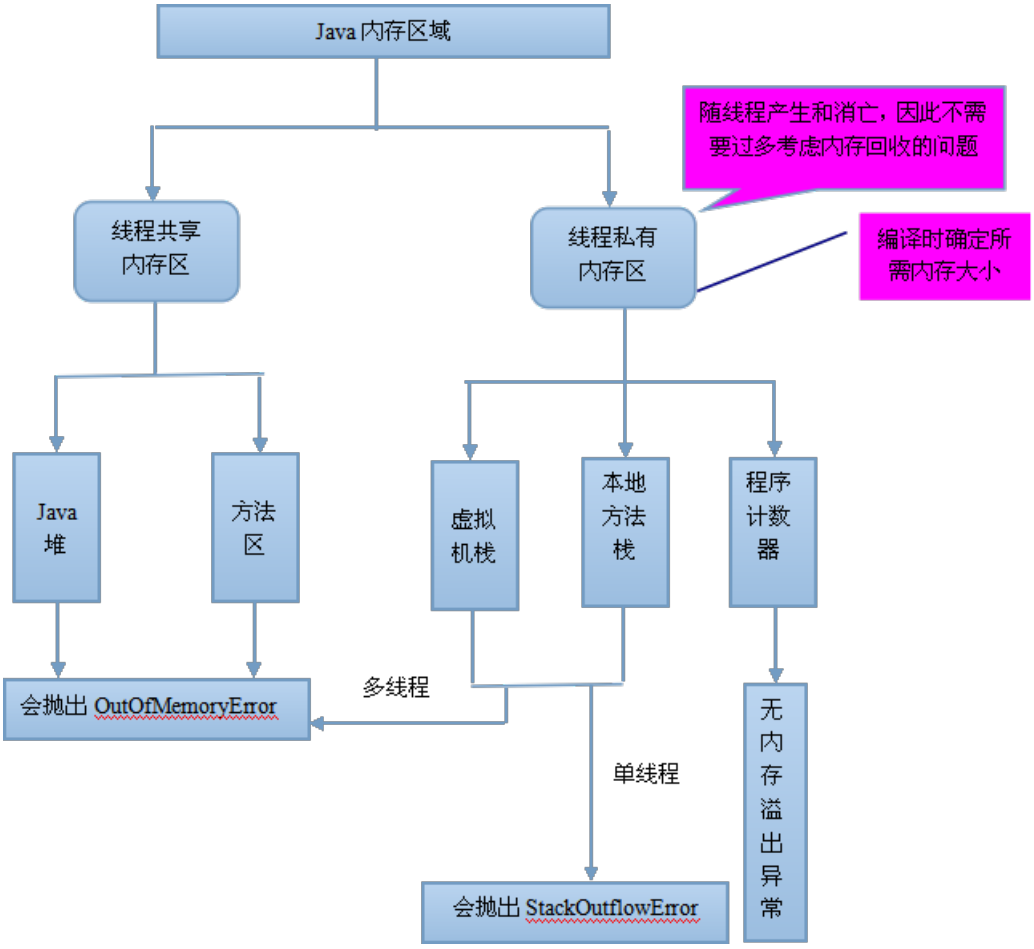
JVM

内存模型以及分区，需要详细到每个区放什么。

http://blog.csdn.net/ns_code/article/details/17565503

JVM所管理的内存分为以下几个运行时数据区：程序计数器、Java虚拟机栈、本地方法栈、Java堆、方法区。



程序计数器(Program Counter Register)

一块较小的内存空间，它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通过改变该计数器的值来选择下一条需要执行的字节码指令，分支、跳转、循环等基础功能都要依赖它来实现。每条线程都有一个独立的的程序计数器，各线程间的计数器互不影响，因此该区域是线程私有的。

当线程在执行一个Java方法时，该计数器记录的是正在执行的虚拟机字节码指令的地址，当线程在执行的是Native方法（调用本地操作系统方法）时，该计数器的值为空。另外，该内存区域是唯一一个在Java虚拟机规范中么有规定任何OOM（内存溢出：OutOfMemoryError）情况的区域。

Java虚拟机栈（Java Virtual Machine Stacks）

该区域也是线程私有的，它的生命周期也与线程相同。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，栈它是用于支持续虚拟机进行方法调用和方法执行的数据结构。对于执行引擎来讲，活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧，这个栈帧所关联的方法称为当前方法，执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。栈帧用于存储局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。在编译程序代码时，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定了，并且写入了方法表的Code属性之中。因此，一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

本地方法栈（Native Method Stacks）

该区域与虚拟机栈所发挥的作用非常相似，只是虚拟机栈为虚拟机执行Java方法服务，而本地方法栈则为使用到的本地操作系统（Native）方法服务。

Java堆（Java Heap）

Java Heap是Java虚拟机所管理的内存中最大的一块，它是所有线程共享的一块内存区域。几乎所有的对象实例和数组都在这类分配内存。Java Heap是垃圾收集器管理的主要区域，因此很多时候也被称为“GC堆”。

根据Java虚拟机规范的规定，Java堆可以处在物理上不连续的内存空间中，只要逻辑上是连续的即可。如果在堆中没有内存可分配时，并且堆也无法扩展时，将会抛出OutOfMemoryError异常。

方法区（Method Area）

方法区也是各个线程共享的内存区域，它用于存储已经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。方法区域又被称为“永久代”，但这仅仅对于Sun HotSpot来讲，JRockit和IBM J9虚拟机中并不存在永久代的概念。Java虚拟机规范把方法区描述为Java堆的一个逻辑部分，而且它和Java Heap一样不需要连续的内存，可以选择固定大小或可扩展，另外，虚拟机规范允许该区域可以选择不实现垃圾回收。相对而言，垃圾收集行为在这个区域比较少出现。该区域的内存回收目标主要针是对废弃常量的和无用类的回收。运行时常量池是方法区的一部分，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（Class文件常量池），用于存放编译器生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。运行时常量池相对于Class文件常量池的另一个重要特征是具备动态性，Java语言并不要求常量一定只能在编译期产生，也就是并非预置入Class文件中的常量池的内容才能进入方法区的运行时常量池，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比较多的是String类的intern（）方法。

根据Java虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

内存泄漏和内存溢出的差别

内存泄露是指分配出去的内存没有被回收回来，由于失去了对该内存区域的控制，因而造成了资源的浪费。Java中一般不会产生内存泄露，因为有垃圾回收器自动回收垃圾，但这也不绝对，当我们new了对象，并保存了其引用，但是后面一直没用它，而垃圾回收器又不会去回收它，这边会造成内存泄露，

内存溢出是指程序所需要的内存超出了系统所能分配的内存（包括动态扩展）的上限。

类型擦除

http://blog.csdn.net/ns_code/article/details/18011009

Java语言在JDK1.5之后引入的泛型实际上只在程序源码中存在，在编译后的字节码文件中，就已经被替换为了原来的原生类型，并且在相应的地方插入了强制转型代码，因此对于运行期的Java语言来说，`ArrayList<String>`和`ArrayList<Integer>`就是同一个类。所以泛型技术实际上是Java语言的一颗语法糖，Java语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型被称为伪泛型。

下面是一段简单的Java泛型代码：

```
Map<Integer,String> map = new HashMap<Integer,String>();
map.put(1,"No.1");
map.put(2,"No.2");
System.out.println(map.get(1));
System.out.println(map.get(2));
````

将这段Java代码编译成Class文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都变回了原生类型，如下面的代码所示：
```

```
Map map = new HashMap();
map.put(1,"No.1");
map.put(2,"No.2");
System.out.println((String)map.get(1));
System.out.println((String)map.get(2));
```

```
为了更详细地说明类型擦除，再看如下代码：
```

```
import java.util.List;
public class FanxingTest{
public void method(List list){
System.out.println("List String");
}
public void method(List list){
System.out.println("List Int");
}
}
```
```

当我用Javac编译器编译这段代码时，报出了如下错误：

FanxingTest.java:3: 名称冲突：method(java.util.List) 和 method

(java.util.List) 具有相同疑符

public void method(List list){

^

FanxingTest.java:6: 名称冲突：method(java.util.List) 和 metho

d(java.util.List) 具有相同疑符

public void method(List list){

^

2 错误

这是因为泛型List<String>和List<Integer>编译后都被擦除了，变成了一样的原生类型List，擦除动作导致这两个方法的特征签名变

把以上代码修改如下：

```
import java.util.List;
public class FanxingTest{
public int method(List list){
System.out.println("List String");
return 1;
}
public boolean method(List list){
System.out.println("List Int");
return true;
}
}
```

发现这时编译可以通过了（注意：Java语言中true和1没有关联，二者属于不同的类型，不能相互转换，不存在C语言中整数值非零即真的情

我们知道，Java代码中的方法特征签名只包括了方法名称、参数顺序和参数类型，并不包括方法的返回值，因此方法的返回值并不参与

堆里面的分区：Eden，survival from to，老年代，各自的特点。

对象创建方法，对象的内存分配，对象的访问定位。

对内存分配情况分析最常见的示例便是对象实例化：

Object obj = new Object(); ```

这段代码的执行会涉及java栈、Java堆、方法区三个最重要的内存区域。假设该语句出现在方法体中，及时对JVM虚拟机不了解的Java使用这，应该也知道obj会作为引用类型（reference）的数据保存在Java栈的本地变量表中，而会在Java堆中保存该引用的实例化对象，但可能并不知道，Java堆中还必须包含能查找到此对象类型数据的地址信息（如对象类型、父类、实现的接口、方法等），这些类型数据则保存在方法区中。

另外，由于reference类型在Java虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到Java堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄池和直接使用指针。

GC的两种判定方法：引用计数与引用链。

引用计数方式最基本的形态就是让每个被管理的对象与一个引用计数器关联在一起，该计数器记录着该对象当前被引用的次数，每当创建一个新的引用指向该对象时其计数器就加1，每当指向该对象的引用失效时计数器就减1。当该计数器的值降到0就认为对象死亡。

Java的内存回收机制可以形象地理解为在堆空间中引入了重力场，已经加载的类的静态变量和处于活动线程的堆栈空间的变量是这个空间的牵引对象。这里牵引对象是指按照Java语言规范，即便没有其它对象保持对它的引用也不能够被回收的对象，即Java内存空间中的本原对象。当然类可能被去加载，活动线程的堆栈也是不断变化的，牵引对象的集合也是不断变化的。对于堆空间中的任何一个对象，如果存在一条或者多条从某个或者某几个牵引对象到该对象的引用链，则就是可达对象，可以形象地理解为从牵引对象伸出的引用链将其拉住，避免掉到回收池中。

GC的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？

标记清除算法是最基础的收集算法，其他收集算法都是基于这种思想。标记清除算法分为“标记”和“清除”两个阶段：首先标记出需要回收的对象，标记完成之后统一清除对象。它的主要缺点：①.标记和清除过程效率不高 。②.标记清除之后会产生大量不连续的内存碎片。

标记整理，标记操作和“标记-清除”算法一致，后续操作不只是直接清理对象，而是在清理无用对象完成后让所有存活的对象都向一端移动，并更新引用其对象的指针。主要缺点：在标记-清除的基础上还需进行对象的移动，成本相对较高，好处则是不会产生内存碎片。

复制算法，它将可用内存容量划分为大小相等的两块，每次只使用其中的一块。当这一块用完之后，就将还存活的对象复制到另外一块上面，然后在把已使用过的内存空间一次理掉。这样使得每次都是对其中的一块进行内存回收，不会产生碎片等情况，只要移动堆订的指针，按顺序分配内存即可，实现简单，运行高效。主要缺点：内存缩小为原来的一半。

Minor GC与Full GC分别在什么时候发生？

Minor GC：通常是指对新生代的回收。指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快

Major GC：通常是指对年老代的回收。

Full GC：Major GC除并发gc外均需对整个堆进行扫描和回收。指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程） 。MajorGC 的速度一般会比 Minor GC 慢 10倍以上。

几种常用的内存调试工具：jmap、jstack、jconsole。

jmap（linux下特有，也是很常用的一个命令）观察运行中的jvm物理内存的占用情况。 参数如下： -heap：打印jvm heap的情况 -histo：打印jvm heap的直方图。其输出信息包括类名，对象数量，对象占用大小。 -histo：live ：同上，但是只答应存活对象的情况 -permstat：打印permanent generation heap情况 jstack（linux下特有）可以观察到jvm中当前所有线程的运行情况和线程当前状态 jconsole一个图形化界面，可以观察到java进程的gc，class，内存等信息 jstat最后要重点介绍下这个命令。这是jdk命令中比较重要，也是相当实用的一个命令，可以观察到classloader，compiler，gc相关信息 具体参数如下： -class：统计class loader行为信息 -compile：统计编译行为信息 -gc：统计jdk gc时heap信息 -gccapacity：统计不同的generations（不知道怎么翻译好，包括新生区，老年区，permanent区）相应的heap容量情况 -gccause：统计gc的情况，（同-gcutil）和引起gc的事件 -gcnew：统计gc时，新生代的情况 -gcnewcapacity：统计gc时，新生代heap容量 -gcold：统计gc时，老年区的情况 -gcoldcapacity：统计gc时，老年区heap容量 -gcpermcapacity：统计gc时，permanent区heap容量 -gcutil：统计gc时，heap情况 -printcompilation：不知道干什么的，一直没用过。

类加载的五个过程：加载、验证、准备、解析、初始化。

类加载过程

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、验证、准备、解析、初始化、使用、卸载。

其中类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始，这是为了支持Java语言的运行时绑定（也成为动态绑定或晚期绑定）。另外注意这里的几个阶段是按顺序开始，而不是按顺序进行或完成，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。

这里简要说明下Java中的绑定：绑定指的是把一个方法的调用与方法所在的类(方法主体)关联起来，对java来说，绑定分为静态绑定和动态绑定：

- 静态绑定：即前期绑定。在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。针对java，简单的可以理解为程序编译期的绑定。java当中的方法只有final，static，private和构造方法是前期绑定的。
- 动态绑定：即晚期绑定，也叫运行时绑定。在运行时根据具体对象的类型进行绑定。在java中，几乎所有的方法都是后期绑定的。

“加载”(Loading)阶段是“类加载”(Class Loading)过程的第一个阶段，在此阶段，虚拟机需要完成以下三件事情：

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Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

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。

类初始化是类加载过程的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

双亲委派模型：Bootstrap ClassLoader、Extension ClassLoader、ApplicationClassLoader。

1. 启动类加载器，负责将存放在\lib目录中的，或者被-Xbootclasspath参数所指定的路径中，并且是虚拟机识别的（仅按照文件名识别，如rt.jar，名字不符合的类库即时放在lib目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被java程序直接引用。
2. 扩展类加载器：负责加载\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用该类加载器。
3. 应用程序类加载器：负责加载用户路径上所指定的类库，开发者可以直接使用这个类加载器，也是默认的类加载器。 三种加载器的关系：启动类加载器->扩展类加载器->应用程序类加载器->自定义类加载器。

这种关系即为类加载器的双亲委派模型。其要求除启动类加载器外，其余的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不以继承关系实现，而是用组合的方式来复用父类的代码。

双亲委派模型的工作过程：如果一个类加载器接收到了类加载的请求，它首先把这个请求委托给他的父类加载器去完成，每个层次的类加载器都是如此，因此所有的加载请求都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它在搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

好处：java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类java.lang.Object，它存放在rt.jar中，无论哪个类加载器要加载这个类，最终都会委派给启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果用户自己写了一个名为java.lang.Object的类，并放在程序的Classpath中，那系统中将会出现多个不同的Object类，java类型体系中最基础的行为也无法保证，应用程序也会变得一片混乱。

实现：在java.lang.ClassLoader的loadClass()方法中，先检查是否已经被加载过，若没有加载则调用父类加载器的loadClass()方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父加载失败，则抛出ClassNotFoundException异常后，再调用自己的findClass()方法进行加载。

分派：静态分派与动态分派。

静态分派与重载有关，虚拟机在重载时是通过参数的静态类型，而不是运行时的实际类型作为判定依据的；静态类型在编译期是可知的； 动态分派与重写（Override）相关，invokevirtual(调用实例方法)指令执行的第一步就是在运行期确定接收者的实际类型，根据实际类型进行方法调用；

GC收集器有哪些？CMS收集器与G1收集器的特点。

自动内存管理机制，GC算法，运行时数据区结构，可达性分析工作原理，如何分配对象内存

反射机制，双亲委派机制，类加载器的种类

Jvm内存模型，先行发生原则，violate关键字作用