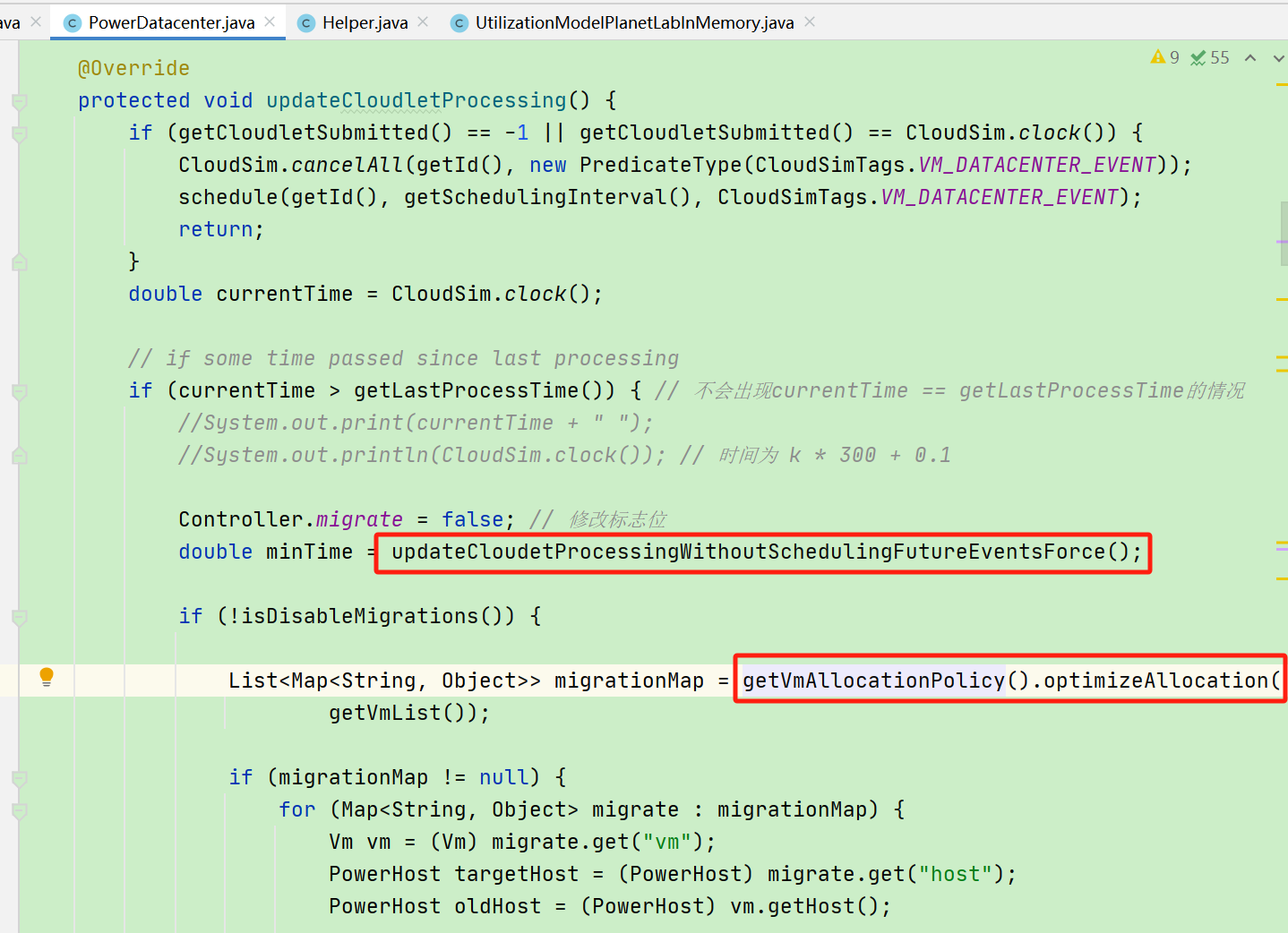
# CloudSim 4.0能耗仿真框架分析

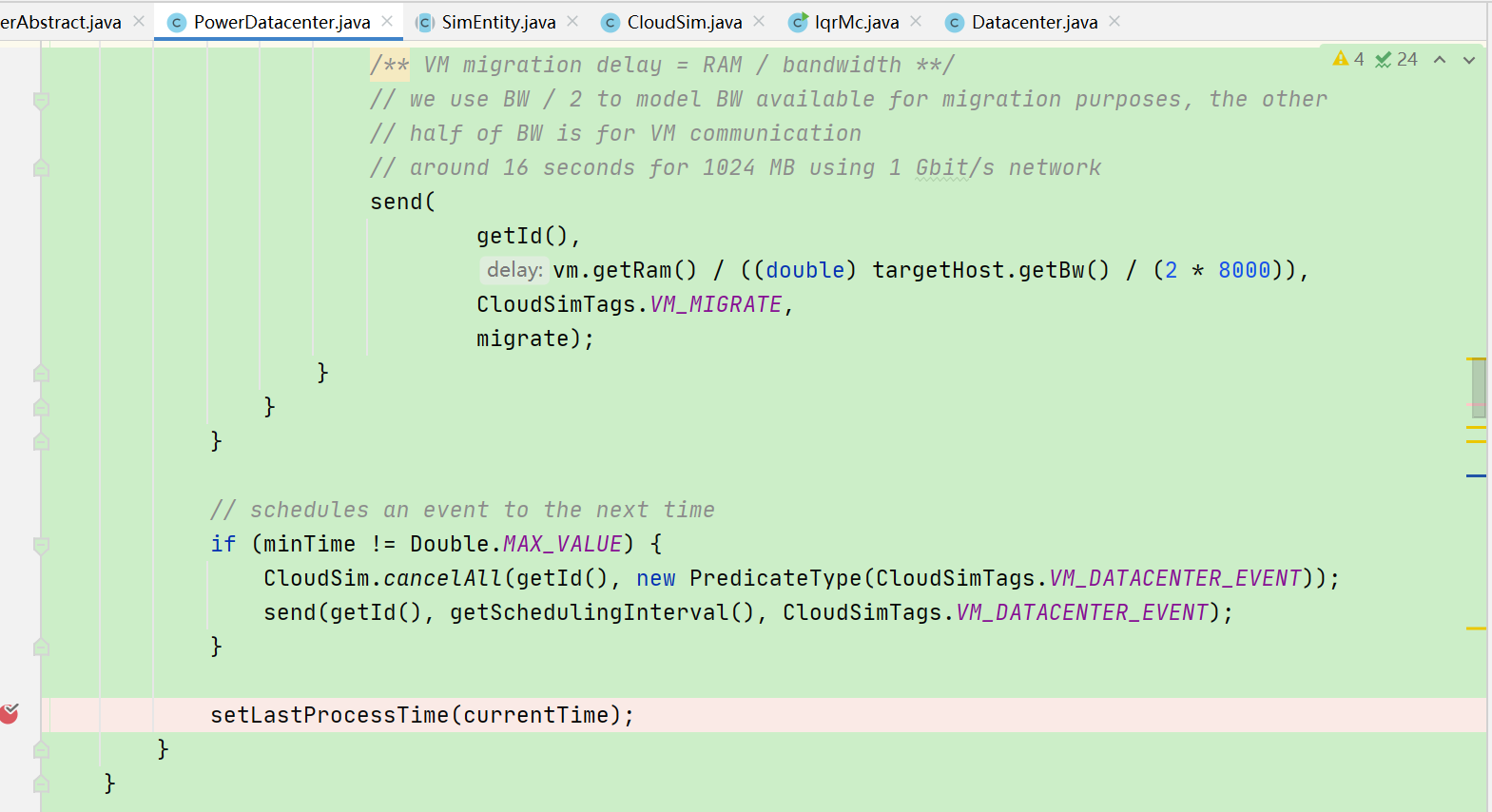
## 1. 虚拟机整合前准备

在k \* 300 + 0.1 (s)时刻（k∈[1,287]），PowerDatacenter类中的updateCloudletProcessing函数会被调用。在该函数内，其将调用updateCloudletProcessingWithoutSchedulingFutureEventsForce函数读取负载轨迹实现云任务利用率的更新，并完成虚拟机与主机状态的更新。随后执行optimizeAllocation函数进行虚拟机整合。上述两函数的执行时间均在k \* 300 + 0.1 (s) 时刻。

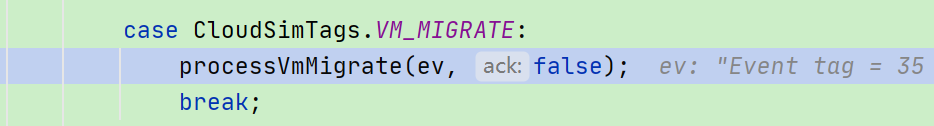


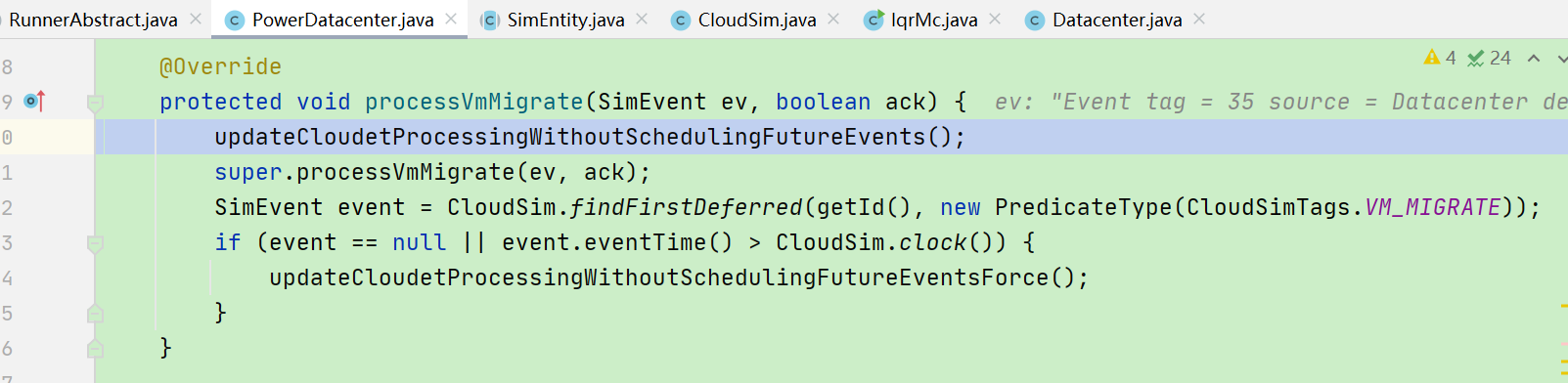
## 2. 虚拟机整合后

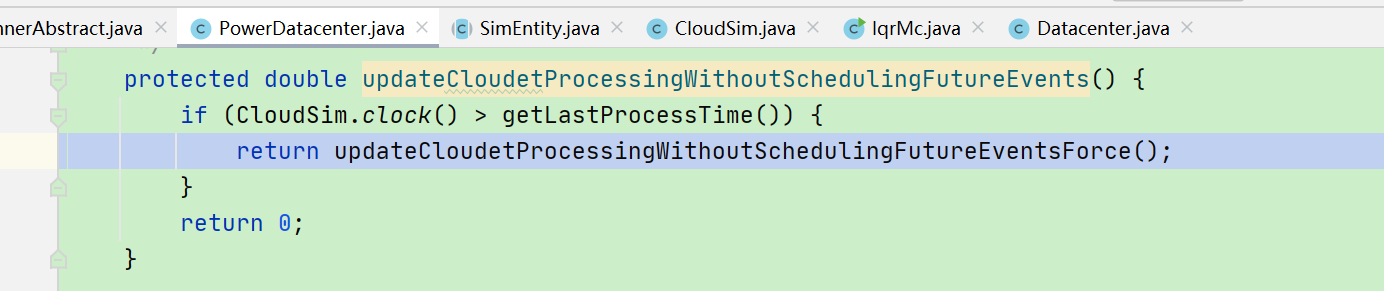
在PowerDatacenter类的updateCloudletProcessing函数中，如果执行optimizeAllocation函数返回的键值对不为空，将发送虚拟机迁移消息进行虚拟机的实际迁移，如下图所示。



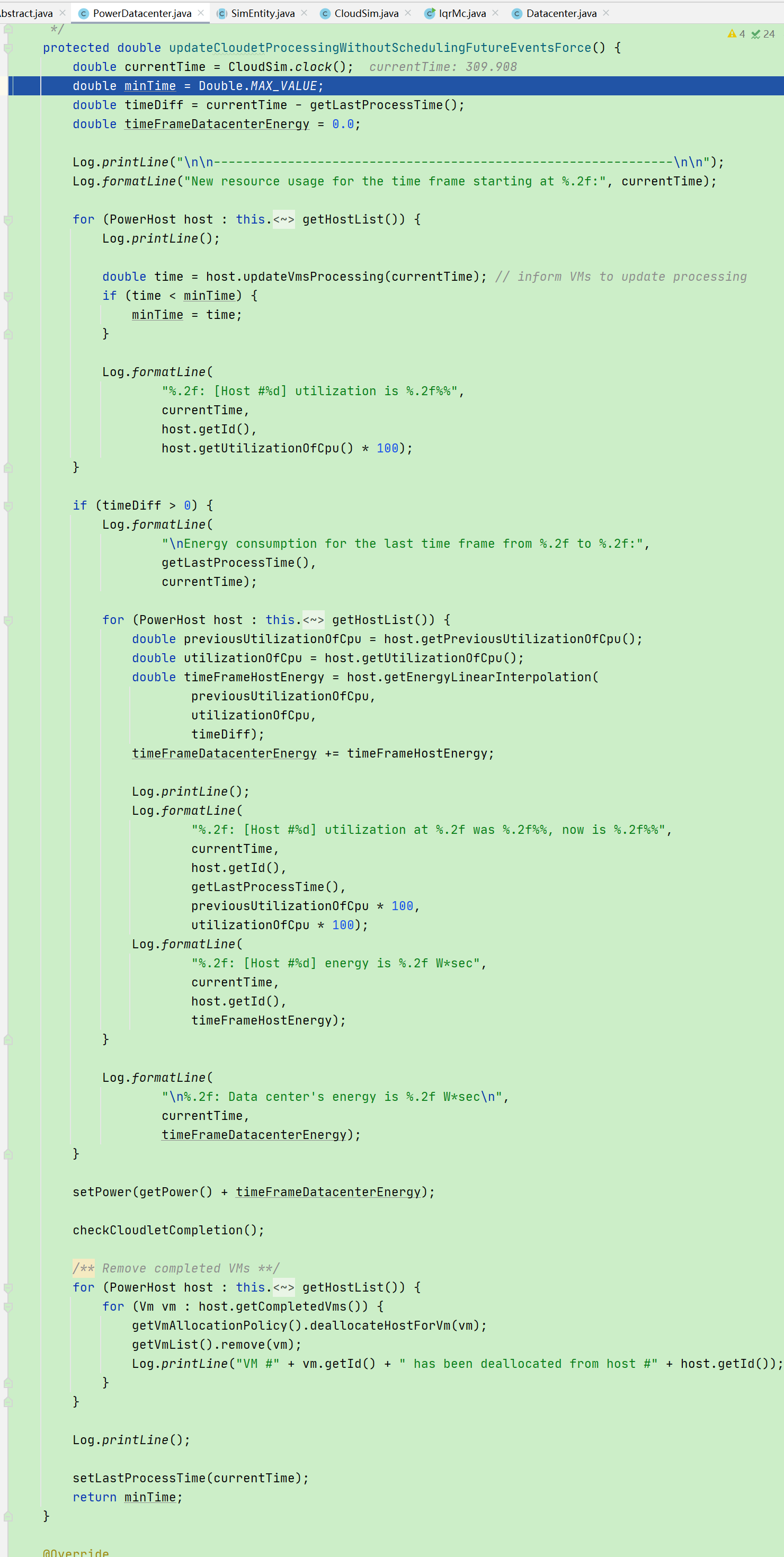
updateCloudletProcessing函数执行完毕后，CloudSim执行虚拟机迁移事件，如下图所示。







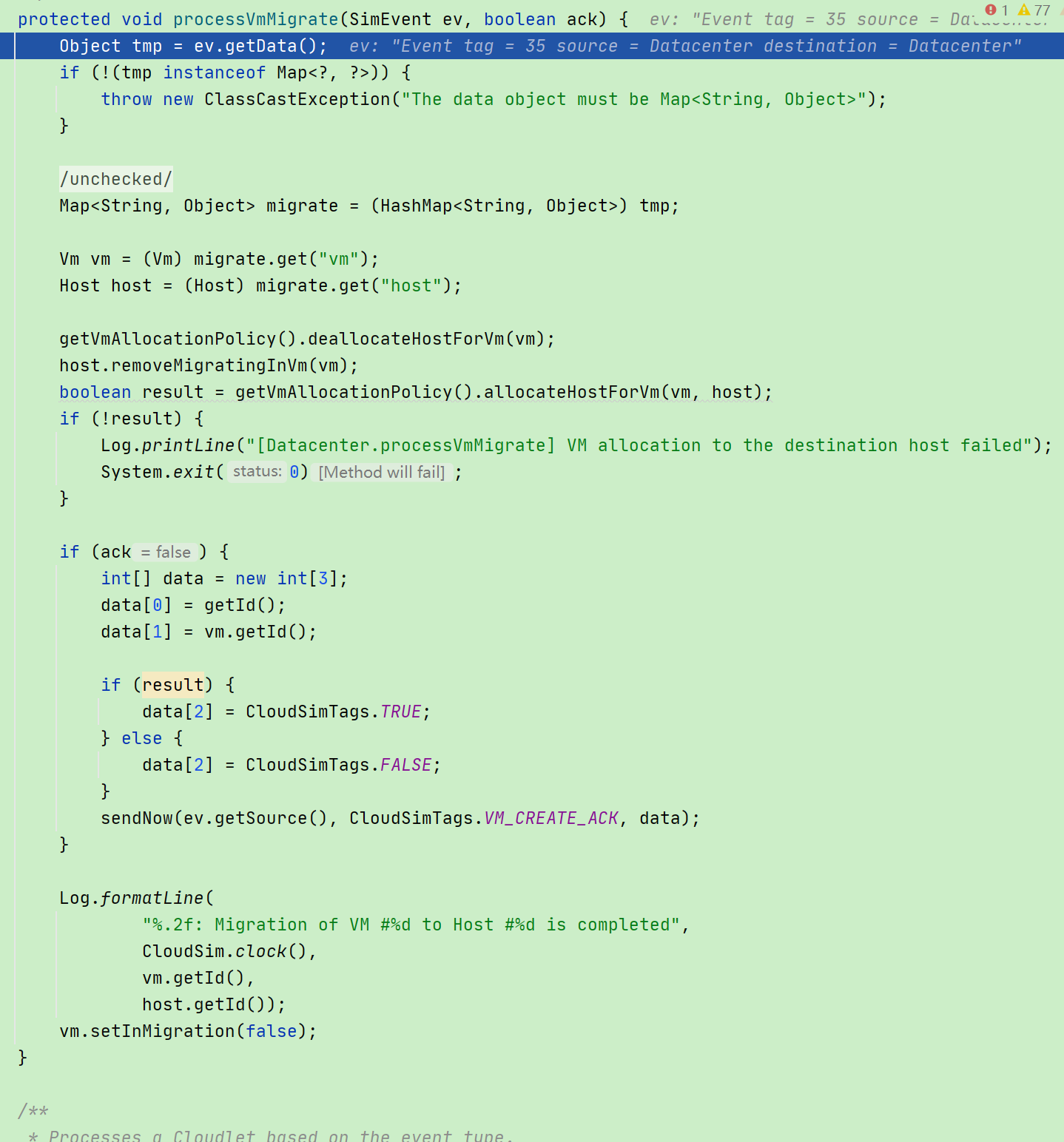
由于虚拟机迁移附带时间开销，因此会导致时间差。



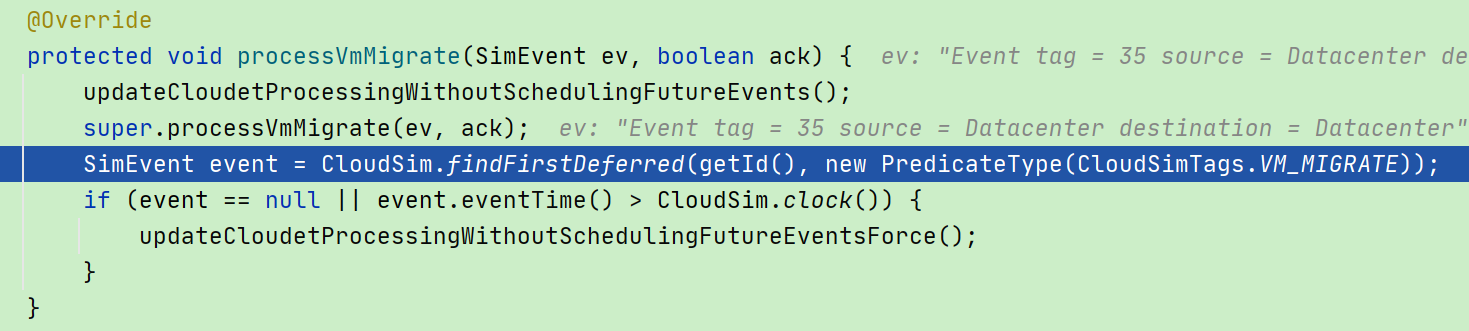
通过updateCloudletProcessingWithoutSchedulingFutureEventsForce函数计算在虚拟机迁移时间延迟这段时间内数据中心（所有主机）产生的能耗，同时主机和虚拟机都记录历史状态。

updateCloudetProcessingWithoutSchedulingFutureEvents函数执行完毕后，当前时刻为上一次整合时刻 + 首次虚拟机迁移的延迟。

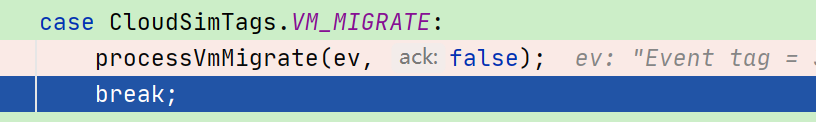
随后调用super.processVmMigrate以完成虚拟机的迁移事务。此函数位于Datacenter类中。



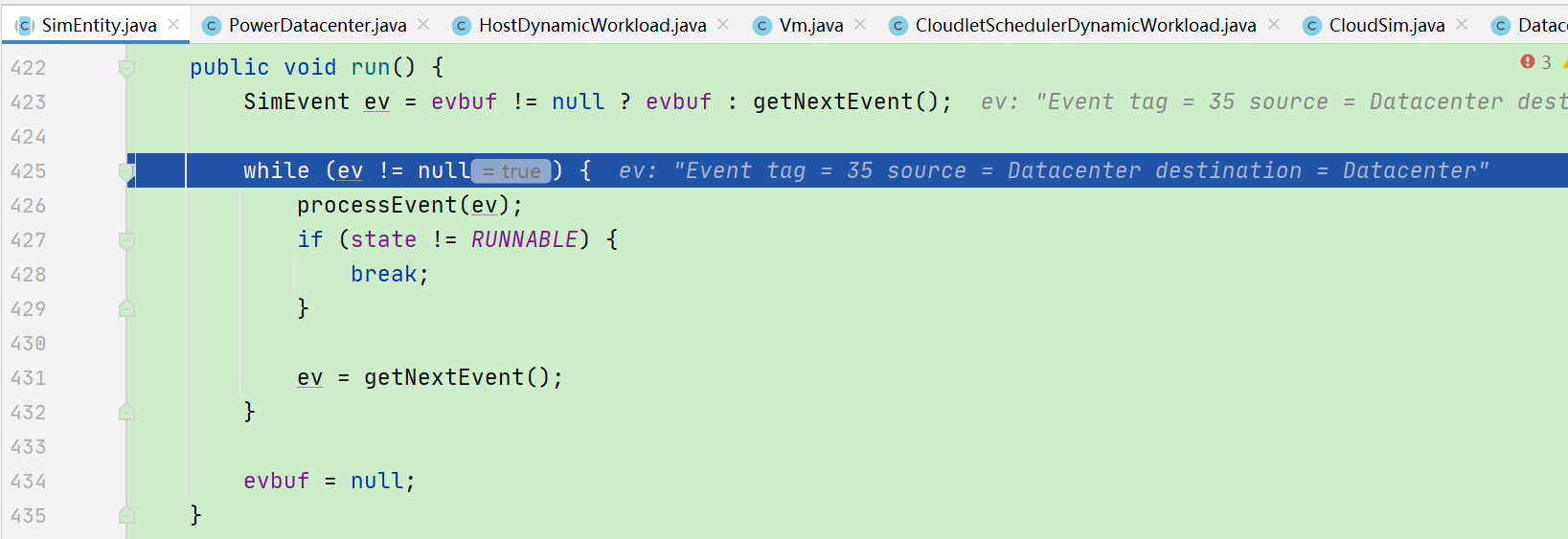
该函数执行完毕后，CloudSim寻找延迟队列中第一个虚拟机迁移事件（updateCloudletProcessing会产生较多的虚拟机迁移事件），如下图所示。

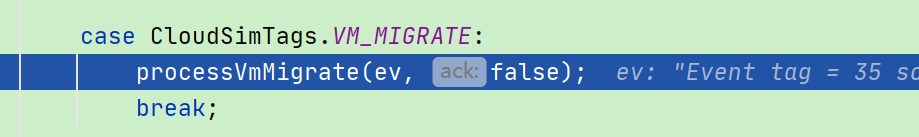


如果该事件的调度时间（虚拟机迁移发生的时刻）大于CloudSim.clock()（CloudSim当前时刻），需要执行updateCloudletProcessingWithoutSchedulingFutureEventsForce函数。之后本次虚拟机迁移处理结束，如下图所示。



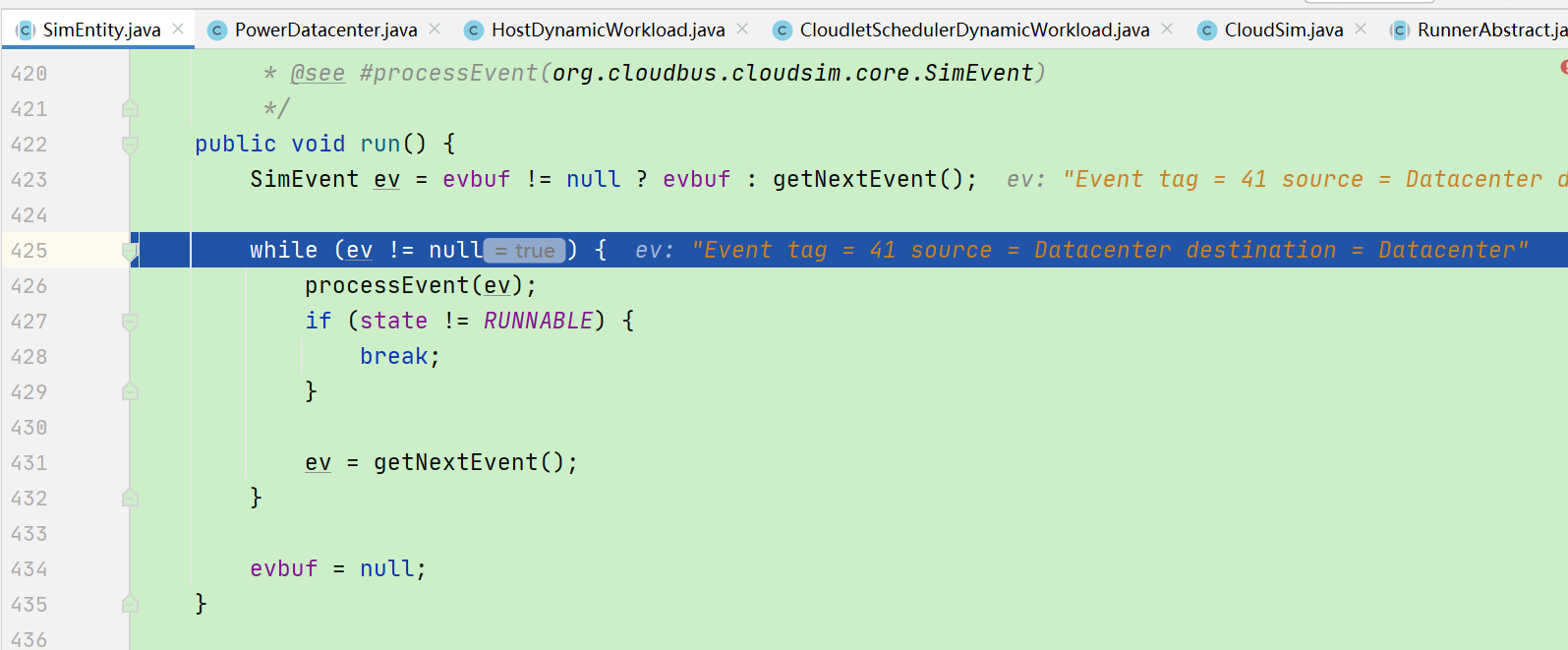
随后，PowerDatacenter实体继续执行下一个事件。

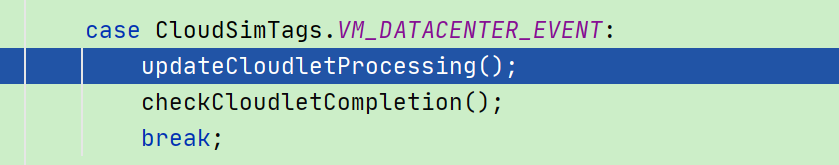


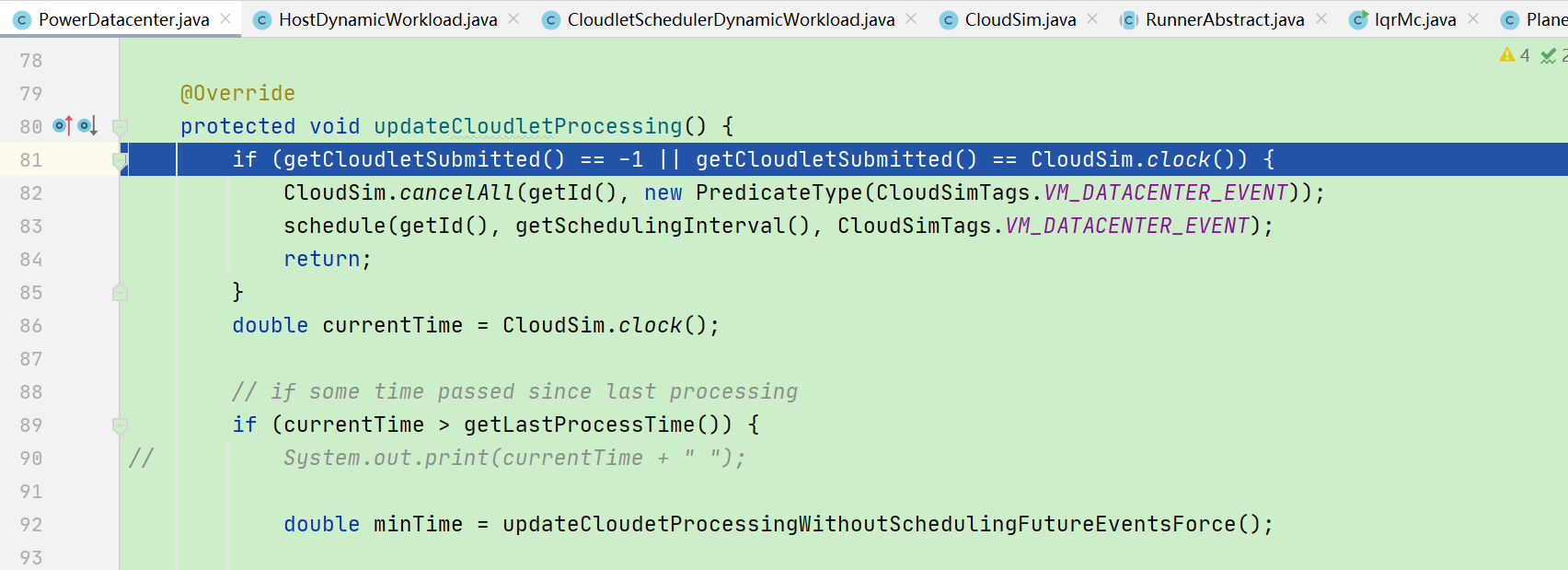


跟前述过程一样循环。

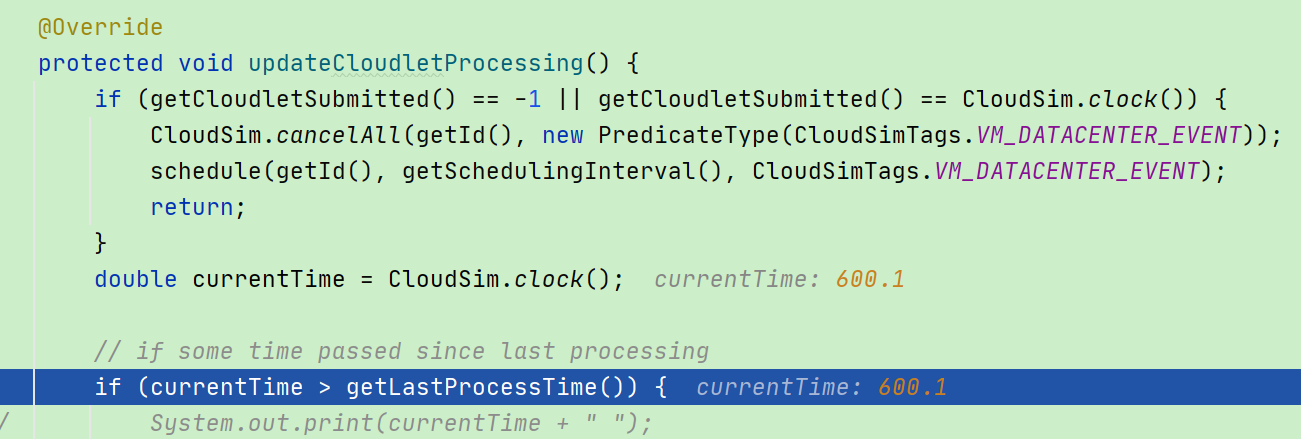
当所有虚拟机迁移事件处理完毕后（经过若干轮runClockTick），结果如下三图所示。







回到了updateCloudletProcessing函数。



如上图所示，CloudSim.clock()来到了600.1s，即将开始第二次虚拟机整合。

## 3.updateCloudetProcessingWithoutSchedulingFutureEventsForce函数详解

## 图3-1 updateCloudetProcessingWithoutSchedulingFutureEventsForce函数

## 调用HostDynamicWorkload类的updateVmsProcessing函数。

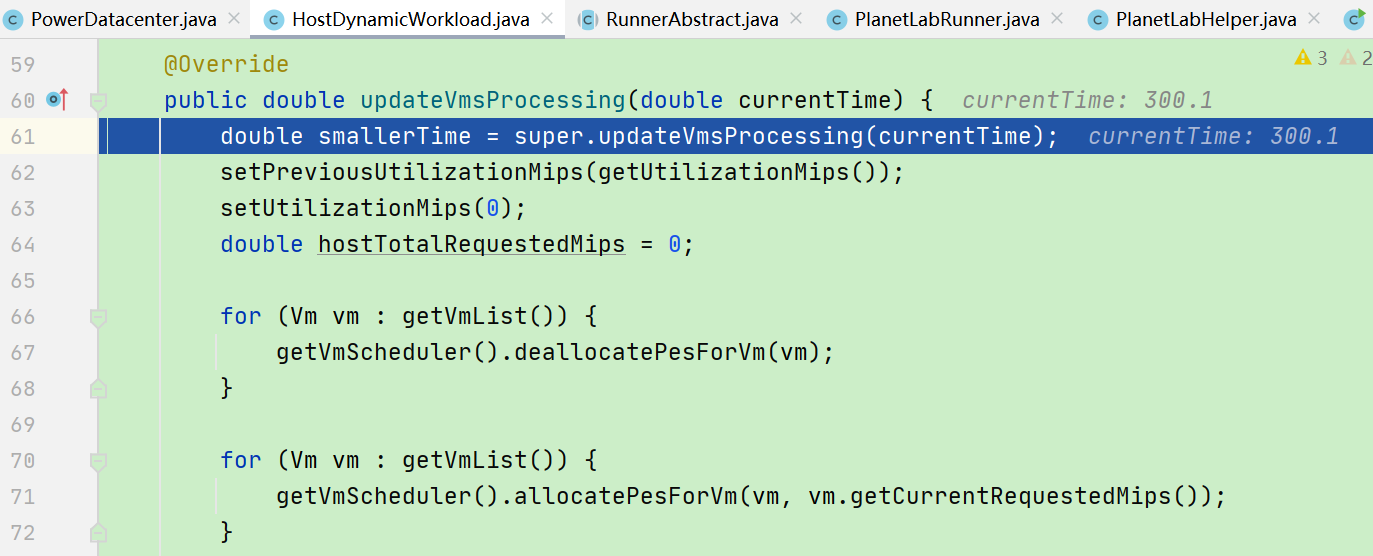


图3-2 HostDynamicWorkload类的updateVmsProcessing函数

调用Host类的updateVmsProcessing函数。

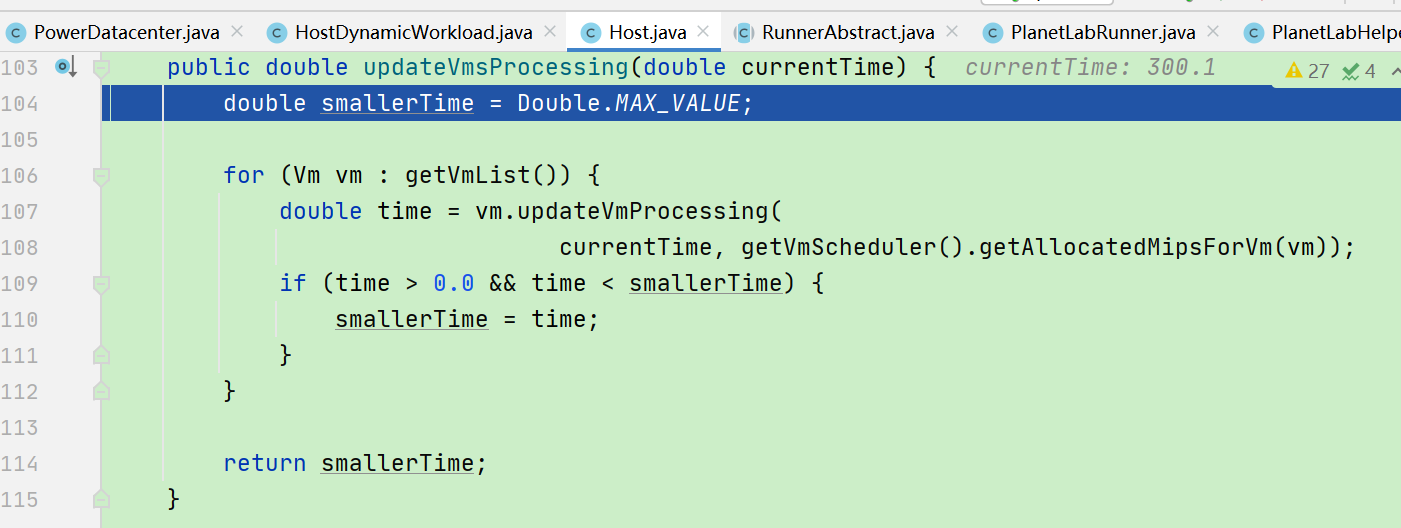


图3-3 Host类的updateVmsProcessing函数

调用PowerVm类的updateVmProcessing函数。

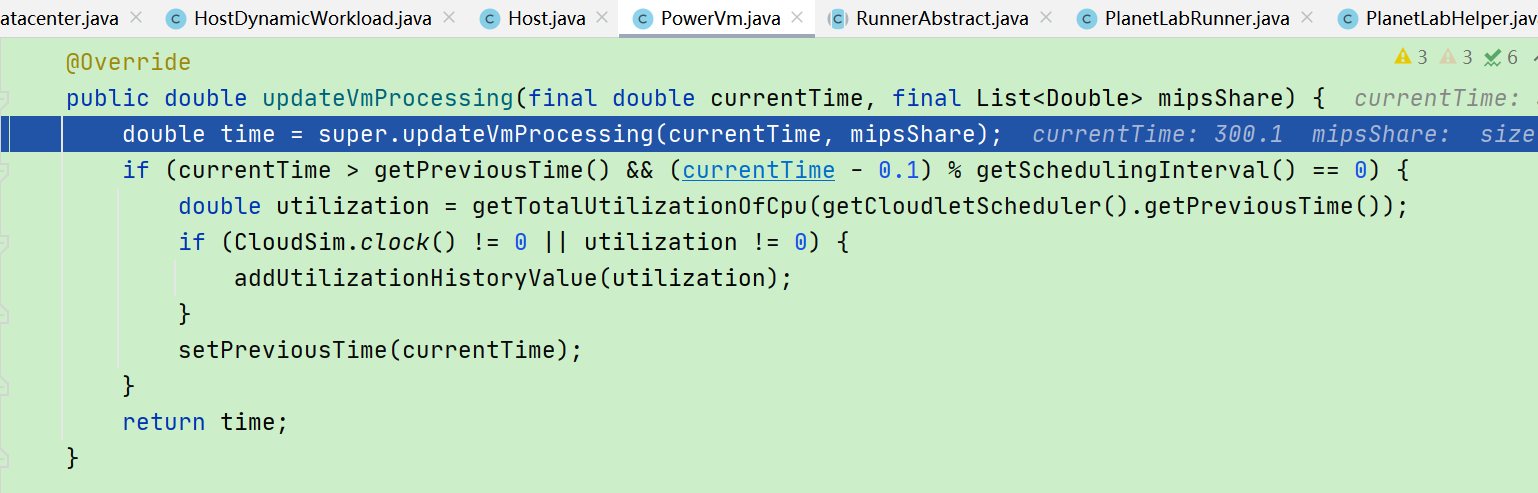


图3-4 PowerVm类的updateVmProcessing函数

调用Vm类的updateVmProcessing函数。

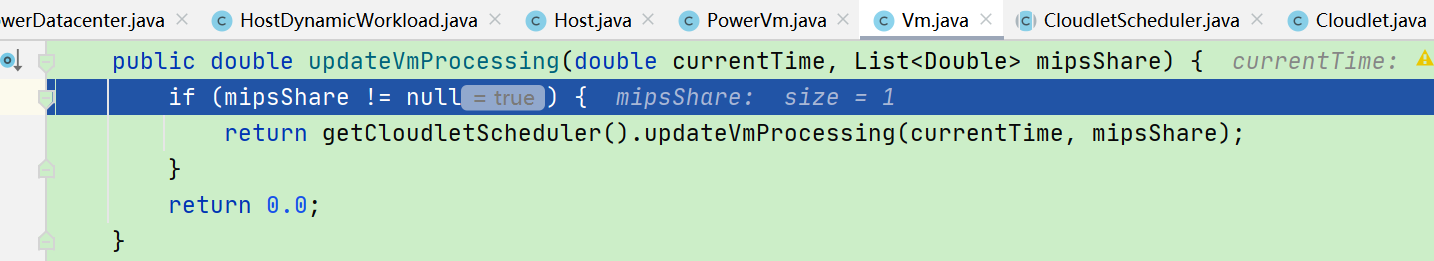


图3-5 Vm类的updateVmProcessing函数

进入CloudletSchedulerDynamicWorkload类的updateVmProcessing函数。

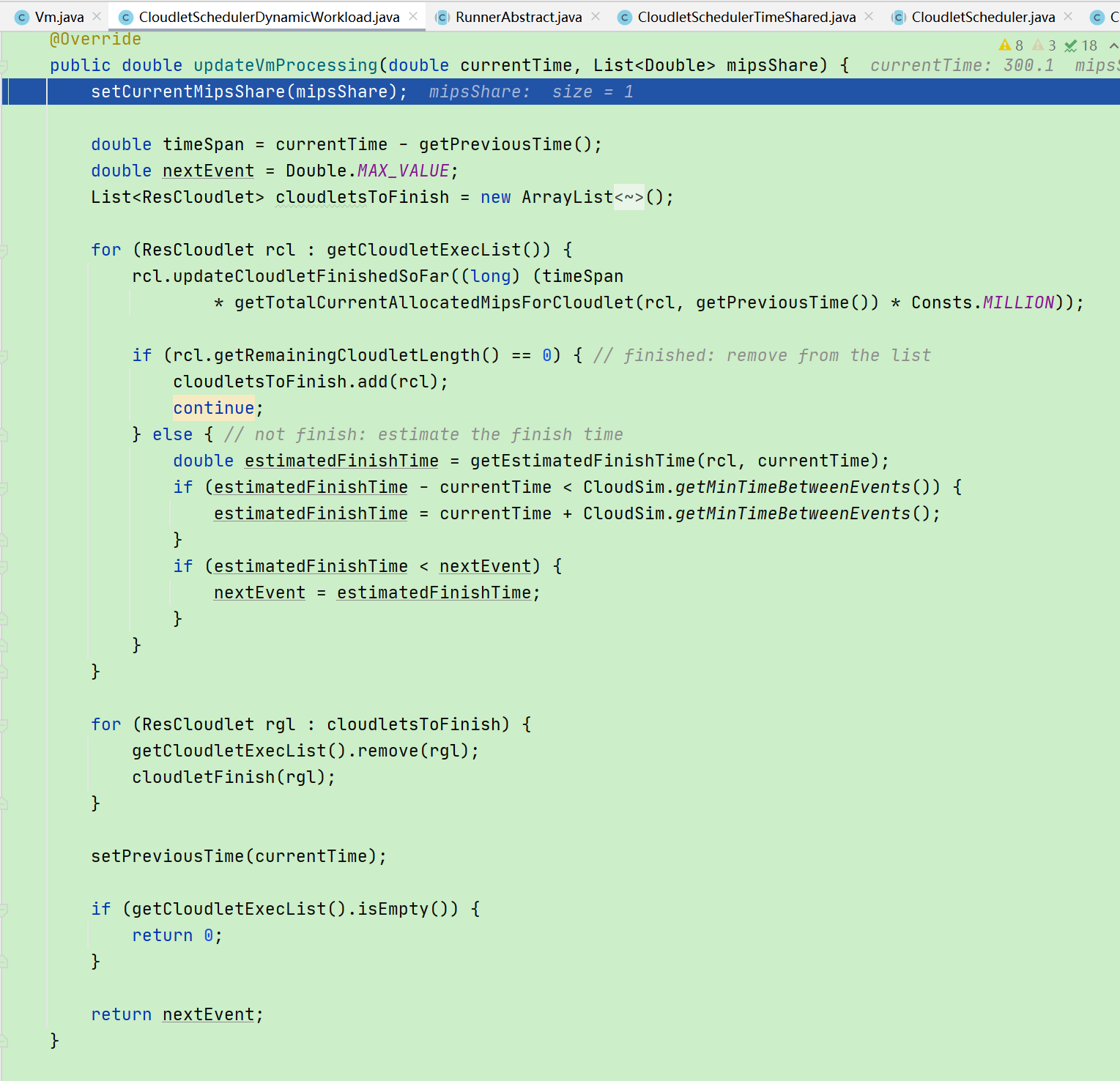
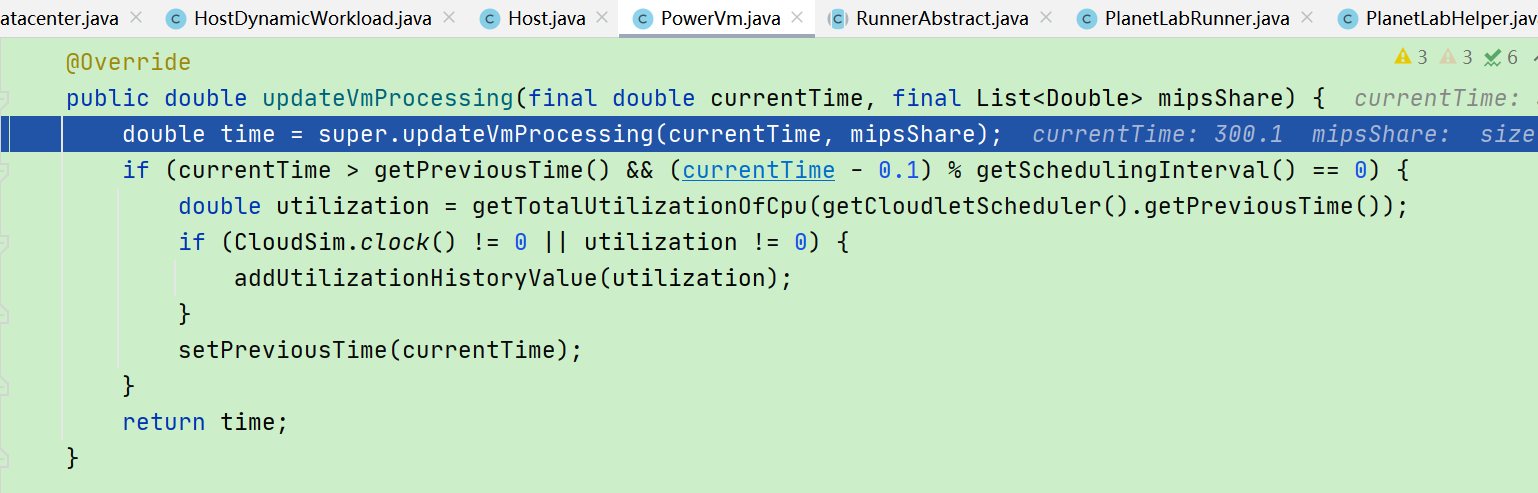


图3-6 CloudletSchedulerDynamicWorkload类的updateVmProcessing函数

这个函数的函数体我们不关心,注意到setPreviousTime(currentTime)，将当前时间设置为了先前时间。执行完该函数后，回退到图3-4，如下所示。



首先进行if判断,如果当前时间>先前时间 并且 在云任务数据的读取节点：读取云任务获取利用率并将其加入到利用率历史中。之后setPreviousTime(currentTime)，将当前时间设置为了先前时间。这里getCloudletScheduler().getPreviousTime()的值已经是当前时间currentTime，前面已经更新过了。

执行完该函数后,回退到图3-3所示函数,图3-3函数执行完后,回退到图3-2所示函数，如下所示。

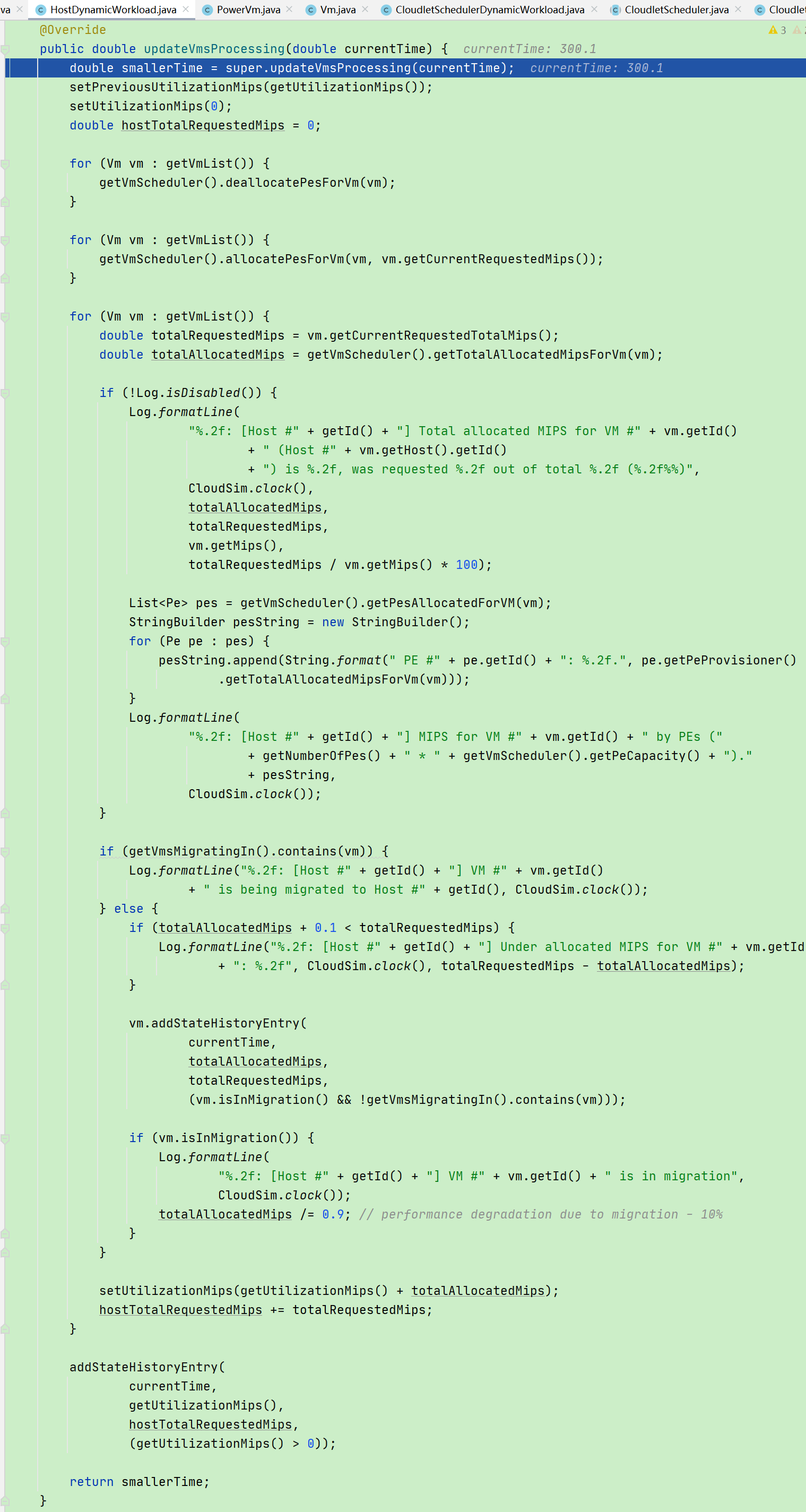


图3-7 HostDynamicWorkload类的updateVmsProcessing函数全景

下面对该函数进行解析：

setPreviousUtilizationMips(getUtilizationMips());

// 将当前利用率设置为先前利用率。根据前面的描述,HostDynamicWorkload类的updateVmsProcessing函数调用了super.updateVmsProcessing，进入到Host类的updateVmsProcessing函数，该函数仅仅完成了Vm的更新，主机的状态并没有发生变化。又根据对HostDynamicWorkload类成员变量utilizationMips和previousUtilizationMips使用的查找，发现除了初始化以外,仅仅在HostDynamicWorikload类的updateVmsProcessing函数中进行更新。因此getUtilizationMips获取的是上一次执行HostDynamicWorkload类的updateVmsProcessing函数中所对主机更新的mips值。

setUtilizationMips(0); // 设当前利用率为0

double hostTotalRequestedMips = 0; // 设该主机上虚拟机所请求的总Mips

对所有虚拟机释放pe，对所有虚拟机配置pe。这里具体的释放和配置pe的函数如何执行我们并不关心。

for (Vm vm: getVmList()) // 迭代主机上的每个虚拟机

double totalRequestedMips = vm.getCurrentRequestedTotalMips(); // 获取vm 的请求量

double totalAllocatedMips = getVmScheduler().getTotalAllocatedMipsForVm(vm);// 获取配置给该vm的 mips量

if (getVmsMigratingIn().contains(vm)) // 如果当前vm正在迁入该主机

Log

else: //当前vm未迁入该主机 在后段分析判断迁入原因

vm.addStateHistoryEntry( // 当前数据加入虚拟机历史

currentTime,

totalAllocatedMips,

totalRequestedMips,

(vm.isInMigration() && !getVmsMigratingIn().contains(vm)));

if (vm.isInMigration) // 如果虚拟机在迁移状态

totalAllocatedMips /= 0.9; // 由于迁移性能退化10%

setUtilizationMips(getUtilizationMips() + totalAllocatedMips);

hostTotalRequestedMips += totalRequestedMips;

addStateHistoryEntry( //当前数据加入主机历史.

currentTime,

getUtilizationMips(),

hostTotalRequestedMips,

(getUtilizationMips() > 0));

在此函数中（HostDynamicWorkload类的updateVmsProcessing），主机完成了利用率的更新，设置了先前利用率previousUtilizationMips和当前利用率UtilizationMips。

函数执行完毕后，返回至图3-1所示函数，如下图所示:



图3-8 updateCloudetProcessingWithoutSchedulingFutureEventsForce函数剩余部分

分析过程如下:

timeDiff = currentTime - getLastProcessTime() ///计算当前时间和上一次处理时间的时间差

timeFrameDatacenterEnergy = 0.0;

if (timeDiff > 0) // 时间差 > 0 计算从上一次处理时刻到现在所产生的能耗

for (PowerHost host: getHostList)

PreivousUtilizationOfCpu = // 获取lastProcessTime时刻时主机的利用率

utilizationOfCpu = // 当前时刻主机的利用率

timeFrameHostEnergy = host.getEnergyLinearInterpolation(timeDiff);

timeFrameDatacenterEnergy += timeFrameHostEnergy;

setPower(getPower() + timeFrameDatacenterEnergy);

// 成员变量power : 服务器集群产生的能耗

checkCloudletCompletion();// 此函数我们不关心

// 经单步调试,函数体内的sendNow函数不会被调用

/\*\* Remove completed VMs \*\*/

for (PowerHost host : this.<PowerHost> getHostList()) {

for (Vm vm : host.getCompletedVms()) {

getVmAllocationPolicy().deallocateHostForVm(vm);

getVmList().remove(vm);

Log.printLine("VM #" + vm.getId() + " has been deallocated from host #" + host.getId());

}

} // 该双层for循环我们并不关心

setLastProcessTime(currentTime); // 将当前时间设置为上次处理时间

至此，updateCloudetProcessingWithoutSchedulingFutureEventsForce函数分析完毕。

### 3.1 Vm.getCurrentRequestedTotalMips函数分析：

功能：计算当前虚拟机的请求量

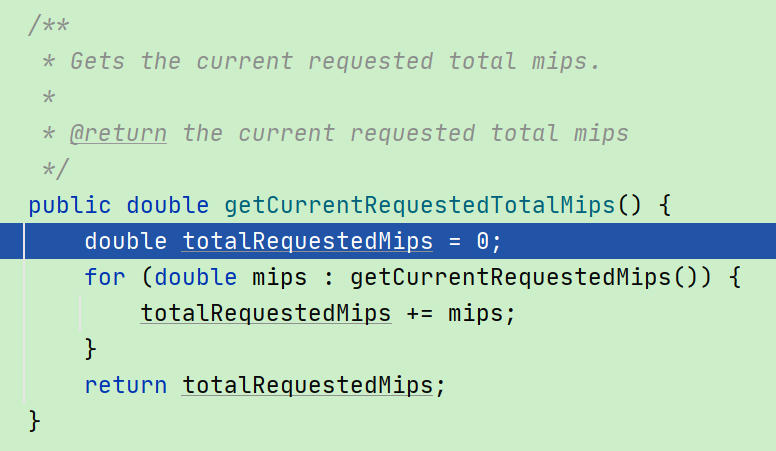


图3-9 vm.getCurrentRequestedTotalMips函数



图3-10 vm.getCurrentRequestedMips函数

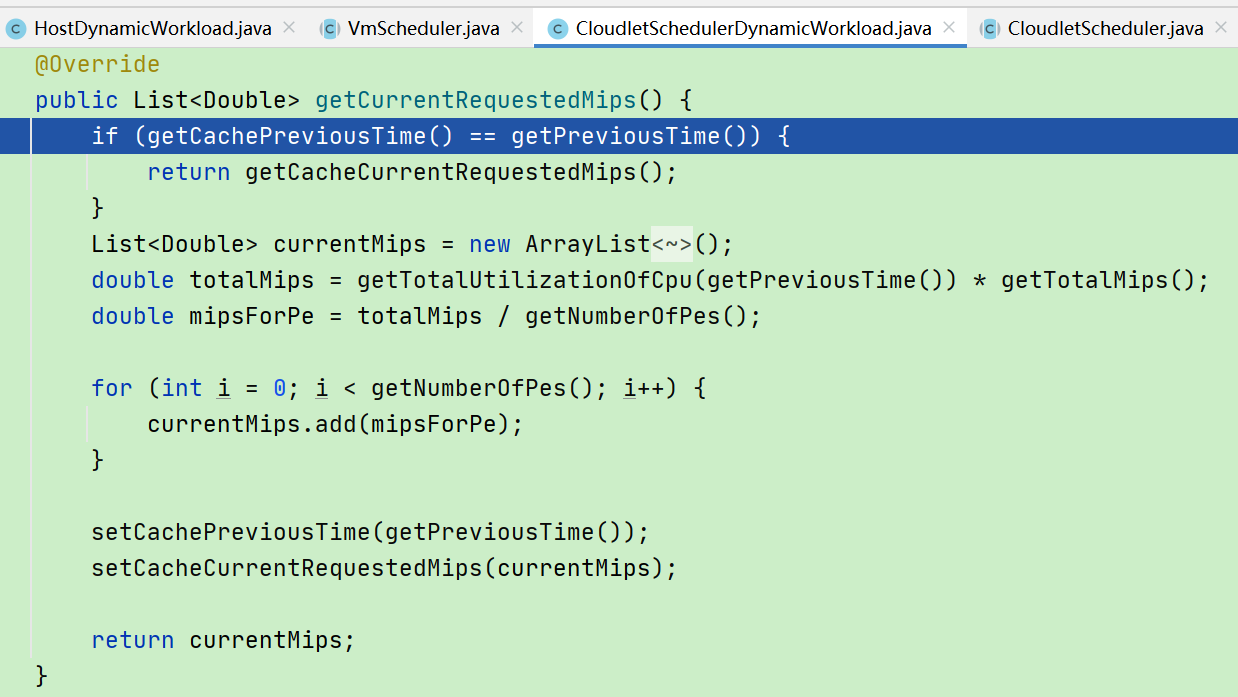


图3-11 CloudletSchedulerDynamicWorkload.getCurrentRequestedMips函数

该函数出现了多个PreviousTime，根据前述分析，PreviousTime已经被赋值为了

当前时间。因此，实际上这些实参就是当前时间。正如函数名:得到当前请求的Mips。

如果cache,返回缓存。

将请求的mips总量平均分配到每个核心(默认虚拟机的核心数为1),组成返回队列currentMips。

将cache值设置为当前数据。

执行完此函数后,回到图3-10所示:

如果虚拟机在进行初始化:则currentRequestedMips由最大容量构成。

当虚拟机创建完毕后,虚拟机就不在初始化阶段,因此在虚拟机整合过程中,都是得到getCurrentRequestedMips函数的结果。

返回currentRequestedMips。

返回至图3-9所示:

将列表中元素相加,得到总请求量。

至此，函数分析结束。

### 3.2 updateCloudletProcessing函数分析：

PowerDatacenter.updateCloudletProcessing



图3-12 PowerDatacenter类的updateCloudletProcessing函数

此函数由Datacenter类的事件调用,如图3-13所示:

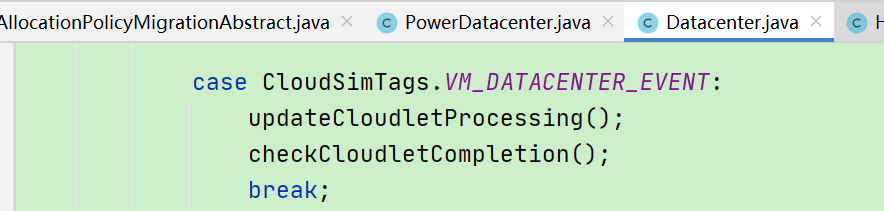


图3-13 VM\_DATACENTER\_EVENT

现解析PowerDatacenter类的updateCloudletProcessing函数。

if (getCloudletSubmitted() == -1 || getCloudletSubmitted() == CloudSim.clock())

schedule;

return; // 该if代码块只在0.1s时刻执行,不影响虚拟机整合阶段

double currentTime = CloudSim.clock(); // 获取当前时间

if (currentTime > getLastProcessTime()) // some time passed since last processing

updateCloudetProcessingWithoutSchedulingFutureEventsForce();// 完成主机、虚拟机的更新(获取新的云任务的利用率)

if (!isDisableMigrations) // 允许虚拟机迁移

migrationMap = optimizeAllocation(getVmList());

if (migrationMap != Null)

for ( Map<String, Object> migrate : migrationMap)

vm;

targetHost;

oldHost;

targetHost.addMigratingInVm(vm); //将vm加入到目标主机的迁入集合中

incrementMigrationCount();

send( // 发送迁移消息

getId(),

vm.getRam() / ((double) targetHost.getBw() / (2 \* 8000)),

CloudSimTags.VM\_MIGRATE,

migrate);

if (minTime != Double.MAX\_VALUE) // schedules an event to the next time

Send(SchedulingInterval);

setLastProcessTime(currentTime); // 将当前时间设置为上次处理时间

PowerDatacenter类的updateCloudletProcessing函数解析结束。

### 3.3 addMigratingInVm函数分析：

Host.addMigratingInVm



图3-14 Host类的addMigratingInVm函数

现解析该函数。

vm.setInMigration(true); // 将虚拟机设置为迁移状态

if (!getVmsMigratingIn().contains(vm))

判断是否满足约束条件

getVmScheduler().getVmsMigratingIn().add(vm.getUid());

getVmsMigratingIn().add(vm); //将虚拟机加入到主机的迁入队列

getVmList().add(vm); //加入到主机的虚拟机列表当中

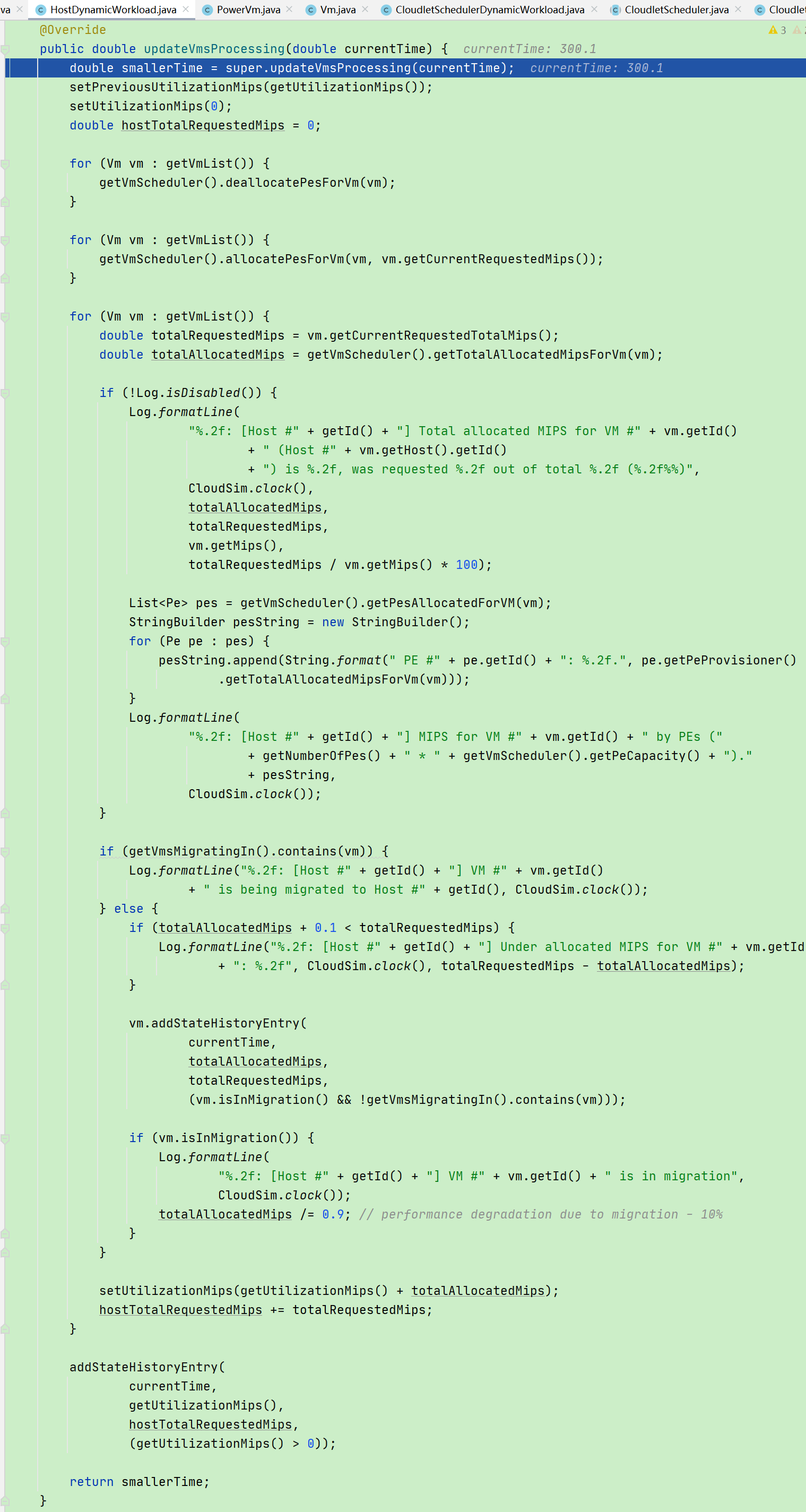
updateVmsProcessing(cloudSim.clock); // 当前目标主更新虚拟机

vm.getHost().updateVmsProcessing(cloudsim.clock); // 宿主机更新虚拟机

以上两条指令都是调用的HostDynamicWorkload类的updateVmsProcessing函数

addMigratingInVm函数解析至此结束。

### 3.4 HostDynamicWorkload类updateVmsProcessing函数补充说明



函数全景图

首先调用super.updateVmsProcessing,进入到Host类的updateVmsProcessing函数,该函数仅仅完成了Vm的更新,主机的状态并没有发生变化.

setPreviousUtilizationMips(getUtilizationMips());

// 将当前利用率设置为先前利用率

setUtilizationMips(0); // 设当前利用率为0

double hostTotalRequestedMips = 0; // 设该主机上虚拟机所请求的总Mips为0

for (Vm vm : getVmList()) {

getVmScheduler().deallocatePesForVm(vm);

}

for (Vm vm : getVmList()) {

getVmScheduler().allocatePesForVm(vm, vm.getCurrentRequestedMips());

}

// 以上两个for循环暂不关心

for (Vm vm: getVmList()) // 迭代主机上的每个虚拟机

double totalRequestedMips = vm.getCurrentRequestedTotalMips(); // 获取vm 的请求量

double totalAllocatedMips = getVmScheduler().getTotalAllocatedMipsForVm(vm);// 获取配置给该vm的 mips量.

if (getVmsMigratingIn().contains(vm)) // 如果当前vm正在迁入该主机

Log

else: //当前vm未迁入该主机

vm.addStateHistoryEntry( // 当前数据加入虚拟机历史

currentTime,

totalAllocatedMips,

totalRequestedMips,

(vm.isInMigration() && !getVmsMigratingIn().contains(vm)));

if (vm.isInMigration) // 如果虚拟机在迁移状态

totalAllocatedMips /= 0.9; // 由于迁移性能退化10%

setUtilizationMips(getUtilizationMips() + totalAllocatedMips);

hostTotalRequestedMips += totalRequestedMips;

addStateHistoryEntry( //当前数据加入主机历史.

currentTime,

getUtilizationMips(),

hostTotalRequestedMips,

(getUtilizationMips() > 0));

该函数有两个调用入口。

1.PowerDatacenter的updateCloudletProcessing函数体中的targetHost.addMigratingInVm函数。其会在addMigratingInVm内被执行。而addMigratingInVm函数只会被PowerDatacenter的updateCloudletProcessing函数调用。

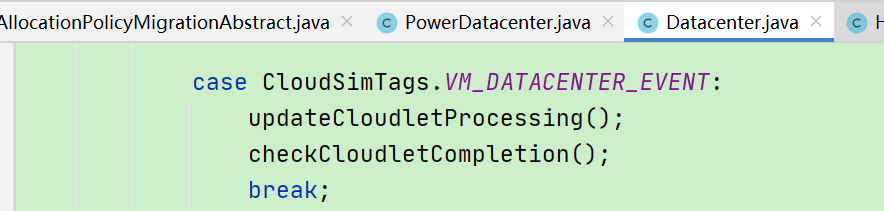
2.PowerDatacenter的updateCloudetProcessingWithoutSchedulingFutureEventsForce函数体内。而updateCloudetProcessingWithoutSchedulingFutureEventsForce函数会被PowerDatacenter.updateCloudletProcessing函数调用、会被PowerDatacenter.processVmMigrate函数调用、会被PowerDatacenter.updateCloudletProcessingWithoutSchedulingFutureEvents函数调用。

函数功能：

HostDynamicWorkload.updateVmsProcessing函数更新主机(CPU利用率)、虚拟机状态:setPreviousUtilizationMips、setUtilizationMips。

它的实参为当前时间currentTime，但此函数和Datacenter成员变量LastProcessTime无关。

### 3.5 PowerDatacenter.updateCloudletProcessing过程总结：



该函数调用入口如上图所示。

该函数在执行过程中，没有对时间进行更新，所有代码均在某一特定时刻进行。

进入函数体后:

double currentTime = CloudSim.clock();

if (currentTime > getLastProcessTime) // if some time passed since last processing

updateCloudetProcessingWithoutSchedulingFutureEventsForce();

// 在此函数中完成上次处理时间与当前时间间隔的数据中心能耗的计算

// 此函数调用了HostDynamicWorkload.updateVmsProcessing函数

// 完成主机和虚拟机的更新. setPreviousUtilizationMips、setUtilizationMips.

// 它的实参为当前时间currentTime,但此函数和Datacenter成员变量

// LastProcessTime无关.

// 同样,updateCloudetProcessingWithoutSchedulingFutureEventsForce函数在

// 执行的过程中都在某一特定时刻,没有对时间进行更新。

// updateCloudetProcessingWithoutSchedulingFutureEventsForce在对数据中心 // 中所有主机调用完host,updateVmsProcessing函数后,开始进行能耗的计算.

// double timeDiff = currentTime - getLastProcessTime();

// if(timeDiff > 0) 根据主机的previousUtilization和Utilization计算能耗.

if (!isDisableMigrations) // 允许迁移

migrationMap = optimizeAllocation; // 获取映射集合

if (migrationMap != null)

for (migrate : migrationMap)

vm;

targetHost;

targetHost.addMigratingInVm(vm);

incrementMigrationCount();

Send(VM\_MIGRATE).

setLastProcessTime(currentTime);

其中targetHost.addMigratingInVm(vm)指令调用了两次HostDynamicWorkload.updateVmsProcessing函数。

一次是目标主机调用，一次是宿主机调用。

因此，目标主机和宿主机的利用率等属性被更新了，注意到：CloudSim.clock()并没有发生变化。

接下来考虑事件，每一个迁移映射都触发一个迁移事件。

send(

getId(),

vm.getRam() / ((double) targetHost.getBw() / (2 \* 8000)),

CloudSimTags.VM\_MIGRATE,

migrate);

注意到：所有涉及虚拟机迁移的目标主机和宿主机的利用率都更新了。

当PowerDatacenter.updateCloudletProcessing函数执行完后，经过虚拟机迁移时间延迟，CloudSim开始执行虚拟机迁移事件，如下图所示。

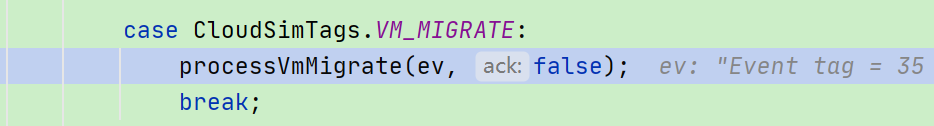


图3-15 虚拟机迁移事件

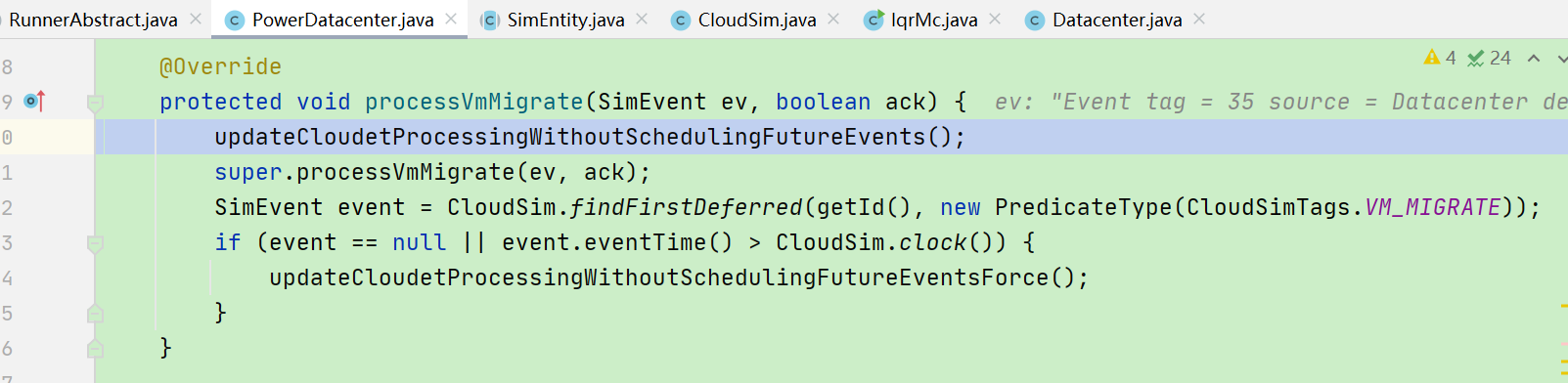


图3-16 PowerDatacenter.processVmMigrate函数

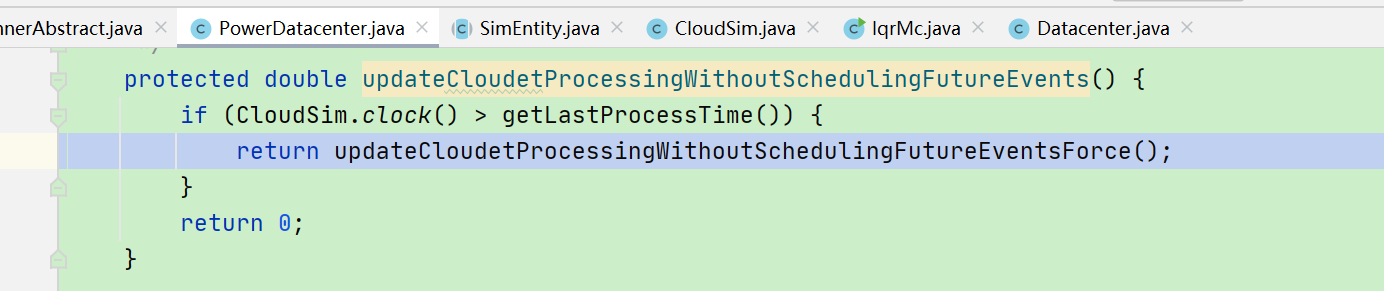


图3-17 PowerDatacenter.updateCloudletProcessingWithoutSchedulingFutureEvents函数

为什么要判断CloudSim.clock>getLastProcessTime()？

因为该函数的核心功能是计算时间差区间内的能耗，没有时间差就没必要调用。

由于虚拟机迁移会有延迟,因此当第一次调用时processVmMigrate函数时,CloudSim.clock > getLastProcessTime,然后执行updateCloudetProcessingWithoutSchedulingFutureEventsForce函数。

该函数的功能是:完成主机状态(利用率)的更新，如果当前时间与上一次执行时间不同,则计算该区间内的能耗。

执行完该函数后，调用super.processVmMigrate，此函数位于Datacenter类中，功能是完成虚拟机的迁移事务，如下图所示。

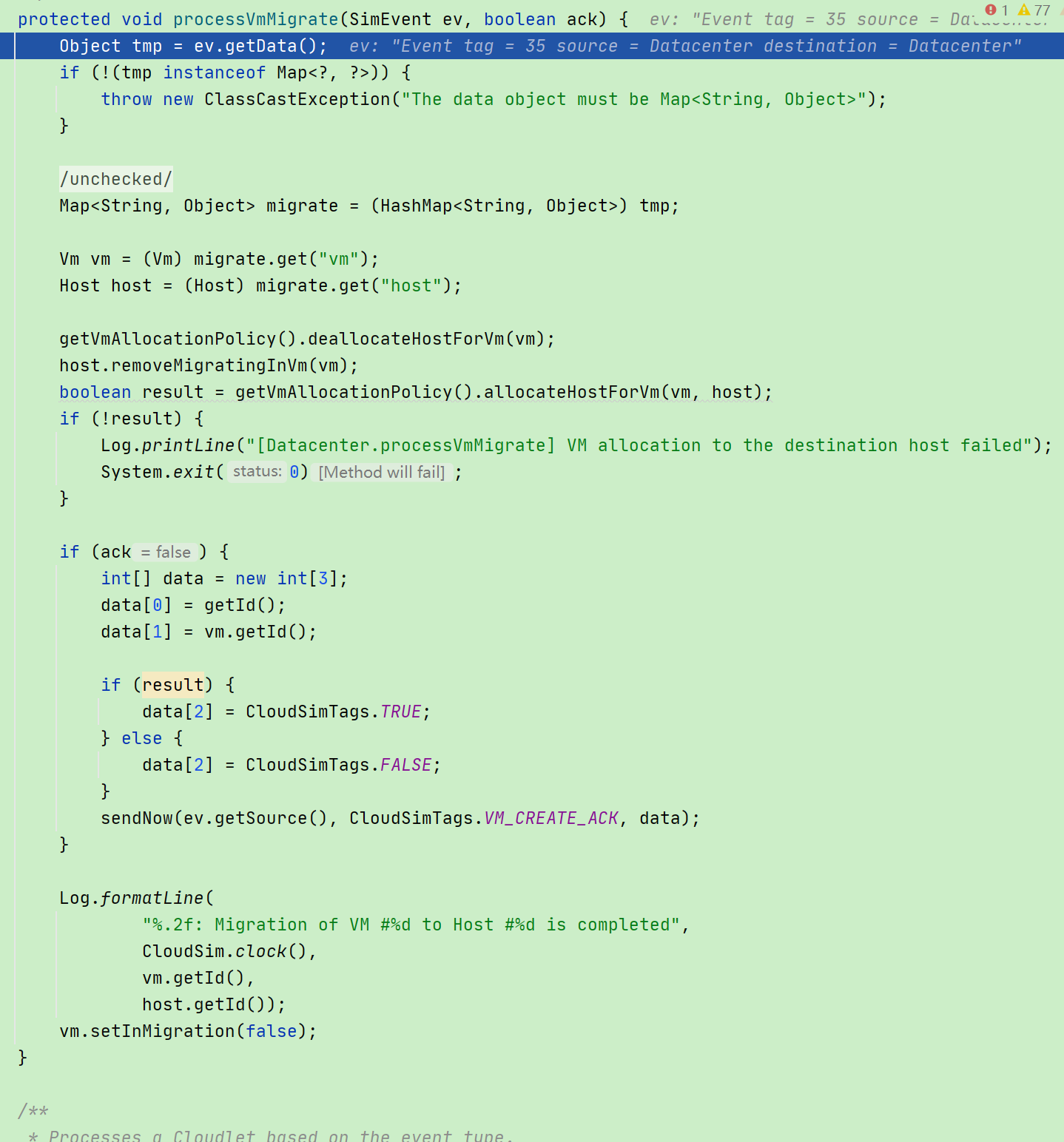


图3-18 Datacenter.processVmMigrate函数

该函数执行完后,如下图所示:

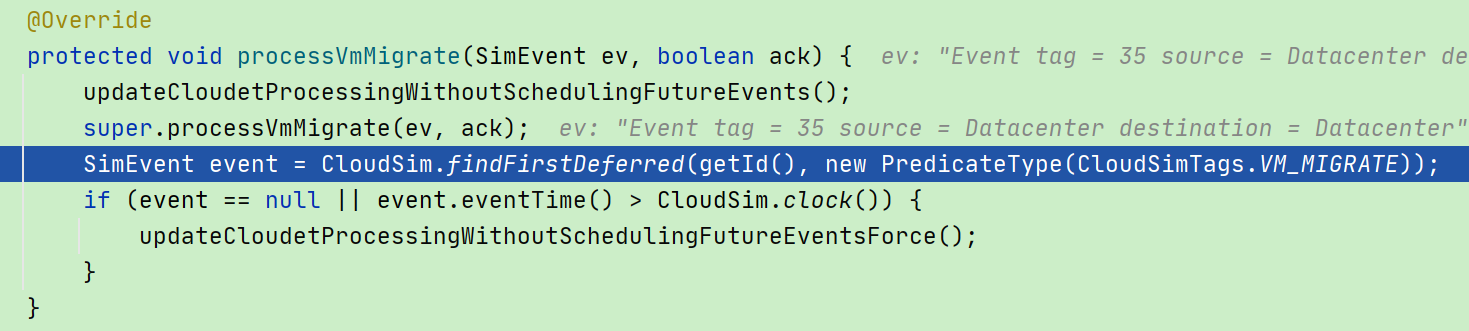


图3-19

寻找延迟队列中第一个虚拟机迁移事件（因为updateCloudletProcessing会产生较多的虚拟机迁移事件）。

如果事件为空或者这个事件的调度时间（虚拟机迁移发生的时刻）大于CloudSim当前时刻，需要再次执行updateCloudletProcessingWithoutSchedulingFutureEventsForce函数。

注意:

如果event == null，代表延迟队列中没有虚拟机迁移事件，此时调用updateCloudletProcessingWithoutSchedulingFutureEventsForce函数的原因是想要对主机利用率进行更新，以计算(interval ~ migration\_delay)时间段内的能耗。因为processVmMigrate操作的对象是虚拟机，通过deallocateHostForVm和allocateHostForVm实现虚拟机的迁移，这两个函数主要涉及PE，而不涉及utilization(CPU利用率)，processVmMigrate并不涉及主机利用率，跟主机利用率无关。

如果event.eventTime() > CloudSim.clock()，代表该迁移事件之后还有别的迁移事件待处理。此时调用updateCloudletProcessingWithoutSchedulingFutureEventsForce函数的原因也是想要对主机利用率进行更新。以进行下次计算{migrate\_delay1~migrate\_delay2期间数据中心产生的能耗}。

当event.eventTime() =CloudSim.clock()时，这是唯一不执行函数的情况。此时意味着延迟队列中存在虚拟机迁移事件的调用时间与当前时间一致。

第一个原因:currentTime - delay ~ currentTime这段时间数据中心的能耗已经计算过。

第二个原因:当前PowerDatacenter.processVmMigrate函数执行完后,由于延迟队列仍存在虚拟机迁移事件。因此还将继续处理虚拟机迁移事件，如图3-19所示。此时再次进入PowerDatacenter.processVmMigrate函数。首先将调用updateCloudletProcessingWithoutSchedulingFutureEvents函数，如图3-17所示。由于CloudSim.clock = getLastProcessTime()，因此该函数的函数体不会执行。之后执行Datacenter.processVmMigrate函数，如图3-18所示。然后继续执行判断event步骤，前面已述。

由于出口只有event == null 和event.eventTime > CloudSim.clock,因此终究必将调用updateCloudletProcessingWithoutSchedulingFutureEventsForce函数对主机利用率进行更新。

之后本次vm\_migrate处理结束:

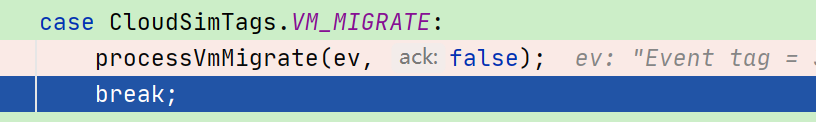


图3-20 vm\_migrate处理结束

PowerDatacenter实体继续执行下一个事件(vm迁移).

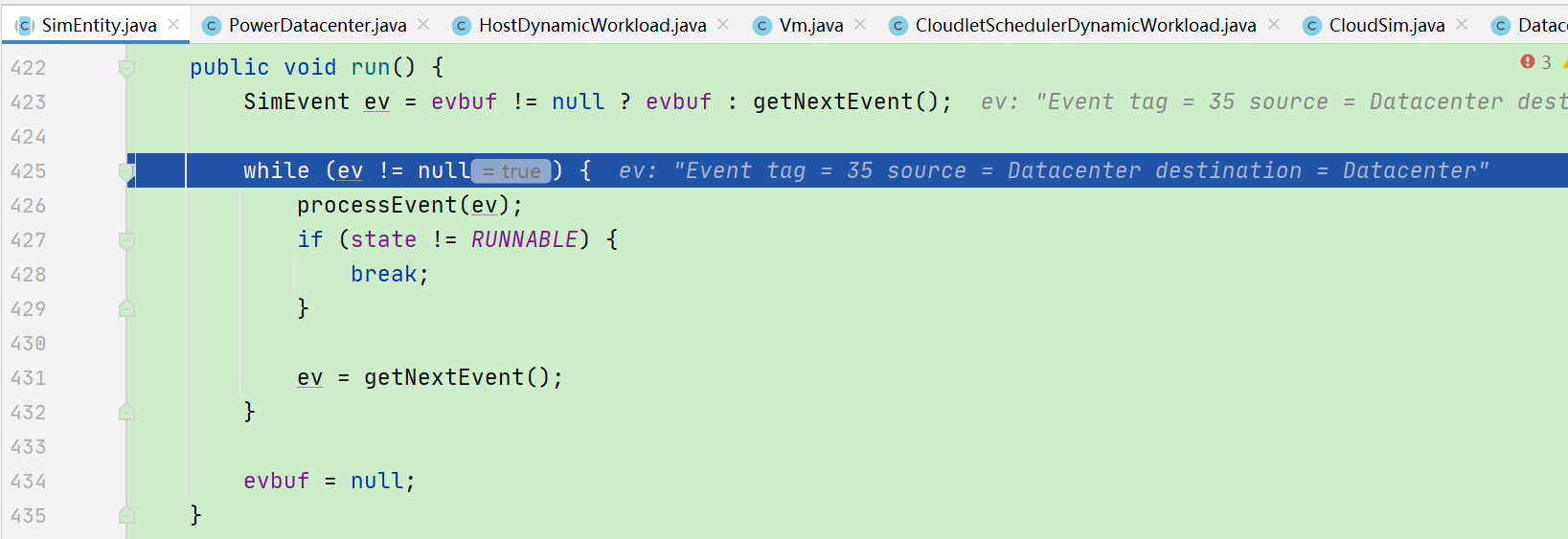


图3-21

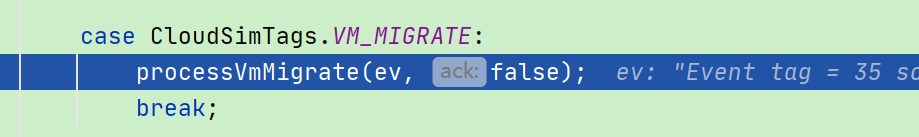


图3-22

跟前述过程一样。

当所有虚拟机迁移事件处理完毕后(经过若干轮runclocktick)。

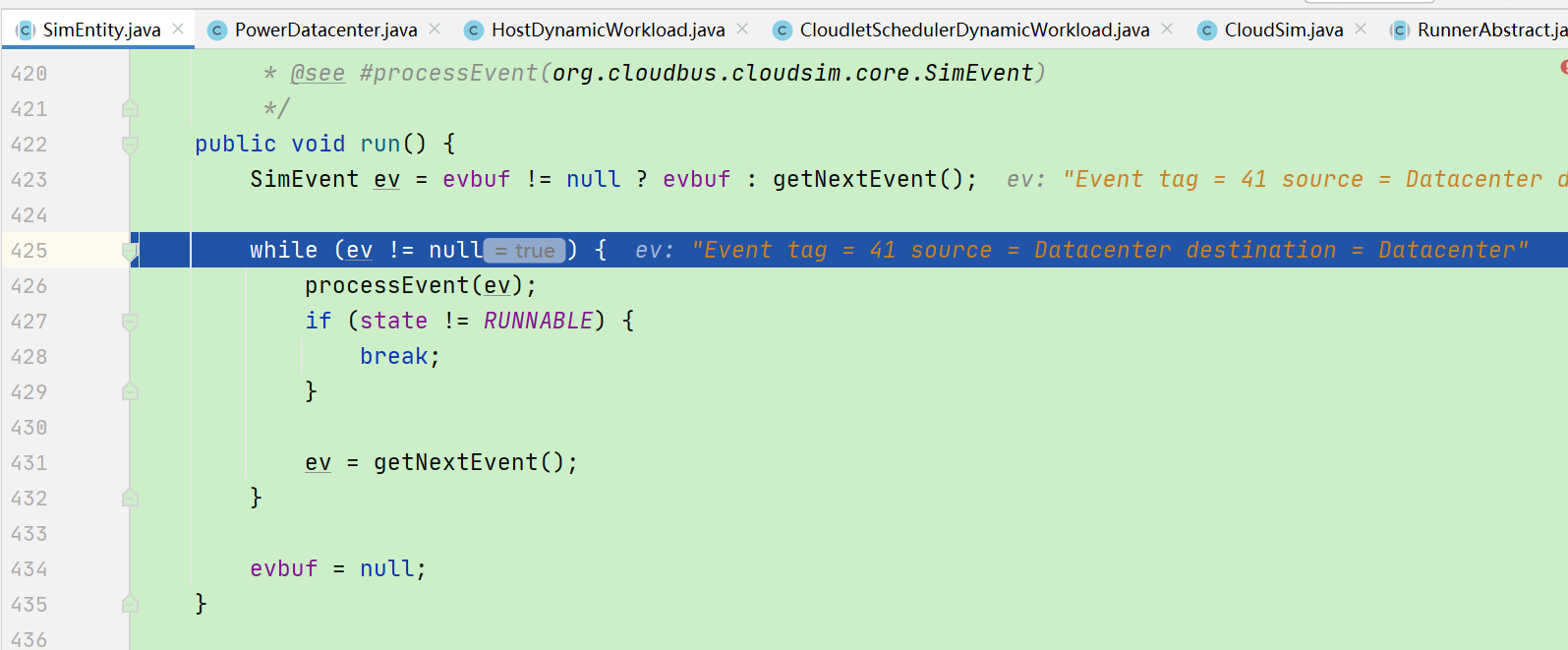


图3-23

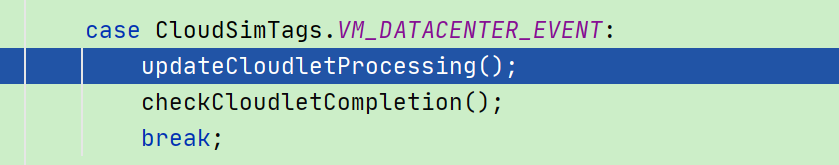


图3-24

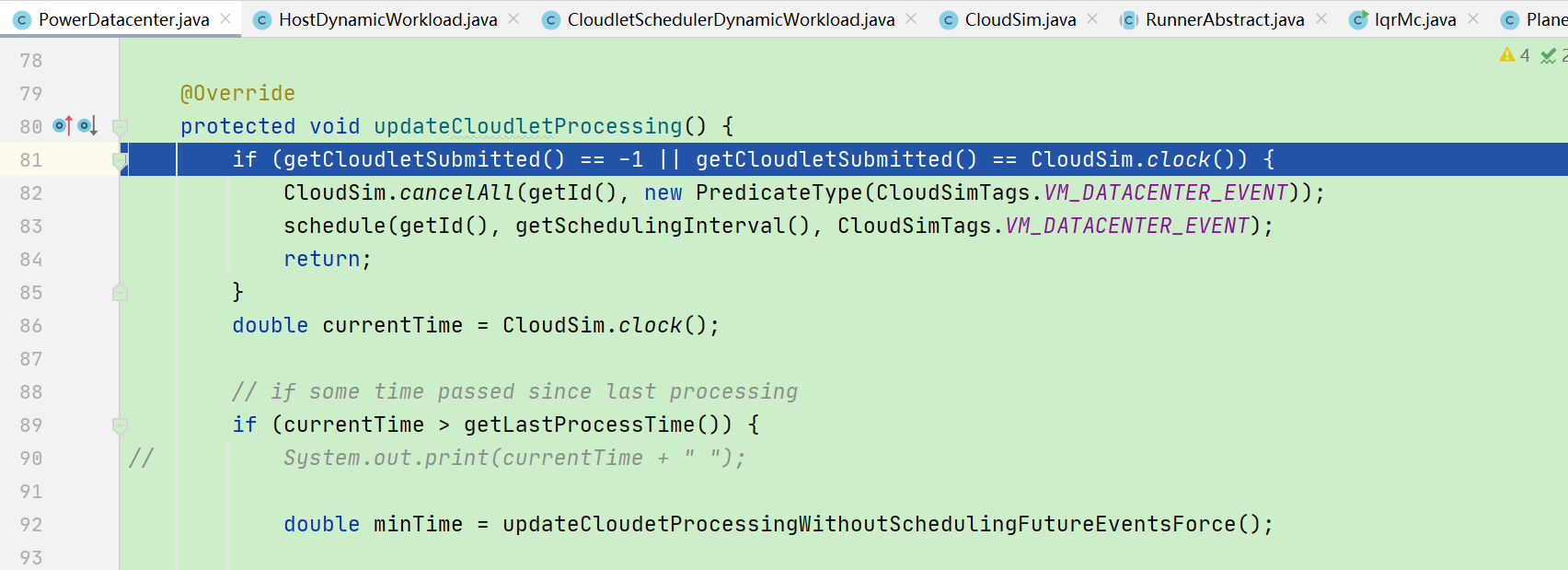


图3-25

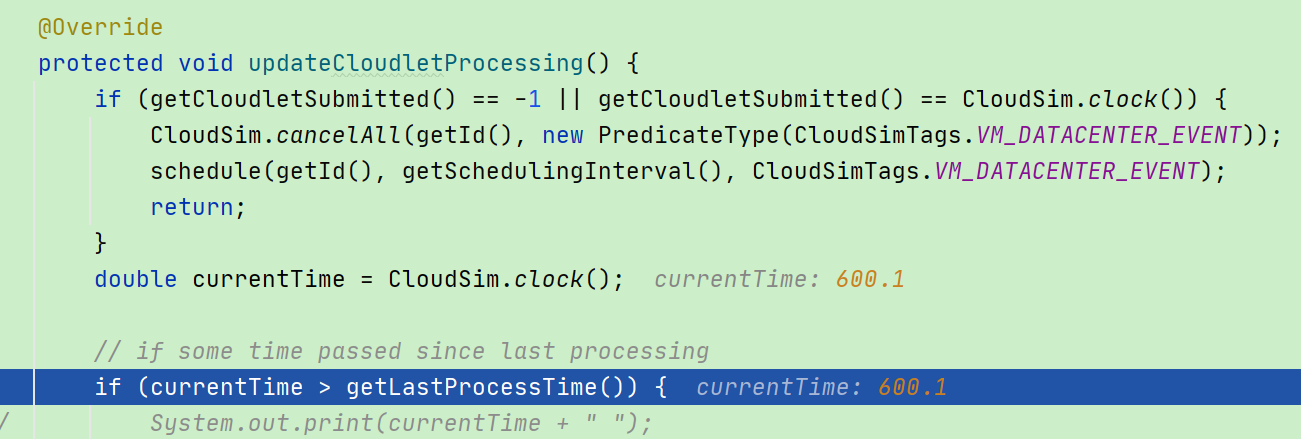


图3-26

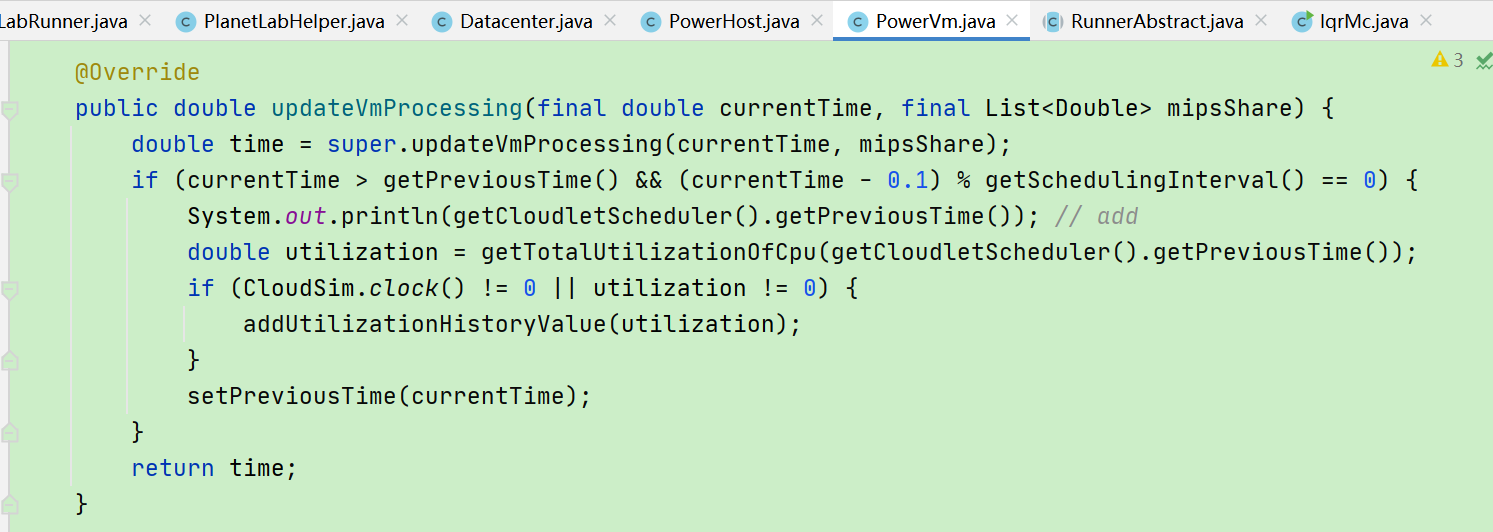
开始了新的循环处理。

## 4. 云任务利用率分析

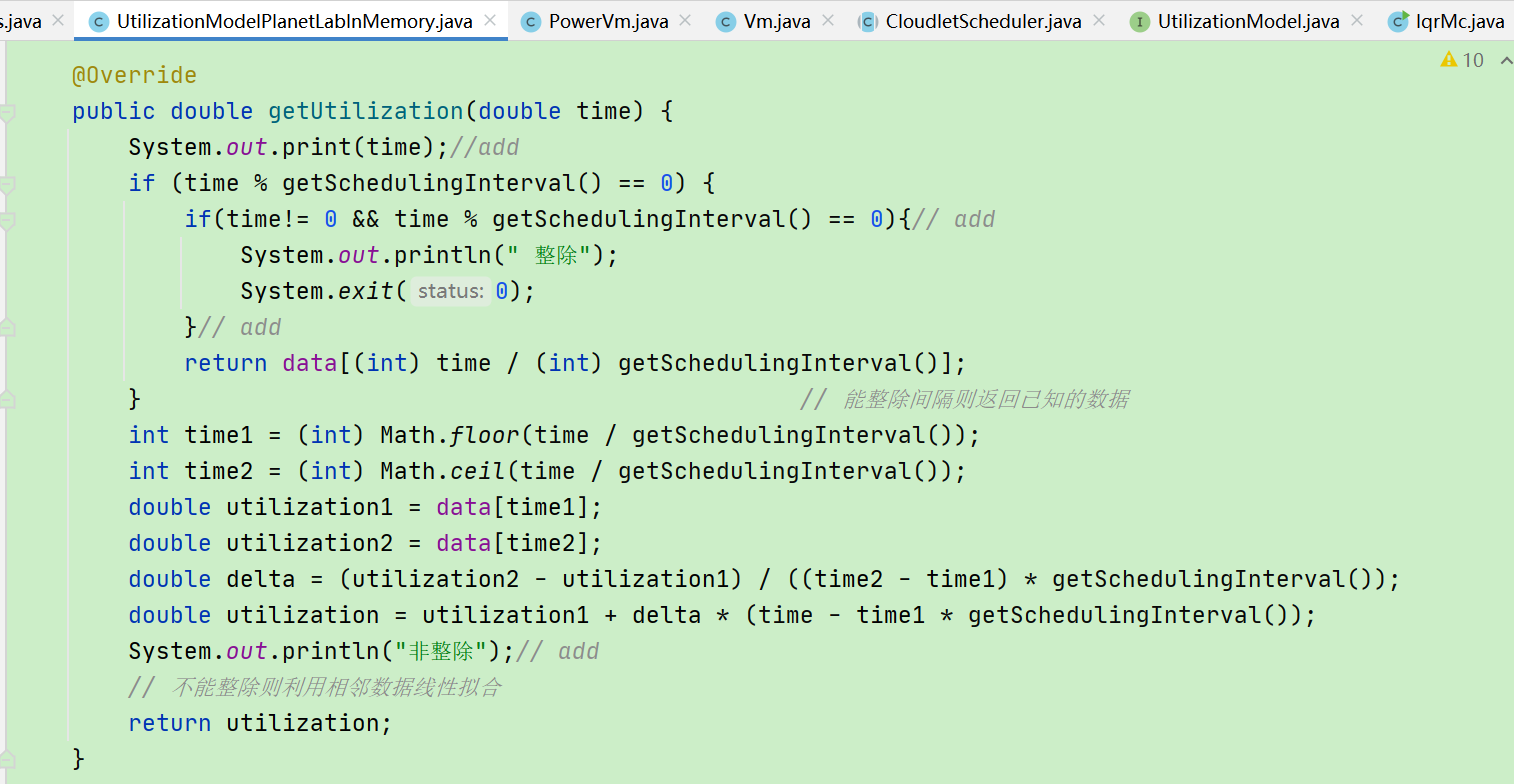
CloudSim能耗仿真实验数据集来自PlanetLab真实数据集。负载数据的采样频率为5分钟（即12次/小时），因此单日（24小时）的负载文件包含288个时序数据点，对应从00:00至23:55的完整周期。每个时隙interval（又称time slot）为5分钟（300s）。仿真开始后，数据中心需初始化。虚拟机整合从300.1秒起每隔300秒触发一次，共287次（300.1~86100.1，对应k∈[1,287]，时刻为k \* 300 + 0.1 (s)）。



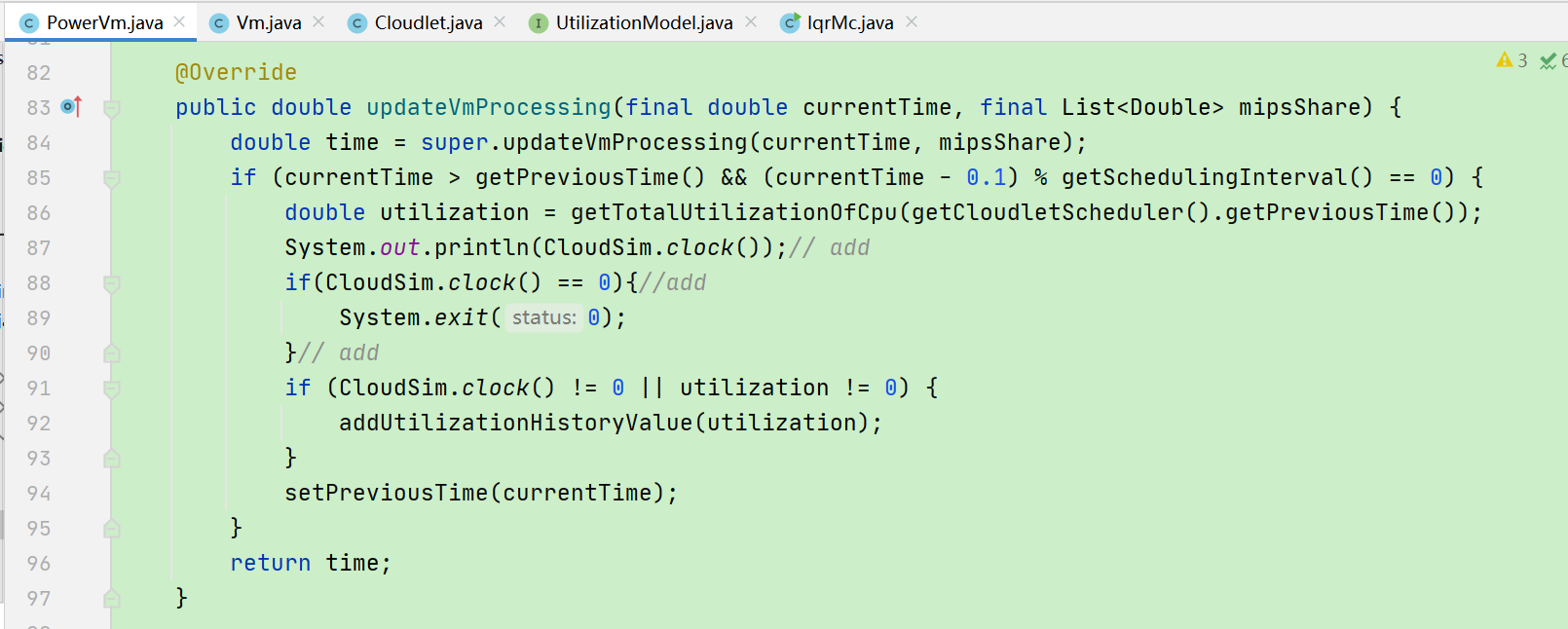
如上图所示，在初始化利用率模型函数中，data数组的长度为289。索引范围为0~287（第1~288个）的数据读取文件得到,索引为288（第289个）的数据与287相同。这样设置的目的是确保在仿真结束前的最后一次整合时刻（86100.1(s)，对应k=287），可以通过索引为287和288的两组相同的数据点进行线性插值，从而拟合86100.1s时刻的负载。



打印的范围为 300.1~86100.1,说明PowerVm没有读取索引为0的(第1个)数据。因此其读取的数据为索引为1~287的数据。



经过验证后，发现当time > 0时，获取的都不是整除数据，都是线性拟合后的数据。PowerVm.updateVmProcessing函数会调用getUtilization函数。以上说明，PowerVm读取的数据并不是planetlab数据集中的原始数据，而是线性拟合后得到的数据。



经过验证,当进入到if的大循环后，CloudSim.Clock的值范围为300.1~86100.1。那么 addUtilizationHistoryValue函数必然执行。因此，实际的利用率数据都存储在PowerVm的utilizationHistory中。

当然，由于0.1相对于interval=300而言数值太小，仅为300的1/3000，因此线性拟合后得到的利用率与原始数据集的利用率差别不大，影响几乎可以忽略。