npc_hp110的专栏

个人资料

内胚层

访问： 73次
积分： 10
等级： BLOG 1
排名： 千里之外

原创： 1篇 转载： 0篇
译文： 0篇 评论： 0条

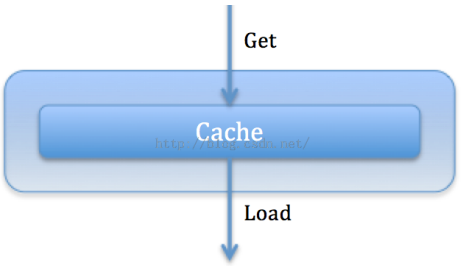
groupcache的设计和实现分析

2015-08-27 22:19 70人阅读 评论(0) 收藏 举报

groupcache的设计和实现分析

本文基于groupcache源码， 分析分布式缓存系统的设计和实现过程。本文代码大部分是来自groupcache的源码，但根据分析的需要做了少许改动。

1.本地缓存系统



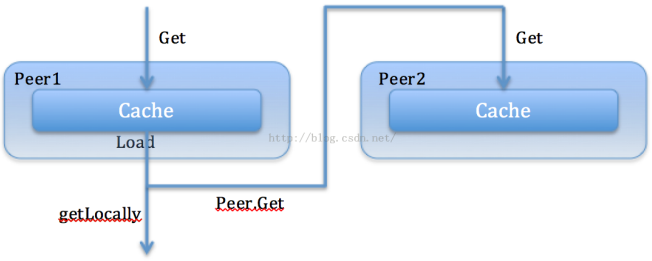
本地缓存系统的基本结构如上图所示。在内存中维护一个cache。查询时，首先查询cache中是否已经缓存查询结果。如果已经缓存，直接返回缓存结果，如果没有缓存，将查询结果Load到cache中，然后返回结果。代码如下：

[plain]

```
type Value interface{}
type Cache interface {
    Get(key string) (Value, bool)
    Add(key string, val Value)
}

type Group struct {
    cache Cache
}

func (g *Group) Get(key string) Value {
    //look up cache first
    val, cacheHit := g.lookupCache(key)
    if cacheHit {
        return val
    }
    //if miss, load value to cache
    val = g.lo
    return val
}

func (g *Group) lookupCache(key) (Value, bool) {
    val, ok := g.cache.Get(key)
    return val, ok
}
```

2.分布式缓存系统

在设计分布式缓存系统的时候，需要让key分布在不同的缓存节点上。当某节点收到查询请求时，如果该key归属于本节点，则在本节点获取查询结果；如果该key归属于其他节点，则本节点向归属节点获取查询结果 。如下图所示：



此分布式缓存架构引入了2个新的问题：

1. 如何判断查询的key归属哪个节点

2. 如何从其他节点获取数据

第1个问题实际上是一个路由问题，即给定key，路由到某个节点。这里忽略具体实现，将问题抽象出来，由2个接口表示（对应于groupcache的ProtoGetter和PeerPicker）：

```
[plain]
type Peer interface{
    Get(key string) Value
}

type Router interface{
    Route(key string) Peer
}
```

接口Peer表示一个远端节点，Get方法从远端节点查询数据；接口Router表示一个路由器，Route方法将给定key路由到其归属的远端节点，如果key归属于本节点，Route方法返回nil。基于此，改写Group结构体和Group.load方法：

```
[plain]
type Group struct {
    router Router
    cache  Cache
}
func (g *Group) load(key string) Value {
    peer := g.router.Route(key)
    if peer == nil {
        val := g.getLocally(key)
        g.cache.Add(key, val) //store result in cache
        return val
    }
    return peer.Get(g.name, key)
}
```

至此，一个分布式缓存框架搭建完成了，接下来分析具体细节的实现和功能的优化完善。

2.1. Cache的内存结构

Cache对象是内存中的一个容器，用来存放查询的结果。我们将Cache对象实现为一个List。另外，为了提升访问效率，用一个map结构来索引key及其对应的值：

```
[plain]
type ListCache struct {
    MaxEntries int
    l list.List
    m map[string]*list.Element
}
```

指定LRU为缓存的淘汰策略：即当Cache满了需要淘汰数据时，优先淘汰最老的数据。我们约定，链表中越靠近表头数据越新，越靠近表尾数据越老。因此在访问cache时，需要把命中的数据移至表头：

```
[plain]
type entry struct {
    key string
    val Value
}
func (c *ListCache) Get(key string) (Value, bool) {
    if e, hit := c.m[key]; hit {
        c.l.MoveToFront(e)
        return e.Value.(*entry).val, hit
    }
    return nil, false
}
func (c *ListCache) Add(key string, val Value) {
    if e, ok := c.m[key]; ok {
        c.l.MoveToFront(e)
    } else {
        ee := c.l.PushFront(&entry{key, val})
        c.m[key] = ee
        if c.l.Len() > c.MaxEntries {
            oldest := c.l.Back()
            c.l.Remove(oldest)
            delete(c.m, oldest.Value.(*entry).key)
        }
    }
}
```

2.2. 路由器的实现——一致性哈希

本节节选自：一致性哈希算法及其在分布式系统中的应用

路由器的基本功能是将key分布到不同的节点。很多方法可以实现这一点，最常用的方法是计算哈希。例如对于每次访问，可以按如下算法计算其哈希值：

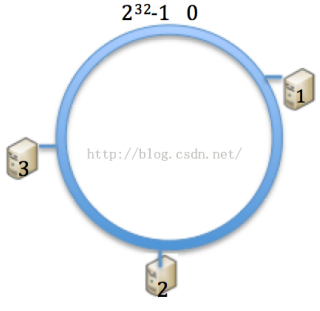
h= Hash(key) % N

这个算式计算每个key应该被路由到哪个节点，其中N为节点总数，所有节点按照0 – (N-1)编号。

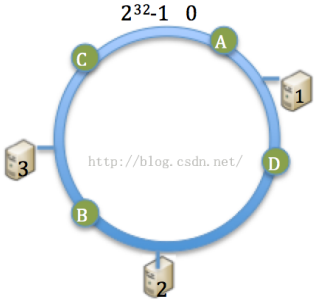
这个算法的问题在于容错性和扩展性不好。假设有一台服务器宕机了，那么为了填补空缺，要将宕机的服务器从编号列表中移除，后面的服务器按顺序前移一位并将其编号值减一，此时每个key就要按h = Hash(key) % (N-1)重新计算；同样，如果新增了一台服务器，虽然原有服务器编号不用改变，但是要按h = Hash(key) % (N+1)重新计算哈希值。因此系统中一旦有服务器变更，大量的key会被重定位到不同的服务器从而造成大量的缓存不命中。

一致性哈希算法就是解决这个问题的一种哈希方案。

简单来说，一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0 - 2^{32}-1（即哈希值是一个32位无符号整形），整个空间按顺时针方向组织。0和2^{32}-1在零点中方向重合。将各个节点使用H进行一个哈希，确定其在哈希环上的位置。假设有三台节点使用ip地址哈希后在环空间的位置如下：



接下来使用如下算法将数据访问路由到相应节点：将数据key使用相同的函数H计算出哈希值h，根据h确定此数据在环上的位置，从该位置沿环顺时针“行走”，第一台遇到的节点就是其应该路由到的节点。例如有A、B、C、D四个数据对象，经过哈希计算后，在环空间上的位置如下：



根据一致性哈希算法：数据A，C路由到节点1；数据D路由到节点2；数据B路由到节点3。

下面基于一致性哈希实现路由器。首先给每个节点命名：

```
[plain]

type Peer interface {
    Get(key string) Value
    Name() string
}
```

一致性哈希路由器HashRouter实现了Router接口（参考groupcache的HTTPPool）：

```
[plain]

type Hash func(key string) uint32
type HashRouter struct {
    self      Peer
    replicas  int
    hash      Hash
    locations []int        //Sorted
    hashMap   map[int]Peer //from location on the ring to Peer
}
func (r *HashRouter) Add(peers ...Peer) {
    for _, peer := range peers {
        for i := 0; i < r.replicas; i++ {
            hash := int(r.hash(peer.Name() + strconv.Itoa(i)))
            if _, ok := r.hashMap[hash]; !ok {
                r.locations = append(r.locations, hash)
            }
            r.hashMap[hash] = peer
        }
    }
    sort.Ints(r.locations)
}
func (r *HashRouter) Route(key string) Peer {
    hash := int(r.hash(key))
    // Binary search for appropriate replica.
    idx := sort.Search(
        len(r.locations),
        func(i int) bool { return r.locations[i] >= hash },
    )
    if idx == len(r.locations) {
        idx = 0
    }
    peer := r.hashMap[r.locations[idx]]
    if peer == r.self {
        return nil
    }
    return peer
}
```

其中HashRouter的locations是一个有序切片，保存所有节点在哈希环上的位置。hashMap保存从哈希环上的位置到节点(Peer)的索引。

2.3. 从远端节点获取缓存数据

我们采用HTTP协议在不同节点间传输缓存数据。数据请求的协议为：GET http://peer/key。HTTPPeer实现了Peer接口：

```
[plain]
```

```
<pre name="code" class="plain">type encoder interface {
    Marshal(v interface{}) []byte
    Unmarshal(data []byte, v interface{})
}
type HTTPPeer struct {
    baseURL string
    enc     encoder
}
func (p *HTTPPeer) Name() string {
    return p.baseURL
}
func (p *HTTPPeer) Get(key string) Value {
    u := fmt.Sprintf("%v%v", p.baseURL, url.QueryEscape(key))
    resp, _ := http.Get(u)
    defer resp.Body.Close()

    b := make([]byte, 0)
    resp.Body.Read(b)
    var val Value
    p.enc.Unmarshal(b, &val)
    return val
}
```

2.4. 处理远端节点的查询请求

每个节点既可以向远端节点请求数据，同时也要处理远端节点的查询请求。HTTPPeer负责处理节点间的数据传输，因此需要处理远端节点的查询请求。

```
[plain]
func (p *HTTPPeer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    key := strings.SplitN(r.URL.Path, "/", 1)[0]
    group := GetGroup()
    val := group.Get(key)

    body := p.enc.Marshal(val)
    w.Write(body)
}
```

2.5. 数据编码

在2.3中我们定义了数据编码器，但未实现它。实际上golang有很多成熟的编码库，如json, gob, protobuf等。groupcache使用的是protobuf。

2.6. 数据Load

当查询数据不在缓存中时，需要从数据源加载数据。加载数据的过程需要用户自定义，因此定义Loader接口，用户自行实现该接口：

```
[plain]
type Loader interface {
    Load(key string) Value
}
type Group struct {
    loader Loader
    router Router
    cache  Cache
}

func (g *Group) getLocally(key string) Value {
    return g.loader.Load(key)
}
```

2.6. Alltogether and Startup

要在一个节点上启动上述的分布式缓存系统，包含如下步骤：

1. 初始化一个Group对象，指定Loader

2. 初始化一个ListCache对象，并将其赋值给Group对象的cache

3. 初始化一个HashRouter对象，并将其赋值给Group对象的router

4. 为集群的每个节点初始化一个HTTPPeer对象，并将它们添加到HashRouter对象。将本节点对应的HTTPPeer对象赋值给HashRouter对象的self

5. http监听指定端口，并指定处理函数为HTTPPeer对象的ServeHTTP方法

可以做一些适当的封装，让此缓存系统使用更加方便。

2.7. 错误处理

3. 功能优化和完善

至此我们已经实现了一个简陋的分布式缓存系统，接下来我们为这个分布式缓存系统增加一些新的功能（groupcache支持的功能）。

3.1 命名空间

给系统增加命名空间，使不同命名空间的缓存互相独立。为此，给Group对象增加一个名称，每个节点可以创建多个不同名称的Group对象，每个对应一个命名空间。

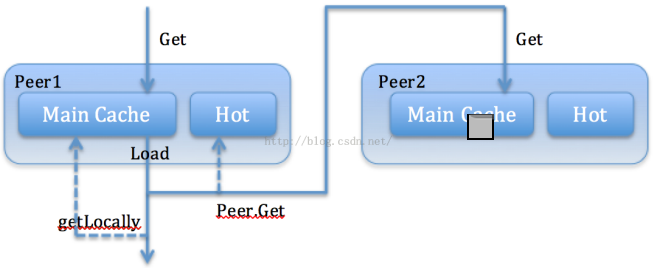
```
[html]

var groups = make(map[string]*Group)
func GetGroup(name string) *Group {
    return groups[name]
}
type Group struct {
    name   string
    loader Loader
    router Router
    cache  Cache
}
```

节点间的数据请求协议变为：GET http://peer[illegible]oupname/key 。

```
[plain]

type Peer interface {
    Get(group, key string) Value
    Name() string
}
func (p *HTTPPeer) Get(group, key string) Value {
    u := fmt.Sprintf("%v%v/%v", p.baseURL, url.QueryUnescape(group), url.QueryEscape(key))
    ...
}
func (p *HTTPPeer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    parts := strings.SplitN(r.URL.Path, "/", 2)
    groupName := parts[0]
    key := parts[1]
    group := GetGroup(groupName)
    val := group.Get(key)

    body := p.enc.Marshal(val)
    w.Write(body)
}
```

3.2 热数据扩散

前面提到，在设计分布式缓存系统的时候，需要让key分布在不同的缓存节点上。如果某些key的访问量特别大，大量请求会路由到该key归属的节点，可能导致该节点无法处理而瘫痪。因此groupcache增加了热数据自动扩散功能。通过在Group中增加一个Cache对象，如果节点的某些key访问特别频繁，而这些key的归属节点不在本节点，此时会将这些key的查询结果缓存到新增的Cache对象中。如下图所示：



代码如下：

```
[plain]

type Group struct {
    name      string
    loader    Loader
    router    Router
    mailCache Cache
    hotCache  Cache
}
func (g *Group) lookupCache(key string) (Value, bool) {
    val, ok := g.mailCache.Get(key)
    if ok {
        return val, ok
    }
    val, ok = g.hotCache.Get(key)
    return val, ok
}
func (g *Group) load(key string) Value {
    peer := g.router.Route(key)
    if peer == nil {
        val := g.getLocally(key)
        g.mailCache.Add(key, val) //store result in cache
        return val
    }
    val := peer.Get(g.name, key)
    if rand.Intn(10) == 0 {
        g.hotCache.Add(val)
    }
    return val
}
```

3.3 缓存Load过程合并机制，避免“惊群效应”

如果某节点对某相同的key存在大量并发查询相，而该key的值不在缓存中，这些并发查询就会触发大量的Load过程（从数据源或远端节点加载数据）。但这些Load过程都是加载相同的数据，造成了资源的大量浪费。为了避免这种现象，需要在查询触发Load过程前，先判断是否已经有相同的 Load过程正在运行。如果存在，本次Load不执行，而是等正在运行的Load过程完成后直接使用其结果。

```
[plain]

type loadResult struct {
    wg  sync.WaitGroup
    val interface{}
}
type LoadGroup struct {
    mu sync.Mutex             // protects m
    m  map[string]*loadResult // map key to loading process
}
type loadFn func() Value
func (g *LoadGroup) Do(key string, load loadFn) Value {
    g.mu.Lock()
    if c, ok := g.m[key]; ok { // if it is loading
        g.mu.Unlock()
        c.wg.Wait() // wait loaded
        return c.val
    }
    c := new(loadResult)
    c.wg.Add(1)
    g.m[key] = c // notify that it is loading
    g.mu.Unlock()

    c.val = load() //start loading
    c.wg.Done()    //notify loaded

```

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

    return c.val
}
```

上面的代码定义了一个LoadGroup对象，调用它的Do方法可以合并多个Load过程，从而避免“惊群效应”。

```
[plain]
type Group struct {
    name      string
    loader    Loader
    router    Router
    mailCache Cache
    hotCache  Cache
    loadGroup LoadGroup
}
func (g *Group) load(key string) Value {
    val := g.loadGroup.Do(key, func() Value {
        peer := g.router.Route(key)
        if peer == nil {
            val := g.getLocally(key)
            g.mailCache.Add(key, val) //store result in cache
            return val
        }
        val := peer.Get(g.name, key)
        if rand.Intn(10) == 0 {
            g.hotCache.Add(key, val)
        }
        return val
    })
    return val
}
```

3.4. 缓存容量控制

前面提到，Cache对象在内存中是以List的结构组织的。通过ListCache.MaxEntries字段控制缓存的容量。然而该字段只能限制缓存对象的数量，不能限制缓存的实际内存容量。要控制缓存的实际内存容量，需要知道缓存对象的大小。缓存的对象结构如下：

```
[plain]
type entry struct {
    key string
    val Value
}
```

key可以通过len(key)计算大小，要计算val的大小，需要 val对象实现一下接口：

```
[html]
type ByteView interface{
    Len() int
}
```

具体实现参考groupcache源码

3.5. 统计功能

参考groupcache源码

猜你在找

《C语言/C++学习指南》加密解密篇（安全相关算法）

大数据时代的<集装箱式>架构设计与Docker潮流

Ceph—分布式存储系统的另一个选择

疯狂IOS讲义之Objective-C面向对象设计

C语言系列之 字符串相关算法

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap