

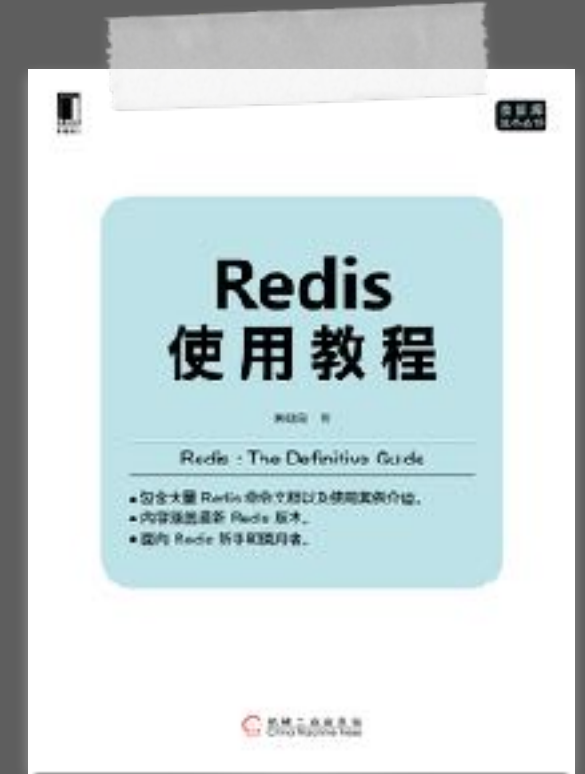
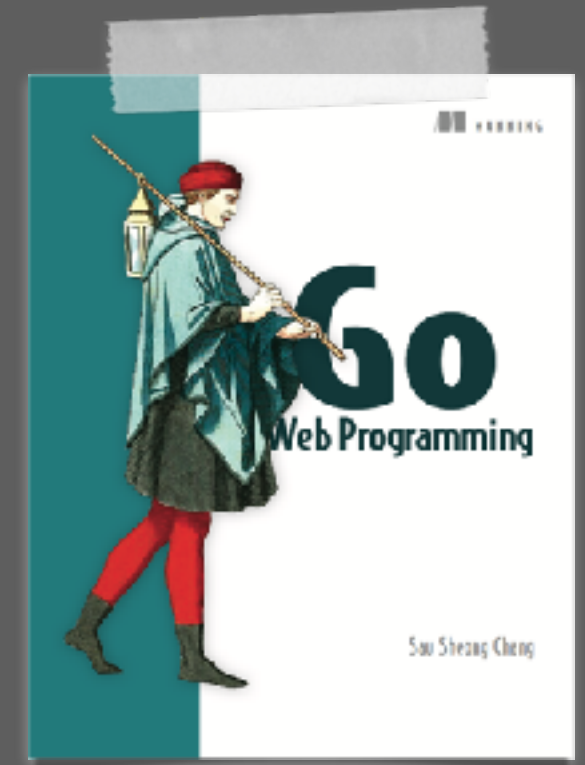
An aerial, high-angle photograph of a large audience seated in a stadium or arena. The seating is arranged in concentric, curved rows, creating a sense of depth and scale. The audience members are diverse in age and appearance, and many are looking towards the center of the arena. The lighting is bright, suggesting an indoor or well-lit outdoor venue.

使用 Go 和 Redis 构建有趣的程序

黄健宏 @ huangz.me

关于我

- 黄健宏，网名 huangz ，广东清远人。
- 计算机技术图书作者和译者，偶尔也写一点小程序自娱自乐。
- 精通 Go、Python 、Ruby 、PHP、 C 等数十种语言.....的 Hello World ！
- 著作：《Redis 设计与实现》，《Redis 使用教程》（写作中）。
- 翻译：《Go Web 编程》，《Redis 实战》。
- 开源文档：《Go 标准库中文文档》，《Redis 命令参考》，《SICP 解题集》等。
- 个人网站： huangz.me 。



路线图

路线图

一. Redis 简介

路线图

一. Redis 简介

二. 使用 Redis 构建锁

路线图

一. Redis 简介

二. 使用 Redis 构建锁

三. 使用 Redis 构建在线用户统计器

路线图

一. Redis 简介

二. 使用 Redis 构建锁

三. 使用 Redis 构建在线用户统计器

四. 使用 Redis 构建自动补完程序

Redis

an open source, in-memory data structure store

特点

特点

- 具有多种不同的数据结构可用，其中包括：字符串、散列、列表、集合、有序集合、位图(bitmap)、HyperLogLog、地理坐标(GEO)

特点

- 具有多种不同的数据结构可用，其中包括：字符串、散列、列表、集合、有序集合、位图(bitmap)、HyperLogLog、地理坐标(GEO)
- 内存存储和基于多路复用的事件响应系统，确保了命令请求的执行速度和效率

特点

- 具有多种不同的数据结构可用，其中包括：字符串、散列、列表、集合、有序集合、位图 (bitmap)、HyperLogLog、地理坐标 (GEO)
- 内存存储和基于多路复用的事件响应系统，确保了命令请求的执行速度和效率
- 丰富的附加功能：事务、Lua 脚本、键过期机制、键淘汰机制、多种持久化方式 (AOF、RDB、RDB+AOF 混合)

特点

- 具有多种不同的数据结构可用，其中包括：字符串、散列、列表、集合、有序集合、位图 (bitmap)、HyperLogLog、地理坐标 (GEO)
- 内存存储和基于多路复用的事件响应系统，确保了命令请求的执行速度和效率
- 丰富的附加功能：事务、Lua 脚本、键过期机制、键淘汰机制、多种持久化方式 (AOF、RDB、RDB+AOF 混合)
- 强大的多机功能支持：主从复制（单主多从）、Sentinel（高可用）、集群（基于 Raft 算法，多主多从，内建高可用）

特点

- 具有多种不同的数据结构可用，其中包括：字符串、散列、列表、集合、有序集合、位图 (bitmap)、HyperLogLog、地理坐标 (GEO)
- 内存存储和基于多路复用的事件响应系统，确保了命令请求的执行速度和效率
- 丰富的附加功能：事务、Lua 脚本、键过期机制、键淘汰机制、多种持久化方式 (AOF、RDB、RDB+AOF 混合)
- 强大的多机功能支持：主从复制（单主多从）、Sentinel（高可用）、集群（基于 Raft 算法，多主多从，内建高可用）
- 拥有无限可能性的扩展模块系统：神经网络、全文搜索、JSON 数据结构等等。

数据结构

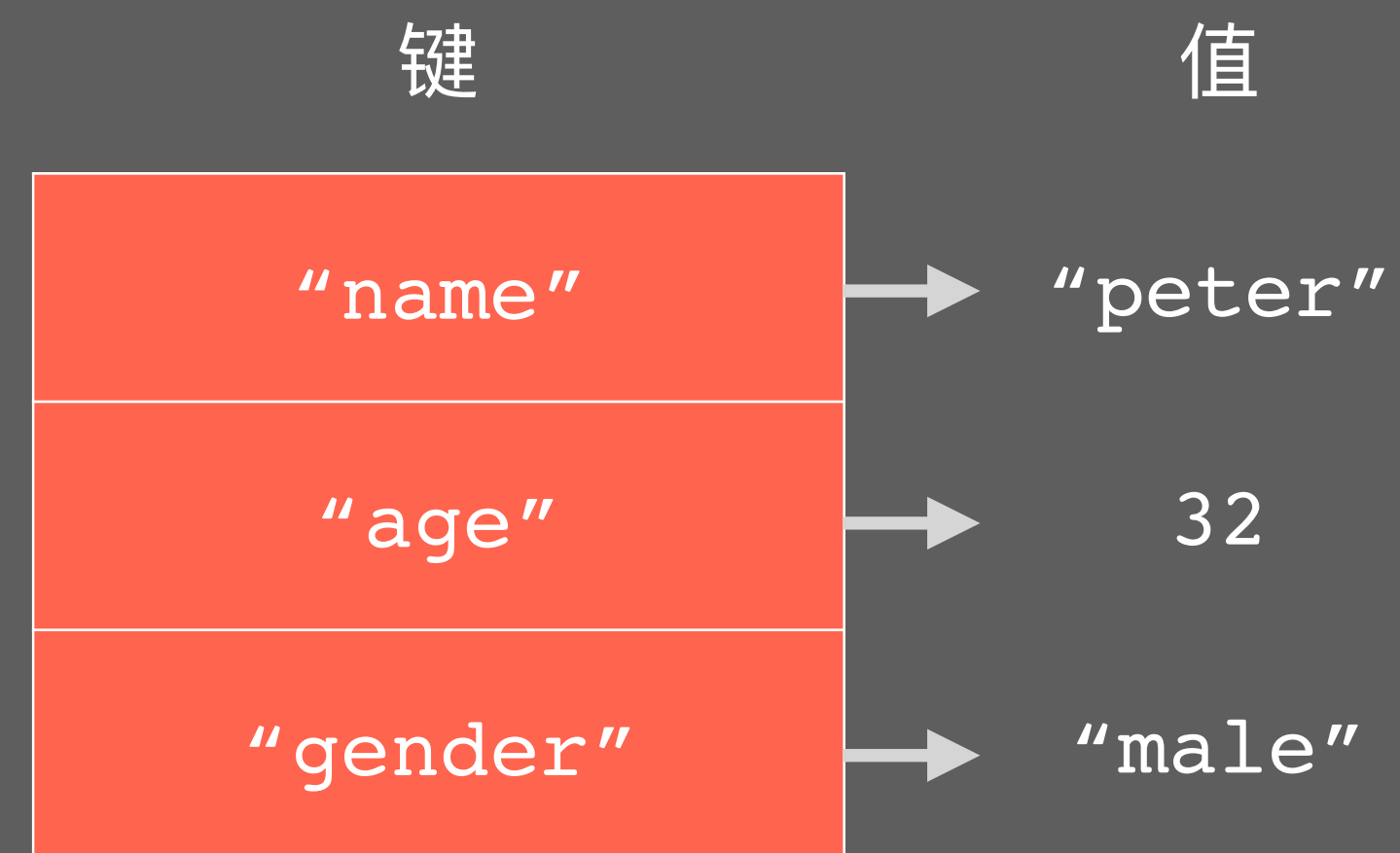
data structures

字符串

索引	0	1	2	3	4	5	6	7	8	9	10	11
字符	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'\0'

带索引、带长度记录、二进制安全、带有内存预分配策略的字符串实现

散列



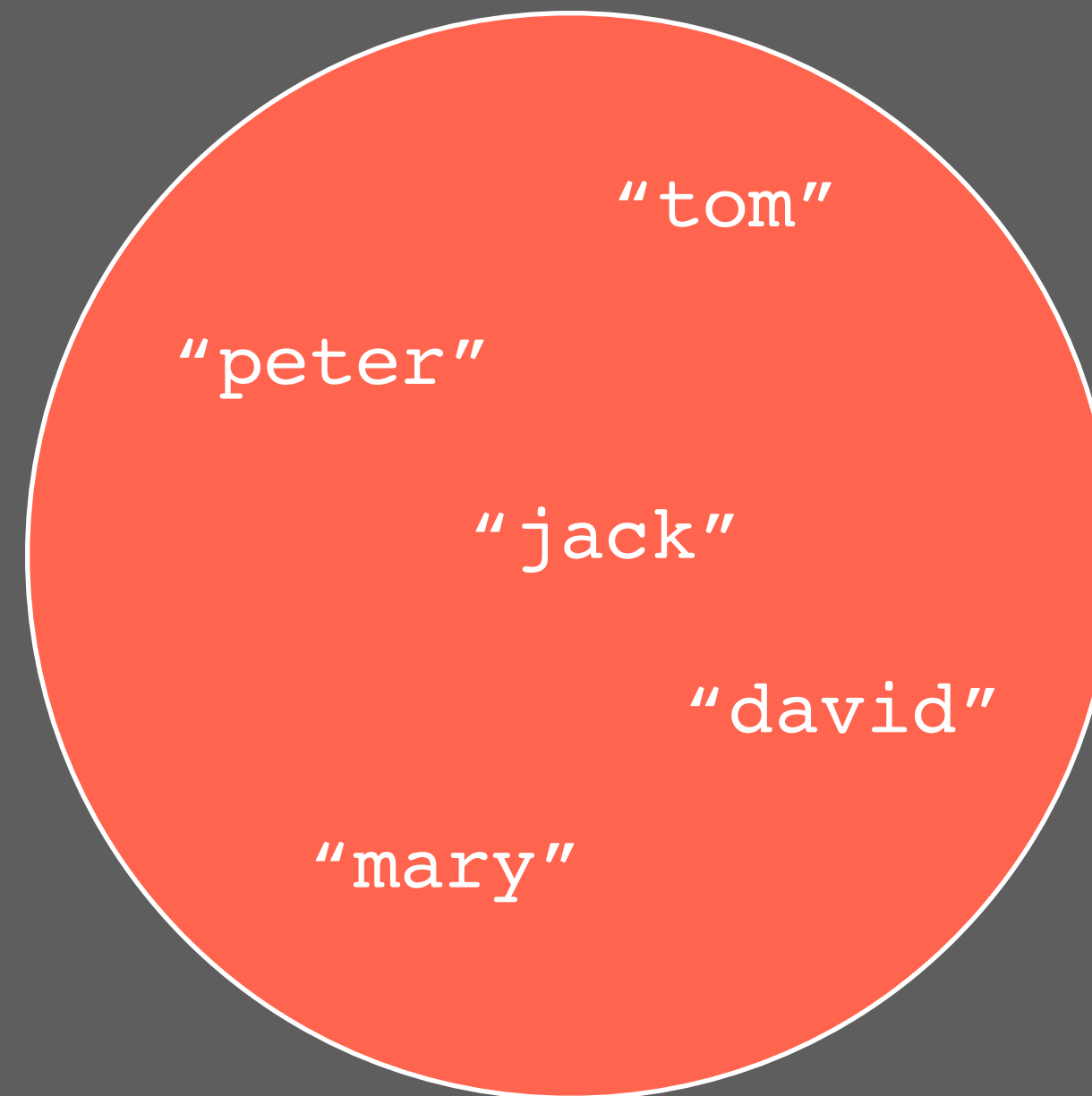
使用哈希表储存键值对、每个键都各不相同、获取单个键值对的复杂度为常数

列表

索引	0	1	2	3
项	"job::6379"	"msg::10086"	"request::256"	"user::peter"

使用双端链表实现的有序序列、针对两端的操作为常数复杂度、其他列表操作多为线性复杂度、允许重复元素

集合



以无序方式储存任意多个各不相同的元素、针对单个元素的操作都为常数复杂度、可执行并集交集等常见的集合计算

有序集合

分值	成员
1.5	"banana"
2.5	"cherry"
3.7	"apple"
8.3	"durian"

各不相同的多个成员按分值大小进行排列、可以按分值顺序或者成员顺序执行多项有序操作、使用 Skip List 实现

位图 (bitmap)

索引	0	1	2	3	4	5	6	7
位	0	1	1	0	1	1	1	0

由一连串二进制位组成、可单独设置指定的位、可获取指定范围的多个位又或者对它们进行计数

HyperLogLog



基于概率算法实现、可以计算出给定集合的近似基数、只使用固定大小的内存

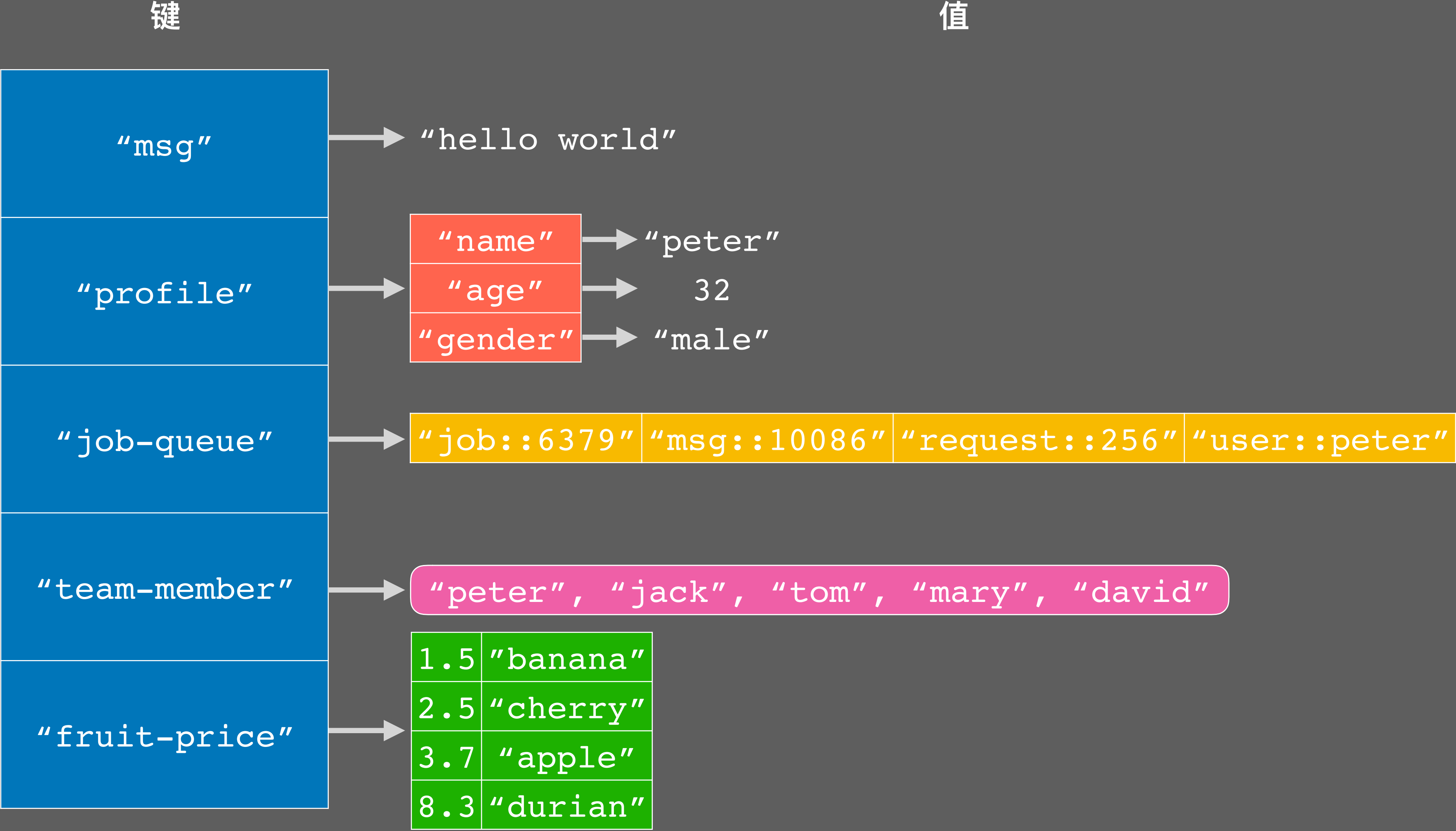
地理位置 (GEO)

113.2099647, 23.593675
“Qingyuan”



113.106308, 23.0088312
“Guangzhou”

储存经纬度坐标、可以进行范围计算、或者计算两地间的距离



命令请求

COMMAND <key> <arg1> <arg2> <arg3> ... OPTION1 <value1> <value2> ...

命令

键

参数

选项

选项的值










命令请求

COMMAND <key> <arg1> <arg2> <arg3> ... OPTION1 <value1> <value2> ...










命令 键 参数 选项 选项的值

PING
SET msg "hello world"
SORT fruit ALPHA GET *-price
HMSET profile "name" "peter" "age" 32 "gender" "male"
RPUSH "job-queue" "job::6379" "msg::10086" "request::256"
...

获取客户端

Radix	 	 MIT licensed Redis client which supports pipelining, pooling, redis cluster, scripting, pub/sub, scanning, and more.	 
Redigo	 	 Redigo is a Go client for the Redis database with support for Print-alike API, Pipelining (including transactions), Pub/Sub, Connection pooling, scripting.	

获取客户端

Radix	 	 MIT licensed Redis client which supports pipelining, pooling, redis cluster, scripting, pub/sub, scanning, and more.	 
Redigo	 	 Redigo is a Go client for the Redis database with support for Print-alike API, Pipelining (including transactions), Pub/Sub, Connection pooling, scripting.	

```
go get github.com/mediocregopher/radix.v2
```

连接客户端

```
package main

import (
    "fmt"
    "github.com/mediocregopher/radix.v2/redis"
)

func main() {
    client, _ := redis.Dial("tcp", "localhost:6379")
    defer client.Close()

    repl := client.Cmd("PING")
    content, _ := repl.Str()

    fmt.Println(content) // "PONG"
}
```

连接客户端

```
package main

import (
    "fmt"
    "github.com/mediocregopher/radix.v2/redis"
)

func main() {
    client, _ := redis.Dial("tcp", "localhost:6379")
    defer client.Close()

    repl := client.Cmd("PING")
    content, _ := repl.Str()

    fmt.Println(content) // "PONG"
}
```

连接服务器

连接客户端

```
package main

import (
    "fmt"
    "github.com/mediocregopher/radix.v2/redis"
)

func main() {
    client, _ := redis.Dial("tcp", "localhost:6379")
    defer client.Close()

    repl := client.Cmd("PING")
    content, _ := repl.Str()

    fmt.Println(content) // "PONG"
}
```

连接服务器

执行命令并获取回复

连接客户端

```
package main

import (
    "fmt"
    "github.com/mediocregopher/radix.v2/redis"
)

func main() {
    client, _ := redis.Dial("tcp", "localhost:6379")
    defer client.Close()

    repl := client.Cmd("PING")
    content, _ := repl.Str()

    fmt.Println(content) // "PONG"
}
```

连接服务器

执行命令并获取回复

将回复转换为字符串

锁

lock

锁

锁是一种同步机制，它可以保证一项资源在任何时候只能被一个进程使用，如果有其他进程想要使用相同的资源，那么它们就必须等待，直到正在使用资源的进程放弃使用权为止。

锁

锁是一种同步机制，它可以保证一项资源在任何时候只能被一个进程使用，如果有其他进程想要使用相同的资源，那么它们就必须等待，直到正在使用资源的进程放弃使用权为止。

一个锁实现通常会有获取（acquire）和释放（release）这两种操作：

锁

锁是一种同步机制，它可以保证一项资源在任何时候只能被一个进程使用，如果有其他进程想要使用相同的资源，那么它们就必须等待，直到正在使用资源的进程放弃使用权为止。

一个锁实现通常会有获取（acquire）和释放（release）这两种操作：

- 获取操作用于取得资源的独占使用权。在任何时候，最多只能有一个进程取得锁，我们把成功取得锁的进程称之为锁的持有者。在锁已经被持有的情况下，所有尝试再次获取锁的操作都会失败。

锁

锁是一种同步机制，它可以保证一项资源在任何时候只能被一个进程使用，如果有其他进程想要使用相同的资源，那么它们就必须等待，直到正在使用资源的进程放弃使用权为止。

一个锁实现通常会有获取（acquire）和释放（release）这两种操作：

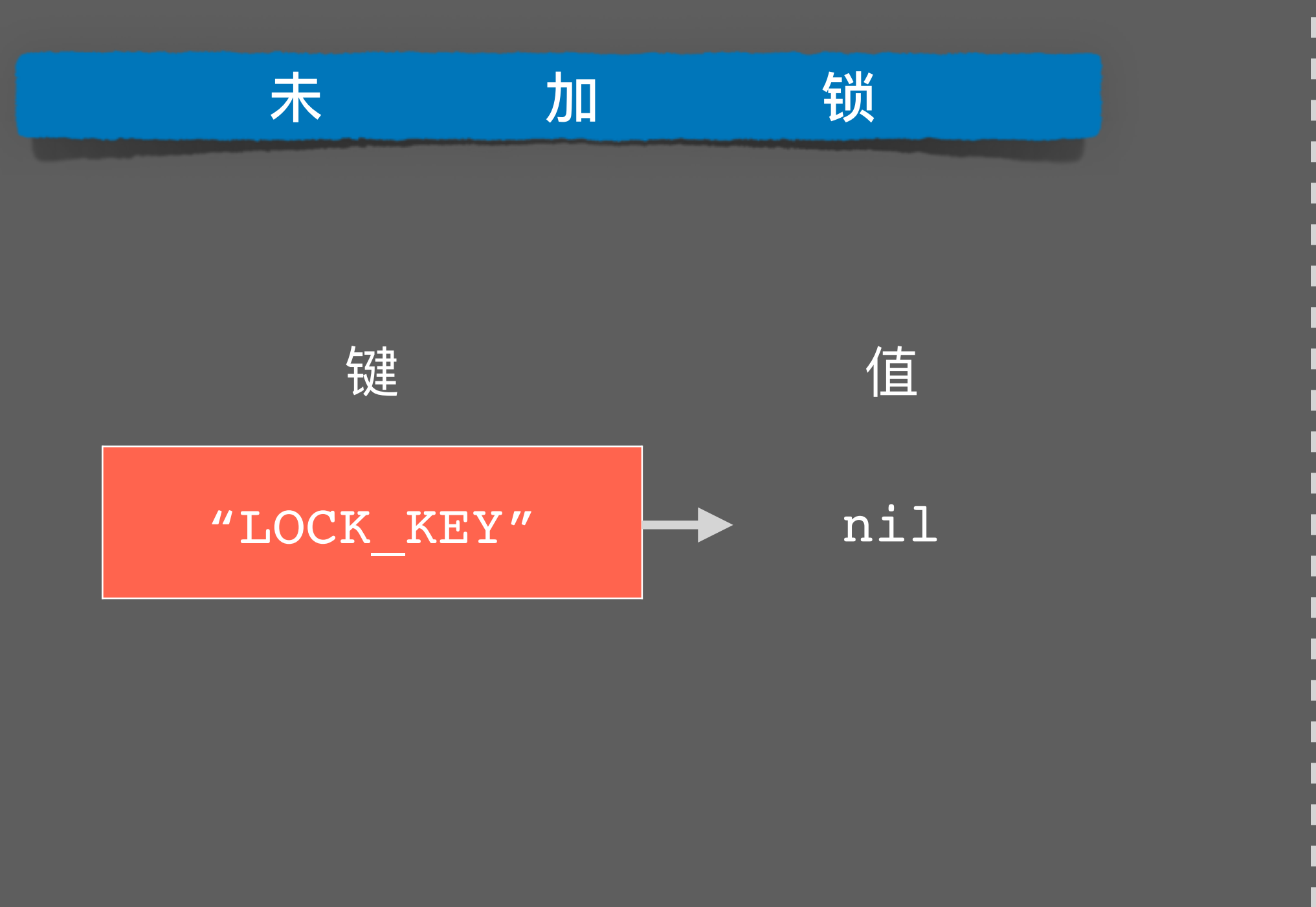
- 获取操作用于取得资源的独占使用权。在任何时候，最多只能有一个进程取得锁，我们把成功取得锁的进程称之为锁的持有者。在锁已经被持有的情况下，所有尝试再次获取锁的操作都会失败。
- 释放操作用于放弃资源的独占使用权，一般由锁的持有者调用。在锁被释放之后，其他进程就可以尝试再次获取这个锁了。

方法一 —— 使用字符串

将一个字符串键用作锁，如果这个键有值，那么说明锁已被获取；反之，如果键没有值，那么说明锁未被获取，程序可以通过为其设置值来获取锁。

方法一 —— 使用字符串

将一个字符串键用作锁，如果这个键有值，那么说明锁已被获取；反之，如果键没有值，那么说明锁未被获取，程序可以通过为其设置值来获取锁。



方法一 —— 使用字符串

将一个字符串键用作锁，如果这个键有值，那么说明锁已被获取；反之，如果键没有值，那么说明锁未被获取，程序可以通过为其设置值来获取锁。

未 加 锁



已 加 锁



需要用到的命令

需要用到的命令

GET *key*

获取字符串键 *key* 的值，如果该键尚未有值，那么返回一个空值 (*nil*)

需要用到的命令

GET *key*

获取字符串键 *key* 的值，如果该键尚未有值，那么返回一个空值 (*nil*)

SET *key value*

将字符串键 *key* 的值设置为 *value*，如果键已经有值，那么默认使用新值去覆盖旧值

需要用到的命令

GET *key*

获取字符串键 *key* 的值，如果该键尚未有值，那么返回一个空值 (*nil*)

SET *key value*

将字符串键 *key* 的值设置为 *value*，如果键已经有值，那么默认使用新值去覆盖旧值

DEL *key*

删除给定的键 *key*

实现代码

```
const lock_key = "LOCK_KEY"
const lock_value = "LOCK_VALUE"

func acquire(client *redis.Client) bool {
    current_value, _ := client.Cmd("GET", lock_key).Str()
    if current_value == "" {
        client.Cmd("SET", lock_key, lock_value)
        return true
    } else {
        return false
    }
}

func release(client *redis.Client) {
    client.Cmd("DEL", lock_key)
}
```

实现代码

```
const lock_key = "LOCK_KEY"
const lock_value = "LOCK_VALUE"

func acquire(client *redis.Client) bool {
    current_value, _ := client.Cmd("GET", lock_key).Str()
    if current_value == "" {
        client.Cmd("SET", lock_key, lock_value)
        return true
    } else {
        return false
    }
}

func release(client *redis.Client) {
    client.Cmd("DEL", lock_key)
}
```

获取键的值

实现代码

```
const lock_key = "LOCK_KEY"  
const lock_value = "LOCK_VALUE"
```

```
func acquire(client *redis.Client) bool {  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("SET", lock_key, lock_value)  
        return true  
    } else {  
        return false  
    }  
}
```

获取键的值

为键设置值

```
func release(client *redis.Client) {  
    client.Cmd("DEL", lock_key)  
}
```

实现代码

```
const lock_key = "LOCK_KEY"  
const lock_value = "LOCK_VALUE"
```

```
func acquire(client *redis.Client) bool {  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("SET", lock_key, lock_value)  
        return true  
    } else {  
        return false  
    }  
}
```

获取键的值

为键设置值

```
func release(client *redis.Client) {  
    client.Cmd("DEL", lock_key)  
}
```

删除键

竞争条件

```
const lock_key = "LOCK_KEY"  
const lock_value = "LOCK_VALUE"
```

```
func acquire(client *redis.Client) bool {  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("SET", lock_key, lock_value)  
        return true  
    } else {  
        return false  
    }  
}
```

```
func release(client *redis.Client) {  
    client.Cmd("DEL", lock_key)  
}
```

因为 GET 和 SET 在两个不同的请求中执行，在它们之间可能有其他请求已经改变了锁键的值，导致“锁键为空”这一判断不再为真，从而引发多个客户端之间的竞争条件。

客户端一

开始

客户端二

开始

客户端一

开始



GET LOCK_KEY

客户端二

开始



GET LOCK_KEY

客户端一

开始

GET LOCK_KEY

客户端二

开始

GET LOCK_KEY

SET LOCK_KEY

客户端一

开始

GET LOCK_KEY

客户端二

开始

GET LOCK_KEY

SET LOCK_KEY

成功获取锁

客户端一

开始

GET LOCK_KEY



客户端二

开始

GET LOCK_KEY

SET LOCK_KEY

成功获取锁



客户端一

开始

GET LOCK_KEY

SET LOCK_KEY

不知道锁键已被
修改，继续执行
SET 命令。

客户端二

开始

GET LOCK_KEY

SET LOCK_KEY

成功获取锁

客户端一

开始

GET LOCK_KEY

SET LOCK_KEY

成功获取锁

不知道锁键已被
修改，继续执行
SET 命令。

同时存在两个锁，出错！

客户端二

开始

GET LOCK_KEY

SET LOCK_KEY

成功获取锁

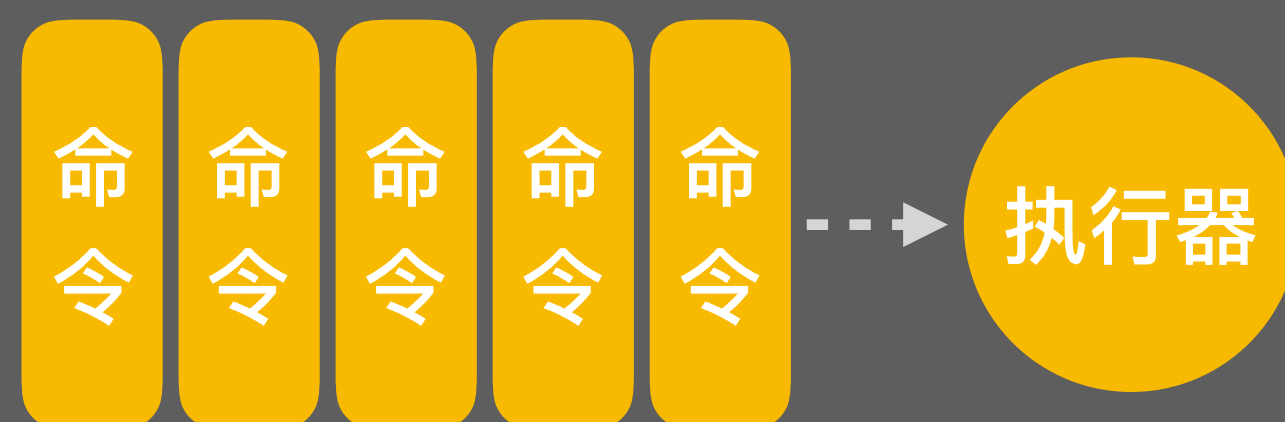
方法二 —— 使用事务保证安全性

锁的基本实现方法跟之前一样，但使用 Redis 的事务特性保证操作的安全性。

方法二 —— 使用事务保证安全性

锁的基本实现方法跟之前一样，但使用 Redis 的事务特性保证操作的安全性。

非 事 务 命 令



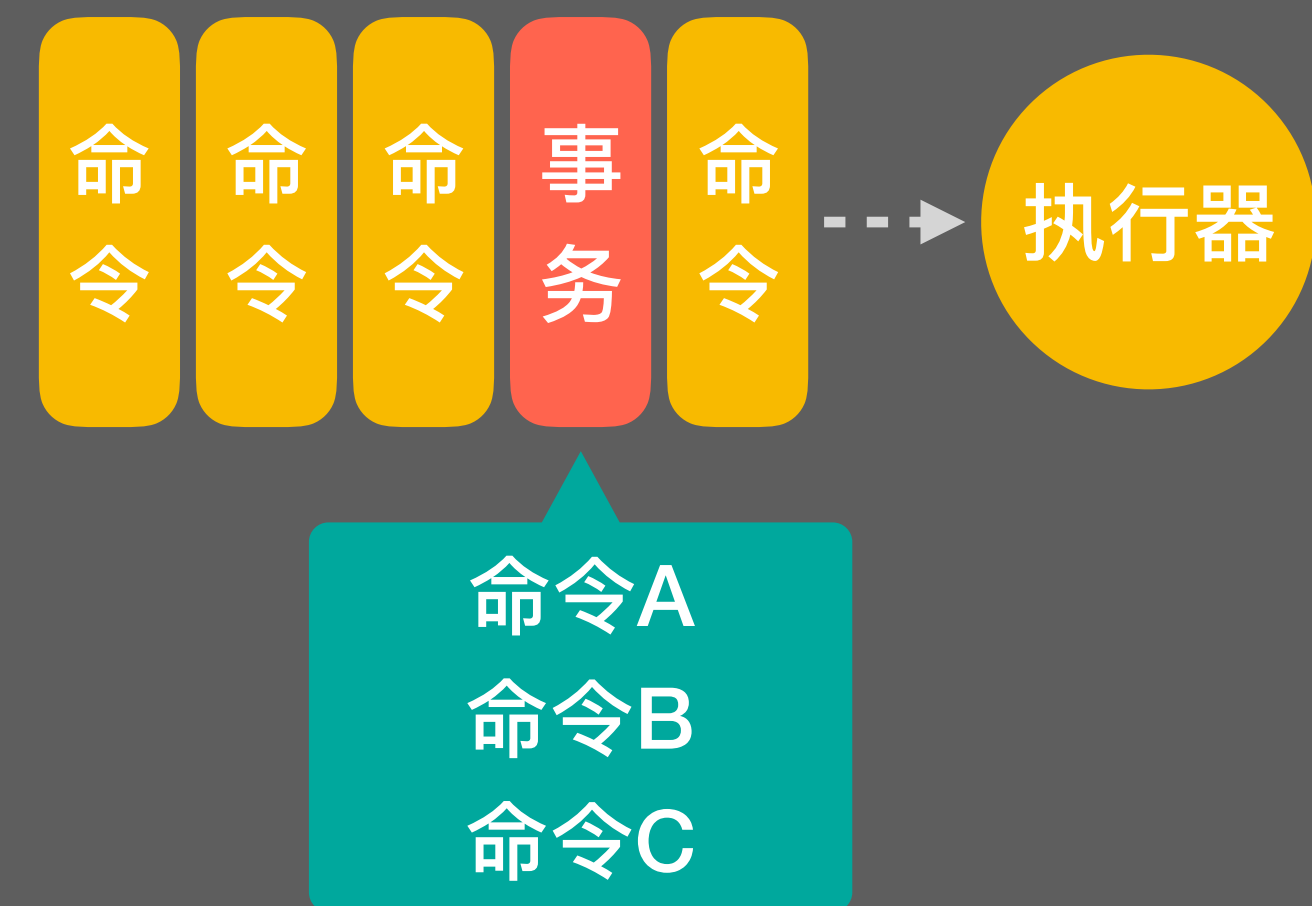
方法二 —— 使用事务保证安全性

锁的基本实现方法跟之前一样，但使用 Redis 的事务特性保证操作的安全性。

非 事 务 命 令



事 务 命 令



需要用到的命令

需要用到的命令

WATCH key [key ...]

监视给定的键，如果这些键在事务执行之前已经被修改，那么拒绝执行事务

需要用到的命令

WATCH key [key ...]

监视给定的键，如果这些键在事务执行之前已经被修改，那么拒绝执行事务

MULTI

开启一个事务，之后客户端发送的所有操作命令都会被放入到事务队列里面

需要用到的命令

WATCH key [key ...]

监视给定的键，如果这些键在事务执行之前已经被修改，那么拒绝执行事务

MULTI

开启一个事务，之后客户端发送的所有操作命令都会被放入到事务队列里面

EXEC

尝试执行事务，成功时将返回一个由多个命令回复组成的队列，失败则返回 `nil`

实现代码

```
func acquire(client *redis.Client) bool {  
    client.Cmd("WATCH", lock_key)  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("MULTI")  
        client.Cmd("SET", lock_key, lock_value)  
        repl, _ := client.Cmd("EXEC").List()  
        if repl != nil {  
            return true  
        }  
    }  
    return false  
}
```

实现代码

监视键

```
func acquire(client *redis.Client) bool {
    client.Cmd("WATCH", lock_key)
    current_value, _ := client.Cmd("GET", lock_key).Str()
    if current_value == "" {
        client.Cmd("MULTI")
        client.Cmd("SET", lock_key, lock_value)
        repl, _ := client.Cmd("EXEC").List()
        if repl != nil {
            return true
        }
    }
    return false
}
```

实现代码

```
func acquire(client *redis.Client) bool {  
    client.Cmd("WATCH", lock_key)  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("MULTI")  
        client.Cmd("SET", lock_key, lock_value)  
        repl, _ := client.Cmd("EXEC").List()  
        if repl != nil {  
            return true  
        }  
    }  
    return false  
}
```

监视键

开启事务

实现代码

```
func acquire(client *redis.Client) bool {  
    client.Cmd("WATCH", lock_key)  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("MULTI")  
        client.Cmd("SET", lock_key, lock_value)  
        repl, _ := client.Cmd("EXEC").List()  
        if repl != nil {  
            return true  
        }  
    }  
    return false  
}
```

监视键

开启事务

放入事务队列

实现代码

```
func acquire(client *redis.Client) bool {  
    client.Cmd("WATCH", lock_key)  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("MULTI")  
        client.Cmd("SET", lock_key, lock_value)  
        repl, _ := client.Cmd("EXEC").List()  
        if repl != nil {  
            return true  
        }  
    }  
    return false  
}
```

监视键

开启事务

放入事务队列

尝试执行事务

实现代码

```
func acquire(client *redis.Client) bool {  
    client.Cmd("WATCH", lock_key)  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("MULTI")  
        client.Cmd("SET", lock_key, lock_value)  
        repl, _ := client.Cmd("EXEC").List()  
        if repl != nil {  
            return true  
        }  
    }  
    return false  
}
```

监视键

开启事务

放入事务队列

尝试执行事务

根据事务是否执行成功
来判断加锁是否成功

客户端一

开始

客户端二

开始

客户端一

开始

WATCH LOCK_KEY

GET LOCK_KEY

客户端二

开始

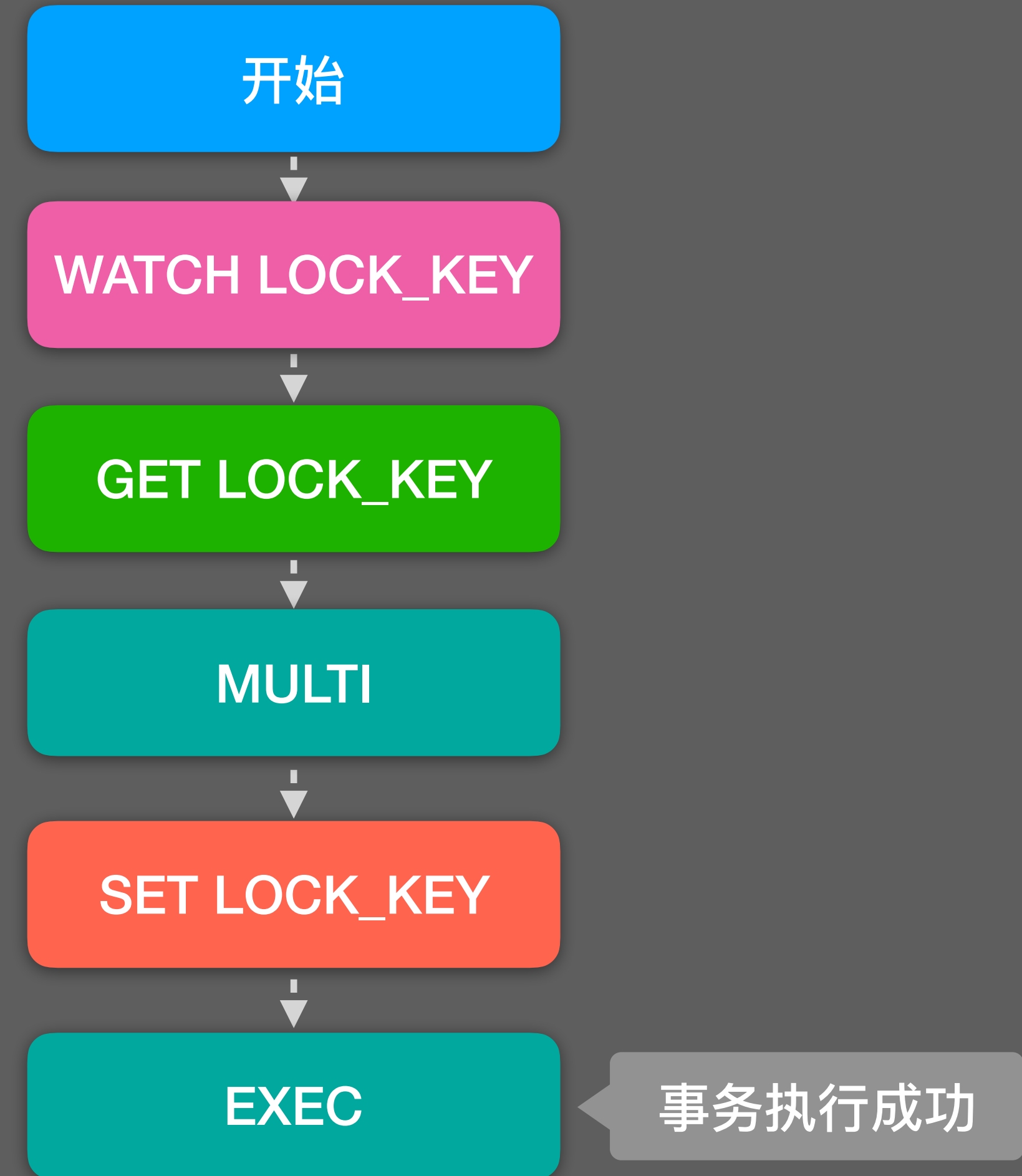
WATCH LOCK_KEY

GET LOCK_KEY

客户端一



客户端二



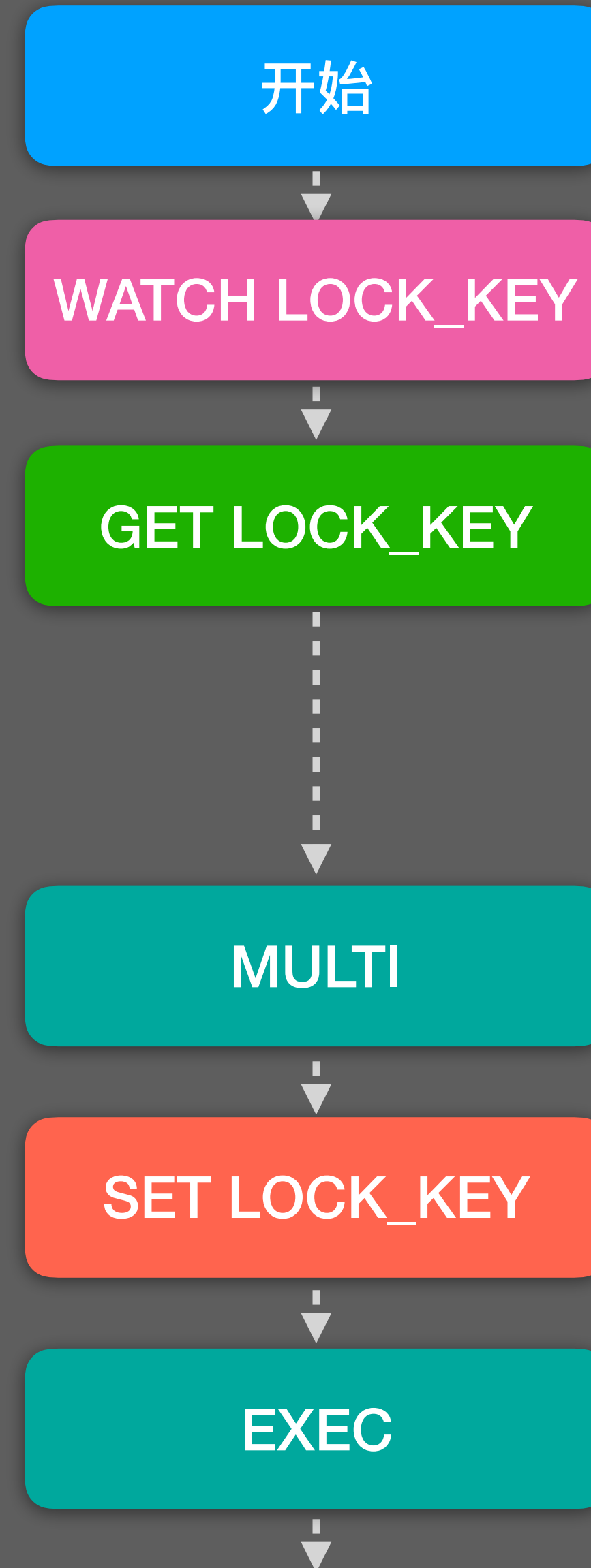
客户端一



客户端二



客户端一



客户端二



客户端一

开始

WATCH LOCK_KEY

GET LOCK_KEY

MULTI

SET LOCK_KEY

EXEC

因为 WATCH 命令的作用，该事务将被拒绝执行。

客户端二

开始

WATCH LOCK_KEY

GET LOCK_KEY

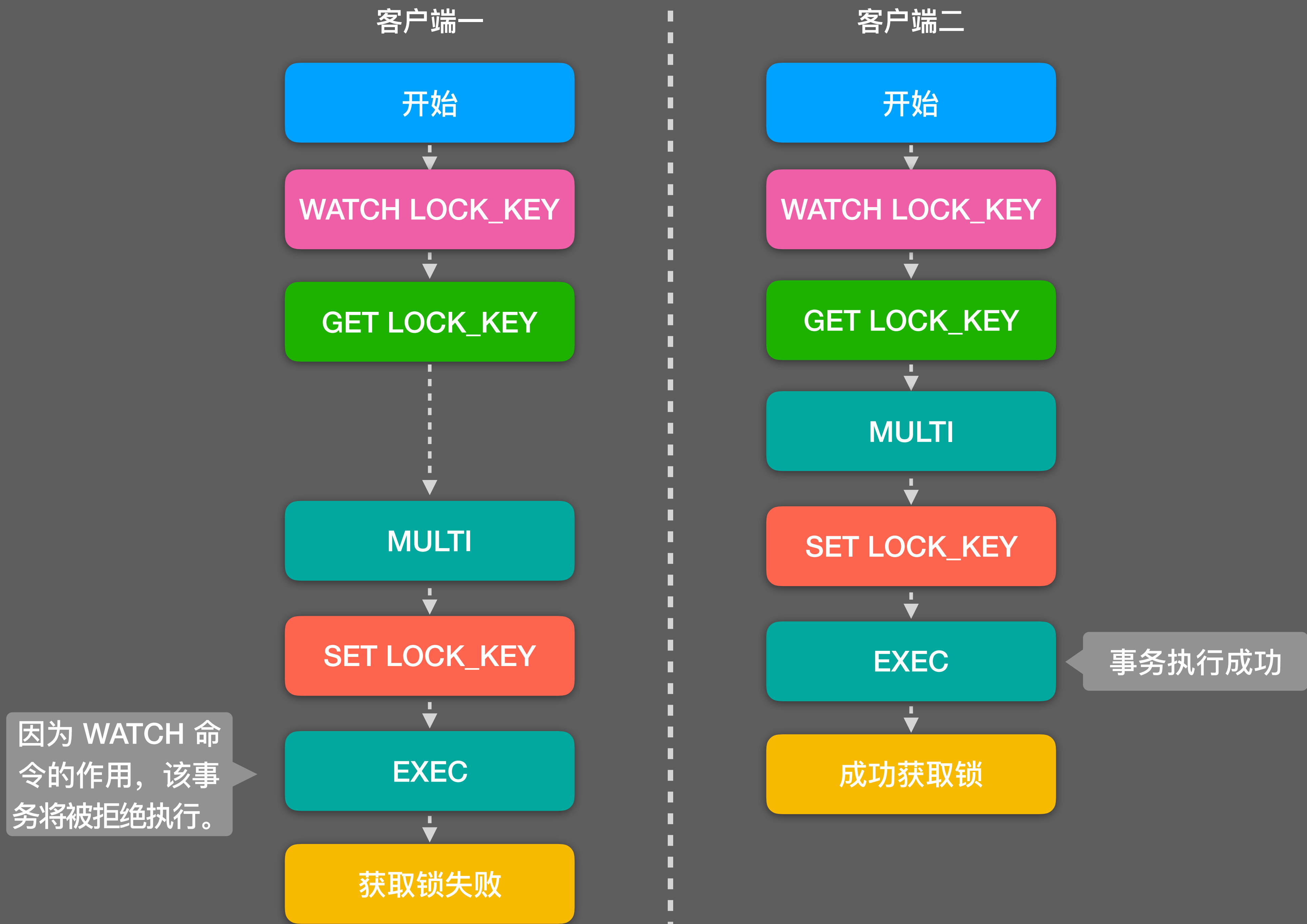
MULTI

SET LOCK_KEY

EXEC

事务执行成功

成功获取锁



使用事务带来的代价

```
func acquire(client *redis.Client) bool {
    client.Cmd("WATCH", lock_key)
    current_value, _ := client.Cmd("GET", lock_key).Str()
    if current_value == "" {
        client.Cmd("MULTI")
        client.Cmd("SET", lock_key, lock_value)
        repl, _ := client.Cmd("EXEC").List()
        if repl != nil {
            return true
        }
    }
    return false
}
```

事务的使用导致程序的逻辑变得复杂起来，并且不正确地使用事务本身就有引发 bug 的危险。

带 NX 选项的 SET 命令

SET key value NX

只在键 `key` 不存在的情况下，将它的值设置为 `value`，然后返回 `OK`；
如果键已经有值，那么放弃执行设置操作，并返回 `nil` 表示设置失败。

NX 代表 *Not eXists*，该选项从 *Redis 2.6.12* 版本开始可用。

NX 选项的作用

```
func acquire(client *redis.Client) bool {  
    current_value, _ := client.Cmd("GET", lock_key).Str()  
    if current_value == "" {  
        client.Cmd("SET", lock_key, lock_value)  
        return true  
    } else {  
        return false  
    }  
}
```

相当于在服务器
里面以原子方式
执行这两项操作

实现三 —— 使用带 NX 选项的 SET 命令

```
func acquire(client *redis.Client) bool {  
    repl, _ := client.Cmd("SET", lock_key, lock_value, "NX").Str()  
    return repl != ""  
}
```

嘿，我在这儿！

客户端一

开始

客户端二

开始

客户端一

开始

客户端二

开始

SET LOCK_KEY NX

键尚未有值，设置成功

客户端一

开始

客户端二

开始

SET LOCK_KEY NX

键尚未有值，设置成功

成功获取锁

客户端一

开始

键已经有值，设置失败

SET LOCK_KEY NX

客户端二

开始

SET LOCK_KEY NX

键尚未有值，设置成功

成功获取锁

客户端一

开始

键已经有值，设置失败

SET LOCK_KEY NX

获取锁失败

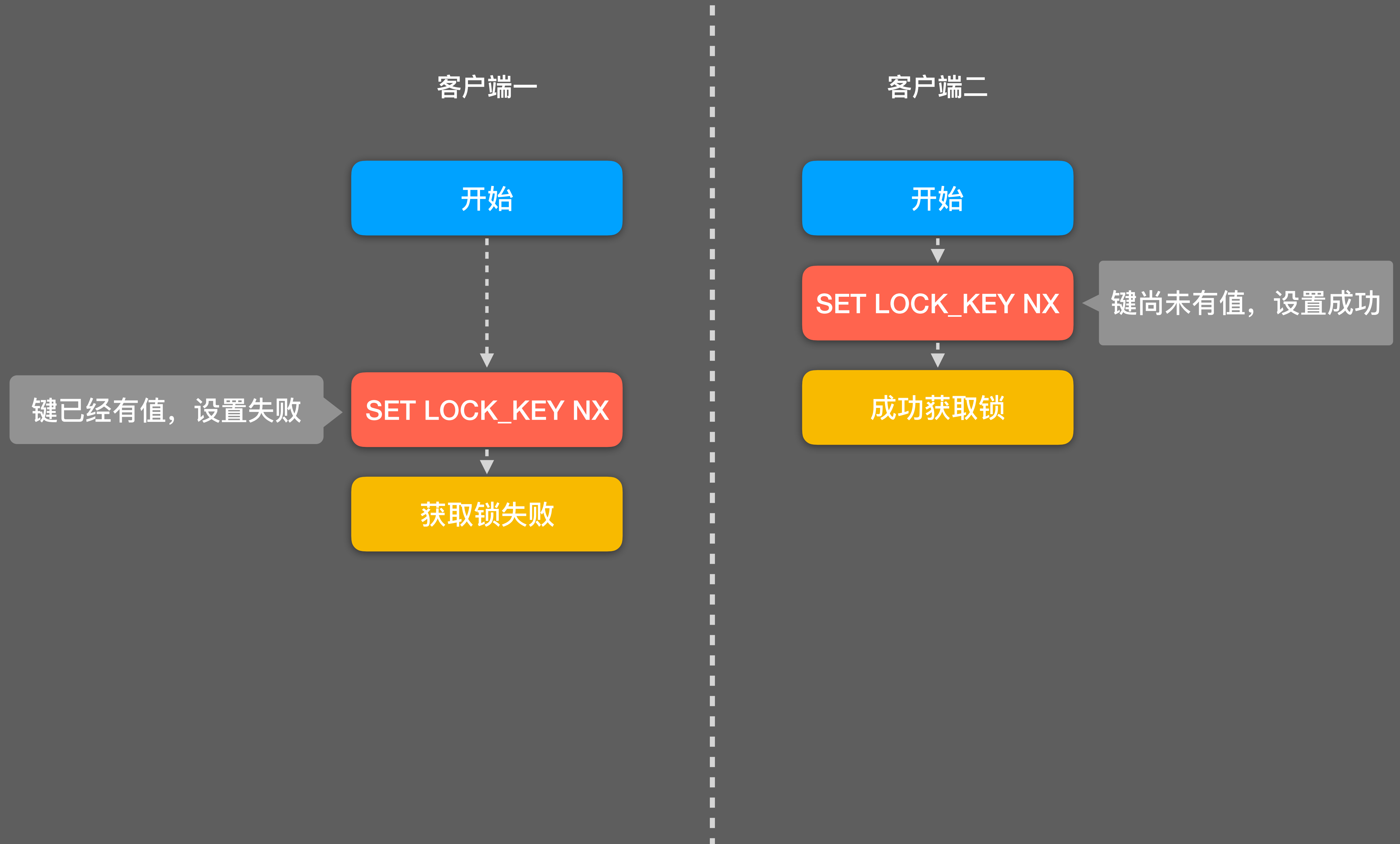
客户端二

开始

SET LOCK_KEY NX

键尚未有值，设置成功

成功获取锁



在线用户统计器

online user counter

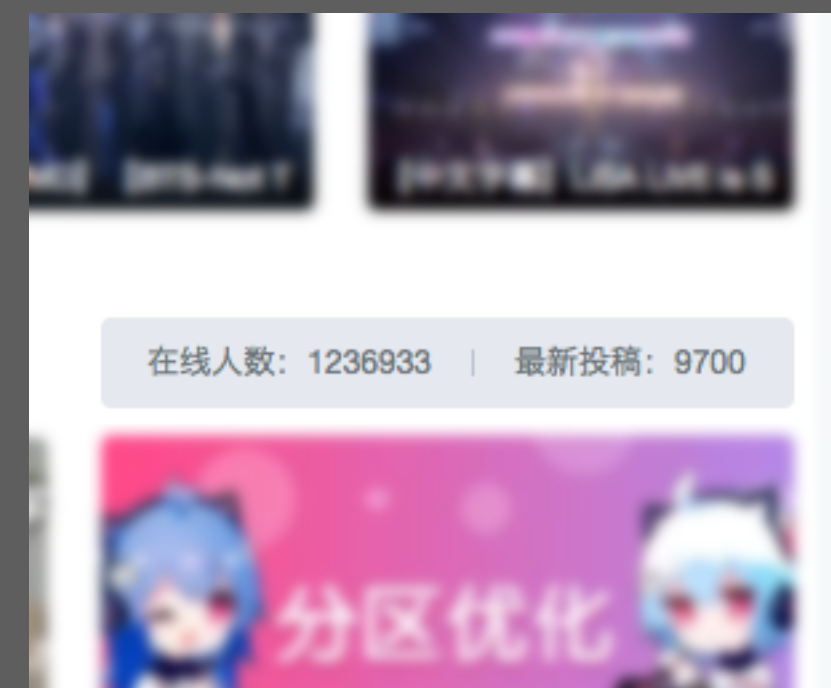
示例



analytics.google.com



panda.tv



bilibili.com



v2ex.com

方法一 —— 使用集合

当一个用户上线时，将它的用户名添加到记录在线用户的集合当中。

方法一 —— 使用集合

当一个用户上线时，将它的用户名添加到记录在线用户的集合当中。

j a c k 未 上 线



方法一 —— 使用集合

当一个用户上线时，将它的用户名添加到记录在线用户的集合当中。

j a c k 未 上 线



j a c k 已 上 线



需要用到的命令

需要用到的命令

SADD set element [element ...]

将给定的元素添加到集合当中

需要用到的命令

SADD set element [element ...]

将给定的元素添加到集合当中

SCARD set

获取集合的基数，也即是集合包含的元素数量

需要用到的命令

SADD set element [element ...]

将给定的元素添加到集合当中

SCARD set

获取集合的基数，也即是集合包含的元素数量

SISMEMBER set element

检查给定的元素是否存在于集合当中，是的话返回 1，不是的话返回 0

实现代码

```
const online_user_set = "ONLINE_USER_SET"

func set_online(client *redis.Client, user string) {
    client.Cmd("SADD", online_user_set, user)
}

func count_online(client *redis.Client) int64 {
    repl, _ := client.Cmd("SCARD", online_user_set).Int64()
    return repl
}

func is_online_or_not(client *redis.Client, user string) bool {
    repl, _ := client.Cmd("SISMEMBER", online_user_set, user).Int()
    return repl == 1
}
```

实现代码

```
const online_user_set = "ONLINE_USER_SET"
```

```
func set_online(client *redis.Client, user string) {  
    client.Cmd("SADD", online_user_set, user)  
}
```

把用户名添加至集合

```
func count_online(client *redis.Client) int64 {  
    repl, _ := client.Cmd("SCARD", online_user_set).Int64()  
    return repl  
}
```

```
func is_online_or_not(client *redis.Client, user string) bool {  
    repl, _ := client.Cmd("SISMEMBER", online_user_set, user).Int()  
    return repl == 1  
}
```

实现代码

```
const online_user_set = "ONLINE_USER_SET"
```

```
func set_online(client *redis.Client, user string) {  
    client.Cmd("SADD", online_user_set, user)  
}
```

把用户名添加至集合

```
func count_online(client *redis.Client) int64 {  
    repl, _ := client.Cmd("SCARD", online_user_set).Int64()  
    return repl  
}
```

获取集合基数

```
func is_online_or_not(client *redis.Client, user string) bool {  
    repl, _ := client.Cmd("SISMEMBER", online_user_set, user).Int()  
    return repl == 1  
}
```

实现代码

```
const online_user_set = "ONLINE_USER_SET"
```

```
func set_online(client *redis.Client, user string) {  
    client.Cmd("SADD", online_user_set, user)  
}
```

把用户名添加至集合

```
func count_online(client *redis.Client) int64 {  
    repl, _ := client.Cmd("SCARD", online_user_set).Int64()  
    return repl  
}
```

获取集合基数

```
func is_online_or_not(client *redis.Client, user string) bool {  
    repl, _ := client.Cmd("SISMEMBER", online_user_set, user).Int()  
    return repl == 1  
}
```

检查用户是否存在
于集合当中

问题

问题

集合的体积将随着元素的增加而增加，集合包含的元素越多，每个元素的体积越大，集合的体积也就越大。

问题

集合的体积将随着元素的增加而增加，集合包含的元素越多，每个元素的体积越大，集合的体积也就越大。

假设平均每个用户的名字长度为 10 字节，那么：

- 拥有一百万用户的网站每天需要使用 10 MB 内存去储存在线用户统计信息
- 拥有一千万用户的网站每天需要使用 100 MB 内存去储存在线用户统计信息

问题

集合的体积将随着元素的增加而增加，集合包含的元素越多，每个元素的体积越大，集合的体积也就越大。

假设平均每个用户的名字长度为 10 字节，那么：

- 拥有一百万用户的网站每天需要使用 10 MB 内存去储存在线用户统计信息
- 拥有一千万用户的网站每天需要使用 100 MB 内存去储存在线用户统计信息

如果我们把这些信息储存一年，那么：

- 拥有一百万用户的网站每年需要为此使用 3.65 GB 内存
- 拥有一千万用户的网站每年需要为此使用 36.5 GB 内存

问题

集合的体积将随着元素的增加而增加，集合包含的元素越多，每个元素的体积越大，集合的体积也就越大。

假设平均每个用户的名字长度为 10 字节，那么：

- 拥有一百万用户的网站每天需要使用 10 MB 内存去储存在线用户统计信息
- 拥有一千万用户的网站每天需要使用 100 MB 内存去储存在线用户统计信息

如果我们把这些信息储存一年，那么：

- 拥有一百万用户的网站每年需要为此使用 3.65 GB 内存
- 拥有一千万用户的网站每年需要为此使用 36.5 GB 内存

除此之外，因为使用 Redis 储存信息还有一些额外的消耗（overhead），所以实际的内存占用数量将比这个估算值更高。

方法二 —— 使用位图

为每个用户创建一个相对应的数字 ID ， 当一个用户上线时，使用他的 ID 作为索引，将位图指定索引上的二进制位设置为 1 。

方法二 —— 使用位图

为每个用户创建一个相对应的数字 ID ， 当一个用户上线时，使用他的 ID 作为索引，将位图指定索引上的二进制位设置为 1 。

用户名

`"peter"`

方法二 —— 使用位图

为每个用户创建一个相对应的数字 ID ， 当一个用户上线时，使用他的 ID 作为索引，将位图指定索引上的二进制位设置为 1 。

用户名

"peter"



ID

10086

方法二 —— 使用位图

为每个用户创建一个相对应的数字 ID ， 当一个用户上线时，使用他的 ID 作为索引，将位图指定索引上的二进制位设置为 1 。



需要用到的命令

需要用到的命令

SETBIT bitmap index value

将位图指定索引上的二进制位设置为给定的值

需要用到的命令

SETBIT bitmap index value

将位图指定索引上的二进制位设置为给定的值

GETBIT bitmap index

获取位图指定索引上的二进制位

需要用到的命令

SETBIT bitmap index value

将位图指定索引上的二进制位设置为给定的值

GETBIT bitmap index

获取位图指定索引上的二进制位

BITCOUNT bitmap

统计位图中值为 1 的二进制位的数量

实现代码

```
const online_user_bitmap = "ONLINE_USER_BITMAP"

func set_online(client *redis.Client, user_id int64) {
    client.Cmd("SETBIT", online_user_bitmap, user_id, 1)
}

func count_online(client *redis.Client) int64 {
    repl, _ := client.Cmd("BITCOUNT", online_user_bitmap).Int64()
    return repl
}

func is_online_or_not(client *redis.Client, user_id int64) bool {
    repl, _ := client.Cmd("GETBIT", online_user_bitmap, user_id).Int()
    return repl == 1
}
```

实现代码

```
const online_user_bitmap = "ONLINE_USER_BITMAP"
```

```
func set_online(client *redis.Client, user_id int64) {  
    client.Cmd("SETBIT", online_user_bitmap, user_id, 1)
```

设置二进制位

```
}  
  
func count_online(client *redis.Client) int64 {  
    repl, _ := client.Cmd("BITCOUNT", online_user_bitmap).Int64()  
    return repl  
}
```

```
func is_online_or_not(client *redis.Client, user_id int64) bool {  
    repl, _ := client.Cmd("GETBIT", online_user_bitmap, user_id).Int()  
    return repl == 1  
}
```

实现代码

```
const online_user_bitmap = "ONLINE_USER_BITMAP"
```

```
func set_online(client *redis.Client, user_id int64) {  
    client.Cmd("SETBIT", online_user_bitmap, user_id, 1)
```

设置二进制位

```
func count_online(client *redis.Client) int64 {  
    repl, _ := client.Cmd("BITCOUNT", online_user_bitmap).Int64()  
    return repl
```

统计值为 1 的二进制位数量

```
func is_online_or_not(client *redis.Client, user_id int64) bool {  
    repl, _ := client.Cmd("GETBIT", online_user_bitmap, user_id).Int()  
    return repl == 1  
}
```

实现代码

```
const online_user_bitmap = "ONLINE_USER_BITMAP"
```

```
func set_online(client *redis.Client, user_id int64) {  
    client.Cmd("SETBIT", online_user_bitmap, user_id, 1)
```

设置二进制位

```
func count_online(client *redis.Client) int64 {  
    repl, _ := client.Cmd("BITCOUNT", online_user_bitmap).Int64()  
    return repl
```

统计值为 1 的二进制位数量

```
func is_online_or_not(client *redis.Client, user_id int64) bool {  
    repl, _ := client.Cmd("GETBIT", online_user_bitmap, user_id).Int()  
    return repl == 1
```

检查指定二进制位的值

改进

虽然位图的体积仍然会随着用户数量的增多而变大，但因为记录每个用户所需的内存数量从原来的平均 10 字节变成了 1 位，所以实现方法二将节约大量内存。

改进

虽然位图的体积仍然会随着用户数量的增多而变大，但因为记录每个用户所需的内存数量从原来的平均 10 字节变成了 1 位，所以实现方法二将节约大量内存。

储存一天用户在线信息所需的内存数量：

- 一百万人，125 KB ；
- 一千万人， 1250 KB = 1.25 MB ；

改进

虽然位图的体积仍然会随着用户数量的增多而变大，但因为记录每个用户所需的内存数量从原来的平均 10 字节变成了 1 位，所以实现方法二将节约大量内存。

储存一天用户在线信息所需的内存数量：

- 一百万人，125 KB ；
- 一千万人， 1250 KB = 1.25 MB ；

储存一年用户在线信息所需的内存数量：

- 一百万人，45.625 MB
- 一千万人，456.25 MB

方法三 —— 使用 HyperLogLog

当一个用户上线时，使用统计在线用户数量的 HyperLogLog 对其进行计数。

方法三 —— 使用 HyperLogLog

当一个用户上线时，使用统计在线用户数量的 HyperLogLog 对其进行计数。

“jack”

方法三 —— 使用 HyperLogLog

当一个用户上线时，使用统计在线用户数量的 HyperLogLog 对其进行计数。

“jack” -----> HyperLogLog 算法

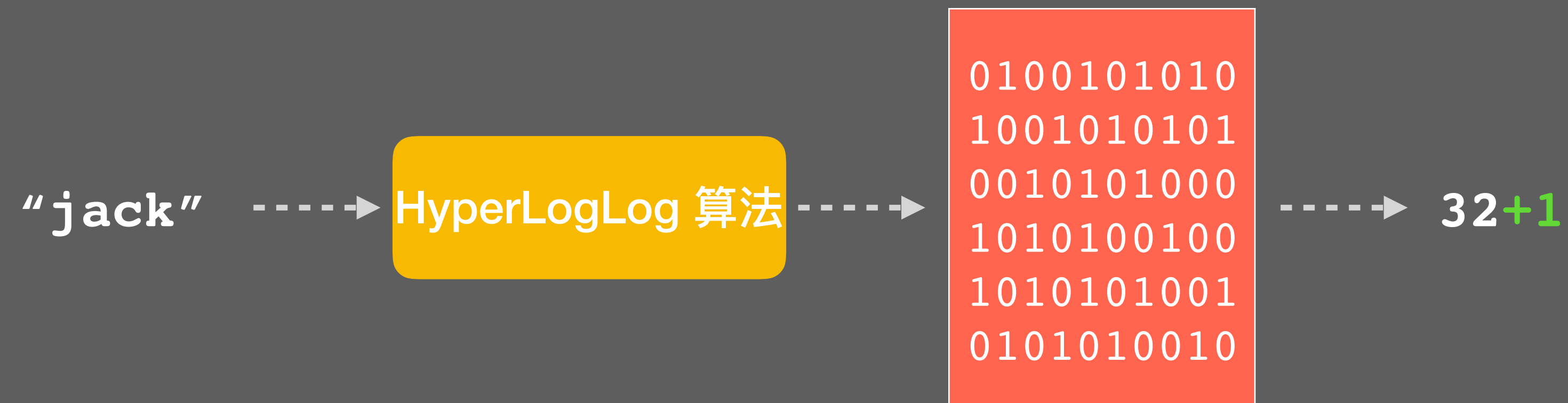
方法三 —— 使用 HyperLogLog

当一个用户上线时，使用统计在线用户数量的 HyperLogLog 对其进行计数。



方法三 —— 使用 HyperLogLog

当一个用户上线时，使用统计在线用户数量的 HyperLogLog 对其进行计数。



需要用到的命令

需要用到的命令

```
PFADD h11 element [element ...]
```

使用 HyperLogLog 对给定元素进行计数

需要用到的命令

```
PFADD h11 element [element ...]
```

使用 HyperLogLog 对给定元素进行计数

```
PFCOUNT h11
```

获取 HyperLogLog 的近似基数，也即是基数的估算值

需要用到的命令

```
PFADD h11 element [element ...]
```

使用 HyperLogLog 对给定元素进行计数

```
PFCOUNT h11
```

获取 HyperLogLog 的近似基数，也即是基数的估算值

缺陷：因为 HyperLogLog 的概率性质，我们无法准确地知道特定的一个用户是否在线，换句话说，这个实现只能给出在线用户的数量，无法检测用户是否在线。

实现代码

```
const online_user_hll = "ONLINE_USER_HLL"

func set_online(client *redis.Client, user string) {
    client.Cmd("PFADD", online_user_hll, user)
}

func count_online(client *redis.Client) int64 {
    repl, _ := client.Cmd("PFCOUNT", online_user_hll).Int64()
    return repl
}
```

实现代码

```
const online_user_hll = "ONLINE_USER_HLL"
```

```
func set_online(client *redis.Client, user string) {  
    client.Cmd("PFADD", online_user_hll, user)  
}
```

对用户进行计数

```
func count_online(client *redis.Client) int64 {  
    repl, _ := client.Cmd("PFCOUNT", online_user_hll).Int64()  
    return repl  
}
```

实现代码

```
const online_user_hll = "ONLINE_USER_HLL"
```

```
func set_online(client *redis.Client, user string) {  
    client.Cmd("PFADD", online_user_hll, user)  
}
```

对用户进行计数

```
func count_online(client *redis.Client) int64 {  
    repl, _ := client.Cmd("PFCOUNT", online_user_hll).Int64()  
    return repl  
}
```

获取近似基数

三种实现的内存消耗对比

规模/实现	集合	位图	HyperLogLog
一百万，一天			
一千万，一天			
一百万，一年			
一千万，一年			

三种实现的内存消耗对比

规模/实现	集合	位图	HyperLogLog
一百万，一天	10 MB		
一千万，一天	100 MB		
一百万，一年	3.65 GB		
一千万，一年	36.5 GB		

三种实现的内存消耗对比

规模/实现	集合	位图	HyperLogLog
一百万，一天	10 MB	125 KB	
一千万，一天	100 MB	1.25 MB	
一百万，一年	3.65 GB	45.625 MB	
一千万，一年	36.5 GB	456.25 MB	

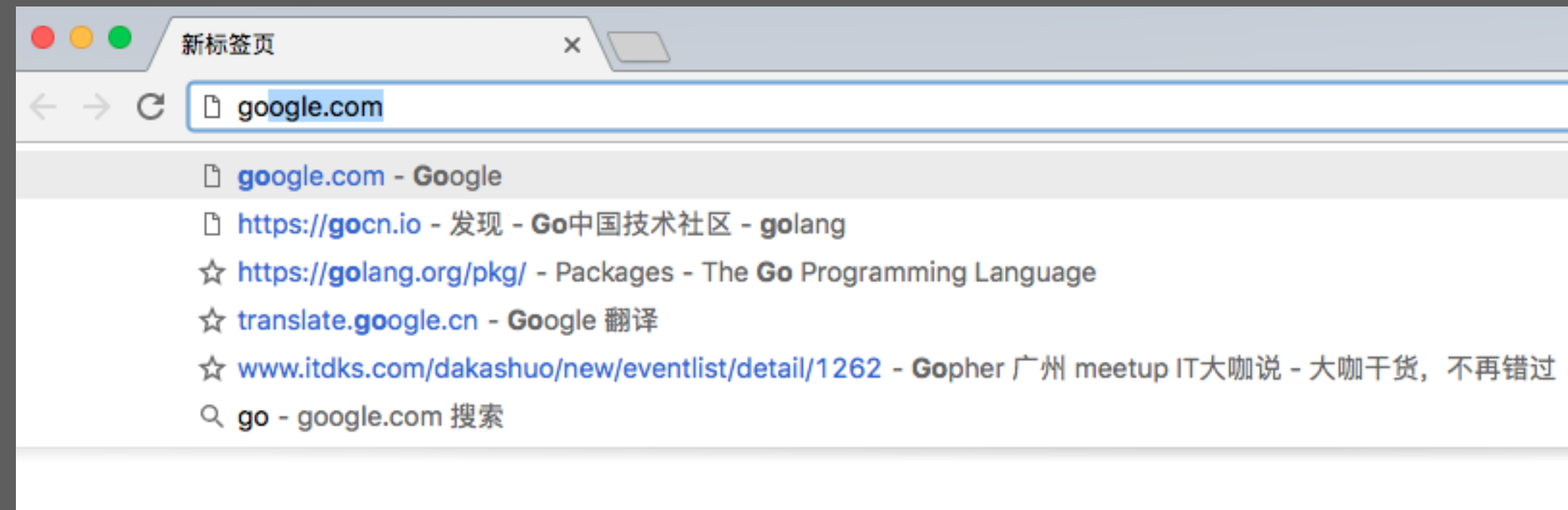
三种实现的内存消耗对比

规模/实现	集合	位图	HyperLogLog
一百万，一天	10 MB	125 KB	12 KB
一千万，一天	100 MB	1.25 MB	12 KB
一百万，一年	3.65 GB	45.625 MB	4.32 MB
一千万，一年	36.5 GB	456.25 MB	4.32 MB

自动补全

autocomplete

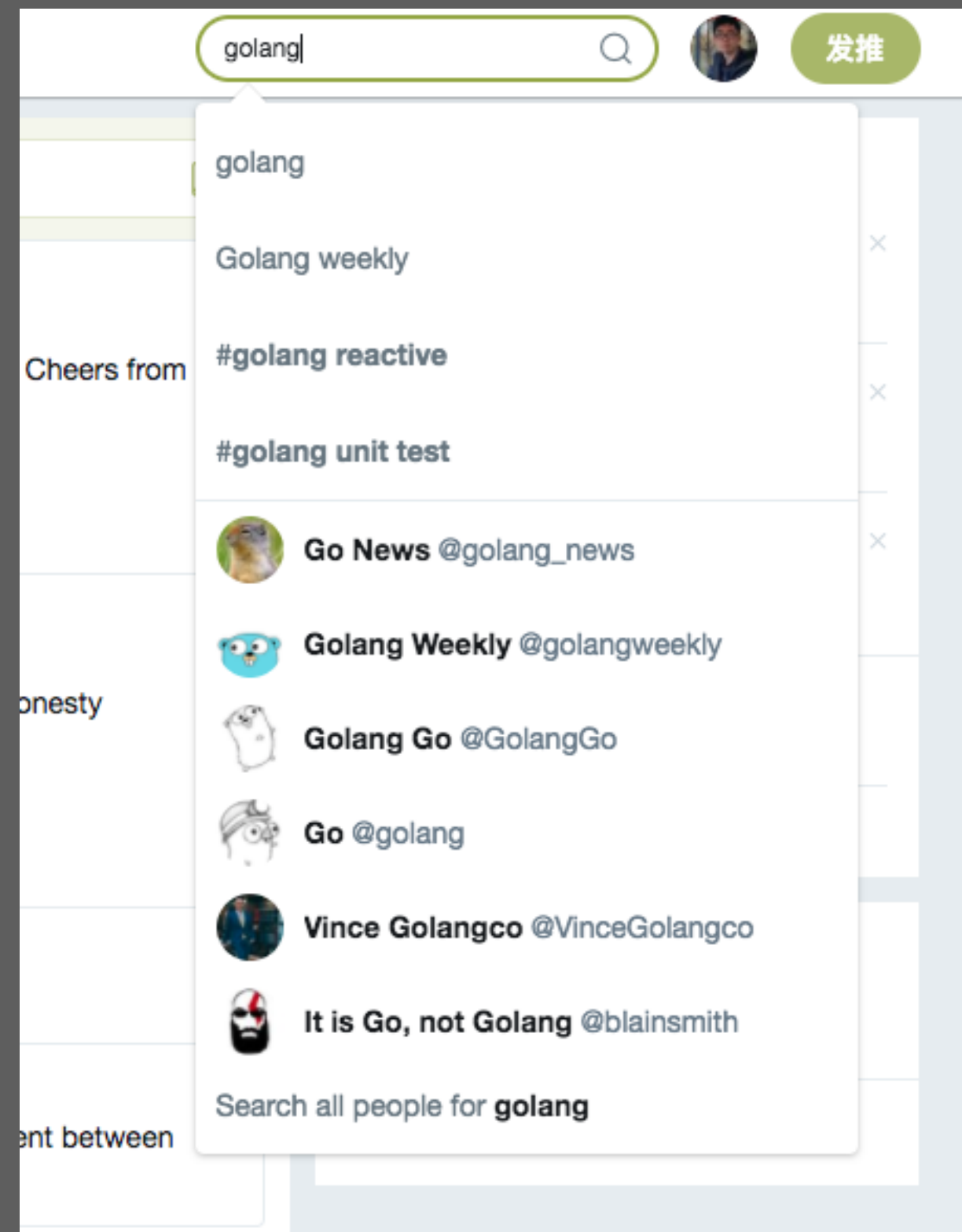
示例



示例



示例



示例



原理解释

原理解释

> go

原理解释

```
> go  
"google"  
"gmail"  
"gopher"  
"great"  
"guide"
```

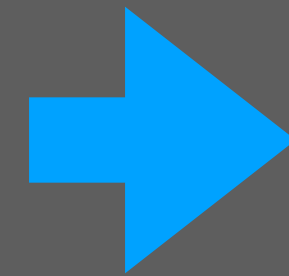

原理解释

```
> go  
"google"  
"gmail"  
"gopher"  
"great"  
"guide"
```

输入与候选结果

原理解释

```
> go  
"google"  
"gmail"  
"gopher"  
"great"  
"guide"
```



```
w1, "google"  
w2, "gmail"  
w3, "gopher"  
w4, "great"  
w5, "guide"
```

输入与候选结果

权重与候选结果

原理解释

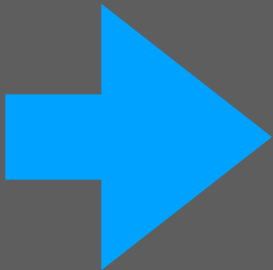
> go
"google"
"gmail"
"gopher"
"great"
"guide"

输入与候选结果



w1, "google"
w2, "gmail"
w3, "gopher"
w4, "great"
w5, "guide"

权重与候选结果



分值	成员
w1	"google"
w2	"gmail"
w3	"gopher"
w4	"great"
w5	"guide"

使用有序集合储存权重表

构建权重表

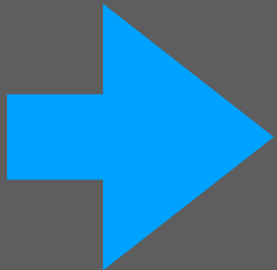
构建权重表

分值	成员
320	“google”
300	“gmail”
278	“great”
277	“gopher”
210	“guide”

现有的权重表

构建权重表

分值	成员
320	"google"
300	"gmail"
278	"great"
277	"gopher"
210	"guide"



> gopher
> gopher
> gopher

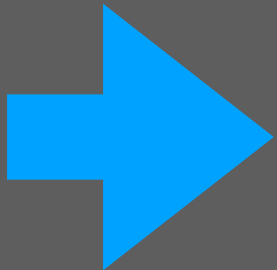
现有的权重表

获得三个新的输入

构建权重表

分值	成员
320	"google"
300	"gmail"
278	"great"
277	"gopher"
210	"guide"

现有的权重表



> gopher
> gopher
> gopher

获得三个新的输入



分值	成员
320	"google"
300	"gmail"
280	"gopher"
278	"great"
210	"guide"

根据输入更新权重

构建多个权重表

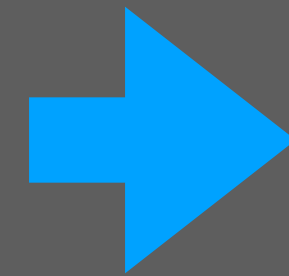
构建多个权重表

> `gopher`

根据输入

构建多个权重表

> gopher



"g"
"go"
"gop"
"goph"
"gophe"
"gopher"

根据输入

枚举出字符串的组成排列

构建多个权重表

> gopher



"g"
"go"
"gop"
"goph"
"gophe"
"gopher"



分值	成员
320	"google"
300	"gmail"
280 +1	"gopher"
278	"great"
210	"guide"

根据输入

枚举出字符串的组成排列

根据排列对各个权重表进行更新

补全过程

补全过程

> g

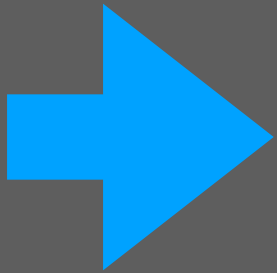
补全过程

分值	成员
320	"google"
300	"gmail"
280	"gopher"
278	"great"
210	"guide"

> g

补全过程

分值	成员
320	"google"
300	"gmail"
280	"gopher"
278	"great"
210	"guide"



> g

> go

补全过程

分值	成员
320	"google"
300	"gmail"
280	"gopher"
278	"great"
210	"guide"

> g



分值	成员
320	"google"
280	"gopher"
255	"goodbye"
230	"gold"
188	"gossip"

> go

补全过程

分值	成员
320	"google"
300	"gmail"
280	"gopher"
278	"great"
210	"guide"

> g



分值	成员
320	"google"
280	"gopher"
255	"goodbye"
230	"gold"
188	"gossip"

> go



> gop

补全过程

分值	成员
320	"google"
300	"gmail"
280	"gopher"
278	"great"
210	"guide"

> g



分值	成员
320	"google"
280	"gopher"
255	"goodbye"
230	"gold"
188	"gossip"

> go



分值	成员
280	"gopher"
255	"gopak"
232	"gopher hold"
180	"gopura"
75	"gophering"

> gop

需要用到的命令

需要用到的命令

ZINCRBY zset increment member

对给定成员的分值执行自增操作

需要用到的命令

ZINCRBY zset increment member

对给定成员的分值执行自增操作

ZREVRANGE zset start end [WITHSCORES]

按照分值从大到小的顺序，从有序集合里面获取指定索引范围内的成员

实现代码

```
const autocomplete = "autocomplete::"

func feed(client *redis.Client, content string, weight int) {
    for i, _ := range content {
        segment := content[:i+1]
        key := autocomplete + segment
        client.Cmd("ZINCRBY", key, weight, content)
    }
}

func hint(client *redis.Client, prefix string, count int) []string {
    key := autocomplete + prefix
    result, _ := client.Cmd("ZREVRANGE", key, 0, count-1).List()
    return result
}
```

实现代码

```
const autocomplete = "autocomplete::"
```

```
func feed(client *redis.Client, content string, weight int) {  
    for i := range content {  
        segment := content[:i+1]  
        key := autocomplete + segment  
        client.Cmd("ZINCRBY", key, weight, content)  
    }  
}
```

枚举字符串
组成排列

```
func hint(client *redis.Client, prefix string, count int) []string {  
    key := autocomplete + prefix  
    result, _ := client.Cmd("ZREVRANGE", key, 0, count-1).List()  
    return result  
}
```

实现代码

```
const autocomplete = "autocomplete::"
```

```
func feed(client *redis.Client, content string, weight int) {  
    for i := range content {  
        segment := content[:i+1]  
        key := autocomplete + segment  
        client.Cmd("ZINCRBY", key, weight, content)  
    }  
}
```

枚举字符串
组成排列

拼接出各个权重表的键名

```
func hint(client *redis.Client, prefix string, count int) []string {  
    key := autocomplete + prefix  
    result, _ := client.Cmd("ZREVRANGE", key, 0, count-1).List()  
    return result  
}
```


实现代码

```
const autocomplete = "autocomplete::"
```

```
func feed(client *redis.Client, content string, weight int) {  
    for i := range content {  
        segment := content[:i+1]  
        key := autocomplete + segment  
        client.Cmd("ZINCRBY", key, weight, content)  
    }  
}
```

枚举字符串
组成排列

拼接出各个权重表的键名

对各个权重表进行更新

```
func hint(client *redis.Client, prefix string, count int) []string {  
    key := autocomplete + prefix  
    result, _ := client.Cmd("ZREVRANGE", key, 0, count-1).List()  
    return result  
}
```

实现代码

```
const autocomplete = "autocomplete::"
```

```
func feed(client *redis.Client, content string, weight int) {  
    for i := range content {  
        segment := content[:i+1]  
        key := autocomplete + segment  
        client.Cmd("ZINCRBY", key, weight, content)  
    }  
}
```

枚举字符串
组成排列

拼接出各个权重表的键名

对各个权重表进行更新

```
func hint(client *redis.Client, prefix string, count int) []string {  
    key := autocomplete + prefix  
    result, _ := client.Cmd("ZREVRANGE", key, 0, count-1).List()  
    return result  
}
```

拼接出权重
表的键名

实现代码

```
const autocomplete = "autocomplete::"
```

```
func feed(client *redis.Client, content string, weight int) {  
    for i := range content {  
        segment := content[:i+1]  
        key := autocomplete + segment  
        client.Cmd("ZINCRBY", key, weight, content)  
    }  
}
```

枚举字符串
组成排列

拼接出各个权重表的键名

对各个权重表进行更新

```
func hint(client *redis.Client, prefix string, count int) []string {  
    key := autocomplete + prefix  
    result, _ := client.Cmd("ZREVRANGE", key, 0, count-1).List()  
    return result  
}
```

拼接出权重
表的键名

按权重从大到小
获取候选结果

总结

总结

- Go 和 Redis 都是简单且强大的工具，组合使用它们能够轻而易举地解决很多过去非常难以实现或者需要很多代码才能实现的特性（又黑我大 JAVA，放学别走！）。

总结

- Go 和 Redis 都是简单且强大的工具，组合使用它们能够轻而易举地解决很多过去非常难以实现或者需要很多代码才能实现的特性（又黑我大 JAVA，放学别走！）。
- 在构建程序的时候一定要确保程序的正确性和安全性，虽然为了保证这两点常常会使得程序变得复杂，但有时候工具本身也会提供一些鱼和熊掌兼得的方案。

总结

- Go 和 Redis 都是简单且强大的工具，组合使用它们能够轻而易举地解决很多过去非常难以实现或者需要很多代码才能实现的特性（又黑我大 JAVA，放学别走！）。
- 在构建程序的时候一定要确保程序的正确性和安全性，虽然为了保证这两点常常会使得程序变得复杂，但有时候工具本身也会提供一些鱼和熊掌兼得的方案。
- 解决一个问题通常会有很多不同的方式可选，但我们必须对正在使用的工具足够熟悉，才能找到这些方法。

总结

- Go 和 Redis 都是简单且强大的工具，组合使用它们能够轻而易举地解决很多过去非常难以实现或者需要很多代码才能实现的特性（又黑我大 JAVA，放学别走！）。
- 在构建程序的时候一定要确保程序的正确性和安全性，虽然为了保证这两点常常会使得程序变得复杂，但有时候工具本身也会提供一些鱼和熊掌兼得的方案。
- 解决一个问题通常会有很多不同的方式可选，但我们必须对正在使用的工具足够熟悉，才能找到这些方法。
- 最后，不同实现的效率和功能通常也会有所不同，我们要根据自身的情况进行选择，不要盲目的相信所谓的“最优解”。

多谢大家，得闲饮茶！

thank you!

©黄健宏, 2017 · 保留所有权利, 禁止未经许可的转载和商用