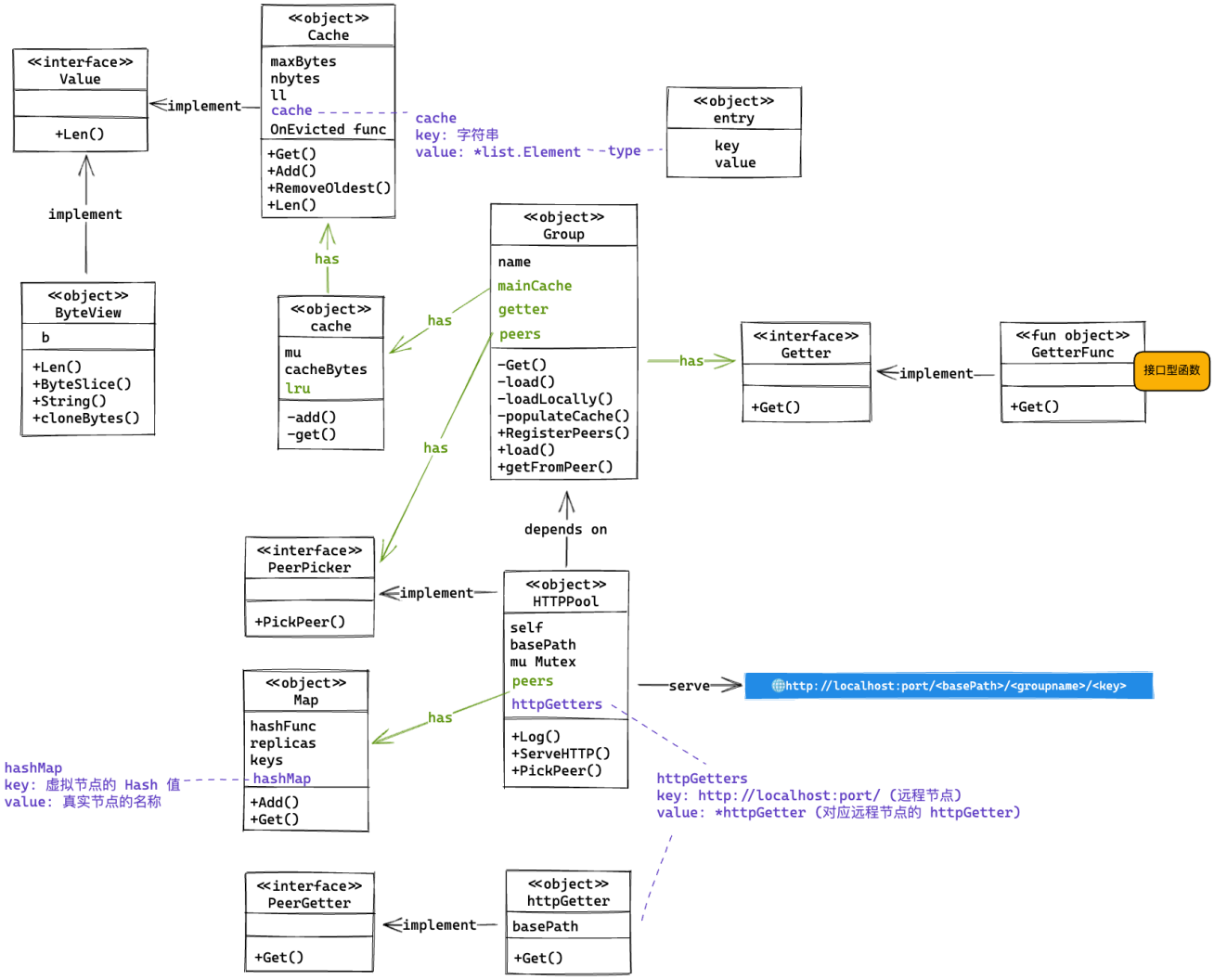
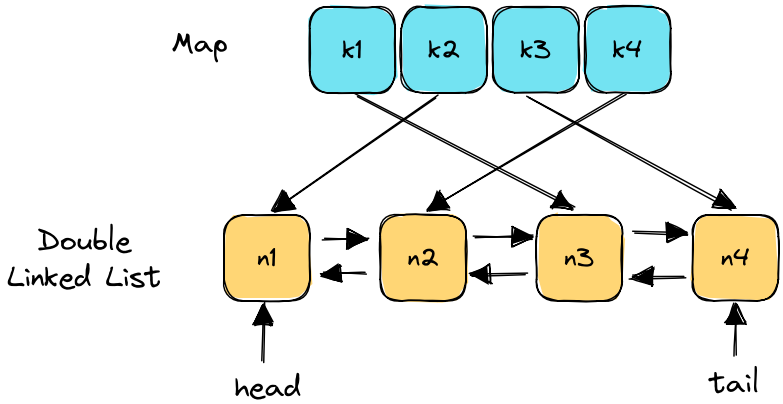
缓存 - GeeCache

#golang #学习



LRU(Least Recently Used)

- 最近最少使用法
- 如果数据最近被访问过，那么将来被访问的概率也会更高；



-

查找缓存

1. 从 Map 中找到对应的双向链表节点；
2. 将节点移动到队尾；约定 head 为队尾；

淘汰缓存

1. 移除最近最少访问的节点（队首 tail）；
2. 取到队首节点 → 删除映射关系 → 更新当前内存 → 调用回调函数；

新增 / 修改缓存

1. 如果节点存在，更新节点值 → 移动到队尾；
2. 如果节点不存在：
    1. 超出最大内存限制 → 删除队首节点 → 在队尾插入新节点；
    2. 未超出最大内存限制 → 在队尾添加新节点；

```
// lru.go
type Cache struct {
        maxBytes  int64                                // 最大使用内存
        nbytes    int64                                // 当前使用内存
        ll        *list.List                           // 双向链表
        cache     map[string]*list.Element             // key: 字符串;
*list.Element 指向双向列表节点的指针
        OnEvicted func(key string, value Value) // 删除某个记录时的回调
函数
}

// entry: 双向列表节点的数据类型
type entry struct {
        key   string
        value Value
}

// Get 从 c.cache 中取出 key 对应的元素指针
// → 将当前元素移动到双向链表的队尾(front)
// → 转换为 entry 类型 → 返回 entry.value
func (c *Cache) Get(key string) (value Value, ok bool)

// RemoveOldesr 从 ll 中取到队首的元素(tail) 并删除
func (c *Cache) RemoveOldest()

// Add 新增缓存
// 1. key 存在 → 更新对应 value → 移动到队尾
```

```
// 2. key 不存在 → 创建新的 entry → 插入到链表及 cache 映射表中
// 判断是否超过了缓存容量
// 超过 → RemoveOldest
func (c *Cache) Add(key string, value Value)
```

单机并发缓存

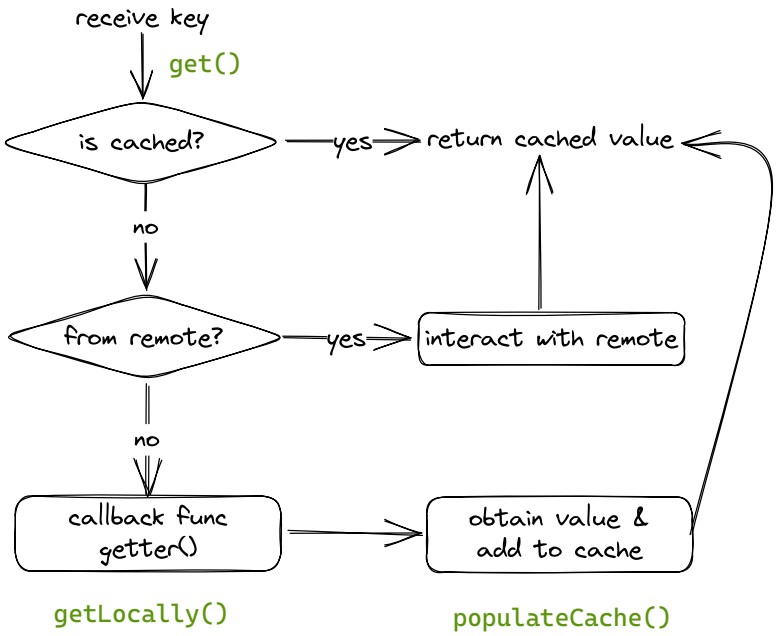
支持并发读写

- 使用 `sync.Mutex` 封装 LRU 的方法，使之支持**并发读写**；
- 抽象只读数据结构 `ByteView` 来表示缓存值；
- cache：对 LRU cache 的进一步封装；
  - 在 add 时才实例化 LRU cache → **Lazy Initialization （延迟初始化）** → 提高性能减少内存需求

```
// byteview.go
type ByteView struct {
        b []byte  // 存储真实的缓存值
}

func (v ByteView) Len()
func (v ByteView) ByteSlice()
func (v ByteView) cloneBytes(b []byte)
func (v ByteView) String()
func cloneBytes(b []byte)

// cache.go
type cache struct {
        mu         sync.Mutex  // 互斥锁
        lru        *lru.Cache  // LRU 缓存
        cacheBytes int64       // 缓存大小
}

// add → 获取 mu → *lru.cache.Add()
func (c *cache) add(key string, value ByteView)

// get → mu → *lru.cache.Get()
func (c *cache) get(key string) (value ByteView, ok bool)
```

主体结构 Group

- 负责与用户交互，并控制缓存值的存储和获取的过程



回调函数 Getter

- **接口型函数**
  - 只能用于接口内部只制定了一个方法的情况；
  - 更加灵活，可读性也更好；
  - 应用场景：`func GetFromSource(getter Getter, key string)[]byte{…}`
    - 将 GetterFunc 作为方法的参数；
    - 可以将实现了 Getter 接口的结构体作为方法的参数；

```
// Getter 缓存未命中时获取源数据的回调函数
type Getter interface {
        Get(key string) ([]byte, error)
}

// 定义函数类型实现 Getter 接口 —— 接口型函数
type GetterFunc func(key string) ([]byte, error)

func (f GetterFunc) Get(key string) ([]byte, error) {
        return f(key)
}
```

缓存组 Group

```
type Group struct {
        name      string // 每个 Group 的唯一标识符 - 命名空间
        getter    Getter // 未命中缓存时用来获取数据源的回调函数
        mainCache cache  // 并发缓存
}
```

```
// Get 获得缓存值
// 1. 命中缓存 → g.mainCache.get(key) 从缓存中取 → return
// 2. 未命中缓存 → load → return
func (g *Group) Get(key string) (ByteView, error)

// load → getlocally
func (g *Group) load(key string) (value ByteView, err error)

// getlocally → g.getter.Get → populateCache
func (g *Group) getlocally(key string) (ByteView, error)

// populateCache → g.mainCache.Add(key, value) 更新缓存
func (g *Group) populateCache(key string, value ByteView)
```

HTTP Server

- 实现节点之间的通信；
- 构造 `HTTPPool` 作为承载节点 HTTP 通信的核心数据结构；
- `/<basepath>/<groupname>/<key>`

```
type HTTPPool struct {
        self     string // 用来记录自己的地址 主机名:端口号
        basePath string // 节点之间通讯的前缀地址
http://example.com/_geecache/
}

func (p *HTTPPool) Log(format string, v ...interface{})

// ServeHTTP
// → parse r.URL.Path → basepath, groupname, key
// → g := GetGroup(groupname) → g.Get(key) → w.Write()
func (p *HTTPPool) ServeHTTP(w http.ResponseWriter, r *http.Request)
```

一致性哈希

> 单节点 → 分布式节点

为什么?

- 访问哪个节点?
  - **对于给定的 key 每次都访问相同的节点获取值；**
  - 缓解缓存性能问题，不需要每个节点都存储相同的数据；
- 节点数目变化了怎么办?

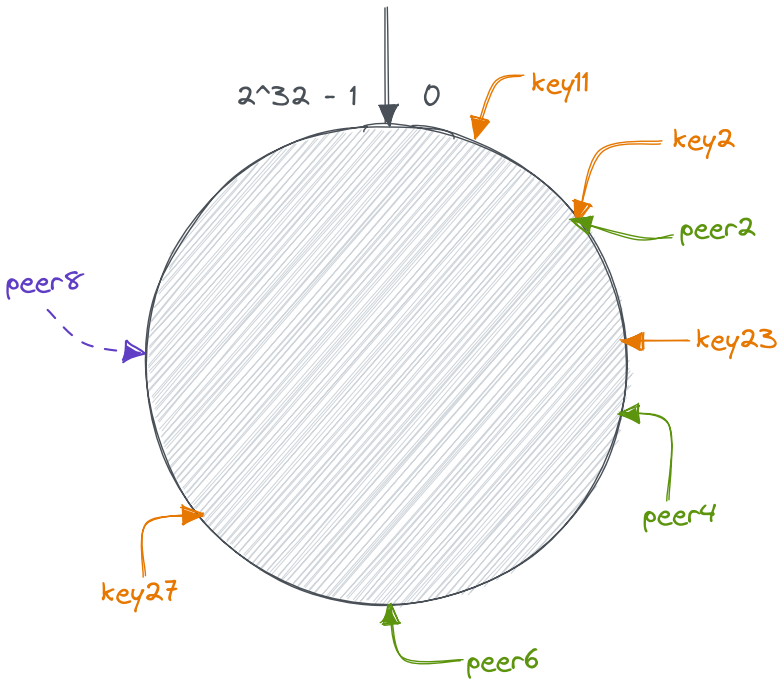
- 缓存雪崩问题
  - 缓存在同一时刻全部失效，造成瞬间 DB 请求量大、压力骤增、引起雪崩；
  - 通常是因为缓存服务器宕机 / 缓存设置了相同的过期时间；
- **一致性哈希算法**

怎么做?

> key → 2^{32} 空间中 → 首尾相连形成一个环；

- 计算节点/机器的哈希值，放在环上；
  - 通常使用节点的名称、编号和 IP 地址进行哈希值的计算；
- 计算 key 的哈希值，放在环上，顺时针找到的第一个节点/机器，就是应该访问的节点/机器；

```
key11, key2, key27 → peer2
key23 → peer4

// 新增节点 peer8
key27 → peer8
```



数据倾斜问题

- 问题概述：如果服务器节点过少，容易引起 key 的倾斜 → 负载不均衡；
- 解决方案：**虚拟节点**
  - 一个真实节点对应多个虚拟节点；`peer1→ peer1-1, peer1-2, peer1-3`

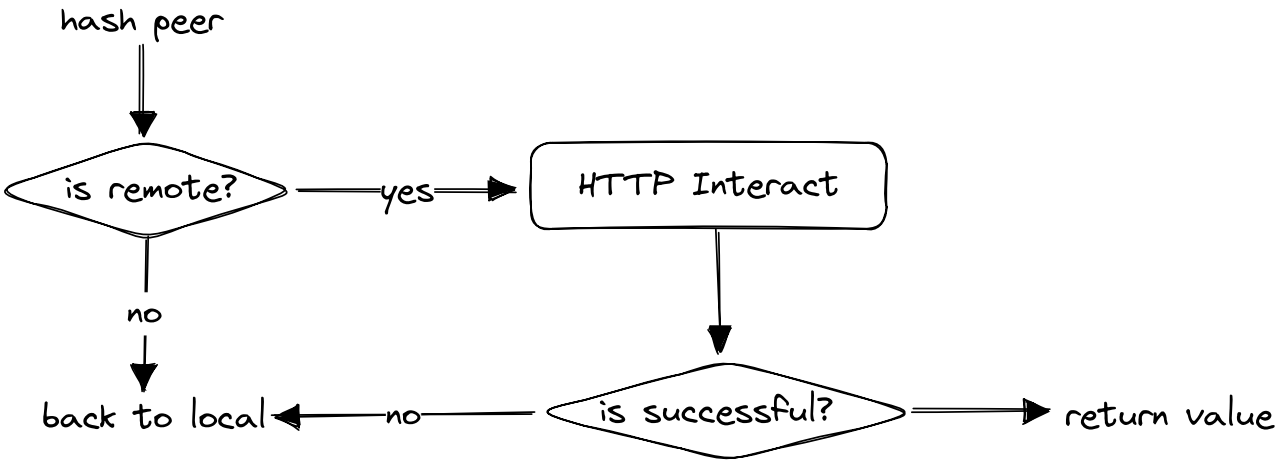
1. 计算虚拟节点的 Hash 值放在环上；

2. 计算 key 的 Hash 值，顺时针寻找最近的虚拟节点；
3. 如果是 `peer2-1`，对应的真实节点为 `peer2`；

具体实现

```
type Map struct {
        hash     Hash           // 注入式 Hash 函数，默认为 crc32.ChecksumIEEE 算法
        replicas int            // 虚拟节点副本数
        keys     []int          // 哈希环
        hashMap  map[int]string // 虚拟节点和真实节点的映射表
                                // key: 虚拟节点的 Hash 值
                                // value: 真实节点的名称
}

// Add →
// 根据 replicas 实现哈希环 keys 上虚拟节点的创建并建立虚拟节点与真实节点之间的映射表
func (m *Map) Add(keys ...string)

// Get
// 计算输入 key 的哈希值 → 在 m.keys 上找到最近的虚拟节点
// 查找映射表 m.hashMap → 返回虚拟节点对应的真实节点名称
func (m *Map) Get(key string) string
```

分布式节点



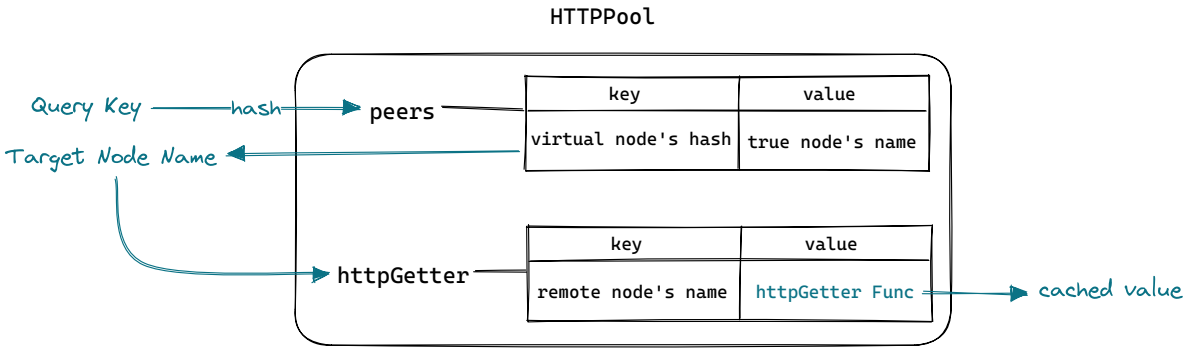
- 注册节点，借助 一致性哈希 选择节点；
- HTTP Client，与远程节点服务器通信；

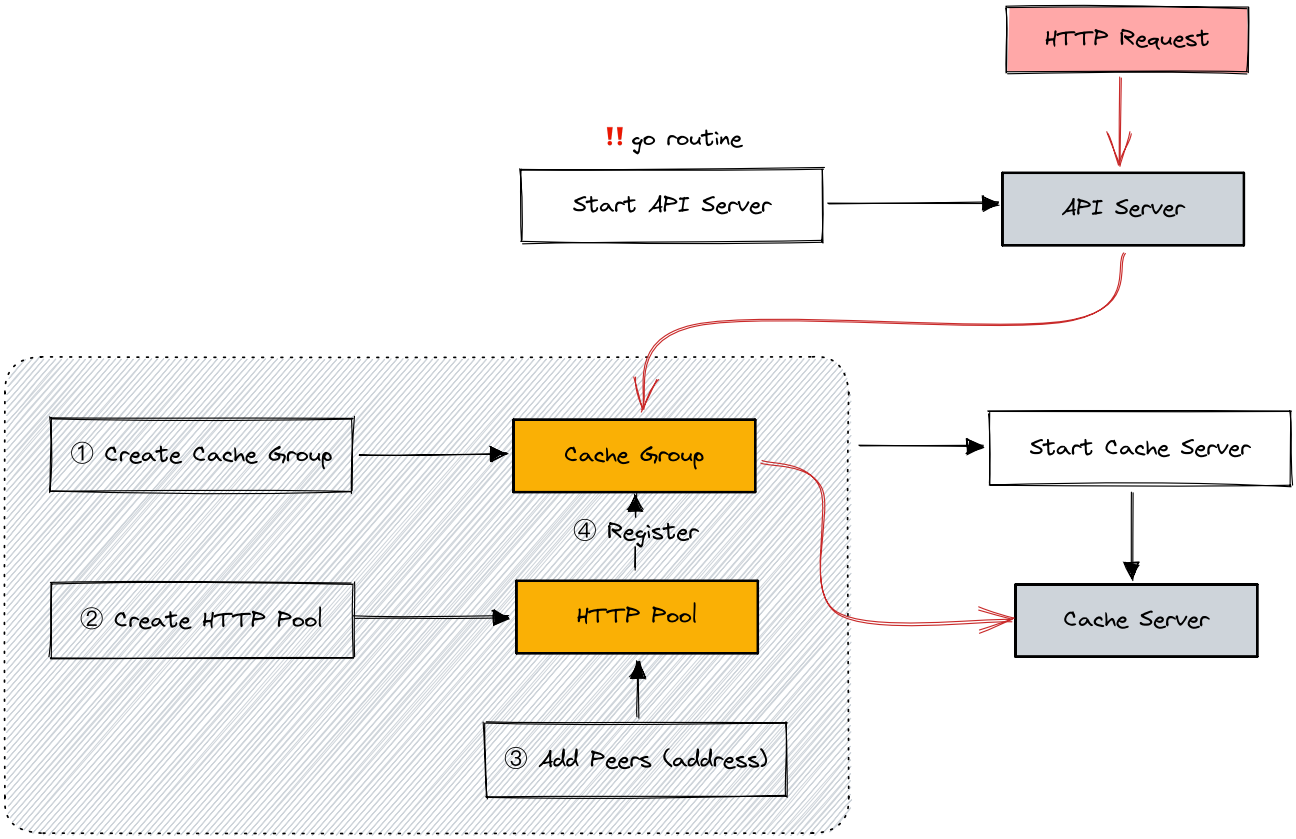
```go
// 根据传入的 key 选择相应节点 PeerGetter，即 HTTP Client
type PeerPicker interface {
        PickPeer(key string) (peer PeerGetter, ok bool)
}

// 从对应的 Group 中查找缓存值
type PeerGetter interface {
        Get(group string, key string) ([]byte, error)
}

type httpGetter struct {
        baseURL string // 要访问的远程节点地址
}

var _ PeerGetter = (*httpGetter)(nil)

// Get → 访问远程节点 http://localhost:port/_geecache/... 获得返回值
func (h *httpGetter) Get(group string, key string) ([]byte, error)
```

```go
type HTTPPool struct {
        self       string // 用来记录自己的地址 主机名:端口号
        basePath   string // 节点之间通讯的前缀地址
        mu         sync.Mutex
        peers      *consistenthash.Map    // 根据 key 选择合适的 peer
        httpGetter map[string]*httpGetter // 远程节点和 httpGetter 的
映射表
}

// Set（peers 为远程节点列表）
// 获得互斥锁 mu → 初始化一致性哈希 p.peers
// → 在一致性哈希中加入所有的远程节点 peer
// → 初始化 m.httpGetter 映射表
// → 在映射表中添加远程节点 peer string 与 httpGetter object 之间的映射
func (p *HTTPPool) Set(peers ...string)

// PickPeer（key 为要查询的值）
// 获得互斥锁 key → p.peers.Get(key) 一致性哈希选择合适的远程节点 peer
// → 查找 m.httpGetter 映射表 → 返回节点对应的 httpGetter
func (p *HTTPPool) PickPeer(key string) (PeerGetter, bool)
```

```go
type Group struct {
        ...
        peers     PeerPicker
}
```

```go
// RegisterPeers 将实现了 PeerPicker 的 HTTPPool 注入到 Group 中
func (g *Group) RegisterPeers(peers PeerPicker)

// load → 使用 g.peers.PickPeer(key) 选择合适的节点
// 1. 若非本机节点 → getFromPeer
// 2. 是本机节点 / 获取远程节点失败 → g.getLocally
func (g *Group) load(key string) (value ByteView, err error)

// getFromPeer 使用实现了 PeerGetter 的 HTTPGetter 从远程节点中获取缓存值
func (g *Group) getFromPeer(peer PeerGetter, key string) (ByteView, error)
```





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。

- singlefilght：确保了并发场景下针对相同的 key，`load` 过程只会调用一次。

```
//in gocache.go
type Group struct {
        name      string // 命名空间
        getter    Getter // 未命中缓存时用来获取数据源的回调函数
        mainCache cache  // 并发缓存
        peers     PeerPicker
        loader    *singleflight.Group
}

// in singlefilght.Group
// 正在进行中的，或者已经结束的请求.
type call struct {
        wg  sync.WaitGroup // 避免重入
        val interface{}
        err error
}

// 管理不同 key 的请求 call.
type Group struct {
        mu sync.Mutex // 保护 m 不会被并发读写
        m  map[string]*call
}

func (g *Group) Do(key string, fn func() (interface{}, error))
(interface{}, error)
```

Protocol Buffers

protobuf 即 Protocol Buffers，Google 开发的一种数据描述语言，是一种轻便高效的结构化数据存储格式，与语言、平台无关，可扩展可序列化。protobuf 以二进制方式存储，占用空间小。

```
brew install protobuf
brew install protoc-gen-go
```

- 按照 protobuf 的语法，在 `.proto` 文件中定义数据结构，并使用 `protoc` 生成 Go 代码（`.proto` 文件是跨平台的，还可以生成 C、Java 等其他源码文件）。
- 在项目代码中引用生成的 Go 代码。
  - `ServeHTTP()` 中使用 `proto.Marshal()` 编码 HTTP 响应。
  - `Get()` 中使用 `proto.Unmarshal()` 解码 HTTP 响应。