一、内存区域与内存溢出异常

运行时数据区

程序计数器

- 是线程私有的；可看做当前线程执行字节码的行号指示器；

虚拟机栈

- 是线程私有的；描述java方法执行的线程内存模型，每个方法被执行时都会在虚拟机栈中创建一个栈帧，用于存放局部变量表、动态链接、方法出口等信息；

本地方法栈

- 是线程私有的；为本地native方法提供服务（HotSpot虚拟机不区分虚拟机栈和本地方法栈）；

堆空间

- 是线程共享的；唯一目的就是存放对象，几乎存放了所有对象（部分对象会通过即时编译器的逃逸分析将其分配在虚拟机栈的栈帧上），不过从内存分配的角度看，可以将堆空间划分为多个线程私有的线程分配缓冲区；

方法区

- 是线程共享的；用于存放已被JVM加载的常量、静态变量、即时编译器编译后的代码缓存等信息；

运行时数据区延伸部分

运行时常量池

- 属于方法区的一部分，用于存放编译期间产生的字面量和符号引用；

字面量：

- 接近java语言层面的概念，像文本字符串、使用final修饰的常量值都属于字面量；

符号引用：

- 接近编译原理的概念，java中的符号引用包括：
- 1.类或接口的全限定名；
- 2.字段的简单名称和描述符；
- 3.方法的简单名称和描述符；
- 4.方法句柄和方法类型；
- 5.动态常量和调用点限定符；
- 6.JDK9中涉及的被模块导出或开放的包；

直接内存

- 直接内存并不属于运行时数据区的一部分，不过对直接内存的频繁IO也可能导致OOM；
- 在JDK1.4中引入了一种基于通道与缓冲区的技术，通过native堆分配堆外内存，然后通过一个存储在java堆上的DirectByteBuffer对象作为这块内存的引用，避免了数据在native堆和java堆之间来回复制，以此显著提高性能；

对象

对象的内存布局

对象头

- 用于存储对象自身的运行时数据信息，包括对象hashcode、GC分代年龄、锁状态标志、是否处于偏向模式、线程持有的锁、偏向线程id、偏向时间戳等信息；
- 对象的类型指针，JVM通过该指针确定对象属于哪个类；
- 如果对象是一个数组，对象头中还有一个属性用于保存数组的长度；

实例数据

- 用于记录对象所属类及所继承父类中各个成员变量的具体值；

对齐填充部分

- 没有实际意义，只是为了保证任何对象都是8字节的整数倍；

对象的创建过程

从JVM的角度出发：

- 从JVM的角度出发，JVM遇到一个new字节码指令，首先检查该指令参数能否在常量池中定位到一个类的符号引用，再去检查符号引用所代表的类是否已被加载、解析和初始化，如果没有，则执行相应的类加载过程；
- 接下来为该新生对象分配内存空间。基于java堆是否规整，划分为两种分配方式：1、空闲列表；2、指针碰撞。为了解决空闲列表分配方式时产生的并发问题，又有两种解决方案：1、将线程共享的对空间划分为多个线程私有的线程分配缓冲区；2、采用CAS加失败重试的机制保证更新操作的原子性（这也是HotSpot虚拟机所采用的解决方案）；
- 对该对象进行必要的值设置，将对象的数据部分、对其填充部分用0填充，并对对象的对象头进行简单的值设置。至此，JVM完成了一个对象的创建。

从程序角度出发

- 执行class文件中的init方法，按照程序员的意愿初始化。一个真正可用的对象被构造出来；

对象的访问定位

句柄访问

- 在堆空间中专门划分一块空间作为句柄池，reference中存放的就是稳定的句柄地址，句柄中包含了对象的数据和类型各自具体的地址信息；

优：

- reference中存放的是稳定的句柄地址，对象在对空间中移动时，只需修改句柄中对象的数据地址信息即可；

缺：

- 多了一次访问定位的消耗；

直接指针

- reference中存放的直接就是对象的地址信息。

优：

- 简单高效；

缺：

- 新生代中的对象移动是非常频繁的，需要频繁修改reference存储的对象地址信息；

问题：

为什么为每个线程分配的栈的内存空间越大，越容易导致OOM的发生？

知识点一：

- 关于虚拟机栈和本地方法栈。《JAVA虚拟机规范》为本地方法栈和虚拟机栈定义了两种异常： 1、如果线程申请的栈深度超过JVM所允许的深度，则会抛出StackOverFlowError异常； 2、如果栈容量支持动态扩展，在扩展栈容量时，因无法申请到足够的内存空间，则抛出OutOfMemory异常； 在HotSpot虚拟机中，既不区分本地方法栈和虚拟机栈，也不支持栈容量动态扩展，所以在线程运行期间只可能抛出StackOverFlowError异常，除非在创建线程时因无法申请到足够的内存空间才会抛出OOM异常；

知识点二：

- 关于操作系统。OS为进程分配的内存空间是有上限的，即OS为JVM分配的内存空间是有上限的。而JVM中的堆空间和方法区可以人为的控制大小、程序计数器几乎不占用内存空间。用OS分配给JVM的内存空间 - 堆空间 - 方法区 - 程序计数器 - JVM自身运行所消耗的内存空间，本地方法栈和虚拟机栈的内存空间大小也就确定了。

- 如果为每条线程分配的栈的内存空间越大，在HotSpot虚拟机中，JVM所能创建的线程数量也就越小，一旦创建线程数量超过JVM的阈值，就会在创建线程时因无法申请到足够的内存空间而抛出OOM异常；

上述代码在JDK6以及之前，flag为false；在JDK7及之后，flag为true，为什么？

```
1  StringBuilder str = new StringBuilder("java");
2  boolean flag = str.intern() == str;
```

知识点一：

- String.intern()方法的作用是，如果在字符串常量池中已经有一个等于此String对象的字符串，则会返回字符串常量池中代表这个字符串的String对象的引用；否则会将该String对象包含的字符串添加到常量池中，然后返回此String对象的引用；

知识点二：

- 关于方法区、永久代、元空间。方法区属于运行时数据区的一部分，是一个概念模型；而永久代、元空间则是对这个概念模型的具体实现方案。JDK6及之前，使用永久代实现方法区，此时静态变量、字符串常量池都保存在方法区中；JDK7中，使用本地内存逐步代替方法区，将静态变量、字符串常量池从方法区移入到堆空间；从JDK8开始，使用元空间实现方法区，彻底废弃永久代；

- 对于java代码，在JDK6及之前，字符串常量池保存在方法区中，intern方法会把首次遇到的字符串实例保存到方法区中，而str对象保存在堆空间中，所以flag为false； 在JDK7及之后，字符串常量池保存在堆空间中，intern方法只需在常量池中记录下首次出现的实例引用即可，故flag为true；