go 实现缓存服务

潘重宇

2022/5/19

缓存服务概述 随着互联网的飞速发展，各行各业对互联网服务的要求也越来越高，服务架构的响应速度、数据容量一直是研究的重点。缓存服务就是为服务提速的一个方案。它需要被设置在其他服务的前端，客户端首先访问缓存查询自己的数据，仅当客户端需要的数据不存在于缓存中时，才会去访问实际的服务。从实际的服务中获取到的数据会被放在缓存中，以备下次使用。

缓存与存储的关系 存储是非易失的，被存储的内容通常会被期望永久保存。存储对性能也有要求，比如要求系统的吞吐量达到每秒多少字节等。但在单个请求的响应时间上，存储一般不会有很高的要求。
缓存被用来提升访问资源的速度。在设计之初，缓存就允许自己的数据出现丢失的情况，甚至有用生存时间、启发式算法来决定缓存内容的写入与回收。缓存的目的就是要快速存取。

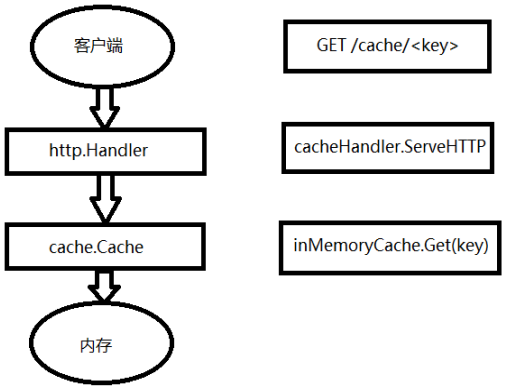
1 基于 HTTP 的内存缓存服务

使用 Go 语言写基于 HTTP 的缓存服务非常方便，我们只需要一个 map 来保存键值对，写一个 handler 来处理请求，调用 http.ListenAndServe 启动服务。Go 的 HTTP 服务框架解决了底层的协议和并发问题。我们会利用 HTTP/REST 协议的 GET/PUT/DEL 方法实现缓存的 Get/Set/Del 操作。

1.1 响应流程

我们以 Get 方法为例展示一下流程。如下图所示：

图 1: in memory 缓存的 Get 流程（http）



客户端使用 HTTP 的 GET 请求提供了 key，服务端 http.Handler 的具体实现 cacheHandler 实现的 SereHTTP 方法负责解析客户端传来的 HTTP 请求，并调用 cache.Cache 接口的 Get 方法。inMemoryCache 结构体的 Get 方法会将内存里的 map（Go 语言里的一种键值对数据结构）中查询 key 对应的 value 并返回。cacheHandler 会将 value 写入 HTTP 的响应正文并返回 200 OK。如果 cache.Cache.Get 返回错误，cacheHandler 会返回 500 Internal Server Error。如果 value 长度为 0 则返回 404 Not Found。

1.2 cache 包的实现

我们希望在 cache 包里实现服务的缓存功能而不涉及 http 服务的相关内容。首先我们声明一个 Cache 接口：

```
type Cache interface {
  Set(string, []byte) error
  Get(string) ([]byte, error)
```

```
    Del(string) error
    GetStat() Stat
  }
```

在这个接口里，我们声明了增删查等方法。任何结构体只要实现了这些方法，我们就认为该结构体实现了 Cache 接口。Set 用于设置键值对，它接受一个 string 类型的 key 和 []byte 类型的 value，返回类型为 error，用于反馈操作是否成功。GetStat 用于返回当前缓存的状态，类型 Stat 为用户定义类型的结构体。

```
type Stat struct {
  Count     int64
  KeySize   int64
  ValueSize int64
}

func (s *Stat) add(k string, v []byte) {
  s.Count += 1
  s.KeySize += int64(len(k))
  s.ValueSize += int64(len(v))
}

func (s *Stat) del(k string, v []byte) {
  ...
}
```

Cache 接口可以有多种实现，在这儿我们选择实现内存内缓存 inMemoryCache。它需要实现 Cache 里的所有方法，代码如下所示：

```
func New(typ string) Cache {
  var c Cache
  if typ == "inmemory" {
    c = newInmemoryCache()
  }
  if c == nil {
    panic("Unknown cachetype" + typ)
  }
  log.Println(typ, "ready to server")
  return c
}

func newInmemoryCache() *inMemoryCache {
  return &inMemoryCache{make(map[string][]byte), sync.RWMutex{}, Stat{}}
}

type inMemoryCache struct {
  c     map[string][]byte
  mutex sync.RWMutex
  Stat
}

func (c *inMemoryCache) Set(k string, v []byte) error {
  c.mutex.Lock()
  defer c.mutex.Unlock()
  tmp, exist := c.c[k]
  if exist {
    c.del(k, tmp)
  }
  c.c[k] = v
  c.add(k, v)
```

```
    return nil
  }

  func (c *inMemoryCache) Get(k string) ([]byte, error) {
    c.mutex.RLock()
    defer c.mutex.RUnlock()
    return c.c[k], nil
  }

  func (c *inMemoryCache) Del(k string) error {
    ...
  }

  func (c *inMemoryCache) GetStat() Stat {
    return c.Stat
  }

```

inMemoryCache 结构体包含一个类型为 map 的成员 c，用于存储键值对；一个锁 mutex 用于对 c 提供并发读写保护；一个 Stat 用于记录缓存状态。Stat 结构提的方法可以直接被 inMemoryCache 调用。

1.3 HTTP 包的实现

HTTP 包用来实现我们的 HTTP 服务功能，Go 的 net/http 包提供了很多便利的接口于方法。服务的 Server 结构体代码如下：

```
  type Server struct {
    cache.Cache
  }

  func (s *Server) Listen() {
    http.Handle("/cache/", s.cacheHandler())
    http.Handle("/status", s.statusHandler())
    http.ListenAndServe(":12345", nil)
  }

  func New(c cache.Cache) *Server {
    return &Server{c}
  }

```

http.Handle 方法用于处理对应 url 路径的 http 请求，它接收一段 string 和 http.Handler 接口。我们需要用 cacheHandler() 返回 http.Handler 接口。具体来说是需要实现 Handler 接口里的 ServeHTTP 方法。

```
  type cacheHandler struct {
    *Server
  }

  func (h *cacheHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    key := strings.Split(r.URL.EscapedPath(), "/")[2]
    if len(key) == 0 {
      w.WriteHeader(http.StatusBadRequest)
      return
    }
    m := r.Method
    if m == http.MethodPut {
      b, _ := ioutil.ReadAll(r.Body)
      if len(b) != 0 {
        e := h.Set(key, b)
```

```
        if e != nil {
          log.Println(e)
          w.WriteHeader(http.StatusInternalServerError)
        }
      }
      return
    }
    ...
    w.WriteHeader(http.StatusMethodNotAllowed)
  }

  func (s *Server) cacheHandler() http.Handler {
    return &cacheHandler{s}
  }

```

1.4 main 包实现

最后是 main 包的内容：

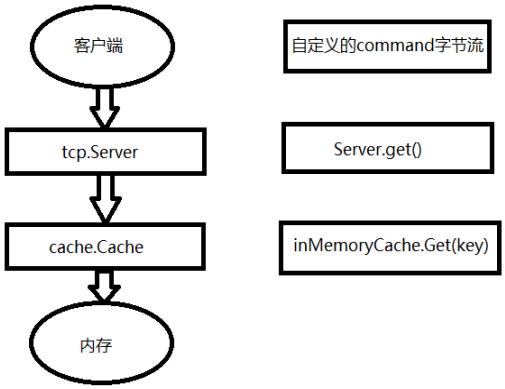
```
  func main() {
    c := cache.New("inmemory")
    HTTP.New(c).Listen()
  }

```

2 基于 TCP 的内存缓存服务

抛弃 Golang 自身的 HTTP 框架意味着我们需要处理 TCP 连接和并发的请求，自己定义和解析 tcp 传输的网络字节流。大致流程如下所示：

图 2: in memory 缓存的 Get 流程（TCP）



幸运的是我们只需要解决这些问题，并且 go 有着便利的 goroutine 用于并发。流程图中的 cache 包已经在上一节中实现了，我们只需要实现 tcp 包的相关内容。

2.1 TCP 包的实现

和 HTTP 包一样，我们需要一个 Server 结构体来处理 TCP 连接以及客户端交互。它的 Listen() 方法使用一个无线循环接受 12346 端口的客户端连接请求，并在 goroutine 上调用 process() 来处理请求：

```
type Server struct {
  cache.Cache
}

func (s *Server) Listen() {
  l, e := net.Listen("tcp", ":12346")
  if e != nil {
    panic(e)
  }
  for {
    c, e := l.Accept()
    if e != nil {
      panic(e)
    }
    go s.process(c)
  }
}

func New(c cache.Cache) *Server {
  return &Server{c}
}
```

其中 Server.process() 的相关实现如下所示：

```
func (s *Server) readKey(r *bufio.Reader) (string, error) {
  klen, e := readLen(r)
  if e != nil {
    return "", e
  }
  k := make([]byte, klen)
  _, e = io.ReadFull(r, k)
  if e != nil {
    return "", e
  }
  return string(k), nil
}

func (s *Server) readKeyAndValue(r *bufio.Reader) (string, []byte, error) {
  ...
}

func readLen(r *bufio.Reader) (int, error) {
  //读取用户命令中下一段字符长度
  ...
}

func sendResponse(value []byte, err error, conn net.Conn) error {
  if err != nil {
    errString := err.Error()
    tmp := fmt.Sprintf("-%d ", len(errString)) + errString
    _, e := conn.Write([]byte(tmp))
    return e
  }
  vlen := fmt.Sprintf("%d ", len(value))
  _, e := conn.Write(append([]byte(vlen), value...))
  return e
}

func (s *Server) get(conn net.Conn, r *bufio.Reader) error {
  k, e := s.readKey(r)
  if e != nil {
```

```
38        return e
39      }
40      v, e := s.Get(k)
41      return sendResponse(v, e, conn)
42    }
43
44    func (s *Server) set(conn net.Conn, r *bufio.Reader) error {
45      ...
46    }
47
48    func (s *Server) del(conn net.Conn, r *bufio.Reader) error {
49      ...
50    }
51
52    func (s *Server) process(conn net.Conn) {
53      defer conn.Close()
54      r := bufio.NewReader(conn)
55      for {
56        op, e := r.ReadByte()
57        if e != nil {
58          if e != io.EOF {
59            log.Println("close connection due to error", e)
60          }
61          return
62        }
63        if op == 'S' {
64          e = s.set(conn, r)
65        } else if op == 'G' {
66          e = s.get(conn, r)
67        } else if op == 'D' {
68          e = s.del(conn, r)
69        } else {
70          log.Println("close connection due to invalid operation:", op)
71          return
72        }
73        if e != nil {
74          log.Println("close connection due to error:", e)
75          return
76        }
77      }
78    }
79
```