# 负载均衡

## 源代码阅读

### 接口定义

kubernetes/pkg/proxy/winuserspace/loadbalancer.go

定义了一个interface的接口

type LoadBlancer interface {  
 NextEndpoint(service proxy.ServicePortName,  
 srcAddr net.Addr,  
 sessionAffinityReset bool) (string, error)  
 NewService(service proxy.ServicePortName,  
 sessionAffinityType v1.ServiceAffinity,  
 stickyMaxAgeMinutes int) error  
 DeleteService(service proxy.ServicePortName)  
 CleanupStaleStickySessions(service proxy.ServicePortName)  
}

可以看出定义了四个方法，其中最重要的是sessionAffinity。sessionAffinity是一种分发策略：

基于客户端IP地址进行会话保持/关联的模式，即第1次将某个客户端发起的请求转发到后端的某个Pod上，之后从相同的客户端发起的请求都将被转发到后端相同的Pod上。

那么方法中的sessionAffinityType和sessionAffinityReset以及stickMaxAgeMinutes就十分容易理解了:

### 轮询算法

kubernetes/pkg/proxy/winuserspace/roundrobin.go

首先定义了两个错误：

var(  
 ErrMissingServiceEntry = errors.New("missing service entry")  
 ErrMissingEndpoints = errors.New("missing endpoints)  
)

其次定义affinityState:

type affinityState struct {  
 clientIP string  
 endpoint string  
 lastUsed time.Time  
}

很好理解。

然后对策略进行定义：

type affinityPolicy struct {  
 affinityType v1.ServciceAffinity  
 affinityMap map[string]\*affinitystate  
 ttlSeconds int  
}

显然构造了一个从client ip到状态的map（为什么？在affinityState中明明有clientIp啊）

然后定义balancerState字段：

type balancerState struct {  
 endpoints []string // a list of "ip:port" style strings  
 index int // current index into endpoints  
 affinity affinityPolicy  
}

最终马上定义了一个LoadBalancerRR，从而实现了轮询的负载均衡算法。

type LoadBalancerRR struct {  
 lock sync.RWMutex  
 services map[proxy.ServicePortName]\*balancerState  
}

注意，定义了一个读写锁。其中第二字段。

最终，定义了负载均衡器：

var \_ LoadBalancer = &LoadBalancerRR{}

从上描述可以看出，字段之间的逻辑结果如下：

LoadBalancerRR是最终的字段，定义了从proxy.ServicePortName到balancerState。每一个balancerState含有其策略、指向目前端点的index并且其中含有endpoints的字符串。逻辑很好理解。

func newAffinityPolicy(affinityType v1.ServiceAffinity,  
 ttlSeconds int) \* affinityPolicy {  
 return &affinityPolicy{  
 affinityType: affinityType,  
 affinityMap: make(map[string]\*affinityState),  
 ttlSeconds: ttlSeconds,  
 }  
}

可以看出这个函数就是进行字段初始化。

func NewLoadBalancerRR() \*LoadBalancerRR {  
 return &LoadBalancerRR{  
 services: map[proxy.ServicePortName]\*balancerState{},  
 }  
}

仍然是初始化。

func (lb \*LoadBalancerRR) NewService  
 (svcPort proxy.ServicePortName,   
 affinityType v1.ServiceAffinity,   
 ttlSeconds int) error {  
 klog.V(4).Infof("LoadBalancerRR NewService %q", svcPort)  
 lb.lock.Lock()  
 defer lb.lock.Unlock()  
 lb.newServiceInternal(svcPort, affinityType, ttlSeconds)  
 return nil  
}

对interface中的NewService方法进行定义了，通过读者写者锁实现同步操作，其中调用了newServiceInternal，如下：

func (lb \*LoadBalancerRR) newServiceInternal  
 (svcPort proxy.ServicePortName,   
 affinityType v1.ServiceAffinity,   
 ttlSeconds int) \*balancerState {  
 if ttlSeconds == 0 {  
 ttlSeconds = int(v1.DefaultClientIPServiceAffinitySeconds)   
 }  
  
 if \_, exists := lb.services[svcPort]; !exists {  
 lb.services[svcPort] = &balancerState{affinity: \*newAffinityPolicy(affinityType, ttlSeconds)}  
 klog.V(4).Infof("LoadBalancerRR service %q did not exist, created", svcPort)  
 } else if affinityType != "" {  
 lb.services[svcPort].affinity.affinityType = affinityType  
 }  
 return lb.services[svcPort]  
}

新建服务操作，无任何较复杂的业务逻辑。

func (lb \*LoadBalancerRR) DeleteService(svcPort proxy.ServicePortName) {  
 klog.V(4).Infof("LoadBalancerRR DeleteService %q", svcPort)  
 lb.lock.Lock()  
 defer lb.lock.Unlock()  
 delete(lb.services, svcPort)  
}

删除服务操作，无任何较复杂的业务逻辑。

func isSessionAffinity(affinity \*affinityPolicy) bool {  
 // Should never be empty string, but checking for it to be safe.  
 if affinity.affinityType == "" || affinity.affinityType == v1.ServiceAffinityNone {  
 return false  
 }  
 return true  
}

判断操作。

下面实现interface中的NextEndPoint方法。

func (lb \*LoadBalancerRR) NextEndpoint(svcPort proxy.ServicePortName,   
 srcAddr net.Addr,   
 sessionAffinityReset bool) (string, error) {  
 // Coarse locking is simple. We can get more fine-grained if/when we  
 // can prove it matters.  
 lb.lock.Lock()  
 defer lb.lock.Unlock()  
  
 state, exists := lb.services[svcPort]  
 if !exists || state == nil {  
 return "", ErrMissingServiceEntry  
 }  
 if len(state.endpoints) == 0 {  
 return "", ErrMissingEndpoints  
 }  
 klog.V(4).Infof("NextEndpoint for service %q, srcAddr=%v: endpoints: %+v", svcPort, srcAddr, state.endpoints)  
  
 sessionAffinityEnabled := isSessionAffinity(&state.affinity)  
  
 var ipaddr string  
 if sessionAffinityEnabled {  
 // Caution: don't shadow ipaddr  
 var err error  
 ipaddr, \_, err = net.SplitHostPort(srcAddr.String())  
 if err != nil {  
 return "", fmt.Errorf("malformed source address %q: %v", srcAddr.String(), err)  
 }  
 if !sessionAffinityReset {  
 sessionAffinity, exists := state.affinity.affinityMap[ipaddr]  
 if exists && int(time.Since(sessionAffinity.lastUsed).Seconds()) < state.affinity.ttlSeconds {  
 // Affinity wins.  
 endpoint := sessionAffinity.endpoint  
 sessionAffinity.lastUsed = time.Now()  
 klog.V(4).Infof("NextEndpoint for service %q from IP %s with sessionAffinity %#v: %s", svcPort, ipaddr, sessionAffinity, endpoint)  
 return endpoint, nil  
 }  
 }  
 }  
 // Take the next endpoint.  
 endpoint := state.endpoints[state.index]  
 state.index = (state.index + 1) % len(state.endpoints)  
  
 if sessionAffinityEnabled {  
 var affinity \*affinityState  
 affinity = state.affinity.affinityMap[ipaddr]  
 if affinity == nil {  
 affinity = new(affinityState) //&affinityState{ipaddr, "TCP", "", endpoint, time.Now()}  
 state.affinity.affinityMap[ipaddr] = affinity  
 }  
 affinity.lastUsed = time.Now()  
 affinity.endpoint = endpoint  
 affinity.clientIP = ipaddr  
 klog.V(4).Infof("Updated affinity key %s: %#v", ipaddr, state.affinity.affinityMap[ipaddr])  
 }  
  
 return endpoint, nil  
}

这段代码的逻辑也比较简单，说白了就是轮询的核心功能。

## cpp实现

* 日志库的实现，找个第三方库（服务发现也能用）
* 数据结构的定义

需要与轩轩交流。(todo)