使用最广jvm sun hotspot vm

运行时数据区

1. 程序计数器：线程所执行的字节码的行号指示器（**线程私有**）
2. Java虚拟机栈：**线程私有，生命周期与线程相同**，每个方法在执行的同时会在其中创建**栈帧**存储：**局部变量表**、操作数、动态链表、方法出口（调用压栈）。其中局部变量包括存放**基本数据类型、对象引用、和returnaddress，**局部变量表所需的内存空间在**编译期间分配完成，其空间不会在运行期间改变。**
3. 本地方法栈，与2类似，不过是提供给native（jni）方法服务的
4. Java堆：所有线程共享，几乎所有对象和全部数组都要在堆上分配内存，垃圾管理器主要管理的也是堆
5. 方法区：**存储类信息、常量、静态变量、和jit代码**（堆的逻辑部分），其中的**运行常量池**存放编译期生成的字面量和符号引用（符号引用是一个字符串，它给出了被引用的内容的名字­­————此处指类名）（运行期也可能将常量放入池中）

对象的创建：

1. new之后根据类名去常量池定位到符号引用，并检查该引用是否被加载、解析初始化过、未加载则加载
2. 分配内存（空闲列表或指针碰撞）
3. 内存空间初始化为0
4. 虚拟机对对象设置（如对象是哪个类的实例），将信息存放在objectheader里
5. Init

Oom：（out of memory）

堆溢出：部分对象生命周期过长，导致它引用的对象也无法释放造成恶性拓展。

类加载过程

拓展栈时无法申请到足够空间：创建过多线程

方法区和运行时常量池溢出

Sof：（stack over flow）

栈深度大于虚拟机允许最大深度

每个**方法在执行**的同时都会创建一个**栈帧**，每一个方法从调用直至执行完成的过程，就对应着一个**栈帧在虚拟机栈中入栈到出栈**的过程，错误**递归**则是一直压栈，超过栈的最大速度

**Javac与jit**

Javac是编译成字节码，运行时还要一句一句转义，jit是编译成机器码

线程状态

存储区域

**垃圾回收**

一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。



如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行finalize（）方法。当对象没有覆盖finalize（）方法，或者finalize（）方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行finalize（）方法，那么这个对象将会放置在一个叫做F-Queue的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。如果对象要在finalize（）中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上它就真的被回收了（和不需要执行finalize的一起，finalize只会执行一次只能逃一次）。

**Gc算法**

最基础的收集算法是“**标记-清除**”（Mark-Sweep）算法，如同它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。(问题：产生碎片)

**停止-复制**：

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了。将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存会被“浪费”。

**Java的类型加载、连接、和初始化都是在程序运行区间完成的。**

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）7个阶段。其中验证、准备、解析3个部分统称为连接（Linking）



对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

**加载：**

1）通过一个类的全限定名来获取定义此类的二进制字节流。

2）将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3）在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

对于数组类而言，情况就有所不同，数组类本身不通过类加载器创建，它是由Java虚拟机直接创建的。

**验证：**

验证是连接阶段的第一步，验证是虚拟机对自身保护的一项重要工作

1.文件格式验证

2.元数据验证（对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求）

3.字节码验证 （通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的）

4.符号引用验证（符号引用中通过字符串描述的全限定名是否能找到对应的类及其方法字段）

**准备：**

准备阶段是正式为**类变量**（非对象变量--static）分配内存并设置类变量初始值的阶段

public static int value=123； 那变量value在准备阶段过后的初始值为0而不是123

**解析：**

解析阶段是虚拟机将常量池内的**符号引用替换为直接引用**的过程

符号引用（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。

直接引用（Direct References）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。

**虚拟机规范之中并未规定解析阶段发生的具体时间**

**初始化：**

到了初始化阶段，才真正开始执行类中定义的Java程序代码（或者说是字节码）。在准备阶段，变量已经赋过一次系统要求的初始值，初始化阶段是执行类构造器＜clinit＞（）方法的过程

＜clinit＞（）方法是由编译器自动收集类中的所有类变量的赋值动作（不包括方法与构造函数中）和静态语句块（static{}块）中的语句合并产生的

**类加载器：**

通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放**到Java虚拟机外部**去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“类加载器”。（外部是指不通过jvm自身的操作，可以是lang包里的程序，可以自己编写。不要过度理解）

示例：

protected synchronized Class＜?＞loadClass（String name,boolean resolve）throws ClassNotFoundException

{

//首先，检查请求的类是否已经被加载过了

Class c=findLoadedClass（name）；

if（c==null）{

try{

if（parent！=null）{

c=parent.loadClass（name,false）；

}else{

c=findBootstrapClassOrNull（name）；

}

}catch（ClassNotFoundException e）{

//如果父类加载器抛出ClassNotFoundException

//说明父类加载器无法完成加载请求

}

if（c==null）{

//在父类加载器无法加载的时候

//再调用本身的findClass方法来进行类加载

c=findClass（name）；

}

}

if（resolve）{

resolveClass（c）；

}

return c；

}

Nio

Java的类型加载、连接、和初始化都是在程序运行区间完成的。

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）7个阶段。其中验证、准备、解析3个部分统称为连接（Linking）



对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

SuperClass[]sca=new SuperClass[10]；

加载顺序

**加载：**

1）通过一个类的全限定名来获取定义此类的二进制字节流。

2）将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3）在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

对于数组类而言，情况就有所不同，数组类本身不通过类加载器创建，它是由Java虚拟机直接创建的。

**验证：**

验证是连接阶段的第一步，验证是虚拟机对自身保护的一项重要工作

1.文件格式验证

2.元数据验证（对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求）

3.字节码验证 （通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的）

4.符号引用验证（符号引用中通过字符串描述的全限定名是否能找到对应的类及其方法字段）

**准备：**

准备阶段是正式为**类变量**（非对象变量--static）分配内存并设置类变量初始值的阶段

public static int value=123； 那变量value在准备阶段过后的初始值为0而不是123

**解析：**

解析阶段是虚拟机将常量池内的**符号引用替换为直接引用**的过程

符号引用（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。

直接引用（Direct References）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。

**虚拟机规范之中并未规定解析阶段发生的具体时间**

**初始化：**

到了初始化阶段，才真正开始执行类中定义的Java程序代码（或者说是字节码）。在准备阶段，变量已经赋过一次系统要求的初始值，初始化阶段是执行类构造器＜clinit＞（）方法的过程

＜clinit＞（）方法是由编译器自动收集类中的所有类变量的赋值动作（不包括方法与构造函数中）和静态语句块（static{}块）中的语句合并产生的

**类加载器：**通过一个类的全限定名来获取描述此类的二进制字节流”这个动作放**到Java虚拟机外部**去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“类加载器”。（外部是指不通过jvm自身的操作，可以是lang包里的程序，可以自己编写。不要过度理解）

示例：

protected synchronized Class＜?＞loadClass（String name,boolean resolve）throws ClassNotFoundException

{

//首先，检查请求的类是否已经被加载过了

Class c=findLoadedClass（name）；

if（c==null）{

try{

if（parent！=null）{

c=parent.loadClass（name,false）；

}else{

c=findBootstrapClassOrNull（name）；

}

}catch（ClassNotFoundException e）{

//如果父类加载器抛出ClassNotFoundException

//说明父类加载器无法完成加载请求

}

if（c==null）{

//在父类加载器无法加载的时候

//再调用本身的findClass方法来进行类加载

c=findClass（name）；

}

}

if（resolve）{

resolveClass（c）；

}

return c；

}