jre、jdk、jvm之间的关系

JDK是Java程序员常用的开发包、目的就是用来编译和调试Java程序的。

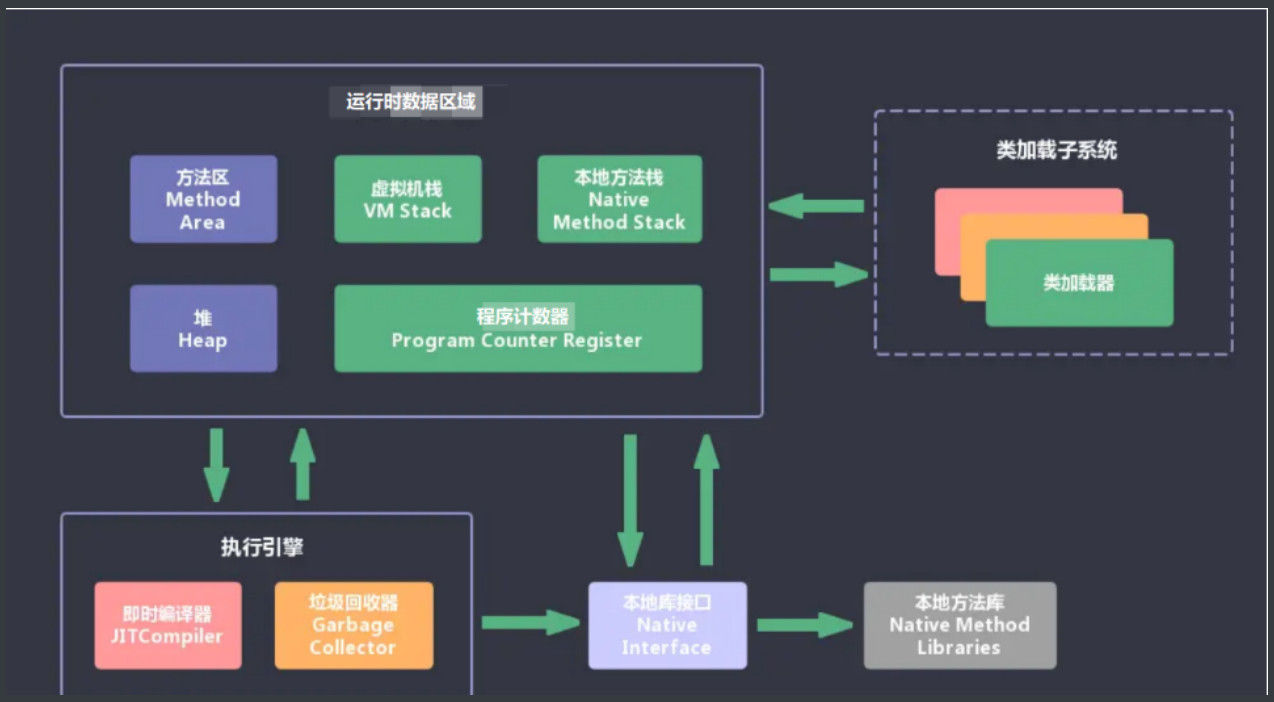
JRE是指Java运行环境，也就是我们的写好的程序必须在JRE才能够运行。

JVM是Java Virtual Machine（Java虚拟机）的缩写，是指负责将字节码解释成为特定的机器码进行运行，值得注意的是在运行过程中，Java源程序需要通过编译器编译为.class文件，否则JVM不认识。

jvm组成结构

我们先来研究一下jvm的组成结构都有哪些

jvm结构图



类加载子系统

负责从文件系统或是网络中加载class信息，加载的信息存放在一个称之为 方法区 的内存空间

方法区

用于存放类的信息、常量信息、常量池信息、包括字符串字面量和数字常量。我们常用的反射就是从这个方法区里读取的类信息

Java堆

堆空间是jvm启动的时候创建的一块内存区域，几乎所有的对象实例都放在这个空间里（可以理解成new出来的那些对象）。

这个区域被划分为新生代和老年的，之后重点讲解，我们常说的GC垃圾回收机制，就是主要回收堆空间的垃圾数据。

堆空间里的数据，是被所有线程共享的，所以会存在线程安全的问题。所以那些锁就是为了解决堆空间数据线程安全的问题而生的。

直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但这部分也是被频繁的读写使用，也可能会导致 `OutOfMemoryError` 异常的出现。

Java的 `NIO` 中的allocateDirect方法是可以直接使用直接内存的，能显著的提高读写的速度。

Java栈

就是我们常说的堆栈两兄弟之一的栈，所有线程共享堆空间里的数据，但是栈空间是每个线程独有的，互相直接不能访问。

栈空间是线程创建的时候所创建的一份内存空间，栈里主要保存一些局部变量、方法参数、Java方法调用，返回值等信息。

本地方法栈

本地方法栈和Java栈不同之处在于，可以直接调用Java本地方法，即JDK中用native修饰的方法。

垃圾收集系统

GC垃圾回收，是一个非常重要的知识点，保证我们程序能够有足够的内存空间运行，回收掉内存中已经无效的数据，大家就可以理解成我们日常中活中的垃圾回收。

回收算法一般有 `标记清除` 算法、 `复制` 算法、 `标记整理` 算法等等，之后的文章，我们会详解讲解每一种算法。

PC寄存器

它是每个线程私有的空间，JVM会为每个线程创建单独的PC寄存器，在任意时刻，一个Java线程总是在执行一个方法，这个方法被称为当前方法，如果当前方法不是本地方法，PC寄存器会执行当前正在被执行的指令，如果是本地方法，则PC寄存器值为undefined，寄存器存放如当前环境指针、程序计数器、操作栈指针、计算的变量指针等信息。

执行引擎

是jvm非常核心的组件，它负责执行jvm的字节码，一般先会编译成机器码后执行。

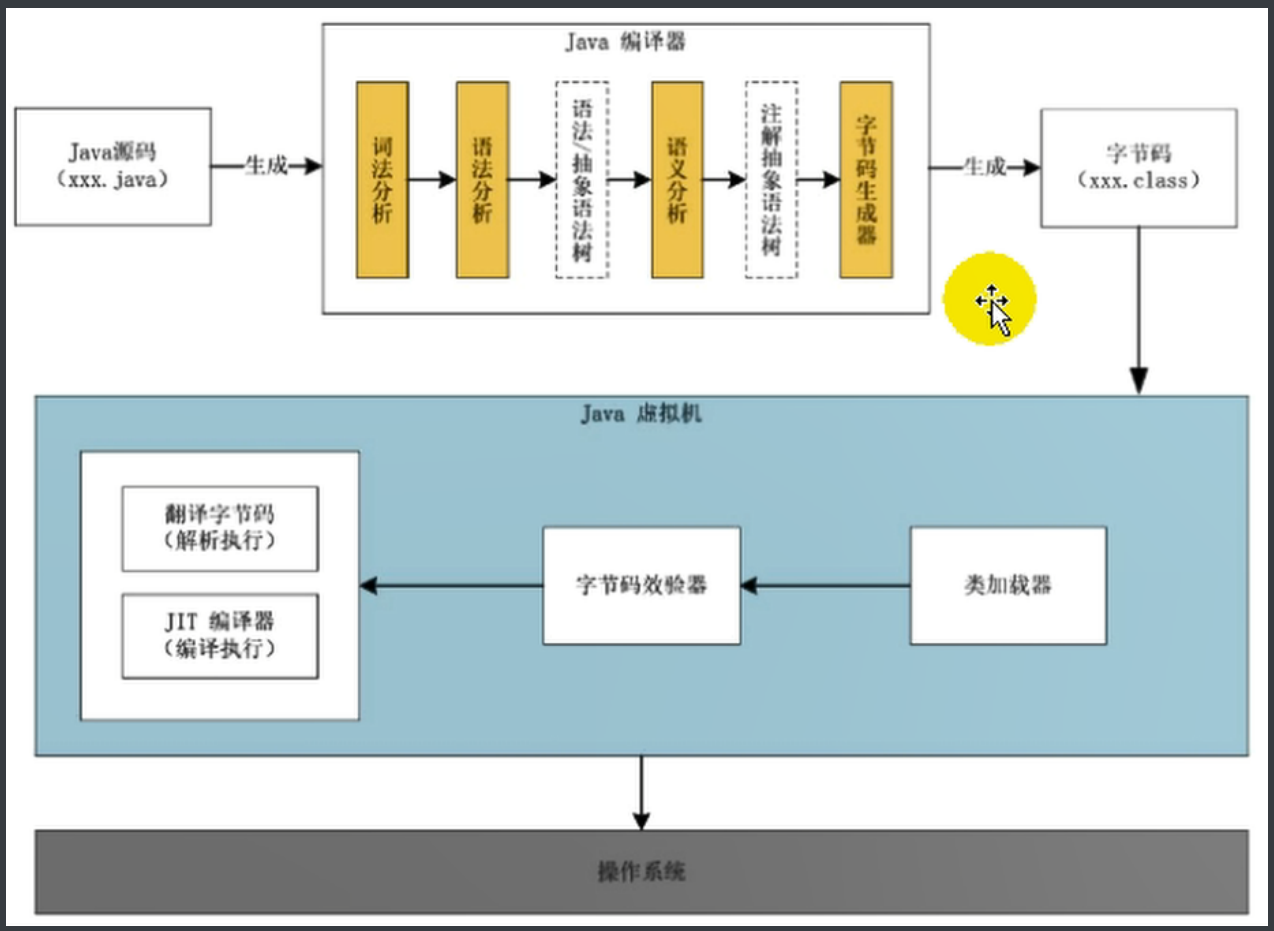
类加载机制

`jvm` 的启动是通过引导类加载器（bootstrap class loader）创建一个初始类（initial class）来完成的，这个类是由 `jvm` 的具体实现指定的。[来自官方规范]

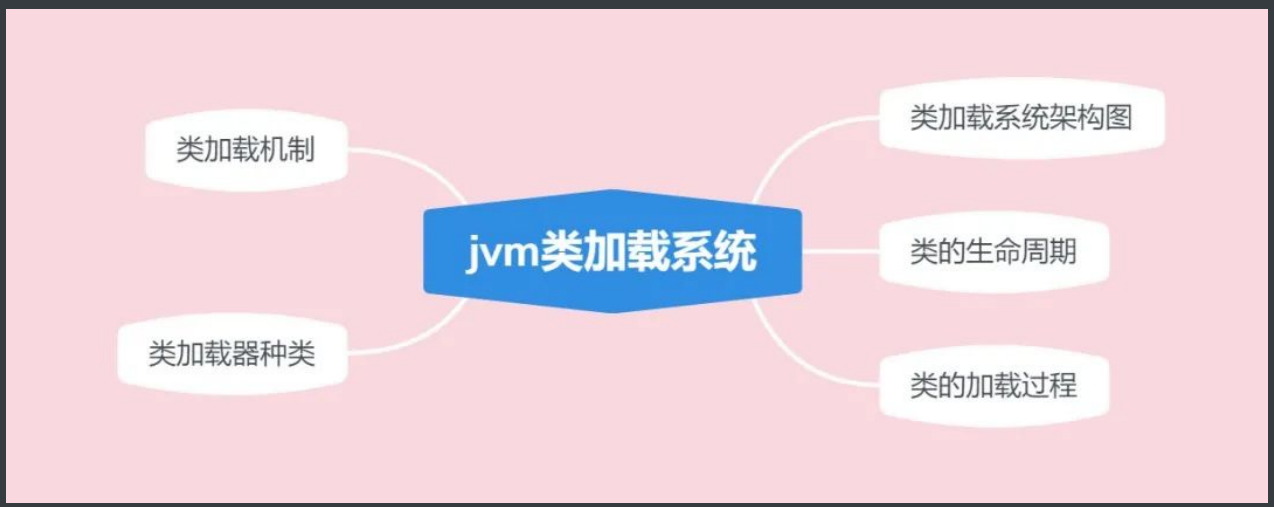
jvm组成结构之一就是 `类装载器子系统` ，我们今天就来仔细讲讲这个组件。

Java代码执行流程图

大家通过这个流程图，了解一下我们写好的Java代码是如何执行的，其中要经历 `类加载器` 这个流程，我们就来仔细讲讲这里面的知识点。

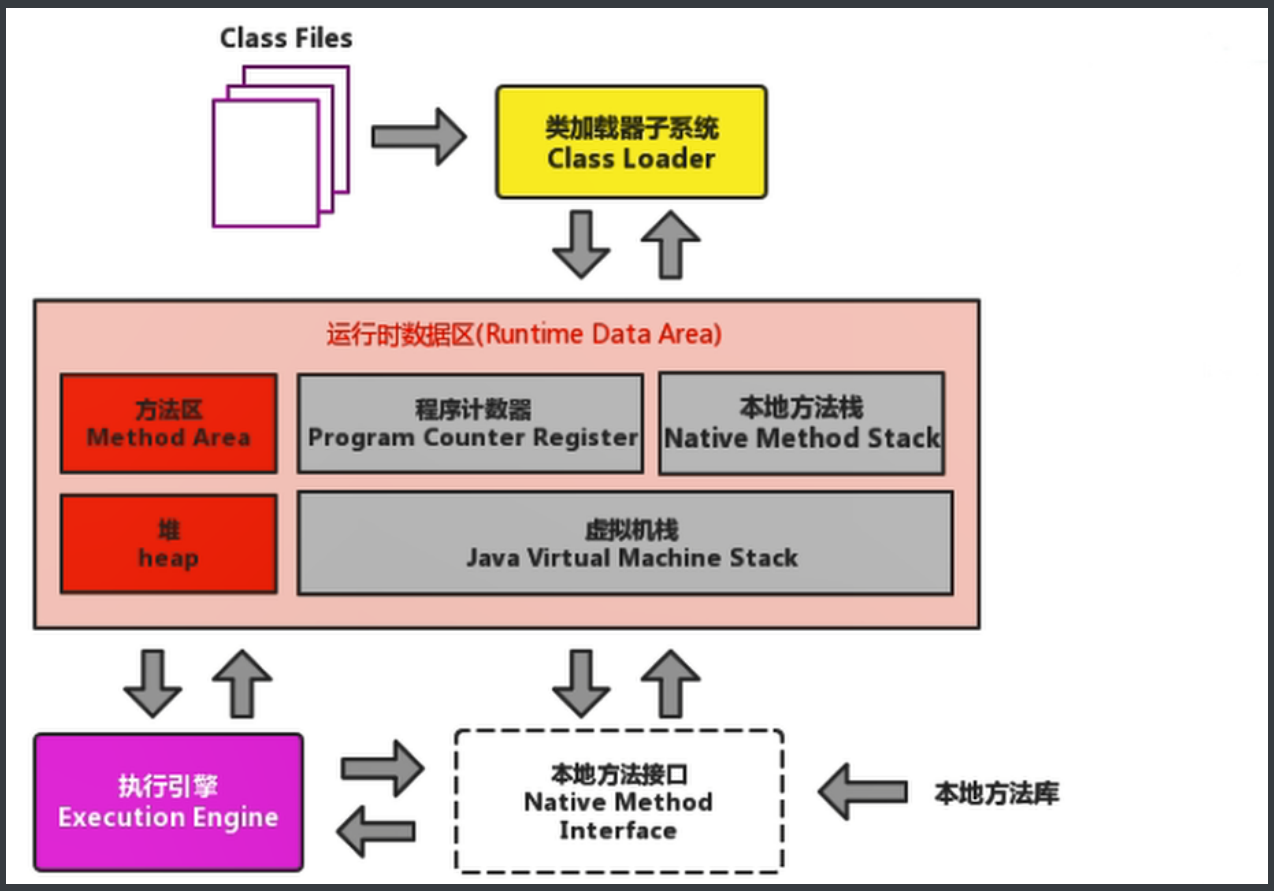


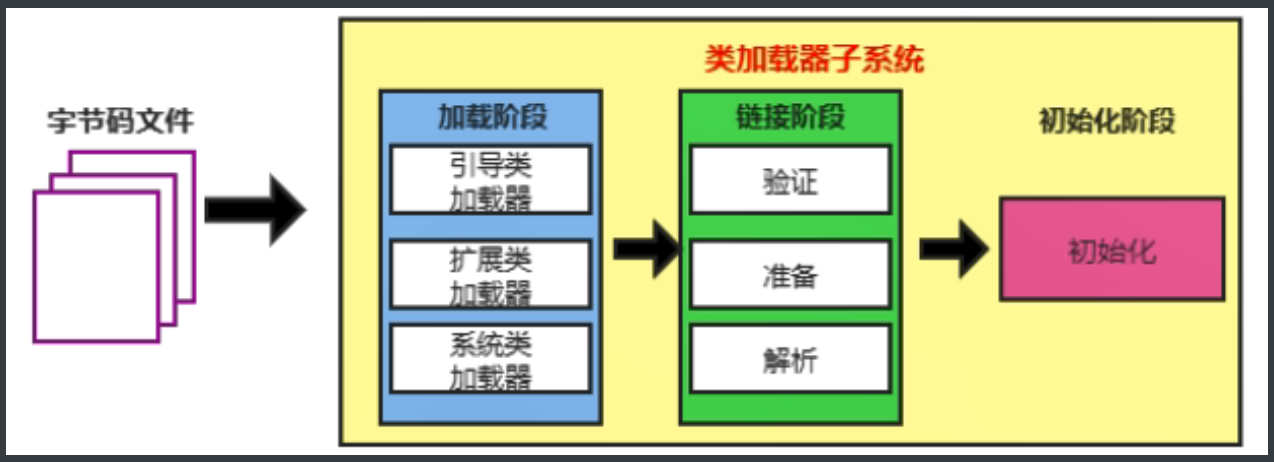
类加载子系统



类加载系统架构图

暂时看不懂这两张图没关系，跟着 `老哥` 往下看

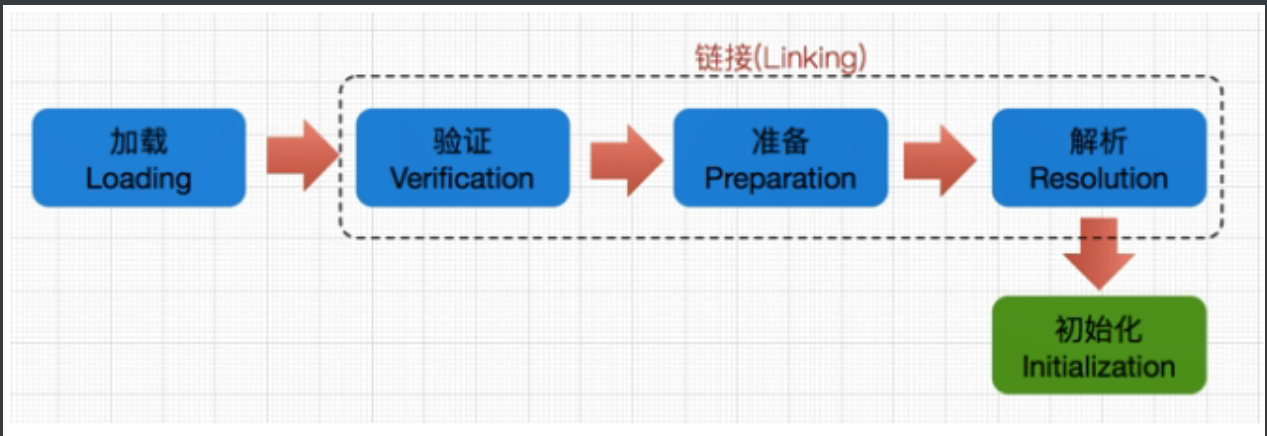




类的生命周期

类的生命周期包括：加载、链接、初始化、使用和卸载，其中 `加载` 、 `链接` 、 `初始化` ，属于 `类加载的过程` ，我们下面仔细讲解。使用是指我们new对象进行使用，卸载指对象被垃圾回收掉了。

类加载的过程



- **第一步：Loading加载**

> 通过类的全限定名（包名 + 类名），获取到该类的 `.class` 文件的二进制字节流

> 将二进制字节流所代表的静态存储结构，转化为方法区运行时的数据结构

> 在 `内存` 中生成一个代表该类的 `java.lang.Class` 对象，作为方法区这个类的各种数据的访问入口

总结：`加载二进制数据到内存` —> `映射成jvm能识别的结构` —> `在内存中生成class文件` 。

- **第二步：Linking链接**

链接是指将上面创建好的class类合并至Java虚拟机中，使之能够执行的过程，可分为 `验证` 、 `准备` 、 `解析` 三个阶段。

① 验证（Verify）

> 确保class文件中的字节流包含的信息，符合当前虚拟机的要求，保证这个被加载的class类的正确性，不会危害到虚拟机的安全。

② 准备（Prepare）

> 为类中的 `静态字段` 分配内存，并设置默认的初始值，比如int类型初始值是0。被final修饰的static字段不会设置，因为final在编译的时候就分配了

③ 解析（Resolve）

> 解析阶段的目的，是将常量池内的符号引用转换为直接引用的过程（将常量池内的符号引用解析成为实际引用）。如果符号引用指向一个未被加载的类，或者未被加载类的字段或方法，那么解析将触发这个类的加载（但未必触发这个类的链接以及初始化。）

> 事实上，解析器操作往往会伴随着 JVM 在执行完初始化之后再执行。 符号引用就是一组符号来描述所引用的目标。符号引用的字面量形式明确定义在《Java 虚拟机规范》的Class文件格式中。直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

> 解析动作主要针对类、接口、字段、类方法、接口方法、方法类型等。对应常量池中的CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info等。

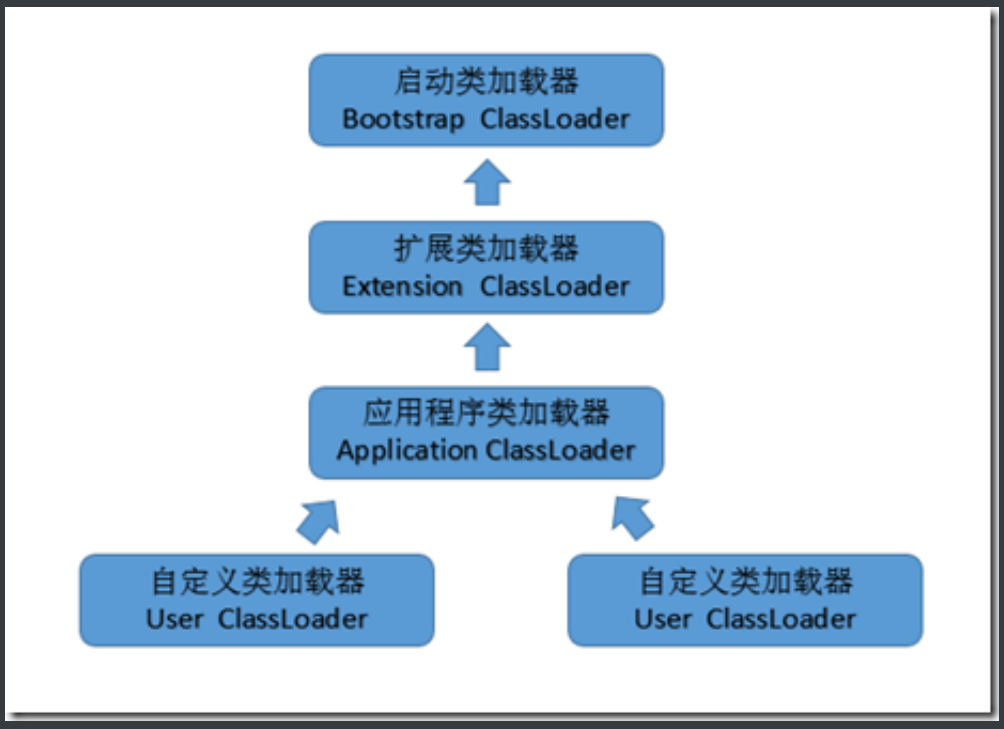
- **第三步：initialization初始化**

> 初始化就是执行类的构造器方法 `init ()`的过程。

> 这个方法不需要定义，是javac编译器自动收集类中所有类变量的赋值动作和静态代码块中的语句合并来的。

> 若该类具有父类， `jvm` 会保证父类的 `init` 先执行，然后在执行子类的 `init` 。

类加载器的分类



- **第一个：启动类/引导类：Bootstrap ClassLoader**

> 这个类加载器使用C/C++语言实现的，嵌套在JVM内部，java程序无法直接操作这个类。

> 它用来加载Java核心类库，
> 如：`JAVA_HOME/jre/lib/rt.jar`、`resources.jar`、`sun.boot.class.path`路径下的包，
> 用于提供jvm运行所需的包。

> 并不是继承自java.lang.ClassLoader，它没有父类加载器

> 它加载`扩展类加载器`和`应用程序类加载器`，并成为他们的父类加载器

> 出于安全考虑，启动类只加载包名为：java、javax、sun开头的类

- **第二个：扩展类加载器：Extension ClassLoader**

> Java语言编写，由`sun.misc.Launcher$ExtClassLoader`实现，我们可以用Java程序操作这个加载器

> 派生继承自java.lang.ClassLoader，父类加载器为`启动类加载器`

> 从系统属性：`java.ext.dirs`目录中加载类库，或者从JDK安装目录：`jre/lib/ext`目录下加载类库。我们就可以将我们自己的包放在以上目录下，就会自动加载进来了。

- **第三个：应用程序类加载器：Application Classloader**

> Java语言编写，由`sun.misc.Launcher$AppClassLoader`实现。

> 派生继承自java.lang.ClassLoader，父类加载器为`启动类加载器`

> 它负责加载`环境变量classpath`或者`系统属性java.class.path`指定路径下的类库

> 它是程序中默认的类加载器，我们Java程序中的类，都是由它加载完成的。

> 我们可以通过`ClassLoader#getSystemClassLoader()`获取并操作这个加载器

- **第四个：自定义加载器**

> 一般情况下，以上3种加载器能满足我们日常的开发工作，不满足时，我们还可以 `自定义加载器`

> 比如用网络加载Java类，为了保证传输中的安全性，采用了加密操作，那么以上3种加载器就无法加载这个类，这时候就需要 `自定义加载器`

自定义加载器实现步骤

> 继承 `java.lang.ClassLoader` 类，重写findClass()方法

> 如果没有太复杂的需求，可以直接继承 `URLClassLoader` 类，重写 `loadClass` 方法，具体可参考 `AppClassLoader` 和 `ExtClassLoader` 。

获取ClassLoader几种方式

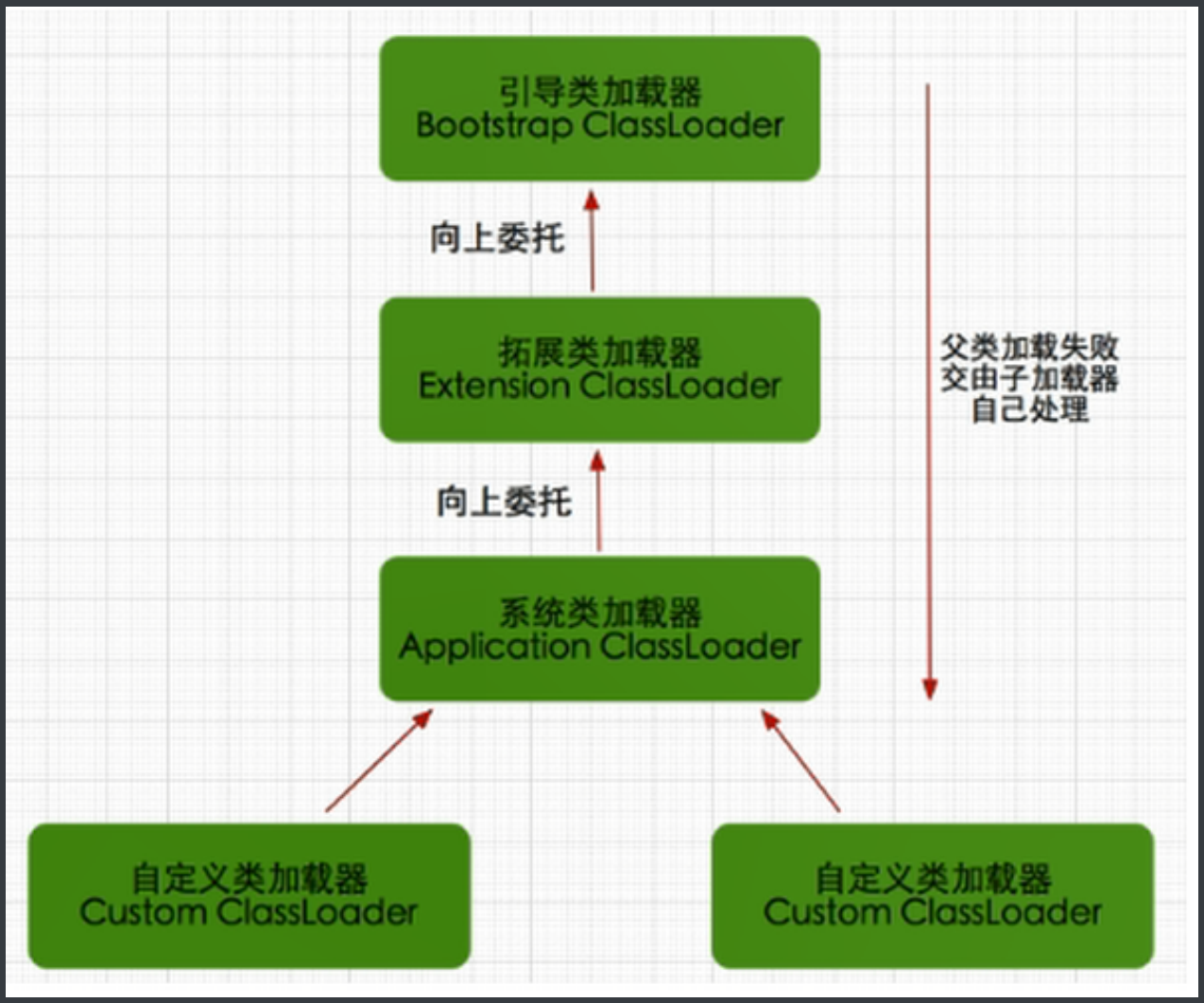
它是一个抽象类，其后所有的类加载器继承自 ClassLoader（不包括启动类加载器）

```
// 方式一：获取当前类的 ClassLoader
clazz.getClassLoader()

// 方式二：获取当前线程上下文的 ClassLoader
Thread.currentThread().getContextClassLoader()

// 方式三：获取系统的 ClassLoader
ClassLoader.getSystemClassLoader()

// 方式四：获取调用者的 ClassLoader
DriverManager.getCallerClassLoader()
```

类加载机制—双亲委派机制

jvm对class文件采用的是按需加载的方式，当需要使用该类时，jvm才会将它的class文件加载到内存中产生class对象。

在加载类的时候，是采用的 `双亲委派机制` ，即把请求交给父类处理的一种任务委派模式。



- **工作原理**

（1）如果一个 `类加载器` 接收到了 `类加载` 的请求，它自己不会先去加载，会把这个请求委托给 `父类加载器` 去执行。

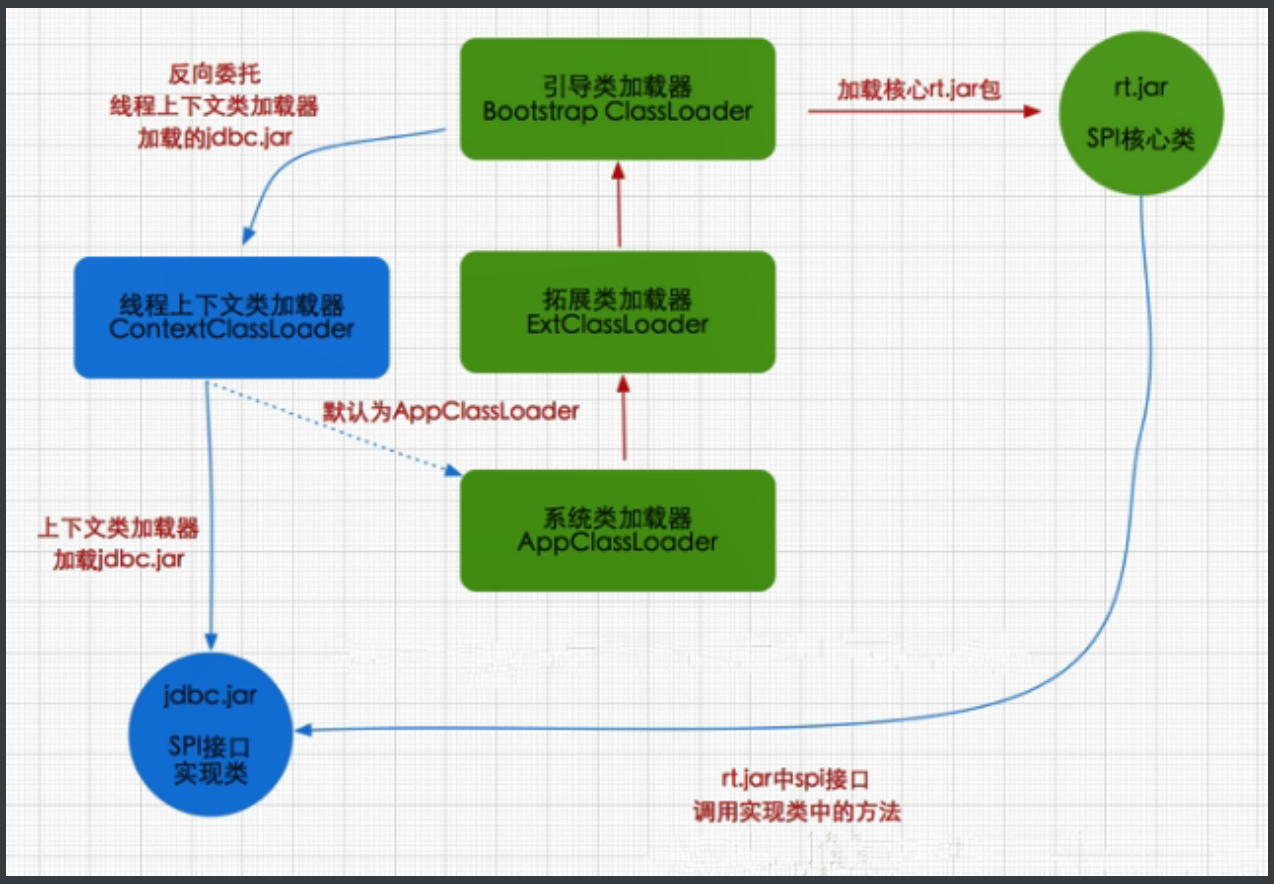
（2）如果父类还存在父类加载器，则继续向上委托，一直委托到 `启动类加载器：Bootstrap ClassLoader`

（3）如果父类加载器可以完成加载任务，就返回成功结果，如果父类加载失败，就由子类自己去尝试加载，如果子类加载失败就会抛出 `ClassNotFoundException` 异常，这就是 `双亲委派模式`

- **第三方包加载方式：反向委派机制**

在Java应用中存在着很多服务提供者接口（Service Provider Interface，SPI），这些接口允许第三方为它们提供实现，如常见的 SPI 有 JDBC、JNDI等，这些 SPI 的接口属于 Java 核心库，一般存在rt.jar包中，由Bootstrap类加载器加载。而Bootstrap类加载器无法直接加载SPI的实现类，同时由于双亲委派模式的存在，Bootstrap类加载器也无法反向委托AppClassLoader加载器SPI的实现类。在这种情况下，我们就需要一种特殊的类加载器来加载第三方的类库，而线程上下文类加载器（双亲委派模型的破坏者）就是很好的选择。

从图可知rt.jar核心包是有Bootstrap类加载器加载的，其内包含SPI核心接口类，由于SPI中的类经常需要调用外部实现类的方法，而jdbc.jar包含外部实现类(jdbc.jar存在于classpath路径)无法通过Bootstrap类加载器加载，因此只能委派线程上下文类加载器把jdbc.jar中的实现类加载到内存以便SPI相关类使用。显然这种线程上下文类加载器的加载方式破坏了“双亲委派模型”，它在执行过程中抛弃双亲委派加载链模式，使程序可以逆向使用类加载器，当然这也使得Java类加载器变得更加灵活。



- **沙箱安全机制**

自定义 String 类，但是在加载自定义 String 类的时候会率先使用引导类加载器加载，而引导类加载器在加载的过程中会先加载 JDK 自带的文件（rt.jar 包中的 java\lang\String.class），报错信息说没有 main 方法就是因为加载的 rt.jar 包中的 String 类。这样可以保证对 Java 核心源代码的保护，这就是沙箱安全机制。

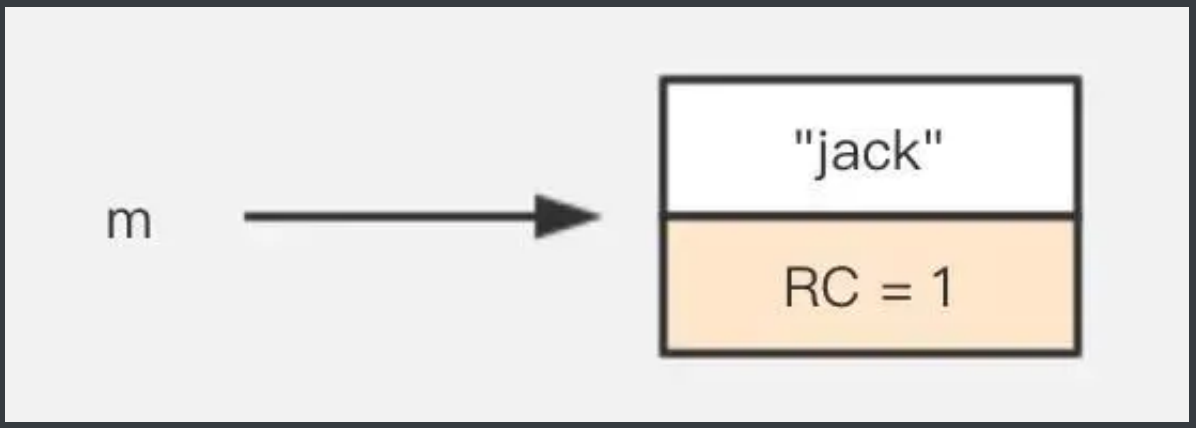
什么是垃圾回收

垃圾回收（Garbage Collection，GC），顾名思义就是释放垃圾占用的空间，防止内存泄露。有效的使用可以使用的内存，对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收。

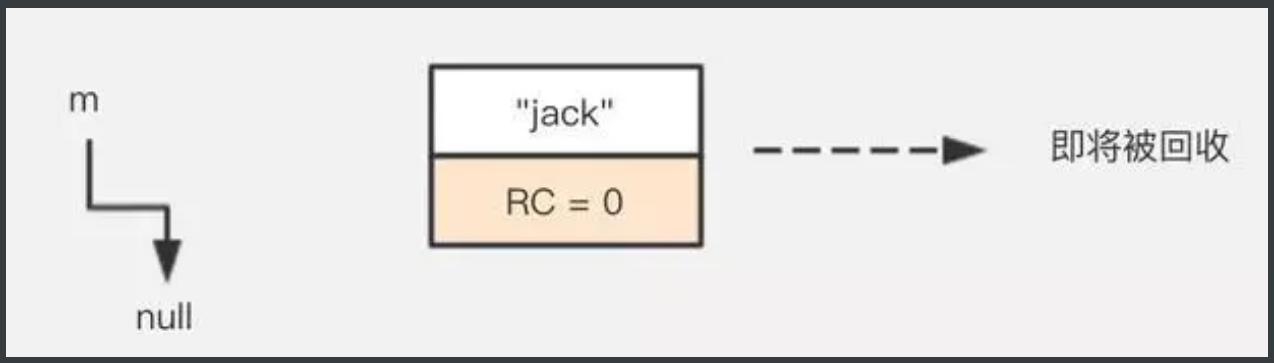
如何找到程序里的垃圾

引用计数法

给每个对象添加一个计数器 RC ，当有地方引用该对象时计数器加1，当引用失效时计数器减1。用对象计数器是否为0来判断对象是否可被回收。 缺点 ： 无法解决循环引用的问题。



先创建一个字符串，String m = new String("jack");，这时候 "jack" 有一个引用，就是m。然后将m设置为null，这时候 "jack" 的引用次数就等于 0 了，在引用计数算法中，意味着这块内容就需要被回收了。



引用计数算法是将垃圾回收分摊到整个应用程序的运行当中了，而不是在进行垃圾收集时，要挂起整个应用的运行，直到对堆中所有对象的处理都结束。因此，采用引用计数的垃圾收集不属于严格意义上的Stop-The-World的垃圾收集机制。

看似很美好，但我们知道JVM的垃圾回收就是Stop-The-World的，那是什么原因导致我们 `最终放弃` 了 `引用计数算法` 呢？看下面的例子。

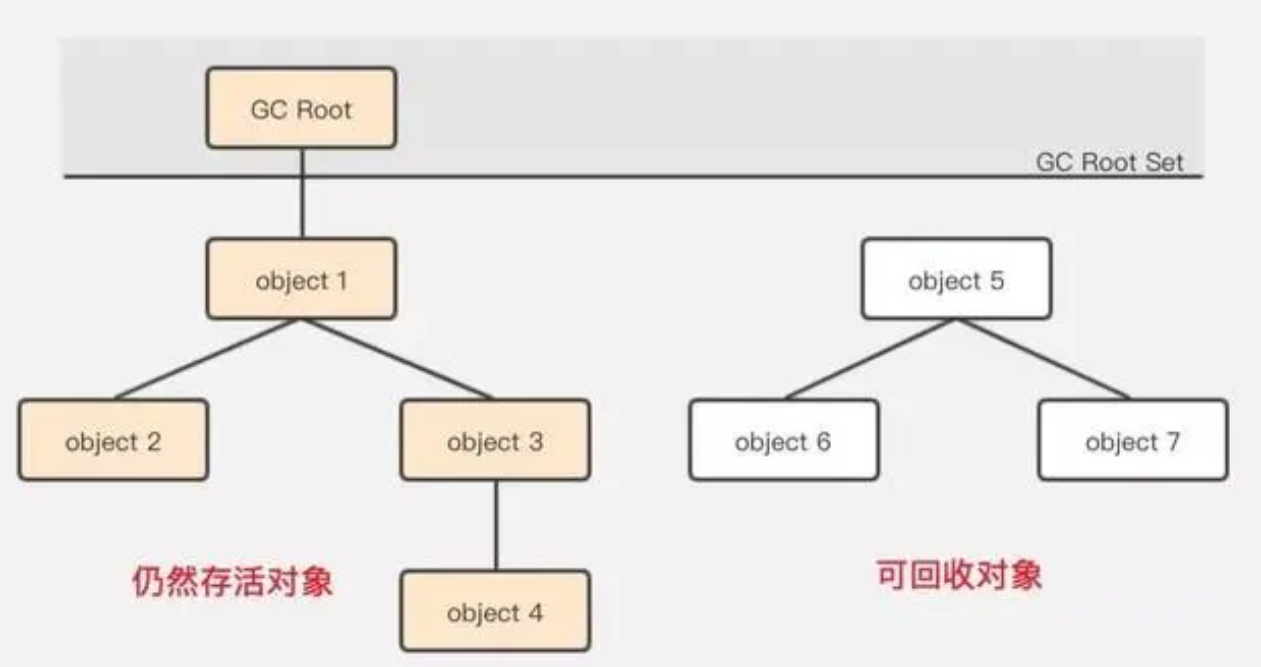
```
1  public class ReferenceCountingGC {
2
3    public Object instance;
4
5    public ReferenceCountingGC(String name) {
6    }
7
8    public static void testGC(){
9
10     ReferenceCountingGC a = new ReferenceCountingGC("objA");
11     ReferenceCountingGC b = new ReferenceCountingGC("objB");
12
13     // a和b互相引用了
14     a.instance = b;
15     b.instance = a;
```

```
16
17      a = null;
18      b = null;
19    }
20  }
```

我们可以看到，最后这2个对象已经不可能再被访问了，但由于他们 相互引用 着对方，导致它们的引用计数永远都不会为0，通过引用计数算法，也就 永远无法通知 GC收集器回收它们。

可达性分析算法

通过 GC ROOT 的对象作为搜索起始点，通过引用向下搜索，所走过的路径称为引用链。通过对象是否有到达引用链的路径来判断对象是否可被回收（可作为GC ROOT的对象：虚拟机栈中引用的对象，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，本地方法栈中JNI引用的对象）



通过可达性算法，成功解决了引用计数所无法解决的循环依赖问题，只要你无法与 GC Root 建立直接或间接的连接，系统就会判定你为可回收对象。那这样就引申出了另一个问题，哪些属于 GC Root 。

Java内存区域中可以作为GC ROOT的对象：

虚拟机栈中引用的对象

```
public class StackLocalParameter {

  public StackLocalParameter(String name) {}

  public static void testGC() {
    StackLocalParameter s = new StackLocalParameter("localParameter");
    s = null;
  }
}
```

> 此时的s，即为GC Root，当s置空时，localParameter对象也断掉了与GC Root的引用链，将被回收。

方法区中类静态属性引用的对象

```
public class MethodAreaStaicProperties {

  public static MethodAreaStaicProperties m;

  public MethodAreaStaicProperties(String name) {}

  public static void testGC(){
    MethodAreaStaicProperties s = new
MethodAreaStaicProperties("properties");
    s.m = new MethodAreaStaicProperties("parameter");
    s = null;
  }
}
```

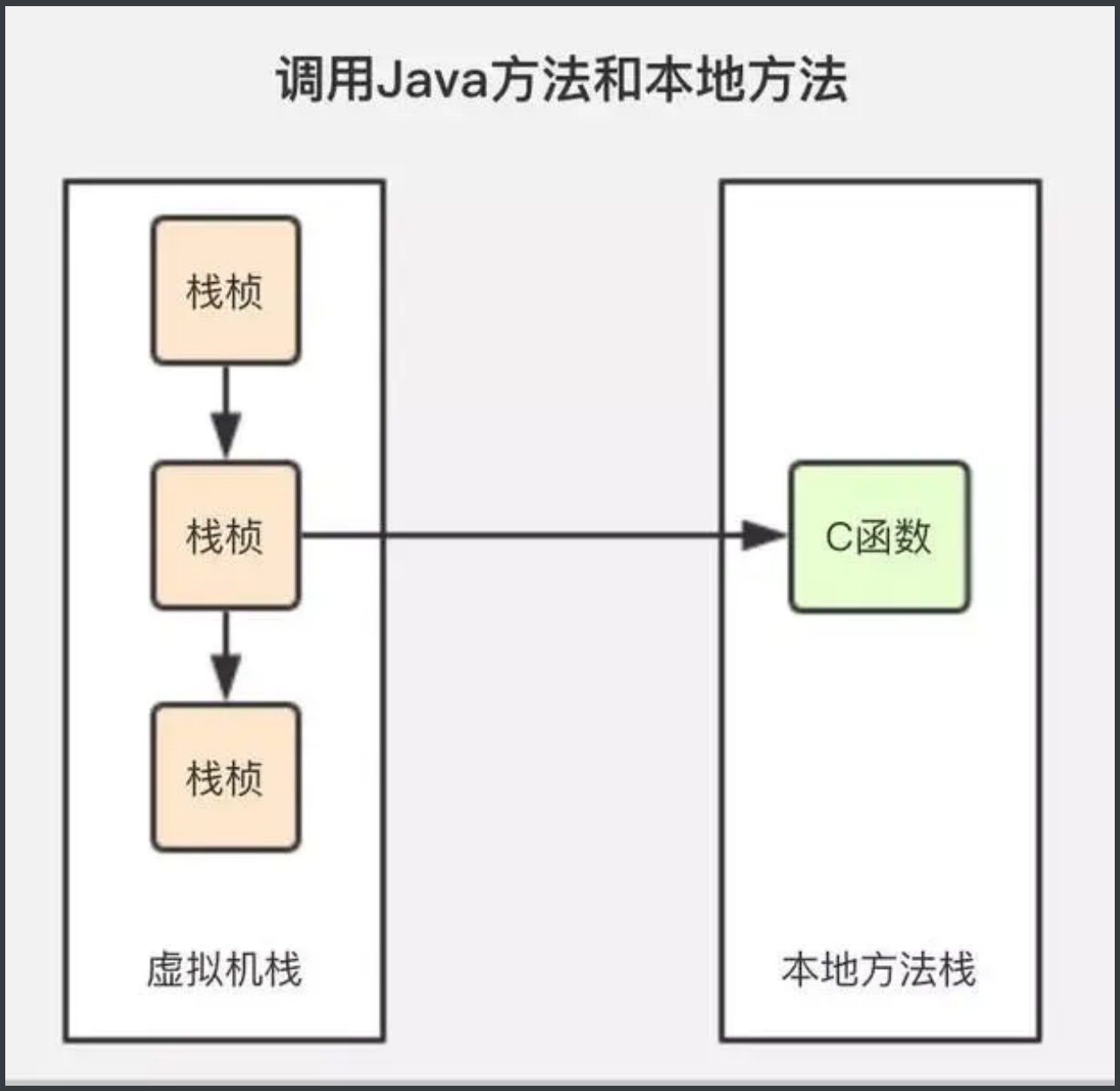
> 此时的s，即为GC Root，s置为null，经过GC后，s所指向的properties对象由于无法与GC Root建立关系被回收。而m作为类的静态属性，也属于GC Root，parameter 对象依然与GC root建立着连接，所以此时parameter对象并不会被回收。

方法区中常量引用的对象

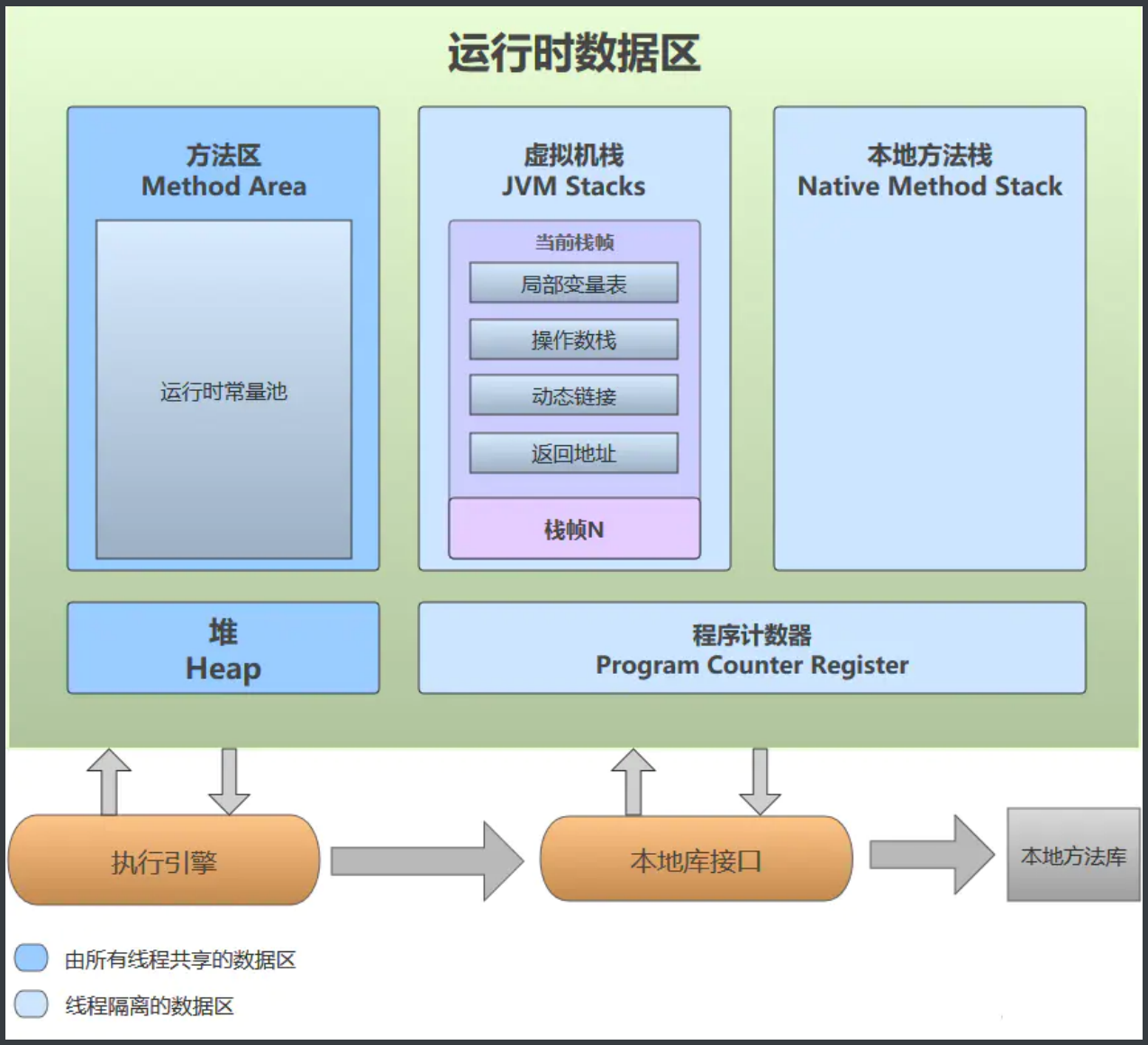
```java
public class MethodAreaStaicProperties {

  public static final MethodAreaStaicProperties m =
MethodAreaStaicProperties("final");

  public MethodAreaStaicProperties(String name) {}

  public static void testGC() {
    MethodAreaStaicProperties s = new
MethodAreaStaicProperties("staticProperties");
    s = null;
  }
}
```

> m即为方法区中的常量引用，也为GC Root，s置为null后，final对象也不会因没有与GC Root建立联系而被回收。

本地方法栈中引用的对象



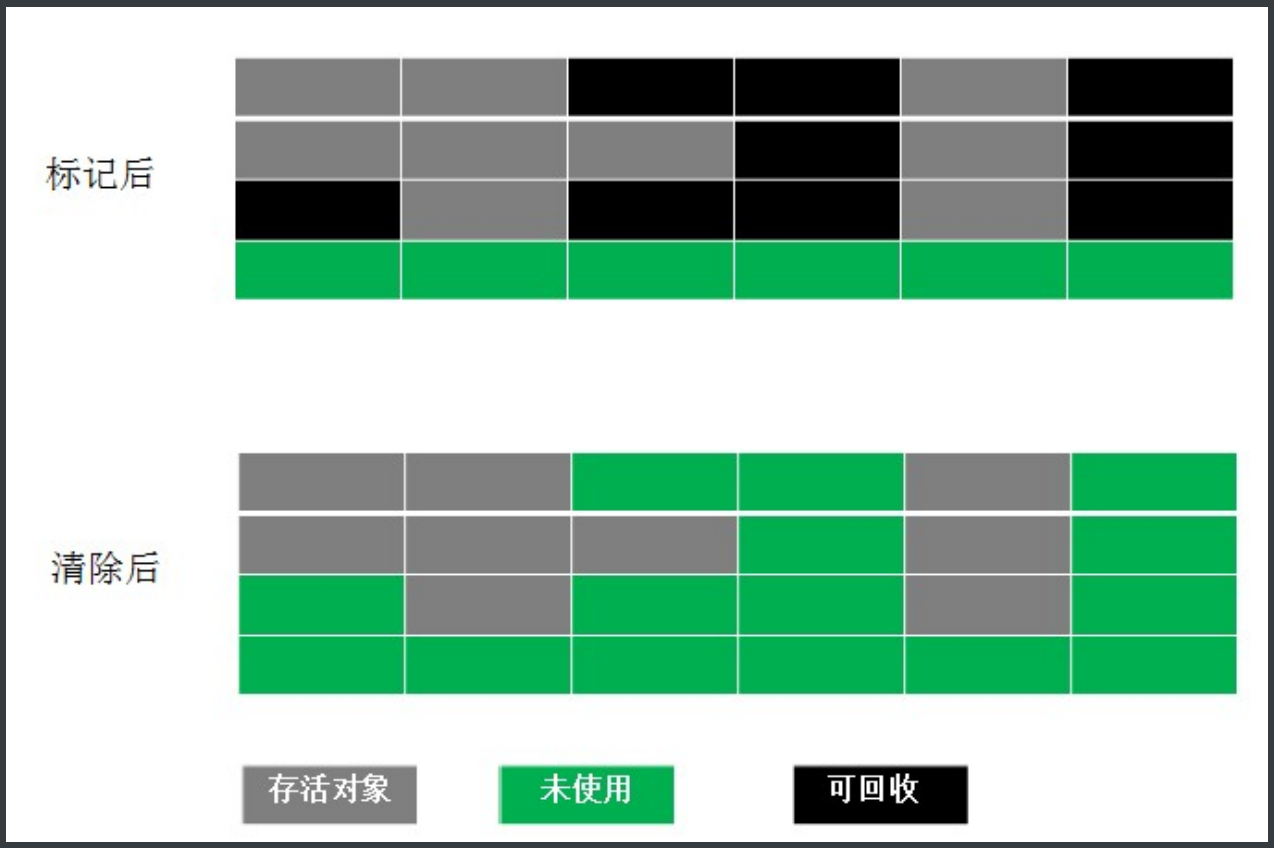
> 任何native接口都会使用某种本地方法栈，实现的本地方法接口是使用C连接模型的话，那么它的本地方法栈就是C栈。当线程调用Java方法时，虚拟机会创建一个新的栈帧并压入Java栈。然而当它调用的是本地方法时，虚拟机会保持Java栈不变，不再在线程的Java栈中压入新的帧，虚拟机只是简单地动态连接并直接调用指定的本地方法。



垃圾回收算法

在确定了哪些垃圾可以被回收后，垃圾收集器要做的事情就是开始进行垃圾回收，但是这里面涉及到一个问题是：如何高效地进行垃圾回收。这里我们讨论几种常见的垃圾收集算法的核心思想。

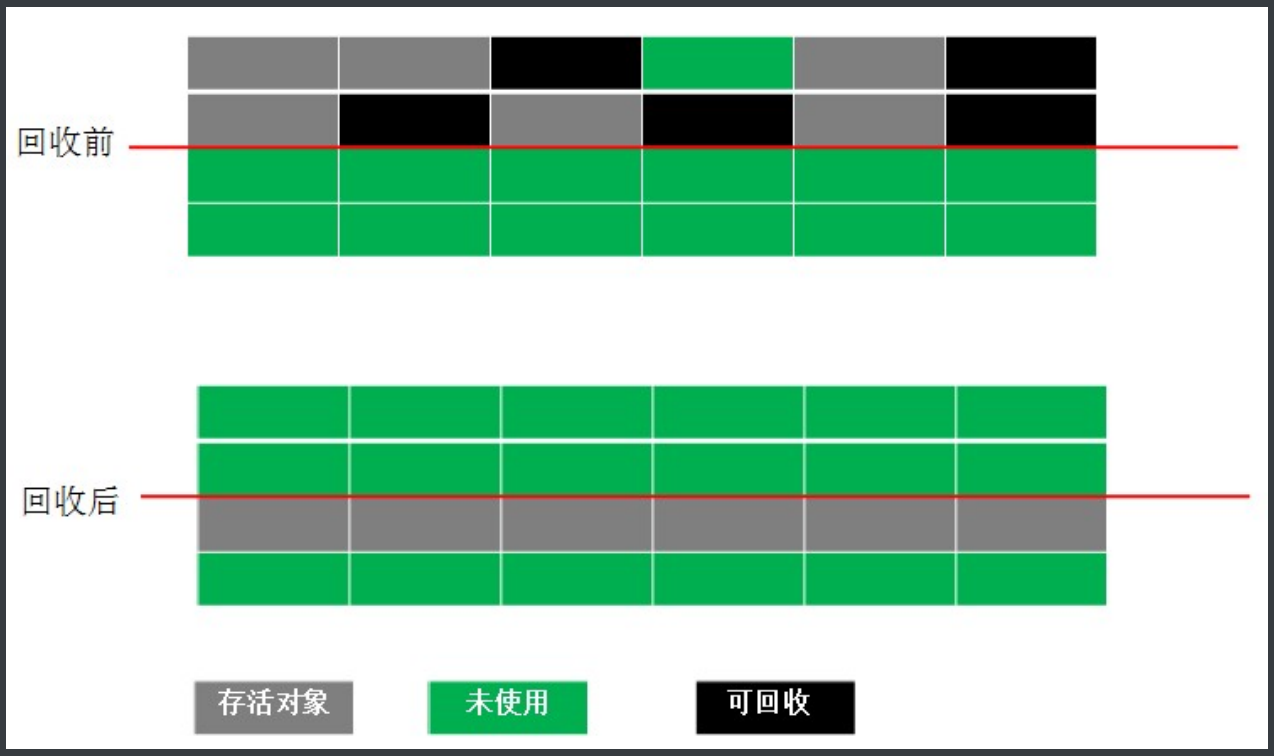
标记-清除算法



标记清除算法（Mark-Sweep）是最基础的一种垃圾回收算法，它分为2部分，先把内存区域中的这些对象进行标记，哪些属于可回收标记出来，然后把这些垃圾拎出来清理掉。就像上图一样，清理掉的垃圾就变成未使用的内存区域，等待被再次使用。但它存在一个很大的问题，那就是内存碎片。

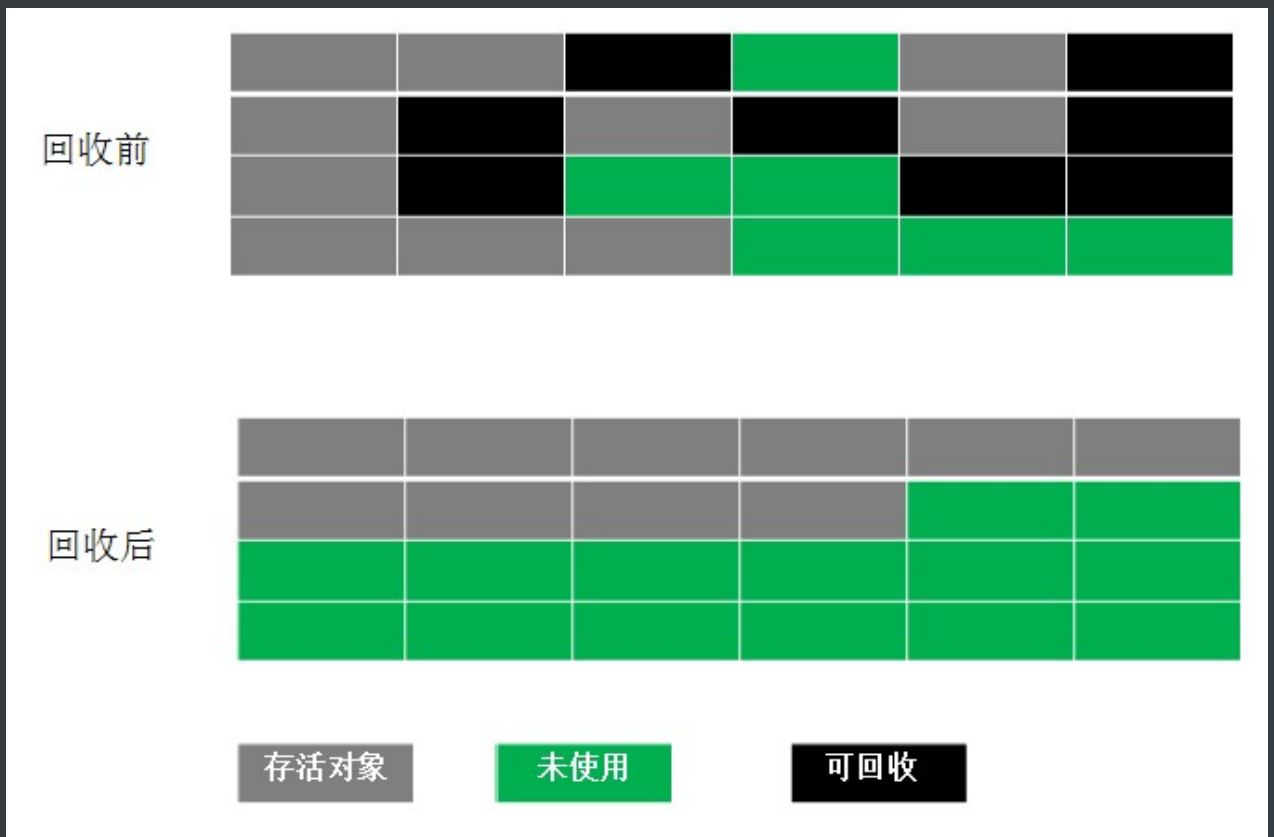
上图中等方块的假设是2M，小一些的是1M，大一些的是4M。等我们回收完，内存就会切成了很多段。我们知道开辟内存空间时，需要的是连续的内存区域，这时候我们需要一个2M的内存区域，其中有2个1M是没法用的。这样就导致，其实我们本身还有这么多的内存的，但却用不了。

复制算法



复制算法（Copying）是在标记清除算法基础上演化而来，解决标记清除算法的内存碎片问题。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。保证了内存的连续可用，内存分配时也就不用考虑内存碎片等复杂情况。复制算法暴露了另一个问题，例如硬盘本来有500G，但却只能用200G，代价实在太高。

标记-压缩算法



标记-压缩算法标记过程仍然与标记-清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，再清理掉端边界以外的内存区域。

标记压缩算法解决了内存碎片的问题，也规避了复制算法只能利用一半内存区域的弊端。标记压缩算法对内存变动更频繁，需要整理所有存活对象的引用地址，在效率上比复制算法要差很多。一般是把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。

分代收集算法

分代收集算法分代收集算法严格来说并不是一种思想或理论，而是融合上述3种基础的算法思想，而产生的针对不同情况所采用不同算法的一套组合拳，根据对象存活周期的不同将内存划分为几块。

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。

在老年代中，因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记-清理算法或者标记-整理算法来进行回收。

堆内存详解

上面这个图大家应该已经很明白了吧。大家就可以理解成一个 `房子` 被分成了几个 `房间` ，每个房间的作用不同而已，有的是婴儿住的，有的是父母住的，有的是爷爷奶奶住的

- 堆内存被划分为 `两块` ，一块的 `年轻代` ，另一块是 `老年代` 。
- 年轻代又分为 `Eden` 和 `survivor` 。他俩空间大小比例默认为8:2,
- 幸存区又分为 `s0` 和 `s1` 。这两个空间大小是一模一样的，就是一对双胞胎，他俩是1:1的比例

堆内存垃圾回收过程

第一步

新生成 的对象首先放到 Eden 区，当Eden区 满了 会触发 Minor GC 。

第二步

第一步GC活下来的对象，会被移动到 survivor 区中的S0区，S0区满了之后会触发 Minor GC ，S0区存活下来的对象会被移动到S1区，S0区空闲。

S1满了之后在GC，存活下来的再次移动到S0区，S1区空闲，这样反反复复GC，每GC一次，对象的年龄就 涨一岁 ，达到某个值后（15），就会进入 老年代 。

第三步

在发生一次 Minor GC 后（前提条件），老年代可能会出现 Major GC ，这个视垃圾回收器而定。

Full GC触发条件

- 手动调用System.gc，会不断的执行Full GC
- 老年代空间不足/满了
- 方法区空间不足/满了

注意

们需要记住一个单词： stop-the-world 。它会在任何一种GC算法中发生。stop-the-world 意味着JVM因为需要执行GC而 停止 应用程序的执行。

当stop-the-world 发生时，除GC所需的线程外，所有的 线程 都进入 等待 状态，直到GC任务完成。GC优化很多时候就是减少stop-the-world 的发生。

回收哪些区域的对象

需要注意的是，JVM GC只回收 堆内存 和 方法区内 的对象。而 栈内存 的数据，在超出作用域后会被JVM自动释放掉，所以其不在JVM GC的管理范围内。

堆内存常见参数配置

参数	描述
-Xms	堆内存初始大小，单位m、g
-Xmx	堆内存最大允许大小，一般不要大于物理内存的80%
-XX:PermSize	非堆内存初始大小，一般应用设置初始化200m，最大1024m就够了
-XX:MaxPermSize	非堆内存最大允许大小
-XX:NewSize（-Xns）	年轻代内存初始大小
-XX:MaxNewSize（-Xmn）	年轻代内存最大允许大小

-XX:SurvivorRatio=8	年轻代中Eden区与Survivor区的容量比例值，默认为8，即8:1
-Xss	堆栈内存大小
-XX:NewRatio=老年代/新生代	设置老年代和新生代的大小比例
-XX:+PrintGC	jvm启动后，只要遇到GC就会打印日志
-XX:+PrintGCDetails	查看GC详细信息，包括各个区的情况
-XX:MaxDirectMemorySize	在NIO中可以直接访问**直接内存**，这个就是设置它的大小，不设置默认就是最大堆空间的值-Xmx
-XX:+DisableExplicitGC	关闭System.gc()
-XX:MaxTenuringThreshold	垃圾可以进入老年代的年龄
-Xnoclassgc	禁用垃圾回收
-XX:TLABWasteTargetPercent	TLAB占eden区的百分比，默认是1%
-XX:+CollectGen0First	FullGC时是否先YGC，默认false

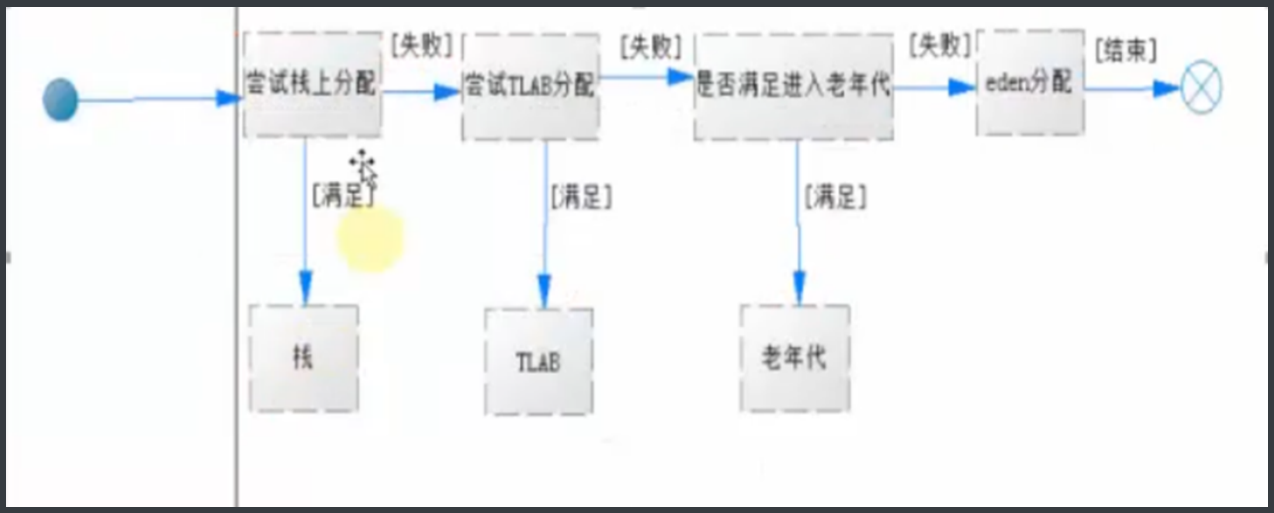
TLAB 内存

TLAB全称是Thread Local Allocation Buffer即 `线程本地分配缓存` ，从名字上看是一个线程专用的内存分配区域，是为了加速对象分配而生的。

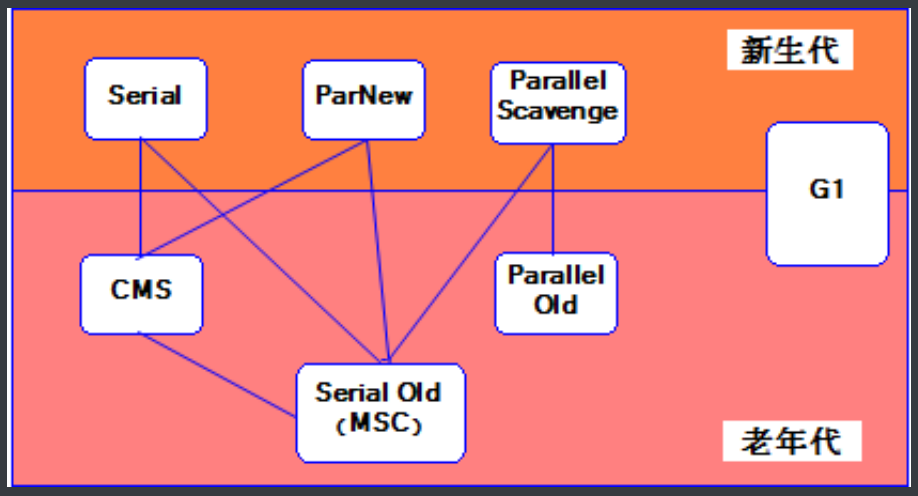
每一个线程都会产生一个TLAB，该线程独享的工作区域，java虚拟机使用这种TLAB区来避免多线程冲突问题，提高了对象分配的效率。

TLAB空间一般不会太大，当大对象无法在TLAB分配时，则会直接分配到堆上。

参数	**描述**
-Xx:+UseTLAB	使用TLAB
-XX:+TLABSize	设置TLAB大小
-XX:TLABRefillWasteFraction	设置维护进入TLAB空间的单个对象大小，他是一个比例值，默认为64，即如果对象大于整个空间的1/64，则在堆创建
-XX:+PrintTLAB	查看TLAB信息
-Xx:ResizeTLAB	自调整TLABRefillWasteFraction阀值。



垃圾回收器总览



新生代可配置的回收器：Serial、ParNew、Parallel Scavenge

老年代配置的回收器：CMS、Serial Old、Parallel Old

新生代和老年代区域的回收器之间进行连线，说明他们之间可以搭配使用。

新生代垃圾回收器

Serial 垃圾回收器

Serial收集器是最基本的、发展历史最悠久的收集器。俗称为： `串行回收器` ， 采用 `复制算法` 进行垃圾回收

特点

串行回收器是指使用单线程进行垃圾回收的回收器。每次回收时，串行回收器只有一个工作线程。

对于并行能力较弱的单CPU计算机来说，串行回收器的专注性和独占性往往有更好的性能表现。

它存在Stop The World问题，及垃圾回收时，要停止程序的运行。

使用 -XX:+UseSerialGC 参数可以设置新生代使用这个串行回收器

ParNew 垃圾回收器

ParNew其实就是Serial的 多线程 版本，除了使用多线程之外，其余参数和Serial一模一样。俗称： 并行垃圾回收器 ， 采用 复制算法 进行垃圾回收

特点

ParNew默认开启的线程数与CPU数量相同，在CPU核数很多的机器上，可以通过参数 -XX:ParallelGCThreads 来设置线程数。

它是目前新生代首选的垃圾回收器，因为除了ParNew之外，它是唯一一个能与老年代CMS配合工作的。

它同样存在Stop The World问题

使用 -XX:+UseParNewGC 参数可以设置新生代使用这个并行回收器

ParallelGC 回收器

ParallelGC使用复制算法回收垃圾，也是多线程的。

特点

就是非常关注系统的吞吐量， 吞吐量 = 代码运行时间 /(代码运行时间 + 垃圾收集时间)

-XX:MaxGCPauseMillis ： 设置最大垃圾收集停顿时间，可用把虚拟机在GC停顿的时间控制在MaxGCPauseMillis范围内，如果希望减少GC停顿时间可以将MaxGCPauseMillis设置的很小，但是会导致 GC频繁 ，从而增加了GC的 总时间 ， 降低 了 吞吐量 。所以需要根据实际情况设置该值。

-Xx:GCTimeRatio ： 设置吞吐量大小，它是一个0到100之间的整数，默认情况下他的取值是 99 ，那么系统将花费不超过 1/(1+n) 的时间用于垃圾回收，也就是 1/(1+99)=1% 的时间。

另外还可以指定 -XX:+UseAdaptiveSizePolicy 打开自适应模式，在这种模式下，新生代的大小、eden、from/to的比例，以及晋升老年代的对象年龄参数会被自动调整，以达到在堆大小、吞吐量和停顿时间之间的平衡点。

使用-XX:+UseParallelGC参数可以设置新生代使用这个并行回收器

老年代垃圾回收器

SerialOld 垃圾回收器

SerialOld是Serial回收器的 老年代 回收器版本，它同样是一个 单线程 回收器。

用途

- 一个是在JDK1.5及之前的版本中与Parallel Scavenge收集器搭配使用，
- 另一个就是作为CMS收集器的后备预案，如果CMS出现Concurrent Mode Failure，则SerialOld将作为后备收集器。

使用算法 ： 标记 - 整理算法

ParallelOldGC 回收器

老年代 ParallelOldGC 回收器也是一种多线程的回收器，和新生代的ParallelGC回收器一样，也是一种关注吞吐量的回收器，他使用了 标记压缩算法 进行实现。

-XX:+UseParallelOldGc 进行设置老年代使用该回收器

-XX:+ParallelGCThreads 也可以设置垃圾收集时的线程数量。

CMS 回收器

CMS全称为:Concurrent Mark Sweep意为并发标记清除，他使用的是 标记清除法 。主要关注系统停顿时间。

使用 -XX:+UseConcMarkSweepGC 进行设置老年代使用该回收器。

使用 -XX:ConcGCThreads 设置并发线程数量。

特点

CMS并不是独占的回收器，也就说CMS回收的过程中，应用程序仍然在不停的工作，又会有新的垃圾不断的产生，所以在使用CMS的过程中应该确保应用程序的内存足够可用。

CMS不会等到应用程序 饱和 的时候才去回收垃圾，而是在某一阀值的时候开始回收，回收阀值可用指定的参数进行配置： -XX:CMSInitiatingoccupancyFraction 来指定，默认为 68 ，也就是说当老年代的空间 使用率 达到 68% 的时候，会 执行 CMS回收。

如果内存使用率增长的很快，在CMS执行的过程中，已经出现了内存不足的情况，此时CMS回收就会失败，虚拟机将启动老年代 串行 回收器； SerialOldGC 进行垃圾回收，这会导致应用程序中断，直到垃圾回收完成后才会正常工作。

这个过程GC的停顿时间可能较长，所以 -XX:CMSInitiatingoccupancyFraction 的设置要根据实际的情况。

之前我们在学习算法的时候说过，标记清除法有个缺点就是存在 内存碎片 的问题，那么CMS有个参数设置 -XX:+UseCMSCompactAtFullCollecion 可以使CMS回收完成之后进行一次 碎片整理 。

-XX:CMSFullGCsBeforeCompaction 参数可以设置进行多少次CMS回收之后，对内存进行一次 压缩 。

Minor Gc和Full GC 有什么不同呢?

- 新生代 GC（Minor GC）:指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- 老年代 GC（Major GC/Full GC）: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（并非绝对），Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。

程序计数器为什么是私有的?

程序计数器主要有下面两个作用：

- 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
- 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。

所以，程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。

常用 GC 调优策略有哪些?

GC 调优目的

将转移到老年代的对象数量降低到最小； 减少 GC 的执行时间。

GC 调优策略

`策略 1`： 将新对象预留在新生代，由于 Full GC 的成本远高于 Minor GC，因此尽可能将对象分配在新生代是明智的做法，实际项目中根据 GC 日志分析新生代空间大小分配是否合理，适当通过“-Xmn”命令调节新生代大小，最大限度降低新对象直接进入老年代的情况。

`策略 2`： 大对象进入老年代，虽然大部分情况下，将对象分配在新生代是合理的。但是对于大对象这种做法却值得商榷，大对象如果首次在新生代分配可能会出现空间不足导致很多年龄不够的小对象被分配的老年代，破坏新生代的对象结构，可能会出现频繁的 full gc。因此，对于大对象，可以设置直接进入老年代（当然短命的大对象对于垃圾回收来说简直就是噩梦）。-XX:PretenureSizeThreshold 可以设置直接进入老年代的对象大小。

`策略 3`： 合理设置进入老年代对象的年龄，-XX:MaxTenuringThreshold 设置对象进入老年代的年龄大小，减少老年代的内存占用，降低 full gc 发生的频率。

`策略 4`： 设置稳定的堆大小，堆大小设置有两个参数：-Xms 初始化堆大小，-Xmx 最大堆大小。

`策略5`： 注意： 如果满足下面的指标，则一般不需要进行 GC 优化：

> MinorGC 执行时间不到50ms； Minor GC 执行不频繁，约10秒一次； Full GC 执行时间不到1s； Full GC 执行频率不算频繁，不低于10分钟1次。

说一下 JVM 运行时数据区?

不同虚拟机的运行时数据区可能略微有所不同，但都会遵从 Java 虚拟机规范， Java 虚拟机规范规定的区域分为以下 5 个部分：

程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的字节码指令，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来完成；

Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息；

本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务 Java 方法的，而本地方法栈是为虚拟机调用 Native 方法服务的；

Java 堆（Java Heap）：Java 虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存；

方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

队列和栈是什么？有什么区别?

队列和栈都是被用来预存储数据的。

队列允许先进先出检索元素，但也有例外的情况，Deque 接口允许从两端检索元素。

栈和队列很相似，但它运行对元素进行后进先出进行检索。

说一下堆栈的区别?

功能方面：堆是用来存放对象的，栈是用来执行程序的。

共享性：堆是线程共享的，栈是线程私有的。

空间大小：堆大小远远大于栈。

说一下 JVM 调优的工具?

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具。

jconsole：用于对 JVM 中的内存、线程和类等进行监控；

jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等。

请问java中内存泄漏是什么意思？什么场景下会出现内存泄漏的情况?

Java中的内存泄露，广义并通俗的说，就是：不再会被使用的对象的内存不能被回收，就是内存泄露。

如果长生命周期的对象持有短生命周期的引用，就很可能会出现内存泄露。