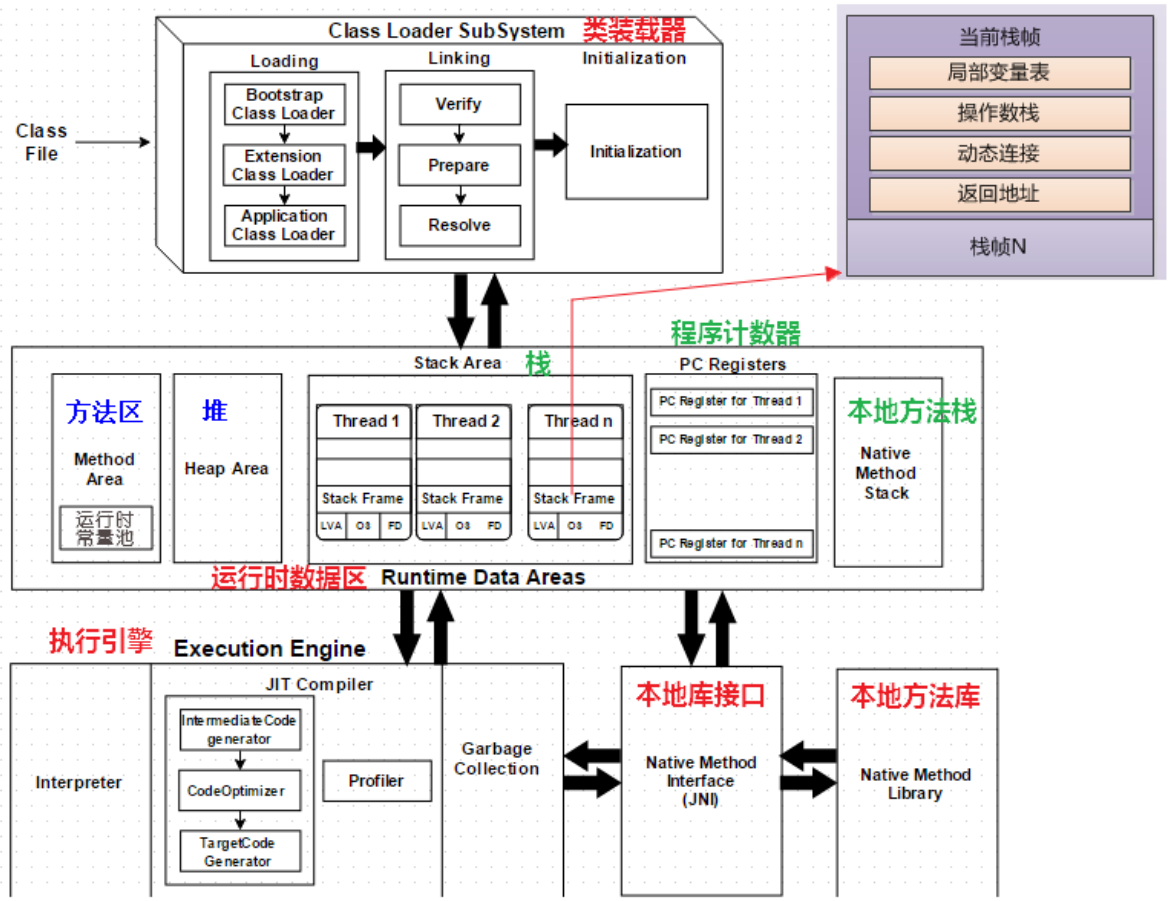
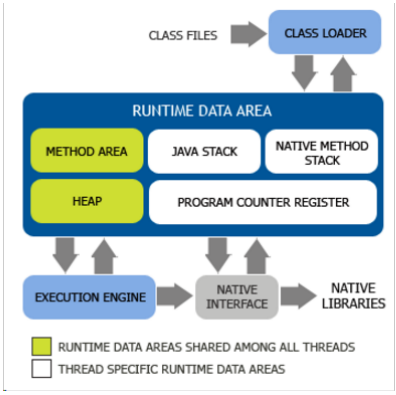
JVM虚拟机



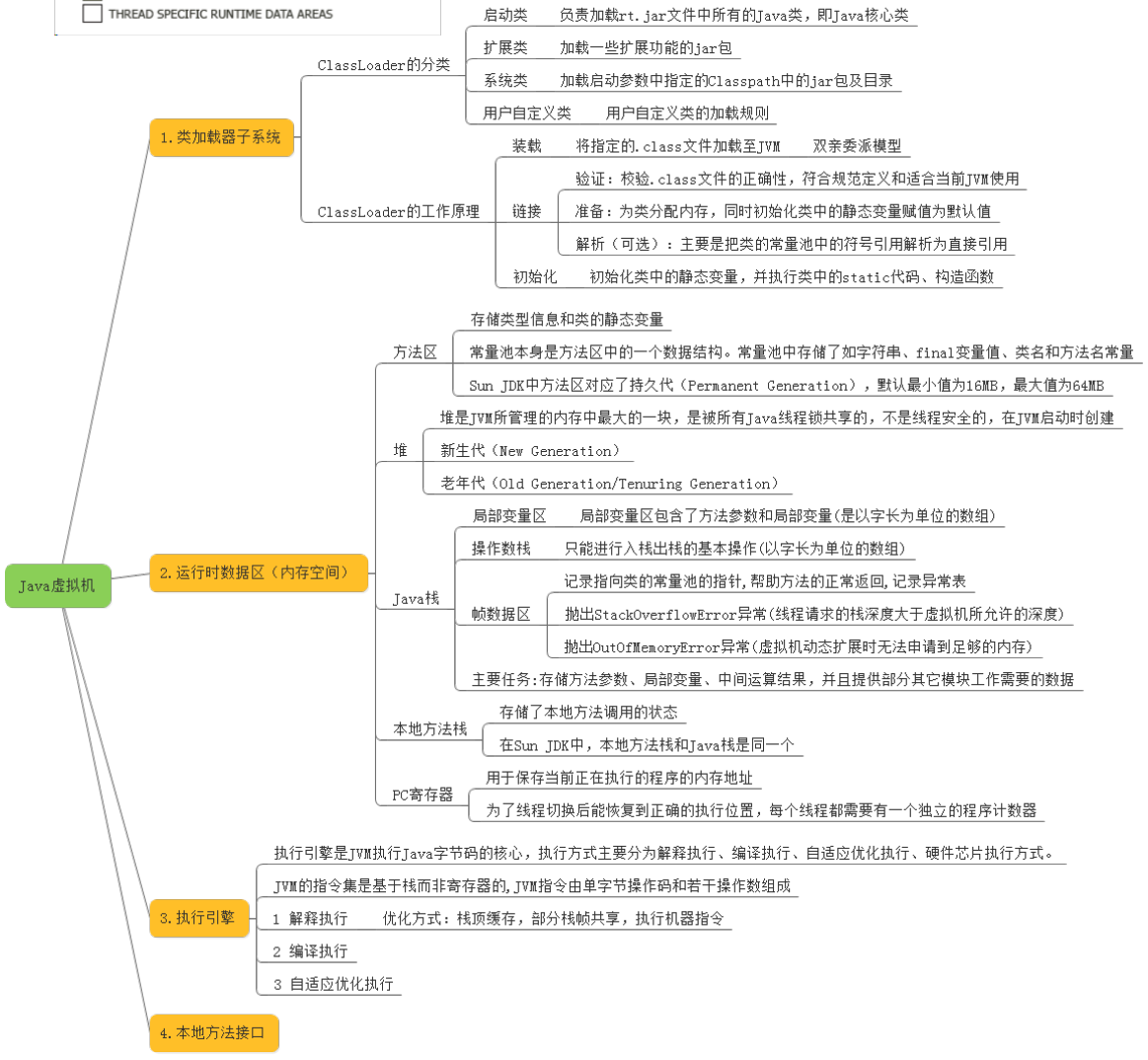
所有线程共享的数据区：方法区、堆

线程隔离的数据区：虚拟机栈、程序计数器、本地方法栈



1. 内存空间中:方法区和堆是所有Java线程共享的，Java栈、本地方法栈、PC寄存器则由每个线程私有

2. Java语言支持通过JNI（Java Native Interface）来实现本地方法的调用(不再具备跨平台)



首先要了解

1. 数据类型

Java虚拟机中，数据类型可以分为两类：**基本类型和引用类型**。基本类型的变量保存原始值，即：他代表的值就是数值本身；而引用类型的变量保存引用值。

“引用值”代表了某个对象的引用，而不是对象本身，对象本身存放在这个引用值所表示的地址的位置。

基本类型包括：**byte, short, int, long, char, float, double, Boolean, returnAddress**

引用类型包括：**类类型，接口类型和数组**

2. 堆与栈

栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。

而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟该线程（或程序）相关的：包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

栈中存的是基本数据类型和堆中对象的引用，堆中存的是对象

3. Java中的参数传递时传值呢？还是传引用？

先要明确两点：

1） 不要试图与C进行类比，Java中没有指针的概念

2） **程序运行永远都是在栈中进行的，因而参数传递时，只存在传递 基本类型和对象引用 的问题。**不会直接传对象本身。

Java在方法调用传递参数时，因为没有指针，所以它都是**进行传值调用**（这点可以参考C的传值调用）。因此，很多书里面都说Java是进行传值调用，这点没有问题，而且也简化的C中复杂性。

首先我们来看下面这段代码：

```
public class Test1 {
    String a = "123";
    public static void change(Test1 test) {
        test.a="abc";
    }
    public static void main(String[] args) {
        Test1 test1=new Test1();
        System.out.println(test1.a);
        change(test1);
        System.out.println(test1.a);
    }
}
```

```
public class Test2{
public static void main(String[] args) {
        String str = "123";
        System.out.println(str);
        change(str);
        System.out.println(str);
    }
    public static void change(String str){
        str = "abc";
    }
}
```

结果白色字体：

= 是赋值操作（任何包含=的如+=、-=、 /=等等，都内含了赋值操作）。不再是你以前理解的数学含义了，而+ - * /和 = 在java中更不是一个级别，换句话说， = 是一个动作，一个可以改变内存状态的操作，一个可以改变变量的符号，而+ - * /却不会。这里的赋值操作其实是包含了两个意思：1、放弃了原有的值或引用；2、得到了 = 右侧变量的值或引用。Java中对 = 的理解很重要！！

对于基本数据类型变量，= 操作是完整地复制了变量的值。换句话说，“=之后，你我已无关联”；赋值运算符会直接改变变量的值，原来的值被覆盖掉。

对于非基本数据类型变量，= 操作是复制了变量的引用。赋值运算符改变了引用中所保存的地址，原来的地址被覆盖掉。**但是原来的对象不会被改变（重要）。**

参数本身是变量，参数传递本质就是一种 = 操作。参数是变量，所有我们对变量的操作、变量能有的行为，参数都有。所以把C语言里参数是传值啊、传指针啊的那套理论全忘掉，参数传递就是 = 操作。

Java中所说的按引用传递实质上是传递该对象的地址，该地址其实是按值传递的，通过这个地址可以修改其指向内存处对象的值。改变该地址的值毫无意义，只会失去对真实对象的掌控。

文档中说String不可变，StringBuffer可变的意思是堆区的那片内存的可变性。
对于String类，通过引用无法修改之前在堆区申请的那段内存，大小是固定的，也就是不能修改他的值，因为他的底层是char数组。当再次给变量new一个值时，他会指向另一个堆区内存，从表面上看也是改变了值。 对于StringBuffer，可以根据实际使用继续分配更多内存，例如调用append方法，这就是可变的意思。

堆和栈中，**栈是程序运行最根本的东西，程序运行可以没有堆，但是不能没有栈。** 而堆是为栈进行数据存储服务。

说白了 **堆就是一块共享的内存。正是因为堆和栈的分离的思想，才使得Java的垃圾回收成为可能。**

Java中，栈的大小通过 -Xss 来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现 java.lang.StackOverflowError 异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是

方法返回的记录点。

4. Java对象的大小

基本数据的类型的大小是固定的，**Java基本数据类型与位运算**，这里就不多说了。对于非基本类型的Java对象，其大小就值得商榷。

在Java中，**一个空Object对象的大小是 8 byte**，这个大小只是保存堆中一个没有任何属性的对象的大小。

看语句：`Object ob = new Object();`

这样在程序中完成了一个Java对象的创建，但是 **它所占的空间为：4 byte + 8 byte =12 byte**。

4 byte是 Java栈中保存引用的所需要的空间，而那 8 byte则是Java堆中对象的信息。因为所有的Java非基本类型的对象都需要默认继承Object对象，因此不论什么样的Java对象，其大小都必须是大于 8 byte。

有了Object对象的大小，我们就可以计算其他对象的大小了。

```
Class NewObject {
    int count;
    boolean flag;
    Object ob; }
```

其大小为：

空对象大小(8byte) + int大小(4byte) + Boolean大小(1byte) + 空Object引用的大小(4byte)= 17byte。

但是因为Java在对对象内存分配时 **都是以8的整数倍来分**，因此大于17byte的最接近8的整数倍的是24，因此此**对象的大小为24 byte**。

这里需要注意一下**基本类型的包装类型的大小**。因为这种包装类型已经成为对象了，因此需要把他们作为对象来看待。

包装类型的大小至少是12byte（声明一个空Object至少需要的空间），而且12byte没有包含任何有效信息，同时，因为Java对象大小是8的整数倍，因此**一个基本类型包装类的大小至少是16byte。**

这个内存占用是很恐怖的，它是使用基本类型的N倍（N>2），有些类型的内存占用更是夸张（随便想下就知道了）。因此，**可能的话应尽量少使用包装类**。在JDK5.0以后，因为加入了自动类型装换，因此，Java虚拟机会在存储方

面进行相应的优化。

5. 引用类型

对象引用类型分为**强引用、软引用、弱引用和虚引用。**

强引用: 就是我们一**般声明对象时虚拟机生成的引用**，强引用环境下，垃圾回收时需要严格判断当前对象是否被强引用，**如果被强引用，则不会被垃圾回收**

软引用: 软引用一**般被做为缓存来使用**。与强引用的区别是，软引用在垃圾回收时，虚拟机会**根据当前系统的剩余内存来决定是否对软引用进行回收**。如果剩余内存比较紧张，则虚拟机会回收软引用所引用的空间；如果剩

余内存相对富裕，则不会进行回收。换句话说，**虚拟机在发生 OutOfMemory 时，肯定是没有软引用存在的**。

弱引用: 弱引用**与软引用类似，都是作为缓存来使用**。但与软引用不同，**弱引用在进行垃圾回收时，是一定会被回收掉的，因此其生命周期只存在于一个垃圾回收周期内。**

强引用不用说，我们系统一般在使用时都是用的强引用。而“软引用”和“弱引用”比较少见，他们一般被作为缓存使用，而且一般是在内存大小比较受限的情况下做为缓存。因为如果内存足够大的话，可以直接使用

强引用作为缓存即可，同时可控性更高。因而，他们常见的是被使用在桌面应用系统的缓存。

JVM的生命周期

一、首先分析两个概念

JVM实例和JVM执行引擎实例

（1）JVM实例 对应了一个独立运行的java程序，它是**进程级别**。

（2）JVM执行引擎实例 则对应了属于用户运行程序的线程，它是**线程级别的**。

二、JVM的生命周期

1）JVM实例的诞生

当启动一个Java程序时，一个JVM实例就产生了，**任何一个拥有public static void main(String[] args)函数的class都可以作为JVM实例运行的起点**。

2）JVM实例的运行 main() 作为该程序初始线程的起点，任何其他线程均由该线程启动。

JVM内部有两种线程：**守护线程和非守护线程，main()属于非守护线程，守护线程通常由JVM自己使用，java程序也可以标明自己创建的线程是守护线程。**

3）JVM实例的消亡

当程序中的所有非守护线程都终止时，JVM才退出；若安全管理器允许，程序也可以使用Runtime类或者System.exit()来退出。

JVM的体系结构

一、JVM的内部体系结构分为三部分，

1）类装载器（ClassLoader）子系统 作用：用来装载.class文件

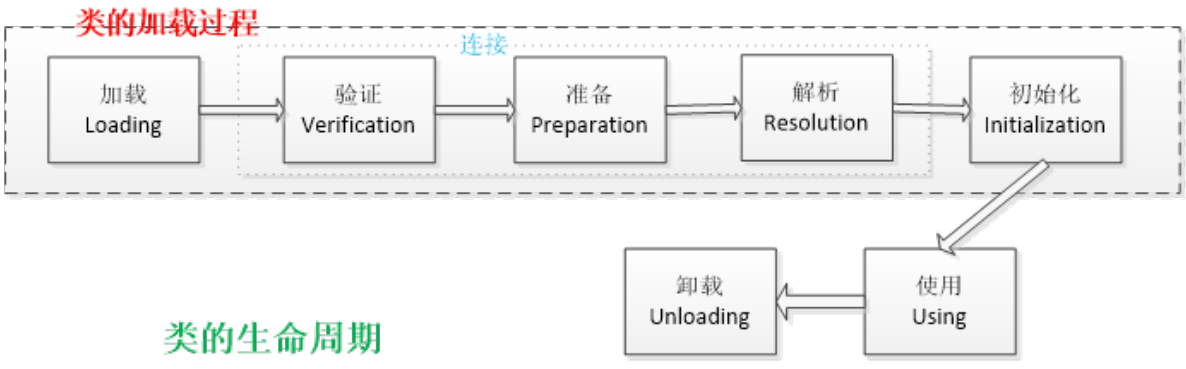
2）运行时数据区 方法区，堆，java栈，PC寄存器，本地方法栈

- Java堆（Heap）：Java虚拟机所管理的内存中最大的一块，被所有线程共享的一块内存区域，在虚拟机启动时创建。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。
- 方法区（Method Area）：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。
- 程序计数器（Program Counter Register）：一块较小的内存空间，线程私有,它的作用可以看做是当前线程所执行的字节码的行号指示器。
- JVM栈（JVM Stacks）: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方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被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从入栈到出栈的过程。
- 本地方法栈（Native Method Stacks）:与虚拟机栈所发挥的作用是非常相似的，区别是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务

73）执行引擎 作用：执行字节码，或者执行本地方法

JVM类加载器

类的生命周期包括这几个部分，加载、连接、初始化、使用和卸载，其中前三部是类的加载的过程,如下图；



- 加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象
- 连接，连接又包含三块内容：验证、准备、初始化。1）验证，文件格式、元数据、字节码、符号引用验证；2）准备，为类的静态变量分配内存，并将其初始化为默认值；3）解析，把类中的符号引用转换为直接引用
- 初始化，为类的静态变量赋予正确的初始值

- 使用，new出对象程序中使用
- 卸载，执行垃圾回收

一、 JVM将整个类加载过程划分为了三个步骤：

1）加载

类的加载指的是将类的.class文件中的二**进制数据**读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。**类的加载的最终产品是位于堆区中的Class对象**，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

装载过程负责找到二进制字节码并加载至JVM中，JVM通过类名、类所在的包名通过ClassLoader来完成类的加载，同样，也采用三个元素来标识一个被加载了的类：类名+包名+ClassLoader实例ID。

装载过程采用了一种被称为“双亲委派模型（Parent Delegation Model）”的方式，当一个ClassLoader要加载类时，它会先请求它的双亲ClassLoader（其实这里只有两个ClassLoader，所以称为父ClassLoader可能更容易理解）加载类，而它的双亲ClassLoader会继续把加载请求提交再上一级的ClassLoader，直到启动类加载器。只有其双亲ClassLoader无法加载指定的类时，它才会自己加载类。

双亲委派模型是JVM的第一道安全防线，它保证了类的安全加载，这里同时依赖了类加载器隔离的原理：不同类加载器加载的类之间是无法直接交互的，即使是同一个类，被不同的ClassLoader加载，它们也无法感知到彼此的存在。这样即使有恶意的类冒充自己在核心包（例如java.lang）下，由于它无法被启动类加载器加载，也造成不了危害。

2）链接

链接过程负责对二进制字节码的格式进行校验、初始化装载类中的静态变量以及解析类中调用的接口、类。在完成了校验后，JVM初始化类中的静态变量，并将其值赋为默认值。

最后一步为对类中的所有属性、方法进行验证，以确保其需要调用的属性、方法存在，以及具备应的权限（例如public、private域权限等），会造成 NoSuchMethodError、NoSuchFieldError 等错误信息。

3）初始化

初始化过程即为执行类中的静态初始化代码、构造器代码以及静态属性的初始化，在以下四种情况下初始化过程会被触发执行：

调用了new；反射调用了类中的方法；子类调用了初始化；JVM启动过程中指定的初始化类。

二、JVM两种类装载器包括：启动类装载器 和 用户自定义类装载器

启动类装载器是JVM实现的一部分，用户自定义类装载器则是Java程序的一部分，必须是ClassLoader 类的子类。

主要分为以下几类：

1) Bootstrap ClassLoader 启动类加载器

JVM的根ClassLoader，它是用C++实现的，JVM启动时初始化此ClassLoader，并由此ClassLoader完成 **$JAVA_HOME中jre/lib/rt.jar（Sun JDK的实现）中所有class文件的加载**，这个jar中包含了java规范定义的所有接口以及实现。

2) Extension ClassLoader 扩展类加载器

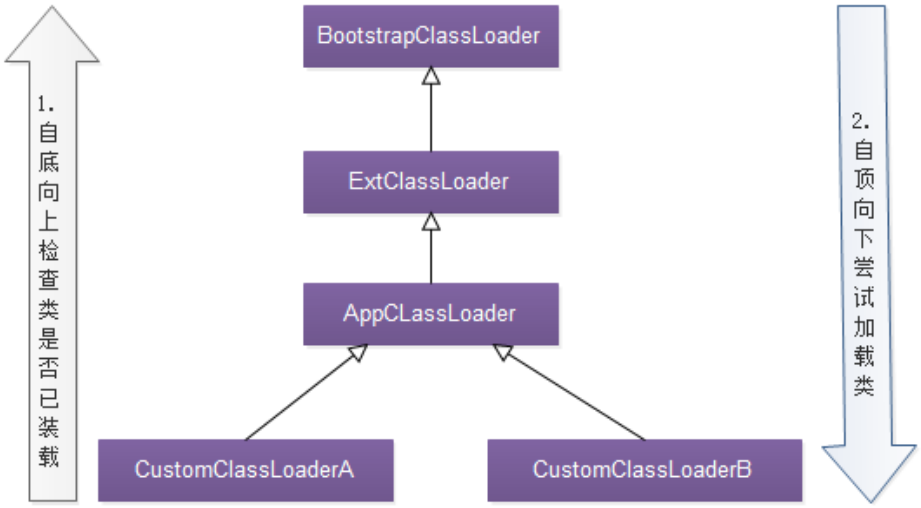
JVM用此classloader来加载扩展功能的一些jar包

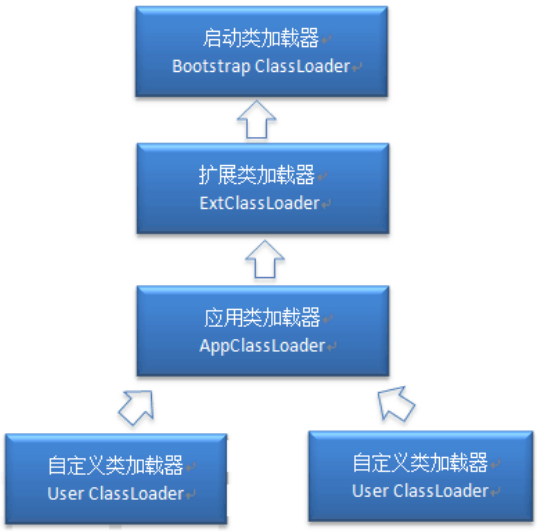
3) System ClassLoader 应用类加载器

JVM用此classloader来加载**启动参数中指定的Classpath中的jar包以及目录**，在Sun JDK中ClassLoader对应的类名为 AppClassLoader。

(4) User-Defined ClassLoader 自定义类加载器

User-DefinedClassLoader是Java开发人员继承ClassLoader抽象类自行实现的ClassLoader，基于自定义的ClassLoader可用于加载非 Classpath中的jar以及目录





三、类加载机制

- 全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入
- 父类委托，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类

- 缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效

JVM执行引擎

一、JVM 通过执行引擎来完成字节码的执行，在执行过程中JVM采用的是自己的一套指令系统，

每个线程在创建后，都会产生一个程序计数器（pc）和栈（Stack），其中程序计数器中存放了下一条将要执行的指令，

Stack中存放 **Stack Frame 栈帧**，表示的为当前正在执行的方法，每个方法的执行都会产生Stack Frame，Stack Frame中存放了**传递给方法的参数、方法内的局部变量以及操作数栈，**

操作数栈用于存放指令运算的中间结果，指令负责从操作数栈中弹出参与运算的操作数，指令执行完毕后再将计算结果压回到操作数栈，当方法执行完毕后则从Stack中弹出，继续其他方法的执行。

在执行方法时JVM提供了**invokestatic、invokevirtual、invokeinterface和invokespecial**四种指令来执行

1）invokestatic：调用类的static方法

2） invokevirtual： 调用对象实例的方法

3） invokeinterface：将属性定义为接口来进行调用

4） invokespecial： JVM对于初始化对象（Java构造器的方法为：<init>）以及调用对象实例中的私有方法时。

二、反射机制是Java的亮点之一，基于反射可动态调用某对象实例中对应的方法、访问查看对象的属性等

而无需在编写代码时就确定需要创建的对象，这使得Java可以实现很灵活的实现对象的调用，代码示例如下：

```
Class actionClass=Class.forName(外部实现类);
Method method=actionClass.getMethod("execute",null);
Object action=actionClass.newInstance();
method.invoke(action,null);
```

反射的关键：**要实现动态的调用，最明显的方法就是动态的生成字节码，加载到JVM中并执行。**

1）Class actionClass=Class.forName(外部实现类);

调用本地方法，使用调用者所在的ClassLoader来加载创建出Class对象；

2）Method method=actionClass.getMethod("execute",null);

校验此Class是否为public类型的，以确定类的执行权限，如不是public类型的，则直接抛出SecurityException；

调用 private GetDeclaredMethods来获取到此Class中所有的方法，在 private GetDeclaredMethods对此Class中所有的方法的集合做了缓存，在第一次时会调用本地方法去获取；

扫描方法集合列表中是否有相同方法名以及参数类型的方法，如有则复制生成一个新的Method对象返回；如没有则继续扫描父类、父接口中是否有此方法，如仍然没找到方法则抛出NoSuchMethodException；

3） Object action=actionClass.newInstance();

第一步：校验此Class是否为public类型，如权限不足则直接抛出SecurityException；

第二步：如没有缓存的构造器对象，则调用本地方法获取到构造器，并复制生成一个新的构造器对象，放入缓存，如没有空构造器则抛出InstantiationException；

第三步：校验构造器对象的权限；

第四步：执行构造器对象的newInstance方法；构造器对象的newInstance方法判断是否有缓存的ConstructorAccessor对象，如果没有则调用sun.reflect.ReflectionFactory生成新的ConstructorAccessor对象；

第五步：sun.reflect.ReflectionFactory判断是否需要调用本地代码，可通过sun.reflect.noInflation=true来设置为不调用本地代码，在不调用本地代码的情况下，就转交给MethodAccessorGenerator来处理了；

第六步：MethodAccessorGenerator中的generate方法根据Java Class格式规范生成字节码，字节码中包括了ConstructorAccessor对象需要的newInstance方法，此newInstance方法对应的指令为invokespecial，所需的参数则从外部压入，生成的Constructor类的名字以：sun/reflect/GeneratedSerializationConstructorAccessor或sun/reflect/GeneratedConstructorAccessor开头，后面跟随一个累计创建的对象的次数；

第七步：在生成了字节码后将其加载到当前的ClassLoader中，并实例化，完成ConstructorAccessor对象的创建过程，并将此对象放入构造器对象的缓存中；

最后一步：执行获取的constructorAccessor.newInstance，这步和标准的方法调用没有任何区别。

4） method.invoke(action,null);

这步执行的过程和上一步基本类似，只是在生成字节码时生成的方法改为了invoke，其调用的目标改为了传入的对象的方法，同时生成的类名改为了：sun/reflect/GeneratedMethodAccessor。

注：但是getMethod是非常耗性能的，一方面是权限的校验，另外一方面所有方法的扫描以及Method对象的复制，因此在使用反射调用多的系统中应缓存getMethod返回的Method对象

2、执行技术

主要的执行技术有：解释执行、即时编译执行、自适应优化执行、硬件芯片执行方式。

1）解释属于第一代JVM，

2）即时编译JIT属于第二代JVM，

3）自适应优化（目前Sun的HotspotJVM采用这种技术）则吸取第一代JVM和第二代JVM的经验，采用两者结合的方式

4）自适应优化：开始对所有的代码都采取解释执行的方式，并监视代码执行情况，然后对那些经常调用的方法启动一个后台线程，将其编译为本地代码，并进行仔细优化。若方法不再频繁使用，则取消编译过的代码，仍对其进行解释执行。

JVM运行时数据区

一、JVM在运行时将数据划分为了6个区域来存储：

第一块： PC寄存器

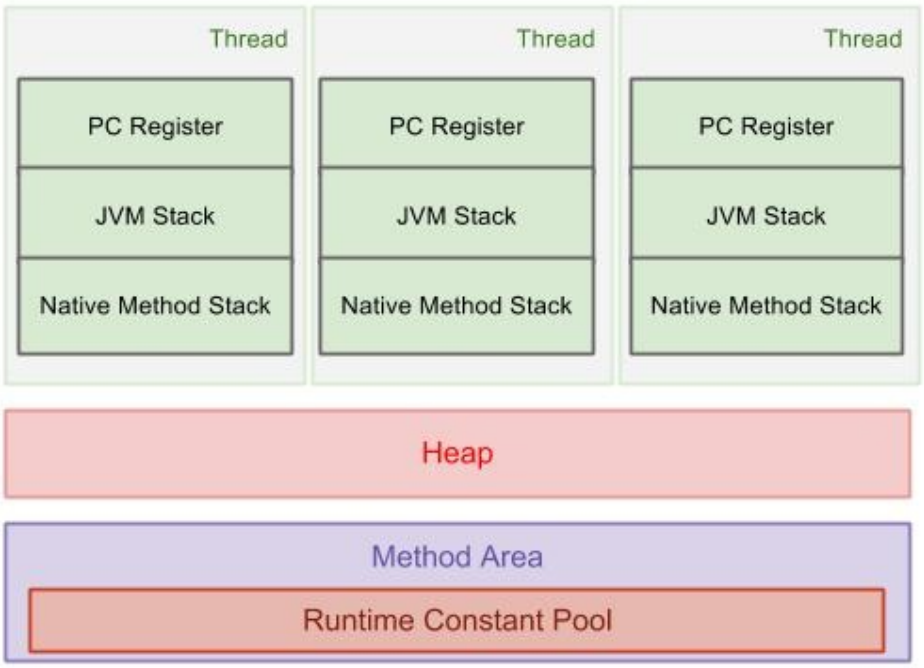
PC寄存器是用于存储每个线程下一步将执行的JVM指令，如该方法为native的，则PC寄存器中不存储任何信息。

程序计数器中存储的数据所占空间的大小不会随程序的执行而发生改变，因此不会发生内存溢出现象(OutOfMemory)。

第二块：JVM栈

Java Vitual Machine Stack

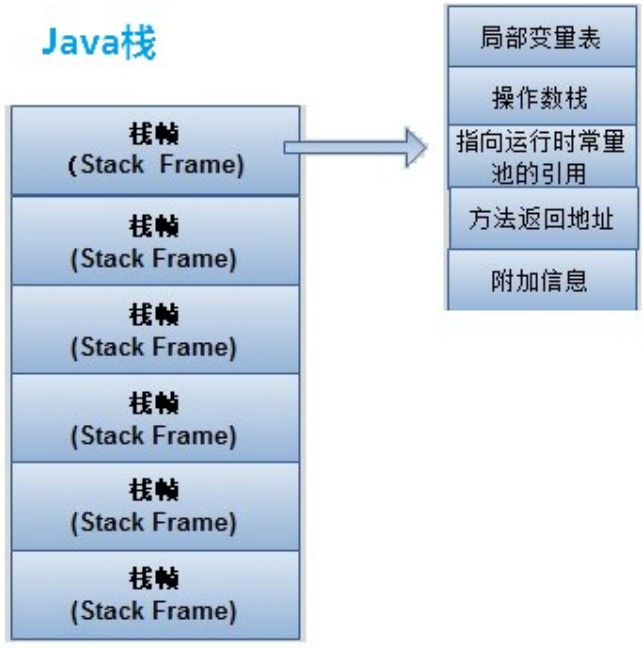
JVM栈是线程私有的，每个线程创建的同时都会创建JVM栈



Java栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法，在栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向当前方法所属的类的运行时常量池

的引用(Reference to runtime constant pool)、方法返回地址(Return Address)和一些额外的附加信息。当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈。因此可知，线程当前执行的方法所对应的栈帧必定位于Java栈的顶部(因而在使用递归方法的时候容易导致栈内存溢出的现象)。

下图表示了一个Java栈的模型：



局部变量（包括在方法中声明的非静态变量以及函数形参）。对于基本数据类型的变量，则直接存储它的值，对于引用类型的变量，则存的是指向对象的引用。局部变量表的大小在编译器就可以确定其大小了，因此在程序执行期间局部变量表的大小是不会改变的。

操作数栈，程序中的所有计算过程都是在借助于操作数栈来完成的。

指向运行时常量池的引用，在方法执行的过程中有可能需要用到类中的常量，所以必须要有一个引用指向运行时常量。

方法返回地址，当一个方法执行完毕之后，要返回之前调用它的地方，在栈帧中必须保存一个方法返回地址。

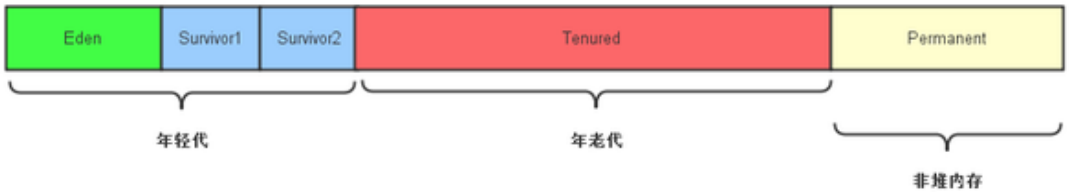
由于每个线程正在执行的方法可能不同，因此每个线程都会有一个自己的Java栈，互不干扰。

第三块：堆（Heap）

Heap是大家最为熟悉的区域，它是JVM用来存储对象实例以及数组值的区域，可以认为Java中

所有通过 new 创建的对象的内存都在此分配，Heap中的对象的内存需要等待GC进行回收。

JVM将Heap分为New Generation 和 Old Generation（或Tenured Generation）两块来进行管理：



新生代 老年代

1） New Generation 新生代，程序中新建的对象都将分配到新生代中，又由Eden Space (伊甸区)和两块Survivor Space （from区和to区) 构成，可通过-Xmn参数来指定其大小

2） Old Generation 又称为老年代，用于存放程序中经过几次垃圾回收还存活的对象，例如缓存的对象等，所占用的内存大小即为 -Xmx指定的大小减去-Xmn 指定的大小。

PermGen space 永久代

指内存的永久保存区域

OutOfMemoryError: PermGen space从表面上看就是内存溢出，解决方法也一定是加大内存。

为什么会内存溢出：这一部分用于存放 Class和Meta 的信息, Class在被 Load的时候被放入 PermGen space区域，它和存放 Instance的Heap区域不同, GC(Garbage Collection)不会在主程序运行期对PermGen space进行清理，所以如果你的 APP会LOAD很多CLASS 的话,就很可能出现PermGen space错误。这种错误常见在web服务器对JSP进行pre compile的时候。 如果你的WEB APP下都用了大量的第三方jar, 其大小 超过了jvm默认的大小(4M)那么就会产生此错误信息了。

Java内存按照分代进行组织，主要是基于这样一个事实：大多数对象都在年轻时候死亡。所以新生代**相对老年代需要更频繁的清理。**

对堆的解释：

1）堆是JVM中所有线程共享的，因此在其上进行对象内存的分配均需要进行加锁，这也导致了new对象的开销是比较大的。

2）鉴于上面的原因，Sun Hotspot JVM为了提升对象内存分配的效率，对于所创建的线程都会分配一块独立的空间，这块空间又称为TLAB（Thread Local Allocation Buffer），其大小由JVM根据运行的情况计算而得，在TLAB上分配对象时不需要加锁，因此JVM在给线程的对象分配内存时会尽量的在TLAB上分配，在这种情况下JVM中分配对象内存的性能和C基本是一样高效的，但如果对象过大的话则仍然是直接使用堆空间分配。

3）TLAB 仅作用于新生代的Eden Space，因此在编写Java程序时，通常多个小的对象比大的对象分配起来更加高效，

但这种方法同时也带来了两个问题，一是空间的浪费，二是对象内存的回收上仍然没法做到像Stack那么高效，同时也会增加回收时的资源的消耗，可通过在启动参数上增加-XX:+PrintTLAB来查看TLAB这块的使用情况。

第四块：方法区域（Method Area） --注： 堆和方法区在逻辑上分开的，物理上是一起的 ，全局共享

1）

存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。当开发人员在程序中通过Class对象中的getName、isInterface等方法来获取信息时，这些数据都来源于方法区域，可见方法区域的重要性，同样，方法区域也是全局共享的，在一定的条件下它也会被GC，当方法区域需要使用的内存超过其允许的大小时，会抛出 OutOfMemory 的错误信息。

用于储存包括**类的元数据**，**常量池**，**普通字段**，**静态变量**，方法内的**局部变量**以及编译好的**字节码**

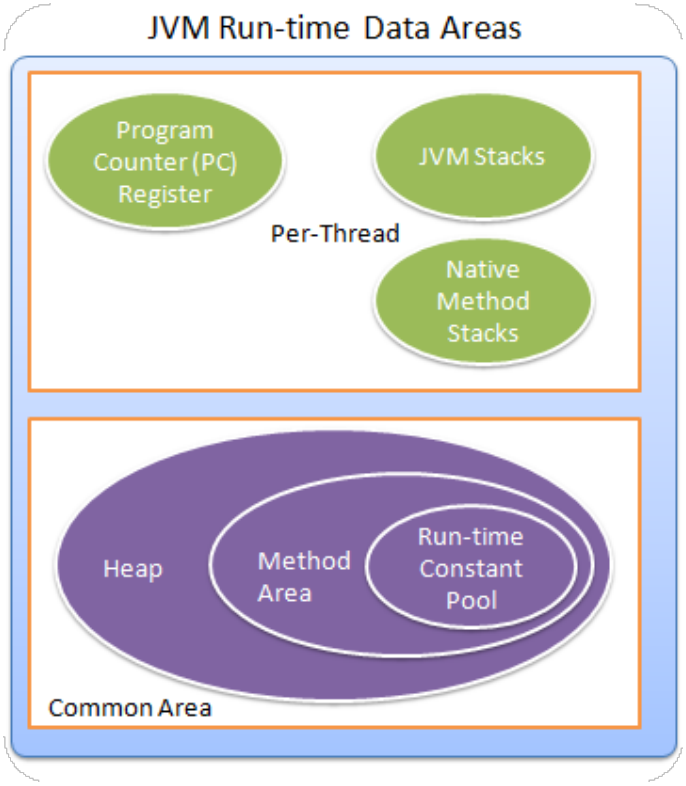
2）在Sun JDK中这块区域对应的为 Permanet Generation，又称为持久代，默认为64M，可通过-XX:PermSize以及-XX:MaxPermSize来指定其大小。sunJDK的方法区可能会抛出 java.lang.OutOfMemoryError: PermGen space异常。

Java 8中已经放弃了对永久代的分配，使用 Native Memory （元数据区，Metaspace）来进行管理。

在方法区中有一个非常重要的部分就是运行时常量池，它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到JVM后，对应的运行时常量池就被创建出来。当然并非Class文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，比如String的intern方法。

第五块：运行时常量池（Runtime Constant Pool）

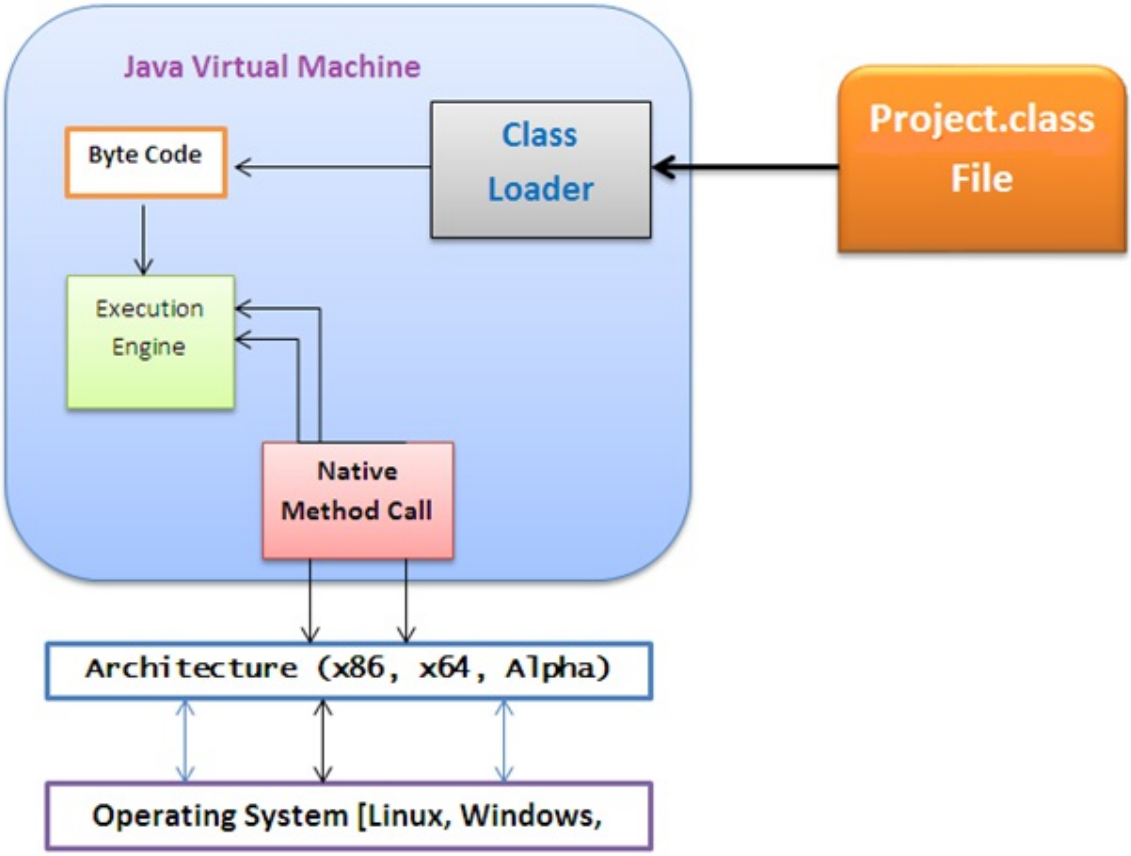
类似C中的符号表，存放的为类中的固定的常量信息、方法和Field的引用信息等，其空间从方法区域中分配。



第六块：本地方法堆栈（Native Method Stacks）

JVM采用本地方法堆栈来支持native方法的执行，此区域用于存储每个native方法调用的状态。

本地方法栈与Java栈的作用和原理非常相似。区别是Java栈是为执行Java方法服务的，而本地方法栈则是为执行本地方法（Native Method）服务的。在JVM规范中，并没有对本地方发展的具体实现方法以及数据结构作强制规定，虚拟机可以自由实现它。在HotSopt虚拟机中直接就把本地方法栈和Java栈合二为一。



基本垃圾回收算法

按照基本回收策略分

1. 引用计数（Reference Counting）：

比较古老的回收算法。原理是 **对象有一个引用，即增加一个计数，删除一个引用则减少一个计数**。垃圾回收时，只收集计数为0的对象。

此算法最致命的是**无法处理循环引用**的问题。

2. 标记-清除（Mark-Sweep）：

此算法执行分两阶段。**第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。**此算法 **需要暂停整个应用，同时，会产生内存碎片**。

3. 复制（Copying）：

此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。次算法每次只处理正在使用中的对象，因此复制成本比较小，**同时复制**

过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是 **需要两倍内存空间**。

4. 标记-整理（Mark-Compact）：

此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，**第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块**，按顺序排放。此

算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

5. 分代收集算法（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

按分区对待的方式分

增量收集（Incremental Collecting）：

实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因JDK5.0中的收集器 **没有使用这种算法**的。

分代收集（Generational Collecting）：

基于对对象生命周期分析后得出的垃圾回收算法。把对象 **分为新生代、老年代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收**。现在的垃圾回收器（从J2SE1.2开始）都是使用此算法的。

按系统线程分

串行收集器（Serial Collector）：

串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，**即无法使用多处理器的优势，所以此收集适合单处理器机器**。当然，此收集器也可以用在小数据量（100M左右）情况下的多处理器机器上。

并行收集器（Parallel Collector）：

并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。**而且理论上CPU数目越多，越能体现出并行收集器的优势。**

CMS（并发标记清理收集器，Concurrent Mark Sweep）：

相对于串行收集和并行收集而言，前面两个在进行垃圾回收工作时，需要暂停整个运行环境，而只有垃圾回收程序在运行，因此，系统在垃圾回收时会有明显的暂停，而且暂停时间会因为堆越大而越长。

CMS（-XX：+ UseConcMarkSweepGC）收集器在老年代中使用，专门收集那些在主要回收中不可能到达的年老对象。与应用程序并发运行，在老年代中保持一直有足够的空间以保证不会发生新生代晋升失败。晋升失败将会触发一次FullGC，CMS会按照下面几个步骤处理：

1) 初始化标记：寻找GC根。

2) 并发标记：标记所有从GC根开始可到达的对象。

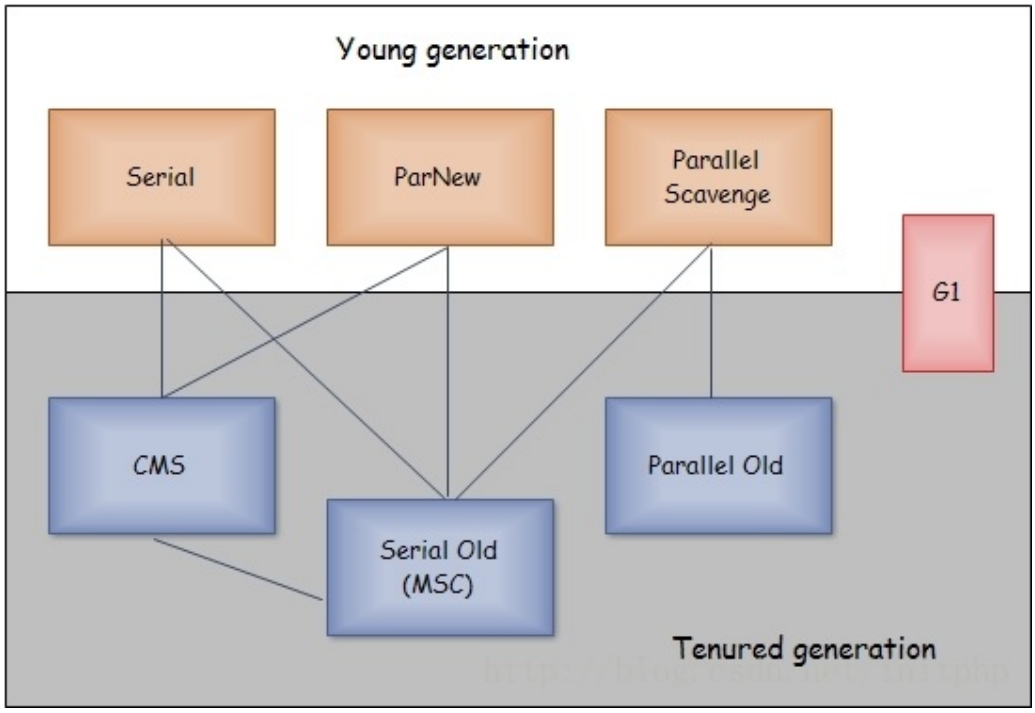
3) 并发预清理：检查被更新过的对象引用和在并发标记阶段晋升的对象。

4) 重标记：捕捉预清洁阶段开始更新的对象引用。

5) 并发清理：通过回收被死对象占用的内存更新可用空间列表。

6) 并发重置：重置数据结构为下一次运行做准备。

新生代和老年代可以用到的垃圾收集器。



Oracle JDK 7 update 4版本后支持G1回收策略 （目前CMS还是默认首选的GC策略）

是面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征

G1在压缩空间方面有优势

G1通过将内存空间分成区域（Region）的方式避免内存碎片问题

Eden, Survivor, Old区不再固定、在内存使用效率上来说更灵活

G1可以通过设置预期停顿时间（Pause Time）来控制垃圾收集时间避免应用雪崩现象

G1在回收内存后会马上同时做合并空闲内存的工作、而CMS默认是在STW（stop the world）的时候做

G1会在Young GC中使用、而CMS只能在O区使用

适用场景

服务端多核CPU、JVM内存占用较大的应用（至少大于4G）

应用在运行过程中会产生大量内存碎片、需要经常压缩空间

想要更可控、可预期的GC停顿周期；防止高并发下应用雪崩现象

JVM垃圾回收

一、 JVM中自动的对象内存回收机制称为：GC（Garbage Collection）

1、 GC的基本原理：

为将内存中不再被使用的对象进行回收，GC中用于回收内存中不被使用的对象的方法称为收集器，由于GC需要消耗一些资源和时间的，Java在对对象的生命周期特征进行分析后，在V 1.2以上的版本采用了分代的方式来进行对象的收集，

即按照新生代、老年代的方式来对对象进行收集，以尽可能的缩短GC对应用造成的暂停

1）对新生代的对象的收集称为minor GC

2）对老年代的对象的收集称为Major GC

3）程序中主动调用System.gc()强制执行的GC为Full GC

二、 JVM中自动内存回收机制

（1）引用计数收集器

原理： 引用计数是标识Heap中对象状态最明显的一种方法，**对象有一个引用，即增加一个计数，删除一个引用则减少一个计数**。当这个计数为零时，说明这个对象已经不再被使用了。

优点： 引用计数的好处是可以不用暂停应用，当计数变为零时，即可将此对象的内存空间回收，但它需要给每个对象附加一个关联引用计数

缺点： 并且引用计数无法解决循环引用的问题，因此JVM并没有采用引用计数。

（2）跟踪收集器

原理：跟踪收集器的方法为停止应用的工作，然后开始跟踪对象，跟踪时从对象根开始沿着引用跟踪，直到检查完所有的对象。

根对象的来源主要有三种：

1. 被加载的类的常量池中的对象引用

2. 传到本地方法中，没有被本地方法"释放"的对象引用

3. 虚拟机运行时数据区中从垃圾收集器的堆中分配的部分

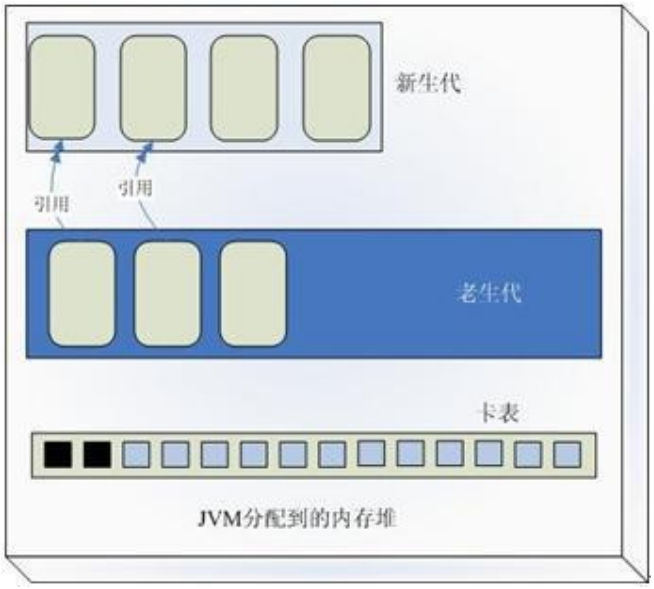
存在问题：跟踪收集器采用的均为扫描的方法，但JVM将Heap分为了新生代和旧生代，在进行minor GC时需要扫描是否有旧生代引用了新生代中的对象，但又不可能每次minor GC都扫描整个旧生代中的对象，

此JVM采用了一种称为卡片标记（Card Marking）的算法来避免这种现象。

（3）卡片标记算法

卡片标记的算法为将老年代以某个大小（例如512字节）进行划分，划分出来的每个区域称为卡片，JVM采用卡表维护卡的状态，每张卡片在卡表中占用一个字节的标识（有些JVM实现可能会不同），

当Java代码执行过程中发现老年代的对象引用或释放了对于新生代对象的引用时，就相应的修改卡表中卡的状态，每次Minor GC只需扫描卡表中标识为脏状态的卡中的对象即可，图示如下：



1、跟踪收集器在扫描时最重要的是要根据这些对象是否被引用来标识其状态

2、JVM中将对象的引用分为了四种类型，不同的对象引用类型会造成GC采用不同的方法进行回收：

1）强引用：默认情况下，对象采用的均为强引用（这个对象的实例没有其他对象引用，GC时才会被回收）

2）软引用：软引用是Java中提供的一种比较适合于缓存场景的应用只有在内存不够用的情况下才会被GC）

3）弱引用：在GC时一定会被GC回收

4）虚引用：由于虚引用只是用来得知对象是否被GC

三、 JVM中内存回收

GC的工作目的：在堆中找到已经无用的对象，并把这些对象占用的空间收回使其可以重新利用.

对象分配规则

- 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。
- 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
- 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。
- 动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

- 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

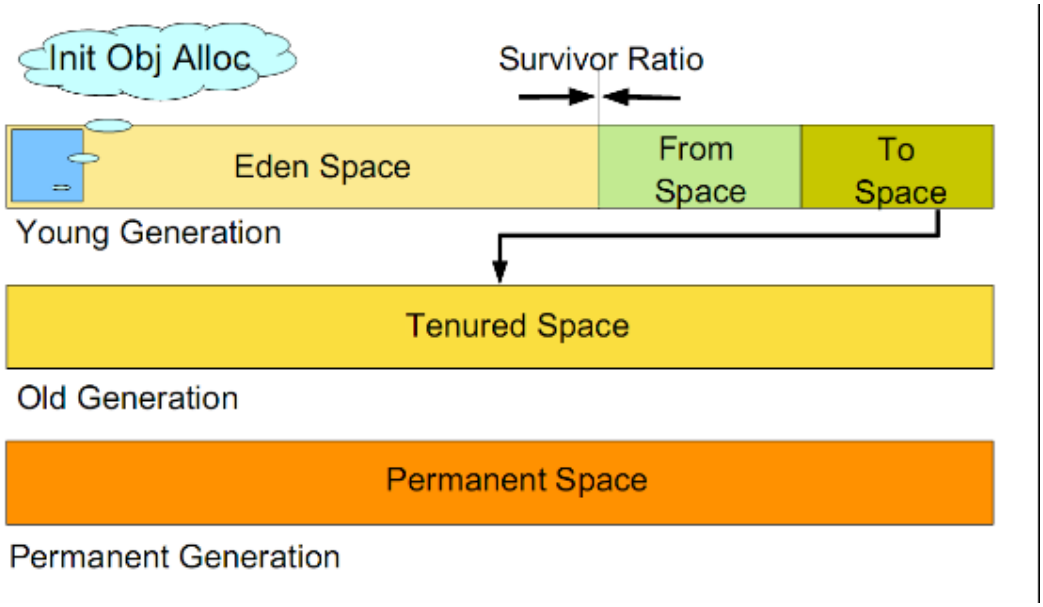
对象存活判断

判断对象是否存活一般有两种方式：

- 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。
- 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，不可达对象。

当一组对象生成时，内存申请过程如下：

1. JVM会试图为相关Java对象在新生代的Eden区中初始化一块内存区域。
2. 当Eden区空间足够时，内存申请结束。否则执行下一步。
3. JVM试图释放在Eden区中所有不活跃的对象（Young GC）。释放后若Eden空间仍然不足以放入新对象，JVM则试图将部分Eden区中活跃对象放入Survivor区。
4. Survivor区被用来作为Eden区及老年代的中间交换区域。当老年代空间足够时，Survivor区中存活了一定次数的对象会被移到老年代。
5. 当老年代空间不够时，JVM会在老年代进行完全的垃圾回收（Full GC）。
6. Full GC后，若Survivor区及老年代仍然无法存放从Eden区复制过来的对象，则会导致JVM无法在Eden区为新生成的对象申请内存，即出现“Out of Memory”。



虚拟机中共划分了三个代：新生代（Young Generation）、老年代（Old Generation）和持久代（Permanent Generation）。

其中持久代主要存放的是java类的类信息，与垃圾收集要收集的java对象关系不大。新生代和老年代的划分是对垃圾收集影响比较大的。

新生代：

所有新生成的对象首先都是放在新生代的。新生代的目标就是尽可能快速的收集掉那些生命周期短的对象。新生代分为三个区。一个Eden区，两个Survivor区（一般而言）。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当这个Survivor区满时，此区的存活将被复制到另外一个Survivor区，当这个Survivor区也满了的时候，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制"老年代（Tenured）"。需要注意，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden复制过来的对象和从前一个Survivor复制过来的对象，而复制到年老区的只有从第一个Survivor区过来的对象。而且，Survivor区总有一个是空的。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在新生代中的存在时间，减少被放到老年代的可能。

老年代：

在新生代中经历了N次垃圾回收后仍然存活的对象，就会被放到老年代中。因此，可以认为老年代中存放的都是一些生命周期较长的对象。

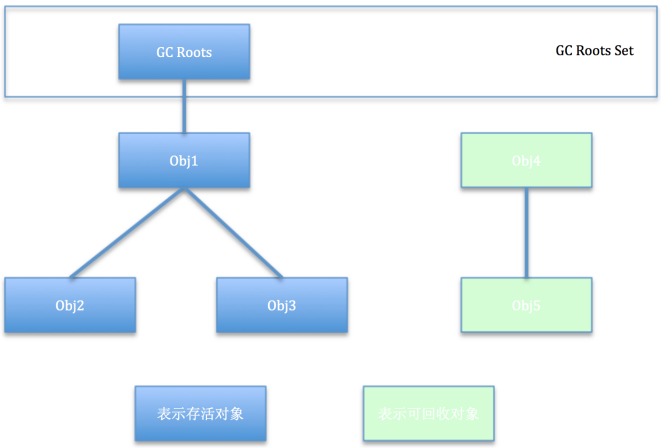
持久代：

用于存放静态文件，如java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class,例如Hibernate等，在这种时候需要设置一个比较大的持久空间来存放这些运行过程中新增的类。持久代大小通过 -XX:MaxPermSize = <N> 进行设置。

默认情况下内存分配 Eden:Survivor 区为8:1

可达性分析算法

原理：通过一系列称为"GC Roots"的对象作为起始起点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连时，则证明此对象没有被使 用，可以被回收。



Java中可作为GC Roots的对象包括以下几种：

1) 虚拟机栈中引用的对象

2) 方法区中类静态属性引用的对象

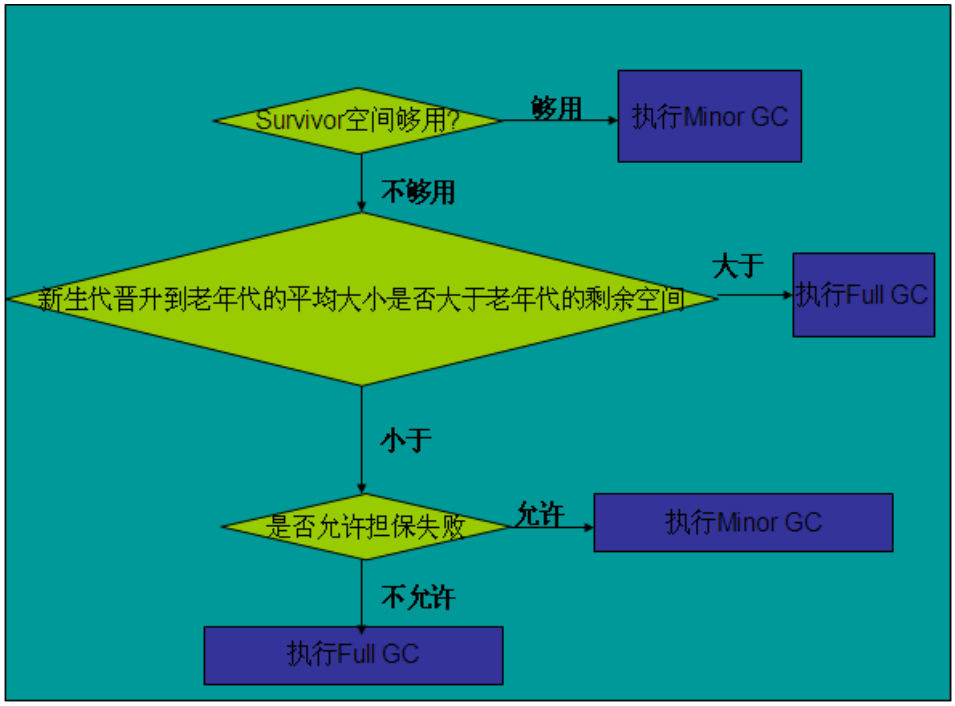
3) 方法区中常量引用的对象

4) 本地方法栈中JNI引用的对象

对young gc来说，gcroot的对象包括：

1) 所有老年带对象
2) 所有全局对象
3) 所有jni句柄
4) 所有上锁对象
5) jvmti持有的对象
6) 代码段code_cache
7) 所有classloader，及其加载的class
8) 所有字典
9) flat_profiler和management
10) 最重要的，所有运行中线程栈上的引用类型变量

一般情况下的GC策略如下图：



1、新生代垃圾回收

新生代垃圾回收使用的是 标记-复制 算法

当 JVM 无法为一个新的对象分配空间时(Eden 区满了)会触发 Minor GC ,清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。 (注意young GC中有部分存活对象会晋升到old gen，所以young GC后old gen的占用量通常会有所升高)

具体过程参看文章：图解Java中的GC（分代收集器） http://blog.csdn.net/hp910315/article/details/50985877

2、老年代垃圾回收

老年代的存活率一般比较高，一般使用"标记-清理"或者"标记-整理"算法进行回收

当minor gc 发生时，又有对象从Survivor区域升级到Tenured区域，但是Tenured区域已经没有空间容纳新的对象了，那么这个时候就会触发老年代上的垃圾回收，我们称之为"Major Garbage Collection".

在老年代上选择的垃圾回收算法则取决于JVM上采用的是什么垃圾回收器。通过的垃圾回收器有两种：Parallel Scavenge(PS) 和Concurrent Mark Sweep(CMS)。他们主要的不同体现在老年代的垃圾回收过程中，

Parallel Scavenge垃圾回收器在执行垃圾回收时使用了多线程，以提高垃圾回收的效率。而Concurrent Mark Sweep回收器主要是应用程序挂起"Stop The World"的时间比较短，更接近并发。

除直接调用System.gc外，触发Full GC执行的情况有如下四种

1. 老年代空间不足

老年代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象

调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。

2. 永久代空间满

永久代中存放的为一些class的信息等，当系统中要加载的类、反射的类和调用的方法较多时可能会被占满，在未配置为采用CMS GC的情况下会执行Full GC。

可采用的方法为增大Perm Gen空间或转为使用CMS GC。

3. CMS GC 时出现 promotion failed 和 concurrent mode failure

promotionfailed是在进行Minor GC时，survivor space放不下、对象只能放入老年代，而此时年老生代也放不下造成的；

concurrent mode failure是在执行CMS GC的过程中同时有对象要放入老年代，而此时老年代空间不足造成的。

应对措施为：增大survivorspace、老年代空间或调低触发并发GC的比率，但在JDK 5.0+、6.0+的版本中有可能会由于JDK的bug29导致CMS在remark完毕后很久才触发sweeping动作。

对于这种状况，可通过设置-XX:CMSMaxAbortablePrecleanTime=5（单位为ms）来避免。

4. 统计得到的 **Minor GC晋升到老年代的平均大小大于剩余空间 时**

四、JVM调优

通常情况下，JVM堆的大小应为物理内存的80%（小于？，默认为物理内容的4分之1）。

整个JVM内存大小=年轻代大小 + 年老代大小 + 持久代大小

调优总结

1. **新生代大小选择**

   响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，新生代收集发生的频率也是最小的。同时，减少到达老年代的对象。

   吞吐量优先的应用：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

2. **老年代大小选择**

   响应时间优先的应用：老年代使用并发收集器，所以其大小需要小心设置，一般要考虑**并发会话率**和**会话持续时间**等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

   并发垃圾收集信息

   持久代并发收集次数

   传统GC信息

   花在新生代和老年代回收上的时间比例

   减少新生代和老年代花费的时间，一般会提高应用的效率

   吞吐量优先的应用：一般吞吐量优先的应用都有一个很大的新生代和一个较小的老年代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而老年代尽存放长期存活对象。

3. **较小堆引起的碎片问题**

   因为老年代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

   -XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对老年代的压缩。

   -XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次Full GC后，对老年代进行压缩

JVM调优工具Jconsole，jProfile，VisualVM

Jconsole:JDK自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪。

JProfiler：商业软件，需要付费。功能强大。

VisualVM：JDK自带，功能强大，与JProfiler类似。推荐

常见配置汇总

堆大小设置:

JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统下，一般限制在1.5G~2G；64为操作系统对内存无限制

1. 堆设置

-Xms: 初始堆大小
-Xmx: 最大堆大小
-XX:NewSize=n: 设置新生代大小
-XX:NewRatio=n:设置新生代和老年代的比值。如:为3，表示新生代与老年代比值为1：3，新生代占整个新生代老年代和的1/4
-XX:SurvivorRatio=n:新生代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个新生代的1/5
-XX:MaxPermSize=n:设置持久代大小

2. 收集器设置
   -XX:+UseSerialGC:设置串行收集器
   -XX:+UseParallelGC:设置并行收集器
   -XX:+UseParalledlOldGC:设置并行老年代收集器
   -XX:+UseConcMarkSweepGC:设置并发收集器
3. 垃圾回收统计信息
   -XX:+PrintGC
   -XX:+PrintGCDetails
   -XX:+PrintGCTimeStamps
   -Xloggc:filename
4. 并行收集器设置
   -XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。
   -XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间
   -XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)
5. 并发收集器设置
   -XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。
   -XX:ParallelGCThreads=n:设置并发收集器新生代收集方式为并行收集时，使用的CPU数。并行收集线程数。
   -XX:MaxTenuringThreshold=0：设置垃圾最大年龄。**如果设置为0的话，则新生代对象不经过Survivor区，直接进入老年代**。对于老年代比较多的应用，可以提高效率。**如果将此值设置为一个较大值，则新生代对象会在Survivor区进行多次复制，这样可以增加对象再新生代的存活时间**，增加在新生代即被回收的概论。

-Xmn2g：设置新生代大小为2G。持久代一般固定大小为64m，所以增大新生代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8

问题

你能不能谈谈，java GC是在什么时候，对什么东西，做了什么事情？

1. 什么时候触发?
   看上面
   GC与非 GC 时间耗时超过了 GCTimeRatio的限制引发 OOM 内存溢出
   程序员不能具体控制时间，系统在不可预测的时间调用 System.gc()
   调优，用 NewRatio 控制 newObject 和 oldObject 的比例，用 MaxTenuringThreshold 控制进入oldObject的次数，使得 oldObject 存储空间延迟达到full gc,
   从而使得计时器引发gc时间延迟OOM的时间延迟，以延长对象生存期。

2.对什么东西？

超出了作用域或引用计数为空的对象；从gc root开始搜索找不到的对象，而且经过一次标记、清理，仍然没有复活的对象。

3. 做什么？

删除不使用的对象，回收内存空间；运行默认的finalize, 当然程序员想立刻调用就用dipose调用以释放资源如文件句柄，JVM用from survivor、to survivor对它进行标记清理，对象序列化后也可以使它复活。

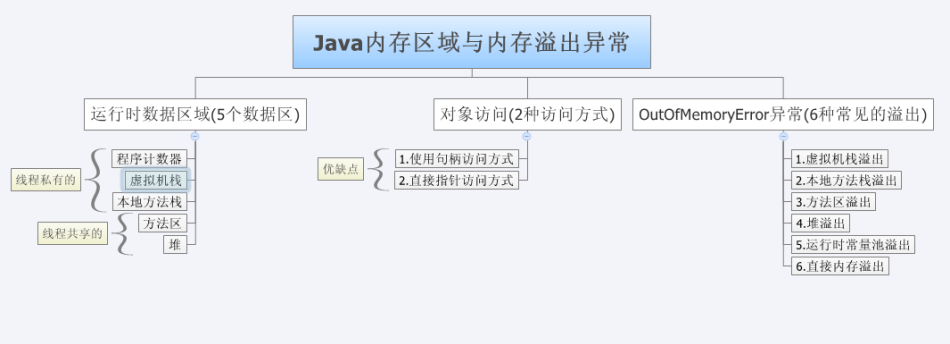
OOM（“Out of Memory”）异常一般主要有如下2种原因：

1. 老年代溢出，表现为：java.lang.OutOfMemoryError:Javaheapspace

这是最常见的情况，产生的原因可能是：设置的内存参数Xmx过小或程序的内存泄露及使用不当问题。**例如循环上万次的字符串处理、创建上千万个对象、在一段代码内申请上百M甚至上G的内存。**还有的时候虽然不会报内存溢出，却会使系统不间断的垃圾回收，也无法处理其它请求。这种情况下除了检查程序、打印堆内存等方法排查，还可以借助一些内存分析工具，比如MAT就很不错。

2. 持久代溢出，表现为：java.lang.OutOfMemoryError:PermGenspace

通常由于持久代设置过小，动态加载了大量Java类而导致溢出，解决办法唯有将参数 -XX:MaxPermSize 调大（一般256m能满足绝大多数应用程序需求）。将部分Java类放到容器共享区（例如Tomcat share lib）去加载的办法也是一个思路，但前提是容器里部署了多个应用，且这些应用有大量的共享类库



Java堆溢出：堆里放的是new出来的对象，不断的new对象 （防止对象new出来之后被GC，把对象new出来的对象放到一个List中去）

虚拟机栈溢出：单线程的堆中不断的让一个成员变量自增，无法承受这个变量了就抛出StackOverflowError。或开尽量多的线程，并在每个线程里调用native的方法就会抛出OutOfMemoryError。

方法区溢出：可采用增强Class加载的方式。基本思路是运行时产生大量的类去填满方法区，直到溢出。借助第三方类库CGLib 直接操作字节码运行，生成大量的动态类。

Java中的内存泄露

在Java中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，

首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；

其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

一般来说是长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收。

一个典型的内存泄露实例：

```
Vector v=new Vector(10);
for (int i=1;i<100; i++){
    Object o=new Object();
    v.add(o);
    /**   * 此时，所有的Object对象都没有被释放，因为变量v引用这些对象     */
   o=null;
 }
```

我们循环申请Object对象，并将所申请的对象放入一个Vector中，如果我们仅仅释放引用本身，那么Vector仍然引用该对象，所以这个对象对GC来说是不可回收的。

因此，如果对象加入到Vector后，还必须从Vector中删除，最简单的方法就是将Vector对象设置为null。

在Java程序中容易发生内存泄露的场景：

1. 集合类，集合类仅仅有添加元素的方法，而没有相应的删除机制，导致内存被占用

这个集合类如果仅仅是局部变量，根本不会造成内存泄露，在方法栈退出后就没有引用了会被jvm正常回收，而如果这个集合类是全局性的变量（比如类中的静态属性，全局性的map等即有静态引用或final一直指向它），那么没有相应的删除机制，很可能导致集合所占用的内存只增不减，因此提供这样的删除机制或者定期清除策略非常必要。

2. 单例模式

不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露，考虑下面的例子：

```
class A{
    public A(){
        B.getInstance().setA(this);
    }
    ....
```

```
        }
    //B类采用单例模式
        class B{
            private A a;
            private static B instance=new B();
            public B(){}
            public static B getInstance(){
            return instance;
        }
        public void setA(A a){
            this.a=a;
        }
        //getter…
        }
```

显然B采用singleton模式，他持有一个A对象的引用，而这个A类的对象将不能被回收，想象下如果A是个比较大的对象或者集合类型会发生什么情况。

所以在Java开发过程中和代码复审的时候要重点关注那些长生命周期对象：全局性的集合、单例模式的使用、类的static变量等等。
在不使用某对象时，显式地将此对象赋空，遵循谁创建谁释放的原则，减少内存泄漏发生的机会。

常见问题

1、内存溢出

要求分配的java虚拟机内存超出了系统能提供的，内存不够产生溢出。

2、内存泄漏

指程序中间动态分配了内存，但在程序结束时没有释放这部分内存，从而造成那部分内存不可用的情况，重启计算机可以解决，但也有可能再次发生内存泄露，内存泄露和硬件没有关系，它是由软件设计缺陷引起的。。

例：

1. Java中也存在内存泄漏。当被分配的对象可达但已无用（未对作废数据内存单元的引用置null）即会引起。

```
Vector v=new Vector(10);
for (int i=1;i<100; i ) {
    Object o=new Object();
    v.add(o);
```

```
    o=null;
}
// 此时，所有的Object对象都没有被释放，因为变量v引用这些对象。
// 对象加入到Vector后，还必须从Vector中删除，最简单释放方法就是将Vector对象设置为null。
```

内存泄漏、溢出的异同？

同： 都会导致应用程序运行出现问题，性能下降或挂起。

异： 1) 内存泄漏是导致内存溢出的原因之一；内存泄露积累起来将导致内存溢出。

2) 内存泄v可以通过完善代码来避免；内存溢出可以通过调整配置来减少发生频率，但无法彻底避免。

如何避免内存泄露、溢出？

1)尽早释放无用对象的引用。

好的办法是使用临时变量的时候，让引用变量在退出活动域后自动设置为null，暗示垃圾收集器来收集该对象，防止发生内存泄露。

2)程序进行字符串处理时，尽量避免使用String，而应使用StringBuffer。

因为每一个String对象都会独立占用内存一块区域，如：

```
String str = "aaa";
String str2 = "bbb";
String str3 = str   str2;
// 假如执行此次之后str，str2再不被调用，那么它们就会在内存中等待GC回收；
// 假如程序中存在过多的类似情况就会出现内存错误；
```

3) 尽量少用静态变量。

因为静态变量是全局的，GC不会回收。

4) 避免集中创建对象尤其是大对象，如果可以的话尽量使用流操作。

JVM会突然需要大量内存，这时会触发GC优化系统内存环境； 一个案例如下：

```
// 使用jspsmartUpload作文件上传，运行过程中经常出现java.outofMemoryError的错误，
// 检查之后发现问题：组件里的代码
m_totalBytes = m_request.getContentLength();
m_binArray = new byte[m_totalBytes];
// totalBytes这个变量得到的数极大，导致该数组分配了很多内存空间，而且该数组不能及时释放。
// 解决办法只能换一种更合适的办法，至少是不会引发outofMemoryError的方式解决。
// 参考：http://bbs.xml.org.cn/blog/more.asp?name=hongrui&id=3747
```

5) 尽量运用对象池技术以提高系统性能。

生命周期长的对象拥有生命周期短的对象时容易引发内存泄漏，例如大集合对象拥有大数据量的业务对象的时候，可以考虑分块进行处理，然后解决一块释放一块的策略。

6) 不要在经常调用的方法中创建对象，尤其是忌讳在循环中创建对象。

可以适当的使用hashtable，vector 创建一组对象容器，然后从容器中去取那些对象，而不用每次

new之后又丢弃。

7) 优化配置。

内存溢出的解决方案？

一是从代码层面进行优化完善，尽量避免该情况发生；

二是调整优化服务器配置：

1) 设置-Xms、-Xmx相等；

2) 设置NewSize、MaxNewSize相等；

3) 设置Heap size, PermGen space:

Tomcat 的配置示例：修改%TOMCAT_HOME%/bin/catalina.bat or catalina.sh

在“echo "Using CATALINA_BASE: $CATALINA_BASE"”上面加入以下行：

Cmd代码

```
set JAVA_OPTS=-Xms800m -Xmx800m -XX:PermSize=128M -XX:MaxNewSize=256m -
XX:MaxPermSize=256m
```

来源： http://www.cnblogs.com/binyue/p/4546205.html