比赛攻略—ONE PIECE 团队

初赛（赛道二，语言 Java）

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：

- 减少判断约束条件的时间，以扩大搜索解空间的范围
- 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。
- 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：

- 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

- 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：

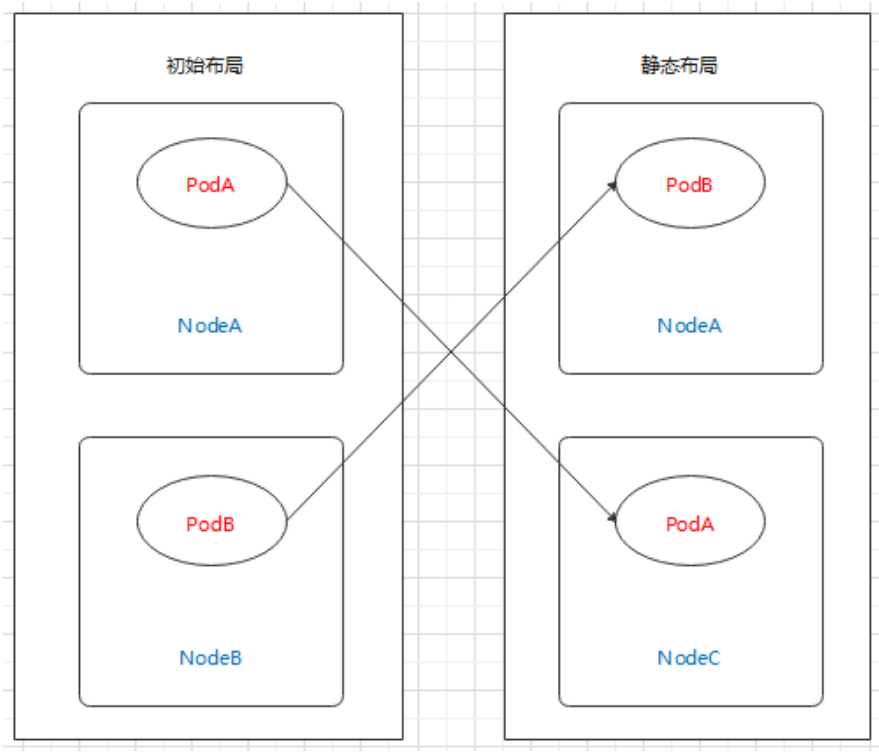

图 1 优化前（PodA 和 PodB 都需要迁移）

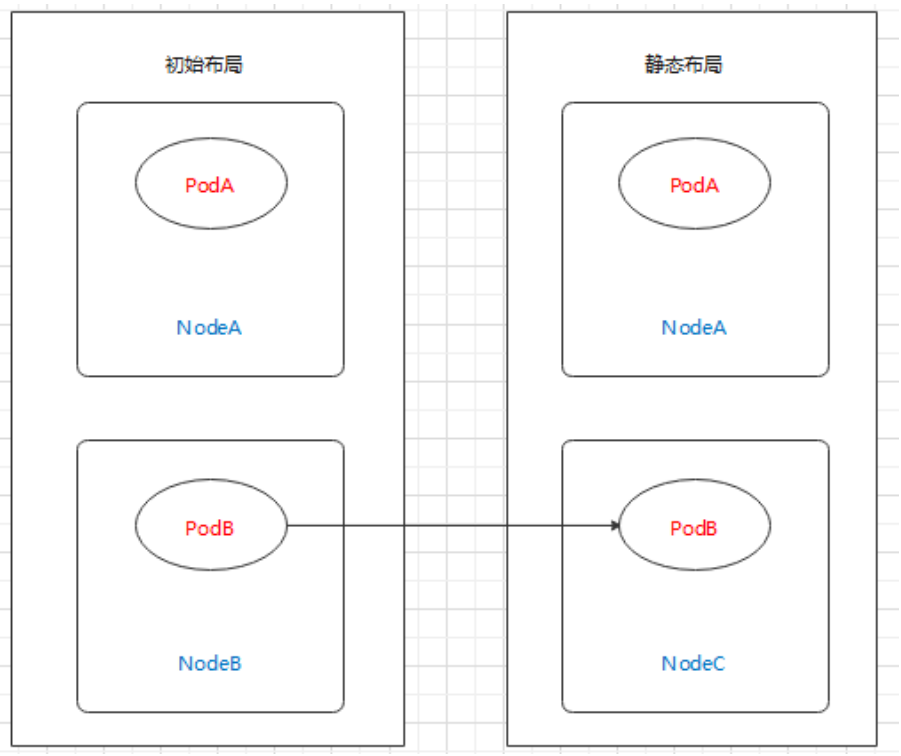

图 2 优化后（PodA 和 PodB 交换位置，PodA 不需要迁移）

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：

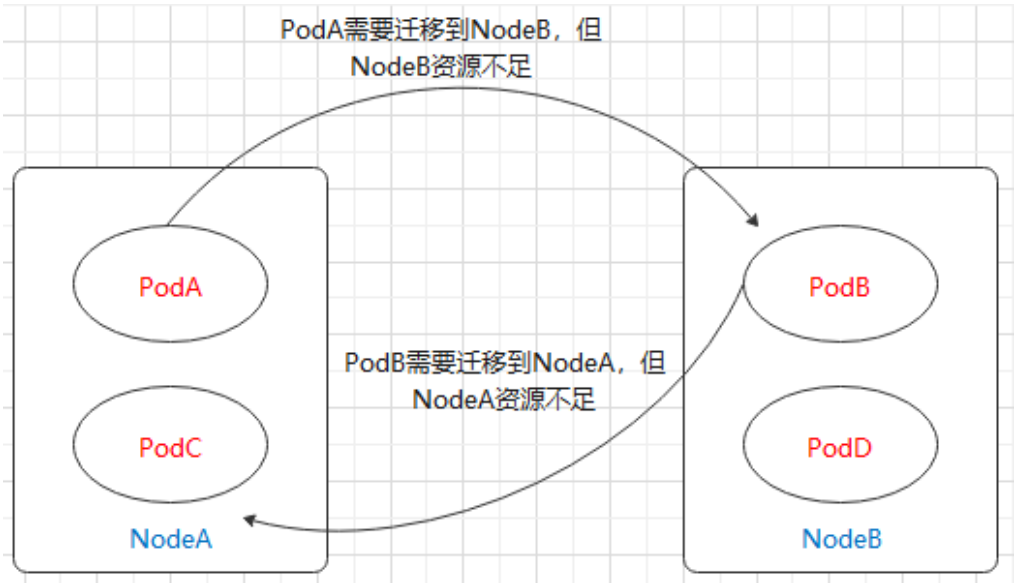

图 3 优化前（PodA 和 PodB 形成死锁）

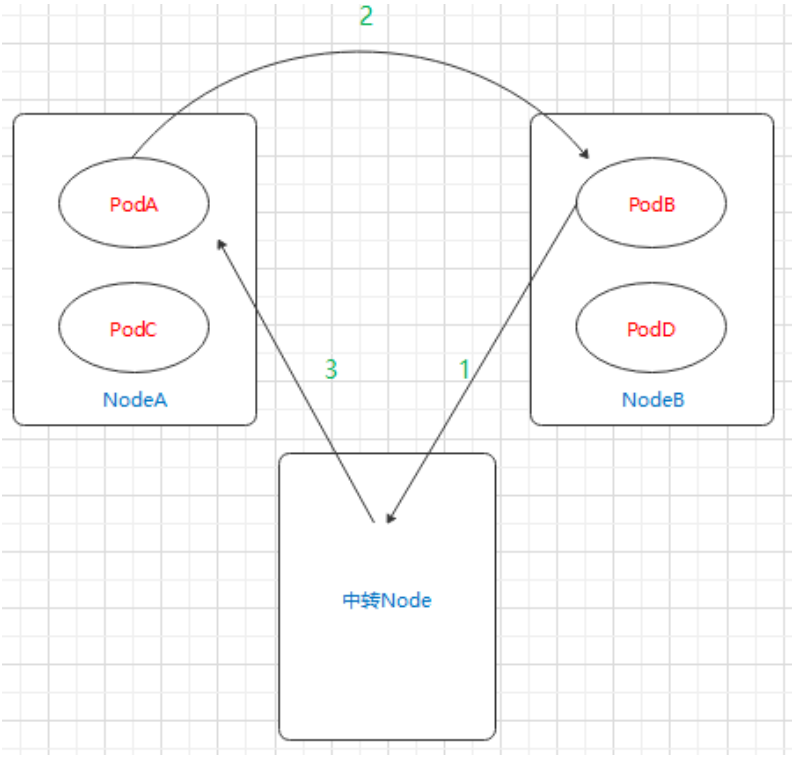

图 4 优化后（根据迁移顺序，解除死锁）

3 关键代码

3.1 静态布局

3.1.1 部署 Pod

Pod 根据 cpu/ram 的比例进行从小到大排序，在部署每个 pod 时，会对所有 node 进行排序，根据 pod 和 node 的资源不匹配分数从小到大排序。计算方式：从 gpu、cpu、ram 三个维度，计算 minRatio = node 可用资源/pod 所需资源。除比率最小的维度外，其他两个维度计算分数，计算公式：

$$score = \left(\frac{NS}{minRatio} - PS\right) * weight$$

其中 NS 为 node 可用资源，PS 为 pod 所需资源，weight 为该资源的比重。node 排序后，选择第一个满足约束条件的 node。

- 部署 pod 代码

```
Map<String, Integer> allMaxInstancePerNodeLimit =
ALL_MAX_INSTANCE_PER_NODE_LIMIT.get();

List<Pod> podList = pods.stream().map(Pod::copy).collect(Collectors.toList());
```

```
for (NodeAndPod nodeAndPod : nodeAndPods) {
        if (CollectionUtils.isEmpty(podList)) {
                break;
        }
        Utils.sortPod(podList, nodeAndPod, rule);

        List<Pod> selectedPods = new ArrayList<>();
        for (Pod pod: podList) {
                int maxInstancePerNodeLimit =
allMaxInstancePerNodeLimit.get(pod.getGroup());
                long beg = System.currentTimeMillis();
                if (Utils.fillOnePod(nodeAndPod, pod, maxInstancePerNodeLimit)) {
                        selectedPods.add(pod);
                }
                System.out.println("fillOnePod time:" + (System.currentTimeMillis() -
beg));
        }
        podList.removeAll(selectedPods);
}
```

- 计算资源不匹配分数的代码

```
double gpuRatio = nodeAndPod.getSurplusGpu() == 0 ? Integer.MAX_VALUE :
(double)nodeAndPod.getSurplusGpu() / pod.getGpu();
double cpuRatio = (double)nodeAndPod.getSurplusCpu() / pod.getCpu();
double ramRatio = (double)nodeAndPod.getSurplusRam() / pod.getRam();

double minRatio = Math.min(gpuRatio, Math.min(cpuRatio, ramRatio));
double score = 0;
if (gpuRatio != minRatio) {
        score += ((double)nodeAndPod.getSurplusGpu() / minRatio - pod.getGpu()) *
gpuValue;
}
if (cpuRatio != minRatio) {
        score += ((double)nodeAndPod.getSurplusCpu() / minRatio - pod.getCpu()) *
cpuValue;
}
if (ramRatio != minRatio) {
        score += ((double)nodeAndPod.getSurplusRam() / minRatio - pod.getRam())
* ramValue;
}
return score;
```

3.1.2 模拟退火

有三个参数来调整退火的速度，INIT_TEMPERATURE 设定初始的温度，初始温度设置的越高，获得更优解的几率更大， 但时间也会随之增加。LOOP 设定在同一温度下循环的次数。DELTA 表示退火的速率。每次产生新解p，都要根据分数S判定是否接受该解，如果分数更优，则接受p为新的当前解，否则按照概率$e^{\frac{-\Delta S}{T}}$，其中ΔS为分数差，T为当前温度，判断是否将p作为新的当前解。

- 模拟退火主体代码

```
AllocationStrategy.allocation(nodeAndPods, pods, rule);
int score =
ScoreUtils.scoreNodeWithPods(nodeAndPods.stream().map(NodeAndPod::incomp
leteCopy).collect(toList()), rule, groupRuleAssociates);

bestSchedule.setAndCopy(nodeAndPods, pods, score);
log.info("initAllocation score is {}", score);
double temperature = INIT_TEMPERATURE;

Allocation curAllcation = new Allocation();
curAllcation.setAndCopy(nodeAndPods, pods, score);

while(temperature > TERMINAL_TEMPERATURE) {
    for (int i = 0; i < LOOP; i++) {
        nodeAndPods =
nodeAndPodList.parallelStream().map(NodeAndPod::copy).collect(toList());
        newAllocationForSchedule(nodeAndPods, pods, rule);

        score =
ScoreUtils.scoreNodeWithPods(nodeAndPods.parallelStream().map(NodeAndPod:
:incompleteCopy).collect(toList()), rule, groupRuleAssociates);
        log.info("schedule score is {}", curAllcation.getScore());

        int scoreDiff = score - curAllcation.getScore();
        if (scoreDiff < 0) {
            curAllcation.setAndCopy(nodeAndPods, pods, score);
            if (score < bestSchedule.getScore()) {
                bestSchedule.setAndCopy(nodeAndPods, pods, score);
            }
        }
        else {
            double random = Math.random();
```

```
                if (Math.exp(-scoreDiff / temperature) > random) {
                    curAllcation.setAndCopy(nodeAndPods, pods, score);
                }
                else {
                    pods =
curAllcation.getPods().parallelStream().map(Pod::copy).collect(toList());
                }
            }
        }
        temperature *= DELTA;
}
```

3.2 动态迁移

3.2.1 统计迁移的 Pod

- 若静态布局前后 Pod 所在的 node 没有变化，则不进行迁移
- node 有改变，若布局后 PodA 初始 node 存在与其规格相同的 PodB，则 PodA 与 PodB 交换布局后的位置，PodA 还是在同一 node，不进行迁移，减少了迁移次数。
- node 有改变，若布局后不存在与 PodA 相同规则的 PodB，则需要进行迁移
  - ◆ 统计迁移的 Pod 的代码

```
List<MigratePod> migratePods = new ArrayList<>();

for (NodeWithPod initNwp : initNwps) {
        for (Pod initPod : initNwp.getPods()) {
            Pair<NodeAndPod, Pod> reAlloPos = isSamePosition(initNwp,
reAlloNaps, initPod);
            if (reAlloPos != null) {
                exchangePosition(initNwp, initPod, reAlloNaps,
reAlloPos.getValue(), reAlloPos.getKey());
            }
        }
}

for (NodeWithPod initNwp : initNwps) {
        for (Pod initPod : initNwp.getPods()) {
            Pair<NodeAndPod, Pod> reAlloPos = isSamePosition(initNwp,
reAlloNaps, initPod);
            if (reAlloPos != null) {
```

```
                NodeWithPod targetNwp = findNodeWithPodBySn(initNwps,
reAlloPos.getKey().getNode().getSn());
                migratePods.add(MigratePod.builder()
                  .sourceNwp(initNwp).targetNwp(targetNwp).initPod(initPod)
                  .reAlloPod(reAlloPos.getValue()).intrm(false).sameNode(false)
                  .build());
            }
        }
}
```

3.2.2 pod 迁移

- 将静态布局后空闲的 node 作为中转 node，pod 迁移时可能会存在 target node 约束不满足的情况，需要先迁移到中转 node，等约束条件满足后，再迁移至 target node。
- 首先尝试迁移至 target node，如果不能迁移，则迁移至中转 node，若中转 node 没有位置，则停止迁移，此次迁移失败，重新静态布局。
  - ◆ Pod 迁移主体代码

```
List<RescheduleResult> rescheduleResults = new ArrayList<>();
List<MigratePod> migratePods = collectMigrate(initNwps, reAlloNaps);
int stage = 1;
List<NodeWithPod> releaseNodes = new ArrayList<>();

for (NodeAndPod reAlloNap : reAlloNaps)
    if (reAlloNap.getPodNum() == 0)
        for (NodeWithPod initNwp : initNwps)
            if (reAlloNap.getNode().getSn().equals(initNwp.getNode().getSn()))
            {
                releaseNodes.add(initNwp);
                break;
            }
        }
    }
}

while (CollectionUtils.isNotEmpty(migratePods)) {
    List<MigratePod> hasMigratePods = new ArrayList<>();
    for (MigratePod migratePod : migratePods) {
        boolean isMigrate = isMigrate(migratePod);
```

```
            if (isMigrate) {
                hasMigratePods.add(migratePod);
                 rescheduleResults.add(RescheduleResult.builder().stage(stage++)
                            .podSn(migratePod.getInitPod().getPodSn())
                            .cpuIDs(migratePod.getReAlloPod().getCpuIDs())
                            .sourceSn(migratePod.getSourceNwp().getNode().getSn()
                            )
                            .targetSn(migratePod.getTargetNwp().getNode().getSn())
                            .build()
                );
            }
        }
        if (CollectionUtils.isEmpty(hasMigratePods)) {
            if (!migrateIntrm(migratePods, releaseNodes, rescheduleResults, stage,
rule)) {
                log.error("handle deadlock failed.");
                break;
            }
            stage++;
        }
        migratePods.removeAll(hasMigratePods);
}
```

复赛（语言 Golang）

4 赛题背景分析及理解

4.1 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。

4.2 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。

5 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：

选取 node 的策略

- 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。
- 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。
- 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。

选取 container 的策略

- 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。
- 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。

函数分类

- 当容器使用内存高于分配内存的 30%时，判定为内存密集型函数。
- 当容器 cpu 使用率大于 10%时，判定为 cpu 密集型函数。
- 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。

动态缩减

- 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。
- 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。

- 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。

回收资源

- 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 3 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：

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。

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。

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。

6 关键代码

6.1 AcquireContainer 接口

先从 containerMap 中搜索符合条件的 container，如果存在直接返回，不存在就去申请 node 创建新容器。

- acquireContainer 接口的主体代码

```
var res *ContainerInfo
funcName := req.FunctionName

r.functionMap.SetIfAbsent(funcName, cmap.New())
fmObj, _ := r.functionMap.Get(funcName)
containerMap := fmObj.(cmap.ConcurrentMap)

res = r.getExistedContainer(containerMap, funcName)

//如果当前有短时间函数运行，则非短时间的 cpu-intensive 和 mem-intensive 类型函数等待其运行完，或者等待超时后继续运行
if shortTimeType.Has(funcName) && res != nil {
    r.shortFuncRunning.SetIfAbsent(res.id, 1)
} else if (memIntensiveType[funcName] == 1|| cpuIntensiveType[funcName] == 1)
   && !shortTimeType.Has(funcName) && res != nil
   && !r.shortFuncRunning.IsEmpty() {
       fmt.Printf("[INFO] req:%s, func{%s} wait for short-time func to finish or
   time out\n", req.RequestId, funcName)
       now := time.Now()
       for !r.shortFuncRunning.IsEmpty() && time.Since(now).Nanoseconds() <=
   shortTimeLine {
           time.Sleep(3 * time.Millisecond)
       }
```

```
        fmt.Printf("[INFO] req:%s, func{%s} wait is over.waiting time is %d\n",
    req.RequestId, funcName, time.Since(now).Nanoseconds())
}

if res == nil {
    lock.Lock()
    r.ctCreating.SetIfAbsent(funcName, 0)
    ctObj, _ := r.ctCreating.Get(funcName)
    ctNum := ctObj.(int)
    if ctNum >= 20 && memIntensiveType[funcName] !=1
        &&cpuIntensiveType[funcName] != 1 {
        lock.Unlock()
        for res == nil {
            res = r.getUnconstrainedContainer(containerMap)
            time.Sleep(5 * time.Millisecond)
        }
    } else {
        node, err := r.getNode(req.AccountId, req.FunctionConfig.MemoryInBytes,
funcName)
        lock.Unlock()
        if err != nil {
            fmt.Printf("[ERROR] failed to get node. req:%s\n", req.RequestId)
            return nil, nil
        }
        res, err = r.createContainer(node, req)
        if err != nil {
            fmt.Printf("[ERROR] failed to create container. req:%s, node:%s\n",
req.RequestId, node.nodeID)
            return nil, nil
        }
        res.requests = 1
        node.containerMap.Set(res.id, res)
        containerMap.Set(res.id, res)
        r.containerMap.Set(res.id, res)
    }
}
```

◆ 获取现有 containers 的代码（有约束）

该段代码有容器运行实例数约束，每个容器只允许同时运行一个实例，当容器满载时，会提前申请预留容器。

```
var res *ContainerInfo
if containerMap.Count() == 0 {
    return nil
}
```

```
for i, key := range containerMap.Keys() {
    cmObj, ok := containerMap.Get(key)
    if !ok {
        continue
    }
    r.ctCreating.SetIfAbsent(funcName, 0)
    ctObj, _ := r.ctCreating.Get(funcName)
    ctNum := ctObj.(int)

    container := cmObj.(*ContainerInfo)
    container.Lock()
    if container.deleting {
        container.Unlock()
        continue
    }
    if i == containerMap.Count() - 1 && ctNum < 20 {
        go r.reserveContainersInAdvance(containerMap, container)
    }
    if container.requests < 1 {
        res = container
        res.requests++
        container.Unlock()
        break
    }
    container.Unlock()
}
return res
```

◆ 获取现有 node 的代码

```
//1）如果并发请求数超过当前 node 数量且不超过 initNodeNum，则获取新的 node，可以更快的创建容器和更快的响应时间
//2）否则尽量分布均匀，首先以 node 内该函数容器的数量排序，优先数量少的 node，如果数量相同，优先可用内存大的 node
//3）如果 initNode 不能满足请求，则获取新的 node 来满足请求，规则如 2）
if r.nodeMap.Count() < initNodeNum && expand {
    for _, key := range r.nodeMap.Keys() {
        nmObj, ok := r.nodeMap.Get(key)
        if !ok {
            continue
        }
        node := nmObj.(*NodeInfo)
        node.Lock()
        if !node.functionMap.Has(funcName) && node.availableMem >= memoryReq {
            res = node
```

```
            }
            node.Unlock()
        }
    }   else {
        var bestFuncNum int
        var bestAvailMem int64
        for _, key := range r.nodeMap.Keys() {
            nmObj, ok := r.nodeMap.Get(key)
            if !ok {
                continue
            }
            node := nmObj.(*NodeInfo)
            if node.nodeID == exclude {
                continue
            }
            node.Lock()

            var funcNum int
            fmObj, ok := node.functionMap.Get(funcName)
            if ok {
                funcNum = fmObj.(int)
            } else {
                funcNum = 0
            }
            if node.availableMem >= memoryReq &&
                (res == nil || funcNum < bestFuncNum || (funcNum == bestFuncNum
&& node.availableMem > bestAvailMem)) {
                res = node
                bestFuncNum = funcNum
                bestAvailMem = node.availableMem
            }
            node.Unlock()
        }
    }
```

6.2 ReturnContainer 接口

- 若执行函数失败，则释放该容器，失败原因有 OOM、执行时间超时等，容器 OOM 则不能再继续执行函数。
- 根据函数的执行时间判断是否为短时间型，getStat 分析容器的 cpu 使用情况，判定是否需要调整资源，当需要调整时，创建新容器后释放旧容器。

◆ ReturnContainer 接口主体代码

```
if req.ErrorCode != "" || req.ErrorMessage != "" {
    //if the function invoke fails, remove the container
    r.handleContainerInvocationErr(req.RequestId, container)
} else {
    container.Lock()
    defer container.Unlock()
    container.requests--
    container.activeTime = time.Now().UnixNano()
    container.usedMem = req.MaxMemoryUsageInBytes
    //根据函数执行时间判断是否为短时间类型函数。如果已经判断为短时间类型，并且当前执行时间超过上限，取消认定该函数为短时间类型
    if req.DurationInNanos <= shortTimeLine
&& !shortTimeType.Has(container.funcName) {
        fmt.Printf("[INFO] analyzed funcName{%s} is short-time type.\n",
container.funcName)
        shortTimeType.SetIfAbsent(container.funcName, 1)
    } else if shortTimeType.Has(container.funcName) && req.DurationInNanos >
limitTimeLine {
        fmt.Printf("[INFO] analyzed funcName{%s} isn't short-time type.\n",
container.funcName)
        shortTimeType.Remove(container.funcName)
    }
    //根据 getStat 的分析，调整容器的资源
    if modifyMap.Has(container.funcName) && container.requests == 0 {
        mmObj, _ := modifyMap.Get(container.funcName)
        memory := mmObj.(int64)
        if container.totalMem < memory {
            cmObj, _ := r.functionMap.Get(container.funcName)
            containerMap := cmObj.(cmap.ConcurrentMap)
            nmObj, ok := r.nodeMap.Get(container.nodeId)
            if ok {
                node := nmObj.(*NodeInfo)
                go r.modifyContainer(container.nodeId, containerMap, container,
memory, node)
            }
        }
    }
}
```

◆ 容器迁移的代码（资源调整复用该段代码）

先创建新容器，创建完成后将旧容器标记 deleting，当有新请求搜索时，会跳过标记 deleting 的容器。旧容器没有在执行函数就会释放，否则等待其执行完毕。

```
//先创建新容器，再删除旧容器，实现转移容器
r.createNewContainer(newNodeId, containerMap, container, memSize)
container.Lock()
container.deleting = true
container.Unlock()
for container.requests > 0 {
    time.Sleep(5 * time.Millisecond)
}
r.releaseContainer(container, containerMap, node)
fmt.Printf("[INFO] modify the container memory size. funcName:%s,
memSize:%d\n", container.funcName, memSize)
```

7 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。复赛在通用性和针对性也很难取舍，通用性方案能在不同数据下保持不错的稳定分数，但始终跑不过针对性方案，只要有足够的提交机会，针对性方案就可以根据数据逐步上升，不过最终还是选择更通用性的方案。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。