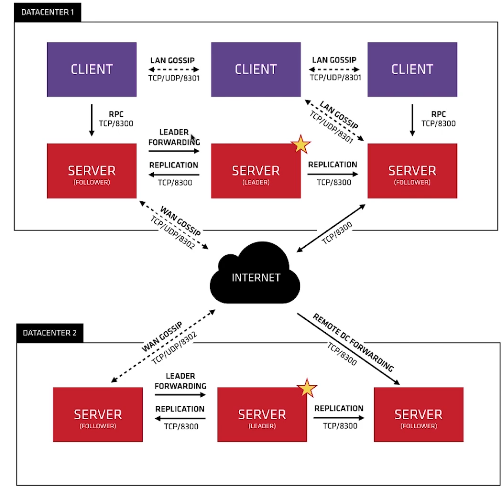
**周记**

**consul 注册中心**

**consul介绍**

consul是一种分布式服务工具，为了避免单点故障常常以集群方式部署。

它提供的关键功能有：服务发现、运行状态检查、kv存储（配置中心）、安全通信、和多数据中心等功能

##### consul的集群架构

在consul集群中有server和client两种节点，server节点保存数据，client节点负责健康检查及转发数据rpc请求到server，server节点中有一个leader节点和多个follower节点。leader会将数据同步到follower节点。leader节点的产生是通过raft（选举）算法实现的。

它可以是多数据中心通信的，同一数据中心的节点通过通过局域网gossip协议和tcp,udp通信，使用8301端口，而不同数据中心通过广域网gossip协议，和tcp ,udp通信，使用8302端口。

##### 两个协议

###### gossip协议

在传统的监控，每个节点都向server发送消息,随着节点的增加，server压力也变大。而在consul,所有节点都运行gossip协议，加入集群中，收发gossip消息。每隔一段时间，每个节点随机选取几个节点发送gossip消息，这样一段时间过后，整个集群都能收到gossip消息。

###### raft一致性算法

为了实现集群中多个ConsulServer中的数据保持一致性，consul使用了基于强一致性的RAFT算法

在Raft中，任何时候一个服务器可以扮演下面角色之一：

　　1. Leader: 处理所有客户端交互，日志复制等，一般一次只有一个Leader.

　　2. Follower: 类似选民，完全被动

　　3. Candidate（候选人）: 可以被选为一个新的领导人。

Leader全权负责所有客户端的请求，以及将数据同步到Follower中（同一时刻系统中只存在一个Leader）。Follower被动响应请求RPC，从不主动发起请求RPC。Candidate由Follower向Leader转换的中间状态。

代码：

package structs  
  
import "math"  
  
//该模块用于计算程序运行所需的时间  
  
func calcLatency(task pb.Basic\_TaskType, from string, to string, net comnet.NetV4) float64 {  
 node := net[from] //得到最优时延矩阵  
 //计算时延  
 w := TypeTaskParamsDict[task].Weight //需要处理的数据量的大小  
 x, y, u := node.Rc[0], node.Rc[1], node.Rc[2] //计算资源 cpu核 cpu的频率 cpu的利用率  
 execDelay := 1e3 \* (w[0]\*math.Exp(-w[1]\*(x\*(1-u)+w[2]\*y)) + w[3]) //执行时延  
 delay := execDelay  
 //  
 if from != to {  
 link := net[from].Links[to]  
 esDelay := link.EsDelay  
 transDelay := float64(TypeTaskParamsDict[task].DataSize) / link.Rate  
 delay += esDelay + transDelay //总时延  
 }  
 return delay  
}

package structs  
  
import (  
 "encoding/json"  
 "fmt"  
 "github.com/shirou/gopsutil/cpu"  
 "github.com/shirou/gopsutil/disk"  
 "github.com/shirou/gopsutil/mem"  
 "github.com/shopspring/decimal"  
 "log"  
 "time"  
)  
  
const *nums* = 4  
  
type Cpu struct {  
 Info []cpu.InfoStat `json:"info"`  
 LogicalCount int `json:"logical\_count"`  
 PhysicalCount int `json:"physical\_count"`  
 Usage []float64 `json:"usage"`  
 Time []cpu.TimesStat `json:"time"`  
}  
  
//获取cpu的使用率  
func GetCpuPercent(cpuChan chan float64) {  
 var second = 5  
 var sum float64  
 for i := 0; i < second; i++ {  
 //创建5s内cpu总使用率  
 percent, err := cpu.Percent(time.*Second*, false)  
 if err != nil {  
 log.Fatal(err)  
 }  
 sum += percent[0]  
 time.Sleep(time.*Second*)  
 }  
 cpuPer, \_ := decimal.NewFromFloat(sum / float64(second)).Round(2).Float64()  
 cpuChan <- cpuPer  
  
}  
  
func GetMemPercent() float64 {  
 memInfo, \_ := mem.VirtualMemory()  
 return memInfo.UsedPercent  
}  
  
func GetDiskPercent() float64 {  
 parts, \_ := disk.Partitions(true)  
 diskInfo, \_ := disk.Usage(parts[0].Mountpoint)  
 return diskInfo.UsedPercent  
}  
  
func GetCpuInfo() float64 {  
 physicalCnt, \_ := cpu.Counts(false) // cpu物理核数  
 logicalCnt, \_ := cpu.Counts(true) // cpu逻辑核数  
 fmt.Printf("物理核数:%d 逻辑核数:%d\n", physicalCnt, logicalCnt)  
 // 获取 3s 内的总 CPU 使用率和每个 CPU 各自的使用率  
 totalPercent, \_ := cpu.Percent(5\*time.*Second*, false) // 总 CPU 使用率  
 perPercents, \_ := cpu.Percent(5\*time.*Second*, true) // 每个 CPU 各自的使用率  
 fmt.Printf("total percent:%v per percents:%v", totalPercent, perPercents)  
 infos, \_ := cpu.Info()  
 for \_, info := range infos {  
 data, \_ := json.MarshalIndent(info, "", " ")  
 fmt.Print(string(data))  
 }  
 y := infos[0].Mhz / 1000 //GHZ  
 //FLOAS=核数\*单核主频\*CPU单个周期浮点计算值 单位GFLOPS  
 tatalCompting := float64(physicalCnt) \* (1 - totalPercent[0]\*1e-2) \* y \* *nums* return tatalCompting  
  
}

package gs  
  
import (  
 "WORK/test/wbg3/wbg"  
 "fmt"  
)  
  
type GS struct {  
}  
  
type People struct {  
 Name string // 名字  
 LikePeople []string // 喜好列表  
 CurrentLike int // 后面算法记录当前表白对象时使用  
 Friend string // 当前匹配对象  
}  
  
// type Graph [][]WeightedEdge  
// type WeightedEdge struct {  
// Src string //男  
// Des string //女  
// Cost float64 //权重  
// }  
// var Graph wbg.Graph  
  
func SortLike(graph wbg.Graph) (boyArr []\*People, girlArr []\*People) {  
 e := len(graph)  
 m := e - len(graph[0])  
 // n := e - m  
 // var boyArr1, girlArr1 []\*People  
 boyArr1 := make([]\*People, 0)  
 girlArr1 := make([]\*People, 0)  
 // likepeople := make([]string, 0)  
 for i, k := range graph {  
 if i < m {  
 // boyArr[i].Name = append(boyArr[i].Name,k[i].Src)  
 // boyArr1[i].Name = k[i].Src  
 // 升序排列  
 likepeople := make([]string, 0)  
 for \_, v := range k {  
 people := v.Des  
 likepeople = append(likepeople, people)  
 // boyArr1[i].LikePeople = append(boyArr1[i].LikePeople, people)  
 }  
 // boyArr1[i] = &People{  
 boyArr1 = append(boyArr1, &People{  
 Name: k[0].Src,  
 LikePeople: likepeople,  
 CurrentLike: 0,  
 Friend: "",  
 })  
 // 要对邻接矩阵的按照cost标准进行升序排列  
  
 } else {  
 likepeople := make([]string, 0)  
 for \_, v := range k {  
 people := v.Des  
 likepeople = append(likepeople, people)  
 // boyArr1[i].LikePeople = append(boyArr1[i].LikePeople, people)  
 }  
 girlArr1 = append(girlArr1, &People{  
 Name: k[0].Src,  
 LikePeople: likepeople,  
 CurrentLike: 0,  
 Friend: "",  
 })  
 }  
  
 }  
 return boyArr1, girlArr1  
}  
func (gs \*GS) GaleShapley(boyArr []\*People, girlArr []\*People) {  
 // func (gs \*GS) GaleShapley(graph Graph) {  
 // E := len(graph) //一维数组的元素个数  
 // boyArr := make([]\*People, E)  
 // girlArr := make([]\*People, E)  
  
 for {  
 // 找到一个没有对象, 且未全部表白的男生  
 var searchBoy \*People  
 for \_, boy := range boyArr {  
 if boy.Friend != "" { // 当前男孩已经有对象了  
 continue  
 }  
 // 男孩向所有女生表白过了  
 if boy.CurrentLike >= len(boy.LikePeople) {  
 fmt.Printf("%v\n", boy.CurrentLike)  
 continue  
 }  
 searchBoy = boy  
 break  
 }  
 if searchBoy == nil { // 已经全部有对象了, 结束  
 break  
 }  
 // 男生向女生依次表白  
 var i int  
 for i := searchBoy.CurrentLike; i < len(searchBoy.LikePeople); i++ {  
 girlName := searchBoy.LikePeople[i]  
 // 找到这个女孩  
 girl := searchPeople(girlArr, girlName)  
 if girl == nil { // 习惯了, 判下空  
 continue  
 }  
 if girl.Friend == "" { // 若女孩没有对象, 则直接配对  
 girl.Friend = searchBoy.Name  
 searchBoy.Friend = girl.Name  
 break  
 } else { // 若女孩有对象, 看下 girl 更喜欢谁  
 searchBoyIdx := searchNameIndex(girl.LikePeople, searchBoy.Name)  
 girlFriendIdx := searchNameIndex(girl.LikePeople, girl.Friend)  
 if girlFriendIdx < searchBoyIdx { // 保持当前  
 continue  
 } else { // 重新组队  
 girlFriend := searchPeople(boyArr, girl.Friend)  
 if girlFriend != nil { // 分手了  
 girlFriend.Friend = ""  
 girlFriend.CurrentLike++  
 }  
 girl.Friend = searchBoy.Name  
 searchBoy.Friend = girl.Name  
 break  
 }  
 }  
  
 }  
 searchBoy.CurrentLike = i  
 }  
}  
func searchPeople(peopleArr []\*People, name string) \*People {  
 for \_, people := range peopleArr {  
 if people.Name == name {  
 return people  
 }  
 }  
 return nil  
}  
  
func searchNameIndex(nameArr []string, name string) int {  
 for i, tmpName := range nameArr {  
 if tmpName == name {  
 return i  
 }  
 }  
 return -1  
}

ackage main  
  
import (  
 "WORK/test/wbg2/gs"  
 "WORK/test/wbg2/wbg"  
 "fmt"  
)  
  
  
var GS gs.GS  
  
func main() {  
  
 girlip := wbg.GirlIP{"w1", "w2", "w3", "w4"}  
 // girlip := wbg.GirlIP{"w1"}  
 boyip := wbg.BoyIP{"m1", "m2", "m3"}  
 // 调用pf 函数  
  
 wbglist, \_ := wbg.Compile(boyip, girlip)  
  
 for \_, v := range wbglist {  
 fmt.Printf("Graph长度是=%v \n", len(v.Graph))  
 graph := v.Graph  
 boyArr1, girlArr1 := gs.SortLike(graph)  
 GS.GaleShapley(boyArr1, girlArr1)  
 fmt.Println("名字:\t最终匹配对象:")  
 for \_, v := range boyArr1 {  
 fmt.Printf("%v\t%v\n", v.Name, v.Friend)  
 }  
  
 fmt.Println("")  
 for \_, v := range girlArr1 {  
 fmt.Printf("%v\t%v\n", v.Name, v.Friend)  
 }  
  
 for i, k := range v.Graph {  
 fmt.Printf("Graph[i]宽度是=%v \n", len(k))  
  
 for j, m := range k {  
 fmt.Printf("Graph[%v][%v]=%v \n", i, j, m)  
  
 }  
 }  
 fmt.Println(v)  
 }  
 fmt.Println(wbglist)  
 fmt.Printf("长度是%v", len(wbglist))  
  
}