缓存组件的实现

一、实验介绍

1.1 实验内容

这一节实验中，我们将设计并使用 golang 语言实现一个缓存组件。

1.2 实验知识点

- 缓存的意义和缓存的设计
- Go 语言 map 实现哈希方法

1.3 实验环境

- Go 1.2.1
- Xfce终端

1.4 适合人群

本课程属于中级级别课程，适合具有Go基础的用户。

1.5 代码获取

该实验的所有实验代码可通过在终端中输入入如下代码来获取

```
$ wget http://labfile.oss-cn-hangzhou.aliyuncs.com/courses/504/cache.tgz
```

二、实验原理

2.1 缓存

缓存（Cache）在计算机硬件中普遍存在。比如在 CPU 中就有一级缓存，二级缓存，甚至三级缓存。缓存的工作原理一般是 CPU 需要读取数据时，会首先从缓存中查找需要的数据，如果找到了就直接进行处理，如果没有找到则从内存中读取数据。由于 CPU 中的缓存工作速度比内存还要快，所以缓存的使用能加快 CPU 处理速度。缓存不仅仅存在于硬件中，在各种软件系统中也处处可见。比如在 Web 系统中，缓存存在于服务器端，客户端或者代理服务器中。广泛使用的 CDN 网络，也可以看作一个巨大的缓存系统。缓存在 Web 系统中的使用有很多好处，不仅可以减少网络流量，降低客户访问延迟，还可以减轻服务器负载。

目前已经存在很多高性能的缓存系统，比如 Memcache (http://memcached.org/)，Redis (http://redis.io/) 等，尤其是 Redis，现已经广泛用于各种 Web 服务中。既然有了这些功能完善的缓存系统，那为什么我们还需要自己实现一个缓存系统呢？这么做主要有两个原因，第一，通过动手实现我们可以了解缓存系统的工作原理，这也是老掉牙的理由了。第二，Redis 之类的缓存系统都是独立存在的，如果只是开发一个简单的应用还需要单独使用 Redis 服务器，难免会过于复杂。这时候如果有一个功能完善的软件包实现了这些功能，只需要引入这个软件包就能实现缓存功能，而不需要单独使用 Redis 服务器，就最好不过了。

2.2缓存系统的设计

缓存系统中，缓存的数据一般都存储在内存中，所以我们设计的缓存系统要以某一种方式管理内存中的数据。既然缓存数据是存储在内存中的，那如果系统停机，那数据岂不就丢失了吗？其实一般情况下，缓存系统还支持将内存中的数据写入到文件中，在系统再次启动时，再将文件中的数据加载到内存中，这样一来就算系统停机，缓存数据也不会丢失。

同时缓存系统还提供过期数据清理机制，也就是说缓存的数据项是有生存时间的，如果数据项过期，则数据项会从内存中被删除，这样一来热数据会一直存在，而冷数据则会被删除，也没有必要进行缓存。

缓存系统还需要对外提供操作的接口，这样系统的其他组件才能使用缓存。一般情况下，缓存系统需要支持 CRUD 操作，也就算创建（添加），读取，更新和删除操作。

通过以上分析，可以总结出缓存系统需要有以下功能：

- 缓存数据的存储
- 过期数据项管理
- 内存数据导出，导入
- 提供 CRUD 借口

三、开发准备

首先，需要创建工作目录，同时设置 GOPATH 环境变量：

```
$ cd /home/shiyanlou/
$ mkdir -p golang/src
$ cd golang
$ export GOPATH='/home/shiyanlou/golang'
```

以上步骤中，我们创建了 /home/shiyanlou/golang 目录，并将它设置为 GOPATH，也是后面实验的工作目录。

四、实验步骤

4.1 缓存系统的实现

缓存数据需要存储在内存中，这样才可以被快速访问。那使用什么数据结构来存储数据项呢？一般情况下，我们使用哈希表来存储数据项，这样访问数据项将获得更好的性能。在 golang 语言中，我们不用自己实现哈希表，因为内建类型 map 已经实现了哈希表，所以我们可以直接将缓存数据项存储在 map 中。同时由于缓存系统支持过期数据清理，所以缓存数据项应该带有生存时间，这说明需要将缓存数据进行封装后，保存到缓存系统中。这样我们就需要先实现缓存数据项，其实现的代码如下：

```
type Item struct {
    Object     interface{} // 真正的数据项
    Expiration int64       // 生存时间
}

// 判断数据项是否已经过期
func (item Item) Expired() bool {
    if item.Expiration == 0 {
        return false
    }
    return time.Now().UnixNano() > item.Expiration
}
```

以上代码中，我们定义了一个 Item 结构，该结构包含两个字段，其中 Object 用于存储任意类型的数据对象，而 Expiration 字段则存储了该数据项的过期时间，同时我们为 Item 类型，提供了 Expired() 方法，该方法返回布尔值表示该数据项是否已经过期。需要注意的是，数据项的过期时间，是 Unix 时间戳，单位是纳秒。怎么样判断数据项有没有过期呢？其实非常简单。我们在每一个数据项中，记录数据项的过期时间，然后缓存系统将定期检查每一项数据项，如果发现数据项的过期时间小于当前时间，则将数据项从缓存系统中删除。这里我们将借助 time (https://godoc.org/time) 模块来实现周期任务。到这里，我们可以实现缓存系统的框架了，见一下代码：

```
const (
    // 没有过期时间标志
    NoExpiration time.Duration = -1

    // 默认的过期时间
    DefaultExpiration time.Duration = 0
)

type Cache struct {
    defaultExpiration time.Duration
    items             map[string]Item // 缓存数据项存储在 map 中
    mu                sync.RWMutex // 读写锁
    gcInterval        time.Duration   // 过期数据项清理周期
    stopGc            chan bool
}

// 过期缓存数据项清理
func (c *Cache) gcLoop() {
    ticker := time.NewTicker(c.gcInterval)
    for {
        select {
        case <-ticker.C:
            c.DeleteExpired()
        case <-c.stopGc:
            ticker.Stop()
            return
        }
    }
}
```


以上代码中，实现了 Cache 结构，该结构也就是缓存系统结构，其中 items 是一个 map，用于存储缓存数据项。同时可以看到，我们实现了 gcLoop() 方法，该方通过 time.Ticker 定期执行 DeleteExpired() 方法，从而清理过期的数据项。通过 time.NewTicker() 方法创建的 **ticker,** 会通过指定的 c.Interval 间隔时间，周期性的从 ticker.C 管道中发送数据过来，我们可以根据这一特性周期性的执行 DeleteExpired() 方法。同时为使 gcLoop() 函数能正常结束，我们通过监听 c.stopGc 管道，如果有数据从该管道中发送过来，我们就停止 gcLoop() 的运行。同时注意到，我们定义了 NoExpiration 和 DefaultExpiration 常量，前者代表数据项永远不过期，后者标记数据项应该拥有一个默认过期时间。 DeleteExpired() 该怎样实现呢？见如下代码：

```
/ 删除缓存数据项
func (c *Cache) delete(k string) {
    delete(c.items, k)
}

// 删除过期数据项
func (c *Cache) DeleteExpired() {
    now := time.Now().UnixNano()
    c.mu.Lock()
    defer c.mu.Unlock()

    for k, v := range c.items {
        if v.Expiration > 0 && now > v.Expiration {
            c.delete(k)
        }
    }
}
```

可以看到 DeleteExpired() 方法非常简单，只需要遍历所有数据项，删除过期数据即可。

现在，我们可以实现缓存系统的 **CRUD** 接口了。我们可以通过以下接口将数据添加到缓存系统中：

```
// 设置缓存数据项，如果数据项存在则覆盖
func (c *Cache) Set(k string, v interface{}, d time.Duration) {
    var e int64
    if d == DefaultExpiration {
        d = c.defaultExpiration
    }
    if d > 0 {
        e = time.Now().Add(d).UnixNano()
    }
    c.mu.Lock()
    defer c.mu.Unlock()
    c.items[k] = Item{
        Object:     v,
        Expiration: e,
    }
}

// 设置数据项，没有锁操作
func (c *Cache) set(k string, v interface{}, d time.Duration) {
    var e int64
    if d == DefaultExpiration {
        d = c.defaultExpiration
    }
    if d > 0 {
        e = time.Now().Add(d).UnixNano()
    }
    c.items[k] = Item{
        Object:     v,
        Expiration: e,
    }
}

// 获取数据项，如果找到数据项，还需要判断数据项是否已经过期
func (c *Cache) get(k string) (interface{}, bool) {
    item, found := c.items[k]
    if !found {
        return nil, false
    }
    if item.Expired() {
        return nil, false
    }
    return item.Object, true
}

// 添加数据项，如果数据项已经存在，则返回错误
func (c *Cache) Add(k string, v interface{}, d time.Duration) error {
    c.mu.Lock()
    _, found := c.get(k)
    if found {
        c.mu.Unlock()
        return fmt.Errorf("Item %s already exists", k)
    }
    c.set(k, v, d)
    c.mu.Unlock()
    return nil
}

// 获取数据项
func (c *Cache) Get(k string) (interface{}, bool) {
    c.mu.RLock()
    item, found := c.items[k]
    if !found {
        c.mu.RUnlock()
        return nil, false
    }
    if item.Expired() {
        return nil, false
    }
    c.mu.RUnlock()
    return item.Object, true
}
```


以上代码中，我们主要实现了 Set() 和 Add() 接口，两者的主要区别是，前者将数据添加到缓存系统中时，如果数据项已经存在则会覆盖，而后者如果发现数据项已经存在则会发生报错，这样能避免缓存被错误的覆盖。同时我们还实现了 Get() 方法， 该方法从缓存系统中获取数据项。还需要注意的是 缓存数据项是否存在的真实含义是，数据项存在且没有过期。接着我们可以实现删除和更新接口了。

```
// 替换一个存在的数据项
func (c *Cache) Replace(k string, v interface{}, d time.Duration) error {
    c.mu.Lock()
    _, found := c.get(k)
    if !found {
        c.mu.Unlock()
        return fmt.Errorf("Item %s doesn't exist", k)
    }
    c.set(k, v, d)
    c.mu.Unlock()
    return nil
}
```



```
// 删除一个数据项
func (c *Cache) Delete(k string) {
    c.mu.Lock()
    c.delete(k)
    c.mu.Unlock()
}
```

以上代码一目了然，就不多做介绍了。 上面我们说到缓存系统支持将数据导入到文件中，并且从文件中加载数据，下面让我们实现该功能。

```
// 将缓存数据项写入到 io.Writer 中
func (c *Cache) Save(w io.Writer) (err error) {
    enc := gob.NewEncoder(w)
    defer func() {
        if x := recover(); x != nil {
            err = fmt.Errorf("Error registering item types with Gob library")
        }
    }()
    c.mu.RLock()
    defer c.mu.RUnlock()
    for _, v := range c.items {
        gob.Register(v.Object)
    }
    err = enc.Encode(&c.items)
    return
}

// 保存数据项到文件中
func (c *Cache) SaveToFile(file string) error {
    f, err := os.Create(file)
    if err != nil {
        return err
    }
    if err = c.Save(f); err != nil {
        f.Close()
        return err
    }
    return f.Close()
}

// 从 io.Reader 中读取数据项
func (c *Cache) Load(r io.Reader) error {
    dec := gob.NewDecoder(r)
    items := map[string]Item{}
    err := dec.Decode(&items)
    if err == nil {
        c.mu.Lock()
        defer c.mu.Unlock()
        for k, v := range items {
            ov, found := c.items[k]
            if !found || ov.Expired() {
                c.items[k] = v
            }
        }
    }
    return err
}

// 从文件中加载缓存数据项
func (c *Cache) LoadFile(file string) error {
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    if err = c.Load(f); err != nil {
        f.Close()
        return err
    }
    return f.Close()
}
```


以上代码中， Save() 方法通过 gob 模块将二进制缓存数据转码写入到实现了 io.Writer 接口的对象中，而 Load() 方法则从 io.Reader 中读取二进制数据，然后通过 gob 模块将数据进行反序列化。其实这里，我们其实就是在对缓存数据进行序列和反序列化。目前为止，整个缓存系统的功能以及大部分完成了，下面我们进行扫尾工作吧。

```
// 返回缓存数据项的数量
func (c *Cache) Count() int {
    c.mu.RLock()
    defer c.mu.RUnlock()
    return len(c.items)
}

// 清空缓存
func (c *Cache) Flush() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.items = map[string]Item{}
}

// 停止过期缓存清理
func (c *Cache) StopGc() {
    c.stopGc <- true
}

// 创建一个缓存系统
func NewCache(defaultExpiration, gcInterval time.Duration) *Cache {
    c := &Cache{
        defaultExpiration: defaultExpiration,
        gcInterval:        gcInterval,
        items:             map[string]Item{},
        stopGc:            make(chan bool),
    }
    // 开始启动过期清理 goroutine
    go c.gcLoop()
    return c
}
```


以上代码中，我们又添加了几个方法。 Count() 会返回系统中缓存的数据数量， Flush() 则会清空整个缓存系统， 而 StopGc() 则会使缓存系统停止清理过期数据项。最后，我们可以使用 NewCache() 方法创建一个新的缓存系统。

目前为止，整个缓存系统就已经完成了，是不是很简单呢？下面我们进行一些测试。

4.2 测试系统

在本实验开始的环节中，我们按照以下步骤设置了 $GOPATH 环境变量。

```
$ cd /home/shiyanlou/
$ mkdir -p golang/src
$ cd golang
$ export GOPATH='/home/shiyanlou/golang'
```

下载我们创建 /home/shiyanlou/golang/src/cache 目录，并在该目录创建 **cache.go** 文件，同时将以上缓存系统实现代码写入到该文件中。最后该文件部分内容如下所示:

```
package cache

import (
    "encoding/gob"
    "fmt"
    "io"
    "os"
    "sync"
    "time"
)

type Item struct {
    Object     interface{} // 真正的数据项
    Expiration int64       // 生存时间
}

// 省略部分内容
//......

// 创建一个缓存系统
func NewCache(defaultExpiration, gcInterval time.Duration) *Cache {
    c := &Cache{
        defaultExpiration: defaultExpiration,
        gcInterval:        gcInterval,
        items:             map[string]Item{},
    }
    // 开始启动过期清理 goroutine
    go c.gcLoop()
    return c
}
```


下面我们写一个示例程序，该示例程序源代码位于 /home/shiyanlou/golang/sample.go 中，其内容如下：

```
package main

import (
    "cache"
    "fmt"
    "time"
)

func main() {
    defaultExpiration, _ := time.ParseDuration("0.5h")
    gcInterval, _ := time.ParseDuration("3s")
    c := cache.NewCache(defaultExpiration, gcInterval)

    k1 := "hello shiyanlou"
    expiration, _ := time.ParseDuration("5s")

    c.Set("k1", k1, expiration)
    s, _ := time.ParseDuration("10s")
    if v, found := c.Get("k1"); found {
        fmt.Println("Found k1: ", v)
    } else {
        fmt.Println("Not found k1")
    }
    // 暂停 10s
    time.Sleep(s)
    // 现在 k1 应该被清理掉了
    if v, found := c.Get("k1"); found {
        fmt.Println("Found k1: ", v)
    } else {
        fmt.Println("Not found k1")
    }
}
```

示例代码非常简单。我们通过 NewCache 方法创建了一个缓存系统，其过期数据清理周期为 3 秒，数据项默认的过期时间为半小时。然后我们设置了一个数据项 **"k1"**, 该数据项的过期时间为 5 秒，可以看到，我们设置数据项后马上获取该数据项，然后暂停 10 秒后，我们再次获取 **"k1"**, 这时候应该发现 **"k1"** 已经被清除了。可以通过以下方法执行：

```
$ cd /home/shiyanlou/golang
$ go run sample.go
```

可以看到以下输出：

```
Found k1:  hello shiyanlou
Not found k1
```

五、实验总结

通过这个实验，我们实现了一个简单的缓存系统，该缓存系统支持数据对象的，添加，删除，替换，和查询操作，同时还支持过期数据的删除。如果你熟悉 Redis (http://redis.io/)，那么你会发现在 Redis 中，支持增加某一数字的值，比如如果设置了 "key" 的值为 "20", 那么可以通过传递参数 "2" 给 `Increment` 接口，增加 "key" 的值到 "22"，你能在本实验的缓存系统基础上实现该功能吗？

同时在测试环节中，我们并未对缓存系统保存数据到文件和从文件中加载数据功能进行测试，你能给出相应的测试代码吗？

课程教师

aiden0z

共发布过**6**门课程

查看老师的所有课程 > (/teacher/5348)