从服务器，例如，为了分离(scale)读请求。简单修改一下第 2 步就可以支持连接从服务器。不是调用下面的命令：

```
SENTINEL get-master-addr-by-name master-name
```

客户端应该调用：

```
SENTINEL slaves master-name
```

用于检索从服务器实例的清单。

相应地，客户端应该使用 ROLE 命令来验证实例真的是一个从服务器，以防止分离读请求到主服务器。

连接池(Connection pools)

对于实现了连接池的客户端，当单个连接重连时，应该要再次联系 Sentinel，如果是主服务器的地址改变了，所有已经存在的连接都要关闭并且重新连接到新的地址。

错误报告(Error reporting)

客户端应该在遇到错误时正确的返回信息给用户，尤其是：

- 如果没有 Sentinel 能够联系上(这样客户端不可能从 SENTINEL get-master-addr-by-name 获得回复)，应该返回明确表明 Redis Sentinel 不可达的错误。
- 如果所有池中的 Sentinel 返回 null 回复，用户必须被通知 Sentinel 不认识这个主服务器名称的错误。

Sentinel 列表自动刷新(Sentinels list automatic refresh)

一旦收到 get-master-addr-by-name 的成功回复，客户端会按照下面的步骤来更新其内部的 Sentinel 节点的列表：

- 使用 SENTINEL sentinels 命令获取这台主服务器的其他 Sentinel 列表。
- 添加每个不在列表中的 ip:port 对到列表的后面。

客户端不需要更新自己的配置文件来持久化列表。更新内存中表示的 Sentinel 列表的能力对改进可靠性已经很有用了。

订阅 Sentinel 事件来改进响应能力(Subscribe to Sentinel events to improve responsiveness)

介绍 Sentinel 的文档中展示了客户端可以使用发布订阅来连接 Sentinel 以订阅 Redis 实例的配置变更。

这种机制可以用来加快客户端的重配置，也就是，客户端可以监听发布订阅，以知道配置变更什么时候发生，从而运行上文解释的三步协议来解析新的 Redis 主服务器(或者从服务器)地址。

但是，通过发布订阅收到的变更消息不能代替上面的步骤，因为不能保证客户端可以收到所有的变更消息。

额外信息(Additional information)

要获得额外信息或者讨论这个指引的特定方面，请发消息到 Redis Google Group。

16

集群（上）

这篇文档是对 Redis 集群的介绍，没有使用复杂难懂的东西来理解分布式系统的概念。本文提供了如何建立，测试和操作一个集群的相关指导，但没有涉及在 Redis 集群规范（参考本系列其他文章，译者注）中的诸多细节，只是从用户的视角来描述系统是如何运作的。

注意，如果你打算来一次认真的 Redis 集群的部署，更正式的规范文档（关注本系列文章，译者注）强烈建议你好好读一读。

Redis 集群当前处于 alpha 阶段，如果你发现任何问题，请联系 Redis 邮件列表，或者在 Redis 的 Github 仓库中开启一个问题（issue）。

Redis 集群（Redis Cluster）

Redis 集群提供一种运行 Redis 的方式，数据被自动的分片到多个 Redis 节点。

集群不支持处理多个键的命令，因为这需要在 Redis 节点间移动数据，使得 Redis 集群不能提供像 Redis 单点那样的性能，在高负载下会表现得不可预知。

Redis 集群也提供在网络分割（partitions）期间的一定程度的可用性，这就是在现实中当一些节点失败或者不能通信时能继续进行运转的能力。

所以，在实践中，你可以从 Redis 集群中得到什么呢?

- 在多个节点间自动拆分你的数据集的能力。
- 当部分节点正在经历失败或者不能与集群其他节点通信时继续运转的能力。

Redis 集群的 TCP 端口（Redis Cluster TCP ports）

每个 Redis 集群节点需要两个 TCP 连接打开。正常的 TCP 端口用来服务客户端，例如 6379，加 10000 的端口用作数据端口，在上面的例子中就是 16379。

第二个大一些的端口用于集群总线（bus），也就是使用二进制协议的点到点通信通道。集群总线被节点用于错误检测，配置更新，故障转移授权等等。客户端不应该尝试连接集群总线端口，而应一直与正常的 Redis 命令端口通信，但是要确保在防火墙中打开了这两个端口，否则 Redis 集群的节点不能相互通信。

命令端口和集群总线端口的偏移量一直固定为 10000。

注意，为了让 Redis 集群工作正常，对每个节点：

1. 用于与客户端通信的正常的客户端通信端口（通常为 6379）需要开放给所有需要连接集群的客户端以及其他集群节点（使用客户端端口来进行键迁移）。
2. 集群总线端口（客户端端口加 10000）必须从所有的其他集群节点可达。

如果你不打开这两个 TCP 端口，你的集群就不会像你期待的那样去工作。

Redis 集群的数据分片（Redis Cluster data sharding）

Redis 集群没有使用一致性哈希，而是另外一种不同的分片形式，每个键概念上是被我们称为哈希槽（hash slot）的东西的一部分。

Redis 集群有 16384 个哈希槽，我们只是使用键的 CRC16 编码对 16384 取模来计算一个指定键所属的哈希槽。

每一个 Redis 集群中的节点都承担一个哈希槽的子集，例如，你可能有一个 3 个节点的集群，其中：

- 节点 A 包含从 0 到 5500 的哈希槽。
- 节点 B 包含从 5501 到 11000 的哈希槽。
- 节点 C 包含从 11001 到 16384 的哈希槽。

这可以让在集群中添加和移除节点非常容易。例如，如果我想添加一个新节点 D，我需要从节点 A，B，C 移动一些哈希槽到节点 D。同样地，如果我想从集群中移除节点 A，我只需要移动 A 的哈希槽到 B 和 C。当节点 A 变成空的以后，我就可以从集群中彻底删除它。

因为从一个节点向另一个节点移动哈希槽并不需要停止操作，所以添加和移除节点，或者改变节点持有的哈希槽百分比，都不需要任何停机时间（downtime）。

Redis 集群的主从模型（Redis Cluster master-slave model）

为了当部分节点失效时，或者无法与大多数节点通信时仍能保持可用，Redis 集群采用每个节点拥有 1（主服务自身）到 N 个副本（N-1 个附加的从服务器）的主从模型。

在我们的例子中，集群拥有 A，B，C 三个节点，如果节点 B 失效集群将不能继续服务，因为我们不再有办法来服务在 5501-11000 范围内的哈希槽。

但是，如果当我们创建集群后（或者稍后），我们为每一个主服务器添加一个从服务器，这样最终的集群就由主服务器 A，B，C 和从服务器 A1，B1，C1 组成，如果 B 节点失效系统仍能继续服务。

B1 节点复制 B 节点，于是集群会选举 B1 节点作为新的主服务器，并继续正确的运转。

Redis 集群的一致性保证（Redis Cluster consistency guarantees）

Redis 集群不保证强一致性。实践中，这意味着在特定的条件下，Redis 集群可能会丢掉一些被系统收到的写入请求命令。

Redis 集群为什么会丢失写请求的第一个原因，是因为采用了异步复制。这意味着在写期间下面的事情发生了：

- 你的客户端向主服务器 B 写入。
- 主服务器 B 回复 OK 给你的客户端。
- 主服务器 B 传播写入操作到其从服务器 B1，B2 和 B3。

你可以看到，B 在回复客户端之前没有等待从 B1，B2，B3 的确认，因为这是一个过高的延迟代价，所以如果你的客户端写入什么东西，B 确认了这个写操作，但是在发送写操作到其从服务器前崩溃了，其中一个从服务器被提升为主服务器，永久性的丢失了这个写操作。

这非常类似于在大多数被配置为每秒刷新数据到磁盘的数据库发生的事情一样，这是一个可以根据以往不包括分布式系统的传统数据库系统的经验来推理的场景。同样的，你可以通过在回复客户端之前强制数据库刷新数据到磁盘来改进一致性，但这通常会极大的降低性能。

基本上，有一个性能和一致性之间的权衡。

注意：未来，Redis 集群在必要时可能或允许用户执行同步写操作。

Redis 集群丢失写操作还有另一个场景，发生在网络分割时，客户端与至少包含一个主服务器的少数实例被孤立起来了。

举个例子，我们的集群由 A，B，C，A1，B1，C1 共 6 个节点组成，3 个主服务器，3 个从服务器。还有一个客户端，我们称为 Z1。

分割发生以后，有可能分割的一侧是 A，C，A1，B1，C1，分割的另一侧是 B 和 Z1。

Z1 仍然可以写入到可接受写请求的 B。如果分割在很短的时间内恢复，集群会正常的继续。但是，如果分割持续了足够的时间，B1 在分割的大多数这一侧被提升为主服务器，Z1 发送给 B 的写请求会丢失。

注意，Z1 发送给 B 的写操作数量有一个最大窗口：如果分割的大多数侧选举一个从服务器为主服务器后过了足够多的时间，少数侧的每一个主服务器节点将停止接受写请求。

这个时间量是 Redis 集群一个非常重要的配置指令，称为节点超时（node timeout）。

节点超时时间过后，主服务器节点被认为失效，可以用其一个副本来取代。同样地，节点超时时间过后，主服务器节点还不能感知其它主服务器节点的大多数，则进入错误状态，并停止接受写请求。

创建和使用 Redis 集群（Creating and using a Redis Cluster）

要创建一个集群，我们要做的第一件事情就是要有若干运行在集群模式下的 Redis 实例。这基本上意味着，集群不是使用正常的 Redis 实例创建的，而是需要配置一种特殊的模式 Redis 实例才会开启集群特定的特性和命令。

下面是最小的 Redis 集群配置文件：

```
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

正如你所看到的，简单的 cluster-enabled 指令开启了集群模式。每个实例包含一个保存这个节点配置的文件的路径，默认是 nodes.conf。这个文件不会被用户接触到，启动时由 Redis 集群实例生成，每次在需要时被更新。

注意，可以正常运转的最小集群需要包含至少 3 个主服务器节点。在你的第一次尝试中，强烈建议开始一个 6 个节点的集群，3 个主服务器，3 个从服务器。

要这么做，先进入一个新的目录，创建下面这些以端口号来命名的目录，我们后面会在每个目录中运行实例。

像这样：

```
mkdir cluster-test
cd cluster-test
mkdir 7000 7001 7002 7003 7004 7005
```

在从 7000 到 7005 的每个目录内创建一个 redis.conf 文件。作为你的配置文件的模板，只使用上面的小例子，但是要确保根据目录名来使用正确的端口号来替换端口号 7000。

现在，复制你从 Github 的不稳定分支的最新的源代码编译出来的 redis-server 可执行文件到 cluster-test 目录中，最后在你喜爱的终端应用程序中打开 6 个终端标签。

像这样在每个标签中启动实例：

```
cd 7000
../redis-server ./redis.conf
```

你可以从每个实例的日志中看到，因为 nodes.conf 文件不存在，每个节点都为自己赋予了一个新 ID。

```
[82462] 26 Nov 11:56:55.329 * No cluster configuration found, I'm 97a3a64667477371c4479320d683e4c8db5858b1
```

这个 ID 会一直被这个实例使用，这样实例就有一个在集群上下文中唯一的名字。每个节点使用这个 ID 来记录每个其它节点，而不是靠 IP 和端口。IP 地址和端口可能会变化，但是唯一的节点标识符在节点的整个生命周期中都不会改变。我们称这个标识符为节点 ID（Node ID）。

创建集群（Creating the cluster）

现在，我们已经有了一些运行中的实例，我们需要创建我们的集群，写一些有意义的配置到节点中。

这很容易完成，因为我们有称为 redis-trib 的 Redis 集群命令行工具来帮忙，这是一个 Ruby 程序，可以在实例上执行特殊的命令来创建一个新的集群，检查或重分片一个已存在的集群，等等。

redis-trib 工具在 Redis 源代码分发版本的 src 目录中。要创建你的集群，简单输入：

```
./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
```

这里使用的命令是 create，因为我们想创建一个新的集群。--replicas 1 选项意思是我们希望每个创建的主服务器有一个从服务器。其他参数是我想用来创建新集群的实例地址列表。

显然，我们要求的唯一布局就是创建一个拥有 3 个主服务器和 3 个从服务器的集群。

Redis-trib 会建议你一个配置。输入 yes 接受。集群会被配置和连接在一起，也就是说，实例会被引导为互相之间对话。最后，如果一切顺利你会看到一个类似这样的消息：

```
[OK] All 16384 slots covered
```

这表示，16384 个槽中的每一个至少有一个主服务器在处理。

与集群共舞（Playing with the cluste）

在当前阶段，Redis 集群的一个问题是缺少客户端库的实现。

据我所知有以下实现：

- redis-rb-cluster 是我（@antirez）写的 Ruby 实现，作为其他语言的参考。这个是对原先的
- redis-rb 进行了简单的封装，实现了与集群高效对话的最小语义。
- redis-py-cluster 看起来就是 redis-rb-cluster 的 Python 版本。最新没有更新（最后一次提交是 6 个月之前）但是这是一个起点。
- 流行的 Predis 有对 Redis 集群的支持，支持最近有更新，并处于活跃开发状态。
- 最多使用的 Java 客户端 Jedis 最近增加了对 Redis 集群的支持，请查看项目 README 中的 Jedis 集群部分。
- StackExchange.Redis 提供对 C#的支持（应该与大多数.NET 语言工作正常：VB，F#等）。
- Github 上 Redis 仓库的不稳定分支上的 redis-cli 工具实现了一个基本的集群支持，使用-c 启动时切换。

测试 Redis 集群的简单办法就是尝试上面这些客户端，或者只是使用 redis-cli 命令行工具。下面的交互例子使用的是后者：

```
$ redis-cli -c -p 7000
redis 127.0.0.1:7000> set foo bar
-> Redirected to slot [12182] located at 127.0.0.1:7002
OK
redis 127.0.0.1:7002> set hello world
-> Redirected to slot [866] located at 127.0.0.1:7000
OK
redis 127.0.0.1:7000> get foo
-> Redirected to slot [12182] located at 127.0.0.1:7002
"bar"
redis 127.0.0.1:7000> get hello
-> Redirected to slot [866] located at 127.0.0.1:7000
"world"
```

redis-cli 的集群支持非常基本，所以总是依赖 Redis 集群节点重定向客户端到正确的节点。一个真正的客户端可以做得更好，缓存哈希槽和节点地址之间的映射，直接使用到正确节点的正确连接。映射只在集群的配置发生某些变化时才重新刷新，例如，故障转移以后，或者系统管理员通过添加或移除节点改变了集群的布局以后。

17

集群（中）

使用 redis-rb-cluster 写一个示例应用

在后面介绍如何操作 Redis 集群之前，像故障转移或者重新分片这样的事情，我们需要创建一个示例应用，或者至少要了解简单的 Redis 集群客户端的交互语义。

我们采用运行一个示例，同时尝试使节点失效，或者开始重新分片这样的方式，来看看在真实世界条件下 Redis 集群如何表现。如果没有人往集群写的话，观察集群发生了什么也没有什么实际用处。

这一小节通过两个例子来解释 redis-rb-cluster 的基本用法。第一个例子在 redis-rb-cluster 发行版本的 example.rb 文件中，如下：

```
require './cluster'

startup_nodes = [
    {:host => "127.0.0.1", :port => 7000},
    {:host => "127.0.0.1", :port => 7001}
]
rc = RedisCluster.new(startup_nodes,32,:timeout => 0.1)

last = false

while not last
    begin
        last = rc.get("__last__")
        last = 0 if !last
    rescue => e
        puts "error #{e.to_s}"
        sleep 1
    end
end

((last.to_i+1)..1000000000).each{|x|
    begin
        rc.set("foo#{x}",x)
        puts rc.get("foo#{x}")
        rc.set("__last__",x)
    rescue => e
        puts "error #{e.to_s}"
    end
    sleep 0.1
}
```

这个程序做了一件很简单的事情，一个一个地设置形式为 foo<number> 的键的值为一个数字。所以如果你运行这个程序，结果就是下面的命令流：

```
SET foo0 0
SET foo1 1
SET foo2 2
And so forth...
```

这个程序看起来要比通常看起来更复杂，因为这个是设计用来在屏幕上展示错误，而不是由于异常退出，所以每一个对集群执行的操作都被 begin rescue 代码块包围起来。

第 7 行是程序中第一个有意思的地方。创建了 Redis 集群对象，使用启动节点（startup nodes）的列表，对象允许的最大连接数，以及指定操作被认为失效的超时时间作为参数。 启动节点不需要是全部的集群节点。重要的是至少有一个节点可达。也要注意，redis-rb-cluster 一旦连接上了第一个节点就会更新启动节点的列表。你可以从任何真实的客户端中看到这样的行为。

现在，我们将 Redis 集群对象实例保存在 rc 变量中，我们准备像一个正常的 Redis 对象实例一样来使用这个对象。

第 11 至 19 行说的是：当我们重启示例的时候，我们不想又从 foo0 开始，所以我们保存计数到 Redis 里面。上面的代码被设计为读取这个计数值，或者，如果这个计数器不存在，就赋值为 0。

但是，注意这里为什么是个 while 循环，因为我们想即使集群下线并返回错误也要不断地重试。一般的程序不必这么小心谨慎。

第 21 到 30 行开始了主循环，键被设置赋值或者展示错误。

注意循环最后 sleep 调用。在你的测试中，如果你想尽可能快地往集群写入，你可以移除这个 sleep（相对来说，这是一个繁忙的循环而不是真实的并发，所以在最好的条件下通常可以得到每秒 10k 次操作）。

正常情况下，写被放慢了速度，让人可以更容易地跟踪程序的输出。

运行程序产生了如下输出：

```
ruby ./example.rb
1
2
3
4
5
6
7
8
```

```
9
^C (I stopped the program here)
```

这不是一个很有趣的程序，稍后我们会使用一个更有意思的例子，看看在程序运行时进行重新分片会发生什么事情。

重新分片集群（Resharding the cluster）

现在，我们准备尝试集群重分片。要做这个请保持 example.rb 程序在运行中，这样你可以看到是否对运行中的程序有一些影响。你也可能想注释掉 sleep 调用，这样在重分片期间就有一些真实的写负载。

重分片基本上就是从部分节点移动哈希槽到另外一部分节点上去，像创建集群一样也是通过使用 redis-trib 工具来完成。

开启重分片只需要输入：

```
./redis-trib.rb reshard 127.0.0.1:7000
```

你只需要指定单个节点，redis-trib 会自动找到其它节点。

当前 redis-trib 只能在管理员的支持下进行重分片，你不能只是说从这个节点移动 5%的哈希槽到另一个节点（但是这也很容易实现）。那么问题就随之而来了。第一个问题就是你想要重分片多少：

你想移动多少哈希槽（从 1 到 16384）？

我们尝试重新分片 1000 个哈希槽，如果没有 sleep 调用的那个例子程序还在运行的话，这些槽里面应该已经包含了不少的键了。

然后，redis-trib 需要知道重分片的目标了，也就是将接收这些哈希槽的节点。我将使用第一个主服务器节点，也就是 `127.0.0.1:7000`，但是我得指定这个实例的节点 ID。这已经被 redis-trib 打印在一个列表中了，但是我总是可以在需要时使用下面的命令找到节点的 ID：

```
$ redis-cli -p 7000 cluster nodes | grep myself
97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5460
```

好了，我的目标节点是 97a3a64667477371c4479320d683e4c8db5858b1。

现在，你会被询问想从哪些节点获取这些键。我会输入 all，这样就会从所有其它的主服务器节点获取一些哈希槽。

在最后的确认后，你会看到每一个被 redis-trib 准备从一个节点移动到另一个节点的槽的消息，并且会为每一个被从一侧移动到另一侧的真实的键打印一个圆点。

在重分片进行的过程中，你应该能够看到你的示例程序运行没有受到影响。如果你愿意的话，你可以在重分片期间多次停止和重启它。

在重分片的最后，你可以使用下面的命令来测试一下集群的健康情况：

```
./redis-trib.rb check 127.0.0.1:7000
```

像平时一样，所有的槽都会被覆盖到，但是这次在 `127.0.0.1:7000` 的主服务器会拥有更多的哈希槽，大约 6461 个左右。

一个更有意思的示例程序

到目前为止一切挺好，但是我们使用的示例程序却不够好。不顾后果地（acritically）往集群里面写，而不检查写入的东西是否是正确的。

从我们的观点看，接收写请求的集群可能一直将每个操作都作为设置键 foo 值为 42，我们却根本没有察觉到。

所以在 redis-rb-cluster 仓库中，有一个叫做 consistency-test.rb 的更有趣的程序。这个程序有意思得多，因为它使用一组计数器，默认 1000 个，发送 INCR 命令来增加这些计数器。

但是，除了写入，程序还做另外两件事情：

- 当计数器使用 INCR 被更新后，程序记住了写操作。
- 在每次写之前读取一个随机计数器，检查这个值是否是期待的值，与其在内存中的值比较。

这个的意思就是，这个程序就是一个一致性检查器，可以告诉你集群是否丢失了一些写操作，或者是否接受了一个我们没有收到确认（acknowledgement）的写操作。在第一种情况下，我们会看到计数器的值小于我们记录的值，而在第二种情况下，这个值会大于。

运行 consistency-test 程序每秒钟产生一行输出：

```
$ ruby consistency-test.rb
925 R (0 err) | 925 W (0 err) |
5030 R (0 err) | 5030 W (0 err) |
9261 R (0 err) | 9261 W (0 err) |
13517 R (0 err) | 13517 W (0 err) |
17780 R (0 err) | 17780 W (0 err) |
22025 R (0 err) | 22025 W (0 err) |
25818 R (0 err) | 25818 W (0 err) |
```

每一行展示了执行的读操作和写操作的次数，以及错误数（错误导致的未被接受的查询是因为系统不可用）。

如果发现了不一致性，输出将增加一些新行。例如，当我在程序运行期间手工重置计数器，就会发生：

```
$ redis 127.0.0.1:7000> set key_217 0
OK

(in the other tab I see...)

94774 R (0 err) | 94774 W (0 err) |
98821 R (0 err) | 98821 W (0 err) |
```

```
102886 R (0 err) | 102886 W (0 err) | 114 lost |
107046 R (0 err) | 107046 W (0 err) | 114 lost |
```

当我把计数器设置为 0 时，真实值是 144，所以程序报告了 144 个写操作丢失（集群没有记住的 INCR 命令执行的次数）。

这个程序作为测试用例很有意思，所以我们会使用它来测试 Redis 集群的故障转移。

测试故障转移（Testing the failover）

注意：在测试期间，你应该打开一个标签窗口，一致性检查的程序在其中运行。

为了触发故障转移，我们可以做的最简单的事情（这也是能发生在分布式系统中语义上最简单的失败）就是让一个进程崩溃，在我们的例子中就是一个主服务器。

我们可以使用下面的命令来识别一个集群并让其崩溃：

```
$ redis-cli -p 7000 cluster nodes | grep master
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385482984082 0 connected 5960-10921
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 master - 0 1385482983582 0 connected 11423-16383
97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-11422
```

好了，7000，7001，7002 都是主服务器。我们使用 DEBUG SEGFAULT 命令来使节点 7002 崩溃：

```
$ redis-cli -p 7002 debug segfault
Error: Server closed the connection
```

现在，我们可以看看一致性测试的输出报告了些什么内容。

```
18849 R (0 err) | 18849 W (0 err) |
23151 R (0 err) | 23151 W (0 err) |
27302 R (0 err) | 27302 W (0 err) |

... many error warnings here ...

29659 R (578 err) | 29660 W (577 err) |
33749 R (578 err) | 33750 W (577 err) |
37918 R (578 err) | 37919 W (577 err) |
42077 R (578 err) | 42078 W (577 err) |
```

你可以看到，在故障转移期间，系统不能接受 578 个读请求和 577 个写请求，但是数据库中没有产生不一致性。这听起来好像和我们在这篇教程的第一部分中陈述的不一样，我们说道，Redis 集群在故障转移期间会丢失写操作，因为它使用异步复制。但是我们没有说过的是，这并不是经常发生，因为 Redis 发送回复给客户端，和发送复制命令给从服务器差不多是同时，所以只有一个很小的丢失数据窗口。但是，很难触发并不意味着不可能发生，所以这并没有改变 Redis 集群提供的一致性保证（即非强一致性，译者注）。

我们现在可以看看故障转移后的集群布局（注意，与此同时，我重启了崩溃的实例，所以它以从服务器的身份重新加入了集群）：

```
$ redis-cli -p 7000 cluster nodes
3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 0
a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64667477371c4479320d683e4c8db5858b1 0
97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-11422
3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 1385503419023 3 connected 11423-16383
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385503417005 0 connected 5960-10921
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0
```

现在，主服务器运行在 7000，7001 和 7005 端口。之前运行在 7002 端口的主服务器现在是 7005 的从服务器了。

CLUSTER NODES 命令的输出看起来挺可怕的，但是实际上相当的简单，由以下部分组成：

- 节点 ID
- ip:port
- flags: master, slave, myself, fail, ...
- 如果是从服务器的话，就是其主服务器的节点 ID
- 最近一次发送 PING 后等待回复的时间
- 最近一次发送 PONG 的时间
- 节点的配置纪元（请看集群规范）
- 节点的连接状态
- 服务的哈希槽

18

集群（下）

手动故障转移（Manual failover）

有时候在主服务器事实上没有任何故障的情况下强制一次故障转移是很有用的。例如，为了升级主服务器节点中的一个进程，可以对其进行故障转移使其变为一个从服务器，这样最小化了对可用性的影响。

Redis 集群支持使用 CLUSTER FAILOVER 命令来手动故障转移，必须在你想进行故障转移的主服务的其中一个从服务器上执行。

手动故障转移很特别，和真正因为主服务器失效而产生的故障转移要更安全，因为采取了避免过程中数据丢失的方式，仅当系统确认新的主服务器处理完了旧的主服务器的复制流时，客户端才从原主服务器切换到新主服务器。

下面是当你手动故障转移时你从从服务器日志中看到的内容：

```
# Manual failover user request accepted.

# Received replication offset for paused master manual failover: 347540

# All master replication stream processed, manual failover can start.

# Start of election delayed for 0 milliseconds (rank #0, offset 347540).

# Starting a failover election for epoch 7545.

# Failover election won: I'm the new master.
```

基本上，连接到我们正在故障转移的主服务器的客户端停止了。与此同时，主服务器发送复制偏移量给从服务器，等待到达这个偏移量。当复制偏移量到达以后，故障转移就开始了，旧的主服务器被通知切换配置。当客户端在旧主服务器上解除阻塞时，就被重定向到新的主服务器。

添加新节点（Adding a new node）

添加一个新节点的过程基本上就是，添加一个空节点，然后，如果是作为主节点则移动一些数据进去，如果是从节点则其作为某个节点的副本。

两种情况我们都会讨论，先从添加一个新的主服务器实例开始。

两种情况下，第一步要完成的都是添加一个空节点。

我们使用与其他节点相同的配置（端口号除外）在 7006 端口（我们已存在的 6 个节点已经使用了从 7000 到 7005 的端口）上开启一个新的节点，那么为了与我们之前的节点布局一致，你得这么做：

- 在你的终端程序中开启一个新的标签窗口。
- 进入 cluster-test 目录。
- 创建一个名为 7006 的目录。
- 在里面创建一个 redis.conf 的文件，类似于其它节点使用的文件，但是使用 7006 作为端口号。
- 最后使用../redis-server ./redis.conf 启动服务器。

此时服务器已经在运行中了。

现在我们可以像通常一样使用 redis-trib 来添加节点到已存在的集群中。

```
./redis-trib.rb add-node 127.0.0.1:7006 127.0.0.1:7000
```

你可以看到，我使用了 addnode 命令，指定新的节点地址为第一个参数，集群中一个随机存在的节点的地址作为第二个参数。

实际上 redis-trib 在这里对我们只有很少的帮助，只是发送了一个 CLUSTER MEET 消息到节点，这些也可以手动完成。但是 redis-trib 也在操作之前检查了集群的状态，所以即便你知道内部是如何工作的，一直通过 redis-trib 来执行集群操作也是一个不错的主意。

现在，我们可以连接到这个新的节点，看看它是否真的加入到了集群中：

```
redis 127.0.0.1:7006> cluster nodes
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385543178575 0 connected 5960-10921
3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 0
f093c80dde814da99c5cf72a7dd01590792b783b :0 myself,master - 0 0 0 connected
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0
a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64667477371c4479320d683e4c8db5858b1 0
```

```
97a3a64667477371c4479320d683e4c8db5858b1 127.0.0.1:7000 master - 0 1385543179080 0 connected 0-5959 10922-
3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 1385543177568 3 connected 11423-16383
```

注意，因为这个节点已经连接到集群了，所以也已经可以正确地重定向客户端查询，简而言之，这个节点已经是集群的一部分了。但是它比其他主节点有两个特殊之处：

- 因为没有分配哈希槽所以没有数据。
- 因为这个主服务器没有分配哈希槽，所以当有从服务器要变成主服务器时不能参与选举过程。

现在可以使用 redis-trib 的重新分片特性来给这个节点赋予哈希槽了。基本上没有必现展示这个了，因为我们已经在之前的小节中展示过了，没有什么不同，只是以空节点为目标的一次重分片。

添加副本节点（Adding a new node as a replica）

添加一个新副本可以有两种方式。显而易见的一种方式是再次使用 redis-trib，但是要使用—slave 选项，像这样：

```
./redis-trib.rb add-node --slave 127.0.0.1:7006 127.0.0.1:7000
```

注意，这里的命令行完全像我们在添加一个新主服务器时使用的一样，所以我们没有指定要给哪个主服务器添加副本。这种情况下，redis-trib 会添加一个新节点作为一个具有较少副本的随机的主服务器的副本。

但是，你可以使用下面的命令行精确地指定你想要的主服务器作为副本的目标：

```
./redis-trib.rb add-node --slave --master-id 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7006 127.0.0.1:7
```

这样我们就把一个新的副本赋予了一个指定的主服务器。

一种更手工的给指定主服务器添加副本的方式，是添加一个新节点作为一个空主服务器，然后使用 CLUSTER REPLICATE 命令将其变为副本。如果节点被作为从服务器添加，但是你想移动它为另一个不同的主服务器的副本，这也是可行的。

例如，为了给节点 127.0.0.1:7005 添加一个副本，这个节点当前服务 11432-16383 范围内的哈希槽，其节点 ID 为 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e，所有我们需要去做的，就是连接这个新的节点（已经作为空主服务器被添加）然后发送命令：

```
redis 127.0.0.1:7006> cluster replicate 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e
```

就是这样。现在我们有了这组哈希槽的一个新副本了，集群中的其它节点也已经知道了（需要几秒钟来更新配置）。我们可以用下面的命令来核实一下：

```
$ redis-cli -p 7000 cluster nodes | grep slave | grep 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e
f093c80dde814da99c5cf72a7dd01590792b783b 127.0.0.1:7006 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0
```

这个 3c3a0c…的节点现在有两个从服务器了，分别运行于 7002（已存在的）和 7006（新的）端口。

移除节点（Removing a node）

要移除一个从服务器节点，只要使用 redis-trib 的 del-node 命令就可以：

```
./redis-trib del-node 127.0.0.1:7000 <node-id>
```

第一个参数只是集群中的一个随机节点，第二个参数是你想移除的节点的 ID。

你也可以用同样的方式移除一个主服务器节点，但是，为了移除一个主服务器节点，它必须是空的。如果主服务器不是空的，你需要先将其数据重分片到其他的主服务器节点。

另一种移除主服务器节点的方式，就是在其从服务器上执行一次手工故障转移，当它变为了新的主服务器的从服务器以后将其移除。显然，当你需要真的减少你的集群中的主服务器的数量时这个没有什么帮助，如果那样的话，就需要重新分片了。

副本迁移（Replicas migration）

在 Redis 集群中，可以使用下面的命令在任何时候重新配置一个从服务器复制一个不同的主服务器：

```
CLUSTER REPLICATE <master-node-id>
```

但是有一种特殊的场景，你想让副本集自动地从一个主服务器移动到另一个主服务器，而不需要系统管理员的帮助。自动重新配置副本集被称为副本集迁移，这可以改善 Redis 集群的可靠性。

注意：你可以在 Redis 集群规范中阅读到副本集迁移的细节，这里我们只提供一般性的信息，以及为了从中受益你该做什么。

为什么你想让你的集群副本在某些特定条件下从一个主服务器移动到另一个的原因，是通常情况下 Redis 集群对失败的抵御能力和连接到指定从服务器的副本数量成正相关。

例如，每个主服务器只有单个副本组成的集群在主服务器及其副本同时失效时就不能够继续运转，因为没有其他的实例拥有这台主服务器服务的哈希槽的副本。但是，网络断裂可能会在同一时间隔绝若干节点，其他类型的故障，例如单个节点的硬件或者软件错误，是值得注意的一类故障，很可能会同时发生，所以在每个主服务器拥有一个从服务器的集群中，有可能从服务器在下午 4 点被干掉，而主服务器在下午 6 点被干掉。这仍然会导致集群不再能运转。

为了改进系统的可靠性，我们有一些增加额外副本集到每个主服务器的选项，但是这代价昂贵。副本迁移允许添加更多的从服务器到少许的主服务器。所以你可以有 10 个主服务器，每个有一个从服务器，总共 20 个实例。但是你为某些主服务器添加例如 3 个以上的实例，那么有些主服务器会有多余一个的从服务器。

有了副本集迁移会发生什么？如果一个主服务器没有从服务器，一个来自于拥有多个从服务器的主服务器上的从服务器会迁移到这个孤独的主服务器上。所以像上面我们举的例子中，在你的从服务器下午 4 点下线以后，另一个从服务器会接替它的位置，当主服务器在下午 5 点也失效的时候，仍然有一个从服务器可以被选举，这样集群就可以继续运转了。

那么，简而言之，你应该了解副本集迁移的什么呢？

- 集群会尝试从在某一个指定时刻拥有最多数量副本集的主服务上迁移一个副本。
- 为了从副本迁移中受益，你需要在集群中添加多一些的副本到单个主服务器，无论是什么主服务器。
- 有一个称为 replica-migration-barrier 的控制副本迁移特性的配置参数。你可以在 Redis 群提供的示例 redis.conf 文件中读到更多信息。

升级节点（Upgrading nodes in a Redis Cluster）

升级从服务器节点很简单，因为你只需要停止节点然后用已更新的 Redis 版本重启。如果有客户端使用从服务器节点分离读请求，它们应该能够在某个节点不可用时重新连接另一个从服务器。

升级主服务器要稍微复杂一些，建议的步骤是：

1. 使用 CLUSTER FAILOVER 来触发一次手工故障转移主服务器（请看本文档的手工故障转移小节）。
2. 等待主服务器变为从服务器。
3. 像升级从服务器那样升级这个节点。
4. 如果你想让你刚刚升级的节点成为主服务器，触发一次新的手工故障转移，让升级的节点重新变回主服务器。

你可以按照这些步骤来一个节点一个节点的升级，直到全部节点升级完毕。

迁移到 Redis 集群（Migrating to Redis Cluster）

想迁移到 Redis 集群的用户可能只有一个单一的主服务器，或者已经使用了已存在的分片布局，通过使用某种内部算法，或者他们的客户端库实现的分片算法，或者 Redis 代理，键被分拆到 N 个节点上。

这两种情况下迁移到 Redis 集群都很简单，但是最重要的细节是，如果程序使用了多键操作，怎么办。有三种不同的情况：

1. 没有使用多键操作，或者事务，或者涉及多个键的 Lua 脚本。键被独立地访问（即使通过事务或者 Lua 脚本组合针对同样的键的多个命令一起来访问）。
2. 使用了多键操作，或者事务，或者涉及多个键的 Lua 脚本，但是键都有相同的哈希标签（hash tag），也就是说这些一起使用的键都碰巧有相同的{…}子串。例如，下面的多键操作是在相同的哈希标签上下文中定义的：SUNION {user:1000}.foo {user:1000}.bar。
3. 使用了多键操作，或者事务，或者涉及多个键的 Lua 脚本，但是键的名字没有一个显式的或者相同的哈希标签。

Redis 不处理第三种情况：应用程序需要被修改为不能使用多键操作，或者只能在相同的哈希标签上下文中使用。

前两种情况覆盖到了，所以我们会聚焦在这两种情况，它们会用相同的方式来处理，所以本文不会去区别对待。

假设你的已存在数据集已经被拆分到了 N 个主服务器上，如果你没有已存在的分片的话 N=1，你需要下面的步骤来迁移你的数据集到 Redis 集群：

1. 停止你的客户端。当前没有自动在线迁移（live-migration）到 Redis 集群的可能。你也许可以通过精心策划一次在你的程序或环境上下文中的在线迁移来办到。
2. 使用 BGREWRITEOF 命令为所有你的 N 个主服务器生成一个追加文件，然后等待 AOF 文件完全生成。
3. 按照 aof-1 到 af-N 保存你的 AOF 文件到某处。此时愿意的话你可以停掉你的旧实例（这很有用，因为在非虚拟化的部署中，你常常需要重用这些计算机）。
4. 创建一个由 N 个主服务器和 0 个从服务器组成的 Redis 集群。你可以稍后添加从服务器。确保所有你的节点都是使用追加文件来持久化。
5. 停止所有的集群节点，用你已存在的追加文件替换他们的朱家文件，aof-1 替换第一节点，aof-2 替换第二个节点，一直到 aof-N。
6. 使用新的 AOF 文件来重启你的 Redis 集群。它们会抱怨按照配置有些键不应该出现。
7. 使用 redis-trib fix 命令来修正集群，这样键就会根据每个节点的哈希槽被迁移了。

8. 最后使用 redis-trib check 来确保集群是正常的。
9. 重启被修改为支持 Redis 集群的客户端。

还有一个方式从外部实例导入数据到 Redis 集群，就是使用 redis-trib import 命令。

这个命令移动一个运行实例（同时删除源实例上的键）上的所有键到一个指定已存在的 Redis 集群。但是，注意如果你使用 Redis 2.8 实例作为来源实例，操作可能很慢，因为 2.8 没有实现迁移连接缓存（migrate connection caching），所以在执行这个操作之前，你可能得重启你的 Redis 3.x 版本的源实例。