Redis 学习教程

书栈(BookStack.CN)

目　录

致谢

当前文档 《Redis 学习教程》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-10-22。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/redis-tutorial

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Redis 简介

redis简介

Redis 是完全开源免费的，是一个高性能的 `key-value` 内存数据库。

> 中文网站 *http://redis.cn*官方网站 *http://redis.io*

Redis 有三个主要的特点，有别于其它很多竞争对手 ：

- Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用
- Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储
- Redis支持数据的备份，即master-slave模式的数据备份。
  Redis优点
- 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
- 支持丰富的数据类型 - Redis支持最大多数开发人员已经知道如列表，集合，可排序集合，哈希等数据类型。这使得在应用中很容易解决的各种问题，因为我们知道哪些问题处理，使用哪种数据类型更好解决。
- 操作都是原子的 - 所有 Redis 的操作都是原子，从而确保当两个客户同时访问 Redis 服务器得到的是更新后的值（最新值）。原子性（atomicity）：一个事务是一个不可分割的最小工作单位，事务中包括的诸操作要么都做，要么都不做。Redis所有单个命令的执行都是原子性的，这与它的单线程机制有关；
  Redis是一个多功能实用工具，可以在很多如:
- 缓存
- 消息传递队列中使用（Redis原生支持发布/订阅），
- 在应用程序中，如：Web应用程序会话，网站页面点击数等任何短暂的数据；

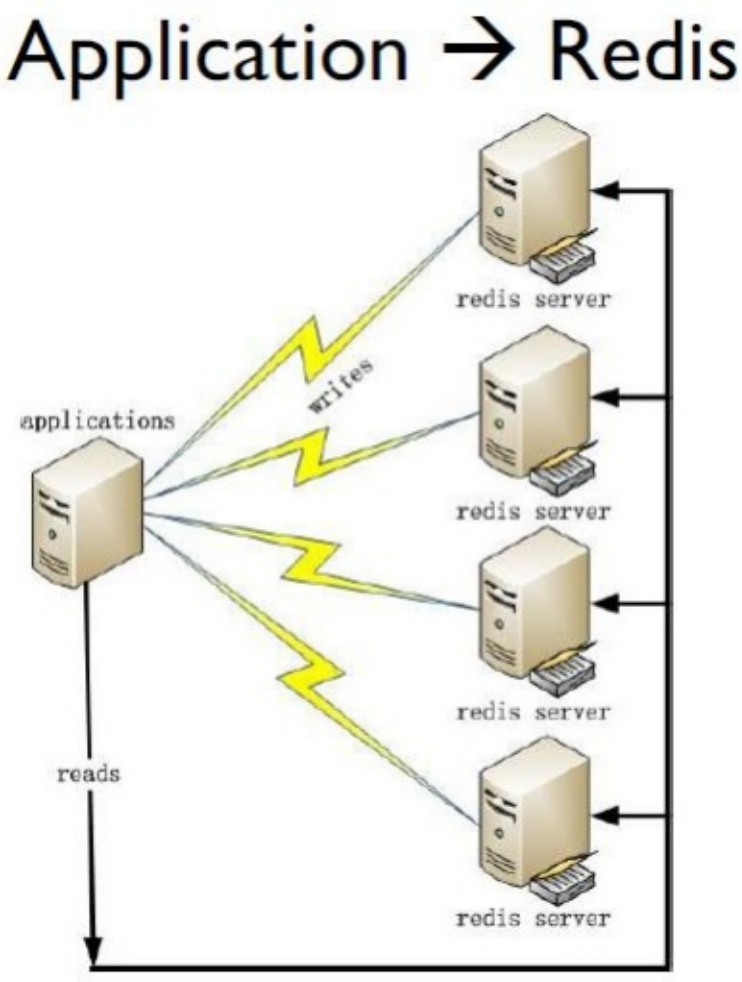
Redis 实际应用案例

目前全球最大的 Redis 用户是新浪微博，公布一下Redis平台实际情况:

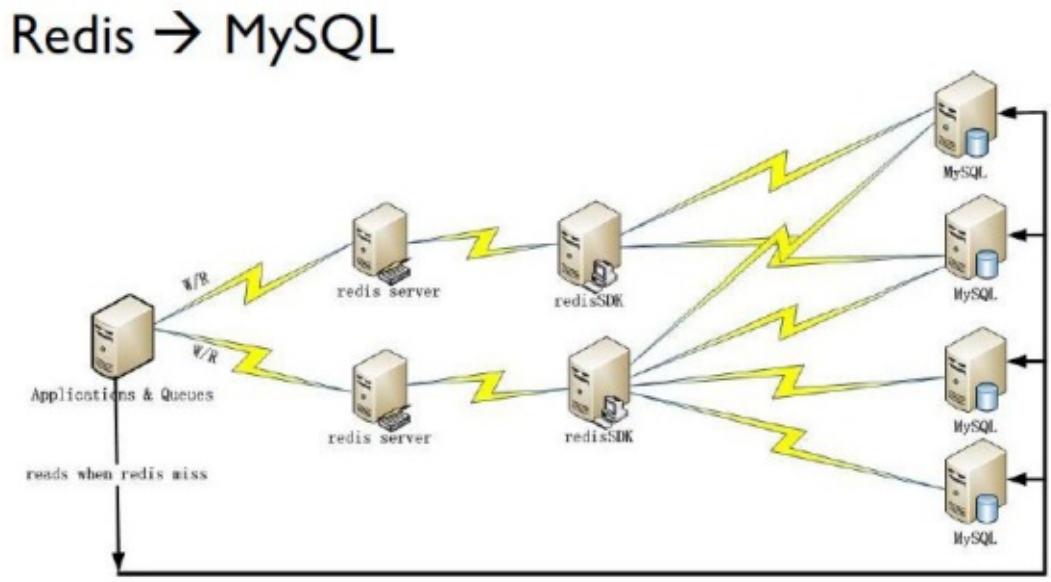
- 2200+亿 commands/day 5000亿Read/day 500亿Write/day
- 18TB+ Memory
- 500+ Servers in 6 IDC（互联网数据中心，机房） 2000+instances

在新浪微博 Redis 的部署场景很多，大概分为如下的 2 种:

第一种是应用程序直接访问 Redis 数据库



第二种是应用程序直接访问 Redis，只有当 Redis 访问失败时才访问 MySQL



> 原文: *https://piaosanlang.gitbooks.io/redis/content/index.html*

1. Redis 环境安装

Redis 环境安装

安装

如果已经安装了老版本3.0.6

```
1. 1. 卸载软件
2. sudo apt-get remove redis-server
3. 2. 清除配置
4. sudo apt-get remove --purge redis-server
5. 3. 删除残留文件
6. sudo find / -name redis
7. 一般设置如下
8. sudo rm -rf var/lib/redis/
9. sudo rm -rf /var/log/redis
10. sudo rm -rf /etc/redis/
11. sudo rm -rf /usr/bin/redis-*
12. sudo rm -rf /usr/local/bin/redis-*
13. sudo rm -rf /usr/local/redis/
```

下载地址：http://redis.io/download，下载最新文档版本，稳定版系列.

- 下载安装文件
  wget http://download.redis.io/releases/redis-3.2.3.tar.gz
- 解压
  tar -zxvf redis-3.2.3.tar.gz
- copy文件并ls查看
  sudo mkdir -p /usr/local/redis/
  sudo cp -r redis-3.2.3/* /usr/local/redis/
  ls /usr/local/redis/
- 进入安装目录
  cd /usr/local/redis/
- 编译

首先打开README.md，翻阅基本build和install方式。

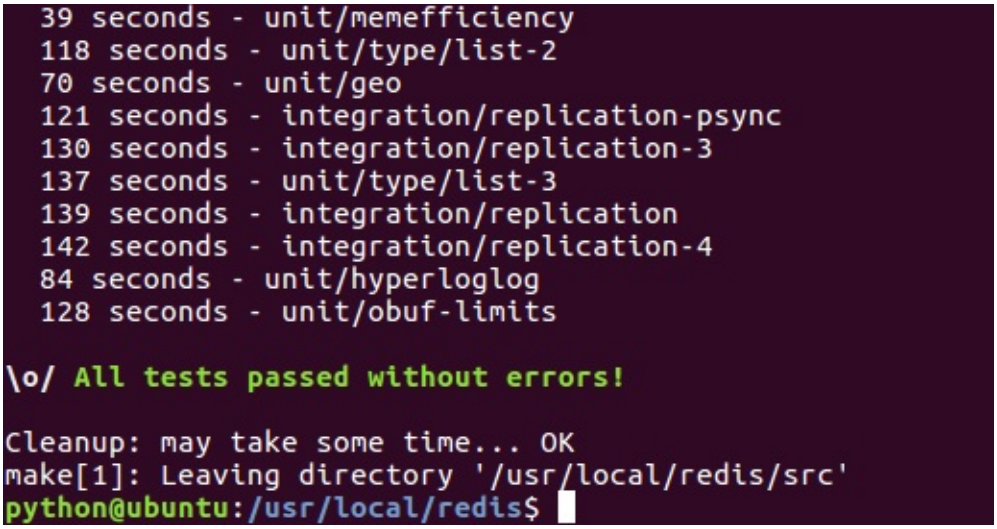
```
vi README.md
sudo make
```

尝试环境是否可以正常使用(Hint: It's a good idea to run 'make test' ;)

```
sudo make test
```

如果出现

```
\o/ All tests passed without errors!
```



表示redis环境没有问题。

- 安装最后安装路径

```
cd src
sudo make install
```

- 查看编译好的命令文件

```
ls /usr/local/bin/redis-*
–/usr/local/bin/redis-benchmark 性能测试工具,例如: redis-benchmark -n 1000000 -c 50 , 50 个客户端, 并发1000000个SETs/GETs查询
–/usr/local/bin/redis-check-aof 更新日志检查
–/usr/local/bin/redis-check-dump 本地数据文件检查
–/usr/local/bin/redis-cli 命令行操作工具
–/usr/local/bin/redis-server 服务器程序
```

- 修改配置文件

```
cd ..
sudo mkdir /etc/redis
sudo cp redis.conf /etc/redis/
ls /etc/redis/redis.conf
```

启动redis

```
python@ubuntu:~$sudo redis-server /etc/redis/redis.conf
```

出现以下代表启动成功

```
python@ubuntu:~$ redis-sever /etc/redis/redis.conf
未找到 'redis-sever' 命令，您要输入的是否是：
命令 'redis-server' 来自于包 'redis-server' (universe)
redis-sever: 未找到命令
python@ubuntu:~$ redis-server /etc/redis/redis.conf
23003:M 17 Sep 10:52:57.535 * Increased maximum number of open files to 10032 (it was
originally set to 1024).
                _.
           .-``__ ''-.
      .-    `  .  `. ''-. Redis 3.2.3 (00000000/0) 64 bit
     .-`` .-  .  \/ .,  ''-.
    (  '  ,  .-  |  ,  ) Running in standalone mode
    |  -._  -…-  __...-.``-._|&#39;  .-'| Port: 6379
    |  `-.  ._ / _.-&#39; | PID: 23003  -.  `-.  -./ _.-&#39; _.-&#39; |  -.`-.  -.__.-&#39; _.-&#39;_.-&#39;|
    |  -.`-. .-'.-' | http://redis.io
     -._  -.`-.__.-'.-'  .-'
    |`-.  -._  -..-'  .-'.-'|
    |  -._  -.  .-'.-'  |
    `-.  -._  -..-'.-'  .-'
     -._  -..-'  .-'
        `-. _.-'
           `-..-'

23003:M 17 Sep 10:52:57.536 # WARNING: The TCP backlog setting of 511 cannot be
enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
23003:M 17 Sep 10:52:57.537 # Server started, Redis version 3.2.3
23003:M 17 Sep 10:52:57.537 # WARNING overcommit_memory is set to 0! Background save
may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1'
to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1'
for this to take effect.
23003:M 17 Sep 10:52:57.537 # WARNING you have Transparent Huge Pages (THP) support
enabled in your kernel. This will create latency and memory usage issues with Redis.
To fix this issue run the command 'echo never >
/sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local
in order to retain the setting after a reboot. Redis must be restarted after THP is
disabled.
23003:M 17 Sep 10:52:57.537 * The server is now ready to accept connections on port
6379
```

连接redis服务器

新开一个窗口，或者redis-server后台启动。

通过

```
1. $ redis-cli
```

得到

```
127.0.0.1:6379>
```

```
127.0.0.1:6379> ping
PONG
```

代表redis服务器已经正常安装并且可以使用了。

> 原文: *https://piaosanlang.gitbooks.io/redis/content/01_huan_jing_an_zhuang.html*

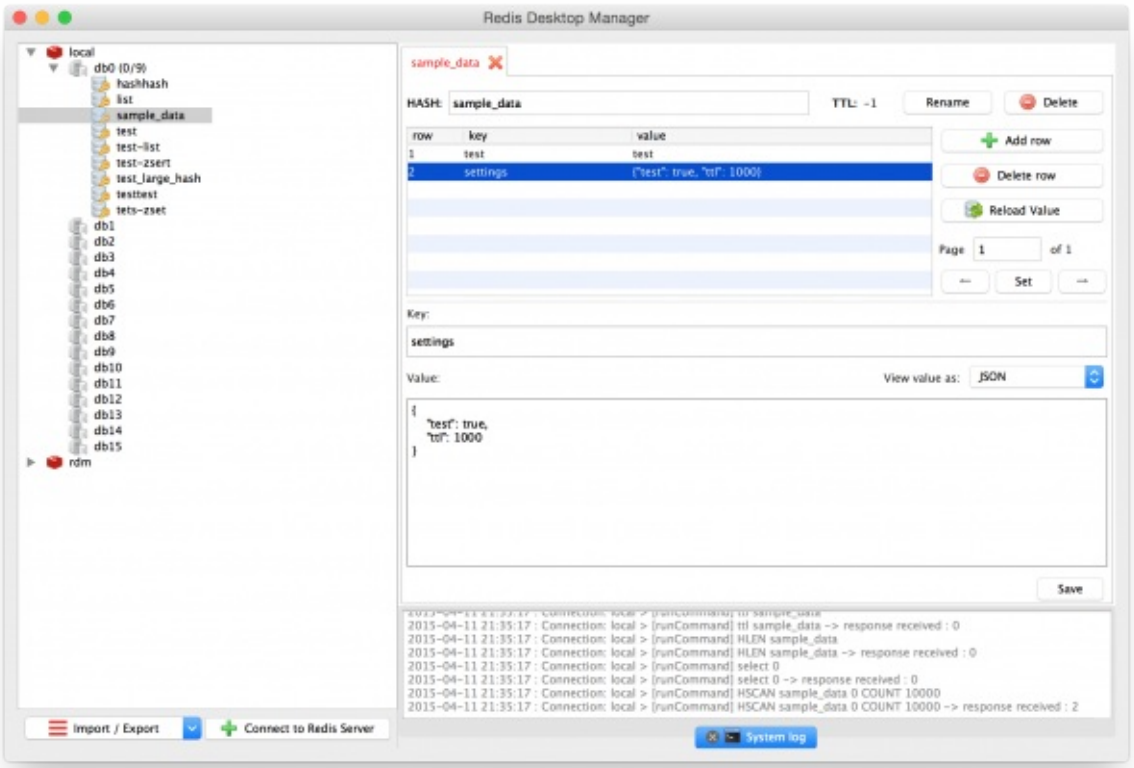
2. Redis 可视化客户端工具

redis 可视化客户端工具

Redis是一个超精简的基于内存的键值对数据库(key-value)，一般对并发有一定要求的应用都用其储存session，乃至整个数据库。不过它公自带一个最小化的命令行式的数据库管理工具，有时侯使用起来并不方便。不过Github上面已经有了很多图形化的管理工具，而且都针对REDIS做了一些优化，如自动折叠带schema的key等。

Redis Desktop Manager

一款基于Qt5的跨平台Redis桌面管理软 件



- 支持: Windows 7+, Mac OS X 10.10+, Ubuntu 14+
- 特点： C++ 编写，响应迅速，性能好。但不支持数据库备份与恢复。
- 项目地址： https://github.com/uglide/RedisDesktopManager

> 原文: *https://piaosanlang.gitbooks.io/redis/content/01_gui.html*

3. Redis 基本数据类型

Redis 基本数据类型

Redis支持五种数据类型: string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set: 有序集合)。

redis类型	含义
String	字符串
Hash	哈希
List	列表
Set	集合
Sorted set	有序集合

String 字符串

string是redis最基本的类型，一个key对应一个value。

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

```
1. redis 127.0.0.1:6379> SET name "itcast"
2. OK
3. redis 127.0.0.1:6379> GET name
4. "itcast"
5. 127.0.0.1:6379> type name
6. string
```

删除键值

```
1. redis 127.0.0.1:6379> del name
```

删除这个 key 及对应的 value

验证键是否存在

```
1. redis 127.0.0.1:6379> exists name
2. (integer) 0
```

在上面的例子中，SET 和 GET 是 Redis STRING命令， `name` 和 `itcast` 是存储在 Redis 的键和字符串值。

Hash 哈希

Redis hash 是一个键值对集合。

Redis hash是一个string类型的 `field` 和 `value` 的映射表，hash特别适合用于存储对象。

```
127.0.0.1:6379> HMSET my_hash_table username itcast age 18 sex male
OK
127.0.0.1:6379> HGETALL my_hash_table
1) "username"
2) "itcast"
3) "age"
4) "18"
5) "sex"
6) "male"
```

在上面的例子中，哈希数据类型用于存储包含用户基本信息的用户对象。

这里 HSET，HGETALL 是 Redis HASHES命令，同时 `my_hash_table` 也是一个键。

List 列表

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）。

```
redis 127.0.0.1:6379> lpush tutorial_list redis
(integer) 1
redis 127.0.0.1:6379> lpush tutorial_list mongodb
(integer) 2
redis 127.0.0.1:6379> lpush tutorial_list rabbitmq
(integer) 3
redis 127.0.0.1:6379> lrange tutorial_list 0 10

1) "rabitmq"
2) "mongodb"
3) "redis"
```

列表最多可存储 2^{32} - 1 元素 (4294967295, 每个列表可存储40多亿)。

Set 集合

Redis Set是 `string` 类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

在 Redis 可以添加，删除和测试成员存在的时间复杂度为 O（1）。

```
127.0.0.1:6379> sadd myset redis
```

```
(integer) 1
127.0.0.1:6379> sadd myset mongodb
(integer) 1
127.0.0.1:6379> sadd myset rabitmq
(integer) 1
127.0.0.1:6379> sadd myset rabitmq
(integer) 0
127.0.0.1:6379> smembers myset
1) "mongodb"
2) "redis"
3) "rabitmq"
```

注：在上面的例子中 rabitmq 被添加两次，但由于它是只集合具有唯一特性。

集合中最大的成员数为 2^{32} - 1(4294967295, 每个集合可存储40多亿个成员)。

zset(sorted set：有序集合)

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数(权重)。redis正是通过分数来为集合中的成员(权重)进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

```
127.0.0.1:6379> zadd mysortset 0 redis
(integer) 1
127.0.0.1:6379> zadd mysortset 2 mongodb
(integer) 1
127.0.0.1:6379> zadd mysortset 1 rabitmq
(integer) 1
127.0.0.1:6379> ZRANGEBYSCORE mysortset 0 1000
1) "redis"
2) "rabitmq"
3) "mongodb"
```

全部数据类型相关操作指令在 http://redis.cn/commands.html有详细介绍

> 原文: *https://piaosanlang.gitbooks.io/redis/content/01day/README2.html*

3.1. 字符串

字符串

String 是redis最基本的类型，value 不仅可以是 String,也可以是数字。

使用 Strings 类型,可以完全实现目前Memcached 的功能,并且效率更高。还可以享受 Redis 的定时持久化(可以选择 RDB 模式或者 AOF 模式).

string类型是二进制安全的。意思是redis的string可以包含任何数据,比如jpg图片或者序列化的对象

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

命令示例:

`set` 设置key对应的值为string类型的value。

```
1. > set name itcast
```

`setnx` 将key设置值为value，如果key不存在，这种情况下等同SET命令。 当key存在时，什么也不做。SETNX是”SET if Not eXists”的简写。

```
1. > get name
2. "itcast"
3. > setnx name itcast_new
4. (integer)0
5. >get name
6. "itcast"
```

`setex` 设置key对应字符串value，并且设置key在给定的 `seconds` 时间之后超时过期。

```
1. > setex color 10 red
2. > get color
3. "red"
4. 10秒后...
5. > get color (nil)
```

`setrange` 覆盖key对应的string的一部分，从指定的offset处开始，覆盖value的长度。

```
1. 127.0.0.1:6379> set email wangbaoqiang@itcast.cn
2. OK
3. 127.0.0.1:6379> setrange email 13 gmail.com
4. (integer) 22
5. 127.0.0.1:6379> get email
6. "wangbaoqiang@gmail.com"
7. 127.0.0.1:6379>STRLEN email
```

```
8. (integer) 22
```

其中的4是指从下标为13(包含13)的字符开始替换

`mset` 一次设置多个key的值,成功返回ok表示所有的值都设置了,失败返回0表示没有任何值被设置。

```
1. > mset key1 python key2 c++
2.   OK
```

`mget` 一次获取多个key的值,如果对应key不存在,则对应返回nil。

```
1. > mget key1 key2 key3
2.   1) "python"
3.   2) "c++"
4.   3) (nil)
```

`msetnx` 对应给定的keys到他们相应的values上。只要有一个key已经存在，MSETNX一个操作都不会执行。

```
1. > MSETNX key11 "Hello" key22 "there"
2. (integer) 1
3. > MSETNX key22 "there" key33 "world"
4. (integer) 0
```

认证了：MSETNX是原子的，所以所有给定的keys是一次性set的

`getset` 设置key的值,并返回key的旧值

```
1. > get name
2. "itcast"
3. > getset name itcast_new
4. "itcast"
5. > get name
6. "itcast_new"
```

`GETRANGE key start end` 获取指定key的value值的子字符串。是由start和end位移决定的

```
1. > getrange name 0 4
2.   "itcas"
```

`incr` 对key的值加1操作

```
1. > set age 20
2. > incr age
3. (integer) 21
```

`incrby` 同incr类似,加指定值 ,key不存在时候会设置key,并认为原来的value是 0

```
> incrby age 5
  (integer) 26
> incrby age1111 5
(integer) 5
> get age1111
"5"
```

`decr` 对key的值做的是减减操作,decr一个不存在key,则设置key为1

`decrby` 同decr,减指定值

`append` 给指定key的字符串值追加value,返回新字符串值的长度。例如我们向name的值追加一个"redis"字符串:

```
127.0.0.1:6379> get name
"itcast_new"
127.0.0.1:6379> append name "value"
(integer) 15
127.0.0.1:6379> get name
"itcast_newvalue"
127.0.0.1:6379>
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/01day/string.html*

3.2. Hash

HASH 哈希

Redis hash 是一个string类型的field和value的映射表，hash特别适合用于存储对象。

Redis 中每个 hash 可以存储 2^{32} - 1 键值对（40多亿）。

示例

`HSET key field value` 设置 key 指定的哈希集中指定字段的值

```
1. > hset myhash field1 Hello
```

`hget` 获取指定的hash field。

```
1. > hget myhash field1
2.  "Hello"
3. > hget myhash field3
4.  (nil)
```

由于数据库没有field3,所以取到的是一个空值nil.

`HSETNX key field value` 只在 key 指定的哈希集中不存在指定的字段时，设置字段的值。如果 key 指定的哈希集不存在，会创建一个新的哈希集并与 key 关联。如果字段已存在，该操作无效果。

```
1. > hsetnx myhash field "Hello"
2. (integer) 1
3. > hsetnx myhash field "Hello"
4. (integer) 0
```

第一次执行是成功的,但第二次执行相同的命令失败,原因是field已经存在了。

`hmset` 同时设置hash的多个field。

```
1. > hmset myhash field1 Hello field2 World
2. > OK
```

`hmget` 获取全部指定的hash filed。

```
1. > hmget myhash field1 field2 field3
2. 1) "Hello"
3. 2) "World"
4. 3) (nil)
```

`hincrby` 指定的hash filed 加上给定值。

```
> hset myhash field3 20
 (integer) 1
> hget myhash field3
 "20"
> hincrby myhash field3 -8
(integer) 12
> hget myhash field3
"12
```

`hexists` 测试指定field是否存在。

```
> hexists myhash field1
  (integer) 1
> hexists myhash field9
  (integer) 0
  通过上例可以说明field1存在,但field9是不存在的。
```

`hdel` 从 key 指定的哈希集中移除指定的域

```
127.0.0.1:6379> hkeys myhash
1) "field1"
2) "field"
3) "field2"
4) "field3"
127.0.0.1:6379> hdel myhash field
(integer) 1
127.0.0.1:6379> hkeys myhash
1) "field1"
2) "field2"
3) "field3"
127.0.0.1:6379>
```

`hlen` 返回指定hash的field数量。

```
> hlen myhash
  (integer) 3
```

`hkeys` 返回hash的所有field。

```
> hkeys myhash
> 1) "field2"
> 2) "field"
> 3) "field3"
```

说明这个hash中有3个field。

`hvals` 返回hash的所有value。

```
> hvals myhash
1) "World"
2)"Hello"
3)"12"
```

说明这个hash中有3个field。

`hgetall` 获取某个hash中全部的filed及value。

```
> hgetall myhash
1) "field2"
2) "World"
3) "field"
4) "Hello"
5) "field3"
6) "12"
```

`HSTRLEN` — 返回 hash指定field的value的字符串长度

```
127.0.0.1:6379> HSTRLEN myhash field1
(integer) 5
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/01day/hash.html*

3.3. List

List 列表

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导列表的头部（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)。

RPUSH key value [value …]

向存于 key 的列表的尾部插入所有指定的值。如果 key 不存在，那么会创建一个空的列表然后再进行 push 操作。

```
redis> RPUSH mylist "hello"
(integer) 1
redis> RPUSH mylist "world"
(integer) 2
redis> LRANGE mylist 0 -1
1) "hello"
2) "world"
redis>
```

LPOP key

移除并且返回 key 对应的 list 的第一个元素。

```
redis> RPUSH mylist "one"
(integer) 1
redis> RPUSH mylist "two"
(integer) 2
redis> RPUSH mylist "three"
(integer) 3
redis> LPOP mylist
"one"
redis> LRANGE mylist 0 -1
1) "two"
2) "three"
redis>
```

LTRIM key start stop

修剪(trim)一个已存在的 list，这样 list 就会只包含指定范围的指定元素。

> *start* 和 *stop* 都是由*0*开始计数的， 这里的 *0* 是列表里的第一个元素（表头），*1* 是第二个元素，以此类推。

例如： LTRIM foobar 0 2 将会对存储在 foobar 的列表进行修剪，只保留列表里的前3个元素。

start 和 end 也可以用负数来表示与表尾的偏移量，比如 -1 表示列表里的最后一个元素， -2 表示倒数第二个，等等。

应用场景：

1.取最新N个数据的操作

比如典型的取你网站的最新文章，通过下面方式，我们可以将最新的5000条评论的ID放在Redis的List集合中，并将超出集合部分从数据库获取

- 使用LPUSH latest.comments命令，向list集合中插入数据
- 插入完成后再用LTRIM latest.comments 0 5000命令使其永远只保存最近5000个ID
- 然后我们在客户端获取某一页评论时可以用下面的逻辑（伪代码）

  FUNCTION get_latest_comments(start,num_items):
  id_list = redis.lrange("latest.comments",start,start+num_items-1)
  IF id_list.length < num_items

  ```
  1. id_list = SQL_DB(&#34;SELECT ... ORDER BY time LIMIT ...&#34;)
  ```

  END
  RETURN id_list

如果你还有不同的筛选维度，比如某个分类的最新N条，那么你可以再建一个按此分类的List，只存ID的话，Redis是非常高效的。

示例

取最新N个评论的操作

```
1. 127.0.0.1:6379> lpush mycomment 100001
2. (integer) 1
3. 127.0.0.1:6379> lpush mycomment 100002
4. (integer) 2
5. 127.0.0.1:6379> lpush mycomment 100003
6. (integer) 3
7. 127.0.0.1:6379> lpush mycomment 100004
8. (integer) 4
9. 127.0.0.1:6379> LRANGE mycomment 0 -1
10. 1) "100004"
11. 2) "100003"
12. 3) "100002"
13. 4) "100001"
14. 127.0.0.1:6379> LTRIM mycomment 0 1
15. OK
16. 127.0.0.1:6379> LRANGE mycomment 0 -1
17. 1) "100004"
18. 2) "100003"
19. 127.0.0.1:6379> lpush mycomment 100005
20. (integer) 3
21. 127.0.0.1:6379> LRANGE mycomment 0 -1
```

```
1) "100005"
2) "100004"
3) "100003"
127.0.0.1:6379> LTRIM mycomment 0 1
OK
127.0.0.1:6379> LRANGE mycomment 0 -1
1) "100005"
2) "100004"
127.0.0.1:6379>
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/01day/list.html*

3.4. Set

Set 集合

Set 就是一个集合,集合的概念就是一堆不重复值的组合。利用 Redis 提供的 Set 数据结构,可以存储一些集合性的数据。

> 比如在 微博应用中,可以将一个用户所有的关注人存在一个集合中,将其所有粉丝存在一个集合。

因为 Redis 非常人性化的为集合提供了 求交集、并集、差集等操作, 那么就可以非常方便的实现如共同关注、共同喜好、二度好友等功能, 对上面的所有集合操作,你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

SADD key member [member …]

添加一个或多个指定的member元素到集合的 key中

```
redis> SADD myset "Hello"
(integer) 1
redis> SADD myset "World"
(integer) 1
redis> SADD myset "World"
(integer) 0
redis> SMEMBERS myset
1) "World"
2) "Hello"
redis>
```

SCARD key

返回集合存储的key的基数 (集合元素的数量).

```
redis> SADD myset "Hello"
(integer) 1
redis> SADD myset "World"
(integer) 1
redis> SCARD myset
(integer) 2
redis>
```

SDIFF key [key …]

返回一个集合与给定集合的差集的元素.

```
redis> SADD key1 'a' 'b' 'c'
(integer) 1
```

```
redis> SADD key2 "c"
(integer) 1
redis> SADD key2 "d"
(integer) 1
redis> SADD key2 "e"
(integer) 1
redis> SDIFF key1 key2
1) "a"
2) "b"
redis>
```

应用场景

1.共同好友、二度好友

2.利用唯一性,可以统计访问网站的所有独立 IP

3.好友推荐的时候,根据 tag 求交集,大于某个 临界值 就可以推荐

示例

以王宝强和马蓉为例，求二度好友，共同好友，推荐系统

```
127.0.0.1:6379> sadd marong_friend 'songdan' 'wangsicong' 'songzhe'
(integer) 1
127.0.0.1:6379> SMEMBERS marong_friend
1) "songzhe"
2) "wangsicong"
3) "songdandan"
127.0.0.1:6379> sadd wangbaoqiang_friend 'dengchao' 'angelababy' 'songzhe'
(integer) 1

#求共同好友
127.0.0.1:6379> SINTER marong_friend wangbaoqiang_friend
1) "songzhe"

#推荐好友系统
127.0.0.1:6379> SDIFF marong_friend wangbaoqiang_friend
1) "wangsicong"
2) "songdandan"
127.0.0.1:6379>
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/01day/set.html*

3.5. Sort-set

Sorted Set 有序集合

Redis 有序集合和集合一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

有序集合的成员是唯一的,但分数(score)却可以重复。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。 集合中最大的成员数为 $2^{32} - 1$ (4294967295, 每个集合可存储40多亿个成员)。

ZADD key score member

将所有指定成员添加到键为key有序集合（sorted set）里面

```
redis> ZADD myzset 1 "one"
(integer) 1
redis> ZADD myzset 1 "uno"
(integer) 1
redis> ZADD myzset 2 "two" 3 "three"
(integer) 2
redis> ZRANGE myzset 0 -1 WITHSCORES
1) "one"
2) "1"
3) "uno"
4) "1"
5) "two"
6) "2"
7) "three"
8) "3"
redis>
```

ZCOUNT key min max

返回有序集key中，score值在min和max之间(默认包括score值等于min或max)的成员

```
redis> ZADD myzset 1 "one"
(integer) 1
redis> ZADD myzset 2 "two"
(integer) 1
redis> ZADD myzset 3 "three"
(integer) 1
redis> ZCOUNT myzset -inf +inf
(integer) 3
redis> ZCOUNT myzset (1 3
```

```
(integer) 2
redis>
```

ZINCRBY key increment member

为有序集key的成员member的score值加上增量increment

```
redis> ZADD myzset 1 "one"
(integer) 1
redis> ZADD myzset 2 "two"
(integer) 1
redis> ZINCRBY myzset 2 "one"
"3"
redis> ZRANGE myzset 0 -1 WITHSCORES
1) "two"
2) "2"
3) "one"
4) "3"
redis>
```

应用场景

1.带有权重的元素,LOL游戏大区最强王者

2 排行榜

案例

斗地主大赛排名

- 初始比赛
  127.0.0.1:6379> ZADD doudizhu_rank 0 "player1"
  (integer) 1
  127.0.0.1:6379> ZADD doudizhu_rank 0 "player2"
  (integer) 1
  127.0.0.1:6379> ZADD doudizhu_rank 0 "player3"
  (integer) 1
- 比赛开始，经过n轮比赛，每次统计，类似计算如下所示
  127.0.0.1:6379> ZINCRBY doudizhu_rank 3 player3
  "3"
  127.0.0.1:6379> ZINCRBY doudizhu_rank -1 player2
  "-1"
  127.0.0.1:6379> ZINCRBY doudizhu_rank -2 player1
  "-2"
- 比赛结束，进行排名
  127.0.0.1:6379> ZRANGE doudizhu_rank 0 -1

```
    1) "player1"
    2) "player2"
    3) "player3"
    127.0.0.1:6379> ZRANGE doudizhu_rank 0 -1 withscores
    1) "player1"
    2) "-2"
    3) "player2"
    4) "-1"
    5) "player3"
    6) "3"
```

逆序排序才对

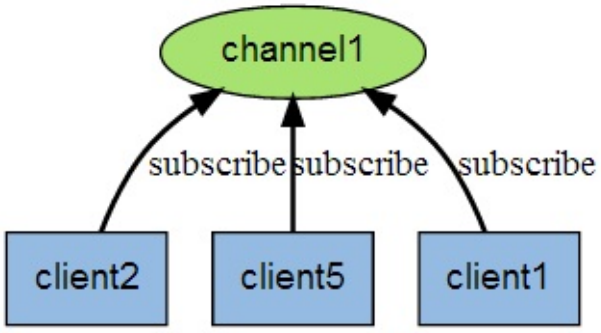
```
127.0.0.1:6379> zrevrange doudizhu_rank 0 -1 withscores
1) "player3"
2) "3"
3) "player2"
4) "-1"
5) "player1"
6) "-2"
127.0.0.1:6379>
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/01day/sort-set.html*

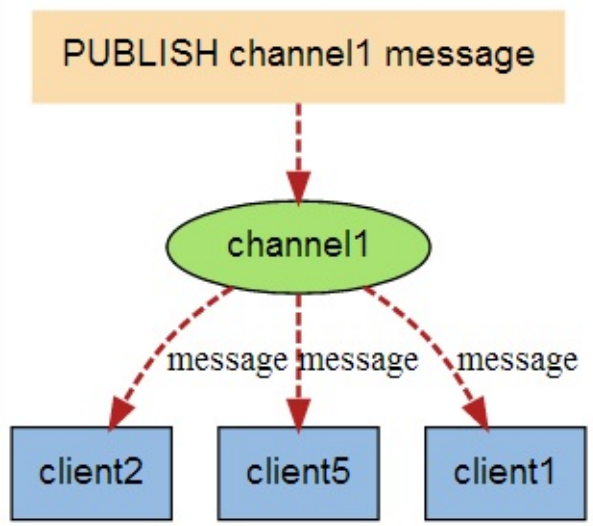
4. Redis 订阅和发布模式

Redis 订阅发布模式

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。Redis 客户端可以订阅任意数量的频道。下图展示了频道 channel1 ， 以及订阅这个频道的三个客户端 —— client2、client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时， 这个消息就会被发送给订阅它的三个客户端：



SUBSCRIBE channel [channel …]

订阅给指定频道的信息。

PUBLISH channel message

将信息 message 发送到指定的频道 channel

应用场景

在Redis中，你可以设定对某一个key值进行消息发布及消息订阅，当一个key值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

- 今日头条订阅号、微信订阅公众号、新浪微博关注、邮件订阅系统
- 即时通信系统（QQ、微信）
- 群聊部落系统（微信群）

案例

微信班级群 `class:20170101` ，发布订阅模型

学生 **A B C:**

学生**C:**

订阅一个 `主题` 名叫： `class:20170101`

```
127.0.0.1:6379> SUBSCRIBE class:20170101
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "redisChat"
3) (integer) 1
```

学生**A:**

针对 `class:20170101` 主题发送 消息，那么所有订阅该主题的用户都能够收到该数据。

```
127.0.0.1:6379> PUBLISH class:20170101 "i love peace!"
(integer) 1
```

学生**B:**

针对 `class:20170101` 主题发送 消息，那么所有订阅该主题的用户都能够收到该数据。

```
127.0.0.1:6379> PUBLISH class:20170101 "go to hell"
(integer) 1
```

最后学生C会收到 A 和 B 发送过来的消息。

```
python@ubuntu:~$ redis-cli
127.0.0.1:6379> SUBSCRIBE class:20170101
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "class:20170101"
3) (integer) 1
1) "message"
2) "class:20170101"
```

```
3) "i love peace!"
1) "message"
2) "class:20170101"
3) "i love peace!"
1) "message"
2) "class:20170101"
3) "go to hell"
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/04_pub_sub.html*

5. Redis 事务

Redis 事务

Redis事务允许一组命令在单一步骤中执行。事务有两个属性，说明如下：

- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- Redis事务是原子的。原子意味着要么所有的命令都执行，要么都不执行；

一个事务从开始到执行会经历以下三个阶段：

- 开始事务
- 命令入队
- 执行事务

```
redis 127.0.0.1:6379> MULTI
OK
List of commands here
redis 127.0.0.1:6379> EXEC
```

案例

银行转账，邓超给宝强转账1万元：

```
127.0.0.1:6379> set dengchao 60000
OK
127.0.0.1:6379> set wangbaoqiang 200
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> INCRBY dengchao -10000
QUEUED
127.0.0.1:6379> INCRBY wangbaoqiang 10000
QUEUED
127.0.0.1:6379> EXEC
1) (integer) 50000
2) (integer) 10200
127.0.0.1:6379>
```

> 原文：*https://piaosanlang.gitbooks.io/redis/content/05_transaction.html*

6. Redis 使用场景

Redis 使用场景

1.取最新N个数据的操作

比如典型的取你网站的最新文章,我们可以将最新的5000条评论的ID放在Redis的List集合中,并将超出集合部分从数据库获取。

2.排行榜应用,取TOP N操作

这个需求与上面需求的不同之处在于,前面操作以时间为权重,这个是以某个条件为权重,比如按顶的次数排序, 这时候就需要我们的sorted set出马了,将你要排序的值设置成sorted set的score, 将具体的数据设置成相应的value,每次只需要执行一条ZADD命令即可。

3.需要精准设定过期时间的应用

比如你可以把上面说到的sorted set的score值设置成过期时间的时间戳,那么就可以简单地通过过期时间排序, 定时清除过期数据了,不仅是清除Redis中的过期数据,你完全可以把Redis里这个过期时间当成是对数据库中数据的索引, 用Redis来找出哪些数据需要过期删除,然后再精准地从数据库中删除相应的记录。

4.计数器应用

Redis的命令都是原子性的,你可以轻松地利用INCR,DECR命令来构建计数器系统。

5.uniq操作,获取某段时间所有数据排重值

这个使用Redis的set数据结构最合适了,只需要不断地将数据往set中扔就行了,set意为集合,所以会自动排重。

6.Pub/Sub构建实时消息系统

Redis的Pub/Sub系统可以构建实时的消息系统,比如很多用Pub/Sub构建的实时聊天系统的例子。

7.构建队列系统

使用list可以构建队列系统,使用sorted set甚至可以构建有优先级的队列系统。

8.缓存

最常用,性能优于Memcached(被libevent拖慢),数据结构更多样化。

> 原文: *https://piaosanlang.gitbooks.io/redis/content/06_redisshi_yong_chang_jing.html*

7. Redis 数据备份与恢复

Redis 数据备份与恢复

Redis SAVE 命令用于创建当前数据库的备份。

语法

redis Save 命令基本语法如下：

```
redis 127.0.0.1:6379> SAVE
```

实例

```
redis 127.0.0.1:6379> SAVE
OK
```

该命令将在 redis 安装目录中创建dump.rdb文件。

恢复数据

如果需要恢复数据，只需将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可。获取 redis 目录可以使用 CONFIG 命令，如下所示：

```
redis 127.0.0.1:6379> CONFIG GET dir
1) "dir"
2) "/home/python"
```

以上命令 CONFIG GET dir 输出的 redis 安装目录为 /home/python。

Bgsave

创建 redis 备份文件也可以使用命令 BGSAVE，该命令在后台执行。

实例

```
127.0.0.1:6379> BGSAVE

Background saving started
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/07_bak_restore.html*

8. Redis 配置文件

Redis 配置文件

在启动Redis服务器时,我们需要为其指定一个配置文件,缺省情况下配置文件在Redis的源码目录下,文件名为**redis.conf**。

redis配置文件使用 `####################` 被分成了几大块区域,

主要有:

- 通用(general)
- 快照(snapshotting)
- 复制(replication)
- 安全(security)
- 限制(limits)
- 追加模式(append only mode)
- LUA脚本(lua scripting)
- REDIS集群(REDIS CLUSTER)
- 慢日志(slow log)
- 事件通知(event notification)
- ADVANCED CONFIG
  为了对Redis的系统实现有一个直接的认识,我们首先来看一下Redis的配置文件中定义了哪些主要参数以及这些参数的作用。

- daemonize no
  默认情况下,redis不是在后台运行的。如果需要在后台运行,把该项的值更改为yes;

- pidfile /var/run/redis_6379.pid
  当Redis在后台运行的时候,Redis默认会把pid文件放在/var/run/redis.pid, 你可以配置到其他地址。
  当运行多个redis服务时,需要指定不同的pid文件和端口;

- port 6379
  指定redis运行的端口,默认是6379;

作者在自己的一篇博文中解释了为什么选用6379作为默认端口，因为6379在手机按键上MERZ对应的号码，而MERZ取自意大利歌女Alessia Merz的名字

- bind 127.0.0.1
  指定redis只接收来自于该IP地址的请求,如果不进行设置,那么将处理所有请求。在生产环境中最好设置该项;

- loglevel notice
  指定日志记录级别,

其中Redis总共支持四个级别: debug 、verbose 、notice 、warning, 默认为 notice 。

- debug表示记录很多信息,用于开发和测试。
- verbose表示记录有用的信息, 但不像debug会记录那么多。
- notice表示普通的verbose,常用于生产环境。
- warning 表示只有非常重要或者严重的信息会记录到日志;

- logfile ""
  配置log文件地址,默认值为stdout。若后台模式会输出到/dev/null("黑洞");(可以自定义配置: /var/log/redis/redis.log)

- databases 16
  可用数据库数,默认值为16,默认数据库为0,数据库范围在0 15之间切换,彼此隔离;

- save
  保存数据到磁盘,格式为 `save <seconds> <changes>`,指出在多长时间内,有多少次更新操作, 就将数据同步到数据文件rdb。相当于条件触发抓取快照,这个可以多个条件配合。

保存数据到磁盘:

- save 900 1 #900秒(15分钟)内至少有1个key被改变
- save 300 10 #300秒(5分钟)内至少有10个key被改变
- save 60 10000 #60秒内至少有10000个key被改变

- rdbcompression yes
  存储至本地数据库时(持久化到rdb文件)是否压缩数据,默认为yes;

如果为了节省CPU时间, 可以关闭该选项, 但会导致数据库文件变的巨大

- dbfilename dump.rdb
  本地持久化数据库文件名,默认值为dump.rdb;

- dir ./
  工作目录,数据库镜像备份的文件放置的路径。

- slaveof
  主从复制,设置该数据库为其他数据库的从数据库。设置当本机为slave服务时,设置master服务的IP地址及端口。 在Redis启动时,它会自动从master进行数据同步;

- masterauth
  当master服务设置了密码保护时(用requirepass制定的密码)slave服务连接master的密码;

- slave-serve-stale-data yes

当一个slave失去和master的连接，或者同步正在进行中，slave的行为有两种可能:

1) 如果 slave-serve-stale-data 设置为 "yes" (默认值)，slave会继续响应客户端请求，可能是正常数据，也可能是还没获得值的空数据。

2) 如果 slave-serve-stale-data 设置为 "no"，slave会回复"正在从master同步（SYNC with master in progress）"来处理各种请求，除了 INFO 和 SLAVEOF 命令。

- repl-ping-slave-period 10
  从库会按照一个时间间隔向主库发送PING,可以通过repl-ping-slave-period设置这个时间间隔,默认是10秒;

- repl-timeout 60
  设置主库批量数据传输时间或者ping回复时间间隔,默认值是60秒,一定要确保repl-timeout大于repl-ping-slave-period ;不然会经常检测到超时。

master检测到slave上次发送的时间超过repl-timeout，即认为slave离线，清除该slave信息。

slave检测到上次和master交互的时间超过repl-timeout，则认为master离线

- requirepass foobared
  设置客户端连接后进行任何其他指定前需要使用的密码。因为redis速度相当快,所以在一台比较好的服务器下， 一个外部的用户可以在一秒钟进行150K次的密码尝试,这意味着你需要指定非常强大的密码来防止暴力破解;

- renamecommand CONFIG ""
  命令重命名,在一个共享环境下可以重命名相对危险的命令,比如把CONFIG重名为一个不容易猜测的字符:rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52。 如果想删除一个命令,直接把它重命名为一个空字符""即可:rename-command CONFIG "";

- maxclients 128
  设置最多同时连接客户端数量。

默认没有限制，这个关系到Redis进程能够打开的文件描述符数量。

特殊值"0"表示没有限制。

一旦达到这个限制，Redis会关闭所有新连接并发送错误"达到最大用户数上限（max number of clients reached）"

- maxmemory
  指定Redis最大内存限制。Redis在启动时会把数据加载到内存中,达到最大内存后,Redis会先尝试清除已到期或即将到期的Key， Redis同时也会移除空的list对象。当此方法处理后,仍然到达最大内存设置,将无法再进行写入操作,但仍然可以进行读取操作。

注意:Redis新的vm机制,会把Key存放内存,Value会存放在swap区;

- maxmemory-policy volatile-lru
  当内存达到最大值的时候Redis会选择删除哪些数据呢?有五种方式可供选择:

- volatile-lru 代表利用LRU算法移除设置过期时间的key(LRU:最近使用 LeastRecentlyUsed)

- allkeys-lru 代表利用LRU算法移除任何key
- volatile-random 代表移除设置过过期时间的随机key
- allkeys_random 代表移除一个随机的key,
- volatile-ttl 代表移除即将过期的key(minor TTL)
- noeviction 代表不移除任何key,只是返回一个写错误。
  注意:对于上面的策略,如果没有合适的key可以移除,写的时候Redis会返回一个错误;

- appendonly no
  是否开启aof功能

默认情况下,redis会在后台异步的把数据库镜像备份到磁盘,但是该备份是非常耗时的,而且备份也不能很频繁。 如果发生诸如拉闸限电、拔插头等状况,那么将造成比较大范围的数据丢失,所以redis提供了另外一种更加高效的数据库备份及灾难恢复方式。

开启appendonly模式之后,redis会把所接收到的每一次写操作请求都追加到appendonly.aof文件中。

当redis重新启动时,会从该文件恢复出之前的状态

- appendfilename appendonly.aof
  AOF文件名称,默认为"appendonly.aof";

- appendfsync everysec
  Redis支持三种同步AOF文件的策略:

- no 代表不进行同步,系统去操作,

- always 代表每次有写操作都进行同步,
- everysec 代表对写操作进行累积,每秒同步一次
  默认是"everysec",按照速度和安全折中这是最好的。

- slowlog-log-slower-than 10000
  记录超过特定执行时间的命令。执行时间不包括I/O计算,比如连接客户端,返回结果等,只是命令执行时间。

可以通过两个参数设置slow log:一个是告诉Redis执行超过多少时间被记录的参数slowlog-log-slower-than(微妙), 另一个是slow log 的长度。当一个新命令被记录的时候最早的命令将被从队列中移除,

下面的时间以微秒(百万分之一秒,1000 * 1000)单位, 因此1000000代表一分钟。

```
1. 1秒=1000毫秒
2. 1秒=1000000微秒
```

注意制定一个负数将关闭慢日志,而设置为0将强制每个命令都会记录;

- slowlog-max-len 128
  慢操作日志"保留的最大条数

"记录"将会被队列化,如果超过了此长度,旧记录将会被移除。可以通过 `SLOWLOG <subcommand> args` 查看慢记录的信息(SLOWLOG get 10,SLOWLOG reset),通过"SLOWLOG get num"指令可以查看最近num条慢速记录,其中包括"记录"操作的时间/指令/K-V等信息。

参考: https://github.com/linli8/cnblogs/blob/master/redis%E5%89%AF%E6%9C%AC.conf

总结

Select 命令

Redis Select 命令用于切换到指定的数据库,数据库索引号 index 用数字值指定,以 0 作为起始索引值。

```
python@ubuntu:/dev$ redis-cli
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> set name 'itcast.cn'
OK
127.0.0.1:6379[1]> select 2
OK
127.0.0.1:6379[2]> set web 'itcast.com'
OK
127.0.0.1:6379[2]> get web
"itcast.com"
127.0.0.1:6379[2]> select 1
OK
127.0.0.1:6379[1]> get name
"itcast.cn"
127.0.0.1:6379[1]>
```

Shutdown 命令

Redis Shutdown 命令执行以下操作:

- 停止所有客户端
- 如果有至少一个保存点在等待,执行 SAVE 命令
- 如果 AOF 选项被打开,更新 AOF 文件
- 关闭 redis 服务器(server)

```
redis 127.0.0.1:6379> PING
PONG
redis 127.0.0.1:6379> SHUTDOWN
$ redis
```

Redis 认证

```
redis-cli
AUTH "password"
```

或者

```
redis-cli -h 127.0.0.1 -p 6379 -a myPassword
```

Redis Showlog

Redis Showlog 是 Redis 用来记录查询执行时间的日志系统。

查询执行时间指的是不包括像客户端响应(talking)、发送回复等 IO 操作，而单单是执行一个查询命令所耗费的时间。

另外，slow log 保存在内存里面，读写速度非常快，因此你可以放心地使用它，不必担心因为开启 slow log 而损害 Redis 的速度。

语法redis Showlog 命令基本语法如下:

```
redis 127.0.0.1:6379> SLOWLOG subcommand [argument]
```

查看日志信息:

```
redis 127.0.0.1:6379> slowlog get 2
1) 1) (integer) 14
   2) (integer) 1309448221
   3) (integer) 15
   4) 1) "ping"
2) 1) (integer) 13
   2) (integer) 1309448128
   3) (integer) 30
   4) 1) "slowlog"
      2) "get"
      3) "100"
```

> 每一个*Showlog*都是由四个字段组成的慢日志标识符记录的命令进行处理的*Unix*时间戳执行所需的时间，以微秒命令、参数。

查看当前日志的数量:

```
redis 127.0.0.1:6379> SLOWLOG LEN
(integer) 14
使用命令 SLOWLOG RESET 可以清空 slow log 。

redis 127.0.0.1:6379> SLOWLOG LEN
(integer) 14

```

```
redis 127.0.0.1:6379> SLOWLOG RESET
OK

redis 127.0.0.1:6379> SLOWLOG LEN
(integer) 0
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/08_redispei_zhi_wen_jian.html*

9. Redis 持久化

对Redis持久化的探讨与理解

目前Redis持久化的方式有两种： RDB 和 AOF

首先，我们应该明确持久化的数据有什么用，答案是：

> 用于重启后的数据恢复

Redis是一个内存数据库，无论是RDB还是AOF，都是其保证数据恢复的措施。

所以Redis在利用RDB和AOF进行恢复的时候，都会读取RDB或AOF文件，重新加载到内存中。

RDB

RDB就是Snapshot快照存储，是默认的持久化方式。可理解为半持久化模式，

> 即按照一定的策略周期性的将数据保存到磁盘。对应产生的数据文件为*dump.rdb*，快照的周期通过配置文件中的*save*参数来定义。

下面是默认的快照设置：

```
1. dbfilename dump.rdb
2. # save <seconds> <changes>
3. save 900 1    #当有一条Keys数据被改变时，900秒刷新到Disk一次
4. save 300 10   #当有10条Keys数据被改变时，300秒刷新到Disk一次
5. save 60 10000 #当有10000条Keys数据被改变时，60秒刷新到Disk一次
```

Redis的RDB文件不会坏掉，因为其写操作是在一个新进程中进行的。当生成一个新的RDB文件时，Redis生成的子进程会先将数据写到一个临时文件中，然后通过原子性rename系统调用将临时文件重命名为RDB文件。

这样在任何时候出现故障，Redis的RDB文件都总是可用的。

同时，Redis的RDB文件也是Redis主从同步内部实现中的一环。

> 第一次*Slave*向*Master*同步的实现是：*Slave*向*Master*发出同步请求，*Master*先*dump*出*rdb*文件，然后将*rdb*文件全量传输给*slave*，然后*Master*把缓存的命令转发给*Slave*，初次同步完成。第二次以及以后的同步实现是：*Master*将变量的快照直接实时依次发送给各个*Slave*。但不管什么原因导致*Slave*和*Master*断开重连都会重复以上两个步骤的过程。

Redis的主从复制是建立在内存快照的持久化基础上的，只要有Slave就一定会有内存快照发生。可以很明显的看到，RDB有它的不足，就是一旦数据库出现问题，那么我们的RDB文件中保存的数据并不是全新的。

从上次RDB文件生成到Redis停机这段时间的数据全部丢掉了。

AOF（Append-only file）方式

AOF(Append-Only File)比RDB方式有更好的持久化性。

- 在使用AOF持久化方式时，Redis会将每一个收到的写命令都通过Write函数追加到文件中，类似于MySQL的binlog。
- 当Redis重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。

在Redis重启时会逐个执行AOF文件中的命令来将硬盘中的数据载入到内存中,所以说，载入的速度相较RDB会慢一些

- 默认情况下,Redis没有开启AOF方式的持久化,可以在redis.conf中通过appendonly参数开启:

  appendonly yes #启用aof持久化方式

  # appendfsync always #每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用

  appendfsync everysec #每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐

  # appendfsync no #完全依赖OS的写入，一般为30秒左右一次，性能最好但是持久化最没有保证，不被推荐。

- AOF文件和RDB文件的保存文件夹位置相同,都是通过dir参数设置的,默认的文件名是appendonly.aof,可以通过appendfilename参数修改

  appendfilename appendonly.aof

- AOF的完全持久化方式同时也带来了另一个问题，持久化文件会变得越来越大。

比如: 我们调用INCR test 命令100次，文件中就必须保存全部的100条命令，但其实99条都是多余的。
因为要恢复数据库的状态其实文件中保存一条SET test 100就够了。

为了压缩AOF的持久化文件，Redis提供了bgrewriteaof命令。收到此命令后Redis将使用与快照类似的方式将内存中的数据以命令的方式保存到临时文件中，最后替换原来的文件，以此来实现控制AOF文件的增长。

配置redis自动重写AOF文件的参数

no-appendfsync-on-rewrite yes #在AOF重写时，不进行命令追加操作，而只是将其放在缓冲区里，避免与命令的追加造成 `DISK IO` 上的冲突。
auto-aof-rewrite-percentage 100 #当前AOF文件大小是上次日志重写得到AOF文件大小的二倍时，自动启动新的日志重写过程。
auto-aof-rewrite-min-size 64mb #当前AOF文件启动新的日志重写过程的最小值，避免刚刚启动Reids时由于文件尺寸较小导致频繁的重写。

到底选择什么呢？

下面是来自官方的建议：

通常，如果你要想提供很高的数据保障性，那么建议你同时使用两种持久化方式。

> 如果你可以接受灾难带来的几分钟的数据丢失，那么你可以仅使用*RDB*。很多用户仅使用了*AOF*，但是我们建议，既然*RDB*可以时不时的给数据做个完整的快照，并且提供更快的重启，所以建议也使用*RDB*。

因此，我们希望可以在未来（长远计划）统一AOF和RDB成一种持久化模式。

在数据恢复方面：RDB的启动时间会更短，原因有两个：

> 一是*RDB*文件中每一条数据只有一条记录，不会像*AOF*日志那样可能有一条数据的多次操作记录。所以每条数据只需要写一次就行了。另一个原因是*RDB*文件的存储格式和*Redis*数据在内存中的编码格式是一致的，不需要再进行数据编码工作，所以在*CPU*消耗上要远小于*AOF*日志的加载。

二、灾难恢复模拟

既然持久化的数据的作用是用于重启后的数据恢复，那么我们有必要进行一次这样的灾难恢复模拟了。

> 如果数据要做持久化又想保证稳定性，则建议留空一半的物理内存。因为在进行快照的时候，*fork*出来进行*dump*操作的子进程会占用与父进程一样的内存，真正的*copy-on-write*，对性能的影响和内存的耗用都是比较大的。

目前，通常的设计思路是利用 `Replication` 机制来弥补aof、snapshot性能上的不足，达到了数据可持久化。

> 即*Master*上*Snapshot*和*AOF*都不做，来保证*Master*的读写性能，而*Slave*上则同时开启*Snapshot*和*AOF*来进行持久化，保证数据的安全性。

首先，修改Master上的如下配置：

```
$ sudo vim /redis/etc/redis.conf
#save 900 1 #禁用Snapshot
#save 300 10
#save 60 10000

appendonly no #禁用(注释)AOF
```

接着，修改Slave上的如下配置：

```
$ sudo vim /redis/etc/redis.conf
save 900 1 #启用Snapshot
save 300 10
save 60 10000

appendonly yes #启用AOF
appendfilename appendonly.aof #AOF文件的名称
# appendfsync always
appendfsync everysec #每秒钟强制写入磁盘一次
# appendfsync no

no-appendfsync-on-rewrite yes   #在日志重写时，不进行命令追加操作
auto-aof-rewrite-percentage 100 #自动启动新的日志重写过程
auto-aof-rewrite-min-size 64mb  #启动新的日志重写过程的最小值
```

分别启动Master与Slave

```
$ redis-server /etc/redis/redis.conf
```

启动完成后在Master中确认未启动Snapshot参数

```
redis 127.0.0.1:6379> CONFIG GET save
1) "save"
2) ""
```

然后通过以下脚本在Master中生成25万条数据：

```
python@redis:$ cat redis-cli-generate.temp.sh
```

```
#!/bin/bash

REDISCLI="redis-cli -a slavepass -n 1 SET"
ID=1

while(($ID<50001))
do
  INSTANCE_NAME="i-2-$ID-VM"
  UUID=`cat /proc/sys/kernel/random/uuid`
  PRIVATE_IP_ADDRESS=10.`echo "$RANDOM % 255 + 1" | bc`.`echo "$RANDOM % 255 + 1" | bc`.`echo "$RANDOM % 255 + 1" | bc`\
  CREATED=`date "+%Y-%m-%d %H:%M:%S"`

  $REDISCLI vm_instance:$ID:instance_name "$INSTANCE_NAME"
  $REDISCLI vm_instance:$ID:uuid "$UUID"
  $REDISCLI vm_instance:$ID:private_ip_address "$PRIVATE_IP_ADDRESS"
  $REDISCLI vm_instance:$ID:created "$CREATED"

  $REDISCLI vm_instance:$INSTANCE_NAME:id "$ID"

  ID=$(($ID+1))
done
```

```
python@redis:$ ./redis-cli-generate.temp.sh
```

在数据的生成过程中，可以很清楚的看到Master上仅在第一次做Slave同步时创建了dump.rdb文件，之后就通过增量传输命令的方式给Slave了。dump.rdb文件没有再增大。

```
python@redis:/opt/redis/data/6379$ ls -lh
total 4.0K
-rw-r--r-- 1 root root 10 Sep 27 00:40 dump.rdb
```

而Slave上则可以看到dump.rdb文件和AOF文件在不断的增大，并且AOF文件的增长速度明显大于dump.rdb文件。

```
python@redis-slave:$ ls -lh
total 24M
```

```
-rw-r--r-- 1 root root 15M Sep 27 12:06 appendonly.aof
-rw-r--r-- 1 root root 9.2M Sep 27 12:06 dump.rdb
```

等待数据插入完成以后，首先确认当前的数据量。

```
redis 127.0.0.1:6379> info
...
...

used_memory:33055824
used_memory_human:31.52M
used_memory_rss:34717696
used_memory_peak:33055800
used_memory_peak_human:31.52M
...

# Keyspace
db0:keys=1,expires=0,avg_ttl=0
db1:keys=250000,expires=0
```

当前的数据量为 `db0:keys=1 db1:keys=250000` 条，占用内存31.52M。

然后我们直接Kill掉Master的Redis进程，模拟灾难。

```
python@redis:$ sudo killall -9 redis-server
```

我们到Slave中查看状态：

```
redis 127.0.0.1:6379> info
...
...

used_memory:33047696
used_memory_human:31.52M
used_memory_rss:34775040
used_memory_peak:33064400
used_memory_peak_human:31.53M
...

master_host:10.6.1.143
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
...

# Keyspace
db0:keys=1,expires=0,avg_ttl=0
db1:keys=250000,expires=0
```

可以看到 `master_link_status` 的状态已经是down了，Master已经不可访问了。而此时，Slave依然运行良好，并且保留有AOF与RDB文件。

下面我们将通过Slave上保存好的AOF与RDB文件来恢复Master上的数据。

> 首先，将*Slave*上的同步状态取消，避免主库在未完成数据恢复前就重启，进而直接覆盖掉从库上的数据，导致所有的数据丢失。

如果Redis服务器已经充当从站命令 `SLAVEOF NO ONE` 会关掉复制，转Redis服务器为主

```
redis 127.0.0.1:6379> SLAVEOF NO ONE
OK
```

确认一下已经没有了master相关的配置信息：

```
redis 127.0.0.1:6379> INFO
...
role:master
...
```

在Slave上复制数据文件：

```
python@redis-slave:$ tar cvf data.tar *
appendonly.aof
dump.rdb
```

将data.tar上传到Master上，尝试恢复数据：可以看到Master目录下有一个初始化Slave的数据文件，很小，将其删除。

```
python@redis:$ ls -l
total 4
-rw-r--r-- 1 root root 10 Sep 27 00:40 dump.rdb
python@redis:/opt/redis/data/6379$ sudo rm -f dump.rdb
```

然后解压缩数据文件：

```
python@redis:$ sudo tar xf data.tar
python@redis:$ ls -lh
total 29M
-rw-r--r-- 1 root root 18M Sep 27 01:22 appendonly.aof
-rw-r--r-- 1 root root 12M Sep 27 01:22 dump.rdb
```

启动Master上的Redis；

```
python@redis:$ redis-server /etc/redis/redis.conf
Starting Redis server...
```

查看数据是否恢复：

```
redis 127.0.0.1:6379> INFO
...
db0:...
db1:keys=250000,expires=0
```

可以看到25万条数据已经完整恢复到了Master上。此时，可以放心的恢复Slave的同步设置了

```
redis 127.0.0.1:6379> SLAVEOF 10.6.1.143 6379
OK
```

查看同步状态：

```
redis 127.0.0.1:6379> INFO
...
role:slave
master_host:10.6.1.143
master_port:6379
master_link_status:up
...
```

恢复数据策略

Redis允许同时开启AOF和RDB,既保证了数据安全又使得进行备份等操作十分容易， 那么到底是哪一个文件完成了数据的恢复呢？

实际上，当Redis服务器挂掉时，重启时将按照以下优先级恢复数据到内存：

- 如果只配置AOF,重启时加载AOF文件恢复数据；
- 如果同时 配置了RDB和AOF,启动是只加载AOF文件恢复数据；
- 如果只配置RDB,启动是将加载dump文件恢复数据。
  也就是说，AOF的优先级要高于RDB，这也很好理解，因为AOF本身对数据的完整性保障要高于RDB。

总结

目前的线上环境中，由于数据都设置有过期时间，采用AOF的方式会不太实用，因为过于频繁的写操作会使AOF文件增长到异常的庞大，大大超过了我们实际的数据量，这也会导致在进行数据恢复时耗用大量的时间。

因此，可以在Slave上仅开启Snapshot来进行本地化，同时可以考虑将save中的频率调高一些或者调用一个计划任务来进行定期bgsave的快照存储，来尽可能的保障本地化数据的完整性。

在这样的架构下，如果仅仅是Master挂掉，Slave完整，数据恢复可达到100%。

如果Master与Slave同时挂掉的话，数据的恢复也可以达到一个可接受的程度。

> 原文: *https://piaosanlang.gitbooks.io/redis/content/09_redischi_jiu_hua.html*

10. Redis 性能测试

Redis 性能测试

Redis 性能测试是通过同时执行多个命令实现的。

语法

redis 性能测试的基本命令如下：

```
1. redis-benchmark [option] [option value]
```

实例

- 测试存取大小为100字节的数据包的性能。

```
1. $ redis-benchmark -h 127.0.0.1 -p 6379 -q -d 100
```

```
PING_INLINE: 85910.65 requests per second PING_BULK: 123762.38 requests per second SET: 85763.29 requests per second
GET: 81699.35 requests per second
INCR: 82372.32 requests per second
LPUSH: 83472.46 requests per second
LPOP: 82712.98 requests per second
SADD: 82236.84 requests per second
SPOP: 83963.05 requests per second
LPUSH (needed to benchmark LRANGE): 82850.04 requests per second LRANGE_100 (first 100 elements): 29585.80 requests per
second LRANGE_300 (first 300 elements): 9348.42 requests per second LRANGE_500 (first 450 elements): 7562.58 requests per
second LRANGE_600 (first 600 elements): 6780.58 requests per second MSET (10 keys): 94428.70 requests per second
```

序号	选项	描述	默认值
1	-h	指定服务器主机名	127.0.0.1
2	-p	指定服务器端口	6379
3	-s	指定服务器 socket	
4	-c	指定并发连接数	50
5	-n	指定请求数	10000
6	-d	以字节的形式指定 SET/GET 值的数据大小	2
7	-k	1=keep alive 0=reconnect	1
8	-r	SET/GET/INCR 使用随机 key, SADD 使用随机值	
9	-P	通过管道传输 <numreq> 请求	1
10	-q	强制退出 redis。仅显示 query/sec 值	
11	—csv	以 CSV 格式输出	
12	-l	生成循环，永久执行测试	

13	**-t**	仅运行以逗号分隔的测试命令列表。
14	**-I**	Idle 模式。仅打开 N 个 idle 连接并等待。

- 100个并发连接,100000个请求,检测host为localhost 端口为6379的redis服务器性能

```
1. $ redis-benchmark -h 127.0.0.1 -p 6379 -c 100 -n 100000
```

```
====== PING_INLINE ======
100000 requests completed in 0.83 seconds 100 parallel clients

3 bytes payload
keep alive: 1
98.95% <= 1 milliseconds
100.00% <= 1 milliseconds
120192.30 requests per second
====== PING_BULK ======
100000 requests
completed in 0.82 seconds
100 parallel clients
3 bytes payload
keep alive: 1
100.00% <= 0 milliseconds
121506.68 requests per second
====== SET ======
100000 requests
completed in 0.82 seconds
100 parallel clients
3 bytes payload
keep alive: 1
99.80% <= 1 milliseconds
100.00% <= 1 milliseconds
122249.38 requests per second
====== GET ======
100000 requests
completed in 0.81 seconds
100 parallel clients
3 bytes payload
keep alive: 1
99.79% <= 1 milliseconds
100.00% <= 1 milliseconds
122699.39 requests per second
====== INCR ======
100000 requests
completed in 0.81 seconds
100 parallel clients
3 bytes payload
keep alive: 1
99.95% <= 1 milliseconds
100.00% <= 1 milliseconds
124223.60 requests per second
====== LPUSH ======
100000 requests
completed in 0.82 seconds
100 parallel clients
3 bytes payload
keep alive: 1
99.82% <= 1 milliseconds
100.00% <= 1 milliseconds
122100.12 requests per second
====== LPOP ======
```

```
100000 requests
completed in 1.30 seconds
100 parallel clients
3 bytes payload
keep alive: 1
99.93% <= 1 milliseconds
100.00% <= 1 milliseconds
77160.49 requests per second
====== SADD ======
100000 requests
completed in 0.88 seconds
100 parallel clients
3 bytes payload
keep alive: 1
99.81% <= 1 milliseconds
99.97% <= 2 milliseconds
100.00% <= 2 milliseconds
113895.21 requests per second
====== SPOP ======
100000 requests completed in 0.82 seconds 100 parallel clients
3 bytes payload
keep alive: 1
99.78% <= 1 milliseconds
100.00% <= 1 milliseconds
121802.68 requests per second
====== LPUSH (needed to benchmark LRANGE) ====== 100000 requests
completed in 0.81 seconds
100 parallel clients
3 bytes payload
keep alive: 1
99.09% <= 1 milliseconds
99.94% <= 2 milliseconds
100.00% <= 2 milliseconds
122850.12 requests per second
====== LRANGE_100 (first 100 elements) ======
100000 requests
completed in 2.13 seconds
100 parallel clients
3 bytes payload
keep alive: 1
28.64% <= 1 milliseconds
99.65% <= 2 milliseconds
99.97% <= 3 milliseconds
100.00% <= 3 milliseconds
47036.69 requests per second
```

```
====== LRANGE_300 (first 300 elements) ======
100000 requests completed
in 5.46 seconds 100 parallel clients
3 bytes payload keep alive: 1
0.01% <= 1 milliseconds
0.50% <= 2 milliseconds
82.99% <= 3 milliseconds
99.11% <= 4 milliseconds
99.75% <= 5 milliseconds
99.95% <= 6 milliseconds
100.00% <= 6 milliseconds
18325.09 requests per second
====== LRANGE_500 (first 450 elements) ======
100000 requests
completed in 7.94 seconds
100 parallel clients
```

```
3 bytes payload
keep alive: 1
0.01% <= 1 milliseconds
0.10% <= 2 milliseconds
3.03% <= 3 milliseconds
58.57% <= 4 milliseconds
93.27% <= 5 milliseconds
99.03% <= 6 milliseconds
99.53% <= 7 milliseconds
99.72% <= 8 milliseconds
99.77% <= 9 milliseconds
99.82% <= 10 milliseconds
99.85% <= 11 milliseconds
99.88% <= 12 milliseconds
99.94% <= 13 milliseconds
99.97% <= 14 milliseconds
99.98% <= 15 milliseconds
99.99% <= 16 milliseconds
100.00% <= 17 milliseconds
100.00% <= 17 milliseconds
12600.81 requests per second
====== LRANGE_600 (first 600 elements) ======
100000 requests
completed in 10.34 seconds
100 parallel clients
3 bytes payload
keep alive: 1
0.00% <= 1 milliseconds
0.01% <= 2 milliseconds
0.10% <= 3 milliseconds
6.40% <= 4 milliseconds
45.93% <= 5 milliseconds
84.86% <= 6 milliseconds
95.54% <= 7 milliseconds
99.47% <= 8 milliseconds
99.81% <= 9 milliseconds
99.94% <= 10 milliseconds
99.99% <= 11 milliseconds
100.00% <= 11 milliseconds
9673.99 requests per second
====== MSET (10 keys) ======
100000 requests
completed in 1.01 seconds
100 parallel clients
3 bytes payload
keep alive: 1
84.16% <= 1 milliseconds
99.59% <= 2 milliseconds
99.85% <= 3 milliseconds
100.00% <= 3 milliseconds
99206.34 requests per second
```

> 原文：*https://piaosanlang.gitbooks.io/redis/content/10_redisxing_neng_ce_shi.html*

11. Python 客户端

Redis 客户端API

http://redis.cn/clients.html

> 原文: *https://piaosanlang.gitbooks.io/redis/content/11_ke_hu_duan_api.html*

11.1. Python 使用 Redis

Python 使用 Redis

参考文档:

http://redis.cn/clients.html#python

https://github.com/andymccurdy/redis-py

安装Redis

```
sudo pip install redis
```

简单的redis操作

字符串string操作

```
In [1]: import redis

In [2]: r = redis.StrictRedis(host='localhost', port=6379, db=0, password='foobared')

In [3]: r.set('foo', 'bar')
Out[3]: True

In [4]: r.get('foo')
Out[4]: 'bar'

In [5]: r['foo']
Out[5]: 'bar'

In [6]: r.delete('foo')
Out[6]: 1

In [7]: r.get('foo')


In [8]:r.sadd('setmy','a')
Out[8]: 1

In [10]: r.s('setmy')
r.sadd                          r.setex
r.save                          r.setnx
r.scan                          r.setrange
r.scan_iter                     r.shutdown
r.scard                         r.sinter
r.script_exists                 r.sinterstore
r.script_flush                  r.sismember
```

```
r.script_kill                       r.slaveof
r.script_load                       r.slowlog_get
r.sdiff                             r.slowlog_len
r.sdiffstore                        r.slowlog_reset
r.sentinel                          r.smembers
r.sentinel_get_master_addr_by_name  r.smove
r.sentinel_master                   r.sort
r.sentinel_masters                  r.spop
r.sentinel_monitor                  r.srandmember
r.sentinel_remove                   r.srem
r.sentinel_sentinels                r.sscan
r.sentinel_set                      r.sscan_iter
r.sentinel_slaves                   r.strlen
r.set                               r.substr
r.set_response_callback             r.sunion
r.setbit                            r.sunionstore

In [10]: r.smembers('setmy')
Out[10]: {'a'}
```

pipeline操作

管道（pipeline）是redis在提供单个请求中缓冲多条服务器命令的基类的子类。它通过减少服务器-客户端之间反复的TCP数据库包，从而大大提高了执行批量命令的功能。

```
>>> p = r.pipeline()        --创建一个管道
>>> p.set('hello','redis')
>>> p.sadd('faz','baz')
>>> p.incr('num')
>>> p.execute()
[True, 1, 1]
>>> r.get('hello')
'redis'
```

管道的命令可以写在一起，如：

```
>>> p.set('hello','redis').sadd('faz','baz').incr('num').execute()
1
```

默认的情况下，管道里执行的命令可以保证执行的原子性，执行 `pipe = r.pipeline(transaction=False)` 可以禁用这一特性。

字符串应用场景 – 页面点击数

假定我们对一系列页面需要记录点击次数。例如论坛的每个帖子都要记录点击次数，而点击次数比回帖的次数的多得多。如果使用关系数据库来存储点击，可能存在大量的行级锁争用。所以，点击数的增加使用redis的INCR命令最好不过了。

当redis服务器启动时，可以从关系数据库读入点击数的初始值（1237这个页面被访问了34634次）

```
>>> r.set("visit:1237:totals",34634)
True
```

每当有一个页面点击，则使用INCR增加点击数即可。

```
>>> r.incr("visit:1237:totals")
34635
>>> r.incr("visit:1237:totals")
34636
```

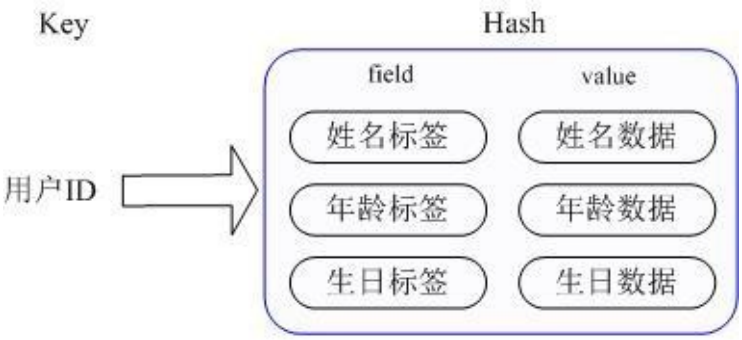
页面载入的时候则可直接获取这个值

```
>>> r.get ("visit:1237:totals")
'34636'
```

使用hash类型保存多样化对象

应用场景

比如我们要存储一个用户信息对象数据，用户的姓名、年龄、生日等，修改某一项的值。Redis的Hash结构可以使像在数据库中 `Update` 一个属性一样只修改某一项属性值。



Redis的Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，如下图：

```
>>> r.hset('users:jdoe',  'name', "John Doe")
1L
>>> r.hset('users:jdoe', 'age', 25)
1L
>>> r.hset('users:jdoe',  'birthday', '19910101')
1L
>>> r.hgetall('users:jdoe')
{'age': '26', 'birthday': '19910101', 'name': 'John Doe'}
>>> r.hkeys('users:jdoe')
['name', 'age', 'birthday']

>>> r.hincrby('users:jdoe', 'age', 1)
26L
>>> r.hgetall('users:jdoe')
>>> {'age': '26', 'birthday': '19910101', 'name': 'John Doe'}
```

Set集合应用场景 - 社交圈子数据

在社交网站中，每一个圈子(circle)都有自己的用户群。通过圈子可以找到有共同特征（比如某一体育活动、游戏、电影等爱好者）的人。当一个用户加入一个或几个圈子后，系统可以向这个用户推荐圈子中的人。我们定义这样两个圈子,并加入一些圈子成员。

```
>>> r.sadd('circle:game:lol','user:debugo')
1
>>> r.sadd('circle:game:lol','user:leo')
1
>>> r.sadd('circle:game:lol','user:Guo')
1
>>> r.sadd('circle:soccer:InterMilan','user:Guo')
1
>>> r.sadd('circle:soccer:InterMilan','user:Levis')
1
>>> r.sadd('circle:soccer:InterMilan','user:leo')
1
```

获得某一圈子的成员

```
>>> r.smembers('circle:game:lol')
set(['user:Guo', 'user:debugo', 'user:leo'])
```

可以使用集合运算来得到几个圈子的共同成员：

```
>>> r.sinter('circle:game:lol', 'circle:soccer:InterMilan')
set(['user:Guo', 'user:leo'])
>>> r.sunion('circle:game:lol', 'circle:soccer:InterMilan')
set(['user:Levis', 'user:Guo', 'user:debugo', 'user:leo'])
```

推荐游戏 `soccer:InterMilan` ：

```
>>> r.sdiff('circle:game:lol','circle:soccer:InterMilan')
>>> {'user:debugo'}
```

> 原文: *https://piaosanlang.gitbooks.io/redis/content/02day/section2.1.html*

12. redis分布式锁实现

背景

在很多互联网产品应用中，有些场景需要加锁处理，比如：秒杀，全局递增ID，楼层生成等等。大部分的解决方案是基于DB实现的，Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对redis的连接并不存在竞争关系。其次Redis提供一些命令SETNX，GETSET，可以方便实现分布式锁机制。

一、使用分布式锁要满足的几个条件：

- 系统是一个分布式系统（关键是分布式，单机的可以使用ReentrantLock或者synchronized代码块来实现）
- 共享资源（各个系统访问同一个资源，资源的载体可能是传统关系型数据库或者NoSQL）
- 同步访问（即有很多个进程同事访问同一个共享资源。没有同步访问，谁管你资源竞争不竞争）
  二、应用的场景例子

管理后台的部署架构（多台tomcat服务器+redis【多台tomcat服务器访问一台redis】+mysql【多台tomcat服务器访问一台服务器上的mysql】）就满足使用分布式锁的条件。多台服务器要访问redis全局缓存的资源，如果不使用分布式锁就会出现问题。 看如下伪代码：

```
1. long N=0L;//N从redis获取值if(N<5){N++；//N写回redis}
```

上面的代码主要实现的功能：

从redis获取值N，对数值N进行边界检查，自加1，然后N写回redis中。 这种应用场景很常见，像秒杀，全局递增ID、IP访问限制等。以IP访问限制来说，恶意攻击者可能发起无限次访问，并发量比较大，分布式环境下对N的边界检查就不可靠，因为从**redis**读的**N**可能已经是脏数据。传统的加锁的做法（如**java**的**synchronized**和**Lock**）也没用，因为这是分布式环境，这个同步问题的救火队员也束手无策。在这危急存亡之秋，分布式锁终于有用武之地了。

分布式锁可以基于很多种方式实现，比如zookeeper、redis…。不管哪种方式，他的基本原理是不变的：用一个状态值表示锁，对锁的占用和释放通过状态值来标识。

这里主要讲如何用redis实现分布式锁。

三、使用redis的setNX命令实现分布式锁

- 、实现的原理
  Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系。redis的SETNX命令可以方便的实现分布式锁。

2、基本命令解析

1）setNX（SET if Not eXists）

语法：

```
1. SETNX key value
```

> 将 key 的值设为 value ，当且仅当 key 不存在。
>
> 若给定的 key 已经存在，则 SETNX 不做任何动作。
>
> SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写
>
> 返回值:
>
> 设置成功，返回 1 。
>
> 设置失败，返回 0 。
>
> 例子:
>
> ```
> 1. redis> EXISTS job                # job 不存在(integer) 0redis> SETNX job "programmer"    # job 设置成功
> (integer) 1redis> SETNX job "code-farmer"   # 尝试覆盖 job ，失败(integer) 0redis> GET job
> # 没有被覆盖"programmer"
> ```

> 所以我们使用执行下面的命令
>
> ```
> 1. SETNX lock.foo <current Unix time + lock timeout + 1>
> ```

- 如返回1，则该客户端获得锁，把lock.foo的键值设置为时间值表示该键已被锁定，该客户端最后可以通过DEL lock.foo来释放该锁。
- 如返回0，表明该锁已被其他客户端取得，这时我们可以先返回或进行重试等对方完成或等待锁超时。

2）getSET

语法:

```
1. GETSET key value
```

将给定 key 的值设为 value，并返回 key 的旧值(old value)。

当 key 存在但不是字符串类型时，返回一个错误。

返回值:

返回给定 key 的旧值。

```
1.     当 key 没有旧值时，也即是， key 不存在时，返回 nil 。
```

3）get

语法:

```
1. GET key
```

返回值：

当 key 不存在时，返回 nil ，否则，返回 key 的值。

```
1.     如果 key 不是字符串类型，那么返回一个错误
```

四、解决死锁

上面的锁定逻辑有一个问题：如果一个持有锁的客户端失败或崩溃了不能释放锁，该怎么解决？

> 1. 我们可以通过锁的键对应的时间戳来判断这种情况是否发生了，如果当前的时间已经大于lock.foo的值，说明该锁已失效，可以被重新使用。

发生这种情况时，可不能简单的通过DEL来删除锁，然后再SETNX一次（讲道理，删除锁的操作应该是锁拥有这执行的，这里只需要等它超时即可），当多个客户端检测到锁超时后都会尝试去释放它，这里就可能出现一个竞态条件，让我们模拟一下这个场景：

> 1. C0操作超时了，但它还持有着锁，C1和C2读取lock.foo检查时间戳，先后发现超时了。 C1 发送DEL lock.foo C1 发送SETNX lock.foo 并且成功了。 C2 发送DEL lock.foo C2 发送SETNX lock.foo 并且成功了。 这样一来，C1，C2都拿到了锁！问题大了！

幸好这种问题是可以避免的，让我们来看看C3这个客户端是怎样做的：

> 1. C3发送SETNX lock.foo 想要获得锁，由于C0还持有锁，所以Redis返回给C3一个0 C3发送GET lock.foo 以检查锁是否超时了，如果没超时，则等待或重试。 反之，如果已超时，C3通过下面的操作来尝试获得锁： GETSET lock.foo <current Unix time + lock timeout + 1> 通过GETSET，C3拿到的时间戳如果仍然是超时的，那就说明，C3如愿以偿拿到锁了。

如果在C3之前，有个叫C4的客户端比C3快一步执行了上面的操作，那么C3拿到的时间戳是个未超时的值，这时，C3没有如期获得锁，需要再次等待或重试。留意一下，尽管C3没拿到锁，但它改写了C4设置的锁的超时值，不过这一点非常微小的误差带来的影响可以忽略不计。

注意：为了让分布式锁的算法更稳键些，持有锁的客户端在解锁之前应该再检查一次自己的锁是否已经超时，再去做DEL操作，因为可能客户端因为某个耗时的操作而挂起，操作完的时候锁因为超时已经被别人获得，这时就不必解锁了

六、一些问题

1、为什么不直接使用expire设置超时时间，而将时间的毫秒数其作为value放在redis中？

如下面的方式，把超时的交给redis处理：

```
1. lock(key, expireSec){
2. isSuccess =
3.  setnx key
```

```
4. if
5.  (isSuccess)
6. expire key expireSec
7. }
```

这种方式貌似没什么问题，但是假如在setnx后，redis崩溃了，expire就没有执行，结果就是死锁了。锁永远不会超时。

2、为什么前面的锁已经超时了，还要用getSet去设置新的时间戳的时间获取旧的值，然后和外面的判断超时时间的时间戳比较呢？

```
String currentValueStr = this.get(lockKey); //redis里的时间
if (currentValueStr != null && Long.parseLong(currentValueStr) < System.currentTimeMillis()) {
    //判断是否为空，不为空的情况下，如果被其他线程设置了值，则第二个条件判断是过不去的
    // lock is expired

    String oldValueStr = this.getSet(lockKey, expiresStr);
    //获取上一个锁到期时间，并设置现在的锁到期时间，
    //只有一个线程才能获取上一个线上的设置时间，因为jedis.getSet是同步的
    if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {
        //防止误删了他人的锁

        //[分布式的情况下]:如过这个时候，多个线程恰好都到了这里，但是只有一个线程的设置值和当前值相同，他才有权利获取锁
        // lock acquired
        locked = true;
        return true;
    }
}
```

因为是分布式的环境下，可以在前一个锁失效的时候，有两个进程进入到锁超时的判断。如：

C0超时了，还持有锁,C1/C2同时请求进入了方法里面

C1/C2获取到了C0的超时时间

C1使用getSet方法

C2也执行了getSet方法

假如我们不加 oldValueStr.equals(currentValueStr) 的判断，将会C1/C2都将获得锁，加了之后，能保证C1和C2只能一个能获得锁，一个只能继续等待。

注意：这里可能导致超时时间不是其原本的超时时间，**C1**的超时时间可能被**C2**覆盖了，但是他们相差的毫秒及其小，这里忽略了。

http://www.cnblogs.com/0201zcr/p/5942748.html

http://blog.csdn.net/ugg/article/details/41894947

> 原文: *https://piaosanlang.gitbooks.io/redis/content/redisfen-bu-shi-suo-shi-xian.html*

13. 总结

总结与建议

总结

1.Redis使用最佳方式是全部数据in-memory。

2.Redis更多场景是作为Memcached的替代者来使用。

3.当需要除key/value之外的更多数据类型支持时,使用Redis更合适。

4.当存储的数据不能被剔除时,使用Redis更合适。

建议

1.批量处理:

redis在处理数据时,最好是要进行批量处理,将一次处理1条数据改为多条,性能可以成倍提高。测试的目的就是要弄清楚批量和 非批量处理之间的差别,性能差异非常大,所以在开发过程中尽量使用批量处理,即每次发送多条数据,以抵消网络速度影响。

2.网络:

redis在处理时受网络影响非常大, 所以, 部署最好能在本机部署, 如果本机部署redis, 能获取10到20倍的性能。集群情况下, 网络硬件、网速要求一定要高。

3.内存:

如果没有足够内存, linux可能将reids一部分数据放到交换分区, 导致读取速度非常慢导致超时。所以一定要预留足够多的内存供 redis使用。

4.少用 `get/set` , 多用 `hashset`

作为一个key value存在,很多开发者自然的使用 `set/get` 方式来使用 `Redis` ,实际上这并不是最优化的使用方法。尤其在未启用VM情况下,Redis全部数据需要放入内存,节约内存尤其重要。

假如一个 `key-value` 单元需要最小占用512字节,即使只存一个字节也占了512字节。这时候就有一个设计模式, 可以把key复用,几个key-value放入一个key中,value再作为一个set存入,这样同样512字节就会存放10-100倍的容量。

这就是为了节约内存,建议使用hashset而不是 `set/get` 的方式来使用 `Redis` 。

命令速查

http://doc.redisfans.com

http://redis.cn

相关文档

《The Little Redis Book中文版》

https://github.com/JasonLai256/the-little-redis-book

《redis官方文档翻译》

http://www.iteye.com/blogs/subjects/redis3

《为什么使用 Redis及其产品定位》

http://www.infoq.com/cn/articles/tq-why-choose-redis/

> 原文: *https://piaosanlang.gitbooks.io/redis/content/99_zong_jie.html*

14. 作业

作业

1. Redis是什么、特点、优势

Redis是一个开源的使用C语言编写、开源、支持网络、可基于内存亦可持久化的日志型、高性能的Key-Value数据库，并提供多种语言的API。

它通常被称为 数据结构服务器 ，因为值（value）可以是 字符串(String)、哈希(Map)、 列表(list)、集合(sets) 和 有序集合(sorted sets)等类型。

Redis 与其他 key - value 缓存产品有以下三个特点:

- Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。
- Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
- Redis支持数据的备份，即master-slave模式的数据备份。
  Redis优势
- 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
- 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
- 原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。
- 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

2. redis安装（Linux）、启动、退出、设置密码、远程连接

2.1 安装redis

下载redis安装包（如: redis-2.8.17.tar.gz ）

```
1. tar -zxvf redis-2.8.17.tar.gz
2. cd redis-2.8.17
3. make
4. sudo make install
```

2.2 后台启动服务端

```
1. nohup redis-server &
```

注 ：redis-server默认启动端口是6379，没有密码

如果不使用默认配置文件，启动时可以加上配置文件

```
nohup redis-server  ~/soft/redis-2.8.17/redis.conf &
```

2.3 启动客户端、验证

```
127.0.0.1:6379> ping
PONG

127.0.0.1:6379> set var "hello world"
OK
127.0.0.1:6379> get var
"hello world"
```

2.4 退出

关闭redis-server

```
redis-cli shutdown
```

例子

```
$ps -ef | grep redis
root     23422 19813  0 10:59 pts/5    00:00:08 redis-server *:6379

$sudo redis-cli  shutdown
[23422] 05 Mar 12:11:29.301 # User requested shutdown...
[23422] 05 Mar 12:11:29.301 * Saving the final RDB snapshot before exiting.
[23422] 05 Mar 12:11:29.314 * DB saved on disk
[23422] 05 Mar 12:11:29.314 # Redis is now ready to exit, bye bye...
[1]+  Done                    sudo redis-server  (wd: ~/soft/redis-2.10.3)
(wd now: ~/soft/redis-2.8.17)

$ps -ef | grep redis
wzh94434 30563 19813  0 12:11 pts/5    00:00:00 grep redis
```

注 ：如果设置上 密码 后，单纯的redis-cli是关不掉的，必须加上ip、port、passwd

```
sudo redis-cli -h host -p port -a passwd shutdown
```

退出客户端

```
localhost:6379> QUIT
```

2.5 设立密码

打开redis.conf找到requirepass，去掉默认，修改

```
requirepass yourpassword
```

验证密码的正确性

```
localhost:6379> auth jihite
OK
```

2.6 远程连接

需要已经安装redis，可以使用redis-cli命令

```
redis-cli -h host -p port -a password
```

2.7 查看redis-server统计信息

4. Redis数据类型

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

4.1 String（字符串）

- 是Redis最基本的数据类型，可以理解成与Memcached一模一样的类型，一个key对应一个value
- 二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象
- 一个键最大能存储 **512MB**
  例子

```
127.0.0.1:6379> set var "String type"
OK
127.0.0.1:6379> get var
"String type"
```

说明：利用set给变量var赋值“String type”；利用get获得变量var的值

4.2 Hash（哈希）

- 是一个键值对集合
- 是一个string类型的field和value的映射表，hash特别适合用于存储对象
  例子

```
127.0.0.1:6379> HMSET var:1  name jihite school pku
OK

127.0.0.1:6379> HGETALL var:1
1) "name"
2) "jihite"
```

```
3) "school"
4) "pku"
```

说明

var:1是键值，每个 hash 可以存储 2 ^{32} - 1 键值对（40多亿）

HMSET用于建立hash对象，HGETALL用于获取hash对象

4.3 LIST（列表）

例子

```
127.0.0.1:6379> lpush lvar 1
(integer) 1
127.0.0.1:6379> lpush lvar a
(integer) 2
127.0.0.1:6379> lpush lvar ab
(integer) 3

127.0.0.1:6379> lrange lvar 0 1
1) "ab"
2) "a"
127.0.0.1:6379> lrange lvar 0 10
1) "ab"
2) "a"
3) "1"
127.0.0.1:6379> lrange lvar 2 2
1) "1"
```

说明

lpush往列表的前边插入；lrange后面的数字是范围（闭区间）

列表最多可存储 2 ^{32} - 1 元素 (4294967295, 每个列表可存储40多亿)。

4.4 Set(集合)

Redis的Set是string类型的无序集合。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)

例子

```
127.0.0.1:6379> sadd setvar redis
(integer) 1
127.0.0.1:6379> sadd setvar mongodb
(integer) 1
127.0.0.1:6379> sadd setvar mongodb
(integer) 0
127.0.0.1:6379> sadd setvar rabbitmq
```

```
(integer) 1
127.0.0.1:6379> smembers setvar
1) "rabbitmq"
2) "redis"
3) "mongodb"
```

说明

set往集合中插入元素，smembers列举出集合中的元素

成功插入返回1；错误插入返回0，例子中mongodb第二次插入时，因已经存在，故插入失败。

4.5 zset(sorted sete: 有序集合)

zset和set一样也是String类型的集合，且不允许元素重复

zset和set不同的地方在于zset关联一个double类型的分数，redis通过分数对集合中的元素排序

zset的元素是唯一的，但是分数是可以重复的

例子

```
127.0.0.1:6379> zadd zvar 1 redis
(integer) 1
127.0.0.1:6379> zadd zvar 1 redis
(integer) 0
127.0.0.1:6379> zadd zvar 2 redis
(integer) 0
127.0.0.1:6379>
127.0.0.1:6379> zadd zvar 2 mongo
(integer) 1
127.0.0.1:6379> zadd zvar 0 rabbitmq
(integer) 1
127.0.0.1:6379>
127.0.0.1:6379> ZRANGEBYSCORE zvar 0 1000
1) "rabbitmq"
2) "mongo"
3) "redis"
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379> zadd zvar -2 celery
(integer) 1
127.0.0.1:6379> ZRANGEBYSCORE zvar 0 1000
1) "rabbitmq"
2) "mongo"
3) "redis"
127.0.0.1:6379> ZRANGEBYSCORE zvar -3 1000
1) "celery"
2) "rabbitmq"
3) "mongo"
4) "redis"
```

说明

成功插入返回1，否则返回0。插入已存在元素失败—返回0

分数为float（可正、负、0）

5. Redis HyperLogLog

Redis HyperLogLog是用来做基数统计的算法。优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。

在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基 数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。

注 ：因为HyperLogLog只会根据输入元素来计算基数，而不会存储输入元素本身，因此不会返回输入的各个元素。

基数是什么？ 对于["abc", "abc", "2", "3"]，基数是["abc", "2", "3"]，个数是3.

例子

```
localhost:6379> pfadd jsh redis
(integer) 1
localhost:6379> pfadd jsh redis
(integer) 0
localhost:6379> pfadd jsh mongodb
(integer) 1
localhost:6379> pfadd jsh rabbitmq
(integer) 1
localhost:6379> pfcount jsh
(integer) 3
localhost:6379> pfadd jsh2 redis
(integer) 1
localhost:6379> pfadd jsh2 a
(integer) 1
localhost:6379> pfcount jsh2
(integer) 2

localhost:6379> pfmerge jsh jsh2
OK
localhost:6379> pfcount jsh
(integer) 4
localhost:6379> pfcount jsh2
(integer) 2
```

说明:

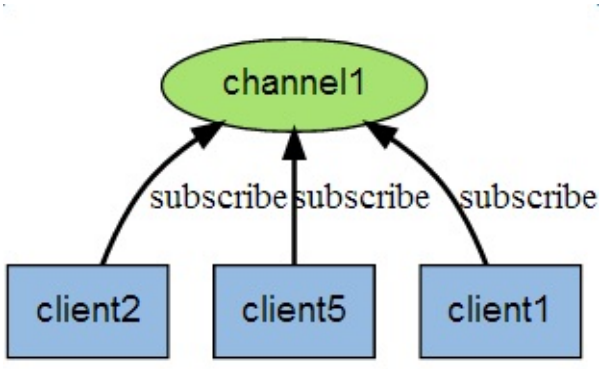
- pfadd key ele [ele2 …]: 添加指定元素到HyperLogLog中，
- pfcount key: 返回给定HyperLogLog的基数估算值
- pfmerge destkey srckey [srckey2….]: 讲多个HyperLogLog合并到一个第一个HyperLogLog中

6. Redis 发布订阅

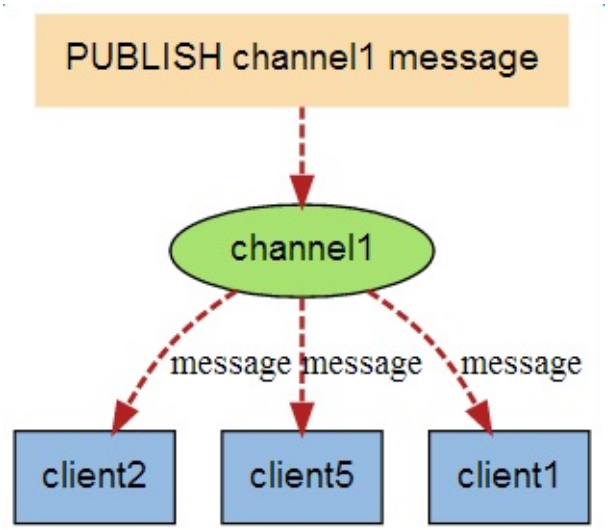
Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

原理：下图展示了三个客户端client1, client2, client5订阅了频道channel1



当有新消息通过PUBLISH发送给channel1时，这时候channel1就会把消息同时发布给订阅者



例子

创建订阅频道redisChat

```
1. localhost:6379> subscribe redisChat
2. Reading messages... (press Ctrl-C to quit)
3. 1) "subscribe"
4. 2) "redisChat"
5. 3) (integer) 1
```

打开几个客户端，订阅channel redisChat

```
1. localhost:6379> psubscribe redisChat
2. Reading messages... (press Ctrl-C to quit)
3. 1) "psubscribe"
```

```
2) "redisChat"
3) (integer) 1
```

然后给channel redisChat发送消息“Hello World”

```
localhost:6379> publish redisChat "Hello World"
(integer) 1
```

客户端会收到消息

```
Reading messages... (press Ctrl-C to quit)
1) "pmessage"
2) "redisChat"
3) "redisChat"
4) "Hello World"
```

7. Redis事务

事务是一个单独的操作集合，事务中的命令有顺序，是一个原子操作（事务中的命令要么全部执行，要么全部不执行），执行一个事务中的命令时不会被其他命令打断。

一个事务从开始到结束经过以下三个阶段：

- 开始事务
- 命令入队
- 执行事务
  例子

```
localhost:6379> MULTI
OK
localhost:6379> set name jihite
QUEUED
localhost:6379> get name
QUEUED
localhost:6379> sadd language "c++" "python" "java"
QUEUED
localhost:6379> smembers language
QUEUED
localhost:6379> exec
1) OK
2) "jihite"
3) (integer) 3
4) 1) "java"
   2) "python"
   3) "c++"
```

说明：事务以MULTI开始，以EXEC结束

8. Redis脚本

Redis 脚本使用 Lua 解释器来执行脚本。执行脚本的常用命令为 **EVAL** 。基本语法

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

例子

```
localhost:6379> EVAL "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

9. 数据备份与恢复

数据备份

```
localhost:6379> save
OK
```

改命令会在redis的安装目录中创建文件dump.rdb，并把数据保存在该文件中。

查看redis的安装目录

```
localhost:6379> config get dir
1) "dir"
2) "/home/jihite/soft/redis-2.8.17"
```

数据恢复

只需将备份文件dump.rdb拷贝到redis的安装目录即可。

10. 数据库操作

3
Redis中，一共有16个数据库，分别是0~15，一般情况下，进入数据库默认编号是0，如果我们要进入指定数据库，可以用select语句

切换到编号为3的数据库

```
localhost:6379> select 3
OK
localhost:6379[3]>
```

查看数据库中所有的键值

```
localhost:6379[1]> set a 1
OK
localhost:6379[1]> set  b 2
OK
localhost:6379[1]> keys *
1) "b"
2) "a"
```

返回当前数据库中所有key的数目: dbsize

删除当前数据库中的所有key: flushdb

清空所有数据库中的所有key: flushall

把当前数据库中的key转移到指定数据库: move a aim_db, 例:

```
localhost:6379[1]> set z sss
OK
localhost:6379[1]> move z 0
(integer) 1
localhost:6379[1]> select 0
OK
localhost:6379> get z
"sss"
```

请用Redis和任意语言实现一段恶意登录保护的代码，限制1小时内每用户Id最多只能登录5次。具体登录函数或功能用空函数即可，不用详细写出。

用列表实现:列表中每个元素代表登陆时间,只要最后的第5次登陆时间和现在时间差不超过1小时就禁止登陆.用Python写的代码如下:

```
!/usr/bin/env python3import redis import sys import timer = redis.StrictRedis(host='127.0.0.1', port=6379, db=0) try: id = sys.argv[1]except: print('input argument error') sys.exit(0)if r.llen(id) >= 5 and time.time() – float(r.lindex(id, 4)) <= 3600: print('you are forbidden logining')else: print('you are allowed to login') r.lpush(id, time.time()) # login_func()
```

> 原文：*https://piaosanlang.gitbooks.io/redis/content/section99.html*

15. 段子集中营

10年编程教会我最重要的10件事

0．“面向对象”比你想象的要难得多

也许只有我有这种想法，不过我曾经以为计算机科学课上学过的“面向对象”是很简单的东西。我的意思是，创建一些类来模拟现实世界能有多难啊？其实，那还真是挺难的。

十年之后，我仍然在学习如何合理地建模。我后悔以前我没有花更多的时间来学习面向对象和设计模式。优秀的建模技术对于每一个开发团队都是非常有价值的。

1．软件开发的难点在于沟通

这里的沟通是指与人的沟通，而不是socket编程。有时你的确会遇上棘手的技术问题，但是这种情况根本不常见。常见的问题在于那些你和项目经理之间的、你和客户之间的、还有你和其他开发者之间的误解。培养你的软技能吧。

2．学会拒绝

当我刚开始工作的时候，我非常急切的想要去讨好别人。这也就是说，我几乎不能去回绝别人对我的要求。我加了很多班，但是还是不能完成他们交代给我的所有事情。结果他们表示不满意，而我也表示要崩溃了。

如果你从不回绝别人，你的答应就显得毫无意义。承担能力所及的事情，如果别人不停地指派给你更多的事情，你需要明确的表示那意味着将会耽误其他的工作。

为了应付这种事情，我会随身携带一张列有待办事项的纸（To-do list）。当人们叫我去做什么事情的时候，我就给他们看这张纸，并且问他们我应该为他们挤掉哪个事情。这是我用来拒绝别人的一种好办法。

3．如果每件事都重要，那就什么事都不重要

我们这一行，总是强调每种特性都是同等重要的，其实并不是这样。敦促你的同事，让他们承担起工作。

如果你不强迫他们选择该做和不该做的事情，你会轻松很多。相反，让他们来为你选择你这周的任务。这会让你生产出来的东西变得最有价值。如果其他的部分都还是乱糟糟的，至少你已经完成了最重要的。

4．不要过度考虑问题

我可以站在白板前面一整天策划事情，但是这并不意味着事情会向更好的方向发展，这仅意味着事情将变得更复杂。

我的意思并不是“你不应该去做任何策划”，只是如果我会在实现程序的时候会很快遇到我没考虑过的问题的话，那为什么我不去尝试把它做好呢？像戴夫·法洛所说的，“魔鬼居住于细节中，而驱走魔鬼的方法是实践，而不是理论”。

5．去钻研一些东西，但不要钻牛角尖

克里斯和我花费了大量的时间钻研SQL服务器的深层部分。那真的很有趣，我也学到了很多知识，但是过了一段时间我意识到，知道了那么多的知识并不能帮助我解决业务上的问题。

举个例子：我知道在数据表层次，SQL服务器不会接受IU锁—它只会接受IX锁。这是一个性能调整，因为在大多数情况下，IU锁都会升级成IX锁。为了了解这些，我花掉了无数天做实验，我读了很多的书，还在会议上向微软的员工了解情况。然而我用过这个知识吗？没有。

6． 了解软件开发系统的其他方面

这对成为一个优秀的开发者是很重要的，但是若要在一个开发软件的系统中成为优秀的一员，你还需要去了解开发系统中剩下的部分在干什么。QA是如何工作的？项目经理在干什么？业务分析员在忙些什么？这些知识会让你与其他员工产生联系，并使你和他们之间的互动顺畅。

向你周围的人寻求帮助，以便学到更多的知识。有什么好书呢？大多数人都会为你的关注而高兴，并且很乐意帮助你。在这上花一点小时间会对你有很大的帮助。

7． 同事是你最好的老师

在我找到第一份工作的一年后，我们和另一所公司合并了。突然之间身边就多出很多聪明又经验丰富的人。我深刻的记得这是我感到多么自卑和愚蠢。我努力地学习，读了一本又一本的书，还是还是赶不上他们。我发现他们和我比起来有非常突出的优势。

现在，我不会因为和优秀的人一起工作而感到难受。我认为我有一生的时间去学习。我提出问题，并且非常努力地去了解我的同事们是怎么做出结论的。这也是为什么我加入了 ThoughtWorks。把你的同事们看成财富，而不是竞争对手。

关于学习，不论是哪个行业，都是永恒的话题，正如 Jonathan Danylko 在总结自己 20 年的编程经验时所说到，“诚然，总有很多你不知道的技术，你可以从中学习以保持不落后。如果你有一种灵巧的方式来获取你需要的新技术，那你每天都应该坚持学习。”（编注：ThoughtWorks是一家全球知名的IT咨询公司。）

8． 做出可用的软件是最终目标

不管你的算法有多酷，不管你的数据库模式有多棒，不管你的什么什么有多么多么好，如果它不能搔到客户的痒处，它就不值一文。专注于做出有用的软件，同时准备继续做出后续软件，这才是正轨。

9． 有些人真的不可理喻

在你身边的大多数人总是很优秀的，你向他们学习，他们也向你学习。共同完成一件事情的感觉总是很好。然而不幸的是，你也有可能遇到例外。因为某些原因，人可能会变得冷漠刻薄。萎靡不振的老板啊，满口谎言的同事啊，无知愚昧的顾客什么的。不要把他们看的太重。尽量避开他们，尽量把他们所带来的痛苦和影响降到最小，但不要自责。只要你保持诚实并且尽力去工作，你就完成了你该做的事情。

> 原文：*https://piaosanlang.gitbooks.io/redis/content/happy/index.html*

15.1. 程序员的样子

程序员的样子

[](#)[](#)[](#)[](#)[](#)[](#)

往运行服务器上直接上传文件时程序员的样子

当老板说项目如果能赶在最后期限前开发完成将会有一笔奖金时程序员的样子

当凌晨3点还在修改bug时程序员的样子

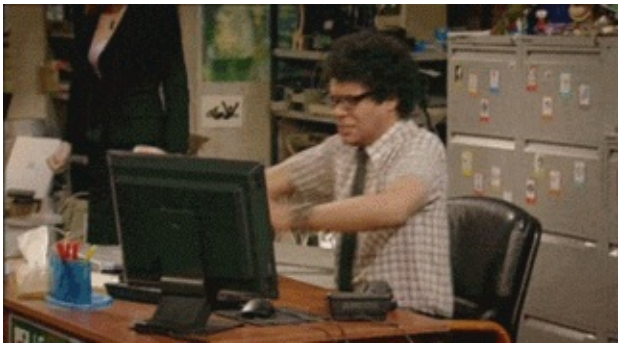
当发现没有按CTRL-S就关闭了文件时程序员的样子

当使用正则表达式返回了想要的结果时程序员的样子

第一次使用CSS美化页面时的效果

当所有人都在办公室挥汗如雨的加班而你却能安然的回家度周末时的样子

当老板想找一个人来修改这个严重bug时程序员的样子



当发现有东西上周五还好用而到了周一不好用了时程序员的样子

当经过了数小时的努力后第一次运行开发出的脚本时程序员的样子

当在没有使用谷歌搜索的情况下就找到了问题解决方案时程序员的样子

做市场的那帮家伙告诉程序员他们是这样销售软件的

> 原文：*https://piaosanlang.gitbooks.io/redis/content/happy/section1.html*

15.2. 要嫁就嫁程序猿

要嫁就嫁程序猿—钱多话少死的早

一、

程序猿问科比：“你为什么这么成功？ ”科比：“你知道洛杉矶凌晨四点是什么样子吗？ ”程序猿：“知道，一般那个时候我还在写代码，怎么了？”科比：“额…….”

二、

女神：你能让这个论坛的人都吵起来，我今晚就跟你走。程序猿：PHP语言是最好的语言！论坛炸锅了，各种吵架。女神：服了你了，我们走吧，你想干啥都行。程序猿：今天不行，我一定要说服他们，PHP语言是最好的语言。

三、

我是一个苦b的程序员，今晚加班到快通宵了，困得快睁不开眼了，女上司很关心，问我要不要吃宵夜。我没好气地说，宵夜就算了，能让我睡一觉就行了。女上司红着脸说了句讨厌啊，然后坐在我身边不动，好像距离我很近，搞得我很紧张，难道她发现我的程序出了bug？

四、

老公一定要找程序员！！！！！！钱多话少死的早

五、

老婆给当程序员的老公打电话：“下班顺路买一斤包子带回来，如果看到卖西瓜的，就买一个。”

当晚，程序员老公手捧一个包子进了家门……

老婆怒道：“你怎么就买了一个包子？！”

老公答曰：“因为看到了卖西瓜的。”

六、

七、

某人发帖子：“各位JR，我想做一个程序猿，请问有什么要注意的……”某猿：“等我下班跟你细说……”

然后……就没有然后了

八、

我问程序员朋友借了1000，他说再多借你24吧，凑个整

九、

程序员A：“我吃鱼香肉丝盖饭，你吃什么?”程序员B：“宫保鸡丁盖饭.”程序员A 在点菜单写上：鱼香肉丝盖饭 1宫保鸡丁盖饭 1程序员B：“我还是要牛肉面吧!”程序员A 更正点菜单:鱼香肉丝盖饭 1// 宫保鸡丁盖饭 1牛肉面 1

十、

我是一个程序猿，一天我坐在路边一边喝水一边苦苦检查bug，这时一个乞丐在我边上坐下了，开始要饭，我觉得他可怜，就给了他一块钱，然后接着调试程序。他可能生意不好，就无聊的看看我在干什么，然后过了一会，他幽幽说，这里少了一个分号。

我惊奇的问：“你也懂这行啊” 乞丐说：“我以前就是做这个的。”

十一、

某程序员退休后决定练习书法，于是重金购买文房四宝。一日，饭后突生雅兴，一番研墨拟纸，并点上上好檀香。定神片刻，泼墨挥毫，郑重地写下一行字：hello world！

十二、

搞IT太辛苦了，想换一行怎么办？”“敲一下Enter键。”

十三、

程序员不喜欢乾隆的第八个儿子，因为八阿哥 bug

十四、

一女同学在食堂吃饭时，一程序猿凑到旁边，“同学，我能和你说话不，我已经一个月没和女生说话了。

十五、

两个程序员在聊天：“我昨天碰到个辣妹。我把她带回家，马上就开始如饥似渴地亲吻，她就坐在我的键盘上，然后……”“你在家里也有台电脑？CPU是什么型号的？”

十六、

程序员的读书历程：x 语言入门 –> x 语言应用实践 –> x 语言高阶编程 –> x 语言的科学与艺术 –> 编程之美 –> 编程之道 –> 编程之禅–> 颈椎病康复指南。

(整理自网络)

> 原文: *https://piaosanlang.gitbooks.io/redis/content/happy/section2.html*

15.3. 程序员心花怒放的七种礼物

能让程序员心花怒放的七种礼物

http://www.open-open.com/news/view/27c2e5

> 原文: *https://piaosanlang.gitbooks.io/redis/content/happy/section3.html*

15.4. 程序员该如何找合租室友？

程序员该如何找合租室友？


第二天
招合租
你说谁呢？
JS一统江湖
PHP是最好的语言！
滚～
《神秘的程序员们》
by 西乔
coderstory

15

0

来自: mp.weixin.qq.com

Udacity 博客园专属学费优惠 ) )

标签: 神秘的程序员们

> 原文: *https://piaosanlang.gitbooks.io/redis/content/happy/section4.html*

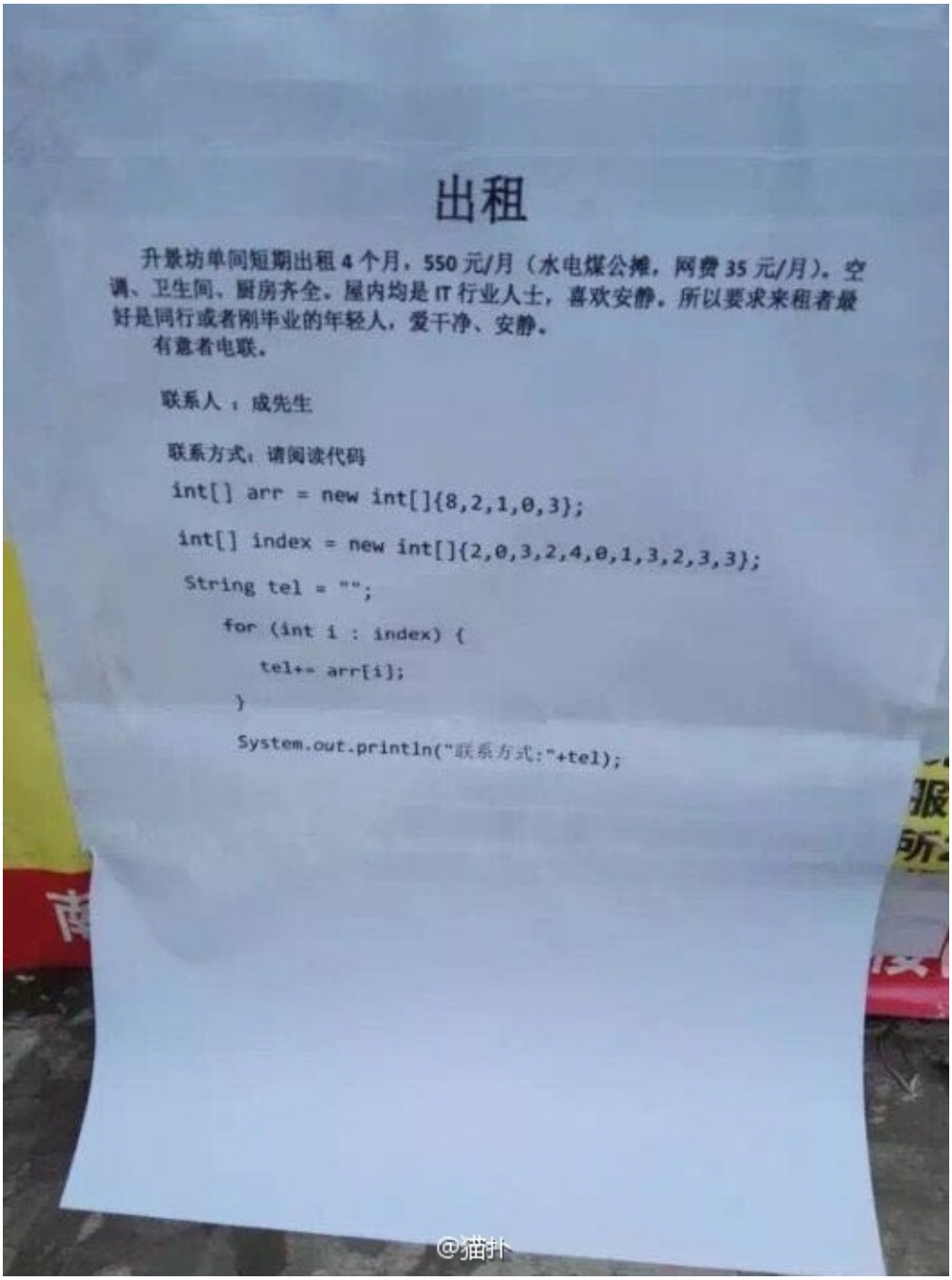
15.5. 程序猿招租广告 手机号竟是一串代码

程序猿招租广告 手机号竟是一串代码

某女：你能让这个论坛的人都吵起来，我就跟你约会。 程序员：PHP 是最好的语言！论坛炸锅了，各种吵架… 某女：服了你了，咱们走吧程序员：今天不行，我一定要说服他们，PHP 必须是最好的语言！！！ 谁说码农没有幽默感？如果你觉得上面这个只是个段子的话，下面这件事也许能让你对程序员的印象有所改观。

日前，有程序员在南京某小区张贴了一份合租广告，表示屋内均是 IT 行业人士，喜欢安静，所以要求来租者最好是同行或者刚毕业的年轻人。

而在联系方式部分，这条广告并没有提供明显的手机号，而是一段代码。。。



不懂没关系，万能的网友给出了解法：

8，2，1，0，3 这几个数分别编号为 0 1 2 3 4（手机号只包含这 5 个数字），然后按照第二行顺序把编号对应的数字写出来就是电话号码。

> 原文：*https://piaosanlang.gitbooks.io/redis/content/happy/section5.html*