

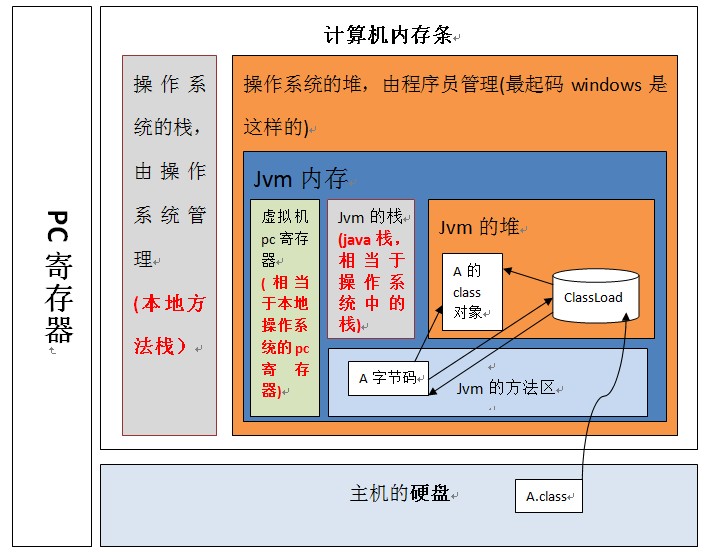
JVM（java虚拟机）：只认识.class文件，能够将class文件中的字节码指令进行识别并调用操作系统上的API完成动作，所以说，JVM是java能够跨平台的核心。

JRE（java运行时环境）：主要包含两部分，JVM的标准实现和java的一些基本类库。

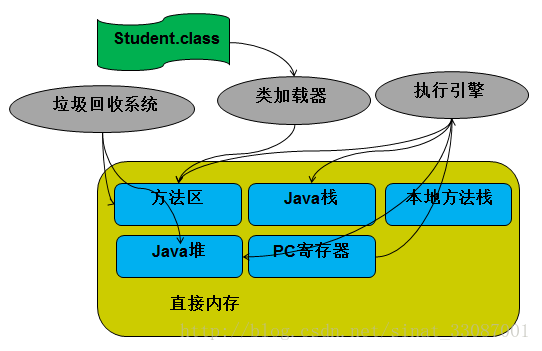
JDK（java开发工具包）：JDK是整个java的核心，它集成了JRE和一些好用的小工具。

JDK是Java语言的软件开发工具包。在JDK的安装目录下有一个JRE目录，里面有两个文件夹bin和lib，在这里可以认为bin里的就是JVM，lib中则是JVM工作所需要的类库，而JVM和lib合起来就是成为JRE。我们利用JDK（调用JAVA API）开发属于我们自己的java程序，通过JDK中的编译程序（javac）将我们的文本java文件编译成JAVA字节码，在JRE上运行这下java字节码，JVM解析这些字节码，映射到CPU指令集或OS的系统调用。

三者关系：JDK>JRE>JVM

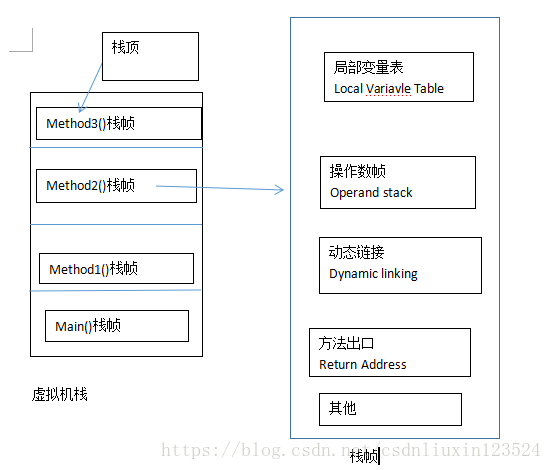


1. java自动管理堆和栈，程序员不能直接的设置堆和栈
2. JVM是分布在操作系统的堆中。因为操作系统的栈是操作系统管理的，它随时会被回收，所以如果JVM放在栈中，那java的null对象就很难确定被谁回收了，那GC的存在意义就没有了，而要对栈做到自动释放也是JVM需要考虑的，所以放在堆中最合适。
3. 操作系统的堆栈：
   1. 堆（操作系统）：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收，分配方式类似于链表
   2. 栈（操作系统）：由操作系统自动分配释放，存放函数的参数值，局部变量值等。操作方式与数据结构中的栈相似。
4. 类加载到JVM的过程：当一个classLoader启动时classLoader的生存地点在JVM的堆中，然后它会去主机硬盘上将A.class装载到JVM的方法区，方法区中的这个字节文件会被虚拟机拿来new A字节码，然后在堆内存中生成一个A字节码的对象，然后**A字节码这个内存文件有两个引用：一个指向A的class对象，一个指向加载自己的classLoader**。
5. Java虚拟机的生命周期：生命周期起点是当一个java应用main函数启动时虚拟机也同时被启动，而只有当虚拟机实例中所有非守护线程都结束时，java虚拟机实例才结束生命。
6. Java虚拟机和main方法的关系：main函数就是一个java应用的入口，main函数被执行时，java虚拟机就启动了。**启动了几个main函数就启动了几个java应用，同时也启动了几个java的虚拟机。**
7. **Java虚拟机中有两种线程：一种叫守护线程，一种叫非守护线程（普通线程）。只要有任何非守护线程还没有结束，java虚拟机的实例就不会退出，守护线程中产生的线程也是守护线程。GC就是典型的守护线程，main函数就是一个非守护线程。**

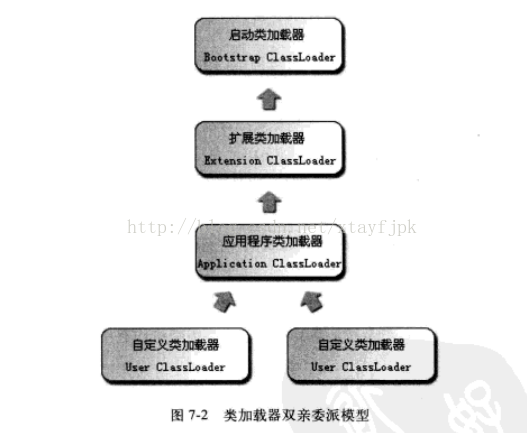


JVM内部执行运行流程图

1. 上图分为“功能区”和“数据区”（一动一静），功能区：垃圾回收系统、类加载器、执行引擎；数据区：方法区、java栈，本地方法栈、java堆、pc寄存器
2. 当一个程序启动前，它的class会被**类装载器**装入**方法区**，**执行引擎**读取方法区的字节码自适应解析，边解析边运行（其中一种方式），然后**pc寄存器**指向main函数所在的位置，虚拟机开始为main函数在**java栈**中预留一个栈帧（每一个方法对应一个栈帧），并且在**java堆**中给对象和数据分配内存，然后开始跑main函数，main函数里的代码被执行引擎映射成本地操作系统里相应的实现，然后调用本地方法接口，本地方法运行时，操作系统会为本地方法分配**本地方法栈**，用来存储一些临时变量，然后运行本地方法，调用操作系统API，并在适当的时候执行**垃圾回收系统**等等。
3. 启动一个java虚拟机程序就是启动了一个进程，启动的同时就是在操作系统的堆内存中开辟一块JVM内存区。
4. **JVM结构图各模块的生命周期总结：java栈、本地方法栈、pc寄存器（程序计数器）这三个模块是线程私有的，有多少个线程就有多少个这三个模块，生命周期跟所属线程的声明周期一致。其余是跟JVM的生命周期一致。**
5. JVM内存包含两个子系统和两个组件。两个子系统是：ClassLoader（类加载）子系统和ExecutionEngine（执行引擎）子系统；两个组件是：Runtime Data Area（运行时数据区域）组件和Native Interface（本地库接口）组件。
6. JVM结构图各模块详解：
   1. **程序计数器（program counter Register）**：也叫pc寄存器，是一块较小的内存空间，它可以看成当前线程所执行的字节码的行号指示器。在虚拟机的概念模型中，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的完成。（1）、PC寄存器的功能时存放伪指令，更确切的说存放的是将要执行指令的地址。（2）、当虚拟机正在执行的方法是一个本地（native）方法的时候，JVM的pc寄存器存储的值是undefined。（3）、线程私有的，它的生命周期和线程相同，每个线程都有一个。（4）、次内存区域是唯一一个在JVM规范中没有规定任务outOfMemoryError情况的区域。
   2. **Java虚拟机栈（java virtual machine stack）**：（1）、线程私有，它的生命周期与线程相同，每个线程都有一个。（2）、每个线程创建的同时会创建一个JVM栈，JVM栈中每一个栈帧存放当前线程中局部基本类型的变量、部分的返回结果、非基本类型的对象在JVM栈中存放一个指向堆上的地址。（3）、每一个方法从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。（4）、JVM栈的最小单位可以理解为一个个栈帧，一个方法对应一个栈帧，一个栈帧可以执行很多指令。（5）、栈运行原理：栈中的数据都是以栈帧的格式存在，栈帧是一个内存去块，是一个数据集，是一个有关方法的运行期数据的数据集，当一个方法A（method1）被调用时就产生了一个栈帧F1，并被压入到栈中，A方法又调用了B(method2)方法，于是产生栈帧F2也被压入栈，B方法又调用了C(method3)方法，于是产生栈帧F3也被压入栈…… 依次执行完毕后，先弹出后进......F3栈帧，再弹出F2栈帧，再弹出F1栈帧。



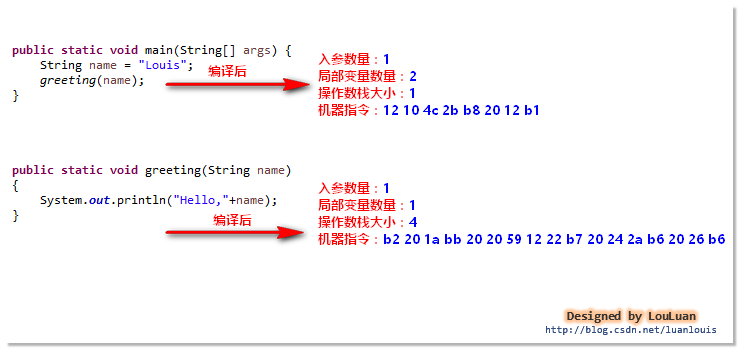
* 1. **本地方法栈（native method stack）**：(1)、本地方法：JVM的本地方法是指方法的修饰符是带有native的，但是方法不是用java代码写的，因此不能与abstract连用，这类方法存在的意义是填补java代码不方便实现的缺陷而提出的。（2）、java栈为虚拟机执行java方法服务，而本地方法栈为虚拟机执行本地方法服务。（3）、线程私有的，生命周期和线程相同，每个线程都有一个。
  2. **Java堆（java heap）**：（1）、java堆是JVM所管理内存最大的一块，是垃圾收集器管理的主要区域，因此很多时候也被称为GC堆（Garbage Collected Heap）。（2）、堆是所有线程共享的，在虚拟机启动的时候创建。（3）、唯一目的就是存放对象实例，几乎所有的对象实例以及数组都要在这里分配内存。（4）、java堆是计算机物理存储上不连续的、逻辑上是连续的，也是大小可调节的（通过-Xms和-Xmx控制）。（5）、如果在堆中没有内存完成实例的分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。
  3. **方法区（method area）**：（1）、在虚拟机启动的时候创建，所有JVM线程共享。（2）、不需要连续的内存空间和可以固定大小或者可扩展，还可以选择不实现垃圾收集。（3）、用于存放已被虚拟机加载的类的信息、常量、静态变量、以及编译后的方法实现的二进制形式的机器指令集等数据。（4）、运行时常量池（Runtime Constant Pool）也是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池（Constant Pool Table）,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。
  4. **类加载器子系统（class loader subsystem）**：根据给定的全限定类名（如java.lang.Object）来装载class文件的内容到运行时数据区域的方法区。Java程序员可以继承java.lang.ClassLoader类来写自己的类加载器。**JVM在加载类时默认采用双亲委派机制：就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成类加载任务时，才自己区加载。**例如：当JVM要加载Test.class的时候：（1）、首先会到**自定义加载器（User ClassLoader）**中查找（其实时看运行时数据区的方法区有没有加载），如果已经加载过，则返回字节码。（2）、如果自定义加载器没有加载过，则询问上一层**应用程序加载器（Application ClassLoader）**是否已经加载过。（3）、如果没有加载过，则询问上一层**扩展类加载器（Extension ClassLoader）**是否加载过。（4）、如果没有加载过，则继续询问上一层**启动类加载器（Bootstrap ClassLoader）**是否加载过。（5）、如果依然没有加载过，则到自己指定类加载路径下查看是否有Test.class字节码，有则返回，没有通知下一层扩展类加载器到自己指定的类加载路径下查看。（6）、依次类推，最后到自定义类加载器指定的路径下还没有找到Test.class字节码，则抛出异常ClassNotFoundException。

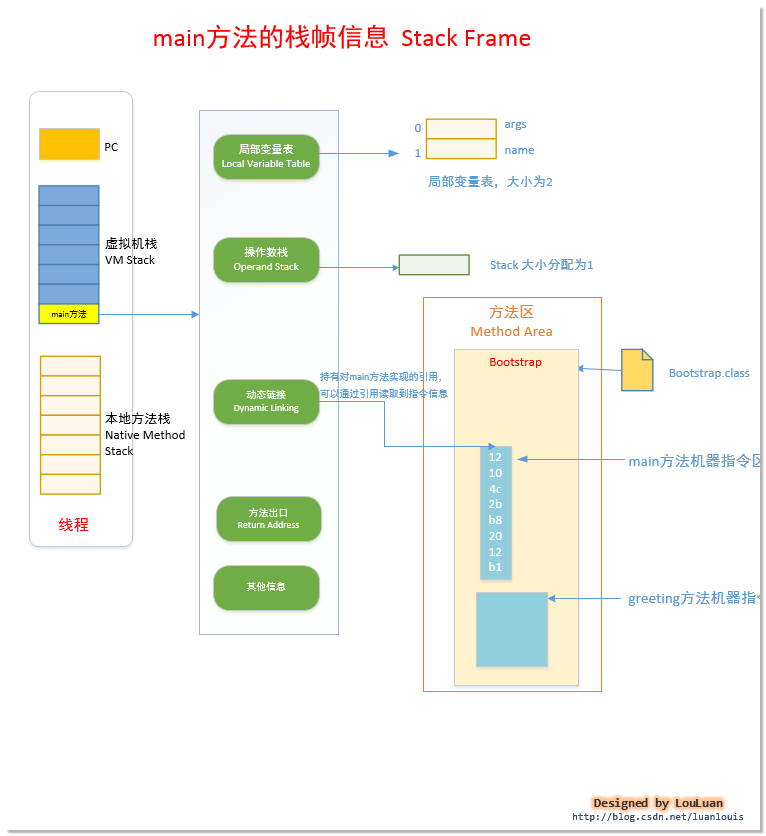


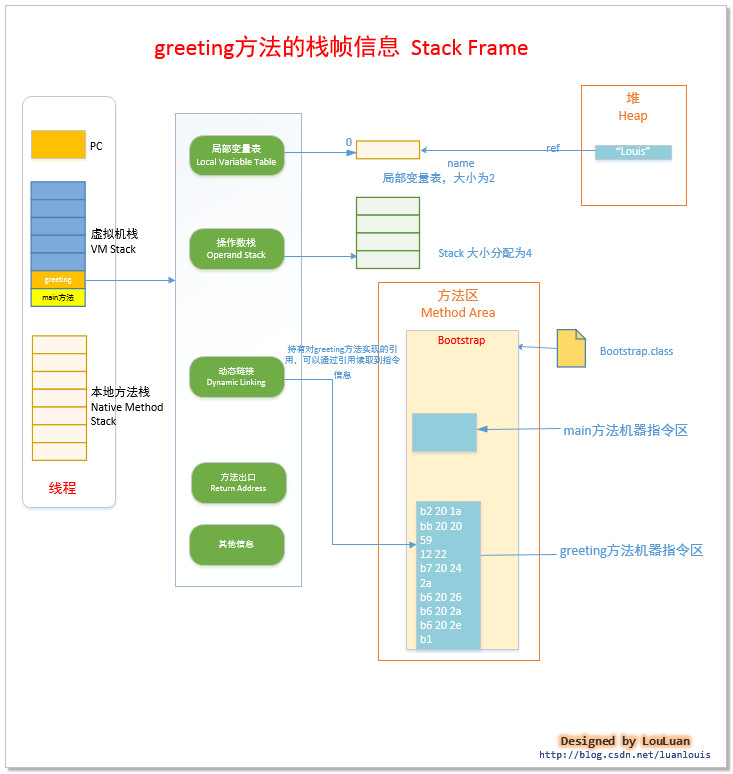
* 1. **执行引擎子系统（Execution engine subsystem）**：（1）、负责执行来自类加载器子系统中被加载类中在方法区包含的指令集，通俗讲就是类加载器子系统把代码逻辑都以指令的形式加载到了方法区，执行引擎就负责执行这些指令就行。（2）、程序在JVM主要执行的过程是执行引擎与运行时数据区不断交互的过程。
  2. **GC（垃圾回收）机制**：堆内存分为三部分：（1）、**新生区**：新生区是类的诞生、成长、消亡的区域，一个类在这里产生、应用、最后被垃圾回收器回，结束生命。新生区又分为两部分：**伊甸区（Eden space）和幸存者区（Survivor space）**，所有类都是在伊甸区被new出来的。幸存者区有两个：**0区（survivor 0 space）和1区（survivor 1 space）**。（2）、**养老区**：养老区用于保存从新生区筛选出来的java对象，一般池对象都在这个区域活跃。（3）、**永久区**：永久存储区是一个常驻内存区域，用于存放JDK自身所携带的CLASS、INTERFACE的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭JVM才会释放此区域所占用的内存。



1. native方法PC寄存器中为undefined原因：类加载时，native方法就被保存在了本地方法栈中，当需要调用native方法时，调用的是一个指向本地方法栈中某方法的地址，然后执行方法直接与操作系统交互，返回运行结果。整个过程并没有经过执行引擎的解释器把字节码解释成操作系统语言，PC计数器也就没有起做用。
2. 类加载器注意：（1）、类加载器代码本身也是java类，因此类加载器本身也是要被加载的，因此必须有一个类加载器不是java类，这就是bootStrap，是使用c++写的其他都是java实现。（2）、Application ClassLoader、Extension ClassLoader、Bootstrap ClassLoader三个是父子类加载器关系，但是并没有使用继承，而是使用了组合关系。具备了一种优先的层次关系，越是基础的类，越是被上层的类加载器进行加载，保证了java的稳定性。
3. 以下时运行时数据区各个模块协同工作图：







**常用的垃圾回收算法**

1. 引用计数法：该算法的实现是假设有一个对象A，任何一个对象对A进行引用，那么A的引用计数器进行+1，当引用失败时，对象A的引用技术器就-1，如果对象A的计数器的值为0，说明对象A没有引用了，可以被回收了。但目前主流的java虚拟机中并没有使用引用计数法，其中主要原因是引用计数法无法回收循环引用的变量。

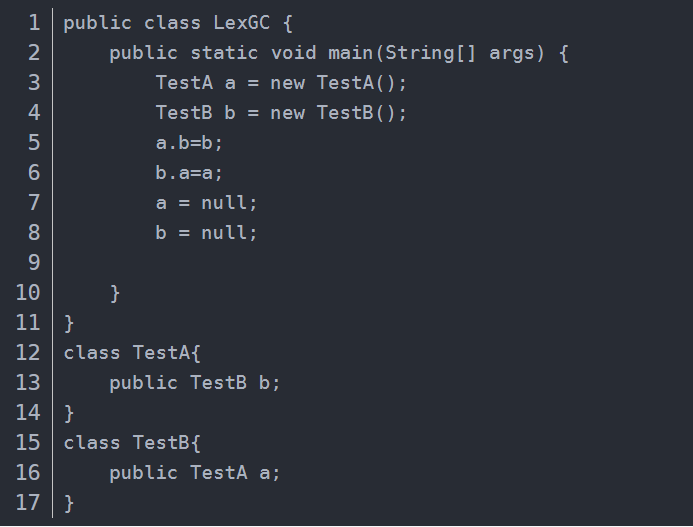
优点：

1. 实时性较高，无需等到内存不够的时候，才开始回收，运行时根据对象的计数器是为0，就可以直接回收。
2. 在垃圾回收过程中，应用无需挂起。如果申请内存时，内存不足，则立刻报内存不足错误。
3. 区域性，更新对象的计数器时，只是影响到该对象，不会扫描全部对象。

缺点：

1. 每次对象被引用时，都需要去更新计数器，有一点时间开销。
2. 浪费cpu资源，即使内存够用，仍然在运行时进行计数器的统计
3. 无法解决循环问题。

例如下面情况：**虽然a和b都为null，但是由于a和b存在循环引用，这样a和b永远都不会被回收。**



1. 标记清除算法：该算法是将垃圾的回收分为两个阶段，分别为标记和清除，首先标记，会采用可达性分析算法，找出可用和不可用的对象，将可用对象标记为1，不可用的为0，然后进入第二阶段--清除，把标记为0的对象进行垃圾回收，然后将剩余的对象标记为0，等待下一次标记。

可达性分析算法：算法的基本思路是以“GC ROOTS“为起始点，从这个节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC ROOTS没有任何引用链时，就说明从GC ROOTS到这个对象不可达，则这个对象会被判定为可回收对象。

优点：解决了计数回收算法中循环引用对象回收的问题。

缺点：

1. 效率较低，在标记和清除阶段都需要遍历所有对象，而且在GC的时候会短暂的停止应用程序，用户体验较差。
2. 通过标记清除算法清理出来的内存，碎片化比较严重，因为被回收的对象可能存在于内存的各个角落，所以清理出来的内存是不连贯的。
3. 标记压缩算法：标记压缩算法是在标记清除算法上做了改进和优化，标记阶段都是一样的，只不过在清理阶段，并不是简单的清理未标记的对象，而是将存活的对象压缩到内存的一端，然后清理边界以外的垃圾。从而解决碎片化的问题。

优点：

1. 解决了计数算法不能处理的循环引用问题。
2. 解决了标记清除算法内存碎片严重，不连贯的问题。

缺点：相比于标记清除算法，多了一步对象移动内存位置的步骤，效率又有一些下降。

1. 复制算法：复制算法就是将内存空间一分为二，存储时只使用其中一块，当进行垃圾回收的时候，找出正在使用的对象，并将这些对象复制到另一块内存空间中，然后将内存清空，交换两个空间的角色，实现垃圾回收。

优点：

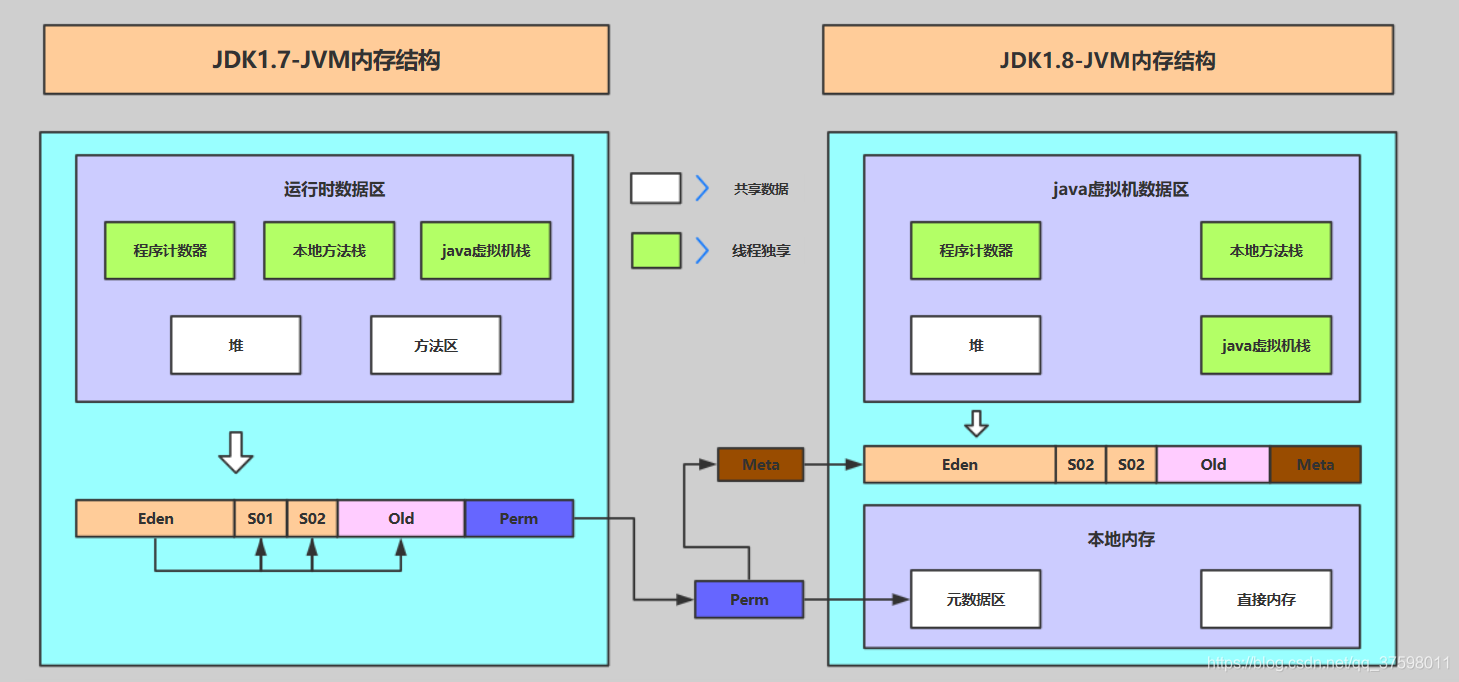
1. 如果垃圾对象较多的情况下，该算法效率较高。
2. 垃圾清理后，内存不会出现碎片化。

缺点：

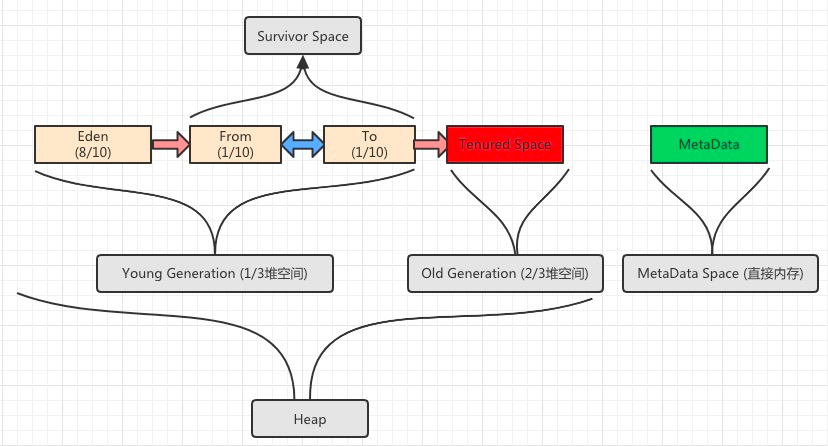
1. 在垃圾较少的情况下不适用，不如老年区。
2. 分配的2块内存空间，在同一时刻，只能使用一半，内存使用率较低。

**JDK1.7和JDK1.8的JVM划分**

1. 元空间区取代了永久代，永久代原本主要存放class和meta的信息。而元空间的本质和永久代类似，都是对JVM规范中方法区的实现。不过元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。



1. Jdk1.7后字符串常量池从永久代中隔离出来，存放在堆中。方法区原本跟堆一样是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、静态变量等数据。有人称为永久代，但它有一个问题，很容易出现内存溢出。
2. 直接内存不是虚拟机运行时数据区的一部分也不是java虚拟机规范中定义的内存区域。在NIO中，它可以使用native函数直接分配堆外内存，然后通过一个存储在java堆中的DirectByteBuffer对象作为这块内存的引用进行操作。这样可以提高性能，因为避免了java堆和native堆中来回赋值数据，该区域不受堆大小限制，但是会受本机总内存（RAM或SWAP）大小等限制。如果忽略这点将堆大小设置过大使得各个内存区域大小总和大于物理内存，他会在动态扩展时出现OutOfMemoryError异常。
3. 元空间取代永久代原因：**\*字符串存在永久代，容易出现性能问题和内存溢出；\*类及方法得信息等比较难确定其大小，因此对于永久代得大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出；\*永久代会为GC带来不必要得复杂度，并且回收效率偏低。**
4. CG过程及默认堆内存分配：



1. Eden区满或者快满的时候进行一次清理（minor GC），不被引用的对象直接被清理，还有引用的对象放到S0或S1区中的一个，假如第一次选择了S0，它会逐步将活着的对象拷贝到S0区域，可若是S0满了，剩下活着的对象只能放到old区，然后将Eden区清空。
2. 当第二次Eden区满的时候，就将Eden区+S0区活着的对象，迁移到S1中，若S1放不下，就会将剩下的放到old区，最后将Eden区+S0区清空。
3. 依次类推，反反复复屡次没有被淘汰的对象，将会放入old区，默认15次。计数器会在对象的头部记录它交换的次数。
4. 依次循环，直到old区和Eden区都快满了的时候，会对整个这个内存区域进行一次大清洗（Full GC）。
5. 还好

**JVM优化参数**