首届

云原生编程挑战赛

挑战Serverless极致弹性

阿里云　阿里云原生 | TIANCHI天池

CONTENT 挑战 Serverless 极致弹性

目录

CATALOGUE

1 比赛攻略——睡衣小英雄

初赛赛道 3：服务网格控制面分治体系构

1 赛题背景分析及理解

赛题的本质是希望在分布一个应用时，在每个节点上，应用或者服务器的分布尽可能均匀，将资源最大可能进行利用。这里有一对矛盾，如何每个节点都去加载所有的服务，那么连接数可以很均衡，但是会浪费大量的内存。

如果节点加载的服务不重复，那么使用的内存最为节省，但是连接压力就可能极不平均。因此，赛题的目的就是在两者间取得平衡。

根据赛题的评分公式： 得分=(M加载内存/M服务内存)*(Dif内存标准差+Dif连接标准差)

也就说，在分配服务到Pilot上时，应该尽可能使加载的总内存最小,平时内存和连接数在各个节点间应该尽可能平均。

2 核心思路

由于评分公式很明确，因此可以采用随机方法，生成若干种解，然后计算每个解的分数，取最好值即可。

同时要充分利用P1和P2的timeout时间，尽量在给定时间内充分计算，计算和尝试尽可能多的解，以取得最好值。

3 粒子群算法

如果单纯的采用随机，得不到太好的效果。因此这里使用粒子群算法。在P1阶段，随机生成500个粒子。然500粒子在可行解的范围内按照标准粒子群算法移动，并不停更新个体最优和全局最优。

在P1 timeout前，将全局最优返回（个体和全局最优使用得分公式计算）。

在P2 阶段，由于timeout时间很短，因此使用50个粒子。其中将P1阶段的最优粒子 也加入P2阶段的计算中。让所有粒子在解空间中按照标准粒子群算法移动，并在P2 timeout前返回全局最优。

在程序尝试将App分配到Pilot时，会选择最合适的Pilot进行分配，分配原则依据以下公式：

Base[0]*incrMem + Base[1]*math.Pow(mem*appMem, 1.0/2.0) + Base[2]*math.Sqrt(conn*app.Sidercar)

各个参数含义如下:

- Base[0],Base[1],Base[2] 粒子群的学习参数，粒子群的学习过程，就是调整这3个参数。
- incrMem表示将整个App分配到这个Pilot时，增加的整体内存。
- appMem表示这个待分配应用的内存

» mem表示这个Pilot的内存

» conn表示这个Pilot的sidecar数量

» app.Sidecar表示 这个App的sidecar数量

上述公式返回了将一个app分配到一个pilot的评分，当要分配一个app时，总是选择这个评分最小的Pilot进行分配。

此公式的意义在于：要选择的Pilot必然是：

1. 增加的内存比较小
2. Pilot自己的内存比较小
3. Pilot资深的连接数比较小

而 Base[0],Base[1],Base[2]几个未知数，则需要使用粒子群求解。

4 关键代码

P1阶段初始化500个例子(注意，为了充分利用多核并行，一半的粒子通过InitSwamPilotsChannel ，在另外一个线程中运行)

```
for  i := 1; i < 500; i++ {
var  base []float64 = make([]float64, BaseCount)
var  pbest []float64 = make([]float64, BaseCount)
var  pv []float64 = make([]float64, BaseCount)
for  i := 0; i < BaseCount; i++ {
base[i] = rand.Float64() * maxRnd
}
var  Point = &Piont{base, pbest, pv, MAXFLOAT}
pilotsSwam = append(pilotsSwam, Point)
if i%2 == 0 {
InitSwamPilotsChannel <- Point
} else {
Point.Score(OrderApplist)
}
now := time.Now().Unix()
if now-P1Begin > P1Timeout {
break
}
}
```

为每一个例子打分，并更新局部最优和全局最优

```
func (p *Pilots) ScoreApps(applist AppArray) {
appCount := len(applist)
```

```
for  i, app := range applist {
p.assignApp(app, i, appCount)
}
score := p.MyScore()

if score < p.Piont.PBestScore {
p.Piont.PBestScore = score
copy(p.Piont.PBest, p.Piont.Base)
}

BestMutex.Lock()
if score < MinScoreP1 {
LastPBestPolit = p
MinScoreP1 = score
copy(BestBase, p.Piont.Base)
log.Printf("score:%f", score)
}
BestMutex.Unlock()
}
```

粒子向全局最优移动:

```
func (p *Pilots) FlyToBest() {
BestMutex.RLock()
for  i := 0; i < len(p.Piont.Base); i++ {
p.Piont.PVelocity[i] = Omiga*p.Piont.PVelocity[i] + (p.Piont.PBest[i]-p.Piont.Base[i])*rand.Float64()*C1 +
(BestBase[i]-p.Piont.Base[i])*rand.Float64()*C2
if p.Piont.PVelocity[i] > MaxVelocity {
p.Piont.PVelocity[i] = MaxVelocity
}
if p.Piont.PVelocity[i] < -MaxVelocity {
p.Piont.PVelocity[i] = -MaxVelocity
}

p.Piont.Base[i] += p.Piont.PVelocity[i]
if p.Piont.Base[i] > maxRnd {
p.Piont.Base[i] = maxRnd
```

```
}
if p.Piont.Base[i] < 0 {
p.Piont.Base[i] = 0
}
}
BestMutex.RUnlock()
}
```

5 比赛经验总结和感想

本赛题是一个最优化问题，但是显然无法通过暴力搜索取得全局最优，因此使用启发式的粒子群算法求解。除了Pilot的评分公式模型外，其余都是采用自学习的方式进化选择，得到最优解。

初赛赛道 1：实现一个分布式统计和过滤的链路

1 赛题背景分析及理解

在本赛题中，需要对2个大Trace文件中的数据进行过滤和合并。最终结果中的每一条Trace都需要由两个输入文件共同产生。因此，不可避免的需要在Backend节点进行数据整合。

但由于数据量巨大，两个前端节点必须对数据进行过滤，只将有错的Trace传入后端节点。后端节点收到一个错误的Trace后，将会向另外一侧的节点请求相同的TraceId的数据，因此，每个前端节点需要对处理过的数据进行一定的缓存。

由于前端节点是处理的主力，因此程序优化的重点放在前端节点，后端节点的压力非常小，对后端的优化，意义不大。

2 核心思路

2.1 数据的基本处理

整个数据处理使用标准的按行读取模式。按字符或者块处理，会有更好的性能，但是同时会大大提升编码难度和程序的可维护性。在本竞赛中，本程序既使用良好的按行处理风格，同时又取得较好的性能。

2.2 数据读取

使用RANGE读取数据。根据赛题中给出的CPU数量，这里将整个文件为了2份处理，以充分利用CPU。

为了保证数据处理的正确性，使用RANGE读取数据时，需要首先查找"\n"所在位置，RANGE读取的中间位置保证是一个"\n"， 确保处理不会出错。

2.3 线程模型

使用2个线程读取数据，每次读取1M，读取后立即按行分割，并判断这个span是否有错。如果有错，在构造Trace时，将整个Trace标记为ERROR。

一个节点上的2个线程分别处理数据的前半段和后半段，完全不交互，各自独立，没有同步开销。

2.4 通信模式

前端节点和后端节点之间，主要采用tcp长连接通信。确保通信的高效性，对于一些数量不多的消息通知，采用http方式通信。

2.5 避免数据复制

通过自定义Scanner，确保数据可以在没有二次复制的情况下被处理。在MyScan中，使用的Buffer和默认Scanner的Buffer是完全不一样的。在MyScan中，Buffer是一块可重复利用的空间，很大，每个线程1G。但每次从http读取数据只读取1M,读取后，追加到Buffer中。同时向外返回指针，这样就避免了数据复制。

2.6 前端节点的协调

为了确保多个节点可以同步处理数据，每个节点在Buffer溢出重写时，需要向Backend节点申请， Backend协调多个节点。使多个前端节点可以在几乎同时重写自己的Buffer，避免一个处理太快，一个处理太慢的问题，导致数据出错的问题。目前，1G空间在当前线上环境中，已经可以处理data1和data2大小的差异。如果将来data1和data2的大小差异进一步扩大，导致无法使用相同大小buffer同步的时候，可以使用正在使用的Trace数量进行两个节点的同步，会有更好的效果（增加Buffer大小也可以增强对data1和data2大小差异的容忍性，但是Buffer本身不适合无限制扩大）。

2.7 数据正确性保证

为了确保数据正确性，2个处理线程在交接点上进行重复读，目前各重复读取32M，这样如果在这段区间内出现错误的Trace，总是可以被其中一个线程完整的捕捉到的。但同时，由于重复读取的存在，Backend节点就有可能收到重复的span行，这就需要在Backend就行二次过滤。当Backend收到每一个span时，都会根据时间排序，并丢弃已经收到过的span。

2.8 使用更快的map

所有的Trace都根据TraceId保存在Hashmap中，这里使用一个比原生map更快的intintmap，它比原生map具有更快的Hash计算能力。同时，将TraceId转成int，也使用目前已知最快的算法，参见 Str2Uint64() 函数。

2.9 如何确定Span的错误

在判断Span是否错误时，主要通过&进行查找，通过判断&之后的参数是否是错误来定位。为了加快查找使用go原生的bytes.IndexByte()该函数使用汇编，并且使用AVX2指令，比直接进行byte匹配快很多。具体参考GetErrorInBytes_Meet200Return()函数

2.10 预分配内存以及Trace对象池

为了加快速度，所有内存都进行预分配，在系统ready之前，已经将需要用到的包含80万Trace的Trace池（每个线程40万），初始化好，并合理的分配每个Trace的Span大小。Trace池中的Trace是重复使用的。整个过程不需要创建额外的Trace。 Scanner用到的每个线程各1G的Buffer空间，也是在系统ready之前分配的。

2.11 充分利用CPU Cache

当数据通过http下载后并按照'\n'分割后，该数据行位于CPU Cache，此时立即进行错误检查的效率要远远高于将这行存起来延后处理。因此，对数据行是否包含错误的判断，会在行分割后立即执行。延迟判断将带来性能损失。

2.12 一些需要避免使用的技术

虽然想channel，闭包等技术在Go语言中很常见，但是在本次比赛中应该避免使用。上述技术都将给性能带来毁灭性的打击。

3 关键代码

为了使用intintmap，需要一个快速将string转为int的函数：

```
func  Str2Uint64(value string) int64 {
var  result  int64
result |= (Hex[(value[0])]) << 32
result |= (Hex[(value[1])]) << 28
result |= (Hex[(value[2])]) << 24
result |= (Hex[(value[3])]) << 20
result |= (Hex[(value[4])]) << 16
result |= (Hex[(value[5])]) << 12
result |= (Hex[(value[6])]) << 8
result |= (Hex[(value[7])]) << 4
result |= (Hex[(value[8])])
return result
}
```

如何判断span是否包含错误：

```
const  SKIPCOUNT = 65
func  GetErrorInBytes_Meet200Return(line []byte) data.ERRORTYPE {
lastIndexOfS := bytes.IndexByte(line[SKIPCOUNT:], '&')
if line[lastIndexOfS-20 + SKIPCOUNT] == 'h' && line[lastIndexOfS-4 +SKIPCOUNT] == '=' && line[lastIndexOfS-3 + SKIPCOUNT] == '2' {
return data.NOERROR
} else  if line[lastIndexOfS-7 +SKIPCOUNT] == 'e' && line[lastIndexOfS-2 + SKIPCOUNT] == '=' && line[lastIndexOfS-1 +SKIPCOUNT] == '1'{
return data.ERROR
} else  if line[lastIndexOfS-20 +SKIPCOUNT] == 'h' && line[lastIndexOfS-4+SKIPCOUNT] == '='{
return data.ERROR
}

tag := line[lastIndexOfS+1 + SKIPCOUNT:]

if tag[0] == '&' {
tag = tag[1:]
}
```

```
var  i  int = 0
var  length  int
for {
length = len(tag)
if length < 7 {
return data.NOERROR
}
if tag[0] == 'h' && length >= 20 {
//stable compare whole http.status_code
if tag[16] == '=' && tag[17] == '2' {
//perf just compre 200 if tag[17:20] == "200" {
return data.NOERROR
} else  if tag[5] == 's' {
//} else if tag[0:6] == "http.s" {
return data.ERROR
}
} else  if tag[0] == 'e' && tag[1] == 'r' && tag[2] == 'r' && tag[3] == 'o' && tag[4] == 'r' && tag[5] == '=' && tag[6] == '1' {
return data.ERROR
}

if tag[0] == '&' {
tag = tag[1:]
continue
}

i = bytes.IndexByte(tag, '&') + 1

// log.Printf("i=%d",i)
if i == 0 || i == length {
return data.NOERROR
} else {
tag = tag[i:]
}
}
```

```
}
```

如何通过MyScan按行读取数据：

```
func (s *MyScanner) Scan() bool {
if s.done {
return  false
}
s.scanCalled = true
// Loop until we have a token.
for {
// See if we can get a token with what we already have.
// If we've run out of data but have an error, give the split function
// a chance to recover any remaining, possibly empty token.
if s.end > s.start || s.err != nil {
advance, token, err ,errorInLine := ScanLines(s.buf[s.start:s.end], s.err != nil)

if err != nil {
if err == ErrFinalToken {
s.errorInLine = errorInLine
s.token = token
s.done = true
return  true
}
s.setErr(err)
return  false
}
s.start += advance

s.errorInLine = errorInLine
s.token = token
if token != nil {
if s.err == nil || advance > 0 {
s.empties = 0
} else {
// Returning tokens not advancing input at EOF.
s.empties++
```

```
if s.empties > 100 {
panic("bufio.Scan: too many empty tokens without progressing")
}
}
return  true
}
}
// We cannot generate a token with what we are holding.
// If we've already hit EOF or an I/O error, we are done.
if s.err != nil {
// Shut it down.
s.start = 0
s.end = 0
return  false
}
// Must read more data.
// First, shift data to beginning of buffer if there's lots of empty space
// or space is needed.
if s.start > 0 && (len(s.buf)-s.end <= readSize) {

s.myStep++
// perf remove this
s.PostMyStepToBackend(s.myStep)
for s.BackendStep < s.myStep {
time.Sleep(time.Duration(1) * time.Microsecond)
}

copy(s.buf, s.buf[s.start:s.end])
s.end -= s.start
s.start = 0

log.Printf("myscan %d buffer full and restart", s.Tid)
}
// Is the buffer full? If so, resize.
if s.end == len(s.buf) {
```

```
//should never get here
log.Println("shoud not here , myscan buffer fully")
}
// Finally we can read some input. Make sure we don't get stuck with
// a misbehaving Reader. Officially we don't need to do this, but let's
// be extra careful: Scanner is for safe, simple jobs.

for  loop := 0; ; {
endPos := s.end + readSize
if endPos > len(s.buf) {
endPos = len(s.buf)
}
n, err := s.r.Read(s.buf[s.end:endPos])
s.end += n
if err != nil {
s.setErr(err)
break
}
if n > 0 {
s.empties = 0
break
}
loop++
if loop > 100 {
s.setErr(io.ErrNoProgress)
break
}
}
}
}


func (s *MyScanner) TextAndErrorInLine() (string,data.ERRORTYPE) {
return *(*string)(unsafe.Pointer(&s.token)),s.errorInLine
}
```

4 比赛经验总结和感想

通过这次比赛，让我对微服务中的链路调用分析有了一定的了解。同时，作为我的第一个Go程序，能够取得前十的成绩，也让人感到兴奋，这证明了Go的强大。经过这次比赛，学习了不少Go语言相关知识，告别了Go小白的身份。但同时，也察觉到了在这种极端性能敏感的场合下，不同计算机语言也会对比赛成绩产生较大的影响（最优秀的Java选手也只能勉强挤进前10）。

复赛：实现一个 Serverless 计算服务调度系统

1 赛题背景分析及理解

根据比赛的评分公式， Score_ti = (RT_f1/RT_f1b + RT_f2/RT_f2b + … + RT_fn/RT_fnb)/n * Weight1 + (ND / NDb) * Weight2, 系统需要在RT和ND之间获得平衡，即，需要在尽可能少的资源下，获得尽可能快的响应速度。

其中，响应速度又可以分为调度速度和函数的执行速度。对于调度来说，因为容器的创建是很慢的，所以，调度器必须尽可能复用(命中)已有容器。同时，为了确保函数的执行速度，函数请求应该尽可能被分配到尚有空闲资源的节点上执行。如果CPU资源不足，则可能导致RT变慢，如果内存资源不足，则可能导致函数执行出错。

2 核心思路

2.1 如何选择容器

- 参考ContainerManager.findAvalibleContainerInCurrentSet() , 首先找到所有这个函数的容器。并对他们进行打乱(Shuffle)，其目的是希望每次选择的容器尽可能随机
- 在全部容器找到一个所在节点当前请求数量最少的容器返回
- 如果是内存型的函数，则根据节点上内存型函数的数量判断
- 如果是CPU型的函数，则根据节点上所有的CPU请求数量判断
- 如果不是CPU型函数，则根据节点上所有的函数请求数量判断（参见 Container.NodeReqCount）

2.2 如何选择节点

- 参考 ContainerManager.findAvalibleNodeInCurrentSet() , 找一个可以新建容器的节点
- 首先打乱所有的Node（ShuffleNodes）
- 按照容器数量对节点进行排序（有限选择容器数量少的节点）
- 对于内存型函数，找一个内存函数占用内存最少的节点
- 对于CPU函数，找一个包含CPU函数容器最少的节点
- 对于非CPU函数，找一个包含当前函数容器数量最少的节点

2.3 如何确定容器的并行度

- FunctionResource.SuggestRunningCountOnOneContaine() 返回一个容器的并行度，即一次同时可以执行几个函数
- 如果函数没有ready，即在没有足够统计信息的情况下，始终为1（不并行，防止OOM）

» ready后，根据CPU和Mem进行估算，对于CPU占用率或者内存占用率高的函数，始终返回1。其他函数根据实际情况估算

» 由于CPU占用率对并行度计算非常重要，为了防止探测出错，允许在函数ready后，继续更新CPU占用率。更新的CPU占用率会马上产生新的并行度，减少在一个容器上堆放太多函数的可能

2.4 如何保证容器的均匀分布

» 均匀分布容器有助于平衡压力，在选择Node创建容器时，已经进行了一定的平衡，为了防止回收算法导致容器分布不平衡，需要通过后台线程强制保证其平衡性

» BalanceAllFunction()的作用就是将分布不均匀的容器进行均匀化的分布。具体算法细节在 BalanceFunction()

» 当需要平衡函数A时

» 对节点进行按照函数A容器数量从小到大排序

» 将最大的容器数量和最小的容器数量比较，如果差异大于2，则进行容器迁移

» 如果最少节点不能接受新容器，则尝试次小容器，依次类推

2.5如何判断一个节点可以接受一个容器

参考FunctionResource.CanAcceptFunctionContainer()，主要通过内存和CPU来判断。即加权内存剩余量大于函数加权内存使用量；并且节点所有容器Cpu使用量小于CpuOverSaleCount*200。

3 关键代码

3.1 程序入口

< 3.11函数派发入口 ContainerManager.AssignFunction()

» 首先尝试查找容器，如果找到就返回容器

» 如果找不到，就找可以新建容器的节点

» 如果找到合适的节点就新建容器

» 如果找不到节点就尝试申请节点

» 如果申请成功则新建容器返回

» 如果申请不成功，则回到函数起始位置，进行重试

» 重试90次不成功，强制释放可以释放的容器

» 重试100次不成功，放弃，返回容器不足的错误，并且在后台继续申请新的节点

< 3.12函数返回入口 ContainerManager.OnFunctionReturn()

维护容器和节点的请求数量，表示调用结束进行回收

如果出现错误，则释放当前容器，并重新申请

更新函数调用的统计信息，比如duration等信息

3.2 后台线程

< 3.21 RecycleContainerAllInOneTask

后台容器清理线程。如果有需要就释放容器。同时进行容器碎片化整理，释放节点。

容器碎片化整理参考DefragmentContainers()函数。回收触发的条件是系统总体内存使用率小于50%或者其中有一个节点的内存使用率小于200M。在碎片化整理中，会尝试将最小内存的节点上的所有容器移动到其他节点上，并释放节点。

回收过期容器，参考RecycleKeepaliveTimeoutContainers()函数。每个容器有一个过期时间。该时间为定义为MaxInt64(fr.MaxIntervalTimeInNano+1*int64(time.Second), BaseSuggestKeepAliveTimeInSec*int64(time.Second)),是一个大于75s的一个时间。一个容器如果超过这个时间没有使用，则可能被回收；如果一个容器超过这个时间的2倍还没有被使用，则一定会被回收。

平衡CPU敏感性函数和内存敏感性函数BalanceAllFunction() 。如果有一个CPU敏感性函数F1。在节点A上有2个容器，在节点B上没有容器。这个函数会把节点A上的一个F1容器移动到节点B上。

节点释放函数 RecycleFreeNode()， 释放没有容器的节点，它对节点容器进行探测，如果连续3次节点 都是空闲的，则释放节点。

3.22 如何确定容器并行度

```
func (fr *FunctionResource) SuggestRunningCountOnOneContainerInternal() int32 {
memSug := int32(fr.ReqMem / fr.MaxMemUsage)
if memSug < 1 {
memSug = 1
}
var  cpuSug  int32 = 0

checkCpu := fr.MaxCpuUsage * MaxCpuBase
if checkCpu < 1 {
cpuSug = 9
} else  if checkCpu < 5 {
cpuSug = 6
} else  if checkCpu < 20 {
cpuSug = 3
} else {
cpuSug = 1
}
fr.SuggestFunctionCountInOneContainer = MinInt32(memSug, cpuSug)
return fr.SuggestFunctionCountInOneContainer
}
```

3.23 如何确定容器的timeout时间

```
func (fr *FunctionResource) SuggestKeepAliveTimeInNamo() int64 {
return MaxInt64(fr.MaxIntervalTimeInNano+1*int64(time.Second), BaseSuggestKeepAliveTimeInSec*int6
```

```
4(time.Second))
    }
```

3.24 如何查找容器

```
    func (cm *ContainerManager) findAvalibleContainerInCurrentSet(fr *FunctionResource, ContainerIds []int,
reqMemInBytes int64, reqId string) int {
    containerCount := len(ContainerIds)
    if containerCount == 0 {
    return -1
    }

    sugcount := fr.SuggestRunningCountOnOneContainer()

    Shuffle(ContainerIds)
    sort.Sort(ContainerSortedByRunningFuncCount(ContainerIds))

    var  minReqCount  int = 99999999999
    var  minContainerIndex  int = -1

    cm.NodeReqCountMutex.Lock()
    for  i := 0; i < len(ContainerIds); i++ {
    if !ContainerPoolInstance.allContainers[ContainerIds[i]].IsUsed || ContainerPoolInstance.
allContainers[ContainerIds[i]].WaitRemove {
    continue
    }
    ContainerPoolInstance.allContainers[ContainerIds[i]].mutex.Lock()
    if ContainerPoolInstance.allContainers[ContainerIds[i]].IsUsed == true && ContainerPoolInstance.
allContainers[ContainerIds[i]].WaitRemove == false &&
    ContainerPoolInstance.allContainers[ContainerIds[i]].RunningFuncCount < sugcount &&
    ContainerPoolInstance.allContainers[ContainerIds[i]].NodeReqCount() < minReqCount {

    minReqCount = ContainerPoolInstance.allContainers[ContainerIds[i]].NodeReqCount()
    minContainerIndex = ContainerIds[i]
    }
    ContainerPoolInstance.allContainers[ContainerIds[i]].mutex.Unlock()
    }
```

```
if minContainerIndex < 0 {
cm.NodeReqCountMutex.Unlock()
PreserveContainerChannel <- PreserveContainerData{fr, 0}
return -1
}

ContainerPoolInstance.allContainers[minContainerIndex].IncrNodeReqCount(reqId)

ContainerPoolInstance.allContainers[minContainerIndex].mutex.Lock()
ContainerPoolInstance.allContainers[minContainerIndex].RunningFuncCount++

ContainerPoolInstance.allContainers[minContainerIndex].MemUsageInBytesByPlan += reqMemInBytes
ContainerPoolInstance.allContainers[minContainerIndex].mutex.Unlock()
 cm.NodeReqCountMutex.Unlock()
 return minContainerIndex
}
```

3.25 如何查找节点

```
func (cm *ContainerManager) findAvalibleNodeInCurrentSet(fr *FunctionResource, reqMemInBytes int64,
exceptNodeIndex int) int {
BeginFindAvalibleNodeInCurrentSet:

cm.nodeRwMutex.RLock()
tmpNodes := make([]*Node, len(cm.nodes))
copy(tmpNodes, cm.nodes)
cm.nodeRwMutex.RUnlock()
 var  nodeSize = len(tmpNodes)
if nodeSize == 0 {
return -1
}
var  minCount  int = 99999999
var  minMem  int64 = math.MaxInt64
var  minIndex = -1
var  i = 0
ShuffleNodes(tmpNodes)
```

```
sort.Sort(NodeOrderByContainerCount(tmpNodes))

fr.NodeContainerCountMutex.Lock()

if fr.MaxMemUsage > 100*1024*1024 {
for  i = 0; i < len(tmpNodes); i++ {
if tmpNodes[i].IndexInPool == exceptNodeIndex || !tmpNodes[i].CanAcceptFunctionContainer(fr) ||
!tmpNodes[i].IsUsed {
continue
}
memcount := tmpNodes[i].GetMemContainerTotalMem()
if memcount < minMem {
minMem = memcount
minIndex = i
}
}
} else  if fr.MaxCpuUsage > 40 {
for  i = 0; i < len(tmpNodes); i++ {
if tmpNodes[i].IndexInPool == exceptNodeIndex || !tmpNodes[i].CanAcceptFunctionContainer(fr) ||
!tmpNodes[i].IsUsed {
continue
}
count := tmpNodes[i].GetCpuContainerCount()
if count < minCount {
minCount = count
minIndex = i
}
}
} else {
for  i = 0; i < len(tmpNodes); i++ {
if tmpNodes[i].IndexInPool == exceptNodeIndex || !tmpNodes[i].CanAcceptFunctionContainer(fr) ||
!tmpNodes[i].IsUsed {
continue
}
count := tmpNodes[i].GetContainerCountByFr(fr)
```

```
if count < minCount {
minCount = count
minIndex = i
}
}
}
if minIndex < 0 {
fr.NodeContainerCountMutex.Unlock()
return -1
}
tmpNodes[minIndex].IncreaseCount(fr)
fr.NodeContainerCountMutex.Unlock()
tmpNodes[minIndex].mutex.Lock()
if tmpNodes[minIndex].IsUsed {
tmpNodes[minIndex].MemUsageInBytesByPlan += reqMemInBytes
tmpNodes[minIndex].ContainerCount++
} else {
tmpNodes[minIndex].mutex.Unlock()
goto BeginFindAvalibleNodeInCurrentSet
}
tmpNodes[minIndex].mutex.Unlock()
return tmpNodes[minIndex].IndexInPool
}
```

3.26 如何确认一个节点可以创建一个容器

```
func (n *Node) CanAcceptFunctionContainer(fr *FunctionResource) bool {
return  float64(n.TotalMemInBytesByPlan)-n.GetWeightedUsedMem() > fr.GetWeightedMem() &&
n.IsUsed == true &&
fr.CpuUsageInNode()+n.ContainerCpuSumWithoutLock() < 200*CpuOverSaleCount
}
```

5 比赛经验总结和感想

这次比赛让我了解了阿里云函数计算这种最新的云计算模式，同时，通过对赛题的解读，对阿里云函数计算的基本工作模式有了初步的了解，让人受益匪浅。

通过比赛中的一次次尝试，对函数计算中的调度模块的功能和行为模式也有了更深入的理解，初步认识了调度系统的核心问题和困难。是一场另人难忘而又不可多得的比赛。

2 比赛攻略——greydog.

初赛赛道 1：实现一个分布式统计和过滤的链路

1 赛题分析

1.1 数据来源

采集自分布式系统中的多个节点上的调用链数据，每个节点一份数据文件。数据格式进行了简化，每行数据(即一个span)包含如下信息：

traceId | startTime | spanId | parentSpanId | duration | serviceName | spanName | host | tags

具体各字段的：

traceId: 全局唯一的Id，用作整个链路的唯一标识与组装

startTime: 调用的开始时间

spanId: 调用链中某条数据(span)的id

parentSpanId: 调用链中某条数据(span)的父亲id，头节点的span的parantSpanId为0

duration: 调用耗时

serviceName: 调用的服务名

spanName: 调用的埋点名

host: 机器标识，比如ip，机器名

tags: 链路信息中tag信息，存在多个tag的key和value信息。格式为key1=val1&key2=val2&key3=val3 比如 http.status_code=200&error=1

数据总量足够大

文件里面有很个调用链路（很多个traceId）。每个调用链路内有很多个span（相同的traceId，不同的spanId）

例如文件1有

d614959183521b4b|1587457762873000|d614959183521b4b|0|311601|order|getOrder|192.168.1.3|http.status_code=200

d614959183521b4b|1587457762877000|69a3876d5c352191|d614959183521b4b|305265|Item|getItem|192.168.1.3|http.status_code=200

文件2有

d614959183521b4b|1587457763183000|dffcd4177c315535|d614959183521b4b|980|Loginc|getLogisticy|192.168.1.2|http.status_code=200

d614959183521b4b|1587457762878000|52637ab771da6ae6|d614959183521b4b|304284|Loginc|getAddress|192.168.1.2|http.status_code=2

1.2 需求

用户需要实现两个程序，一个是数量流的处理程序，该机器可以获取数据源的http地址，拉取数据后进行处理，一个是后端程序，和客户端数据流处理程序通信，将最终数据结果在后端引擎机器上计算。

1.3 要求

找到所有tags中存在 http.status_code 不为 200，或者 error 等于1的调用链路。输出数据按照 traceId 分组，并按照startTime升序排序。记录每个 traceId 分组下的全部数据，最终输出每个traceId下所有数据的CheckSum(Md5值）。

1.4 难点

如何快速找到error trace，这也是赛题的核心，由于每个trace会分布在多个节点，某trace在当前节点不为error trace，但在其他节点却可能为error trace。

2 缓存设计

数据流缓存：采用环的形式，缓存大小：2GB+最大行长度

metadata缓存：用于记录数据流缓存中的数据，采用环的形式，最大容量：(WINDOW_SIZE * 64 * 2)条

```
struct TraceOffsetLink
{
  uint64_t traceId;      //traceId
  uint64_t globaloffset; //文件中的偏移  缓存中的偏移 =（文件中的偏移%(数据流缓存大小−最大行长度)）
  uint64_t lineIndex;    //行号
  int32_t prev;          //用于记录该span 前一个tracespan的位置，如果不使用map,可以去掉
  uint16_t len;          //长度
  uint16_t flag;         //标志
};
```

2.1 客户端分别记录处理长度和最大下载长度

拉数据时确保： 最大下载长度 − 处理长度 < 数据流缓存大小 − 最大行长度 − 单次下载大小

处理时确保： 最大下载长度 − 处理长度 > 最大行长度

2.2 客户端分别记录本地处理行数和所有客户端的最小处理行数

处理时确保： 本地处理行数 − 最小处理行数 < metadata最大长度 − 窗口大小

3 客户端 <-> 后端引擎 通信逻辑

3.1 同步

客户端每隔一段时间（2个窗口大小的时间，可调整），向后端引擎发送其处理行数，后端引擎会记录所有客户端的处理行数，并计算一个最小值，每当最小值发生变化时，会将这个最小值发送给所有客户端。

3.2 发送wrongtrace

< 3.21客户端使用map的方案

每个客户端独自维护一个map，用于记录每个trace在metadata缓存中最后一次出现的位置。（同一trace下的span组成一个链表）。

每个客户端按行处理判断span是否为wrongtrace。

如果当前span为wrong trace，发送traceid给后端引擎，后端引擎收到后发给其它客户端，其它客户端收到后到，在map中查找该trace最后一次出现的位置。对于当前客户端则直接查找。

1. 如果map中查找到该trace有记录位置且为-1，表示曾经已经判断该trace为wrong了，不需要做处理，只需要发当前span给后端引擎(trace一旦已经判断为wrong trace了，前面出现的该trace下的span会全部发给后端引擎，这时再出现不需要考虑前面的)。

2. 如果map中查找到该trace有记录位置且为不为-1，这时需要将前面出现的该trace下的span全部发给后端引擎，并更新map中的位置为-1，发送时根据prev回溯(同一trace下的span组成一个链表)，直到遇到-1。

3. 如果map中查找到该trace没有记录位置，在map中插入-1。

如果当前span不为wrong trace，在map中查找该trace最后一次出现的位置:

1. 如果map中查找到该trace有记录位置且为-1，表示曾经已经判断该trace为wrong了，不需要做处理，只需要发当前span给后端引擎。

2. 如果map中查找到该trace有记录位置且不为-1，更新map中的位置为当前span的位置，并把当前span的prev指向map原来的记录的位置。

3. 如果map中查找到该trace没有记录位置，在map中插入当前span的位置，并把当前span的prev指向-1。

< 3.22 客户端不使用map的方案

1. 每个客户端按行处理判断span是否为wrongtrace，如果是wrong trace，则发送traceid和对应行数给后端引擎。

2. 后端引擎收到后先检查一下是否已经收到过该traceid，没有收到过，则将该traceid和对应行数发给所有客户端。

3. 客户端收到后，在metadata中对应行数位置遍历前后2万条，对应traceid的span数据发送给后端引擎。

4 查找 wrong trace(核心)

查找wrong trace最终有2个方案，纯处理的话方案2大概比方案1快2百多ms。(后面由于加入了长tag，方案1应该比原来快一点)。

2个方案在线上用时无差异，都是下载速度跟不上处理速度（前期测试的，后期只测试了方案2）。

4.1 基于单字符搜索&再匹配

此方法很准确，速度也快，但相比方案2较慢(后面由于加入了长tag，应该比原来快一点)。

第一步，同时寻找&和\n。

第二步，如果为\n，退出查找，如果为&，判断下一字符是否为'h'或'e'。

第三步，如果下一字符为'h'或'e'，则**进一步判断是否为"http.status_code=200&"**或**"error=1&"**或**"http.status_code=200\n"**或**"error=1\n"**，否则回到第一步。

4.2 基于双字符搜索直接查找字符串

在"http.status_code=200"中选取1对交叉的长度为2的低频子串，例如

"s_" "_c"

或

".s" "_c"

当搜索到这2个子串时，再进一步判断是否为 **"http.status_code=200"**

" error=1" 同理

此方法速度稍快，但有一些缺陷

1. 因为此方法没有以&为边界，而是相当于直接匹配"http.status_code=200"或"error=1"这2个字符串， 因此如果出现某个tag包含子串"http.status_code=200"或"error=1"的情况时，将会误判。

2. 在判断行结尾时，因为是双字符查找，所以导致每行至少需要3个字符，用于判断是否结束(如果是单字符搜索，可以用'\n'判断是否结束)，这里选用了下一行的startTime前的'|' 以及startTime前2个字符判断结束(所以此方法有一定取巧)，当然也可用低频单字符查找来搜索匹配字符串，但速度将降低，因为单字符出现频率较高。

初赛赛道 3：服务网格控制面分治体系构建

1 赛题分析

https://code.aliyun.com/middleware-contest-2020/pilotset
https://tianchi.aliyun.com/forum/postDetail?postId=109732

2 思路分析

2.1 数据处理

原本的appName 和 pilotName都是字符串，程序内部统一采用唯一的索引(int32)表示，最终输出再转换为字符串

由于需要记录每个pilot的app，且需要进行增删查操作，使用hash table记录是一个不错的选择。

但是优化过程中频繁涉及到随机选择(查找)，hash table效率低下。

最终的方案是将数据堆砌在一起(便于随机选取)，并为其构造一个索引表。(便于增删查)

2.2 初始化

随机初始化

随机为每一个app选择pilot

贪心算法

首先将app按 sidecarNum/SrvNode排序，然后按顺序选择局部最优的pilot

两种初始化方法差异不大

2.3 优化

不管是贪心算法还是随机初始化，解的质量都是比较差的

注意点：每一轮只能对本次新增的app随意移动，如果要移动之前的app，要保证原pilot加载的数据一直存在

优化算法大体上有3个，效果差异不大

这里主要介绍排行榜最终得分的算法

核心思想，随机选择pilot和app，进行迭代迁移

首先将pilot按 sidecarNum/SrvNode排序

选择时pilot保证sidecarNum SrvNode 不均匀的pilot被选中的概率更大，

sidecarNum偏大的pilot选中SrvNode偏大的概率更大(互补)

选择app服从均匀分布，然后进行迭代迁移

模拟退火，递归枚举最优解(局部)

多线程加速

由于线上评测机器是4核，评测时间时间卡的比较严，为了充分利用cpu。

将原pilot随机划分为几个子集单独优化，再合并，再划分…

复赛：实现一个 Serverless 计算服务调度系统

1 关键代码

/cpp/include/scheduler、/cpp/src/scheduler

文件夹下的代码为scheduler的实现，其余基本为本地评测相关代码

/cpp/include/config.h

配置文件，里面有详细注释这里不再赘述

/cpp/include/scheduler/container.hpp

记录容器信息(id、函数、Node)，运行统计

/cpp/include/scheduler/functionInfo.hpp

函数类，记录函数的mete信息，容器，内存、cpu等信息统计。函数cpu、mem更新，调用统计(压力)

/cpp/include/scheduler/node.hpp

Node类，记录地址，端口，内存、cpu等信息统计。Node上的容器创建，回收。函数信息探测

/cpp/include/scheduler/resourceManager.hpp

资源管理类，Node的创建，回收。容器选择，创建，回收

/cpp/include/scheduler/server.hpp

apiserver->scheduler入口

AcquireContainer位置: /cpp/include/scheduler/server.hpp 53行

ReturnContainer位置: /cpp/include/scheduler/server.hpp 140行

2 赛题分析

2.1 赛题背景

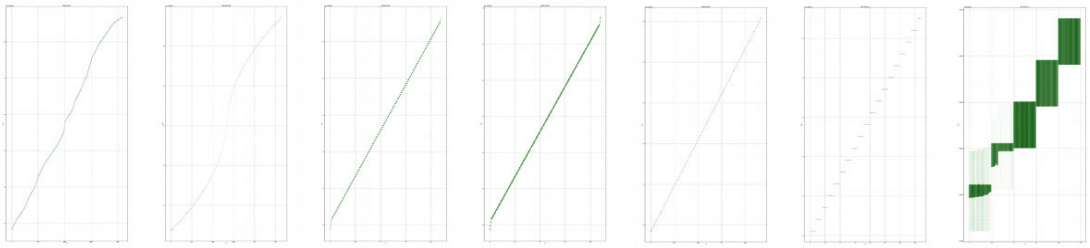
（https://tianchi.aliyun.com/competition/entrance/231793/information ）

Serverless 计算服务（如阿里云函数计算）让用户无需管理服务器等基础设施，只需编写并上传代码，服务会准备好计算资源，并以弹性、可靠的方式运行用户代码。服务根据代码实际执行时间收费，让使用者无需为闲置资源付费，其背后是服务采用各种调度策略降低请求的响应时间，并且使用尽可能少的资源。

传统的应用通常根据某些指标来扩展所需要的计算资源，比如根据CPU，内存等硬件资源的使用情况，或者根据延迟，队列积压等业务指标。而一些云厂商提供的函数即服务（FaaS）如阿里云函数计算，AWS Lambda采用了一种基于请求的调度方式，根据当前系统的资源所能够支撑的并发请求数和实际请求数作为主要依据调度资源。本题目是关于设计和实现一个函数计算服务的调度系统，为函数调用请求高效的管理计算资源，不限制使用何种方式作为调度的依据。

2.2 赛题特点

1. 函数类型多样，CPU紧密型，内存紧密型，周期请求型



2. 评测过程短(20min)，函数周期短(秒级)，执行时间短(微秒级-秒级)
3. 分配Node耗时极短

128	130	133	133	146	147	148	148	150	151	152	154	154	156	160	161	162	174	222	224	252
8	1	3	3	1	7	4	8	3	6	5	1	1	2	4	0	6	1	2	3	7

①. 平均时间在1614μs左右

②. 创建容器时间较长，且同一Node并行创建容器明显性能下降，445176μs - 4997857μs，最快可以在几百毫秒，如果Node负载过高，创建容器时间甚至需要好几秒

③. 创建、迁移容器成本太高，有的函数创建一个容器的时间远大于函数所有调用时间之和，总响应时间基本取决于首次创建容器的时间

④. 成本需同时考虑响应时间和资源使用量，即要求请求能够被及时处理，又使用较少的资源

2.3 优化方向

1. 冷启动优化

由于无法提前知道函数信息，同一函数前几次请求必然无法提前准备容器，必然会有冷启动

2. 容器选择优化

尽量做到负载均衡，充分利用CPU、MEM，提高响应速度

3. 资源优化

尽量使用更少的Node

2.4 具体优化点

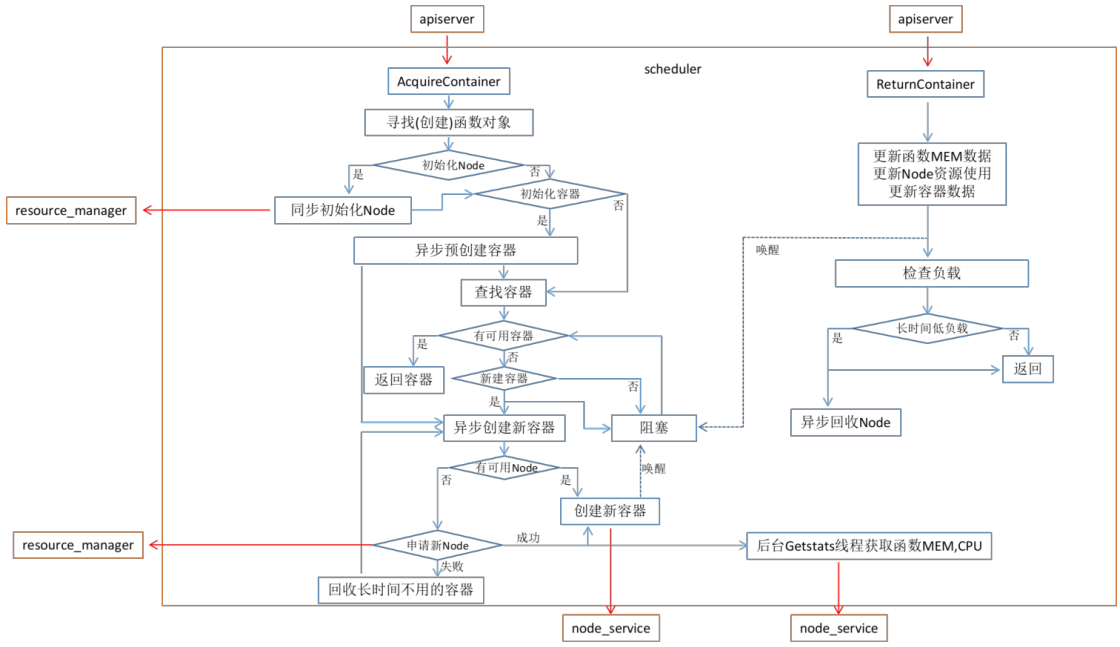
1. 创建容器耗时太长，应尽量在请求到来之前，准备好容器，减少冷启动次数。
2. 冷启动时避免在同一Node同时创建容器，保证Node创建容器任务均衡。
3. 创建、迁移容器成本太高，应该尽量在创建容器的时候就保持资源均衡。
4. 评测过程短，函数周期短，执行时间短，在请求到来时容器的选择上做到负载均衡。
5. 容器尽量分散，sleep型和CPU紧密型函数保持均衡， CPU使用量和MEM使用量保持均衡。
6. 同一函数Node之间并行数量尽量分配均衡。
7. 尽量并行，充分利用CPU资源，特别是后面增加了一种模型内存常驻的函数，并行就显得更为重要，并行可以减少模型加载次数，减少内存浪费，从而使用更少的资源。

3 核心思路

1. 容器的创建。
2. 请求到来时容器的选择。

4 具体思路

4.1 程序流程



4.2 说明

1. 在第一个请求到来的时候会同步进行Node的初始化，预分配几个Node，后面会根据压力自动扩展。

2. 每个Node申请后，会后台启动GetStats线程探测函数信息,当超过一段时间没有新函数出现时，暂停探测。

3. 在某函数第一个请求到来的时候会异步进行容器的初始化，预创建几个容器，然后会根据压力再分配，当获取到某个函数的统计信息后，会对小内存容器进一步预分配，后面会根据压力再分配。

4. 当创建容器时，如果没有可用Node，会申请新的Node，如果没有可申请的Node， 则回收长时间不使用的容器，创建完成后，唤醒正在等待容器的请求。创建容器时，尽量选择负载低的Node。

5. 当整体负载长时间过低时，后台回收Node(及其上面创建的容器)。

6. 最开始做了后台容器优化(预分配、迁移、回收)，后台Node回收，由于迁移代价巨大，所以尽量在创建、选择容器时保持均衡，不再进行后台优化。

7. 周期请求型函数处理，由于周期请求型函数周期太小，最开始加了周期性回收、预创建，由于回收创建成本太高，效果明显不好，故周期请求型函数不做特殊处理。

8. 选择容器的原则，首先内存必须充足，并行数量少，Node上sleep型和CPU紧密型函数比例保持均衡，Node上CPU使用量和MEM使用量比例保持均衡（ scheduler会在本地记录、更新各个Node的CPU、MEM、并行数量等使用详情）。

5 健壮性分析

5.1 冷启动的处理

程序会通过预创建容器，尽量减少冷启动的次数。

5.2 OOM的处理

程序会严格计算、记录资源，控制Node内存并预留一定空间，一般不会出现OOM。

假设由于GetStats误差出现OOM，或出现其它错误，会移除出错的容器，防止后续调用失败。

5.3 大压力请求或超多函数的情景

对于大压力请求或超多函数的情况下，不会轻易崩溃。

程序支持自动扩容资源，支持回收长时间不用的容器，支持回收Node释放资源。

即使资源不足，程序也不会返回错误。如果资源不足且没有资源可回收时，则阻塞该请求到有足够资源运行，或该函数其它请求调用完毕（但是超时时间过短，可能会出现调用超时，但是这在资源不足的情况下也是无法避免）。

6 比赛经验总结和感想

感谢举办方提供这样一个机会和平台，特别是复赛，应该动用了不少的机器评测。

无论是初赛还是复赛，我觉得都是比较有趣的，特别是赛道一。这次比赛我学到了很多，比如socket编程、http协议、grpc、proto、docker相关知识，sse指令优化，同时也认识了一些朋友，对阿里云的技术以及云原生有了更加深刻的认识和了解。

3 比赛攻略—ONE PIECE 团队

初赛赛道 2：实现规模化容器静态布局和动态迁

1 赛题背景分析及理解

1.1 赛题介绍

实现规模化容器静态布局和动态迁移：

1）规模化容器静态布局场景。我们准备了确定数量多规格机器资源，使得在满足调度约束条件下，确定数量、多种类的应用容器能在这些机器资源上满足扩容诉求，保证建站的确定性(赛题中我们会给出充足的机器，但真正建站场景我们会先通过计算预估机器)。

2）规模化容器动态迁移场景。我们会准备一个集群容器静态布局数据，此时集群是一种碎片态。然后通过容器的迁移，按照规则要求尽可能腾空机器资源，并过滤空机器，使得碎片态集群现状重新成为饱满态。

1.2 赛题分析

静态布局问题可以理解为多约束的多维装箱问题，约束有资源、绑核、堆叠和打散，约束条件多，判断约束条件的时间较长。

考虑在三个方面着手：

1. 减少判断约束条件的时间，以扩大搜索解空间的范围
2. 选择一个好的初始解，解空间很大，在比赛给出的时间内能搜索的范围很小，这时好的初始解很重要。
3. node的选择，对于不同规格的容器，node的选择顺序决定了node资源的使用情况。

动态迁移问题可以从两个思路解决：

1. 先将所有容器按照静态布局的方案进行放置，再通过前后位置的差异进行迁移
2. 使用策略判断是否需要迁移，逐渐减少node的使用数

综上，初赛考察的是复杂的装箱问题，在有限的时间内找到较优解，动态迁移使用较少的迁移次数达到饱和态。

2 核心思路

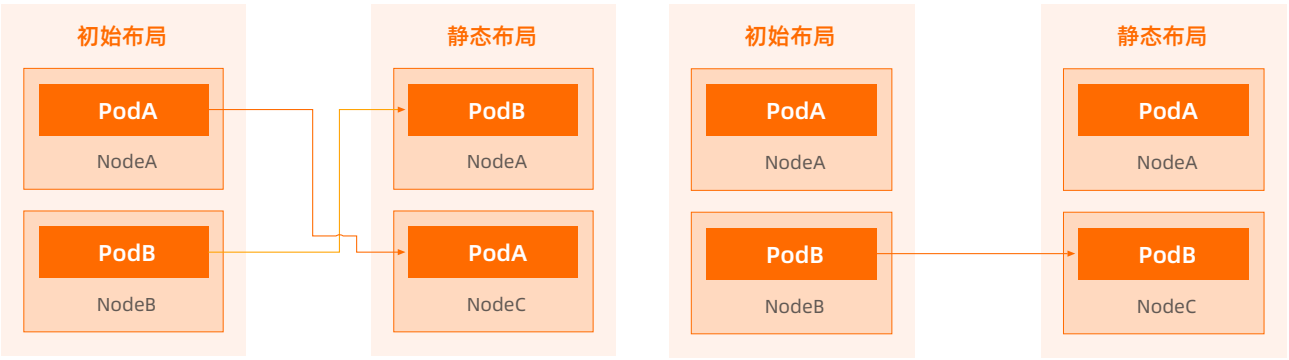
静态布局考虑了两个方案，一是使用模拟退火，二是模拟退火+局部DP。因为使用模拟退火时，后面的node资源利用率较差，可以用DP来改进这一缺陷。首先先试验使用DP能够跑多大的数据规模，每次判断约束条件的时间大约在500ms，在

调整数据结构后，时间降低至200~300ms，但DP也只能跑几个node，无法配合模拟退火。

最终方案主体采用模拟退火，pods根据cpu/ram从小到大排序，nodes根据与pod的资源不匹配分数从小到大排序。

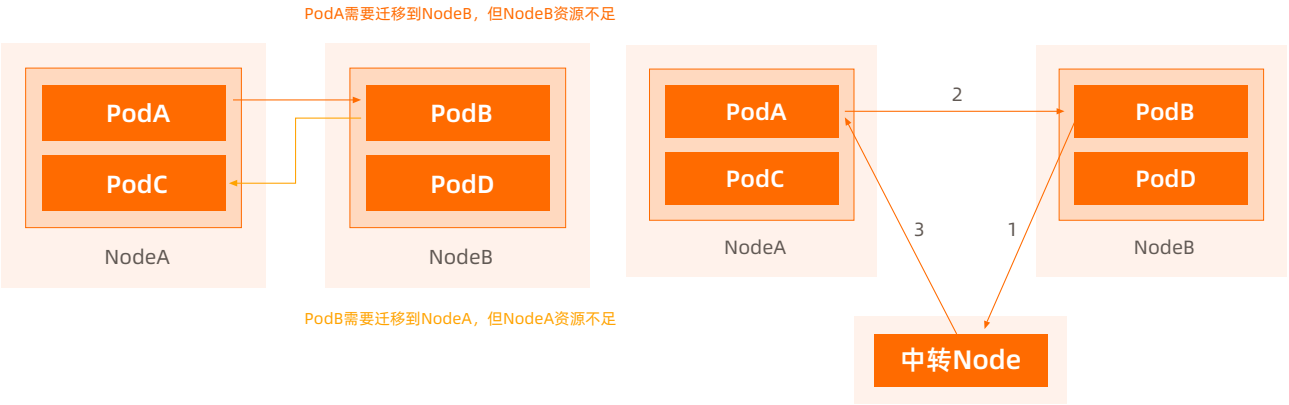
动态迁移先将不满足约束的pod进行迁移，然后采用静态布局的方法将每个pod重新放置，根据布局前后的位置不同决定是否需要迁移。不采取任何优化时，迁移次数在8000-8300。考虑到静态布局后，PodA的原node可能存在同规格的PodB，可以交换两者位置，PodA就不用迁移，经过优化后迁移次数在4000~4200。

优化策略如图所示：



优化前（PodA和PodB都需要迁移）　　优化后（PodA和PodB交换位置，PodA不需要迁移）

在迁移过程中，可能存在死锁情况，都在等待对方迁移空出位置，需要跳转到中转node解除死锁。方案如图所示：



优化前（PodA和PodB形成死锁）　　优化后（根据迁移顺序，解除死锁）

复赛：实现一个 Serverless 计算服务调度系

1 赛题背景分析及理解

1.1 赛题介绍

一个简化的Faas系统分为APIServer，Scheduler，ResourceManager，NodeService，ContainerService 5个组件，本题目中APIServer，ResourceManager，NodeService，ContainerService由平台提供，Scheduler的AcquireContainer和ReturnContainer API由选手实现（gRPC服务，语言不限），Scheduler会以容器方式单实例运行，无需考虑分布式多实例问题。

其中测试函数由平台提供，可能包含但不局限于helloworld，CPU intensive，内存intensive，sleep等类型；调用模式

包括稀疏调用，密集调用，周期调用等；执行时间包括时长基本固定，和因输入而异等。

选手对函数的实现无感知，可以通过Scheduler的AcquireContainer和ReturnContainer API的调用情况，以及NodeService.GetStats API获得一些信息，用于设计和实现调度策略。

1.2 赛题分析

赛题需要完成AcquireContainer和ReturnContainer两个接口，根据请求函数分配相应的node和container，主要考察node的资源分配，以及请求的响应时间。

评分从两个方面，总的响应时间RT和资源使用时间ND，两者的权重都是0.5，分数是与基准分相比得出的，所以基准的实现策略对选手的策略有很大影响。

node的初始可用数只有10个，需要过一段时间才可以申请新的node，总的可用node数为20个。要规划好node的资源分配，避免前期请求申请超过初始node时出错，造成惩罚性分数。

内存密集型函数占用内存较大，同一container同时运行多个实例，可能会导致OOM，cpu密集型函数也尽量避免同一container运行多个实例，可能会造成超时，这两种情况也会有惩罚性分数。

有些函数（如helloworld、sleep等）占用资源很小，同一container同时运行多个实例，响应时间变化不大，可以减少创建容器的时间。

2 核心思路

由于赛题使用grpc，不限制语言的使用，对于比拼速度的比赛，语言的选择也是考虑的一方面，分别尝试使用c++，java，golang三种语言实现demo，经测试发现，c++与golang的速度差不多，java要比其他两个慢10%~20%。Golang对于并发开发具有天生的优势，再加上官方提供了demo，所以最终选择了golang作为开发语言。

官方前期没有提供评测系统，每次提交系统需要等待1到3小时才能得到结果，等待结果的同时并不能并行开发，效率极低。通过线上跑出的日志，自己使用java实现了一个评测demo，每个函数的调用频率、调用时间、首次调用时间都可以通过日志分析出。由于无法模拟真实环境，所以NodeService只是实现基本功能，无法根据负载情况做出相应的执行时间。评测demo可以检验程序的正确性，以及资源的使用情况，避免了线上等待很久才发现出错的情况，提升了开发效率。

程序的整体策略就是负载均衡和动态扩缩，负载均衡提升响应速度，动态扩缩应对调用压力的变化以及不同数据，避免出现大规模的调用失败的情况。分别从以下几点着手：

2.1 选取node的策略

1. 如果当前请求函数的并发数大于现有的node数，且现有node不超过初始node数量（设定10个node），则获取新的node创建容器，目的是充分利用前期仅能获取的初始node，让每个node响应较少的请求，可以更快的创建容器以及更快的响应速度。

2. 使函数均匀分布在每个node，首先按照node内该函数容器数量排序，优先数量少的node，如果数量相同，优先可用内存多的node。目的是资源分配更加平均，避免资源抢占。

3. 当初始node数不能再满足新的请求时，申请新的node，满足高压力的请求。

2.2 选取container的策略

1. 当容器数大于等于请求数，每个容器同时满足一个请求。如果容器数不够，则创建新的容器。

2. 如果该函数（不包括内存密集型和cpu密集型）当前不能再创建容器，则每个容器同时满足多个请求。

2.3 函数分类

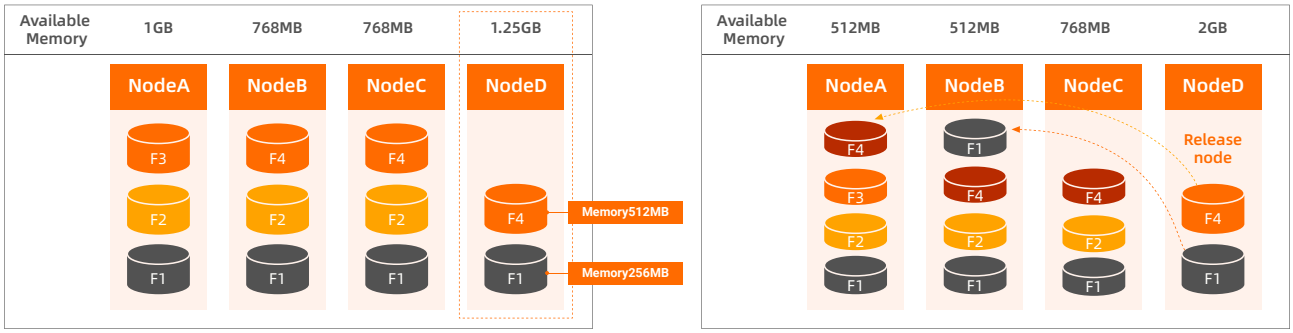
1. 当使用内存高于分配内存的30%时，判定为内存密集型函数。
2. 当cpu使用率大于10%时，判定为cpu密集型函数。
3. 当函数执行时间小于50ms时，判定为短时间型函数，若再超过65ms的上限，取消判定为短时间型函数。

2.4 动态缩减

1. 根据cpu密集型容器的数量，初步判断可以缩减的node数量，判断依据是缩减后每个node内cpu密集型容器的数量不超过一定值（更优的方案是根据cpu负载情况判断数量），防止cpu资源紧张，响应时间过长。

2. 选择使用内存最少的node，先校验是否满足迁移条件。迁移条件是剩余node的可用内存是否满足该node所有容器的需求，重点是内存密集型函数。

3. 当满足迁移条件后，将该node的所有容器迁移到其他node，然后释放该node。当缩减数量达到预判值或者不再满足迁移条件，则停止该轮缩减。



2.5 回收资源

1. 部分函数可能会从密集调用变为稀疏调用，或者调用一段时间后就不再调用，这样会导致一些空闲容器，导致资源浪费（尤其是内存密集型会一直占用内存）。所以，当容器超过一定时间（设5分钟）没有执行函数时，则释放该容器。若node中没有容器，则释放该node。

根据以上几个策略，程序没有设定最终使用的node数，会根据实时压力动态扩缩。程序每次分配node，都是根据node的可用内存和函数的分配内存进行选择，并且内存密集型容器只会同时执行一个请求，避免发生OOM。cpu密集型容器只会同时执行一个请求，避免出现执行超时的情况。

根据基准分来看，RT相对与ND优化空间更大，所以重点优化响应时间，根据函数的分类指定多种策略：

2.6 函数执行优先级

函数： 短时间型、cpu密集型（非短时间）、内存密集型（非短时间）

分析： cpu密集型和内存密集型函数一般执行时间较长，占用资源较多，与短时间型同时执行时，会存在资源竞争，导致短时间型执行时间增加至两到三倍。

策略： 短时间型优先级比cpu密集型和内存密集型高，当短时间型正在执行时，cpu密集型和内存密集型等待其执行完毕或者等待超时。由于cpu密集型和内存密集型执行时间较长，等待的时间对其影响不大。

2.7 预留容器

函数： 所有函数

分析： 创建容器的时间在0.4~1.0s，大大增加了请求的响应时间，要尽量避免在请求到来时创建容器。部分函数由稀疏

调用逐渐变为密集调用，可以在空闲时间提前创建容器。

策略：当某函数的容器满载时，则提前申请一个空闲容器，之后并发请求数增加时，省去创建容器的时间，减少了响应时间。

2.8 资源调整

函数：cpu密集型

分析：容器规格是由内存决定的，cpu和内存成比例，每1GB内存对应0.67cpu。cpu密集型执行时，可能存在cpu满载的情况，执行时间受资源限制不能再减少。

策略：当函数执行时cpu使用率高于分配的90%，则认为当前分配的cpu资源不足，先创建新容器再删除旧容器，新容器分配的内存是旧容器的200%，对应的cpu资源也是200%，减少函数的执行时间。

3 比赛经验总结和感想

初赛动态迁移部分，一开始选择直接迁移这个方案，简单实现后发现并不理想，然后就放弃了，赛后才发现这个方案要更优，应该多做一些尝试。在比赛中，学到了很多东西，也结识了很多大佬，感谢官方举办这次比赛。

云原生编程挑战赛

阿里云 阿里云原生 | TIANCHI天池