第十一章 Go语言操作缓存

11.1 Memcached

Memcached是一个自由开源的，高性能，分布式内存对象缓存系统。

Memcached是以LiveJournal旗下Danga Interactive公司的Brad Fitzpatric为首开发的一款软件。现在已成为mixi、hatena、Facebook、Vox、LiveJournal等众多服务中提高Web应用扩展性的重要因素。

Memcached是一种基于内存的key-value存储，用来存储小块的任意数据（字符串、对象）。这些数据可以是数据库调用、API调用或者是页面渲染的结果。

Memcached简洁而强大。它的简洁设计便于快速开发，减轻开发难度，解决了大数据量缓存的很多问题。它的API兼容大部分流行的开发语言。

本质上，它是一个简洁的key-value存储系统。

一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度、提高可扩展性。

memcached作为高速运行的分布式缓存服务器，具有以下的特点。

* 协议简单
* 基于libevent的事件处理
* 内置内存存储方式
* memcached不互相通信的分布式

11.1.2安装MemCached驱动

获取memcached第三方库的方法是使用go命令直接下载即可：

go get github.com/bradfitz/gomemcache/memcache

获取到可用的库之后，我们尝试着开始连接memcached服务器。

例11.1

package main

import (

"fmt"

"github.com/bradfitz/gomemcache/memcache" //此行代码文中后面的例子不会给出，请注意

)

func main() {

mc := memcache.New("127.0.0.1:11211")

if mc == nil

fmt.Println("memcache New failed")

return

}

fmt.Println("memcache New success")

}

通过例11.1的代码，我们就实现了连接memcached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。通过测试我们发现，memcache.New方法的返回值是一个\*Clien结构体，所以不会存在返回值为nil的情况。我们也无法根据返回的内容进行判断连接是否成功，只有在具体操作的时候，才会返回连接超时的信息，所以此处的判断代码只有象征意义，并不能说明连接是否成功。

11.1.3Go语言操作memcached

11.1.3.1Go语言Memcache库set操作

使用set操作命令格式如下：

func (c \*Client) Set(item \*Item) error

参数：item是一个item结构体

item结构体的结构如下：

type Item struct {

// Item的key(最大250字节).

Key [string](https://godoc.org/builtin#string)

// 需要存储的值

Value [][byte](https://godoc.org/builtin#byte)

//服务标识位，取决于应用的不同。.

Flags [uint32](https://godoc.org/builtin#uint32)

// 缓存过期时间。（单位：秒）

//0代表永不过期

Expiration [int32](https://godoc.org/builtin#int32)

// contains filtered or unexported fields

}

例11.2

func main() {

mc := memcache.New("127.0.0.1:11211")

err := mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value")})

if err != nil {

fmt.Println(err)

return

}

err = mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value1")})

fmt.Println("set is success")

}

通过11.2的例子我们需要注意两个细节。

1.value是使用[]byte存储的，所以在获取数据的时候，需要使用类型转换将[]byte转换成string才可以正常使用

2.set操作是强制覆盖的，重名情况下不会有任何提示，直接将内容重新写入。

11.1.3.2 Go语言Memcached库get操作

使用get操作命令格式如下：

func (c \*[Client](https://godoc.org/github.com/bradfitz/gomemcache/memcache#Client)) Get(key [string](https://godoc.org/builtin#string)) (item \*[Item](https://godoc.org/github.com/bradfitz/gomemcache/memcache#Item), err [error](https://godoc.org/builtin#error))

通过key获取item。如果获取的内容不存在，返回” memcache: cache miss”。key的长度不能大于250位。

例11.3

func main() {

mc := memcache.New("127.0.0.1:11211")

err := mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value")})

if err != nil {

fmt.Println(err)

return

}

err = mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value1")})

it, err := mc.Get("foo")

if err != nil {

fmt.Println(err)

return

}

fmt.Println(string(it.Value)) //my value1

\_, err = mc.Get("t")

fmt.Println(err) //memcache: cache miss

}

通过上例，验证了11.2中同键名情况下，值会被直接覆盖。Get取到的数据是[]byte类型，需要使用string()进行类型转换。

11.1.3.3 Go语言Memcached库add操作

使用add操作命令格式如下：

func (c \*[Client](https://godoc.org/github.com/bradfitz/gomemcache/memcache#Client)) Add(item \*[Item](https://godoc.org/github.com/bradfitz/gomemcache/memcache#Item)) [error](https://godoc.org/builtin#error)

add用于将value存储在指定的key中，如果add的key已经存在，则不会更新数据（过期的key会更新），之前的值不变，并且获得响应NOT\_STORED

例11.4

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

if mc == nil {

fmt.Println("memcache New failed")

return

}

mc.Add(&memcache.Item{Key: "foo3", Value: []byte("bluegogo1")})

it, err := mc.Get("foo3")

if err != nil {

fmt.Println("Add failed")

} else {

if string(it.Key) == "foo3" {

fmt.Println("Add value is ", string(it.Value))// Add value is bluegogo1

} else {

fmt.Println("Get failed")

}

}

}

11.1.3.4 Go语言Memcached库 replace操作

使用replace操作命令格式如下：

func (c \*[Client](https://godoc.org/github.com/bradfitz/gomemcache/memcache#Client)) Replace(item \*[Item](https://godoc.org/github.com/bradfitz/gomemcache/memcache#Item)) [error](https://godoc.org/builtin#error)

replace 命令用于替换已存在的 key(键) 的 value(数据值)。

如果 key 不存在，则替换失败，并且您将获得响应 NOT\_STORED。

例11.5

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

if mc == nil {

fmt.Println("memcache New failed")

return

}

mc.Set(&memcache.Item{Key: "foo2", Value: []byte("my value1")})

mc.Replace(&memcache.Item{Key: "foo2", Value: []byte("mobike")})

it, err := mc.Get("foo2")

if err != nil {

fmt.Println("foo2 failed")

} else {

if string(it.Key) == "foo2" {

fmt.Println("foo2 value is ", string(it.Value)) //foo2 value is mobike

} else {

fmt.Println("Get failed")

}

}

}

11.1.3.5 Go语言Memcached库 delete操作

使用delete操作命令格式如下：

func (c \*Client) Delete(key string) error

delete 命令用于删除已存在的 key(键)。

例11.6

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

if mc == nil {

fmt.Println("memcache New failed")

return

}

errd := mc.Delete("foo")

if errd != nil {

fmt.Println("Delete failed:", errd.Error())

}

\_, err := mc.Get("foo")

if err != nil {

fmt.Println("delete success")

}

}

11.1.3.6 Go语言Memcached库incrby操作

使用incrby操作命令格式如下：

func (c \*[Client](https://godoc.org/github.com/bradfitz/gomemcache/memcache#Client)) Increment(key [string](https://godoc.org/builtin#string), delta [uint64](https://godoc.org/builtin#uint64)) (newValue [uint64](https://godoc.org/builtin#uint64), err [error](https://godoc.org/builtin#error))

incr命令用于对已存在的 key(键) 的数字值进行自增或自减操作。

incr 命令操作的数据必须是十进制的32位无符号整数。

如果 key 不存在返回 NOT\_FOUND，如果键的值不为数字，则返回 CLIENT\_ERROR，其他错误返回 ERROR。

例11.7

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

err := mc.Set(&memcache.Item{Key: "aaa", Value: []byte("1")})

if err != nil {

fmt.Println("Set failed :", err.Error())

}

it, err := mc.Get("foo")

if err != nil {

fmt.Println("Get failed ", err.Error())

} else {

fmt.Println("src value is:", it.Value)

}

value, err := mc.Increment("aaa", 7)

if err != nil {

fmt.Println("Increment failed")

} else {

fmt.Println("after increment the value is :", value)

}

}

11.1.3.7 Go语言Memcached库decrby操作

使用decrby操作命令格式如下：

func (c \*Client) Decrement(key string, delta uint64) (newValue uint64, err error)

decr 命令用于对已存在的 key(键) 的数字值进行自增或自减操作。

decr 命令操作的数据必须是十进制的32位无符号整数。

如果 key 不存在返回 NOT\_FOUND，如果键的值不为数字，则返回 CLIENT\_ERROR，其他错误返回 ERROR。

例11.8

func main() {

// mc := memcache.New("192.1.1.1:11211")

mc := memcache.New("127.0.0.1:11211")

value, err := mc.Decrement("aaa", 4)

if err != nil {

fmt.Println("Decrement failed", err.Error())

} else {

fmt.Println("after decrement the value is ", value)

}

}

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。

11.2.1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。

11.2.2Go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。

11.2.3Go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()

}

11.3Go语言操groupcache

11.3.1groupcache简介

groupcache 是使用 Go 语言编写的缓存及缓存过滤库，作为 memcached 许多场景下的替代版本。

groupcache 与 memcached 的相似之处：通过 key 分片，并且通过 key 来查询响应的 peer。

groupcache 与 memcached 的不同之处：

1. groupcache 不需要对服务器进行单独的设置，这将大幅度减少部署和配置的工作量。groupcache 既是客户端库也是服务器库，并连接到自己的 peer 上。

2. groupcache 具有缓存过滤机制。众所周知，在 memcached 出现“Sorry，cache miss（缓存丢失）”时，经常会因为不受控制用户数量的请求而导致数据库（或者其它组件）产生“惊群效应（thundering herd）”；groupcache 会协调缓存填充，只会将重复调用中的一个放于缓存，而处理结果将发送给所有相同的调用者。

3. groupcache 不支持多个版本的值。如果“foo”键对应的值是“bar”，那么键“foo”的值永远都是“bar”。这里既没有缓存的有效期，也没有明确的缓存回收机制，因此同样也没有 CAS 或者 Increment/Decrement。

4. 基于上一点的改变，groupcache 就具备了自动备份“超热”项进行多重处理，这就避免了 memcached 中对某些键值过量访问而造成所在机器 CPU 或者 NIC 过载。

5. groupcache 当下只支持Go

11.3.2groupcache运行机制

简而言之，groupcache查找一个Get（“foo”）的过程类似下面的情景（机器#5上，它是运行相同代码N台机器集中的一台）：

1. key“foo”的值是否会因为“过热”而储存在本地内存，如果是，就直接使用

2. key“foo”的值是否会因为peer #5是其拥有者而储存在本地内存，如果是，就直接使用

3. 首先确定key “fool”是否归属自己N个机器集合的peer中，如果是，就直接加载。如果有其它的调用者介入（通过相同的进程或者是peer的RPC请求，这些请求将会被阻塞，而处理结束后，他们将直接获得相同的结果）。如果不是，将key的所有者RPC到响应的peer。如果RPC失败，那么直接在本地加载（仍然通过备份来应对负载）。

目前groupcache已经在dl.Google.com、Blogger、Google Code、Google Fiber、Google生产监视系统等项目中投入使用。

11.3.3使用groupcache构建服务端

例：groupcache

package main

import (

"fmt"

"io/ioutil"

"log"

"net/http"

"os"

"github.com/golang/groupcache"

)

var (

peers\_addrs = []string{"http://127.0.0.1:8001", "http://127.0.0.1:8002", "http://127.0.0.1:8003"}

)

func main() {

// 验证启动参数

if len(os.Args) <= 1 {

fmt.Println("invalid hosts")

os.Exit(1)

}

local\_addr := os.Args[1]

// 创建HTTP池

peers := groupcache.NewHTTPPool("http://" + local\_addr)

peers.Set(peers\_addrs...)

// 创建Group

var image\_cache = groupcache.NewGroup("cache", 8<<30, groupcache.GetterFunc(

// 无法命中缓存的处理方式

func(ctx groupcache.Context, key string, dest groupcache.Sink) error {

// 从文件中读取内容

result, err := ioutil.ReadFile(key)

if err != nil {

fmt.Printf("read file error %s.\n", err.Error())

message := "{'status':'0','msg':'" + err.Error() + "'}"

dest.SetBytes([]byte(message))

return nil

}

fmt.Printf("asking for %s from local file system\n", key)

// 设置缓存内容

dest.SetBytes([]byte(result))

return nil

}))

// 创建请求接口

http.HandleFunc("/cache", func(rw http.ResponseWriter, r \*http.Request) {

// 响应cache请求

var data []byte

// 获取参数

k := r.URL.Query().Get("id")

fmt.Printf("user get %s from groupcache\n", k)

// 根据key获取缓存信息

image\_cache.Get(nil, k, groupcache.AllocatingByteSliceSink(&data))

// 提供返回值

rw.Write([]byte(data))

})

log.Fatal(http.ListenAndServe(local\_addr, nil))

}

以上代码在命令行中执行以后，可以使用浏览器通过http://127.0.0.1:8001/cache访问

使用：http://127.0.0.1:8001/image?id=1.ini

第一次请求会看到控制台输出user get 1.ini from groupcache，同时会输出asking for 1.ini from local file system。由于是第一次访问，缓存中没有装载数据，所以会到本地去寻找要加载的文件，将文件内容装载到缓存中，显示到页面中。

第二次请求访问http://127.0.0.1:8001/image?id=1.ini1

控制台输出read file error open 1.ini1: no such file or directory.提示没有找到1.ini1这个文件。大家在测试代码的时候，需要提前创建文件，以防止无法找到。

对于1.ini由于已经存在了。在后续的多次访问中，都会显示从groupcache中获取。由于groupcache设计的时候就只允许添加，所以我们目前可以考虑将配置类信息，和持续不变的信息放到groupcache中。替代部分Memcached的功能。本书仅给出一个简单的示例，具体的使用还需要读者在实际工作中去寻找合适的场景。

