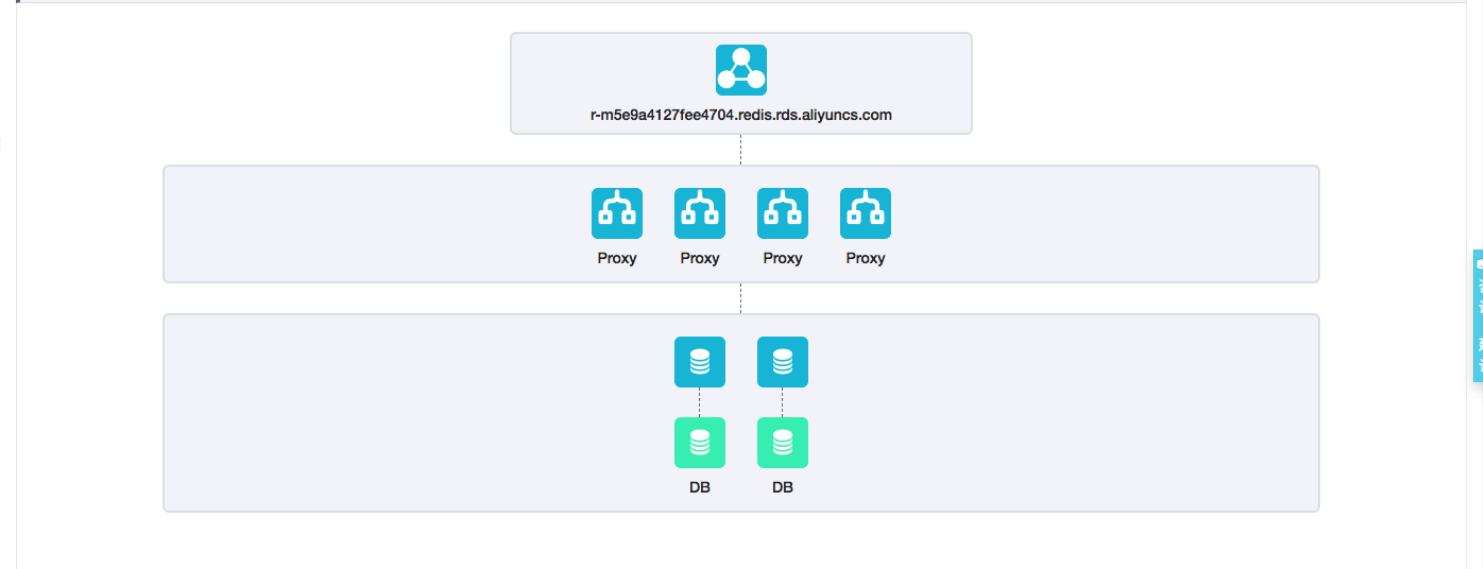


Redis 分享

版本: Redis 4.0	Lua状态:	最大连接数 20000	云命令 (Cloud Shell) X
---------------	--------	-------------	---------------------

连接信息		Redis连接问题排查文档	修改连接地址	关闭免密访问
内网连接地址（host）： r-m5e9a4127fee4704.redis.rds.aliyuncs.com		端口号（port） 6379		
连接密码： 				
温馨提示： 请使用以上访问连接串进行实例连接，VIP在业务维护中可能会变化。				

实例架构图 (集群模式)



Redis 的诞生

Redis 的创建者

Salvatore Sanfilippo (antirez), 男, 意大利人, 出生并居住在西西里岛, 个人网站 <http://invece.org/>。

早年为系统管理员, 关注计算机安全领域, 于 1999 年发明了 idle scan 扫描技术, 该技术现在在 nmap 扫描器上也有实现。

2004 年~2006 年期间在做嵌入式方面的工作, 并为此写了名为 Jim 的 Tcl 解释器、《Tcl Wise: Guide to the Tcl programming language》一书以及《Tcl the Misunderstood》文档。(Redis 的事件处理器就重写自 Jim 的事件循环, 而 Redis 的测试套件也使用 Tcl 语言来写的)。除此之外, 他在 2006 还写了 Hping —— 一个 TCP/IP 包分析器。

之后开始接触 web, 在 2007 年和另一个朋友共同创建了 LLOOGG.com, 并因为解决这个网站的负载问题而在 2009 年 2 月 26 日发明了 Redis。



LLOOGG.com

一个访客信息追踪网站, 网站可以通过 JavaScript 脚本, 将访客的 IP 地址、所属国家、浏览器信息、被访问页面的地址等数据传送给 LLOOGG.com 。

然后 LLOOGG.com 会将这些浏览数据通过 web 页面实时地展示给用户, 并储存起最新的 5 至 10,000 条浏览记录以便进行查阅。

Redis 的诞生

为了在不升级 VPS 的前提下，解决 LLOOGG.com 的负载问题，antirez 决定自己写一个具有列表结构的内存数据库原型(prototype)。

这个数据库原型支持 $O(1)$ 复杂的推入和弹出操作，并且将数据储存在内存而不是硬盘，所以程序的性能不会受到硬盘 I/O 限制，可以以极快的速度执行针对列表的推入和弹出操作。

于是 antirez 使用 C 语言重写了这个内存数据库，并给它加上了持久化功能，Redis 就此诞生！

Redis 的演进

经过五年时间的演进, Redis 发生了以下变化.....

刚开始	现在
只支持列表结构	支持字符串、列表、散列等六种结构, 以及丰富的附加功能
只能单机运行, 没有内置的方法可以方便地将数据库分布到多台机器上	支持多机运行 (包括复制、自动故障转移以及分布式数据库)
少有人知的开源项目	广为人知并广泛使用在很多大型网站的热门开源项目
接受捐款支持, antirez 自己无偿开发	Pivotal 公司出资支持开发, 并且有非常多的开发者通过 GitHub 和论坛为这个项目添砖加瓦

独特的键值对模型

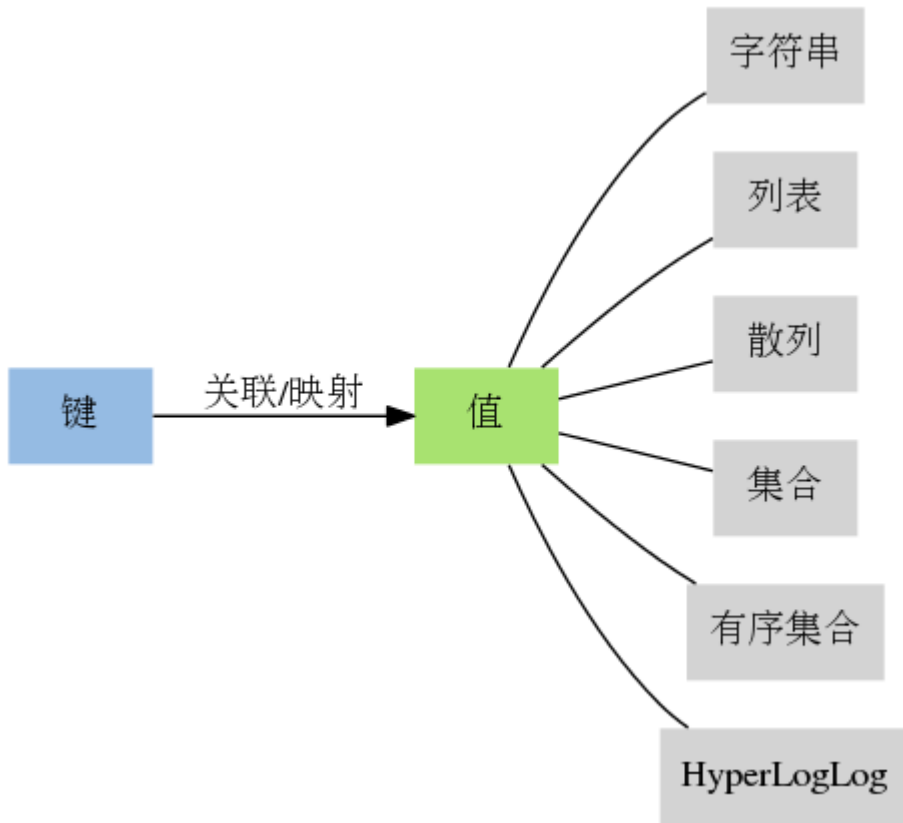
很多数据库只能处理一种数据结构：

- SQL 数据库—— 表格
- Memcached —— 键值对数据库，键和值都是字符串
- 文档数据库(CouchDB、MongoDB) —— 由 JSON/BSON 组成的文档(document)

而一旦数据库提供的数据结构不适合去做某件事的话，程序写起来就会非常地麻烦和不自然。

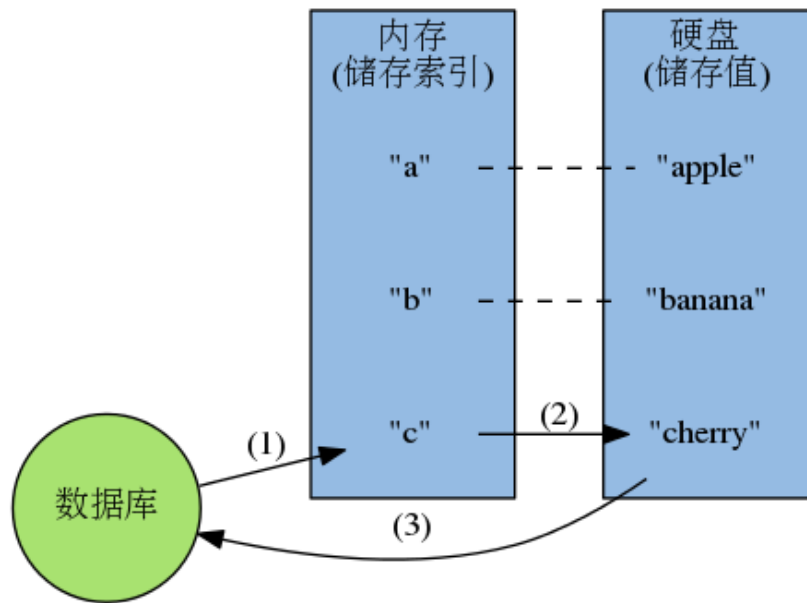
Redis 也是键值对数据库，但和 Memcached 不同的是，Redis 的值不仅可以是字符串，它还可以其他五种数据结构中的任意一种。

通过选用不同的数据结构，用户可以使用 Redis 解决各式各样的问题。

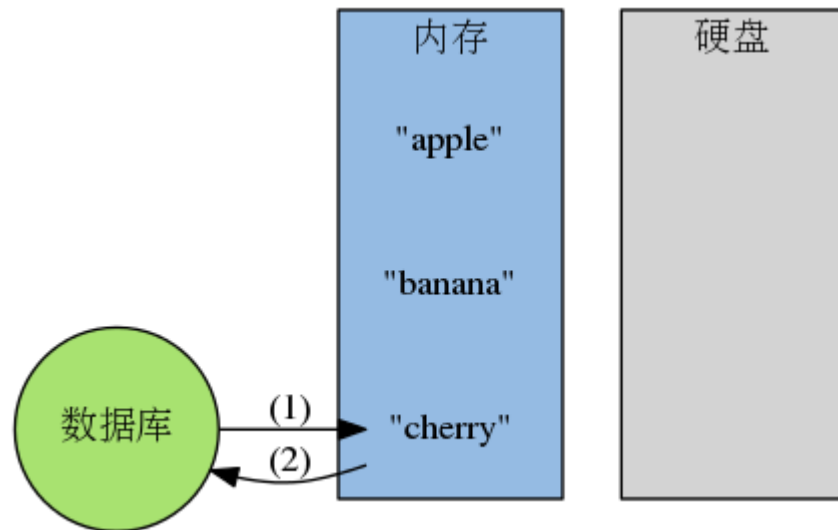


内存储存，速度极快

Redis 将数据储存在内存里面，读写数据的时候都不会受到硬盘 I/O 速度的限制，所以速度极快。



硬盘数据库的工作模式。



内存数据库数据库的工作模式。

丰富的附加功能

持久化功能:将储存在内存里面的数据保存到硬盘里面,保障数据安全,方便进行数据备份和恢复。

发布与订阅功能:将消息同时分发给多个客户端,用于构建广播系统。

过期键功能:为键设置一个过期时间,让它在指定的时间之后自动被删除。

事务功能:原子地执行多个操作,并提供乐观锁功能,保证处理数据时的安全性。

脚本功能:在服务器端原子地执行多个操作,完成复杂的功能,并减少客户端与服务器之间的通信往返次数。


复制:为指定的 Redis 服务器创建一个或多个复制品,用于提升数据安全性,并分担读请求的负载。Sentinel:

监控 Redis 服务器的状态,并在服务器发生故障时,进行自动故障转移。

集群:创建分布式数据库,每个服务器分别执行一部分写操作和读操作。

完善的文档

Redis 具有完善、易读的文档, 加上 Redis 本身功能的简单性, 就算是新手也可以轻松上手。

 **redis**

Commands

Clients

Documentation

Community

Download

Issues

Support

License

All

Keys

Strings

Hashes

Lists

Sets

Sorted Sets

HyperLogLog

Pub/Sub

Transactions

Scripting

Connection

Server

APPEND key value
Append a value to a key

GETBIT key offset
Returns the bit value at offset in the string value stored at key

PEXPIREAT key milliseconds-timestamp
Set the expiration for a key as a UNIX timestamp specified in milliseconds

SMEMBERS key
Get all the members in a set

AUTH password
Authenticate to the server

GETRANGE key start end
Get a substring of the string stored at a key

PFADD key element [element ...]
Adds the specified elements to the specified HyperLogLog.

SMOVE source destination member
Move a member from one set to another

BGREWRITEAOF
Asynchronously rewrite the append-only file

GETSET key value
Set the string value of a key and return its old value

PFCOUNT key [key ...]
Return the approximated cardinality of the set(s) observed by the HyperLogLog at key(s).

SORT key [BY pattern] [LIMIT offset count] [GET pattern [...]]
Sort the elements in a list, set or sorted set

BGSAVE
Asynchronously save the dataset to disk

HDEL key field [field ...]
Delete one or more hash fields

PFMERGE destkey sourcekey [sourcekey ...]
Merge N different HyperLogLogs into a single one.

SPOP key
Remove and return a random member from a set

BITCOUNT key start end [start end ...]
Count set bits in a string

HEXISTS key field
Determine if a hash field exists

PING
Ping the server

SRANDMEMBER key [count]
Get one or multiple random members from a set

BITOP operation destkey key [key ...]
Perform bitwise operations between strings

HGET key field
Get the value of a hash field

PSETEX key milliseconds value
Set the value and expiration in milliseconds of a key

SREM key member [member ...]
Remove one or more members from a set

BITPOS key bit [start] [end]
Find first bit set or clear in a string

HGETALL key
Get all the fields and values in a hash

PSUBSCRIBE pattern [pattern ...]
Listen for messages published to channels matching the given patterns

STRLEN key
Get the length of the value stored in a key

良好的支持

antirez 非常勤奋, 在每个版本都会不断地增加有用的新功能:

- 2.6 新增脚本功能, 并为很多命令添加了多参数支持(比如 SADD、ZADD、等等);
- 2.8 添加了数据库通知功能, HyperLogLog 数据结构以及 SCAN 命令, 实现了部分重同步;
- 3.0 将推出稳定版的 Redis 集群, 另外还有更多新功能陆续开发中.....
- 4.0 Redis 现在的 Stable 版本是4.0.11, 4.0引入了 Module
- 5.0.4 最新稳定版本 <https://redis.io/modules>

Bug 一旦出现就会很快被修复, 齐全测试套件以及稳扎稳打的开发策略, 使得 Redis 非常健壮可靠。

有问题时, 在 Redis 的论坛上发贴, 或者到 Redis 的 GitHub 页面发 issue, 又或者直接和作者 antirez 联系, 通常都可以很快得到回 应。

Pivotal 公司雇用 antirez 全力开发Redis, 无后顾之忧;这间公司也提供专门的 Redis 开发和维护咨询服

阿里云、百度云、Amazon、RedisLab 等公司都提供了基于 Redis 的应用服务。

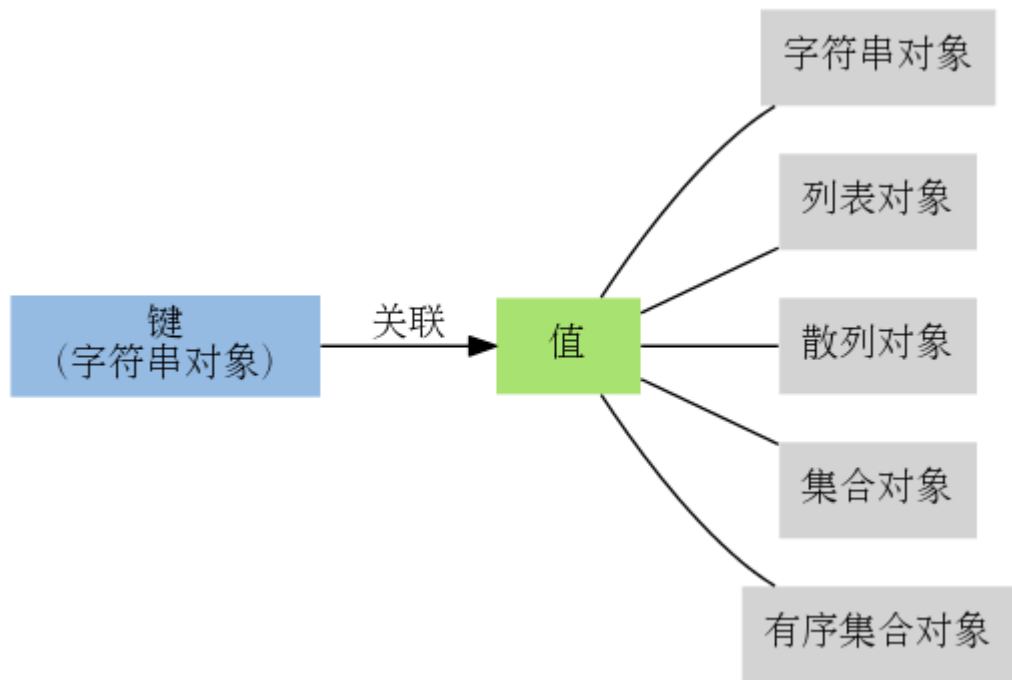
广泛的使用

1. Twitter 使用 Redis 来储存用户时间线(user timeline)
2. StackOverflow 使用 Redis 来进行缓存消息分发。
Pinterest 使用 Redis 来构建关注模型(follow model)和兴趣图谱(interest graph)。
3. Flickr 使用 Redis 来构建队列。
4. Github 使用 Redis 作为持久化的键值对数据库, 并使用 Resque 来实现消息队列。
5. 新浪微博使用 Redis 来实现计数器、反向索引、排行榜、消息队列, 并储存用户关系。
6. 知乎使用 Redis 来进行计数、缓存、消息分发和任务调度。

Redis 实现原理简介

<https://blog.csdn.net/snakorse/article/details/78154402>

对象关系图



对象的实现

Redis 中的每种对象都有与之相对应的底层数据结构，并且为了让对象在各种不同的应用场景下都有优秀的性能表现，Redis 为每种对象都提供了两种类型的数据结构实现：

- 第一类是为了进行性能优化而特制的**编码数据结构** (encoded data structure)，这些数据结构主要使用“以 CPU 换内存的方式”来节约内存。
编码数据结构主要在对象包含的值数量比较少、或者值的体积比较小时使用：比如在字符串对象包含的字符串比较短时，集合只包含三五个小元素 时，又或者列表只包含十来个 项时，这些情况都可以使用编码数据结构。
- 另一类是**普通数据结构**，也即是我们在书上、论文上看见的数据结构，比如双向链表、字典、跳跃 表等等。比起编码数据结构，这些普通的数据结构需要消耗更多内存，但是能够提供更强大的功能，普通数据结构主要在对象包含的值比较多、或者值的体积比较大时使用。

在创建新对象时，Redis 会优先使用编码数据结构来表示对象，并在有需要时，自动将对象的表示方式转换为普通数据结构。

数据结构

编码数据结构：

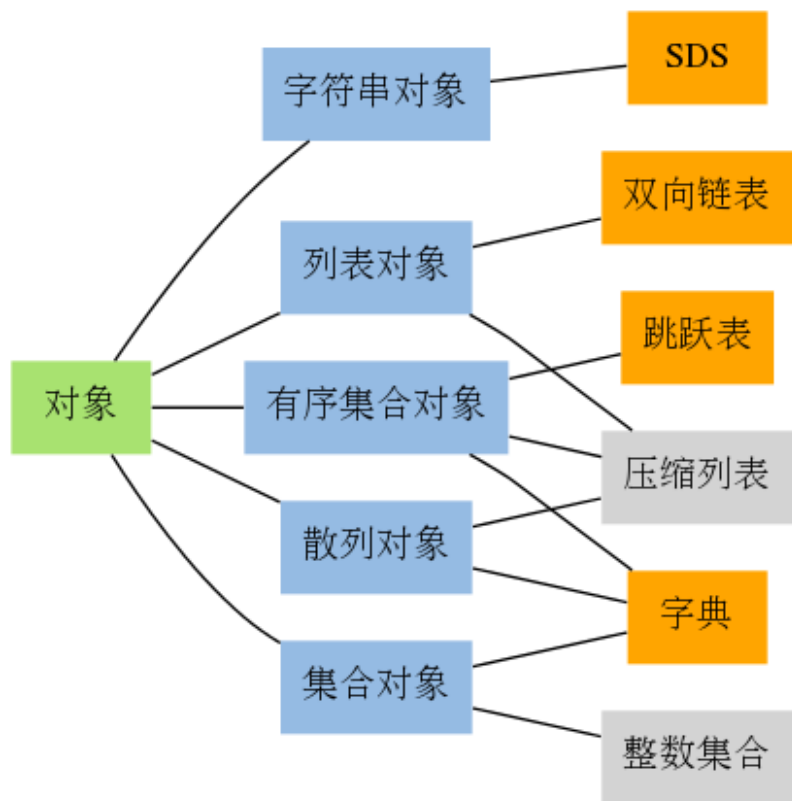
- 压缩列表(zip list)
- 整数集合(int set)

普通数据结构：

- 简单动态字符串(SDS, simple dynamic string)
- 双向链表
- 字典
- 跳跃表

另外需要注意的是，节约内存并不是编码数据结构所特有的——普通数据结构也有相应的内存优化模式，比如 SDS 就有两种或以上的表示方式，不同的表示可以让同一对象在不同的使用场景下也尽可能地保持高效。

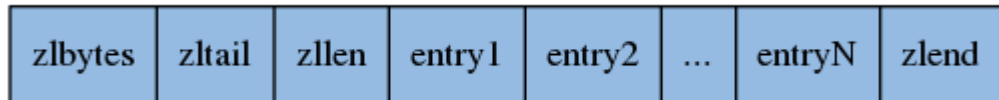
对象与数据结构之间的关系



压缩列表

压缩列表具有以下特点：

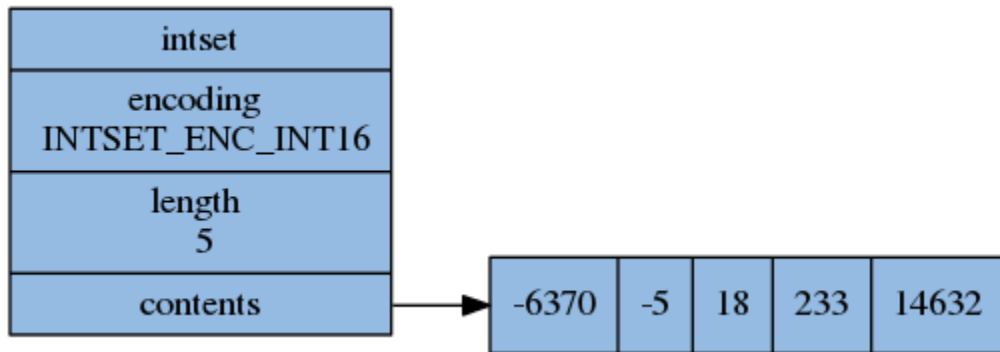
- 压缩列表包含的项都是有序的，列表的两端分别为表头和表尾。
- 每个项可以储存一个字符串、整数或者浮点数。
- 可以从表头开始或者从表尾开始遍历整个压缩列表，复杂度为 $O(N)$ 。
- 定位压缩列表中指定索引上的项，复杂度为 $O(N)$ 。
- 使用压缩列表来储存值消耗的内存比使用双向链表来储存值消耗的内存要少。



整数集合

整数集合具有以下特点：

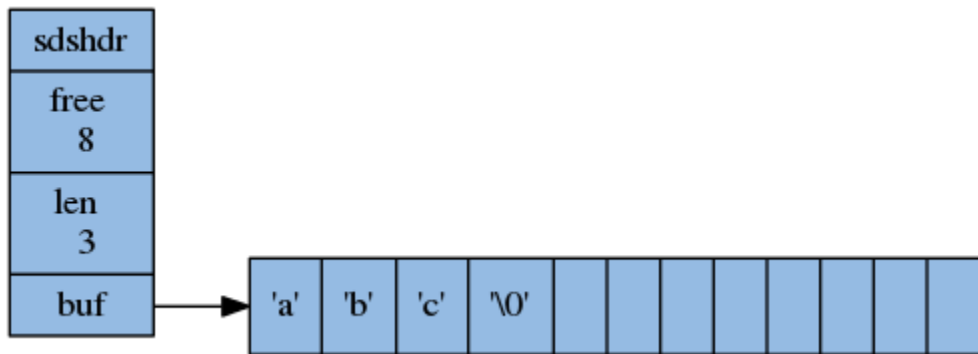
- 集合元素只能是整数(最大为64 位)，并且集合中不会出现重复的元素。
- 集合的底层使用有序的整数数组来表示。
- 数组的类型会随着新添加元素的类型而改变:举个例子，如果**集合中位长度最大的元素**可以使用 16 位整数来保存，那么数组的类型就是 int16_t，而如果集合中位长度最大的元素可以使用 32 位整数来保存的话，那么数组的类型就是 int32_t，诸如此类。
- 数组的类型只会自动增大，但不会减小。



SDS

Redis 使用 SDS (simple dynamic string) 而不是 C 语言的字符串格式(以空字符为结尾的字符数组)来作为 Redis 的默认字符串表示, SDS 具有以下特点:

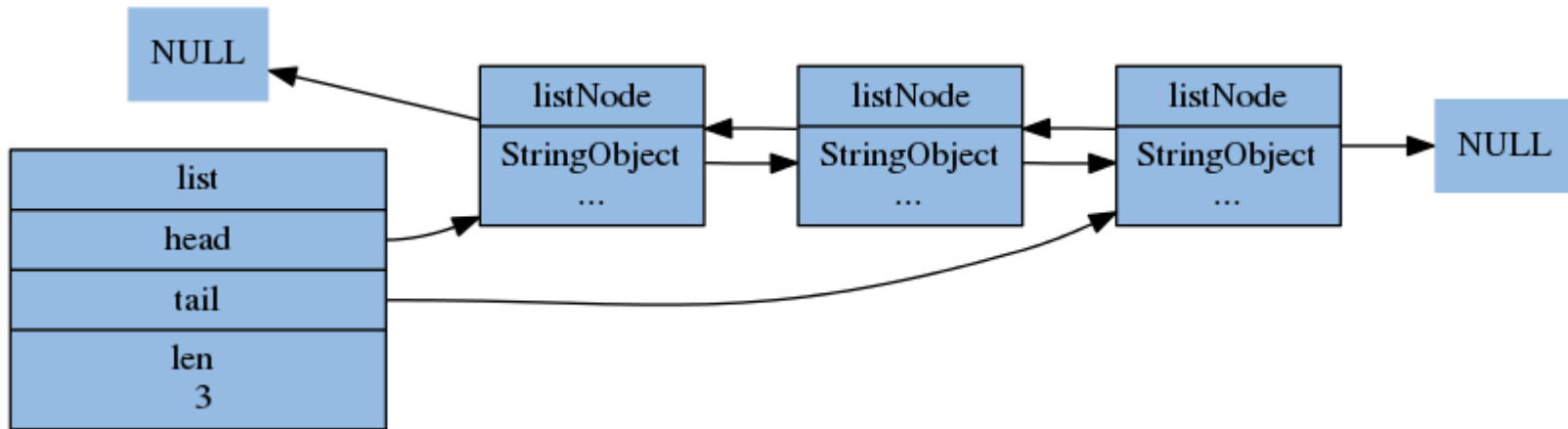
- 可以储存位数组(实现BITOP 和 HyperLogLog)、字符串、整数和浮点数, 其中超过64 位的整数和超过IEEE 754 标准的浮点数使用字符串来表示。
- 具有 int、embstr 和 raw 三种表示形式可选, 其中 int 表示用于储存小于等于 64 位的整数, embstr 用来储存比较短的位数组和字符串, 而其他格式的值则由 raw 格式储存。
- 比起 C 语言的字符串格式, SDS 具有以下四个优点: 1) 常数复杂度获取长度值; 2) 不会引起缓冲区溢出; 3) 通过预分配和惰性释放两种策略来减少内存重分配的 执行次数; 4) 可以储存二进制位。



双向链表

Redis 的双向链表实现具有以下特性：

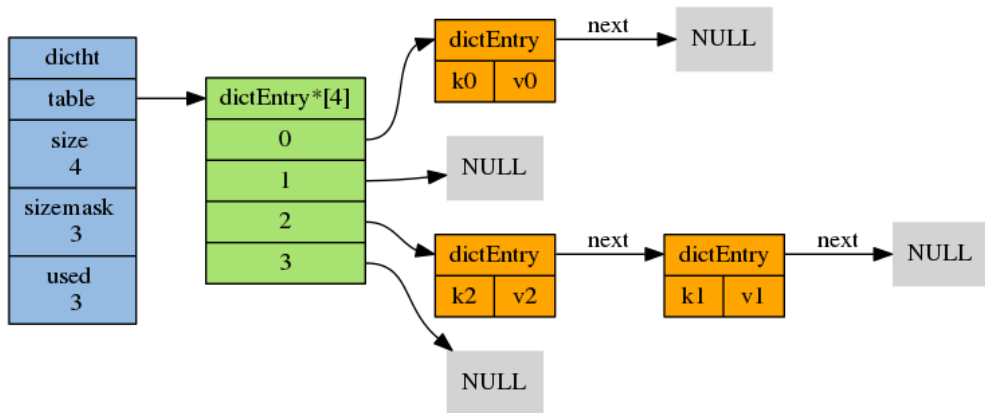
- 双向、无环、带有表头和表尾指针。
- 一个链表包含多个项，每个项都是一个字符串对象，换句话说，一个列表对象可以包含多个字符串对象。
- 可以从表头或者表尾遍历整个链表，复杂度为 $O(N)$ 。
- 定位特定索引上的项，复杂度为 $O(N)$ 。
- 链表带有长度记录属性，获取链表的当前长度的复杂度为 $O(1)$ 。



字典

Redis 的字典实现具有以下特性：

- 查找、添加、删除键值对的复杂度为 $O(1)$ ，键和值都是字符串对象。
- 使用散列表(hash table)为底层实现，使用链地址法(separate chaining)来解决键冲突。
- Redis 会在不同的地方使用不同的散列算法，其中最常用的是 MurmurHash2 算法。
- 在键值对数量大增或者大减的时候会对散列表进行重新散列(rehash)，并且这个 rehash 是渐进式、分多次进行的，不会在短时间内耗费大量 CPU 时间，造成服务器阻塞。

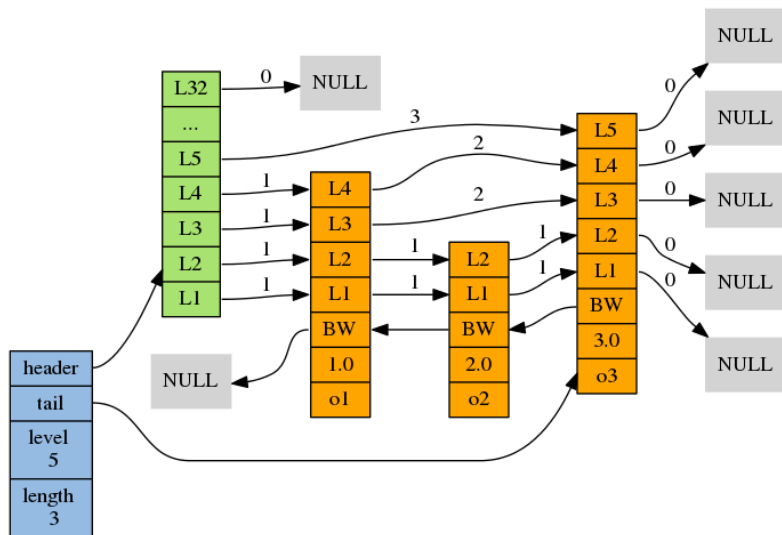


跳跃表

<https://www.cnblogs.com/acfox/p/3688607.html>

Redis 的跳跃表实现具有以下特点：

- 支持平均 $O(\log N)$ 最坏 $O(N)$ 复杂度的节点查找操作，并且可以通过执行范围性(range)操作来批量地获取有序的节点。
- 跳跃表节点除了实现跳跃表所需的层(level)之外，还具有 score 属性和 obj 属性：前者是一个浮点数，用于记录成员的分值；而后者则是一个字符串对象，用来记录成员本身。
- 和字典一起构成 ZSET 结构，用于实现Redis 的有序集合结构：其中字典用于快速获取元素的分值（比如实现ZSCORE 命令），以及判断元素是否存在；而跳跃表则用于执行范围操作（比如实现ZRANGE 命令）。



Redis 数据库的实现方法

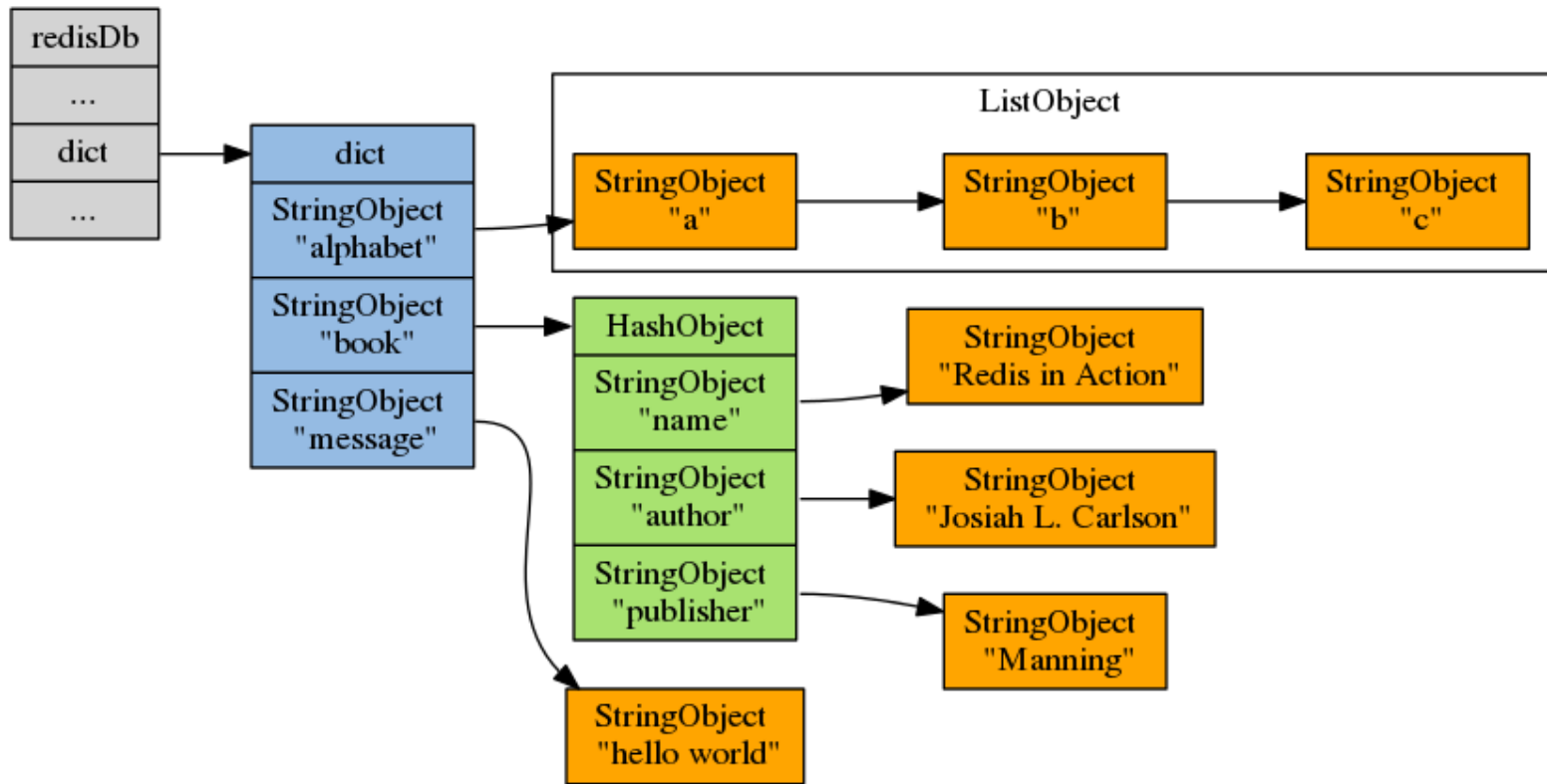
在 Redis 里面, 每个数据库都是一个字典, 该字典的键和值都是我们之前提到的对象, 其中:

- 字典的键总是一个字符串对象, 它储存了用户为键设置的键名。
- 字典的值则可以是字符串对象、列表对象、散列对象、集合对象或者有序集合对象的其中一个。

因为数据库就是字典, 所以针对数据库的操作都是基于字典操作来实现的:

- 比如说, 使用 DEL 命令删除一个数据库键, 就是删除数据库对应的字典的键值对。
- 又比如说, 使用 FLUSHDB 清空数据库, 就是清空数据库对应的字典。
- 诸如此类。

数据库示例

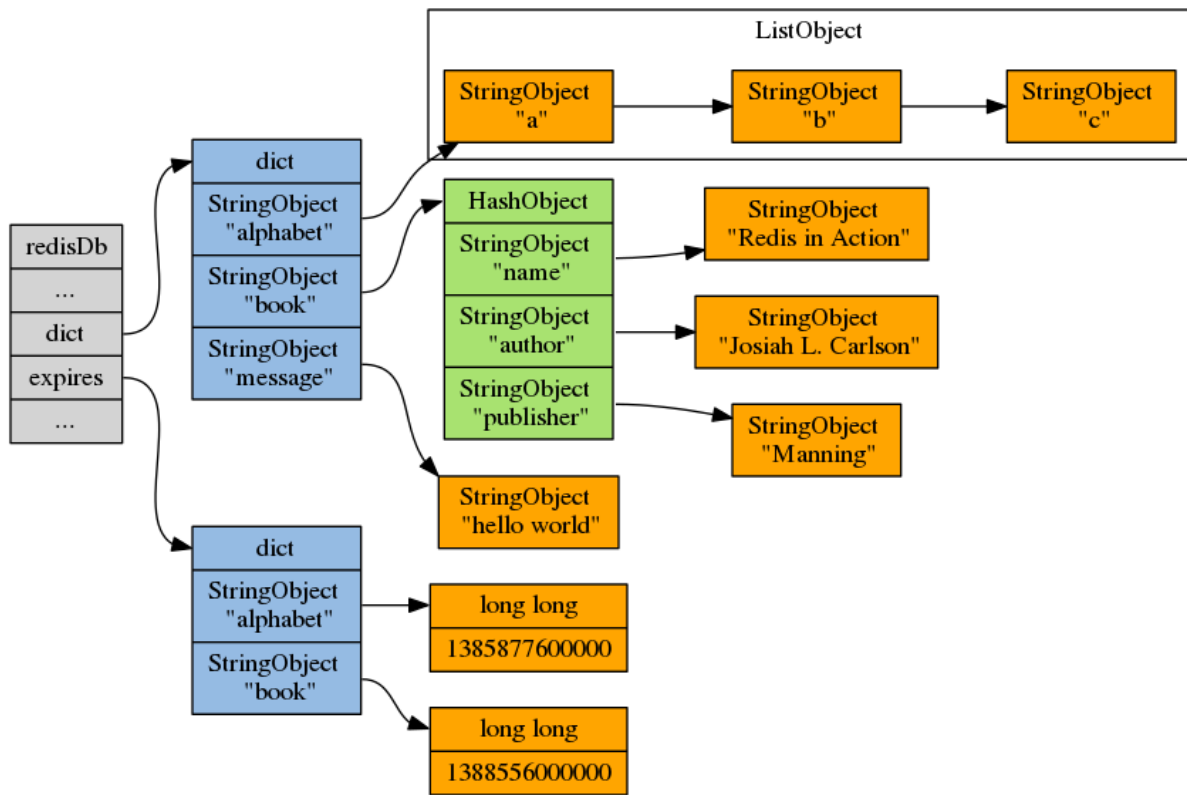


记录过期时间

为了记录数据库键的过期时间，Redis 为每个数据库创建了另一个字典，专门使用这个字典来记录键的过期时间，其中：

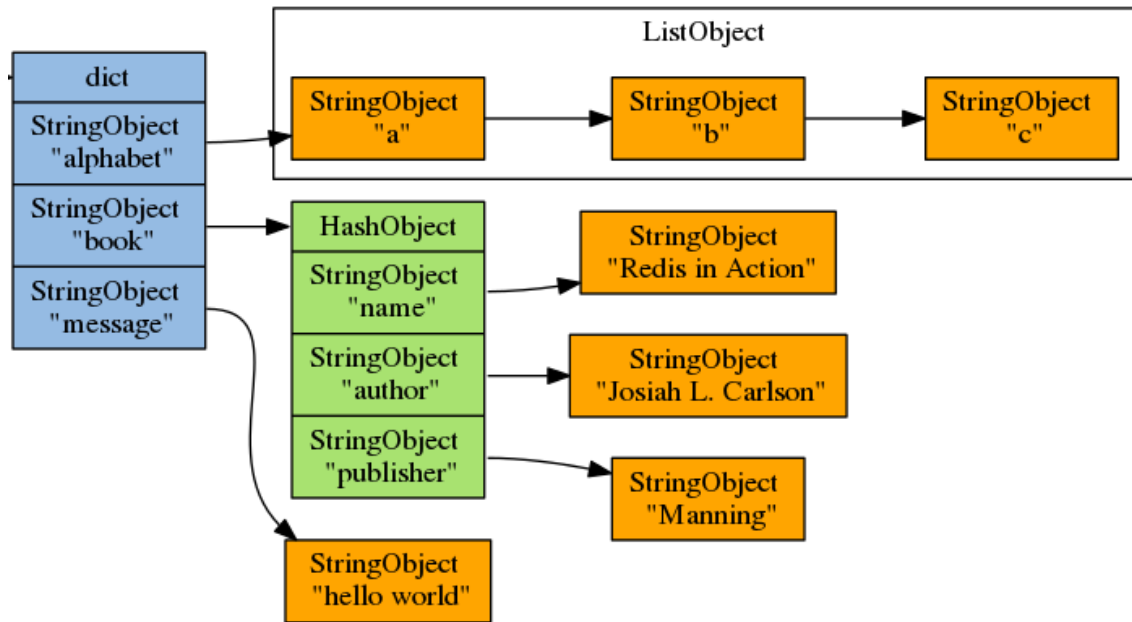
- 字典的键指向数据库键对象，也即是带有过期时间的那个键（数据库字典和储存过期时间的字典通过指针使用同一个键对象，不会造成任何资源浪费）。
- 键的值则是一个毫秒格式的 UNIX 时间戳，记录了键到期的时间。

带有过期时间的数据库示例



RDB 持久化实现原理

Redis 会遍历服务器中的所有数据库，访问数据库中的所有键值对，并根据键值对的类型，将这些键值对以及它们的过期时间写入到 RDB 文件里面。



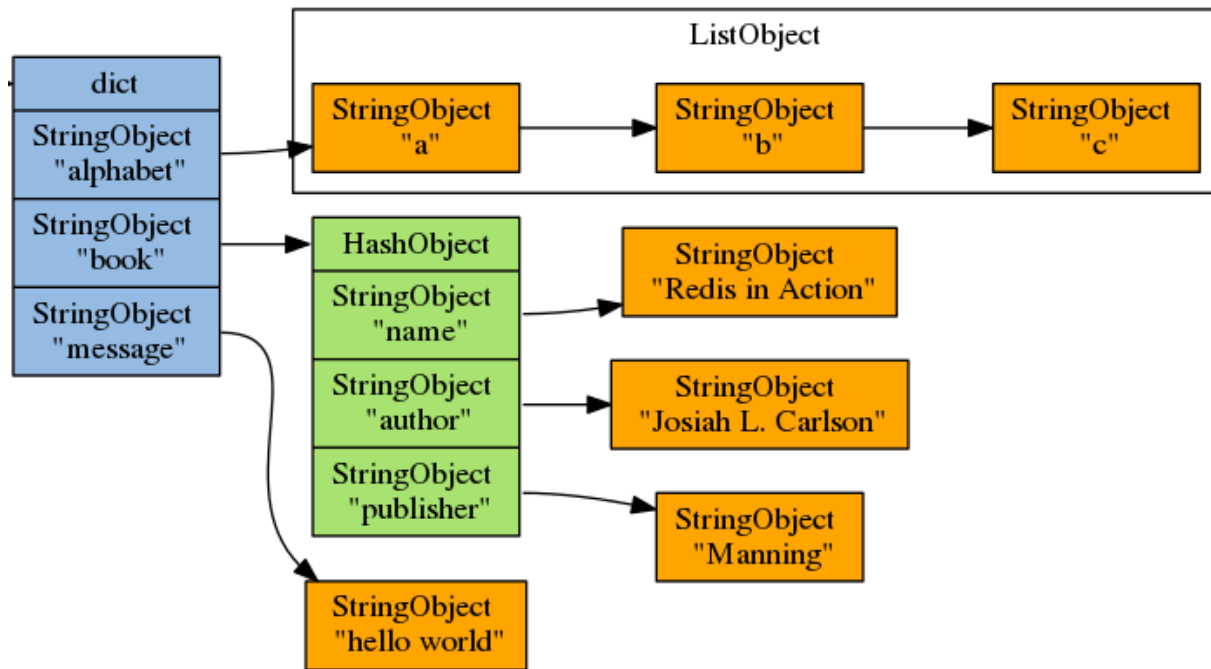
AOF 持久化实现原理

AOF 持久化功能在每次执行命令之后就将协议格式的命令写入到 AOF 缓冲区，然后服务器再定期将缓冲区的内容写入到 AOF 文件，还原数据时只要重新执行 AOF 文件里面的命令即可。

命令执行状态	AOF 文件的内容
未执行任何命令，空数据库	
SET msg "hello world"	SET msg "hello world"
SADD fruits "apple" "banana" "cherry"	SET msg "hello world" SADD fruits "apple" "banana" "cherry"
RPush lst 1 3 5 7	SET msg "hello world" SADD fruits "apple" "banana" "cherry" RPush list 1 3 5 7
DEL fruits	SET msg "hello world" SADD fruits "apple" "banana" "cherry" RPush list 1 3 5 7 DEL fruits

AOF 文件重写的实现原理

无须对现有的 AOF 文件进行处理, 直接根据数据库目前的状态来生成新的 AOF 文件。

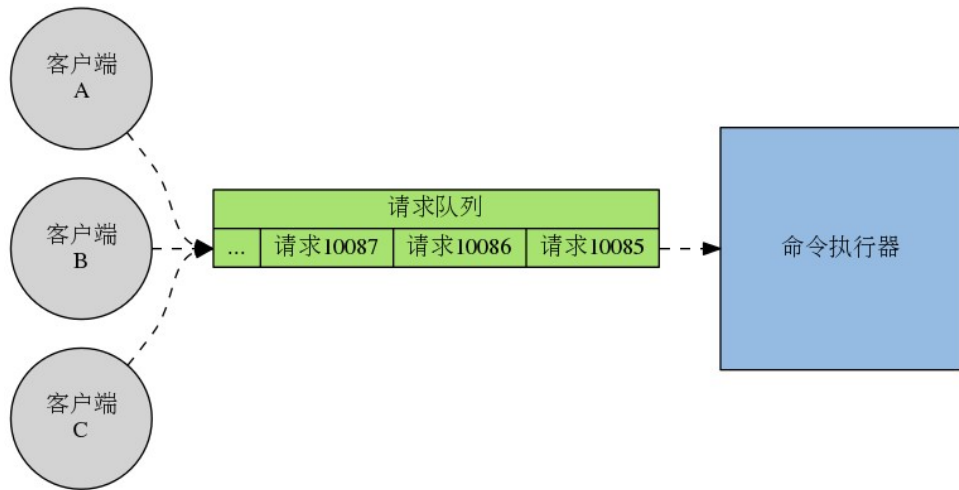


```
SELECT <db>
RPUSH alphabet ...
HMSET book ... SET
message ...
```

命令处理模型

Redis 服务器使用 Reactor 模式 来连接多个客户端并处理命令请求，其中：

- 客户端发送的命令请求会被放到一个有序的队列里面。
- 服务器使用单线程方式来执行命令 —— 服务器每次从队列里面取出一个请求并处理它，只有在当前的命令请求处理完毕之后，服务器才会去处理下一个命令请求。
- 单线程的命令处理方式使得针对服务器以及数据库的操作都不需要加锁，好处是极大地方便了功能的实现，减少了代码出错的可能性；而坏处则是不能最大化地使用硬件的多线程能力。



Redis 数据库中的键和值都是对象，其中键总是一个字符串对象，而值则可以是多种类型对象的其中一个。

Redis 为每种对象都设置了两两种或以上的表示方式，使得 对象可以在不同的应用场景中都有最好的性能表现。

Redis 的数据库就是一个字典，数据库操作都是通过字典操作来实现的；除此之外，Redis 还使用了另一个字典来专门记录键值对的过期时间。

RDB 持久化通过遍历数据库并将键值对写入到文件来实现，而 AOF 持久化则通过记录服务器执行过的命令请求来实现。

Redis 服务器使用 Reactor 模式来处理客户端的命令

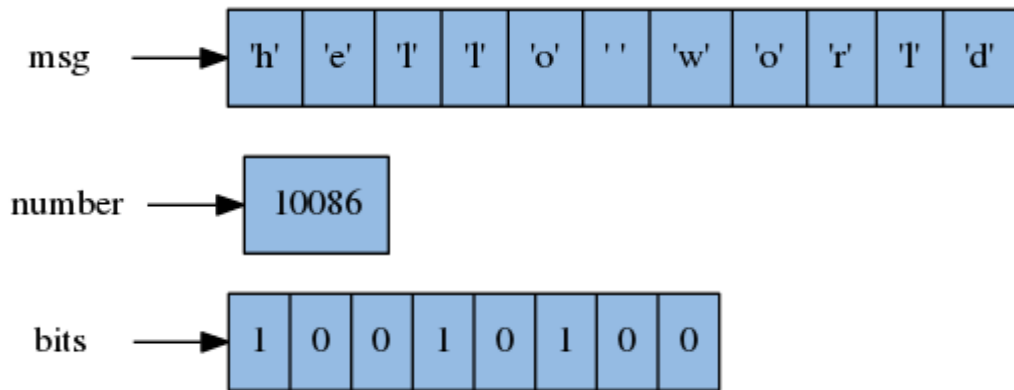
字符串

储存文字、数字或者二进制数据。

字符串(string)

Redis 中最简单的数据结构，它既可以储存文字(比如 "hello world")，又可以储存数字(比如整数 10086 和浮点数 3.14)，还可以储存二进制数据(比如 10010100)。

Redis 为这几种类型的值分别设置了相应的操作命令，让用户可以针对不同的值做不同的处理。



三个字符串，分别储存了文字 "hello world"、数字 10086 以及二进制数据 10010100。

为字符串键设置值

SET key value

将字符串键 key 的值设置为 value，命令返回 OK 表示设置成功。如

果字符串键 key 已经存在，那么用新值覆盖原来的旧值。

复杂度为 $O(1)$ 。

```
redis> SET msg "hello world"
```

OK

```
redis> SET msg "goodbye"      # 覆盖原来的值"hello world"
```

OK

SET key value [NX|XX]

SET 命令还支持可选的 NX 选项和 XX 选项：

- 如果给定了 NX 选项，那么命令仅在键key 不存在的情况下，才进行设置操作；如果键key 已经存在，那么 SET ... NX 命令不做动作(不会覆盖旧值)。
- 如果给定了 XX 选项，那么命令仅在键key 已经存在的情况下，才进行设置操作；如果键key 不存在，那么 SET ... XX 命令不做动作(一定会覆盖旧值)。

在给定 NX 选项和 XX 选项的情况下，SET 命令在设置成功时返回 OK，设置失败时返回 nil。

```
redis> SET nx-str "this will fail" XX           # 键不存在，指定 XX 选项导致设置失败
(nil)

redis> SET nx-str "this will success" NX         # 键不存在，所以指定 NX 选项是可行的
OK

redis> SET nx-str "this will fail" NX           # 键已经存在，指定 NX 选项导致设置失败
(nil)

redis> SET nx-str "this will success again!" XX # 键已经存在，指定 XX 选项是可行的
OK
```

获取字符串的值

GET key

返回字符串键 key 储存的值。

复杂度为 $O(1)$ 。

```
redis> SET msg "hello world"
```

```
OK
```

```
redis> GET msg
```

```
hello world
```

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

```
redis> GET number
```

```
10086
```

仅在键不存在的情况下进行设置

SETNX key value

仅在键 **key** 不存在的情况下，将键 key 的值设置为 value，效果和 SET key value NX 一样。NX 的意思为“Not eXists”（不存在）。

键不存在并且设置成功时，命令返回 1；因为键已经存在而导致设置失败时，命令返回 0。

复杂度为 $O(1)$ 。

```
redis> SETNX new-key "i am a new key!"
```

```
1
```

```
redis> SETNX new-key "another new key here!"
```

键已经存在，设置失败

```
0
```

```
redis> GET new-key
```

键的值没有改变

```
i am a new key!
```

同时设置或获取多个字符串键的值

命令	效果	复杂度
MSET key value [key value ...]	一次为一个或多个字符串键设置值, 效果和同时执行多个 SET 命令一样。 命令返回 OK 。	$O(N)$, N 为要设置的字符串键数量。
MGET key [key ...]	一次返回一个或多个字符串键的值, 效果和同时执行多个 GET 命令一样。	$O(N)$, N 为要获取的字符串键数量。

示例: 设置或获取个人信息

很多网站都会给你一个地方, 填写自己的个人信息、联系信息、个人简介等等, 比如右图就是某个网站上的个人信息设置页面。

通过将每项信息储存在一个字符串键里面 (比如电子邮件在 `huangz::email` 键、个人网站在 `huangz::homepage` 键、公司在 `huangz::company` 键, 等等), 我们可以通过调用 MSET 来一次性设置多个项, 并使用 MGET 来一次性获取多个项的信息。

用户名	huangz
电子邮件	<input type="text" value="huangz1990@gmail.com"/>
个人网站	<input type="text" value="http://huangz.me/"/>
所在公司	<input type="text" value="FakeCompany"/>
工作职位	<input type="text" value="Programmer"/>
所在地	<input type="text" value="广东"/>
签名	<input type="text" value="time waits for no one"/>

```
MSET huangz::email "huangz1990@gmail.com" huangz::homepage "http://huangz.me/" huangz::company "FakeCompany" huangz::position "Programmer" huangz::location "广东" huangz::sign "time waits for no one"
```

```
MGET huangz::email huangz::homepage huangz::company huangz::position ...
```

键的命名

因为 Redis 的数据库不能出现两个同名的键，所以我们通常会使用 `field1::field2::field3` 这样的格式来区分同一类型的多个字符串键。

举个例子，像前面储存个人信息例子，因为网站里面不可能只有 `huangz` 一个用户，所以我们不能用 `email` 键来直接储存 `huangz` 的邮件地址，而是使用 `huangz::email`，这样 `huangz` 的邮件地址就不会和其他用户的邮件地址发生冲突——比如用户名为 `peter` 的用户可以将它的邮件地址储存到 `peter::email` 键，而用户名为 `jack` 的用户也可以将它的邮件地址储存到 `jack::email` 键，大家各不相关，互不影响。

一些更为复杂的键名例子：`user::10086::info`，ID 为 10086 的用户的信息；`news::sport::cache`，新闻网站体育分类的缓存；`message::123321::content`，ID 为 123321 的消息的内容。

`::` 是比较常用的分割符，你也可以选择自己喜欢的其他分割符来命名键，比如斜线 `huangz/email`、竖线 `huangz|email`、或者面向对象风格的 `huangz.email`。

一次设置多个不存在的键

MSETNX key value [key value ...]

只有在所有给定键都不存在的情况下，MSETNX 会为所有给定键设置值，效果和同时执行多个 SETNX 一样。如果给定的键**至少有一个是存在的**，那么 MSETNX 将**不执行任何设置操作**。返回

1 表示设置成功，返回 0 表示设置失败。复杂度为 $O(N)$ ， N 为给定的键数量。

```
redis> MSETNX nx-1 "hello" nx-2 "world" nx-3 "good luck"
```

```
1
```

```
redis> SET ex-key "bad key here"
```

```
OK
```

```
redis> MSETNX nx-4 "apple" nx-5 "banana" ex-key "cherry" nx-6 "durian" 0
```

因为 **ex-key** 键已经存在，所以第二个 MSETNX 会执行失败，所有键都不会被设置。

设置新值并返回旧值

GETSET key new-value

将字符串键的值设置为 new-value，并返回字符串键在设置新值之前储存的旧值(old value)。复

杂度为 $O(1)$ 。

```
redis> SET getset-str "i'm old value"  
OK
```

先给字符串键设置一个值

```
redis> GETSET getset-str "i'm new value"  
i'm old value
```

更新字符串键的值, 并返回之前储存的旧值

```
redis> GET getset-str  
i'm new value
```

确认一下, 新值已被设置

追加内容到字符串末尾

APPEND key value

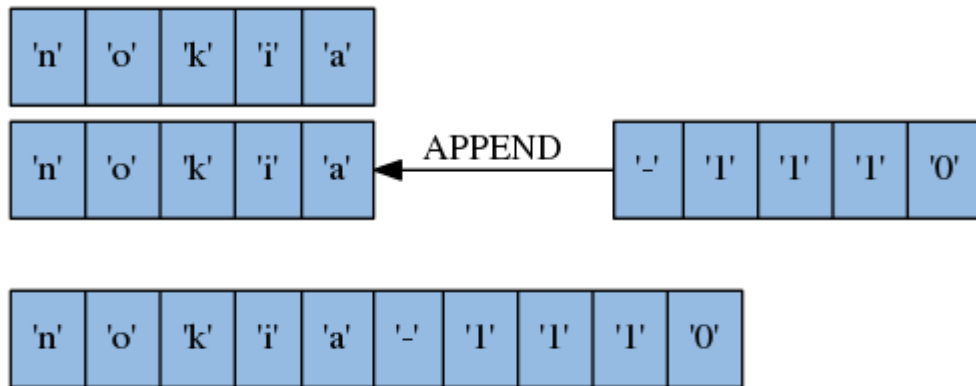
将值 value 推入到字符串键 key 已储存内容的末尾。

$O(N)$, 其中 N 为被推入值的长度。

```
redis> SET myPhone "nokia" OK
```

```
redis> APPEND myPhone "-1110"  
(integer) 10
```

```
redis> GET myPhone  
"nokia-1110"
```



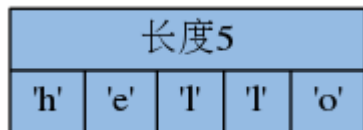
返回值的长度

STRLEN key

返回字符串键 key 储存的值的长度。

因为 Redis 会记录每个字符串值的长度, 所以获取该值的复杂度为 $O(1)$ 。

```
redis> SET msg "hello" OK
redis> STRLEN msg (integer)
5
redis> APPEND msg " world"
(integer) 11
redis> STRLEN msg
(integer) 11
```

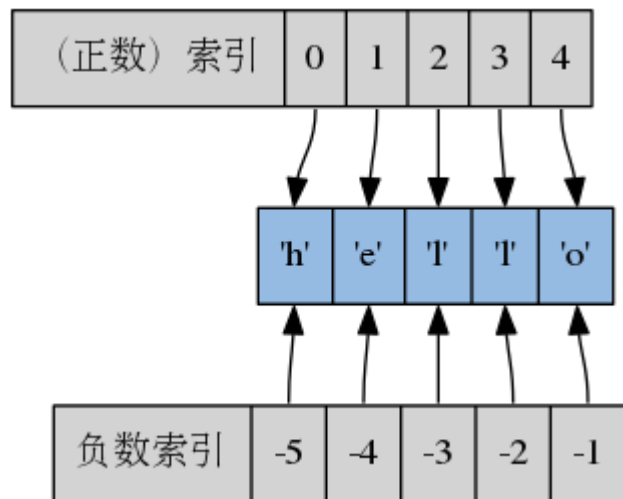


索引

字符串的索引(index)以 0 为开始, 从字符串的开头向字符串的结尾依次递增, 字符串第一个字符的索

引为0，字符串最后一个字符的索引为 $N-1$ ，其中 N 为字符串的长度。

除了(正数)索引之外，字符串还有负数索引：负数索引以 -1 为开始，从字符串的结尾向字符串的开头依次递减，字符串的最后一个字符的索引为 $-N$ ，其中 N 为字符串的长度。



范围设置

SETRANGE key index value

从索引 index 开始，用 value 覆写 (overwrite) 给定键 key 所储存的字符串值。**只接受正数索引**。命令

返回覆写之后，字符串值的长度。复杂度为 $O(N)$ ， N 为 value 的长度。

```
redis> SET msg "hello"
```

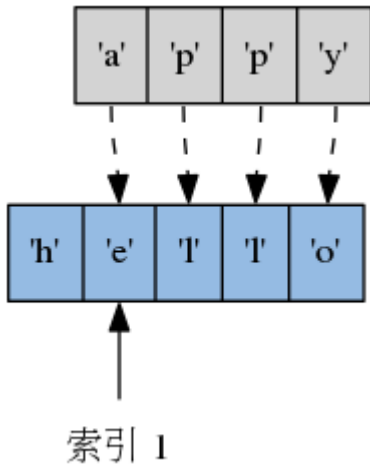
OK

```
redis> SETRANGE msg 1 "appy" (integer)
```

5

```
redis> GET msg
```

"happy"



范围取值

GETRANGE key start end

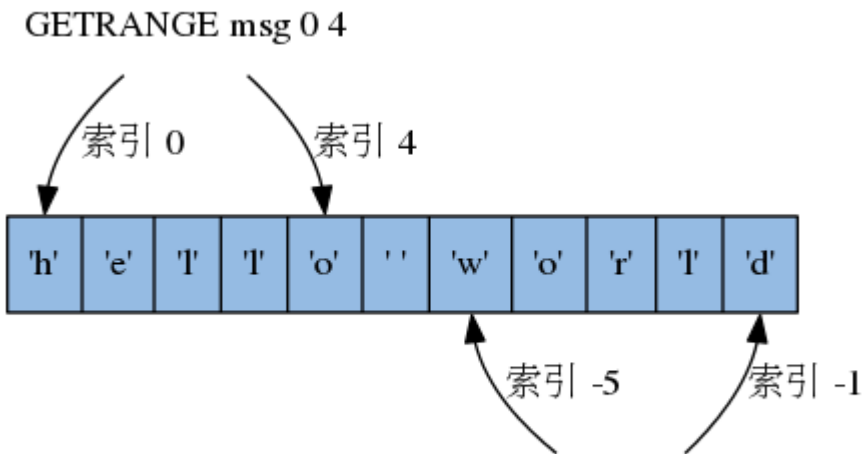
返回键 key 储存的字符串值中，位于 start 和 end 两个索引之间的内容(闭区间，start 和 end 会被包括在内)。和 SETRANGE 只接受正数索引不同，GETRANGE 的索引可以是正数或者负数。

复杂度为 $O(N)$ ，N 为被选中内容的长度。

```
redis> SET msg "hello world" OK
```

```
redis> GETRANGE msg 0 4  
"hello"
```

```
redis> GETRANGE msg -5 -1 "world"
```



设置和获取数字

只要储存在字符串键里面的值可以被解释为 64 位整数, 或者 IEEE-754 标准的 64 位浮点数, 那么用户就可以对这个字符串键执行针对数字值的命令。

值	能否执行数字值命令？	原因
10086	可以	值可以被解释为整数
3.14	可以	值可以被解释为浮点数
+123	可以	值可以被解释为整数
123456789123456789123456789	不可以	值太大, 没办法使用 64 位整数来储存
2.0e7	不可以	Redis 不解释以科学记数法表示的浮点数
123ABC	不可以	值包含文字
ABC	不可以	值为文字

增加或者减少数字的值

对于一个保存着数字的字符串键 key，我们可以使用 INCRBY 命令来增加它的值，或者使用 DECRBY 命令来减少它的值。

命令	效果	复杂度
INCRBY key increment	将 key 所储存的值加上增量 increment，命令返回操作执行之后，键 key 的当前值。	O(1)
DECRBY key decrement	将 key 所储存的值减去减量 decrement，命令返回操作执行之后，键 key 的当前值。	O(1)

如果执行 INCRBY 或者 DECRBY 时，键 key 不存在，那么命令会将键 key 的值初始化为 0，然后再执行增加或者减少操作。

INCRBY / DECRBY 示例

redis> INCRBY num 100 # 键num 不存在, 命令先将 num 的值初始化为0 ,
(integer) 100 # 然后再执行加 100 操作

redis> INCRBY num 25 # 将值再加上 25
(integer) 125

redis> DECRBY num 10 # 将值减少 10
(integer) 115

redis> DECRBY num 50 # 将值减少 50
(integer) 65

增一和减一

因为针对数字值的增一和减一操作非常常见, 所以 Redis 特别为这两个操作创建了 INCR 命令和 DECR 命令。

命令	效果	复杂度
INCR key	等同于执行 INCRBY key 1	O(1)
DECR key	等同于执行 DECRBY key 1	O(1)

```
redis> SET num 10 OK
```

```
redis> INCR num
```

```
(integer) 11 redis>
```

```
DECR num (integer)
```

```
10
```

示例: 计数器(counter)

大家码代码的时候，都听什么音乐？

2 popeyelay · 1 天前 · 2870 次点击

分享一个比较适合码代码时听的歌单(post-rock)。

<http://music.163.com/#/playlist?id=6948994>

一张图理解 Git 对象模型

Git · xiaoronglv · 于 3 天前发布 · 最后由 cricy 于 2 小时前回复 · 1960 次阅读

很多网站都使用了计数器来记录页面被访问的次数。

每当用户访问页面时，程序首先将页面访问计数器的值增一，然后将计数器当前的值返回给用户观看，以使用户通过页面的访问次数来判断页面内容的受关注程度。

使用字符串键以及 INCR、GET 等命令，我们也可以实现这样的计数器。

浮点数的自增和自减

INCRBYFLOAT key increment

为字符串键 key 储存的值加上浮点数增量 increment，命令返回操作执行之后，键 key 的值。没

有相应的 DECRBYFLOAT，但可以通过给定负值来达到 DECRBYFLOAT 的效果。

复杂度为 $O(1)$ 。

```
redis> SET num 10
```

```
OK
```

```
redis> INCRBYFLOAT num 3.14 "13.14"
```

```
redis> INCRBYFLOAT num -2.04      # 通过传递负值来达到做减法的效果  
"11.1"
```

注意事项

即使字符串键储存的是数字值，它也可以执行 APPEND、STRLEN、SETRANGE 和 GETRANGE。当用户针

对一个数字值执行这些命令的时候，Redis 会先将数字值转换为字符串，然后再执行命令。

```
redis> SET number 123
```

```
OK
```

```
redis> STRLEN number
```

转换为 "123"，然后计算这个字符串的长度

```
3
```

```
redis > APPEND number 456
```

转换为 "123"，然后与 "456" 进行拼接

```
6
```

```
redis> GET number
```

```
123456
```


设置和获取二进制数据

SET、GET、SETNX、APPEND 等命令同样可以用于设置二进制数据。

因为 Redis 自带的客户端 redis-cli 没办法方便的设置二进制数据

所以这里使用 Python 客户端来进行

```
>>> import redis
```

```
>>> r = redis.Redis()
```

```
>>> r.set('bits', 0b10010100) # 将字符串键 bits 的值设置为二进制 10010100 True
```

```
>>> bin(int(r.get('bits')))      # 获取字符串键bits 储存的二进制值(需要进行转换)
```

```
'0b10010100'
```

```
>>> r.append('bits', 0b111)      # 将 0b111 (也即是十进制的 7)推入到 bits 已有二进制位的末尾
```

```
4L
```

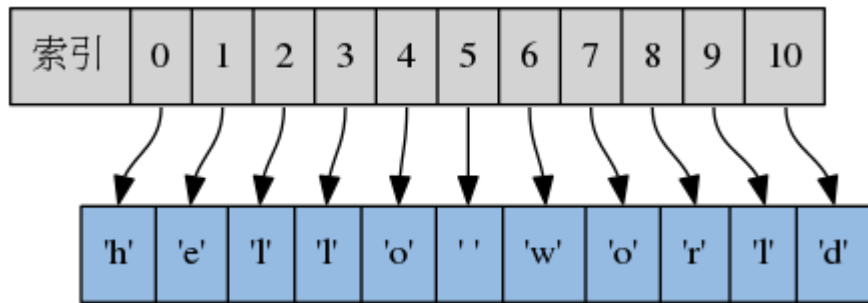
```
>>> bin(int(r.get('bits')))      # 推入之前的值为0b10010100 = 148
```

```
'0b10111001111'                # 推入之后的值为0b10111001111 = 1487
```

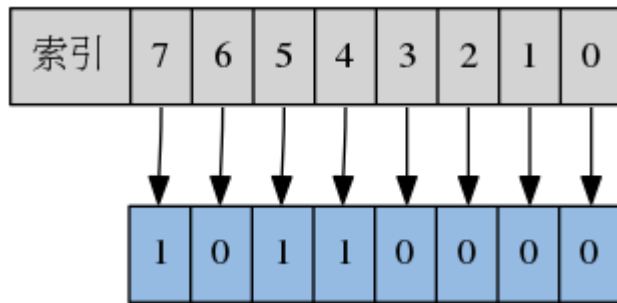
二进制位的索引

和储存文字时一样，字符串键在储存二进制位时，索引也是从 0 开始的。

但是和储存文字时，索引**从左到右依次递增**不同，当字符串键储存的是二进制位时，二进制位的索引会**从左到右依次递减**。



字符串索引



二进制索引

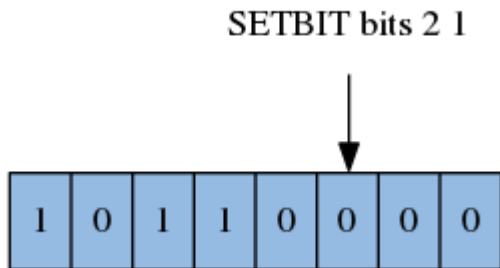
设置二进制位的值

SETBIT key index value

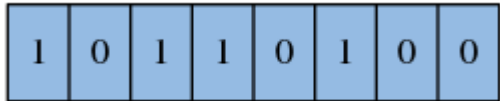
将给定索引上的二进制位的值设置为 value，命令返回被设置的位原来储存的旧值。

复杂度为 $O(1)$ 。

```
redis> SETBIT bits 2 1  
(integer) 0
```



命令执行后



获取二进制位的值

GETBIT key index

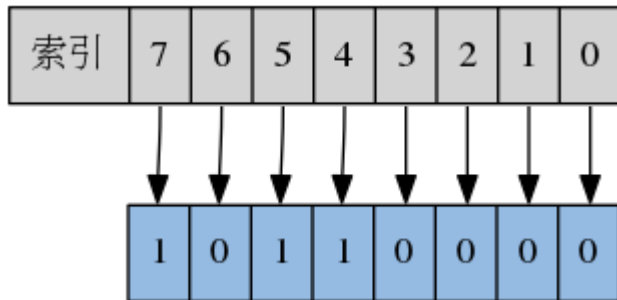
返回给定索引上的二进制位的值。

复杂度为 $O(1)$ 。

redis> GETBIT bits 7
(integer) 1

redis> GETBIT bits 6
(integer) 0

redis> GETBIT bits 4
(integer) 1



计算值为 1 的二进制位的数量

BITCOUNT key [start] [end]

计算并返回字符串键储存的值中，被设置为 1 的二进制位的数量。

一般情况下，给定的整个字符串键都会进行计数操作，但通过指定额外的 start 或 end 参数，可以让计数只在特定索引范围的位上进行。

start 和 end 参数的设置和 GETRANGE 命令类似，都可以使用负数值：比如 -1 表示最后一个位，而 -2 表示倒数第二个位，以此类推。

复杂度为 $O(N)$ ，其中 N 为被计算二进制位的数量。

二进制位运算

BITOP operation destkey key [key ...]

对一个或多个保存二进制位的字符串键执行位元操作，并将结果保存到 destkey 上。
operation 可以是 AND、OR、NOT、XOR 这四种操作中的任意一种：

命令	效果
BITOP AND destkey key [key ...]	对一个或多个 key 求 逻辑并 ，并将结果保存到 destkey 。
BITOP OR destkey key [key ...]	对一个或多个 key 求 逻辑或 ，并将结果保存到 destkey 。
BITOP XOR destkey key [key ...]	对一个或多个 key 求 逻辑异或 ，并将结果保存到 destkey 。
BITOP NOT destkey key	对给定 key 求 逻辑非 ，并将结果保存到 destkey 。

除了 NOT 操作之外，其他操作都可以接受一个或以上数量的 key 作为输入。复杂度为 $O(N)$ ， N 为进行计算的二进制位数量的总和。

命令的返回值为计算所得结果的字节长度，相当于对 destkey 执行 STRLEN 。

小象科技

BITOP 示例

假设现在 b1 键储存了二进制 01001101，而 b2 键储存了二进制 10110101。

```
redis> BITOP AND b1-and-b2 b1 b2      # b1-and-b2 = 00000101  
(integer) 1
```

```
redis> BITOP OR b1-or-b2 b1 b2        # b1-or-b2 = 11111101  
(integer) 1
```

```
redis> BITOP XOR b1-xor-b2 b1 b2      # b1-xor-b2 = 11111000  
(integer) 1
```

```
redis> BITOP NOT not-b1 b1            # not-b1 = 10110010  
(integer) 1
```

示例：实现在线人数统计

一些网站具备了在线人数统计功能，通过这个功能可以看到一段时间以内（比如这个小时，或者这一天），曾经登录过这个网站的会员人数。

首页 · 289 人在线 最高记录 815

某网站的在线人数统计结果，显示目前有 289 个会员在线。

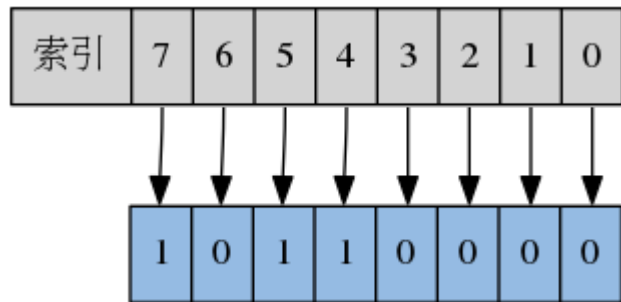
通过使用字符串键以及二进制数据处理命令，我们也可以构建一个高效并且节约内存的在线人数统计实现。

在用户id 和位索引之间进行关联

之前说过，字符串键储存的每个二进制位都有与之对应的索引，比如对于一个 8 位长的二进制值来说，它的各个二进制位的索引值为 0 至 7。

因为通常网站的每个会员都有一个自己的数字 id，比如 peter 的 id 可能是 3，而 jack 的 id 可能是 5，所以我们可以用户在用户id 和二进制位的索引之间进行关联：

- 如果 id 为N 的用户在线，我们就将索引为N 的二进制位的值设置为1。
- 如果索引为N 的二进制位的值为0，这表示 id 为N 用户不在线。
- 使用 BITCOUNT 可以统计有多少个用户在线。
- 通过为每段时间分别储存一个二进制值，我们就可以为每段时间都记录在线用户的数量。（每小时创建一个键或者每天创建一个键，诸如此类。）



peter 的 id 为 3，相对应的二进制位的值为 0，表示 peter 未在线。

jack 的 id 为 5，相对应的二进制位的值为 1，表示 jack 在线。

BITCOUNT 结果为 3，表示共有 3 人在线。

关于用户在线统计的更多信息

目前这个实现的优点：

- 将用户设置为在线的速度非常快， $O(1)$ 。
- 即使用户数量非常大，占用的内存也不多：记录一百万用户仅需一百万位，也即是0.125 MB；记录一千万用户仅需一千万位，也即是1.25 MB。
- 可以在现有程序的基础上，做进一步的操作。举个例子，我们可以使用 BITOP AND 命令，将多个在线记录作为输入，计算出全勤用户的数量（全勤指的是，用户在所有输入的在线统计记录中，都显示为在线）。

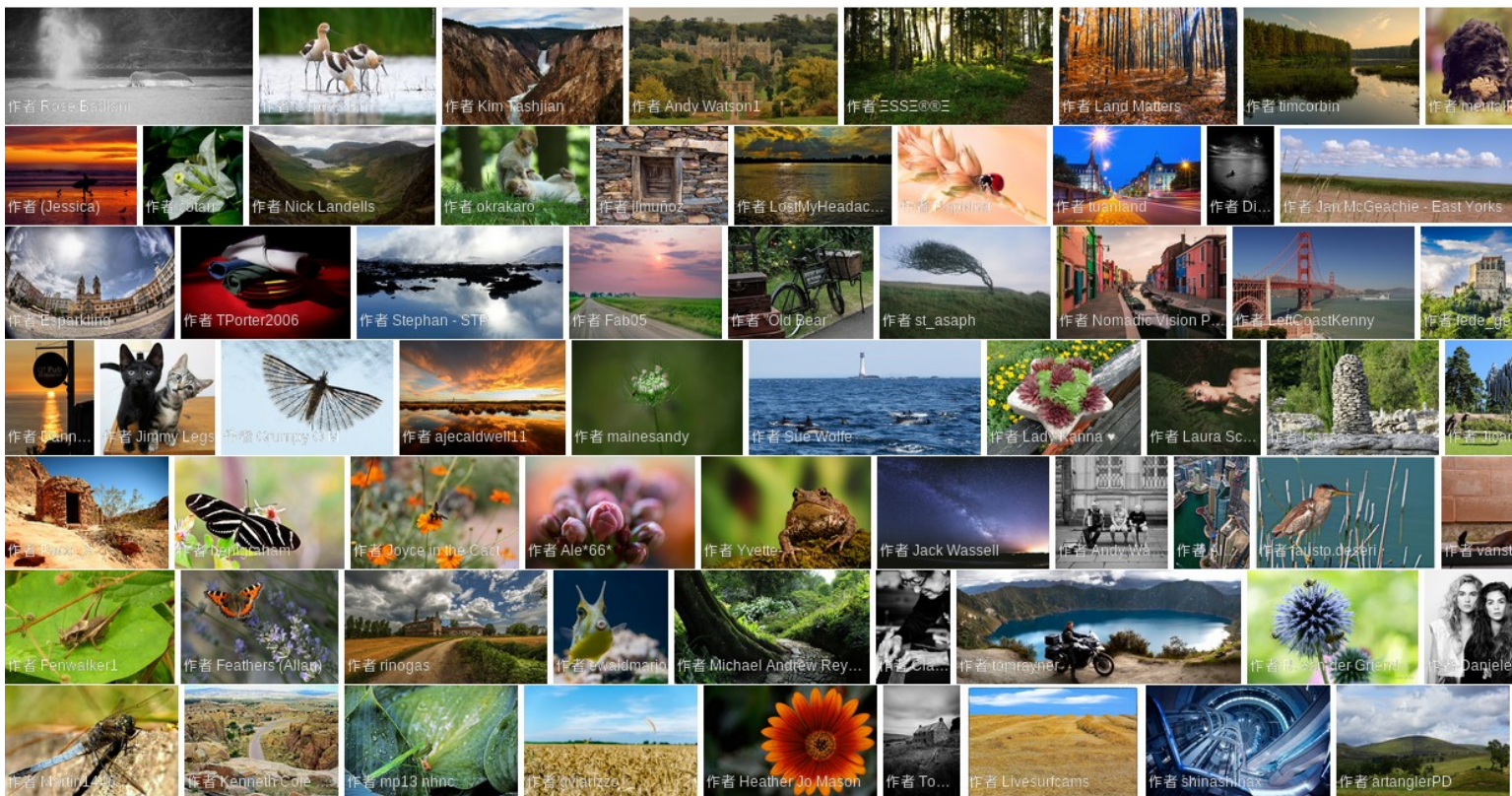
目前这个实现的缺点：

- 每次进行统计的复杂度为 $O(N)$ 。
- 没办法轻易地获取所有在线用户的名单，只能遍历整个二进制值，复杂度为 $O(N)$ ，其中 N 为二进制位数。

进一步的优化：

- 用户量不大并且需要获取在线用户名单的话，可以使用之后介绍的集合数据结构来实现。
- 不需要获取在线用户名单，并且不需要精确的在线统计数量，可以使用之后介绍的 HyperLogLog 来实现。

照片分享



注意事项

一个英文字符只需要使用单个字节来储存，而一个中文字符却需要使用多个字节来储存。

字节	1	2	3	4	5	6	7	8	9	10	11
字符	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'

字节	1	2	3	4	5	6	7	8	9	10	11	12
字符	'世'			'界'			'你'			'好'		

STRLEN、SETRANGE 和 GETRANGE 都是为英文设置的，它们只会在字符为单个字节的情况下正常工作，而一旦我们储存的是类似中文这样的多字节字符，那么这三个命令就不再适用了。

STRLEN 示例

\$ redis-cli --raw # 在 redis-cli 中使用中文时, 必须打开 --raw 选项, 才能正常显示中文

redis> SET msg "世界你好" # 设置四个中文字符

OK

redis> GET msg # 储存中文没有问题

世界你好

redis> STRLEN msg # 这里 STRLEN 显示了“世界你好”的字节长度为12 字节

12 # 但我们真正想知道的是 msg 键里面包含多少个字符

字节	1	2	3	4	5	6	7	8	9	10	11	12
字符	'世'			'界'			'你'			'好'		

SETRANGE 和 GETRANGE 示例

SETRANGE 和 GETRANGE 的情况也是类似的: 因为这两个命令所使用的索引是根据字节而不是字符来编排的, 所以调用 SETRANGE 或者 GETRANGE 来处理中文, 得不到我们想要的结果。

```
redis> SET msg "世界你好" OK
```

```
redis> GETRANGE msg 2 3
```

试图获取 "你好"



```
redis> SETRANGE msg 2 "欢迎你"
```

试图构建 "世界欢迎你", 其中"欢迎你"为9 字节长

```
12
```

```
redis> GET msg
```

??欢迎你?

索引	0	1	2	3	4	5	6	7	8	9	10	11
字节	1	2	3	4	5	6	7	8	9	10	11	12
字符	'世'		'界'		'你'		'好'					

结论

不要使用 STRLEN、SETRANGE 和 GETRANGE 来处理中文。

例外情况:如果你想知道被储存的中文包含多少个字节,那么可以使用 STRLEN

散列

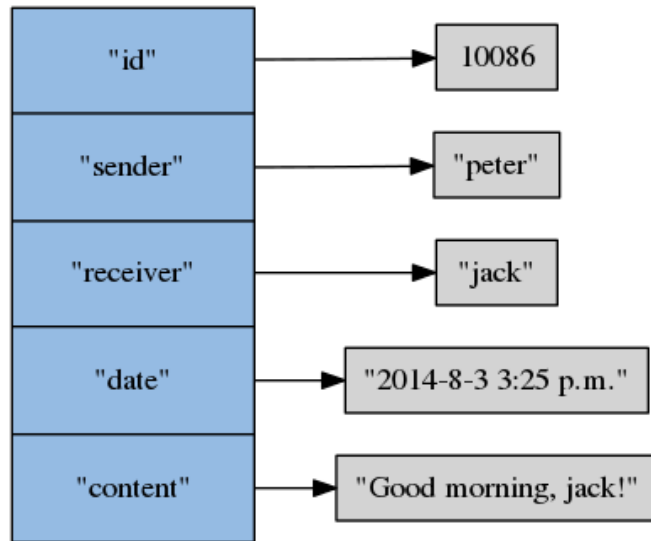
储存多个域值对。

散列(hash)

一个散列由多个域值对(field-value pair)组成, 散列的域和值都可以是文字、整数、浮点数或者二进制数据。

同一个散列里面的每个域必须是独一无二、各不相同的, 而域的值则没有这一要求, 换句话说, 不同域的值可以是重复的。

通过命令, 用户可以对散列执行设置域值对、获取域的值、检查域是否存在等操作, 也可以让 Redis 返回散列包含的所有域、所有值或者所有域值对。



一个储存了消息(message)信息的散列。

关联域值对

HSET key field value

在散列键key 中关联给定的域值对field 和 value 。如果域 field 之前没有关联值, 那么命令返回 1 ; 如果域 field 已经有关联值, 那么命令用新值覆盖旧值, 并返回 0 。

复杂度为 $O(1)$ 。

```
redis> HSET message "id" 10086
```

```
(integer) 1
```

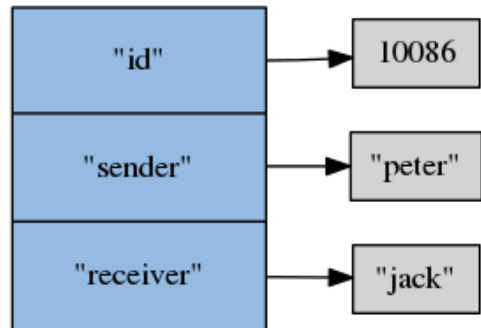
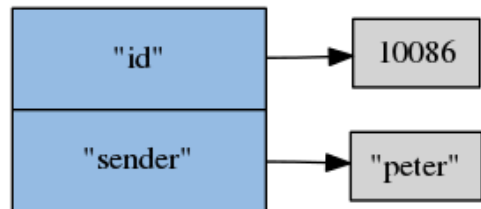
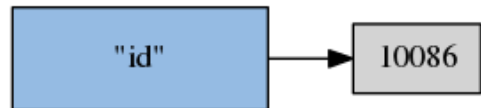
```
redis> HSET message "sender" "peter"
```

```
(integer) 1
```

```
redis> HSET message "receiver" "jack"
```

```
(integer) 1
```

获取域关联的值



HGET key field

返回散列键 key 中, 域 field 所关联的值。如果域 field 没有关联值, 那么返回 nil 。复

杂度为 $O(1)$ 。

```
redis> HGET message "id"
```

```
"10086"
```

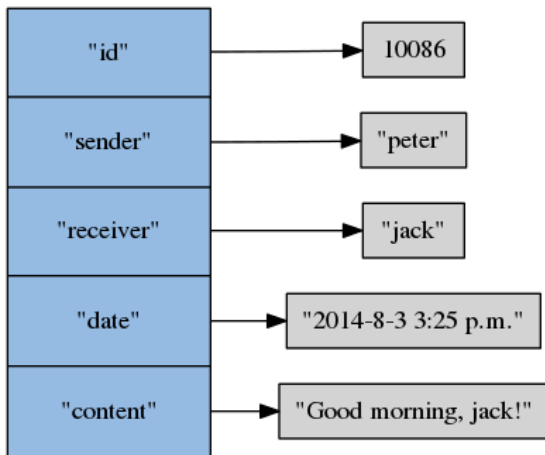
```
redis> HGET message "sender"
```

```
"peter"
```

```
redis> HGET message "content"
```

```
"Good morning, jack!"
```

```
redis> HGET message "NotExistsField" (nil)
```



仅当域不存在时, 关联域值对

HSETNX key field value

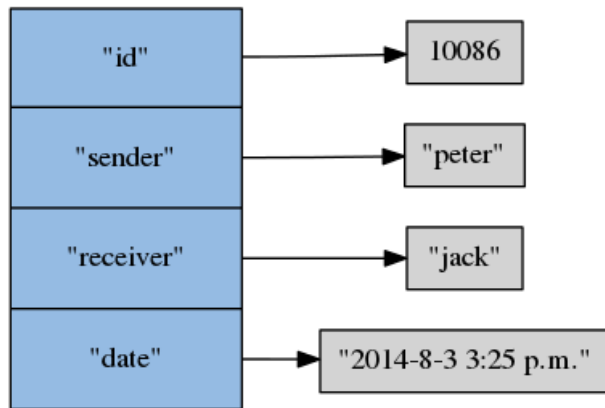
如果散列键 key 中，域 field 不存在(也即是，还没有与之相关联的值)，那么关联给定的域值对 field 和 value 。

如果域 field 已经有与之相关联的值，那么命令不做动作。

复杂度为 $O(1)$ 。

```
redis> HSETNX message "content" "Good morning,jack!"  
(integer) 1
```

```
redis> HSETNX message "content" "Good morning,jack!"  
(integer) 0
```



检查域是否存在

HEXISTS key field

查看散列键 key 中, 给定域 field 是否存在: 存在返回 1, 不存在返回 0。复

杂度为 $O(1)$ 。

```
redis> HEXISTS message "id"
```

```
(integer) 1
```

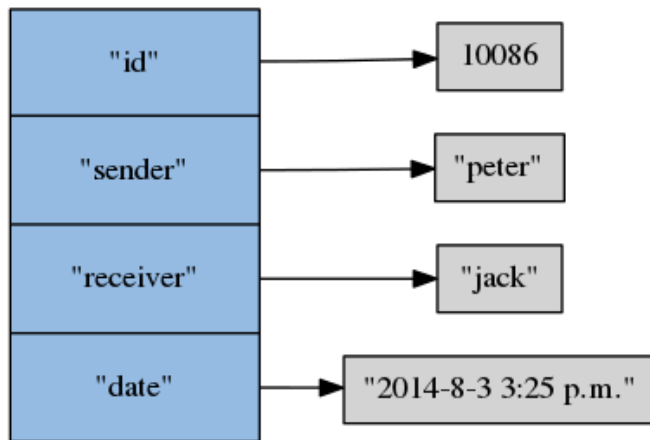
```
redis> HEXISTS message "sender"
```

```
(integer) 1
```

```
redis> HEXISTS message "content"
```

```
(integer) 0
```

```
redis> HEXISTS message "NotExistsField" (integer) 0
```



删除给定的域值对



HDEL key field [field ...]

删除散列键 `key` 中的一个或多个指定域，以及那些域的值。
不存在的域将被忽略。命令返回被成功删除的域值对数量。

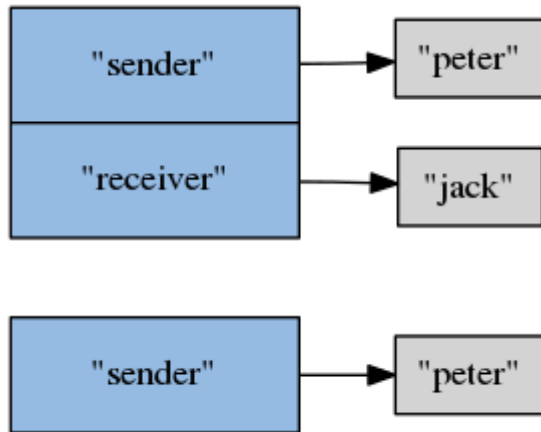
复杂度为 $O(N)$ ， N 为被删除的域值对数量。

```
redis> HDEL message "id"  
(integer) 1
```

```
redis> HDEL message "receiver"  
(integer) 1
```

```
redis> HDEL message "sender"  
(integer) 1
```

获取散列包含的键值对数量



HLEN key

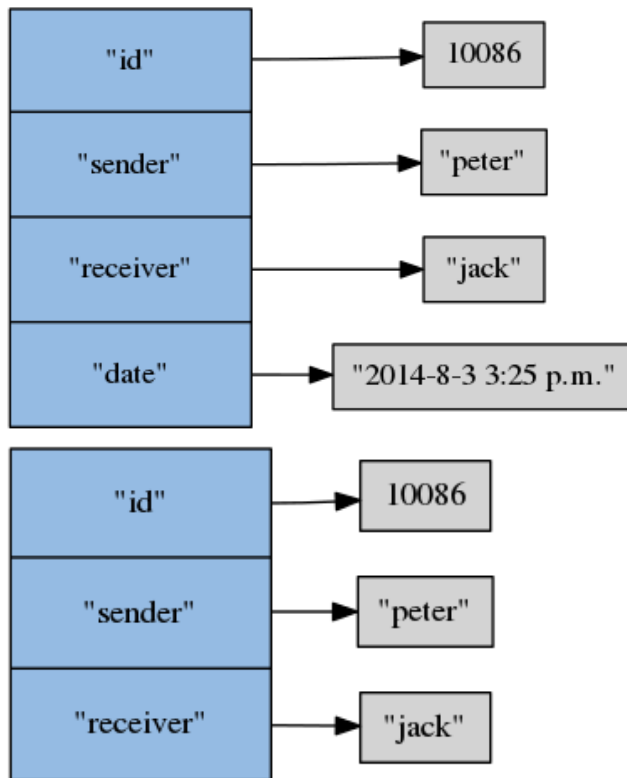
返回散列键 key 包含的域值对数量。

复杂度为 $O(1)$ 。

```
redis> HLEN message  
(integer) 4
```

```
redis> HDEL message "date"  
(integer) 1
```

```
redis> HLEN message  
(integer) 3
```



一次设置或获取散列中的多个域值对

命令	效果	复杂度
----	----	-----

HMSET key field value [field value ...]	在散列键 key 中关联多个域值对，相当于同时执行多个 HSET。	$O(N)$ ，N 为输入的域值对数量。
HMGET key field [field ...]	返回散列键 key 中，一个或多个域的值，相当于同时执行多个 HGET。	$O(N)$ ，N 为输入的域数量。

```
redis> HMSET message "id" 10086 "sender" "peter" "receiver" "jack" OK
```

```
redis> HMGET message "id" "sender" "receiver" 1)
```

```
"10086"
```

```
2) "peter"
```

```
3) "jack"
```

获取散列包含的所有域、值、或者域值对

命令	效果	复杂度
----	----	-----

HKEYS key	返回散列键 key 包含的所有域。	$O(N)$, N 为被返回域的数量。
HVALS key	返回散列键 key 中, 所有域的值。	$O(N)$, N 为被返回值的数量。
HGETALL key	返回散列键 key 包含的所有域值对。	$O(N)$, N 为被返回域值对的数量。

为什么命令叫 HKEYS 而不是 HFIELDS ?

对于散列来说, key 和 field 表示的是同一个意思, 并且 key 比 field 更容易拼写, 所以 Redis 选择使用 HKEYS 来做命令的名字, 而不是 HFIELDS。

HKEYS、HVALS 和 HGETALL 示例

```
redis> HKEYS message
```

```
1) "id"
```

```
2) "sender"
```

```
3) "receiver"
```

4) "date"

5) "content"

redis> HVALS message 1)

"10086"

2) "peter"

3) "jack"

4) "2014-8-3 3:25 p.m."

5) "Good morning, jack!"

redis> HGETALL message

1) "id" # 域

2) "10086" # 值

3) "sender" # 域

4) "peter" # 值

5) "receiver"

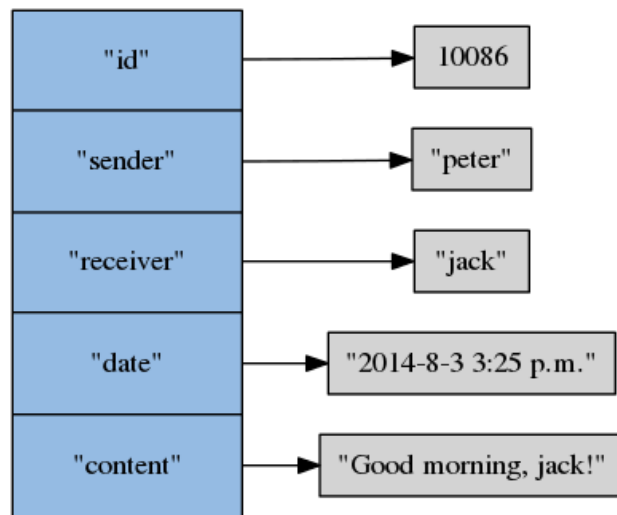
6) "jack"

7) "date"

8) "2014-8-3 3:25 p.m."

9) "content"

10) "Good morning, jack!"



对域的值执行自增操作

命令	作用	复杂度
HINCRBY key field increment	为散列键 key 中，域 field 的值加上整数增量 increment 。	O(1)
HINCRBYFLOAT key field increment	为散列键 key 中，域 field 的值加上浮点数增量 increment 。	O(1)

虽然 Redis 没有提供与以上两个命令相匹配的HDECRBY 命令和 HDECRBYFLOAT 命令，但我们同样可以通过将 increment 设为负数来达到做减法的效果。

```
redis> HINCRBY numbers x 100      # 域不存在, 先将值初始化为0, 然后再执行 HINCRBY 操作
(integer) 100
```

```
redis> HINCRBY numbers x -50      # 传入负值, 做减法
(integer) 50
```

```
redis> HINCRBYFLOAT numbers x 3.14  # 浮点数计算
"53.14"
```

效果类似的命令

散列命令	字符串/数据库命令
HSET	SET
HGET	GET
HSETNX	SETNX
HDEL	DEL(删除一个键, 不仅限于字符串键)
HMSET	MSET
HMGET	MGET
HINCRBY	INCRBY
HINCRBYFLOAT	INCRBYFLOAT
HEXISTS	EXISTS(检查一个键是否存在, 不仅限于字符串键)

如果散列键能做的事情, 字符串键也能做, 那么我们为什么不直接使用字符串键呢?

使用散列的好处(1): 将数据放到同一个地方

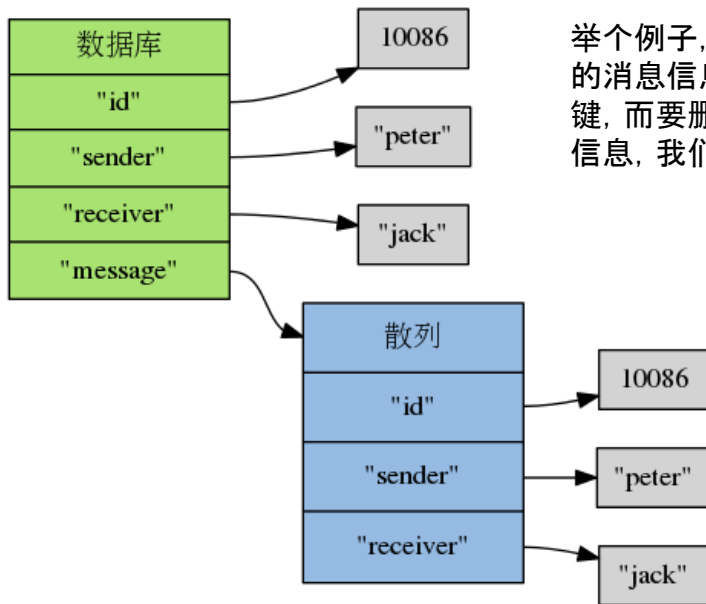
散列可以让我们将一些相关的信息储存在同一个地方, 而不是直接分散地 储存在整个数据库里面, 这不仅方便数据管理, 还可以尽量避免误操作发生。

```
redis> MSET "id" 10086  
         "sender" "peter"  
         "receiver" "jack"
```

OK

```
redis> HMSET message  
         "id" 10086  
         "sender" "peter"  
         "receiver" "jack"
```

OK



举个例子, 要删除字符串键记录的消息信息, 我们需要输入三个键, 而要删除散列键储存的消息信息, 我们只要输入一个键。

使用散列的好处(2):避免键名冲突

在介绍字符串键的时候,我们说过,可以在命名键的时候,使用**分割符来避免命名冲突**,但更好的办法是**直接使用散列键来储存键值对数据**。

举个例子,如果我们需要储存多条消息,那么我们可能会按照格式 `message::<id>::<field>` 的方式来创建字符串键并储存数据,比如使用 `message::12345::receiver` 键来储存 id 为 12345 的消息的接收者,使用 `message::10086::sender` 键来储存 id 为 10086 的发送者,诸如此类。

但更好的办法是直接使用 `message:<id>` 散列键来表示消息信息,并将与消息有关的各项信息储存到散列的各个域里面,比如创建 `message::12345` 散列,并将该消息的各项信息储存分别储存到这个散列的id域、receiver域和sender域里面。

这保证了数据库中每个键的作用都是固定的、单一的,储存的信息都是被隔离的,从而最大限度地避免键名冲突。

对比

以下两个数据库分别以字符串键和散列键两种形式保存了相同的信息，但是很明显，使用字符串键的数据库创建的键数量多，而使用散列键的数据库创建的键数量则少。

随着域数量的增加，使用散列会比使用字符串少 创建很多数据库键。

数据库
"message::10086::id"
"message::10086::sender"
"message::10086::receiver"
"message::12345::id"
"message::12345::sender"
"message::12345::receiver"
"message::15000::id"
"message::15000::sender "
"message::15000::receiver"

数据库
"message::10086"
"message::12345"
"message::15000"

使用散列的好处(3):减少内存占用

在一般情况下,保存相同数量的键值对信息,使用散列键比使用字符串键更节约内存。

因为在数据库里面创建的每个键都带有数据库附加的管理信息(比如这个键的类型、最后一次被访问的时间等等),所以数据库里面的键越多,服务器在储存附加管理信息方面耗费的内存就越多,花在管理数据库键上的 CPU 也会越多。

除此之外,当散列包含的域值对数量比较少的时候,Redis 会自动使用一种占用内存非常少的数据结构来做散列的底层实现,在散列的数量比较多时,这一措施对减少内存有很大的帮助。

结论

只要有可能的话, 就尽量使用散列键而不是字符串键来储存键值对数据, 因为散列键管理方便、能够避免键名冲突、并且还能够节约内存。

一些没办法使用散列键来代替字符串键的情况:

1. **使用二进制位操作命令**: 因为Redis 目前支持对字符串键进行 SETBIT、GETBIT、BITOP 等操作, 如果你想使用这些操作, 那么只能使用字符串键。
2. **使用过期功能**: Redis 的键过期功能目前只能对键进行过期操作, 而不能对散列的域进行过期操作, 因此如果你要对键值对数据使用过期功能的话, 那么只能把键值对储存在字符串里面。关于过期键的详细信息会在之后介绍。

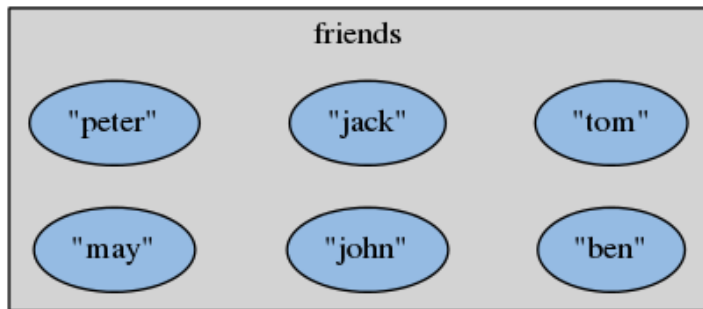
集合

储存多个各不相同的元素。

集合(set)

Redis 的集合以无序的方式储存多个各不相同的元素。

用户可以快速地向集合添加元素，或者从集合里面 删除元素，也可以对多个集合进行集合运算操作，比如计算并集、交集和差集。



添加元素

SADD key element [element ...]

将一个或多个元素添加到给定的集合里面，
已经存在于集合的元素会自动被忽略，
命令返回新添加到集合的元素数量。

```
redis> SADD friends "peter"
```

```
(integer) 1
```

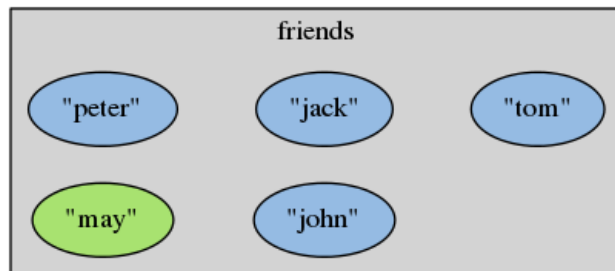
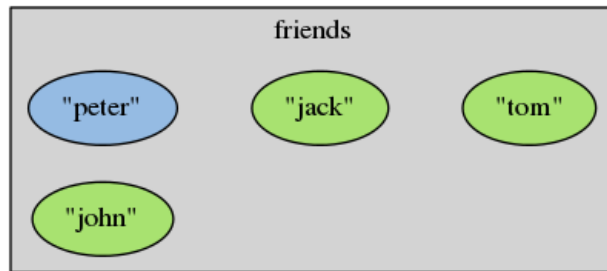
```
redis> SADD friends "jack" "tom" "john"
```

```
(integer) 3
```

```
redis> SADD friends "may" "tom"
```

```
(integer) 1
```

命令的复杂度为 $O(N)$, N 为成功添加的元素数量。



移除元素

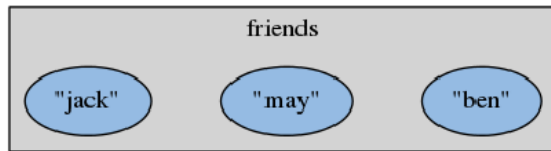
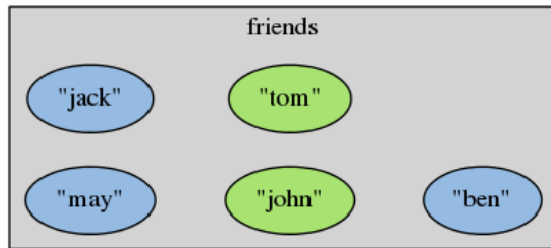
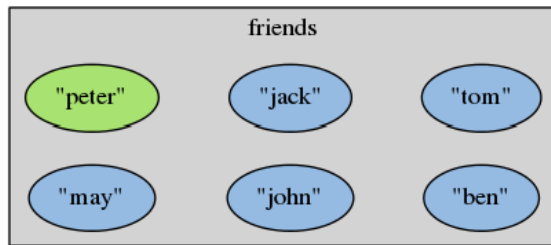
SREM key element [element ...]

移除集合中的一个或者多个元素，
不存在于集合中的元素会自动被忽略，
命令返回存在并且被移除的元素数量。

```
redis> SREM friends "peter"  
(integer) 1
```

```
redis> SREM friends "tom" "john"  
(integer) 2
```

命令的复杂度为 $O(N)$, N 为被移除元素的数量。



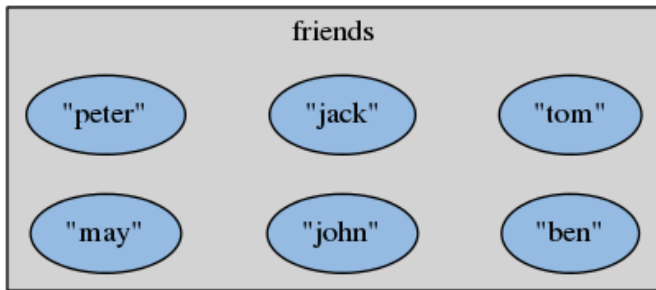
检查给定元素是否存在于集合

SISMEMBER key element

检查给定的元素是否存在于集合，存在的话返回 1；
如果元素不存在，或者给定的键不存在，那么返回 0。

```
redis> SISMEMBER friends "peter"  
(integer) 1
```

```
redis> SISMEMBER friends "li lei"  
(integer) 0
```



```
redis> SISMEMBER NOT-EXISTS-KEY "element" (integer) 0
```

命令的复杂度为 $O(1)$ 。

返回集合的大小

SCARD key

返回集合包含的元素数量(也即是集合的基数)。

```
redis> SCARD friends
```

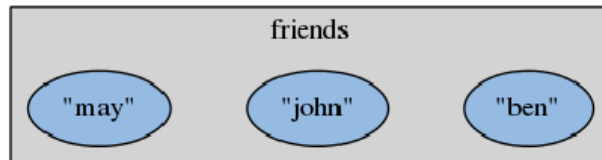
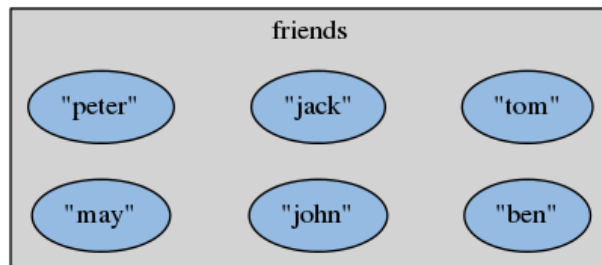
```
(integer) 6
```

```
redis> SREM friends "peter" "jack" "tom"
```

```
(integer) 3
```

```
redis> SCARD friends
```

```
(integer) 3
```



因为 Redis 会储存集合的长度, 所以命令的复杂度为 $O(1)$ 。

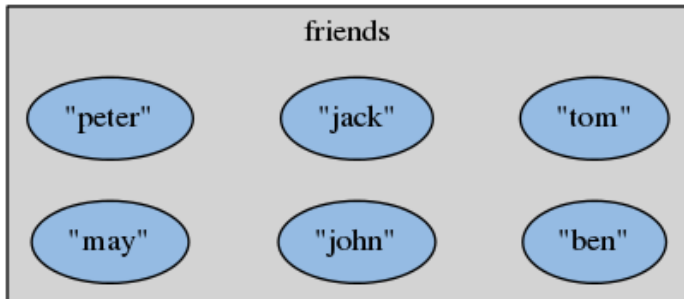
返回集合包含的所有元素

SMEMBERS key

返回集合包含的所有元素。

```
redis> SMEMBERS friends
```

- 1) "jack"
- 2) "peter"
- 3) "may"
- 4) "tom"
- 5) "john"
- 6) "ben"



命令的复杂度为 $O(N)$ ， N 为集合的大小。

当集合的基数比较大时，执行这个命令有可能会造成服务器阻塞，将来会介绍更好的方式来迭代集合中的元素。

集合的无序性质

对于相同的一集元素, 同一个集合命令可能会返回不同的 结果。

```
redis> SADD friends "peter" "jack" "tom" "john"  
"may" "ben"  
(integer) 6
```

```
redis> SMEMBERS friends  
1) "jack"  
2) "peter"  
3) "may"  
4) "tom"  
5) "john"  
6) "ben"
```

```
redis> SADD another-friends "ben" "may" "john" "tom"  
"jack" "peter"  
(integer) 6
```

```
redis> SMEMBERS another-friends  
1) "may"  
2) "ben"  
3) "john"  
4) "tom"  
5) "jack"  
6) "peter"
```

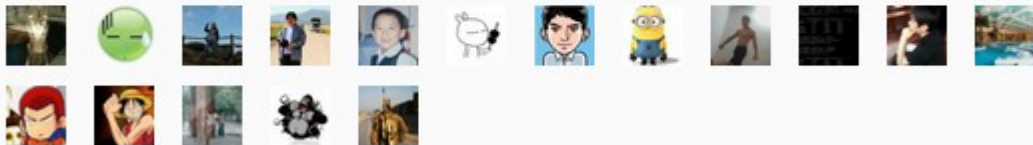
结论: 不要使用集合来储存有序的数据。如果想要储存有序且重复的值, 可以使用列表; 如果想要储存有序且无重复的值, 可以使用之后介绍的有序集合。

示例：赞、喜欢、Like、签到.....

今天从编辑那里获得消息，《Redis 设计与实现》的纸书要加印了，真的非常感谢大家的支持。

8月5日 20:25 来自微博 weibo.com

阅读(4890) 推广 |  (17) | 转发(2) | 收藏 | 评论(9)



每个用户可以点一次赞，
也可以取消赞，
也可以查看多少人赞了，
以及哪些人赞了。

除了微博的赞之外，网上还有很多作用类似的功能，比如豆瓣的“喜欢”、Facebook 的“Like”、以及一些网站上的“签到”等等。

这些功能实际上都是一种投票功能，比如“赞”就是投一票，而“取消赞”就是撤销自己的投票。

通过使用 Redis 的集合，我们也可以实现类似的投票功能。

示例：打标签功能

Linux 系统编程（第二版）

 更新描述或封面

作者: Robert Love
出版社: 东南大学出版社
原作名: Linux in a Nutshell
出版年: 2011年
页数: 429
定价: 78.00元
装帧: 平装
ISBN: 9787112121111

我最近在读这本书 [修改](#) [删除](#)
我的评价: ☆☆☆☆☆
标签: Linux 系统编程
[写笔记](#) [写书评](#) [加入购书单](#)

作者简介

洛夫 (Robert Love) 在Linux早期开发中扮演了重要角色。他是Google公司的资深软件工程师，也是Linux内核的早期贡献者之一。

添加收藏：修改

☐ 想读 ☒ 在读 ☐ 读过 给个评价吧?(可选) ☆☆☆☆☆ [收起](#)

标签(多个标签用空格分隔):

我的标签:

O'Reilly 计算机 系统编程 Linux 算法 函数式编程 数据库
设计 程序设计 操作系统 数据结构 C 日本 lisp 创业
历史 软件工程 编程语言 分布式 C++

常用标签:

Linux 系统编程 编程 计算机 O'Reilly C Unix 经典

简短附注:
350

☐ 仅自己可见

分享到 ☐ 我的广播 ☒ 去绑定新浪微博

保存

豆瓣允许你在标记一本书的同时，为这本书添加标签，比如“Linux”、“系统编程”、“计算机”、“C”、“编程”，等等。

使用 Redis 的集合也可以实现这样的加标签功能。

打标签功能的 API 及其实现

API	作用	实现原理
Tag(key, client)	使用指定的键来储存某种东西的标签。	使用集合来储存标签。
Tag.add(*tags)	添加任意多个标签。	调用 SADD 命令。
Tag.remove(*tags)	移除任意多个标签。	调用 SREM 命令。
Tag.is_include(tag)	检查某个标签是否存在。	调用 SISMEMBER 命令。
Tag.get_all()	返回所有标签。	调用 SMEMBERS 命令。
Tag.count()	返回标签的数量。	调用 SCARD 命令。

从集合里面随机地弹出一个元素

SPOP key

随机地从集合中移除并返回一个元素，复杂度为 $O(1)$ 。

```
redis> SADD friends "peter" "jack" "tom" "john" "may" "ben" (integer)
```

```
6
```

```
redis> SPOP friends
```

```
"may"
```

```
redis> SMEMBERS friends
```

```
1) "tom"
```

```
2) "john"
```

```
3) "jack"
```

```
4) "peter"
```

```
5) "ben"
```

从集合里面随机地返回元素

SRANDMEMBER key [count]

如果没有给定可选的 count 参数, 那么命令随机地返回集合中的一个元素。

如果给定了 count 参数, 那么:

- 当 count 为正数, 并且少于集合基数时, 命令返回一个包含 count 个元素的数组, 数组中的每个元素**各不相同**。如果 count 大于或等于集合基数, 那么命令返回整个集合。
- 当 count 为负数时, 命令返回一个数组, 数组中的元素**可能会重复出现多次**, 而数组的长度为 count 的绝对值。

和 SPOP 不同, SRANDMEMBER **不会**移除被返回的元素。命令

的复杂度为 $O(N)$, N 为被返回元素的数量。

SRANDMEMBER 的使用示例

```
redis> SADD friends "peter" "jack"  
"tom" "john" "may" "ben" (integer)
```

1

```
redis> SRANDMEMBER friends      # 随机地返回一个元素
```

```
"ben"
```

```
redis> SRANDMEMBER friends 3    # 随机地返回三个无重复的元素
```

```
1) "john"
```

```
2) "jack"
```

```
3) "peter"
```

```
redis> SRANDMEMBER friends -3   # 随机地返回三个可能有重复的元  
素
```

```
1) "may"
```

```
2) "peter"
```

```
3) "may"
```

[他的主页](#)[微博](#)[个人资料](#)[相册](#)[赞](#)

#转发赠书#由@黄健宏_huangz 同学精心打造的《Redis设计与实现》
(<http://t.cn/RvQPwtE> 正式出版了！可能不少同学都读过网上的电子版，但本书比电子版内容更翔实更精良。只要转发就有机会免费得新书，第一期提供10本新书随机送，7.7日截止~图书由@华章图书 提供~



6月26日 10:09 来自微博 weibo.com | 举报

(9) | [转发\(221\)](#) | [收藏](#) | [评论\(64\)](#)



☐ 同时转发到我的微博

评论

微博上的转发抽奖活动：每个参与者只需要进行转发，就有机会获得奖品。

通过将参与抽奖的所有人都添加到一个集合里面，并使用 SRANDMEMBER 命令来抽取获奖得主，我们也可以构建一个类似的抽奖功能。

抽奖程序的 API 及其实现

API	作用	实现原理
Loterry(key, client)	使用指定的键来储存参与抽奖的人。	使用集合来储存参与抽奖的人。
Loterry.add_player(*user)	添加参与者。	调用 SADD 命令。
Loterry.get_all_players()	返回所有参与者。	调用 SMEMBERS 命令。
Loterry.player_count()	返回参与者数量。	调用 SCARD 命令。
Loterry.draw(n)	抽出 n 个获奖者。	调用 SRANDMEMBER 命令。

抽奖程序的使用示例

创建一个赠送 Redis 书的抽奖活动

```
book_loterry('loterry::redis_book', client)
```

不断地添加参与者

```
book_loterry.add_player('peter', 'jack', 'tom', 'may', ..., 'ben')
```

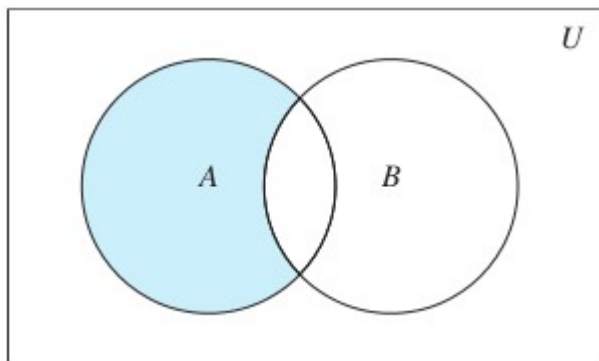
抽出三名幸运的获奖者

```
book_loterry.draw(3)
```

差集运算

命令	作用	复杂度
----	----	-----

SDIFF key [key ...]	计算所有给定集合的差集, 并返回结果。	$O(N)$, N 为所有参与差集计算的元素数量之和。
SDIFFSTORE destkey key [key ...]	计算所有给定集合的差集, 并将结果储存到 destkey 。	$O(N)$, N 为所有参与差集计算的元素数量之和。



图片来自
《离散数学及其应用(第7版)》

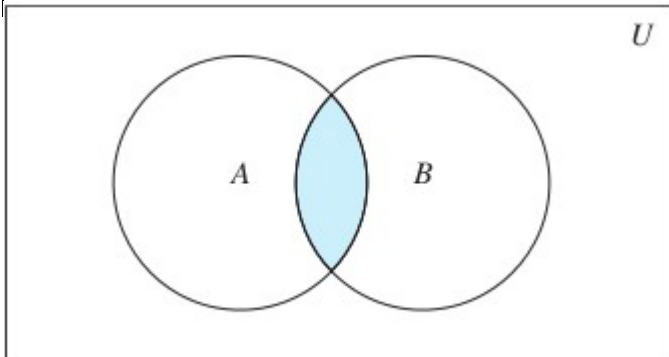
```
redis> SADD number1 "123" "456" "789"
(integer) 3
```

```
redis> SADD number2 "123" "456" "999"
(integer) 3
```

```
redis> SDIFF number1 number2 1)
"789"
```

交集运算

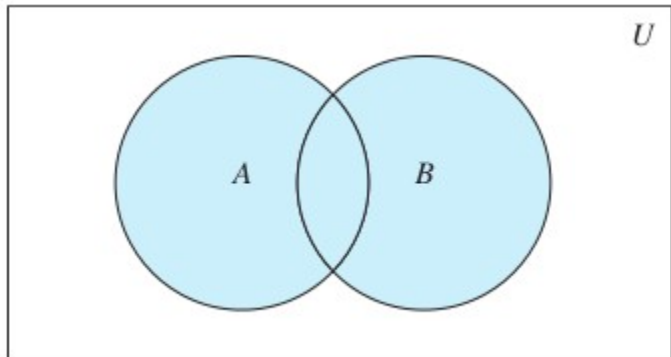
命令	作用	复杂度
SINTER key [key ...]	计算所有给定集合的交集，并返回结果。	$O(N * M)$ ， N 为给定集合当中基数最小的集合， M 为给定集合的个数。
SINTERSTORE destkey key [key ...]	计算所有给定集合的交集，并将结果储存到 destkey 。	$O(N * M)$ ， N 为给定集合当中基数最小的集合， M 为给定集合的个数。



```
redis> SADD number1 "123" "456" "789"
(integer) 3
redis> SADD number2 "123" "456" "887"
(integer) 3
redis> SINTER number1 number2 1)
"123"
2) "456"
```

并集运算

命令	作用	复杂度
SUNION key [key ...]	计算所有给定集合的并集，并返回结果。	$O(N)$, N 为所有参与计算的元素数量。
SUNIONSTORE destkey key [key ...]	计算所有给定集合的并集，并将结果储存到 destkey。	$O(N)$, N 为所有参与计算的元素数量。



```
redis> SADD number1 "123" "456" "789"
(integer) 3
redis> SADD number2 "123" "456" "887"
(integer) 3
redis> SUNION number1 number2 1)
"123"
2) "456"
3) "789"
4) "887"
```

示例:使用集合实现共同关注功能

新浪微博的共同关注功能:当用户访问另一个用户时,该功能会显示出两个用户关注了哪些相同的用户。

如果我们把每个用户的关注对象都分别放到一个集合里面的话,那么我们可以使用 SINTER 或者 SINTERSTORE 命令来实现这个共同关注功能。

例如:

```
peter = {'john', 'jack', 'may'}
```

```
ben = {'john', 'jack', 'tom'}
```

那么 peter 和 ben 的共同关注为:

```
SINTER peter ben = {'john', 'jack'}
```

还可以使用 SINTERSTORE 来避免重复计算。



示例:构建商品筛选功能

手机 - 商品筛选

品牌：

 HUAWEI	 SAMSUNG		 Coolpad 酷派	 htc	 NOKIA 诺基亚	 ZTE中兴
 lenovo 联想	 MEIZU	 SONY make.believe		 nubia 努比亚	 DAXIAN	 Uniscope

价格：0-199 200-399 400-799 800-1499 1500-2899 2900-4099 4100以上

网络：移动4G (TD-LTE) 联通3G (WCDMA) 移动3G (TD-SCDMA) 电信3G (CDMA2000)

屏幕尺寸：5.6英寸及以上 5.5-5.0英寸 4.9-4.1英寸 4.0-3.1英寸 3.0英寸及以下

更多选项 (机身颜色, 系统, 特点) ▾

京东上的一个手机筛选系统, 用户可以根据品牌、价格、网 络、屏幕尺寸等条件, 对商品进行筛选。

对于每个条件的每个选项, 我们可以使用一个集合来保存 该选项对应的所有商品, 并通过在多个选项之间进行交集计算, 从而达到筛选的目的。

示例：构建商品筛选功能

对于品牌条件，我们可以创建这样的集合，比如 `apple = { 'iPhone 5c', 'iPhons 5s', ... }`，而 `samsung = { 'Galaxy S5', 'Galaxy S4', ... }`，诸如此类。

而对于价格条件，我们可以创建这样的集合，比如 `price_2900_to_4099 = { 'Galaxy S5', 'Galaxy S4', ... }`，以及 `price_uppon_4100 = { 'iPhone 5c', 'iPhone 5S', ... }`，诸如此类。

每当用户添加一个过滤选项时，我们就计算出所有被选中选项的交集，而交集的计算结果就是符合筛选条件的商品。

示例：构建商品筛选功能

手机 - 商品筛选

已选条件：品牌：苹果（Apple）× 价格：5200以上 × 网络：联通3G（WCDMA）×

排序：销量 价格 评论数 上架时间

库存：北京朝阳区管庄 ▾ ☒ 仅显示有货 商品类型：☒ 全部 ☐ 京东配送 ☐ 第三方配送



苹果（APPLE）iPhone 5s 32G版 4G手机（金色）TD-LTE/TD-SCDMA/WCDMA

¥5888.00

已有504人评价

北京有货

加入购物车

关注

☐ 对比

ZINTER apple price_uppon_5200 unicom_wcdma ‘iPhone 5s

32G’

当然，每次过滤都要计算一次交集的话，速度就太慢了，因此我们可以预先计算好，然后把结果储存在一个固定的地方，比如：

ZINTERSTORE

apple&price_uppon_5200&unicom_wcdma

apple price_uppon_5200 unicom_wcdma

SMEMBER

apple&price_uppon_5200&unicom_wcdma

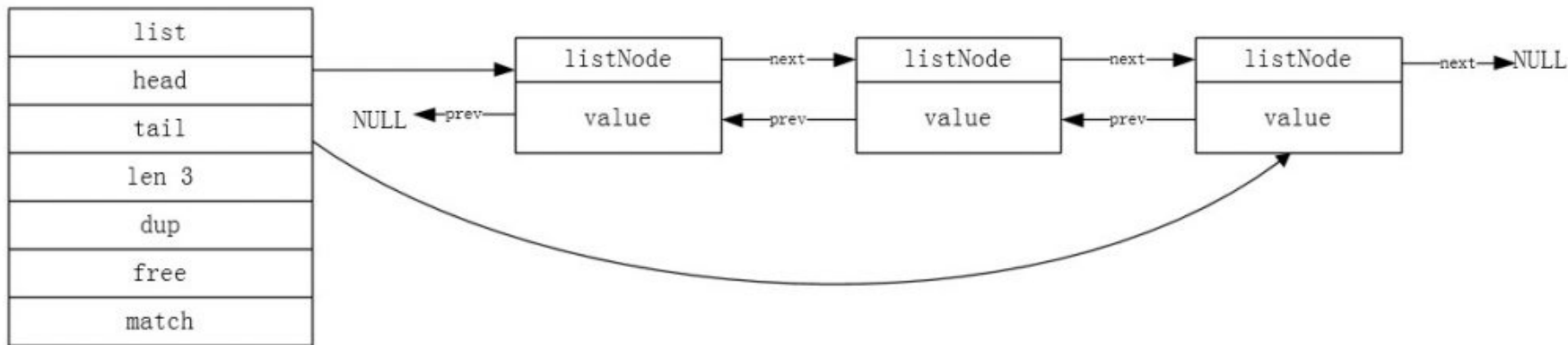
商品筛选功能的实现及其 API

API	作用	实现原理
ItemFilter(client)	指定客户端。	
ItemFilter.add_option(item_set)	添加一个筛选条件选项, item_set 集合储存了所有符合该选项的商品。	把 item_set 添加到一个集合里面, 等待进行筛选。
ItemFilter.result()	根据目前已给定的选项进行筛选, 并返回符合条件的商品。	使用 SINTER, 对所有已给定选项的集合进行计算。
ItemFilter.store_result(key)	根据目前已给定的选项进行筛选, 并把筛选结果储存到给定的键里面。	使用 SINTERSTORE, 计算并储存交集结果。

Redis 列表(List)

Redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部(左边)或者尾部

(右边)一个列表最多可以包含 $2^{32} - 1$ 个元素 (4294967295, 每个列表超过40亿个元素)。



Redis 的链表的实现的主要特性如下：

- 1.双端:链表节点都有 prev 和 next 指针, 这样获取一个节点的前置节点和后置节点的算法复杂度都为 $O(1)$ 。
- 2.无环:list 的第一个节点(头节点)的 prev 和最后一个节点(尾节点)的 next 都指向 NULL。
- 3.带表头指针和表尾指针:通过 list 的 head 和 tail 两个指针, 可以随意的从链表的头和尾进行操作。
- 4.带链表长度计数器:可以通过 len 成员来获取链表的节点的个数, 复杂度 $O(1)$ 。
- 5.多态:链表使用 void *指针来保存 value, 并且可以通过 dup, free, match 来操控节点的 value 值, 因此, 该链表可以保存任意类型的值。

一、PUSH操作

1, RPUSH key value [value ...]

从队列的右边入队一个元素或多个元素，复杂度 $O(1)$ 。

将所有指定的值插入存于key的列表的尾部（从右侧插入）。如果key不存在，那么PUSH之前，会先自动创建一个空的列表。如果key对应的值不是一个list的话，则会返回一个错误。如果同时push多个值的话，值会依次从左到右PUSH从尾部进入list。

PUSH和POP操作，其实是队列的基本操作。Redis的list就是一个极其强大的队列系统。我们在哪些地方会用到队列呢？下面，我们说两个例子：

a, 评论系统

逛过微博的筒子们应该都对评论系统有了解。我们在看完一条微博之后，常常会评论一番，或者看看其他人的吐槽。每条评论的记录都是按照时间顺序排序的。我们读的时候也是这个顺序。这时，队列就是一个很好的存储结构。每提交一次评论，都向list的末尾添加一个新的节点。

当然，博客本身也可以是这样的结构。

b, 并行转串行

我们做后台开发的筒子们应该都遇到过类似的情景。用户每时每刻都可能发出请求，而且请求的频率经常变化。这时，后台的程序不可能立刻响应每一个用户的请求，尤其是请求特别占资源的服务的时候（双11的时候，你有没有看到404页面？）。有什么好的办法来解决这个问题呢？我们需要一个排队系统。根据用户的请求时间，将用户的请求放入队列中，后台程序依次从队列中获取任务，处理并将结果返回到结果队列。这其实也是一个生产者消费者模型。通过队列，我们将并行的请求转换成串行的任务队列，之后依次处理（当然后台的程序也可以多核并行处理）。

```
redis> rpush queue a
(integer) 1
redis> rpush queue b
(integer) 2
redis> rpush queue c
(integer) 3
redis> rpush queue d e f
(integer) 6
redis> lrange queue 0 -1
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
```

lrange这个命令是获取指定范围的list中的数据，我们下面再具体介绍。

这个例子中，我们依次将a、b、c、d、e、f、g从尾部(右侧)追加到queue中，最后通过lrange查看queue中的数据的顺序。

2, LPUSH key value [value ...]

从队列的左边入队一个或多个元素，复杂度O(1)。

这个指令和RPUSH几乎一样，只是插入节点的方向相反了，是从list的头部(左侧)进行插入的。

```
1 redis> del queue
2 (integer) 1
3 redis> lpush queue a
4 (integer) 1
5 redis> lpush queue b
6 (integer) 2
7 redis> lpush queue c
8 (integer) 3
9 redis> lpush queue e f g
10 (integer) 6
11 redis> lrange queue 0 -1
12 1) "g"
13 2) "f"
14 3) "e"
15 4) "c"
```

```
16 5) "b"  
17 6) "a"
```

可以看出，结果正好和RPUSH相反。

3, RPUSHX key value

从队列的右边入队一个元素，仅队列存在时有效。当队列不存在时，不进行任何操作。

```
1 # 之前queue已经存在，且有a、b、c、d、e、f、g这6个元素  
2 redis> rpushx queue z  
3 (integer) 7  
4 redis> del queue  
5 (integer) 1  
6 redis> rpushx queue z  
7 (integer) 0
```

可以看出，最开始rpushx向queue中新增了一个节点，但当我们删掉了queue时，再rpushx，就没有插入成功（返回值为0）。

4, LPUSHX key value

从队列的左边入队一个元素，仅队列存在时有效。当队列不存在时，不进行任何操作。

参考RPUSHX。

二、POP操作

PUSH操作，是从队列头部和尾部增加节点的操作。而POP是获取并删除头尾节点的操作。

1, LPOP key

从队列的左边出队一个元素，复杂度 $O(1)$ 。如果list为空，则返回nil。

```
1 redis> del queue
2 (integer) 0
3 redis> rpush queue a b c d e f
4 (integer) 6
5 redis> lrange queue 0 -1
6 1) "a"
7 2) "b"
8 3) "c"
9 4) "d"
10 5) "e"
11 6) "f"
12 redis> lpop queue
13 "a"
14 redis> lpop queue
15 "b"
16 redis> lrange queue 0 -1
17 1) "c"
18 2) "d"
```

```
19 3) "e"
20 4) "f"
21 redis> rpop queue
22 "f"
23 redis> rpop queue
24 "e"
25 redis> lrange queue 0 -1
26 1) "c"
27 2) "d"
```

我们首先向空的list中添加了a、b、c、d、e、f这6个值，之后从左边POP(LPOP)出两个值，再从右侧POP(RPOP)出两个值。

2, RPOP key

从队列的右边出队一个元素，复杂度 $O(1)$ 。如果list为空，则返回nil。

见LPOP。

3, BLPOP key [key ...] timeout

删除，并获得该列表中的第一元素，或阻塞，直到有一个可用。

这是LPOP的阻塞版本。在LPOP的时候，如果队列中没有值，则会返回一个nil。而BLPOP则会等待一段时间，如果list中有值（等待的时候，被添加的），则返回对应值；如果在给定时间内仍没有得到结果，则返回nil。

```
1 redis> lrange queue 0 -1
2 1) "c"
3 2) "d"
4 redis> BLPOP queue 1
5 1) "queue"
6 2) "c"
7 redis> BLPOP queue 1
8 1) "queue"
9 2) "d"
10 redis> BLPOP queue 1
11 (nil)
12 (1.10s)
13 redis> LPOP queue
14 (nil)
```

我们仍接着上面的实验继续，这时queue里面只有2个元素了，我们使用BLPOP取值，前两次都成功地得到了值，效果和LPOP一样。但第三次的时候，由于list已经为空，但是BLPOP并没有立刻返回nil，而是阻塞了一点时间(timeout的时间)，之后才返回了nil。最后，我们试验了一下LPOP，证实了LPOP是立刻返回结果的。timeout表示等待的时间，单位是秒。当设为0时，表示永远阻塞，非0时，表示等待的最长时间。

要注意的是，LBPOP支持多个key，也就是说可以同时监听多个list，并按照key的顺序，依次检查list是否为空，如果不为空，则返回最优先的list中的值。如果都为空，则阻塞，直到有一个list不为空，那么返回这个list对应的值。这里进行试验不是特别的方便，更具体的介绍可以查看中文官网的文档：

<http://redis.cn/commands/blpop.html>。

4, BRPOP key [key ...] timeout

删除，并获得该列表中的最后一个元素，或阻塞，直到有一个可用。

参考BLPOP。

三、POP and PUSH

1, RPOPLPUSH source destination

删除列表中的最后一个元素，将其追加到另一个列表。

这个命令可以原子性地返回并删除source对应的列表的最后一个元素（尾部元素），并将钙元素放入destination对应的列表的第一个元素位置（列表头部）。

```
1 redis> rpush q1 1 2 3 4 5
```

```
2 (integer) 5
```

```
3 redis> lrange q1 0 -1
4 1) "1"
5 2) "2"
6 3) "3"
7 4) "4"
8 5) "5"
9 redis> rpoplpush q1 q2
10 "5"
11 redis> rpoplpush q1 q2
12 "4"
13 redis> lrange q1 0 -1
14 1) "1"
15 2) "2"
16 3) "3"
17 redis> lrange q2 0 -1
18 1) "4"
19 2) "5"
```

发布与订阅

定义与模型

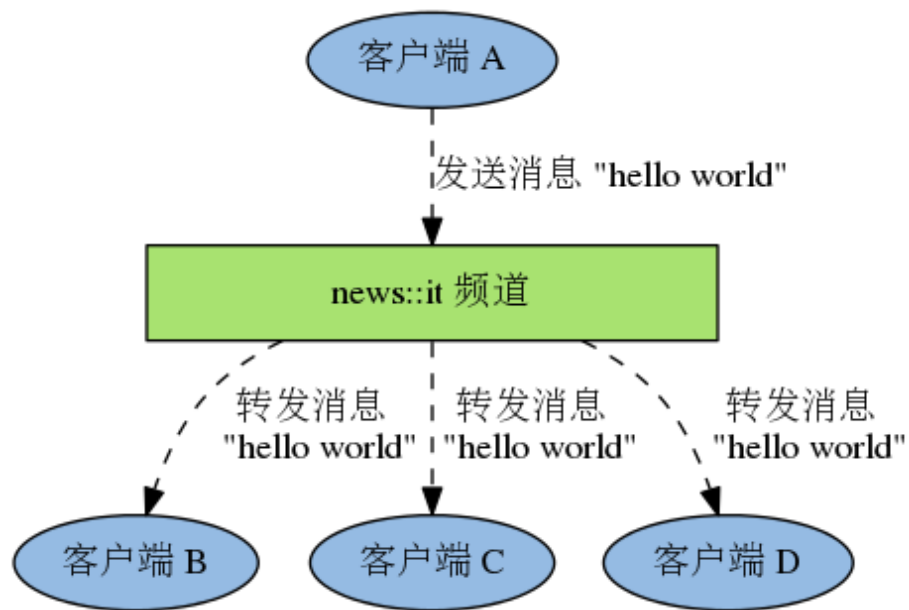
发布与订阅功能

Redis 的发布与订阅功能可以让用户将消息同时发送给多个客户端。

这个功能由几个不同的角色协作组成：

- 发布者(publisher)：发布消息的客户端。
- 频道(channel)：构建在服务器内部，负责接收发布者发送的消息，并将消息转发给频道的订阅者。
- 模式(pattern)：构建在服务器内部，负责对频道进行匹配，当被匹配的频道接到消息时，模式也会将消息转发给模式的订阅者。
- 订阅者(subscriber)：通过订阅频道或者模式来获取消息的客户端。每个频道或者模式都可以有任意多个订阅者。

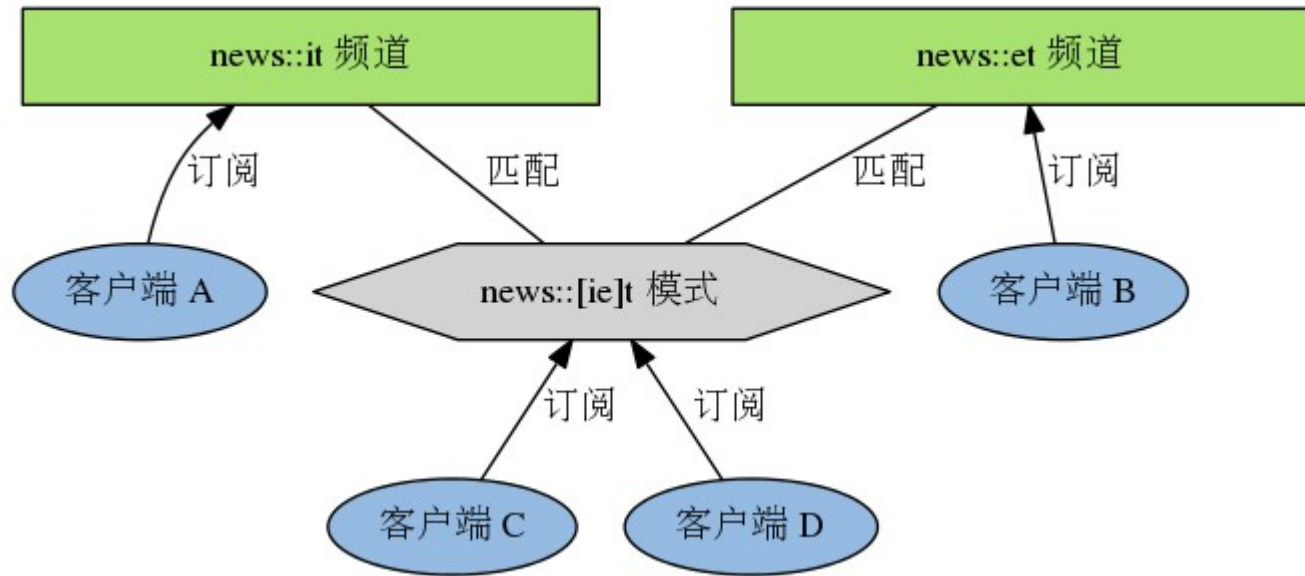
频道的订阅与消息发布



B、C、D 三个客户端正在订阅 news::it 频道。

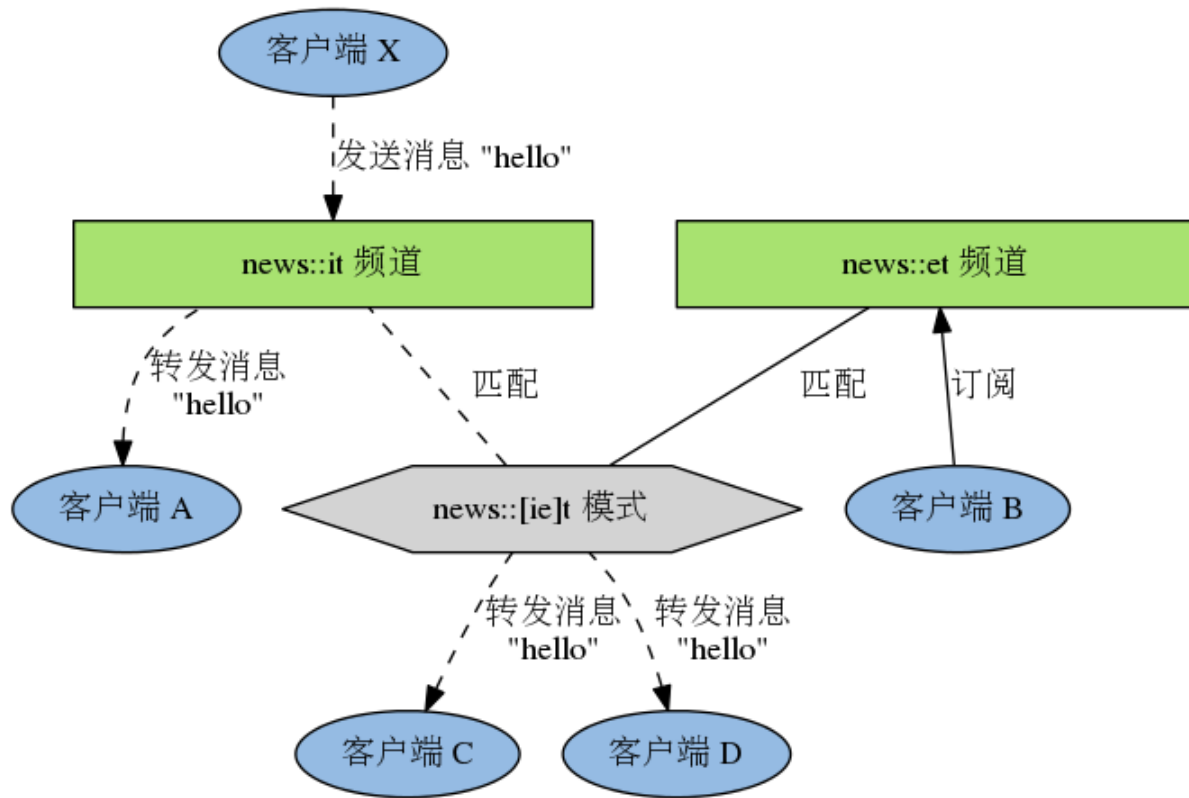
当客户端 A 向 news::it 频道发送消息“hello world”时，该消息将被频道转发至 B、C、D 三个客户端。

模式的订阅与消息发布

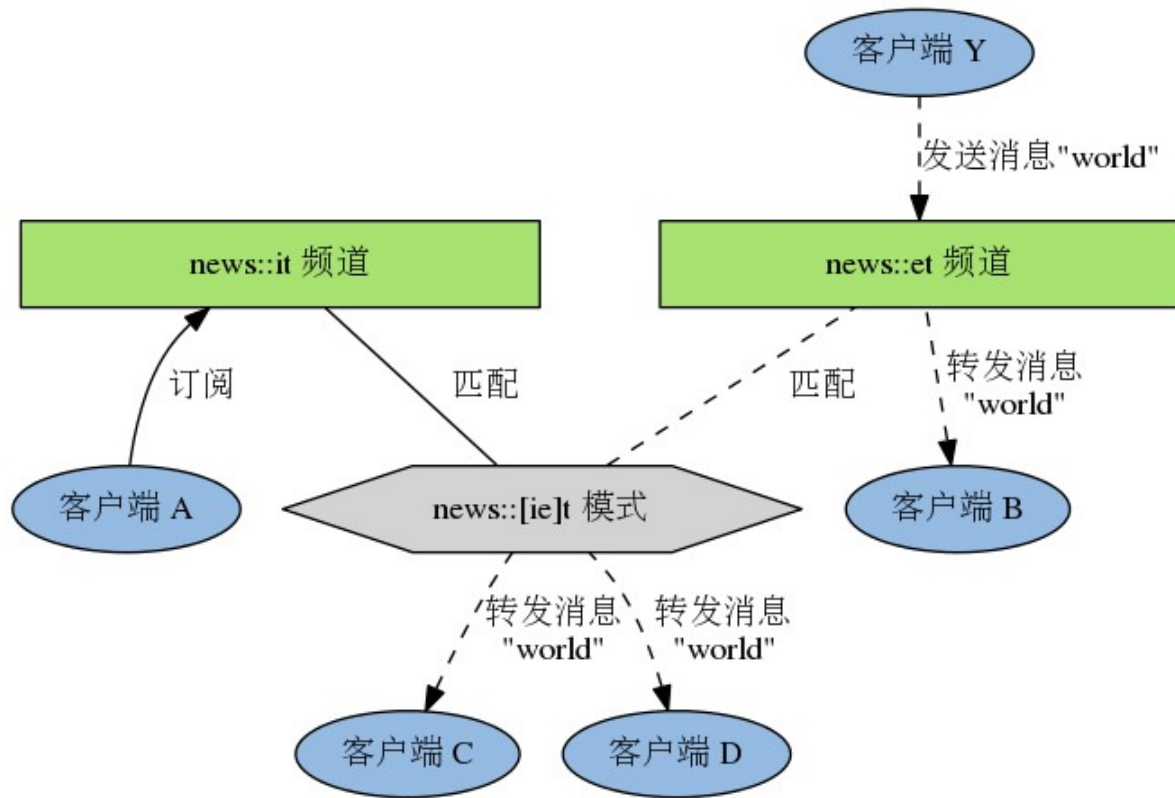


客户端 C 和 D 订阅了 `news::[ie]t 模式`，而这个模式又和 `news::it`、`news::et` 两个频道匹配，当 `news::it` 频道或者 `news::et` 频道接收到消息的时候，这些消息不仅会被转发给频道的订阅者，也会被转发给客户端 C 和 D。

模式订阅者接收消息示例(1)



模式订阅者接收消息示例(2)



订阅频道

SUBSCRIBE channel [channel ...]

订阅给定的一个或多个频道。

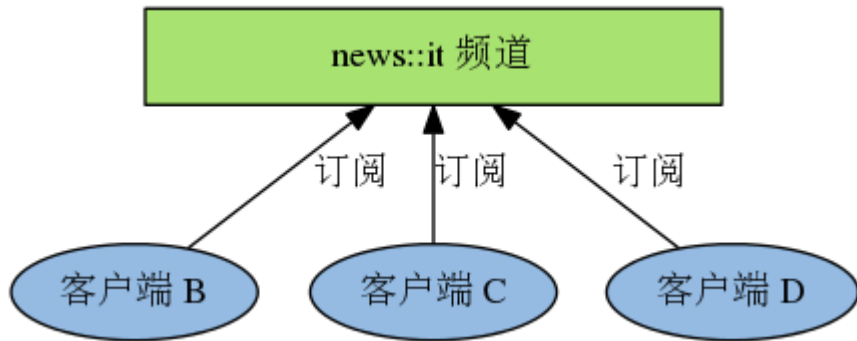
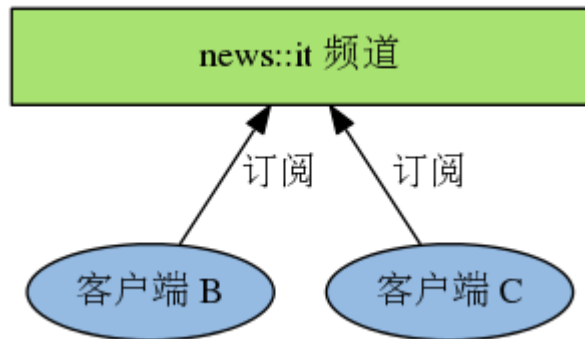
复杂度为 $O(N)$ ， N 为被订阅频道的数量。

```
redis> SUBSCRIBE news::it
```

Reading messages... (press Ctrl-C to quit)

- 1) "subscribe" # 订阅频道时返回的消息
- 2) "news::it" # 被订阅的频道
- 3) (integer) 1 # 客户端目前订阅频道的数量

- 1) "message" # 这是从频道接收到的消息
- 2) "news::it" # 消息来源的频道
- 3) "hello" # 消息内容



订阅模式

PSUBSCRIBE pattern [pattern ...]

订阅一个或多个模式，pattern 参数可以包含 glob 风格的匹配符，比如：

- news::* 模式可以匹配 news::bussiness、news::it、news::sports::football 等频道；
- news::[ie]t 模式可以匹配 news::it 频道或者 news::et 频道；
- news::?t 模式可以匹配 news::it、news::et、news::at 等频道；

诸如此类。

复杂度为 $O(N)$ ， N 为被订阅模式的数量。

订阅模式示例

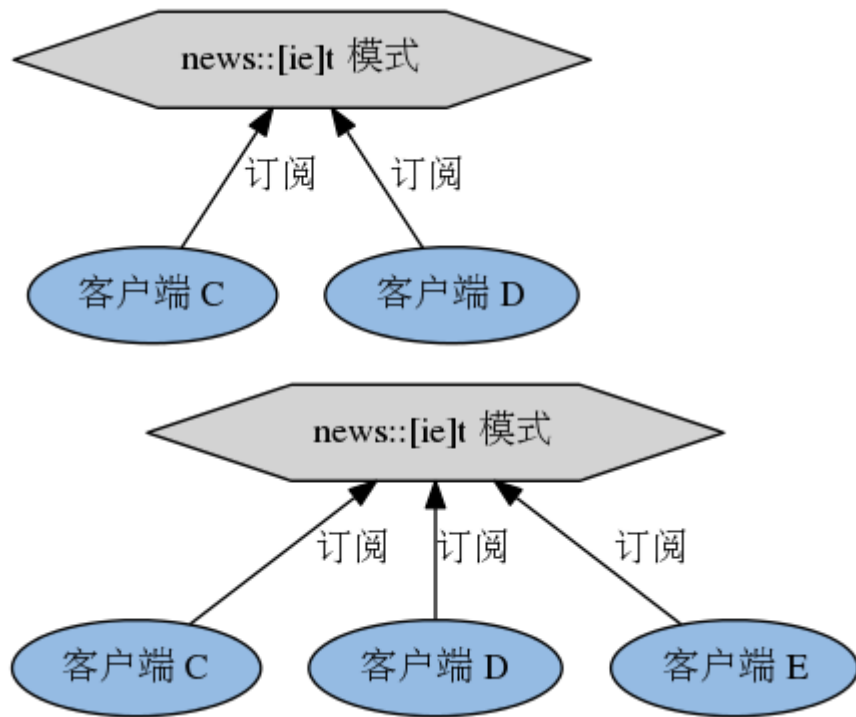
```
redis> PSUBSCRIBE news::[ie]t
```

Reading messages... (press Ctrl-C to quit)

- 1) "psubscribe" # 订阅模式时返回的信息
- 2) "news::[ie]t" # 被订阅的模式
- 3) (integer) 1 # 客户端目前订阅的模式数量

- 1) "pmessage" # 这是从模式接收到的消息
- 2) "news::[ie]t" # 被匹配的模式
- 3) "news::it" # 消息的来源频道(被匹配的频道)
- 4) "hello" # 消息正文

- 1) "pmessage"
- 2) "news::[ie]t"
- 3) "news::et"
- 4) "world"



退订频道和退订模式

命令	作用	复杂度
UNSUBSCRIBE [channel [channel ...]]	退订指定的频道。 如果执行时没有指定任何频道， 那么退订已订阅的所有频道。	$O(N)$, N 为被退订的 频道数量。
PUNSUBSCRIBE [pattern [pattern ...]]	退订指定的模式。 如果执行时没有指定任何模式， 那么退订已订阅的所有模式。	$O(M)$, M 为服务器中 被订阅模式的数量。

退订命令的行为在各个客户端的表现都不同，比如 redis-cli 客户端就是通过直接退出客户端来进行退订的，而 Python 和 Ruby 的客户端则需要显示地执行退订命令。

退订示例

redis-cli 客户端

```
redis> SUBSCRIBE news.it
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news::it"
3) (integer) 1
^C
```

Python 的 Redis 客户端

```
>>> from redis import Redis
>>> client = Redis()
>>> pubsub = client.pubsub()
>>> pubsub.subscribe('news::it')      # 订阅频道
>>> pubsub.channels                    # 列出已订阅的频道
set(['news::it'])
>>> pubsub.unsubscribe('news::it') # 退订频道
>>> pubsub.channels
set([])
```


发布消息

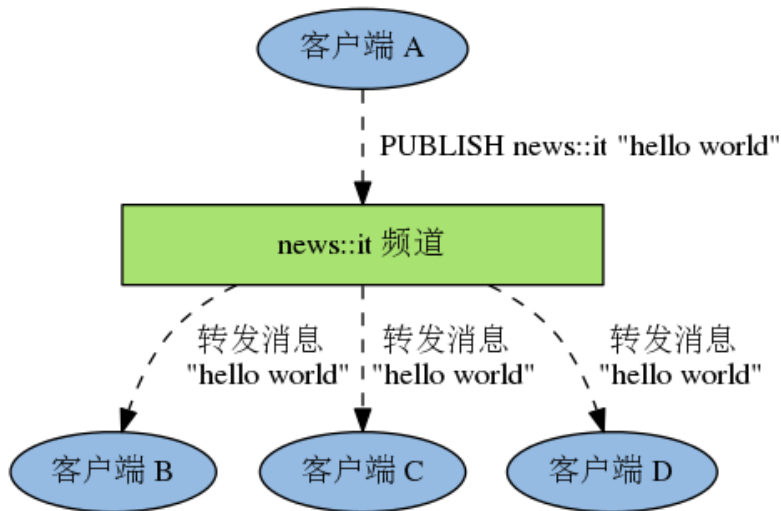
PUBLISH channel message

将消息发送至指定的频道, 命令返回接收到消息的订阅者数量。

复杂度为 $O(N)$, N 为接收到消息的订阅者数量(包括通过订阅频道来接收消息的订阅者和通过订阅模式来接收消息的订阅者)。

```
redis> PUBLISH news::it "hello world"  
(integer) 2
```

```
redis> PUBLISH news::et "hello again"  
(integer) 1
```



查看被订阅的频道

PUBSUB CHANNELS [pattern]

列出目前至少有一个订阅者的频道。

如果给定了可选的 pattern 参数, 那么只列出与模式相匹配的 频道。

复杂度为 $O(N)$, N 为服务器中被订阅频道的总数量。

```
redis> PUBSUB CHANNELS          # 没有任何频道被订阅  
(empty list or set)
```

```
redis> PUBSUB CHANNELS          # 有客户端正在订阅news::et 和 news::it 频道  
1) "news::et"  
2) "news::it"
```

查看频道的订阅者数量

PUBSUB NUMSUB [channel-1 ... channel-N]

返回给定频道的订阅者数量。

复杂度为 $O(N)$ ， N 为给定频道的数量。

```
redis> PUBSUB NUMSUB news::it news::et
```

1) "news::it" # 有两个客户端正在订阅news::it 频道

2) "2"

3) "news::et" # 有一个客户端正在订阅news::et 频道

4) "1"

查看被订阅模式的数量

PUBSUB NUMPAT

返回服务器目前被订阅的模式数量。

复杂度为 $O(1)$ 。

```
redis> PUBSUB NUMPAT
```

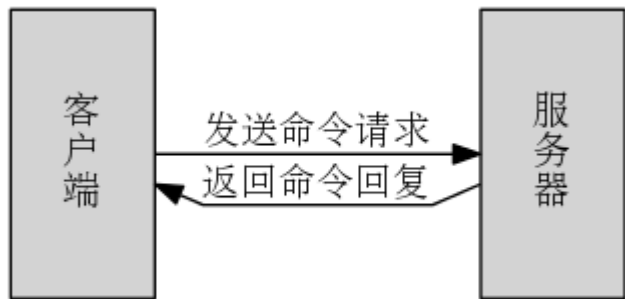
```
(integer) 3    # 服务器目前有三个模式被订阅
```

流水线功能

通过减少客户端与服务器之间的通信次数来提高程序的执行效率

通信

在一般情况下，用户每执行一个 Redis 命令，客户端与服务器都需要进行一次通信：客户端会将命令请求发送给服务器，而服务器则会将执行命令所得的结果返回给客户端。



当程序执行一些复杂的操作时，客户端可能需要执行多个命令，并与服务器进行多次通信。

多次通信示例(1/2)

举个例子，假设我们正在构建一个为图书打标签(tag)的网站，这个网站上的每本图书都可以被打上任意多个标签。

并且为了记录哪些标签的图书是最多人阅览的，我们会为每个标签创建一个点击计数器，每当用户浏览网站上的某本书时，程序就会对该书拥有的各个标签的点击计数器执行增一操作。

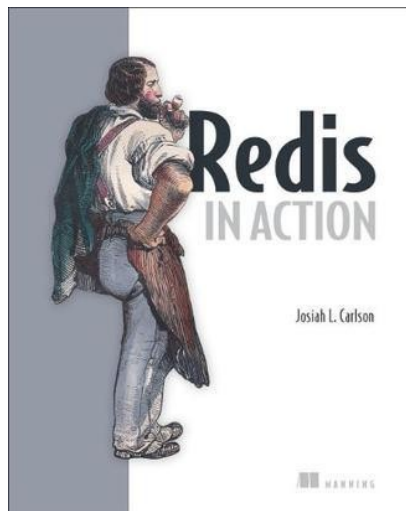
举个例子，对于《Redis in Action》这本书，用户可能会给它打上“计算机”、“编程”、“数据库”和“Redis”这四个标签，每次当《Redis in Action》的页面被访问时，程序就会对这四个标签的点击计数器执行增一操作：

```
INCR tag::计算机::click_counter
```

```
INCR tag::编程::click_counter
```

```
INCR tag::数据库::click_counter
```

```
INCR tag::Redis::click_counter
```



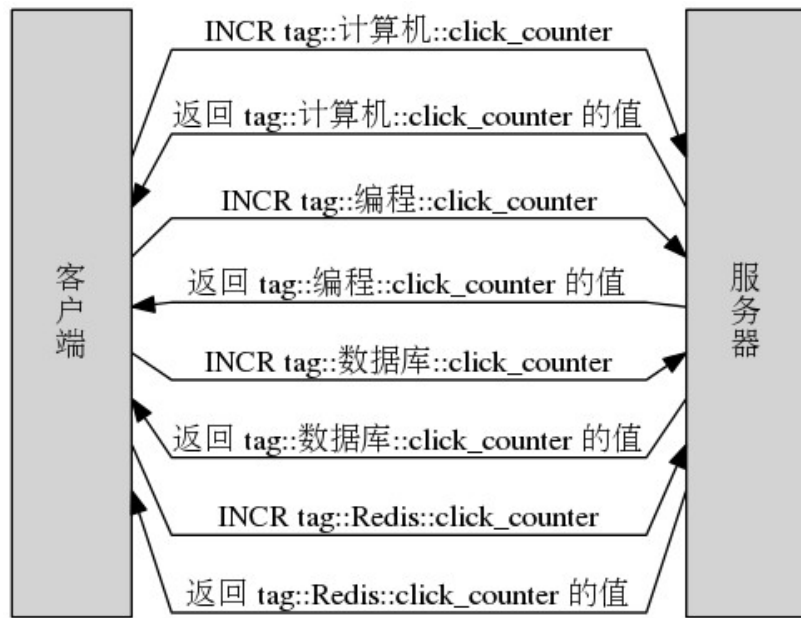
多次通信示例(2/2)

对于这个点击计数程序来说, 客户端要执行的 INCR 命令次数取决于书本的标签数量, 而标签的数量越多, 要执行的 INCR 命令就越多。

右图就展示了《Redis in Action》的页面被查看时, 客户端和服务端之间进行的通信情况。

因为Redis 服务器的性能非常高, 所以当一个命令请求抵达服务器之后, 服务器通常很快就会将命令执行完毕, 并向客户端返回命令的执行结果。

在多数情况下,
客户端在执行 Redis 命令时,
大部分等待时间都耗费在了
发送命令请求和接收命令回复上面。



流水线功能

对于前面展示的点击计数器这种需要执行多个命令的程序来说，如果我们能减少程序执行时，客户端与服务器之间的通信次数，就能够有效地提升程序的性能，而 Redis 的流水线功能(pipeline)就是为此而设置的。

Redis 的流水线功能允许客户端一次将多个命令请求发送给服务器，并将被执行的多个命令请求的结果在一个命令回复中全部返回给客户端，使用这个功能可以有效地减少客户端在执行多个命令时需要与服务器进行通信的次数。

流水线应用示例

以前面列出的点击计数器为例，我们可以将多条 INCR 命令都包裹到一个流水线里面执行，使得程序在执行 N 个 INCR 命令时所需的通信次数从 N 次降低为一次。

下图展示了流水线功能是如何将更新《Redis in Action》各个标签的点击计数器所需的通信次数从四次降低为一次的。



Lua 脚本

在服务器端执行复杂的操作

流水线:打包发送多条命令,并在一个回复里面接收所有被执行命令的结果。

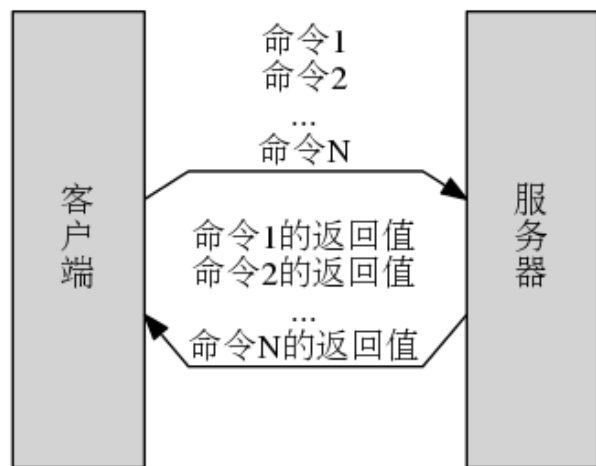
事务:一次执行多条命令,被执行的命令要么就全部都被执行,要么就一个也不执行。并且事务执行过程中不会被其他工作打断。

乐观锁:监视特定的键,防止事务出现竞争条件。(watch)

虽然这些附加功能都非常有用,但它们也有一些缺陷。

流水线的缺陷

尽管使用流水线可以一次发送多个命令，但是对于一个由多个命令组成的复杂操作来说，为了执行该操作而不断地重复发送相同的命令，这并不是最高效的做法，会对网络资源造成浪费。



如果我们有办法避免重复地发送相同的命令，那么客户端就可以减少花在网络传输方面的时间，操作就可以执行得更快。

事务和乐观锁的缺陷

虽然使用事务可以一次执行多个命令，并且通过乐观锁可以防止事务产生竞争条件，但是在实际中，要正确地使用事务和乐观锁并不是一件容易的事情。

1. 对于一个复杂的事务来说，通常需要仔细思考才能知道应该对哪些键进行加锁：锁了不应该锁的键会增加事务失败的机会，甚至可能会造成程序出错；而忘了对应该锁的键进行加锁的话，程序又会产生竞争条件。
2. 有时候为了防止竞争条件发生，即使操作本身不需要用到事务，但是为了让乐观锁生效，我们也会使用事务将命令包裹起来，这增加了实现的复杂度，并且带来了额外的性能损耗。

避免事务被误用的办法

如果有一种方法，可以让我们以事务方式来执行多个命令，并且这种方法不会引入任何竞争条件，那么我们就可以使用这种方法来代替事务和乐观锁。

Lua 脚本

为了解决以上提到的问题，Redis 从 2.6 版本开始在服务器内部嵌入了一个 Lua 解释器，使得用户可以在服务器端执行 Lua 脚本。

这个功能有以下好处：

1. 使用脚本可以直接在服务器端执行 Redis 命令，一般的数据处理操作可以直接使用 Lua 语言或者 Lua 解释器提供的函数库来完成，不必再返回给客户端进行处理。
2. 所有脚本都是以事务的形式来执行的，脚本在执行过程中不会被其他工作打断，也不会引起任何竞争条件，完全可以使用 Lua 脚本来代替事务和乐观锁。
3. 所有脚本都是可重用的，也即是说，重复执行相同的操作时，只要调用储存在服务器内部的脚本缓存就可以了，不用重新发送整个脚本，从而尽可能地节约网络资源。

执行 Lua 脚本

EVAL script numkeys key [key ...] arg [arg ...]

script 参数是要执行的 Lua 脚本。

numkeys 是脚本要处理的数据库键的数量，之后的 key [key ...] 参数指定了脚本要处理的数据库键，被传入的键可以在脚本里面通过访问 KEYS 数组来取得，比如 KEYS[1] 就取出第一个输入的键，KEYS[2] 取出第二个输入的键，诸如此类。

arg [arg ...] 参数指定了脚本要用到的参数，在脚本里面可以通过访问 ARGV 数组来获取这些参数。

显式地指定脚本里面用到的键是为了配合 Redis 集群对键的检查，如果不这样做的话，在集群里面使用脚本可能会出错。

另外，通过显式地指定脚本要用到的数据库键以及相关参数，而不是将数据库键和参数硬写在脚本里面，用户可以更方便地重用同一个脚本。

EVAL 命令使用示例

```
redis> EVAL "return 'hello world'" 0 "hello  
world"
```

```
redis> EVAL "return 1+1" 0  
(integer) 2
```

```
redis> EVAL "return {KEYS[1], KEYS[2], ARGV[1], ARGV[2]}" 2 "msg" "age" 123 "hello world"
```

```
1) "msg"      # KEYS[1]  
2) "age"      # KEYS[2] 3)  
"123"         # ARGV[1]  
4) "hello world" # ARGV[2]
```

在 Lua 脚本中执行 Redis 命令

通过调用 `redis.call()` 函数或者 `redis.pcall()` 函数，我们可以直接在 Lua 脚本里面执行 Redis 命令。

```
redis> EVAL "return redis.call('PING')" 0      # 在 Lua 脚本里面执行 PING 命令
PONG
```

```
redis> EVAL "return redis.call('DBSIZE')" 0      # 在 Lua 脚本里面执行 DBSIZE 命令
(integer) 4
```

在 Lua 脚本里面执行 GET 命令，取出键 msg 的值，并对值进行字符串拼接操作

```
redis> SET msg "hello world" OK
redis> EVAL "return 'The message is: ' .. redis.call('GET', KEYS[1])" 1 msg "The
message is: hello world"
```

redis.call() 和 redis.pcall() 的区别

`redis.call()` 和 `redis.pcall()` 都可以用来执行 Redis 命令，它们的不同之处在于，当被执行的脚本出错时，`redis.call()` 会返回**出错脚本的名字**以及 EVAL 命令的错误信息，而 `redis.pcall()` 只返回 EVAL 命令的错误信

息。

```
redis> EVAL "return redis.call('NotExistsCommand')" 0
```

```
(error) ERR Error running script (call to f_ddabd662fa0a8e105765181ee7606562c1e6f1ce): @user_script:1:
```

```
@user_script: 1: Unknown Redis command called from Lua script
```

```
redis> EVAL "return redis.pcall('NotExistsCommand')" 0
```

```
(error) @user_script: 1: Unknown Redis command called from Lua script
```

换句话说，在被执行的脚本出错时，`redis.call()` 可以提供更详细的错误信息，方便进行查错。

示例：使用 Lua 脚本重新实现 ZDECRBY 命令

创建一个包含以下内容的 `zdecrby.lua` 文件：

```
local old_score = redis.call('ZSCORE', KEYS[1], ARGV[2])
```

```
local new_score = old_score - ARGV[1]
return redis.call('ZADD', KEYS[1], new_score, ARGV[2])
```

然后通过以下命令来执行脚本：

```
$ redis-cli --eval zdecrby.lua salary , 300 peter
(integer) 0
```

这和在 redis-cli 里面执行 EVAL “local ...” 1 salary 300 peter 效果一样，但先将脚本内容保存到文件里面，再执行脚本文件的做法，比起直接在客户端里面一个个字输入要容易一些。

另外，这个脚本实现的 ZDECRBY 也比使用事务和乐观锁实现的 ZDECRBY 要简单得多。

使用 EVALSHA 来减少网络资源损耗

任何 Lua 脚本，只要被 EVAL 命令执行过一次，就会被储存到服务器的脚本缓存里面，用户只要通过 EVALSHA 命令，指定被缓存脚本的 SHA1 值，就可以在不发送脚本的情况下，再次执行脚本：

```
EVALSHA sha1 numkeys key [key ...] arg [arg ...]
```

通过 SHA1 值来重用返回 'hello world' 信息的脚本：

```
redis> EVAL "return 'hello world'" 0
```

```
"hello world"
```

```
redis> EVALSHA 5332031c6b470dc5a0dd9b4bf2030dea6d65de910 "hello world"
```

通过 SHA1 值来重用之前实现的 ZDECRBY 命令，这样就不用每次都发送整个脚本了：

```
redis> EVALSHA 918130cae39ff0759b8256948742f77300a91cb2 1 salary 500 peter (integer) 0
```

脚本管理命令

SCRIPT EXISTS sha1 [sha1 ...]

检查 sha1 值所代表的脚本是否已经被加入到脚本缓存里面，是的话返回 1，不是的话返回 0。

SCRIPT LOAD script

将脚本储存到脚本缓存里面，等待将来EVALSHA 使用。

SCRIPT FLUSH

清除脚本缓存储存的所有脚本。

SCRIPT KILL

杀死运行超时的脚本。如果脚本已经执行过写入操作，那么还需要使用 SHUTDOWN NOSAVE 命令来强制服务器不保存数据，以免错误的数据被保存到数据库里面。

函数库

Redis 在 Lua 环境里面载入了一些常用的函数库, 我们可以使用这些函数库, 直接在脚本里面处理数据, 它们分别是标准库:

- base 库: 包含 Lua 的核心(core)函数, 比如 assert、tostring、error、type 等。
- string 库: 包含用于处理字符串的函数, 比如 find、format、len、reverse 等。
- table 库: 包含用于处理表格的函数, 比如 concat、insert、remove、sort 等。
- math 库: 包含常用的数学计算函数, 比如 abs、sqrt、log 等。
- debug 库: 包含调试程序所需的函数, 比如 sethook、gethook 等。

以及外部库

- struct 库: 在 C 语言的结构和 Lua 语言的值之间进行转换。
- cjson 库: 将 Lua 值转换为 JSON 对象, 或者将 JSON 对象转换为 Lua 值。
- cmsgpack 库: 将 Lua 值编码为 MessagePack 格式, 或者从 MessagePack 格式里面解码出 Lua 值。

另外还有一个用于计算 sha1 值的外部函数 redis.sha1hex

大礼包