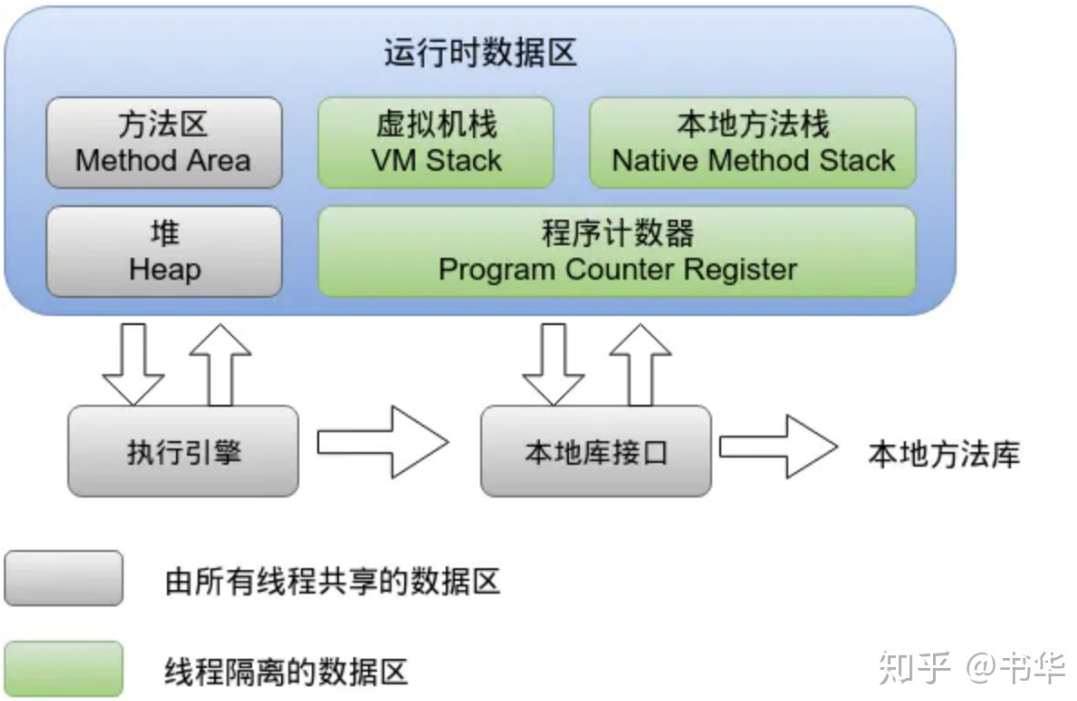
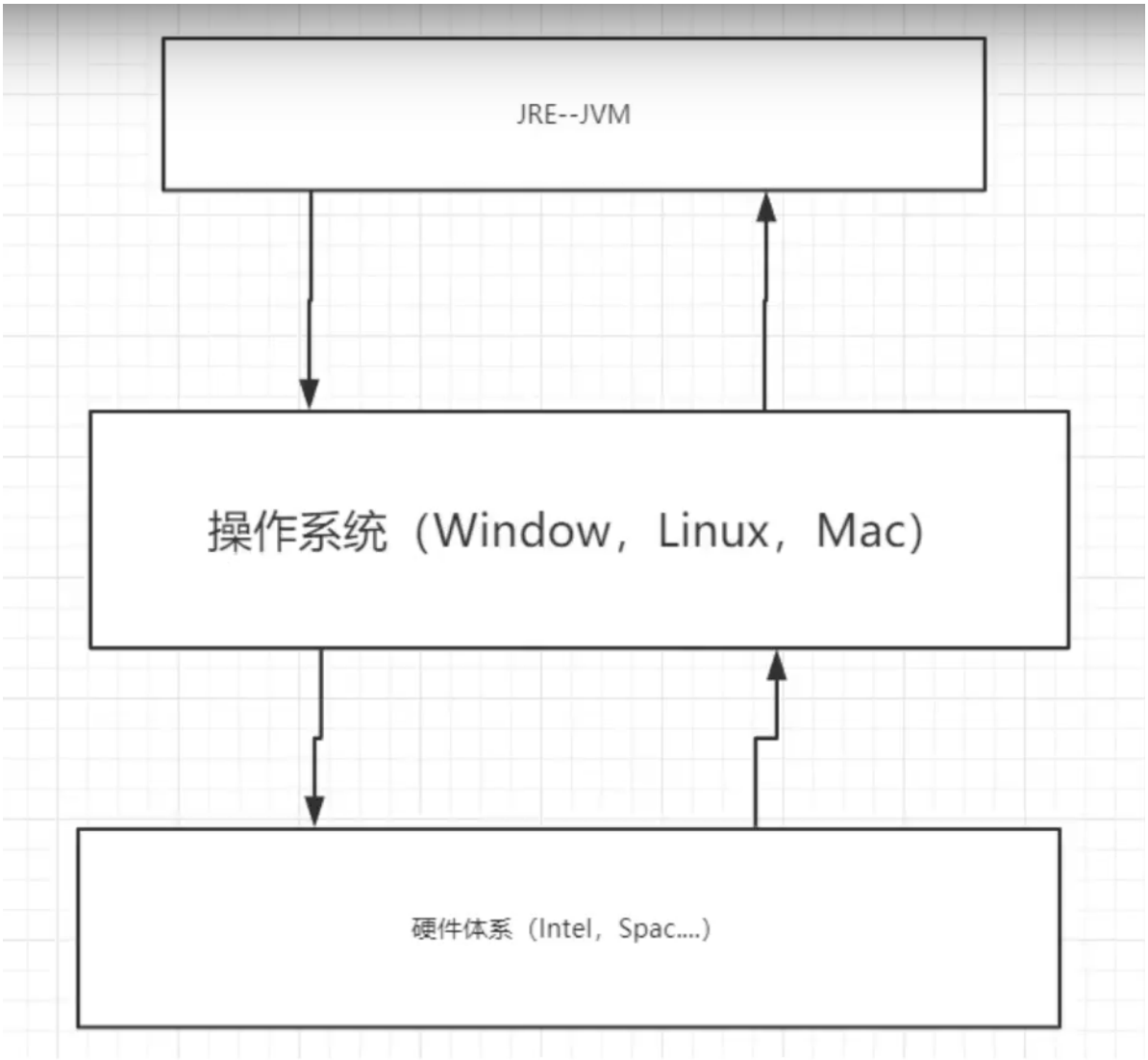
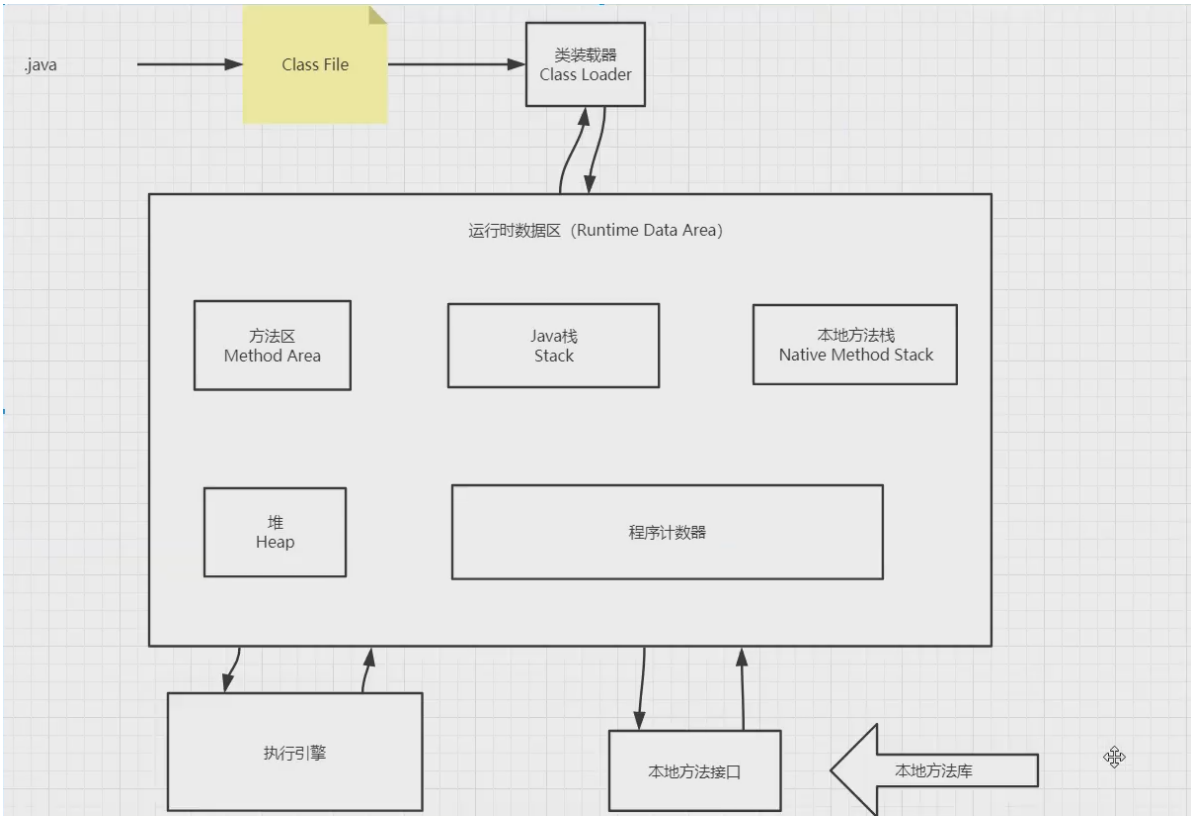


1、JVM的位置

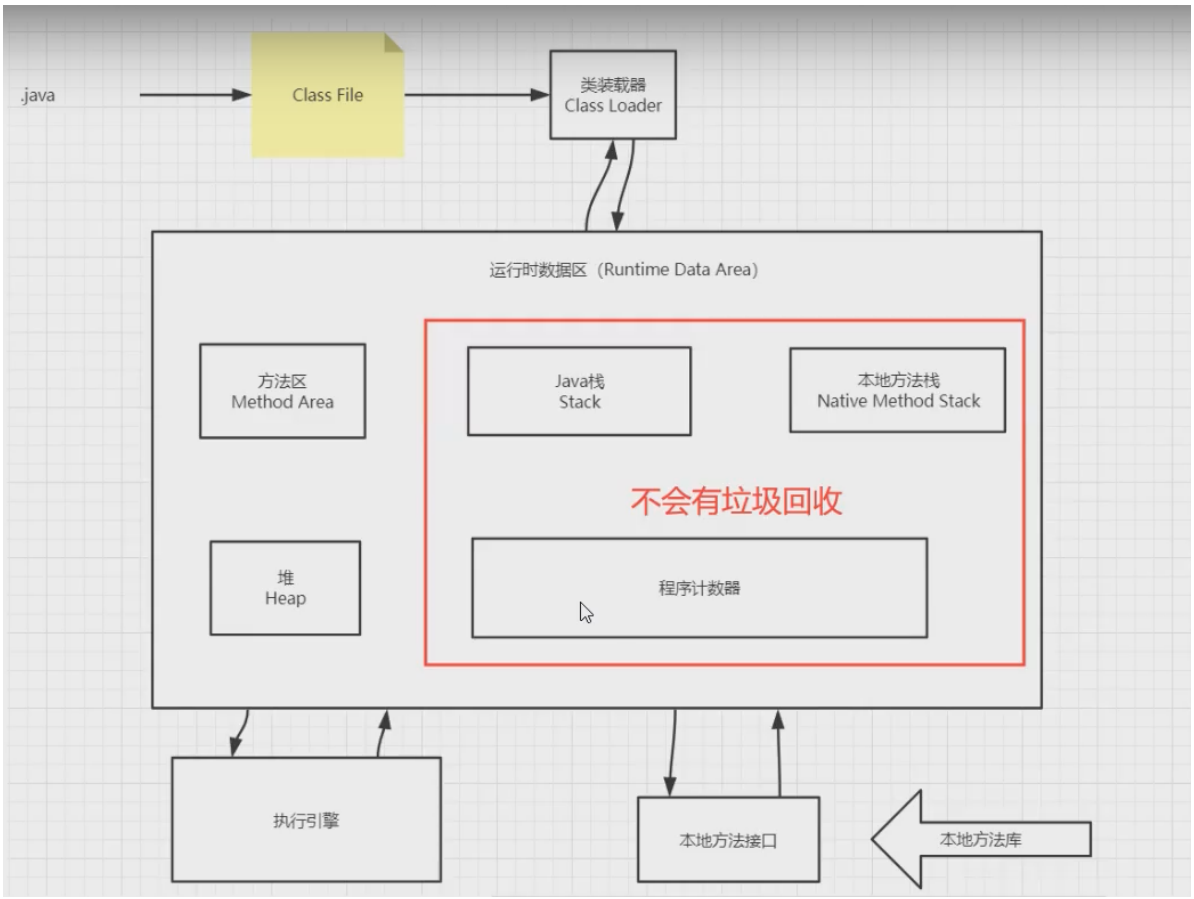
- JVM运行在操作系统之上
- JVM与操作系统打交道
- JVM是用 C++写的
- JRE包含了JVM


JRE--JVM
操作系统（Window，Linux，Mac）
硬件体系（Intel，Spac....）

2、JVM的体系结构

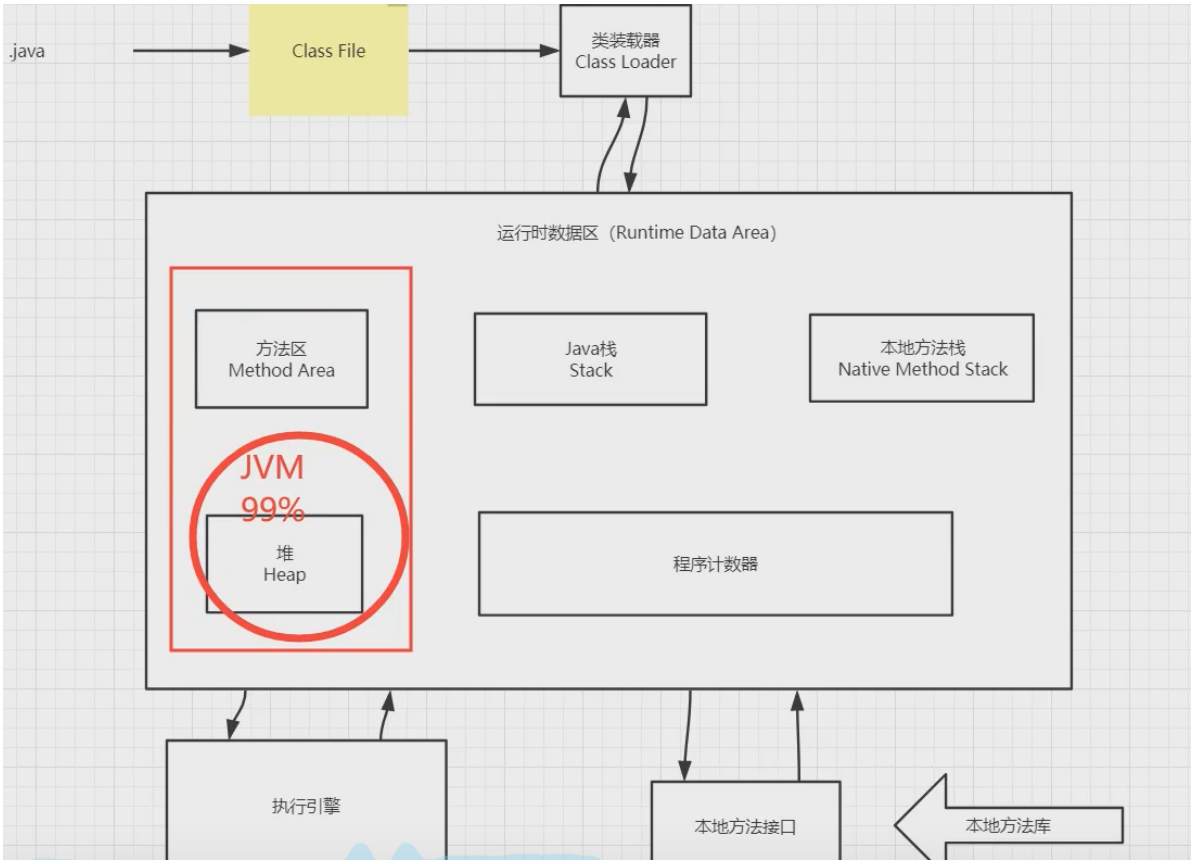

.java
Class File
类装载器
Class Loader
运行时数据区（Runtime Data Area）
方法区
Method Area
Java栈
Stack
本地方法栈
Native Method Stack
堆
Heap
程序计数器
执行引擎
本地方法接口
本地方法库

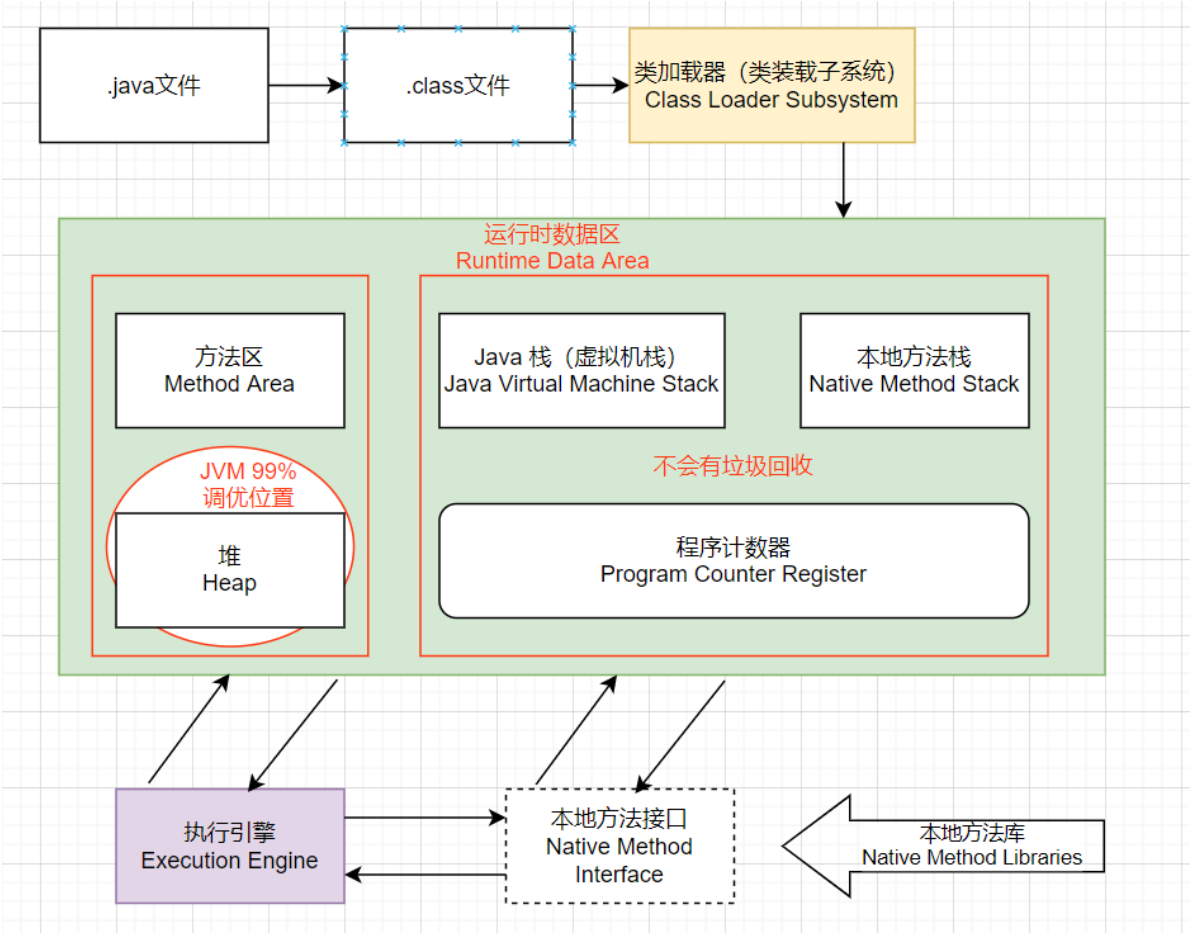
JVM 垃圾回收



JVM调优

JVM调优绝大多数都是指的是针对堆调优

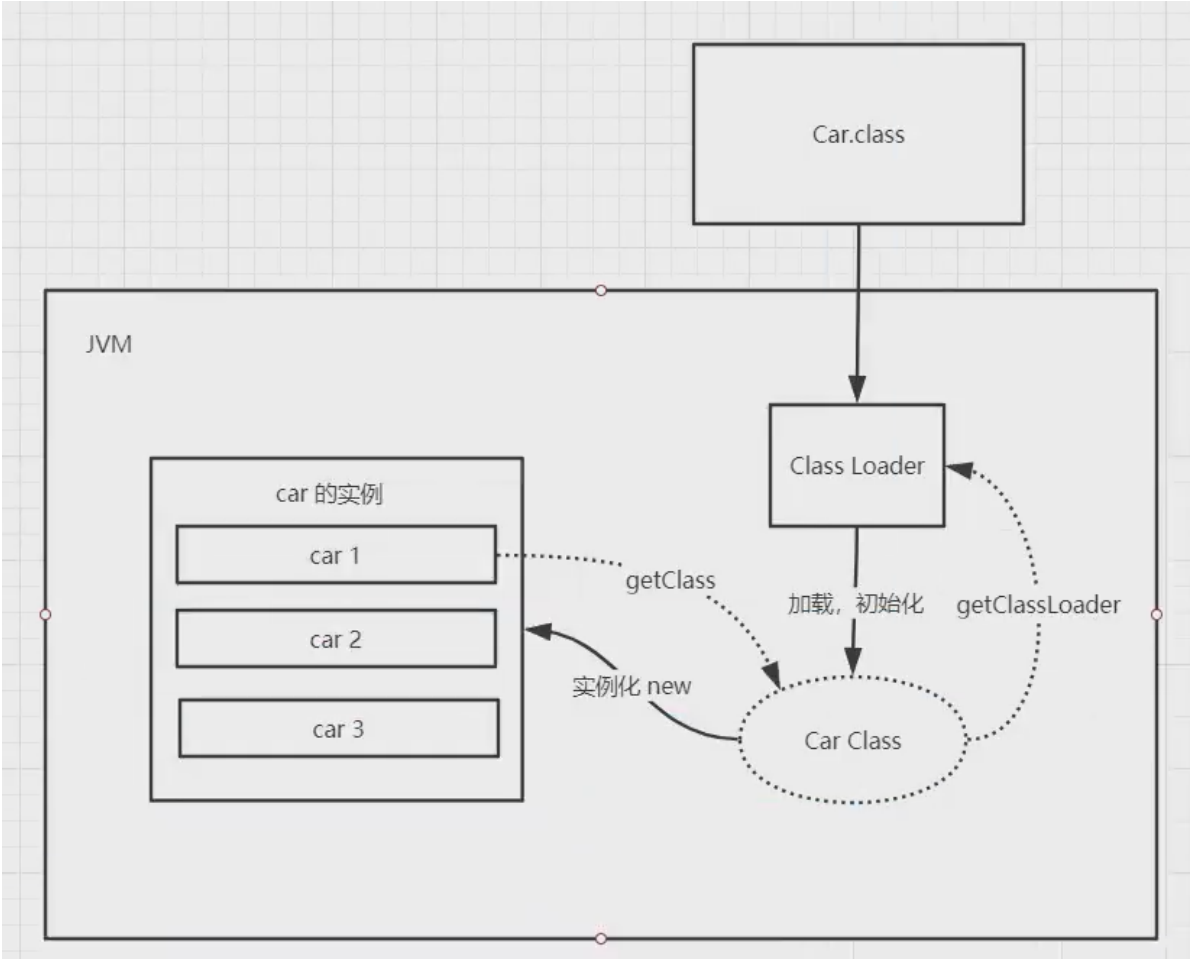




3、类加载器及双亲委派机制

1、类装载器

作用：加载 Class文件



对象实例化的过程（new 关键字）

- 类是模板，是抽象的，new 出来的对象是实例化的，是具体的
- **对象的引用存放在栈中，对象存放在堆中**

```
package jvm;

/**
 * @author miemiehoho
 * @date 2022/2/8 17:21
 */
public class Car {
    private int id;

    public static void main(String[] args) {
        // 类是模板，对象是具体的
        Car car1 = new Car();
        Car car2 = new Car();
        Car car3 = new Car();

        car1.id = 1;
        car2.id = 2;
        car3.id = 3;

        System.out.println(car1.hashCode());
        System.out.println(car2.hashCode());
        System.out.println(car3.hashCode());
```

```
        Class<? extends Car> aClass1 = car1.getClass();
        Class<? extends Car> aClass2 = car1.getClass();
        Class<? extends Car> aClass3 = car1.getClass();

        System.out.println(aClass1.hashCode());
        System.out.println(aClass2.hashCode());
        System.out.println(aClass3.hashCode());
    }
}
```

类加载器分类

1. 虚拟机自带的加载器
2. 启动类（根）加载器　BootstrapClassLoader
3. 扩展类加载器　ExtClassLoader
4. 应用程序加载器　AppClassLoader
5. 用户自定义类加载器　CustomClassLoader

类加载器的类别

BootstrapClassLoader（启动类加载器）

- 主要负责加载核心的类库(java.lang.*等)，构造ExtClassLoader和APPClassLoader。

`C++`编写，加载`java`核心库`java.*`,构造`ExtClassLoader`和`AppClassLoader`。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作

ExtClassLoader （标准扩展类加载器）

- 主要负责加载jre/lib/ext目录下的一些扩展的jar

`java`编写，加载扩展库，如`classpath`中的`jre`，`javax.*`或者`java.ext.dir`指定位置中的类，开发者可以直接使用标准扩展类加载器。

AppClassLoader（系统类加载器）

- 主要负责加载应用程序的主函数类

```
java`编写，加载程序所在的目录，如`user.dir`所在的位置的`class
```

CustomClassLoader（用户自定义类加载器）

`java`编写,用户自定义的类加载器,可加载指定路径的`class`文件

类加载器的加载过程

类加载器会逐层查找

1. 类加载器收到类加载的请求
2. 将这个请求向上委托给父类加载器去完成，一直向上委托，直到启动类加载器
3. 启动加载器检查是否能够加载当前这个类，能加载就结束，使用当前的加载器，否则抛出异常，通知子加载器进行加载
4. 重复步骤3

类加载器与类的唯一性

类加载器虽然只用于实现类的加载动作，但是对于任意一个类，都需要由加载它的类加载器和这个类本身共同确立其在Java虚拟机中的**唯一性**。通俗的说，JVM中两个类是否“相等”，首先就必须是同一个类加载器加载的，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要类加载器不同，那么这两个类必定是不相等的。

这里的“相等”，包括代表类的Class对象的equals()方法、isAssignableFrom()方法、isInstance()方法的返回结果，也包括使用instanceof关键字做对象所属关系判定等情况。

2、双亲委派机制

双亲委派机制的作用：保证安全

运行一个类之前会先向上找：APP-->EXC-->BOOT(最终执行)，如果找不到会倒着往回找

```
package jvm;

/**
 * @author miemiehoho
 * @date 2022/2/8 17:21
 */
public class Car {
    private int id;

    public static void main(String[] args) {
        // 类是模板，对象是具体的
        Car car1 = new Car();

        car1.id = 1;

        System.out.println(car1.hashCode());

        Class<? extends Car> aClass1 = car1.getClass();

        ClassLoader classLoader = aClass1.getClassLoader();// AppClassLoader
        System.out.println(classLoader);
        System.out.println(classLoader.getParent());// ExtClassLoader
        System.out.println(classLoader.getParent().getParent());// null: null的两种可能情况：1，不存在；2，java程序获取不到（C、C++）
    }
}
```

```
package jvm;

/**
 * @author miemiehoho
 * @date 2022/2/8 18:22
 */
public class Student {

    @Override
    public String toString() {
```

```
        return "Hello";
    }

    public static void main(String[] args) {
        Student student = new Student();// 先逐层向上找，EXC、BOOT中都没有Student类，所以往回找，在APP中找到了该Student类
        System.out.println(student.toString());
    }
}
```

总结

前言

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**双亲委派机制**，即**把请求交由父类处理**，它是一种任务委派模式

什么是类加载

通过javac将`.java`文件编译成`.class`字节码文件后，则需要将.class加载到 JVM中运行，哪么是谁将`.class`加载到JVM的呢？那就是类加载器啦。

什么是双亲委派机制

当某个类加载器需要加载某个`.class`文件时，它首先把这个任务委托给他的上级类加载器，递归这个操作，如果上级的类加载器没有加载，自己才会去加载这个类。

工作原理

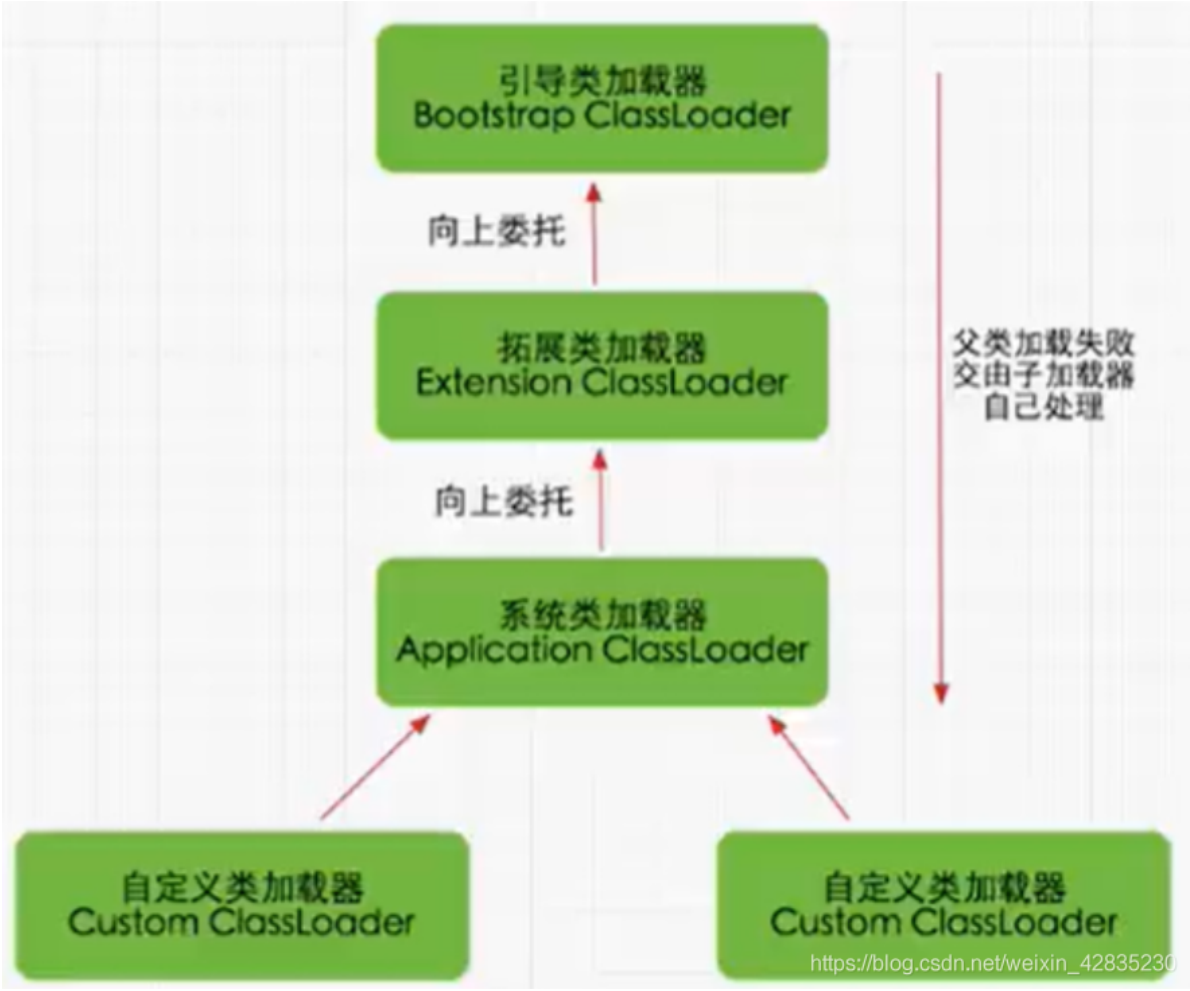
（1）如果一个类加载器收到了类加载请求，它并不会自己先加载，而是把这个请求委托给父类的加载器去执行

（2）如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的引导类加载器；

（3）如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成加载任务，子加载器才会尝试自己去加载，这就是双亲委派机制

（4）父类加载器一层一层往下分配任务，如果子类加载器能加载，则加载此类，如果将加载任务分配至系统类加载器也无法加载此类，则抛出异常

```
1.当Application ClassLoader 收到一个类加载请求时，他首先不会自己去尝试加载这个类，而是将这个请求委派给父类加载器Extension ClassLoader去完成。

2.当Extension ClassLoader收到一个类加载请求时，他首先也不会自己去尝试加载这个类，而是将请求委派给父类加载器Bootstrap ClassLoader去完成。

3.如果Bootstrap ClassLoader加载失败(在<JAVA_HOME>\lib中未找到所需类)，就会让Extension ClassLoader尝试加载。

4.如果Extension ClassLoader也加载失败，就会使用Application ClassLoader加载。

5.如果Application ClassLoader也加载失败，就会使用自定义加载器去尝试加载。

6.如果均加载失败，就会抛出ClassNotFoundException异常。
```



代码示例

代码示例1

自己建立一个 java.lang.String 类，写上 static 代码块

```
package java.lang;

public class String {
    static{
        System.out.println("我是自定义的String类的静态代码块");
    }
}
```

在另外的程序中加载 String 类，看看加载的 String 类是 JDK 自带的 String 类，还是我们自己编写的 String 类

```
/**
 * @author miemiehoho
 * @date 2022/2/8 22:50
 */
public class StringTest {

    public static void main(String[] args) {
        java.lang.String string = new java.lang.String();
        System.out.println("========");
        System.out.println(string.getClass());
        System.out.println(string.getClass().getClassLoader());
    }
}
```

程序并没有输出我们静态代码块中的内容，可见仍然加载的是 JDK 自带的 String 类

输出：

```
========
class java.lang.String
null
```

为什么呢?
由于我们定义的String类 应用系统类加载器，但它并不会自己先加载，而是把这个请求委托给父类的加载器去执行，到了扩展类加载器发现String类不归自己管，再委托给父类加载器（引导类加载器），这时发现是java.lang包，这事就归引导类加载器管，所以加载的是 JDK 自带的 String 类

代码示例2

```
package java.lang;

/**
 * @author miemiehoho
 * @date 2022/2/8 22:49
 */
public class String {
    static {
        System.out.println("自己定义的String类的静态代码块");
    }

    public static void main(String[] args) {
        System.out.println("Hello String!");
    }
}
```

输出：

```
错误: 在类 java.lang.String 中找不到 main 方法, 请将 main 方法定义为:
   public static void main(String[] args)
否则 JavaFX 应用程序类必须扩展javafx.application.Application
```

由于双亲委派机制找到的是 JDK 自带的 String 类，但在引导类加载器的核心类库API里的 String 类中并没有 main() 方法

代码示例3

在 java.lang 包下新建 ShkStart 类 (自定义类名)

```
package java.lang;

public class ShkStart {
    public static void main(String[] args) {
        System.out.println("hello!");
    }
}
```

输出:

```
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.SecurityException: Prohibited package name:
java.lang
    at java.lang.ClassLoader.preDefineClass(ClassLoader.java:655)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:754)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:473)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:74)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:369)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:363)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:362)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:418)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:355)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:351)
    at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:601)
```

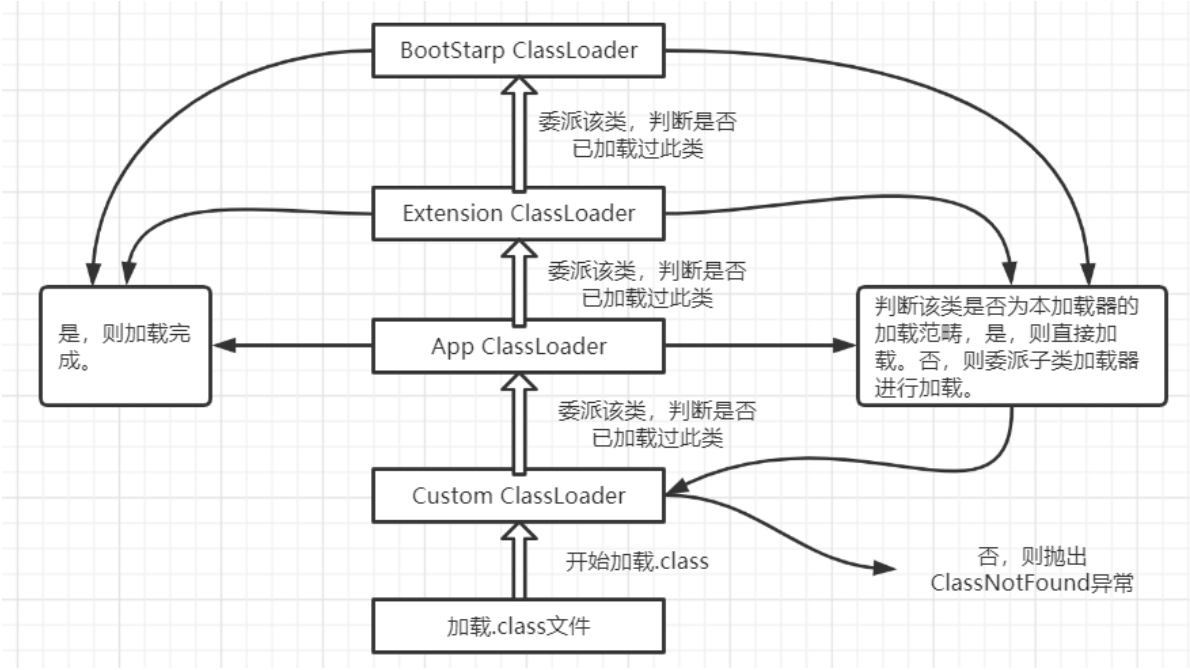
出于保护机制，java.lang 包下不允许我们自定义类

双亲委派机制的作用（优势）

```
双亲委派机制可以

- 避免类的重复加载
- 保护程序安全，防止核心API被随意篡改
  - 自定义类：java.lang.String （没用）
  - 自定义类：java.lang.ShkStart（报错：阻止创建 java.lang开头的类）
```

1、防止重复加载同一个`.class`。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、保证核心`.class`不能被篡改。通过委托方式，不会去篡改核心`.class`，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。不同的加载器加载同一个`.class`也不是同一个`Class`对象。这样保证了`Class`执行安全。



3、自定义类加载器

若要实现自定义类加载器，只需要继承java.lang.ClassLoader 类，并且重写其findClass()方法即可。

java.lang.ClassLoader 类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class 类的一个实例。除此之外，ClassLoader 还负责加载 Java 应用所需的资源，如图像文件和配置文件等，ClassLoader 中与加载类相关的方法如下：

方法说明：

- getParent() 返回该类加载器的父类加载器。
- loadClass(String name) 加载名称为 二进制名称为name 的类，返回的结果是 java.lang.Class 类的实例。
- findClass(String name) 查找名称为 name 的类，返回的结果是 java.lang.Class 类的实例。
- findLoadedClass(String name) 查找名称为 name 的已经被加载过的类，返回的结果是 java.lang.Class 类的实例。
- resolveClass(Class<?> c) 链接指定的 Java 类。

4、Java历史-沙箱安全机制

相关介绍:

我们都知道，程序员编写一个Java程序，默认的情况下可以访问该机器的任意资源，比如读取，删除一些文件或者网络操作等。当你把程序部署到正式的服务器上，系统管理员要为服务器的安全承担责任，那么他可能不敢确定你的程序会不会访问不该访问的资源，为了消除潜在的安全隐患，他可能有两种办法：

1. 让你的程序在一个限定权限的帐号下运行。
2. 利用Java的沙箱机制来限定你的程序不能为非作歹。以下用于介绍该机制。

什么是沙箱?

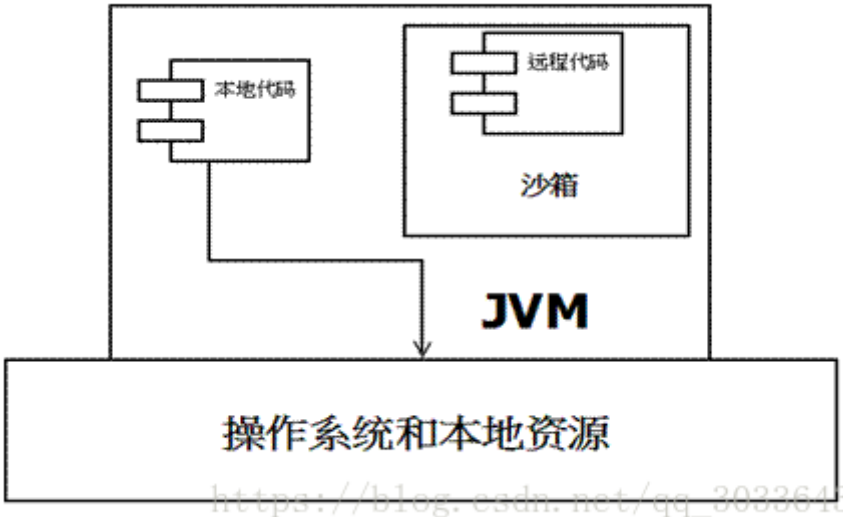
Java安全模型的核心就是Java沙箱（sandbox），什么是沙箱？沙箱是一个限制程序运行的环境。沙箱机制就是将 Java 代码限定在虚拟机(JVM)特定的运行范围中，并且严格限制代码对本地系统资源访问，通过这样的措施来保证对代码的有效隔离，防止对本地系统造成破坏。沙箱**主要限制系统资源访问**，那系统资源包括什么？——`CPU、内存、文件系统、网络`。不同级别的沙箱对这些资源访问的限制也

可以不一样。

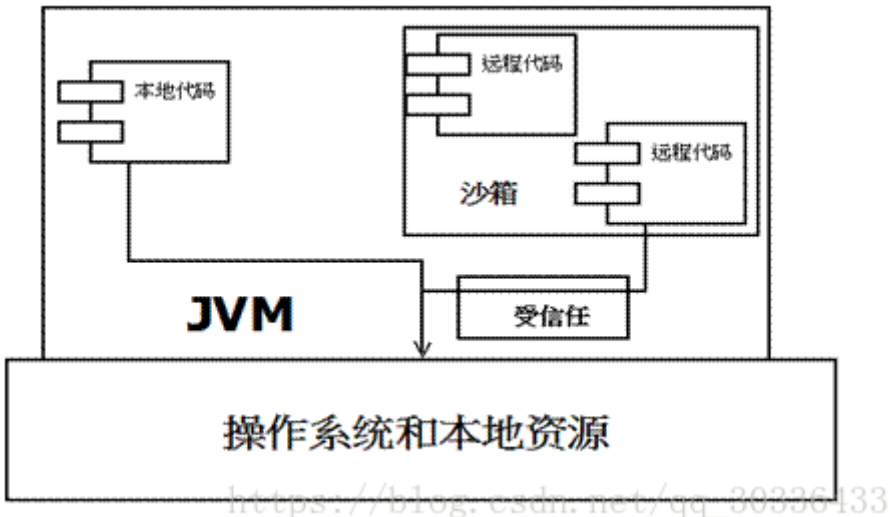
所有的Java程序运行都可以指定沙箱，可以定制安全策略。

Java中的安全模型：

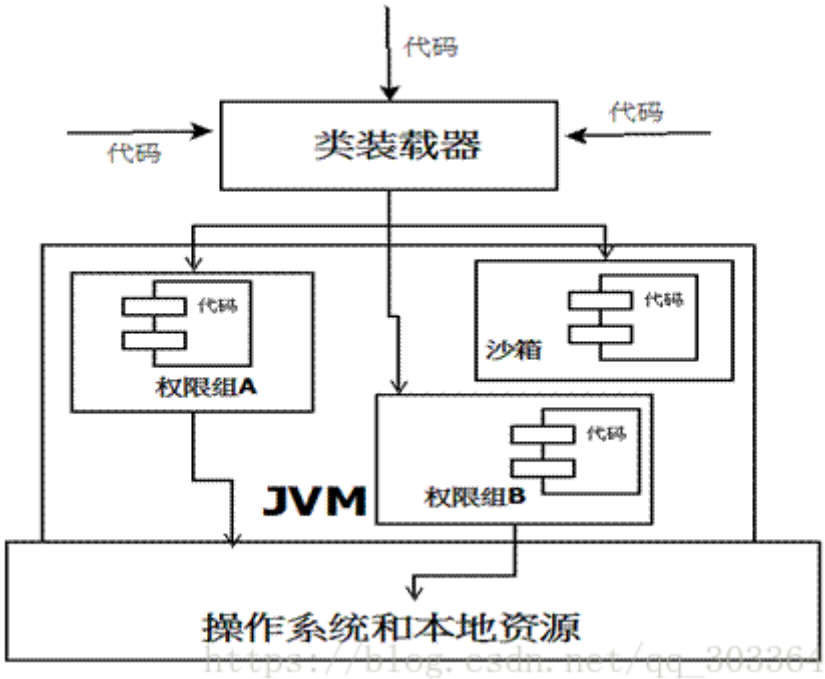
在Java中将执行程序分成本地代码和远程代码两种，本地代码默认视为可信任的，而远程代码则被看作是不受信的。对于授信的本地代码，可以访问一切本地资源。而对于非授信的远程代码在早期的Java实现中，安全依赖于沙箱 (Sandbox) 机制。如下图所示 JDK1.0安全模型



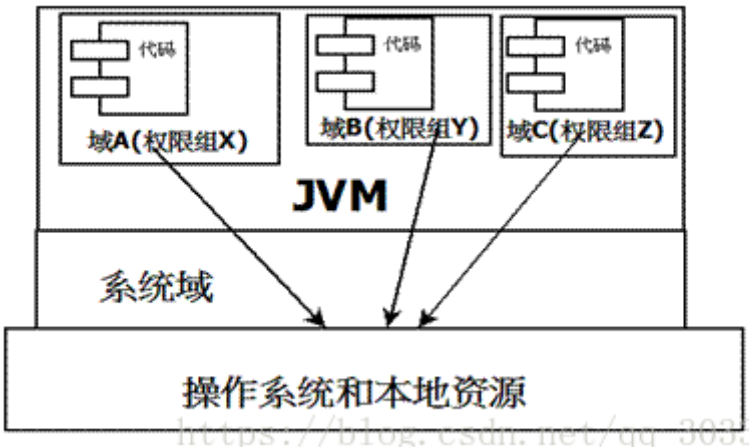
但如此严格的安全机制也给程序的功能扩展带来障碍，比如当用户希望远程代码访问本地系统的文件时候，就无法实现。因此在后续的 Java1.1 版本中，针对安全机制做了改进，增加了`安全策略`，允许用户指定代码对本地资源的访问权限。如下图所示 JDK1.1安全模型



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



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(jdk 1.6)



以上提到的都是基本的`Java 安全模型概念`，在应用开发中还有一些`关于安全的复杂用法`，其中最常用到的 API 就是 doPrivileged。`doPrivileged 方法能够使一段受信任代码获得更大的权限，甚至比调用它的应用程序还要多，可做到临时访问更多的资源`。有时候这是非常必要的，可以应付一些特殊的应用场景。例如，应用程序可能无法直接访问某些系统资源，但这样的应用程序必须得到这些资源才能够完成功能。

组成沙箱的基本组件：

- `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。但并不是所有的类文件都会经过字节码校验，比如核心类。
- 类装载器（class loader）：其中类装载器在3个方面对Java沙箱起作用
  - 它防止恶意代码去干涉善意的代码；
  - 它守护了被信任的类库边界；
  - 它将代码归入保护域，确定了代码可以进行哪些操作。

虚拟机为不同的类加载器载入的类提供不同的命名空间，命名空间由一系列唯一的名称组成，每一个被装载的类将有一个名字，这个命名空间是由Java虚拟机为每一个类装载器维护的，它们互相之间甚至不可见。

类装载器采用的机制是**双亲委派模式。**

1. 从最内层JVM自带类加载器开始加载，外层恶意同名类得不到加载从而无法使用；
2. 由于严格通过包来区分了访问域，外层恶意的类通过内置代码也无法获得权限访问到内层类，破坏代码就自然无法生效。

- `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

沙箱包含的要素:

1. 权限

权限是指允许代码执行的操作。包含三部分：权限类型、权限名和允许的操作。

2. 代码源

代码源是类所在的位置，表示为URL地址。

3. 保护域

保护域用来组合代码源和权限，这是沙箱的基本概念。保护域就在于声明了比如由代码A可以做权限B这样的事情。

4. 策略文件

策略文件是控制沙箱的管理要素，一个策略文件包含一个或多个保护域的项。策略文件完成了代码权限的指定任务，策略文件包括全局和用户专属两种。

5. 密钥库

保存密钥证书的地方。

默认沙箱

通过Java命令行启动的Java应用程序，默认不启用沙箱。要想启用沙箱，启动命令需要做如下形式的变更：

```
java -Djava.security.manager <other args>
```

沙箱启动后，安全管理器会使用两个默认的策略文件来确定沙箱启动参数。当然也可以通过命令指定：

```
java -Djava.security.policy=<URL>
```

如果要求启动时只遵循一个策略文件，那么启动参数要加个等号，如下：

`java -Djava.security.policy==<URL>`

5、Native、方法区

Native

```
package jvm;

/**
 * @author miemiehoho
 * @date 2022/2/9 16:30
 */
public class Demo {
    public static void main(String[] args) {
        new Thread(() -> System.out.println("Hello Native!"),
"myThread").start();
    }

    // native: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
    // 会进入本地方法栈
    // 进入本地方法栈后会调用本地方法接口（JNI）
    // 本地方法接口会调用本地方法库
    // JNI作用：扩展Java的使用，融合不同的编程语言为Java所用，如 C、C++
    // 它在内存区域中专门开辟了一块标记区域：Native Method Stack（本地方法栈）
    // 在最终执行的时候，加载本地方法库中的方法，通过JNI

    // 需要调用 native方法的场景：java程序驱动打印机、管理系统、robot类等
    private native void start0();
}
```

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，大概过程：

1. 会进入本地方法栈
2. 进入本地方法栈后会调用本地方法接口（JNI）
3. 本地方法接口会调用本地方法库

JNI作用：

- 扩展Java的使用，融合不同的编程语言为Java所用，如 c、c++
- 它在内存区域中专门开辟了一块标记区域：Native Method Stack（本地方法栈）
- 在最终执行的时候，加载本地方法库中的方法

需要调用 native方法的场景：java程序驱动打印机、管理系统、robot类等

除了native外，调用其他接口的方法还有：http、socket、WebService、

PC 寄存器

程序计数寄存器(Program Counter Register)

JVM中的PC寄存器是对物理PC寄存器的一种抽象模拟

作用：

PC寄存器用来存储指向下一条指令的地址，也即将要执行的指令代码。

特性：

- 它是一块很小的内存空间，几乎可以忽略不记。也是运行速度最快的存储区域
- 在JVM规范中，每个线程都有它自己的程序计数器，是线程私有的，生命周期与线程的生命周期保持一致。
- 任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。
- 程序计数器会存储当前线程正在执行的Java方法的JVM指令地址，若在执行native方法，则是未指定值(undefined)

使用PC寄存器存储字节码指令地址有什么用呢?

因为CPU需要不停的切换各个线程，这时候切换回来以后，就得知道接着从哪开始继续执行。

JVM的字节码解释器就是需要通过改变PC寄存器的值来明确下一条应该执行什么样的字节码指令。

PC寄存器为什么会被设定为线程私有?

我们都知道所谓的多线程在一个特定的时间段内只会执行其中某一个线程的方法，CPU会不停地做任务切换，这样必然导致经常中断或恢复，如何保证分毫无差呢?

为了能够准确地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个线程都分配一个PC寄存器，这样一来各个线程之间便可以进行独立计算，从而不会出现相互干扰的情况。

补充：关于CPU时间片

CPU时间片即CPU分配给各个程序的时间，每个线程被分配一个时间段，称作它为时间片。

在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。

但在微观上：由于只有一个CPU(单核CPU)，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，每个程序轮流执行。

方法区

方法区是被所有线程共享的，**静态变量、常量、类信息、运行时的常量池存在方法区中，但是 实例变量存在堆内存中，和方法区无关**（static、final、Class、运行时的常量池）

6、深入理解栈

栈主管程序的运行，生命周期和线程同步，线程结束，栈内存也就释放了，不存在垃圾回收问题

栈中存放的内容：8大基本类型、对象的引用、实例的方法

栈+堆+方法区的交互关系：

Java是值传递，不是引用传递?

7、堆

三种 JVM

目前有三大Java虚拟机：HotSpot，oracle JRockit，IBM J9

一个 JVM只有一个堆内存，堆内存的大小是可以调节的。

类加载器读取了类文件后，会把 类、方法、常量、变量 保存到堆中，保存的是所有引用类型的真实对象

8、新生区、永久区，堆内存调优

堆内存中细分为三个区域：

- 新生
  - 对象实例 诞生的地方、甚至死亡
  - 所有对象都是在 伊甸园区 new 出来的

- 老年
- 永久

  常驻内存中，用来存放JDK自身携带的 Class对象，Interface元数据，存储的是Java运行时的一些环境和类信息，这个区域不存在垃圾回收，关闭虚拟机就会释放永久代区域
  - JDK 1.6：永久代，常量池在方法区
  - JDK1.7：永久代，字符常量池在堆中，运行时常量池在方法区中
  - JDK1.8之后：没有永久代，方法区变成了元空间，字符常量池在堆中，运行时常量池在元空间中?

jdk1.8元空间在本地内存中，取代jvm中的方法区，堆在jvm中；运行时常量池在元空间，字符串常量池在堆中

方法区是堆中的一个逻辑部分，但是别名叫非堆

1.8元空间占用本地内存非堆内存

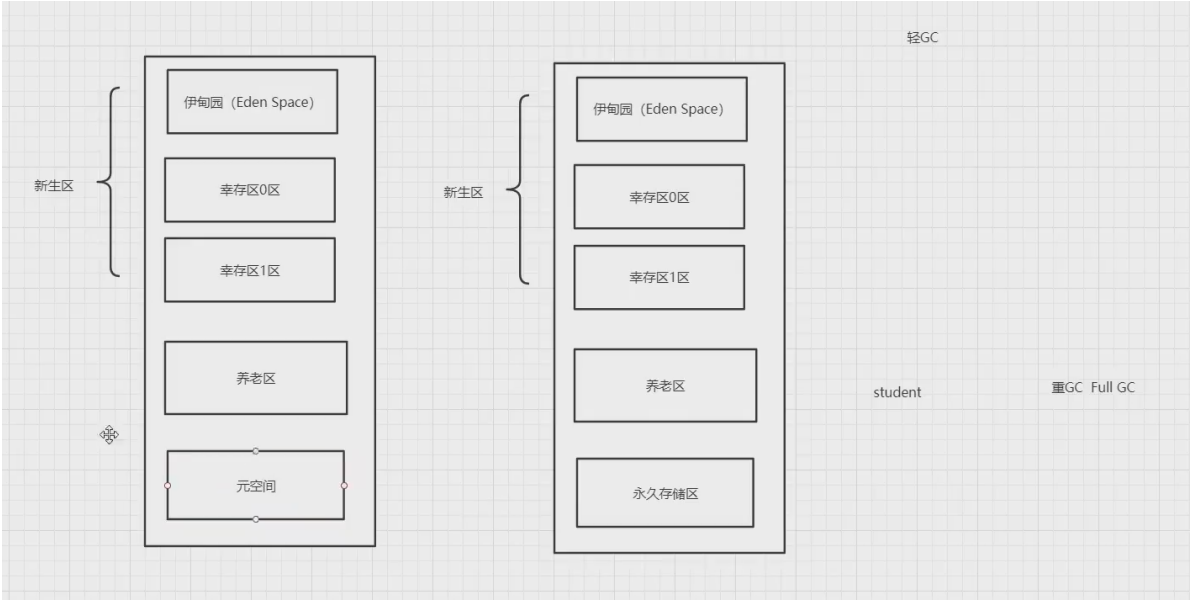
JVM规范中，方法区逻辑上是堆的一部分

方法区是一个特殊的堆 jdk8以后在堆(元空间里面)

方法区只是一种规范，而永久代或者元空间是方法区的实现，他们并不是从属关系。

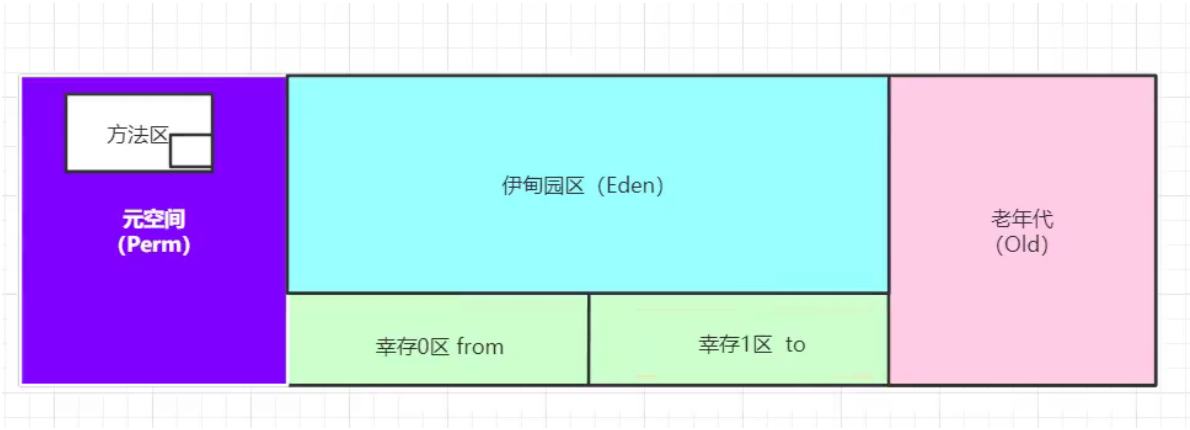
什么情况下永久区会导致OOM?

一个启动类加载了大量的第三方jar包、tomcat部署了太多的应用、大量动态生成反射类，不断被加载，直到堆内存满，就会出现OOM



GC垃圾回收主要在伊甸园区和老年区

GC 分为 轻GC、重GC




```
package jvm;

import java.util.Random;

/**
 * @author miemiehoho
 * @date 2022/2/9 19:26
 */
// -Xms1024m -Xmx1024m -XX:+PrintGCDetails
public class Demo2 {
    public static void main(String[] args) {
        String str = "abcdefghijklmmopqrstuvwxyz";
        while (true) {
            str += str + new Random().nextInt(999999999) + new
Random().nextInt(999999999);
        }
    }
}
```

```
package jvm;
```

```java
/**
 * @author miemiehoho
 * @date 2022/2/9 19:30
 */
public class Demo3 {
    public static void main(String[] args) {
        // JVM试图使用的最大内存
        long max = Runtime.getRuntime().maxMemory();
        // JVM的总内存
        long total = Runtime.getRuntime().totalMemory();

        System.out.println("max=" + max + "byte," + max / (1024 * 1024) + "MB");
        System.out.println("total=" + total + "byte," + total / (1024 * 1024) +
"MB");

        // 默认分配的总内存是电脑内存的 1/4，初始化的内存是电脑内存的 1/16

        // 修改JVM内存：-Xms1024m -Xmx1024m -XX:+PrintGCDetails

        // OOM的处理：
        // 1.尝试扩大堆内存看结果
        // 2.分析内存，看一下哪个地方出现了问题（专业工具）

        // 元空间，逻辑上和堆是联系的，但是物理上不相关
        // 元空间是在本地内存中的，并没有在JVM内存中，所以打印出来的结果新生区+老年区就已经是
JVM内存了
        // 元空间使用的是计算机物理内存，不是JVM内存
    }
}
```

```java
    // 默认分配的总内存是电脑内存的 1/4，初始化的内存是电脑内存的 1/16

    //

    // OOM的处理：
    // 1.尝试扩大堆内存看结果
    // 2.

    // 元空间，逻辑上和堆是联系的，但是物理上不相关
    // 元空间是在本地内存中的，并没有在JVM内存中，所以打印出来的结果新生区+老年区就已经是JVM内
存了
    // 元空间使用的是计算机物理内存，不是JVM内存
```

- JVM 默认分配的总内存是电脑内存的 1/4，初始化的内存是电脑内存的 1/16
- 修改JVM内存：-Xms1024m -Xmx1024m -XX:+PrintGCDetails

OOM的处理：

1. 尝试扩大堆内存看结果
2. 分析内存，看一下哪个地方出现了问题（专业工具）

- 元空间，逻辑上和堆是联系的，但是物理上不相关
- 元空间是在本地内存中的，并没有在JVM内存中，所以打印出来的结果新生区+老年区就已经是JVM内存了

- 元空间使用的是计算机物理内存，**不是JVM内存**

9、使用 JPofiler工具分析 OOM原因

在一个项目中，出现了OOM，该如何排除?

- 能够看到代码第几行出错：内存快照分析工具：MAT、JPofiler
- Debug，一行行分析代码

MAT、JPofiler 作用：

- 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

实例

编辑 VM参数：`-Xms1m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError`

- -Xms 设置初始化内存分配大小， 默认 1/64
- -Xmx 设置最大分配内存， 默认 1/4
- -XX:+PrintGCDetails 打印GC垃圾回收信息
- -XX:+HeapDumpOnOutOfMemoryError oom 时 DUMP

```
package jvm;

import java.util.ArrayList;
import java.util.List;

/**
 * @author miemiehoho
 * @date 2022/2/9 22:50
 */
public class Demo4 {

    byte[] array = new byte[1 * 1024 * 1024];

    public static void main(String[] args) {
        List<Demo4> list = new ArrayList<>();
        int count = 0;
        try {
            while (true) {
                list.add(new Demo4());
                count++;
            }
        } catch (Exception e) {
            System.out.println("count:" + count);
            e.printStackTrace();
        }
    }
}
// -Xms1m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError
```

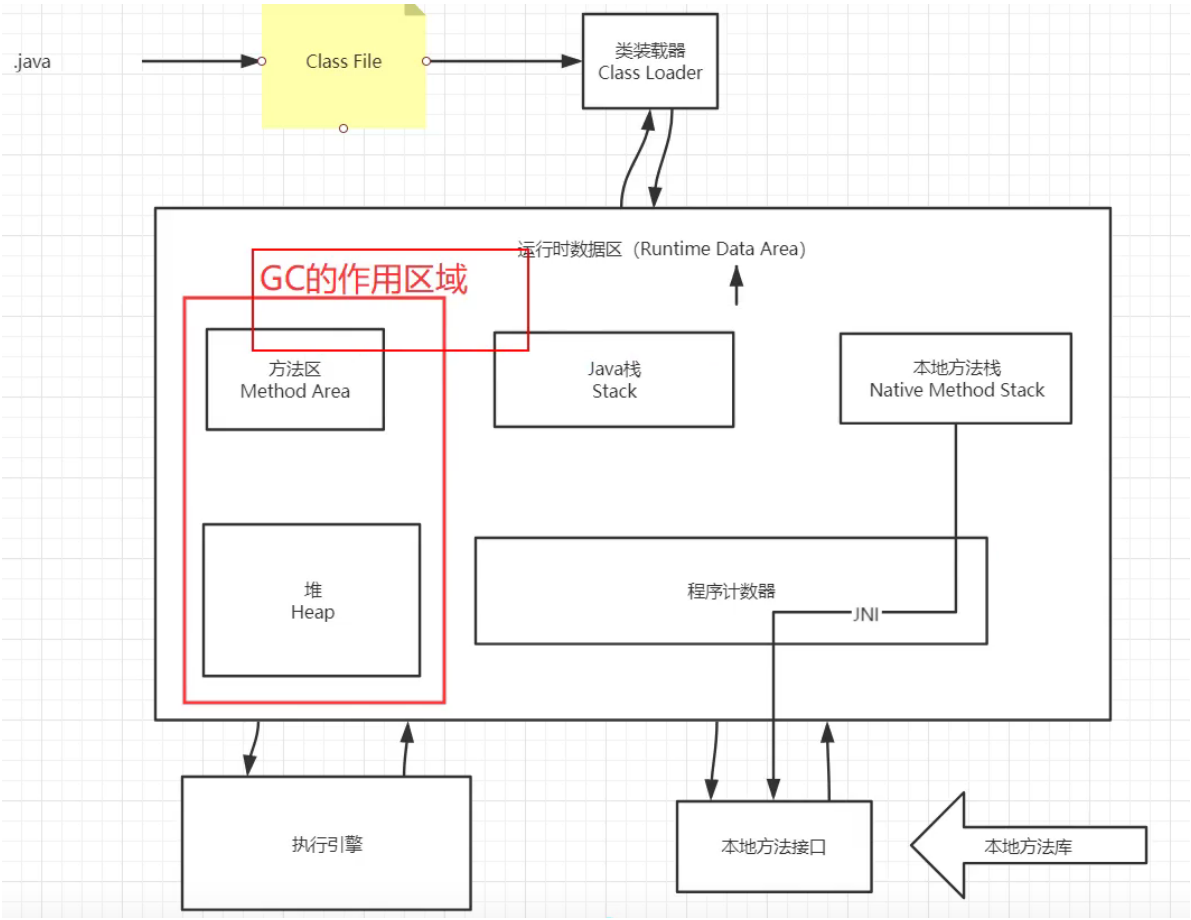
扩展知识

Runtime 类

（用来调优，java运行时环境的一些东西）

10、GC

GC的作用区域



- JVM在进行 GC时，大部分时候回收都是 新生代
- GC分为两种：轻GC（普通GC）、重GC（全局GC）
- 堆内存中细分为三个区域：
  - 新生
    - 伊甸园区
    - 幸存区（from、to）
  - 老年
  - 永久

GC题目

JVM的内存模型和分区，详细说明每个区放什么

堆中的分区有哪些？Eden、from、to，老年区，说说它们的特点

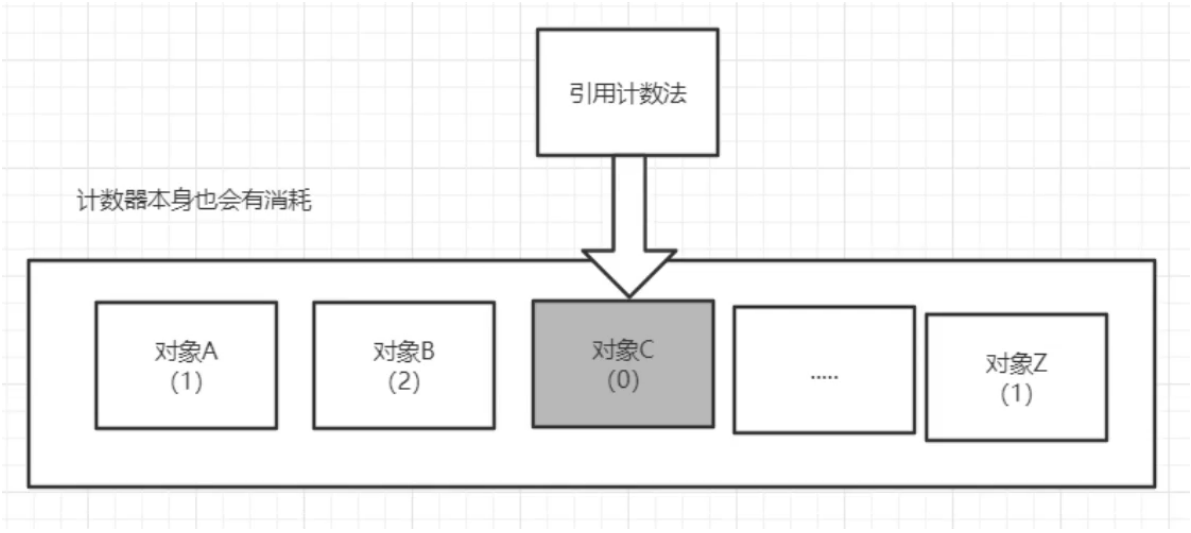
GC的算法有哪些？怎么用？

GC四大算法：标记整理算法、复制算法、标记清除算法、分代收集算法

> GC四大算法中没有引用计数法
>
> 四大算法应该为：复制算法，标记整理算法，标记清除算法，分代收集算法
>
> 引用计数法应该与可达性分析归为一类

轻GC和重GC什么时候发生？

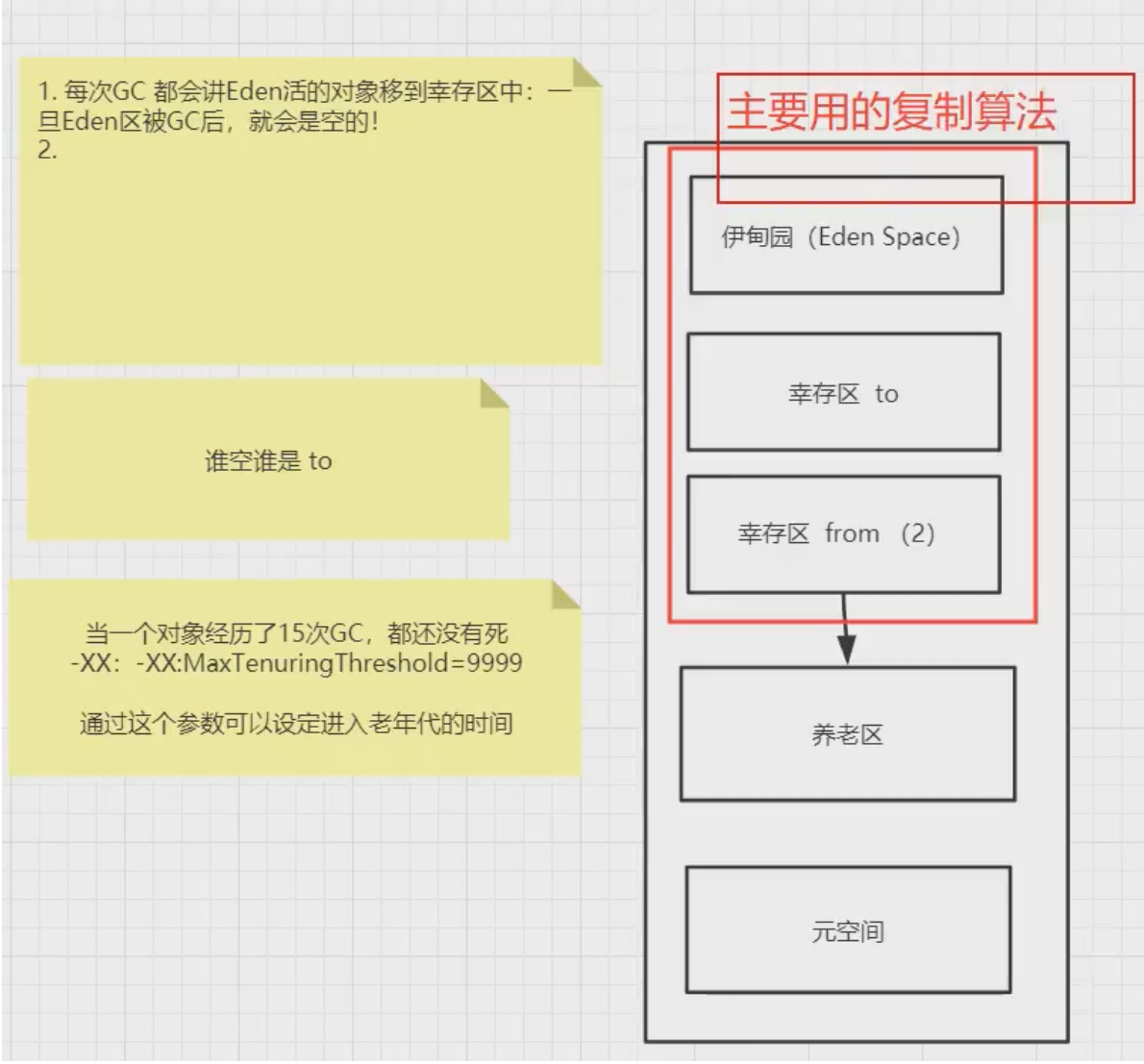
11、GC 之 引用计数法



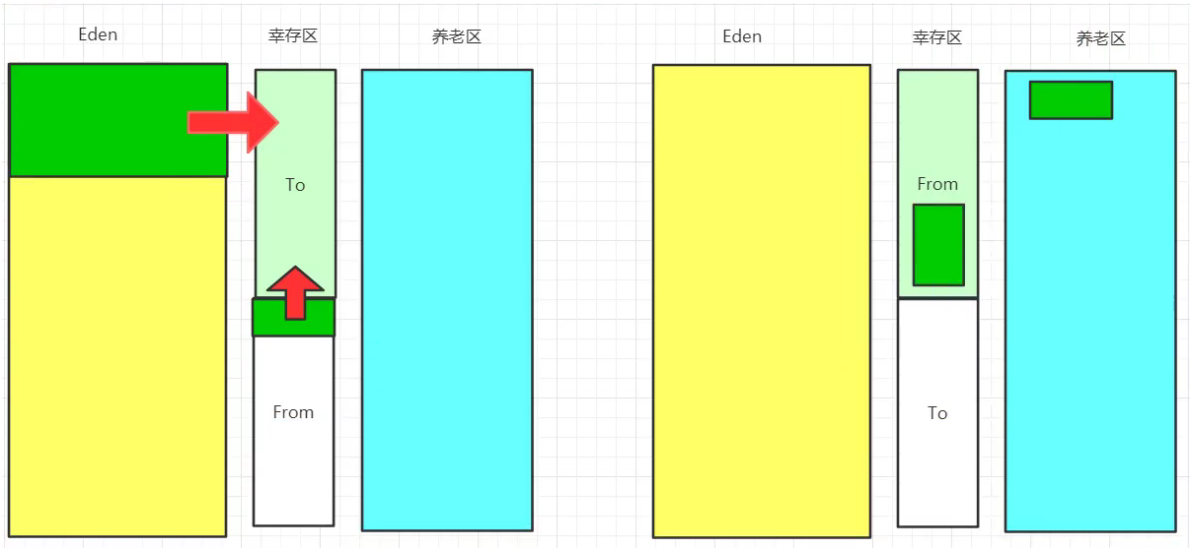
12、 GC 之复制算法

- 新生区
  - 伊甸园区
    - 每次GC后都会将 Eden活着的对象移动到幸存区中，一旦Eden区被GC后，就会是空的！
  - 幸存区（from、to）
    - 谁空谁是 to
- 养老区
  - 当一个对象经历了15次GC（默认是15次），都还没有死，就会移动到养老区
  - 可以通过参数：`-XX:+MaxTenuringThreshold=15` 设定进入老年代的时间

- 元空间



复制算法过程



- 很少有放不下的情况的，幸存区有两个就是为了解决对象放不下内存碎片化的问题。因为永远有一个空的幸存区可以放。
- To区放不下了，剩下的全部晋升到养老区，养老区也放不下直接Full GC，Full GC后还放不下，抛出OOM

好处：没有内存的碎片

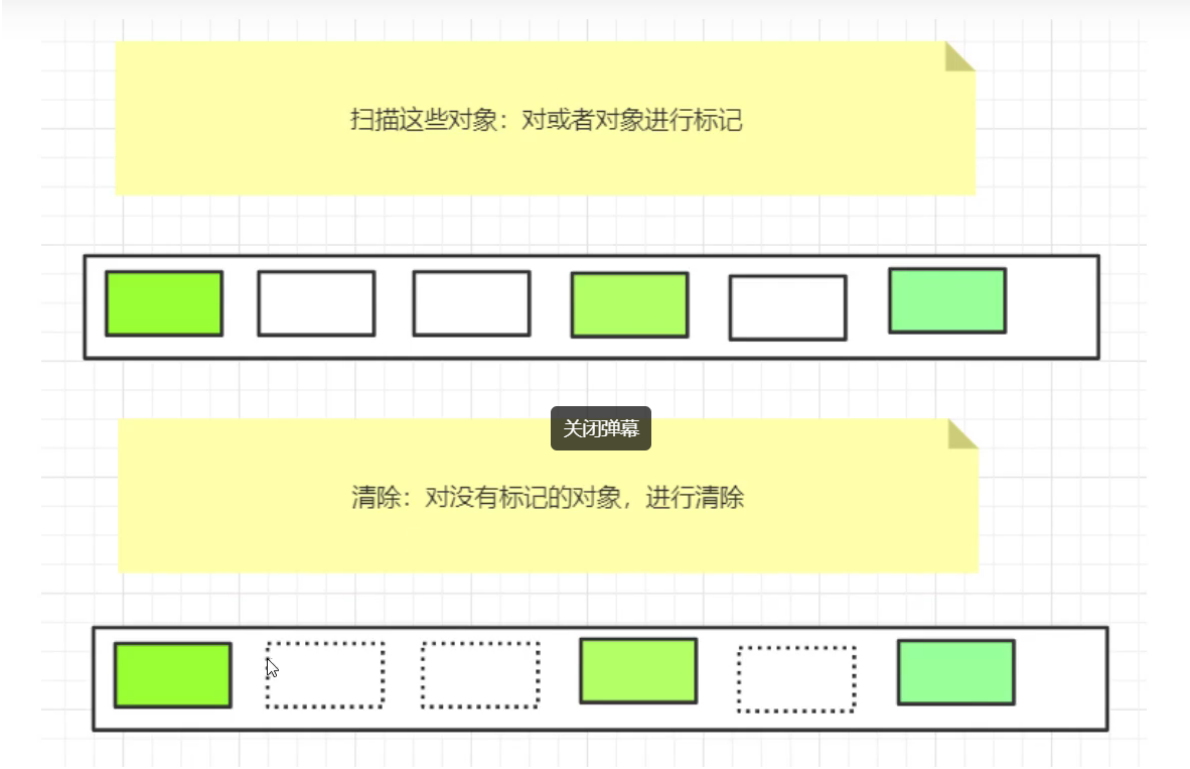
坏处：空间利用率不高，浪费了内存空间，多了一半空间永远是空的（to区）

复制算法最佳使用场景：对象存活度较低的时候（新生区）

13、 GC 之标记压缩清除算法

标记清除

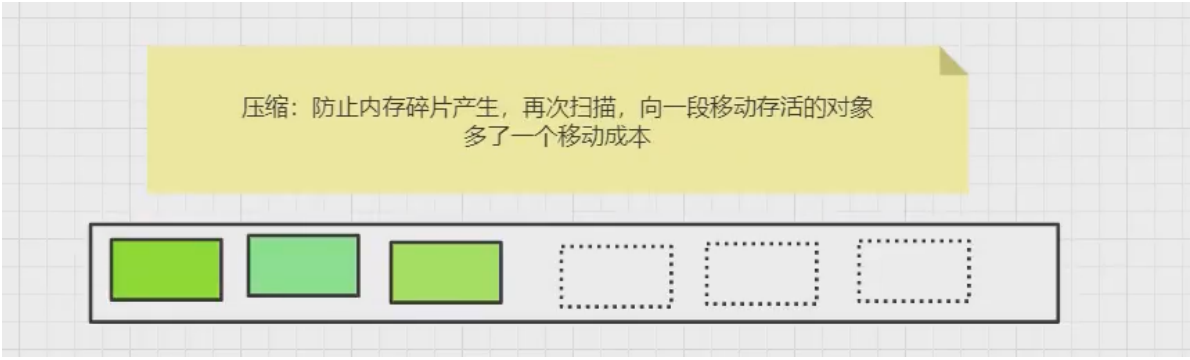
这里判断对象用的可达性分析算法（GCRoot根）



优点：不需要额外的空间

缺点：两次扫描，严重浪费时间，会产生内存碎片

标记压缩



14、 GC 算法总结

内存效率(时间复杂度)：复制算法 > 标记清除算法 > 标记压缩算法

内存整齐度：复制算法 = 标记压缩算法 > 标记清除算法

内存利用率：标记压缩算法 = 标记清除算法 > 复制算法

没有最好的算法，只有最合适的算法 ---> GC：分代收集算法

- 年轻代：存活率低，使用复制算法
- 老年代：区域大，存活率高，使用 标记清除（碎片不多的时候）、标记压缩算法混合实现

Voliate保证可见性，不保证原子性，防止指令重排

15、JMM：Java Memory Model

1、什么是 JMM

Java Memory Model的缩写，Java内存模型

> JMM：Java内存模型，是java虚拟机规范中所定义的一种内存模型，Java内存模型是标准化的，屏蔽掉了底层不同计算机的区别（注意这个跟JVM完全不是一个东西，现在还有小伙伴搞错的）。
>
> 其实早期计算机中cpu和内存的速度是差不多的，但在现代计算机中，cpu的指令速度远超内存的存取速度，由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存（Cache）来作为内存与处理器之间的缓冲。
>
> 将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理器就无须等待缓慢的内存读写了。
>
> 基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是也为计算机系统带来更高的复杂度，因为它引入了一个新的问题：缓存一致性（CacheCoherence）。
>
> 在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存（MainMemory）

2、它是干嘛的

- 是一种**缓存一致性协议，用于定义数据读写的规则**
- JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory）

JVM是Java实现的虚拟计算机（Java Virtual Machine），对于熟悉计算机结构的同学，我感觉把这些概念和物理机对应起来更好理解。

JVM对应的就是物理机，它有存放数据的存储区：堆、栈等由JVM管理的内存（对应于物理机的内存）、执行数据计算的执行单元：线程（对应于物理机的CPU）、加速线程执行的本次存储区：可能会从存储区里分配一块空间来存储线程本地数据，比如栈（对应于物理机的cache）。

众所周知，现代计算机一般都会包含多个处理器，多个处理器共享主内存。为了提升性能，会在每个处理器上增加一个小容量的cache加速数据读写。cache会导致了缓存一致性问题，为了解决缓存一致性问题又引入了一系列Cache一致性协议（比如MSI、MESI、MOSI、Synapse、Firefly及Dragon Protocol）来解决CPU本地缓存和主内存数据不一致问题。

而JVM中管理下的存储空间（包括堆、栈等）就对应与物理机的内存；

线程本次存储区（例如栈）就对应于物理机的cache；

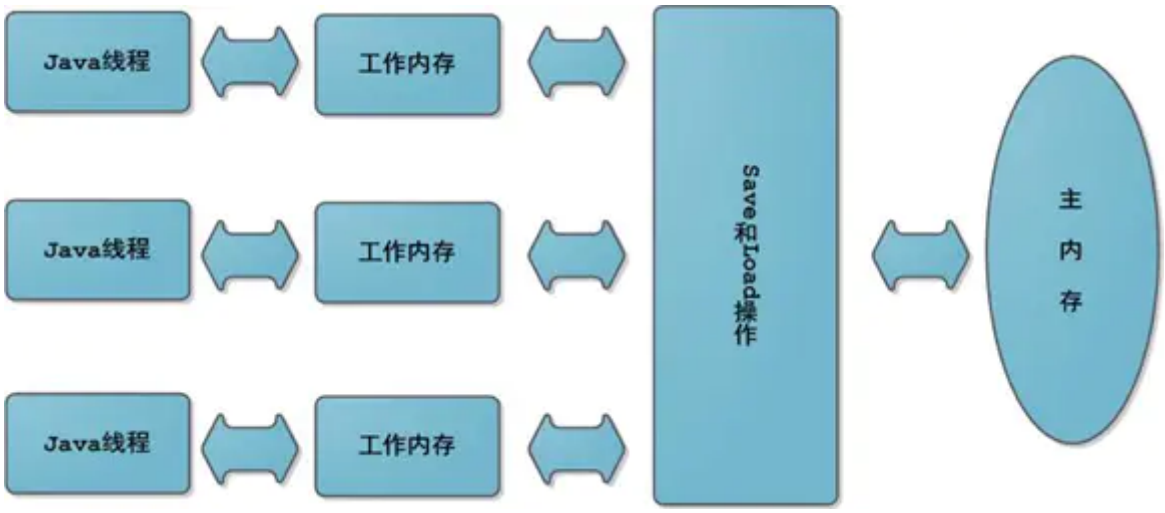
而JMM就对应于类似于MSI、MESI、MOSI、Synapse、Firefly及Dragon Protocol这样的**缓存一致性协议，用于定义数据读写的规则。**

JMM相对于物理机的缓存一致性协议来说它还要处理JVM自身特有的问题：重排序问题，参见： http://cmsblogs.com/?p=2116。

那么JMM都有哪些内容呢?

- 官方文档： http://www.cs.umd.edu/~pugh/java/memoryModel/CommunityReview.pdf
- 通俗理解就是happens-before原则 https://www.cnblogs.com/chenssy/p/6393321.html

JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM是整个计算机虚拟模型，所以JMM是隶属于JVM的。从抽象的角度来看，**JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory）**，本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。



Java内存模型带来的问题

可见性问题

CPU中运行的线程从主存中拷贝共享对象obj到它的CPU缓存，把对象obj的count变量改为2。但这个变更对运行在右边CPU中的线程不可见，因为这个更改还没有flush到主存中：要解决共享对象可见性这个问题，我们可以使用java `volatile`关键字或者是加锁

```
volatile 不能保证原子性，保证了可见性
```

3、 它该如何学习

- JMM是一个抽象的概念，理论
- 内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可在分的
    - lock （锁定）：作用于主内存的变量，把一个变量标识为线程独占状态
    - unlock （解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
    - read （读取）：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用

    - load （载入）：作用于工作内存的变量，它把read操作从主存中变量放入工作内存中
    - use （使用）：作用于工作内存中的变量，它把工作内存中的变量传输给执行引擎，每当虚拟机遇到一个需要使用到变量的值，就会使用到这个指令
    - assign （赋值）：作用于工作内存中的变量，它把一个从执行引擎中接受到的值放入工作内存的变量副本中
    - store （存储）：作用于主内存中的变量，它把一个从工作内存中一个变量的值传送到主内存中，以便后续的write使用
    - write （写入）：作用于主内存中的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中
- JMM对这八种指令的使用，制定了如下规则：
    - 不允许read和load、store和write操作之一单独出现。即使用了read必须load，使用了store必须write
    - 不允许线程丢弃他最近的assign操作，即工作变量的数据改变了之后，必须告知主存
    - 不允许一个线程将没有assign的数据从工作内存同步回主内存
    - 一个新的变量必须在主内存中诞生，不允许工作内存直接使用一个未被初始化的变量。就是怼变量实施use、store操作之前，必须经过assign和load操作
    - 一个变量同一时间只有一个线程能对其进行lock。多次lock后，必须执行相同次数的unlock才能解锁
    - 如果对一个变量进行lock操作，会清空所有工作内存中此变量的值，在执行引擎使用这个变量前，必须重新load或assign操作初始化变量的值
    - 如果一个变量没有被lock，就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量
    - 对一个变量进行unlock操作之前，必须把此变量同步回主内存

JMM对这八种操作规则和对volatile的一些特殊规则就能确定哪里操作是线程安全，哪些操作是线程不安全的了。但是这些规则实在复杂，很难在实践中直接分析。所以一般我们也不会通过上述规则进行分析。更多的时候，使用java的happen-before规则来进行分析

模型特征

原子性：例如上面八项操作，在操作系统里面是不可分割的单元。被synchronized关键字或其他锁包裹起来的操作也可以认为是原子的。从一个线程观察另外一个线程的时候，看到的都是一个个原子性的操作。

```
synchronized (this) {
    a = 1;
    b = 2;
}
```

例如一个线程观察另外一个线程执行上面的代码，只能看到a、b都被赋值成功结果，或者a、b都尚未被赋值的结果。

可见性：每个工作线程都有自己的工作内存，所以当某个线程修改完某个变量之后，在其他的线程中，未必能观察到该变量已经被修改。volatile关键字要求被修改之后的变量要求立即更新到主内存，每次使用前从主内存处进行读取。因此volatile可以保证可见性。除了volatile以外，synchronized和final也能实现可见性。synchronized保证unlock之前必须先把变量刷新回主内存。final修饰的字段在构造器中一旦完成初始化，并且构造器没有this逸出，那么其他线程就能看到final字段的值。

有序性：java的有序性跟线程相关。如果在线程内部观察，会发现当前线程的一切操作都是有序的。如果在线程的外部来观察的话，会发现线程的所有操作都是无序的。因为JMM的工作内存和主内存之间存在延迟，而且java会对一些指令进行重新排序。volatile和synchronized可以保证程序的有序性，很多程序员只理解这两个关键字的执行互斥，而没有很好的理解到volatile和synchronized也能保证指令

不进行重排序。

16、如何学习

针对面试学习

1. 什么是 JMM （百度百科）
2. 它是干嘛的 （帖子）
3. 它该如何学习（帖子）
4. 面试题

- netty
- Spring Cloud Alibaba