**第一章 HotSpot源码结构**

├─agent                            Serviceability Agent的客户端实现

├─make                             用来build出HotSpot的各种配置文件

├─src                              HotSpot VM的源代码

│  ├─cpu                            CPU相关代码（汇编器、模板解释器、ad文件、部分runtime函数在这里实现）

│  ├─os                             操作系相关代码

│  ├─os\_cpu                         操作系统+CPU的组合相关的代码

│  └─share                          平台无关的共通代码

│      ├─tools                        工具

│      │  ├─hsdis                      反汇编插件

│      │  ├─IdealGraphVisualizer       将server编译器的中间代码可视化的工具

│      │  ├─launcher                   启动程序“java”

│      │  ├─LogCompilation             将-XX:+LogCompilation输出的日志（hotspot.log）整理成更容易阅读的格式的工具

│      │  └─ProjectCreator             生成Visual Studio的project文件的工具

│      └─vm                           HotSpot VM的核心代码

│          ├─adlc                       平台描述文件（上面的cpu或os\_cpu里的\*.ad文件）的编译器

│          ├─asm                        汇编器接口

│          ├─c1                         client编译器（又称“C1”）

│          ├─ci                         动态编译器的公共服务/从动态编译器到VM的接口

│          ├─classfile                  类文件的处理（包括类加载和系统符号表等）

│          ├─code                       动态生成的代码的管理

│          ├─compiler                   从VM调用动态编译器的接口

│          ├─gc\_implementation          GC的实现

│          │  ├─concurrentMarkSweep      Concurrent Mark Sweep GC的实现

│          │  ├─g1                       Garbage-First GC的实现（不使用老的分代式GC框架）

│          │  ├─parallelScavenge         ParallelScavenge GC的实现（server VM默认，不使用老的分代式GC框架）

│          │  ├─parNew                   ParNew GC的实现

│          │  └─shared                   GC的共通实现

│          ├─gc\_interface               GC的接口

│          ├─interpreter                解释器，包括“模板解释器”（官方版在用）和“C++解释器”（官方版不在用）

│          ├─libadt                     一些抽象数据结构

│          ├─memory                     内存管理相关（老的分代式GC框架也在这里）

│          ├─oops                       HotSpot VM的对象系统的实现

│          ├─opto                       server编译器（又称“C2”或“Opto”）

│          ├─prims                      HotSpot VM的对外接口，包括部分标准库的**native**部分和JVMTI实现

│          ├─runtime                    运行时支持库（包括线程管理、编译器调度、锁、反射等）

│          ├─services                   主要是用来支持JMX之类的管理功能的接口

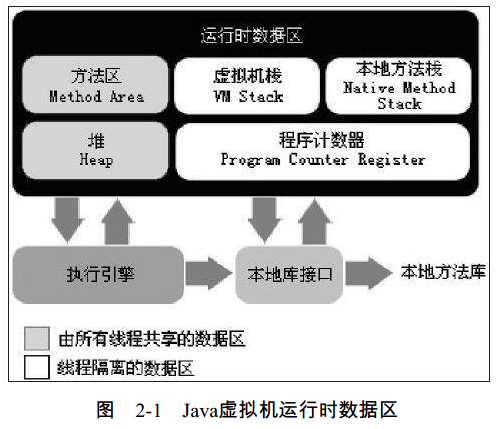
│          ├─shark                      基于LLVM的JIT编译器（官方版里没有使用）

│          └─utilities                  一些基本的工具类

└─test                             单元测试

**第二章 自动内存管理机制**

**2.1运行时数据区域**



运行时数据区域分为5个部分：虚拟机栈、本地方法栈、程序计数器、堆和方法区。

2.1.1程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看作是当前线

程所执行的字节码的行号指示器。也就是当虚拟机中的字节码解释器工作的时候会通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、 循环、 跳转、 异常处理、 线程恢复等基础功能。

在Java虚拟机中，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）都只会执行一条线程中的指令。为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器。因此，程序计数器所占有的内存区域是线程私有（线程隔离）的内存。

**注意：**如果线程正在执行的是一个Java方法，程序计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，程序计数器的值则为空（Undefined）。 程序计数器所在的内存区域是唯一一个在Java虚拟机规范中没有规定任OutOfMemoryError情况的区域。

2.2.2Java虚拟机栈

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、 操作数栈、 动态链接、 方法出口等信息。栈帧在虚拟机栈中入栈到出栈的过程就是一个java方法从开始调用到执行结束的过程。

局部变量表存放了编译期可知的各种基本数据类型（boolean、 byte、 char、 short、 int、

float、 long、 double）、 对象引用（reference类型）和returnAddress类型（指向了一条字节码指令的地址）。其中64位长度的long和double类型的数据会占用2个局部变量空间（Slot），其余的数据类型只占用1个。当进入一个方法时，这个方法需要在栈帧中分配多大的局部变量空间是完全确定的，也就是说，在方法运行期间不会改变局部变量表的大小。

**注意：**如果线程请求的虚拟机栈深度大于所允许的深度，则将抛出StackOverflowError异常；如果虚拟机栈可以动态扩展，但是无法申请到足够的内存时，就会抛出OutOfMemoryError异常。

2.2.3本地方法（Native）栈

本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是非常相似的，它们之间

的区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法服务。在虚拟机规范中对本地方法栈中方法使用的语言、 使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如Sun HotSpot虚拟机）直接就把本地方法栈和虚拟机栈合二为一。 与虚拟机栈一样，本地方法栈也是线程私有，也会抛出StackOverflowError和OutOfMemoryError异常。

2.2.4Java堆

对于大多数应用来说，Java堆（Java Heap）是Java虚拟机所管理的内存中最大的一块。

Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。 此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC堆”（Garbage Collected Heap）。 从内存回收的角度来看，由于现在收集器基本都采用分代收集算法，所以Java堆中还可以细分为：新生代和老年代；再细致一点的有Eden空间、 From Survivor空间、 To Survivor空间等。 从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer,TLAB）。 不过无论如何划分，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。 在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx和-Xms控制）。 如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

2.2.5方法区

方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、 常量、 静态变量、 即时编译器编译后的代码等数据。对于习惯在HotSpot虚拟机上开发、 部署程序的开发者来说，通常会把方法区称为“永久代”（Permanent Generation）。

Java虚拟机规范对方法区的限制非常宽松，除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。 相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。 这区域的内存回收目标主要是针对常量池的回收和对类型的卸载。当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

运行时常量池（Runtime Constant Pool）是方法区的一部分。 Class文件中除了有类的版本、 字段、 方法、 接口等描述信息外，还有一项信息是常量池（Constant Pool Table），它用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

Java虚拟机对Class文件每一部分（自然也包括常量池）的格式都有严格规定，每一个字节用于存储哪种数据都必须符合规范上的要求才会被虚拟机认可、 装载和执行，但对于运行时常量池，Java虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。 不过，一般来说，除了保存Class文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个重要特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的intern（）方法。既然运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError异常。