关于云计算资源调度方法的研究综述

摘要

云资源如何调度才能使系统收益最大是云计算领域的关键问题之一，不同场景有着不同的调度优化目标，设备之间的异构性更加大了对云资源调度器与调度算法的设计难度。本文分别通过对工业界与学术界两个具体的调度方法的解析来总结现有方法中值得借鉴与学习的地方。工业界主要介绍 Nova-Scheduler 的调度方法，其采用过滤-称重的方法筛选任务所需的主机；学术界主要介绍 WM-RDQN 算法的调度方法，其算法对如何卸载任务（即选择在本地、边缘端或者云端处理任务）能使调度系统的时延、能耗以及用户的服务质量这三者的加权收益达到最大值。本文通过分析两个不同业界对相同问题的解决方式总结出两者的不同点以及对于未来调度方法的设计值得学习借鉴的地方。

关键词： 云计算；云资源调度；Nova；深度学习；时延；能耗；服务质量

A Survey of Cloud Computing Resource Scheduling Method

Abstract

How to schedule cloud resources to maximize system revenue is one of the key issues in the field of cloud computing. Different scenarios have different scheduling optimization objectives, and the heterogeneity between devices makes it more difficult to design cloud resource scheduler and scheduling algorithm. This paper analyzes two specific scheduling methods of industry and academia respectively to summarize the existing methods worth learning and learning. The industry mainly introduces the scheduling method of Nova-scheduler, which uses the filter-weight method to screen the hosts required by the task. The academic community mainly introduces the scheduling method of WM-RDQN algorithm. Its algorithm can make the weighted revenue of time delay, energy consumption and quality of service of the scheduling system reach the maximum on how to offload tasks (that is, choose to process tasks in the local, edge or cloud). This paper analyzes the solutions of two different industries to the same problem and summarizes the differences between the two and the design of future scheduling methods worth learning.

Keywords： Cloud Computing; Cloud Resource Scheduling; Nova; Deep-Learning Delay; Energy Consumption; Quality of Service

0 引言

随着移动互联网技术和移动通信技术的高速发展，AR/VR、物联网、车联网等新业务场景也随之出现。然而这些业务场景的实现不仅需要网络带宽资源的保证，还需要计算处理数据能力足够强大的服务器以及大量的计算资源支持。

云计算中心与边缘计算设备凭借着其强大的计算与数据处理能力成为支持新业务的重要信息基础设施。如今，已经有许多互联网巨头建立了自己的云计算数据中心，如亚马逊、阿里、华为、腾讯等。他们将云计算资源集中起来形成资源池，对资源池进行统一管理。当客户端需要使用这些资源时，客户端通过向资源池发送任务请求来获取相应的云服务。然而如何高效、高服务质量、低成本的调用这些资源是目前云服务行业中面临的重大挑战之一。

资源调度任务本身是一个 NP-难问题，而云中心与边缘设备、边缘设备之间的异构性以及虚拟化技术的发展更加大了资源调度任务的难度与复杂度。本文对工业界以及学术界的云计算资源调度方法进行了研究，按照资源调度任务的执行流程对相关的调度方法进行整理并提炼其可以在实际应用当中可以借鉴的思想。

1 工业界的调度方法（Nova-Scheduler）

工业界的调度方法相比学术界的更加贴近实际生产应用，追求的是高效性与实用性，所以在调度方法的设计与测试时会不断地进行简化，包括应用更简单有效的算法以及删改过于繁琐的步骤。在调度执行的过程中主要实现调度功能的硬件设备叫做调度器（scheduler），也是调度执行中最为重要的组件。下面将以 OpenStack 的 Nova-Scheduler 为例进行分析。

1.1 Nova 组件

Nova 是 OpenStack 中最早出现的模块之一，是 OpenStack 中最核心的服务模块，主要负责管理和维护云计算环境的计算资源以及整个云环境虚拟机生命周期。

Nova 由 API、Compute、Conductor、Scheduler4 个核心组件所组成，它们之间通过 RPC 行通信，对外通过 HTTP 进行通信，分别实现不同的功能，下面将详细介绍:

Nova-API：

作为 Nova 组件对外的唯一窗口，向客户暴露 Nova 能够提供的功能。 当客户需要执行虚拟机相关的操作，能且只能向 Nova-API 发送 REST 请求，创建虚拟机实例的请求也要通过该接口才能调用 Nova 中的其他组件的服务。

Nova-Compute：

在计算节点上运行，负责管理节点上的实例。每隔一段时间，Nova-Compute 就会报告当前计算节点的资源使用情况和 Nova-Compute 服务状态。这样 OpenStack 就能得知每个计算节点的 Vcpu、Ram、Disk 等信息。Nova-Scheduler 的很多 Filter 才能根据计算节点的资源使用情况进行过滤，选择符合 Flavor 要求的计算节点。同时，OpenStack 对实例最主

要的操作都是通过 Nova-Compute 实现的，包括实例的启动、关闭、重启、暂停、恢复、删除、调整实例大小、迁移、创建快照等。

Nova-Conductor：

为计算节点提供数据库访问支持。Nova-Compute 服务和数据库之间的中间件。将 nova-compute 访问数据库的全部操作都放到 Nova-Conductor 中，而且 Nova-Conductor 是部署在控制节点上的。这样就避免了 Nova-Compute 直接访问数据库，增加了系统的安全性。与此同时，这种松散的架构允许配置多个 Nova-Conductor 实例。在一个大规模的 OpenStack 部署环境里，管理员可以通过增加 Nova-Conductor 的数量来应对日益增长的计算节点对数据库的访问，更好的实现了系统的伸缩性。

Nova-Scheduler：

将创建新虚拟机的请求调度到正确的节点。当创建实例时，用户会提出资源需求，例如 CPU、内存、磁盘各需要多少。OpenStack 将这些需求定义在 Flavor 中，用户只需要指定用哪个 Flavor 就可以了。Nova-Scheduler 会按照 Flavor 去选择合适的计算节点。

DB：用于数据存储的 SQL 数据库，通常是 MariaDB。

图 1-1 是 Nova 架构图，红色框内为 Nova 架构的组件，框外为其他 OpenStack 组件模块。

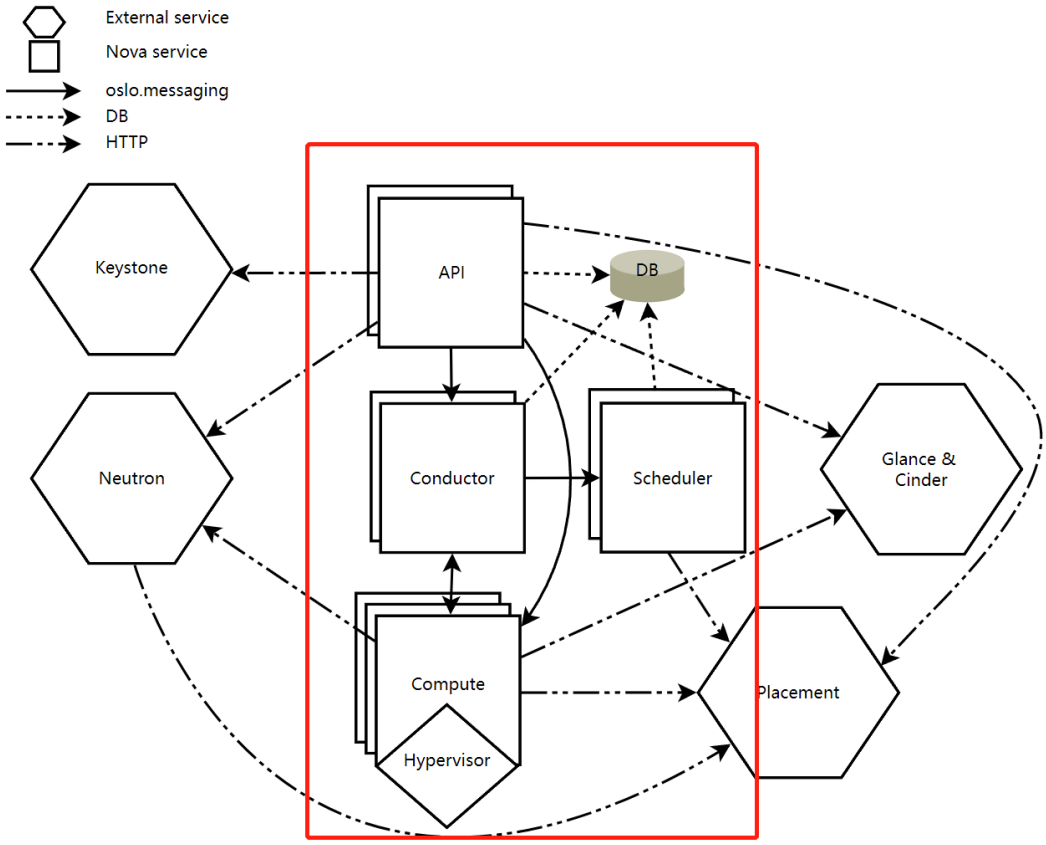

图 1-1

1.2 Nova-Scheduler 调度流程

OpenStack 对用户请求进行调度，都是通过 Nova 实现的，其中最关键的 Nova 组件为 Nova-Scheduler，步骤主要分为两步，先通过过滤器过滤掉不符合要求的主机，再通过权值计算对通过过滤的主机进行最优选择，下面是详细的调度流程。

用户通过 Novaclient 生成创建虚拟机实例请求。API 在监听到 HTTP 请求之后，将其转化为 AMQP 信息(RPC 的一种，为完整配置服务线路信息)。同时 API 通过 RPC 调用 Conductor 服务，Conductor 在收到消息队列中的 AMQP 信息后将其转发给 Scheduler。

Scheduler 收到信息之后将信息初始化，如果选择用 FilterScheduler（过滤器调度器）对主机进行筛选则流程如下（如图 1-2）：获取需要创建的虚拟机实例信息，通过解析虚拟机实例信息来更新过滤器属性。过滤器属性更新之后，对主机进行过滤，首先获取可用的计算节点并在节点上设置基本信息先将不可用的主机以及计算状态不活跃的主机删除，留下可用的主机。之后根据解析虚拟机实例信息过后的信息定义所要使用的过滤器（决定使用哪些过滤器，哪些不使用），用这些过滤器将主机筛选出来并返回给 Scheduler。Scheduler 再用加权的方式对返回的主机进行称重，得到每个主机的权值，根据这些权值将主机排序。再根据需要选择的主机数目选择排名最优的几个主机返回给 Conductor 并更新主机状态和主机资源列表，为下一次主机调用做好准备。

如果选择 ChanceScheduler（随机调度器）对主机进行随机选择，那么就从 compute 服务正常且该不在指定的 ignore_hosts 列表（不符合基本要求的主机列表）中的主机当中随机选择一台或者几台主机返回给 Conductor。其中，ChanceScheduler 中还有一种特殊的调度器叫 Caching Scheduler，它在随机调度的基础上将主机资源信息缓存在本地内存中，然后通过后台的定时任务定时从数据库中获取最新的主机资源信息。

Conductor 收到返回的主机之后调用 Compute 服务用来创建虚拟机实例，最后 Compute 通过与 Hypervisor 通信对虚拟机实例的生命周期和主机状态进行管理。

此外，具体的一些 Filter 类型以及 weigher 类型以及它们相关的作用和源代码我都整理在了 nova 图表当中。链接（按住 ctrl 点击访问）：工业界-nova\nova 图表.xlsx

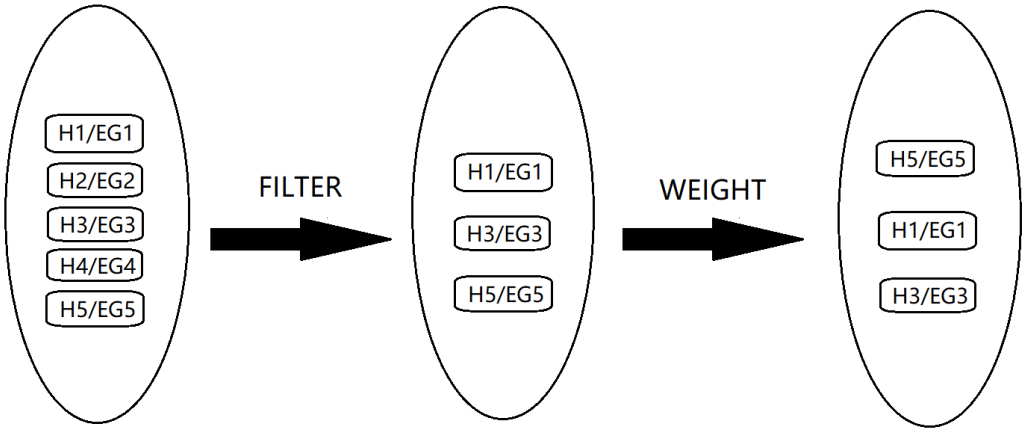

图 1-2 FilterScheduler 调度流程

1.3 Nova-Scheduler 错误应对机制

作为工业界的调度方法是需要投入到实际应用当中的，自然也会有错误应对机制（容灾机制），下面会对 Nova-Scheduler 一些调度错误的应对机制进行举例分析。

1.3.1 过滤之后发现没有可用主机

在对主机进行过滤的时候，所有主机都不符合要求，全部被过滤掉，这时会给用户返回一个报错：“no valid host was found”。

这时有两个方法应对此错误。

如果是因为资源不足可以添加节点或者服务器，以达到过滤要求。

如果不想添加节点或者服务器就需要增大 CPU，内存，磁盘的超配比，以达到处理任务的要求。但是如果物理机的本身的剩余资源不多了，增大超配比是一个很危险的操作，很容易造成系统本身的崩溃，所以在操作之前建议通过，top 等命令查看系统本身资源是否充足。

1.3.2 在选择的主机上创建虚拟机实例失败

在过滤-称重得到目标主机之后，在主机上创建虚拟机实例失败，这时会给用户返回报错“no valid host”。

首先需要检查是否是由于主机资源不足造成的。先通过 Nova host-describe 检查主机资源是否真的不足，若 host UUID 真的资源不足，需要扩容计算节点，或者降低虚拟机规格。

如果看不是，CPU 资源还很多，那就是有问题了，因为 Nova-Scheduler 日志打印的和 Nova host-describe 查出来的不一致，这种情况出现的原因是上层下发 FS 创建虚拟机请求，而 FS 上创建虚拟机失败，且上层不在自动删除失败虚拟机的场景，会导致 Nova-Compute 和 Nova- Scheduler 中资源数据不一致，可以使用以下方法解决。

1. 如果虚拟机还没有删除掉，为保证上下（如 vcmm 和 fs）数据一致，需要从上层触发删除虚拟机操作，否则，使用如下指令查询创建失败的虚拟机：

   ```
   root# nova list --all-te | grep ERROR
   root# nova delete vm-error01 vm-error02
   ```

2. 还可以使用如下指令重启 nova-scheduler 组件，使 Nova-Compute 和 Nova-Scheduler 中资源数据一致：

   ```
   root# cps host-template-instance-operate --action stop --service nova nova-scheduler
   root# cps host-template-instance-operate --action start --service nova nova-scheduler
   ```

1.4 工业界在 Nova 架构基础上增加优化算法

在工业界有不少云服务平台是基于 OpenStack 的框架搭建的，华为云就是其中之一，他们所采用的调度策略包括了 Nova-Scheduler 的过滤和称重，但也加入了一些其他的算法和步骤对其进行优化，其中应用最广泛的两个算法分别是贪心算法和遗传算法，它们逻辑简单、代码简洁、筛选效果突出，大大加快了主机的筛选速率，使得执行算法后面的流程当中所需要的遍历的主机实例更少，能够减小大部分的处理时延，使得总处理时延更低，提升用户的服务质量。下面将以贪心算法举例说明增加算法步骤所能带来的筛选效益。

贪心算法的主要思想就是在每一个决策点，都是做出当前看来的最优选择，一般流程如下：

1. 将最优化问题简化为这样的形式：做出一个选择以后，只剩下一个子问题需要求解
2. 证明在做出贪心选择以后，原问题总是存在最优解，即贪心选择总是安全的
3. 证明在做出贪心选择以后，剩下的子问题满足性质：其最优解与做出选择的组合在一起得到原问题的最优解，即最优子结构

由此可以看出主要的两个影响因素为贪心选择和最优子结构。

贪心选择即如何对需要进行贪心算法的变量进行选择才能得到最优解。

最优子结构即如果一个问题的最优解包含其子问题的最优解，那么就称这个问题具有最优子结构性质。在贪心算法中，我们总是在原问题的基础上做出一个选择，然后求解剩下的唯一子问题。

下面将结合 Nova-Scheduler 举例说明：

问题提出：

在过滤器过滤之后剩下 100 台主机，还想做进一步的过滤，希望得到处理时延尽量低且能耗尽量低的主机再进入计算权值步骤，并希望进一步的过滤能够使筛选步骤的时延比原先更小。

分析：

由于处理时延和能耗均与 CPU 的计算周期有关，而且两者为类反比例关系，即一定范围内能耗越高可以使得处理时延越低，那么想要使得两者都尽可能低，可以在原有问题上面建立一个贪心选择，即选择减小单位时延所需要增加的能耗尽量小的主机。

求解：

由于贪心算法的所得到的解不一定为最优解，且使用贪心算法之后还有遍历主机的计算权值步骤，因此不需要在贪心算法步骤就得到最终需要返回的主机，可以将子问题的解转化为一个相对的筛选比例，比如过滤之后还剩 100 台主机，设置的筛选比例为 20%，那么经过贪心算法步骤之后，剩下不超过 20 台主机进入计算权值步骤。接下来，遍历所有主机的“减小单位时延所需要增加的能耗”的值，进行排序，筛选出最优的 20 台主机，即 “减小单位时延所需要增加的能耗”最小的 20 台主机，进入下一步的权值计算。

筛选过程是否得到了优化：

增加了贪心算法的步骤之后，所需要遍历的主机从 100 台变成了 120 台，原先对遍历的 100 台主机进行的操作为权值计算，而求解之后的对主机遍历进行的操作为 100 台主机进行贪心算法，20 台主机进行权值计算，若满足 80 台主机的权值计算时延比 100 台主机的贪心算法时延高，则说明加入贪心算法能够降低整个筛选主机的处理时延。权值计算的计算量包括了 9 个系统自带称重量以及若干自定义称重量，而贪心算法的计算量为减小单位时延所需要增加的能耗，从计算量以及代码的繁杂程度来看，有理由相信 80 台主机的权值计算时延比 100 台主机的贪心算法时延高，因此筛选过程得到了优化。

1.5 工业界调度方法总结

总的来说，工业界的调度方法逻辑简单有效，优化算法也是简单高效的，其可以借鉴的地方是 Nova 架构与优化算法如何进行结合以及如何选择优化算法，我有以下看法：

1. Nova-Scheduler 的过滤-称重这种简单明确且有效的筛选主机机制是值得借鉴的。通过逻辑简单的两步即可高效的筛选出需要的主机或主机群，同时能最大可能保证筛选出的主机或主机群是符合用户的要求，这点很好的符合了实际应用中的简洁高效的要求。

2. Nova-Scheduler 可以自定义过滤器和称重函数也是值得学习的。用户可以结合自身对主机或者主机群的要求自定义过滤器和称重函数，同时也可以通过自定义的过滤器和称重函数来实现一些优化算法的功能，使算法在 Nova-Scheduler 可以很好地与其他过滤器并存，即其他算法或者过滤器的加入不会影响原先过滤器的正常使用，是 Nova-Scheduler 自主性、灵活性的一种体现。

3. Nova 模块如何实现不重复的功能同时保证其整体的容灾能力也是值得借鉴学习的。对于工业界来说，不同的架构会对应不同的调度方法，但其架构是否优秀更多的取决于架构中各个模块的分工是否明确以及模块之间是否解耦，对于 Nova-Scheduler 来说，其模块分工明确，而且不会因为某一模块出故障就会导致整体的功能不可用。

4. 对于优化算法的选择同样也要基于实际应用效果，即是否能达到优化的效果。其逻辑应是尽可能简单的条件下保证算法能有效地解决所需要优化的问题。目前工业界中应用最广泛的两种算法为贪心算法与遗传算法，以及二者的结合算法，但不限于这两种算法，算法选择的思路可以但不限于寻找逻辑与优化问题的解题思路相符合的算法以及多个算法的逻辑结合使问题能够得到最大程度的优化。

2 学术界的调度方法（WM-RDQN 算法）

学术界的调度方法相比于工业界，适用的环境相对理想化，方法流程相对复杂，但是不完全是没有学习与借鉴的价值，下面将结合 WM-RDQN 算法调度流程对其值得学习与借鉴的地方进行提炼总结。

2.1 问题建模

问题的建模分为几步，首先是对问题的建模，再接着是分别对时延、能耗以及服务模型这三个影响因素进行建模分析，将问题模型化。原则是让三个影响因素用相同的变量来进行衡量，在这篇文章当中，以任务所需计算周期数以及 CPU 的计算频率作为变量。其他一些变量的符号以及对应的名称如图 2-1:

变量	解释
U	用户集合
M	MEC 服务器
C	云端服务器
W	串行任务集合
t	单位时间步
W_u	用户终端 u 当前拥有的串行任务
T_u	用户终端 u 当前任务的预计完成时间
R_u	用户终端 u 与无线基站的通信带宽
Q_u	用户终端 u 的服务质量保证
C_m	MEC 服务器的计算能力
T_m	MEC 服务器当前任务的预计完成时间
R_m	MEC 服务器与云服务器的连接带宽
C_c	云服务器的计算能力
T_c	云服务器当前任务的预计完成时间
$maxT$	任务时延上限要求
X	卸载策略
dT	任务时延
E	计算能耗
Qos	用户服务质量保证
c_t	任务所需 CPU 周期数
d_t	任务计算所需数据量
d_t'	任务计算结果回传数据量
dt	MEC 服务器与云端服务器传播时延
f_u	用户终端 u 的 CPU 频率
f_m	边缘服务器的 CPU 频率
f_c	云端服务器的 CPU 频率
P_{up}	用户终端在无线传输时消耗的能量
P_{idle}	用户终端空闲时消耗的能量
q_u	用户设备优先级
$\mathbf{A_u}$	用户设备权值矩阵

图 2-1

2.1.1 时延模型

在调度过程当中，主要的卸载时延分为本地、边缘和云端。

本地主要为处理时延，如下：

$$dT_u^{local} = \frac{c_t}{f_u}$$

边缘端离用户端较近，本文当中忽略了传播时延，因此本文边缘端时延主要为上传回传的传输时延，任务处理时延以及任务排队等待时延:如下：

$$dT_u^{up} = \frac{d_t}{R_u} + \frac{c_t}{f_m} + \frac{d_t'}{R_u} + T_m$$

云端离用户端较远，需要考虑传播时延，因此云端时延主要为上传回传的传输传播时延，任务处理时延以及任务排队等待时延，如下：

$$dT_u^{upc} = \frac{d_t + d_t'}{R_u} + \frac{d_t + d_t'}{R_c} + \frac{c_t}{f_c} + 2dt_{mc} + T_c$$

因此，时延模型可建为：

$$dT_{task} = \sum_{n=1}^{N} X|dT_u^{local}\ dT_u^{up}\ dT_u^{upc}|$$

2.1.2 能耗模型

在调度过程当中，能耗主要与 CPU 的计算功率有关，其功率与计算频率的关系为：

$$P = \kappa f^3$$

本地计算能耗主要为本地任务处理能耗，可表示为：

$$E_u^{local} = \kappa(f_u)^2 c_t$$

边缘端由于要上传以及回传，所以主要能耗为传输期间的能耗，边缘服务器任务处理能耗以及边缘服务器中排队等待传输的能耗，可表示为：

$$E_u^{up} = P_{up}(\frac{d_t}{R_u} + \frac{d_t'}{R_u}) + P_{idle}(\frac{c_t}{f_m} + T_m)$$

云端与边缘端相比，还要考虑到在传播前等待的能耗，因此主要能耗为传播等待能耗，传输期间的能耗，云服务器任务处理能耗以及云服务器中排队等待传输的能耗，可表示为：

$$E_u^{upc} = P_{up}\frac{d_t}{R_u} + P_{idle}(\frac{d_t + d_t'}{R_c} + \frac{c_t}{f_c} + 2dt_{mc} + T_c)$$

因此，总的能耗模型可以表示为：

$$E_{task} = \sum_{n=1}^{N} X|E_u^{local}\ E_u^{up}\ E_u^{upc}|$$

2.1.3 服务质量模型

在调度过程中，用户感受到的服务质量主要考虑动态的时延以及用户设备的优先级对服务质量的影响，其中动态时延对服务质量的衡量评估结果如下：

$$Q_t = e^{-\frac{dT_{t-1} - maxT_{t-1}}{(dT - maxT)}}$$

总的服务质量模型如下，其中α和β为加权系数：

$$Qos_u = \alpha Q_t + \beta q_u$$

2.1.4 汇总问题建模

在对三个影响因素建模完成之后，可以将问题的各个条件用如下公式来表示。

$x_{local} = 1$表示任务卸载在本地处理，$x_{mec} = 1$表示任务卸载在边缘服务器处理，$x_{cloud} = 1$表示任务卸载在云服务器处理，0 表示空操作，即不卸载在本地或边缘服务器或云服务器。可用以下表达式表示：

$$X_t = [x_{local} \quad x_{mec} \quad x_{cloud}]$$

$$x_{local} + x_{mec} + x_{cloud} \leqslant 1, \forall X_t \in X$$

$$x = \{0, 1\}, \forall x \in X_t$$

任务的处理时延不可以超过任务的最大时延上限：

$$\sum_{task}^{N} dT_{u,task} \leqslant maxT_w, \forall u \in U, \forall w \in W$$

在单位时间当中全部用户端产生的总的任务计算需求，不能超过系统中所有 CPU 的能够提供的总算力：

$$\sum_{w}^{W}\sum_{u}^{U} c_t^{u,w} dt \leqslant \sum_{u}^{U} f_u + f_m + f_c, \forall t \in \Delta T, \forall w \in W$$

串行任务是实际应用的任务类型当中最常见的任务类型，每个串行任务W由N个微任务以及对大完成时延上限构成，并且串行任务的起始任务（上传任务）必须在本地完成：

$$W = \{task_1, \ldots, task_N, maxT\}$$

$$x_{local} = 1, \forall task_1 \in w, \forall X_t \in X$$

每个用户端在任意的时间，最多拥有一项串行任务：

$$w_u \leqslant 1, \forall t \in \Delta T, \forall u \in U$$

在以上条件的制约下，最终的问题建模可这样表示，在任务最优化的卸载策略X_t下，如何使系统总收益最大化，下式为目标优化值，值越小，系统总收益越大：

$$F = \min \sum_{u=1}^{U} \sum_{task=1}^{N} (\lambda_t dT_{u,task} + \lambda_e E_{u,task} - \lambda_q Qos_{u,task})$$

$$\lambda_t + \lambda_e + \lambda_q = 1$$

2.2 深度 Q 学习算法

强化学习的核心思想是使用智能体与环境进行交互，从交互的反馈-奖励循环来学习到最优策略，其主要构成如图 2-2：

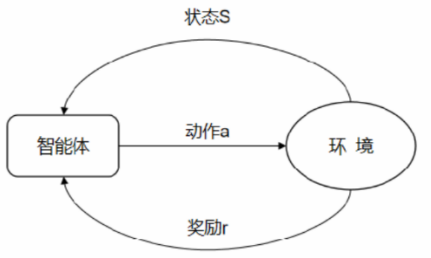

图 2-2

强化学习主要可以分为基于值函数与基于策略函数两种类型，值函数类型中的 Q 学习算法是最为常见的，其函数公式如下：

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

s为状态向量，a为动作向量，r为奖励值，γ是期望回报（0 到 1 取值，越大越趋向于更长远的利益），s'为上一次状态向量，a'为上一次动作向量。中括号内为每次对 Q 函数更新的部分，通过在状态 s 下选择动作 a 产生的 Q 函数值来更新 Q 函数。通过对 Q 函数不断地更新让每次动作获得最大的奖励值，从而实现决策动作的最优化。

策略函数类型中往往采用策略梯度算法，公式如下：

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta (s_t, a_t) v_t$$

s_t指步骤 t 时的状态向量，a_t指步骤 t 时的动作向量，$\pi_\theta(s,a)$为参数θ下的策略概率分布函数，v_t为蒙特卡洛法在步骤 t 时模拟的价值函数，通过梯度方法使函数的选择尽可能的

接近策略。

Q 函数所对应的 Q 值为在某状态下选择某动作的价值，Q 值越大说明在该状态下选择该动作对系统的回报值越大（即系统收益越大）。但是如果 Q 值表中对应的状态动作较多，穷举 Q 值表来寻找最优解的难度将几何式上升，于是文中定义用θ来调整系统对状态-动作的 Q 值，用$Q(s,a,\theta)$来取代$Q(s,a)$，如下：

$$TargetQ = r + \gamma \max_{a'} Q(s', a'; \theta)$$

2.3 具体调度流程

具体调度流程分为两步，首先要理解权值矩阵算法，其是 WM-RDQN 算法中的一部分，调度流程为 WM-RDQN 算法流程。

2.3.1 权值矩阵算法

将状态向量S设置为当前待处理的任务信息和当前各处理节点、终端设备的预计处理完成时间，用如下式子表示：

$$S = [w_u, T_1, \cdots, T_u, T_c, T_m, \mathrm{A_u}]$$

其中包含任务信息与各服务器终端运行状态以及用户的优先级权值，用户设备权值矩阵A_u主要为状态向量提供动态的用户终端优先级参数，平衡各用户终端得到的服务质量，如下：

$$\mathrm{A_u} = \begin{vmatrix} \overline{E}_1 & \overline{dT}_1 & q_1 \\ \overline{E}_2 & \overline{dT}_2 & q_2 \\ & \cdots & \\ \overline{E}_u & \overline{dT}_u & q_u \end{vmatrix}$$

其中的平均时延标准与平均能耗标准是根据用户终端的实际情况由$z-score$公式计算得到的，$z-score$公式如下：

$$\begin{cases} \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \\ \sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \\ y_i = \frac{x_i - \bar{x}}{\sigma} \end{cases}$$

首先初始化权值矩阵A^0，输入用户端所需要的服务质量保证q_u。

接着产生任务卸载决策X_t，根据X_t进行任务卸载处理，得到用户的时延、能耗，分别记录进时延集合与能耗集合。

然后使用$z-score$公式对时延集合与能耗集合进行处理，将处理结果更新至A_u^t（如果产生的X_t为空集，则不对A_u^t进行更新）。

2.3.2 WM-RDQN 算法

在具体执行流程之前先明确两个概念：动作空间和奖励函数。

动作空间需要包括所有可能的任务卸载决策，0 表示跳过请求（空操作），1 表示任务卸载在本地，2 表示任务卸载在边缘服务器，3 表示任务卸载在云服务器。

$$a \in \{0,1,2,3\}$$

奖励函数计算的是某状态下的某动作对于系统收益的结果，值越大说明系统收益越好具体计算方式如下：

$$R_u^t = \begin{cases} 1 - F(u,t,A), & a \neq 0 \\ 0, & a = 0 \\ -1, & dT_u^t > maxT_u \end{cases}$$

下面是具体的调度流程：

首先初始化权值矩阵A^0、经验池以及时间步 t，输入用户端所需要的服务质量保证q_u。

接着初始化状态序列S_1，其包含当前待处理的任务信息，同时初始化预处理序列，其包含当前各处理节点、终端设备的预计处理完成时间，进入下面的迭代循环。

根据任务信息，在状态对应下的多个动作当中进行动作选择，以ε的概率选择最优奖励动作（Q 值最大的动作），以$(1-\varepsilon)$的概率在所有动作当中随机选择动作，这一步骤过后，选择的动作为a_t并向系统执行动作a_t。

执行完毕之后记录下系统和用户的状态，根据用户状态更新权值矩阵A_u^t，再根据权值矩阵A_u^t和系统状态来计算奖励值并返回奖励值r_t和S_{t+1}。

之后令$S_t = S_{t+1}$，以此计算得到$\varphi(t+1) = \varphi(S_{t+1})$，得到向量$(\varphi(t), a_t, r_t, \varphi(t+1))$并将向量存入经验池。

在经验池中按优先经验回放策略抽取向量，更新网络参数θ。

若S_{t+1}为结束状态，则结束本次迭代，否则继续进行迭代，直到S_{t+1}为结束状态，得到最优的状态-动作对应的 Q 值表。

2.4 学术界调度方法总结

学术界的调度方法大多具有以下特点：

1. 切入点相比于工业界来说较小，解决一些具体的问题，比如只考虑时延与成本或者只考虑服务质量，具有局限性，但不能否认这些具体的问题得以解决且效果较工业界的收益更高。
2. 适用的环境相对理想化，对于算法的效果评估绝大多数都是采用实验模拟仿真的方式进行，比如在一台或者多台电脑上搭建框架并进行模拟仿真，环境的参数相对固定，没有体现出实际应用场景的随机性。但不可否认的是在一些特定的场景下，一些特定的算法的收益要比工业界好，甚至远超实际应用的一些调度方法。
3. 调度方法中的算法相对复杂，代码量相较于工业界实际应用的调度方法的相关部分会大很多，流程相对复杂，如果一步出错，极有可能会出现连锁反应，容灾能力相对较差。

其可以值得借鉴学习的地方：

1. 对于本文提到的深度 Q 学习算法而言，其根据先前动作的“调度经验”指导后面的调度动作思路值得借鉴，就好像在自学一门课程与老师指导学习一门课程的学习效率是不一样的，有好的老师指导会学习得更快。这样可以减少对整个状态-动作表的直接遍历，有效减少任务调度的决策时延。除此之外还可以将其应用于寻找任务处理服务器的流程，通过迭代寻找任务的次数来学习其过程并有效减少对于主机的遍历，这是一种思路。中心思想是如何利用好其能通过迭代学习来大幅减少对目标进行遍历的次数。
2. 对于学术界的一些特定算法的算法而言，可以考虑在特定的情景下使用特定的算法，比如在任务数量远超平时的时候可以将调度算法切换成面向大规模任务群的算法，这样能有效利用其特定算法在特定场景下的系统高收益的特点。但是要保证其架构中的模块功能支持这种算法与算法之间的切换，并且要保证切换之后整个系统的容灾能力，这是思路之一。

3 总结

工业界要求调度方法简化有效，在能够有效解决问题的条件下还要保证系统其他的功能，比如容灾能力，新增模块的适应能力，同时工业界调度方法的逻辑简单，步骤明确，所以理解起来相对容易。学术界更加追求理想化的解决一个问题，这也导致只能从一个相对较小的切入点出发对问题进行分析求解，其方法对实时变化的环境的适应能力不如工业界，但不可否认其方法在特定的一些环境下（甚至是极端环境），有着比工业界更好的解决问题的效益。也正是因为工业界与学术界对待这个问题的看待方式不同，导致两者对现在以及未来的调度方法和调度系统的学习借鉴意义是不一样的。

工业界的调度方法值得借鉴的地方在于其筛选主机的机制，可自定义过滤器与称重函数，在实现自身功能的条件下还能保证系统整体的容灾能力，同时在添加优化算法时要注意的是要基于实际应用效果，即算法的加入是否能达到优化调度方法以及调度系统的效果。

学术界的调度方法值得借鉴的地方在于其筛选主机的逻辑，将这些逻辑加入到工业界的调度系统中也许能够提高工作效率，提升用户服务质量，同时一些特定的调度方法在特定的场景下有着比工业界更好地系统收益，能更好的对任务以及主机进行调度，可以考虑在特定的场景下使用特定的调度方法以保证调度任务的高效进行。

如何将学术界以及工业界的调度方法进行融合以达到优化调度方法和调度系统的目的以及如何减少对主机的大幅度遍历是未来可以考虑的研究方向。

参考文献

[1] Nova OpenDev: Free Software Needs Free Tools, Compute schedulers https://opendev.org/openstack/nova/src/commit/ad7249b3fcbda8ec179956f9784bceed1a633b5e/doc/source/admin/scheduling.rst#allhostsfilter

[2] 熊孩子会撒野(CSDN ID): 深挖 Openstack Nova - Scheduler 调度策略（1） https://blog.csdn.net/u011692924/article/details/80578686

[3] 赵佳君. 面向工业智能的移动边缘计算任务调度研究[D].电子科技大学,2022.DOI:10.27005/d.cnki.gdzku.2022.000480.

[4] 李成严,孙巍,唐立民.一种权重自适应的强化学习云资源调度算法[J].哈尔滨理工大学学报,2021,26(02):17-25.DOI:10.15938/j.jhust.2021.02.003.