

JVM概述

Java的平台无关性

为什么要先编译成字节码再解析成机器码？

JVM有哪些组成部分，各部分的主要功能？

类加载机制

类加载过程

类的生命周期

加载

连接

初始化

什么时候会进行类的初始化（主动引用）？

类加载器与双亲委派模型

三种类加载方式及区别 `loadClass()` 和 `forName()`

`forName()` 和 `loadClass()` 区别

类加载器

双亲委派模型

自定义类加载器的实现

什么时候需要打破双亲委派？ todo

字节码执行机制 todo

JVM内存区域

线程私有

程序计数器

Java 虚拟机栈

本地方法栈

所有线程共享

堆

堆和栈的区别与联系

方法区

Java对象

对象的创建过程

对象的内存布局：对象头

对象的访问定位

垃圾回收

如何判断对象已无效、对象被判定为垃圾的标准

引用计数法

可达性分析算法

可以作为GC Root的对象

finalize方法的作用

方法区的回收

如何判断一个常量是废弃常量

如何判断一个类是无用的类

Java中对象的引用方式

ReferenceQueue的作用

垃圾回收算法

标记-清除算法

复制算法

标记-整理算法

分代收集算法

Java堆内存分配与回收策略

内存分配策略

Full GC 的触发条件

垃圾回收器

STW——stop the world

1. Serial收集器和Serial Old收集器

2. ParNew 收集器 -XX:+UseParNewGC

3. Parallel Scavenge收集器和Parallel Old收集器

4. CMS收集器

5. G1 (Garbage-First) 垃圾收集器

JVM性能调优

性能调优

何时进行JVM调优

JVM调优的基本原则

JVM调优目标

JVM调优量化目标

JVM调优的步骤

JVM 配置常用参数

推荐阅读

JVM概述

什么是JVM?

JVM(Java Virtual Machine), 也就是Java虚拟机。Java 虚拟机可以看作是一台抽象的计算机。如同真实的计算机那样，它有自己的指令集以及各种运行时内存区域。

任何平台只要装有针对于该平台的Java虚拟机，字节码文件 (.class) 就可以在该平台上运行。

JVM的主要功能

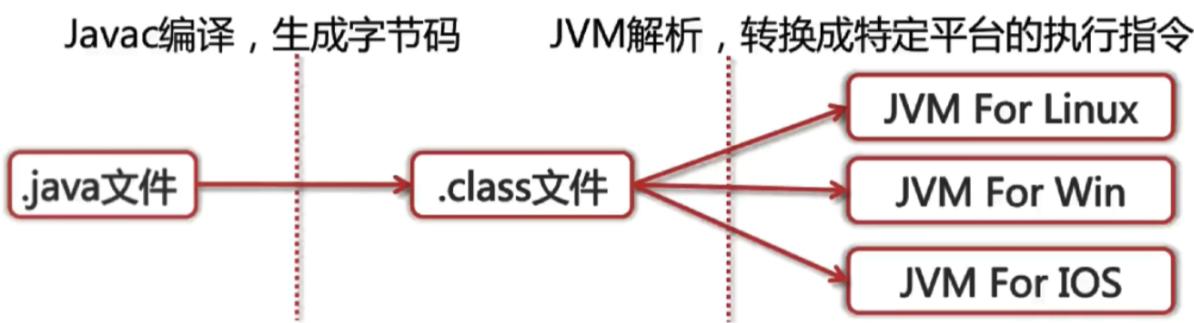
- 通过ClassLoader寻找和装在class文件
- 解释字节码成为指令并执行，提供class文件的运行环境
- 进行运行期间的内存分配和垃圾回收
- 提供与硬件交互的平台

Java的平台无关性

todo<https://www.cnblogs.com/helloworld2048/p/10916296.html>

JVM是Java平台无关的保障。一次编译，到处运行。

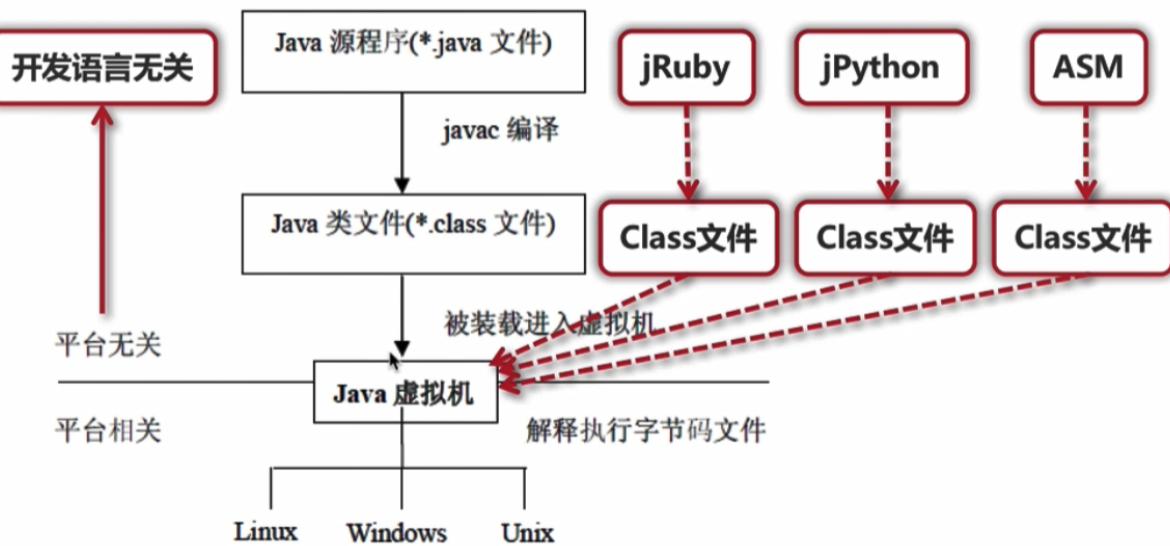
Compile Once, Run Anywhere如何实现



Java 源码首先被编译成字节码，再由不同平台的 JVM 进行解析，Java 语言在不同的平台上运行时不需要进行重新编译，Java 虚拟机在执行字节码的时候，把字节码转换成具体平台上机器指令。

Java与平台无关，而JVM与平台有关，不同的平台安装不同的JRE。Java的运行环境(JRE)由JVM、类库、核心文件组成。

实际上虚拟机不关心.class文件是怎么来的，**只要.class文件符合虚拟机规范，虚拟机就认**。因此在这一层面上JVM虚拟机**与开发语言无关**。可以是除了java以外的其他开发语言，比如jRuby, jPython。也可以是ASM，直接写.class文件。



为什么要先编译成字节码再解析成机器码？

为什么不直接将源码解析成机器码去执行呢？

1. 不然每次执行都需要各种检查。如果直接将源码解析成机器码去执行，每次执行都需要重新编译，每一次执行都要进行各种语法、句法、语义的检查，做这么多重复的东西，整体性能会受影响。因此引入了中间字节码，保证在被编译成字节码之后多次执行程序时不需要进行各种校验和补全。

2. JVM语言无关性、拓展性——可以将别的语言解析成字节码。脱离java的束缚，比如 Scala, Groovy、Jython这些语言生成字节码，同样可以被JVM进行调用执行（只需要满足JVM规范即可），进而可以增加平台的兼容扩展能力，符合软件设计的中庸之道。

如何查看字节码？

javap是jdk自带的反汇编工具（从字节码到JVM汇编指令集-助记符）。它的作用就是根据 class字节码文件，反解析出当前类对应的code区（汇编指令）、本地变量表、异常表和代码行偏移量映射表、常量池等等信息。

```

1 javap -c Demo.class -c 反汇编
2 javap -verbose Demo.class -verbose 输出附加信息

```

Java字节码.class文件是由java文件javac编译生成的16进制文件。

Java虚拟机支持大约248个字节码。每个字节码执行一种基本的CPU运算，例如，把一个整数加到寄存器，子程序转移等。

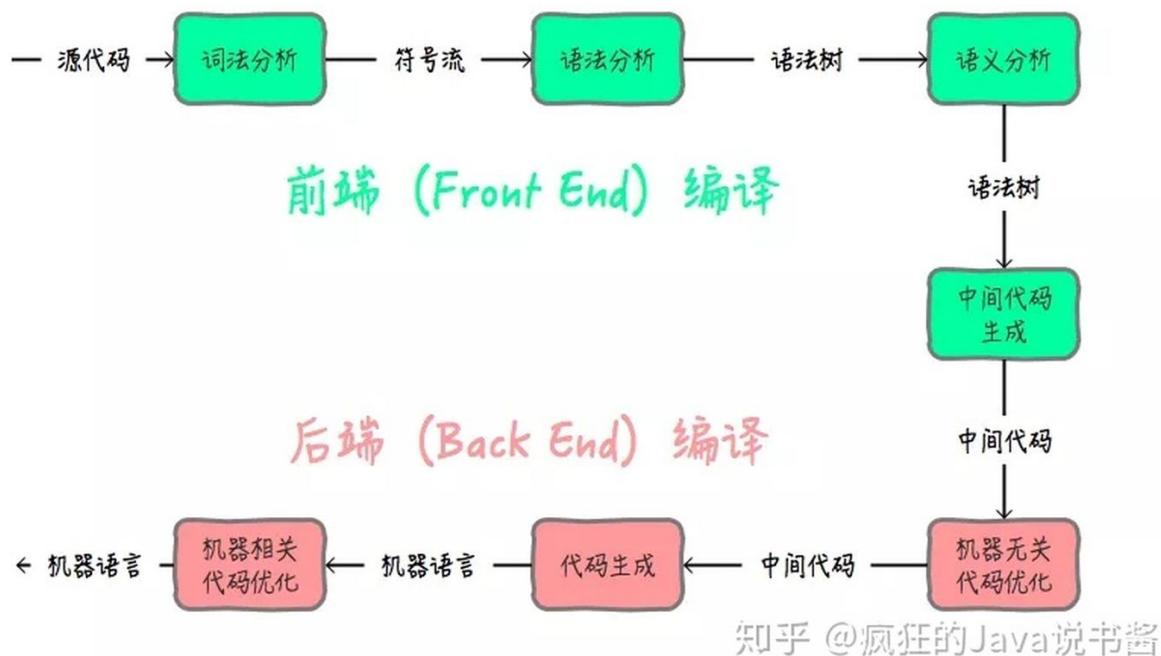
Java指令集相当于Java程序的汇编语言--助记符。由一个字节长度的、代表着某种特定操作含义的操作码（Opcode）以及跟随其后的零至多个代表此操作所需参数的操作数

(Operands) 所构成。

Java的编译与反编译

Java语言作为一种高级语言，想要被执行，就需要通过编译的手段将其转换为机器语言。

Java语言的源文件是一个java文件，要将一个java文件，转换为二进制文件一共要经过两个步骤。



首先经过前端编译器，将java文件编译成中间代码，这种中间代码就是class文件，即字节码文件。

然后，在经过后端编译器，将class字节码文件，编译成机器语言。

Java的前端编译器主要是javac，Eclipse JDT 中的增量式编译器 ECJ 等。

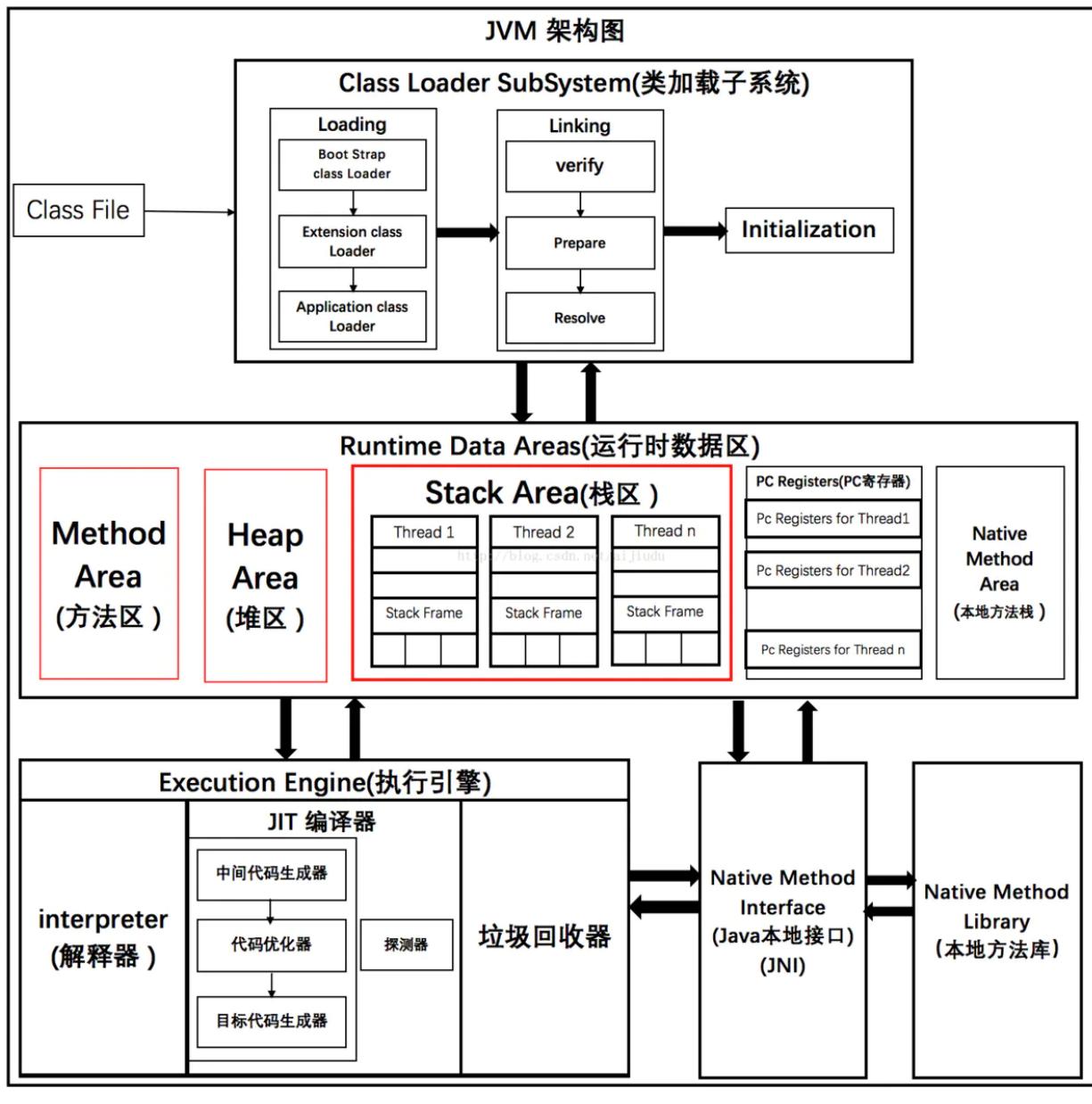
Java的后端编译器主要是各大虚拟机实现的，如HotSpot中的JIT编译器。

JVM有哪些组成部分，各部分的主要功能？

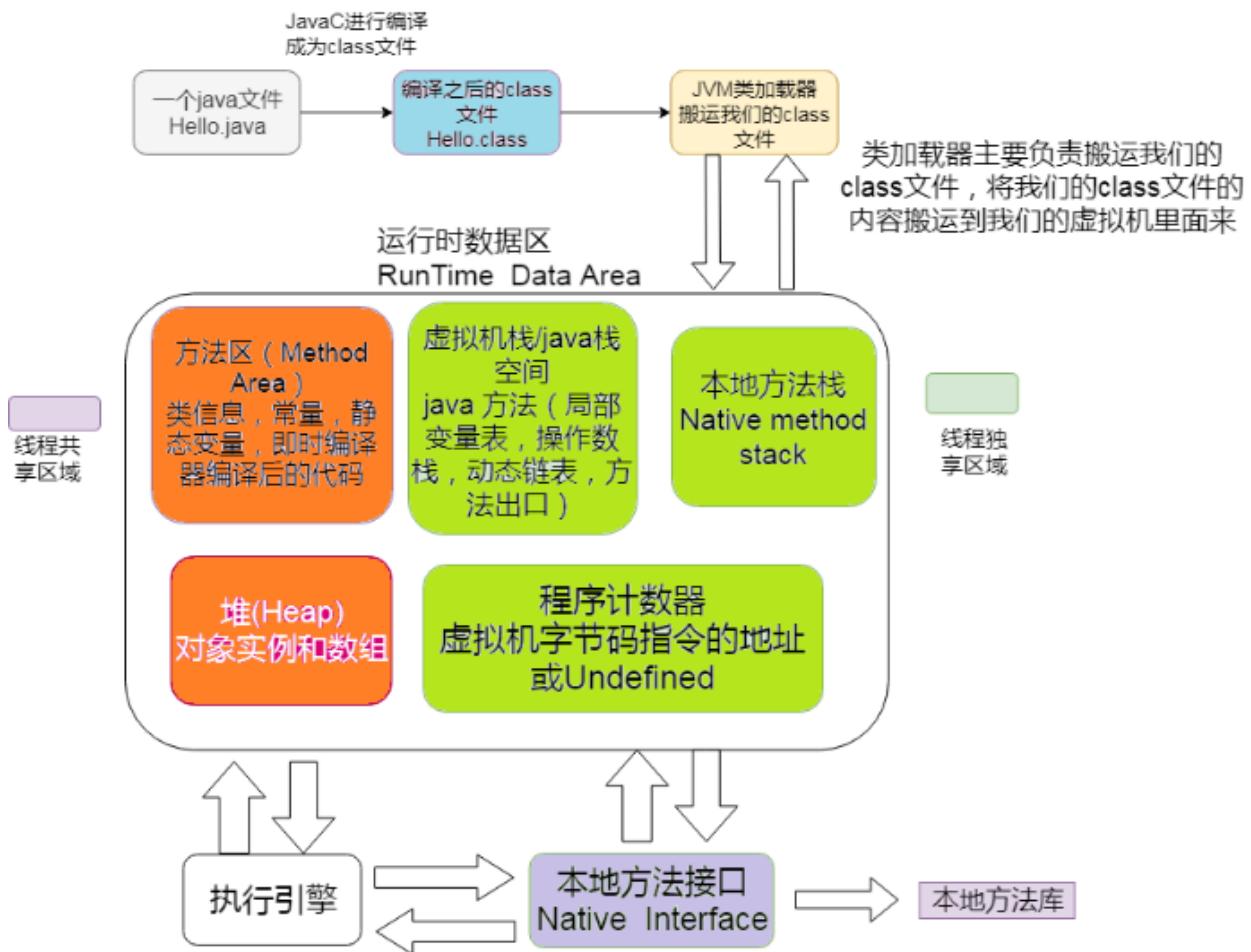
如果没有特殊说明，都是针对的是 HotSpot 虚拟机。

JVM架构图

Java虚拟机主要分为5大模块，类装载器子系统，运行时数据区，执行引擎，本地方法接口，垃圾收集模块。



JVM架构图



常见面试题

JVM结构

各部分主要功能：

类加载器: 负责将class字节码加载到JVM内存区域中（需要符合格式要求，具体过程详见类加载机制）。

执行引擎: 负责解析、执行文件中包含的字节码指令（解释执行，即时编译，OSR）

运行时数据区 (JVM内存模型) :

- **方法区(元空间)**: 用于存储类结构信息的地方，包括常量池、静态变量、构造函数等
- **Java堆 (Heap)** : 存储java实例的地方。这块是GC的主要区域。方法区和堆是被线程共享的。
- **Java栈 (Stack)** : java栈总是和线程关联在一起，每当创建一个线程时，JVM就会为这个线程创建一个对应的java栈。每运行一个方法就创建一个栈帧，用于存储局部变量表、操作栈，方法返回等。
- **程序计数器 (PC Register)** : 一块较小的内存空间，可以看做是当前线程所执行的字节码行号指示器，分支，循环，跳转，异常处理，线程恢复等基础功能都需要一来这个计数器来完成。

- 本地方法栈 (Native Method Stack) : 和java栈作用差不多，只不过是为了JVM使用到的本地方法服务。

本地方法接口: 主要提供调用C或C++实现的本地方法。特别是操作系统相关，底层相关。融合不同开发语言的原生库为Java所用 (C、C++更高效；不重复造轮子)

垃圾回收模块: 主要负责方法区和堆的垃圾回收。

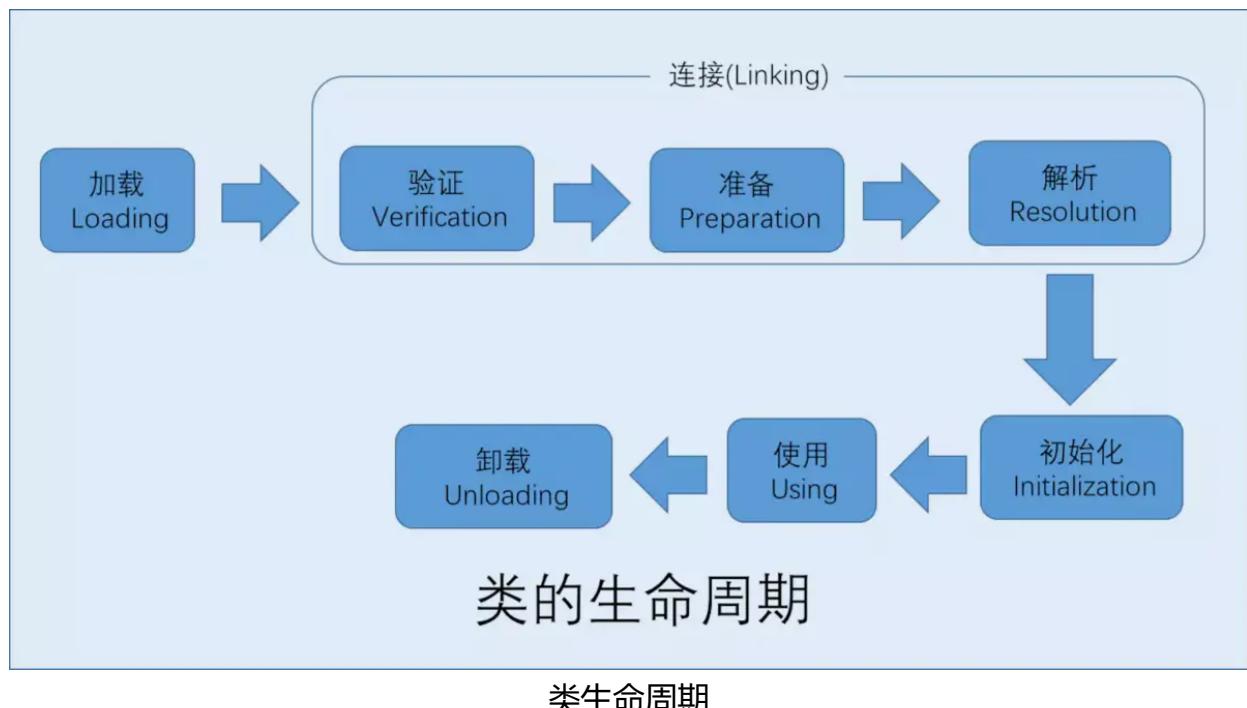
类加载机制

类加载过程

实例化不是类加载的一个过程，类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。

类的生命周期

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、连接（验证、准备、解析）、初始化、使用、卸载



类的加载、验证、准备、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（也称为动态绑定或晚期绑定）。

加载阶段和连接阶段的部分内容是互相交叉进行的（按顺序开始，而不是按顺序进行或完成），通常在一个阶段执行的过程中调用或激活另一个阶段。加载阶段尚未结束，连接阶段可能就已经开始了。

加载

通过ClassLoader加载class字节码文件，生成Class对象。

通过-XX:+TraceClassLoading参数可以观察

类加载过程的第一步，主要完成下面3件事情：

1. 通过类的全限定名获取定义这个类的二进制字节流
2. 将这个字节流所代表的静态存储结构转换为方法区的运行时数据结构
3. 在内存(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

虚拟机规范上面这3点并不具体，因此是非常灵活的。比如："通过类的全限定名获取定义这个类的二进制字节流" 并没有指明具体从哪里获取、怎样获取。

可以从以下方式中获取：

- 从 ZIP 包读取，成为 JAR、EAR、WAR 格式的基础。
- 从网络中获取，最典型的应用是 Applet。
- 运行时计算生成，例如动态代理技术，在 java.lang.reflect.Proxy 使用 ProxyGenerator.generateProxyClass 的代理类的二进制字节流。
- 由其他文件生成，例如由 JSP 文件生成对应的 Class 类。

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）。数组类型不通过类加载器创建，它由 Java 虚拟机直接创建。

连接

验证

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。

不同虚拟机对类验证的实现可能有所不同，但大致都会完成以下四个阶段的验证。

1. 文件格式验证：验证字节流是否符合Class文件格式的规范，并且能被当前版本虚拟机处理。例如：是否以0xCAFEBAE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。

经过验证后，字节流才会进入内存方法区中存储，后面的三个验证都是基于方法区的存储结构进行的。

2. 元数据验证：对类的元数据信息进行语义校验，保证不存在不符合Java语法规范的元数据信息（注意对比javac编译阶段的语义分析）。例如：这个类是否有父类，除

Java.lang.Object之类所有的类都有父类、这个类是否被继承了不允许继承的类（被final修饰的类）等。

3. 字节码验证：数据流和控制流分析。最复杂的一个阶段，对类的方法体进行校验分析，确保程序语义是合法的、符合逻辑的，防止方法在运行时做出危害虚拟机的行为。比如：保证任意时刻操作数栈和指令代码序列都能配合工作。

4. 符号引用验证：保证解析动作（将符号引用转化为直接引用）能正确执行

准备

为类变量分配内存并设置初始值。

类变量是被static修饰的变量

实例变量不会在这阶段分配内存，实例变量会在对象实例化时随着对象一块分配在 Java 堆中。应该注意到，实例化不是类加载的一个过程，**类加载发生在所有实例化操作之前，并且类加载只进行一次，实例化可以进行多次。实例化-对象创建的过程在后面的JVM内存区部分。**

初始值一般为 0 值，例如下面的类变量 value 被初始化为 0 而不是 123。

```
1 public static int value = 123;
```

如果类变量是常量（即同时被**static final**修饰），那么它将初始化为表达式所定义的值而不是 0。例如下面的常量 value 被初始化为 123 而不是 0。

```
1 public static final int value = 123;
```

Java中所有基本数据类型以及reference类型的默认零值：

数据类型	默认零值	数据类型	默认零值
boolean	false	long	0L
char	'\u0000'	float	0.0f
int	0	double	0.0d
short	(short) 0	byte	(byte) 0
reference	null		

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，也就是得到类或者字段、方法在内存中的指针或者偏移量。

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。

解析过程在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 的动态绑定。

符号引用就是一组符号来描述目标，可以是任何字面量。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。在程序实际运行时，只有符号引用是不够的，举个例子：在程序执行方法时，系统需要明确知道这个方法所在的位置。**Java 虚拟机为每个类都准备了一张方法表来存放类中所有的方法**。当需要调用一个类的方法的时候，只要知道这个方法在方法表中的偏移量就可以直接调用该方法了。**通过解析操作符号引用就可以直接转变为方法在类中方法表的位置，从而使得方法可以被调用。**

初始化

执行类变量赋值和静态代码块。

初始化是类加载过程的最后一步，初始化阶段才真正开始执行类中定义的 Java 程序代码。初始化阶段是虚拟机执行类构造器 <clinit>() 方法的过程。在准备阶段，类变量 (static) 已经赋过一次系统要求的初始值，而在初始化阶段，根据程序员通过程序制定的主观计划去初始化类变量和其它资源。

<clinit>()方法与实例构造器<init>()（类的构造函数）方法不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此，在虚拟机中第一个被执行的<clinit>()方法肯定是java.lang.Object

接口中不能使用静态语句块（JDK1.8之后有静态方法），但仍然有类变量（final static）初始化的赋值操作，因此接口也会生成<clinit>()方法。

什么时候会进行类的初始化（主动引用）？

对于初始化阶段，虚拟机严格规范了有且只有6种情况下，必须对类进行初始化(只有主动去使用类才会初始化类)：

- 当遇到 new、getstatic、putstatic 或 invokestatic 这 4 条直接码指令时，比如 new 一个类，读取一个静态字段(未被 final 修饰)、或调用一个类的静态方法时。最常见的生成这 4 条指令的场景是：使用 new 关键字实例化对象的时候；读取或设置

一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候；以及调用一个类的静态方法的时候。

- 使用 java.lang.reflect 包的方法对类进行反射调用时如 Class.forName("...").newInstance() 等等，如果类没初始化，需要触发其初始化。
- 初始化一个类，如果其父类还未初始化，则先触发该父类的初始化。
- 当虚拟机启动时，用户需要定义一个要执行的主类（包含 main 方法的那个类），虚拟机会先初始化这个类。
- MethodHandle 和 VarHandle 可以看作是轻量级的反射调用机制，而要想使用这 2 个调用，就必须先使用 findStaticVarHandle 来初始化要调用的类。
- 当一个接口中定义了 JDK8 新加入的默认方法（被 default 关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那该接口要在其之前被初始化。

其余情况都不会进行类的初始化，成为**被动引用**。

例如：

1. 通过子类引用父类的静态字段，不会导致子类初始化。

实例：<https://blog.csdn.net/SMonkeyKing/article/details/82831302>

2. 通过数组定义来引用类，不会触发此类的初始化。该过程会对数组类进行初始化，数组类是一个由虚拟机自动生成的、直接继承自 Object 的子类，其中包含了数组的属性和方法。

```
1 SuperClass[] sca = new SuperClass[10];
```

3. 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。

```
1 System.out.println(ConstClass.HELLOWORLD);
```

卸载、卸载的前提(要求)

卸载类即该类的 Class 对象被 GC。

卸载类需要满足 3 个要求：

1. 该类的所有实例对象都已被 GC，也就是说堆不存在该类的实例对象。
2. 该类没有在其他任何地方被引用
3. 该类的类加载器的实例已被 GC

所以，在 JVM 生命周期类，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。

只要想通一点就好了， 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。

结束

在如下几种情况JVM将结束生命周期

- 执行了System.exit(0)方法
- 程序正常执行结束
- 程序再执行过程中遇到了异常或错误而异常终止
- 由于操作系统出现错误而导致JVM虚拟机进程终止

类加载器与双亲委派模型

<http://gityuan.com/2016/01/24/java-classloader/>

<https://blog.csdn.net/xyang81/article/details/7292380>

<https://juejin.cn/post/6844903729435508750>

所有的类都由类加载器加载，**加载的作用就是将 .class文件加载到内存**。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**）。数组类型不通过类加载器创建，它由 Java 虚拟机直接创建。

三种类加载方式及区别 loadClass()和forName()

隐式加载：new关键字

显式加载

- 1.通过Class.forName()来加载类，然后调用类的newInstance()方法实例化对象。
- 2.通过类加载器的loadClass()方法来加载类，然后调用类的newInstance()方法实例化对象。

new关键字不需要调用类对象的newInstance来实例化

new支持构造器传入参数。class对象的newInstance不支持传入参数，需要通过反射获取constructor。

forName()和loadClass()区别

forName()会初始化类：会进行类变量赋值和执行静态代码块。loadClass只进行了加载过程，第二步的链接和初始化都没有进行。

代码实例

从代码看：

```
1 loadClass(string name, boolean resolve);
```

resolve变量决定是否链接这个类。

```

1  @CallerSensitive
2  public static Class<?> forName(String className)
3  throws ClassNotFoundException {
4  Class<?> caller = Reflection.getCallerClass();
5  return forName0(className, true, ClassLoader.getClassLoader(caller), caller);
6 }
7  private static native Class<?> forName0(String name, boolean initialize,
8  ClassLoader loader,
9  Class<?> caller)

```

forName中调用forName0, initialize变量为true, 会进行初始化。

这样区别的作用：

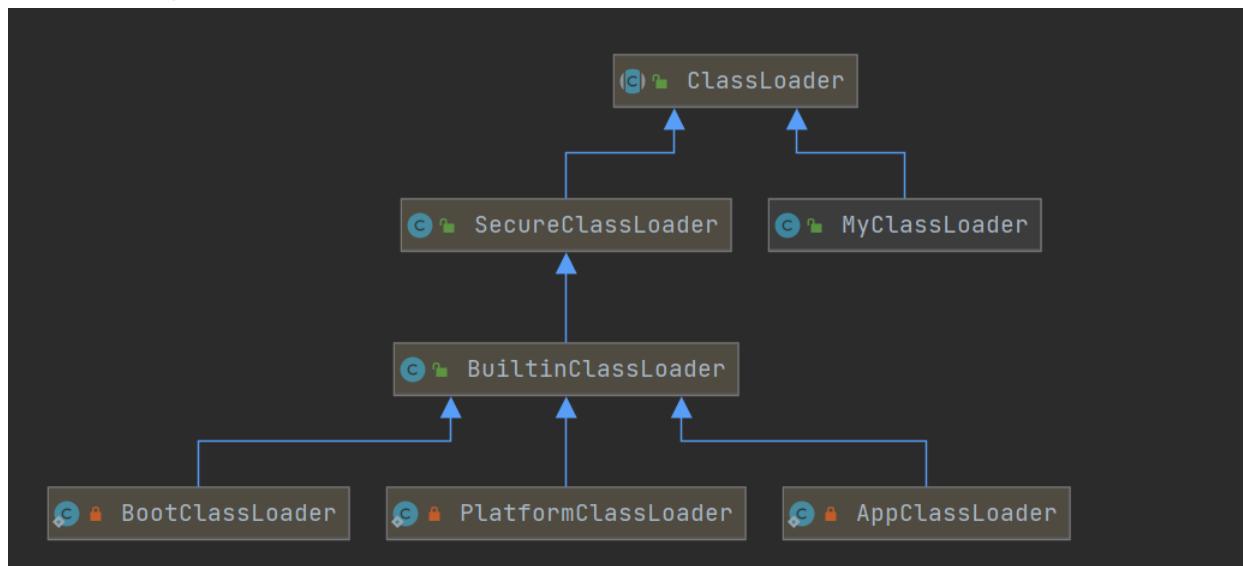
forName举例：在程序中要连接MySQL，需要加载MySQL的driver，要用forName()，不能用loadClass()。因为driver中有静态语句块需要被执行：生成Driver对象，创建数据库驱动。

loadClass举例：在Spring IOC中资源加载器读取bean的配置资源时，如果以classpath的形式加载，需要用loadClass加载，spring IOC是lazy loading，为了加快初始化速度，延迟加载。使用loadClass不需要执行类的初始化代码和连接步骤，可以加快初始化速度，把类的初始化工作留到实际使用时再做。

类加载器

作用：在Class装载的加载阶段，从Class文件中的二进制数据流装载进系统。

几种类加载器都在`jdk.internal.loader.ClassLoaders`中，static内部类，查看源码可以看到相互关系（parent）。



从 Java 虚拟机的角度来讲，只存在以下两种不同的类加载器：

- 启动类加载器 (Bootstrap ClassLoader) , 使用 C++ 实现，是虚拟机自身的一部分；
- 所有其它类的加载器，使用 Java 实现，独立于虚拟机，**继承自抽象类 java.lang.ClassLoader.**

从 Java 开发人员的角度看，类加载器可以划分得更细致一些：

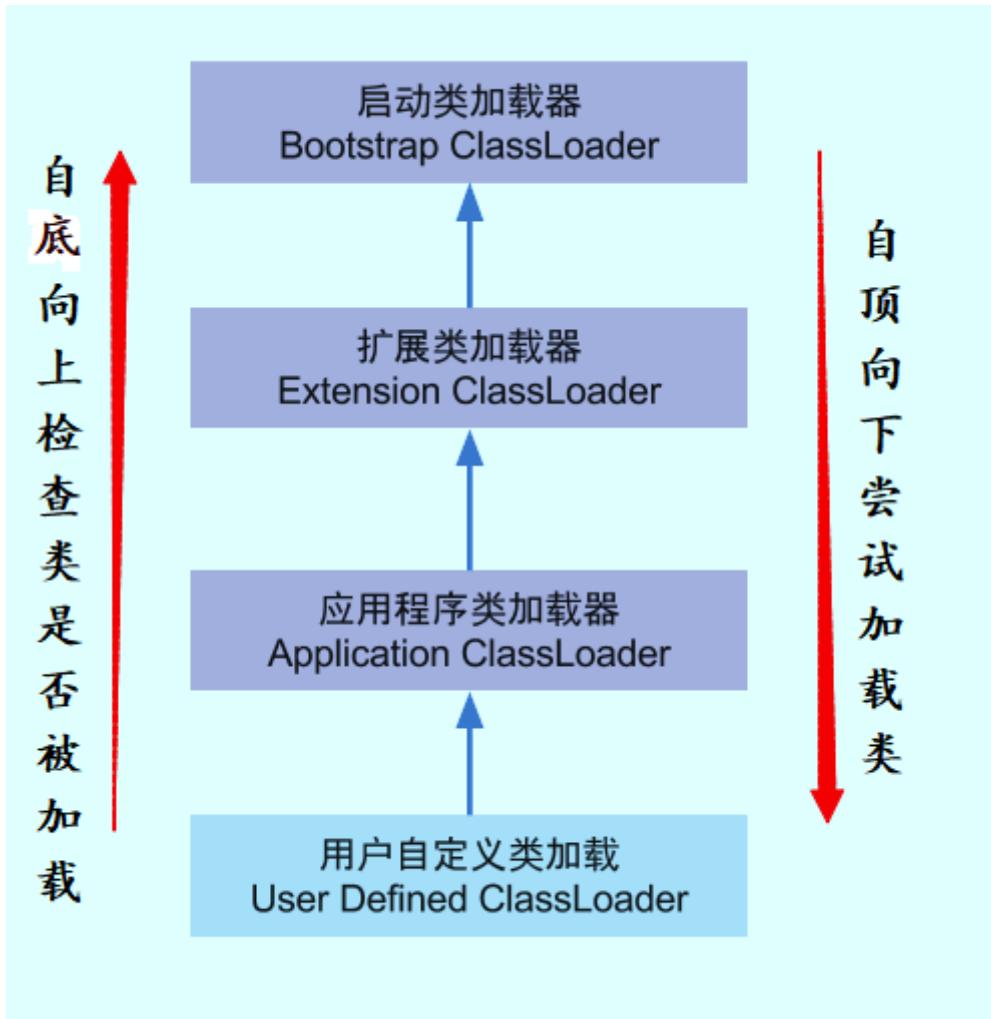
1. BootstrapClassLoader(启动类加载器)：最顶层的加载类，由C++实现，负责加载 %JAVA_HOME%/lib 目录下的jar包和类或者或被 -Xbootclasspath 参数指定的路径中的所有类库。
2. ExtensionClassLoader(扩展类加载器， JKD1.9之后为PlatformClassLoader)：主要负责加载目录 %JRE_HOME%/lib/ext 目录下的jar包和类，或被 java.ext.dirs 系统变量所指定的路径下的jar包。
3. AppClassLoader(应用程序类加载器) :面向我们用户的加载器，负责加载**当前应用 classpath 下的所有jar包和类**。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，因此一般称为系统类加载器。如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

双亲委派模型

The Java platform uses a delegation model for loading classes. The basic idea is that **every class loader has a "parent" class loader**. When loading a class, a class loader first "**delegates**" the search for the class to its parent class loader before attempting to find the class itself.

加载时一个类加载器首先将类加载请求转发到父类加载器，只有当父类加载器无法完成时才尝试自己加载。

并且，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。



双亲委派模型实现源码

双亲委派模型的实现代码逻辑非常清晰，在`java.lang.ClassLoader`的`loadClass()`中，先检查类是否已经加载过，如果没有则让父类加载器去加载。当父类加载器加载失败时抛出`ClassNotFoundException`，此时尝试自己去加载。

`java.lang.ClassLoader`中的`loadClass()`是`ClassLoader`的默认实现方式，如果用户自定义`extends ClassLoader`，采用的是这种实现方式。对于`parent`的`appClassLoader`，以及它的`parent` `ExtensionClassLoader`，调用的`loadClass`不在这里，他们都继承了`ClassLoader`，重写了`loadClass`方法，但步骤也是一样的，可以简单的认为就是这样的递归调用。

```

1 public abstract class ClassLoader {
2     // The parent class loader for delegation
3     private final ClassLoader parent;
4
5     public Class<?> loadClass(String name) throws ClassNotFoundException {
6         return loadClass(name, false);
7     }
8     // 返回的是Class对象，可以通过这个反射对象获取加载的类的任意方法和属性。
9     protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {
10        //synchronized 防止多个线程同时加载同一个类

```

```
11 synchronized (getClassLoadingLock(name)) {
12     // 先检查这个类是不是已经被加载
13     Class<?> c = findLoadedClass(name);
14     if (c == null) {
15         try { //没有被加载，调用parent的loadClass，又继续递归查找。
16             if (parent != null) {
17                 c = parent.loadClass(name, false);
18             } else {
19                 //BootstrapClassLoader是C++编写，parent为null
20                 //Bootstrap用jvm native方法加载
21                 //源码位置: http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/1fd7ad9f225
22                 c = findBootstrapClassOrNull(name);
23             }
24         } catch (ClassNotFoundException e) {
25             // ClassNotFoundException thrown if class not found
26             // from the non-null parent class loader
27             //抛出异常说明父类加载器无法完成加载请求
28         }
29
30         if (c == null) {
31             //parent一层一层递归都没找到，自己再尝试加载
32             c = findClass(name);
33         }
34     }
35     if (resolve) {
36         resolveClass(c);
37     }
38     return c;
39 }
40 }
41
42 protected Class<?> findClass(String name) throws ClassNotFoundException
{
43     throw new ClassNotFoundException(name);
44 }
45 }
46 }
```

双亲委派模型的好处

1. 可以避免类的重复加载。通过委托去向上面问一问，加载过了，就不用再加载一遍。保证数据安全。

2. 保证了 Java 的核心类库不被篡改，保证了Class执行安全。通过委托方式，**不会去篡改核心.clas，即使篡改也不会去加载，即使加载也不会是同一个.class对象了。**

换句话说：使得 Java 类随着它的类加载器一起具有一种带有优先级的层次关系，从而使得基础类得到统一。

例如 java.lang.Object 存放在 rt.jar 中，如果编写另外一个 java.lang.Object 并放到 ClassPath 中，程序可以编译通过。由于双亲委派模型的存在，所以在 rt.jar 中的 Object 比在 ClassPath 中的 Object 优先级更高，这是因为 rt.jar 中的 Object 使用的是启动类加载器，而 ClassPath 中的 Object 使用的是应用程序类加载器。rt.jar 中的 Object 优先级更高，那么程序中所有的 Object 都是这个 Object。

同一个类 被两个不同的类加载器加载 它们会被当成同一个类吗

不会 JVM 区分不同类的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类。**会在方法区产生两个不同的类，彼此不可见，并且在堆中生成不同 Class实例。**

自定义类加载器的实现

- 除了 BootstrapClassLoader 其他类加载器均由 Java 实现且全部继承自 java.lang.ClassLoader。如果我们要自定义自己的类加载器，很明显**需要继承 ClassLoader**。
- 如果我们不想打破双亲委派模型，就**重写 ClassLoader类中的findClass() 方法**即可，无法被父类加载器加载的类最终会通过这个方法被加载。
- 但是，如果想打破双亲委派模型则需要**重写loadClass() 方法**。

通过重写findClass()自定义加载类：

```
1 //1.根据类的全名在文件系统上查找类的字节代码文件 (.class 文件) ,
2 //a. 用java文件io读取该文件内容
3 //b. 通过 defineClass() 方法来把这些字节代码转换成java.lang.Class类的实例。
4 // defineClass是native方法
5 package com.interview.javabasic.reflect;
6
7 import java.io.ByteArrayOutputStream;
8 import java.io.File;
9 import java.io.FileInputStream;
10 import java.io.InputStream;
```

```
11
12 public class MyClassLoader extends ClassLoader {
13     private String path;
14     private String classLoaderName;
15
16     public MyClassLoader(String path, String classLoaderName) {
17         this.path = path;
18         this.classLoaderName = classLoaderName;
19     }
20
21     //用于寻找类文件
22     @Override
23     public Class findClass(String name) {
24         byte[] b = loadClassData(name);
25         //Converts an array of bytes into an instance of class Class.
26         return defineClass(name, b, 0, b.length);
27     }
28
29     //用于加载类文件
30     private byte[] loadClassData(String name) {
31         name = path + name + ".class";
32         InputStream in = null;
33         ByteArrayOutputStream out = null;
34         try {
35             in = new FileInputStream(new File(name));
36             out = new ByteArrayOutputStream();
37             int i = 0;
38             while ((i = in.read()) != -1) {
39                 out.write(i);
40             }
41         } catch (Exception e) {
42             e.printStackTrace();
43         } finally {
44             try {
45                 out.close();
46                 in.close();
47             } catch (Exception e) {
48                 e.printStackTrace();
49             }
50         }
```

```
51     return out.toByteArray();
52 }
53 }
```

用自定义的类加载器加载指定的class文件，获取Class的反射对象，然后就可以做一系列对对象的操作。

```
1 package com.interview.javabasic.reflect;
2
3 public class ClassLoaderChecker {
4     public static void main(String[] args) throws ClassNotFoundException, IllegalAccessException, InstantiationException {
5         MyClassLoader m = new MyClassLoader("/Users/baidu/Desktop/", "myClassLoader");
6         Class c = m.loadClass("Wali");
7         System.out.println(c.getClassLoader());
8         System.out.println(c.getClassLoader().getParent());
9         System.out.println(c.getClassLoader().getParent().getParent());
10        System.out.println(c.getClassLoader().getParent().getParent().getParent());
11        c.newInstance();
12    }
13 }
14
```

自定义加载类（重写findClass）的意义

可以以不同形式加载类，比如上例的本地文件中的类，网络二进制流获取需要的类。

可以对某些敏感的class文件进行加密，在findClass中进行解密。

还可以对生成的二进制代码做修改，给类添加一些信息。ASM——字节码增强技术

可以延伸到AOP的实现。

什么时候需要打破双亲委派？ todo

原因在于双亲委派的局限性：父级加载器无法加载子级类加载器路径中的类。

和SPI紧密相关，目的JDK提供接口，供应商提供服务实现。第三方实现只能放在classpath，但需要BootstrapClassLoader来加载，这时候就需要打破双亲委派机制。

以java.sql.Driver举例，JDBC在DriverManager中调用sql的Driver，DriverManager当前类的加载器是BootstrapClassLoader。BootstrapClassLoader去加载第三方类库的实现类，lib中找不到。SPI ServiceLoader通过从线程上下文（ThreadContext）获取application classloader，借助这个classloader可以拿到实现类的Class。逻辑上打破了双亲委派。

SPI逻辑上破坏了双亲委派。

JDK提供接口，供应商提供服务。比如`java.sql.Driver`

java提供SPI接口，第三方实现类，这些第三方实现类不能添加到jdk的lib目录下，不能用BootstrapClassLoader加载。通常打包放到classpath。

Java给数据库操作提供了一个Driver接口，然后Java有一个DriverManager类来管理所有的加载的Driver驱动。

驱动加载过程：

从META-INF/services/java.sql.Driver文件得到实现类名字DriverA

Class.forName("xx.xx.DriverA")来加载实现类

Class.forName()方法默认使用当前类的ClassLoader，JDBC是在DriverManager类里调用Driver的，当前类也就是DriverManager，它的加载器是BootstrapClassLoader。

用BootstrapClassLoader去加载非rt.jar包里的类xx.xx.DriverA，就会找不到

要加载xx.xx.DriverA需要用到AppClassLoader或其他自定义ClassLoader

最终矛盾出现在，要在BootstrapClassLoader加载的类里，调用AppClassLoader去加载实现类

但是BootstrapClassLoader不可见application classloader加载的类，需要父类加载器请求子类加载器去完成类加载的动作。SPI ServiceLoader通过从线程上下文

(ThreadContext) 获取 classloader (一般情况下是 application classloader)，借助这个classloader 可以拿到实现类的 Class。逻辑上打破了双亲委派。

字节码执行机制 todo

<https://www.jianshu.com/p/744c87f2d1ab>

JVM内存区域

对于 Java 程序员来说，在虚拟机自动内存管理机制下，不再需要像 C/C++程序开发程序员这样为每一个 new 操作去写对应的 delete/free 操作，不容易出现内存泄漏和内存溢出问题。正是因为 Java 程序员把内存控制权利交给 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那么排查错误将会是一个非常艰巨的任务。

jvm内存使用用户空间。

线程私有

程序计数器

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程序计数器主要有两个作用：

1. 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。
2. 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。

Java 虚拟机栈

与程序计数器一样，Java 虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是 Java 方法执行的内存模型，每次方法调用的数据都是通过栈传递的。

Java 虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。

局部变量表：主要存放了方法执行过程中的所有变量，编译期可知的各种数据类型

(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)。因此得出栈和堆的关系：**栈中的局部变量表中的引用类型的变量指向堆中的对象。**

操作数栈：入栈、出栈、复制、交换、产生消费变量。用来临时存放代码在运行过程中临时的操作数。比如：a=10*3 10和3会先放入操作数栈

iload 指令，将一个局部变量加载到操作数栈。

istore 指令，将一个数值从操作数栈存储到局部变量表。

动态链接：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连接(Dynamic Linking)。

<https://blog.csdn.net/xyh930929/article/details/84067186>

<https://www.cnblogs.com/ding-dang/p/13051143.html>

<https://www.zhihu.com/question/30300585> **JVM里的符号引用如何存储？**

方法出口：记录位置，与方法绑定，一个方法运行结束以后，回到 main 方法的哪一行接着运行。比如 compute 方法运行结束以后，main() 中应该从哪里开始继续执行。

Java 虚拟机栈会出现两种错误：StackOverFlowError 和 OutOfMemoryError。

- StackOverFlowError：若 Java 虚拟机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：栈进行动态扩展时如果无法申请到足够内存，会抛出 **OutOfMemoryError** 异常。若 Java 虚拟机堆中没有空闲内存，并且垃圾回收器也无法提供更多内存的话。就会抛出 OutOfMemoryError 错误。

Java 虚拟机栈也是线程私有的，每个线程都有各自的 Java 虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

可以通过 -Xss 这个虚拟机参数来指定每个线程的 Java 虚拟机栈内存大小，在 JDK 1.4 中默认为 256K，而在 **JDK 1.5+** 默认为 1M：

```
1 java -Xss2M HackTheJava
```

本地方法栈

本地方法栈与 Java 虚拟机栈类似，它们之间的区别只不过是本地方法栈为本地方法服务。本地方法一般是用其它语言（C、C++ 或汇编语言等）编写的，并且被编译为基于本机硬件和操作系统的程序，对待这些方法需要特别处理。

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。

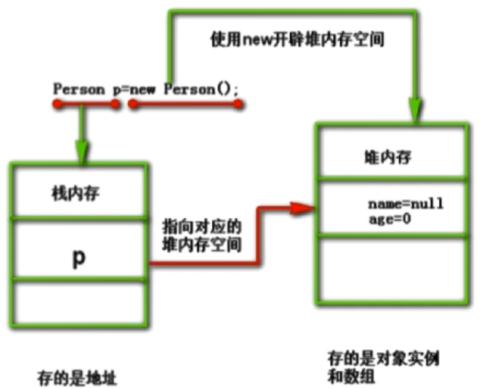
方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 StackOverFlowError 和 OutOfMemoryError 两种错误。

所有线程共享

堆

堆和栈的区别与联系

联系：对象和数组存储在堆中，在引用对象、数组时，栈中的引用变量保存堆中对象、数组的首地址。栈中的引用变量是普通变量，在程序运行到其作用域之外就会被释放，而对象和数组在堆中分配，即使程序运行到了使用 new 产生的变量、数组所在的代码块之外，对象和数组占用的堆内存也不会被释放，它们只有在没有引用变量指向的时候才会变成垃圾，在随后一个不确定的时间被垃圾回收器释放掉。



区别

1. 管理方式：栈自动释放，堆需要GC。

即使堆空间有GC，还需要考虑释放堆空间，栈不需要考虑这个问题。

2. 空间大小：栈远比堆小。

3. 分配方式：栈支持静态和动态分配，而堆只支持动态分配。

4. 栈的效率比堆的高，内存本身就是堆栈结构的排列，栈和底层结构更符合，操作简单，只有入栈和出栈，效率更高。但相比堆灵活程度不够，特别是在动态管理的时候。堆底层可能是双向的链表。

5. 栈产生的碎片远比堆小。

堆栈的设计很巧妙，堆向上涨，栈往下涨。即堆由低到高长，栈由高到低长

内存分配策略

静态存储：编译时确定每个数据目标在运行时的存储空间需求。

栈式存储：数据区需求在编译时未知，运行时模块入口前确定。

堆式存储：编译时或运行时模块入口都无法确定，动态分配。

方法区

用于存放已被加载的类信息（method、field）、类文件、常量、静态变量等数据。

方法区是一个JVM规范，永久代与元空间都是其一种实现方式。**在JDK8之后，用元空间替代了永久代**。原来永久代的数据被分到了堆和元空间中。静态变量和字符串常量池等放入堆中，类的元信息放在元空间中。**元空间使用的是直接内存**。

字符串常量池放在堆中，运行时常量池放在哪里？运行时常量池包含哪些内容？Todo总结

[链接1](#)、[链接2](#)

```

1 public static final int initData = 666; //常量--放在方法区
2 public static User user=new User(); //静态变量--放在方法区 指向堆中的对象

```

方法区和堆之间的关系：方法区中的静态的引用类型的变量存的是堆中的地址，指向堆中的对象。

直接内存

在 JDK 1.4 中新引入了 NIO 类，它可以使用 Native 函数库直接分配堆外内存，然后通过 Java 堆里的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为**避免了在堆内存和堆外内存来回拷贝数据**。

本机直接内存的分配不会受到 Java 堆的限制，但是，既然是内存就会受到本机总内存大小以及处理器寻址空间的限制。

当元空间溢出时会得到如下错误：java.lang.OutOfMemoryError: MetaSpace

JDK8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

```
1 -XX:PermSize=N //方法区（永久代）初始大小  
2 -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
```

在IDEA中edit configurations中，VM option中填入。

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区后就“永远存在”了。

下面是一些元空间的常用参数：

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

可以使用 -XX: MaxMetaspaceSize 标志设置最大元空间大小，**默认值为 unlimited**，这意味着它只受系统内存的限制。-XX: MetaspaceSize 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。

元空间(MetaSpace)相比永久代(PermGen)的优势

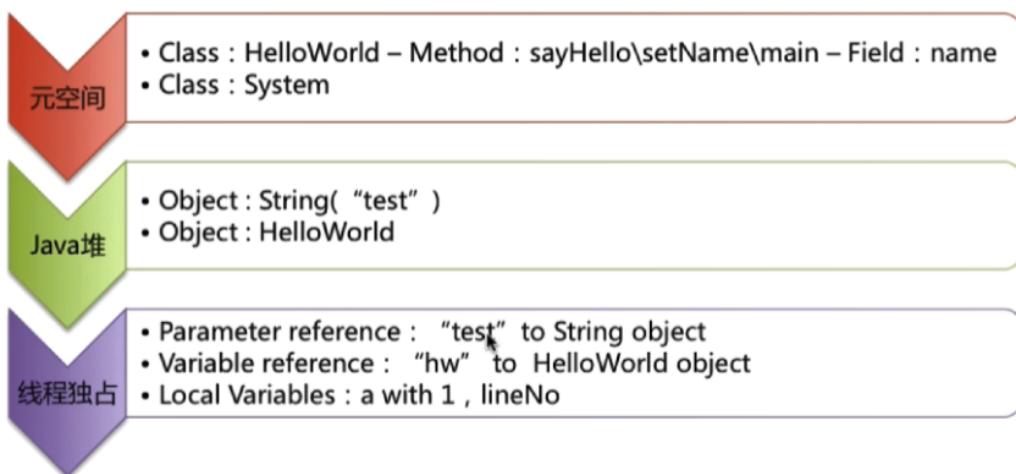
- 整个永久代有一个 JVM 本身设置固定大小上限，无法进行调整，而**元空间使用的是直接内存**，受本机可用内存的限制，可以加载更多的类，且出现OOM的几率会更小。
- 字符串常量池存在永久代中，若频繁调用intern()，容易出现性能问题和OOM。
字符串常量池与intern()讲解：[2021-Java基础.note](#) 有了元空间，字符串常量池放在了堆中就没有这个问题了。
- 类和方法的信息大小难以确定，给永久代的大小指定带来困难。太小，容易导致永久代溢出，太大容易导致老年代溢出。

了解：

- 方便HotSpot与其他JVM的继承，永久代是hotspot特有的。

- **方便对字符串常量池的回收。** 永久代垃圾回收的复杂性：对这块区域进行垃圾回收的主要目标是对常量池的回收和对类的卸载，但是一般比较难实现。HotSpot 虚拟机把它当成永久代来进行垃圾回收。但很难确定永久代的大小，因为它受到很多因素影响，并且每次 Full GC 之后永久代的大小都会改变，所以经常会抛出 OutOfMemoryError 异常。把字符串常量池放到堆中一定程度上避免了这个问题。

元空间、堆、线程独占部分间的联系—内存角度



lineNo 行号，用来记录代码执行，追踪程序。

```

1 public class HelloWorld {
2     private String name;
3     public void sayHello(){
4         System.out.println("Hello " + name);
5     }
6     public void setName(String name){
7         this.name = name;
8     }
9     public static void main(String[] args){
10        int a = 1;
11        HelloWorld hw = new HelloWorld();
12        hw.setName("test");
13        hw.sayHello();
14    }
15 }
```

Java对象

对象的创建过程

类加载检查、分配内存、初始化零值、设置对象头、执行init方法

Step1:类加载检查

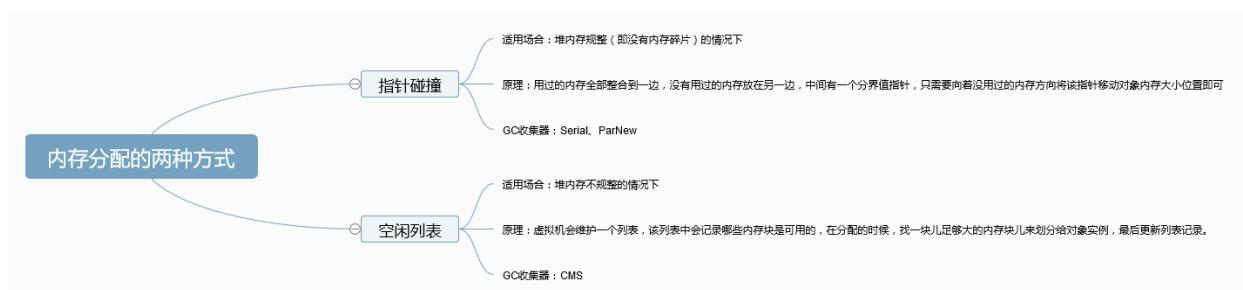
虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

Step2:分配内存

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。**分配方式有“指针碰撞”和“空闲列表”两种**，选择哪种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

内存分配的两种方式：（补充内容，需要掌握）

选择“指针碰撞”和“空闲列表”哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是“**标记-清除**”，还是“**标记-整理**”或“**复制**”。也就是说，只有 CMS 的情况存在内存碎片，使用空闲列表方法。



内存分配并发问题（补充内容，需要掌握）

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- CAS+失败重试:** CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。
- TLAB:** 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

Step3: 初始化零值

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

Step4:设置对象头

初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在对象头中。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

Step5:执行 init 方法

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始，`<init>` 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 `<init>` 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

对象的内存布局：对象头

<https://www.jianshu.com/p/3d38cba67f8b>

在 Hotspot 虚拟机中，对象在内存中的布局可以分为 3 块区域：**对象头、实例数据和对齐填充**。

实例数据部分是对象真正存储的有效信息，也是在程序中所定义的各种类型的字段内容。

对齐填充部分不是必然存在的，也没有什么特别的含义，仅仅起占位作用。因为 Hotspot 虚拟机的自动内存管理系统要求对象起始地址必须是 8 字节的整数倍，换句话说就是对象的大小必须是 8 字节的整数倍。而对象头部分正好是 8 字节的倍数（1 倍或 2 倍），因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

对象头

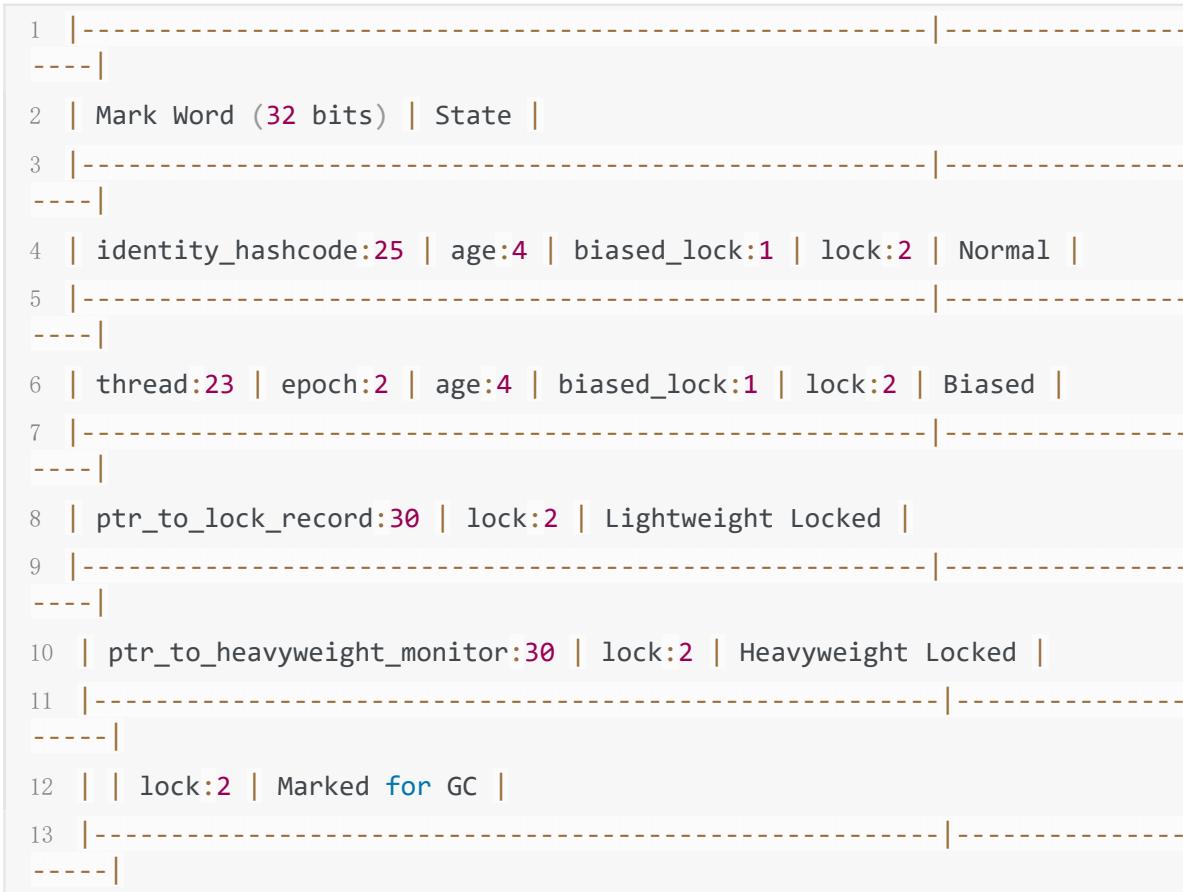
Java 对象头详解 <https://www.jianshu.com/p/3d38cba67f8b>

Hotspot 虚拟机的对象头包括两部分信息，第一部分**Mark Word**，用于存储对象自身的运行时数据（哈希码、GC 分代年龄、锁类型、锁状态标志等等），另一部分是**类型指针**，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是那个类的实例。

Mark Word

锁状态	25bit		4bit	1bit	2bit	
	23bit	2bit		是否是偏向锁	锁标志位	
无锁状态	对象 hashCode、对象分代年龄				01	
轻量级锁	指向锁记录的指针				00	
重量级锁	指向重量级锁的指针				10	
GC 标记	空，不需要记录信息				11	
偏向锁	线程 ID	Epoch	对象分代年龄	1	01	

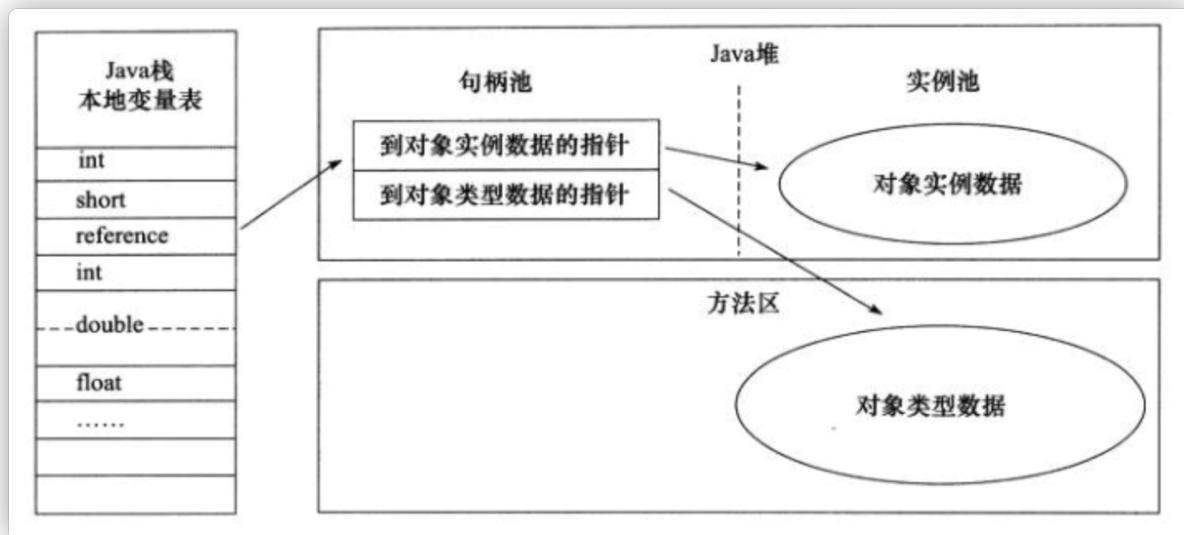
当对象加锁后（偏向、轻量级、重量级），MarkWord的字节没有足够的空间保存 hashCode， hashCode 的值会移动到管程 Monitor 中。



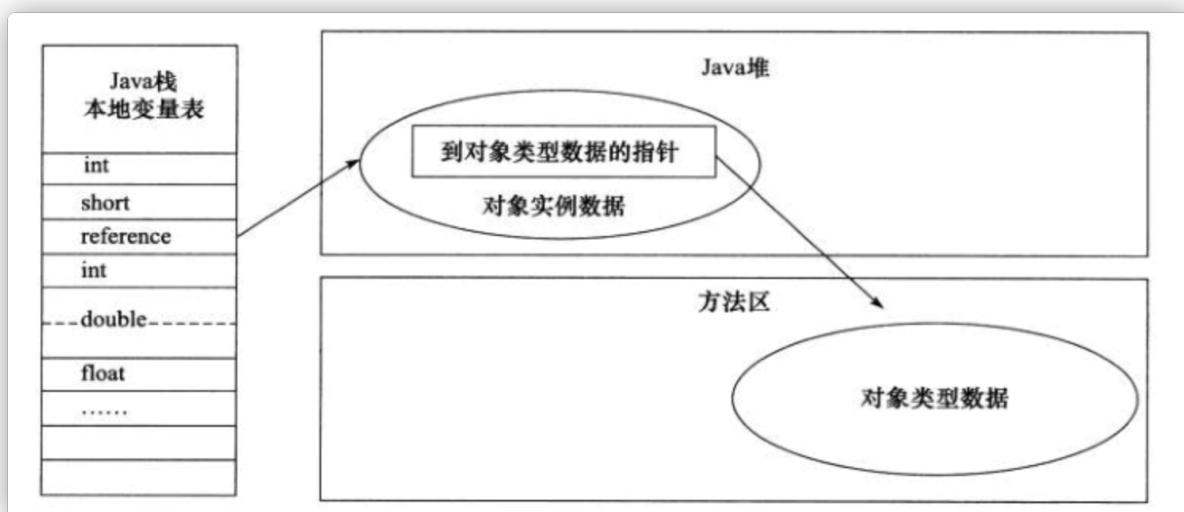
对象的访问定位

建立对象就是为了使用对象，我们的 Java 程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式由虚拟机实现而定，目前主流的访问方式有①使用句柄和②直接指针两种：

1. 句柄：如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息；



2. 直接指针：如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问**类型数据**（**对象头中的类型指针**）的相关信息，而 **reference** 中存储的直接就是对象的地址。



这两种对象访问方式各有优势。使用句柄来访问的最大好处是 **reference** 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 **reference** 本身不需要修改。特别是到处都有对象引用的时候，如果不使用句柄，每个引用类型的局部变量都得修改。

使用直接指针访问方式最大的好处就是**速度快**，它节省了一次指针定位的时间开销。

垃圾回收

垃圾收集主要是针对堆和方法区进行。程序计数器、虚拟机栈和本地方法栈这三个区域属于线程私有的，只存在于线程的生命周期内，线程结束之后就会消失，因此不需要对这三个区域进行垃圾回收。

如何判断对象已无效、对象被判定为垃圾的标准

引用计数法

每个对象有一个引用计数器，每当有一个地方引用它，计数器就加 1；当引用失效（**超过引用变量作用范围-局部变量或者被设置为新值**），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。

+ 简单高效

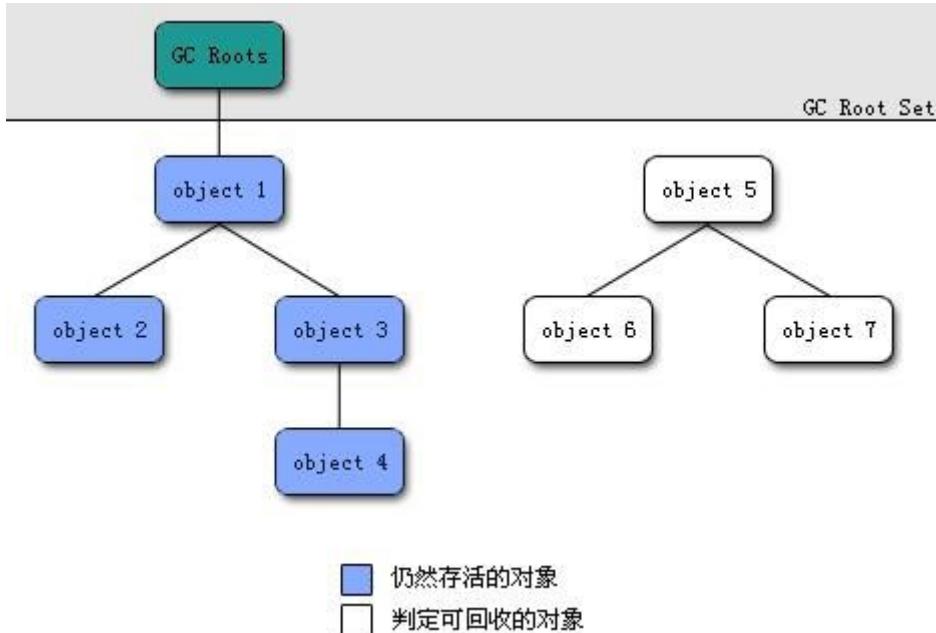
- 无法检测出循环引用，造成内存泄露。因此主流的虚拟机中没有选择这个算法来管理内存。

如下例：object1和object2 互相引用对方，导致它们的引用计数器都不为 0，于是引用计数算法无法通知 GC 回收器回收他们。

```
1 public class ReferenceCountingGc {  
2     Object instance = null; //成员变量是一个Object  
3     public static void main(String[] args) {  
4         ReferenceCountingGc objA = new ReferenceCountingGc();  
5         ReferenceCountingGc objB = new ReferenceCountingGc();  
6         objA.instance = objB;  
7         objB.instance = objA;  
8         objA = null;  
9         objB = null;  
10    }  
11 }
```

可达性分析算法

以 GC Roots 为起始点进行搜索，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的话，则证明此对象是不可用的。即：可达的对象都是存活的，不可达的对象可被回收。



可以作为GC Root的对象

2020美团一面问到，没答上来。

- 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
- 方法区中常量引用的对象 final对象
- 方法区中类静态属性引用的对象 static对象
- 所有被同步锁持有的对象
- 活跃线程的引用对象
- JVM自身持有的对象，比如系统类加载器等
- 通过System Class Loader或者Boot Class Loader加载的class对象，通过自定义类加载器加载的class不一定是GC Root

finalize方法的作用

补充一句：并不提倡在程序中调用finalize()来进行自救。建议忘掉Java程序中该方法的存在。因为它执行的时间不确定，甚至是否被执行也不确定（Java程序的不正常退出），而且运行代价高昂，无法保证各个对象的调用顺序（甚至有不同线程中调用）。在Java9中已经被标记为 **deprecated**，且java.lang.ref.Cleaner（也就是强、软、弱、幻象引用的那一套）中已经逐步替换掉它，会比 finalize来的更加的轻量及可靠。

不可达的对象并非“非死不可”

即使在可达性分析法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑阶段”，要真正宣告一个对象死亡，至少要经历两次标记过程：可达性分析法中不可达的对象被第一次标记；并且进行一次筛选，如果对象覆盖了 finalize 方法，且 finalize 方法没有被虚拟机调用过时，就会被放置在F-Queue中，并在稍后由虚拟机自动建立的低优先级的finalize线程去执行触发finalize方法（该线程运行优先级低，因此该方法随时可能被终止）。

给对象创造最后一次逃脱死亡的机会。

代码示例链接

```
1  /**
2   * finalize()使用示例 在gc之后，由于finalize()方法中重新赋值finalization指向自己，
3   * 对象没有被销毁。
4  */
5  public class Finalization {
6      public static Finalization finalization;
7
8      @Override
9      protected void finalize() {
10         System.out.println("Finalized");
11         finalization = this;
12     }
13
14     public static void main(String[] args) {
15         Finalization f = new Finalization();
16         System.out.println("First print: " + f);
17         f = null;
18         System.gc();
19         try {// 休息一段时间，让上面的垃圾回收线程执行完成
20             Thread.sleep(1000);
21         } catch (InterruptedException e) {
22             e.printStackTrace();
23         }
24         System.out.println("Second print: " + f);
25         System.out.println(f.finalization);
26     }
27 }
```

为什么必须用static类型的成员变量，去掉static就报空指针了呢？

finalize()在java9之后被Deprecated了。

由于finalize()运行的不确定性较大，没办法保证调用顺序，且运行代价高昂，因此不建议使用finalize()。

方法区的回收

因为方法区主要存放永久代对象，而永久代对象的回收率比新生代低很多，所以在方法区上进行回收性价比不高。

主要是对常量池的回收和对类的卸载（主要）。

如何判断一个常量是废弃常量

todo总结

运行时常量池主要回收的是废弃的常量。那么，我们如何判断一个常量是废弃常量呢？

假如在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话，"abc" 就会被系统清理出常量池。

注意：我们在 [可能是把 Java 内存区域讲的最清楚的一篇文章](#) 也讲了 JDK1.7 及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。

如何判断一个类是无用的类

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。

Java 中对象的引用方式

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收都与引用有关。

Java 提供了四种强度不同的引用类型。级别由高到低依次为强引用、软引用、弱引用、虚引用。

1. 强引用 对象的一般状态

被强引用关联的对象不会被回收，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。

使用 new 一个新对象的方式来创建强引用。

```
1 Object obj = new Object();
```

2. 软引用 对象缓存

被软引用关联的对象只有在内存不够的情况下才会被回收。可用来实现高速缓存，会大大降低OOM的概率。

使用 SoftReference 类来创建软引用。

```
1 Object obj = new Object();
2 SoftReference<Object> sf = new SoftReference<Object>(obj);
3
4 obj = null; // 使对象只被软引用关联 obj指向null 而之前地址的对象被软引用关联了
```

3. 弱引用 对象缓存

被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前。

使用 WeakReference 类来创建弱引用。

```
1 Object obj = new Object();
2 WeakReference<Object> wf = new WeakReference<Object>(obj);
3
4 obj = null;
```

4. 虚引用 哨兵

为一个对象设置虚引用的唯一目的是能在这个对象被回收时收到一个系统通知。
又称为幽灵引用或者幻影引用，虚引用并不会决定对象的生命周期。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。

跟踪对象被垃圾收集器回收的活动，起哨兵作用。

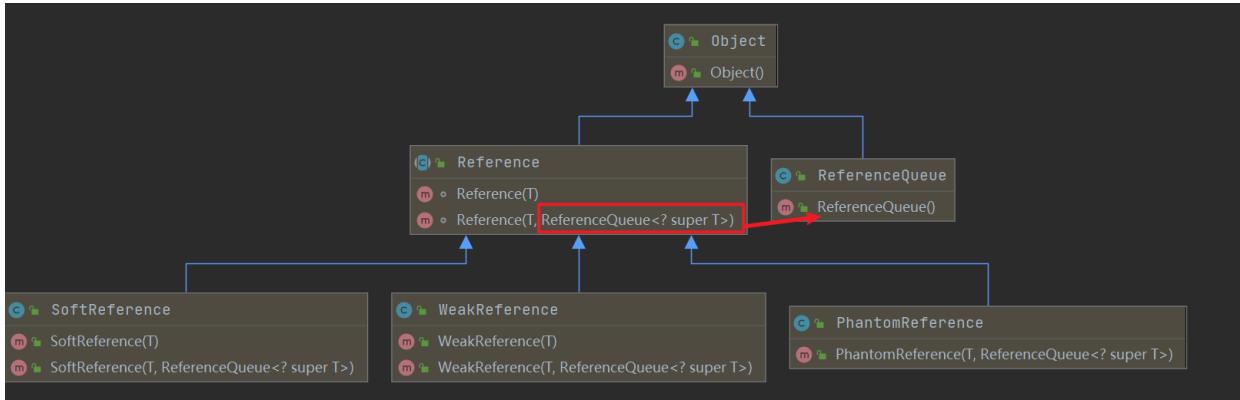
必须和引用队列 ReferenceQueue 联合使用。GC 在回收对象时，如果发现对象有虚引用，在回收之前，会首先把对象的虚引用加入到与之关联的队列中，**程序可以通过判断引用队列是否已经加入虚引用，来了解被引用的对象是否被GC回收。因此起到了哨兵作用。**

```
1 Object obj = new Object();
2 ReferenceQueue queue = new ReferenceQueue<>();
3 PhantomReference<Object> pf = new PhantomReference<Object>(obj, queue);
4
5 obj = null;
```

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 (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。

ReferenceQueue的作用

GC完成之后，存储除强引用以外的与之关联的引用对象。



ReferenceQueue可以结合SoftReference、WeakReference、PhantomReference使用，构造方法可以传入ReferenceQueue。GC在回收对象时，如果发现对象有虚引用，在回收之前，会首先把对象的引用加入到与之关联的队列中。

如果没有ReferenceQueue，就必须不断轮询Reference对象，通过reference.get()获取对象，不断判断返回是否为null来判断关联的对象是否被回收。PhantomReference.get()始终为null，因此这种方法不适用于虚引用，这也是为什么虚引用只有带有ReferenceQueue这一种构造函数。

特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为**软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出(OutOfMemory)等问题的产生。**

垃圾回收算法

标记-清除算法

分为“标记”和“清除”阶段

在标记阶段，用**可达性分析算法**检查每个对象是否存活，对存活对象头部打上标记。

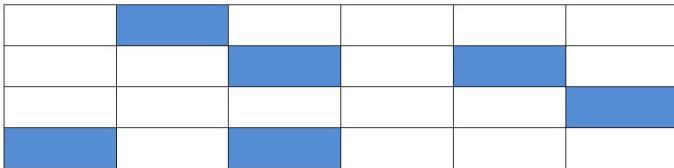
在清除阶段，对堆内存进行线性遍历，回收不可达对象并取消可达对象标志位，用于下一次标记清除。

以链表的形式记录空闲内存块，在分配时，程序会搜索空闲链表寻找空间大于等于新对象大小 size 的块 block。

内存整理前



内存整理后



可用内存 可回收内存 存活对象

不足

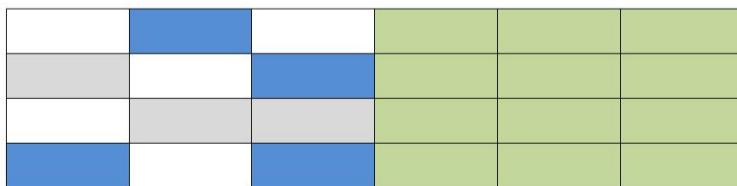
标记和清除过程效率都不高；

会产生大量不连续的内存碎片，导致无法给大对象分配内存。

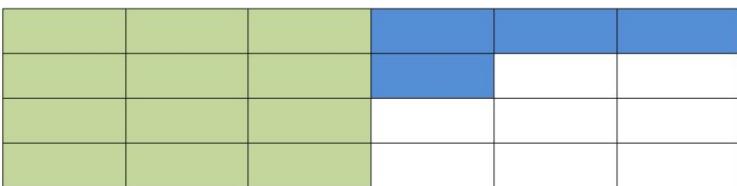
复制算法

将内存划分为大小相等的两块，每次只使用其中一块，当这一块内存用完了就将还存活的对象复制到另一块上面，然后再把使用过的内存空间进行一次清理。

内存整理前



内存整理后



可用内存 可回收内存 存活对象 保留内存

+ 解决了碎片化问题

+ 顺序分配内存，简单高效

- 只使用了一半的内存

适用于对象存活率低的场景：用于年轻代，年轻代中的对象基本每次回收只有10%左右的存活，需要复制的对象很少。

现在的商业虚拟机都采用这种收集算法回收新生代，但是并不是划分为大小相等的两块，而是一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。在回收时，将 Eden 和 Survivor 中还活着的对象全部复制到另一块 Survivor 上，最后清理 Eden 和使用过的那一块 Survivor。

HotSpot 虚拟机的 Eden 和 Survivor 大小比例默认为 8:1，保证了内存的利用率达到 90%。**如果每次回收有多于 10% 的对象存活**，那么一块 Survivor 就不够用了，此时需要依赖于老年代进行空间分配担保，也就是借用老年代的空间存储放不下的对象。

复制算法不适用于对象存活率较高的情景，比如老年代，要进行较多的复制操作，效率低，还得常备 50% 的内存避免全部复制的情况。这种情况下提出了标记整理算法。

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。



- + 解决了碎片化问题
- + 不需要设置两块内存互换

适用于对象存活率高的场景：用于老年代。

分代收集算法

现在的商业虚拟机采用分代收集算法，它根据对象存活周期将内存划分为几块，不同块采用适当的收集算法。

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。

在新生代中，对象存活几率低，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。

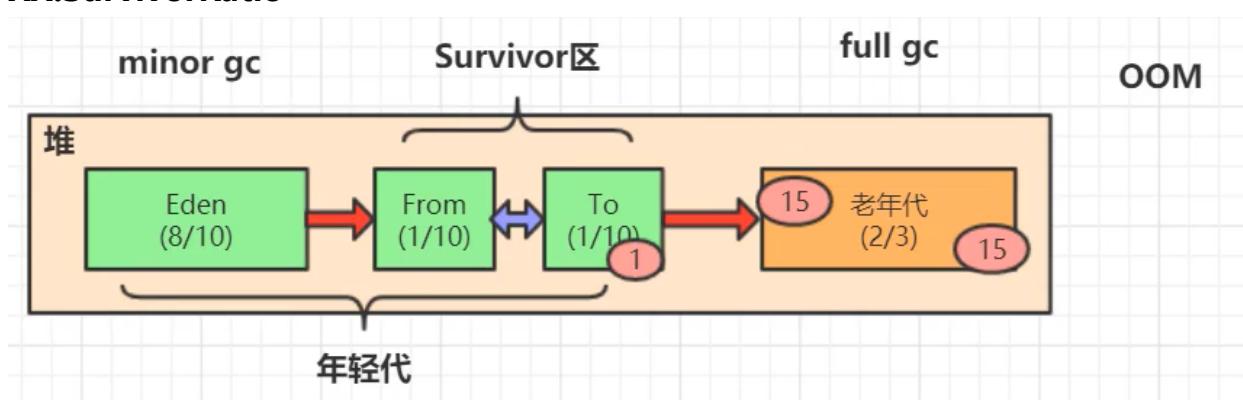
Java堆内存分配与回收策略

Minor GC 和 Full GC

- Minor GC：回收新生代，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般也会比较快。
- Full GC：回收老年代和新生代，老年代对象其存活时间长，因此 Full GC 很少执行，执行速度会比 Minor GC 慢很多。

堆内存分为老年代和年轻代，默认比例是2: 1。-XX:NewRatio

年轻代分为Eden区和两个Survivor区(from区和to区)，默认比例是8: 1: 1。 -XX:SurvivorRatio



Minor GC

1. 当Eden区满的时候，会触发第一次Minor gc，把还活着的对象拷贝到Survivor From区；当Eden区再次触发Minor gc的时候，会扫描Eden区和From区，对两个区域进行垃圾回收，经过这次回收后还存活的对象，则直接复制到To区域，并将Eden区和From区清空。
2. 当后续Eden区又发生Minor gc的时候，会对Eden区和To区进行垃圾回收，存活的对象复制到From区，并将Eden区和To区清空
3. 部分对象会在From区域和To区域中复制来复制去，如此交换15次(由JVM参数**MaxTenuringThreshold**决定，这个参数默认是15)，最终如果还存活，就存入老年代。

为什么要分为Eden和Survivor?

- 如果没有Survivor，Eden区每进行一次Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发Full GC。老年代的内存空间远大于新生代，进行一次Full GC消耗的时间比Minor GC长得多，所以需要分为Eden和Survivor。
- Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生，Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。

为什么要设置两个Survivor区？

即复制算法的好处：解决碎片化。

内存分配策略

1. 对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 上分配，当 Eden 空间不够时，发起 Minor GC。

2. 大对象直接进入老年代

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。

除了Survivor区放不下的对象会直接进入老年代，还可以通过设置-

XX:PretenureSizeThreshold，大于此值的对象直接在老年代分配，避免在 Eden 和 Survivor 之间的大量内存复制。

3. 长期存活的对象进入老年代

为对象定义年龄计数器，对象在 Eden 出生并经过 Minor GC 依然存活，将移动到 Survivor 中，年龄就增加 1 岁，增加到一定年龄则移动到老年代中。

web应用中：线程池，静态变量引用的对象，spring容器里的bean，service，代码初始化的缓存对象，都一直存活，会放在老年代。

-XX:MaxTenuringThreshold 用来定义年龄的阈值。

4. 动态对象年龄判定

虚拟机并不是永远要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 空间的一半，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄。

5. 空间分配担保

在发生 Minor GC 之前，虚拟机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，如果条件成立的话，那么 Minor GC 可以确认是安全的。

如果不成立的话虚拟机会查看 HandlePromotionFailure 的值是否允许担保失败，如果允许那么就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，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 的值不允许冒险，那么就要进行一次 Full GC。

Full GC 的触发条件

对于 Minor GC，其触发条件非常简单，当 Eden 空间满时，就将触发一次 Minor GC。而 Full GC 则相对复杂，有以下条件：

1. 调用 System.gc()

只是建议虚拟机执行 Full GC，但是虚拟机不一定真正去执行。不建议使用这种方式，而是让虚拟机管理内存。

2. 老年代空间不足

老年代空间不足的常见场景为前文所讲的大对象直接进入老年代、长期存活的对象进入老年代等。

为了避免以上原因引起的 Full GC，应当尽量不要创建过大的对象以及数组。除此之外，可以通过 -Xmn 虚拟机参数调大新生代的大小，让对象尽量在新生代被回收掉，不进入老年代。还可以通过 -XX:MaxTenuringThreshold 调大对象进入老年代的年龄，让对象在新生代多存活一段时间。

3. 空间分配担保失败

空间分配担保：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，如果担保失败会执行一次 Full GC。

4. JDK 1.7 及以前的永久代空间不足

在 JDK 1.7 及以前，HotSpot 虚拟机中的方法区是用永久代实现的，永久代中存放的为一些 Class 的信息、常量、静态变量等数据。

当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用 CMS GC 的情况下也会执行 Full GC。如果经过 Full GC 仍然回收不了，那么虚拟机会抛出 java.lang.OutOfMemoryError。

为避免以上原因引起的 Full GC，可采用的方法为增大永久代空间或转为使用 CMS GC。

JDK1.8之后取消永久代，也就没有这个问题了。

5. CMS GC的Promotion failed 和 Concurrent Mode Failure

执行 CMS GC 的过程中同时有对象要放入老年代，而此时老年代空间不足（可能是 GC 过程中浮动垃圾过多导致暂时性的空间不足），便会报 Concurrent Mode Failure 错误，并触发 Full GC。

Promotion failed-minor GC时survivor区域放不下，只能放到老年代，此时老年代也放不下。

垃圾回收器

- **Serial**既不能并发，也不能并行。
- **Parallel**和**ParNew**都可以并行，**Parallel**注重吞吐量。
- **CMS**并发，采用标记清除。
- **G1**并发，标记整理，且有可预测的停顿。
- **Serial**和**Parallel**有Old，**Serial**、**ParNew**可以和**CMS**结合(**Parallel**和**G1**没有用传统GC代码框架)，**G1**新老都自己抗。
- [**未来还有ZGC**](#)

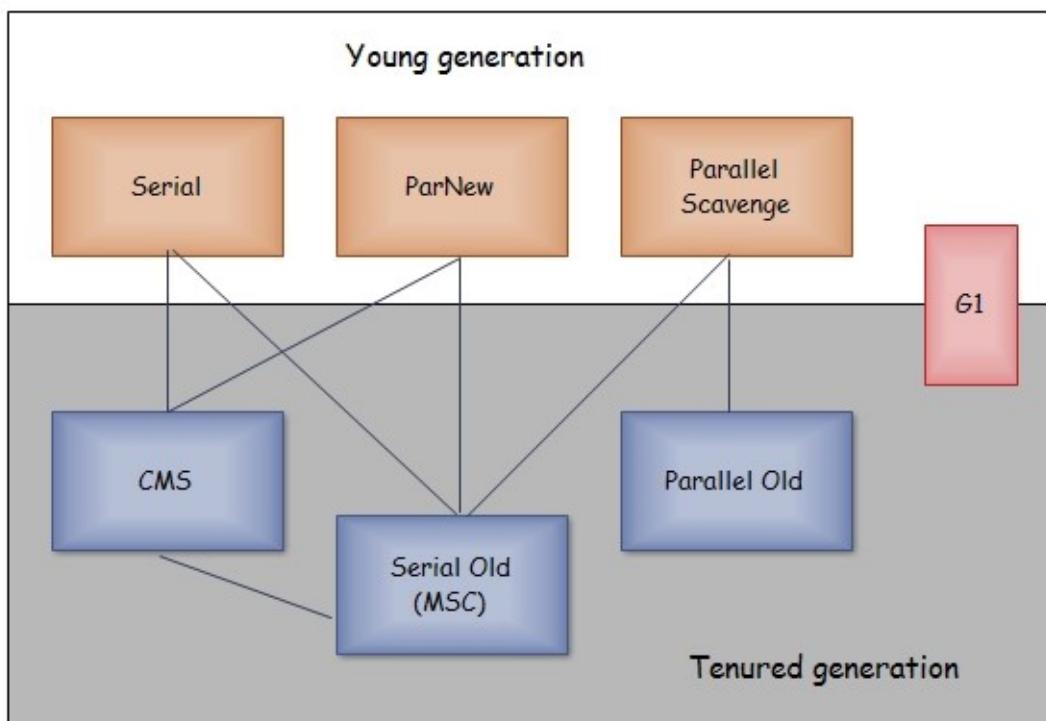
jdk1.7 默认垃圾收集器Parallel Scavenge (新生代) + Parallel Old (老年代)

jdk1.8 默认垃圾收集器Parallel Scavenge (新生代) + Parallel Old (老年代)

jdk1.9 默认垃圾收集器G1

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。

不存在哪个好哪个坏的垃圾收集器，没有万能的垃圾收集器，只有适合于具体应用场景的垃圾收集器。



以上是 HotSpot 虚拟机中的 7 个垃圾收集器，连线表示垃圾收集器可以配合使用。

- **并行 (Parallel)**：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- **并发 (Concurrent)**：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。 **只有 CMS 和 G1**

STW——stop the world

JVM在GC的时候，停止用户线程，专心去垃圾收集。给用户的体验可能是卡死。
任何一种GC收集器中都会发生。

多数GC优化通过减少stop the world发生的时间来提高程序性能。

为什么要STW？

如果GC过程中用户线程也在运行，GC过程中刚找过的不是垃圾的，可能被用户线程清理掉，又变成了垃圾对象，GC的过程情况很复杂，出现了各种乱七八糟的工作，因此Java GC过程中会把用户线程先停掉，专心做GC。

SafePoint 了解

分析过程中对象引用关系不会发生变化的点。一般在方法调用；循环跳转；异常跳转的地方。

一旦GC发生，会让所有线程都跑到安全点再停顿下来。

安全点的数量要适中，太少会让GC等待太久；太多会让程序增加很多负荷。

JVM的运行模式 了解

- Client
- Server

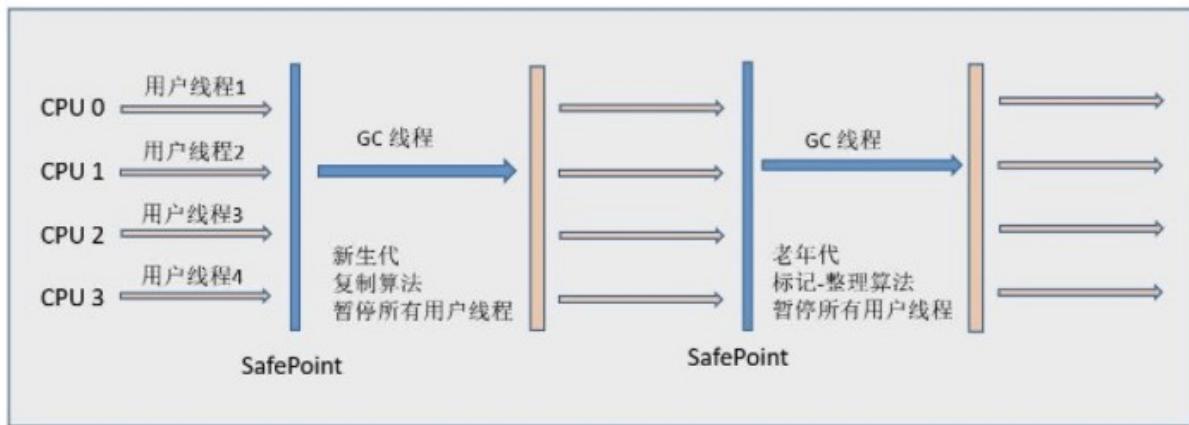
Server模式采用重量级虚拟机，对程序有更多的优化。Client模式采用轻量级的虚拟机。

Client模式启动快，Server启动较慢。但运行之后进入稳定期，Server模式更快。

java -version 就可以查看

```
1 $ java -version
2 java version "11.0.10" 2021-01-19 LTS
3 Java(TM) SE Runtime Environment 18.9 (build 11.0.10+8-LTS-162)
4 Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.10+8-LTS-162, mixed mode)
```

1. Serial收集器和Serial Old收集器



-XX:+UseSerialGC -XX:+UseSerialOldGC

程序启动时用-XX:+UseSerialGC，就可以切换成这种。

Serial 翻译为串行，单线程串行的垃圾回收器，只会使用一个线程进行垃圾收集工作，并且进行垃圾回收时，必须暂停所有工作线程。

优点是简单高效，在单个 CPU 环境下，由于没有线程交互的开销，因此拥有最高的单线程收集效率。

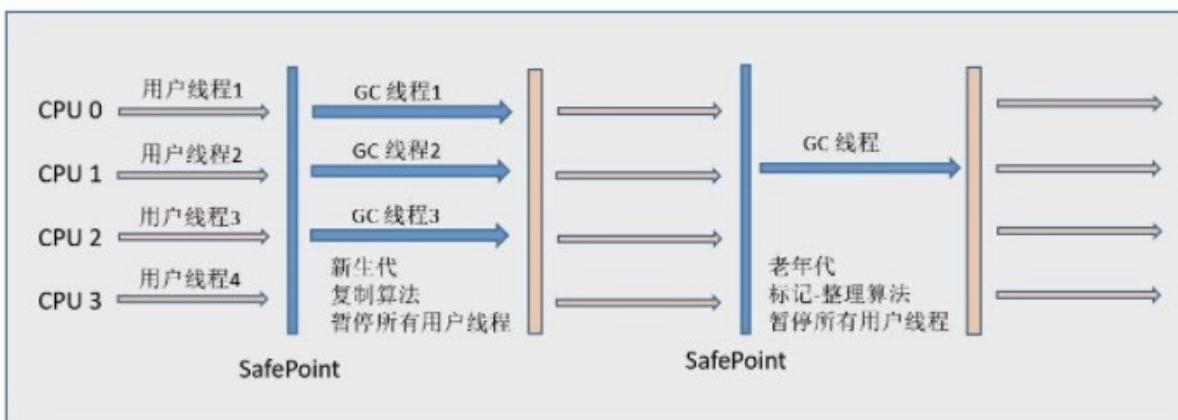
是 Client 场景下的默认新生代收集器，因为在该场景下内存一般来说不会很大。它收集一两百兆垃圾的停顿时间可以控制在一百多毫秒以内，只要不是太频繁，这点停顿时间是可以接受的。

如果用在 Server 场景下，Serial Old 收集器有两大用途：

作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。

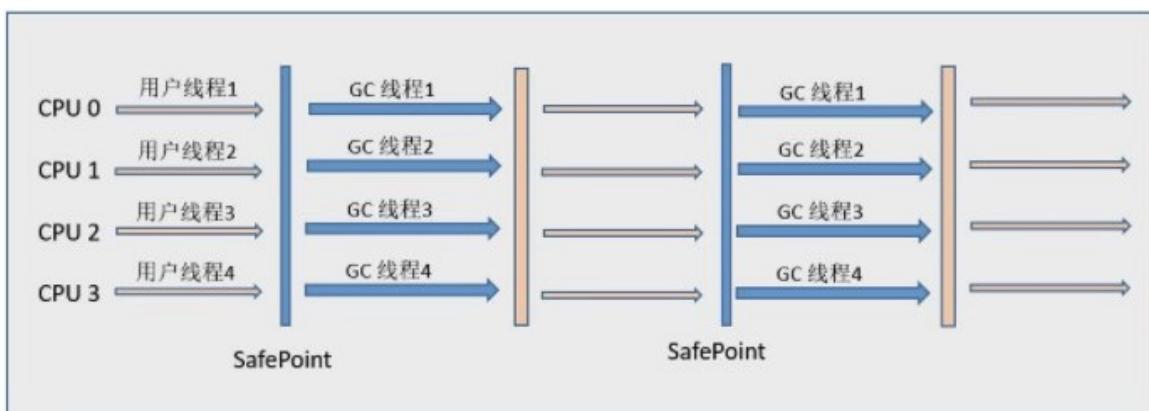
在 JDK 1.5 以及之前版本（Parallel Old 诞生以前）中与 Parallel Scavenge 收集器搭配使用。诞生以后就不这么组合了，之前是没办法的办法。

2. ParNew 收集器 -XX:+UseParNewGC



它是 Serial 收集器的多线程版本（只用关注上图的左半部分），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。单核执行效率不如Serial，但多核下效率更高。默认CPU开启数量与CPU数量相同。它是 Server 场景下默认的新生代收集器，除了性能原因外，主要是因为**除了 Serial 收集器，只有它能与 CMS 收集器配合使用。**

3. Parallel Scavenge收集器和Parallel Old收集器



-XX:+UseParallelGC -XX:+UseParallelOldGC

Parallel并行的。与 ParNew 一样是多线程收集器。

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）。而 Parallel Scavenge 收集器**关注点是吞吐量**（高效率的利用 CPU）。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**。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验。而高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，适合在后台运算而不需要太多交互的任务。

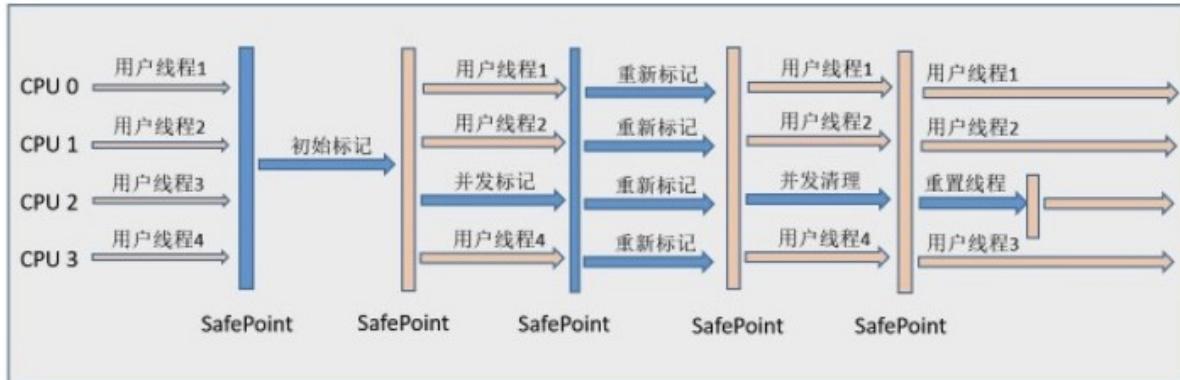
缩短停顿时间是以牺牲吞吐量和新生代空间来换取的：新生代空间变小，垃圾回收变得频繁，导致吞吐量下降。

启动时可以通过一个开关参数打开 GC 自适应的调节策略（**-XX:+UseAdaptiveSizePolicy**），就不需要手工指定新生代的大小（-Xmn）、Eden 和 Survivor 区的比例、晋升老年对象年龄等细节参数了。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量。

在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

4. CMS收集器

CMS (Concurrent Mark Sweep) , **Mark Sweep** 指的是标记 - 清除算法。



- **初始标记**: 仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，**需要 stop the world.**
- **并发标记**: 并发进行 GC Roots Tracing 的过程，它在整个回收过程中耗时最长，不需要停顿。
- 并发预清理：查找执行并发标记阶段从年轻代晋升到老年代的对象。
- **重新标记**: 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，**需要stop the world.**
- **并发清理**: 不需要停顿。
- **重发重置**: 重置CMS的数据结构，准备下一次垃圾清理。

缺点：

吞吐量低: 低停顿时间是以牺牲吞吐量为代价的，导致 CPU 利用率不够高。

无法处理浮动垃圾，可能出现 Concurrent Mode Failure。浮动垃圾是指并发清除阶段由于用户线程继续运行而产生的垃圾，这部分垃圾只能到下一次 GC 时才能进行回收。由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。

标记 - 清除算法导致的空间碎片，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象，不得不提前触发一次 Full GC。

5. G1 (Garbage-First) 垃圾收集器

G1 (Garbage-First) , 它是一款面向服务端应用的垃圾收集器，在多 CPU 和大内存的场景下有很好的性能。HotSpot 开发团队赋予它的使命是未来可以替换掉 CMS 收集器。

具备如下优点

并行与并发: G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU (CPU 或者 CPU 核心) 来缩短 Stop-The-World 停顿时间。部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 java 程序继续执行。

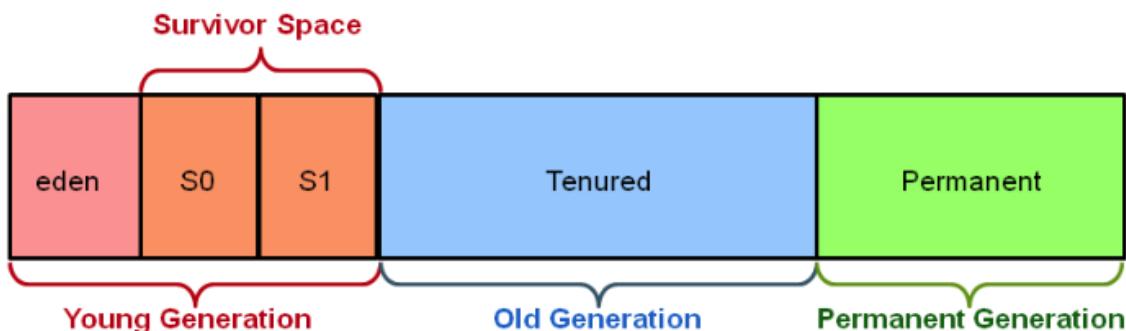
可预测的停顿: 这是 G1 相对于 CMS 的另一个大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立**可预测的停顿时间模型**，能让使用者明确指定在一个长度为 M 毫秒的时间片段内。

分代收集: 虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但是还是保留了分代的概念。

空间整合: 与 CMS 的“标记-清理”算法不同，G1 从整体来看是基于“标记-整理”算法实现的收集器；从局部上来看是基于“**标记-复制**”算法实现的。

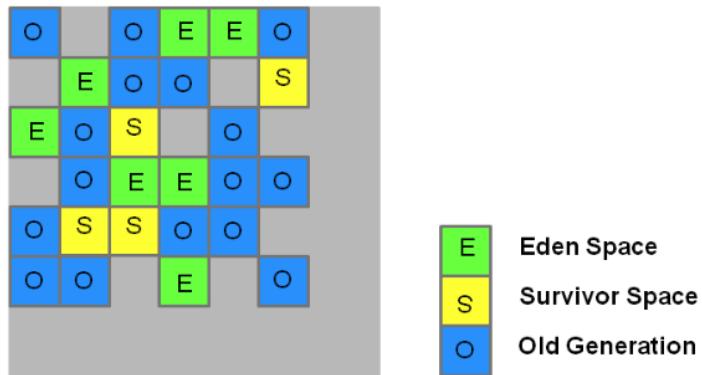
堆被分为新生代和老年代，其它收集器进行收集的范围都是整个新生代或者老年代，而 G1 可以直接对新生代和老年代一起回收。

Hotspot Heap Structure



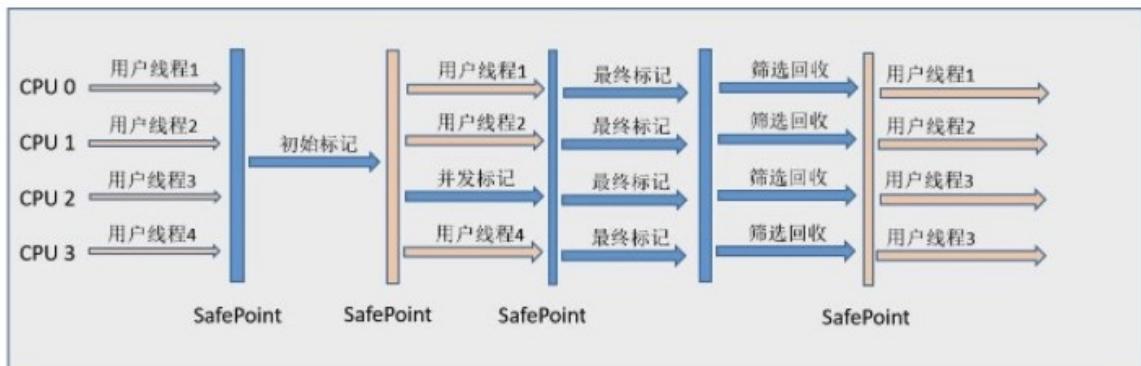
G1 把堆划分成多个大小相等的独立区域 (Region)，新生代和老年代不再物理隔离。

G1 Heap Allocation



通过引入 Region 的概念，从而将原来的一整块内存空间划分成多个的小空间，使得每个小空间可以单独进行垃圾回收。这种划分方法带来了很大的灵活性，使得**可预测的停顿时间**模型成为可能。通过记录每个 Region 垃圾回收时间以及回收所获得的空间（这两个值是通过过去回收的经验获得），并维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region。

每个 Region 都有一个 Remembered Set，用来记录该 Region 对象的引用对象所在的 Region。通过使用 Remembered Set，在做可达性分析的时候就可以避免全堆扫描。



如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤：

- 初始标记
- 并发标记
- 最终标记：为了修正在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标记记录，虚拟机将这段时间对象变化记录在线程的 Remembered Set

Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中。这阶段需要停顿线程

- 筛回收：首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。

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）。

CMS vs G1

CMS一般用于老年代，G1全可用，都注重响应时间

最主要的区别是老年代算法不同，CMS是标记清除，而G1是标记整理。CMS标记-清除算法会导致收集结束时会有大量空间碎片产生。

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)

JVM性能调优

性能调优

性能调优包含多个层次，比如：架构调优、代码调优、JVM调优、数据库调优、操作系统调优等。

架构调优和代码调优是JVM调优的基础，其中架构调优是对系统影响最大的。

性能调优基本上按照以下步骤进行：明确优化目标、发现性能瓶颈、性能调优、通过监控及数据统计工具获得数据、确认是否达到目标。

何时进行JVM调优

遇到以下情况，就需要考虑进行JVM调优了：

- Heap内存（老年代）持续上涨达到设置的最大内存值；
- Full GC 次数频繁；
- GC 停顿时间过长（超过1秒）；
- 应用出现OutOfMemory 等内存异常；
- 应用中有使用本地缓存且占用大量内存空间；

- 系统吞吐量与响应性能不高或下降。

JVM调优的基本原则

JVM调优是一个手段，但并不一定所有问题都可以通过JVM进行调优解决，因此，在进行JVM调优时，我们要遵循一些原则：

- 大多数的Java应用不需要进行JVM优化；
- 大多数导致GC问题的原因是代码层面的问题导致的（代码层面）；
- 上线之前，应先考虑将机器的JVM参数设置到最优；
- 减少创建对象的数量（代码层面）；
- 减少使用全局变量和大对象（代码层面）；
- 优先架构调优和代码调优，JVM优化是不得已的手段（代码、架构层面）；
- 分析GC情况优化代码比优化JVM参数更好（代码层面）；

通过以上原则，我们发现，其实最有效的优化手段是架构和代码层面的优化，而JVM优化则是最后不得已的手段，也可以说是对服务器配置的最后一次“压榨”。

JVM调优目标

调优的最终目的都是为了令应用程序使用最小的硬件消耗来承载更大的吞吐。jvm调优主要是针对垃圾收集器的收集性能优化，令运行在虚拟机上的应用能够使用更少的内存以及延迟获取更大的吞吐量。

- 延迟：GC低停顿和GC低频率；
- 低内存占用；
- 高吞吐量；

其中，任何一个属性性能的提高，几乎都是以牺牲其他属性性能的损为代价的，不可兼得。具体根据在业务中的重要性确定。

JVM调优量化目标

下面展示了一些JVM调优的量化目标参考实例：

- Heap 内存使用率 <= 70%;
- Old generation内存使用率<= 70%;
- avg pause <= 1秒;
- Full gc 次数0 或 avg pause interval >= 24小时；

注意：不同应用的JVM调优量化目标是不一样的。

JVM调优的步骤

一般情况下，JVM调优可通过以下步骤进行：

- 分析GC日志及dump文件，判断是否需要优化，确定瓶颈问题点；

- 确定JVM调优量化目标；
- 确定JVM调优参数（根据历史JVM参数来调整）；
- 依次调优内存、延迟、吞吐量等指标；
- 对比观察调优前后的差异；
- 不断的分析和调整，直到找到合适的JVM参数配置；
- 找到最合适的参数，将这些参数应用到所有服务器，并进行后续跟踪。

以上操作步骤中，某些步骤是需要多次不断迭代完成的。一般是从满足程序的内存使用需求开始的，之后是时间延迟的要求，最后才是吞吐量的要求，要基于这个步骤来不断优化，每一个步骤都是进行下一步的基础，不可逆行之。

JVM 配置常用参数

1. 堆参数；
2. 回收器参数；
3. 项目中常用配置；
4. 常用组合；

堆参数

`-XX:MaxTenuringThreshold=15`：设置垃圾最大年龄。

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，元空间使用的是直接内存。

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，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。

回收器参数

如上表所示，目前主要有串行、并行和并发三种，对于大内存的应用而言，串行的性能太低，因此使用到的主要是并行和并发两种。并行和并发 GC 的策略通过 `UseParallelGC` 和 `UseConcMarkSweepGC` 来指定，还有一些细节的配置参数用来配置策略的执行方式。例如：`XX:ParallelGCThreads`, `XX:CMSInitiatingOccupancyFraction` 等。通常：Young 区对象回收只可选择并行（耗时间），Old 区选择并发（耗 CPU）。

项目中常用配置

备注：在Java8中永久代的参数`-XX:PermSize` 和`-XX: MaxPermSize`已经失效。

常用组合

GC记录

为了严格监控应用程序的运行状况，我们应该始终检查JVM的垃圾回收性能。最简单的方法是以人类可读的格式记录GC活动。

使用以下参数，我们可以记录GC活动：

```
-XX:+UseGCLogFileRotation  
-XX:NumberOfGCLogFiles=< number of log files >  
-XX:GCLogFileSize=< file size >[ unit ]  
-Xloggc:/path/to/gc.log
```

常用 GC 调优策略

1. GC 调优原则；
2. GC 调优目的；
3. GC 调优策略；

GC 调优原则

在调优之前，我们需要记住下面的原则：

多数的 Java 应用不需要在服务器上进行 GC 优化； 多数导致 GC 问题的 Java 应用，都不是因为我们参数设置错误，而是代码问题； 在应用上线之前，先考虑将机器的 JVM 参数设置到最优（最适合）； 减少创建对象的数量； 减少使用全局变量和大对象； GC 优化是到最后不得已才采用的手段； 在实际使用中，分析 GC 情况优化代码比优化 GC 参数要多得多。

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：大对象直接进入老年代，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 初始堆大小，-Xmx 最大堆大小。默认设置成相等

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。

推荐阅读

- [CMS GC 默认新生代是多大？](#)
- [CMS GC启动参数优化配置](#)
- [从实际案例聊聊Java应用的GC优化-美团技术团队](#)
- [JVM参数使用手册](#)
- [JVM性能调优详解](#)

JVM虚拟机调优 何时进行JVM调优？

- Heap内存（老年代）持续上涨达到设置的最大内存值；
- Full GC 次数频繁；
- GC 停顿时间过长（超过1秒）；
- 应用出现OutOfMemory 等内存异常；
- 应用中有使用本地缓存且占用大量内存空间；
- 系统吞吐量与响应性能不高或下降。

STW——stop the world

full GC的时候，会把用户线程先停掉，专心去垃圾收集。给用户的体验可能是卡死。

为什么要STW？如果GC过程中用户线程也在运行，GC过程中刚找过的不是垃圾的，可能被用户线程清理掉，又变成了垃圾对象，GC的过程情况很复杂，出现了各种乱七八糟的工作，因此Java GC过程中会把用户线程先停掉，专心做GC。

Java虚拟机调优最主要的目的：减少STW的时间，也就是减少用户停顿的时间。——减少full GC——STW发生的次数 以及每次GC的时间

正常情况下，应该几天甚至几周才做一次JVM full GC。

JVM的参数都是可以设置的，堆，元空间等等多大空间都可以改

对象头大小很小，基本不用考虑，根据对象的字段占用大小的和来估计对象的大小。

JVM调优的基本原则

JVM调优是一个手段，但并不一定所有问题都可以通过JVM进行调优解决，因此，在进行JVM调优时，我们要遵循一些原则：

- 大多数的Java应用不需要进行JVM优化；
- 大多数导致GC问题的原因是代码层面的问题导致的（代码层面）；
- 上线之前，应先考虑将机器的JVM参数设置到最优；
- 减少创建对象的数量（代码层面）；
- 减少使用全局变量和大对象（代码层面）；
- 优先架构调优和代码调优，JVM优化是不得已的手段（代码、架构层面）；
- 分析GC情况优化代码比优化JVM参数更好（代码层面）；

通过以上原则，我们发现，其实最有效的优化手段是架构和代码层面的优化，而JVM优化则是最后不得已的手段，也可以说是对服务器配置的最后一次“压榨”。

JVM调优的步骤

一般情况下，JVM调优可通过以下步骤进行：

- 分析GC日志及dump文件，判断是否需要优化，确定瓶颈问题点；
- 确定JVM调优量化目标；
- 确定JVM调优参数（根据历史JVM参数来调整）；
- 依次调优内存、延迟、吞吐量等指标；
- 对比观察调优前后的差异；
- 不断的分析和调整，直到找到合适的JVM参数配置；
- 找到最合适的参数，将这些参数应用到所有服务器，并进行后续跟踪。

以上操作步骤中，某些步骤是需要多次不断迭代完成的。一般是从满足程序的内存使用需求开始的，之后是时间延迟的要求，最后才是吞吐量的要求，要基于这个步骤来不断优化，每一个步骤都是进行下一步的基础，不可逆行之。

JVM参数 IDEA edit configuration-VM options中输入

非堆内存分配：

-Xss 设置每个线程可使用的内存大小，即栈的大小。在相同物理内存下，减小这个值能生成更多的线程。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

线程栈的大小是个双刃剑，如果设置过小，可能会出现栈溢出，特别是在该线程内有递归、大的循环时出现溢出的可能性更大，如果该值设置过大，就有影响到创建栈的数量，如果是多线程的应用，就会出现内存溢出的错误。

-Xms:初始堆大小

-Xmx:最大堆大小

-Xmn:新生代大小

-XX:NewRatio:设置新生代和老年代的比值。例如：-XX:NewRatio=4，表示新生代:老年代=1:4，即新生代占整个堆的1/5。在Xms=Xmx并且设置了Xmn的情况下，该参数不需要进行设置。

-XX:SurvivorRatio:新生代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：默认为8，表示Eden: Survivor=8: 2 8: 1: 1

-XX:NewSize --- 设置年轻代大小

-XX:MaxNewSize --- 设置年轻代最大值

-XX:MaxTenuringThreshold:设置转入老年代的存活次数。如果是0，则直接跳过新生代进入老年代

-XX:PermSize、-XX:MaxPermSize:分别设置永久代最小大小与最大大小
(Java8以前)

-XX:MetaspaceSize、-XX:MaxMetaspaceSize:分别设置元空间最小大小与最大大小 (Java8以后)

收集器设置

默认G1

-XX:+UseSerialGC:设置串行收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParallelOldGC:设置并行老年代收集器

-XX:+UseParNewGC 可与CMS收集同时使用 JDK1.5+默认是这种

-XX:+UseConcMarkSweepGC:设置并发收集器

垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

并行收集器设置

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间

-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为
 $1/(1+n)$

并发收集器设置

-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。

-XX:ParallelGCThreads=n:设置并发收集器新生代收集方式为并行收集时，使用的CPU数。并行收集线程数。

推荐阅读

为什么要读JVM规范

《Java虚拟机规范》目录，我们可以看到规范分为下面几个部分：

- 第1章 引言
- 第2章 Java虚拟机结构
 - 介绍了Class文件格式、数据类型、原始类型、引用类型、运行时数据区、栈帧、字节码指令等。
- 第3章 为Java虚拟机编译
- 第4章 Class文件格式
- 第5章 加载、链接与初始化
- 第6章 Java虚拟机指令集
 - 可以把JVM理解成一台机器，它有自己的指令集与助记符，类似于Arm, x86的指令集。
- 第7章 操作码助记符

2.11.2 加载和存储指令

加载和存储指令用于将数据从栈帧 (§2.6) 的局部变量表 (§2.6.1) 和操作数栈之间来回传输 (§2.6.2)

2.12 类库

(JRE)由JVM、类库、核心文件组成。

Java 虚拟机必须对不同平台下 Java 类库的实现提供充分的支持，因为其中有一些类库如果

没有 Java 虚拟机的支持的话是根本无法实现的。

可能需要 Java 虚拟机特殊支持的类库包括有：

- 反射，譬如在 java.lang.reflect 包中的各个类和 java.lang.Class 类 第 50 页 / 共 387 页
- 类和接口的加载和创建，最显而易见的例子就是 java.lang.ClassLoader 类
- 类和接口的链接和初始化，上一点的例子也适用于这点
- 安全，譬如在 java.security 包中的各个类和 java.lang.SecurityManager 等其他类
- 多线程，譬如 java.lang.Thread 类
- 弱引用，譬如在 java.lang.ref 包中的各个类

上面列举的几点意在简单说明一下而不是详细介绍这些类库，介绍这些类库和功能的详细信息

已经超出了本书的范围，如果读者想了解这些类库，阅读 Java 平台类库的说明书可以
获得想要的
信息。