第三章 动手写缓存2

主讲人：陈向
湖南科技大学
计算机科学与工程学院

HTTP 服务端

```go
package main

import (
    "log"
    "net/http"
)

type server int

func (h *server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    log.Println(r.URL.Path)
    w.Write([]byte("Hello World!"))
}

func main() {
    var s server
    http.ListenAndServe("localhost:9999", &s)
}
```

执行 go run .

```
1  $ curl http://localhost:9999
2  Hello World!
3  $ curl http://localhost:9999/abc
4  Hello World!
```

创建HttpServer

```
geecache/
    |--lru/
        |--lru.go  // lru 缓存淘汰策略
    |--byteview.go // 缓存值的抽象与封装
    |--cache.go    // 并发控制
    |--geecache.go // 负责与外部交互，控制缓存存储和获取的主流程
    |--http.go     // 提供被其他节点访问的能力(基于http)
```

```go
package geecache

import (
    "fmt"
    "log"
    "net/http"
    "strings"
)

const defaultBasePath = "/_geecache/"

// HTTPPool implements PeerPicker for a pool of HTTP peers.
type HTTPPool struct {
    // this peer's base URL, e.g. "https://example.net:8000"
    self     string
    basePath string
}

// NewHTTPPool initializes an HTTP pool of peers.
func NewHTTPPool(self string) *HTTPPool {
    return &HTTPPool{
        self:     self,
        basePath: defaultBasePath,
    }
}
```

接下来，实现最核心的 ServeHTTP 方法

- // Log info with server name
  func (p *HTTPPool) Log(format string, v ...interface{}) {
  log.Printf("[Server %s] %s", p.self, fmt.Sprintf(format, v...))
  }

  // ServeHTTP handle all http requests
  func (p *HTTPPool) ServeHTTP(w http.ResponseWriter, r *http.Request) {

  }

```
if !strings.HasPrefix(r.URL.Path, p.basePath) {
        panic("HTTPPool serving unexpected path: " + r.URL.Path)
}
p.Log("%s %s", r.Method, r.URL.Path)
// /<basepath>/<groupname>/<key> required
parts := strings.SplitN(r.URL.Path[len(p.basePath):], "/", 2)
if len(parts) != 2 {
http.Error(w, "bad request", http.StatusBadRequest)
return
}

groupName := parts[0]
key := parts[1]

group := GetGroup(groupName)
if group == nil {
http.Error(w, "no such group: "+groupName, http.StatusNotFound)
return
}

view, err := group.Get(key)
if err != nil {
http.Error(w, err.Error(), http.StatusInternalServerError)
return
}

w.Header().Set("Content-Type", "application/octet-stream")
w.Write(view.ByteSlice())
```

测试

```
package main

import (
"fmt"
"geecache"
"log"
"net/http"
)

var db = map[string]string{
"Tom": "630",
"Jack": "589",
"Sam": "567",
}
```

```
func main() {
        geecache.NewGroup( “scores” , 2<<10,
        geecache.GetterFunc(
        func(key string) ([]byte, error) {
                log.Println( “[SlowDB] search key” , key)
                if v, ok := db[key]; ok {
                        return []byte(v), nil }
                return nil, fmt.Errorf("%s not exist", key)
        }))

        addr := "localhost:9999"
        peers := geecache.NewHTTPPool(addr)
        log.Println("geecache is running at", addr)
        log.Fatal(http.ListenAndServe(addr, peers))
}
```

运行 main 函数，使用 curl 做一些简单测试

$ curl http://localhost:9999/_geecache/scores/Tom
630
$ curl http://localhost:9999/_geecache/scores/kkk
kkk not exist

日志输出如下：

```
2020/02/11 23:28:39 geecache is running at localhost:9999
2020/02/11 23:29:08 [Server localhost:9999] GET /_geecache/scores/Tom
2020/02/11 23:29:08 [SlowDB] search key Tom
2020/02/11 23:29:16 [Server localhost:9999] GET /_geecache/scores/kkk
2020/02/11 23:29:16 [SlowDB] search key kkk
```

一致性Hash

```go
package consistenthash

import (
  “hash/crc32”
  “sort”
  “strconv”
)

// Hash maps bytes to uint32
type Hash func(data []byte) uint32

// Map constains all hashed keys
type Map struct {
    hash Hash //Hash 函数
    replicas int  //虚拟节点倍数
    keys []int // Sorted 哈希环keys
    hashMap map[int]string //虚拟节点与真实节点映射表
}
```

```go
// New creates a Map instance
func New(replicas int, fn Hash) *Map {
    m := &Map{
            replicas: replicas,
            hash: fn,
            hashMap: make(map[int]string),
    }
    if m.hash == nil {
            m.hash = crc32.ChecksumIEEE
    }
    return m
}
```

实现添加真实节点/机器的 Add() 方法

```
// Add adds some keys to the hash.
func (m *Map) Add(keys ...string) {
  for _, key := range keys {
      for i := 0; i < m.replicas; i++ {
          hash := int(m.hash([]byte(strconv.Itoa(i) + key)))
          m.keys = append(m.keys, hash)
          m.hashMap[hash] = key
      }
   }
   sort.Ints(m.keys)
}
```

实现选择节点的Get()方法

```go
// Get gets the closest item in the hash to the provided key.
func (m *Map) Get(key string) string {
    if len(m.keys) == 0 {
        return ""
    }

    hash := int(m.hash([]byte(key)))
    // Binary search for appropriate replica.
    idx := sort.Search(len(m.keys), func(i int) bool {
        return m.keys[i] >= hash
    })

    return m.hashMap[m.keys[idx%len(m.keys)]]
}
```

防止缓存击穿

缓存雪崩：缓存在同一时刻全部失效，造成瞬时DB请求量大、压力骤增，引起雪崩。缓存雪崩通常因为缓存服务器宕机、缓存的 key 设置了相同的过期时间等引起。

缓存击穿：一个存在的key，在缓存过期的一刻，同时有大量的请求，这些请求都会击穿到 DB，造成瞬时DB请求量大、压力骤增。

缓存穿透：查询一个不存在的数据，因为不存在则不会写到缓存中，所以每次都会去请求 DB，如果瞬间流量过大，穿透到 DB，导致宕机。

首先创建 call 和 Group 类型

```
1   package singleflight
2
3   import "sync"
4
5   type call struct {
6       wg  sync.WaitGroup
7       val interface{}
8       err error
9   }
10
11  type Group struct {
12      mu sync.Mutex       // protects m
13      m  map[string]*call
14  }
```

实现do方法

```go
func (g *Group) Do(key string, fn func() (interface{}, error)) (interface{}, error) {
    g.mu.Lock()
    if g.m == nil {
        g.m = make(map[string]*call)
    }
    if c, ok := g.m[key]; ok {
        g.mu.Unlock()
        c.wg.Wait()
        return c.val, c.err
    }
    c := new(call)
    c.wg.Add(1)
    g.m[key] = c
    g.mu.Unlock()

    c.val, c.err = fn()
    c.wg.Done()

    g.mu.Lock()
    delete(g.m, key)
    g.mu.Unlock()

    return c.val, c.err
}
```

精简

```go
func (g *Group) Do(key string, fn func() (interface{}, error)) (interface{}, error) {
    if c, ok := g.m[key]; ok {
        c.wg.Wait()   // 如果请求正在进行中，则等待
        return c.val, c.err  // 请求结束，返回结果
    }
    c := new(call)
    c.wg.Add(1)       // 发起请求前加锁
    g.m[key] = c      // 添加到 g.m，表明 key 已经有对应的请求在处理

    c.val, c.err = fn() // 调用 fn，发起请求
    c.wg.Done()         // 请求结束

    delete(g.m, key)    // 更新 g.m

    return c.val, c.err // 返回结果
}
```

singleflight 的使用

修改 geecache.go 中的 Group，添加成员变量 loader，并更新构建函数 NewGroup。

修改 load 函数，将原来的 load 的逻辑，使用 g.loader.Do 包裹起来即可，这样确保了并发场景下针对相同的 key，load 过程只会调用一次。

```go
type Group struct {
    name      string
    getter    Getter
    mainCache cache
    peers     PeerPicker
    // use singleflight.Group to make sure that
    // each key is only fetched once
    loader *singleflight.Group
}

func NewGroup(name string, cacheBytes int64, getter Getter) *Group {
    // ...
    g := &Group{
        // ...
        loader:    &singleflight.Group{},
    }
    return g
}
```

```go
func (g *Group) load(key string) (value ByteView, err error) {
    // each key is only fetched once (either locally or remotely)
    // regardless of the number of concurrent callers.
    viewi, err := g.loader.Do(key, func() (interface{}, error) {
        if g.peers != nil {
            if peer, ok := g.peers.PickPeer(key); ok {
                if value, err = g.getFromPeer(peer, key); err == nil {
                    return value, nil
                }
                log.Println("[GeeCache] Failed to get from peer", err)
            }
        }

        return g.getLocally(key)
    })

    if err == nil {
        return viewi.(ByteView), nil
    }
    return
}
```

测试

- 执行 run.sh

```
$ ./run.sh
2020/02/16 22:36:00 [Server http://localhost:8003] Pick peer http://localhost:8001
2020/02/16 22:36:00 [Server http://localhost:8001] GET /_geecache/scores/Tom
2020/02/16 22:36:00 [SlowDB] search key Tom
630630630
```

可以看到，向 API 发起了三次并发请求，但8003 只向 8001 发起了一次请求，就搞定了。