redis技术

[1 概述 4](#_Toc3880663)

[2 安装redis 5](#_Toc3880664)

[2.1 windows 5](#_Toc3880665)

[2.2 linux 8](#_Toc3880666)

[2.3 服务器运行时典型的配置 10](#_Toc3880667)

[2.3.1 bind，绑定监听的IP接口 10](#_Toc3880668)

[2.3.2 port，监听的端口 10](#_Toc3880669)

[2.3.3 daemonize，是否以守护进程方式运行 10](#_Toc3880670)

[2.3.4 logfile, 日志文件 11](#_Toc3880671)

[2.3.5 pidfile, 服务器主进程id文件 12](#_Toc3880672)

[2.3.6 databases，数据库的数量 13](#_Toc3880673)

[3 操作redis 14](#_Toc3880674)

[3.1 C/S 14](#_Toc3880675)

[3.2 支持认证 15](#_Toc3880676)

[3.3 key-value型结结构 17](#_Toc3880677)

[3.4 值支持多种类型（结构） 17](#_Toc3880678)

[3.5 Go操作redis 17](#_Toc3880679)

[4 字符串型数据以及属性操作，string 20](#_Toc3880680)

[4.1 # help @string 20](#_Toc3880681)

[4.2 设置相关操作，有效期，addonly，replaceonly 20](#_Toc3880682)

[4.3 获取，GET key 23](#_Toc3880683)

[4.4 删除，del 23](#_Toc3880684)

[4.5 递增，递减 23](#_Toc3880685)

[4.6 追加内容 24](#_Toc3880686)

[4.7 通用 24](#_Toc3880687)

[5 集合Set 26](#_Toc3880688)

[5.1 help @set 26](#_Toc3880689)

[5.2 sadd, set add 26](#_Toc3880690)

[5.3 sismember，判断集合中是否存在某个成员，判断某个成员是否在集合中 27](#_Toc3880691)

[5.4 smembers，获取全部的成员 27](#_Toc3880692)

[5.5 srem，移除成员 27](#_Toc3880693)

[5.6 scard，成员个数 28](#_Toc3880694)

[5.7 spop，弹出成员 28](#_Toc3880695)

[5.8 srandmember随机选择成员 28](#_Toc3880696)

[5.9 smove，移动成员，由A集合移动到B集合 29](#_Toc3880697)

[5.10 交集sinter，并集sunion，差集sdiff 29](#_Toc3880698)

[6 Sorted-Set有序集合（加权（加分）集合） 31](#_Toc3880699)

[6.1 help @sorted\_set 31](#_Toc3880700)

[6.2 ZADD 31](#_Toc3880701)

[6.3 zrange，zrevrange，依据分值升序、降序获取成员 32](#_Toc3880702)

[6.4 zincreby，分值变化，累计变化 33](#_Toc3880703)

[6.5 zcore，确定成员分值 33](#_Toc3880704)

[6.6 zrem，删除集合成员 34](#_Toc3880705)

[6.7 zrangebyscore, zrevrangebyscore 依据分数获取成员 34](#_Toc3880706)

[6.8 zcard，统计集合内的元素个数 34](#_Toc3880707)

[6.9 zcout，统计满足某个分值的元素个数 35](#_Toc3880708)

[7 list，列表类型 36](#_Toc3880709)

[7.1 lpush，rpush 36](#_Toc3880710)

[7.2 lindex，通过索引访问元素 37](#_Toc3880711)

[7.3 lpop，rpop， 37](#_Toc3880712)

[7.4 队列，先进先出，一端进，另一端出 38](#_Toc3880713)

[7.5 栈，先进先出，一端进，同一端出 38](#_Toc3880714)

[7.6 lset，设置某个索引元素的值 38](#_Toc3880715)

[7.7 llen,长度 39](#_Toc3880716)

[7.8 lrange，依据索引查看某个范围 39](#_Toc3880717)

[7.9 blpop, brpop，阻塞弹出 39](#_Toc3880718)

# 概述

官网：

<http://www.redis.net.cn/>

<https://redis.io/>

Redis是一个开源（BSD许可），内存存储的数据结构服务器，可用作数据库，高速缓存和消息队列代理。它支持字符串、哈希表、列表、集合、有序集合，位图，hyperloglogs等数据类型。内置复制、Lua脚本、LRU收回、事务以及不同级别磁盘持久化功能，同时通过Redis Sentinel提供高可用，通过Redis Cluster提供自动分区。

redis主要将数据存储在内存中，同时提供将数据存储在硬盘上的支持！

类比：

数据服务器：MySQL，MongoDB，redis，将数据持久化存储到硬盘上。

内存存储：go的memory，memcache，..

非关系型（非二维表，而是key-value型），缓存型（以内存存储数据为主，支持同步到硬盘）数据库系统。

支持多种数据类型，string（字符串），set（集合），order-set(有序集合)，list（列表），hashmap（哈希表），位图（bitmap），HyperLogLogs（集合统计），streams（流数据）。

基于这些类型完成特定的业务逻辑。

排队（队列），list，解决并发时资源争用问题。并发下订单，一个个的解决，保证库存不超卖。

排行榜（top 10），order-set，来解决

分布式锁定，利用key的nx（不存在才能设置：non-exists）设置完成。

订阅/发布模式。

# 安装redis

## windows

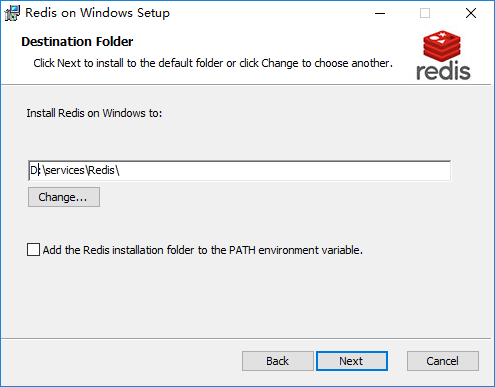
下载：

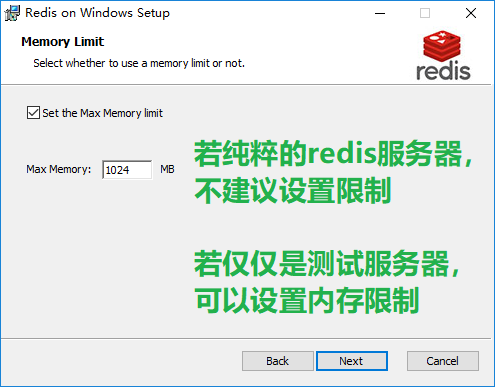
<https://github.com/MicrosoftArchive/redis/releases>

（redis不提供官方的windows版本）

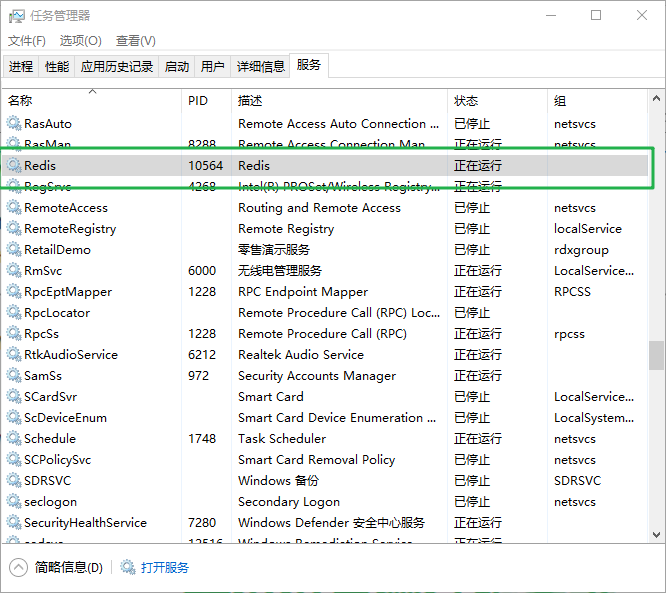


双击安装：



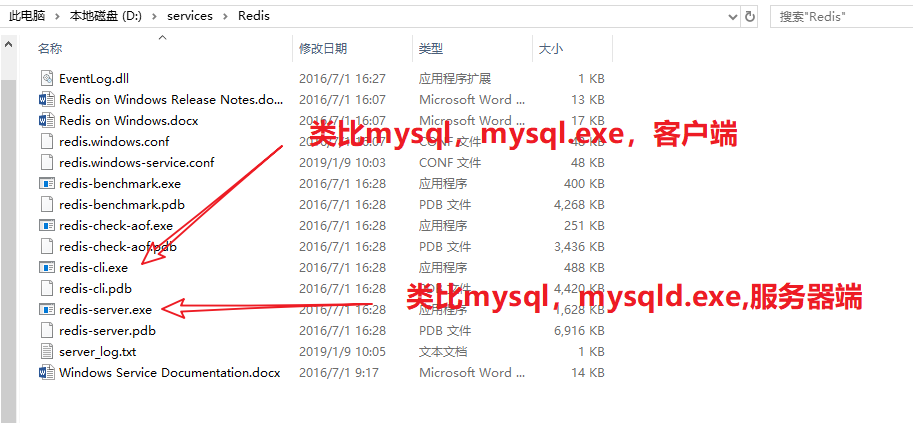


安装完毕后，回自动安装redis的windows服务：



可以服务的方式进管理！

安装目录结构：

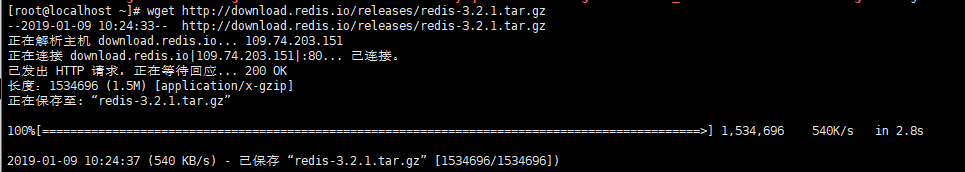


## linux

最典型的redis的运行操作系统！

下载：

wget <http://download.redis.io/releases/redis-3.2.1.tar.gz>



tarball 源码安装方式！

解压，配置编译，编译，安装（源码包中，已经存在部分安装需要的文件，例如Makefile,./configure生成的文件）

进入源码目录，make（编译生成可执行性文件）

# tar zxvf redis-3.2.1.tar.gz

# cd redis-3.2.1

# make

此时，安装结束。不需要执行make install,在编译时，配置生成了最终的程序。（没有生成安装包）

将需要的执行文件，及其其他文件，独立拷贝到合理的位置：/usr/loca/redis

复制或移动可执行文件和配置文件：即可：

# mkdir /usr/local/redis

# cp src/mkreleasehdr.sh src/redis-server src/redis-cli src/redis-benchmark src/redis-check-aof src/redis-check-rdb src/redis-sentinel src/redis-trib.rb /usr/local/redis/

# cp redis.conf sentinel.conf /usr/local/redis/

结果：

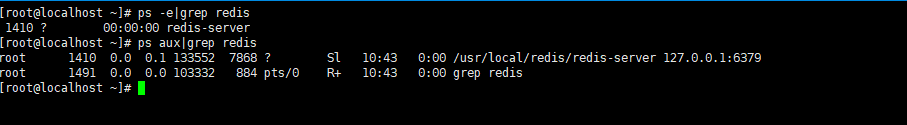


可以选择开启启动redis服务器

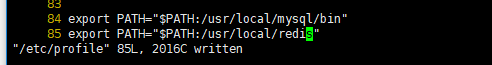
/etc/rc.local



重启测试：



也可以将redis目录加入到环境变量PATH中，可以任意位置redis-cli

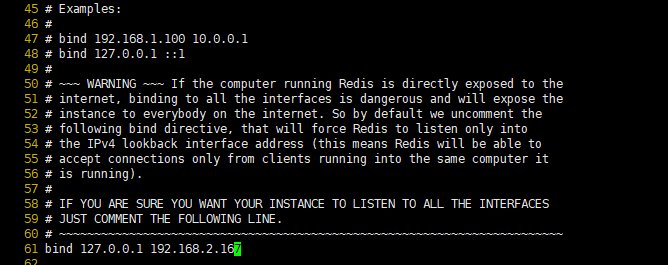


## 服务器运行时典型的配置

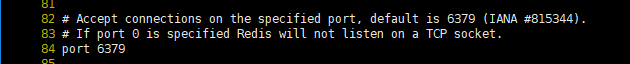
在启动redis服务器时，需要通过配置文件（或者配置参数）的方式完成对服务器的配置。

配置文件，需要在redis启动时进行指定：

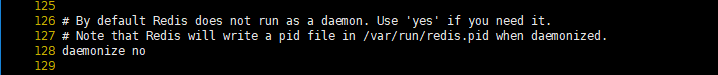
### bind，绑定监听的IP接口



### port，监听的端口

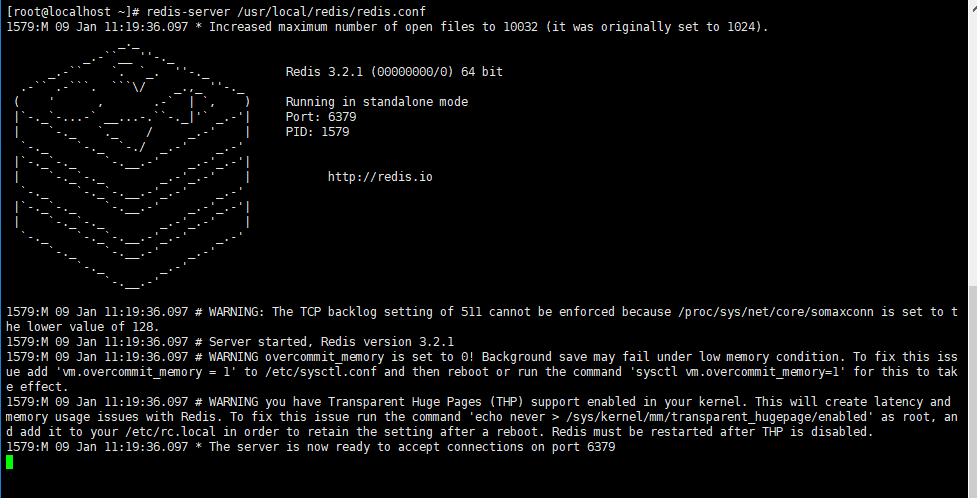


### daemonize，是否以守护进程方式运行

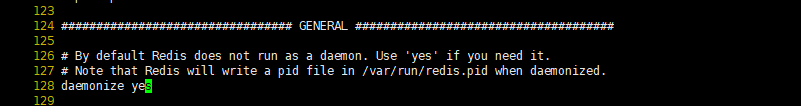


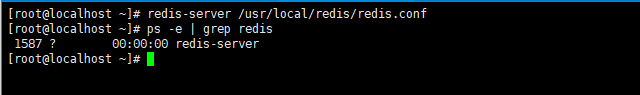
最直观的理解，是是否在后端运行。

非Daemonize模式：占用终端：



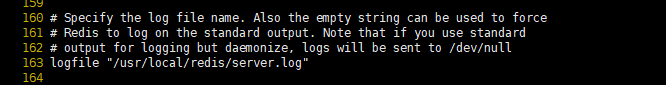
Daemonize模式：



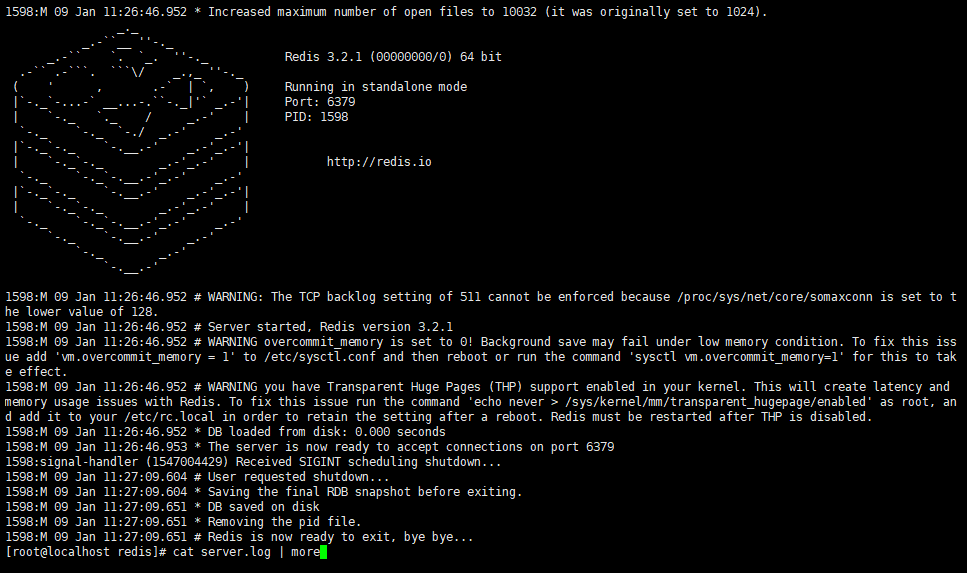


### logfile, 日志文件

在以守护进程方式运行时，通常需要通过日志文件来记录redis服务器的运行状态。



日志内容：



### pidfile, 服务器主进程id文件

用于存储redis服务器的进程ID号。redis.conf



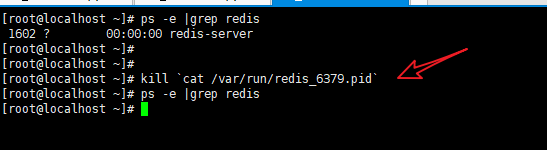
PID：



pidfile:



经常使用该文件，完成redis服务器的管理，例如关闭：



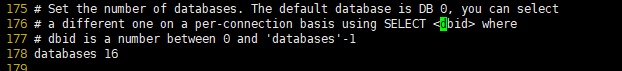
变为通用的命令，不必每次去查询进程ID了。

`` 反引号，标识执行内部命令的意思！

### databases，数据库的数量

redis服务器启动时，支持多少个数据库的数量。

通过索引下标访问。



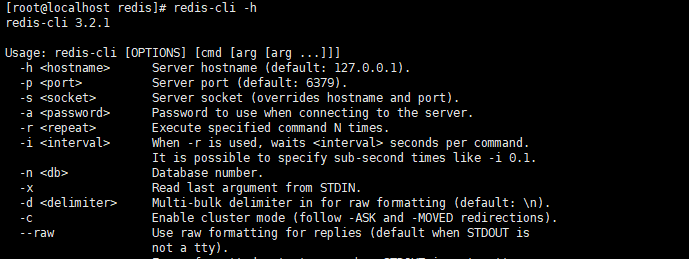
0 - 15 数据库！

# 操作redis

## C/S

使用客户端，去操作redis服务器端！

redis-cli 命令行客户端程序！



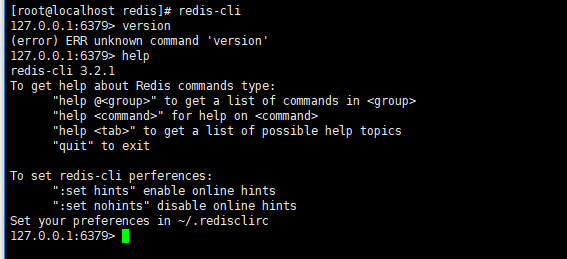
典型的选项：

-h 服务器主机，默认为本机

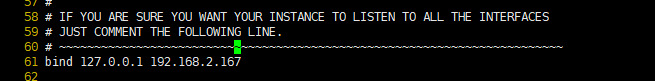
-p 服务器端口，默认为6379

-a 密码

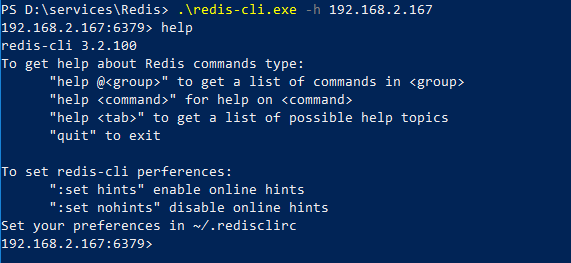
本地默认连接：



远程客户端连接，保证服务器端监听可以远程连接的端口地址：



客户端连接：



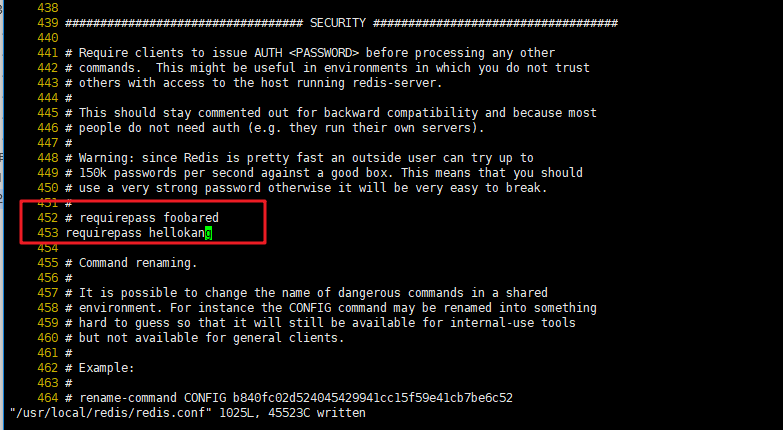
如果是Go操作（或者其他语言操作），程序就作为redis-server 的客户端出现。

## 支持认证

默认没有开启认证，可以直接连接。目的提升效率，因为缓存服务器，通常都是内网服务器。因此通常通过网络配置管理，例如防火墙就可以保证安全，通常没有开启认证机制。

若需要，可以通过配置文件增加认证：

redis.conf 配置 requirepass，可以设置密码。

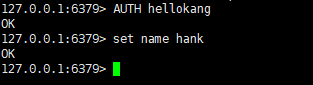


重启服务器后，客户端操作：

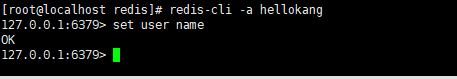


提示， NOAUTH ,未认证！

通过 AUTH 客户端命令，发送认证密码即可



也可以，在连接连接时指定密码：-a



## key-value型结结构

一个键对应一个值。

类似于：Go中的map。

## 值支持多种类型（结构）

key => string

key => Set

key => list

支持的类型：

* 字符串string
* 哈希Hash，类比Go中映射类型
* 列表List，类比Go语言中数组（切片）
* 集合Set，类比Go中的映射类型的key
* 有序集合OrderSet，类比Go中的映射类型的key=>value
* 集合基数HyperLogLog，类似于计数器类型。

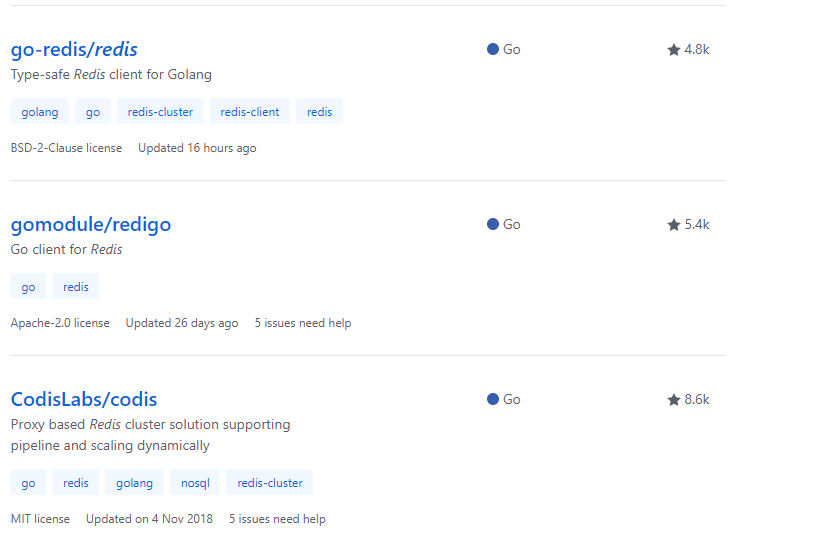
综合起来看：

键， Go中的变量

值， Go中的变量值

## Go操作redis

使用典型包：redigo 操作redis。需要在github上使用



安装：

配置环境变量GOPATH

# go get github.com/gomodule/redigo/redis

基本操作过程：

连接服务器，（延迟关闭连接），去执行命令，处理执行结果



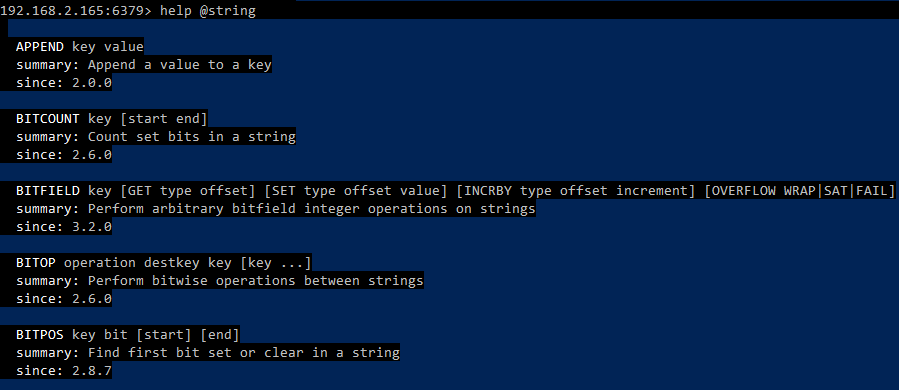
# 字符串型数据以及属性操作，string

典型的数据缓存类型。

如果我们在项目中，使用字符串类型作为缓存。没有附加业务逻辑，没有附加的数据结构，体现的纯粹的数据。

## # help @string

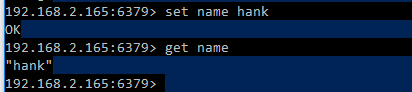
得到关于字符串型数据的操作的动作列表！



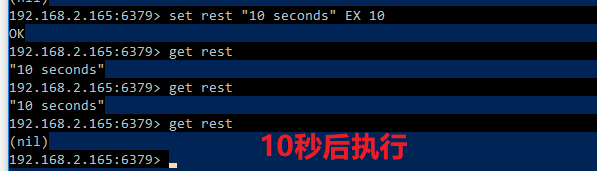
## 设置相关操作，有效期，addonly，replaceonly

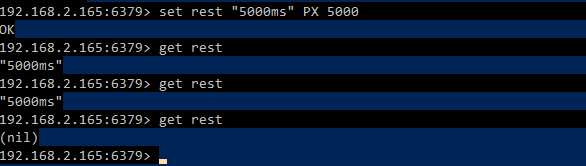
set 命令，仅仅针对字符串数据。

基本用法，SET key value，设置一个永久有效的key



同时设置有效期，支持秒级EX，和毫秒级PX





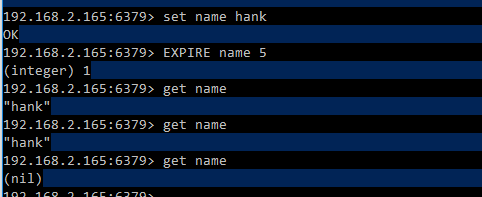
有效期的设置，可以使用命令：SETEX key seconds value，SETPX key seconds value

若在设置的时候没有指定有效期，可以通过命令：

EXPIRE key seconds，秒级单位

PEXPIRE key seconds，毫秒级单位

来完成设置，演示如下：就是说，可以将值和有效期的设置分开来完成！



该命令，针对于所有类型有意义。

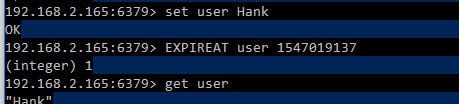
对于有效期的设置，还支持在某个时间点失效的设置，使用命令，时刻的标识方式为时间戳。

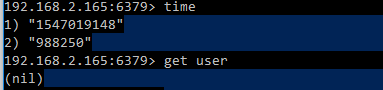
EXPIREAT key timestamp

PEXPIREAT key milliseconds-timestamp

time可以获取服务器时间，返回时间戳，一部分秒，另一部分是毫秒



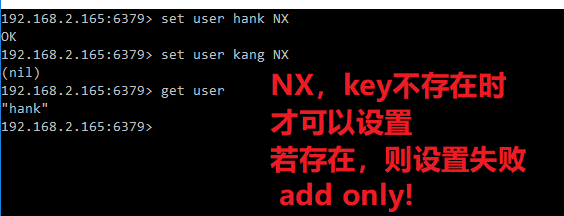




比较适合，每天一个缓存项的操作。例如，统计每天的网站请求数量，key:couter20190109

支持addonly（NX），或replaceonly（XX）操作：

if !is\_exists(key) { set key value}:



NX模式，非常适合做互斥锁定，独占锁定相关逻辑。

if is\_exists(key) { set key value }



有快捷语法：SETNX, 完成set NX

## 获取，GET key

get key

获取响应的数据！

## 删除，del

针对于全部的类型都是用delete完成删除！

del key

## 递增，递减

INCR key，递增

summary: Increment the integer value of a key by one

since: 1.0.0

group: string

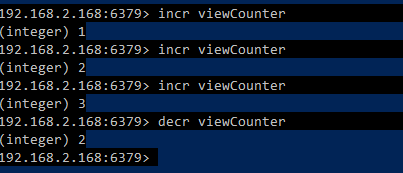
DECR key，递减

summary: Decrement the integer value of a key by one

since: 1.0.0

group: string

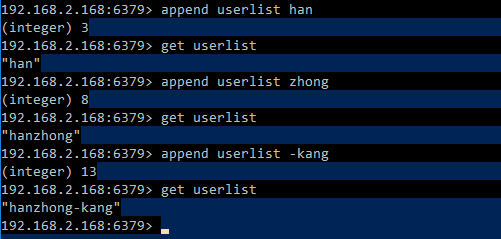
针对于 数值（int）类 数据



非常适合做计数器类工作！统计访问量，操作次数。

## 追加内容

append操作。



value = get key

set key value+”content”

典型逻辑：签到列表，

## 通用

del

expire

expireAt

pexpire

pexpireAt

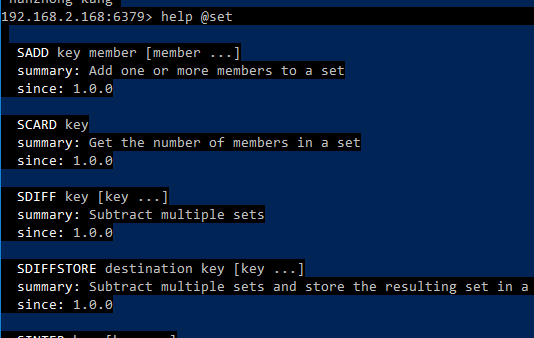
# 集合Set

复合数据类型，一个key，可以存储多个元素（成员）数据值。典型特点：

* 没有重复成员member
* 无序。

（不是redis定义的，典型的集合数据结构就是如此，redis支持集合型数据结构）

## help @set

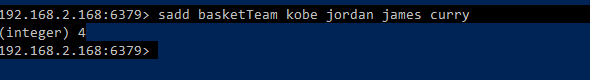


## sadd, set add

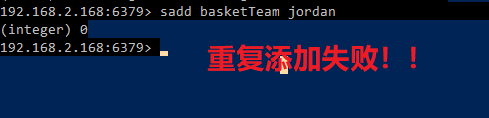
在集合中添加元素。

支持一次性添加一个或者多个成员member。

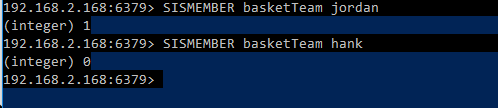
在集合 basketTeam中增加了4个元素：



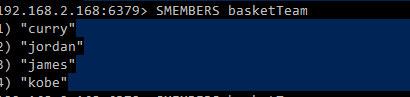
尝试重复添加元素：



## sismember，判断集合中是否存在某个成员，判断某个成员是否在集合中

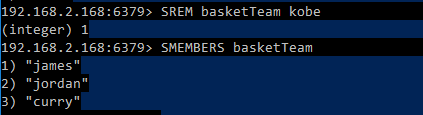


## smembers，获取全部的成员



## srem，移除成员

支持同时移除多个成员。



若需要删除整个集合，使用 del key 操作

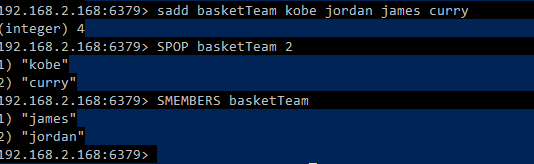
## scard，成员个数



## spop，弹出成员

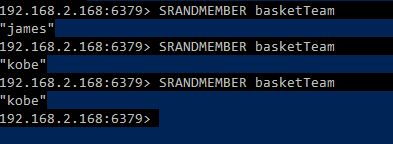
srem，随机删除该成员，同时得到成员。

随机的，因为集合是无序的。



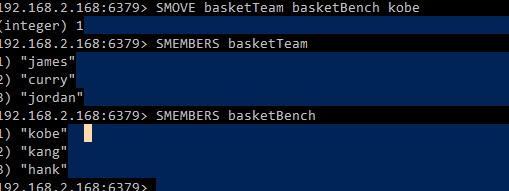
## srandmember随机选择成员

选择，并不会删除！



## smove，移动成员，由A集合移动到B集合

smove A B member

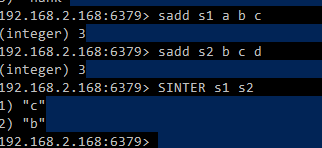


## 交集sinter，并集sunion，差集sdiff

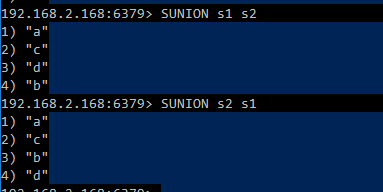
a b c

b c d

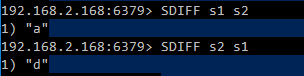
交集（intersect）就是b，c， sinter



并集（union）,就是，a b c d



差集（different），1-2 = a, 2-1 = d，从集合中去掉与另一个集合相同的元素。

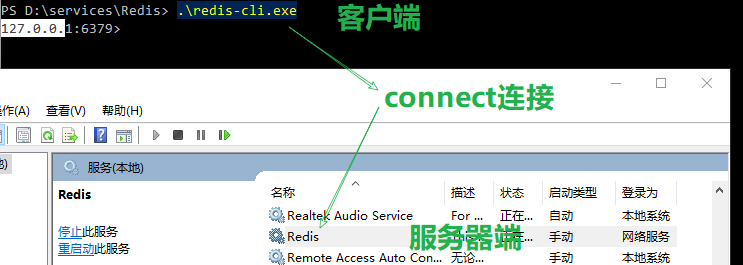


# Sorted-Set有序集合（加权（加分）集合）

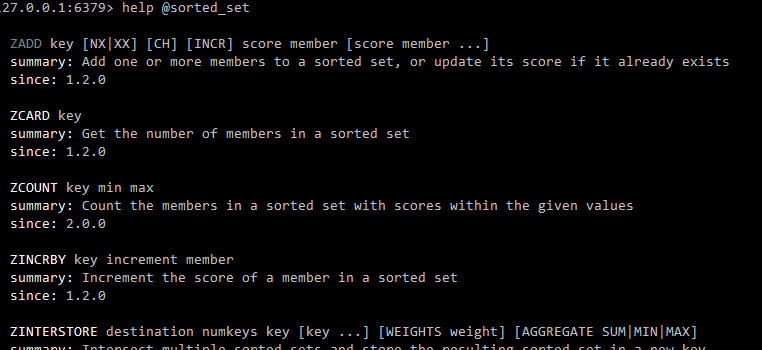
在 set的基础上，为每个成员，增加了score（分值）的属性。利用score完成对集合成员的排序。称之为有序集合。

每个成员（member）都应该有分值（score）

常见的命令如下：



## help @sorted\_set



## ZADD

添加一个成员到有序集合，需要指定成员的score：





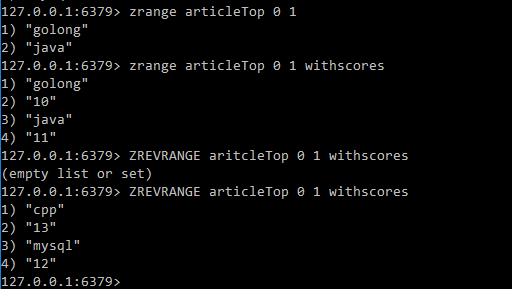
## zrange，zrevrange，依据分值升序、降序获取成员

提供索引值，strat，end

从索引值start开始，到end结束，包含end索引的元素。

默认升序，rev为降序（reverse）

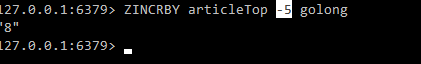
提供withscores的选项，表示获取时，同时获取元素成员的分数！



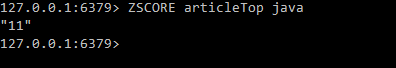
## zincreby，分值变化，累计变化

increment, 递增！

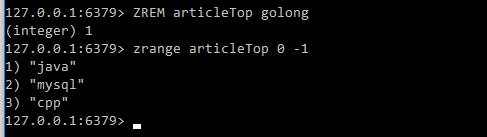




## zcore，确定成员分值

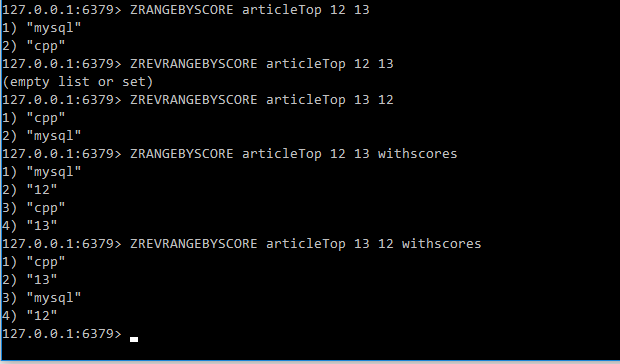


## zrem，删除集合成员

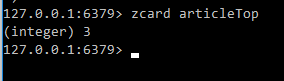


## zrangebyscore, zrevrangebyscore 依据分数获取成员

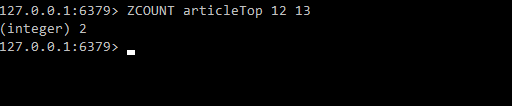
socre的闭区间（包含）



## zcard，统计集合内的元素个数

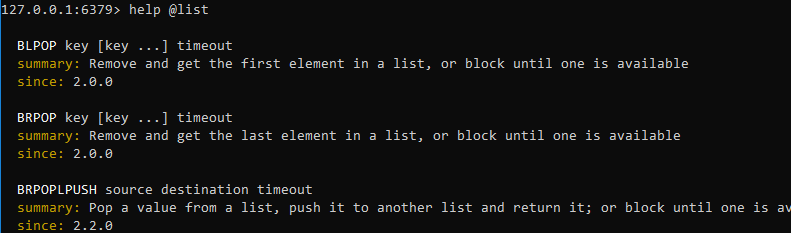


## zcout，统计满足某个分值的元素个数



# list，列表类型

列表中的元素，依据位置存储的（类似数组数据类型）。设置和访问都需要提供索引才可以。

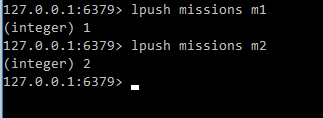


|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| m2 | m1 | 元素C | 元素D | 元素E | 元素F |
| 0 | 1 | 2 | 3 | 4 | 5 |

left right

## lpush，rpush

push：推进去。将元素，推入到某个list中。

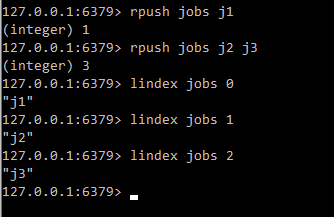


lpush，理解成left push 从左边推入，导致已有的元素向右移动一个索引。

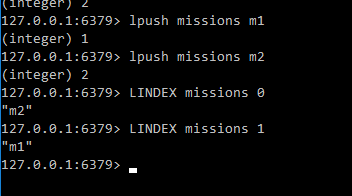
lpush m1 m2 m3 的结果就是： m3 m2 m1

rpush, 理解成 right push ,从右边推入。导致已有元素向左边移动一个索引。

rpush m1 m2 m3 的结果： m1 m2 m3



## lindex，通过索引访问元素



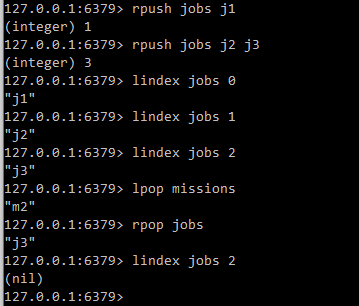
## lpop，rpop，

pop，弹出。从某一端弹出元素。

lpop从左边取出

rpop从右边取出

弹出操作会删除list中的元素，得到被弹出的元素：



## 队列，先进先出，一端进，另一端出

lpush+rpop

rpush+lpop

实现队列操作

## 栈，先进先出，一端进，同一端出

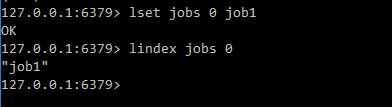
lpush+lpop

rpush+rpop

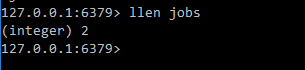
实现栈的操作。

## lset，设置某个索引元素的值

arr[3] = 10

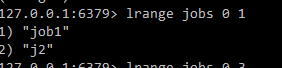


## llen,长度



## lrange，依据索引查看某个范围

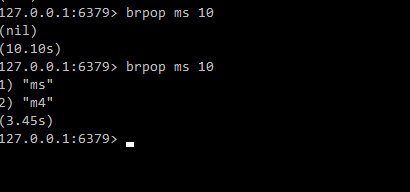
start end 闭区间索引！

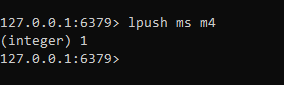


## blpop, brpop，阻塞弹出

block pop 阻塞弹出

当链表（队列）中没有办法pop到元素时（链表为空），会处于阻塞状态：





会阻塞到超时，或者pop到元素！

在循环处理任务队列时：

while true {

如果没有任务

rpop jobs，会无限次数执行pop

brpop jobs，不会执行额外的pop

}

# 任务队列、消息队列，队列，订阅发布模式

队列的应用在实际使用中，及其常见。典型场景：

1. 并发，多个任务同时发生，需要一个个的处理。一定需要队列。处理能力与业务数量不完全匹配。
2. 任务复杂需要大量的处理时间，同时需要即时反馈给用户。例如：下订单，发送邮件。

生产订单：服务器端需要，检查库存，检查价格，检查订单用户的合理性，要处理并发订单。下订单任务，服务器端不能立即就解决。通知给用户：订单在排队处理中

发送邮件：尤其是批量发送邮件，（选择了10个用户，同时发送广告邮件），完成10个邮件 的发送，也是需要大量的时间。通知给用户：邮件正在发送中

将生成订单，或发送邮件的任务，先放在队列中，将结果告知给用户增在处理。等待任务真正完成，再告知用户结果。



lpush+rpop即可完成队列的处理。

典型的模式：关注发布模式sub/pub pub/sub 模式， 来处理消息队列。

用户（客户端），选择关注某个频道（channel）。

服务商（service），向某个频道发布消息。

服务商一旦在某个频道发布了消息，关注该频道的用户（客户端）就可以接收到消息。



redis提供了一下的客户端命令：

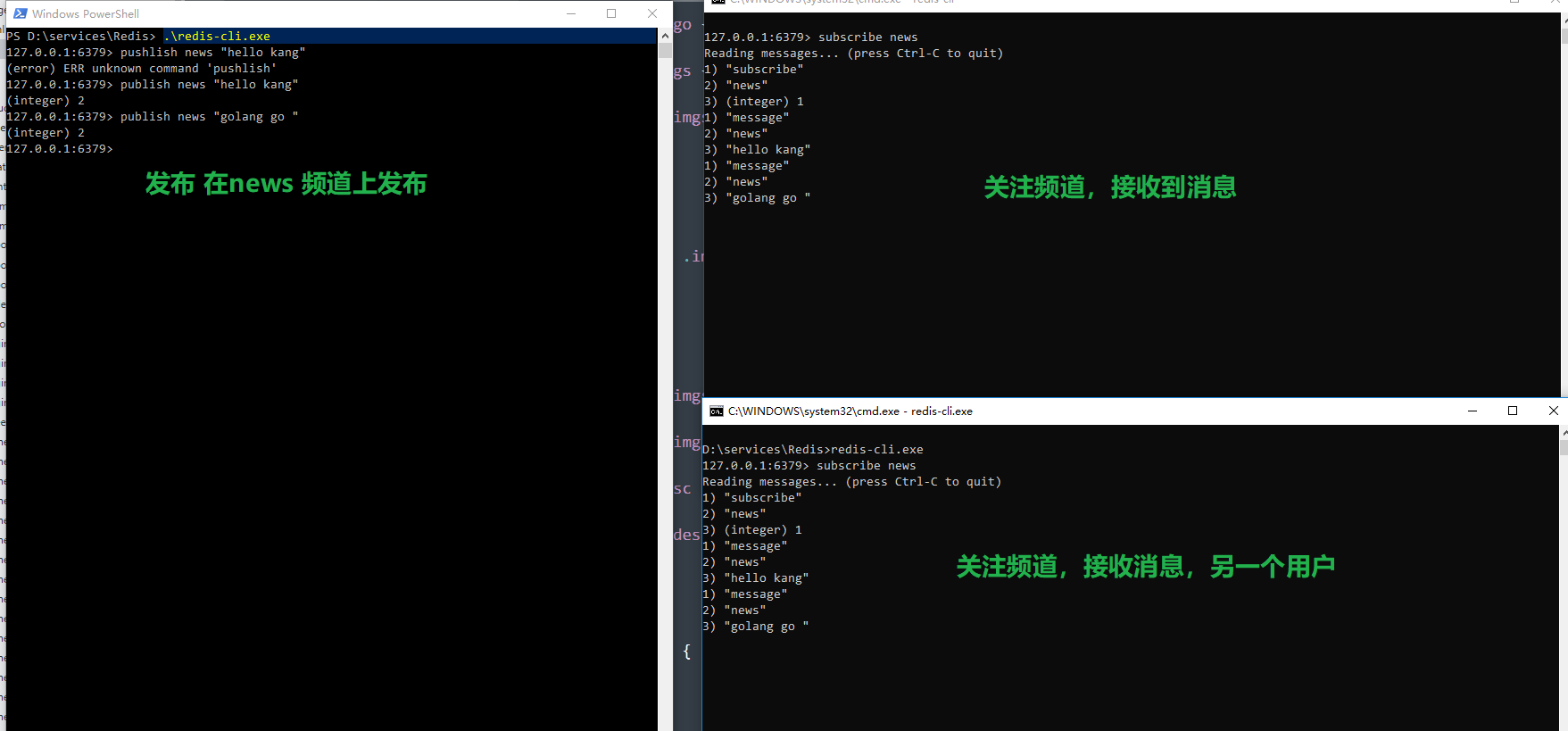


演示：

一个用于发布，多个用于关注，接收消息。

publish 频道 消息，发布

subscribe 频道，关注



左边发布，右上右下关注接收！

## go（redigo）的实现

关于订阅接收消息，redigo扩展，封装了额外的方法，不是典型的使用redis.Do()来实现了。

<https://godoc.org/github.com/gomodule/redigo/redis#hdr-Publish_and_Subscribe>

订阅的操作，由 redis.PubSubConn 该结构体实现

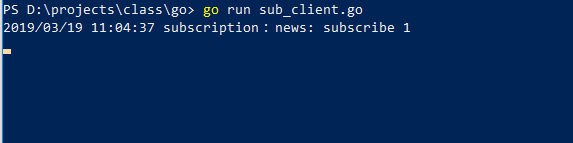
该结构体对象提供了：Subscribe()用于关注和Recive()方法用于接收。

演示代码如下：

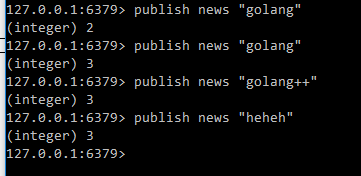
|  |
| --- |
| package main  import (  "github.com/gomodule/redigo/redis"  "log" )  func main() {  *// 1 连接redis服务器* conn, errDial := redis.Dial("tcp", "127.0.0.1:6379")  if nil != errDial {  log.Println(errDial)  return  }  *// 2（4） 关闭连接* defer conn.Close()   *// 通常的默认操作，选择数据库（0-15）  //conn.Do("SELECT", 0)   // 3 执行操作，利用PubSubConn对象，完成关注接收的处理  // 得到 PubSubConn对象* psc := redis.PubSubConn{  Conn: conn,  }  *// psc 关注(订阅）某个频道* psc.Subscribe("news")  *// 循环进程，一直关注，守护执行* for {  *//接收频道消息* msg := psc.Receive() *// 阻塞执行，（brpop），等待有消息，才处理  // 利用类型断言的switch结构，确定接收到到是什么？* switch message := msg.(type) {  case redis.Message: *// 常规消息类型* log.Printf("%s: message: %s\n", message.Channel, message.Data)  *//自定义的逻辑  //得到消息队列中的消息（任务），完成后续处理* case redis.Subscription: *// 订阅事件类型* log.Printf("subscription：%s: %s %d\n", message.Channel, message.Kind, message.Count)  case error:  log.Println("error:", message)  }   } } |

主要关注，如何接收消息！

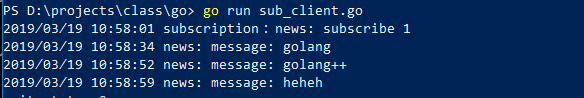
独立运行该程序：



其他进程（程序），向频道发布消息：以cmd演示：

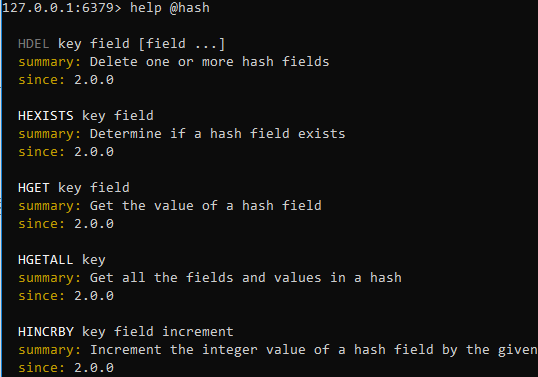


该Go的程序就可以接收到这些消息：



# hashes，哈希表

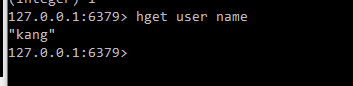
典型的map结构，key-value 集合。



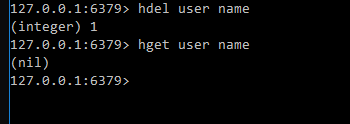
## Hset，设置某个field（字段）



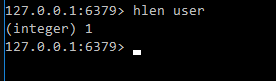
## hget，利用field获取值



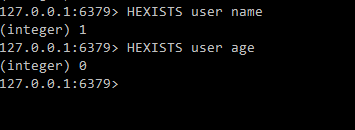
## HDEL，删除某个field



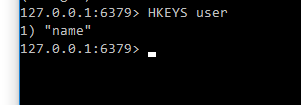
## hlen，存在个field



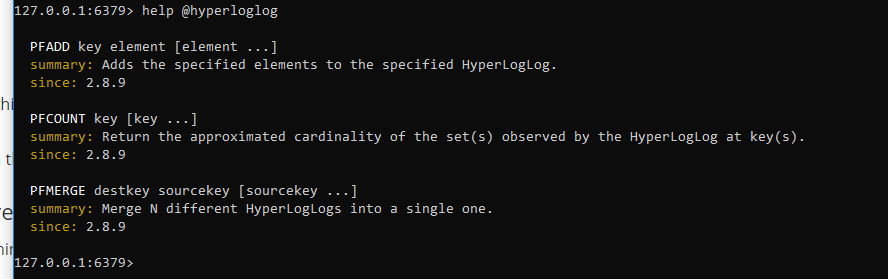
## hexists，判断某个field是否存在



## hkeys，获取全部的fields



# HyperLogLog



思考：

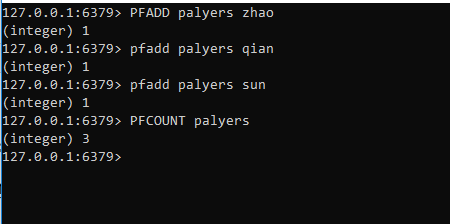
scard，统计集合set中的成员个数。

zcard，统计有序集合sorted-set中的成员个数。

hyperlog功能仅仅可以统计集合的成员个数。

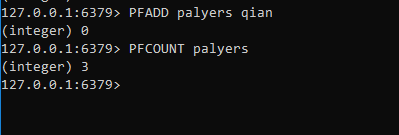
pfadd 在集合中添加成员

pfcount统计集合中的成员个数。



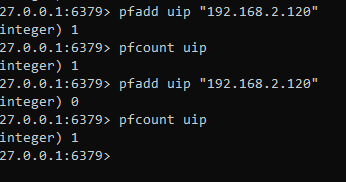
功能仅此而已。

hyperlog在添加成员时，满足集合特征的，成员唯一。



明显：当不关系成员内容，而仅仅关系成员数量时，使用该结构。结构上比set要简单的多。

有一个非常典型的应用：网站统计：独立IP数量，独立用户数量（session-id）概念。



# bitmap（bitarray），位集合

bit：比特，位，一个位2种情况。1和0.

bitarray, bit位的数组（集合）：一串010101010101010 就是bitarray

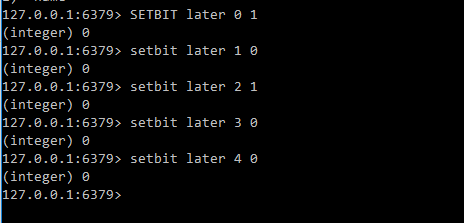
用于记录，一组开关状态，一组逻辑（真假）状态。

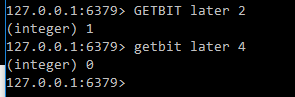
使用某一位，表示某件事的开关状态（真假状态）：

10010 记录用户一周五天是否迟到。第一位（右边，低位）表示星期一，以此类推。

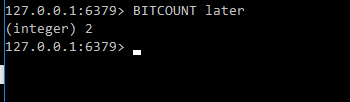
bitarray的操作，就是设置或获取某个位置上的0或1。

典型操作就是 setbit和getbit





统计：bitcount,统计1的数量。



# 类型（数据结构）总结

不同的数据结构，支持的典型操作是不同的。

string，hash：

将redis作为缓存去看待，将程序（go）处理好的串行化的数据，缓存在redis中。

string，每个key的数据间相互独立。

hash：一个hash表中的数据应该存在逻辑上的联系。

复合的（一个key可以存储多个值）类型：

list，set，sorted-set，hashes，bitarray

链式的结构：存储的元素之间存在顺序关系，选择list。（物理顺序，存储顺序）

元素可以使用某个属性进行排序，属性值变化，排序也会发生改变，选择sorted-set.(逻辑顺序)

仅仅关心一个集合，而不去考虑顺序，人人平等。选择set结构。

特殊的：简化了某种结构，对于特定现象特殊操作。

只关心集合的元素个数（不在乎内容）：hyperlog

存储的内容仅仅是 真假（开关），不存在额外信息：bitarray。

5.0 新增了stream类型。支持二进制内容的存储。（此前都是字节型字符串）。5.0之后，可以将图像，视频，声音，各种文件，都存储在redis内存中！

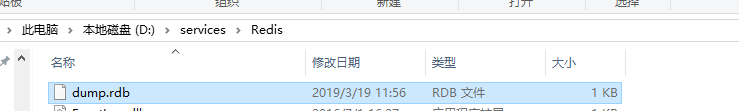
（mySQL也支持二进制数据的存储！）

# 持久化

redis数据需要同步到磁盘上，持久化存储。

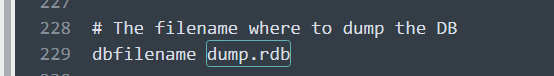
## .rdb,（redis database）

redis用于在磁盘上存储数据的文件，可以配置：



配置：

windows：redis-windows-services.conf



存储机制，使用配置项save来实现：

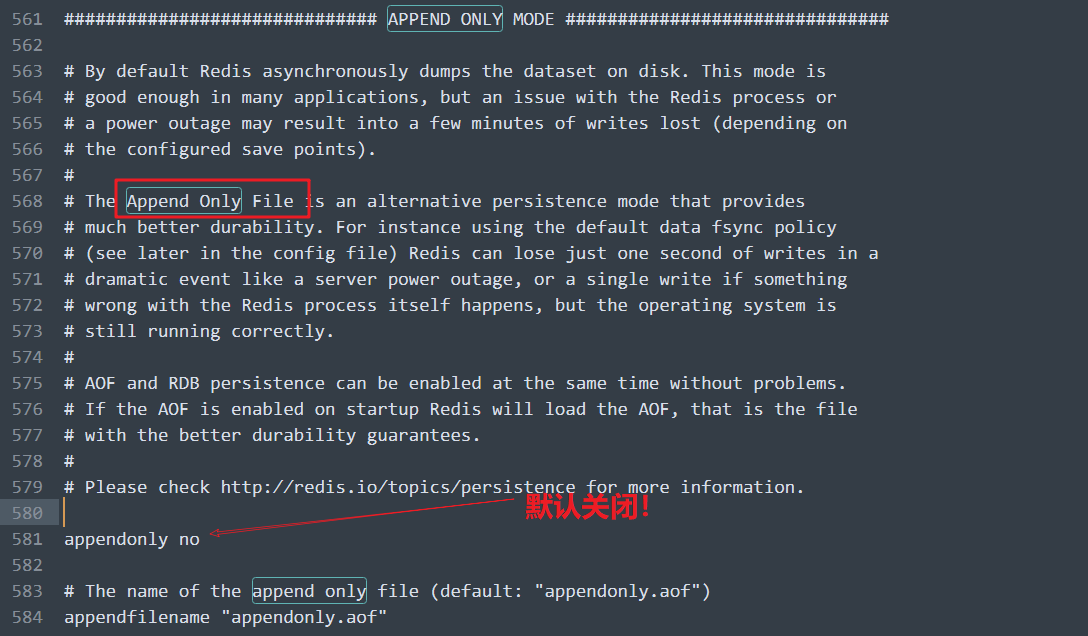


当关闭redis服务器时，会执行一次写入操作。

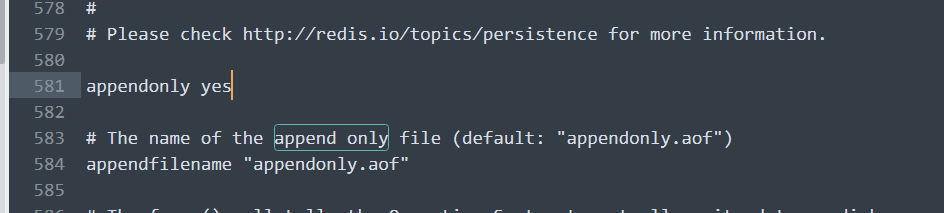
（由于是延时写入，若服务器异常关闭， 会出现数据丢失的可能性）

## AOF，append only file

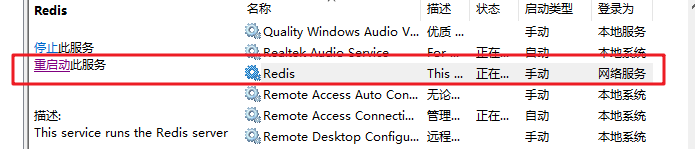
修改追加日志



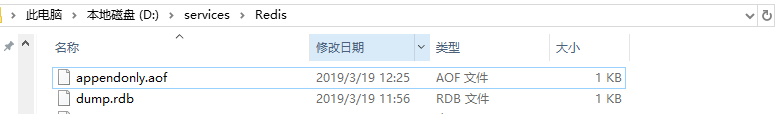
开启测试：



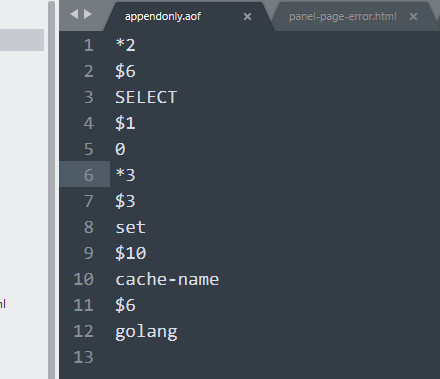
重启redis生效；



再次使用客户端连接并修改缓存项之后，在redis的服务器目录：



该文件中记录的修改：



修改的命令时即时（实时）记录的。

当数据恢复时，重做aof中的命令即可！

最大程度的保证数据不丢失。（save策略是延时的）

当服务器启动时，会使用.rdb和AOF文件来恢复数据！

当重启成功后，手动删除aof内容。

# 批处理（事务）支持

语法：

multi，启动批处理（事务）

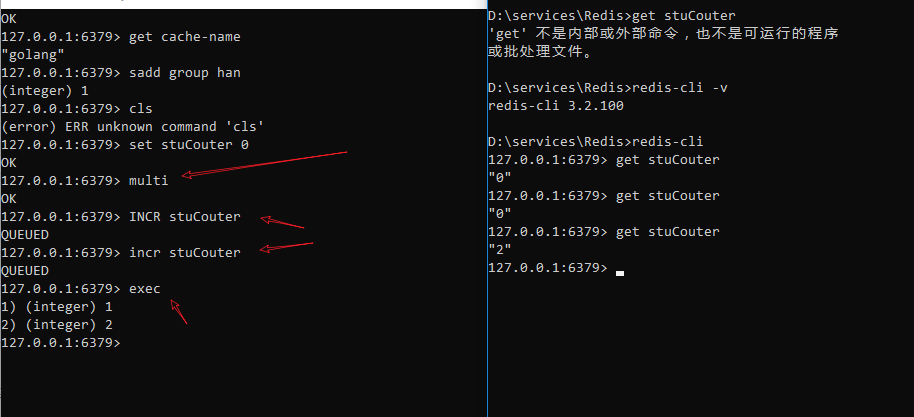
exec，执行批处理（事务）

discard，放弃批处理（事务）

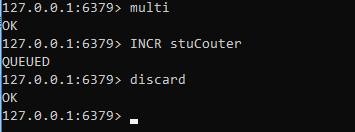
先执行multi，然后执行的语句就被放入到执行队列中。

exec，让执行队列的命令依次执行。

discard，放弃执行

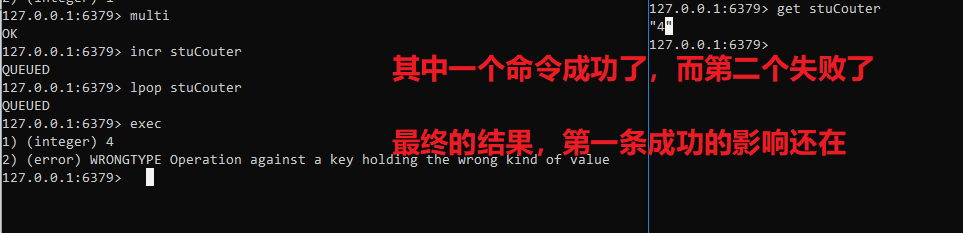


放弃演示：



当批处理执行时，若存在部分命令失败，其他成功的语句的影响其实还在：

（没有完全体现事务的ACID属性）



在redis中，multi事务，主要体现的是多条命令的一起执行。在多条命令指向期间，其他命令时插入不进来的。

当存在多个客户端，同时连接一个服务器端时。批处理的命令被视为一条整体命令，必须要执行完，才能执行其他命令。（服务器在指向客户端命令时，是原子的，是序列化的）

例如：

set stuCounter 0

multi

incr stuCounter

incr stuCounter

exec

//stuCounter 2

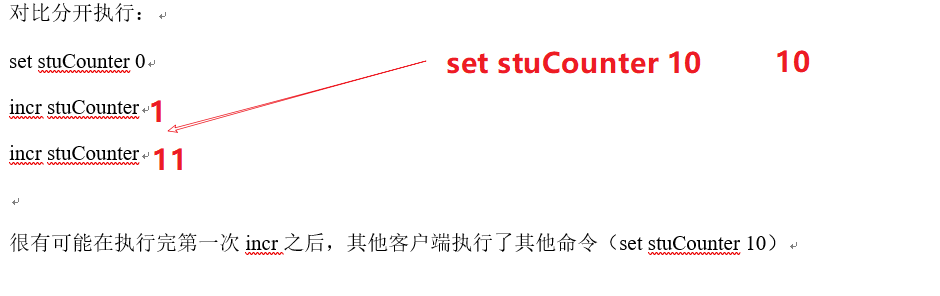
对比分开执行：

set stuCounter 0

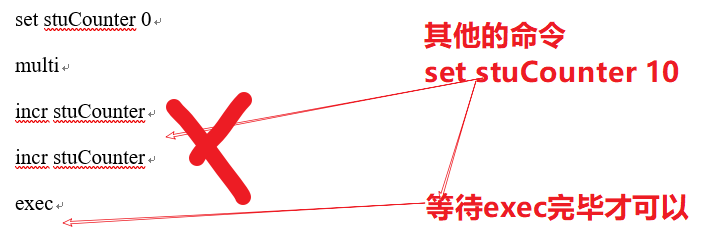
incr stuCounter

incr stuCounter

很有可能在执行完第一次incr之后，其他客户端执行了其他命令（set stuCounter 10）



上面的multi的例子：



批处理的串行化执行。

# 部署方面的支持

partition：分区：当一台服务器的内存不足以承担项目所需时。内存扩容的解决方案。多台redis共同承担一个缓存服务器的功能。字面上的意思，数据分区存储。十个数据，分在4台服务器存储：1-10（1-3，4-6，7-9， 10）

master-slave, replicate：复制，主从：当一台服务器提供的IO性能不足时，可以将数据分散到多台服务器上。此时多台服务器中的缓存数据是一致的，将对缓存的读写分散开。解决的问题时一台服务器能够同时提供的读写IO不足。十个数据，分在4台服务器存储，每台数据上的服务器一样：1-10（1-10， 1-10， 1-10， 1-10）。

模式：将某台写服务器的内容复制给多台读服务器，分散读的压力。写的压力通常要保证数据一致性，服务器少更能保证一致性

cluster：集群：主要解决的高可用的问题。避免单点失败。

# 典型应用

## 数据缓存

set ,get,delete

配合有效期去使用！

## session存储

服务器端的session适合存储在redis中

通常：使用session-id为key，session的内容为值，有效期session的垃圾回收周期。



## 队列

任务队列，消息队列

lpush + rpop

## pub/sub模式，推送，订阅（接收）

消息队列

## 集合，集合统计

独立IP，独立用户，独立会话的数量统计

使用hyperlog类型

## 分布式锁

利用setnx, 在不存在时，才能设置key的特性， setnx是单（原子）线程。

保证，分布式（多个客户端）操作资源时，某一时刻，只能有一个客户端操作。

可以在操作前使用setnx 去设置一个key，设置成功才能操作。

## 计数器

使用了 incr，incrby 这种特性。

## 排行榜

使用 有序集合中的score来完成

# 编码实现

基于beego实现。

## 初始化beego

mkdir beegoRedis

cd beegoRedis

go get -u github.com/astaxie/beego

go get -u github.com/beego/bee

创建main.go

src/app/main.go

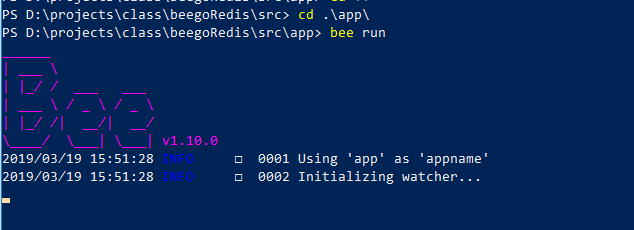
暂时使用测试代码：



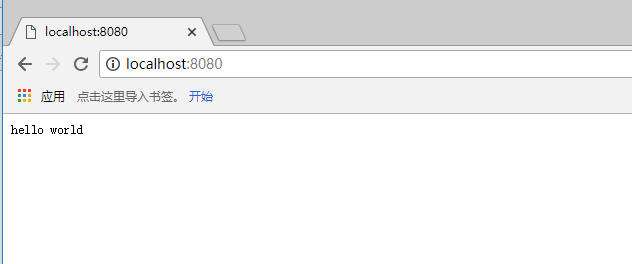
运行项目：

进入 src/app

bee run

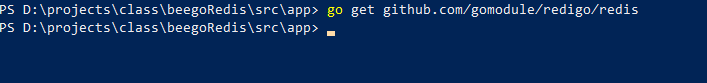


结果：



## 安装redis支持

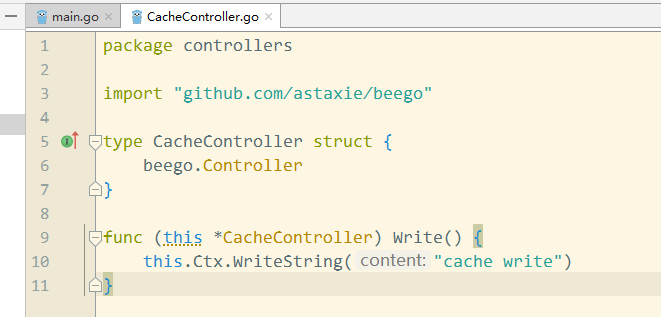
配置好 GOPATH的前提下：



## 数据缓存

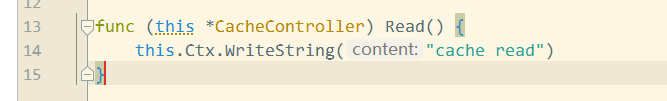
### 准备写缓存动作

app/controllers/CacheController

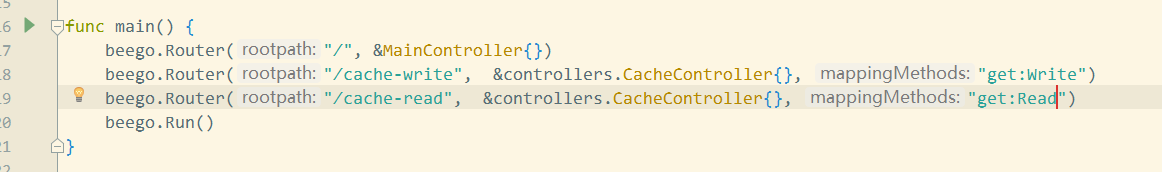


### 准备读缓存动作

app/controllers/CacheController



main.go中的路由：



### 配缓存

安装缓存模块：

go get github.com/astaxie/beego/cache

安装对应的缓存驱动：

go get github.com/astaxie/beego/cache/redis

### 使用缓存

导入抽象层，导入驱动，初始化缓存管理对象，缓存操作：



## session存储到redis中

### 保证session模块存在

go get github.com/astaxie/beego/session

### 配置session

main.go



### 写session

### 读session



## 消息队列

选择方案有2：

lpush,+brpop

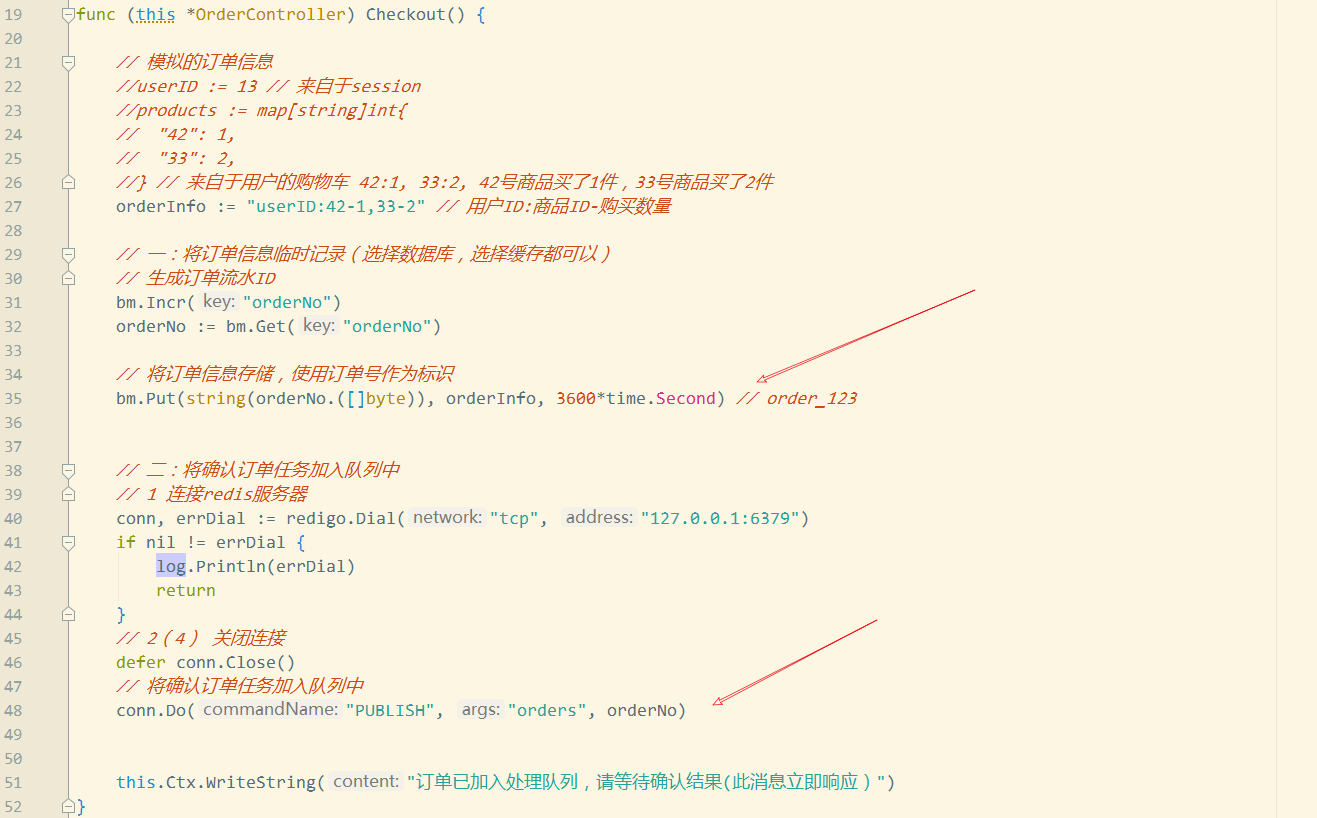
publish + describe(recive)

以用户下订单为例作为演示

### 将消息入队列（用户生成订单）

/checkout ,下单

用户选择下单，将订单信息（买了什么，买了几个，哪个用户）存储，生成（确认）订单的任务放入队列



### 从队列中获取消息

从任务队列中获取消息，检测订单是否合理，如果合理，确认订单完成：



### 通知结果