# 概述

has对用户的接口API函数均在has.cpp文件中实现。分为5个部分：

1. Init/Shutdown：初始化和关闭HSA
2. System: 获得HAS系统环境信息
3. Agent: 主要包括Queue和cache操作
4. Memory
5. Signal

HAS总体由**core::Runtime**类管理。该类在*src/core/inc/runtime.h*和*src/core/runtime/runtime.cpp*文件中定义。该类为单例，创建的单例对象为**runtime\_singleton\_**。

HAS的Init和System部分的功能，都由此类实现。此外Runtime类还包括：

|  |  |  |
| --- | --- | --- |
| 变量名 | 类型 | 说明 |
| gpu\_agents\_ | vector<Agent\*> | GPU设备 |
| link\_matrix\_ | vector<LinkInfo> | IO链接信息 |
| loader\_ | Loader\* |  |
| flag\_ | Flag |  |
|  |  |  |

# 初始化

API初始化函数**has\_init()**会调用**core::Runtime::Acquire()**函数。

core::Runtime::Acquire()函数根据引用计数，创建core::Runtime单例**runtime\_singleton\_**。并且在第一次创建实例时调用runtime\_singleton\_->Load()函数。

**core::Runtime::Load()**函数：

1. flag
2. Agent和拓扑关系初始化：amd::Load()
3. BindVmFaultHandler()
4. Creat loader\_
5. LoadExtensions()
6. foreach agent: PostToolsInit
7. LoadTools

## flag

flag初始化负责读取系统环境变量，以确定HAS配置。flag为Flag类，在*src/core/util/flag.h*文件定义。包括interrupt使能、sdma使能、flat\_scrath配置等。这些flag包括如下：

|  |  |
| --- | --- |
| HSA\_CHECK\_FLAT\_SCRATCH |  |
| HSA\_ENABLE\_VM\_FAULT\_MESSAGE |  |
| HSA\_ENABLE\_QUEUE\_FAULT\_MESSAGE |  |
| HSA\_ENABLE\_INTERRUPT |  |
| HSA\_ENABLE\_SDMA |  |
| HSA\_RUNNING\_UNDER\_VALGRIND |  |
| HSA\_SDMA\_WAIT\_IDLE |  |
| HSA\_MAX\_QUEUES | 设置Queue的最大个数，如果没有定义HAS\_LARGE\_MODEL（非64位系统）, 最大个数为10，否则最大个数为128 |
| HSA\_SCRATCH\_MEM |  |
| HSA\_TOOLS\_LIB |  |
| HSA\_TOOLS\_REPORT\_LOAD\_FAILURE |  |
| HSA\_DISABLE\_FRAGMENT\_ALLOCATOR |  |
| HSA\_ENABLE\_SDMA\_HDP\_FLUSH |  |
| HSA\_REV\_COPY\_DIR |  |
|  |  |

## amd::Load

此时进入amd空间，amd空间在*src/core/runtime/amd\_xxxx.c*文件中描述。

1. 打开KFD设备
2. 建立拓扑结构
   1. 发现Agent
      1. 创建amd::GpuAgent或者amd::CpuAgent对象
      2. 注册Agent对象
   2. 注册link

该函数为amd空间的初始化，负责打开KFD驱动设备，以及建立拓扑结构表。打开驱动设备会调用驱动接口，HSA大部分工作为建立拓扑表。

建立拓扑结构部分在*/src/core/runtime/amd\_topology.cpp*文件中描述。

该函数使用**amd::BuildTopology()**函数建立拓扑结构表（发现并注册系统中HAS硬件）。建立拓扑结构表主要负责两件事情：注册Agent和注册link信息。该函数先获得HAS系统的属性，该属性包含该函数利用驱动层的API函数遍历系统中的每个node（连接在系统中的HAS硬件——CPU或者GPU），根据该node的属性（node的属性在驱动层定义），将其创建为**amd::CpuAgent**或者**amd::GpuAgent**对象并注册到core::Runtime单例。创建Agent使用**amd::DiscoverCpu()**和**amd::DiscoverGpu()**函数。然后使用**amd::RegisterLinkInfo()**函数将该node的连接信息注册到core::Runtime单例。

**amd::GpuAgent**类在*src/core/inc/amd\_gpu\_agent.h*和*/src/core/runtime/amd\_gpu\_agent.cpp*文件中描述。类的继承关系如下：amd::GpuAgent 🡪 amd::GpuAgentInt 🡪 core::Agent 🡪 core::Checked<>。其中**core::Agent**类为HAS定义的Agent接口基类，各个厂商的设备要基于该类，AMD创建的接口类为**amd::GpuAgentInt**。接口类除了Convert, node\_id, device\_type和public\_handle外，都必须为虚函数。

2.a

**amd::DiscoverGpu()**函数负责两件事情：创建**amd::GpuAgent**对象和注册该对象。

2.a.i

amd::GpuAgent的构造函数获得并设置硬件信息，包括设置isa、设置queue最大最小个数等。然后amd::GpuAgent::InitRegionList()和amd::GpuAgent::InitCacheList()。

**InitRegionList()**根据节点的properties信息，来填充HSA可见的内存区域**regions\_**，然后记录内存位宽和频率。mem类型包括HSA\_HEAPTYPE\_FRAME\_BUFFER\_PRIVATE为GPU上对CPU不可见的内存区域，HAS\_HEAPTYPE\_FRAME\_BUFFER\_PUBLIC为GPU上对CPU可见的内存区域。

**InitCacheList()**函数根据节点信息，填充cache信息**cache\_props\_**和**caches\_**。前者填充的为GPU定义的cache信息，后者填充的为HAS定义的cache信息。

2.a.ii

创建完agent对象后，使用runtime单例的**runtime\_singleton\_**->**RegisterAgent()**函数。该函数会将agent填充进runtime\_singletion\_的**agents\_by\_node\_**队列、**gpu\_agents\_**队列和**gpu\_ids\_**队列。

2.b

注册link信息使用**amd::RegisterLinkInfo()**函数。该函数通过驱动接口，获得节点的所有link的属性信息，然后遍历所有接口，填充has\_amd\_memory\_pool\_link\_info\_t的变量，包括atomic支持、link类别、带宽、延迟等。然后利用**runtime\_singleton\_**->**RegisterLinkInfo()**函数为该节点注册链接信息到runtime\_singleton\_的**link\_matrix\_**变量。link\_matrix\_的元素的个数为系统中节点数量。

## 绑定VmFault处理

绑定VMFault处理程序，使用的是**Runtime::BindVmFaultHandler()**函数。

## 创建loader\_

这一步主要创建Runtime类中的loader\_变量和loader\_context\_变量。调用了**amd::has::loader::Loader::Create()**函数

**loader\_**变量是**amd::hsa::loader::Loader**类。该类在/src/core/inc/amd\_hsa\_loader.hpp文件中定义。调用Loaer的Create()函数时，会创建并返回一个**amd::has::loader::AmdHsaCodeLoader**类型。

**loader\_context\_**变量是amd::LoaderContext类型。该类继承自has::loader::Context类型。loaer\_context\_变量在创建loader\_变量时需要被传入，即创建Loaer类型时，需要传入Context类型的变量，用来创建AmdHsaCodeLoader类型。

## 加载扩展库

## PostToolInit

## 加载指定工具

# Agent

has中的Agent基类为**core::Agent**，该类在*/src/core/inc/agent.h*文件中。这是一个接口类，定义了Agent需要满足的操作。包括DMA操作、队列创建、MemoryRegion列表等。

厂商定义的所有Agent类都需要派生自该类。amd定义的GpuAgent类的派生关系如下：**amd::GpuAgent** 🡪 **amd::GpuAgentInt** 🡪 core::Agent。这两个类在*/src/core/inc/amd\_gpu\_agent.h*和*/src/core/runtime/amd\_gpu\_agent.cpp*两个文件中。其中GpuAgentInt为厂商定义的Agent接口，GpuAgent为Gpu实际的节点类。

# MemoryRegion

has中的mem基类为**core::MemoryRegion**，该类在*/src/core/inc/memory\_region.h*文件中定义。

厂商定义的mem类需要派生自该类，amd定义的mem类为**amd::MemoryRegion**。该类在*/src/core/inc/amd\_memory\_region.h*和*/src/core/runtime/amd\_memory\_region.cpp*文件中定义。

一个Agent包含的所有memory region会实例化该类，作为Agent类中的内存队列变量**regions\_**。

Agent中的内存列表regions\_是在Agent构造时，根据驱动获得的Agent属性，利用**amd::GpuAgent::InitRegionList()**函数填充。

## 创建mem region对象

对于CPU，一个memory region就是一个Agent所看到的全部内存空间。当一个CPU有两个NUMA node，每个Node会被当做一个agent，每个node的bank就是这个agent的memory region。

对于GPU，memory region包括global memory, GDS和LDS。

memory region的类型由**HSA\_HEAPTYPE**枚举类型定义：

|  |  |
| --- | --- |
|  | 说明 |
| **HSA\_HEAPTYPE\_SYSTEM** | CPU内存 |
| **HSA\_HEAPTYPE\_FRAME\_BUFFER\_PUBLIC** | CPU可见的GPU显存 |
| **HSA\_HEAPTYPE\_FRAME\_BUFFER\_PRIVATE** | CPU不可见的GPU显存 |
| **HSA\_HEAPTYPE\_GPU\_GDS** |  |
| **HSA\_HEAPTYPE\_GPU\_LDS** |  |
| **HSA\_HEAPTYPE\_GPU\_SCRATCH** |  |
| **HSA\_HEAPTYPE\_DEVICE\_SVM** | sys-memory mapped by device page tables |
| **HSA\_HEAPTYPE\_MMIO\_REMAP** | 重映射的memory io(读写的寄存器) |

其中HAS\_HEAPTYPE\_FRAME\_BUFFER\_PRIVATE是仅在渲染中使用的显存，在计算任务中，所有显存应为CPU可见的，即HAS\_HEAPTYPE\_FRAME\_BUFFER\_PUBLIC。但经实验发现目前返回的显存类型为private。

## 分配memory空间

不管GPU还是CPU的内存分配，均使用**hsa\_memory\_allocate()**函数，该函数在<has.cpp>中定义。该函数调用**core::Runtime::runtime\_singleton\_->AllocateMemory()**函数，该函数在<runtime.cpp>中定义。该函数调用**region->Allocate()**函数，最终实现内存分配。该函数在<amd\_memory\_region.cpp>中定义。

### memory申请位置

GPU和CPU上memory的分配都会调用相同的函数，因此用户需要传入memory region参数以告诉HAS是在那个region上申请。对于CPU，只能在HAS\_HEAPTYPE\_SYSTEM上申请，也就是系统内存空间。对于GPU，只能在HAS\_HEAPTYPE\_FRAME\_BUFFER\_PUBLIC和HAS\_HEAPTYPE\_FRAME\_BUFFER\_PRIVATE上申请，也就是只能在global memory显存上申请。

### memory申请大小

memory分配的大小会向上对齐到page大小，比如page是4KB，则只会分配4KB,8KB,12KB,…大小的memory。

### memory申请步骤

memory分配分为两个步骤：分配memory和map到GPU节点上。两个步骤分别调用kmt的**hsaKmtAllocMemory()**函数和**hsaKmtMapMemoryToGPUNodes()**函数。

这两个步骤分别需要两个配置参数：**HsaMemFlags**，**HsaMemMapFlags**和。这两个参数用来配置memory属型和map属性，分别在<hsakmttypes.h>中定义，在memory region对象创建时会被初始化。这两个参数对于用户是不开放的，因此region->Allocate()函数接受用户传入一个**AllocateFlags**参数来配置上述两个参数。但是这个参数目前没有开放给用户，传入的为0。

在map到GPU节点这个步骤时，CPU端分配好的内存，会map到所有GPU node上，而在GPU上分配的显存只会map到自己的node上。

# 队列

has中队列的基类为**core::Queue**，该类在*/src/core/inc/queue.h*文件中定义。这是一个接口类，只定义了队列需要支持的操作。厂商定义的队列类都需要派生自core::Queue。

amd定义的队列类为**amd::AqlQueue**类，在*src/core/inc/amd\_aql\_queue.h*和*src/core/runtime/amd\_aql\_queue.cpp*文件中定义。

队列中所有的packet的结构均由**core::AqlPacket**结构体定义。包的大小为64字节。共有四种类型的包结构：

* kernel dispatch packet
* agent dispatch packet
* barrier and packet
* barrier or packet

队列的大小为1KB~4GB之间，或者4KB~8MB之间。这里源码中有冲突。

has创建的队列和硬件创建的队列资源，通过AqlQueue的**amd\_queue\_**成员变量进行映射。该变量为**amd\_queue\_t**类型结构体，在amd\_hsa\_queue.h文件中定义。驱动层创建队列后，会返回携带该队列信息的结构体，需要用该结构体对amd\_queue\_进行赋值。

## 创建队列

### 队列大小

创建队列需要首选确定队列的大小。队列的最大大小为128KB，由**GpuAgent**类的**maxAqlSize\_**常数变量确定。最小大小为4KB，由**minAqlSize\_**确定。这里4K~128K怀疑时AqlPacket的个数。

在创建AqlQueue对象时也会限制Queue的大小范围。为1KB ~ 4GB。

queue\_size\_pkts是申请的AqlPacket个数

queue\_size\_bytes是申请的byte数

### 创建队列对象

创建队列需要调用has的**HAS::has\_queue\_create()**函数。该函数在*src/core/runtime/has.cpp*文件中定义。 该函数将会调用对应的agent的QueueCreate()函数来创建队列。

**amd::GpuAgent::QueueCreate()**函数在*src/core/runtime/amd\_gpu\_agent.cpp*文件中定义，该函数将创建并返回**AqlQueue**类型的对象。

创建AqlQueue对象有以下步骤：

1. 计算并为ring\_buffer分配空间。这个步骤会调用runtime单例的**system\_allocator()**函数，这个函数最终由amd\_hsa\_memory\_region负责实现。
2. 调用驱动接口，创建硬件队列。然后将创建好的硬件队列的信息赋值给**amd\_queue\_**变量

这里主要调用roct的hsaKmtCreateQueue()接口

1. 初始化Scratch：InitScratchSRD()

队列

创建队列使用hsa\_queue\_create()函数，获得hsa\_queue\_t结构体变量。

该函数调用agent的QueueCreate()函数，获得core::Queue类型的对象指针，转换成has\_queue\_t结构体返回给用户。

该函数会创建AqlQueue类型对象，AqlQueue继承自Queue类型。

AqlQueue类的构造函数会调用hsaKmtCreateQueue()。KMT会分配ring\_buff和queue。

AqlQueue类的构造函数会调用AllocRegisteredRingBuffer(),创建ring buffer。

AllocRegisteredRingBuffer()函数会调用runtime单例的system\_allocator()函数为ring\_buf\_分配空间。

AqlQueue在构造函数负责为自身的amd\_queue\_t结构体类型变量amd\_queue\_赋值。同时为ring\_buf\_分配空间。

信号

创建信号使用hsa\_signal\_create()函数，获得hsa\_signal\_t结构体类型的变量。

该函数会调用AMD::hsa\_amd\_signal\_create()函数。

该函数会创建core::InterruptSignal类的对象。然后转换为hsa\_signal\_t类型返回给用户。

InterruptSignal类继承自LocalSignal和Signal类。

代码对象

创建代码对象使用hsa\_code\_object\_deserialize()函数，获得hsa\_code\_object\_t结构体类型变量。

实际即使将代码申请空间，然后返回内存地址。

执行对象

创建执行对象使用hsa\_executable\_create()函数，获得hsa\_executable\_t结构体类型变量。

该函数调用HSA::hsa\_executable\_create\_alt()函数，

该函数会调用runtime单例的loader的CreateExecutable()函数，获得Executable类型对象，转换成has\_executable\_t地址，返回给用户。

CreateExecutable()函数会创建ExecutableImpl类型对象，并转换成Executable类型指针给上层。

# 其他

## 层次关系

OpenCL/hip

rocr->HAS -> libhsa\_

roct->Thunk -> libhsakmt.so

rock->Driver

## 项目地址

HSA:https://github.com/RadeonOpenCompute/ROCR-Runtime

Thunk:https://github.com/RadeonOpenCompute/ROCT-Thunk-Interface

## 编译

需要安装libpci：

apt-get install libpciaccess-dev

apt-get install pciutils-dev

hsa编译时会include /opt/rocm/libhsakmt/include/libhsakmt/下的文件，如果hsa与其版本不一致，则可能导致常量未定义，比如HSA\_IOLINK\_TYPE\_XGMI未定义。此时需要把/home/feifei/ROCR-Runtime/src/core/runtime/amd\_topology.cpp文件中HSA\_IOLINK\_TYPE\_XGMI相关case注释掉即可

cd src

mkdir build

cd build

cmake ..

make

## 使用

include: /home/feifei/ROCR-Runtime/src/inc/

link: /home/feifei/ROCR-Runtime/src/build/

library: hsa-runtime64