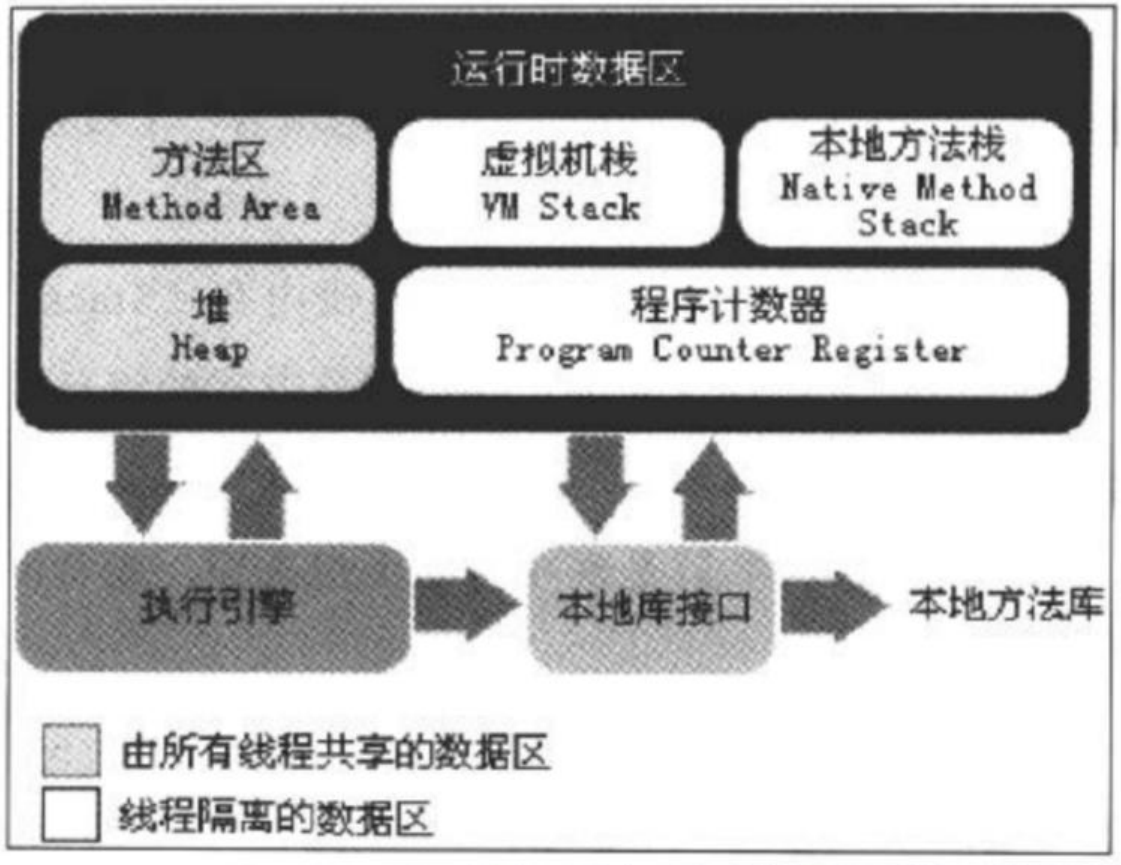
链滴

JVM 相关工作原理

作者：alex18595752445

原文链接：https://ld246.com/article/1583666947310

来源网站：链滴



方法区：**是线程共享的内存区域**，用来存储类加载的信息、常量、静态变量、即时编译器编译后的代等。其中方法区中还有个经常会用到的区域叫做运行时常量池，主要用于存储一些常量，当创建一个量时，首先会在运行时常量池查看是否有，有则直接使用，否则重新创建。

原文链接：JVM 相关工作原理

堆：堆是最大的一块内存区域，也是垃圾回收管理的主要区域，主要用于存放对象实例。

程序计时器：**线程私有的**，每个线程都会分配一个线程计时器，用来表示当前线程执行的字节码的行指示器。在多线程中，一个线程执行的时候释放锁，另一个线程执行完，再回来执行前面线程的时候就是通过程序计时器来获取继续执行的位置。

虚拟机栈：虚拟机栈主要存储基本数据类型变量和引用类型变量。其中与堆的区别就是如：Obj obj= ew Object()；等号左边则是在虚拟机栈上分配栈区存储引用类型变量的句柄 Obj obj，等号右边则是储对象实例，栈区的句柄是指向堆区的对象实例的，一般通过句柄访问堆区的对象实例。

本地方法栈（线程私有）：与虚拟机栈意义相似，区别在于虚拟机栈用于使 Java 方法，而本地方法则是针对于 Native 方法服务。

一、JVM的生命周期

1.

JVM实例对应了一个独立运行的Java程序它是进程级别

a)　启动。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

b)　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：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 程序也可标明自己创建的线程是守护线程

c)　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

1.

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的体系结构

- 类装载器（ClassLoader）（用来装载。class 文件）

特别注意：双亲委派机制

- 执行引擎（执行字节码，或者执行本地方法）
- 运行时数据区（方法区、堆、Java 栈、PC 寄存器、本地方法栈）

类加载的时机：

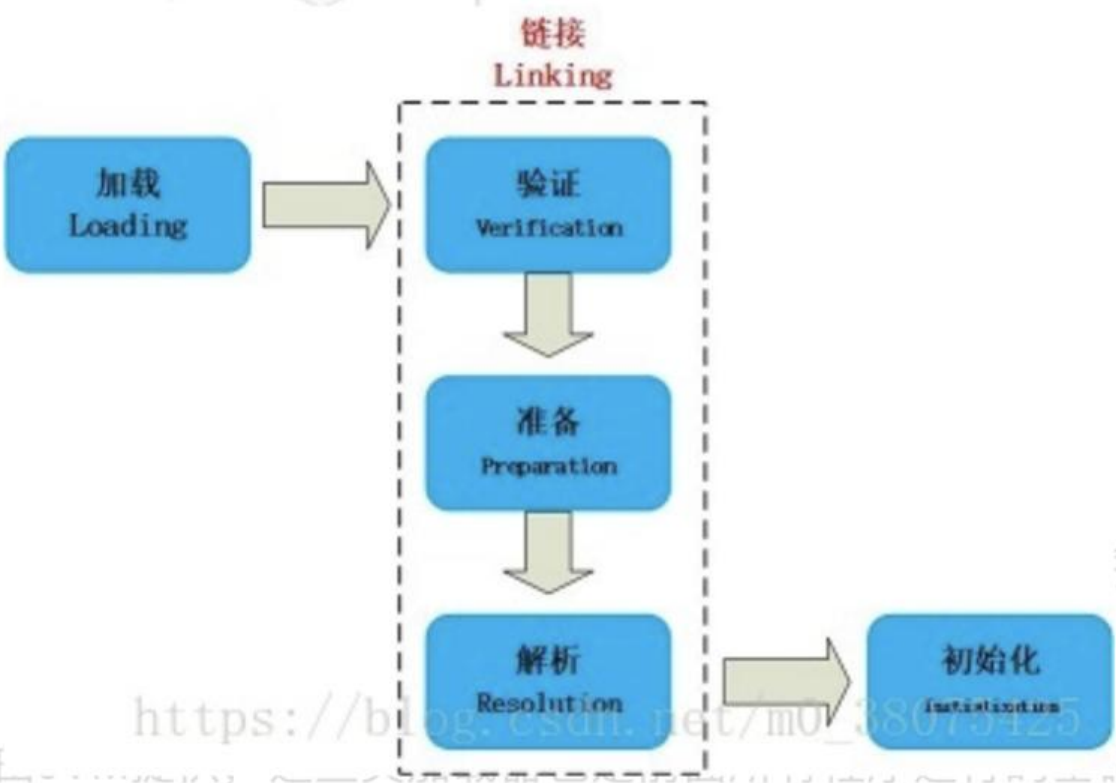
- 创建类的实例，也就是 new 一个对象
- 访问某个类或接口的静态变量，或者对该静态变量赋值
- 调用类的静态方法
- 反射（Class.forName("com.lyj.load")）
- 初始化一个类的子类（会首先初始化子类的父类）

● JVM 启动时标明的启动类，即文件名和类名相同的那个类

注意：

对于一个 final 类型的静态变量，如果该变量的值在编译时就可以确定下来，那么这个变量相当于“变量”。Java 编译器会在编译时直接把这个变量出现的地方替换成它的值，因此即使程序使用该静态量，也不会导致该类的初始化。反之，如果 final 类型的静态 Field 的值不能在编译时确定下来，则须等到运行时才可以确定该变量的值，如果通过该类来访问它的静态变量，则会导致该类被初始化。

三、JVM 类加载器（类加载的过程（5 个阶段））



JVM 整个类加载过程的步骤：

1. 装载

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 完成类的加载，同样，也采用以上三个元素来标识一个被加载了的类：类名 +

包名 +ClassLoader 实例 ID。

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。

完成校验后，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（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，在四种情况下初始过程会被触发执行：

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。

类加载器分为两种：根加载器、用户自定义加载器

根类装载器是 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。

JVM 装载顺序：

Jvm启动时，由Bootstrap向User-Defined方向加载类；

应用进行ClassLoader时，由User-Defined向Bootstrap方向查找并加载类；

JVM 预定义有三种类加载器，当一个 JVM 启动的时候，Java 开始使用如下三种类加载器：前三个为 JVM 预设类加载器

1.Bootstrap ClassLoader

这是 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

JVM 用此 classloader 来加载扩展功能的一些 jar 包。

3. System ClassLoader

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

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

ClassLoader 抽象类的几个关键方法：

（1）loadClass

此方法负责加载指定名字的类，ClassLoader 的实现方法为先从已经加载的类中寻找，如没有则继续从 parent ClassLoader 中寻找，如仍然没找到，则从 System ClassLoader 中寻找，最后再调用 findClass 方法来寻找，如要改变类的加载顺序，则可覆盖此方法

（2） findLoadedClass

此方法负责从当前 ClassLoader 实例对象的缓存中寻找已加载的类，调用的为 native 的方法。

（3）findClass

此方法直接抛出 ClassNotFoundException，因此需要通过覆盖 loadClass 或此方法来以自定义的方加载相应的类。

（4） findSystemClass

此方法负责从 System ClassLoader 中寻找类，如未找到，则继续从 Bootstrap ClassLoader 中寻找如仍然为找到，则返回 null。

（5） defineClass

此方法负责将二进制的字节码转换为 Class 对象

（6）resolveClass

此方法负责完成 Class 对象的链接，如已链接过，则会直接返回。

四、JVM 执行引擎

在执行方法时 JVM 提供了四种指令来执行：

（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私有方法时。

主要的执行技术有：

解释，即时编译，自适应优化、芯片级直接执行

（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

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后台线程，将其编译为本地代码，并进行优化。若方法不再频繁使用，则取消编译过的代码，仍对其行解释执行。

五、JVM 运行时数据区

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 寄存器中存储任何信息。

第二块：JVM 栈

JVM 栈是线程私有的，每个线程创建的同时都会创建 JVM 栈，JVM 栈中存放的为当前线程中局部基类型的变量（Java 中定义的八种基本类型：boolean、char、byte、short、int、long、float、double）、部分的返回结果以及 Stack Frame，非基本类型的对象在 JVM 栈上仅存放一个指向堆上的地址

第三块：堆（Heap）

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 创建的对象的内存在此分配，Heap 中的对象的内存需要等待 GC 进行回收。

（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）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 LAB（Thread Local Allocation Buffer），其大小由 JVM 根据运行的情况计算而得，在 TLAB 上分对象时不需要加锁，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对象分配起来更加高效。

第四块：方法区域（Method Area）

（1）在 Sun JDK 中这块区域对应的为 PermanetGeneration，又称为持久代。

（2）方法区域存放了所加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为 final 类的常量、类中的 Field 信息、类中的方法信息，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，同时方法区域是全局共享的，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，会出 OutOfMemory 的错误信息。

第五块：运行时常量池（Runtime Constant Pool）

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。

六、JVM 垃圾回收

GC 的基本原理：**将内存中不再被使用的对象进行回收，GC 中用于回收的方法称为收集器，由于 GC 需要消耗一些资源和时间，Java 在对对象的生命周期特征进行分析后，按照新生代、旧生代的方式来对象进行收集，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）对旧生代的对象的收集称为 Full GC；

（3）程序中主动调用 System.gc()强制执行的 GC 为 Full GC。

不同的对象引用类型， GC 会采用不同的方法进行回收，JVM 对象的引用分为了四种类型：（JDK1 2 之后引入）

（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 时才会回收）。 eg：new a（）；、

```java
public class Referred {

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Referred对象被垃圾收集");
    }
}

public class StrongRef {

    public static void collect() throws InterruptedException {
        System.out.println("开始垃圾回收...");
        System.gc();
        System.out.println("结束垃圾回收.....");
        Thread.sleep(2000);
    }

    public static void main(String[] args) throws InterruptedException{

        System.out.println("创建一个强引用---->");

        Referred strong = new Referred();

        StrongRef.collect();

        System.out.println("删去引用---->");

        strong = null;
        StrongRef.collect();;
        System.out.println("done");
    }
}
```

出：

```
创建一个强引用—>
开始垃圾回收...
结束垃圾回收.....
开始垃圾回收...
结束垃圾回收.....
> Referred对象被垃圾收集
done
```

（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会被 GC）。**提供了 SoftReference 类来实现软引用**

```java
public class SoftRef {

    public static void collect() throws InterruptedException {
        System.out.println("开始垃圾收集...");
        System.gc();
        System.out.println("结束垃圾收集...");
        Thread.sleep(2000);
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("创建一个软引用--->");

        Referred strong = new Referred();
        SoftReference<Referred> soft = new SoftReference<Referred>(strong);

        SoftRef.collect();

        System.out.println("删除引用--->");

        strong = null;
        SoftRef.collect();

        System.out.println("开始堆占用");

        try {
            List<SoftRef> heap = new ArrayList<>(100);
            while (true) {
                heap.add(new SoftRef());
            }
        } catch (OutOfMemoryError e) {
            // 软引用对象应该在这个之前被收集
            System.out.println("内存溢出...");
        }

        System.out.println("Done");
    }
}
```

m options： `-Xmx100m -Xms100m`

```
创建一个软引用—>
开始垃圾收集...
结束垃圾收集...
删除引用—>
开始垃圾收集...
结束垃圾收集...
开始堆占用
Referred对象被垃圾收集
内存溢出...
Done
```

原文链接：JVM 相关工作原理

（3）弱引用：在 GC 时一定会被 GC 回收。**提供了 WeakReference 类来实现弱引用。**

```
public class WeakRef {

    public static void collect() throws InterruptedException {
        System.out.println("开始垃圾收集...");
        System.gc();
        System.out.println("结束垃圾收集...");
        Thread.sleep(2000);
    }

    /**
   * @param args
   * @throws InterruptedException
   */
    public static void main(String[] args) throws InterruptedException {
        System.out.println("创建一个弱引用--->");

        Referred strong = new Referred();
        WeakReference<Referred> weak = new WeakReference<>(strong);

        WeakRef.collect();
        System.out.println("删除引用--->");

        strong = null;
        WeakRef.collect();

        System.out.println("Done");
    }
}
```

输出：

```
创建一个弱引用—>
开始垃圾收集...
结束垃圾收集...
删除引用—>
开始垃圾收集...
结束垃圾收集...
Referred对象被垃圾收集
Done
```

原文链接：JVM 相关工作原理

（4）虚引用（幽灵引用、幻影引用）：由于虚引用只是用来得知对象是否被 GC。PhantomReference 类来实现虚引用。

```
    public static class Referred {
        // Note! 如果这里重写了finalize方法,那么PhantomReference不会追加到ReferenceQueue中
//      @Override
        protected void finalize() throws Throwable {
            System.out.println("Referred对象被垃圾收集");
        }
    }

    public static void collect() throws InterruptedException {
        System.out.println("开始垃圾收集...");
        System.gc();
        System.out.println("结束垃圾收集...");
        Thread.sleep(2000);
    }

    /**
   * 执行结果
   * 创建一个虚引用--->
   * 开始垃圾收集...
   * 结束垃圾收集...
   * 你需要清理一些东西了
   * Done
   * @param args
   * @throws InterruptedException
   */
    public static void main(String[] args) throws InterruptedException {
        System.out.println("创建一个虚引用--->");

        ReferenceQueue dead = new ReferenceQueue();
        Map<Reference, String> cleanUpMap = new HashMap<>();

        Referred strong = new Referred();
        PhantomReference<Referred> phantom = new PhantomReference<>(strong,
        cleanUpMap.put(phantom, "你需要清理一些东西了");

        strong = null;
        PhantomRef.collect();

        Reference reference = dead.poll();
        if (reference != null) {
            System.out.println(cleanUpMap.remove(reference));
        } else {
            System.out.println("reference为空");
        }
        System.out.println("Done");
    }
}
```

输出：

```
创建一个强引用—->
开始垃圾回收...
结束垃圾回收.....
删去引用—->
开始垃圾回收...
结束垃圾回收.....
Referred对象被垃圾收集
done
```

输出：

```
创建一个强引用—->
开始垃圾回收...
结束垃圾回收.....
删去引用—->
开始垃圾回收...
结束垃圾回收.....
Referred对象被垃圾收集
done
```

七、类加载机制

1.JVM 的类加载机制主要有如下 3 种。

● 全盘负责：所谓全盘负责，就是当一个类加载器负责加载某个 Class 时，该 Class 所依赖和引用其他 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。（1+n）

● 双亲委派：所谓的双亲委派，则是先让父类加载器试图加载该 Class，只有在父类加载器无法加载类时才尝试从自己的类路径中加载该类。通俗的讲，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加载器，依次递归，如果父加载器可以完成类加载任务，就成功返回；只有加载器无法完成此加载任务时，才自己去加载。（父级先加载、子集在上）

● 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，加载器先从缓存区中搜寻该 Class，只有当缓存区中不存在该 Class 对象时，系统才会读取该类对应

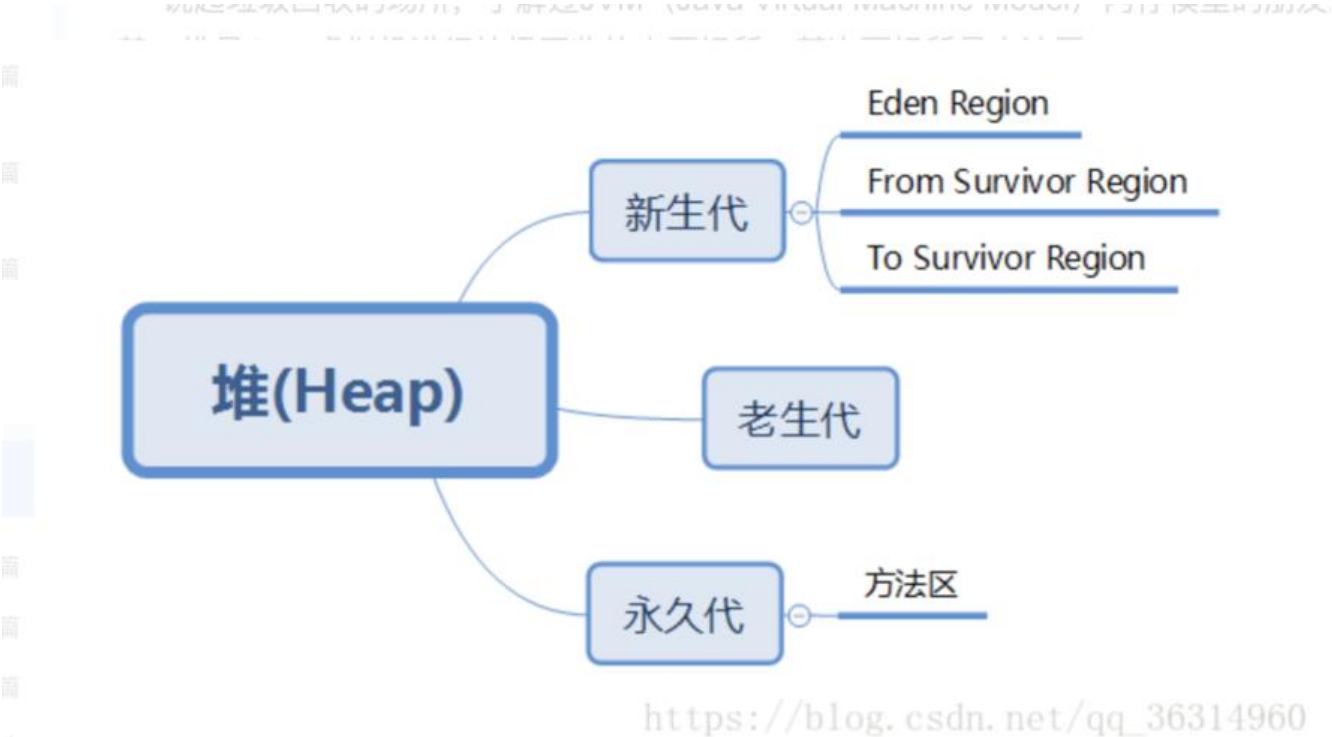
二进制数据，并将其转换成 Class 对象，存入缓冲区中。这就是为很么修改了 Class 后，必须重新启动 JVM，程序所做的修改才会生效的原因。

2.这里说明一下双亲委派机制：

双亲委派机制，其工作原理的是，如果一个类加载器收到了类加载请求，它并不会自己先去加载而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，即每儿子都很懒，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子自己才想办法去完成。

双亲委派机制的优势：采用双亲委派模式的是好处是 Java 类随着它的类加载器一起具备了一种有优先级的层次关系，通过这种层级关可以避免类的重复加载，当父亲已经加载了该类时，就没有必子 ClassLoader 再加载一次。其次是考虑到安全因素，Java 核心 API 中定义类型不会被随意替换，设通过网络传递一个名为 java.lang.Integer 的类，通过双亲委托模式传递到启动类加载器，而启动加载器在核心 Java API 发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的 java.lang.Integer，而直接返回已加载过的 Integer.class，这样便可以防止核心 API 库被随意篡改。

八、GC



我们都知道在 Java 虚拟机中进行垃圾回收的场所有两个，一个是堆，一个是方法区。在堆中存储了 Java 程序运行时的所有对象信息，而垃圾回收其实就是对那些“死亡的”对象进行其所侵占的内存的放，让后续对象再能分配到内存，从而完成程序运行的需要。关于何种对象为死亡对象，在下一部分做详细介绍。Java 虚拟机将堆内存进行了“分块处理”，从广义上讲，在堆中进行垃圾回收分为新生（Young Generation）和老生代（Old Generation）；从细微之处来看，为了提高 Java 虚拟机进垃圾回收的效率，又将新生代分成了三个独立的区域（这里的独立区域只是一个相对的概念，并不是分成三个区域以后就不再互相联合工作了），分别为：Eden 区（Eden Region）、From Survivor（Form Survivor Region）以及 To Survivor（To Survivor Region），而 Eden 区分配的内存较大其他两个区较小，每次使用 Eden 和其中一块 Survivor。Java 虚拟机在进行垃圾回收时，将 Eden 和urvivor 中还存活着的对象进行一次性地复制到另一块 Survivor 空间上，直到其两个区域中对象被回完成，当 Survivor 空间不够用时，需要依赖其他老年代的内存进行分配担保。当另外一块 Survivor 没有足够的空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老

代，在老生代中不仅存放着这一种类型的对象，还存放着大对象（需要很多连续的内存的对象），当 Java 程序运行时，如果遇到大对象将会被直接存放到老生代中，长期存活的对象也会直接进入老年代如果老生代的空间也被占满，当来自新生代的对象再次请求进入老生代时就会报 OutOfMemory 异。新生代中的垃圾回收频率高，且回收的速度也较快。就 GC 回收机制而言，JVM 内存模型中的方法更被人们倾向的称为永久代（Perm Generation），保存在永久代中的对象一般不会被回收。其永久进行垃圾回收的频率就较低，速度也较慢。永久代的垃圾收集主要回收废弃常量和无用类。以 String 常量 abc 为例，当我们声明了此常量，那么它就会被放到运行时常量池中，如果在常量池中没有任何象对 abc 进行引用，那么 abc 这个常量就算是废弃常量而被回收；判断一个类是否“无用”，则需时满足三个条件：

（1）、该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例；

（2）、加载该类的 ClassLoader 已经被回收

（3）、该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的是可以回收而不是必然回收。

大多数情况下，对象在新生代 Eden 区中分配，当 Eden 区没有足够空间进行分配时，虚拟机将发一次 Minor GC；同理，当老年代中没有足够的内存空间来存放对象时，虚拟机会发起一次 Major GC Full GC。只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小就会进行 Minor GC 否则将进行 Full CG。

虚拟机通过一个对象年龄计数器来判定哪些对象放在新生代，哪些对象应该放在老生代。如果对象 Eden 出生并经过一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 间中，并将该对象的年龄设为 1。对象每在 Survivor 中熬过一次 Minor GC，年龄就增加 1 岁，当的年龄增加到最大值 15 时，就将会被晋升到老年代中。虚拟机并不是永远地要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升到老年代，如果在 Survivor 空间中所有相同年龄的对象大小的总大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需等到 MaxTenringThreshold 中要求的年龄。

垃圾回收算法

1. 、引用计数算法（Reference Counting）

给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加 1；当引用失效时，数器值就减 1；任何时刻计数器为 0 的对象就是不可能再被使用的，这就是引用计数算法的核心。客来讲，引用计数算法实现简单，判定效率也很高，在大部分情况下都是一个不错的算法。**但是 Java 拟机并没有采用这个算法来判断何种对象为死亡对象，因为它很难解决对象之间相互循环引用的问题。**

```
public class ReferenceCountingGC{
  public Object object = null;

  private static final int OenM = 1024 * 1024;
  private byte[] bigSize = new byte[2 * OneM];

  public static void testCG(){
     ReferenceCountingGC objA = new ReferenceCountingGC();
      ReferenceCountingGC objB = new ReferenceCountingGC();

      objA.object = null;
      objB.object = null;
```

```
    System.gc();
}
}
```

在上述代码段中，objA 与 objB 互相循环引用，没有结束循环的判断条件，运行结果显示 Full GC 就说明当 Java 虚拟机并不是使用引用计数算法来判断对象是否存活的。

（2）、可达性分析算法（Reachability Analysis）

这是 Java 虚拟机采用的判定对象是否存活的算法。通过一系列的称为“GC Roots"的对象作起始点，从这些结点开始向下搜索，搜索所走过的路径称为饮用链（Reference Chain），当一个对到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。可作为 GC Roots 的对象包括：虚机栈中引用的对象、方法区中类静态属性引用的对象、方法区中常量引用的对象。本地方法栈 JNI 引的对象。

在上图可以看到 GC Roots 左边的对象都有引用链相关联，所以他们不是死亡对象，而在 GCRoots 边有几个零散的对象没有引用链相关联，所以他们就会别 Java 虚拟机判定为死亡对象而被回收。

3.标记-清除算法

该算法先标记，后清除，将所有需要回收的算法进行标记，然后清除；这种算法的缺点是：效率较低；标记清除后会出现大量不连续的内存碎片，这些碎片太多可能会使存储大对象会触发 GC 回收造成内存浪费以及时间的消耗。

4.复制算法

复制算法将可用的内存分成两份，每次使用其中一块，当这块回收之后把未回收的复制到另一块存中，然后把使用的清除。这种算法运行简单，解决了标记-清除算法的碎片问题，但是这种算法代过高，需要将可用内存缩小一半，对象存活率较高时，需要持续的复制工作，效率比较低。

5.标记整理算法

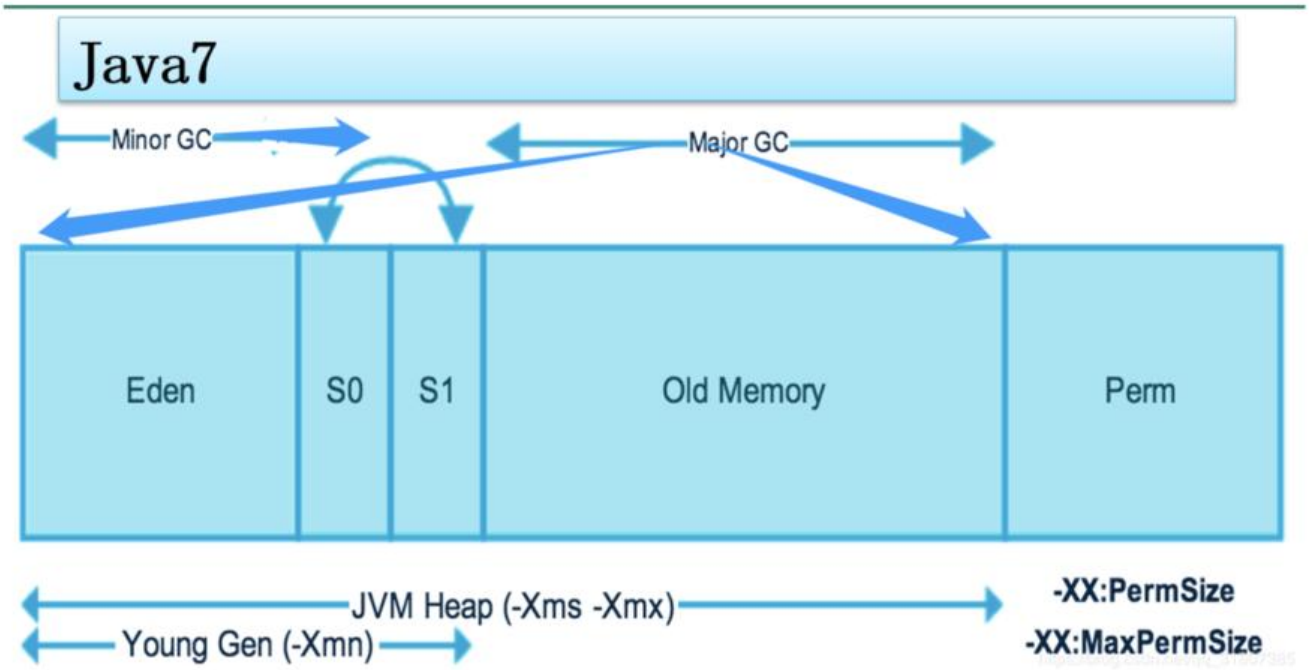
标记整理算法是针对复制算法在对象存活率较高时持续复制导致效率较低的缺点进行改进的，该法是在标记-清除算法基础上，不直接清理，而是使存活对象往一端游走，然后清除一端边界以外的存，这样既可以避免不连续空间出现，还可以避免对象存活率较高时的持续复制。这种算法适合老生。

6.分代收集算法

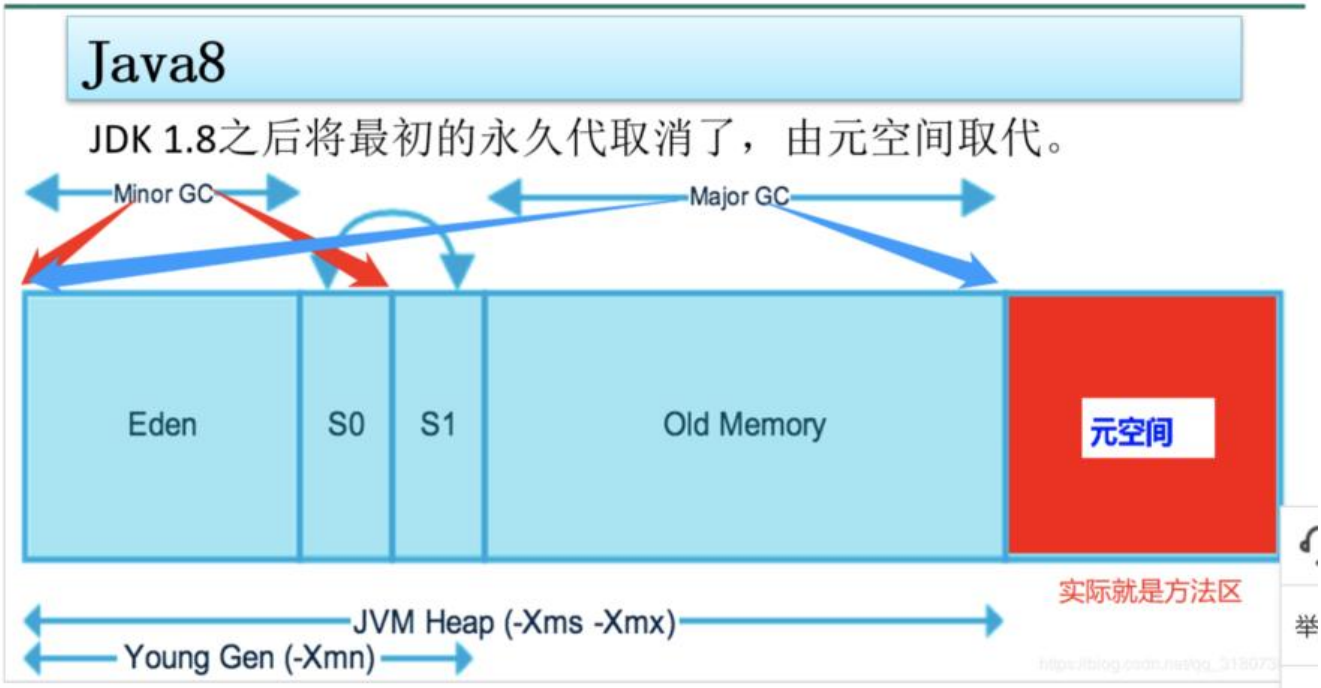
分代收集算法就是目前虚拟机使用的回收算法，它解决了标记整理不适用于老年代的问题，将存分为各个年代，在不同年代使用不同的算法，从而使用最合适的算法，新生代存活率低，可以使用制算法。而老年代对象存活率高，没有额外空间对它进行分配担保，所以使用标记整理算法。

10、jdk1.7与jdk1.8的区别

jdk1.7:



jdk1.8:



原文链接：JVM 相关工作原理