Golang redis 学习指南

安装

我使用的是 **[https://github.com/go-redis/r...](https://github.com/go-redis/redis)** 这个 golang 客户端, 因此安装方式如下:

go get gopkg.in/redis.v4

接着在代码中导入此包即可:

import "gopkg.in/redis.v4"

基本操作

创建客户端

通过 redis.NewClient 函数即可创建一个 redis 客户端, 这个方法接收一个 redis.Options 对象参数, 通过这个参数, 我们可以配置 redis 相关的属性, 例如 redis 服务器地址, 数据库名, 数据库密码等.  
下面是一个连接的例子:

// 创建 redis 客户端

func createClient() \*redis.Client {

client := redis.NewClient(&redis.Options{

Addr: "localhost:6379",

Password: "",

DB: 0,

})

// 通过 cient.Ping() 来检查是否成功连接到了 redis 服务器

pong, err := client.Ping().Result()

fmt.Println(pong, err)

return client

}

String 操作

redis 的 String 操作有:

set(key, value)：给数据库中名称为key的string赋予值value

get(key)：返回数据库中名称为key的string的value

getset(key, value)：给名称为key的string赋予上一次的value

mget(key1, key2,…, key N)：返回库中多个string的value

setnx(key, value)：添加string，名称为key，值为value

setex(key, time, value)：向库中添加string，设定过期时间time

mset(key N, value N)：批量设置多个string的值

msetnx(key N, value N)：如果所有名称为key i的string都不存在

incr(key)：名称为key的string增1操作

incrby(key, integer)：名称为key的string增加integer

decr(key)：名称为key的string减1操作

decrby(key, integer)：名称为key的string减少integer

append(key, value)：名称为key的string的值附加value

substr(key, start, end)：返回名称为key的string的value的子串

在 go-redis 中, 我们可以直接找到对应的操作方法, 直接上代码:

// String 操作

func stringOperation(client \*redis.Client) {

// 第三个参数是过期时间, 如果是0, 则表示没有过期时间.

err := client.Set("name", "xys", 0).Err()

if err != nil {

panic(err)

}

val, err := client.Get("name").Result()

if err != nil {

panic(err)

}

fmt.Println("name", val)

// 这里设置过期时间.

err = client.Set("age", "20", 1 \* time.Second).Err()

if err != nil {

panic(err)

}

client.Incr("age") // 自增

client.Incr("age") // 自增

client.Decr("age") // 自减

val, err = client.Get("age").Result()

if err != nil {

panic(err)

}

fmt.Println("age", val) // age 的值为21

// 因为 key "age" 的过期时间是一秒钟, 因此当一秒后, 此 key 会自动被删除了.

time.Sleep(1 \* time.Second)

val, err = client.Get("age").Result()

if err != nil {

// 因为 key "age" 已经过期了, 因此会有一个 redis: nil 的错误.

fmt.Printf("error: %v\n", err)

}

fmt.Println("age", val)

}

list 操作

redis 的 list 操作有:

rpush(key, value)：在名称为key的list尾添加一个值为value的元素

lpush(key, value)：在名称为key的list头添加一个值为value的 元素

llen(key)：返回名称为key的list的长度

lrange(key, start, end)：返回名称为key的list中start至end之间的元素

ltrim(key, start, end)：截取名称为key的list

lindex(key, index)：返回名称为key的list中index位置的元素

lset(key, index, value)：给名称为key的list中index位置的元素赋值

lrem(key, count, value)：删除count个key的list中值为value的元素

lpop(key)：返回并删除名称为key的list中的首元素

rpop(key)：返回并删除名称为key的list中的尾元素

blpop(key1, key2,… key N, timeout)：lpop命令的block版本。

brpop(key1, key2,… key N, timeout)：rpop的block版本。

rpoplpush(srckey, dstkey)：返回并删除名称为srckey的list的尾元素，并将该元素添加到名称为dstkey的list的头部

同样地, 在 go-redis 中也可以找到对应的方法, 下面是一个简单的示例:

// list 操作

func listOperation(client \*redis.Client) {

client.RPush("fruit", "apple") //在名称为 fruit 的list尾添加一个值为value的元素

client.LPush("fruit", "banana") //在名称为 fruit 的list头添加一个值为value的 元素

length, err := client.LLen("fruit").Result() //返回名称为 fruit 的list的长度

if err != nil {

panic(err)

}

fmt.Println("length: ", length) // 长度为2

value, err := client.LPop("fruit").Result() //返回并删除名称为 fruit 的list中的首元素

if err != nil {

panic(err)

}

fmt.Println("fruit: ", value)

value, err = client.RPop("fruit").Result() // 返回并删除名称为 fruit 的list中的尾元素

if err != nil {

panic(err)

}

fmt.Println("fruit: ", value)

}

set 操作

redis 的 set 操作:

sadd(key, member)：向名称为key的set中添加元素member

srem(key, member) ：删除名称为key的set中的元素member

spop(key) ：随机返回并删除名称为key的set中一个元素

smove(srckey, dstkey, member) ：移到集合元素

scard(key) ：返回名称为key的set的基数

sismember(key, member) ：member是否是名称为key的set的元素

sinter(key1, key2,…key N) ：求交集

sinterstore(dstkey, (keys)) ：求交集并将交集保存到dstkey的集合

sunion(key1, (keys)) ：求并集

sunionstore(dstkey, (keys)) ：求并集并将并集保存到dstkey的集合

sdiff(key1, (keys)) ：求差集

sdiffstore(dstkey, (keys)) ：求差集并将差集保存到dstkey的集合

smembers(key) ：返回名称为key的set的所有元素

srandmember(key) ：随机返回名称为key的set的一个元素

接下来是 go-redis 的 set 操作:

// set 操作

func setOperation(client \*redis.Client) {

client.SAdd("blacklist", "Obama") // 向 blacklist 中添加元素

client.SAdd("blacklist", "Hillary") // 再次添加

client.SAdd("blacklist", "the Elder") // 添加新元素

client.SAdd("whitelist", "the Elder") // 向 whitelist 添加元素

// 判断元素是否在集合中

isMember, err := client.SIsMember("blacklist", "Bush").Result()

if err != nil {

panic(err)

}

fmt.Println("Is Bush in blacklist: ", isMember)

// 求交集, 即既在黑名单中, 又在白名单中的元素

names, err := client.SInter("blacklist", "whitelist").Result()

if err != nil {

panic(err)

}

// 获取到的元素是 "the Elder"

fmt.Println("Inter result: ", names)

// 获取指定集合的所有元素

all, err := client.SMembers("blacklist").Result()

if err != nil {

panic(err)

}

fmt.Println("All member: ", all)

}

hash 操作

redis 的 hash 操作:

hset(key, field, value)：向名称为key的hash中添加元素field

hget(key, field)：返回名称为key的hash中field对应的value

hmget(key, (fields))：返回名称为key的hash中field i对应的value

hmset(key, (fields))：向名称为key的hash中添加元素field

hincrby(key, field, integer)：将名称为key的hash中field的value增加integer

hexists(key, field)：名称为key的hash中是否存在键为field的域

hdel(key, field)：删除名称为key的hash中键为field的域

hlen(key)：返回名称为key的hash中元素个数

hkeys(key)：返回名称为key的hash中所有键

hvals(key)：返回名称为key的hash中所有键对应的value

hgetall(key)：返回名称为key的hash中所有的键（field）及其对应的value

go-redis 中的 hash 操作:

// hash 操作

func hashOperation(client \*redis.Client) {

client.HSet("user\_xys", "name", "xys"); // 向名称为 user\_xys 的 hash 中添加元素 name

client.HSet("user\_xys", "age", "18"); // 向名称为 user\_xys 的 hash 中添加元素 age

// 批量地向名称为 user\_test 的 hash 中添加元素 name 和 age

client.HMSet("user\_test", map[string]string{"name": "test", "age":"20"})

// 批量获取名为 user\_test 的 hash 中的指定字段的值.

fields, err := client.HMGet("user\_test", "name", "age").Result()

if err != nil {

panic(err)

}

fmt.Println("fields in user\_test: ", fields)

// 获取名为 user\_xys 的 hash 中的字段个数

length, err := client.HLen("user\_xys").Result()

if err != nil {

panic(err)

}

fmt.Println("field count in user\_xys: ", length) // 字段个数为2

// 删除名为 user\_test 的 age 字段

client.HDel("user\_test", "age")

age, err := client.HGet("user\_test", "age").Result()

if err != nil {

fmt.Printf("Get user\_test age error: %v\n", err)

} else {

fmt.Println("user\_test age is: ", age) // 字段个数为2

}

}

关于连接池

redis.v4 包实现了 redis 的连接池管理, 因此我们就不需要自己手动管理 redis 的连接了.  
默认情况下, redis.v4 的 redis 连接池大小是10, 不过我们可以在初始化 redis 客户端时自行设置连接池的大小, 例如:

client := redis.NewClient(&redis.Options{

Addr: "localhost:6379",

Password: "",

DB: 0,

PoolSize: 5,

})

通过 redis.Options 的 PoolSize 属性, 我们设置了 redis 连接池的大小为5.  
那么接下来我们来看一下这个设置有什么效果吧:

// redis.v4 的连接池管理

func connectPool(client \*redis.Client) {

wg := sync.WaitGroup{}

wg.Add(10)

for i := 0; i < 10; i++ {

go func() {

defer wg.Done()

for j := 0; j < 100; j++ {

client.Set(fmt.Sprintf("name%d", j), fmt.Sprintf("xys%d", j), 0).Err()

client.Get(fmt.Sprintf("name%d", j)).Result()

}

fmt.Printf("PoolStats, TotalConns: %d, FreeConns: %d\n", client.PoolStats().TotalConns, client.PoolStats().FreeConns);

}()

}

wg.Wait()

}

上面的例子启动了10个 routine 来不断向 redis 读写数据, 然后我们通过 client.PoolStats() 获取连接池的信息. 运行这个例子, 输出如下:

PoolStats, TotalConns: 5, FreeConns: 1

PoolStats, TotalConns: 5, FreeConns: 1

PoolStats, TotalConns: 5, FreeConns: 1

PoolStats, TotalConns: 5, FreeConns: 1

PoolStats, TotalConns: 5, FreeConns: 1

PoolStats, TotalConns: 5, FreeConns: 2

PoolStats, TotalConns: 5, FreeConns: 2

PoolStats, TotalConns: 5, FreeConns: 3

PoolStats, TotalConns: 5, FreeConns: 4

PoolStats, TotalConns: 5, FreeConns: 5

通过输出可以看到, 此时最大的连接池数量确实是 5 了, 并且一开始时, 因为 coroutine 的数量大于5, 会造成 redis 连接不足的情况(反映在 FreeConns 上就是前几次的输出 FreeConns 一直是1), 当某个 coroutine 结束后, 会释放此 redis 连接, 因此 FreeConns 会增加.

完整示例

//

// author xiongyongshun

// project go\_redis

// version 1.0

// created 16/10/6 03:49

//

package main

import (

"fmt"

"gopkg.in/redis.v4"

"time"

"sync"

)

func main() {

client := createClient()

defer client.Close()

stringOperation(client)

listOperation(client)

setOperation(client)

hashOperation(client)

connectPool(client)

}

// 创建 redis 客户端

func createClient() \*redis.Client {

client := redis.NewClient(&redis.Options{

Addr: "localhost:6379",

Password: "",

DB: 0,

PoolSize: 5,

})

pong, err := client.Ping().Result()

fmt.Println(pong, err)

return client

}

// String 操作

func stringOperation(client \*redis.Client) {

// 第三个参数是过期时间, 如果是0, 则表示没有过期时间.

err := client.Set("name", "xys", 0).Err()

if err != nil {

panic(err)

}

val, err := client.Get("name").Result()

if err != nil {

panic(err)

}

fmt.Println("name", val)

// 这里设置过期时间.

err = client.Set("age", "20", 1 \* time.Second).Err()

if err != nil {

panic(err)

}

client.Incr("age") // 自增

client.Incr("age") // 自增

client.Decr("age") // 自减

val, err = client.Get("age").Result()

if err != nil {

panic(err)

}

fmt.Println("age", val) // age 的值为21

// 因为 key "age" 的过期时间是一秒钟, 因此当一秒后, 此 key 会自动被删除了.

time.Sleep(1 \* time.Second)

val, err = client.Get("age").Result()

if err != nil {

// 因为 key "age" 已经过期了, 因此会有一个 redis: nil 的错误.

fmt.Printf("error: %v\n", err)

}

fmt.Println("age", val)

}

// list 操作

func listOperation(client \*redis.Client) {

client.RPush("fruit", "apple") //在名称为 fruit 的list尾添加一个值为value的元素

client.LPush("fruit", "banana") //在名称为 fruit 的list头添加一个值为value的 元素

length, err := client.LLen("fruit").Result() //返回名称为 fruit 的list的长度

if err != nil {

panic(err)

}

fmt.Println("length: ", length) // 长度为2

value, err := client.LPop("fruit").Result() //返回并删除名称为 fruit 的list中的首元素

if err != nil {

panic(err)

}

fmt.Println("fruit: ", value)

value, err = client.RPop("fruit").Result() // 返回并删除名称为 fruit 的list中的尾元素

if err != nil {

panic(err)

}

fmt.Println("fruit: ", value)

}

// set 操作

func setOperation(client \*redis.Client) {

client.SAdd("blacklist", "Obama") // 向 blacklist 中添加元素

client.SAdd("blacklist", "Hillary") // 再次添加

client.SAdd("blacklist", "the Elder") // 添加新元素

client.SAdd("whitelist", "the Elder") // 向 whitelist 添加元素

// 判断元素是否在集合中

isMember, err := client.SIsMember("blacklist", "Bush").Result()

if err != nil {

panic(err)

}

fmt.Println("Is Bush in blacklist: ", isMember)

// 求交集, 即既在黑名单中, 又在白名单中的元素

names, err := client.SInter("blacklist", "whitelist").Result()

if err != nil {

panic(err)

}

// 获取到的元素是 "the Elder"

fmt.Println("Inter result: ", names)

// 获取指定集合的所有元素

all, err := client.SMembers("blacklist").Result()

if err != nil {

panic(err)

}

fmt.Println("All member: ", all)

}

// hash 操作

func hashOperation(client \*redis.Client) {

client.HSet("user\_xys", "name", "xys"); // 向名称为 user\_xys 的 hash 中添加元素 name

client.HSet("user\_xys", "age", "18"); // 向名称为 user\_xys 的 hash 中添加元素 age

// 批量地向名称为 user\_test 的 hash 中添加元素 name 和 age

client.HMSet("user\_test", map[string]string{"name": "test", "age":"20"})

// 批量获取名为 user\_test 的 hash 中的指定字段的值.

fields, err := client.HMGet("user\_test", "name", "age").Result()

if err != nil {

panic(err)

}

fmt.Println("fields in user\_test: ", fields)

// 获取名为 user\_xys 的 hash 中的字段个数

length, err := client.HLen("user\_xys").Result()

if err != nil {

panic(err)

}

fmt.Println("field count in user\_xys: ", length) // 字段个数为2

// 删除名为 user\_test 的 age 字段

client.HDel("user\_test", "age")

age, err := client.HGet("user\_test", "age").Result()

if err != nil {

fmt.Printf("Get user\_test age error: %v\n", err)

} else {

fmt.Println("user\_test age is: ", age) // 字段个数为2

}

}

// redis.v4 的连接池管理

func connectPool(client \*redis.Client) {

wg := sync.WaitGroup{}

wg.Add(10)

for i := 0; i < 10; i++ {

go func() {

defer wg.Done()

for j := 0; j < 100; j++ {

client.Set(fmt.Sprintf("name%d", j), fmt.Sprintf("xys%d", j), 0).Err()

client.Get(fmt.Sprintf("name%d", j)).Result()

}

fmt.Printf("PoolStats, TotalConns: %d, FreeConns: %d\n", client.PoolStats().TotalConns, client.PoolStats().FreeConns);

}()

}

wg.Wait()

}