第十章 Go语言操作Redis

11.2GO语言操作Redis

Redis是一个开源的，在内存中存储数据的缓存系统。可以用作数据库的缓存和消息代理。它支持的数据结构包括：字符串、哈希、列表、set集合、有序set集合、位图等类型。Redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

Redis支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得Redis可执行单层树复制。存盘可以有意无意的对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。

10.1安装redigo

获取redigo的方法是使用go命令直接下载即可：

go get github.com/garyburd/redigo/redis

获取到可用的库之后，我们尝试着开始连接redis服务器。

例10.1

package main

import (

"fmt"

"github.com/garyburd/redigo/redis" //此行代码文中后面的例子不会给出，请注意

)

func main() {

host := "127.0.0.1"

port := "6379"

protocol := "tcp"

redis, err := redis.Dial(protocol, host+":"+port)

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

fmt.Println("Connect to redis succeed")

defer redis.Close()

}

通过例10.1的代码，我们就实现了连接redis服务器的功能，连接成功之后，我们就可以进行一系列的基本操作。接下来我们会根据redigo提供的各种功能，详细的为读者介绍，如何与redis进行交互。

10.2Go语言Redis基本操作

使用Redigo操作数据的方式非常简单，命令格式如下：

Do(commandName string, args ...interface{}) (reply interface{}, err error)

参数：commandName是命令名称，包括“GET”，“SET“，“DEL”等

参数：args是一个字符串集，可以同时操作多个信息。

例10.2

func main() {

redigo, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

v, err := redigo.Do("SET", "color", "red")

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出ok

v, err = redis.String(redigo.Do("GET", "color"))

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出red

}

我们使用上面的例子了解到，对于字符串类型的数据，如何操作。我们使用do方法，使用“SET”命令，将数据保存到redis中，在下面使用”GET”命令，将存储的数据取出来

例10.3

func main() {

redigo, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

v, err := redigo.Do("SET", "color", "red")

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出ok

v, err = redigo.Do("DEL", "color")

if err != nil {

fmt.Println(err)

return

}

v, err = redis.String(redigo.Do("GET", "color"))

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出：redigo: nil returned

}

上面例子中我们使用了DEL命令将已经存入的数据删除后，再获取数据，得到的结果是nil returned，说明我们已经把刚才的值删除成功了。

例10.4

package main

import (

"fmt"

"github.com/garyburd/redigo/redis"

)

func main() {

redigo, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

redigo.Do("lpush", "redlist", "qqq")

redigo.Do("lpush", "redlist", "www")

redigo.Do("lpush", "redlist", "eee")

values, \_ := redis.Values(redigo.Do("lrange", "redlist", "0", "100"))

for \_, v := range values {

fmt.Println(string(v.([]byte)))

}

//输出：

// eee

// www

// qqq

}

列表的操作与普通字符串的操作一致，都是使用Do方法，由于使用的是“lpush”操作，所以我们在遍历的时候发现输出的顺序与我们写入的不一致。如果改成”rpush”，顺序就会一致了。读者可以自行尝试一下使用”rpush”操作查看一下效果。

例：10.5

func main() {

redigo, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

redigo.Do("SADD", "tt", "1")

redigo.Do("SADD", "tt", "2")

redigo.Do("SADD", "tt", "3")

redigo.Do("SADD", "tt", "1")

values, \_ := redis.Values(redigo.Do("SMEMBERS", "tt"))

for \_, v := range values {

fmt.Println(string(v.([]byte)))

}

//输出：

// 1

// 2.

// 3

}

对于set类型的数据，我们使用”SADD”命令可以将数据写入。

通过上面几个例子。我们看到redigo操作redis的方式与我们用命令行的方式几乎完全一样，读者在熟悉基本redis操作的情况下，可以非常容易上手。

在大量数据需要一次性操作完成的时候，可以使用Redis为我们提供的管道操作。管道操作可以将一组操作合并成一次操作，可以减少时间消耗。

我们先看一下基本的管道操作的例子。

例：10.6

func main() {

c, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

c.Send("SET", "foo", "bar")

c.Send("GET", "foo")

c.Flush()

s, err := c.Receive() // reply from SET

fmt.Println(s) //输出:OK

v, err := c.Receive() // reply from GET

fmt.Printf("%s", v) //输出：bar

}

上面的例子我们可以了解管道的基本用法，现在我们来比较下管理操作的速度差异。

我们先看一下不使用管道的写入操作。

例：10.7

func main() {

c, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

start := time.Now().UnixNano() //获取开始时间戳

for i := 0; i < 100000; i++ {

c.Do("SET", "foo"+string(i), "bar"+string(i))

}

end := time.Now().UnixNano() //获取结束时间戳

fmt.Println(end - start) //输出4896866986纳秒=4.896867秒

}

我们使用循环向redis写入100000条数据，不使用管道的情况下使用接近5秒的时间。

例10.8

func main() {

c, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

fmt.Println("Connect to redis error", err)

return

}

start := time.Now().UnixNano() //获取开始时间戳

for i := 0; i < 100000; i++ {

c.Send("SET", "foo"+string(i), "bar"+string(i))

}

end := time.Now().UnixNano() //获取结束时间戳

fmt.Println(end - start) //输出141727169纳秒=0.1417272秒

}

我们使用管道方式写入100000条数据，耗时仅仅0.14秒多一点点。时间效率之间的差异，由此可见一斑。

10.3Go语言操作Redis连接池管理

redigo提供了池来管理连接。为了避免每次操作redis时，建立连接，用完后再关闭，会造成大量的连接处于TIME\_WAIT状态，我们建议开发者在使用的时候，通过池来管理连接。

例10.9

var RedisClient \*redis.Pool

func init() {

// 建立连接池

RedisClient = &redis.Pool{

//最大空闲连接数

MaxIdle: 1,

//最大激活连接数

MaxActive: 10,

//最大的空闲连接等待时间，超过此时间后，空闲连接将被关闭

IdleTimeout: 180 \* time.Second,

Dial: func() (redis.Conn, error) {

c, err := redis.Dial("tcp", "127.0.0.1:6379")

if err != nil {

return nil, err

}

return c, nil

},

}

}

func main() {

// 从池里获取连接

rc := RedisClient.Get()

v, err := rc.Do("SET", "color", "red")

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出ok

v, err = redis.String(rc.Do("GET", "color"))

if err != nil {

fmt.Println(err)

return

}

fmt.Println(v) //输出red

// 用完后将连接放回连接池

defer rc.Close()

}