

JVM

- 谈谈你对 JVM 的理解？ java 文件 先编译成--> class 文件 -->然后再加载到 JVM 中去运行
- Java 8 虚拟机和之前相比有什么变化更新？
- 什么是 OOM？什么是栈溢出 StackOverFlowError？怎么分析？
- JVM 的常用调优参数有哪些？
- 内存快照如何抓取？怎么分析 Dump 文件？
- 谈谈 JVM 中类加载器你的认识？ rt-jar ext application

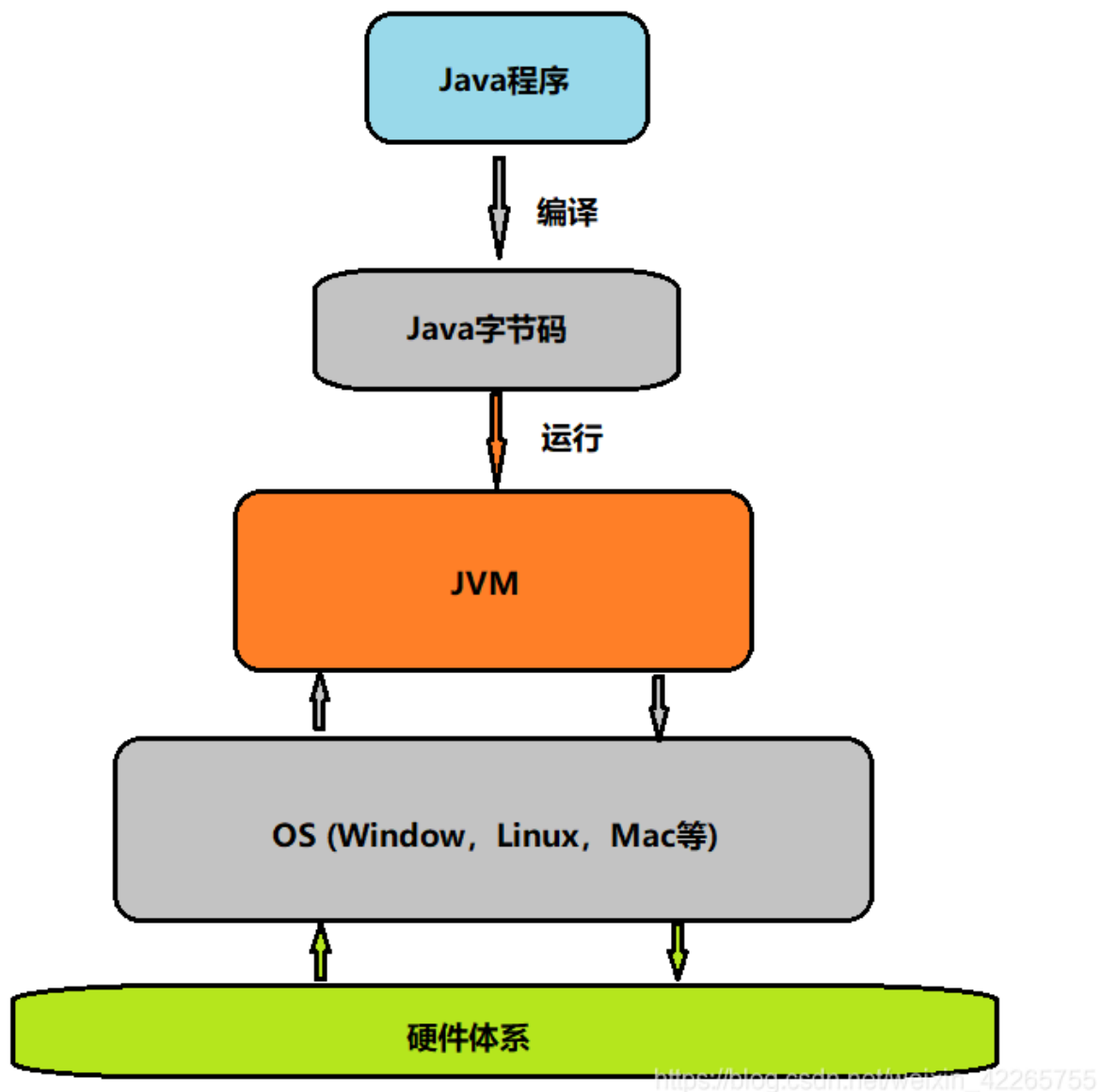
（学习方式：可以通过百度/看别人总结的思维导图- Process-On 搜索 JVM）

一、JVM是什么

JVM（Java Virtual Machine）是 Java 虚拟机，用于运行 Java 编译后的二进制字节码，最后生成机器指令。

JVM是Java能够跨平台的核心

JVM的位置



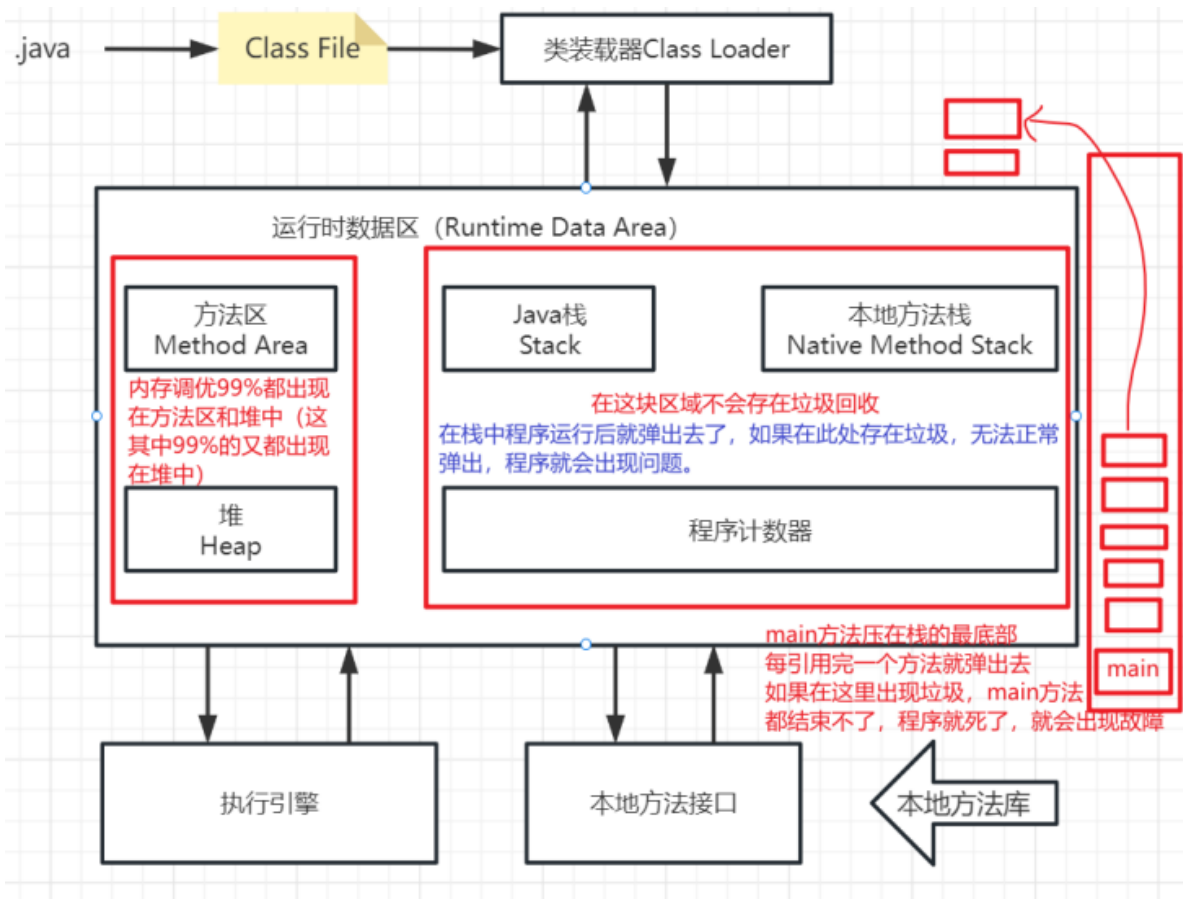
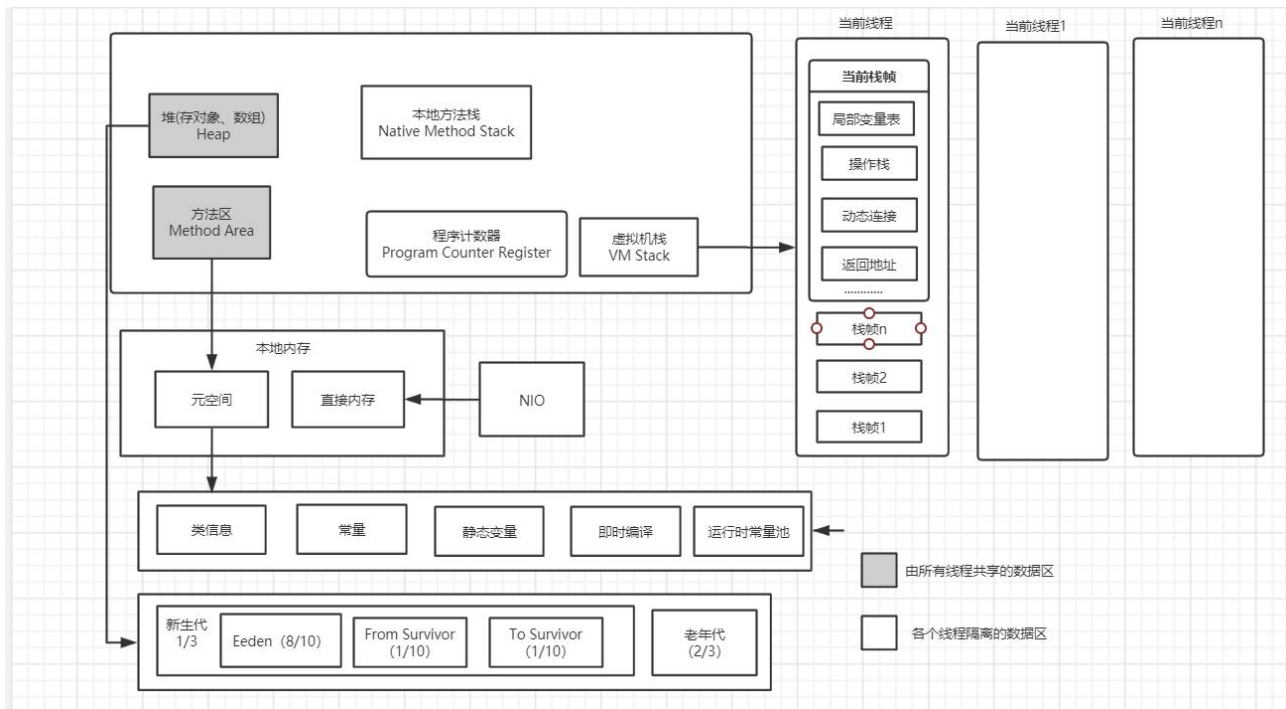
Java虚拟机 存在于操作系统上的 JRE 构建环境中

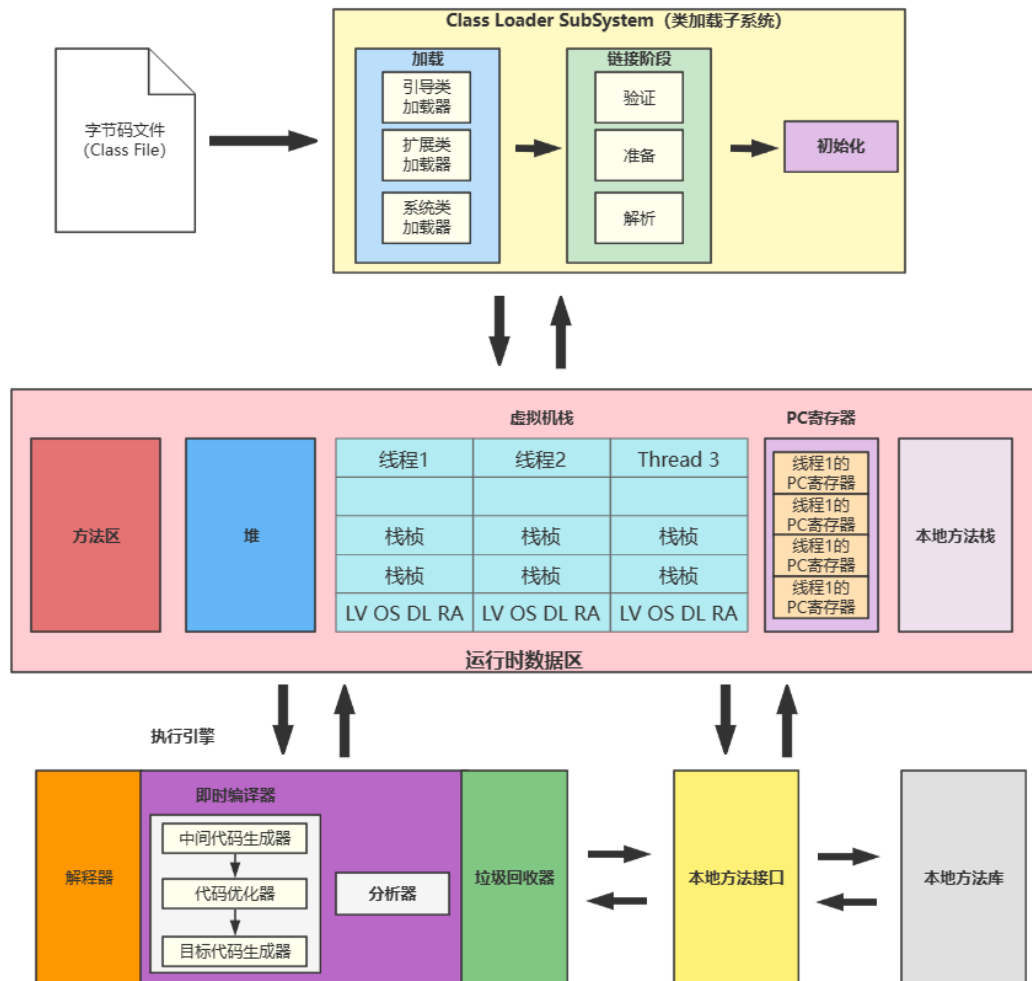
二、JVM的体系结构(和Jvm内存模型不是一个问题)

《深入理解Java虚拟机》学习笔记-知乎

一图看懂JVM内存分布，永久记住！ - 知乎 (zhihu.com)

终于搞懂了Java8的内存结构，再也不纠结方法区和常量池了！ - 腾讯云开发者社区-腾讯云 (tencent.com)





三、类加载器

- 类加载过程：加载->连接->初始化。
- 连接过程又可分为三步：验证->准备->解析。

加载是类加载过程的第一步，主要完成下面 3 件事情：

- 通过全类名获取定义此类的二进制字节流
- 将字节流所代表的静态存储结构转换为方法区的运行时数据结构
- 在内存中生成一个代表该类的 `Class` 对象，作为方法区这些数据的访问入口

类加载过程

- [类加载过程详解 | JavaGuide\(Java面试+学习指南\)](#)

类的生命周期主要有七个阶段：加载、验证、准备、解析、初始化、使用、卸载

其中，验证、准备和解析阶段又可以被称为连接阶段

1.加载：将 *Class* 的字节码形式转换成内存形式的 *Class* 对象

- 通过全类名获取二进制字节文件流
- 将字节流所代表的静态存储结构转换为方法区的运行时数据结构
- 在内存中生成一个代表该类的 *class* 对象，作为方法区中这些数据的访问入口

2.验证：验证是连接阶段的第一步，这一阶段的目的是确保 *Class* 文件的字节流中包含的信息符合《*Java* 虚拟机规范》的全部约束要求，保证这些信息被当作代码运行后不会危害虚拟机自身的安全。

- 文件格式验证：对二进制字节流操作，检查是否符合 *class* 文件格式规范
- 元数据验证：在方法区中进行，对字节码描述的信息进行语义分析，例如是否继承了不被允许继承的类
- 字节码验证：在方法区中进行，通过数据流和控制流分析，确定程序的语法合法。例如：参数类型是否正确等，但是不是所有的类文件都会经过字节码校验，比如核心类
- 符号引用验证：在方法区中进行，验证类的正确性，例如：正确的访问权限，以及方法名等字段是否存在等，主要是为了保证解析阶段正常执行。

3.准备：准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

- 只分配类变量
- *jdk7*之前永久代中，7之后存放在元空间（堆中）。注意！永久代和元空间是方法区的不同实现存储方式。方法区可以理解成一种规范的逻辑空间，用来存储类信息、字段信息、方法信息、常量、静态变量、即时编译器编译后的代码缓存等数据。
- 给类变量赋初始值是赋的变量类型的默认值，而不是用户提供的初始值，例如 `public static int value=111`，初始值为0，但是，*final*修饰的特殊，`public static final int value=111`初始值为111。

4.解析：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。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用限定符 7 类符号引用进行。也就是符号代表的引用对象替换为直接指向目标的指针或者偏移量。

5.初始化过程：初始化阶段是执行初始化方法 `<clinit> ()` 方法的过程，是类加载的最后一步，这一步 *JVM* 才开始真正执行类中定义的 *Java* 程序代码(字节码)。根据不同的字节码指令，对类进行初始化，比如调用类的静态信息或者对类变量进行赋值。

6.类卸载：卸载类即该类的 *Class* 对象被 *GC*。

卸载类需要满足 3 个要求：

- 该类的所有的实例对象都已被 *GC*，也就是说堆不存在该类的实例对象。
- 该类没有在其他任何地方被引用

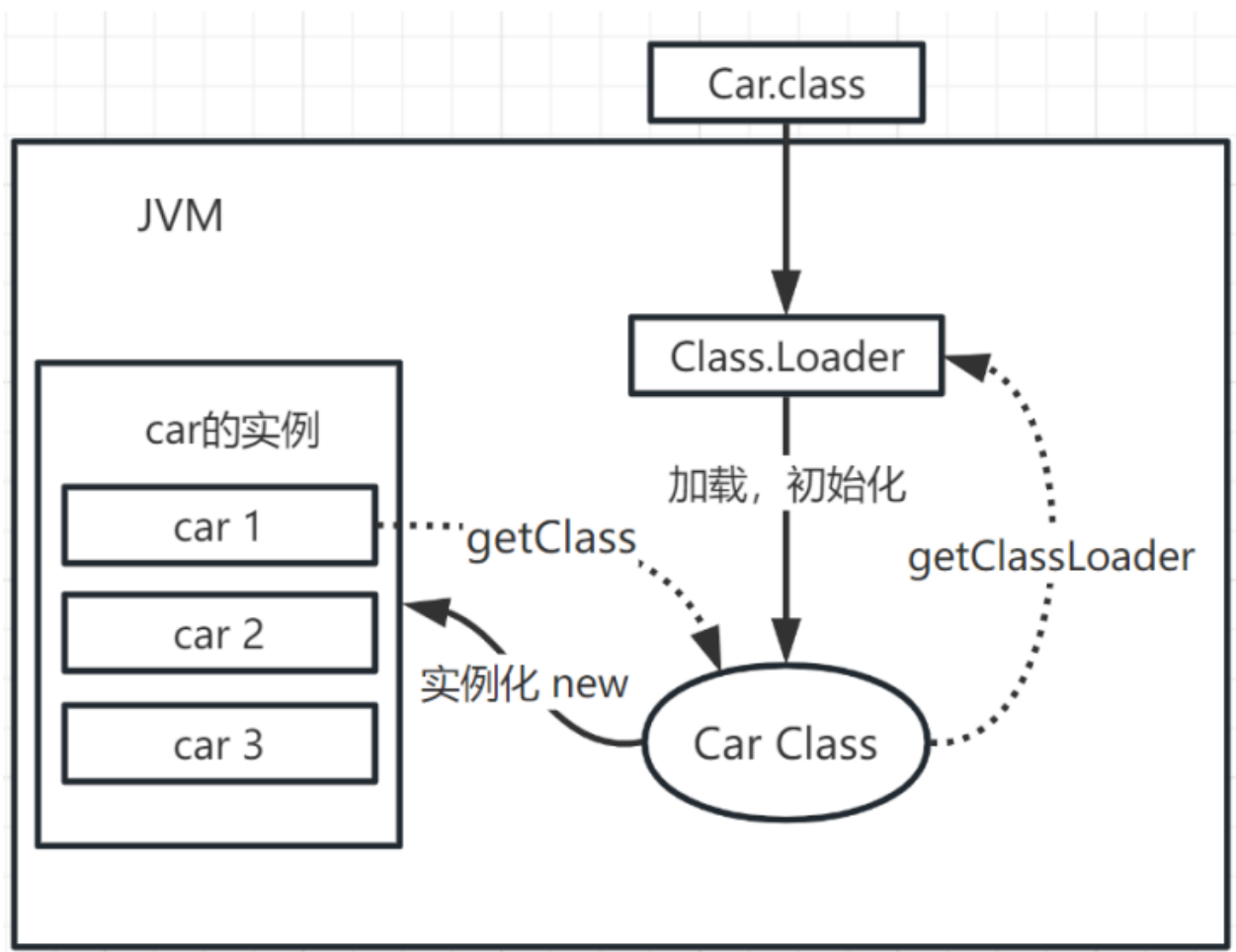
- 该类的类加载器的实例已被 *GC*

所以，在 *JVM* 生命周期内，由 *jvm* 自带的类加载器加载的类是不会被卸载的。但是由我们自定义的类加载器加载的类是可能被卸载的。*JDK* 自带的 `BootstrapClassLoader`, `ExtClassLoader`, `AppClassLoader` 负责加载 *JDK* 提供的类，所以它们(类加载器的实例)肯定不会被回收。而我们自定义的类加载器的实例是可以被回收的，所以使用我们自定义加载器加载的类是可以被卸载掉的

作用

- 类加载器的主要作用就是加载 **Java** 类的字节码（`.class` 文件）到 **JVM** 中（在内存中生成一个代表该类的 `Class` 对象）。字节码(`.class`)可以是 **Java** 源程序（`.java` 文件）经过 `javac` 编译得来，也可以是通过工具动态生成或者通过网络下载得来。（负责将 **Class** 的字节码形式转换成内存形式的 **Class** 对象）
- 其实除了加载类之外，类加载器还可以加载 **Java** 应用所需的资源如文本、图像、配置文件、视频等等文件资源。

每个 *Java* 类都有一个引用指向加载它的 `ClassLoader`。不过，数组类(*Arrays*)不是通过 `ClassLoader` 创建的，而是 *JVM* 在需要的时候自动创建的，数组类通过 `getClassLoader()` 方法获取 `ClassLoader` 的时候和该数组的元素类型的 `ClassLoader` 是一致的。



类加载器分类

Java中的类加载器大致分为两类：

- 一类是系统提供的，JVM内置的classLoader
- 另外一类则是由 Java 应用开发人员编写的。

JVM 中内置了三个重要的 **ClassLoader**：

BootstrapClassLoader(启动类加载器)：最顶层的加载类，由 C++实现，通常表示为 `null`（无法获取引用），并且没有父级，主要用来加载 JDK 内部的核心类库（`%JAVA_HOME%/lib` 目录下的 `rt.jar`、`resources.jar`、`charsets.jar` 等 jar 包和类）以及被 `-xbootclasspath` 参数指定的路径下的所有类，它们都是用原生代码来实现的，并不继承自 `java.lang.ClassLoader`。

ExtensionClassLoader(扩展类加载器)：主要负责加载 `%JRE_HOME%/lib/ext` 目录下的 jar 包和类以及被 `java.ext.dirs` 系统变量所指定的路径下的所有类。

AppClassLoader(应用程序类加载器)：面向我们用户的加载器，负责加载当前应用 `classpath` 下的所有 jar 包和类。

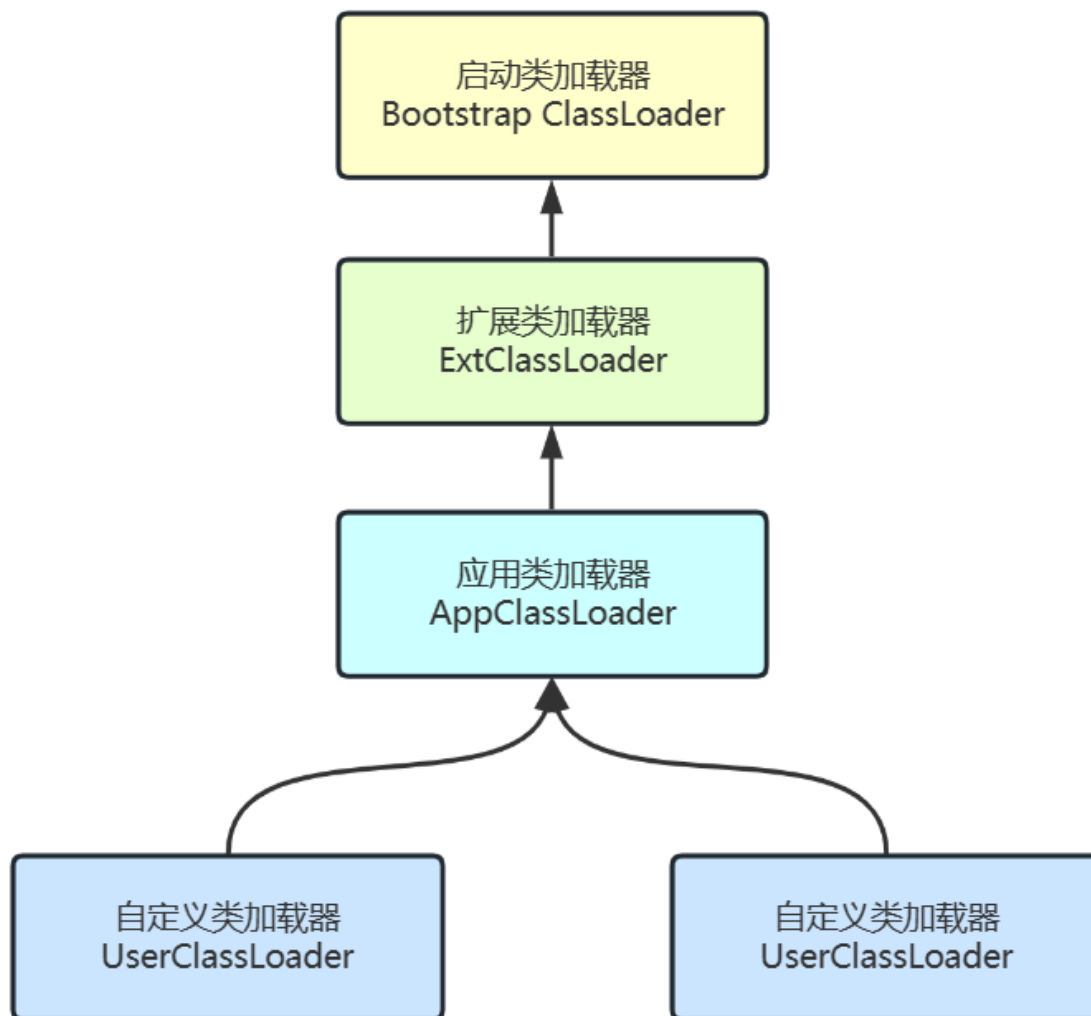
拓展:

- **rt.jar**: *rt* 代表“RunTime”，**rt.jar**是Java基础类库，包含Java doc里面看到的所有的类的类文件。也就是说，我们常用内置库 **java.xxx.***都在里面，比如**java.util.***、**java.io.***、**java.nio.***、**java.lang.***、**java.sql.***、**java.math.***。
- Java 9 引入了模块系统，并且略微更改了上述的类加载器。扩展类加载器被改名为平台类加载器（*platform class loader*）。Java SE 中除了少数几个关键模块，比如说 **java.base** 是由启动类加载器加载之外，其他的模块均由平台类加载器所加载。
- **BootstrapClassLoader** 由 C++ 实现，由于这个 C++ 实现的类加载器在 Java 中是没有与之对应的类的，所以拿到的结果是 *null*。
- 这个三个类加载器之间并不存在继承关系

除了系统提供的类加载器以外，开发人员可以通过继承 **java.lang.ClassLoader**类的方式实现自己的类加载器，以满足一些特殊的需求。

除了 **BootstrapClassLoader** 是 JVM 自身的一部分之外，其他所有的类加载器都是在 JVM 外部实现的，并且全都继承自 **ClassLoader** 抽象类。这样做的好处是用户可以自定义类加载器，以便让应用程序自己决定如何去获取所需的类。

每个 **ClassLoader** 可以通过 **getParent()** 获取其父 **ClassLoader**，如果获取到 **ClassLoader** 为 *null* 的话，那么该类是通过 **BootstrapClassLoader** 加载的。

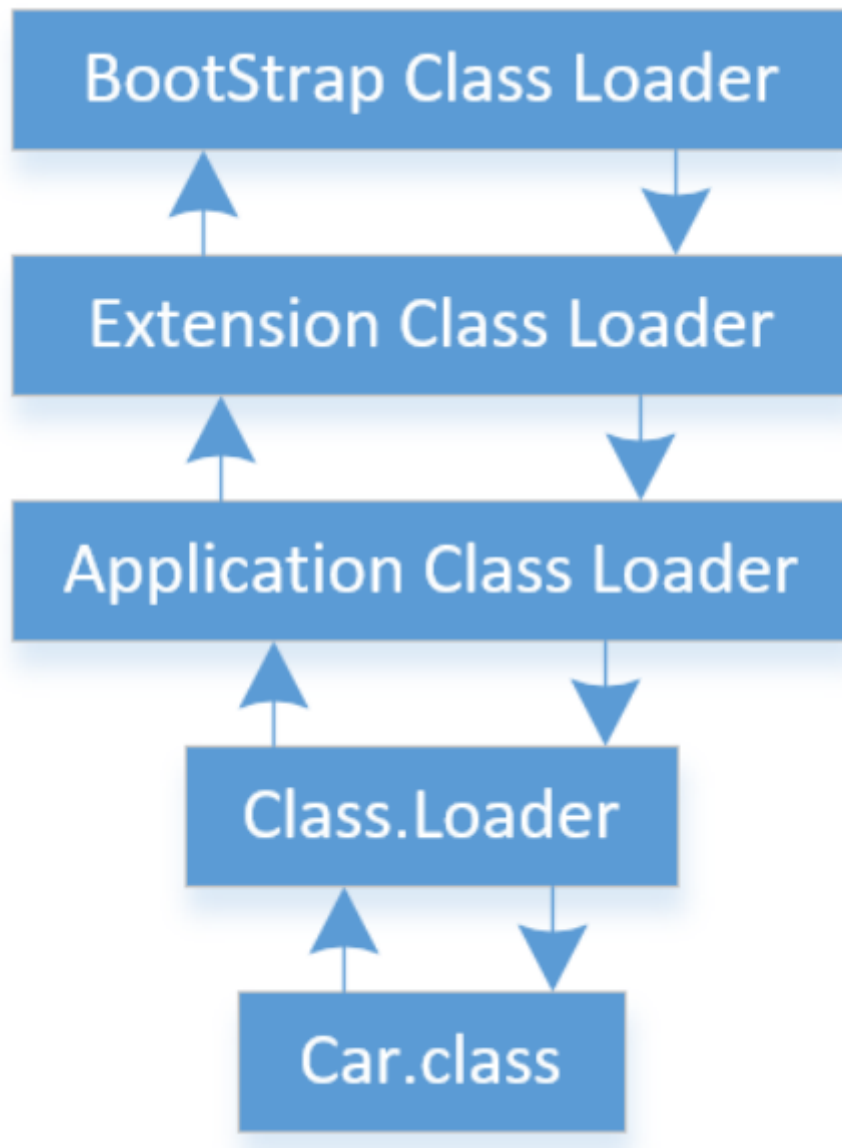


四、双亲委派机制

Java虚拟机对class文件采用的是按需加载的方式，也就是说当需要使用该类时才会将它的class文件加载到内存生成class对象，而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式即把请求交由父类加载器处理，它是一种任务委派模式，

`ClassLoader` 类使用委托模型来搜索类和资源。每个 `ClassLoader` 实例都有一个相关的父类加载器。需要查找类或资源时，`ClassLoader` 实例会在试图亲自查找类或资源之前，将搜索类或资源的任务委托给其父类加载器。虚拟机中被称为 "bootstrap class loader" 的内置类加载器本身没有父类加载器，但是可以作为 `ClassLoader` 实例的父类加载器。[类加载器详解（重点） | JavaGuide\(Java面试+学习指南\)](#)

- `ClassLoader` 类使用双亲委托模型来搜索类和资源。
- 双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应有自己的父类加载器。
- `ClassLoader` 实例会在试图亲自查找类或资源之前，将搜索类或资源的任务委托给其父类加载器。




1. 在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载（每个父类加载器都会走一遍这个流程）。
2. 类加载器在进行类加载的时候，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成（调用父加载器 `loadClass()` 方法来加载类）。这样的话，所有的请求最终都会传送到顶层的启动类加载器 `BootstrapClassLoader` 中。
3. 只有当父加载器反馈自己无法完成这个加载请求（它的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（调用自己的 `findClass()` 方法来加载类）。

作用/意义

（避免类的重复加载、保证了安全性）

1. 通过双亲委派的方式，可以避免类的重复加载，当父类加载器已经加载过某一个子类时，加载器就 不会再重新加载这个类。

2. 通过双亲委派的方式，还保证了安全性，因为`BootstrapClassLoader` 在加载的时候，只会加载 `JAVA_HOME` 中的jar包里面的类，如 `java.lang.Integer`，那么这个类是不会被随意替换的，除非破坏了 `JDK`。这样可以有效防止核心 `JavaAPI` 被篡改。

 **JVM**判定两个 **Java**类是否相同的依据：**JVM** 不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即使两个类来源于同一个 `.class` 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相同。

如何破坏双亲委派

一个非数组类的加载阶段（加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，这一步我们可以去完成还可以自定义类加载器去控制字节流的获取方式（重写一个类加载器的 `loadClass()` 方法）。

如果不想破坏双亲委派机制，那么只需要重写 `findClass()` 方法即可，如果想要破坏双亲委派机制，那么需要重写 `loadClass()`。[类加载器详解（重点） | JavaGuide\(Java面试+学习指南\)](#)

五、Java安全模型->沙箱安全机制

【叁】JVM-沙箱安全机制 - 简书 ([jianshu.com](https://www.jianshu.com/p/1e1e1e1e1e1e))

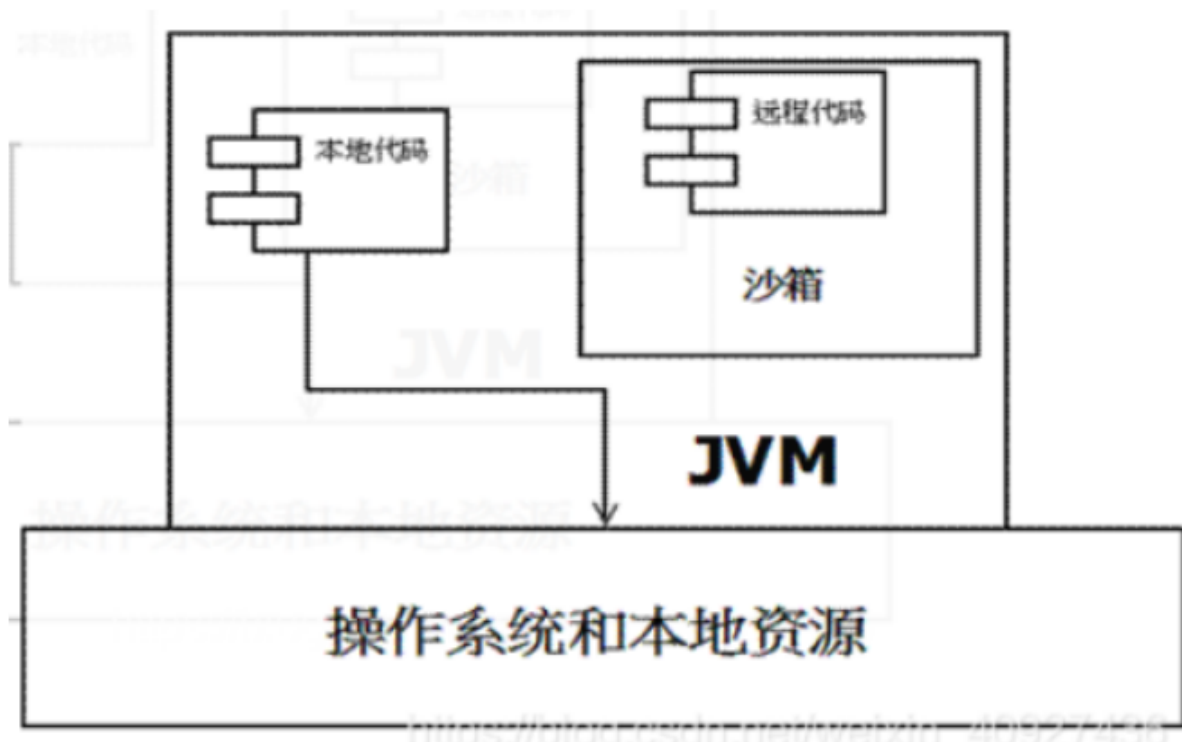
Java安全——理解Java沙箱-阿里云开发者社区 ([aliyun.com](https://developer.aliyun.com/article/111111))

Java沙箱机制的实现——安全管理器、访问控制器 - 掘金 ([juejin.cn](https://juejin.cn/post/111111))

Java安全模型的核心就是Java沙箱（`sandbox`），什么是沙箱？沙箱是一个限制程序运行的环境。沙箱机制就是将 `Java` 代码限定在虚拟机(`JVM`)特定的运行范围中，并且严格限制代码对本地系统资源访问，通过这样的措施来保证对代码的有效隔离，防止对本地系统造成破坏。沙箱主要限制系统资源访问，那系统资源包括什么？**CPU**、内存、文件系统、网络。不同级别的沙箱对这些资源访问的限制也可以不一样。

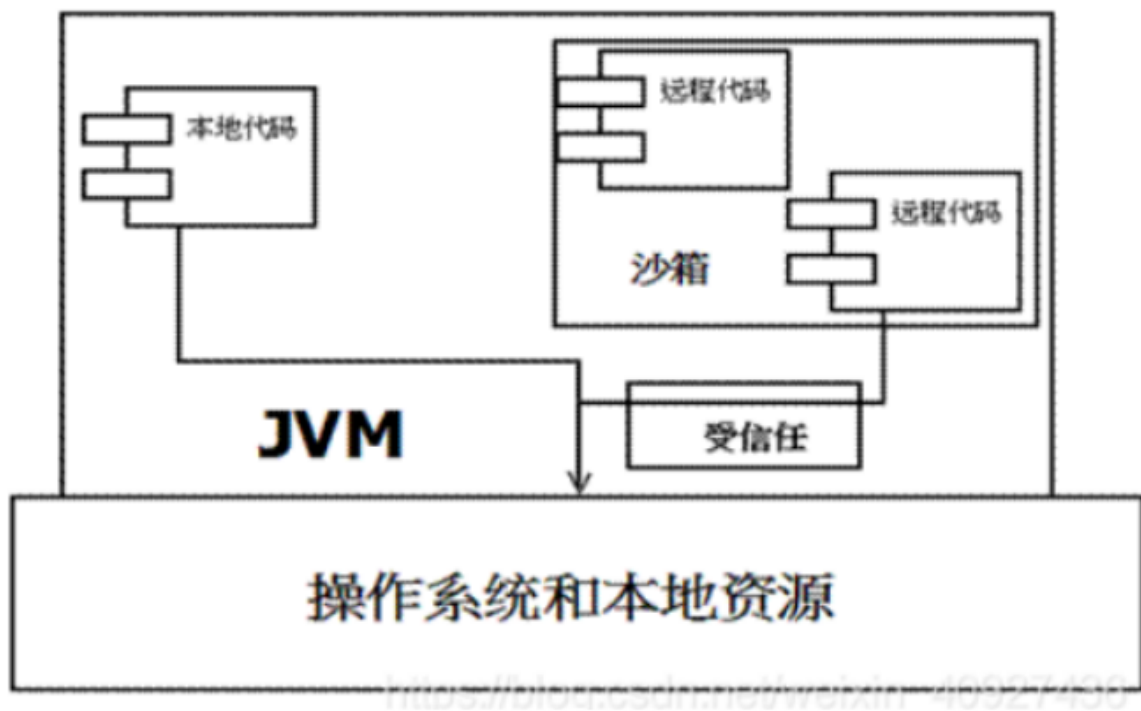
所有的Java程序运行都可以指定沙箱，可以定制安全策略。

在 `Java` 中将执行程序分成本地代码和远程代码两种，本地代码默认视为可信任的，而远程代码则被看作是不受信的。对于授信的本地代码，可以访问一切本地资源。而对于非授信的远程代码在早期的 `Java` 实现中，安全依赖于沙箱（`Sandbox`）机制。如下图所示 `JDK1.0` 安全模型



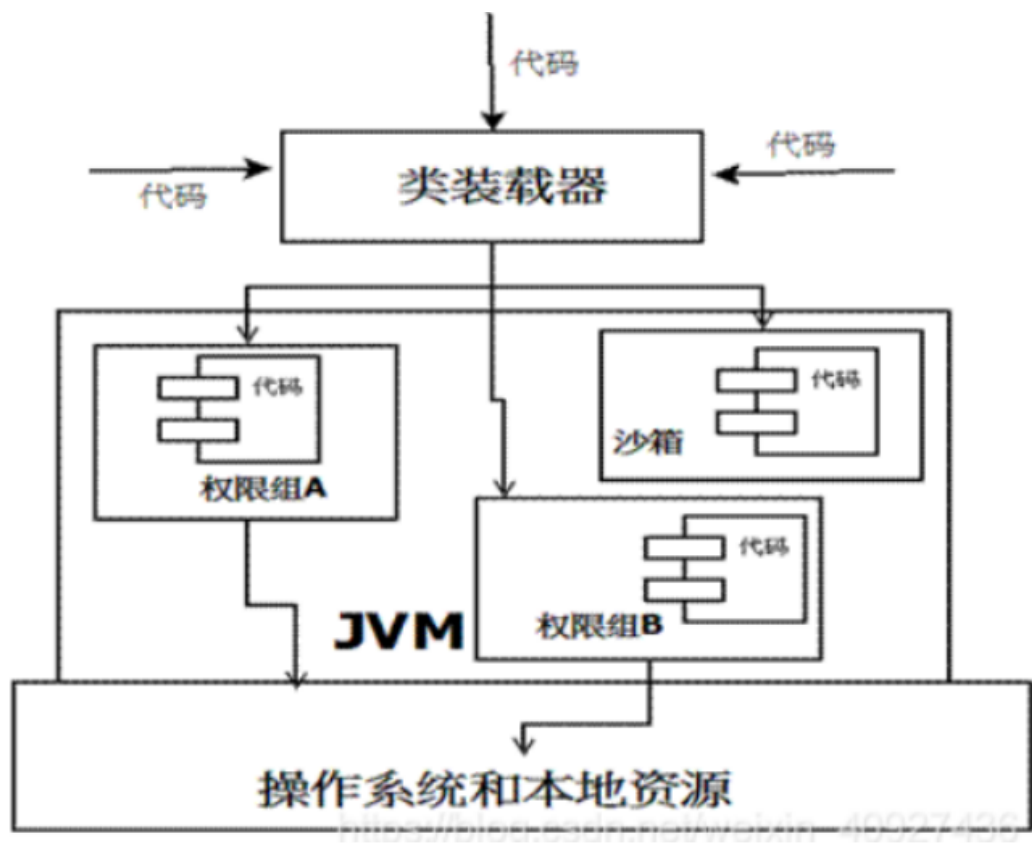
JDK1.0沙箱安全模型

但如此严格的安全机制也给程序的功能扩展带来障碍，比如当用户希望远程代码访问本地系统文件的时候，就无法实现。因此在后续的 Java1.1 版本中，针对安全机制做了改进，增加了安全策略，允许用户指定代码对本地资源的访问权限。如下图所示 JDK1.1 安全模型



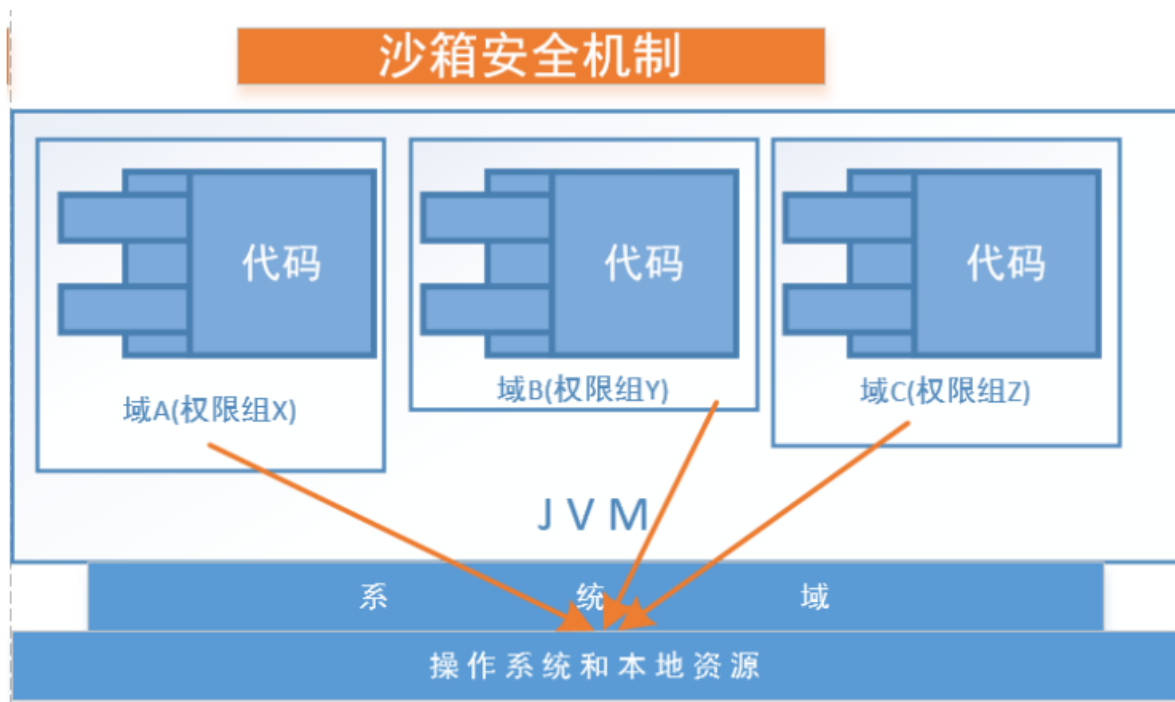
JDK1.1沙箱安全模型

在 Java1.2 版本中，再次改进了安全机制，增加了代码签名。不论本地代码或是远程代码，都会按照用户的安全策略设定，由类加载器加载到虚拟机中权限不同的运行空间，来实现差异化的代码执行权限控制。如下图所示 JDK1.2 安全模型



JDK1.2沙箱安全模型

当前最新的安全机制实现，则引入了域 (Domain) 的概念。虚拟机会把所有代码加载到不同的系统域和应用域，系统域部分专门负责与关键资源进行交互，而各个应用域部分则通过系统域的部分代理来对各种需要的资源进行访问。虚拟机中不同的受保护域 (Protected Domain)，对应不一样的权限 (Permission)。存在于不同域中的类文件就具有了当前域的全部权限，如下图所示最新的安全模型 JDK1.6



JDK1.6沙箱安全模型

- 双亲委派机制-保证JVM不被加载的代码破坏
- 安全权限机制-保证机器资源不被JVM里运行的程序破坏

沙箱构成基础

- 字节码校验器 (*bytecode verifier*)：确保Java类文件遵循Java语言规范。这样可以帮助Java程序实现内存保护。但是不是所有的类文件都会经过字节码校验，比如核心类。
- 类加载器 (*class loader*)：所有的Java类都是通过类加载器加载的，可以自定义类加载器来设置加载类的权限。
- 存取控制器 (*access controller*)：存取控制器可以控制核心API对操作系统的存取权限，而这个控制的策略设定，可以由用户指定。
- 安全管理器 (*security manager*)：是核心API和操作系统之间的主要接口。实现权限控制，比存取控制器优先级高。
- 安全软件包 (*security package*)：java.security下的类和扩展包下的类，允许用户为自己的应用增加新的安全特性，包括：
 - 安全提供者
 - 消息摘要
 - 数字签名
 - 加密
 - 鉴别

六、Native关键字

1. 凡是带了 **native** 关键字的，说明 **java** 的作用范围达不到了，会去调用底层C语言的库
2. **native**是一个计算机的函数,一个Native Method就是一个调用非Java代码的接口
3. 会进入本地方法栈，调用本地方法接口 **JNI** （ **Java Native Interface** ）

*JNI*全称为*Java Native Interface*. 它可以简单理解为是本地方法的接口，即允许在*Java*虚拟机里面的*Java*代码可以和如C，C++等其他底层语言进行交互（即可互相调用）

一般情况下，当你无法用纯*Java*来实现需求的时候，就需要使用*JNI*来用底层语言编写的本地方法来满足这些该需求

4. **JNI** 的作用：扩展 **Java** 的使用，融合不同的编程语言为 **Java** 所用（最初就是想融合C、C++，因为*Java*刚出来的时候C、C++横行，想要立足，就必须要有调用C、C++的程序，于是，它在内存区域中专门开辟了一块标记区域：**Native Method Stack**，登记**native**方法，在最终执行的时候，加载本地方法库中的方法通过 **JNI** 进行调用）
5. 目前存在的一些*Java*调用其他程序语言所编写方法的接口：
 - a. 网景的 **Java Runtime Interface(JRI)**
 - b. 微软的 **Raw Native Interface and Java/COM Interface(RNI)**

七、程序计数器

程序计数器：Program Counter Register

程序计数器是一块较小的内存空间，可以看作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、线程恢复等功能都需要依赖这个计数器来完成。

另外，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

从上面的介绍中我们知道了程序计数器主要有两个作用：

- 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。

⚠ 注意：程序计数器是唯一一个不会出现 `OutOfMemoryError` 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。`OutOfMemoryError`和GC没有关系，是两回事不要弄混。

八、方法区 ❤️❤️❤️❤️❤️❤️

Java 内存区域详解（重点） | [JavaGuide\(Java面试+学习指南\)](#)

方法区定义以及结构

方法区属于是 JVM 运行时数据区域的一块逻辑区域，是各个线程共享的内存区域。

方法区是一种概念，因此，在不同的虚拟机实现上，方法区的实现也是不同的。

🦋 对于 *hotSpot* 虚拟机而言，在 *jdk7* 之前、*jdk7*、*jdk8* 及之后的版本对方法区的实现都不一样。

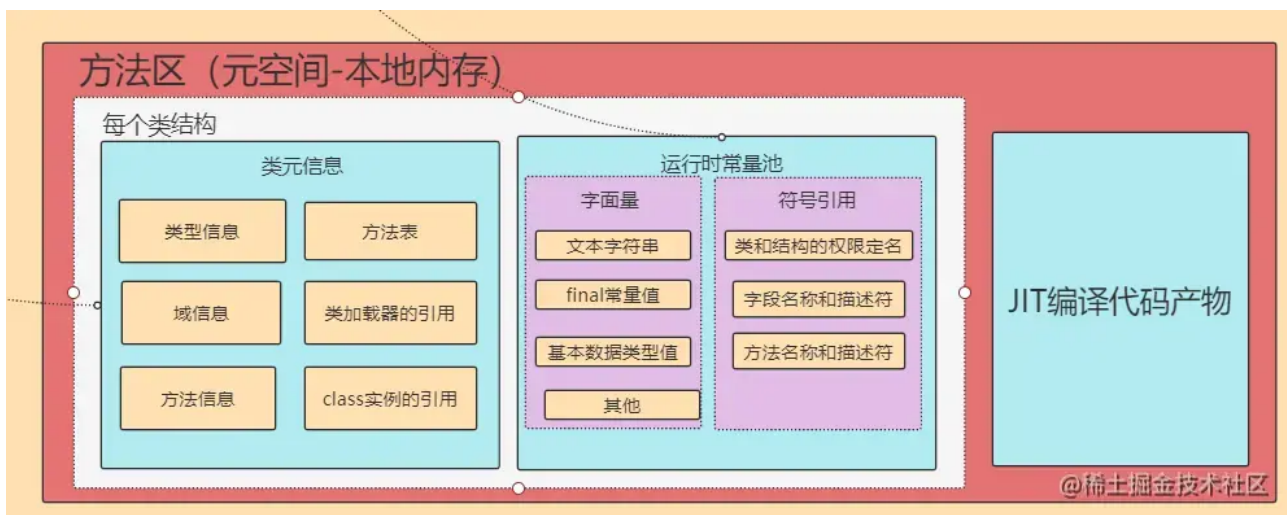
理解JVM运行时数据区（五）方法区 - 掘金 ([juejin.cn](#))

jdk1.6 及以前，使用永久代来实现方法区，其中保存有类型信息、字段(域)信息、方法信息、常量、静态变量、运行时常量池、*JIT* 代码缓存等。并且永久代放在堆空间中。

jdk1.7 中，静态变量和字符串常量池被移出永久代，并放入了堆空间中。

到了 *jdk8* 的时候，永久代被彻底废除，采用元空间来实现方法区，此时元空间是放在本地内存中的。而原先永久代里面存放的数据也相应的存放到了元空间，不过静态变量和字符串常量池依旧放在堆空间中。

当虚拟机要使用一个类时，它需要读取并解析 `Class` 文件获取相关信息，再将信息存入到方法区。方法区存储已经被虚拟机加载的类信息（构造方法，接口定义）、字段信息、方法信息、运行时常量池以及即时编译器编译后的代码缓存等数据。[理解JVM运行时数据区（五）方法区 - 掘金 \(juejin.cn\)](#)



类信息：

- 完整的有效名称，也可以说类的全限定名
- 直接父类的全限定名
- 修饰符（*public*、*abstract*、*final*的某个子集）
- 实现的所有接口的信息，这个是放在一个有序列表里面，因为可以实现多接口。

字段信息/域信息：

- 字段声明的顺序
- 字段名称
- 字段类型
- 字段的修饰符

方法信息：

- 方法名称
- 方法返回类型
- 方法参数的数量和类型
- 方法的修饰符
- 方法的字节码、操作数栈、局部变量表及大小（*abstract*和*native*除外）
- 异常表：每个异常处理的开始位置、结束位置、代码处理在程序计数器中的偏移地址、被捕获的异常类和常量池引用。（*abstract* 和 *native* 除外）

运行时常量池：在类加载时，也会将.class的字节码文件中的常量池载入到内存中，并保存在方法区中。我们常说的常量池就是指方法区中的运行时常量池。

即使编译器编译后的缓存代码(JIT编译后的缓存代码)：从字面意思理解就是代码缓存区，它缓存的是JIT（Just in Time）即时编译期编译的代码。JVM会对频繁使用的代码即热点代码，在达到一定的使用次数后，会编译成本地平台相关的机器码，这样在下次执行的时候就能更快的运行。

- 被多次调用的方法
- 被多次执行的循环体

运行时常量池

理解JVM运行时数据区（五）方法区 - 掘金 (juejin.cn)

面试题系列第5篇：JDK的运行时常量池、字符串常量池、静态常量池，还傻傻分不清？ - 腾讯云开发者社区-腾讯云 (tencent.com)

Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有用于存放编译期生成的各种字面量（Literal）和符号引用（Symbolic Reference）的常量池表（**Constant Pool Table**）

```
interfaces: 0, fields: 4, methods: 2, attributes: 1
Constant pool:
 #1 = Methodref      #11.#35      // java/lang/Object."<init>":()V
 #2 = String         #36          // zhuanghz
 #3 = Fieldref       #9.#37       // com/zhz/leetcode/test/Test.name:Ljava/lang/String;
 #4 = Fieldref       #9.#38       // com/zhz/leetcode/test/Test.age:I
 #5 = Long           666666666L
 #7 = Fieldref       #9.#39       // com/zhz/leetcode/test/Test.id:J
 #8 = Fieldref       #9.#40       // com/zhz/leetcode/test/Test.num:I
 #9 = Class          #41          // com/zhz/leetcode/test/Test
#10 = Methodref      #9.#35       // com/zhz/leetcode/test/Test."<init>":()V
#11 = Class          #42          // java/lang/Object
#12 = Utf8           name
#13 = Utf8           Ljava/lang/String;
#14 = Utf8           age
#15 = Utf8           I
#16 = Utf8           id
#17 = Utf8           J
#18 = Utf8           num
#19 = Utf8           ConstantValue
#20 = Integer        666
#21 = Utf8           <init>
#22 = Utf8           ()V
#23 = Utf8           Code
#24 = Utf8           LineNumberTable
#25 = Utf8           LocalVariableTable
```

@稀土掘金技术社区

字面量：

- 文本字符串
- 被声明为**final**的常量值
- 基本数据类型值（如果是**int**值的话 大于-32768并小于32767 是不会有在常量池里面的，而是跟在字节码指令后面）

符号引用：

- 类和接口的全限定名 如 **#9 = class**

- 字段的名称和描述符 如 `#7 = Fieldref`
- 方法的名称和描述符 如 `#1 = Methodref`

定义：

JVM在完成类装载操作后，会将class文件中的常量(constant pool)载入到内存中，并保存在方法区。而放在方法区里的这块内存被称为运行时常量池。

运行时常量池就是将编译后的类信息放入方法区中，也就是说它是方法区的一部分。

运行时常量池用来动态获取类信息，包括：**class**文件元信息描述、编译后的代码数据、引用类型数据、类文件常量池等。

运行时常量池是在类加载完成之后，将每个class常量池中的符号引用值转存到运行时常量池中。每个class都有一个运行时常量池，类在解析之后将符号引用替换成直接引用，与全局常量池中的引用值保持一致。

java虚拟机为每个类和接口维护一个运行时常量池

jvm在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。而当类加载到内存中后，jvm就会将class常量池中的内容存放到运行时常量池中，由此可知，运行时常量池也是每个类都有一个。在上面我也说了，**class**常量池中存的是字面量和符号引用，也就是说他们存的并不是对象的实例，而是对象的符号引用值。而经过解析（**resolve**）之后，也就是把符号引用替换为直接引用，解析的过程会去查询全局字符串池，也就是我们上面所说的StringTable，以保证运行时常量池所引用的字符串与全局字符串池中所引用的是一致的。

 各种常量池的说明可以参考 [Java中几种常量池的区分 // Emanuel's Notes \(tangxman.github.io\)](https://tangxman.github.io)

字符串常量池

面试题系列第5篇：[JDK的运行时常量池、字符串常量池、静态常量池，还傻傻分不清？ - 腾讯云开发者社区-腾讯云 \(tencent.com\)](#)

字符串常量池 是 JVM 为了提升性能和减少内存消耗针对字符串（**String** 类）专门开辟的一块区域，主要目的是为了避免字符串的重复创建

HotSpot 虚拟机中字符串常量池的实现是

`src/hotspot/share/classfile/stringTable.cpp`, `StringTable` 本质上就是一个 `HashSet<String>`, 容量为 `StringTableSize` (可以通过 `-XX:StringTableSize` 参数来设置)。

`StringTable` 中保存的是字符串对象的引用, 字符串对象的引用指向堆中的字符串对象。

字符串池里的内容是在类加载完成, 经过验证、准备阶段之后存放在字符串常量池中。

基本流程是: 创建字符串之前检查常量池中是否存在, 如果存在则获取其引用, 如果不存在则创建并存入, 返回新对象引用。

永久代和元空间以及方法区的关系

方法区和永久代以及元空间的关系很像 Java 中接口和类的关系, 类实现了接口, 这里的类就可以看作是永久代和元空间, 接口可以看作是方法区, 也就是说永久代以及元空间是 HotSpot 虚拟机对虚拟机规范中方法区的两种实现方式。并且, 永久代是 JDK 1.8 之前的方法区实现, JDK 1.8 及以后方法区的实现变成了元空间。

为什么取消永久代使用元空间?

1. 由于使用永久代实现方法区的方式在后面发现了诸多问题, 相比其他的虚拟机更容易出现 OOM
 - 永久代调优困难
 - 垃圾回收效果不好
 2. 永久代在虚拟机内部的堆空间中, 本身大小就受到了限制, 就算再大也无法突破堆空间的大小限制。
 3. 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的限制, 虽然仍旧可能存在内存溢出, 但是比原来出现的概率会更小。内存溢出概率变小。
 4. Oracle 收购了号称世界最快的 JRockit 虚拟机, 并整合了 JRockit 虚拟机的优秀功能。既然 JRockit 使用的是比永久代更好的元空间, 那就干脆去掉永久代, 使用元空间。
-

常用参数

JDK 1.8 之前永久代还没被彻底移除的时候通常通过下面这些参数来调节方法区大小。

```
-XX:PermSize=N //方法区（永久代）初始大小  
-XX:MaxPermSize=N //方法区（永久代）最大大小,超过这个值将会抛出  
OutOfMemoryError 异常:java.lang.OutOfMemoryError: PermGen
```

JDK 1.8 的时候，方法区（HotSpot 的永久代）被彻底移除了（JDK1.7 就已经开始了），取而代之是元空间，元空间使用的是本地内存。下面是一些常用参数：

```
-XX:MetaspaceSize=N //设置 Metaspace 的初始（和最小大小）  
-XX:MaxMetaspaceSize=N //设置 Metaspace 的最大大小
```

与永久代很大的不同就是，如果不指定大小的话，随着更多类的创建，虚拟机会耗尽所有可用的系统内存。

特点

1. 方法区是线程共享的，多个线程都用到一个类的时候，若这个类还未被加载，应该只有一个线程去加载类，其他线程等待；
2. 方法区的大小可以是非固定的，jvm可以根据应用需要动态调整，jvm也支持用户和程序指定方法区的初始大小；
3. 方法区有垃圾回收机制，一些类不再被使用则变为垃圾，需要进行垃圾清理。

九、Java虚拟机栈

[Java的JVM运行时栈结构和方法调用详解 - 掘金 \(juejin.cn\)](http://juejin.cn)

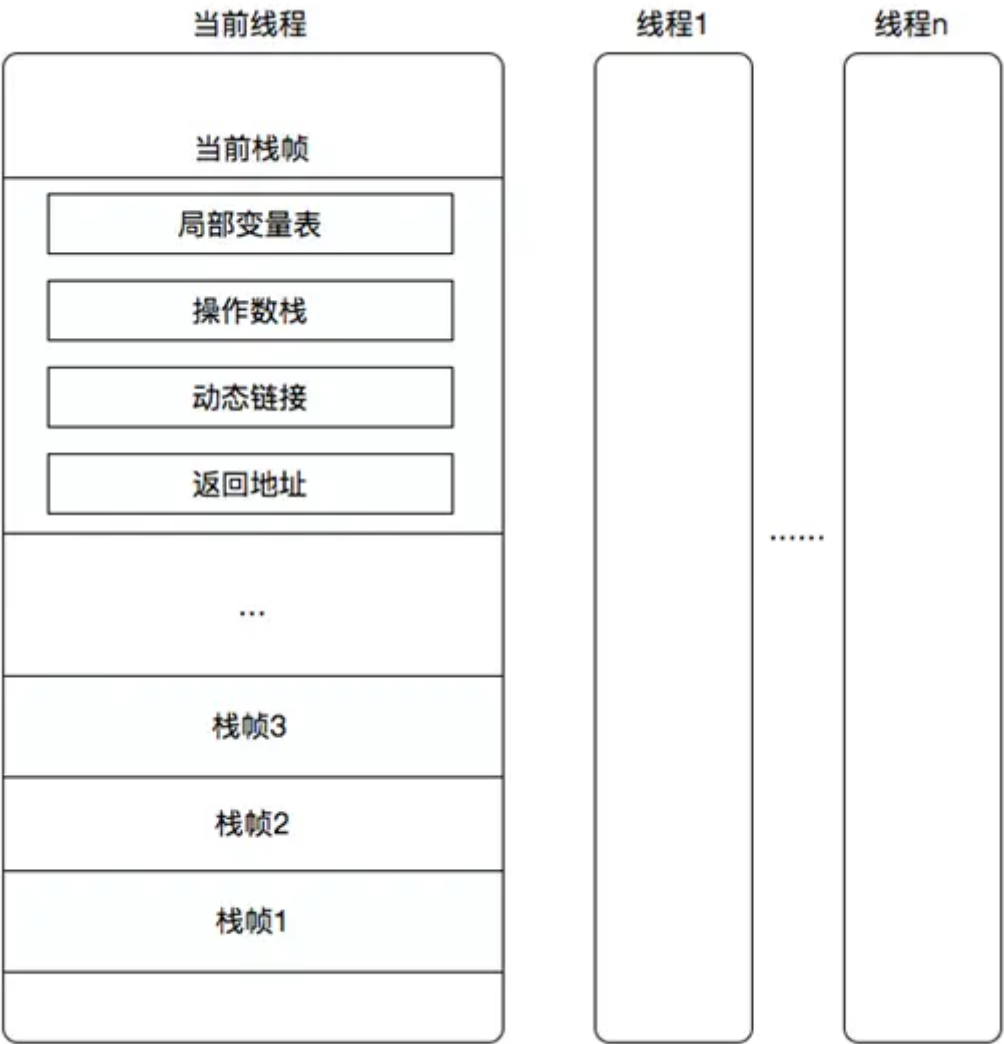
栈：是一种数据结构，先进后出、后进先出（桶） ⚡ 队列：先进先出、后进后出 FIFO，和程序计数器一样，Java虚拟机栈也是线程私有的。生命周期与线程相同。

除了一些native本地方法需要本地方法栈实现之外，其他所有的Java方法都需要通过Java虚拟机栈来实现。方法调用的数据需要通过栈进行传递，每一个方法的调用都会有一个栈帧被压入栈中，每一个方法结束，则会弹出对应栈帧。

栈由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法返回地址。和数据结构上的栈类似，两者都是先进后出的数据结构，只支持出栈和入栈两种操作。

每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

在编译程序代码时，栈帧需要最大多大的局部变量表、最深多深的操作数栈都已经完全确定，并写入方法表**Code**属性中，因此一个栈帧需要分配多少内存，不会受程序运行期数据的影响。



局部变量表

局部变量表是一组变量值的存储空间，用于存放方法参数和方法内部定义的局部变量。局部变量表中的变量只在当前方法调用中有效，当方法调用结束后，随着方法栈帧的销毁，局部变量表也会随之销毁。

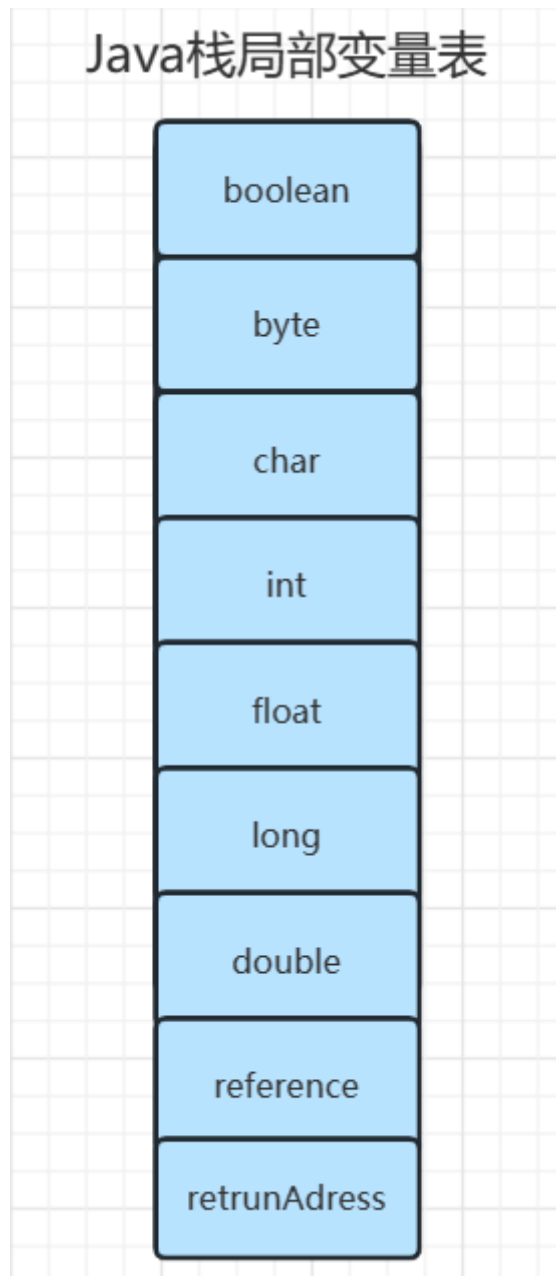
局部变量表的容量以变量槽（**Variable Slot**，下称 **Slot**）为最小单位，局部变量表中主要存放了编译期可知的八种基本数据类型（**boolean**、**byte**、**char**、**short**、**int**、**float**、**long**、**double**）、对象引用（**reference** 类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）以及 **returnAddress** 类型。

- 局部变量表，局部变量更少的方法的递归调用深度可以更深。
- 每一个局部变量都有自己的作用范围(作用字节码范围)，为了尽可能节省栈帧空间，局部变量表中的变量所在的**Slot**是可以重用的，重用slot可能会影响GC。



https://blog.csdn.net/weixin_41864000 @稀土掘金技术社区

Java栈局部变量表



操作数栈

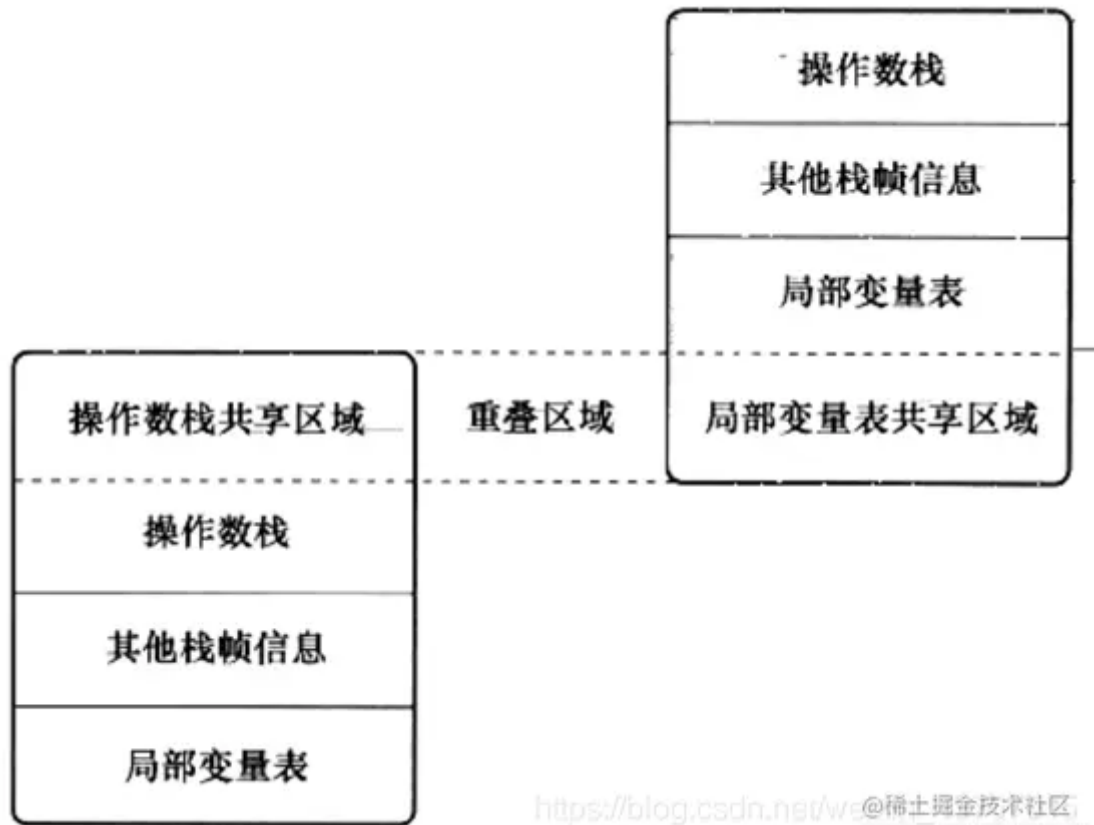
操作数栈 主要作为方法调用的中转站使用，用于存放方法执行过程中产生的中间计算结果。同时作为计算过程中变量临时的存储空间。

操作数栈的每一个元素可以是任意Java数据类型，32位的数据类型占一个栈容量，64位的数据类型占2个栈容量,且在方法执行的任意时刻，操作数栈的深度都不会超过`max_stacks`中设置的最大值。

当一个方法刚刚开始执行的时候，操作数栈是空的，在方法执行过程中，会有各种字节码指令往操作数栈中写入和提取内容，也就是出栈、入栈操作。操作数栈中的数据类型必须与字节码指令序列匹配，在编译程序代码时，编译时必须严格保证这一点，在类校验阶段的数据流分析中还要在此验证这一点。

虚拟机的执行引擎又被称为“基于栈的执行引擎”，其中的“栈”就是操作数栈。

两个栈帧作为虚拟机栈的元素，理论上是完全相互独立的，但在大多数虚拟机的实现里，都会做一些优化处理，令两个栈帧出现一部分重叠。让下面栈帧的操作数栈和上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用时可以共用一部分数据，无需进行额外的参数复制传递，重叠过程如下图所示：



动态链接

动态链接和方法返回地址也被统称为栈帧信息。

主要服务一个方法中调用另一个方法的情况。每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态链接

（**Dynamic Linking**）。Class 文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候就转化为直接引用，这种转化称为静态解析。另一部分在每次运行期间转化为直接引用，这部分称为动态链接。

Java 方法有两种返回方式，一种是 **return** 语句正常返回，一种是抛出异常。不管哪种返回方式，都会导致栈帧被弹出。也就是说，栈帧随着方法调用而创建，随着方法结束而销毁。无论方法正常完成还是异常完成都算作方法结束。

程序运行中栈可能会出现两种错误：[Java 内存区域详解（重点） | JavaGuide\(Java面试+学习指南\)](#)

- **StackOverflowError**：若栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前 *Java* 虚拟机栈的最大深度的时候，就抛出 **StackOverflowError** 错误。
- **OutOfMemoryError**：如果栈的内存大小可以动态扩展，如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出 **OutOfMemoryError** 异常。

但是 *hotspot* 虚拟机中出现 OOM 可能是因为线程申请栈空间失败，因为 *HotSpot* 的栈容量不支持动态扩展。

方法返回地址

[Java的JVM运行时栈结构和方法调用详解 - 掘金 \(juejin.cn\)](#)

一个方法开始执行后，只有两种方式可以退出这个方法：

1. 当执行遇到返回指令，会将返回值传递给上层的方法调用者，这种退出的方式称为正常完成出口（**Normal Method Invocation Completion**），一般来说，调用者的 **PC** 计数器可以作为返回地址。
2. 当执行遇到异常，并且当前方法体内没有得到处理，就会导致方法退出，此时是没有返回值的，称为异常完成出口（**Abrupt Method Invocation Completion**），返回地址要通过异常表来确定。

方法退出时，需要返回到方法被调用的位置，程序才能继续执行。方法返回时可能需要在栈帧中保存一些信息，用来恢复它的上层方法的执行状态。一般来说，调用者的 **PC** 计数器的值可以作为返回地址，栈帧中很可能会保存这个计数器值；而方法异常退出时，返回地址是要通过异常器表来确定的，栈帧中一般不会保存这部分信息。

无论是哪种方法退出，本质上都等同于把当前栈帧出栈，因此退出时可能执行的操作有：

1. 恢复上层方法的局部变量表和操作数栈。
2. 把返回值（如果有）压入调用者栈帧的操作数栈。
3. 调整 **PC** 计数器的值以指向方法调用指令后面的一条指令。（呼应程序计数器的作用）

方法结束的两种方式：

[\(59条消息\) JVM8: Java虚拟机栈——方法返回地址（Return Address）_库隐的博客-CSDN博客](#)

异常退出：

通过异常退出的，返回值是要通过异常表来确定，栈帧种的一般不会保存这部分的信息。

- 在方法执行过程中遇到了异常（*Exception*），并且这个异常没有在方法内进行处理，也就是只有在本方法的异常中没有搜索到匹配的异常处理器，就会导致方法退出，称为异常完成出口
- 在方法执行过程中抛出异常时的异常处理，存储在一个异常处理表，方便再发生异常的时候找到处理异常的代码。

 正常完成出口和异常出口区别在于：通过异常出口退出的不会给它的上层调用者产生任何的返回值。

十、本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是：虚拟机栈为虚拟机执行 **Java** 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 **Native** 方法服务。在 HotSpot 虚拟机中和 **Java** 虚拟机栈合二为一。

本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 `StackOverflowError` 和 `OutOfMemoryError` 两种错误。

十一、堆(Heap) ❤️❤️❤️❤️❤️

定义及目的

Java 虚拟机所管理的内存中最大的一块，**Java** 堆是所有线程共享的一块内存区域(也就是一个JVM只有一个堆)，在虚拟机启动时创建。此内存区域的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在这里分配内存。

Java堆既可以是固定大小的，也可以是可扩展的（通过参数-Xmx和-Xms设定），如果堆无法扩展或者无法分配内存时也会报OOM。

之所以是“几乎所有对象”的原因是因为在`jdk1.7`之后虚拟机加入了逃逸分析优化技术，使得一部分无法逃逸出去的对象可以直接在栈中分配空间，而不需要在堆中分配。具体可以参考：

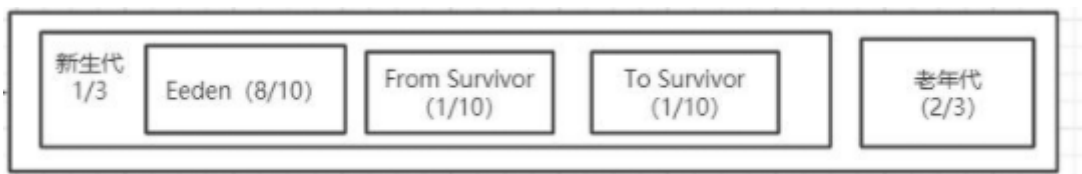
每日一面试题-什么是逃逸分析？ - 知乎 ([zhihu.com](https://www.zhihu.com))

逃逸分析好处：

- 栈上分配空间，减轻GC压力
- 同步消除，消除掉只有一个线程在访问的方法上的同步锁
- 可以进行标量替换，将对象分解为多个标量，进行分析和优化，可以自由地分配空间，无需为对象分配整体空间。（标量可以理解为基本数据类型这些不可分解的数据，如果一个数据可以分解，成为聚合量）

Java 堆是垃圾收集器管理的主要区域，因此也被称作 **GC 堆**（**Garbage Collected Heap**）。从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代；再细致一点有：Eden、Survivor、Old 等空间。进一步划分的目的是更好地回收内存，或者更快地分配内存。

其中空间大小为，新生代:老年代=1:2 Eden:Survivor From:Survivor TO=8:1:1



`jdk1.7`及以前，永久代也处于堆中，`1.7`之后就位于本地内存中了。

GC 垃圾回收，主要是在伊甸园区和老年区

堆中存储内容



- 对象实例

- 类初始化生成的对象
- 基本数据类型的数组也是对象实例
- 字符串常量池
 - 字符串常量池原本存放于方法区，jdk7开始放置于堆中。
 - 字符串常量池存储的是string对象的直接引用，而不是直接存放的对象，是一张string table
- 静态变量
 - 静态变量是有static修饰的变量，jdk7时从方法区迁移至堆中
- 线程分配缓冲区（Thread Local Allocation Buffer）
 - 线程私有，但是不影响java堆的共性
 - 增加线程分配缓冲区是为了提升对象分配时的效率

堆中对象空间分配

大部分情况，对象都会首先在 **Eden** 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 **S0** 或者 **S1**，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

对象晋升到老年代的年龄阈值也可以通过动态调节：

“*Hotspot* 遍历所有对象时，按照年龄从小到大对其所占用的大小进行累积，当累积的某个年龄大小超过了 *survivor* 区的一半时，取这个年龄和 *MaxTenuringThreshold* 中更小的一个值，作为新的晋升年龄阈值”。

也就是说 $tenuringThreshold = \min[TenuringThreshold, \text{if}(\sum age > desired_survivor_size) \rightarrow age]$;

堆中常见错误

- `java.lang.OutOfMemoryError: GC Overhead Limit Exceeded`：当 JVM 花太多时间执行垃圾回收并且只能回收很少的堆空间时，就会发生此错误。
- `java.lang.OutOfMemoryError: Java heap space`：假如在创建新的对象时，堆内存中的空间不足以存放新创建的对象，就会引发此错误。(和配置的最大堆内存有关，且受制于物理内存大小。最大堆内存可通过 `-Xmx` 参数配置，若没有特别配置，将会使用默认配置。

[Java 内存区域详解（重点） | JavaGuide\(Java面试+学习指南\)](#)

十二、直接内存

直接内存是一种特殊的内存缓冲区，并不在 Java 堆或方法区中分配的，而是通过 JNI 的方式在本地内存上分配的。

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，但是这部分内存也被频繁地使用。而且也可能导致 `OutOfMemoryError` 错误出现。

详情见[Java 内存区域详解（重点） | JavaGuide\(Java面试+学习指南\)](#)

总结🔑（待重新整理）

名称	作用	异常
程序计数器	线程记录运行到哪里的	永不异常
Java虚拟机栈	线程记录方法调用	栈不够且不能扩展StackOverflowError, 可扩展内存不够OutOfMemoryError
本地方法栈	为Native方法服务	栈不够且不能扩展StackOverflowError, 可扩展内存不够OutOfMemoryError
Java堆	存对象的	没有内存完成实例分配，堆无法再扩展，OutOfMemoryError
方法区	存类信息，常量等不变的部分	内存不足抛出OutOfMemoryError
直接内存	NIO调用native方法使用	内存不足抛出OutOfMemoryError

十三、GC垃圾回收

新生代

对象诞生和成长的地方，甚至凋亡（在新生区也有可能直接被垃圾回收清理掉，没有被清理掉的才会进入老年区）

🍌Eden区：所有的对象都是在伊甸园区new出来的，新对象或者生命周期很短的对象会存储在这个区域中

👉 **Survivor区**：又分为**From区**和**To区**，这两块区域可以相互转化，并且这两块区域必须有一块是空着的，就是**To区域**

新生代:老年代=1:2 **Eden:Survivor From:Survivor TO=8:1:1**

*HotSpot JVM*把年轻代分为了三部分：

👉 1个**Eden**区和2个**Survivor**区（分别叫**from**和**to**）。默认比例为8（**Eden**）：1（一个**survivor**）：1，

👉 一般情况下，新创建的对象都会被分配到**Eden**区(一些大对象特殊处理)。当 **Eden** 区没有足够空间进行分配时，虚拟机将发起一次 **Minor GC**。这些对象经过第一次 **Minor GC** 后，如果仍然存活，将会被移到**Survivor**区。对象在**Survivor**区中每熬过一次**Minor GC**，年龄就会增加1岁，当它的年龄增加到一定程度时，就会被移动到老年代中。默认晋升年龄并不都是 **15**，这个是要区分垃圾收集器的，**CMS** 就是 **6**，关于默认的晋升年龄是 **15**，这个说法的来源大部分都是《深入理解 **Java** 虚拟机》这本书

👉 如果在**GC**期间虚拟机发现对象无法存入**Survivor**空间的时候，就会通过分配担保机制把新生代的对象提前转移到老年代中，如果老年代空间不足，则会执行**Full GC**。

👉 大对象会直接被放入老年代：大对象就是需要大量连续内存空间的对象（比如：字符串、数组）。大对象直接进入老年代主要是为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

👉 因为年轻代中的对象基本都是朝生夕死的(80%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

👉 在**GC**开始的时候，对象只会存在于**Eden**区和名为“**From**”的**Survivor**区，**Survivor**区“**To**”是空的。紧接着进行**GC**，**Eden**区中所有存活的对象都会被复制到“**To**”中，而在“**From**”区中，经过这次**GC**仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过**XX:MaxTenuringThreshold**来设置)的对象会被移动到老年代中，没有达到阈值的对象会被复制到“**To**”区域。经过这次**GC**后，**Eden**区和**From**区已经被清空。这个时候，“**From**”和“**To**”会交换他们的角色，也就是新的“**To**”就是上次**GC**前的“**From**”，新的“**From**”就是上次**GC**前的“**To**”。不管怎样，都会保证名为**To**的**Survivor**区域是空的。**Minor GC**会一直重复这样的过程，直到“**To**”区被填满，“**To**”区被填满之后，会将所有对象移动到老年代中。（**Minor GC**后有交换，谁空谁是**To**区）

🤔为什么要设置两个Survivor区？设置两个Survivor区最大的好处就是解决了碎片化；假设现在只有一个survivor区，我们来模拟一下流程：刚刚新建的对象在Eden中，一旦Eden满了，触发一次Minor GC，Eden中的存活对象就会被移动到Survivor区。这样继续循环下去，下一次Eden满了的时候，问题来了，此时进行Minor GC，Eden和Survivor各有一些存活对象，如果此时把Eden区的存活对象硬放到Survivor区，很明显这两部分对象所占有的内存是不连续的，也就导致了内存碎片化。碎片化带来的风险是极大的，严重影响Java程序的性能。堆空间被散布的对象占据不连续的内存，最直接的结果就是，堆中没有足够大的连续内存空间，接下去如果程序需要给一个内存需求很大的对象分配内存。。。画面太美不敢看。。。这就好比我们上学时背包里所有东西紧挨着放，最后就可能省出一块完整的空间放饭盒。如果每件东西之间隔一点空隙乱放，很可能最后就要手提一路了。

老年代

那些在经历了Eden区和Survivor区的多次GC后仍能存活下来的对象会存储在这个区里。这个区会由一个特殊的垃圾回收器来负责，老年代中的对象的回收都是由老年代的GC（major GC）来进行的。

❓满足什么条件时，从新生代转移到老年代？

💡幸存区对象年龄达到年龄阈值(默认是15，不过不同垃圾收集器默认值可能不同，CMS是6，可以通过参数调节)，则进入老年区。 `XX:MaxTenuringThreshold`

💡大对象，当某个对象分配需要大量的连续内存时（比如：字符串、数组），此时对象的创建不会分配在Eden区，会直接分配在老年代，因为如果把大对象分配在Eden区，Minor GC后再移动到From区，To区会有很大的开销（对象比较大，复制会比较慢，也占空间），也很快会占满From区，To区，所以干脆就直接移到老年代。大对象直接进入老年代主要是为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

💡在From区（或To区）相同年龄的对象大小之和大于From区（或To区）空间一半以上时，则年龄大于等于该年龄的对象也会晋升到老年代。（也就是动态调节了年龄阈值，某个年龄age的累计空间大小大于了Survivor区大小的一半，则将最大阈值更改为age和Tenuring Threshold中小的那个。）

空间分配担保机制

空间分配担保是为了确保在Minor GC之前老年代本身还有容纳新生代所有对象的剩余空间。

《深入理解Java虚拟机》第三章对于空间分配担保的描述如下：

JDK 6 Update 24 之前，在发生 *Minor GC* 之前，虚拟机必须先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那这一次 *Minor GC* 可以确保是安全的。如果不成立，则虚拟机会先查看

`XX:HandlePromotionFailure` 参数的设置值是否允许担保失败(*Handle Promotion Failure*);如果允许，那会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行一次 *Minor GC*，尽管这次 *Minor GC* 是有风险的;如果小于，或者 `-XX: HandlePromotionFailure` 设置不允许冒险，那这时就要改为进行一次 *Full GC*。

JDK 6 Update 24 之后的规则变为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 GC*

GC分类

针对 HotSpot VM 的实现，它里面的 GC 其实准确分类只有两大类：

部分收集 (Partial GC)：

- 新生代收集 (Minor GC / Young GC)：只对新生代进行垃圾收集；
- 老年代收集 (Major GC / Old GC)：只对老年代进行垃圾收集。需要注意的是 Major GC 在有的语境中也用于指代整堆收集；
- 混合收集 (Mixed GC)：对整个新生代和部分老年代进行垃圾收集。

整堆收集 (Full GC)：收集整个 Java 堆和方法区。

GC引发条件：

Minor GC :发生在Eden区，当Eden区满的时候触发 Minor GC ， From区满的时候并不会触发 Minor GC ；

Full GC： 全面收集也被称为Full GC，是所有GC类型中，耗时最长、停顿最久的GC，Full GC会对于所有可发生 GC的区域进行全面回收，其中涵盖新生区、老年区以及元数据空间，而一般触发Full GC的原因有如下几种：

- ①调用System.gc()时，JVM在内存占用较多时会尝试发生FullGC，但并非100%触发。
- ②除CMS之外的收集器，当老年区内存不足时也会触发Full GC。
- ③元数据空间内存不足时，也会触发Full GC。

④对象晋升时，老年区空间无法承载晋升对象时也会触发Full GC。

⑤新生代空间分配担保机制触发时，也会先触发Full GC。

Mixed GC

混合收集Mixed GC是指收集范围覆盖整个新生代空间及部分年老代空间的GC，但目前只有G1存在该行为，其他收集器均不支持，因为G1收集器是逻辑分代，物理分区结构，所以可以针对不同的分区进行单独的收集，在发生GC时，可以选取新生代分区+部分年老代分区进行回收。

STW和安全点

Minor GC 会引发 STW（Stop The World），STW 是指 GC 事件发生的过程中，会产生应用程序的停顿，停顿产生时整个应用程序线程都会被暂停，没有任何响应，这个停顿称为 STW；在这个阶段，全局停顿，所有 Java 代码停止，Native 代码可以执行，但不能与 JVM 交互；这些现象多半是由于 GC 引起

安全点

[OopMap理论篇 - 知乎 \(zhihu.com\)](#)

在 OopMap的协助下，HotSpot可以快速且准确地完成GC Roots枚举，但是一个很现实的问题随之而来：可能导致引用关系变化，或者说OopMap内容变化的指令非常多，如果为每一条指令都生成对应的OopMap，那将会需要大量的额外空间，这样GC的空间成本将会变得很高。

实际上,HotSpot也的确没有为每条指令都生成OopMap，前面已经提到，只是在“特定的位置”记录了这些信息，这些位置称为安全点（Safepoint），即程序执行时并非在所有地方都停顿下来开始GC，只有在到达安全点时才能暂停。Safepoint的选定既不能太少以至于让GC等待时间太长，也不能过于频繁以致于过分增加运行时的负荷。所以安全点的选定基本上是以程序“是否具有让程序长时间执行的特征”为标准进行选定的----因为每条指令执行的时间都非常短暂，程序不太可能因为指令流长度太长这个原因而过长时间运行，“长时间执行”的最明显特征就是指令序列复用，例如

- 方法调用
- 循环跳转
- 异常跳转等，所以具有这些功能的指令才会产生Safepoint。

对于Safepoint，另一种需要考虑到问题是如何在GC发生时让所有线程（这里不包括执行JNI调用的线程）都“跑”到最近的安全点上再停顿下来。这里有两种方案可供选择：抢先式中断（**Preemptive Suspension**）和主动式中断（**Voluntary Suspension**），其中抢断式中断不需要线程的执行代码主动配合，在GC发生时，首先把所有线程全部中断，如果发现线程在中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上，现在几乎没有虚拟机采用抢先式中断来暂停线程从而响应GC事件。

而主动式中断的思想是当GC需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起。轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配内存的地方。

安全区域

使用Safepoint似乎完美地解决了如何进入GC的问题，但实际情况却并不一定。Safepoint机制保证了程序执行时，在不太长的时间内就会遇到可进入GC的Safepoint。但是程序“不执行”的时候呢？所谓的程序不执行就是没有分配CPU时间，典型的例子就是线程处于Sleep状态或者Blocked状态，这时候线程无法响应JVM的中断请求，“走”到安全的地方去中断挂起，JVM也显然不太可能等待线程重新被分配CPU时间。对于这种情况，就需要安全区域（Safe Region）来解决。

安全区域是指一段代码片段之中，引用关系不会发生变化。在这个区域中的任意地方开始GC都是安全的。我们把Safe Region看做是被拓展的SafePoint。

在线程执行到Safe Region中的代码时，首先标识自己已经进入了Safe Region，那样，当在这段时间里JVM发起GC时，就不用管便是自己为Safe Region状态的线程了。在线程将要离开Safe Region时，它要检查系统是否已经完成了根节点枚举（或者整个GC过程），如果完成了，那线程就继续执行，否则就必须等待，直到收到可以安全离开Safe Region的信号为止。

死亡对象判断方法

要回收垃圾首先要判断堆中哪些对象是死亡对象（即不能再被任何途径使用的对象）

引用计数法

给对象中添加一个引用计数器：

- 每当有一个地方引用它，计数器就加 1；
- 当引用失效，计数器就减 1；

- 任何时候计数器为 0 的对象就是不可能再被使用的。

? 为什么有引用计数法还要使用可达性分析法

- 「引用计数」对程序执行性能的影响比「可达性分析」还要大。这主要是由于「引用计数」要求所有的指针赋值都要改变计数值，多线程的情况下这个操作还要加锁。而「可达性分析」只有在用完内存的情况下才需要做，相对指针赋值而言这个频率几乎可以忽略不计，其他时候对程序执行的性能没有影响。
- 引用计数法虽然也可以使用*Recycle*算法解决循环依赖的问题，但是相比可达性分析法消耗更大。

可达性分析算法

[JVM 垃圾回收详解（重点） | JavaGuide\(Java面试+学习指南\)](#)

[引用计数法和可达性算法 - 腾讯云开发者社区-腾讯云 \(tencent.com\)](#)

这个算法的基本思想就是通过一系列的称为“**GC Roots**”的对象作为起点，从这些节点开始向下搜索，节点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。

下图中的object6~8虽然互相联系，但是GC root无法访问到，也就是说Object6~8是不可达的。下次垃圾回收时，可能会被回收。

哪些对象可以作为 **GC Roots** 呢？

- 虚拟机栈(栈帧中的本地变量表)中引用的对象
- 本地方法栈(Native 方法)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
- 所有被同步锁持有的对象

引用的两次标记过程

虽然对象不可达，但是也不是一定会被回收，一个对象真正被宣判死亡至少要经历两次标记过程。

🔧 第一次标记：如果对象在进行可达性分析后发现没有 GCRoots 相连接的引用链，那么它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 `finalize()` 方法。当对象没有覆盖 `finalize()` 方法，或者 `finalize()` 方法已经被虚拟机调用过，虚拟机将这两种情况都视为没有必要执行。如果对象被判定为有必要执行，则会被放到一个 F-Queue 队列。

🔧 第二次标记：被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就会被真的回收。`finalize()` 方法是对象跳脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中对象进行第二次小规模标记，如果对象要在 `finalize()` 中重新拯救自己：只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（`this` 关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移出即将回收的集合。

强引用、软引用、弱引用和虚引用

无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

JDK1.2之后，对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）4种。1.2之前，只有被引用和没有被引用两种状态。

强引用

阿里面试：说说强引用、软引用、弱引用、虚引用吧 - 腾讯云开发者社区-腾讯云 ([tencent.com](https://cloud.tencent.com/))

强引用：指在程序代码之中普遍存在的，类似 `Object obj = new Object()` 这类的引用，只要强引用还存在，垃圾收集器永远不会回收被引用的对象。当内存空间不足，Java 虚拟机宁愿抛出 `OutOfMemoryError` 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

软引用

软引用：用来描述一些还有用但并非必需的对象。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。

软引用可用来实现内存敏感的高速缓存。例如 Mybatis 缓存类 `SoftCache` 用到的软引用。

对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在JDK1.2之后，提供了SoftReference类来实现软引用

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。再通过判断引用队列是否来清除reference对象。[\(59条消息\) 软引用（SoftReference）和引用队列（ReferenceQueue）_weixin_33826609的博客-CSDN博客](#)

弱引用

弱引用：用来描述非必需对象，但是他的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK1.2之后，提供了WeakReference类来实现弱引用

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

虚引用

虚引用：也被称为幽灵引用或者幻影引用。它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。

如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。在JDK1.2之后，提供了PhantomReference类来实现虚引用。供对象被finalize之后，执行指定的逻辑的机制（cleaner）。

虚引用主要用来跟踪对象被垃圾回收的活动。

虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

判断废弃常量

假如在字符串常量池中存在字符串 "abc"，如果当前没有任何 **String** 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的話，"abc" 就会被系统清理出常量池了。

如何判断一个类是无用的类

[JVM 垃圾回收详解（重点） | JavaGuide\(Java面试+学习指南\)](#)

方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢？

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是“无用的类”：

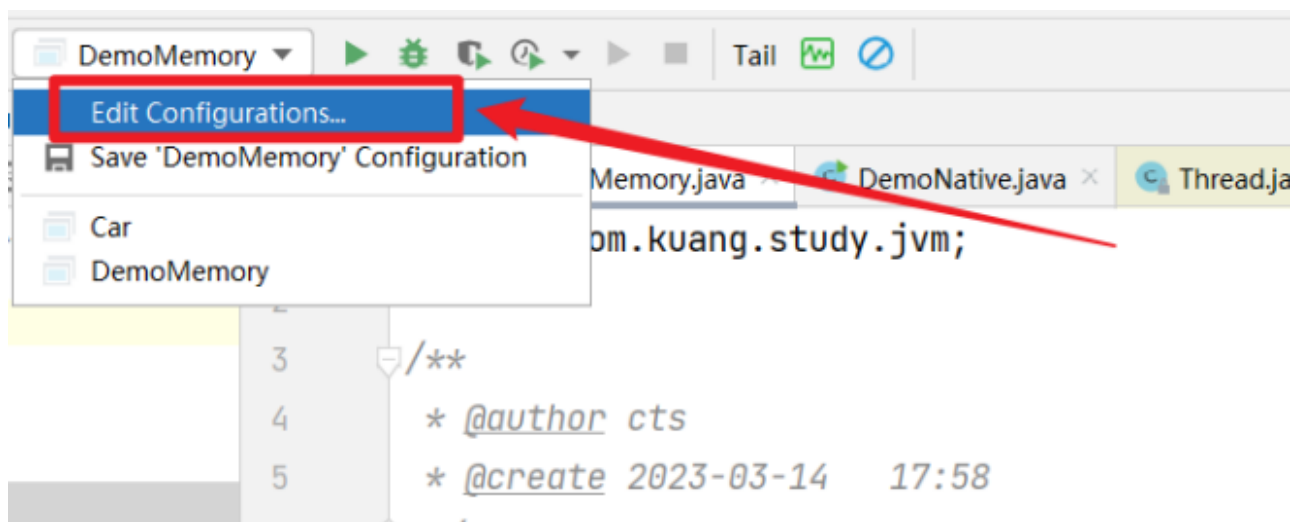
- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 `ClassLoader` 已经被回收。
- 该类对应的 `java.lang.Class` 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

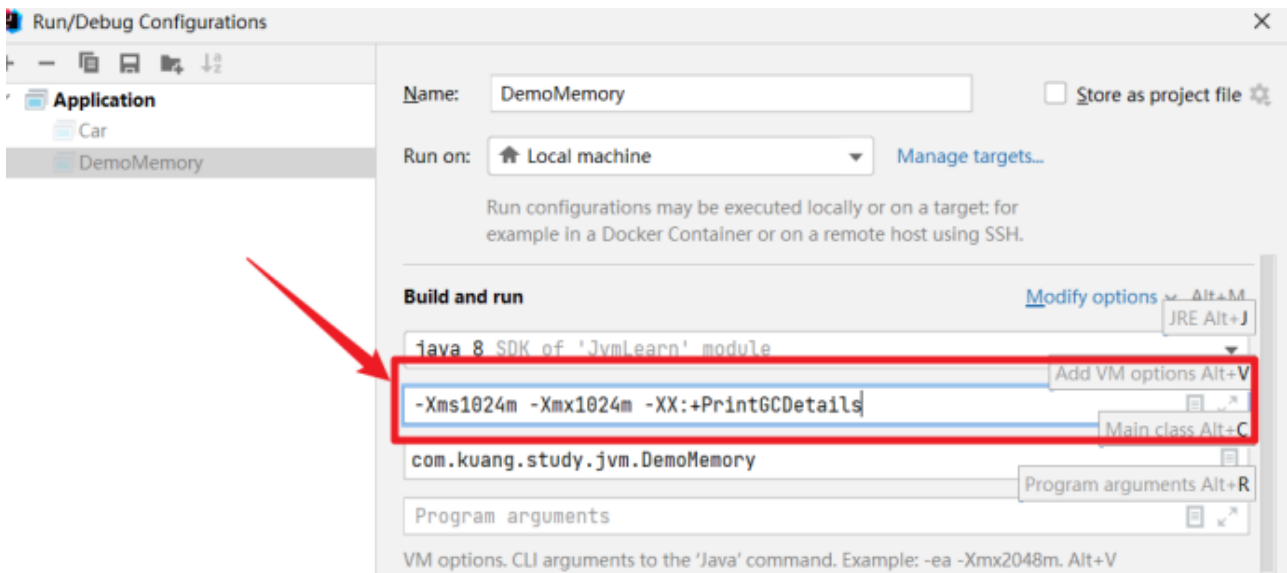
虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收

十四、堆内存调优

14.1 如果出现了OOM，该如何处理？

1. 尝试扩大堆内存，观察结果如何 运行程序之后出现 OOM，点击配置参数进行调整堆内存





另外，设置了参数后，会打印出详细的堆内存使用情况




$PSYoungGen = 305664K = 262144K + 43520K = \text{eden} + \text{from}$

From和To这两个区域同时只会使用一个，因此相当于有一个Survivor的空间是无法使用到的。

2.如果扩大了堆内存还是不行，就要考虑代码是否存在问题了，分析内存，看一下哪个地方出现了问题（专业工具）

分析出现 OOM 故障的方法：

 最快：能够看到代码第几行出错：内存快照分析工具 MAT、Jprofiler（MAT是基于 eclipse 开发的、Jprofiler是ej-technologies公司开发的）

 最慢：Debug，一行行分析代码（不适用于已上线的项目）

MAT、Jprofiler 的好处：

 分析Dump内存文件，快速定位内存泄漏；

✓ 获得堆中的数据;

✓ 获得大的对象;

✓ ...

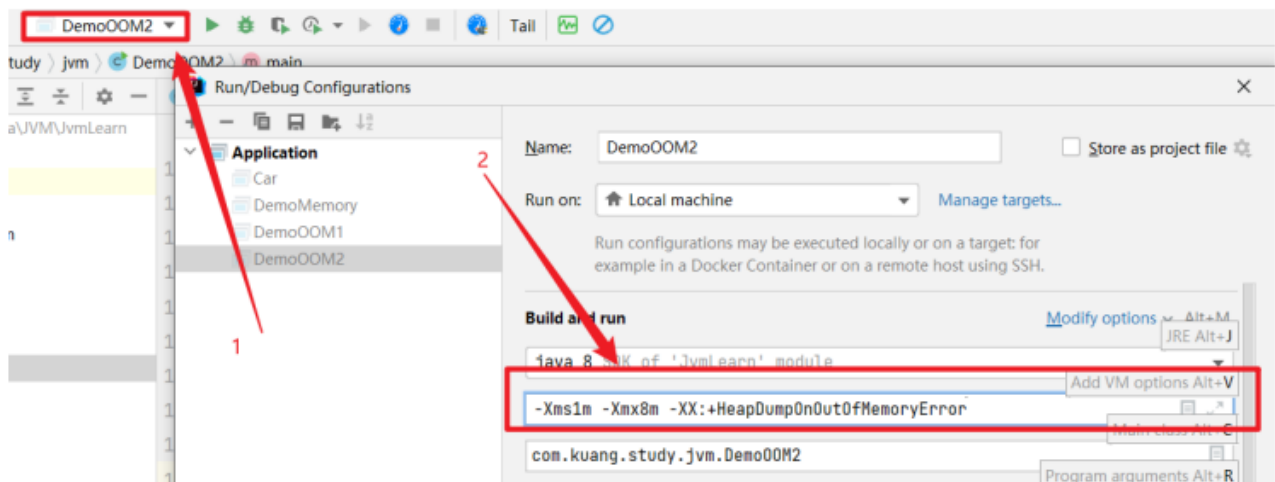
14.2 如何获取Dump文件

内存调优时, 需要获取 **Dump** 文件检查是哪块代码出现问题

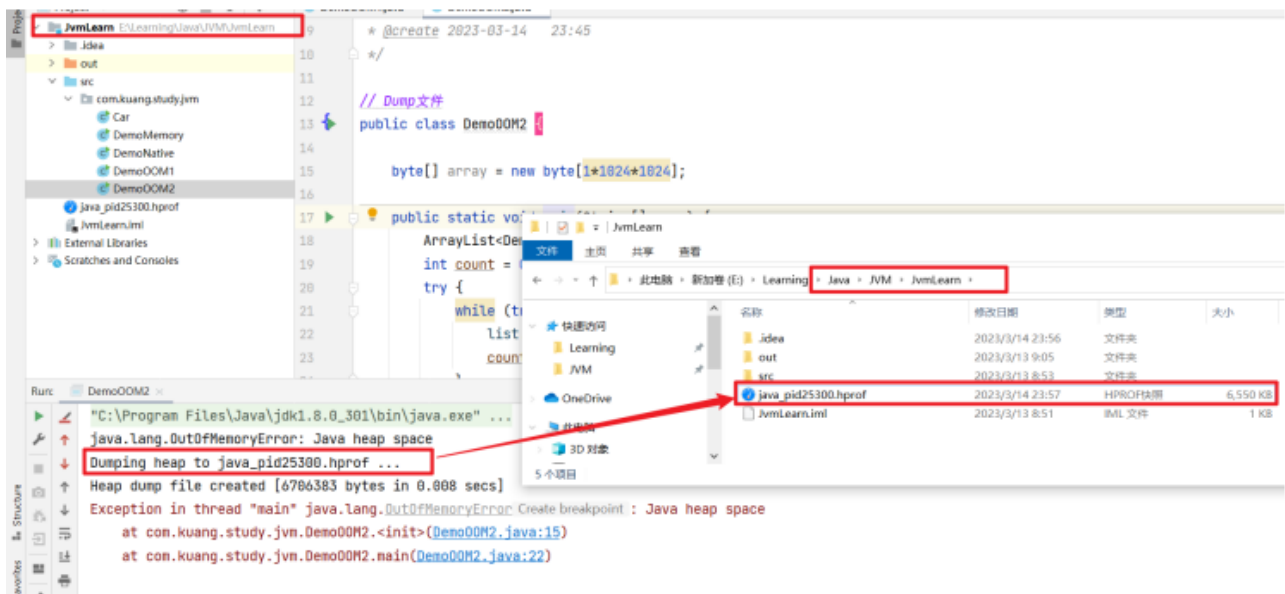
获取Dump文件需要配置程序的 VM options 参数: `-Xms1m -Xmx8m -`

`XX:+HeapDumpOnOutOfMemoryError`

-xms 设置初始化内存分配大小 默认是总内存的1/64 -Xmx 设置最大分配内存大小 默认是总内存的1/4
-XX:+PrintGCDetails 打印GC垃圾回收信息
-XX:+HeapDumpOnOutOfMemoryError -XX:+HeapDumpOn+错误名



然后运行程序就会生成Dump文件, 文件就在java项目的根文件夹里面



然后双击打开该 .hprof 文件即可

十五、GC垃圾回收算法

标记-清除算法（Mark-Sweep）

该算法分为“标记”和“清除”阶段：首先标记出所有不需要回收的对象，在标记完成后统一回收掉所有没有被标记的对象。它是最基础的收集算法，后续的算法都是对其不足进行改进得到。这种垃圾收集算法会带来两个明显的问题：

1. 效率问题
2. 空间问题（标记清除后会产生大量不连续的碎片）



标记-复制算法（新生代）

为了解决效率问题，“标记-复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

标记-整理算法

根据老年代的特点提出的一种标记算法，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回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

分代收集算法

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将 **java** 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

为什么分成新生代和老年代？

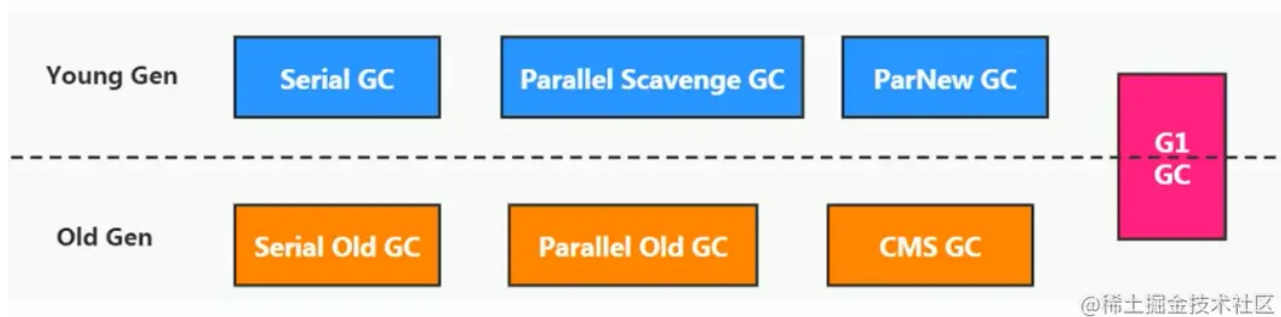
比如在新生代中，每次收集都会有大量对象死去，所以可以选择“标记-复制”算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

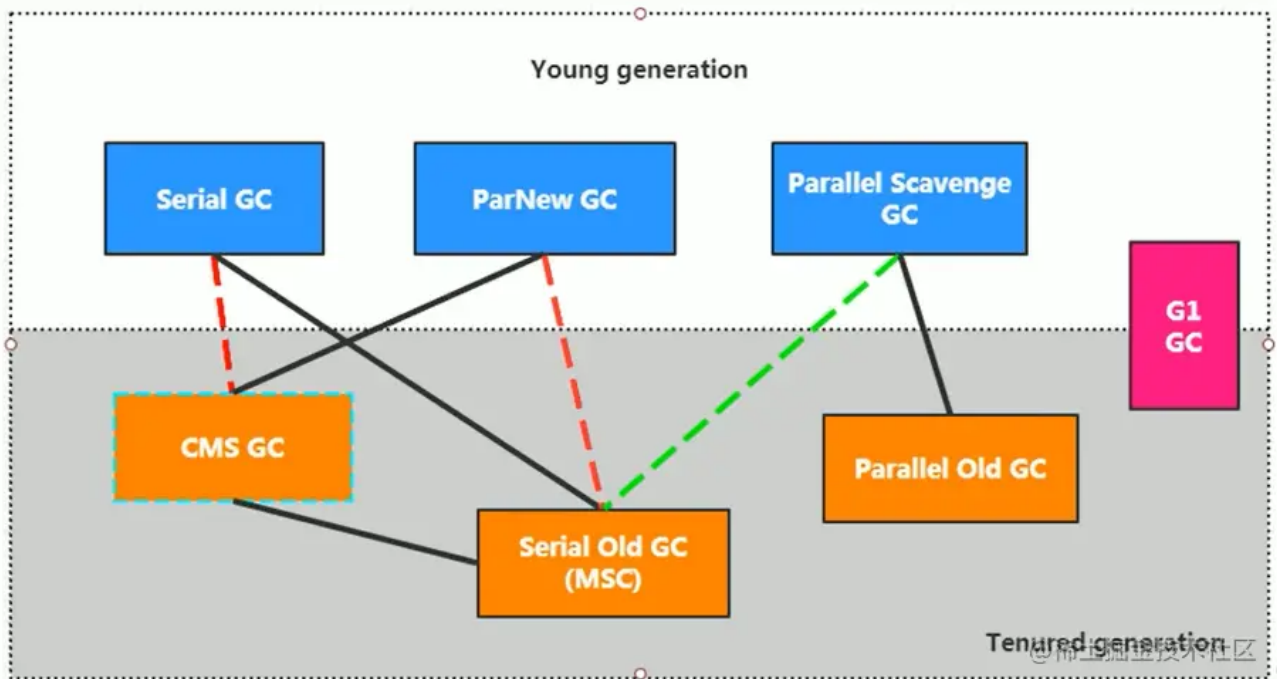
十六、GC垃圾收集器

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

虽然我们对各个收集器进行比较，但并非要挑选出一个最好的收集器。因为直到现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，我们能做的就是根据具体应用场景选择适合自己的垃圾收集器。试想一下：如果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的 **HotSpot** 虚拟机就不会实现那么多不同的垃圾收集器了

没有最优的收集器，只有最合适的





新生代收集器：Serial、ParNew、Parallel Scavenge；

老年代收集器：Serial old、Parallel old、CMS；

整堆收集器：G1；

垃圾回收器 - 掘金 (juejin.cn)

GC回收算法及回收器实现 - 掘金 (juejin.cn)

G1垃圾收集器详解 - 掘金 (juejin.cn)

JVM 垃圾回收详解（重点） | JavaGuide(Java面试+学习指南)

Serial 收集器（串行回收）

Serial收集器是最基本、历史最悠久的垃圾收集器了。JDK1.3之前回收新生代唯一的选择。它的“单线程”的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（**"Stop The World"**），直到它收集结束。

Serial收集器作为HotSpot中client模式下的默认新生代垃圾收集器。Client模式下默认开启，其他模式默认关闭

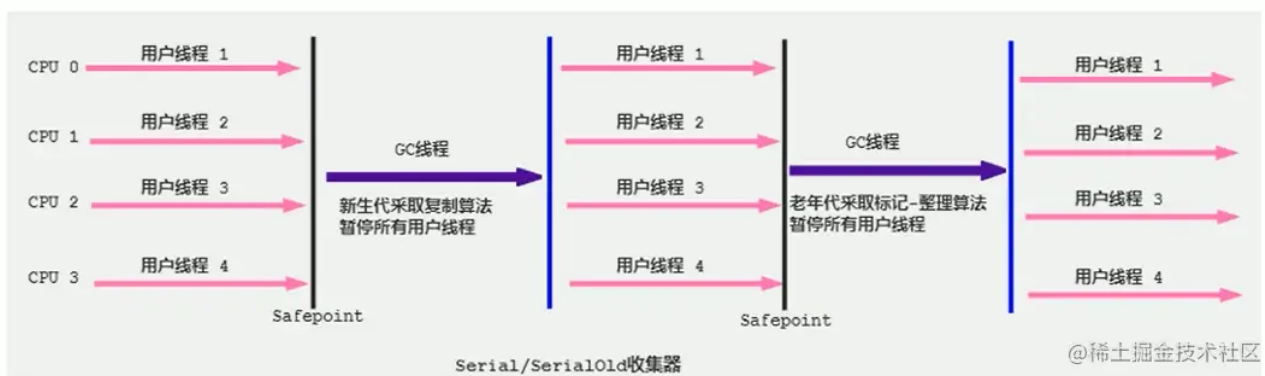
Serial收集器采用复制算法、串行回收和**"stop-the-World"**机制的方式执行内存回收。

特点

- 针对新生代
- 采用复制算法
- 单线程收集
- 会出现GC停顿
- Stop The World
- 对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率，简单高效(对比其他收集器)。

应用场景

- 依然是HotSpot在 Client 模式下默认的新生代收集器



调用参数	GC 组合
<code>-XX: +UseSerialGC</code>	Serial(年轻代)+Serial Old(老年代)

Serial Old收集器

除了年轻代之外，Serial收集器还提供用于执行老年代垃圾收集的Serial old收集器。Serial old收集器同样也采用了串行回收和"stop the World"机制，只不过内存回收算法使用的是标记-压缩算法。

特点

- 针对老年代
- 采用标记-整理算法
- 单线程收集
- Stop The World

应用场景

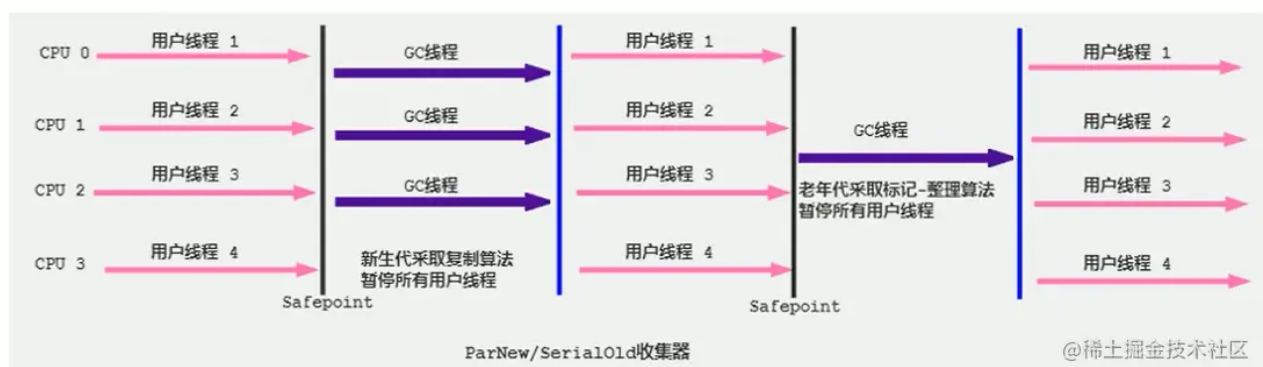
- Serial old是运行在Client模式下默认的老年代的垃圾回收器
- Serial Old在Server模式下主要有两个用途：
 - JDK1.5之前与新生代的Parallel scavenge配合使用（1.6的时候有Parallel Old搭配）
 - 作为老年代CMS收集器的后备垃圾收集方案

调用参数	GC 组合
<code>-XX: +UseSerialOldGC</code>	废弃，JVM不再支持

ParNew收集器（并行回收）

*Par*是*Parallel*的缩写，*New*：只能处理的是新生代

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。



特点

- 针对新生代
- 采用标记-复制算法
- 多线程收集
- 会出现GC停顿
- Stop The World

应用场景

- ParNew 是很多JVM运行在Server模式下新生代的默认垃圾收集器。因为除Serial外，目前只有ParNew GC能与CMS收集器配合工作
- 在单CPU环境中，不会比Serial收集器有更好的效果，因为存在线程交叉开销

调用参数	GC 组合
<code>-XX: +ParNewGC</code>	<code>ParNew+Serial old</code> 指定开启新生代为ParNew不影响老年代
<code>-XX: +UseConcMarkSweepGC</code>	<code>ParNew+CMS</code> 指定使用CMS后会默认使用ParNew作为新生代收集器

Parallel Scavenge收集器(吞吐量优先)

Parallel Scavenge 收集器也是一个并行多线程新生代收集器，它的关注点是吞吐量（高效率的利用 **CPU**）。**CMS** 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是 **CPU** 中用于运行用户代码的时间与 **CPU** 总消耗时间的比值。**Parallel Scavenge** 收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器运作不太了解，手工优化存在困难的时候，使用 **Parallel Scavenge** 收集器配合自适应调节策略，把内存管理优化交给虚拟机去完成也是一个不错的选择。

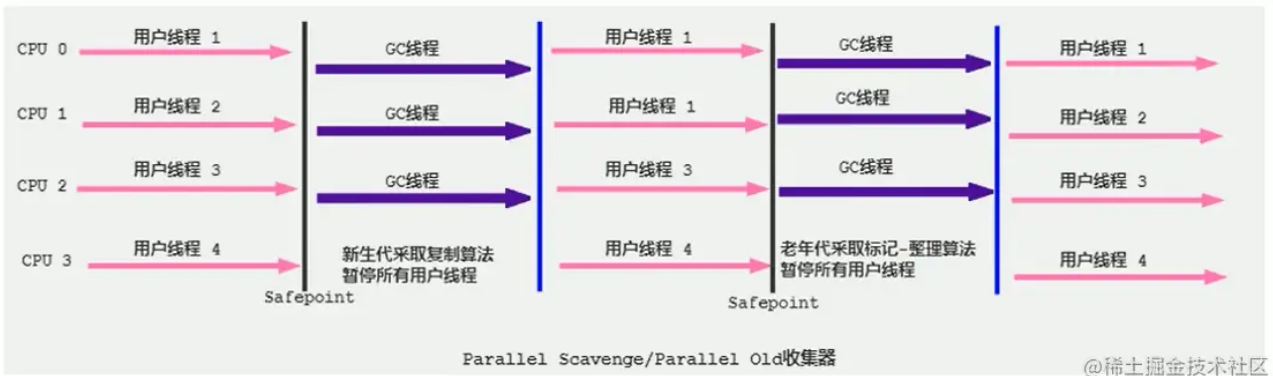
特点

- 针对新生代
- 采用标记复制算法
- 多线程收集
- 高吞吐量为目标
- Stop The World

- 新生代和老年代都使用并行收集器。打印出的GC会带PSYoungGen、ParOldGen关键字

应用场景

- 对暂停时间没有特别高的要求时，即程序主要在后台进行计算，不需要与用户进行太多交互，例如：执行批量处理、订单处理、工资支付、科学计算等应用程序
- 这是 **JDK1.8** 默认收集器，JDK8默认使用的是 Parallel Scavenge + Parallel Old



调用参数	GC 组合
<code>-XX:+UseParallelGC</code>	Parallel Scavenge(年轻代) + Parallel Old(老年代)
<code>-XX: +UseParallelOldGC</code>	Parallel Scavenge(年轻代) + Parallel Old(老年代)
这两个命令指定了其中任一个都会自动激活另一个	

Parallel Old 收集器

Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。在注重吞吐量以及 CPU 资源的场合，都可以优先考虑 Parallel Scavenge 收集器和 Parallel Old 收集器

特点

- 针对老年代
- 采用 标记-整理 算法
- 多线程收集
- Stop The World
- 新生代和老年代都使用并行收集器。打印出的GC会带PSYoungGen、ParOldGen关键字

应用场景

- JDK1.6及之后用来替代老年代Serial Old收集器
 - 特别是在Server模式，多CPU的情况下
-

CMS 收集器(低延迟)

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。它非常符合在注重用户体验的应用上使用。

CMS（Concurrent Mark Sweep）收集器是 **HotSpot** 虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。

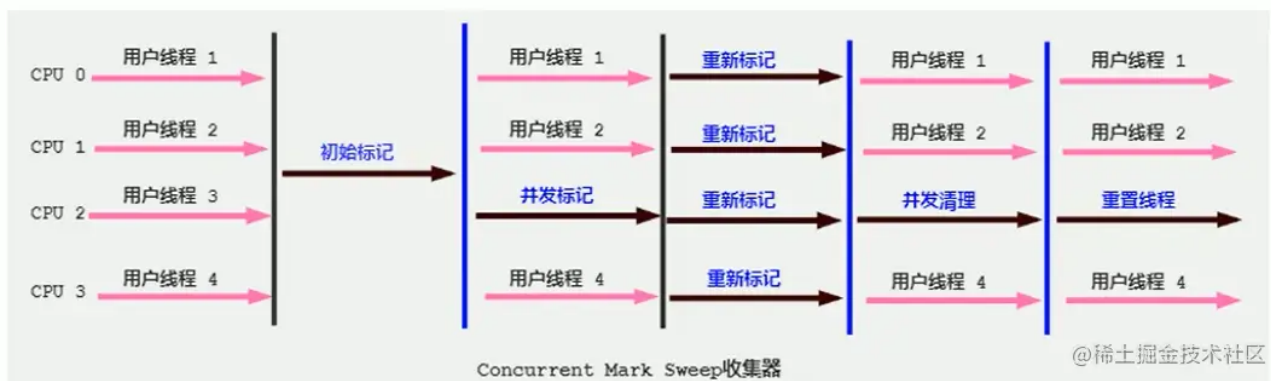
从名字中的**Mark Sweep**这两个词可以看出，CMS 收集器是一种“标记-清除”算法实现的，也会“stop-the-world”，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- 初始标记： 暂停所有的其他线程，并记录下直接与 root 相连的对象，速度很快；在这个阶段中，程序中所有的工作线程都将会因为“stop-the-world”机制而出现短暂的暂停。(STW)
- 并发标记： 同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。简单来说就是从Gc Roots的直接关联对象开始遍历整个对象图的过程，这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行。

解决该问题就是——针对老年代的对象，可以借助类*card table*的存储，将老年代对象发生变化所对应的卡页标记为*dirty*。

- 重新标记： 重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短(STW)
- 并发清除： 开启用户线程，同时 GC 线程开始对未标记的区域做清扫。

不幸的是，CMS作为老年代的收集器，却无法与JDK1.4.0中已经存在的新生代收集器Parallel Scavenge配合工作，所以在JDK1.5中使用CMS来收集老年代的时候，新生代只能选择ParNew或者Serial收集器中的一个。



由于最耗费时间的并发标记与并发清除阶段都不需要暂停工作，所以整体的回收是低停顿的。

特点

- 针对老年代
- 采用 标记-清除 算法
- 并发收集、低停顿
- 以获取最短回收停顿时间为目的

应用场景

- 与用户交互较多的场景
- 希望系统停顿时间较短，注重服务的响应速度

缺点

- 对CPU资源非常敏感
 - 并发收集虽然不会暂停用户线程，但因为占用一部分CPU资源，还是会导致应用程序变慢，总吞吐量降低
 - CMS默认收集线程数量 = $(ParallelGCThreads + 3) / 4$ 。
 - $ParallelGCThreads$ 是年轻代并行收集器的线程数量。当CPU数量小于8时， $ParallelGCThreads = CPU$ 数量，大于8时， $ParallelGCThreads = 3 + [5 * CPU Count] / 8$
- 无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败
 - 浮动垃圾（Floating Garbage）

- 在并发清除时，用户线程新产生的垃圾称为浮动垃圾；这使得并发清除时需要预留一定的内存空间，不能像其他收集器在老年代几乎填满再进行收集
- `-XX:CMSInitiatingOccupancyFraction` 设置老年代占用多少比例的时候触发CMS垃圾回收。JDK1.5默认68%，JDK1.6默认92%，也就是8%的CMS预留内存空间。
- **Concurrent Mode Failure**
 - 如果CMS预留内存空间无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败
 - 此时JVM启用后备预案：临时启用Serial Old收集器，而导致另一次Full GC的产生
- 产生大量内存碎片
 - 由于CMS基于“标记-清除”算法，清除后不进行压缩操作，因此产生大量不连续的内存碎片，有可能提前触发另一次Full GC动作
 - `-XX:+UseCMSCompactAtFullCollection` 使得CMS出现上面情况时不进行Full GC，而开启内存碎片的合并整理过程。默认开启
 - `-XX:+CMSFullGCsBeforeCompaction` 设置执行多少次不压缩的Full GC后，来一次压缩整理。默认0
 - 由于空间不再连续，CMS需要使用可用“空闲列表”内存分配方式

CMS无法做到消除STW

尽管CMS收集器采用的是并发回收（非独占式），但是在其初始化标记和再次标记这两个阶段中仍然需要执行“Stop-the-World”机制暂停程序中的工作线程，不过暂停时间并不会太长，因此可以说明目前所有的垃圾收集器都做不到完全不需要“stop-the-World”，只是尽可能地缩短暂停时间。

CMS Concurrent Mode Failure

由于在垃圾收集阶段用户线程没有中断，所以在CMS回收过程中，还应该确保应用程序用户线程有足够的内存可用。因此，CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，而是当堆内存使用率达到某一阈值时，便开始进行回收，以确保应用程序在CMS工作过程中依然有足够的空间支持应用程序运行。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

CMS内存分配方式

CMS收集器的垃圾收集算法采用的是标记清除算法，这意味着每次执行完内存回收后，由于被执行内存回收的无用对象所占用的内存空间极有可能是非连续的一些内存块，不可避免地将会产生一些内存碎片。那么CMS在为新对象分配内存空间时，将无法使用指针碰撞（Bump the Pointer）技术，而只能选择空闲列表（Free List）执行内存分配。

CMS为什么不使用标记整理算法？

答案其实很简答，因为当并发清除的时候，用Compact整理内存的话，原来的用户线程使用的内存还怎么用呢？要保证用户线程能继续执行，前提的它运行的资源不受影响嘛。Mark Compact更适合“stop the world” 这种场景下使用

指针碰撞

指针碰撞：假设Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞”（Bump the Pointer）。

空闲队列

空闲列表：假设Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没办法进行简单的指针碰撞了，虚拟机就必须维护一个列表，记录上哪块内存是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种方式称为“空闲列表”（Free List）

调用参数	GC 组合
<code>-XX:+UseConcMarkSweepGC</code>	<code>ParNew(年轻代) + CMS(老年代)</code>

G1收集器

垃圾回收器 - 掘金 (juejin.cn)

G1垃圾收集器详解 - 掘金 (juejin.cn)



为什么还需要G1

原因就在于应用程序所应对的业务越来越庞大、复杂，用户越来越多，没有GC就不能保证应用程序正常进行，而经常造成STW的GC又跟不上实际的需求，所以才会不断地尝试对GC进行优化。G1（Garbage-First）垃圾回收器是在Java7 update4之后引入的一个新的垃圾回收器，是当今收集器技术发展的最前沿成果之一。

- 官方给**G1**设定的目标是在延迟可控的情况下获得尽可能高的吞吐量,所以才担当起"全功能收集器"的重任与期望。

G1 (Garbage-First) 是一款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.

JDK7开始出现，JDK9的默认垃圾收集器。

G1工作流程

G1 收集器的运作大致分为以下几个步骤：

- 初始标记(Concurrent Marking)
 - 标记一下GC Roots能直接关联到的对象；
 - 修改TAMS（Next Top at Mark Start），让下一阶段并发运行时，用户程序能在正确可用的Region中创建新对象；
 - 需要“Stop The World”，速度很快
- 并发标记(Concurrent Marking)
 - 进行GC Roots 开始对堆中的对象进行可达性分析，找出存活的对象。
 - 上一步产生的集合中标记出存活对象，并不能保证可以标记出所有的存活对象；
 - 应用程序也在运行，在整个过程中耗时最长
- 最终标记(Final Marking)
 - 为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录；
 - 上一阶段对象的变化记录在线程的Remembered Set Log；这里把Remembered Set Log合并到Remembered Set中
 - 需要“Stop The World”，且停顿时间比初始标记阶段稍长，但远比并发标记的时间短；
 - 采用多线程并发执行来提升效率；

- 清除
 - 整理堆分区，识别高收益的老年代分区集合
 - STW
 - 识别空闲分区，即发现无存活对象的分区。该分区可在清除阶段直接回收，无需等待下次收集周期。
 - 在这个阶段只进行统计识别不进行真正的清理。
- 筛选回收(Live Data Counting And Evacuation)
 - 首先排序各个Region的回收价值和成本；
 - 根据用户期望的GC停顿时间来定制回收计划；
 - 按计划回收一些价值高的Region中垃圾对象；
 - STW

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 **Region**(这也就是它的名字 **Garbage-First** 的由来)。这种使用 **Region** 划分内存空间以及有优先级的区域回收方式，保证了 **G1** 收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）

ZGC收集器

新一代垃圾回收器—ZGC - 掘金 (juejin.cn)

新一代垃圾回收器ZGC的探索与实践 - 美团技术团队 (meituan.com)

是JDK 11中推出的一款追求极致低延迟的实验性质的垃圾收集器

ZGC中管理物理内存的基本单位是segment。segment默认与small page size一样，都是2MB。引入segment是为了避免频繁的申请和释放内存的系统调用，一次申请2MB，当segment空闲时，将加入空闲列表，等待之后重复使用。

ZGC为了能高效、灵活地管理内存，实现了两级内存管理：虚拟内存和物理内存，并且实现了物理内存和虚拟内存的映射关系。这和操作系统中虚拟地址和物理地址设计思路基本一致。

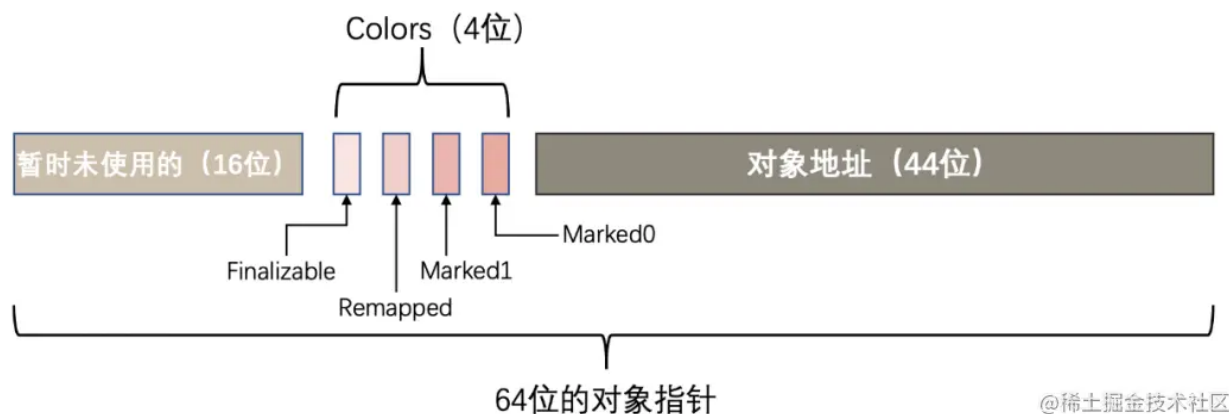
当应用程序创建对象时，首先在堆空间申请一个虚拟地址，ZGC同时会为该对象在Marked0、Marked1和Remapped三个视图空间分别申请一个虚拟地址，且这三个虚拟地址对应同一个物理地址。在ZGC中这三个空间在同一时间点有且仅有一个空间有效。

ZGC利用虚拟空间换时间。

染色指针

之前的垃圾收集器都是把GC信息（标记信息、GC分代年龄..）存在对象头的Mark Word里，而ZGC通过染色指针直接在对象信息中标注这个对象是个垃圾。

ZGC将对象信息存储在指针中，这种技术叫做——染色指针（Colored Pointer）。



在64位的机器中，对象指针是64位的。

ZGC使用64位地址空间的第0~43位存储对象地址， $2^{44} = 16\text{TB}$ ，所以ZGC最大支持16TB的堆。

而第44~47位作为颜色标志位，Marked0、Marked1和Remapped代表三个视图标志位，Finalizable表示这个对象只能通过finalizer才能访问。

第48~63位固定为0没有利用。

读屏障

读屏障是JVM向应用代码插入一小段代码的技术。当应用线程从堆中读取对象引用时，就会执行这段代码。

JVM参数

[JVM参数解析 Xmx、Xms、Xmn、NewRatio、SurvivorRatio、PermSize、PrintGC「建议收藏」 - 腾讯云开发者社区-腾讯云 \(tencent.com\)](#)

[一文带你了解JVM堆详解 - 知乎 \(zhihu.com\)](#)

[Java HotSpot VM Options \(oracle.com\)](#)

-X表示非标准参数，可调用java -X查看

参数	描述
-Xmx	指定最大堆，即堆内存的上线，默认是物理内存的1/4。当实际内存接近上线时会发生GC，默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；
-Xms	初始分配的堆内存大小，也是堆大小的最小值，默认是物理内存的1/64（且小于1G），默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；
-Xmn	新生代内存大小，包括E区和两个S区的总和
-Xss	这个参数用于设置每个线程的栈内存，默认1M
-Xprof	跟踪正运行的程序，并将跟踪数据在标准输出；适合于开发环境调试。
-Xnocompare	关闭针对class的gc功能；因为其阻止内存回收，所以可能会导致OutOfMemoryError错误，慎用；
-Xincgc	开启增量gc（默认为关闭）；这有助于减少长时间GC时应用程序出现的停顿；但由于可能和应用程序并发执行，所以会降低CPU对应用的处理能力。
-Xloggc:file	只是将每次GC事件的相关情况记录到一个文件中

-XX表示非Stable参数（非静态参数），JVM（HotSpot）中主要有三类

性能参数（Performance Options）：用于JVM的性能调优和内存分配控制，如初始化内存大小的设置；

行为参数（Behavioral Options）：用于改变JVM的基础行为，如GC的方式和算法的选择；

调试参数（Debugging Options）：用于监控、打印、输出等jvm参数，用于显示jvm更加详细的信息；

使用方法：

- `-XX:+<option>` 启用选项
- `-XX:-<option>` 不启用选项
- `-XX:<option>=<number>` 给选项设置一个数字类型值，可跟单位，例如 32k, 1024m, 2g
- `-XX:<option>=<string>` 给选项设置一个字符串值，例如-XX:HeapDumpPath=./dump.core

性能参数：性能参数往往用来定义内存分配的大小和比例，相比于行为参数和调试参数，一个比较明显的区别是性能参数后面往往跟的有数值，常用如下：

参数	描述
-XX:NewSize=2.125m	新生代对象生成时占用内存的默认值
-XX:MaxNewSize=size	新生代对象能占用内存的最大值
-XX:MaxPermSize=64m	方法区所能占用的最大内存（非堆内存）
-XX:PermSize=64m	方法区分配的初始内存
- XX:MaxTenuringThreshold=15	对象在新生代存活区切换的次数（坚持过MinorGC的次数，每坚持过一次，该值就增加1），大于该值会进入老年代(年龄阈值)
-XX:MaxHeapFreeRatio=70	GC后java堆中空闲量占的最大比例，大于该值，则堆内存会减少
-XX:MinHeapFreeRatio=40	GC后java堆中空闲量占的最小比例，小于该值，则堆内存会增加
-XX:NewRatio=2	新生代内存容量与老年代内存容量的比例
-XX:ReservedCodeCacheSize=32m	保留代码占用的内存容量
- XX:LargePageSizeInBytes=4m	设置用于Java堆的大页面尺寸
-XX:ThreadStackSize=512	设置线程栈大小，若为0则使用系统默认值
-XX:PretenureSizeThreshold=size	大于该值的对象直接晋升入老年代（这种对象少用为好）
-XX:SurvivorRatio=8	Eden区域Survivor区的容量比值，如默认值为8，代表Eden: Survivor1: Survivor2=8:1:1

行为参数：行为参数主要用来选择使用什么样的垃圾收集器组合，以及控制运行过程中的GC策略等

参数	描述
-XX:+UseSerialGC	启用串行GC，即采用Serial+Serial Old模式
-XX:+UseParallelGC	启用并行GC，即采用Parallel Scavenge+Serial Old收集器组合（-Server模式下的默认组合）
-XX:GCTimeRatio=99	设置用户执行时间占总时间的比例（默认值99，即1%的时间用于GC）（即吞吐量）
- XX:MaxGCPauseMillis=time	设置GC的最大停顿时间（这个参数只对Parallel Scavenge有效）
-XX:+UseParNewGC	使用ParNew+Serial Old收集器组合

参数	描述
-XX:ParallelGCThreads	设置执行内存回收的线程数，在+UseParNewGC的情况下使用
-XX:+UseParallelOldGC	使用Parallel Scavenge +Parallel Old组合收集器
-XX:+UseConcMarkSweepGC	使用ParNew+CMS+Serial Old组合并发收集，优先使用ParNew+CMS，当用户线程内存不足时，采用备用方案Serial Old收集。
-XX:-DisableExplicitGC	禁止调用System.gc()；但jvm的gc仍然有效
-XX:+ScavengeBeforeFullGC	新生代GC优先于Full GC执行

调试参数：主要用于监控和打印GC的信息

参数	描述
-XX:-CITime	打印消耗在JIT编译的时间
-XX:ErrorFile=./hs_err_pid.log	保存错误日志或者数据到文件中
-XX:-ExtendedDTraceProbes	开启solaris特有的dtrace探针
-XX:HeapDumpPath=./java_pid.hprof	指定导出堆信息时的路径或文件名
-XX:-HeapDumpOnOutOfMemoryError	当首次遭遇OOM时导出此时堆中相关信息
-XX:OnError=";"	出现致命ERROR之后运行自定义命令
-XX:OnOutOfMemoryError=";"	当首次遭遇OOM时执行自定义命令
-XX:-PrintClassHistogram	遇到Ctrl-Break后打印类实例的柱状信息，与jmap -histo功能相同
-XX:-PrintConcurrentLocks	遇到Ctrl-Break后打印并发锁的相关信息，与jstack -l功能相同
-XX:-PrintCommandLineFlags	打印在命令行中出现过的标记
-XX:-PrintCompilation	当一个方法被编译时打印相关信息
-XX:-PrintGC	每次GC时打印相关信息
-XX:-PrintGC Details	每次GC时打印详细信息
-XX:-PrintGCTimeStamps	打印每次GC的时间戳
-XX:-TraceClassLoading	跟踪类的加载信息
-XX:-TraceClassLoadingPreorder	跟踪被引用到的所有类的加载信息
-XX:-TraceClassResolution	跟踪常量池

参数	描述
-XX:-TraceClassUnloading	跟踪类的卸载信息
-XX:-TraceLoaderConstraints	跟踪类加载器约束的相关信息
-XX:+PrintHeapAtGC	每次一次GC后，都打印堆信息