分布式缓存系统

分布式缓存就是指在分布式环境或系统下，将一些热门数据存储到离用户近、离应用近的位置，尽量存储到更快的设备，以减少远程数据传输的延迟，让用户和应用可以很快访问到想要的数据

业界具有代表性的分布式缓存系统是**Redis(**远程字典服务器**)**，它将数据存储在内存中，应用可直接到内存读写**Redis**存储的数据

本系统模仿**Redis**，实现分布式的键值存储

实现**:**

1. 单机缓存和基于**HTTP**分布式缓存
2. 最近最少访问缓存策略
3. 利用锁机制防止缓存击穿
4. 使用一致性哈希选择节点，实现负载均衡

项目地址**:** https://github.com/T4t4KAU/Documents/tree/main/cache

缓存淘汰算法

缓存系统的数据全部存储在内存中，但内存是有限的，所以不可能无限制添加数据，当数据所占用的内存超过了容量，那么就要从缓存中移除一条或多条数据，该操作有如下算法**:**

1. **FIFO (First in First Out):** 淘汰缓存中最早添加的记录，如果记录较早但经常被访问，那么这类数据会被频繁添加到缓存，又被淘汰出去，导致缓存命中降低
2. **LFU (Least Frequenty Used):** 淘汰缓存中访问频率最低的记录，维护每个记录的访问次数对内存的消耗较高，如果数据的访问模式发生变化，**LFU**要较长时间去适应，受历史数据的影响比较大
3. **LRU (Least Recenty Used):** 平衡了**FIFO**和**LFU**，如果某个数据最近被访问过，那么将来被访问的概率也会更高

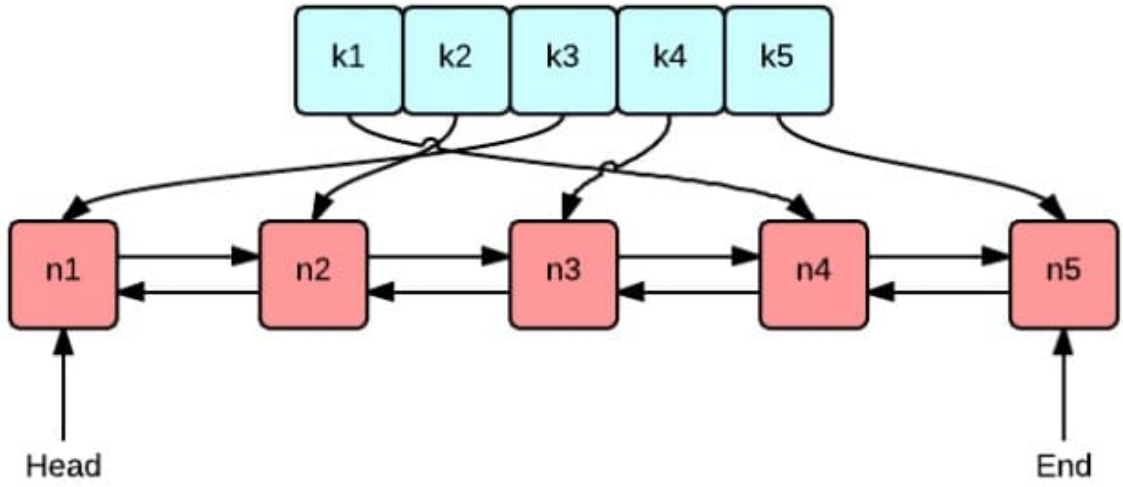
本系统选择**LRU**算法

算法实现**:**

维护一个队列，如果某条记录被访问了，则移动到队尾，那么队首就是最近最少访问的数据

程序有一个存储数据的字典，存储键和值的映射关系，在字典中插入一条记录的复杂度是**O(1)**

同时维护一个双向链表实现的队列，将所有的值放到双向链表中，这样访问到某个值时，将其移动到队尾的复杂度是**O(1)**，在队尾增删数据的复杂度是**O(1)**



流程:

1. 当有新数据插入时，**LRU** 算法会把该数据插入到链表头部，同时把原来链表头部的数据及其之后的数据，都向尾部移动一位
2. 当有数据刚被访问了一次之后，**LRU** 算法就会把该数据从它在链表中的当前位置，移动到链表头部。同时，把从链表头部到它当前位置的其他数据，都向尾部移动一位
3. 当链表长度无法再容纳更多数据时，若再有新数据插入，**LRU** 算法就会去除链表尾部的数据，这也相当于将数据从缓存中淘汰掉

Go代码实现:

```go
// LRU算法：最近最少使用，如果数据最近被访问过，那么将来被访问的概率也会更高
// 维护一个队列，则移动到队尾，那么队首则是最近最少访问的数据，淘汰该记录

type Cache struct {
    maxBytes  int64                          // 最大内存
    nBytes    int64                          // 当前已使用内存
    List      *list.List                     // 双向链表
    cache     map[string]*list.Element       // 字典
    OnEvicted func(key string, value Value) // 回调函数
}

type entry struct {
    key   string
    value Value
}

type Value interface {
    Len() int
}

// New 实例化
func New(maxBytes int64, onEvicted func(string, Value)) *Cache {
    return &Cache{
        maxBytes:  maxBytes,
        List:      list.New(),
        cache:     make(map[string]*list.Element),
        OnEvicted: onEvicted,
    }
}
```

```go

// Get 查找: 从字典中找到对应的双向链表的节点 将该节点移动到队尾
func (c *Cache) Get(key string) (value Value, ok bool) {
    if element, ok := c.cache[key]; ok {
        c.List.MoveToFront(element)
        kv := element.Value.(*entry)
        return kv.value, true
    }
    return
}

// RemoveOldest 删除: 淘汰缓存 移除最近最少访问的节点
func (c *Cache) RemoveOldest() {
    element := c.List.Back() // 队首元素
    if element != nil {
        c.List.Remove(element)
        kv := element.Value.(*entry)
        delete(c.cache, kv.key) // 在字典中删除
        c.nBytes -= int64(len(kv.key)) + int64(kv.value.Len())
        if c.OnEvicted != nil {
            c.OnEvicted(kv.key, kv.value)
        }
    }
}

// Add 增加/修改: 如果键存在 则更新对应的节点 将该节点移动到队尾
func (c *Cache) Add(key string, value Value) {
    // 如果键存在 则更新对应节点的值 将该节点移动到队尾
    // 不存在则新增 在队尾添加新节点 并在字典中添加KV
    if element, ok := c.cache[key]; ok {
        c.List.MoveToFront(element)
        kv := element.Value.(*entry)
        c.nBytes += int64(value.Len()) - int64(kv.value.Len())
    } else {
        element := c.List.PushFront(&entry{key, value})
        c.cache[key] = element
        c.nBytes += int64(len(key)) + int64(value.Len())
    }

    // 如果超过了设定的最大值 则移除最少访问的节点
    for c.maxBytes != 0 && c.maxBytes < c.nBytes {
        c.RemoveOldest()
    }
}

func (c *Cache) Len() int {
    return c.List.Len()
}
```

节点选取地址

当一个节点接收到请求，但如果该节点并没有存储缓存值，那么要选择一个节点获取数据

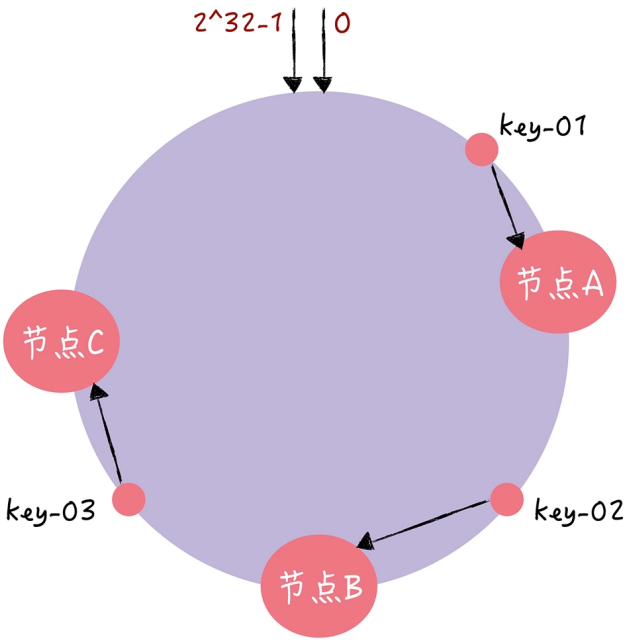
假设有**10**个节点，下面有几个算法可以考虑**:**

1. 随机选取，假设第一次随机选取了节点**1**，节点**1**从数据源获取到数据的同时缓存该数据，那么第二次只有**1/10**的可能性再次选择节点**1**，有**9/10**的概率选择其他节点，如果选择了其他节点，就意味着要再一次从数据源获取数据，这个操作的时间开销较大，这样做，首先是缓存效率低，其次是各个节点上存储着相同的数据，浪费大量的存储空间
2. 普通哈希，对于给定的**key**，每一次都选择同一个节点，可以将**key**的每一个字符的**ASCII**码加起来，再除以**10**取余数，可以解决上述的问题，但是节点数量变化后之前的`hash(key)%10`变成了`hash(key)%9`，这意味着存储值对应的节点都发生了改变，几乎所有的缓存值都失效了，节点在接收到对应的请求时，均要重新去数据源获取数据，容易引起缓存雪崩，要解决这个问题的话要进行数据迁移，但是带来的开销也是巨大的。
3. 一致性哈希，利用一致性哈希算法可以高效的实现负载均衡，将**key**映射到**2^32**的空间中，将这个数字首尾相连，形成一个环，计算节点**/**机器**(**通常使用节点的名称、编号和**IP**地址**)**的哈希值，放置在环上，计算**key**的哈希值，放置在环上，顺时针寻找到第一个节点，就是应选取的节点**/**机器。在新增**/**删除节点时，只要重新定位该节点附近的一小部分数据，而不用重新定位所有的节点，这就解决了上述的问题，与此同时，如果服务器节点过少，容易引起**key**的倾斜，即缓存节点负载不均衡，于是引入了虚拟节点的概念，一个真实的节点对应多个虚拟节点

设一致性哈希函数为**c-hash()**，当要对指定的**key**的值进行读写时，通过下面两步寻址**:**

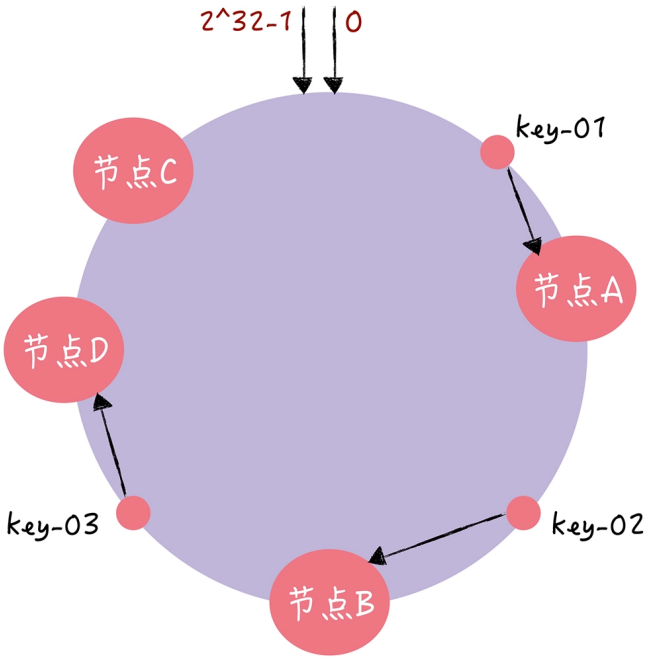
1. 首先将**key**作为参数执行**c-hash()**计算哈希值，并确定**key**在环上的位置
2. 从这个位置沿着哈希环顺时针行走，遇到的第一个节点就是**key**对应的节点

例如，现在有**3**个**key: key-01**、**key-02**、**key-03**，经过哈希算法**c-hash()**计算后，在哈希环的位置如下所示**:**



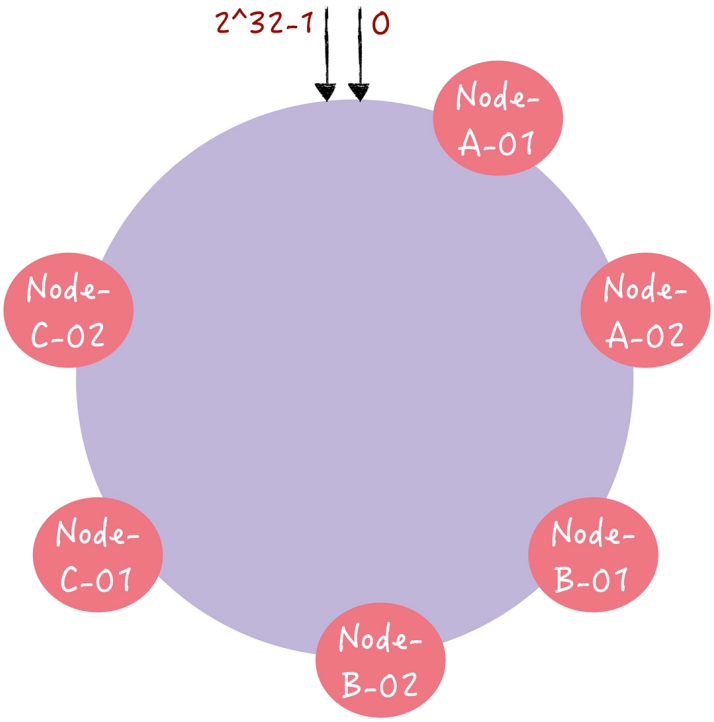
按照顺时针方向，**key-01**寻址到节点**A**，**key-02**寻址到节点**B**，**key-03**寻址到节点**C**

此时如果增加节点**D:**



那么**key-03**的寻址被重新定位到节点**D**，在一致性算法中，如果增加一个节点，受影响的数据仅仅是会寻址到新节点和前一节点的数据，因此在该算法下，数据迁移量要远远小于普通哈希

并且，对一个服务器节点要计算多个哈希值，在每个计算结果上，都放置一个虚拟节点，并将虚拟节点映射到实际节点:



例如访问上方的**Node-A-01**就会定位到**Node A**

代码实现:

```
// 一致性哈希: 将key映射到2^32的空间中 将数字首位相连 形成一个环
// 计算节点/机器(通常使用节点的名称、编号和IP地址)的哈希值 放置在环上
// 计算key的哈希值 放置在环上 顺时针寻找到的一个节点 就是应选取的节点/机器
// 一致性哈希算法在新增/删除节点时 只要重新定位该节点附近的一小部分数据 而无需重新定位所有的节点

// 数据倾斜: 如果服务器节点过少 容易引起key的倾斜 最终使得缓存节点负载不均
```

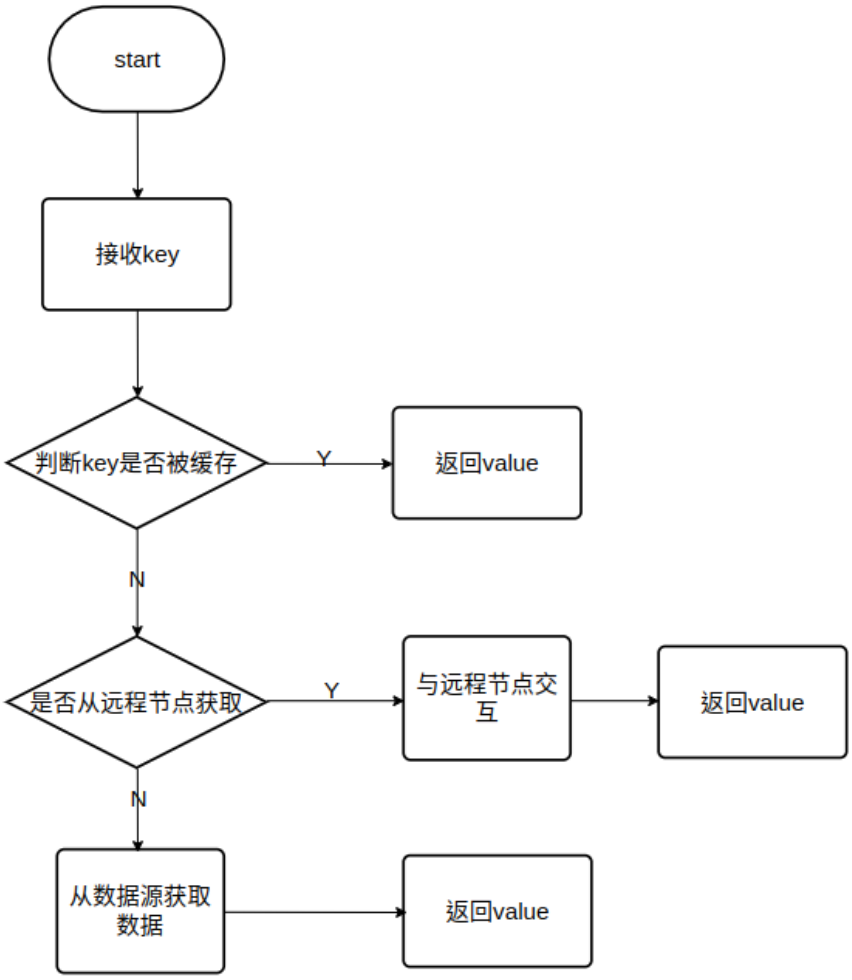
```go
// 引入虚拟节点 一个真实节点对应多个虚拟节点

type Hash func(data []byte) uint32

type Map struct {
    hash     Hash
    replicas int
    keys     []int
    hashMap  map[int]string
}

// New 创建一个Map实例
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
}

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)
}

func (m *Map) Get(key string) string {
    if len(m.keys) == 0 {
        return ""
    }
    hash := int(m.hash([]byte(key)))
    index := sort.Search(len(m.keys), func(i int) bool {
        return m.keys[i] >= hash
    })
    return m.hashMap[m.keys[index%len(m.keys)]]
}
```

分布式节点

本系统能够注册节点，借助一致性哈希算法选择节点，采用**HTTP**与远程节点服务端通信，运行程序时会开启一个**API**节点**(**可选**)**，和一系列存储节点，**API**节点负责与用户交互，其他存储节点负责缓存数据

每一个节点运行一个**HTTP**服务器，来接收客户端的连接，提供服务

服务流程**:**




```go
const (
    defaultBasePath = "/_cache/"
    defaultReplicas = 50
)

type HTTPPool struct {
    self        string
    basePath    string
    mu          sync.Mutex
    peers       *consist.Map
    httpGetters map[string]*httpGetter
}

// NewHTTPPool 初始化HTTP连接池
func NewHTTPPool(self string) *HTTPPool {
    return &HTTPPool{
        self:     self,
        basePath: defaultBasePath,
    }
}

// Log 记录日志信息
func (p *HTTPPool) Log(format string, v ...interface{}) {
    clog.Info("[Server %s] %s", p.self, fmt.Sprintf(format, v...))
}

// 处理所有HTTP请求
```

```go
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)
    }
    clog.Info(fmt.Sprintf("%s %s", r.Method, r.URL.Path))
    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 := cache.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())
}

// Set 添加传入节点
func (p *HTTPPool) Set(peers ...string) {
    p.mu.Lock()
    defer p.mu.Unlock()

    // 实例化一致性哈希算法
    p.peers = consist.New(defaultReplicas, nil)
    p.peers.Add(peers...)
    p.httpGetters = make(map[string]*httpGetter, len(peers))

    for _, peer := range peers {
        p.httpGetters[peer] = &httpGetter{baseURL: peer + p.basePath}
    }
}

// PickPeer 获取key对应的节点
func (p *HTTPPool) PickPeer(key string) (peers.PeerGetter, bool) {
    p.mu.Lock()
    defer p.mu.Unlock()
    if peer := p.peers.Get(key); peer != "" && peer != p.self {
        clog.Info("Pick peer %s", peer)
        return p.httpGetters[peer], true
    }
    return nil, false
}

var _ peers.PeerPicker = (*HTTPPool)(nil)
```

抗缓存击穿

缓存击穿是一个热点的**Key**，有大并发集中对其进行访问，突然间这个**Key**失效了，导致大并发全部打在数据库上，导致数据库压力剧增

那么在一瞬间有大量请求**get(key)**，而且**key**未被缓存或者未被缓存在当前节点 如果不用**singleflight**，那么这些请求都会发送远端节点或者从本地数据库读取，会造成远端节点或本地数据库压力猛增。使用**singleflight**，第一个**get(key)**请求到来时，**singleflight**会记录当前**key**正在被处理，后续的请求只需要等待第一个请求处理完成，取返回值即可

```go
// 正在进行中或已经结束的请求
type call struct {
    wg  sync.WaitGroup
    val interface{}
    err error
}

type Group struct {
    mutex sync.Mutex
    calls map[string]*call
}

func (group *Group) Do(key string, fn func() (interface{}, error))
(interface{}, error) {
    group.mutex.Lock()
    if group.calls == nil {
        group.calls = make(map[string]*call)
    }

    // 检查是否有key的请求 如果有请求则等待并返回
    if c, ok := group.calls[key]; ok {
        group.mutex.Unlock()
        c.wg.Wait()
        return c.val, c.err
    }

    // 第一次key的请求 记录到表
    c := new(call)
    c.wg.Add(1)
    group.calls[key] = c
    group.mutex.Unlock()

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

    group.mutex.Lock()
    delete(group.calls, key)
    group.mutex.Unlock()

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

运行测试

不提供数据源，先在缓存中预设一些**key-value**

创建**group**，命名为**score**，储存一些人名和对应分数，之后启动服务器

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

// 创建group
func createGroup() *cache.Group {
    return cache.NewGroup("score", 2<<10,
        cache.GetterFunc(func(key string) ([]byte, error) {
            clog.Info("[SlowDB] search key", key)
            if v, ok := db[key]; ok {
                return []byte(v), nil
            }
            return nil, fmt.Errorf("%s not exist", key)
        }))
}

// 启动缓存服务器
func startCacheServer(addr string, addrs []string, group *cache.Group) {
    peers := service.NewHTTPPool(addr)
    peers.Set(addrs...)        // 添加节点信息
    group.RegisterPeers(peers) // 注册并启动HTTP服务
    clog.Info("cache is running at:", addr)
    clog.Fatal(http.ListenAndServe(addr[7:], peers))
}

// 启动API服务与用户交互
func startAPIServer(apiAddr string, group *cache.Group) {
    http.Handle("/api", http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
        key := r.URL.Query().Get("key")
        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())
    }))
    clog.Info("fronted server is running at", apiAddr)
    clog.Fatal(http.ListenAndServe(apiAddr[7:], nil))
}

func main() {
    var port int
    var api bool

    flag.IntVar(&port, "port", 8001, "cache server port")
    flag.BoolVar(&api, "api", false, "start a api server?")
    flag.Parse()

    apiAddr := "http://localhost:9999"
    addrMap := map[int]string{
        8001: "http://localhost:8001",
```

```
        8002: "http://localhost:8002",
        8003: "http://localhost:8003",
    }

    var addrs []string
    for _, v := range addrMap {
        addrs = append(addrs, v)
    }

    group := createGroup()
    if api {
        go startAPIServer(apiAddr, group)
    }
    startCacheServer(addrMap[port], addrs, group)
}
```

创建**3**个缓存节点，**1**个**API**节点，运行在不同的端口

编写一个测试脚本**:**

```
#!/bin/bash
trap "rm server;kill 0" EXIT
go build -o server
./server -port=8001 &
./server -port=8002 &
./server -port=8003 -api=1 &

sleep 2
echo ">>> start test"
curl "http://localhost:9999/api?key=Tom" &
curl "http://localhost:9999/api?key=Tom" &
curl "http://localhost:9999/api?key=Tom" &

wait
```

运行**:**

```
$ bash run.sh
[INFO][cache.go:34] 2022/12/19 08:49:56 cache is running at:
http://localhost:8003
[INFO][cache.go:50] 2022/12/19 08:49:56 fronted server is running at
http://localhost:9999
[INFO][cache.go:34] 2022/12/19 08:49:56 cache is running at:
http://localhost:8001
[INFO][cache.go:34] 2022/12/19 08:49:56 cache is running at:
http://localhost:8002
>>> start test
[INFO][pool.go:87] 2022/12/19 08:49:58 Pick peer %s http://localhost:8001
[INFO][pool.go:45] 2022/12/19 08:49:58 GET /_cache/score/Tom
[INFO][cache.go:21] 2022/12/19 08:49:58 [SlowDB] search key Tom
630630630
```