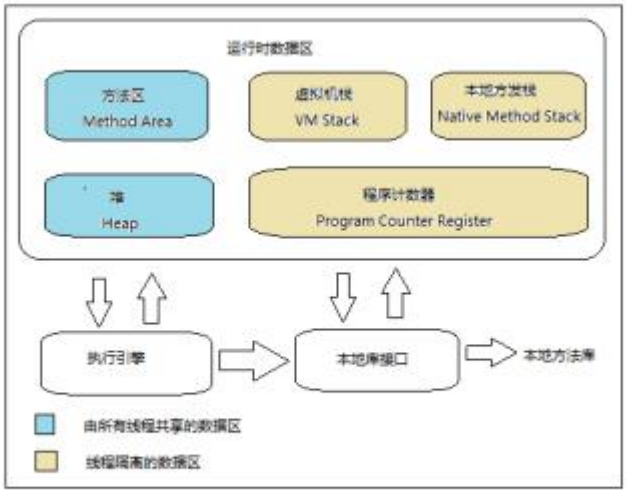
第2章 Java内存区域与内存溢出异常

2.2 运行时内存区域

Java虚拟机所管理的内存包括如下几个运行时数据区：



2.2.1 程序计数器

==程序计数器==是一块比较小的内存空间，可以看作是==当前线程所执行的字节码的行号指示器==。==字节码解释器工作时就是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都需要依赖这个计数器来完成。

Java虚拟机的多线程是通过轮流切换并分配处理器执行时间的方式实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。为了切换线程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==。==各条线程之间计数器互不影响，独立存储==，我们称这类==内存区域为“线程私有”的内存。==

执行方法	计数器中记录的值
Java方法	正在执行的虚拟机字节码指令的地址
Native方法	空

==程序计数器是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。==

2.2.2 Java虚拟机栈

==Java虚拟机栈==是==线程私有==的，它的==生命周期与线程相同。==

虚拟机栈描述的是==Java方法执行时的内存模型==：每个方法在执行的同时都会产生一个==栈帧==，用于==存储局部变量表、操作数栈、动态链接、方法出口等信息==。每一个方法从调用到执行完成的过程，就对应着一个栈帧从入栈到出栈的过程。

==局部变量表存放了编译期可知的各种基本数据类型、对象引用和returnAddress类型。== 其中，64位长的long和double类型占用两个局部变量空间、其余只占一个。==局部变量表所需的内存空间时在编译期间完成分配的==，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间时完全确定的，在==方法运行期间不会更改局部变量表的大小==。

==如果线程请求的栈深度大于虚拟机所需的深度，抛出StackOverflowError异常。如果虚拟机可以自动扩展，如果扩展过程中无法申请到足够的内存，就会抛出OutOfMemoryError异常。==

2.2.3 本地方法栈

本地方法栈和虚拟机栈发挥作用相似，区别只是==虚拟机栈为虚拟机执行Java方法服务，而本地方法栈则为虚拟机使用到的Native方法服务==。

本地方法栈也会抛出==StackOverflowError==和==OutOfMemoryError==异常。

2.2.4 Java堆

==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==，很多时候被称为“GC堆”（Garbage Collection Heap）。

==Java堆==可以处于物理上不连续的内存空间，==只要逻辑上是连续的即可==，实现时，==既可以实现成固定大小的，也可以是可扩展的，当前主流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分配实例，并且堆也无法再扩展，则抛出OutOfMemoryError。==

2.2.5 方法区

==方法区==也是==各个线程共享的内存区域==，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==。

Java虚拟机规范对方法区的限制十分宽松，==可以是逻辑连续的，无须物理连续，大小可固定可扩展==，除此之外，甚至==可以不实现垃圾收集==。

==当方法区无法满足内存分配需求时，将会抛出OutOfMemoryError异常。==

2.2.6 运行时常量池

==运行时常量池是方法区的一部分==。Class文件除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是==常量池==，==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存放。==

==当常量池无法申请到内存时将会抛出OutOfMemoryError异常。==

2.2.7 直接内存

==服务器管理员在配置虚拟机参数时，会根据实际内存设置-Xmx等配置信息，但经常会忽略直接内存，使得各个内存区域总和大于物理内存限制，从而导致动态扩展时出现OutOfMemoryError异常。==

2.3 HotSpot对象探秘

2.3.1 对象的创建

1. 虚拟机遇到一条new指令时，首先将去==检查==这个==指令的参数是否能在常量池中定位到一个类的符号引用==，并且==检查这个符号引用代表的类是否已被加载、解析和初始化过==。如果==没有==，那么必须==先执行响应的类加载过程==。
2. 在类加载检查通过后，接下来虚拟机将==为新生对象分配内存==。对象所需内存大小在类加载完毕后便可以完全确定，为对象分配空间的任务等同于吧一块确定大小的内存从Java堆中划分出来。
3. 内存分配完成后，==虚拟机==需要==将分配到的内存空间都初始化为0（不包括对象头）==。
4. 虚拟机对对象进行必要的设置，==设置对象头==。
5. 最后==执行init方法，把对象按照程序员的意愿进行初始化==。

==分配内存空间==通常具有以下两种方式：

1. 假设Java堆中内存时规整的，所有的用过的内存放在一边，未用的放在另一边，中间放着一个指针作为分界点的指示器，那么分配内存仅仅是将指示器向左右进行挪动。这种分配方式称为 ==“指针碰撞”==。
2. 如果Java堆内存空间不是规整的，虚拟机必须维护一个列表，记录哪些内存块是可用的，在分配的时候找到一块足够大的空间进行划分，并更新记录。这种方式叫做 ==“空闲列表”==。

分配内存过程中，可能存在==并发问题==，针对这种问题，存在如下方案：

1. ==对分配内存空间的动作进行同步处理-实际上虚拟机采用CAS配上失败重试的方法保证更新操作的原子性==。
2. ==把内存分配的动作按照线程划分在不同的空间之中进行==，即每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓存（TLAB）。哪个线程要分配内存，就在哪个线程的TLAB进行分配，只有TLAB被用完并分配新的TLAB时，才需要同步锁定。虚拟机是否使用TLAB，可以通过-XX:+/-UserTLAB参数来设定。

2.3.2 对象的内存布局

==对象在内存中的布局==可以分为3块区域：==对象头==、==实例数据==、==对齐填充==。

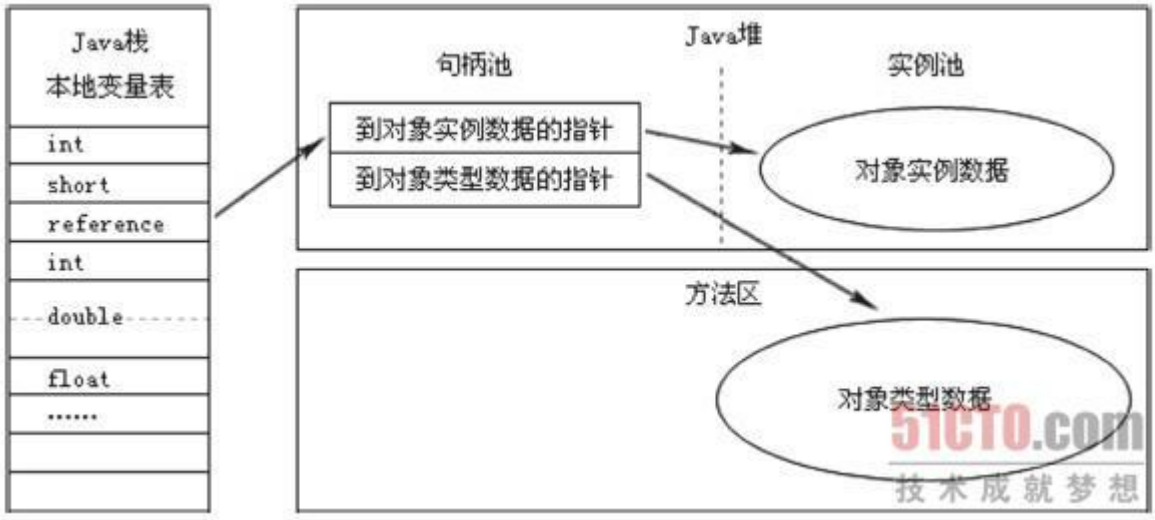
==对象头包含几部分信息==：

1. ==第一部分用于存储对象自身的运行时数据==，如哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳。
2. ==第二部分是类型指针，即对象指向它的类元数据的指针==。如果对象是一个数组，那么在对象头中还必须有一块用于记录长度的数据。
3. ==第三部分对齐填充并不是必须存在的，仅仅起到占位符的作用==。

2.2.3 对象访问定位

==通过栈中的reference查找Java对象的方式：==

1. ==使用句柄==。Java堆中会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄地址包含了对象实例数据和类型数据各自的具体地址信息。



2. ==使用直接地址访问==，那么Java堆对象的布局中必须考虑如何访问类型数据的相关信息，而reference中存储的直接就是对象地址。

优势：==使用句柄的优势是对象被移动时只用更新句柄，而不用更新reference。使用直接指针的优势是速度更快，节省了一次指针定位的时间开销。==

2.4 实战：OutOfMemoryError异常

2.4.1 Java堆溢出

==将堆的最小值-Xms参数与最大值-Xmx参数设置为一样可以避免自动扩展。通过参数-XX：+HeapDumpOnOutOfMemoryError可以让虚拟机在出现内存溢出异常时Dump出当前的内存转储快照。==

==Java堆内存溢出==时，异常堆栈信息“java.lang.OutOfNenoryError”会跟着进一步提示 =="Java heap space"==。

2.4.2 虚拟机栈和本地方法栈溢出

在==单个线程==下，无论是由于栈帧太大还是虚拟机栈容量太小，当内存无法分配时，虚拟机抛出的都是==StackOverFlowError异常==。

2.4.3 方法区和运行时常量池溢出

==运行时常量池溢出==，在OutOfMemoryError后跟随的提示信息是“==PermGen space==”。

2.4.4 本机直接内存溢出

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError
    at sun.misc.Unsafe.allocateMemory(Native Method)
    ...
```

总结

1. Java虚拟机所管理的内存包括的运行时数据区域：方法区、堆、虚拟机栈、本地方法栈、程序计数器。
2. 线程私有的数据区域：程序计数器、虚拟机栈、本地方法栈。线程共享的：方法区、Java堆
3. 各个运行时数据区域产生异常类型：

数据区域	StackOverflowError	OutOfMemoryError
程序计数器	×	×
虚拟机栈	√	√
本地方法栈	√	√
Java堆	×	√
方法区	×	√
运行时常量池	×	√
直接内存	×	√

4.对象创建过程：