中国科学院大学

CloudSim 云任务调度实验报告

小组成员：柳建国、卞景亮
所　　部：数字所
专　　业：电子信息

CloudSim 云任务调度实验报告

目录

一、CloudSim 介绍

1. CloudSim 体系结构[1]

CloudSim 云平台是由澳大利亚墨尔本大学的网格实验室和 Gridbus 项目宣布推出的云计算仿真软件，是一种开源模拟引擎，基于 GridSim 和离散事件驱动，可以模拟创建多种云计算环境中的实体，包括云数据中心、物理主机与虚拟机、各组件间的消息传输及时钟管理等。并且，CloudSim 作为通用的可扩展的模拟框架，支持模拟新兴的云计算基础设施和管理服务。图 1 是 CloudSim 体系结构。

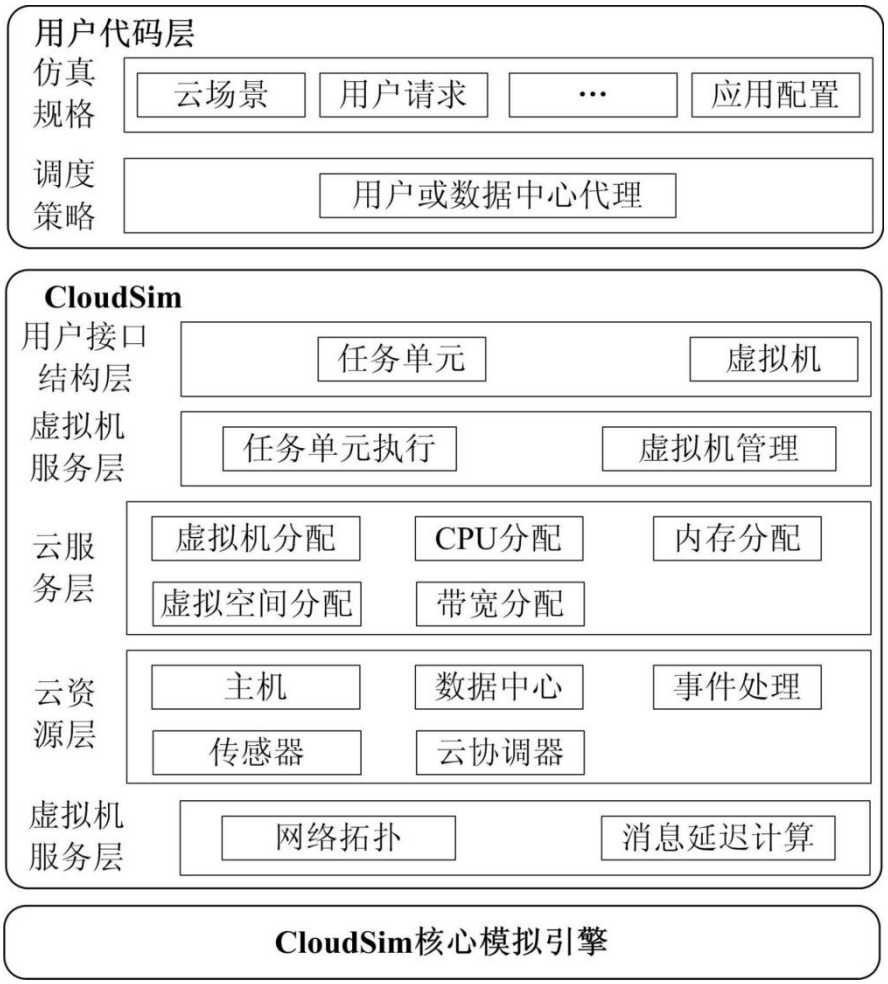

图 1 CloudSim 体系结构

2. CloudSim 工作方式[2]

CloudSim 的工作方式如图 2 所示，不同的用户提交的任务是相互独立的，数据中也可以有多个，代表来自不同云供应商的云资源，数据中也包含性能差异的物理机，通过虚拟化技术将物理资源抽象重组成不同性能的虚拟机。CloudSim 通过 VmAllocationPolicy 方法实现虚拟机和主机之间的映射策略。虚拟机之间的共享策略由 VmScheduler 来实现，包括时间共享策略和空间共享策略。用户自定义的任务和资源的映射方式需要在 DatacenterBroker 中

扩展实现。云信息服务中心（CIS）就是模拟调度的核心。

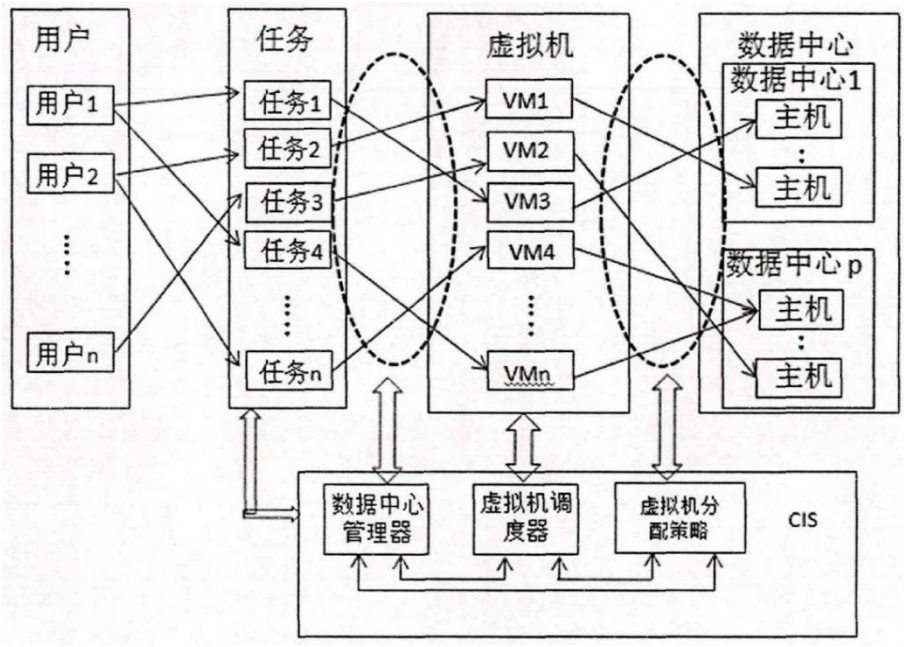

图 2 CloudSim 工作方式

由图 2 可以看出，在云计算环境下，从用户提交任务请求到获得相应的执行资源过程可以分为两个层次。

第一个层次是任务分配阶段，即用户提交的海量应用任务经过云数据中心代理均衡分配到集群中各虚拟机上执行的过程。该过程实现的是任务到虚拟机的映射，完成任务所需资源和虚拟机提供的虚拟资源之间的匹配。

第二个层次是虚拟机部署迁移阶段，即通过虚拟机合理部署和动态迁移实现对物理资源的高效利用。该阶段实现的是虚拟机到物理宿主机的映射，完成物理资源如何均衡的被虚拟机利用。

通过对 CloudSim 的工作方式和自带案例分析得知，任务调度在云平台仿真系统中的一般流程如下：

(1) 初始化 CloudSim 的环境，初始化用户数量、日历和标志；

CloudSim.init(num_user, calendar, trace_flag);

(2) 创建数据中心。数据中心由多个主机组成，一个主机代表一个或多个虚拟机，数据中心还可以设置不同的虚拟机调度策略；

Datacenter datacenter0 = createDatacenter("Datacenter_0");

(3) 创建数据中心代理；

创建数据中心代理。数据中心代理会管理用户提交的任务和数据中心的虚拟机，利用调度策略完成调度。本文编写的算法主要在此处进行调用，完成调度。

DatacenterBroker broker = createBroker();

(4) 创建虚拟机列表。对虚拟机参数进行设置；

Vmlist = createVM(brokerId,)

(5) 创建云任务列表；

Cloudlet cloudlet = new Cloudlet(id, lengths[i], pes Number, filesize, output Size, utilization Model, utilization Model, utilization Model);

(6) 根据自定义的调度策略，讲云任务绑定到虚拟机；

Broker.bindCloudletToVM(Cloudletlist, Vmlist);

(7) 开始仿真；

CloudSim.startSimulation();

(8) 结束仿真。

CloudSim.stopSimulation();

云计算的任务调度，实质上是任务对云计算中的资源节点进行竞争，因此，云计算没办法对每一个任务都提供最好的资源，所以该类型的任务调度是 NPC 问题，即在任务数量增加时，求解最佳解决方案所需要的时间呈指数级增长。

3. 云服务层共享[4]

尽管 VM 上下文是相互独立的（通常指主存和辅存空间），但仍会共享 CPU 内核和系统总线。因此，VM 的可用资源仍受主机处理能力限制。为了实现不同环境下对不同调度策略的模拟，目前的 CloudSim 支持两层 VM 调度：主机层和 VM 层。主机层中的 VM 调度直接指定 VM 可获取的处理能力，而 VM 层中，VM 为在其内运行的独立任务单元分配固定的处理能力。

两层 VM 调度均实现了时间共享和空间共享。以下分析两者在应用任务调度性能上的区别。如图 2，某主机可运行两个 VM，该主机拥有两个 CPU 内核，每个 VM 请求两个内核并执行四个任务，T_1、T_2、T_3、T_4、占用 VM1，而T_5、T_6、T_7、T_8占用 VM2。

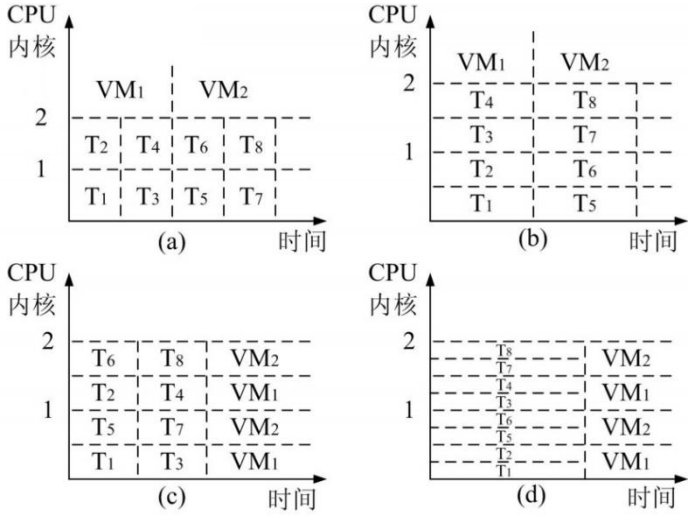

图 3 四种 VM 调度场景

图 3(a)中，VM 主机层和任务单元均采用空间共享。空间共享中，VM 请求两个 CPU 内核，对应时间内只能运行一个 VM。同样的原因，对于 VM1，其任务单元分配模式也是一样的，由于一个任务单元只需一个 CPU 内核，因此可以同时运行任务单元 T1 和 T2。此时，T3 和 T4 队列中等候。

图 3(b)中 VM 采用空间共享，任务单元采用时间共享。

图 3(c)中 VM 采用时间共享，任务单元采用空间共享。此时，内核通过时间片原理将其处理能力在 VM 中进行分配，而时间片本身以空间共享方式分配至任务。由于内核共享，VM 的可用处理能力是变化的。而任务单元是空间共享，表明内核只执行一个任务单元。

图 3(d)中 VM 和任务单元均采用时间共享。此时，VM 共享处理能力，并且同时将共享内核分配至所有任务单元。

4. 云计算任务调度[3]

云计算任务的资源调度通常有两种途径：一是物理机将自身计算资源按需分配给相应的计算任务；二是为达到负载均衡的目的，将计算任务迁移到资源利用率低的主机上继续运行。

云计算环境下的任务分配需要综合考虑的因素更多。这些因素主要包括以下几点：

(1) QoS 要求。满足 QoS 保障要求是对任务分配策略最基础的要求。任何一种分配策略如果达不到用户任务调度的 QoS 要求，那么它就无法投入使用，失去实际价值。一般而言，在用户进行任务调度时，会和云服务商签订相应的服务等级协议（SLA）。在该 SLA 中明确了任务的调度开支、截止时间、可靠性以及安全性等约束条件。实际云计算平台建设中，为给用户提供更加优质的服务从而获得更多的收益，云计算服务商必须充分考虑 QoS 目标约束要求。

(2) 任务完成时间。任务完成时间是衡量任务分配算法好坏的一个十分重要的性能指标。任务完成时间即从用户向云计算中心提交任务到所有任务执行完毕所用时间，在可以接受情况下，完成时间越短，则延时也就越短，用户体验就越好，证明算法执行效率越高，系统性能越好。

(3) 负载均衡水平。云计算作为一种商业计算模式，在满足用户任务调度所要求的 QoS 约束条件下，应充分考虑系统的负载均衡情况，以降低能耗，提高资源使用效率，以最低的成本获得最大的收益。

(4) 服务收益。由于云计算规模十分庞大，其用于海量的数据存储和超强的计算能力，在大数据时代，在制定任务分配策略时应该充分分析用户行为，建立相应的数学和经济学模型，更加智能和有针对性的进行任务分配，以获得更高经济利润。

5. 常见的云计算任务分配策略

云计算相关实现技术及软件多采用比较成熟和简单的任务分配策略，以 Hadoop 为例，其采用最多的还是经典的 FIFO 算法，此外，还有 Facebook 公司为满足并行执行多种任务的需要而提出的公平份额调度算法和 Yahooo 公司提出更够有效管理集群资源的计算能力调度算法等。

目前，研究人员基于 CloudSim 实现了多种方式的任务分配策略。文献[5]综合考虑云任务与虚拟机资源的特征，提出了一种改进的贪心策略。文献[6]通过使用优化后的蚁群算法实现了任务调度，实现了降低任务执行时间的同时提高虚拟机资源的负载。文献[7]把任务执行时间和系统负载均衡度作为优化目标，提出了一种基于混沌猫群算法实现多目标调度。文献[2]基于粒子群算法实现了任务调度。文献[8]将遗传算法和模拟退火算法结合，综合各自优点，实现了任务任务调度。文献[9]将蚁群算法和模拟退火算法相结合实现了任务调度。文献[1]基于蚁群算法，综合考虑 SLA 服务保证和负载均衡实现任务分配，引入了基于灰色预测模型的虚拟机迁移算法。文献[3]将遗传算法和蚁群算法结合，实现了任务调度。

二、CloudSim 任务调度实验原理介绍

1. 任务调度数学描述[8]

虚拟资源节点由 $VM = \{V_1, V_2, \ldots, V_j, \ldots V_m\}$表示，其中 m 表示虚拟资源节点的数量，$V_i$表示第 j 个可供选择的资源节点。

切割后的子任务用 $Task = \{T_1, T_2, \ldots, T_j, \ldots T_n\}$表示，其中 n 表示子任务的数量，$T_j$表示第 i 个需要执行的子任务。

$T_{run}(i,j)$表示任务 i 在虚拟资源节点 j 上执行完毕所需要的时间。$T_i(mips)$表示计算任务 i 所需要的指令数，$V_j(mips)$表示虚拟资源节点 j 的平均执行速度，那么$T_{run}(i,j)$可以表示为：

$$T_{run}(i,j) = \frac{T_i(mips)}{T_j(mips)}$$

$T_{tran}(i,j)$表示任务 i 传输到虚拟资源节点 j 所需时间。$T_i(filesize)$表示任务 i 的数据量大小，$V_j(bw)$表示虚拟资源节点 j 的带宽，那么$T_{tran}(i,j)$可以表示为：

$$T_{tran}(i,j) = \frac{T_i(filesize)}{V_j(bw)}$$

$T_{sum}(i,j)$表示由虚拟资源节点完成任务 i 所需要的全部时间，其计算方式可以表示为：

$$T_{sum}(i,j) = T_{run}(i,j) + T_{tran}(i,j)$$

由于云环境中虚拟资源节点是可以并行工作的，所以每个资源节点独立完成自己的工作，系统处理完所有子任务所需要的时间T_{cost}为：

$$T_{cost} = max\left(\sum_{i=1}^{m} T_{sum}(i,j)\right)$$

2. 循环调度算法

CloudSim 平台中默认实现了循环调度策略。循环调度策略是把一组任务顺序分配给一组虚拟机，尽量保证每个虚拟机运行相同数量的任务以平衡负载，但没有考虑任务的需求和虚拟机之间的差别。

3. Min-Min 和 Max-Min 调度策略[5]

Min-Min 云计算任务调度算法采用先易后难的策略。先执行完成时间短的任务，然后执行完成时间长的任务，并采取贪心策略把每个任务优先指派给执行它最早完成的计算资源。

Max-Min 云计算任务调度算法则恰恰相反，采用先难后易和贪心策略。每次选取完成时间最长的任务，再执行完成时间短的任务，并采取贪心策略把每个任务优先指派给执行它最早完成的计算资源。

4. 遗传调度算法

4.1 遗传算法执行流程见图 1。

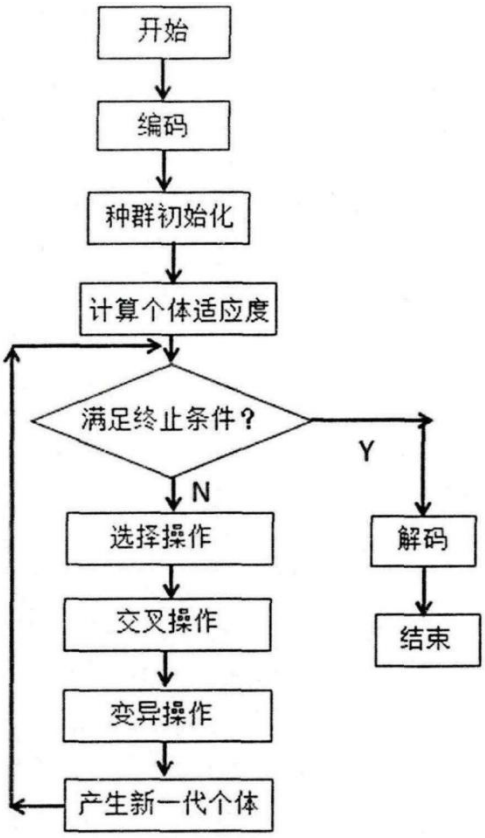

图 4 遗传算法执行流程图[3]

4.2 遗传算法求解任务调度的数学描述[3]

(1) 遗传编码

使用遗传算法求解问题的第一步是对所求问题进行编码，即将实际问题中的参数转换成染色体的形式。介于二进制编码存在长度过长，求解过程占用内存过多的缺点，实验采用实数编码的方式进行编码。根据前文中云计算中资源分配模型的设定，本文研究问题假设有 n 个子任务和 m 个虚拟资源节点，那么染色体的总长度即为子任务的个数 n，染色体的每一位取值范围为[0,m-1]。假设 n=20，m=5，则染色体总长度为 20，每一位取值为 0~4 之间的随机数，可假定一组编码实例为{1，3，1，2，0，2，4，4，2，1，3，0，3，2，0，4，2，1，4，4}对上述染色体解码为：

资源节点 0：{子任务 5，子任务 12，子任务 15}

资源节点 2：{子任务 1，子任务 3，子任务 10，子任务 14，子任务 17}

资源节点 3：{子任务 2，子任务 11，子任务 13}

资源节点 4：{子任务 7，子任务 8，子任务 16，子任务 19，子任务 20}

从上面的解码可以得到每个虚拟资源所执行的任务情况，通过获取每个子任务的完成时

间就可以通过前面公式求得所有任务的总完成时间。

(2) 初始化种群

解决了遗传算法的编码问题后，需要对所求问题进行种群的初始化。遗传算法中种群规模需要提前设定，它决定了每次迭代后种群中的个体数，初始种群一般是用完全随机的方式生。实验中设定种群规模为 scale=n×3，子任务个数为 n，虚拟资源数为 m。则种群初始化过程为：随机生成 scale 个个体，个体的染色体长度均为 n，染色体每一位的取值范围为[0,m-1]。

(3) 适应度函数

初代种群确定之后，算法会通过适应度函数来对所有染色体进行优劣评估。每一个个体都有自己对应的适应度值，由此可以确定个体的优劣性，从而为下一步的选择做准备。在本文中，适应度高的个体代表对任务的调度方案效率越高。适应度函数如果选取不当，会导致前期适应度高的个体被选择的概率增加，影响算法的全局搜索能力；后期还容易导致快速产生近似最优解。同时由于算法对适应度函数调用的频繁性，所以对其设计应尽量简单，实验中选用目标函数的倒数作为适应度函数：

$$\mathrm{fitness} = 1/\mathrm{T_{cost}}$$

遗传算法的核心就是通过模拟自然界中生物进化的方式达到搜索最优解的目的，交叉操作可以提高算法的全局搜索能力，变异操作决定了算法的局部搜索能力，选择操作则可以根据一定的规则，对经过交叉和变异后的新种群进行筛选，保留适应度高的个体组成新的种群，这样算法就可以通过迭代，实现对最优解的寻找。遗传算法对最优解的寻找是有方向的，并不是盲目的搜索，同时交叉和变异概率的取值也和算法的搜索能力密切相关。

(4) 选择操作

选择操作可以将子代中适应度高的解保留下来，为后面的交叉和遗传操作提供优秀解。选择操作通过计算每一个个体的适应度值，按照一定的规则将适应度低的个体淘汰，即“优胜劣汰”。保存下来的优值解又称为精英解，它们有进行下一步交叉和变异的权利，它们的子代适应度同样很高。本文主要通过轮盘赌的方式进行个体选择，即适应度高的个体被选择的概率大。

(5) 交叉操作

遗传算法全局搜索能力的实现主要靠交叉操作。交叉操作发生在两个染色体上，通过两个染色体之间的基因进行交叉和重组，实现产生新个体的目的。实验中采用的是实数编码的方式，基因的取值即代表虚拟资源的序号，所以考虑采用两点交叉的方式进行操作。

假设父代个体为 $\mathrm{x} = \{\mathrm{x_1}, \mathrm{x_2}, \mathrm{x_3}, \ldots, \mathrm{x_i}, \ldots, \mathrm{x_j}, \ldots, \mathrm{x_n}\}, \mathrm{x_i} \in \mathrm{R}, \mathrm{i} = 1,2, \ldots \mathrm{n}$ 和 $\mathrm{y} = \{\mathrm{y_1}, \mathrm{y_2}, \mathrm{y_3}, \ldots, \mathrm{y_i}, \ldots, \mathrm{y_j}, \ldots, \mathrm{y_n}\}, \mathrm{y_i} \in \mathrm{R}, \mathrm{i} = 1,2, \ldots \mathrm{n}$。两点交叉的位置为 i 和 j，则交叉后子代为：

$$\begin{cases} \textit{子代个体}\,1：\ \mathrm{x} = \{\mathrm{x_1}, \mathrm{x_2}, \mathrm{x_3}, \ldots, \mathrm{y_i}, \ldots, \mathrm{y_j}, \ldots, \mathrm{x_n}\} \\ \textit{子代个体}\,2：\ \mathrm{y} = \{\mathrm{y_1}, \mathrm{y_2}, \mathrm{y_3}, \ldots, \mathrm{x_i}, \ldots, \mathrm{x_j}, \ldots, \mathrm{y_n}\} \end{cases}$$

(6) 变异操作

变异操作通过对少量染色体进行改变，可以实现算法的局部搜索并加速最优解收敛。因为染色体的每一位都代表一个虚拟资源，所以进行过变异操作时应尽可能少的对基因进行修改。且进行修改时，基因的取值范围为[0,m-1]。

假设父代个体为：$\mathrm{x} = \{\mathrm{x_1}, \mathrm{x_2}, \mathrm{x_3}, \ldots, \mathrm{x_i}, \ldots, \mathrm{x_n}\}, \mathrm{x_i} \in \mathrm{R}, \mathrm{i} = 1,2, \ldots \mathrm{n}$，对$x_i$进行变异操作，所得子代为：$\mathrm{x} = \{\mathrm{x_1}, \mathrm{x_2}, \mathrm{x_3}, \ldots, \mathrm{x_j}, \ldots, \mathrm{x_n}\}, \mathrm{x_i} \in \mathrm{R}, \mathrm{i} = 1,2, \ldots \mathrm{n}$。

传统的遗传算法常使用固定值作为变异的概率，这样容易使算法陷入局部最优状，本次实验采用固定交叉概率 5%，最大变异步长为 n/5。

(7) 停止准则

由于交叉和变异的特性，遗传算法会不断产生新的个体，选择适当的循环结束准则可以

减少不必要的内存和时间开销。为算法设置迭代次数较简单的方式，但是也有可能会进行不必要的迭代。本次实验采用固定迭代次数结束迭代，迭代次数等于 n×10。

4.3 遗传算法的优缺点：

遗传算法的优点：

(1) 遗传算法具有良好的并行性。因为每个染色体代表一个解，在进行适应度计算时可以同时对染色体进行求解。这样可以节省算法的执行时间。

(2) 具有自组织、自适应和学习性。通过给算法设置合适的适应度函数，就可对种群提供正确的优化方向，经过交叉编译后的新种群也会在适应度函数的作用下不断朝着最优的方向进化，避免陷入局部最优。

(3) 通过交叉和变异操作，算法可以同时具有良好的全局搜索能力和局部搜索能力。

(4) 算法通过概率机制进行迭代，增加了算法的鲁棒性。

(5) 算法简单容易理解，且可扩展性高，易于其他算法融合。

(6) 应用范围广泛，在流行的机器学习和人工智能领域都有应用。

遗传算法的缺点：

(1) 算法的编码和解码操作是不可省略的，编码方式的好坏对求解结果有很大影响。

(2) 算法是在初始种群的基础上进行交叉和变异操作的，初始种群的选取会影响求解质量。

(3) 算法需要用到的参数较多，目前参数的选取主要靠经验，参数选取的好坏同样会影响求解质量。

(4) 算法设计的计算量较大，当问题规模达到一定程度时会增加求解时间。

(5) 初期如果出现超级染色体，会使得种群多样性减少，算法过早收敛，形成局部最优解。

三、CloudSim 实验参数配置

1. 虚拟机配置

在 CloudSim 平台中，首先配置数据中心相关参数，保证虚拟机有足够多的硬件资源可用。其次配置虚拟机相关参数，实验中设置了 5 个虚拟机，其中的关键参数（CPU 频率和带宽）如表 2 所示：

表 1 虚拟机参数信息

序号	MIPS	带宽
1	278	1000
2	289	1200
3	132	1100
4	209	1200
5	286	900

2. 云任务配置

在实验中，初始手工配置 40 个任务，任务的关键参数如表 2 所示。随后利用随机函数生成 100、150、200、300、400、500 个任务，任务指令长度分布在 1000~5000，任务文件大小分布在 10000~30000 之间，通过设置随机种子来让每种策略调度时使用相同的随机结果。

表 2 云任务相关参数

任务数量	任务指令长度	任务文件长度
40	19365, 49809, 30218, 44157, 16754, 26785, 12348, 28894, 33889, 58967, 35045, 12236, 20085, 31123, 32227, 41727, 51017, 44787, 65854, 39836, 18336, 20047, 31493, 30727, 31017, 30218, 44157, 16754, 26785, 12348, 49809, 30218, 44157, 16754, 26785, 44157, 16754, 26785, 12348, 28894	30000, 50000, 10000, 40000, 20000, 41000, 27000, 43000, 36000, 33000, 23000, 22000, 41000, 42000, 24000, 23000, 36000, 42000, 46000, 33000, 23000, 22000, 41000, 42000, 50000, 10000, 40000, 20000, 41000, 10000, 40000, 20000, 41000, 27000, 30000, 50000, 10000, 40000, 20000, 17000
100/150/200/300/400/500	任务指令长度分布在 1000~5000	任务文件大小分布在 10000~30000 之间

四、运行结果及分析

1. 运行结果

通过运行代码，得到在不同的任务情况下各种策略的分配结果，其中，任务的最迟完成时间如表 3 所示，对应的图见图 5 所示；不同策略下虚拟机的负载均衡度（5 个虚拟机运行时间的标准差）的如表 4 所示，对应的图见图 6 所示。

表 3 任务最迟完成时间

策略 任务数	40	100	150	200	300	400	500
循环	1842.282	441.142	646.637	868.608	1330.327	1822.426	2326.543
Man-Min	1150.911	340.508	513.894	680.822	1042.729	1358.032	1689.661
Min-min	1139.437	246.314	376.064	488.641	744.306	1003.848	1259.356
GA	1191.208	289.124	514.796	609.695	847.348	1170.343	1597.705

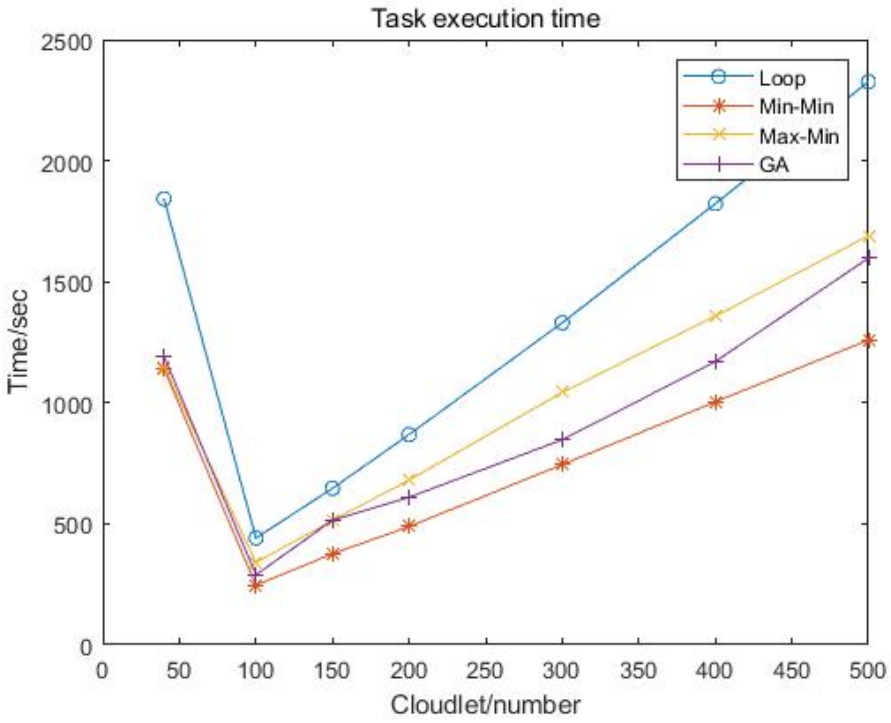

图 5 不同的任务数量在不同的算法调度情况下的云任务最迟完成时间

表 4 虚拟机负载均衡度

策略 任务数	40	100	150	200	300	400	500
循环	15.146	7.188	8.965	10.274	12.706	14.794	16.594
Min-Min	14.569	7.039	8.822	10.082	12.480	14.465	16.178
Max-min	14.434	6.871	8.583	9.796	12.152	14.136	15.826
GA	14.534	6.969	8.811	10.024	12.295	14.307	16.135

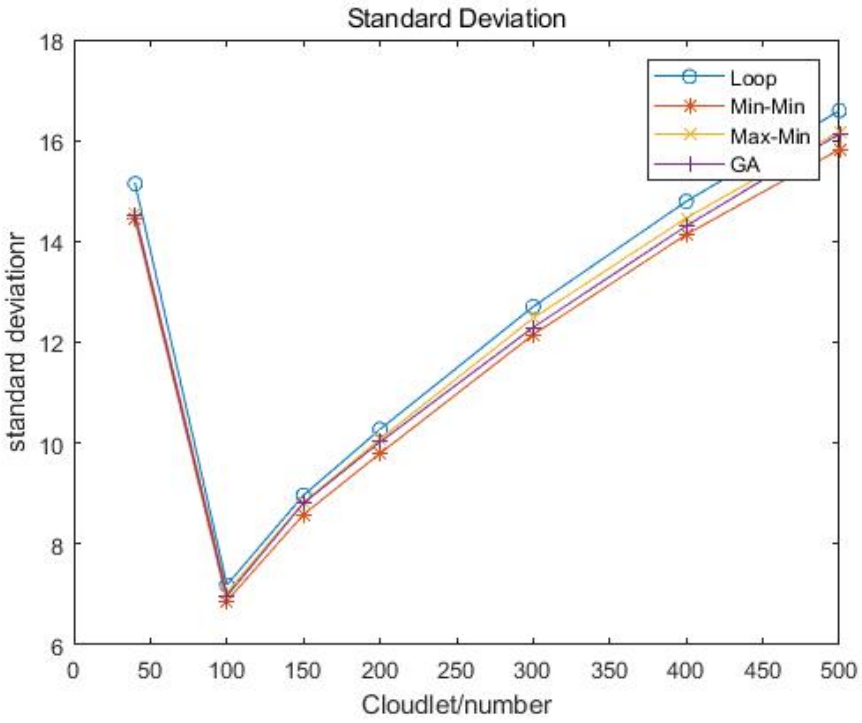

图 6 不同数量的任务在不同的算法调度下的虚拟机负载均衡度

2. 总结分析

通过实验结果可以看到使用遗传算法的任务调度执行时间在 Min-min 和 Max-min 之间，相对于 CloudSim 默认的循环调度算法有了很大的提升，但 Min-min 和 Max-min 不可避免地存在饥饿现象，而遗传算法可以很好地保证各个任务的公平性。

五、参考文献

[1] 于彦波. 基于 CloudSim 平台的云资源调度策略研究[D/OL]. 华北电力大学(北京), 2017. https://kns.cnki.net/KCMS/detail/detail.aspx?dbcode=CMFD&dbname=CMFD201801&filename=1017222046.nh&v=.

[2] 张家铭. 基于粒子群算法的云计算资源调度优化研究[D/OL]. 华北电力大学(北京), 2019. https://kns.cnki.net/KCMS/detail/detail.aspx?dbcode=CMFD&dbname=CMFD202001&filename=1019237444.nh&v=.

[3] 张本志. 云计算中基于 CloudSim 的任务调度研究[D/OL]. 东北财经大学, 2019. https://kns.cnki.net/KCMS/detail/detail.aspx?dbcode=CMFD&dbname=CMFD202002&filename=1020033677.nh&v=.

[4] 罗肖辉, 徐美霞. 基于 CloudSim 云平台的任务调度与能耗分析研究[J]. 计算机与数字工程, 2019, 47(12): 3228-3234.

[5] 王鑫, 王人福, 蒋华. 改进贪婪算法的云任务调度研究[J/OL]. 微电子学与计算机, 2018, 35(2): 109-112+117. https://doi.org/10.19304/j.cnki.issn1000-7180.2018.02.023.

[6] 张焕青, 张学平, 王海涛, 等. 基于负载均衡蚁群优化算法的云计算任务调度[J/OL]. 微

电子学与计算机, 2015, 32(5): 31-35+40. https://doi.org/10.19304/j.cnki.issn1000-7180.2015.05.007.
[7] 黄伟建, 辛风俊, 黄远. 基于混沌猫群算法的云计算多目标任务调度[J/OL]. 微电子学与计算机, 2019, 36(6): 55-59. https://doi.org/10.19304/j.cnki.issn1000-7180.2019.06.012.
[8] 黄璐. 基于遗传算法的云计算任务调度算法研究[D/OL]. 厦门大学, 2014. https://kns.cnki.net/KCMS/detail/detail.aspx?dbcode=CMFD&dbname=CMFD201402&filename=1014223219.nh&v=.
[9] 张浩荣, 陈平华, 熊建斌. 基于蚁群模拟退火算法的云环境任务调度[J]. 广东工业大学学报, 2014, 31(03): 77-82.

六、附录

1. 代码 GitHub 地址

https://github.com/JianguoLiu1996/CloudSim_Scheduling.git

2. 循环调度策略

```
package myTest;

/*
 * Title:        CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation
 *               of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
 *
 * Copyright (c) 2009, The University of Melbourne, Australia
 */

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.util.Collections;
import java.util.Comparator;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
```

```
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.lists.VmList;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;

/**
 * 使用 DatacenterBroker 中的默认调度算法执行，即使用“循环调度算法”执行任务调度，
 * 每个任务根据顺序，轮巡分配给每一个虚拟机。
 */
public class MyAllocationFIFO {
	/** The cloudlet list. */
	private static List<Cloudlet> cloudletList;

	/** The vmlist. */
	private static List<Vm> vmlist;

	// 设置类中全局变量，云任务数量和虚拟机数量，方便后面使用
	private static int cloudletNum = 40;//云任务数量
	private static int vmNum = 5;//虚拟机数量

	/**
	 * Creates main() to run this example.
	 *
	 * @param args the args
	 */
	@SuppressWarnings("unused")
	public static void main(String[] args) {

		Log.printLine("Starting CloudSimExample1...");

		try {
			// First step: Initialize the CloudSim package. It should be called
```

```
// before creating any entities.
// 第一步，即，初始化。
int num_user = 1; // number of cloud users
Calendar calendar = Calendar.getInstance();//日历
boolean trace_flag = false; // mean trace events

// Initialize the CloudSim library
// 初始化 CloudSim 库。
CloudSim.init(num_user, calendar, trace_flag);


// Second step: Create Datacenters
// Datacenters are the resource providers in CloudSim. We need at
// list one of them to run a CloudSim simulation
// 第二步，创建数据中心
Datacenter datacenter0 = createDatacenter("Datacenter_0");


// Third step: Create Broker
// 第三步，创建代理
DatacenterBroker broker = createBroker();
int brokerId = broker.getId();


// Fourth step: Create five virtual machine
// 第四步，创建 5 个虚拟机
vmlist = new ArrayList<Vm>();

// VM description（虚拟机参数设置）
int vmid = 0;
int[] mips = new int[] {278, 289, 132, 209, 286};//虚拟机 CPU 频率
long size = 10000; // image size (MB)
int ram = 2048; // vm memory (MB)
long[] bw = new long[] {1000, 1200, 1100, 1300, 900};//虚拟机带宽
int pesNumber = 1; // number of cpus
String vmm = "Xen"; // VMM name

// create VM
// add the VM to the vmList
for(int i=0; i<vmNum; i++) {
    vmlist.add(new Vm(vmid, brokerId, mips[i], pesNumber, ram, bw[i], size,
vmm, new CloudletSchedulerTimeShared()));
    vmid++;
}
```

```
            // submit vm list to the broker(将虚拟机列表提交给代理商)
            broker.submitVmList(vmlist);


            // Fifth step: Create one Cloudlet
            // 第五步，创建 40 个任务
            cloudletList = new ArrayList<Cloudlet>();

            // Cloudlet properties（任务列表）
            int id = 0;
            long[] length = new long[] {
                    19365, 49809, 30218, 44157, 16754, 26785,12348, 28894, 33889,
58967,
                    35045, 12236, 20085, 31123, 32227, 41727, 51017, 44787, 65854,
39836,
                    18336, 20047, 31493, 30727, 31017, 30218, 44157, 16754, 26785,
12348,
                    49809, 30218, 44157, 16754, 26785, 44157, 16754, 26785, 12348,
28894};//云任务指令数
            long[] fileSize = new long[] {
                    30000, 50000, 10000, 40000, 20000, 41000, 27000, 43000, 36000,
33000,
                    23000, 22000, 41000, 42000, 24000, 23000, 36000, 42000, 46000,
33000,
                    23000, 22000, 41000, 42000, 50000, 10000, 40000, 20000, 41000,
10000,
                    40000, 20000, 41000, 27000, 30000, 50000, 10000, 40000, 20000,
17000};//云任务文件大小

//          // 使用随机的方法生成指令长度和文件数据长度。
//          long[] length = new long[cloudletNum];
//          long[] fileSize = new long[cloudletNum];
//          Random random = new Random();
//          random.setSeed(10000L);//设置种子，让每次运行产生的随机数相同
//          for(int i = 0 ; i < cloudletNum ; i ++) {
//              length[i] = random.nextInt(4000) + 1000;
//          }
//          random.setSeed(5000L);//设置种子，让每次运行产生的随机数相同
//          for(int i = 0 ; i < cloudletNum ; i ++) {
//              fileSize[i] = random.nextInt(20000) + 10000;
//          }
//
            long outputSize = 300;
```

```
                UtilizationModel utilizationModel = new UtilizationModelFull();

                // add the cloudlet to the list
                for(int i=0; i<cloudletNum; i++) {
                    Cloudlet cloudlet = new Cloudlet(id, length[i], pesNumber, fileSize[i],
outputSize, utilizationModel, utilizationModel, utilizationModel);
                    cloudlet.setUserId(brokerId);
                    cloudletList.add(cloudlet);
                    id++;
                }

                // submit cloudlet list to the broker.（将任务提交给代理商）
                broker.submitCloudletList(cloudletList);

                // 不用将虚拟机和任务进行绑定，使用 DatacenterBroker.java 中的默认绑定方
法。

                // Sixth step: Starts the simulation
                //第六步，开始仿真
                CloudSim.startSimulation();

                CloudSim.stopSimulation();


                //Final step: Print results when simulation is over
                //最后一步，输出仿真结果
                List<Cloudlet> newList = broker.getCloudletReceivedList();
                printCloudletList(newList);

                Log.printLine("CloudSimExample1 finished!");
            } catch (Exception e) {
                e.printStackTrace();
                Log.printLine("Unwanted errors happen");
            }
        }

        /**
         * Creates the datacenter.
         *
         * @param name the name
         *
         * @return the datacenter
         */
        private static Datacenter createDatacenter(String name) {
```

```
        // Here are the steps needed to create a PowerDatacenter:
        // 1. We need to create a list to store
        // our machine
        List<Host> hostList = new ArrayList<Host>();

        // 2. A Machine contains one or more PEs or CPUs/Cores.
        // In this example, it will have only one core.
        List<Pe> peList = new ArrayList<Pe>();

        int mips = 1000;

        // 4. Create Host with its id and list of PEs and add them to the list
        // of machines
        int hostId = 0;
        int ram = 2048; // host memory (MB)
        long storage = 1000000; // host storage
        int bw = 10000; //带宽

        for(int i=0; i<vmNum; i++) {
            // 3. Create PEs and add these into a list.
            // 为每个虚拟机创建一个 CPU，CPU 能力大于虚拟机能力。
            peList.add(new Pe(i, new PeProvisionerSimple(mips))); // need to store Pe id and
MIPS Rating

            hostList.add(
                    new Host(
                        hostId,
                        new RamProvisionerSimple(ram),
                        new BwProvisionerSimple(bw),
                        storage,
                        peList,
                        new VmSchedulerTimeShared(peList)
                    )
                ); // This is our machine
            hostId++;
        }

        // 5. Create a DatacenterCharacteristics object that stores the
        // properties of a data center: architecture, OS, list of
        // Machines, allocation policy: time- or space-shared, time zone
        // and its price (G$/Pe time unit).
        String arch = "x86"; // system architecture
        String os = "Linux"; // operating system
```

```
            String vmm = "Xen";
            double time_zone = 10.0; // time zone this resource located
            double cost = 3.0; // the cost of using processing in this resource
            double costPerMem = 0.05; // the cost of using memory in this resource
            double costPerStorage = 0.001; // the cost of using storage in this
                                                    // resource
            double costPerBw = 0.0; // the cost of using bw in this resource
            LinkedList<Storage> storageList = new LinkedList<Storage>(); // we are not adding SAN
                                                                    // devices by now

            DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
                    arch, os, vmm, hostList, time_zone, cost, costPerMem,
                    costPerStorage, costPerBw);

            // 6. Finally, we need to create a PowerDatacenter object.
            Datacenter datacenter = null;
            try {
                datacenter = new Datacenter(name, characteristics, new
VmAllocationPolicySimple(hostList), storageList, 0);
            } catch (Exception e) {
                e.printStackTrace();
            }

            return datacenter;
        }

        // We strongly encourage users to develop their own broker policies, to
        // submit vms and cloudlets according
        // to the specific rules of the simulated scenario
        /**
         * Creates the broker.
         *
         * @return the datacenter broker
         */
        private static DatacenterBroker createBroker() {
            DatacenterBroker broker = null;
            try {
                broker = new DatacenterBroker("Broker");
            } catch (Exception e) {
                e.printStackTrace();
                return null;
            }
            return broker;
        }
```

```
    /**
     * Prints the Cloudlet objects.
     *
     * @param list list of Cloudlets
     */
    private static void printCloudletList(List<Cloudlet> list) {
        int size = list.size();

        double[] executeTimeOfVM = new double[vmNum];//记录每个虚拟机VM的最后一个任务完成时间
        double meanOfExecuteTimeOfVM = 0;//虚拟机平均运行时间
        for(int i=0;i<vmNum;i++) {//初始化数组
            executeTimeOfVM[i] = 0;
        }
        double LB=0;//负载平衡因子

        Cloudlet cloudlet;

        String indent = "    ";
        Log.printLine();
        Log.printLine("========== OUTPUT ==========");
        Log.printLine("Cloudlet ID" + indent + "STATUS" + indent
                + "Data center ID" + indent + "VM ID" + indent + "Time" + indent
                + "Start Time" + indent + "Finish Time");

        DecimalFormat dft = new DecimalFormat("###.##");
        for (int i = 0; i < size; i++) {
            cloudlet = list.get(i);
            Log.print(indent + cloudlet.getCloudletId() + indent + indent);

            if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS) {
                Log.print("SUCCESS");

                Log.printLine(indent + indent + cloudlet.getResourceId()
                        + indent + indent + indent + cloudlet.getVmId()
                        + indent + indent
                        + dft.format(cloudlet.getActualCPUTime()) + indent
                        + indent + dft.format(cloudlet.getExecStartTime())
                        + indent + indent
                        + dft.format(cloudlet.getFinishTime()));

                //计算每个虚拟机最后完成的时间
                if(cloudlet.getFinishTime() > executeTimeOfVM[cloudlet.getVmId()]) {
```

```
                        executeTimeOfVM[cloudlet.getVmId()] = cloudlet.getFinishTime();
                    }
                }
            }

            //求所有虚拟机平均运行时间
            for(int i=0;i<vmNum;i++) {
                meanOfExecuteTimeOfVM += executeTimeOfVM[i];
                Log.printLine("VM" + i +" executeTime:" + executeTimeOfVM[i] + "\n");
            }
            meanOfExecuteTimeOfVM /= vmNum;
            Log.printLine("meanOfExecuteTimeOfVM:" + meanOfExecuteTimeOfVM + "\n");

            //计算负载平衡因子（即标准差）
            for(int i=0; i<vmNum; i++) {
                LB += Math.pow(executeTimeOfVM[i]-meanOfExecuteTimeOfVM, 2);
            }
            LB = Math.sqrt(meanOfExecuteTimeOfVM/vmNum);
            Log.printLine("LB:" + LB + "\n");
        }
    }
```

3. Min-Min 策略

```
    //Min-Min+贪心算法
    //Cloudlet 根据 MI 升序排列,MinMin 算法
    public static void bindCloudletsToVmsTimeAwared(){
        int cloudletNum=cloudletList.size();
        int vmNum=vmlist.size();

        //time[i][j]  表示任务 i 在虚拟机 j 上的执行时间
        double[][] time=new double[cloudletNum][vmNum];

        //cloudletList 按 MI 升序排列, vm 按 MIPS 升序排列
        Collections.sort(cloudletList,new MyAllocationMinMin().new CloudletComparator());
        Collections.sort(vmlist,new MyAllocationMinMin().new VmComparator());

        //For test (查看 cloudletList 和 vmlist 排序结果)
        System. out .println("///////////For test///////////////");
        for(int i=0;i<cloudletNum;i++){
            System. out .print(cloudletList.get(i).getCloudletId()+":"+cloudletList.get
```

```
                        (i).getCloudletLength()+" ");
            }
            System. out .println();
            for (int i=0;i< vmNum;i++){
                System. out .print(vmlist.get(i).getId()+":"+vmlist.get(i).getMips()+" ");
            }
            System. out .println();
            System. out .println("////////////////////////////////");

            //计算 time[i][j]，即：计算任务 i 在虚拟机 j 上执行时间
          for (int i=0;i<cloudletNum;i++){
              for(int j=0;j<vmNum;j++){
                                                                    time[i][j]=
(double)cloudletList.get(i).getCloudletLength()/vmlist.get(j).getMips()
                                                                    +
(double)cloudletList.get(i).getCloudletFileSize()/vmlist.get(j).getBw();
                  System. out .print("time["+i+"]["+j+"]="+time[i][j]);
              }

              //For test,输出 time[i][j]计算结果
              System. out .println();
          }

          double[] vmLoad=new double[vmNum];//在某个虚拟机上任务的总执行时间
          int[] vmTasks=new int[vmNum]; //在某个 Vm 上运行的任务数量
          double minLoad=0;//记录当前任务分配方式的最优值
          int idx=0;//记录当前任务最优分配方式对应的虚拟机列号

          //第一个 cloudlet（最短的任务）先分配给最快的 vm
          vmLoad[vmNum-1]=time[0][vmNum-1];
          vmTasks[vmNum-1]=1;
          cloudletList.get(0).setVmId(vmlist.get(vmNum-1).getId());

          //遍历任务，对于每个虚拟机，如果该任务加到这个虚拟机上时，总执行时间最短，
则将该任务分配给该虚拟机
          for(int i=1;i<cloudletNum;i++){
              minLoad=vmLoad[vmNum-1]+time[i][vmNum-1];
              idx=vmNum-1;

              //对于每个虚拟机，如果任务加到虚拟机上，所有的任务执行时间最短，则将
该任务分配给该虚拟机
              for(int j=vmNum-2;j>=0;j--){
                  //如果当前虚拟机未分配任务,则比较完当前任务分配给该虚拟机是否最
优。
```

```
                    if(vmLoad[j]==0){
                        if(minLoad>=time[i][j]) {
                            idx = j;
                        }
                        break;
                    }
                    //如果将该任务分配到该虚拟机上，总执行时间最短，则将该任务分配该该虚拟机。
                    if(minLoad>vmLoad[j]+time[i][j]){
                        minLoad=vmLoad[j]+time[i][j];
                        idx=j;
                    }
                    //如果分配到该虚拟机，总的执行时间相同，则将任务分配给任务数量较少的虚拟机。
                    else if(minLoad==vmLoad[j]+time[i][j]&&vmTasks[j]<vmTasks[idx]) {
                        idx = j;
                    }
                }
                vmLoad[idx]+=time[i][idx];
                vmTasks[idx]++;
                cloudletList.get(i).setVmId(vmlist.get(idx).getId());
                                                    System.          out          .print(i+"th
"+"vmLoad["+idx+"]="+vmLoad[idx]+"minLoad="+minLoad);
                System. out .println();
            }
        }

        //Cloudlet 根据 length 升序排列
        private class CloudletComparator implements Comparator<Cloudlet>{
            @Override
            public int compare(Cloudlet cl1, Cloudlet cl2){
                return (int)(cl1.getCloudletLength()-cl2.getCloudletLength());
            }
        }

        //Vm 根据 MIPS 升序排列
            private class VmComparator implements Comparator<Vm>{
            @Override
            public int compare(Vm vm1, Vm vm2){
                return (int)(vm1.getMips()-vm2.getMips());
            }
        }
```

4. Max-Min 策略

```
//Max-Min+贪心算法
    //Cloudlet 根据 MI 降序排列,MaxMin 算法
    public static void bindCloudletsToVmsTimeAwared(){
        int cloudletNum=cloudletList.size();
        int vmNum=vmlist.size();

        //time[i][j]  表示任务 i 在虚拟机 j 上的执行时间
        double[][] time=new double[cloudletNum][vmNum];

        //第一步，cloudletList 按 MI 降序排列, vm 按 MIPS 升序排列
        Collections.sort(cloudletList,new MyAllocationMaxMin().new CloudletComparator());
        Collections.sort(vmlist,new MyAllocationMaxMin().new VmComparator());

        //For test (查看 cloudletList 和 vmlist 排序结果)
        System. out .println("////////////For test///////////////");
        for(int i=0;i<cloudletNum;i++){
            System. out .print(cloudletList.get(i).getCloudletId()+":"+cloudletList.get
                    (i).getCloudletLength()+" ");
        }
        System. out .println();
        for (int i=0;i< vmNum;i++){
            System. out .print(vmlist.get(i).getId()+":"+vmlist.get(i).getMips()+" ");
        }
        System. out .println();
        System. out .println("////////////////////////////////");

        //第二步，计算 time[i][j]，即：计算任务 i 在虚拟机 j 上执行时间
       for (int i=0;i<cloudletNum;i++){
            for(int j=0;j<vmNum;j++){
                                                                    time[i][j]=
(double)cloudletList.get(i).getCloudletLength()/vmlist.get(j).getMips();
                System. out .print("time["+i+"]["+j+"]="+time[i][j]);
            }

            //For test,输出 time[i][j]计算结果
            System. out .println();
       }

       double[] vmLoad=new double[vmNum];//在某个虚拟机上任务的总执行时间
       int[] vmTasks=new int[vmNum]; //在某个 Vm 上运行的任务数量
       double minLoad=0;//记录当前任务分配方式的最优值
```

```
        int idx=0;//记录当前任务最优分配方式对应的虚拟机列号

        //第三步，第一个 cloudlet（最长的任务）优先分配给最快的 vm
        vmLoad[vmNum-1]=time[0][vmNum-1];
        vmTasks[vmNum-1]=1;
        cloudletList.get(0).setVmId(vmlist.get(vmNum-1).getId());

        //从长任务到短任务，对于每个虚拟机，如果该任务加到这个虚拟机上时，总执行时间最短，则将该任务分配给该虚拟机
        for(int i=1;i<cloudletNum;i++){
            minLoad=vmLoad[vmNum-1]+time[i][vmNum-1];
            idx=vmNum-1;

            //对于每个虚拟机，如果任务加到虚拟机上，所有的任务执行时间最短，则将该任务分配给该虚拟机
            for(int j=vmNum-2;j>=0;j--){
                //如果当前虚拟机未分配任务,则比较完当前任务分配给该虚拟机是否最优
                if(vmLoad[j]==0){
                    if(minLoad>=time[i][j]) {
                        idx = j;
                    }
                    break;
                }
                if(minLoad>vmLoad[j]+time[i][j]){
                    minLoad=vmLoad[j]+time[i][j];
                    idx=j;
                }
                //简单的负载均衡
                else if(minLoad==vmLoad[j]+time[i][j]&&vmTasks[j]<vmTasks[idx]) {
                    idx = j;
                }
            }
            vmLoad[idx]+=time[i][idx];
            vmTasks[idx]++;
            cloudletList.get(i).setVmId(vmlist.get(idx).getId());
                                                    System.        out        .print(i+"th
"+"vmLoad["+idx+"]="+vmLoad[idx]+"minLoad="+minLoad);
            System. out .println();
        }
    }

    //Cloudlet 根据 length 降序排列
    private class CloudletComparator implements Comparator<Cloudlet>{
        @Override
```

```
        public int compare(Cloudlet cl1, Cloudlet cl2){
            return (int)(cl2.getCloudletLength()-cl1.getCloudletLength());
        }
    }

    //Vm 根据 MIPS 升序排列
        private class VmComparator implements Comparator<Vm>{
        @Override
        public int compare(Vm vm1, Vm vm2){
            return (int)(vm1.getMips()-vm2.getMips());
        }
    }
```

5. 遗传算法

5.1 MyAllocationGA.java

```
// GA 操作，使用遗传算法计算结果进行任务分配。
                GA2 test = new GA2(cloudletList,vmlist);
                int[] resultGene = test.caculte();
                for(int i=0; i<cloudletList.size(); i++) {
                    cloudletList.get(i).setVmId(vmlist.get(resultGene[i]).getId());
                }
```

5.2 GA2.java

```
package myTest;

import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.Vm;

/**
*@Description:
* 使用遗传算法计算最优的任务分配策略，
* 然后根据遗传算法结果，进行实际任务分配。
*/

public class GA2 extends GeneticAlgorithm{
```

```
    private static List<Cloudlet> cloudletList;//云任务列表
    private static List<Vm> vmList;//虚拟机列表

    public GA2(List<Cloudlet> cL, List<Vm>vL) {
        super(cL.size()); //调用父方法，设置基因长度。

        //获取任务列表和虚拟机列表
        GA2.cloudletList = cL;
        GA2.vmList = vL;
    }

    // 计算染色体得分。
    // 输入一个染色体
    // 返回该染色体译码得到的分配策略，在分配策略下，任务的最迟完成时间。
    public double changeX(Chromosome chro) {
        int[] gene = chro.getNum();
        double x_completeTime;//储存染色体最迟完成时间。
        double tempx[] = new double[] {0.0, 0.0, 0.0, 0.0, 0.0};
        for(int i=0; i<gene.length; i++) {
            switch(gene[i]) {
                case 0:
                    tempx[0] +=
(cloudletList.get(i).getCloudletFileSize()/vmList.get(gene[i]).getBw()) +
                    (cloudletList.get(i).getCloudletLength()/vmList.get(gene[i]).getMips());
                    break;
                case 1:
                    tempx[1] +=
(cloudletList.get(i).getCloudletFileSize()/vmList.get(gene[i]).getBw()) +
                    (cloudletList.get(i).getCloudletLength()/vmList.get(gene[i]).getMips());
                    break;
                case 2:
                    tempx[2] +=
(cloudletList.get(i).getCloudletFileSize()/vmList.get(gene[i]).getBw()) +
                    (cloudletList.get(i).getCloudletLength()/vmList.get(gene[i]).getMips());
                    break;
                case 3:
                    tempx[3] +=
(cloudletList.get(i).getCloudletFileSize()/vmList.get(gene[i]).getBw()) +
                    (cloudletList.get(i).getCloudletLength()/vmList.get(gene[i]).getMips());
                    break;
                case 4:
                    tempx[4] +=
(cloudletList.get(i).getCloudletFileSize()/vmList.get(gene[i]).getBw()) +
                    (cloudletList.get(i).getCloudletLength()/vmList.get(gene[i]).getMips());
```

```
                        break;
                    default:break;
                }
            }
            x_completeTime = tempx[0];
            for(int i=1; i<5; i++) {
                if(x_completeTime < tempx[i]) {
                    x_completeTime = tempx[i];
                }
            }
            return (1/x_completeTime);//注意，返回的时任务最迟完成时间的倒数。
        }

        @Override
        public double caculateY(double x) {
            // TODO Auto-generated method stub
            return x;
        }
}
```

5.3 GeneticAlgorithm.java

```
package myTest;

/**
*@Description:
* 遗传算法具体实现
* 该类为虚拟类，需要进行继承该类，并重写类中的方法，实现遗传算法。
*/

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;



public abstract class GeneticAlgorithm {
    private List<Chromosome> population = new ArrayList<Chromosome>();
    private int popSize = 40*3;//种群数量，需手动设置
    private int geneSize;//基因最大长度
    private int maxIterNum = 40*10;//最大迭代次数，需手动设置
    private double mutationRate = 0.05;//基因变异的概率，需手动设置
    private int maxMutationNum = 40/5;//最大变异步长，需手动设置
```

```
    private int generation = 1;//当前遗传到第几代

    private double bestScore;//最好得分
    private double worstScore;//最坏得分
    private double totalScore;//总得分
    private double averageScore;//平均得分

    //private double x = Double.POSITIVE_INFINITY; //记录历史种群中最好的 X 值，基因序列最好的适应值，初始化为无穷大。
    private double x; //记录历史种群中最好的 X 值，基因序列最好的适应值。
    private static int[] bestGenex = new int[1000];//记录历史最好的基因序列
    //private double y = Double.POSITIVE_INFINITY; //记录历史种群中最好的Y值，在CloudSim实现中 y==x，初始化为无穷大。
    private double y;//记录历史种群中最好的 Y 值.
    private int geneI;//x y 所在代数

    // 设置基因长度
    public GeneticAlgorithm(int geneSize) {
        this.geneSize = geneSize;
    }

    // 主方法，使用遗传算法计算最好基因，并返回最好基因。
    public int[] caculte() {
        // 初始化种群
        generation = 1;
        init();

        // 当种群代数不满足要求时，迭代执行进化步骤。
        while (generation < maxIterNum) {
            //种群遗传
            evolve();
            print();
            generation++;
        }

        //输出最好的基因序列结果和适应值
        System.out.print("The best Gene list is ");
        for(int i=0; i<geneSize; i++) {
            System.out.print(bestGenex[i] + "; ");
        }
        System.out.println("\nThe best socre is " + y);
        return bestGenex;
    }
```

```
    /**
     * @Description: 输出结果
     */
    private void print() {
        System.out.println("--------------------------------");
        System.out.println("the generation is:" + generation);
        System.out.println("the best y is:" + bestScore);
        System.out.println("the worst fitness is:" + worstScore);
        System.out.println("the average fitness is:" + averageScore);
        System.out.println("the total fitness is:" + totalScore);
        System.out.println("geneI:" + geneI + "\tx:" + x + "\ty:" + y);
    }


    /**
     * @Description:
     * 初始化种群
     */
    private void init() {
        // 随机生成初始化种群。
        for (int i = 0; i < popSize; i++) {
            //population = new ArrayList<Chromosome>();
            Chromosome chro = new Chromosome(geneSize);
            population.add(chro);
        }

        // 计算每个基因序列的得分，并计算种群的最好、最坏、平均得分。
        caculteScore();

        //给最好的基因序列赋初值
        //bestGenex = Arrays.copyOf(population.get(0).getNum(), geneSize);
    }

    /**
     * @Description:种群进行遗传
     */
    private void evolve() {
        List<Chromosome> childPopulation = new ArrayList<Chromosome>();
        //step1，生成下一代种群
        while (childPopulation.size() < popSize) {
            Chromosome p1 = getParentChromosome();//轮盘赌法选择可以遗传下一代的
染色体。
            Chromosome p2 = getParentChromosome();//轮盘赌法选择可以遗传下一代的
```

```
染色体。
                List<Chromosome> children = Chromosome.genetic(p1, p2);//对父代基因进行交叉变换。
                if (children != null) {
                    for (Chromosome chro : children) {
                        childPopulation.add(chro);
                    }
                }
            }
            //新种群替换旧种群
            List<Chromosome> t = population;
            population = childPopulation;
            t.clear();
            t = null;

            //step2，基因突变
            mutation();

            //step3，计算新种群的适应度
            caculteScore();
        }

        /**
         * @Description: 轮盘赌法选择可以遗传下一代的染色体
         */
        private Chromosome getParentChromosome (){
            double slice = Math.random() * totalScore;
            double sum = 0;
            for (Chromosome chro : population) {
                sum += chro.getScore();
                if (sum > slice && chro.getScore() >= averageScore) {
                    return chro;
                }
            }
            return null;
        }

        /**
         * @Description: 计算种群适应度
         * 遍历种群中的每一个基因序列，计算基因序列得分，和计算种群中的最好、最坏、总得分、平均得分。
         */
        private void caculteScore() {
            setChromosomeScore(population.get(0));
```

```
            bestScore = population.get(0).getScore();
            worstScore = population.get(0).getScore();
            totalScore = 0;

            for (Chromosome chro : population) {
                setChromosomeScore(chro);//计算该基因序列的得分（适应度）。
                if (chro.getScore() > bestScore) { //设置最好基因值。
                    bestScore = chro.getScore();
                    if (y < bestScore) {
                        x = changeX(chro);
                        y = bestScore;

                        //获得最好的基因序列，并记录。
                        bestGenex = Arrays.copyOf(chro.getNum(), geneSize);
                        geneI = generation;
                    }
                }
                if (chro.getScore() < worstScore) { //设置最坏基因值
                    worstScore = chro.getScore();
                }
                totalScore += chro.getScore();
            }
            averageScore = totalScore / popSize;
            //因为精度问题导致的平均值大于最好值，将平均值设置成最好值
            averageScore = averageScore > bestScore ? bestScore : averageScore;
        }

        /**
         * 基因突变
         */
        private void mutation() {
            for (Chromosome chro : population) {
                if (Math.random() < mutationRate) { //发生基因突变
                    int mutationNum = (int) (Math.random() * maxMutationNum);//根据最大变异步长，随机生成最大变异步长，进行突变。
                    chro.mutation(mutationNum);
                }
            }
        }

        /**
         * @Description: 设置染色体得分
         */
        private void setChromosomeScore(Chromosome chro) {
```

```
        if (chro == null) {
            return;
        }
        double x_temp = changeX(chro);
        double y_temp = caculateY(x_temp);
        chro.setScore(y_temp);

    }

    /**
     * @Description: 将二进制转化为对应的 X
     */
    public abstract double changeX(Chromosome chro);


    /**
     * @Description: 根据 X 计算 Y 值  Y=F(X)
     */
    public abstract double caculateY(double x);

    public void setPopulation(List<Chromosome> population) {
        this.population = population;
    }

    public void setPopSize(int popSize) {
        this.popSize = popSize;
    }

    public void setGeneSize(int geneSize) {
        this.geneSize = geneSize;
    }

    public void setMaxIterNum(int maxIterNum) {
        this.maxIterNum = maxIterNum;
    }

    public void setMutationRate(double mutationRate) {
        this.mutationRate = mutationRate;
    }

    public void setMaxMutationNum(int maxMutationNum) {
        this.maxMutationNum = maxMutationNum;
    }
```

```
	public double getBestScore() {
		return bestScore;
	}

	public double getWorstScore() {
		return worstScore;
	}

	public double getTotalScore() {
		return totalScore;
	}

	public double getAverageScore() {
		return averageScore;
	}

	public double getX() {
		return x;
	}

	public double getY() {
		return y;
	}
}
```

5.4 Chromosome.java

```
package myTest;

/**
*@Description: 基因遗传染色体
*/

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import org.cloudbus.cloudsim.Cloudlet;

public class Chromosome {
	private int[] gene;//基因序列
```

```
private double score;//对应的函数得分

public double getScore() {
	return score;
}

public void setScore(double score) {
	this.score = score;
}

/**
 * @param size
 * 随机生成基因序列
 */
public Chromosome(int size) {
	if (size <= 0) {
		return;
	}
	initGeneSize(size);
	Random random = new Random();
	for (int i = 0; i < size; i++) {

		//gene[i] = (int) (Math.random()*10) ;
		gene[i] = random.nextInt(5);
	}
}

/**
 * 生成一个新基因（默认方法）
 */
public Chromosome() {

}

/**
 * @param c
 * @Description: 克隆基因
 */
public static Chromosome clone(final Chromosome c) {
	if (c == null || c.gene == null) {
		return null;
	}
	Chromosome copy = new Chromosome();
	copy.initGeneSize(c.gene.length);
```

```
        for (int i = 0; i < c.gene.length; i++) {
            copy.gene[i] = c.gene[i];
        }
        return copy;
    }

    /**
     * @param size
     * @Description: 初始化基因长度
     */
    private void initGeneSize(int size) {
        if (size <= 0) {
            return;
        }
        gene = new int[size];
    }


    /**
     * @param c1
     * @param c2
     * @Description: 遗传产生下一代
     */
    public static List<Chromosome> genetic(Chromosome p1, Chromosome p2) {
        if (p1 == null || p2 == null) { //染色体有一个为空，不产生下一代
            return null;
        }
        if (p1.gene == null || p2.gene == null) { //染色体有一个没有基因序列，不产生下一代
            return null;
        }
        if (p1.gene.length != p2.gene.length) { //染色体基因序列长度不同，不产生下一代
            return null;
        }
        Chromosome c1 = clone(p1);
        Chromosome c2 = clone(p2);

        //随机产生交叉互换位置
        int size = c1.gene.length;
        int a = ((int) (Math.random() * size)) % size;
        int b = ((int) (Math.random() * size)) % size;
        int min = a > b ? b : a;
        int max = a > b ? a : b;

        //对位置上的基因进行交叉互换
```

```
            for (int i = min; i <= max; i++) {
                int t = c1.gene[i];
                c1.gene[i] = c2.gene[i];
                c2.gene[i] = t;
            }

            //返回交叉变换后的子代。
            List<Chromosome> list = new ArrayList<Chromosome>();
            list.add(c1);
            list.add(c2);
            return list;
        }

        /**
         * @param num
         * @Description: 基因 num 个位置发生变异
         */
        public void mutation(int num) {
            //允许变异
            int size = gene.length;
            Random random = new Random();
            for (int i = 0; i < num; i++) {
                //寻找变异位置
                int at = ((int) (Math.random() * size)) % size;
                //变异后的值
                //int bool = (int) (Math.random() * 10) % 5;
                int bool = random.nextInt(5);
                gene[at] = bool;
            }
        }

        /**
         * @return
         * @Description: 将基因转化为对应的数字，在本次实例中，返回染色体数组。
         */
        public int[] getNum() {
            return gene;
        }
}
```