# 动态负载均衡系统设计总览

## 系统总体设计

### 动态负载均衡系统需求

基于Hyper-V的动态负载均衡系统是一款服务于多域PC系统的优化插件。动态负载均衡系统可以在虚拟机开机时分配更多资源以加速开机过程，在开机后回退到正常状态。在使用过程中，动态负载均衡系统会分析虚拟机的CPU和内存压力，动态调节虚拟机的CPU和内存资源，以达到动态负载均衡的目的。

同时，为满足用户对虚拟机个性化的需求，用户可以在Host配置虚拟机的黑名单和白名单进程，黑名单进程可以包含占用资源的不必要的服务，垃圾软件等，白名单进程可以包含关键业务对应的进程名。域内监测所有进程的详细信息，若发现黑名单进程出现且满足一定条件，则杀死该黑名单进程。若发现白名单进程出现且满足条件，则通知负载均衡系统，提高虚拟机的资源分配等级，有先保证运行关键业务虚拟机的性能。

### 动态负载均衡系统总览

基于Hyper-V的动态负载均衡系统为C/S架构，由两大部分构成，分别是运行于Host的负载均衡器（Load\_Balancer\_Server）和运行于VM的域内性能监测器（Perf\_Detector\_Client），域内性能监测器为客户端，Host资源动态分配器为服务端，二者通过基于共享内存的Hyper-V socket进行通信。此外，多域PC系统操作各虚拟机，可能有开、关机，切入等使用操作，和删除域、创建域等配置操作，负载均衡器需要及时得到各个虚拟机实时状态，因此多域PC需要虚拟机状态传输器(SetVMStatus)，在虚拟机状态改变时，通知负载均衡器系统总览图如下图1.1.1所示：

虚拟机状态变化

传输虚拟机状态

互联网域2

互联网域1

本地域

多域PC系统

负载均衡器

（Load\_Balancer\_Server）

虚拟机状态传输器

(SetVMStatus)

域内性能信息

域内性能探测器

(Perf\_Detector\_

Client)

域内性能探测器

(Perf\_Detector\_

Client)

域内性能探测器

(Perf\_Detector\_

Client)

动态资源调节

图1,1,1 动态负载均衡系统总览图

域内性能监测器实时运行在开启状态的虚拟机，实时监测域内的性能指标，包括性能计数器中CPU、内存相关指标，更细粒度监测进程列表以及各进程占用的资源。每个开启的虚拟机中的性能监测器通过基于共享内存的HvSocket与Host通信，将性能信息发送给Host端的资源动态均衡器。资源动态均衡器可以直接监测各虚拟机的CPU占用率，但在静态内存分配的情况下，Host无法获取域内实际内存占用情况，更无法获取域内进程相关信息，因此资源动态分配器需要接收域内性能监测器发来的性能相关信息。资源动态均衡器根据各虚拟机的性能指标，以及Host的剩余资源情况，选择为资源紧张或空闲的虚拟机进行调度，回收空闲虚拟机已分配的内存和CPU资源，为负载高的虚拟机分配内存和CPU资源，可以提高负载高虚拟机的性能表现。

首先要解决的问题是：监测域内哪些指标能较为准确地反映虚拟机的负载情况，域内监测器不仅要监测CPU占用率和内存占用率，还需要监测反映资源需求情况的指标如线程等待队列长度Processor Queue Length、每秒产生的页错误次数Page Faults/secs。进一步地问题是，Host动态均衡器获取了所有性能指标，并进行分析后，应如何进行动态资源调度，Hyper-V提供了内存热插拔机制，在静态内存分配情况下，开机状态下可以在一定范围内调节静态分配给虚拟机的内存大小。

Host动态均衡器通过设置开启虚拟机的内存大小来调度各虚拟机的内存资源。至于CPU资源，可以通过设置开启后虚拟机的“CPU限制”、“CPU保留”参数设置虚拟机实际可占用的CPU资源。

给某虚拟机分配2个物理CPU，虚拟机中又两个vCPU，如果“CPU限制”为100，则该虚拟机最多占用全部2个物理CPU，将“CPU限制”改为500，则限制该虚拟机最多占用50%的CPU资源，即1个物理CPU。类似地，“CPU保留”指定了虚拟机最低保留的CPU资源比例，该比例需小于“CPU限制”。综上所述，可以有效动态调度CPU和内存资源。

## 系统模块设计

#### 域内性能监测器

域内性能监视器运行于开启的虚拟机，开机自启并定时检测域内内存、CPU性能相关的程序计数器，将收集到的内存、CPU相关性能信息发送到Host负载均衡器，为支持进程粒度的性能分析，以及供负载均衡器进行性能分析。

域内性能检测器按功能分为如下模块，各模块功能如下：

1. 性能计数器接口模块，该模块封装了实时获取部分Windows系统性能计数器指标的API，供其他模块调用；
2. 内存探测模块，通过调用性能计数器接口模块相关API，定时获取实时内存性能相关指标，如：内存占用率，页错误频率等；
3. CPU探测模块，通过调用性能计数器接口模块相关API，定时获取实时CPU性能相关指标，如：CPU占用率，Process Queue Length等；
4. 进程探测控制模块，进程探测功能：定时获取进程详细信息，包括占用的内存大小等，并将结果重定向到文件中。进程控制功能：读取用户设置的进程白名单和黑名单，若发现黑名单进程出现或占用大量资源，Host传输一个进程控制文件(json格式)到域内。此时，进程控制模块监测到该文件并杀死黑名单进程
5. 性能信息发送模块，建立与Host负载均衡器基于共享内存的hvSocket，定义性能消息传输的接口，通过hvSocket定时发送性能信息；
6. 主控模块，调用CPU、内存、进程信息探测模块定时获取综合的性能信息，定时将性能信息通过性能信息发送模块发送到Host负载均衡器。

各模块之间结构如下图1.2.1所示：

告知白名单进程启动，

获取进程详细信息

杀死必要的黑名单进程

获得并刷新性能信息

调用1模块获取性能指标

1. 性能计数器接口模块

2.内存探测模块

5.性能信息发送模块

3.CPU探测模块

4.进程探测控制模块

6.主控模块

域内性能监测器

Host负载均衡器

操作系统

（进程信息）

域内性能信息

性能信息接收模块

图1.2.1 域内性能监测器架构图

各模块的输入输出，如下表# 所示：

|  |  |  |
| --- | --- | --- |
| **模块名称** | **输入** | **输出** |
| 性能计数器接口模块 | 性能指标 | 所选性能指标的值 |
| 内存探测模块 | 无 | 虚拟机内存性能信息 |
| CPU探测模块 | 无 | 虚拟机CPU性能信息 |
| 进程探测控制模块 | ProcConfig.json文件 | 需要性能信息发送模块发送的消息 |
| 性能信息发送模块 | 需要传输的CPU、内存性能信息 | 性能消息发送结果 |
| 主控模块 | 各个性能探测模块获取的信息 | 无 |

#### 负载均衡器

负载均衡器运行在Host物理机，通过两种方式：Host获取虚拟机性能信息和接收各个虚拟机发来的性能信息，综合分析各虚拟机的性能表现，根据调度策略分配、回收虚拟机资源，以达到负载均衡的效果。

负载均衡器分为如下几个模块：

1. 配置文件读写模块，本系统通过一个配置文件记录用户的设置信息，该模块可以读取配置文件，并初始化相应的数据结构，该模块也可以写配置文件某些字段；
2. 系统信息获取模块，该模块获取物理主机的相关信息，包括硬件配置和物理机性能信息，如CPU核心数、物理内存大小、剩余内存大小等；
3. 虚拟机操作模块。该模块定义虚拟机的实例以及管理虚拟机的方法，包括对虚拟机实例进行抽象表示，获取Host可以得到的虚拟机相关信息，并定义修改虚拟机设置的通用接口等；
4. 动态调节接口模块，本模块封装了对CPU、内存资源进行动态调节的接口，如修改内存大小、设置CPU资源限制、CPU资源保留的接口，该模块需调用虚拟机管理模块，使负载均衡器能够调节开启状态的虚拟机资源；
5. 虚拟机状态监控模块，本模块接收虚拟机状态传输工具发来的消息，修改虚拟机的状态如打开、关闭、切入，以及新建虚拟机、删除虚拟机等操作，及时更新各虚拟机的状态信息如：是否安装、是否开机、记录当前切入的虚拟机等；
6. 性能接收模块，该模块接收虚拟机的连接请求，并建立基于共享内存的hvSocket连接，定义接收虚拟机性能信息传输的接口；
7. HyperV性能监测模块，本模块监测Windows中性能计数器中有关Hyper-V性能的部分，如虚拟机的当前内存压力、平均内存压力等；
8. 负载分析均衡模块，负载分析功能：该模块综合性能接收模块获取的各虚拟机性能信息，系统信息获取模块得到的主机信息，以及HyperV性能监测模块获取的信息，分析CPU和内存性能。负载均衡功能：根据设定的负载均衡策略，调用动态调节模块，分别对虚拟机的CPU和内存进行动态调节；
9. 单元测试模块，该模块设计了其他子模块的测试用例，根据测试需要对不同模块进行测试；
10. 主控模块，该模块调用性能接收模块，定时刷新虚拟机的性能信息，作为输入调用系统负载分析模块，启动整个系统流程。

各模块之间的关系如下图1.2.2所示：

获得物理机信息

域内性能监测器

更新虚拟机状态信息

多域PC开、关机，切入域，或安装、卸载域

调用模块4调节虚拟机资源

获取域内发来的性能信息

获得虚拟机开关机、切入状态

1. 配置文件读写模块

2. 系统信息获取模块

5. 虚拟机开关机状态监控模块

7. HyperV性能监测模块

3.虚拟机管理模块

8.负载分析平衡模块

Host负载均衡器

6.性能接收模块

4.动态资源调节接口模块

10.主控模块

虚拟机状态传输工具

(SetVMState.exe)

性能信息发送模块

域内性能信息

多域PC系统

图1.2.2 Host负载均衡器架构图

各模块的输入输出，如下表# 所示：

|  |  |  |
| --- | --- | --- |
| **模块名称** | **输入** | **输出** |
| 配置文件读写模块 | 配置文件路径 | 与系统配置相关的数据结构 |
| 系统信息获取模块 | 无 | 物理机相关信息 |
| 虚拟机管理模块 | 虚拟机名 | 虚拟机管理类实例 |
| 动态调节接口模块 | 待调节的选项名，以及对应的值 | 动态调节的结果 |
| 性能接收模块 | 域内传输来的消息字符串 | 虚拟机性能类VMPerf实例 |
| 负载分析模块 | 虚拟机管理类实例，Hyper | 是否杀死黑名单进程 |
| 单元测试模块 | 各个性能探测模块获取的信息 | 无 |
| 主控模块 |  |  |

#### 虚拟机状态传输工具

负载均衡器需要及时获得虚拟机的开机、关机、切入等运行状态信息，以及各域是否安全的配置信息，用户在多域PC中操作虚拟机更改上述状态。因为负载均衡系统作为一个插件，独立于多域PC系统。因此需要虚拟机状态传输工具(SetVMStatus.exe)，在多域PC更改虚拟机状态时，及时将虚拟机状态的信息发送给负载均衡器。

多域PC系统和负载均衡器均运行在Host，二者通过共享内存的方式进行通信：多域PC调用虚拟机状态传输工具(SetVMStatus.exe),传入参数，SetVMStatus.exe通过共享内存的方式发送信息，到负载均衡器的虚拟机开关机状态监控模块，使负载均衡器及时更新各虚拟机状态。

多域PC为支持本系统，只需要在域开、关机，切入，安装，卸载后插桩调用SetVMStatus.exe的代码即可。

如”.\SetVMStatus.exe VMStatus LocalVM GetIn”命令，通知负载均衡器：切入本地域。

#### 虚拟机性能信息动态库

该工程为一个DLL，域内性能探测器和Host负载均衡器的消息传输，需要对消息进行序列化和反序列化，以完成性能信息类的传输，该类的定义需要被域内性能探测器和Host负载均衡器共享，因此虚拟机性能信息动态库定义了虚拟机性能消息类VMPerf class，提供给域内性能监测器和Host负载均衡器。

# 域内性能监测器

### 性能计数器接口模块

#### 功能

本模块提供获取Windows性能计数器部分内存、CPU性能相关指标的API。

#### 设计

#### 异常和错误处理

#### 相关数据结构

1. Windows性能计数器接口类 WinPerfCounter

数据类型：class WinPerfCounter

功能：提供Windows性能计数器获取部分指标的方法，并描述性能计数器获取的性能指标。

详细数据结构：

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| CPUProcessorTime | CPU占用率 | float |
| CPUPrivilegedTime | CPU特权态时间 | float |
| CPUInterruptTime | CPU中断时间 | float |
| CPUDPCTime | CPU DPC状态时间 | float |
| MEMAvailable | 可用内存大小，单位字节 | float |
| MEMCommited |  | float |
| MEMCommitLimit |  | float |
| MEMCommitedPerc |  | float |
| MEMPoolPaged |  | float |
| MEMPoolNonPaged |  | float |
| MEMCached |  | float |
| PageFile |  | float |
| ProcessorQueueLengh | 等待分配CPU的线程数 | float |
| HANDLECountCounter |  | float |
| THREADCount | 线程数量 | float |
| CONTENTSwitches | 每秒产生的上下文切换次数 | int |
| SYSTEMCalls | 每秒产生的系统调用 | int |
| PagesPerSec |  | float |
| PageFaultsPerSec | 每秒产生的 | float |
| PageReadsPerSec |  | float |
| PageWritesPerSec |  | float |
| PagesInputPerSec |  | float |
| PagesOutputPerSec |  | float |

#### 函数及流程图

该部分函数均非常短，为了获取上述数据结构中的指标，每个指标对应一个函数，逻辑简单，因此流程图略。

【包含的方法】

|  |  |
| --- | --- |
| 函数签名 | 作用 |
| public float getProcessorCpuTime() | 查询指标：ProcessorCpuTime |
| public float getCpuPrivilegedTime() | 查询指标：CpuPrivilegedTime |
| public float getCpuinterruptTime() | 查询指标：CPUInterruptTime |
| public float getcpuDPCTime() | 查询指标：CPUDPCTime |
| public float getPageFile() | 查询指标：PageFile |
| public float getProcessorQueueLengh() | 查询指标：ProcessorQueueLengh |
| public float getMemAvailable() | 查询指标：MemAvailable |
| public float getMemCommited() | 查询指标：MemCommited |
| public float getMemCommitLimit() | 查询指标：MemCommitLimit |
| public float getMemCommitedPerc() | 查询指标：MemCommitedPerc |
| public float getMemPoolPaged() | 查询指标：MemPoolPaged |
| public float getMemPoolNonPaged() | 查询指标：MemPoolNonPaged |
| public float getMemCachedBytes() | 查询指标：MemCachedBytes |
| public float getHandleCountCounter() | 查询指标：HANDLECountCounter |
| public float getThreadCount() | 查询指标：THREADCount |
| public int getContentSwitches() | 查询指标：CONTENTSwitches |
| public int getsystemCalls() | 查询指标：SYSTEMCalls |
| public float getPageFaultsPerSec() | 查询指标：PageFaultsPerSec |
| public float getPageReadsPerSec() | 查询指标：PageReadsPerSec |
| public float getPageWritesPerSec() | 查询指标：PageWritesPerSec |
| public float getPageInputsPerSec() | 查询指标：PageInputsPerSec |
| public float getPageOutputsPerSec() | 查询指标：PageOutputsPerSec |
| public float getPagesPerSec() | 查询指标：PagesPerSec |

### CPU探测模块

#### 功能

本模块定时获取并刷新CPU性能相关信息，如：CPU占用率，Process Queue Length等。该模块对应代码中的CpuSpoofer.cs文件。

#### 设计

#### 异常和错误处理

#### 相关数据结构

1. CPU探测类 CpuSpoofer

【数据类型】

class

【功能】

描述收集到的CPU性能信息，并提供定时刷新CPU性能信息的方法。

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| CPUCount | CPU核心数 | int |
| ProcessorQueueLength | 表示host管理器的授权码 | float |
| CPUProcessorTime | CPU占用率 | float |
| CPUPrivilegedTime |  | float |
| CPUInterruptTime | 表示时间戳 | float |
| CPUDPCTime |  | float |
| winPerfCounter |  | WinPerfCounter |
|  |  | Timer |

#### 函数及流程图

CpuSpoofer()

【所属类及所在文件】

CpuSpoofer类，CpuSpoofe.cs文件

【功能描述】

CpuSpoofer类构造函数。

【输入参数】

无

【返回值】

无

【流程图】

开始

WinPerfCounter.initAllCounterValue();

初始化性能计数器的初始值

以RefreshStateByTime函数新建一个线程并启动

结束

否

**RefreshStatesByTime()**

【所属类及所在文件】

CpuSpoofer类，CpuSpoofe.cs文件

【功能描述】

启动一个定时器，以一定时间间隔，获取并刷新CPU性能相关参数。

【输入参数】

无

【返回值】

无

【流程图】

开始

RUtimer = new System.Timers.Timer(100);

创建一个100ms定时的定时器

RUtimer.Elapsed += RefreshCPUArg;

定时时间到，处理函数为RefreshCPUArg;

RUtimer.AutoReset = true;

RUtimer.Enabled = true;

RUtimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

结束

否

是

**RefreshCPUArg(object sender, ElapsedEventArgs e)**

【所属类及所在文件】

CpuSpoofer类，CpuSpoofe.cs文件

【功能描述】

获取并刷新CpuSpoofer类中CPU性能相关参数。

【输入参数】

object sender:表示触发事件的控件对象，EventArgs e：表示事件数据的类的基类

【返回值】

无

【流程图】

开始

CPUProcessorTime = winPerfCounter.getProcessorCpuTime();

ProcessorQueueLength = winPerfCounter.getProcessorQueueLengh();

CPUPrivilegedTime = winPerfCounter.getCpuPrivilegedTime();

CPUInterruptTime = winPerfCounter.getThreadCount();

调用性能计数器接口模块获取上述参数

结束

否

是否捕获异常

打印异常信息

是

否

### 内存探测模块

#### 功能

本模块定时获取并刷新内存性能相关信息，如：可用内存大小，页错误频率等。

#### 设计

#### 相关数据结构

1. 内存探测类 MemSpoofer

【数据类型】

class

【功能】

描述收集到的内存性能信息，并提供定时刷新内存性能信息的方法。

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| MEMAvailable | CPU核心数 | float |
| MEMCommited | 表示host管理器的授权码 | float |
| MEMCommitLimit | CPU占用率 | float |
| MEMCommitedPerc |  | float |
| MEMPoolPaged | 表示时间戳 | float |
| MEMPoolNonPaged |  | float |
| MEMCached |  | float |
| PageFile |  | float |
| PagesPerSec |  | float |
| PageFaultsPerSec |  | float |
| PageReadsPerSec |  | float |
| PageWritesPerSec |  | float |
| PagesInputPerSec |  | float |
| PagesOutputPerSec |  | float |
| MEMPerfCounter |  | WinPerfCounter |
| RUtimer |  | Timer |

#### 函数及流程图

* MemSpoofer ()

【所属类及所在文件】

MemSpoofer类，MemSpoofe.cs文件

【功能描述】

MemSpoofer类构造函数。

【输入参数】

无

【返回值】

无

【流程图】

开始

MEMPerfCounter.initAllCounterValue();

初始化性能计数器的初始值

以RefreshStateByTime函数新建一个线程并启动

结束

否

**RefreshMemStatesByTime()**

【所属类及所在文件】

MemSpoofer类，MemSpoofe.cs文件

【功能描述】

启动一个定时器，以一定时间间隔，获取并刷新内存性能相关参数。

【输入参数】

无

【返回值】

无

【流程图】

开始

RUtimer = new System.Timers.Timer(100);

创建一个100ms定时的定时器

RUtimer.Elapsed += RefreshMEMArg;

定时时间到，处理函数为RefreshMEMArg;

RUtimer.AutoReset = true;

RUtimer.Enabled = true;

RUtimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

结束

否

是

**RefreshMEMArg(object sender, ElapsedEventArgs e)**

【所属类及所在文件】

MemSpoofer类，MemSpoofe.cs文件

【功能描述】

获取并刷新MemSpoofer类中内存性能相关参数。

【输入参数】

object sender:表示触发事件的对象，EventArgs e：表示事件数据的类的基类

【返回值】

无

【流程图】

开始

MEMAvailable = MEMPerfCounter.getMemAvailable();

MEMCommited = MEMPerfCounter.getMemCommited();

MEMCommitLimit = MEMPerfCounter.getMemCommitLimit();

MEMCommitedPerc = MEMPerfCounter.getMemCommitedPerc();

……

调用性能计数器接口模块获取上述参数

结束

否

是否捕获异常

打印异常信息

否

是

### 进程监测控制模块

#### 功能

本模块提供域内进程详细信息的方法，并读取用户的进程控制文件，监控黑名单进程，若黑名单进程出现且满足一定条件，则杀死黑名单进程。

#### 设计

1. 开启一个线程，该线程定时调用监测函数，监测是否有黑名单进程出现；

#### 异常和错误处理

1. 用户输入远程服务器IP，并点击确认连接后，本线程会首先尝试ping远程服务器，若无法正常连接，则提示用户重新检查服务器IP，并测试网络；

#### 相关数据结构

* ProcConfig文件

【文件类型】

json

【功能】

记录用户配置的黑名单和白名单信息

【详细结构】

示例如下：

{

"BlackList":

// 列表，列举所有黑名单进程

[

{

// 进程名

"name": "cmd",

// 黑名单进程不能超过memLimit大小的内存

"memLimit": 0

},

{

"name": "chrome",

"memLimit": 100

}

]

“WhiteList":

// 列表，列举所有白名单进程

[

{

"name": "MirosoftEdge",

"memLimit": 200

}

]

}

* ProcessInfo类

【数据类型】

class

【功能】

表示单个进程对应的信息

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| procID | 进程ID | int |
| procName | 进程名 | string |
| memSize | 进程占用的虚拟内存大小 | long |

【包含的方法】

|  |  |
| --- | --- |
| 函数签名 | 作用 |
| public ProcessInfo(int id, string name, long memorySize) | 构造函数，根据输入初始化类实例 |

* ProcessControler类

【数据类型】

class

【功能】

进程控制类，包含了获取所有进程信息，监测黑名单进程的方法

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| ProcConfigPath | ProcConfig.json配置文件路径 | string |
| BlackProcDict | 读取的黑名单进程字典，key为进程名，value为限制内存大小 | Dictionary<string, Int64> |
| WhiteProcDict | 读取的白名单进程字典，key为进程名，value为规定的内存大小 | Dictionary<string, Int64> |
| RUtimer | 定时器对象 | Timers.Timer |
| CheckingTimeSpan | 监测进程的时间间隔 | int |

【包含的方法】

|  |  |
| --- | --- |
| 函数签名 | 作用 |
| public ProcessControler(string procConfigPath, int checkingTimeSpan) | 构造函数，输入的procConfigPath初始化ProcConfigPath；checkingTimeSpan初始化  CheckingTimeSpan |
| public List<ProcessInfo> GetProcInfoList() | 遍历收集系统中所有进程，返回List<ProcessInfo>列表 |
| public bool GetBlackListFromConfig() | 读取ProcConfig.json配置文件，初始化BlackProcDict |
| public bool GetWhiteListFromConfig () | 读取ProcConfig.json配置文件，初始化WhiteProcDict |
| public void TryKillBlackProcs(List<ProcessInfo> procList) | 读取ProcInfo列表procList，判断是否有黑名单进程并根据条件杀死 |
| public void CheckProcess(object sender, ElapsedEventArgs e) | CheckProcessByTime的回调函数，该函数先调用GetProcInfoList()获得进程信息，再调用TryKillBlackProcs()监测黑名单进程 |
| public void CheckProcessByTime() | 设置定时器，定时调用CheckProcess()函数 |
| public void StartUpCheckingProcess() | 以CheckProcessByTime()函数启动一个线程 |

#### 函数及流程图

* GetProcInfoList()

【所属类及所在文件】

ProcessControler类，ProcessControler.cs文件

【功能描述】

获得所有进程的信息，返回ProcessInfo的列表。

【输入参数】

无

【返回值】

List<ProcessInfo> ，ProcessInfo类的列表

【流程图】

开始

返回procInfoList

List<ProcessInfo> procInfoList = new List<ProcessInfo>();

Process[] ps = Process.GetProcesses();

初始化返回的ProcessInfo列表，调用Process.GetProcesses()获得所有进程数组ps

procInfoList.Add(new ProcessInfo(p.Id, p.ProcessName, p.VirtualMemorySize64));;

获取进程p的进程ID，进程名和占用的虚拟机内存，初始化ProcessInfo类实例，添加到列表中

遍历ps中每一个进程p，是否遍历完

输出异常信息;

是否出现异常

否

是

是

否

* CheckProcess(object sender, ElapsedEventArgs e)

【所属类及所在文件】

ProcessControler类，ProcessControler.cs文件

【功能描述】

先调用GetProcInfoList()函数获取进程列表，然后调用TryKillBlackProcs函数尝试杀死黑名单进程。

【输入参数】

object sender:表示触发事件的对象，EventArgs e：表示事件数据的类的基类

【返回值】

无

【流程图】

开始

List<ProcessInfo> processInfoList = GetProcInfoList();

获取当前所有进程信息

TryKillBlackProcs(processInfoList);

监测黑名单进程

返回

否

* CheckProcessByTime()

【所属类及所在文件】

ProcessControler类，ProcessControler.cs文件

【功能描述】

启动一个定时器，以一定时间间隔，调用CheckProcess函数。

【输入参数】

无

【返回值】

无

【流程图】

开始

RUtimer = new System.Timers.Timer(CheckingTimeSpan);

创建一个CheckingTimeSpan定时的定时器

RUtimer.Elapsed += CheckProcess;

定时时间到，处理函数为CheckProcess;

RUtimer.AutoReset = true;

RUtimer.Enabled = true;

RUtimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

返回

否

是

ProcConfigPath文件是否存在

Thread.Sleep(10000);睡眠10s

bool ret = GetBlackListFromConfig();读取配置文件

ret

输出：读取配置文件失败，检查json格式

否

否

是

是

* StartUpCheckingProcess()

【所属类及所在文件】

ProcessControler类，ProcessControler.cs文件

【功能描述】

以CheckProcessByTime函数启动一个线程，定时监测黑名单进程。

【输入参数】

无

【返回值】

无

【流程图】

开始

以CheckProcessByTime函数新建一个线程并启动

返回

否

### 性能信息发送模块

#### 功能

本模块整合CPU探测模块和内存探测模块收集的性能信息，通过基于共享内存的Hyper-V Socket发送给Host负载均衡器。

#### 设计

在C#的WCF模型中，客户端与服务端需要定义ServiceContract服务契约和OperationContract操作契约，ServiceContract作为服务端和客户端交互的接口，OperationContract约定的操作。

#### 相关数据结构

1. PerformanceInfo类

【数据类型】

class

【功能】

收集内存探测模块和CPU探测模块的性能信息

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| perf\_Transfer | 与Host传输的性能信息 | VMPerf class |
| VMName | 虚拟机名 | string |
| memSpoofer | 内存性能探测器实例 | MemSpoofer class |
| cpuSpoofer | CPU性能探测器实例 | CpuSpoofer class |
| processInfo | 进程信息监测类实例 | ProcessInfo class |
| RUtimer | 定时器 | Timers.Timer |

1. 接口IServer

【数据类型】

interface

【功能】

定义服务契约IServer，其中包含传输性能信息的操作契约函数TransferPerfStr(string perf\_Str)

【详细数据结构】：

[ServiceContract]

public interface IServer

{

[OperationContract]

VMPerf TransferPerfStr(string perf\_Str);

}

1. DetectorClient类

【数据类型】

class

【继承关系】

继承ClientBase<IServer>, IServer

【功能】

客户端代理类，继承ClientBase<>类，这是一个泛型类，接受服务契约IServer作为泛型参数。DetectorClient类通过基类的信道，调用操作契约函数TransferPerfStr。

【详细数据结构】

public class DetectorClient : ClientBase<IServer>, IServer

{

public DetectorClient(EndpointAddress addy)

: base(new HyperVNetBinding(), addy)

{

}

public VMPerf TransferPerfStr(string perf\_Str) => Channel.TransferPerfStr(perf\_Str);

}

1. Client类

【数据类型】

class

【功能】

根据唯一URI创建HvSocket连接连接Host，提供定时发送性能信息的接口

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| RUtimer | 定时器 | Timers.Timer |
| HvAddr | 客户端连接的地址 | string |
| clientPerfInfo | 客户端收集的性能信息 | PerformanceInfo class |
| detectorClient | 与Host通信客户端代理 | DetectorClient class |
| sendTimeGap | 发送信息的时间间隔，单位ms | int |

#### 函数及流程图

* PerformanceInfo()

【所属类及所在文件】

PerformanceInfo类，PerformanceInfo.cs文件

【功能描述】

PerformanceInfo类构造函数。

【输入参数】

无

【返回值】

无

【流程图】

是

否

开始

memSpoofer = new MemSpoofer();

cpuSpoofer = new CpuSpoofer();

结束

memSpoofer.RefreshMEMArg();

cpuSpoofer.RefreshCPUArg();

刷新memSpoofer和cpuSpoofer中的性能信息

bool ret = SetPerfTransfer(memSpoofer, cpuSpoofer);

调用SetPerfTransfer函数将性能信息写入perf\_Transfer字段中

bool vmIDRet = GetVMName(@"C:\VMID.txt");

读取VMName.txt文件获取虚拟机名

! vmIDRet

VMName = "Unknown";

* GetVMName(string vmIDFilePath)

【所属类及所在文件】

PerformanceInfo类，PerformanceInfo.cs文件

【功能描述】

读取配置文件，获取虚拟机名称。

【输入参数】

vmIDFilePath，配置文件路径

【返回值】

bool值，表示读取是否成功

【流程图】

开始

返回ret

VMName = File.ReadAllText(vmIDFilePath);

bool ret = true;

读取配置文件中vmname字段

是否产生异常

bool ret = false;

!File.Exists(vmIDFilePath)

否

否

是

是

* SetPerfTransfer(MemSpoofer memSpf, CpuSpoofer cpuSpf)

【所属类及所在文件】

PerformanceInfo类，PerformanceInfo.cs文件

【功能描述】

输入内存、CPU监视器实例，从中获取待传输的性能指标，并据此对perf\_Transfer字段进行复制。

【输入参数】

MemSpoofer memSpf, 内存监视器实例，CpuSpoofer cpuSpf，CPU监视器实例

【返回值】

bool值，表示设置是否成功

【流程图】

开始

返回ret

perf\_Transfer.VMName = VMName;

perf\_Transfer.CPUCount = cpuSpf.CPUCount;

perf\_Transfer.CPUProcessorTime = cpuSpf.CPUProcessorTime

……

根据cpuSpf中CPU性能指标设置perf\_Transfer对应指标

是否产生异常

ret = false;

perf\_Transfer.MEMAvailable = memSpf.MEMAvailable;

perf\_Transfer.PagesPerSec = memSpf.PagesPerSec;

……

根据memSpf中内存性能指标设置perf\_Transfer对应指标，bool ret = true

是

否

* Client(string Addr, int TimeGap = 100)

【所属类及所在文件】

Client类，Client.cs文件

【功能描述】

Client类构造函数。

【输入参数】

Addr，客户端连接的Host地址; TimeGap，性能信息发送的时间间隔，默认100ms

【返回值】

无

【流程图】

开始

clientPerfInfo = new PerformanceInfo();

初始化性能信息

HvAddr = Addr;

设置类变量HvAddr

sendTimeGap = TimeGap;

结束

否

* StartUpClient ()

【所属类及所在文件】

Client类，Client.cs文件

【功能描述】

初始化客户端代理detectorClient并启动，调用SendByTime方法定时发送性能信息。

【输入参数】

无

【返回值】

无

【流程图】

开始

detectorClient = new DetectorClient(new EndpointAddress(HvAddr));

初始化客户端代理detectorClient

detectorClient.Open();

启动客户端代理

SendByTime();;

调用SendByTime函数定时发送

结束

否

是

* SendByTime()

【所属类及所在文件】

Client类，Client.cs文件

【功能描述】

以sendTimeGap为定时间隔，定时调用SendPerfTransfer函数发送性能信息。

【输入参数】

无

【返回值】

无

【流程图】

开始

RUtimer = new System.Timers.Timer(sendTimeGap);

创建一个sendTimeGap定时的定时器

RUtimer.Elapsed += SendPerfTransfer;

定时时间到，处理函数为SendPerfTransfer;

RUtimer.AutoReset = true;

RUtimer.Enabled = true;

RUtimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

结束

否

是

* SendPerfTransfer(object sender, ElapsedEventArgs e)

【所属类及所在文件】

Client类，Client.cs文件

【功能描述】

获取内存、CPU性能信息，将类序列化为字符串后，客户端代理detectorClient调用TransferPerfStr函数通知服务端接收。

【输入参数】

object sender:表示触发事件的控件对象，EventArgs e：表示事件数据的类的基类

【返回值】

无

【流程图】

开始

clientPerfInfo.SetPerfTransfer(clientPerfInfo.memSpoofer, clientPerfInfo.cpuSpoofer);

调用SetPerfTransfer函数，将memSpoofer个cpuSpoofer中的性能信息写入clientPerfInfo.perf\_Transfer字段中

返回

否

是否捕获异常

打印异常信息

将VMPerf类型的clientPerfInfo.perf\_Transfer字段序列化为string perfTransferStrSerial，调用

detectorClient.TransferPerfStr(perfTransferStrSerial);

函数通知服务端接收

是

否

### 主控模块

#### 功能

主控模块设置定时时间，启动性能信息发送模块，定时向Host负载均衡器发送性能信息，同时启动进程探测控制模块，读取Host负载均衡器发送来的进程配置文件，监测进程信息，发现黑名单进程占用资源超过配置值时，进程探测控制模块杀死该进程。

#### 设计

#### 数据结构

C#控制台程序默认主函数为Program类，其中只包含Main函数。

#### 函数及流程图

* Main(string[] args)

【所属类及所在文件】

Program类，Program.cs文件

【功能描述】

程序main函数。

【输入参数】

string[] args，输入的命令行参数

【返回值】

无

【流程图】

开始

string HvAddr = "hypervnb://a42e7cda-d03f-480c-9cc2-a4de20abb878/C7240163-6E2B-4466-9E41-FF74E7F0DE47";地址前半部分表示物理机，后半部分表示注册表中注册的可信程序ID。

Client client = new Client(HvAddr，Timegap)

初始化客户端

结束

client.StartUpClient();

启动客户端，定时发送性能信息

是否捕获异常

打印异常消息，并输出客户端启动失败

输出客户端启动成功

否

是

ProcessControler processControler = new ProcessControler(@"C:\TEMP\ProcConfig.json", 5000);

初始化ProcessControler类示例processControler，设置监测时间间隔为5s。

processControler.StartUpCheckingProcess()

调用StartUpCheckingProcess函数启动进程监测

# Host负载均衡器

### 配置文件读取模块

#### 功能

本模块提供获取Windows性能计数器部分内存、CPU性能相关指标的API。

#### 设计

#### 异常和错误处理

#### 相关数据结构

* VMConfig类

【数据类型】

class

【功能】

虚拟机的配置信息，包括虚拟机名、预设定内存大小、CPU核心数等

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| VMName | 虚拟机名 | string |
| Installed | 是否安装 | bool |
| MemorySize | 预设内存大小 | UInt64 |
| CPUNum | 预设CPU核心数 | int |
| CPULimt | 预设CPU限制大小 | int |

* SysConfig类

【数据类型】

class

【功能】

负载均衡器的配置选项信息

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| BalanceStrategy | 均衡策略 | string |
| CPUDefaultCount | 系统默认CPU核心数 | int |
| MemoryDefaultSize | 系统默认内存大小 | UInt64 |
| CPUDefaultLimit | 系统默认的CPU核心数 | int |

* MpcVMInfo类

【数据类型】

class

【功能】

多域PC域管理器中，虚拟机配置文件对应的配置信息。

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| VMName | 虚拟机名 | string |
| Installed | 是否安装 | bool |
| VMPath | 虚拟机安装路径 | string |
| Description | 描述信息 | string |
| DefaultUser | 域内用户名 | string |
| Password | 域内密码 | string |
| IPAddress | IP地址 | string |
| StateSnapID | 当前使用快照ID | string |
| RenamePC | 是否重命名域内主机名 | bool |

* GetConfig类

【数据类型】

class

【功能】

解析配置文件，设置配置文件对应的内存数据结构。

【详细数据结构】

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 含义 | 类型 | 初始值 |
| ConfigFileRelPath | 存储配置文件的相对路径 | string | @"..\..\.." |
| VMConfigName | 虚拟机配置文件名 | const string | "VMConfig.json" |
| SysConfigName | 系统配置文件名 | const string | "SystemConfig.json" |
| ProcessConfigName | 进程配置信息文件名 | const string | "ProcessWhiteList.json" |
| VMconfigPath | 虚拟机配置文件相对路径 | string | ConfigFileRelPath |
| SysConfigPath | 系统配置文件相对路径 | string | 拼接ConfigFileRelPath与SysConfigName |
| ProcessConfigPath | 进程配置信息文件相对路径 | string | 拼接ConfigFileRelPath与ProcessConfigName |
| VMStateJsonPath | 当前使用快照ID | string | 拼接ConfigFileRelPath与VMConfigName |
| currentVMConfig | 当前解析到的虚拟机配置信息 | VMConfig | 无 |
| currentSysConfig | 当前解析到的系统配置信息 | SysConfig | 无 |
| currentMpcVMInfo | 当前解析到的多域PC虚拟机信息 | MpcVMInfo | 无 |

#### 函数及流程图

* GetVMConfig(string VMName)

【所属类及所在文件】

GetConfig类，GetConfig.cs文件

【功能描述】

解析虚拟机配置文件，根据配置文件初始化currentVMConfig变量

【输入参数】

VMName，虚拟机名

【返回值】

VMConfig类实例

【流程图】

否

是

开始

读取路径VMconfigPath下的虚拟机配置文件，初始化JsonTextReader类型的reader

JObject o = (JObject)JToken.ReadFrom(reader)

新建用于操作Json对象的JObject

返回cureentVMConfig

currentVMConfig.VMName = VMName;

JObject vmObj = ((JObject)o[VMName];

currentVMConfig.Installed = Convert.ToBoolean(vmObj["Installed"].ToString()

currentVMConfig.CPUNum = Convert.ToInt32(vmObj["CPUCount"].ToString());

currentVMConfig.MemorySize = Convert.ToUInt64(vmObj["MemorySize"].ToString())；

currentVMConfig.CPULimt = Convert.ToInt32(vmObj["CPULimit"].ToString());

读取Json对象并对currentVMConfig对应字段赋值

o.ContainsKey(VMName)

currentVMConfig=null;

* GetSysConfig ()

【所属类及所在文件】

GetConfig类，GetConfig.cs文件

【功能描述】

解析负载均衡器配置文件，根据配置文件初始化currentSysConfig变量

【输入参数】

无

【返回值】

SysConfig类实例

【流程图】

开始

读取路径SysConfigPath下的虚拟机配置文件，初始化JsonTextReader类型的reader

JObject o = (JObject)JToken.ReadFrom(reader)

新建用于操作Json对象的JObject

返回cureentVMConfig

currentSysConfig.BalanceStrategy = o["BalanceStrategy"].ToString();

currentSysConfig.CPUDefaultCount = Convert.ToInt32(o["CPUDefaultCount"];

…

读取Json对象并对currentSysConfig对应字段赋值

* GetMpcVMInfo(string VMStateJsonPath, string VMName)

【所属类及所在文件】

GetConfig类，GetConfig.cs文件

【功能描述】

解析多域PC虚拟机配置文件，根据配置文件初始化currentMpcConfig变量

【输入参数】

VMStateJsonPath，多域PC虚拟机配置文件路径，VMName，虚拟机名

【返回值】

MpcConfig类实例

【流程图】

开始

返回currentMpcVMInfo

通过StreamReader读取虚拟机配置文件内容

将读取到Host配置文件内容初始化为JObject o

依次将vmObj如下字段读入currentMpcVMInfo对应字段：

VMNane, Installed, VMPath,DefaultUser, Password, IPAddress, StateSnapID, ,Description, RenamePC

JObject vmObj = ((JObject)o["LocalVM"])

VMName == "LocalVM" && o.ContainsKey("LocalVM")

是

否

VMName == "NetVM1" && o.ContainsKey("LocalVM")

依次将vmObj如下字段读入currentMpcVMInfo对应字段：

VMName == "NetVM2" && o.ContainsKey("LocalVM")

依次将vmObj如下字段读入currentMpcVMInfo对应字段：

否

否

是

currentMpcVMInfo=null

是

### 系统信息获取模块

#### 功能

本模块提供获取Windows性能计数器部分内存、CPU性能相关指标的API。

#### 设计

#### 异常和错误处理

#### 相关数据结构

* SystemInfo类

【数据类型】

class

【功能】

虚拟机的配置信息，包括虚拟机名、预设定内存大小、CPU核心数等

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| VMName | 虚拟机名 | string |
| Installed | 是否安装 | bool |
| MemorySize | 预设内存大小 | UInt64 |
| CPUNum | 预设CPU核心数 | int |
| CPULimt | 预设CPU限制大小 | int |

#### 函数及流程图

* SysConfig类

【数据类型】

class

【功能】

负载均衡器的配置选项信息

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| BalanceStrategy | 均衡策略 | string |
| CPUDefaultCount | 系统默认CPU核心数 | int |
| MemoryDefaultSize | 系统默认内存大小 | UInt64 |
| CPUDefaultLimit | 系统默认的CPU核心数 | int |

是

否

开始

memSpoofer = new MemSpoofer();

cpuSpoofer = new CpuSpoofer();

结束

memSpoofer.RefreshMEMArg();

cpuSpoofer.RefreshCPUArg();

刷新memSpoofer和cpuSpoofer中的性能信息

bool ret = SetPerfTransfer(memSpoofer, cpuSpoofer);

调用SetPerfTransfer函数将性能信息写入perf\_Transfer字段中

bool vmIDRet = GetVMName(@"C:\VMID.txt");

读取VMName.txt文件获取虚拟机名

! vmIDRet

VMName = "Unknown";

* GetVMName(string vmIDFilePath)

【所属类及所在文件】

PerformanceInfo类，PerformanceInfo.cs文件

【功能描述】

读取配置文件，获取虚拟机名称。

【输入参数】

vmIDFilePath，配置文件路径

【返回值】

bool值，表示读取是否成功

【流程图】

开始

返回ret

VMName = File.ReadAllText(vmIDFilePath);

bool ret = true;

读取配置文件中vmname字段

是否产生异常

bool ret = false;

!File.Exists(vmIDFilePath)

否

否

是

是

* SetPerfTransfer(MemSpoofer memSpf, CpuSpoofer cpuSpf)

【所属类及所在文件】

PerformanceInfo类，PerformanceInfo.cs文件

【功能描述】

输入内存、CPU监视器实例，从中获取待传输的性能指标，并据此对perf\_Transfer字段进行复制。

【输入参数】

MemSpoofer memSpf, 内存监视器实例，CpuSpoofer cpuSpf，CPU监视器实例

【返回值】

bool值，表示设置是否成功

【流程图】

开始

返回ret

perf\_Transfer.VMName = VMName;

perf\_Transfer.CPUCount = cpuSpf.CPUCount;

perf\_Transfer.CPUProcessorTime = cpuSpf.CPUProcessorTime

……

根据cpuSpf中CPU性能指标设置perf\_Transfer对应指标

是否产生异常

ret = false;

perf\_Transfer.MEMAvailable = memSpf.MEMAvailable;

perf\_Transfer.PagesPerSec = memSpf.PagesPerSec;

……

根据memSpf中内存性能指标设置perf\_Transfer对应指标，bool ret = true

是

否

* Client(string Addr, int TimeGap = 100)

### 虚拟机管理模块

#### 功能

本模块提供虚拟机对象的属性，以及操作配置虚拟机的对应方法。

#### 设计

#### 异常和错误处理

#### 相关数据结构

* PerformanceSetting结构体

【数据类型】

Struct

【功能】

从Host可以直接获得的性能相关的虚拟机信息；

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| Name | 系统内唯一的实例名 | string |
| ElementName | 虚拟机名 | string |
| InstanceID | 虚拟机ID | string |
| NumberOfProcessors | CPU核心数 | UInt16 |
| EnabledState | 预设CPU限制大小 | UInt16 |
| HealthState | 虚拟机健康状态 | UInt16 |
| ProcessorLoad | 当前虚拟机CPU占总CPU资源的负载率 | UInt16 |
| ProcessorLoadHistory | 记录前100次CPU负载率的数组 | UInt16[] |
| MemoryUsage | 分配的内存大小 | UInt64 |
| CPU\_Reservation | CPU保留 | UInt16[] |
| CPU\_Limit | CPU限制 | UInt64 |
| CPU\_Weight | CPU权重 | UInt32 |
| RAM\_VirtualQuantity | 分配的内存大小 | UInt64 |
| RAM\_Weight | 内存权重 | UInt32 |
| RAM\_DynamicMemoryEnabled | 是否打开动态内存 | bool |

* VirtualMachine类

【数据类型】

class

【功能】

表示虚拟机的属性及操作虚拟机的方法；

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| vmName | 虚拟机名 | string |
| performanceSetting | 与性能相关的虚拟机设置 | PerformanceSetting |
| VirtualMachineStatus | 枚举虚拟机所有状态 | enum |
| vmStatus | 虚拟机状态 | VirtualMachineStatus |
| scope | C# WMI类库中ManagementScope类实例，用于同虚拟机建立连接 | ManagementScope |
| managementService | C# WMI类库中ManagementObject类实例，提供操作虚拟机的服务 | ManagementObject |
| virtualMachine | C# WMI类库中ManagementObject类实例，表示操作的虚拟机 | ManagementObject |

#### 函数及流程图

* VirtualMachine(String \_vmName, ManagementScope scopeM, ManagementObject managementServiceM)

【所属类及所在文件】

VirutualMachine类，VMManagement.cs文件

【功能描述】

VirutualMachine类构造函数。

【输入参数】

\_vmName，虚拟机名；scopeM，虚拟机所在的名空间；managementServiceM，虚拟机管理服务对象

【返回值】

无

【流程图】

否

是

开始

vmName = \_vmName;

scope = scopeM;

managementService = managementServiceM;

将上述输入参数赋值给类变量

结束

调用GetPerformanceSetting();设置performanceSetting属性。

virtualMachine = WmiUtilities.GetVirtualMachine(vmName, scope);

调用GetVitualMchine方法获取虚拟机对象，赋值给类变量virtualMachine

是否产生异常

抛出异常“没有找到虚拟机”

performanceSetting.EnabledState == 2

vmState = PowerOff

设置虚拟机状态为关机

vmState = PowerOn

设置虚拟机状态为开机

是

否

* GetPerformanceSetting()

【所属类及所在文件】

VirutualMachine类，VMManagement.cs文件

【功能描述】

获得虚拟机性能相关设置，并给performanceSetting类变量赋值。

【输入参数】

无

【返回值】

PerformanceSetting class类型

【流程图】

开始

返回performanceSetting

获取虚拟机的summaryInfo集合:

virtualSystemSettings=virtualMachine.GetRelated(“Msvm\_SummaryInformation”)

读取virtualSystemSettings，依次对performanceSetting各个字段进行赋值

* IsPowerOn()

【所属类及所在文件】

VirutualMachine类，VMManagement.cs文件

【功能描述】

判断是否开机。

【输入参数】

无

【返回值】

bool值，表示是否开机

【流程图】

开始

返回ret

调用GetPerformanceSetting ()函数，获取当前信息

是

performanceSetting.EnabledState == 2（开机状态为2）

否

bool ret = false

bool ret = true

* IsPowerOff()

【所属类及所在文件】

VirutualMachine类，VMManagement.cs文件

【功能描述】

判断是否关机。

【输入参数】

无

【返回值】

bool值，表示是否关机

【流程图】

开始

返回ret

调用GetPerformanceSetting ()函数，获取当前信息

是

performanceSetting.EnabledState == 3（关机状态为3）

否

bool ret = false

bool ret = true

* PowerOn()

【所属类及所在文件】

VirutualMachine类，VMManagement.cs文件

【功能描述】

执行虚拟机的开机操作。

【输入参数】

无

【返回值】

bool值，表示是否成功开机

【流程图】

开始

返回result

bool result = false;

ManagementBaseObject inParams = virtualMachine.GetMethodParameters("RequestStateChange")

将inParams参数设定为需要修改状态的格式

inParams["RequestedState"] = StartID; 将需要修改的状态设置为开机

ManagementBaseObject outParams = virtualMachine.InvokeMethod("RequestStateChange", inParams, null)

设置对应outParams参数

result =WmiUtilities.ValidateOutput(outParams, scope)

调用函数执行开机操作

是否出现异常

输出“无法开机”及异常原因

是

否

* PowerOff()

【所属类及所在文件】

VirutualMachine类，VMManagement.cs文件

【功能描述】

执行虚拟机的关机操作。

【输入参数】

无

【返回值】

bool值，表示是否成功关机

【流程图】

开始

返回result

bool result = false;

ManagementBaseObject inParams = virtualMachine.GetMethodParameters("RequestStateChange")

将inParams参数设定为需要修改状态的格式

inParams["RequestedState"] = SthutdownID; 将需要修改的状态设置为关机

ManagementBaseObject outParams = virtualMachine.InvokeMethod("RequestStateChange", inParams, null)

设置对应outParams参数

result =WmiUtilities.ValidateOutput(outParams, scope)

调用函数执行开机操作

是否出现异常

输出“无法开机”及异常原因

是

否

* ModifySettingData(String settingName, String settingData)

【所属类及所在文件】

VirutualMachine类，VMManagement.cs文件

【功能描述】

修改虚拟机的内存或CPU配置。

【输入参数】

settingName，要修改的配置项；settingData，修改后的数据

【返回值】

bool值，表示是否修改成功

【流程图】

①

判断是否修改CPU配置:

settingName.Substring(0,3) == “CPU”

获取要修改的字段名: settingName = settingName.Substring(4)

获取处理器配置对象集合:processorSettingDatas=

virtualSystemSetting.GetRelated(“Msvm\_ProcessorSettingData”)

获取集合processorSettingDatas中的第一个元素:

processorSettingData = processorSettingDatas.FirstItem()

获取processorSettingData[settingName]的类型type, 并将settingData转换为该类型,然后赋值:

processorSettingData[settingName] = (Type)SettingData

获取方法ModifyResourceSettings的输入参数

inParams=managementService.GetMethodParameters(“ModifyResourceSettings”)

设置输入参数inParams[“ResourceSettings”]=processorSettingData.GetText(...)

调用方法ModifyResourceSettings修改系统设置outParams=

managementService.InvokeMethod(“ModifyResourceSettings”, inParams, null)

验证调用ModifyResourceSettings的输出参数:

result=WmiUtilities.ValidateOutput(outParams, scope)

是否出现异常?

否

是

否

是

②

result = false; 获取虚拟机的配置对象:

virtualSystemSettings=WmiUtilities.GetVirtualMachineSettings(virtualMachine)

开始

输出“无法修改CPU参数”及异常原因

返回result

①

判断是否修改内存配置:

settingName.Substring(0,3) == “RAM”

获取要修改的字段名: settingName = settingName.Substring(4)

获取内存配置对象集合:memorySettingDatas=

virtualSystemSetting.GetRelated(“Msvm\_MemorySettingData”)

获取集合memorySettingDatas中的第一个元素:

memorySettingData= memorySettingDatas.FirstItem()

获取memorySettingData[settingName]的类型type, 并将settingData转换为该类型,然后赋值:

memorySettingData[settingName] = (Type)SettingData

获取方法ModifyResourceSettings的输入参数

inParams=managementService.GetMethodParameters(“ModifyResourceSettings”)

设置输入参数inParams[“ResourceSettings”]

调用方法ModifyResourceSettings修改系统设置outParams=

managementService.InvokeMethod(“ModifyResourceSettings”, inParams, null)

验证调用ModifyResourceSettings的输出参数:

result=WmiUtilities.ValidateOutput(outParams, scope)

是否出现异常?

否

是

是

②

输出“无法修改内存参数”及异常原因

返回result

### 动态调节接口模块

#### 功能

本模块提供了动态调节CPU、内存指标的API。

#### 设计

#### 异常和错误处理

#### 相关数据结构

#### 函数及流程图

* isDynamicMem(VirtualMachine VM)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

判断是否开启了动态内存。

【输入参数】

VM，VirtualMachine类实例

【返回值】

bool值，表示是否开启动态内存

【流程图】

开始

返回ret

bool ret = (VM.performanceSetting.RAM\_DynamicMemoryEnabled == true)

* AdjustMemorySize(VirtualMachine VM, UInt64 MemorySize)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

调整虚拟机的内存大小。

【输入参数】

VM，VirtualMachine类实例；MemorySize，调整后的内存大小

【返回值】

bool值，表示内存大小调节是否成功

【流程图】

开始

返回ret

bool ret = VM.ModifySettingData("RAM\_VirtualQuantity", Convert.ToString(MemorySize));

* AppendVMMemory(VirtualMachine VM, UInt64 MemorySizeBaseLine, UInt64 MemorySize, int rank)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

按比例扩展虚拟机内存大小。

【输入参数】

VM，VirtualMachine类实例；MemorySizeBaseLine，用户设置的基准内存；MemorySize，当前的内存大小；rank，扩展的等级

【返回值】

bool值，表示内存扩展大小是否成功

【流程图】

开始

返回ret

UInt64 appendSize = MemorySizeBaseLine / 8;内存扩展大小为基准内存的1/8

bool ret = VM.ModifySettingData("RAM\_VirtualQuantity", Convert.ToString(MemorySize + appendSize));

rank == 1

ret = false

是

否

UInt64 appendSize = MemorySizeBaseLine / 4;内存扩展大小为基准内存的1/4

bool ret = VM.ModifySettingData("RAM\_VirtualQuantity", Convert.ToString(MemorySize + appendSize));

rank == 2

是

是

否

否

* RecycleVMMemory(VirtualMachine VM, UInt64 MemorySizeBaseLine, UInt64 MemorySize, int rank)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

按比例回收虚拟机内存大小。

【输入参数】

VM，VirtualMachine类实例；MemorySizeBaseLine，用户设置的基准内存；MemorySize，当前的内存大小；rank，内存回收的等级

【返回值】

bool值，表示内存回收是否成功

【流程图】

开始

返回ret

UInt64 recycleSize= MemorySizeBaseLine / 8;内存回收大小为基准内存的1/8

bool ret = VM.ModifySettingData("RAM\_VirtualQuantity", Convert.ToString(MemorySize-recycleSize));

rank == 1

ret = false

是

否

UInt64 recycleSize= MemorySizeBaseLine / 4;内存回收大小为基准内存的1/4

bool ret = VM.ModifySettingData("RAM\_VirtualQuantity", Convert.ToString(MemorySize -recycleSize));

rank == 2

是

是

否

否

* AdjustMemoryWeight(VirtualMachine VM, int MemoryWeight)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

调整虚拟机的内存大小。

【输入参数】

VM，VirtualMachine类实例；MemoryWeight，调整后的内存权重

【返回值】

bool值，表示内存权重调节是否成功

【流程图】

开始

返回ret

bool ret = VM.ModifySettingData("RAM\_Weight", Convert.ToString(MemoryWeight));

* AdjustCPUCount(VirtualMachine VM, int Count)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

关机状态下，调整虚拟机分配的CPU核心数。

【输入参数】

VM，VirtualMachine类实例；Count，调整后的CPU核心数

【返回值】

bool值，表示CPU核心数调节是否成功

【流程图】

开始

返回ret

bool ret = VM.ModifySettingData("CPU\_VirtualQuantity", Convert.ToString(Count));

VM.IsPowerOn()

否

是

ret = false

* AdjustCPUWeight (VirtualMachine VM, int CPUWeight)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

调整虚拟机的CPU权重。

【输入参数】

VM，VirtualMachine类实例；CPUWeight，调整后的CPU权重

【返回值】

bool值，表示CPU权重调节是否成功

【流程图】

开始

返回ret

bool ret = VM.ModifySettingData("CPU\_Weight", Convert.ToString(CPUWeight));

* AdjustCPULimit(VirtualMachine VM, UInt64 CPULimit)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

调整虚拟机的CPU限制参数。

【输入参数】

VM，VirtualMachine类实例；CPULimit，调整后的CPU限制参数

【返回值】

bool值，表示CPU限制参数调节是否成功

【流程图】

开始

返回ret

bool ret = VM.ModifySettingData("CPU\_Limit", Convert.ToString(CPULimit));

* AdjustCPUReservation(VirtualMachine VM, UInt64 CPUReservation)

【所属类及所在文件】

DynamicAdjustment类，DynamicAdjustment.cs文件

【功能描述】

调整虚拟机的CPU保留参数。

【输入参数】

VM，VirtualMachine类实例；CPUReservation，调整后的CPU保留参数

【返回值】

bool值，表示CPU保留参数调节是否成功

【流程图】

开始

返回ret

bool ret = VM.ModifySettingData("CPU\_Reservation", Convert.ToString(CPUReservation));

### HyperV性能监测模块

#### 功能

监测Windows中性能计数器中有关Hyper-V性能的部分，如虚拟机的当前内存压力、平均内存压力等。

#### 设计

微软官方建议，正常状态的虚拟机内存压力在80左右，若内存压力计数器大于100，则考虑为虚拟机分配内存。

#### 异常和错误处理

#### 相关数据结构

* VMHvPerfCounterInfo类

【数据类型】

class

【功能】

表示虚拟机对象对应的Hyper-V性能计数器中内存当前压力、平均压力信息；

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| vmName | 虚拟机名 | string |
| averagePressure | 虚拟机平均内存压力 | UInt16 |
| currentPressure | 虚拟机当前内存压力 | UInt16 |

* HyperVPerfCounter类

【数据类型】

class

【功能】

获得Hyper-V相关的相关性能计数器的值；

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| memoryAveragePressure | 虚拟机名 | PerformanceCounter |
| memoryCurrentPressure | 虚拟机平均内存压力 | PerformanceCounter |

【包含的方法】

|  |  |
| --- | --- |
| 函数签名 | 作用 |
| public HyperVPerfCounter() | 无参数的构造函数，函数体为空 |
| public HyperVPerfCounter(string VMName) | 构造函数，输入虚拟机名，调用SetPerCounter函数初始化该虚拟机对应的Hyper-V性能计数器 |
| public bool SetPerCounter(string VMName) | 判断输入的虚拟机是否开机，若开机则初始化该虚拟机的Hyper-V性能计数器 |
| public VMHvPerfCounterInfo GetVMHyperVPerfInfo(string VMName) | 查询该虚拟机的Hyper-V相关性能计数器，返回VMHvPerfCounterInfo类实例 |

#### 函数及流程图

* HyperVPerfCounter(string VMName)

【所属类及所在文件】

HyperVPerfCounter类，HyperVPerfCounter.cs文件

【功能描述】

HyperVPerfCounter类构造函数。

【输入参数】

VMName，虚拟机名

【返回值】

无

【流程图】

开始

返回performanceSetting

SetPerCounter(VMName);

初始化该虚拟机的性能计数器

否

是

* SetPerCounter(string VMName )

【所属类及所在文件】

HyperVPerfCounter类，HyperVPerfCounter.cs文件

【功能描述】

判断VMName对应的虚拟机是否安装且开机，然后初始化该虚拟机对应的性能计数器，若获取失败则返回false。

【输入参数】

VMName，虚拟机名

【返回值】

bool值，表示初始化该虚拟机对应的Hyper-V性能计数器是否成功

【流程图】

开始

返回ret

memoryCurrentPressure = new PerformanceCounter("Hyper-V Dynamic Memory VM", "Current Pressure", VMName);

memoryAveragePressure = new PerformanceCounter("Hyper-V Dynamic Memory VM", "Average Pressure", VMName);

初始化虚拟机对应的当前内存压力和平均内存压力的计数器。

bool ret = true;

判断VMName是否合法

bool ret = false

判断该虚拟机是否安装

判断该虚拟机是否开机状态

否

否

否

是

是

是

* GetVMHyperVPerfInfo(string VMName)

【所属类及所在文件】

HyperVPerfCounter类，HyperVPerfCounter.cs文件

【功能描述】

获得虚拟机当前的Hyper-V相关性能计数器中的性能信息。

【输入参数】

VMName，虚拟机名

【返回值】

VMHvPerfCounterInfo类实例，表示虚拟机对象对应的Hyper-V性能计数器中内存当前压力、平均压力等信息

【流程图】

开始

返回vmHvPerfCounterInfo

VMHvPerfCounterInfo vmHvPerfCounterInfo = new VMHvPerfCounterInfo();

vmHvPerfCounterInfo.VMName = VMName;

vmHvPerfCounterInfo.averagePressure = memoryAveragePressure.NextValue();

vmHvPerfCounterInfo.currentPressure = memoryCurrentPressure.NextValue();

获取性能计数器中的值，设置vmHvPerfCounterInfo中的对应字段。

bool ret = SetPerCounter(VMName);

调用SetPerCounter函数设置计数器

ret

vmHvPerfCounterInfo = null;

否

是

### 性能接收模块

#### 功能

本模块接收所有域內性能监测器发送来的性能信息。

#### 设计

#### 异常和错误处理

#### 相关数据结构

1. Server类

【数据类型】

class

【功能】

性能接收的服务器类，定义了消息传输所需的属性和方法。

【详细数据结构】

服务端绑定的地址，分两部分，前一部分表示接收所有虚拟机的连接，后一部分是注册表中注册的该服务的ID；

public string DetectorServerAddr = "hypervnb://00000000-0000-0000-0000-000000000000/C7240163-6E2B-4466-9E41-FF74E7F0DE47";

接口Iserver，定义服务端和客户端交互服务契约和契约函数：

public interface Iserver

继承IServer接口，实现契约函数：

public class SampleServer : Iserver

SampleServer类实例；

public static SampleServer mySampleServer;

1. 接口IServer

【数据类型】

interface

【功能】

定义服务契约IServer，其中包含传输性能信息的操作契约函数TransferPerfStr(string perf\_Str)；接口IServer的定义与客户端相同。

【详细数据结构】

[ServiceContract]

public interface IServer

{

[OperationContract]

VMPerf TransferPerfStr(string perf\_Str);

}

1. SampleServer类

【数据类型】

class

【功能】

实现TransferPerfStr(string perf\_Str)函数，接收性能信息，将收到的字符串反序列化为VMPerf类实例。

【详细数据结构】

通过[ServiceBehavior]特性，在服务级别应用规则和行为，InstanceContextMode设置服务实例的生命周期，ConcurrencyMode = ConcurrencyMode.Multiple，设置服务以多线程运行，需要注意线程安全的问题，UseSynchronizationContext = false表示用户接口线程与所需服务之间没有相关性。

[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single, ConcurrencyMode = ConcurrencyMode.Multiple, UseSynchronizationContext = false)]

public class SampleServer : Iserver{

字典类型，key为string类型的虚拟机名，value是其对应的虚拟机的VMPerf类性能信息：

public Dictionary<string, VMPerf> vmPerfDict

TransferPerfStr(string perf\_Str)方法的实现，详见函数和流程图部分：

public VMPerf TransferPerfStr(string perf\_Str)

{ 函数实现…… }

#### 函数及流程图

* TransferPerfStr(string perf\_Str)

【所属类及所在文件】

SampleServer类，Server.cs文件

【功能描述】

SampleServer继承接口Iserver，实现TransferPerfStr(string perf\_Str)。方法，接收域内发来的字符串形式的性能信息，对其反序列化，返回VMPerf类实例。

【输入参数】

perf\_Str，string类型的性能消息。

【返回值】

VMPerf类实例，虚拟机性能信息类实例

【流程图】

开始

返回perf\_Transfer

byte[] buffer = Convert.FromBase64String(perf\_Str);

MemoryStream stream = new MemoryStream(buffer);

将输入的perf\_Str先转换为字节数组，再转换为MemoryStream

vmPerfDict[perf\_Transfer.VMName] = perf\_Transfer；设置vmPerfDict字典中对应的value

vmPerfDict.ContainsKey(perf\_Transfer.VMName)，判断字典是否含有该虚拟机

是

否

BinaryFormatter formatter = new BinaryFormatter();

perf\_Transfer = (VMPerf)formatter.Deserialize(stream);

将MemoryStream类型的stream反序列化为VMPerf类型的perf\_Transfer，然后对stream进行刷新操作和关闭操作

mPerfDict.Add(perf\_Transfer.VMName, perf\_Transfer)；vmPerfDict字典中添加key和value

* Server(string Addr)

【所属类及所在文件】

Server类，Server.cs文件

【功能描述】

Server类构造函数。

【输入参数】

Addr，string类型，服务端绑定的地址

【返回值】

无

【异常处理】

启动服务器的整个流程中，捕获可能产生的异常

【流程图】

开始

结束

DetectorServerAddr = Addr,

* StartUpServer()

【所属类及所在文件】

Server类，Server.cs文件

【功能描述】

启动性能接收服务器。

【输入参数】

无

【返回值】

无

【流程图】

开始

结束

mySampleServer = new SampleServer();

var sh = new ServiceHost(mySampleServer);

ServiceHost来进行WCF程序的服务寄宿：新建一个承载SampleServer服务的宿主，

输出异常详细信息

是否捕获异常

是

否

var binding = new HyperVNetBinding();

sh.AddServiceEndpoint(typeof(IServer), binding, DetectorServerAddr);新建HyperV Socket的地址绑定，调用AddServiceEndpoint将DetectorServerAddr地址绑定到服务端

sh.Open();

Console.ReadLine();

sh.Close();

启动服务，Console.ReadLine阻塞使其一直提供风月无涯

### 虚拟机开关机状态监测模块

#### 功能

本模块记录并监测用户在多域PC系统中对域的开机、关机和切入使用操作，以及卸载域、安装域的配置操作。多域PC系统调用本系统中的可执行程序-虚拟机状态通知工具通知负载均衡器域使用中状态的变化。

#### 设计

#### 异常和错误处理

#### 相关数据结构

* MemoryMapping类

【数据类型】

class

【功能】

虚拟机状态转换工具作为一个exe，与负载均衡器通过共享内存进行通信，该类定义了共享内存通信所需的属性如信号量，以及通信的方法；

【所在文件】

MemoryMapping.cs

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| capacity | 共享内存文件的大小，初始值为10MB | long |
| file | 共享内存文件 | MemoryMappedFile |
| m\_Write | 可写的信号量 | Semaphore |
| m\_Read | 可读的信号量 | Semaphore |
| m\_Received | 是否收到信息，并处理完请求的信号量 | Semaphore |

【包含的方法】

|  |  |
| --- | --- |
| 函数签名 | 作用 |
| public MemoryMapping(string fileName) | 构造函数，初始化各个信号量，并根据文件名新建共享内存文件 |
| public void WriteString(string msg) | 向共享内存区域写一个字符串，通过信号量进行互斥和同步 |
| public string ReadString() | 从共享内存中读取字符串，通过信号量进行互斥和同步 |
| public void ReleaseReceiveSemaphore() | 释放m\_Received信号量 |

* VMState类

【数据类型】

class

【功能】

表示各个虚拟机的开关机、切入状态，提供监测虚拟机状态的方法；

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| currentUsedVM | 当前切入的虚拟机名，若无切入虚拟机则为”GetOff” | string |
| LocalVM | VirtualMachine类实例，本地域，若该域未安装则为null | static VirtualMachine |
| LocalVMConfig | 配置文件中读取的本地域配置 | static VMConfig |
| NetVM1 | VirtualMachine类实例，联网域1，若该域未安装则为null | static VirtualMachine |
| NetVM1Config | 配置文件中读取的联网域1配置 | static VMConfig |
| NetVM2 | VirtualMachine类实例，联网域1，若该域未安装则为null | static VirtualMachine |
| NetVM2Config | 配置文件中读取的联网域1配置 | static VMConfig |
| VMPowerOnStatus | 长度为3的bool数组，分别表示三个域是否开机 | static bool[] |

【包含的方法】

|  |  |
| --- | --- |
| 函数签名 | 作用 |
| public VMState(string MpcVMConfigPath) | 构造函数，输入多域PC的虚拟机配置文件路径，对类中包含的各个变量初始化 |
| public static void SetVMState(string VMName, string State) | 设置虚拟机的状态，VMName为虚拟机名，State为要转换到的虚拟机状态 |
| public static void ReceiveMessage() | 循环调用 memoryMapping.ReadString()函数，  从共享内存区域尝试读取消息 |
| public void StartMessageReceiver() | 以ReceiveMessage()函数启动一个线程 |

#### 函数及流程图

* VMState(string MpcVMConfigPath)

【所属类及所在文件】

VMState类，VMState文件

【功能描述】

VMState类构造函数，。

【输入参数】

perf\_Str，string类型的性能消息。

【返回值】

VMPerf类实例，虚拟机性能信息类实例

【流程图】

开始

返回perf\_Transfer

byte[] buffer = Convert.FromBase64String(perf\_Str);

MemoryStream stream = new MemoryStream(buffer);

将输入的perf\_Str先转换为字节数组，再转换为MemoryStream

vmPerfDict[perf\_Transfer.VMName] = perf\_Transfer；设置vmPerfDict字典中对应的value

vmPerfDict.ContainsKey(perf\_Transfer.VMName)，判断字典是否含有该虚拟机

是

否

BinaryFormatter formatter = new BinaryFormatter();

perf\_Transfer = (VMPerf)formatter.Deserialize(stream);

将MemoryStream类型的stream反序列化为VMPerf类型的perf\_Transfer，然后对stream进行刷新操作和关闭操作

mPerfDict.Add(perf\_Transfer.VMName, perf\_Transfer)；vmPerfDict字典中添加key和value

* Server(string Addr)

【所属类及所在文件】

Server类，Server.cs文件

【功能描述】

Server类构造函数。

【输入参数】

Addr，string类型，服务端绑定的地址

【返回值】

无

【异常处理】

启动服务器的整个流程中，捕获可能产生的异常

【流程图】

开始

结束

DetectorServerAddr = Addr,

* StartUpServer()

【所属类及所在文件】

Server类，Server.cs文件

【功能描述】

启动性能接收服务器。

【输入参数】

无

【返回值】

无

【流程图】

开始

结束

mySampleServer = new SampleServer();

var sh = new ServiceHost(mySampleServer);

ServiceHost来进行WCF程序的服务寄宿：新建一个承载SampleServer服务的宿主，

输出异常详细信息

是否捕获异常

是

否

var binding = new HyperVNetBinding();

sh.AddServiceEndpoint(typeof(IServer), binding, DetectorServerAddr);新建HyperV Socket的地址绑定，调用AddServiceEndpoint将DetectorServerAddr地址绑定到服务端

sh.Open();

Console.ReadLine();

sh.Close();

启动服务，Console.ReadLine阻塞使其一直提供风月无涯

### 负载分析均衡模块

#### 功能

本模块提供获取Windows性能计数器部分内存、CPU性能相关指标的API。

#### 设计

#### 异常和错误处理

#### 相关数据结构

* MemoryAnalysor类

【数据类型】

class

【功能】

内存负载分析类，包含分析内存性能所需要的的属性，以及定时进行内存负载分析的方法；

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| currentVirtualMachine | 当前分析的虚拟机 | VirtualMachine类 |
| percentageThredHold | 内存占用率的阈值 | double |
| currentVMPerf | 当前得到的域内虚拟机性能信息 | VMPerf 类 |
| detectAlarmTimes | 内存告警的次数 | int |
| memAlarmTimesLimit | 内存告警次数的阈值 | int |
| memFreeRanking | 内存空闲状态，1-10,,5为中间状态，1表示内存最为空闲 | int |
| detectTimeGap | 监测的时间间隔 | int |
| isMemAlarm | 是否产生报警信号 | bool |
| RUtimer | 监测的定时器 | Timers.Timer |
| Cleartimer | 清空内存告警的计时器 | Timers.Timer |

* CpuAnalysor类

【数据类型】

class

【功能】

CPU负载分析类，包含监测CPU性能所需要的的属性，以及定时进行CPU负载分析的方法。

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| currentVirtualMachine | 当前分析的虚拟机 | VirtualMachine类 |
| processQueueLengthThredHold | CPU占用率的阈值 | int |
| currentVMPerf | 当前得到的域内虚拟机性能信息 | VMPerf 类 |
| detectAlarmTimes | CPU告警的次数 | int |
| cpuAlarmTimesLimit | CPU告警次数的阈值 | int |
| cpuFreeRanking | CPU空闲状态，1-10,,5为中间状态，1表示内存最为空闲 | int |
| detectTimeGap | 监测的时间间隔 | int |
| isCpuAlarm | 是否产生报警信号 | bool |
| RUtimer | 监测的定时器 | Timers.Timer |
| Cleartimer | 清空CPU告警的计时器 | Timers.Timer |

* LoadBalancer类

【数据类型】

class

【功能】

负载均衡调节类，调用内存、CPU分析类，获取实时性能信息，定时进行资源负载均衡分配与调节。

【详细数据结构】

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 类型 |
| currentUsedVM | 当前切入的虚拟机 | string |
| memoryAnalysorDict | 字典类型，key为虚拟机实例，value为其对应的内存分析器实例 | Dictionary<VirtualMachine, MemoryAnalysor> |
| cpuAnalysorrDict | 字典类型，key为虚拟机实例，value为其对应的CPU分析器实例 | Dictionary<VirtualMachine, CpuAnalysor> |
| systemInfo | 系统信息类实例，包含Host物理机的物理硬件及性能信息 | SystemInfo类 |
| hyperVPerfCounter | Hyper-V相关的性能计数器实例 | HyperVPerfCounter类 |
| hostMemReservedPercentage | Host物理机内存的保留的最低比例 | double |
| detectTimeGap | 均衡器监测的时间间隔(ms) | int |
| Balancetimer | 监测的定时器 | Timers.Timer |
| myDetectorServer | 性能接收模块实例 | Server类 |
| dynamicAdjustment | 调节虚拟机性能设置参数的类实例 | DynamicAdjustment类 |

#### 函数及流程图

* MemoryAnalysor(VirtualMachine vm, double memthredhold, int alarmTimesLimit, int detectTime)

【所属类及所在文件】

MemoryAnalysor类，LoadBalancer.cs文件

【功能描述】

MemoryAnalysor类构造函数。

【输入参数】

vm，虚拟机对象；memThredhold，内存占用率的门限值；alarmTimesLimit，内存报警的次数门限值，detectTime，分析内存性能的时间间隔

【返回值】

无

【流程图】

开始

返回performanceSetting

currentVirtualMachine = vm;

percentageThredHold = memThredhold;

detectAlarmTimes = 0;

cpuAlarmTimesLimit = alarmTimesLimit;

detectTimeGap = detectTime;

cpuFreeRanking = 5;

根据输入值设置对应参数，初始化所有类变量

* DetectMemByTime()

【所属类及所在文件】

MemoryAnalysor类，LoadBalancer.cs文件

【功能描述】

设置内存监测定时器，定时分析内存性能；同时设置内存报警清空的定时器，定时清空内存告警的次数。

【输入参数】

无

【返回值】

无

【流程图】

开始

RUtimer = new System.Timers.Timer(detectTimeGap);

创建一个定时为detectTimeGap的计时器

RUtimer.Elapsed += DetectMemoryState;

定时时间到，处理函数为DetectMemoryState;

RUtimer.AutoReset = true;

RUtimer.Enabled = true;

RUtimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

结束

否

是

Cleartimer= new System.Timers.Timer(detectTimeGap\*100);

创建一个定时为detectTimeGap\*100的计时器

Cleartimer.Elapsed += ClearMemState;

定时时间到，处理函数为ClearMemState;

Cleartimer.AutoReset = true;

Cleartimer.Enabled = true;

Cleartimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

否

是

* DetectMemoryState(object sender, ElapsedEventArgs e)

【所属类及所在文件】

MemoryAnalysor类，LoadBalancer.cs文件

【功能描述】

分析内存的性能信息，若内存出现性能瓶颈，设置内存报警信号。

【输入参数】

object sender:表示触发事件的对象，EventArgs e：表示事件数据的类的基类

【返回值】

无

【流程图】

开始

结束

获取分配给虚拟机的内存allocMemSize，计算虚拟机内存占用率memPercentage

currentVMPerf = Server.mySampleServer.vmPerfDict[currentVirtualMachine.vmName];

float memAvailable = currentVMPerf.MEMAvailable;

float pagesPerSec = currentVMPerf.PagesPerSec;

从性能接收模块获取当前虚拟机的性能信息，得到当前域内可用内存和Page Fault的频率

当前虚拟机是否开机

!Server.mySampleServer.vmPerfDict.ContainsKey(currentVirtualMachine.vmName)

detectAlarmTimes += 1;

内存告警次数加一

是

否

memPercentage > percentageThredHold

detectAlarmTimes > memAlarmTimesLimit

isMemAlarm = true;

memFreeRanking > 1

memFreeRanking -= 1;标记内存更加紧张

是

是

是

否

是

否

否

否

* ClearMemState(object sender, ElapsedEventArgs e)

【所属类及所在文件】

MemoryAnalysor类，LoadBalancer.cs文件

【功能描述】

清空内存告警次数，如果有内存报警，则取消内存报警信号。

【输入参数】

无

【返回值】

无

【流程图】

开始

返回performanceSetting

SetAlarmTimesZero():清空告警次数

CancelMemAlarm();取消内存报警信号

* CpuAnalysor(VirtualMachine vm, double percentagethredhold, int queuelengththredhold, int alarmTimesLimit, int detectionGap)

【所属类及所在文件】

CpuAnalysor类，LoadBalancer.cs文件

【功能描述】

CpuAnalysor类构造函数。

【输入参数】

vm，虚拟机对象；percentagethredhold，CPU占用率的门限值；queuelengththredhold ,等待CPU的线程数门限值；alarmTimesLimit，CPU报警的次数门限值，detectionGap，分析CPU性能的时间间隔

【返回值】

无

【流程图】

开始

返回performanceSetting

currentVirtualMachine = vm;

percentageThredHold = percentagethredhold;

processQueueLengthThredHold = queuelengththredhold;

detectAlarmTimes = 0;

cpuAlarmTimesLimit = alarmTimesLimit;

detectTimeGap = detectionGap;

cpuFreeRanking = 5;

根据输入值设置对应参数，初始化所有类变量

* DetectCpuByTime ()

【所属类及所在文件】

CpuAnalysor类，LoadBalancer.cs文件

【功能描述】

设置CPU监测定时器，定时分析CPU性能；同时设置CPU报警信号清空的定时器，定时清空内存告警的次数。

【输入参数】

无

【返回值】

无

【流程图】

开始

RUtimer = new System.Timers.Timer(detectTimeGap);

创建一个定时为detectTimeGap的计时器

RUtimer.Elapsed += DetectCpuState;

定时时间到，处理函数为DetectCpuState;

RUtimer.AutoReset = true;

RUtimer.Enabled = true;

RUtimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

结束

否

是

Cleartimer= new System.Timers.Timer(detectTimeGap\*100);

创建一个定时为detectTimeGap\*100的计时器

Cleartimer.Elapsed += ClearMemState;

定时时间到，处理函数为ClearCpuState;

Cleartimer.AutoReset = true;

Cleartimer.Enabled = true;

Cleartimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

否

是

* DetectCpuState (object sender, ElapsedEventArgs e)

【所属类及所在文件】

CpuAnalysor类，LoadBalancer.cs文件

【功能描述】

分析CPU的性能信息，若CPU出现性能瓶颈，发出CPU报警信号。

【输入参数】

object sender:表示触发事件的对象，EventArgs e：表示事件数据的类的基类

【返回值】

无

【流程图】

开始

结束

currentVMPerf = Server.mySampleServer.vmPerfDict[currentVirtualMachine.vmName];

float CpuPercentage = currentVMPerf.CPUProcessorTime;

float processQueueLength = currentVMPerf.ProcessorQueueLength;

从性能接收模块获取当前虚拟机的性能信息，得到当前域内CPU占用率，以及等待分配处理器的线程数

当前虚拟机是否开机

!Server.mySampleServer.vmPerfDict.ContainsKey(currentVirtualMachine.vmName)

detectAlarmTimes += 1;

内存告警次数加一

是

否

CPU占用率大于阈值或processQueueLength大于阈值

detectAlarmTimes > cpuAlarmTimesLimit

isCpuAlarm = true;

cpuFreeRanking > 1

cpuFreeRanking -= 1;标记CPU更加紧张

是

是

是

否

是

否

否

否

* ClearCpuState (object sender, ElapsedEventArgs e)

【所属类及所在文件】

CpuAnalysor类，LoadBalancer.cs文件

【功能描述】

清空CPU次数，如果有CPU则取消CPU报警信号。

【输入参数】

无

【返回值】

无

【流程图】

开始

返回performanceSetting

SetAlarmTimesZero():清空告警次数

CancelCpuAlarm();取消CPU报警信号

* BalanceByTime ()

【所属类及所在文件】

LoadBalancer类，LoadBalancer.cs文件

【功能描述】

LoadBalancer负载均衡器设置定时器，定时执行BalanceAllVM函数，对各虚拟机进行负载均衡调整。

【输入参数】

无

【返回值】

无

【流程图】

开始

RUtimer = new System.Timers.Timer(detectTimeGap);

创建一个定时为detectTimeGap的计时器

RUtimer.Elapsed += BalanceAllVM;

定时时间到，处理函数为BalanceAllVM;

RUtimer.AutoReset = true;

RUtimer.Enabled = true;

RUtimer.Start();

设置定时时间到自动重新计时，使能定时器并启动

返回

否

是

* BalanceAllVM (object sender, ElapsedEventArgs e)

【所属类及所在文件】

LoadBalancer类，LoadBalancer.cs文件

【功能描述】

分析内存、CPU性能分析器收集的信息，如有必要，调用资源动态调节接口，对虚拟机的资源进行均衡调节。

【输入参数】

object sender:表示触发事件的对象，EventArgs e：表示事件数据的类的基类

【返回值】

无

【流程图】

开始

VirtualMachine currentVM = kvp.Key;

MemoryAnalysor currentAnalysor = kvp.Value;

调用虚拟机资源动态调节接口函数AppendVMMemory，扩展内存，扩展的值为基准分配值的1/8，

遍历memoryAnalysorDict中的每一项kvp，判断是否遍历完

是

(VMState.LocalVMPerfCounterInfo.currentPressure > 100

是

否

否

否

否

VMState.LocalVMPerfCounterInfo = hyperVPerfCounter .GetVMHyperVPerfInfo("LocalVM");

设置并获取VMState类静态变量LocalVMPerfCounterInfo，即本地域的Hyper-V性能计数器信息。

currentVM.GetPerformanceSetting();刷新虚拟机性能设置信息

是否为本地域，且本地域为开机状态

虚拟机是否为正在开机中的状态，即未完全启动

调用虚拟机资源动态调节接口函数RecycleVMMemory，回收内存，回收的大小为基准分配值的1/8，

(VMState.LocalVMPerfCounterInfo.averagePressure < 70

是

互联网域1，执行相同流程

否

互联网域2，执行相同流程

currentAnalysor.isMemAlarm == true

是

是

是

否

遍历cpuAnalysorrDict中的每一项kvp，判断是否遍历完

VirtualMachine currentVM = kvp.Key;

currentVM.GetPerformanceSetting();

获取cpuAnalysorrDict中的key，即当前的虚拟机，调用GetPerformanceSetting()刷新最新的性能相关设置。

CpuAnalysor currentAnalysor = kvp.Value;

获取当前虚拟机的CPU限制和CPU保留参数：currentCpuLimit和currentCpuReserve

currentAnalysor.isCpuAlarm == true

currentCpuLimit > 90000

bool ret = dynamicAdjustment.AdjustCPULimit(currentVM, currentCpuLimit + 10000); 将虚拟机限制提高10%

currentCpuReserve < 50000

ret &= dynamicAdjustment.AdjustCPUReservation(currentVM, currentCpuReserve + 10000); 将虚拟机保留提高10%

ret

currentAnalysor.isCpuAlarm = false;取消CPU告警信号，cpuAnalysorrDict[currentVM].cpuFreeRanking += 1; 空闲等级＋1

currentAnalysor.cpuFreeRanking > 6

是

是

否

是

bool ret = dynamicAdjustment.AppendVMMemory(currentVM, Config.MemoryBaseLine, currentVM.performanceSetting.RAM\_VirtualQuantity, 1);

扩展虚拟机内存，，扩展的值为基准分配值的1/8，

是

ret

currentAnalysor.isMemAlarm = false;取消内存告警信号，memoryAnalysorDict[currentVM].memFreeRanking += 1; 空闲等级＋1

是

是

否

否

否

否

否

否

是

currentAnalysor.cpuFreeRanking > 8

currentCpuReserve > 0

先调低虚拟机保留值，使保留值－10

currentCpuLimit > 40000

调低虚拟机限制值，使限制值－10(最低调整到40)

currentAnalysor.cpuFreeRanking -= 2;空闲等级减2

currentCpuReserve > 0

bool ret = dynamicAdjustment.AdjustCPUReservation(currentVM, currentCpuReserve - 10000); 将虚拟机保留调低10%

cpuAnalysorrDict[currentVM].cpuFreeRanking += 1; 空闲等级＋1

ret

返回

否

否

是

否

是

是

是

是

否

否

### 主控模块

#### 功能

本模块启动性能接收模块接收域内的性能信息，启动虚拟机状态监控模块监控虚拟机开、关机等状态信息，初始化负载分析模块，对各虚拟机进行负载均衡调节。

#### 设计

#### 异常和错误处理

#### 相关数据结构

无

#### 函数及流程图

* Main(string[] args)

【所属类及所在文件】

Program类，Program.cs文件

【功能描述】

程序main函数。

【输入参数】

string[] args，输入的命令行参数

【返回值】

无

【流程图】

开始

返回

VMState vmState = new VMState(MpcVmConfigpath);

初始化虚拟机状态监控类实例vmState

vmState.StartMessageReceiver();

调用StartMessageReceiver()函数接收多域PC发送来的虚拟机状态改变信息

!File.Exists(MpcVmConfigpath)

输出：控制台输出配置文件不存在

string HvAddr = "hypervnb://00000000-0000-0000-0000-000000000000/C7240163-6E2B-4466-9E41-FF74E7F0DE47";前部分都是0表示接收所有虚拟机发来的连接请求，后半部分为注册表中注册的可信程序ID。

Server detectorServer = new Server(HvAddr);

输入负载均衡器的URI，初始化性能接收模块的服务端

是

否

var task = new Task(() =>{detectorServer.StartUpServer();})

task.Strart();

将服务器的StartUpServe()函数作为一个线程启动，接收域内发送来的性能信息。

输入参数，初始化负载均衡模块，LoadBalancer testLoadBalancer；

testLoadBalancer.setDetectorServer(detectorServer);

设置负载均衡模块的消息接收服务器

testLoadBalancer.BalanceByTime();

调用BalanceByTime()函数定时分析性能信息，必要时进行资源调节

Console.ReadLine();

是否捕获异常

输出异常消息

否

是

## 初步测试实验数据

开机时间

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| CPU核心数 | 内存大小 | 保留 | 限制 | 是否开机负载均衡器 | 开机时间 |
| 2 | 2046MB | 100 | 100 | 是 | 29s |
|  |  |  |  |  | 21.21s/20.15s |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

## 未解决的问题

1. 线程同步和互斥问题：

客户端，线程读写锁还未设置，CPU、内存探测模块写性能指标和性能信息发送模块读性能指标之前未加锁，存在潜在风险；

服务端，负载分析均衡模块，

重要参考文献：

<https://docs.microsoft.com/en-us/azure/monitoring/infrastructure-health/vmhealth-windows/winserver-memory-pagespersec>

<https://docs.microsoft.com/zh-cn/virtualization/hyper-v-on-windows/user-guide/make-integration-service#register-a-new-application>).



