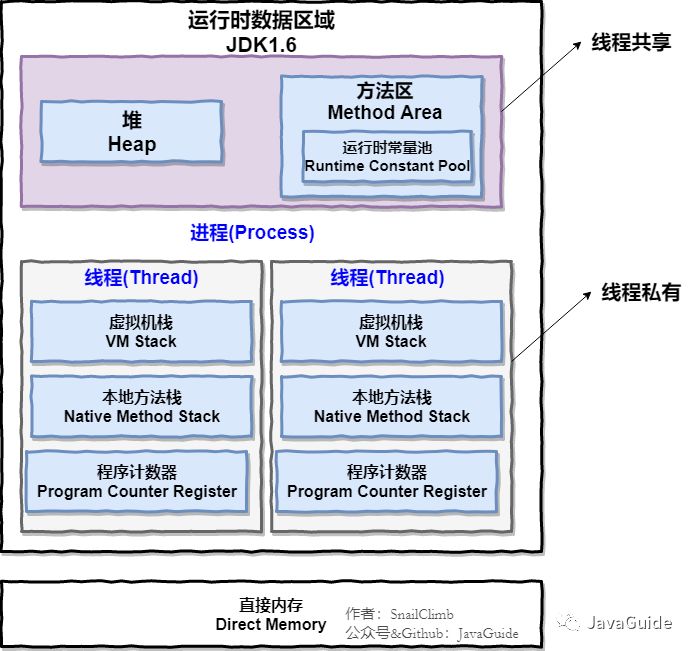
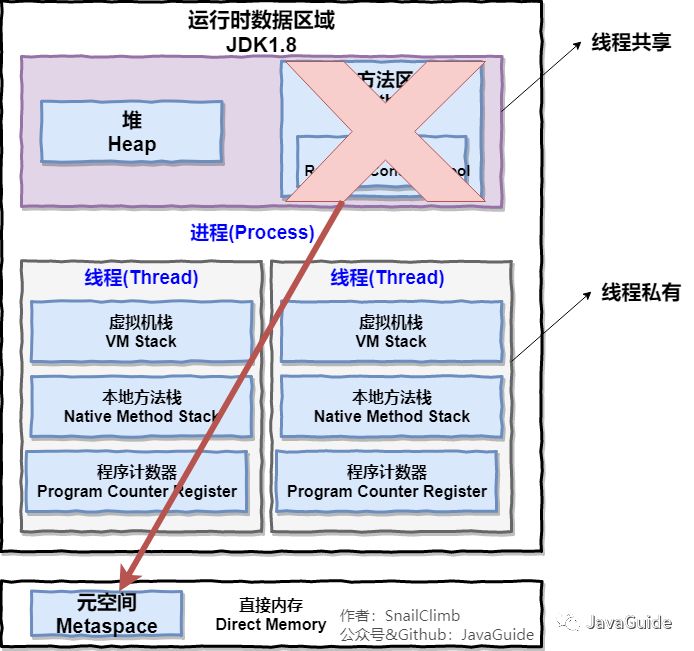
# JVM

运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。JDK. 1.8 和之前的版本略有不同





## 程序计数器

从上面的介绍中我们知道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。

## 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。

（实际上，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编译器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）。

Java 虚拟机栈会出现两种异常：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异常。

•OutOfMemoryError： 若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出OutOfMemoryError异常。

Java方法有两种返回方式：

•return 语句。

•抛出异常。

不管哪种返回方式都会导致栈帧被弹出。

## 本地方法栈

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。

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 两种异常。

## 堆

Java 堆是所有线程共享的一块内存区域，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区、s0区、s1区都属于新生代，tentired 区属于老年代。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。

## 方法区

方法区与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

HotSpot 虚拟机中方法区也常被称为 “永久代”，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“永久存在”了。

JDK 1.8 的时候，方法区被彻底移除了（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。

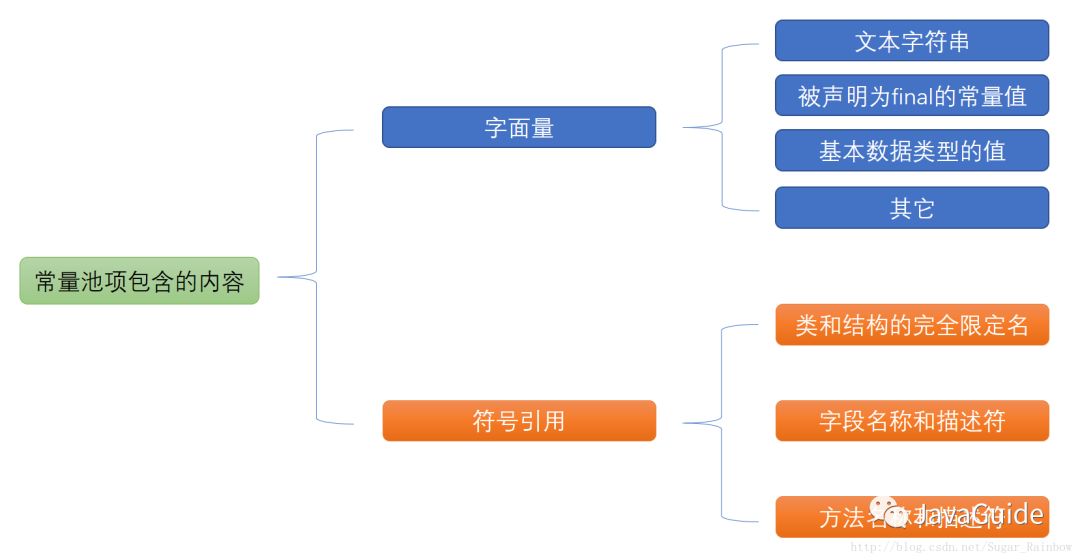
我们可以使用参数： -XX:MetaspaceSize 来指定元数据区的大小。与永久区很大的不同就是，如果不指定大小的话，随着更多类的创建，虚拟机会耗尽所有可用的系统内存。

## 运行时常量池

运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池信息（用于存放编译期生成的各种字面量和符号引用）

既然运行时常量池时方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

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。



## 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 OutOfMemoryError 异常出现。

JDK1.4 中新加入的 NIO(New Input/Output) 类，引入了一种基于通道（Channel） 与缓存区（Buffer） 的 I/O 方式，它可以直接使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样就能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆之间来回复制数据。

# 类加载

## 类加载七个阶段以及案例分析

而 JVM 虚拟机执行 class 字节码的过程可以分为七个阶段：加载、验证、准备、解析、初始化、使用、卸载。

加载：在这个阶段，JVM 的主要目的是将字节码从各个位置（网络、磁盘等）转化为二进制字节流加载到内存中，接着会为这个类在 JVM 的方法区创建一个对应的 Class 对象，这个 Class 对象就是这个类各种数据的访问入口。

验证：JVM规范校验，代码逻辑校验。

准备（重要）：

1.内存分配的对象：为类成员变量分配即静态变量内存，在准备阶段，JVM 只会为「类变量」分配内存，而不会为「类成员变量」分配内存。

2.初始化的类型。在准备阶段，JVM 会为类变量分配内存，并为其初始化。但是这里的初始化指的是为变量赋予 Java 语言中该数据类型的零值，而不是用户代码里初始化的值。

例如下面的代码在准备阶段之后，sector 的值将是 0，而不是 3。public static int sector = 3;

但如果一个变量是常量（被 static final 修饰）的话，那么在准备阶段，属性便会被赋予用户希望的值。

例如下面的代码在准备阶段之后，number 的值将是 3，而不是 0。

public static final int number = 3;

解析：当通过准备阶段之后，JVM 针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类引用进行解析。这个阶段的主要任务是将其在常量池中的符号引用替换成直接其在内存中的直接引用。

初始化（重要）：到了初始化阶段，用户定义的 Java 程序代码才真正开始执行。在这个阶段，JVM 会根据语句执行顺序对类对象进行初始化，一般来说当 JVM 遇到下面 5 种情况的时候会触发初始化：

遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令的最常见的Java代码场景是：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、已在编译器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的时候，如果类没有进行过初始化，则需要先触发其初始化。

当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

当使用 JDK1.7 动态语言支持时，如果一个 java.lang.invoke.MethodHandle实例最后的解析结果 REF\_getstatic,REF\_putstatic,REF\_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

使用：当 JVM 完成初始化阶段之后，JVM 便开始从入口方法开始执行用户的程序代码。这个阶段也只是了解一下就可以。

卸载：当用户程序代码执行完毕后，JVM 便开始销毁创建的 Class 对象，最后负责运行的 JVM 也退出内存。这个阶段也只是了解一下就可以。

案例分析1:

class Grandpa

{

static

{

System.out.println("爷爷在静态代码块");

}

public Grandpa() {

System.out.println("我是爷爷~");

}

}

class Father extends Grandpa

{

static

{

System.out.println("爸爸在静态代码块");

}

public Father()

{

System.out.println("我是爸爸~");

}

}

class Son extends Father

{

static

{

System.out.println("儿子在静态代码块");

}

public Son()

{

System.out.println("我是儿子~");

}

}

public class InitializationDemo

{

public static void main(String[] args)

{

new Son(); //入口

}

}

输出结果：

爷爷在静态代码块

爸爸在静态代码块

儿子在静态代码块

我是爷爷~

我是爸爸~

我是儿子~

静态字段，只有直接定义这个字段的类才会被初始化（执行静态代码块），因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

案例分析2

public class Book {

public static void main(String[] args)

{

staticFunction();

}

static Book book = new Book();

static

{

System.out.println("书的静态代码块");

}

{

System.out.println("书的普通代码块");

}

Book()

{

System.out.println("书的构造方法");

System.out.println("price=" + price +",amount=" + amount);

}

public static void staticFunction(){

System.out.println("书的静态方法");

}

int price = 110;

static int amount = 112;

}

上面这个例子的输出结果是：

书的普通代码块

书的构造方法

price=110,amount=0

书的静态代码块

书的静态方法

下面我们一步步来分析一下代码的整个执行流程。

在上面两个例子中，因为 main 方法所在类并没有多余的代码，我们都直接忽略了 main 方法所在类的初始化。但在这个例子中，main 方法所在类有许多代码，我们就并不能直接忽略了。

当 JVM 在准备阶段的时候，便会为类变量分配内存和进行初始化。此时，我们的 book 实例变量被初始化为 null，amount 变量被初始化为 0。

当进入初始化阶段后，因为 Book 方法是程序的入口，根据我们上面说到的类初始化的五种情况的第四种：当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。JVM 会对 Book 类进行初始化。

JVM 对 Book 类进行初始化首先是执行类构造器（按顺序收集类中所有静态代码块和类变量赋值语句就组成了类构造器），后执行对象的构造器（先收集成员变量赋值，后收集普通代码块，最后收集对象构造器，最终组成对象构造器）。

对于 Book 类，其类构造方法可以简单表示如下：

static Book book = new Book();

static

{

System.out.println("书的静态代码块");

}

static int amount = 112;

于是首先执行static Book book = new Book();这一条语句，这条语句又触发了类的实例化。与类构造器不同，于是 JVM 执行 Book 类的成员变量，再搜集普通代码块，最后执行类的构造方法，于是其执行语句可以表示如下：

int price = 110;

{

System.out.println("书的普通代码块");

}

Book()

{

System.out.println("书的构造方法");

System.out.println("price=" + price +", amount=" + amount);

}

于是此时 price 赋予 110 的值，输出：「书的普通代码块」、「书的构造方法」。而此时 price 为 110 的值，而 amount 的赋值语句并未执行，所以只有在准备阶段赋予的零值，所以之后输出「price=110,amount=0」。

当类实例化完成之后，JVM 继续进行类构造器的初始化：

static Book book = new Book(); //完成类实例化

static

{

System.out.println("书的静态代码块");

}

static int amount = 112;

即输出：「书的静态代码块」，之后对 amount 赋予 112 的值。

到这里，类的初始化已经完成，JVM 执行 main 方法的内容。

public static void main(String[] args)

{

staticFunction();

}

即输出：「书的静态方法」。

分析方法论

从上面几个例子可以看出，分析一个类的执行顺序大概可以按照如下步骤：

确定类变量的初始值。在类加载的准备阶段，JVM 会为类变量初始化零值，这时候类变量会有一个初始的零值。如果是被 final 修饰的类变量，则直接会被初始成用户想要的值。

初始化入口方法。当进入类加载的初始化阶段后，JVM 会寻找整个 main 方法入口，从而初始化 main 方法所在的整个类。当需要对一个类进行初始化时，会首先初始化类构造器，之后初始化对象构造器。

初始化类构造器。初始化类构造器是初始化类的第一步，其会按顺序收集类变量的赋值语句、静态代码块，最终组成类构造器由 JVM 执行。

初始化对象构造器。初始化对象构造器是在类构造器执行完成之后的第二部操作，其会按照执行类成员变成赋值、普通代码块、对象构造方法的顺序收集代码，最终组成对象构造器，最终由 JVM 执行。

如果在初始化 main 方法所在类的时候遇到了其他类的初始化，那么继续按照初始化类构造器、初始化对象构造器的顺序继续初始化。如此反复循环，最终返回 main 方法所在类。

# HotSpot 虚拟机对象探秘

## 对象的创建

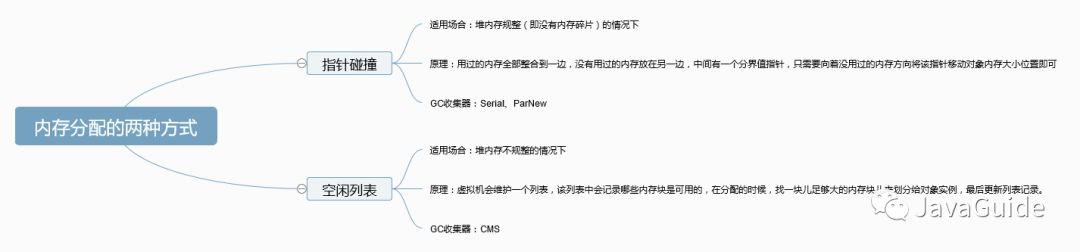


①类加载检查： 虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

②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，选择那种分配方式由 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）

选择以上两种方式中的哪一种，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"（也称作"标记-压缩"），值得注意的是，复制算法内存也是规整的



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

③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。

④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，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希吗、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。

⑤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

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。

Hotspot虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的自身运行时数据（哈希码、GC分代年龄、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。

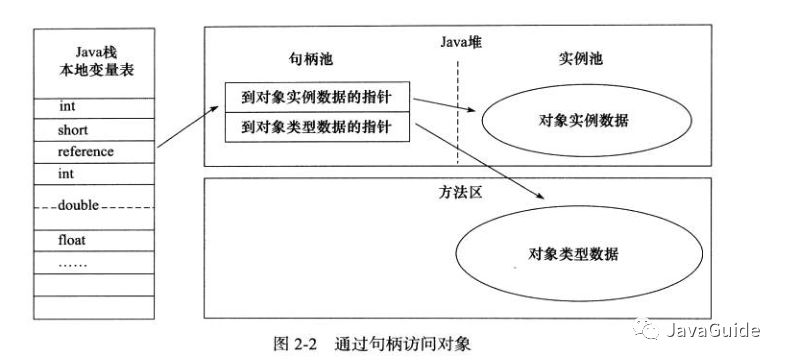
实例数据部分是对象真正存储的有效信息，也是在程序中所定义的各种类型的字段内容。

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。

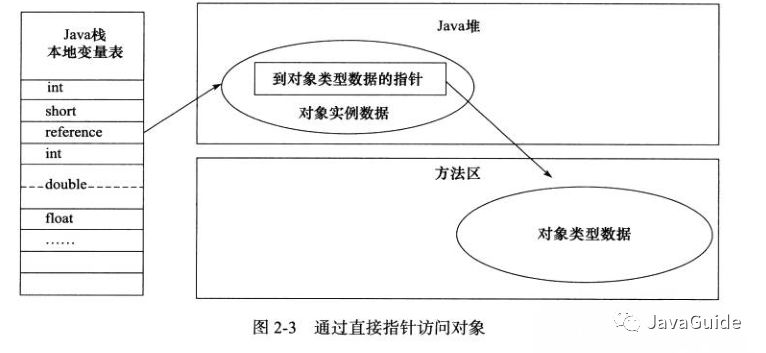
对象的访问定位

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。对象的访问方式有虚拟机实现而定，目前主流的访问方式有①使用句柄和②直接指针两种：

•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；



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。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。

# String 类和常量池

**String 对象的两种创建方式**：

String str1 = "abcd";

String str2 = new String("abcd");

这两种不同的创建方法是有差别的，第一种方式是在常量池中拿对象，第二种方式是直接在堆内存空间创建一个新的对象。只要使用new方法，便需要创建新的对象。

**String 类型的常量池比较特殊。它的主要使用方法有两种：**

•直接使用双引号声明出来的 String 对象会直接存储在常量池中。

•如果不是用双引号声明的 String 对象，可以使用 String 提供的 intern 方法。String.intern() 是一个 Native 方法，它的作用是：如果运行时常量池中已经包含一个等于此 String 对象内容的字符串，则返回常量池中该字符串的引用；如果没有，则在常量池中创建与此 String 内容相同的字符串，并返回常量池中创建的字符串的引用。

String s1 = new String("计算机");

String s2 = s1.intern();

String s3 = "计算机";

System.out.println(s2);//计算机

System.out.println(s1 == s2);//false，因为一个是堆内存中的String对象一个是常量池中的String对象，

System.out.println(s3 == s2);//true，因为两个都是常量池中的String对象

**String 字符串拼接**

String str1 = "str";

String str2 = "ing";

String str3 = "str" + "ing";//常量池中的对象

String str4 = str1 + str2; //在堆上创建的新的对象

String str5 = "string";//常量池中的对象

System.out.println(str3 == str4);//false

System.out.println(str3 == str5);//true

System.out.println(str4 == str5);//false

**String s1 = new String("abc");这句话创建了几个对象？**

创建了两个对象:先有字符串"abc"放入常量池，然后 new 了一份字符串"abc"放入Java堆(字符串常量"abc"在编译期就已经确定放入常量池，而 Java 堆上的"abc"是在运行期初始化阶段才确定)，然后 Java 栈的 str1 指向Java堆上的"abc"。

**8种基本类型的包装类和常量池**

•Java 基本类型的包装类的大部分都实现了常量池技术，即Byte,Short,Integer,Long,Character,Boolean；这5种包装类默认创建了数值[-128，127]的相应类型的缓存数据，但是超出此范围仍然会去创建新的对象。

•两种浮点数类型的包装类 Float,Double 并没有实现常量池技术。

Integer i1 = 33;

Integer i2 = 33;

System.out.println(i1 == i2);// 输出true

Integer i11 = 333;

Integer i22 = 333;

System.out.println(i11 == i22);// 输出false

Double i3 = 1.2;

Double i4 = 1.2;

System.out.println(i3 == i4);// 输出false

应用场景：

•Integer i1=40；Java 在编译的时候会直接将代码封装成Integer i1=Integer.valueOf(40);，从而使用常量池中的对象。

•Integer i1 = new Integer(40);这种情况下会创建新的对象。

Integer i1 = 40;

Integer i2 = new Integer(40);

System.out.println(i1==i2);//输出false

Integer比较更丰富的一个例子:

Integer i1 = 40;

Integer i2 = 40;

Integer i3 = 0;

Integer i4 = new Integer(40);

Integer i5 = new Integer(40);

Integer i6 = new Integer(0);

System.out.println("i1=i2 " + (i1 == i2));

System.out.println("i1=i2+i3 " + (i1 == i2 + i3));

System.out.println("i1=i4 " + (i1 == i4));

System.out.println("i4=i5 " + (i4 == i5));

System.out.println("i4=i5+i6 " + (i4 == i5 + i6));

System.out.println("40=i5+i6 " + (40 == i5 + i6));

结果：

i1=i2 true

i1=i2+i3 true

i1=i4 false

i4=i5 false

i4=i5+i6 true

40=i5+i6 true

语句i4 == i5 + i6，因为+这个操作符不适用于Integer对象，首先i5和i6进行自动拆箱操作，进行数值相加，即i4 == 40。然后Integer对象无法与数值进行直接比较，所以i4自动拆箱转为int值40，最终这条语句转为40 == 40进行数值比较。

# 垃圾回收器

## G1

G1垃圾回收器是为多处理器和大内存的服务器而设计的，它根据运行JVM过程中构建的停顿预测模型（Pause Prediction Model）计算出来的历史数据来预测本次收集需要选择的Region数量，然后尽可能（不是绝对）满足GC的停顿时间，G1期望能让JVM的GC成为一件简单的事情。G1旨在延迟性和吞吐量之间取得最佳的平衡，它尝试解决有如下问题的Java应用：

堆大小能达到几十G甚至更大，超过50%的堆空间都是存活的对象；

对象分配和晋升的速度随着时间的推移有很大的影响；

堆上严重的碎片化问题；

可预测的停顿时间，避免长时间的停顿。

开启G1

在JDK9之前，JDK7和JDK8默认都是ParallelGC垃圾回收。到了JDK9，G1才是默认的垃圾回收器。所以如果JDK7或者JDK8需要使用G1的话，需要通过参数（-XX:+UseG1GC）显示执行垃圾回收器。

# ClassLoader

## ClassLoader 做什么的？

顾名思义，它是用来加载 Class 的。它负责将 Class 的字节码形式转换成内存形式的 Class 对象。字节码的本质就是一个字节数组 []byte，它有特定的复杂的内部格式。

每个 Class 对象的内部都有一个 classLoader 字段来标识自己是由哪个 ClassLoader 加载的。

## 延迟加载

JVM 运行并不是一次性加载所需要的全部类的，它是按需加载，也就是延迟加载。程序在运行的过程中会逐渐遇到很多不认识的新类，这时候就会调用 ClassLoader 来加载这些类。加载完成后就会将 Class 对象存在 ClassLoader 里面，下次就不需要重新加载了。

比如你在调用某个类的静态方法时，首先这个类肯定是需要被加载的，但是并不会触及这个类的实例字段，那么实例字段的类别 Class 就可以暂时不必去加载，但是它可能会加载静态字段相关的类别，因为静态方法会访问静态字段。而实例字段的类别需要等到你实例化对象的时候才可能会加载。

## 各司其职

JVM 运行实例中会存在多个 ClassLoader，不同的 ClassLoader 会从不同的地方加载字节码文件。

JVM 中内置了三个重要的 ClassLoader，分别是 BootstrapClassLoader、ExtensionClassLoader 和 AppClassLoader。

BootstrapClassLoader 负责加载 JVM 运行时核心类，这些类位于 JAVA\_HOME/lib/rt.jar 文件中，比如 java.util.\*、java.io.\*、java.nio.\*、java.lang.\* 等等。

ExtensionClassLoader 负责加载 JVM 扩展类，比如 swing 系列、内置的 js 引擎、xml 解析器 等等，这些库名通常以 javax 开头，它们的 jar 包位于 JAVA\_HOME/lib/ext/\*.jar 中。

AppClassLoader 才是直接面向我们用户的加载器，它会加载 Classpath 环境变量里定义的路径中的 jar 包和目录。

那些位于网络上静态文件服务器提供的 jar 包和 class文件，jdk 内置了一个 URLClassLoader，ExtensionClassLoader 和 AppClassLoader 都是 URLClassLoader 的子类。

## ClassLoader 传递性

程序在运行过程中，遇到了一个未知的类，它会选择哪个 ClassLoader 来加载它呢？虚拟机的策略是使用调用者 Class 对象的 ClassLoader 来加载当前未知的类。何为调用者 Class 对象？就是在遇到这个未知的类时，虚拟机肯定正在运行一个方法调用（静态方法或者实例方法），这个方法挂在哪个类上面，那这个类就是调用者 Class 对象。前面我们提到每个 Class 对象里面都有一个 classLoader 属性记录了当前的类是由谁来加载的。

因为 ClassLoader 的传递性，所有延迟加载的类都会由初始调用 main 方法的这个 ClassLoader 全全负责，它就是 AppClassLoader。

## 双亲委派

前面我们提到 AppClassLoader 只负责加载 Classpath 下面的类库，如果遇到没有加载的系统类库怎么办，AppClassLoader 必须将系统类库的加载工作交给 BootstrapClassLoader 和 ExtensionClassLoader 来做，这就是我们常说的「双亲委派」。

## Class.forName

当我们在使用 jdbc 驱动时，经常会使用 Class.forName 方法来动态加载驱动类。

其原理是 mysql 驱动的 Driver 类里有一个静态代码块，它会在 Driver 类被加载的时候执行。这个静态代码块会将 mysql 驱动实例注册到全局的 jdbc 驱动管理器里。

class Driver {

static {

try {

java.sql.DriverManager.registerDriver(new Driver());

} catch (SQLException E) {

throw new RuntimeException("Can't register driver!");

}

}

...

}

forName 方法同样也是使用调用者 Class 对象的 ClassLoader 来加载目标类。不过 forName 还提供了多参数版本，可以指定使用哪个 ClassLoader 来加载

通过这种形式的 forName 方法可以突破内置加载器的限制，通过使用自定类加载器允许我们自由加载其它任意来源的类库。根据 ClassLoader 的传递性，目标类库传递引用到的其它类库也将会使用自定义加载器加载。

## 自定义加载器

ClassLoader 里面有三个重要的方法 loadClass()、findClass() 和 defineClass()。

loadClass() 方法是加载目标类的入口，它首先会查找当前 ClassLoader 以及它的双亲里面是否已经加载了目标类，如果没有找到就会让双亲尝试加载，如果双亲都加载不了，就会调用 findClass() 让自定义加载器自己来加载目标类。ClassLoader 的 findClass() 方法是需要子类来覆盖的，不同的加载器将使用不同的逻辑来获取目标类的字节码。拿到这个字节码之后再调用 defineClass() 方法将字节码转换成 Class 对象。

自定义类加载器不易破坏双亲委派规则，不要轻易覆盖 loadClass 方法。否则可能会导致自定义加载器无法加载内置的核心类库。在使用自定义加载器时，要明确好它的父加载器是谁，将父加载器通过子类的构造器传入。如果父类加载器是 null，那就表示父加载器是「根加载器」。

## Class.forName vs ClassLoader.loadClass

这两个方法都可以用来加载目标类，它们之间有一个小小的区别，那就是 Class.forName() 方法可以获取原生类型的 Class，而 ClassLoader.loadClass() 则会报错。

URLClassLoader v1 = new URLClassLoader(new URL[]{new URL(v2dir)});

Class<?> depv1Class = v1.loadClass("Dep");

Object depv1 = depv1Class.getConstructor().newInstance();

depv1Class.getMethod("print").invoke(depv1);

即使是同样的字节码用不同的 ClassLoader 加载出来的类都不能算同一个类