Go语言 数据操作 Redis

Redis 简介

redis是一个开源的、使用C语言编写的、支持网络交互的、可基于内存也可持久化，完全开源免费的，遵守BSD协议，是一个高性能的Key-Value数据库。

redis的官网地址，非常好记，是redis.io。

- Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，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string（字符串）、list（链表）、set（集合）、sorted set（有序集合）、hash表等数据结构的存储。
- Redis支持数据的备份，即master-slave模式的数据备份。
- 6379作为默认端口

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 ，单机能够达到15w qps，通常适合做缓存。

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，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。

丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis与其他key-value存储有什么不同?

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

Redis 键(key)

关于key，有几个点要提醒大家：

1. key不要太长，尽量不要超过1024字节，这不仅消耗内存，而且会降低查找的效率；
2. key也不要太短，太短的话，key的可读性会降低；
3. 在一个项目中，key最好使用统一的命名模式，例如user:baozi:passwd。

常用 Redis keys 命令

命令	描述	返回值
DEL KEY_NAME	删除已存在的键。不存在的 key 会被忽略	被删除 key 的数量
DUMP KEY_NAME	序列化给定 key ， 并返回被序列化的值	如果 key 不存在，那么返回 nil ， 否则，返回序列化之后的值
EXISTS KEY_NAME	检查给定 key 是否存在	若 key 存在返回 1 ， 否则返回 0
Expire KEY_NAME TIME_IN_SECONDS	设置 key 的过期时间，key 过期后将不再可用。单位以秒计	设置成功返回 1 。 当 key 不存在或者不能为 key 设置过期时间时返回 0
Expireat KEY_NAME TIME_IN_UNIX_TIMESTAMP	以 UNIX 时间戳(unix timestamp)格式设置 key 的过期时间。key 过期后将不再可用	设置成功返回 1 。 当 key 不存在或者不能为 key 设置过期时间时返回 0
KEYS PATTERN	查找所有符合给定模式 pattern 的 key	符合给定模式的 key 列表 (Array)
MOVE KEY_NAME DESTINATION_DATABASE	将当前数据库的 key 移动到给定的数据库 db 当中	移动成功返回 1 ， 失败则返回 0
PERSIST KEY_NAME	移除给定 key 的过期时间，使得 key 永不过期	当过期时间移除成功时，返回 1 。 如果 key 不存在或 key 没有设置过期时间，返回 0
RENAME OLD_KEY_NAME NEW_KEY_NAME	修改 key 的名称	改名成功时提示 OK ， 失败时候返回一个错误
SCAN cursor [MATCH pattern] [COUNT count]	迭代数据库中的数据库键	数组列表
TYPE KEY_NAME	返回 key 所储存的值的类型	返回 key 的数据类型

Redis 数据结构

redis是一种高级的key:value存储系统，其中value支持五种数据类型：

1. 字符串（strings）
2. 字符串列表（lists）
3. 字符串集合（sets）
4. 有序字符串集合（sorted sets）
5. 哈希（hashes）

字符串（strings）

有人说，如果只使用redis中的字符串类型，且不使用redis的持久化功能，那么，redis就和memcache非常非常的像了。这说明strings类型是一个很基础的数据类型，也是任何存储系统都必备的数据类型。

```
set mystr "hello world!" //设置字符串类型
get mystr //读取字符串类型
```

字符串类型的用法就是这么简单，因为是二进制安全的，所以你完全可以把一个图片文件的内容作为字符串来存储。

我们还可以通过字符串类型进行数值操作：

```
127.0.0.1:6379> set mynum "2"
OK
127.0.0.1:6379> get mynum
"2"
127.0.0.1:6379> incr mynum
(integer) 3
127.0.0.1:6379> get mynum
"3"
```

在遇到数值操作时，redis会将字符串类型转换成数值。

由于INCR等指令本身就具有原子操作的特性，所以我们完全可以利用redis的INCR、INCRBY、DECR、DECRBY等指令来实现原子计数的效果，假如，在某种场景下有3个客户端同时读取了mynum的值（值为2），然后对其同时进行了加1的操作，那么，最后mynum的值一定是5。不少网站都利用redis的这个特性来实现业务上的统计计数需求。

常用的字符串命令

命令	说明	返回值
SET KEY_NAME VALUE	设置给定 key 的值。如果 key 已经存储其他值， SET 就覆写旧值，且无视类型	SET 在设置操作成功完成时，才返回 OK 。
GET KEY_NAME	获取指定 key 的值。如果 key 不存在，返回 nil 。如果key 储存的值不是字符串类型，返回一个错误	返回 key 的值，如果 key 不存在时，返回 nil。 如果 key 不是字符串类型，那么返回一个错误
GETRANGE KEY_NAME start end	获取存储在指定 key 中字符串的子字符串。字符串的截取范围由 start 和 end 两个偏移量决定(包括 start 和 end 在内)	截取得到的子字符串
GETSET KEY_NAME VALUE	设置指定 key 的值，并返回 key 的旧值	返回给定 key 的旧值。 当 key 没有旧值时，即 key 不存在时，返回 nil ，当 key 存在但不是字符串类型时，返回一个错误
SETNX KEY_NAME VALUE	指定的 key 不存在时，为 key 设置指定的值	设置成功，返回 1 。设置失败，返回 0
MSET key1 value1 key2 value2 .. keyN valueN	同时设置一个或多个 key-value 对	总是返回 OK

命令	说明	返回值
MGET KEY1 KEY2 .. KEYN	返回所有(一个或多个)给定 key 的值。 如果给定的 key 里面，有某个 key 不存在，那么这个 key 返回特殊值 nil	一个包含所有给定 key 的值的列表
STRLEN KEY_NAME	获取指定 key 所储存的字符串值的长度。当 key 储存的不是字符串值时，返回一个错误	字符串值的长度。 当 key 不存在时，返回 0
INCR KEY_NAME	将 key 中储存的数字值增一，如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCR 操作，如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误，本操作的值限制在 64 位(bit)有符号数字表示之内	执行 INCR 命令之后 key 的值
INCRBY KEY_NAME INCR_AMOUNT	将 key 中储存的数字加上指定的增量值，如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCRBY 命令	加上指定的增量值之后， key 的值
DECR KEY_NAME	将 key 中储存的数字值减一，如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 DECR 操作	执行命令之后 key 的值
DECRBY KEY_NAME DECREMENT_AMOUNT	将 key 所储存的值减去指定的减量值，如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 DECRBY 操作	减去指定减量值之后， key 的值
APPEND KEY_NAME NEW_VALUE	为指定的 key 追加值，如果 key 已经存在并且是一个字符串， APPEND 命令将 value 追加到 key 原来的值的末尾，如果 key 不存在， APPEND 就简单地将给定 key 设为 value ，就像执行 SET key value 一样	追加指定值之后， key 中字符串的长度

列表（lists）

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）

一个列表最多可以包含 232 - 1 个元素 (4294967295, 每个列表超过40亿个元素)。

redis中的lists在底层实现上并不是数组，而是链表，也就是说对于一个具有上百万个元素的lists来说，在头部和尾部插入一个新元素，其时间复杂度是常数级别的，比如用LPUSH在10个元素的lists头部插入新元素，和在上千万元素的lists头部插入新元素的速度应该是相同的。

虽然lists有这样的优势，但同样有其弊端，那就是，链表型lists的元素定位会比较慢，而数组型lists的元素定位就会快得多。

常用的列表命令

命令	描述	返回值
LPUSH KEY_NAME VALUE1.. VALUEN	将一个或多个值插入到列表头部。 如果 key 不存在，一个空列表会被创建并执行 LPUSH 操作。 当 key 存在但不是列表类型时，返回一个错误	列表的长度
Lpop KEY_NAME	移除并返回列表的第一个元素	列表的第一个元素。 当key 不存在时，返回 nil
LRANGE KEY_NAME START END	返回列表中指定区间内的元素，区间以偏移量 START 和 END 指定	一个列表
RPUSH KEY_NAME VALUE1..VALUEN	将一个或多个值插入到列表的尾部(最右边)	列表的长度
RPOP KEY_NAME	移除列表的最后一个元素，返回值为移除的元素	被移除的元素
LLEN KEY_NAME	返回列表的长度。 如果列表 key 不存在，则 key 被解释为一个空列表，返回 0 。 如果 key 不是列表类型，返回一个错误	列表的长度
LINDEX KEY_NAME INDEX_POSITION	通过索引获取列表中的元素，可以使用负数下标	下标为指定索引值的元素
LINSERT key BEFORE|AFTER pivot value	在列表的元素前或者后插入元素	列表的长度

集合（set）

Redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

集合对象的编码可以是 intset 或者 hashtable。

Redis 中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 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

常用的集合命令

命令	描述	返回值
SADD KEY_NAME VALUE1..VALUEN	将一个或多个成员元素加入到集合中，已经存在于集合的成员元素将被忽略	被添加到集合中的新元素的数量，不包括被忽略的元素
SCARD KEY_NAME	返回集合中元素的数量	集合的数量
SDIFF FIRST_KEY OTHER_KEY1..OTHER_KEYN	第一个集合与其他集合之间的差异	包含差集成员的列表
SDIFFSTORE DESTINATION_KEY KEY1..KEYN	将给定集合之间的差集存储在指定的集合中。如果指定的集合 key 已存在，则会被覆盖	结果集中的元素数量
SINTER KEY KEY1..KEYN	返回给定所有给定集合的交集	交集成员的列表
SISMEMBER KEY VALUE	判断成员元素是否是集合的成员	是集合的成员返回 1 。不是集合的成员或 key 不存在返回 0
SMEMBERS key	返回集合中的所有的成员	集合中的所有的成员
SMOVE SOURCE DESTINATION MEMBER	将指定成员 member 元素从 source 集合移动到 destination 集合	成功移除，返回 1 ，否则 0
SREM KEY MEMBER1..MEMBERN	移除集合中的一个或多个成员元素，不存在的成员元素会被忽略	成功移除的元素的数量
SUNION KEY KEY1..KEYN	返回给定集合的并集	并集成员的列表
SUNIONSTORE destination key [key ...]	将给定集合的并集存储在指定的集合 destination 中。如果 destination 已经存在，则将其覆盖	结果集中的元素数量
SSCAN key cursor [MATCH pattern] [COUNT count]	用于迭代集合中键的元素	数组列表

有序集合（sorted set）

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。有序集合中的每个元素都关联一个序号（score），这便是排序的依据。

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 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

很多时候，我们都将redis中的有序集合叫做zsets。

常用的有序集合命令

命令	描述	返回值
ZADD KEY_NAME SCORE1 VALUE1.. SCOREN VALUEN	将一个或多个成员元素及其分数值加入到有序集当中	被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员

命令	描述	返回值
ZCARD KEY_NAME	用于计算集合中元素的数量	集合中元素的数量
ZCOUNT key min max	用于计算有序集合中指定分数区间的成员数量	分数值在 min 和 max 之间的成员的数量
ZINCRBY key increment member	对有序集合中指定成员的分数加上增量 increment	member 成员的新分数值，以字符串形式表示
ZRANGE key start stop [WITHSCORES]	返回有序集中，指定区间内的成员	带有分数值(可选)的有序集成员的列表
ZRANK key member	返回有序集中指定成员的排名	返回 member 的排名
ZREM key member [member ...]	移除有序集中的一个或多个成员，不存在的成员将被忽略	被成功移除的成员的数量，不包括被忽略的成员
ZSCORE key member	返回有序集中，成员的分数值	成员的分数值，以字符串形式表示
ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]	计算给定的一个或多个有序集的交集，其中给定 key 的数量必须以 numkeys 参数指定，并将该交集(结果集)储存到 destination 。	保存到目标结果集的的成员数量
ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]	计算给定的一个或多个有序集的并集，其中给定 key 的数量必须以 numkeys 参数指定，并将该并集(结果集)储存到 destination 。	保存到 destination 的结果集的成员数量
ZSCAN key cursor [MATCH pattern] [COUNT count]	用于迭代有序集合中的元素（包括元素成员和元素分值）	返回的每个元素都是一个有序集合元素，一个有序集合元素由一个成员（member）和一个分值（score）组成

哈希（hashes）

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

Redis 中每个 hash 可以存储 232 - 1 键值对（40多亿）。

常用的哈希命令

命令	描述	返回值
HDEL KEY_NAME FIELD1.. FIELDN	删除哈希表 key 中的一个或多个指定字段，不存在的字段将被忽略	被成功删除字段的数量
HEXISTS KEY_NAME FIELD_NAME	查看哈希表的指定字段是否存在	哈希表含有返回 1 ，不含有或 key 不存在返回 0
HGET KEY_NAME FIELD_NAME	返回哈希表中指定字段的值	给定字段的值
HGETALL KEY_NAME	返回哈希表中，所有的字段和值	以列表形式返回哈希表的字段及字段值
HKEYS key	获取哈希表中的所有域（field）	包含哈希表中所有域（field）列表
HLEN KEY_NAME	获取哈希表中字段的数量	哈希表中字段的数量
HMGET KEY_NAME FIELD1...FIELDN	返回哈希表中，一个或多个给定字段的值	一个包含多个给定字段关联值的表
HMSET KEY_NAME FIELD1 VALUE1 ...FIELDN VALUEN	同时将多个 field-value (字段-值)对设置到哈希表中	执行成功返回 OK
HSET KEY_NAME FIELD VALUE	为哈希表中的字段赋值	如果字段是哈希表中的一个新建字段，并且值设置成功返回 1 ， 如果字段已经存在且旧值已被新值覆盖返回 0
HSETNX KEY_NAME FIELD VALUE	为哈希表中不存在的的字段赋值	设置成功返回 1 ，如果给定字段已经存在且没有操作被执行返回 0
HVALS key	获取哈希表所有的值	一个包含哈希表中所有值的列表
HSCAN key cursor [MATCH pattern] [COUNT count]	迭代哈希表中的键值对	返回的每个元素都是一个元组，每一个元组元素由一个字段(field) 和值（value）组成

Redis 使用 go-redis库

go-redis 支持连接哨兵及集群模式的Redis

使用第三方开源的redis库: https://github.com/go-redis/redis

```
go get -u github.com/go-redis/redis
```

链接Redis

```
package main

import (
    "fmt"
    "github.com/go-redis/redis"
)

// redis

// 定义一个全局变量
var redisdb *redis.Client

func initRedis()(err error){
    redisdb = redis.NewClient(&redis.Options{
        Addr: "127.0.0.1:6379",  // 指定
        Password: "",
        DB:0,       // redis一共16个库，指定其中一个库即可
    })
    _,err = redisdb.Ping().Result()
    return
}

func main() {
    err := initRedis()
    if err != nil {
        fmt.Printf("connect redis failed! err : %v\n",err)
        return
    }
    fmt.Println("redis连接成功！")
}
```

基本的使用

```
package main

import (
    "fmt"
    "time"

    "github.com/go-redis/redis"
)

// redis

// 定义一个全局变量
var redisdb *redis.Client

func initRedis() (err error) {
```

```
    redisdb = redis.NewClient(&redis.Options{
        Addr:     "127.0.0.1:6379", // 指定
        Password: "",
        DB:       0, // redis一共16个库，指定其中一个库即可
    })
    _, err = redisdb.Ping().Result()
    return
}

func main() {
    err := initRedis()
    if err != nil {
        fmt.Printf("connect redis failed! err : %v\n", err)
        return
    }
    defer redisdb.Close()
    fmt.Println("redis连接成功！")

    // 存取普通string类型，10分钟过期
    redisdb.Set("test_name", "baozi", time.Minute*10)
    sc := redisdb.Get("test_name")
    fmt.Printf("sc: %v\n", sc)

    // 存取hash数据
    redisdb.HSet("test_class", "521", 42)
    sc2 := redisdb.HGetAll("test_class")
    fmt.Printf("sc2: %v\n", sc2)

    // 存取list数据
    redisdb.RPush("test_list", 1) // 向右边添加元素
    redisdb.LPush("test_list", 2) // 向左边添加元素
    ssc := redisdb.LRange("test_list", 0, -1)
    fmt.Printf("ssc: %v\n", ssc)

    // 存取set数据
    redisdb.SAdd("test_set", "apple")
    redisdb.SAdd("test_set", "pear")
    ssc2 := redisdb.SMembers("test_set")
    fmt.Printf("ssc2: %v\n", ssc2)

    // zset 存取
    key := "test_zset"
    items := []redis.Z{
        redis.Z{Score: 90, Member: "PHP"},
        redis.Z{Score: 93, Member: "JAVA"},
        redis.Z{Score: 96, Member: "C++"},
        redis.Z{Score: 92, Member: "PYTHON"},
        redis.Z{Score: 100, Member: "GOLANG"},
    }
    redisdb.ZAdd(key, items...)
    ssc3 := redisdb.ZRevRange(key, 0, -1)
    fmt.Printf("ssc3: %v\n", ssc3)
}
```

Redis 使用 redis client库：redigo

redigo 三方库只有一个 Do 函数执行 Redis 命令，更接近使用 redis-cli 操作 Redis。

使用第三方开源的redis库: github.com/garyburd/redigo/redis

```
go get github.com/garyburd/redigo/redis
```

链接Redis

```
package main

import (
    "log"

    "github.com/garyburd/redigo/redis"
)

// redis客户端连接
func redisConnect() redis.Conn {
    c, err := redis.Dial("tcp", "127.0.0.1:6379")
    if err != nil {
        log.Fatal(err)
    }
    return c
}
```

基本的使用

```
package main

import (
    "fmt"
    "log"
    "time"

    "github.com/garyburd/redigo/redis"
)

// redis客户端连接
func redisConnect() redis.Conn {
    c, err := redis.Dial("tcp", "127.0.0.1:6379")
    if err != nil {
        log.Fatal(err)
    }
    return c
}

func main() {
    // 链接客户端
    c := redisConnect()
    defer c.Close()

    // string类型数据命令
    // set值
```

```
    c.Do("set", "id", 1001)
    // get值
    res, _ := redis.Int(c.Do("get", "id"))
    fmt.Printf("res: %v\n", res)

    // hash操作：hset+hget
    c.Do("hset", "stu1", "name", "lena")
    c.Do("hset", "stu1", "age", 10)
    name, _ := redis.String(c.Do("hget", "stu1", "name"))
    fmt.Println("name =", name)
    age, _ := redis.Int(c.Do("hget", "stu1", "age"))
    fmt.Println("age =", age)

    // hmset
    c.Do("hmset", "stu2", "name", "lena", "age", 20, "school", "职业学院")
    // hmget：用strings接收多个结果
    r, _ := redis.Strings(c.Do("hmget", "stu2", "name", "school", "age"))
    fmt.Printf("type = %T\n", r) // type = []string
    for _, value := range r {
        fmt.Println(value)
    }

    // list操作
    c.Do("lpush", "list", 1, 2, 3, 4, 5)
    // lpop
    res1, _ := redis.Int(c.Do("lpop", "list"))
    fmt.Println("lpop =", res1) // lpop = 5
    // rpush
    c.Do("rpush", "list", 10, 20)
    // rpop
    res1, _ = redis.Int(c.Do("rpop", "list"))
    fmt.Println("rpop =", res1) // rpop = 20

    // 设置有效时间
    // set值
    c.Do("set", "id", 1002)
    // 设置过期时间
    c.Do("expire", "id", 5)
    // get值
    res2, _ := redis.Int(c.Do("get", "id"))
    fmt.Println("data =", res2) // data = 1002
    time.Sleep(time.Second * 5)

    res2, _ = redis.Int(c.Do("get", "id"))
    fmt.Println("data =", res2)
}
```

Redis连接池

```
package main

import (
    "fmt"

    "github.com/garyburd/redigo/redis"
```

```
)

var pool *redis.Pool //创建redis连接池

func init() {
    pool = &redis.Pool{ //实例化一个连接池
        MaxIdle: 16, //最初的连接数量
        // MaxActive:1000000,    //最大连接数量
        MaxActive:   0,   //连接池最大连接数量,不确定可以用0（0表示自动定义），按需分配
        IdleTimeout: 300, //连接关闭时间 300秒 （300秒不使用自动关闭）
        Dial: func() (redis.Conn, error) { //要连接的redis数据库
            return redis.Dial("tcp", "localhost:6379")
        },
    }
}

func main() {
    c := pool.Get() //从连接池，取一个链接
    defer c.Close() //函数运行结束 ，把连接放回连接池

    _, err := c.Do("Set", "abc", 200)
    if err != nil {
        fmt.Println(err)
        return
    }

    r, err := redis.Int(c.Do("Get", "abc"))
    if err != nil {
        fmt.Println("get abc faild :", err)
        return
    }
    fmt.Println(r)
    pool.Close() //关闭连接池
}
```