目录

[Java学习笔记 2](#_Toc16499)

[Java环境配置 2](#_Toc24409)

[多态 2](#_Toc30906)

[final关键字 3](#_Toc3402)

[static关键字 3](#_Toc9392)

[序列化，反序列化，transient 关键字的作用 4](#_Toc14769)

[String ，StringBuffer,StringBuilder 5](#_Toc6593)

[反射 6](#_Toc19016)

[泛型 7](#_Toc31932)

[类加载器 8](#_Toc14006)

[双亲委派模型 9](#_Toc31862)

[类加载机制 9](#_Toc22813)

[Java虚拟机 12](#_Toc24227)

[垃圾回收机制 17](#_Toc19477)

# Java学习笔记

## Java环境配置

定义 系统 变量

JAVA\_HOME 值为 jdk的 本地路径 和 jre

TIM截图20181012192317

在 path 中 添加 %JAVA\_HOME%\bin

## 多态

我们都知道面向对象有四个基本特性：抽象、封装、继承、多态。这四个特性，概括起来可以这么理解，抽象、封装、继承是多态的基础，多态是抽象、封装、继承的表现。多态，是Java中非常重要的一个部分，所以今天来谈一下多态（Polymorphism）。

多态的作用：解耦。多态是设计模式的基础，大多数设计模式都用到了多态。

多态存在的三个条件：

1、有继承关系

2、子类重写父类方法

3、父类引用指向子类对象

一下三种无法体现多态，因为方法不能被重写。或者被外部引用

1、static方法，因为被static修饰的方法是属于类的，而不是属于实例的

2、final方法，因为被final修饰的方法无法被子类重写

3、private方法和protected方法，前者是因为被private修饰的方法对子类不可见，后者是因为尽管被protected修饰的方法可以被子类见到，也可以被子类重写，但是它是无法被外部所引用的，一个不能被外部引用的方法，怎么能谈多态呢

**多态的分类**

1、编译时多态，即方法的重载，从JVM的角度来讲，这是一种静态分派（static dispatch）

2、运行时多态，即方法的重写，从JVM的角度来讲，这是一种动态分派（dynamic dispatch）

**多态的总结：**

1、向上转型是自动的。即Father f = new Children()是自动的，不需要强转

2、向下转型要强转。即Children c = new Father()是无法编译通过的，必须要Children c = (Children)new Father()，让父类知道它要转成具体哪个子类

1. 父类引用指向子类对象，子类重写了父类的方法，调用父类的方法，实际调用的是子类重写了的父类的该方法。即Father f = new Children()，f.toString()实际上调用的是Children中的toString()方法

## final关键字

1、被final修饰的类不可以被继承

2、被final修饰的方法不可以被重写

3、被final修饰的变量不可以被改变

被final修饰不可变的是变量的引用，而不是引用指向的内容，引用指向的内容是可以改变的

**被final修饰的方法**，JVM会尝试为之寻求内联，这对于**提升Java的效率**是非常重要的。因此，假如能确定方法不会被继承，那么尽量将方法定义为final的

被final修饰的常量，在编译阶段会存入调用类的常量池中

## static关键字

1、被static修饰的变量属于类的而不属于对象，可以通过类名.变量名直接引用，而不需要new出一个类来

2、被static修饰的方法属于类方法，可以通过类名.方法名直接引用

被static修饰的变量、被static修饰的方法统一属于类的静态资源，是**类实例之间共享的**，换言之，一处变、处处变（通过实例改变静态变量，其他实例也随之改变）

静态资源属于类，但是是独立于类存在的。从JVM的类加载机制的角度讲，静态资源是类初始化的时候加载的，而非静态资源是类new的时候加载的。类的初始化早于类的new，比如Class.forName(“xxx”)方法，就是初始化了一个类，但是并没有new它，只是加载这个类的静态资源罢了。所以对于静态资源来说，它是不可能知道一个类中有哪些非静态资源的；但是对于非静态资源来说就不一样了，由于它是new出来之后产生的，因此属于类的这些东西它都能认识。所以上面的几个问题答案就很明确了：

1、静态方法能不能引用非静态资源？不能，new的时候才会产生的东西，对于初始化后就存在的静态资源来说，根本不认识它

2、静态方法里面能不能引用静态资源？可以，因为都是类初始化的时候加载的，大家相互都认识。

3、非静态方法里面能不能引用静态资源？可以，非静态方法就是实例方法，那是new之后才产生的，那么属于类的内容它都认识。

静态代码块：

静态块也是static的重要应用之一。也是用于初始化一个类的时候做操作用的，和静态变量、静态方法一样，静态块里面的代**码只执行一次**，且只在初始化类的时候执行

**静态资源的加载顺序是严格按照静态资源的定义顺序来加载的**

**静态代码块对于定义在它之后的静态变量，可以赋值，但是不能访问**。

静态代码块是严格按照父类静态代码块->子类静态代码块的顺序加载的，且只加载一次。

**import static** java.lang.Math.\*;

这是jdk1.5之后的新特性：指定导入类中的某个静态资源，可以直接引用

## 序列化，反序列化，transient 关键字的作用

平时我们在Java内存中的对象，是无法进行IO操作或者网络通信的，因为在进行IO操作或者网络通信的时候，人家根本不知道内存中的对象是个什么东西，因此必须将对象以某种方式表示出来，即存储对象中的状态。一个Java对象的表示有各种各样的方式，Java本身也提供给了用户一种表示对象的方式，那就是序列化。换句话说，序列化只是表示对象的一种方式而已

序列化：将一个对象转换成一串二进制表示的字节数组，通过保存或转移这些字节数据来达到持久化的目的。

反序列化：将字节数组重新构造成对象。

默认序列化：

序列化只需要实现java.io.Serializable接口就可以了。序列化的时候有一个serialVersionUID参数，Java序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化，Java虚拟机会把传过来的字节流中的serialVersionUID和本地相应实体类的serialVersionUID进行比较，如果相同就认为是一致的实体类，可以进行反序列化，否则Java虚拟机会拒绝对这个实体类进行反序列化并抛出异常。serialVersionUID有两种生成方式：

Java为用户定义了默认的序列化、反序列化方法，其实就是ObjectOutputStream的defaultWriteObject方法和ObjectInputStream的defaultReadObject方法。

实现Serializable接口的类才能被序列化。被 transient 和static 修饰的属性不能被序列化。

可以这么理解，序列化保存的是对象的状态（初始化过的属性值都被保存），而static修饰的属性是属于类的。

1、当父类继承Serializable接口时，所有子类都可以被序列化

2、子类实现了Serializable接口，父类没有，父类中的属性不能序列化（不报错，数据丢失），但是在子类中属性仍能正确序列化

3、如果序列化的属性是对象，则这个对象也必须实现Serializable接口，否则会报错

4、反序列化时，如果对象的属性有修改或删减，则修改的部分属性会丢失，但不会报错

5、反序列化时，如果serialVersionUID被修改，则反序列化时会失败

## String ，StringBuffer,StringBuilder

引言：String s1=”123” String s2=”123” s1==s2 结果返回true

String s1 =new String(“123”) String s2 =new String(“123”)

s1== s2 结果 false

== 比较的是两个对象的引用，而不是两个对象的值

解释：

在JVM中有一块区域叫做常量池，**常量池**中的数据是那些在**编译期间被确定**，并被保存在已编译的.class文件中的一些数据。除了包含所有的8种基本数据类型（char、byte、short、int、long、float、double、boolean）外，还有String及其数组的常量值，另外还有一些以文本形式出现的符号引用。

Java栈的特点是存取速度快（比堆块），但是空间小，数据生命周期固定，只能生存到方法结束。我们定义的boolean b = true、char c = 'c'、String str = “123”，这些语句，我们拆分为几部分来看：

1：true、c、123，这些等号右边的指的**是编译期间可以被确定的内容**，都被维护在常量池中

2：b、c、str这些等号左边第一个出现的指的是一个引用，引用的内容是等号右边数据在**常量池中的地址**

3：boolean、char、String这些是引用的类型

栈有一个特点，就是数据共享。回到我们第一个例子，第五行String str0 = "123"，编译的时候，在常量池中创建了一个常量"123"，然后走第六行String str1 = "123"，先去常量池中找有没有这个"123"，发现有，str1也指向常量池中的"123"，所以第七行的str0 == str1返回的是true，因为str0和str1指向的都是常量池中的"123"这个字符串的地址。当然如果String str1 = "234"，就又不一样了，因为常量池中没有"234"，所以会在常量池中创建一个"234"，然后str1代表的是这个"234"的地址。分析了String，其实其他基本数据类型也都是一样的：先看常量池中有没有要创建的数据，有就返回数据的地址，没有就创建一个。

第二个例子呢？Java虚拟机的解释器每遇到一个new关键字，都会在**堆内存中**开辟一块内存来存放一个String对象，所以str2、str3指向的堆内存中虽然存储的是相等的"234"，但是由于是两块不同的堆内存，因此str2 == str3返回的仍然是false

一个开发原则： 利用StringBuilder和StringBuffer拼接字符串

尽量少用 + 号来拼接字符串

原因：编译器每次碰到"+"的时候，会new一个StringBuilder出来，接着调用append方法，在调用toString方法，生成新字符串那么，这意味着，如果代码中有很多的"+"，就会每个"+"生成一次StringBuilder，这种方式对内存是一种浪费，效率很不好。

StringBuffer和StringBuilder原理一样，在底层维护了一个char数组，每次append的时候就往char数组里面放字符而已，在最终sb.toString()的时候，用一个new String()方法把char数组里面的内容都转成String，这样，整个过程中只产生了一个StringBuilder对象与一个String对象，非常节省空间。StringBuilder唯一的性能损耗点在于char数组不够的时候需要进行扩容，扩容需要进行数组拷贝

StringBuffer和StringBuilder用法一模一样，唯一的区别只是StringBuffer是线程安全的，它对所有方法都做了同步，StringBuilder是线程非安全的，所以在不涉及线程安全的场景，比如方法内部，尽量使用StringBuilder，避免同步带来的消耗。

另外，StringBuffer和StringBuilder还有一个优化点，上面说了，扩容的时候有性能上的损耗，那么如果可以估计到要拼接的字符串的长度的话，尽量利用构造函数指定他们的长度。

## 反射

运用反射 会降低程序的执行效率

setAccessible(true)跳过安全检测会提高程序的运行速度

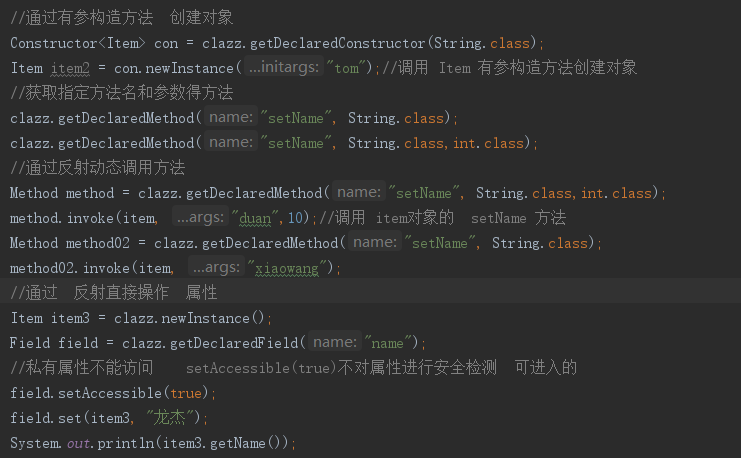
获取 某个类的 Class对象 有3 中方式

类名.class 对象.getClass() Class.forName(“类全限定名”)

Class 类的方法

newInstance() //通过无参构造器创建对象

Class 类的其他方法



## 泛型

泛型的目的是 解决类型强转的问题

**泛型类 在实例化的时候指明泛型的具体类型**

public class Test<T>{

private T t;

} // T 只是一个标识符，别的字母表示也可以

**泛型接口**

public interface Test<T>{

private T t;

**}**

实现类的两种实现方式

public class Son<T> implements Test<T>

public class Son implements Test<String>

**泛型方法 在调用方法的时候指明泛型的具体类型**

<T> 在public 后面的方法 才是泛型方法，方法形参 使用泛型，并不能为泛型方法

只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。

<T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。

public <T> T function(Class<T> clazz){

T instance = clazz.newInstance();

}

public <T> void function( T... args)

**泛型通配符 ？**

public void function(Test<?> obj)

泛型上下界

public void function(Test<? extends String> t）

**DateFomat**

**HashCode的作用**

## 类加载器

虚拟机设计团队把类加载阶段张的"通过一个类的全限定名来获取此类的二进制字节流"这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为"类加载器"。类加载器虽然只用于实现类的加载动作，但它在Java程序中起到的作用却远远不限定于类加载阶段。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一同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话表达地再简单一点就是：比较两个类是否"相等"，只有在这两个类是由同一个类加载器加载的前提下才有意义，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，只要加载它们的类加载器不同，这两个类必定不相等。

从Java虚拟机的角度讲，只有两种不同的类加载器：启动类加载器Bootstrap ClassLoader，这个类加载器是由C++语言实现的，是虚拟机自身的一部分；其他类加载器，这些类加载器都由Java语言实现，独立于虚拟机外部，并且全部继承自java.lang.ClassLoader。从开发人员的角度讲，类加载器还可以划分地更加细致一些

启动类加载器Bootstrap ClassLoader：jvm内核中的加载器，负责加载JAVA\_HOME/lib下的类库

扩展类加载器Extension ClassLoader：加载JAVA\_HOME/lib/ext目录中类库

应用程序类加载器Application ClassLoader（系统类加载器）：

只会加载java.class.path下的.class文件（项目bin目录下的字节码文件或者类路径下的）

## 双亲委派模型

1、如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此。

2、只有当父加载器反馈自己无法完成这这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载

所以，其实所有的加载请求最终都应该传送到顶层的启动类加载器中。双亲委派模型对于Java程序的稳定运作很重要，因为Java类随着它的加载器一起具备了一种带有优先级的层次关系。例如java.lang.Object，存放于rt.jar中，无论哪一个类加载器要去加载这个类，最终都是由Bootstrap ClassLoader去加载，因此Object类在程序的各种类加载器环境中都是一个类。相反，如果没有双亲委派模型，由各个类自己去加载的话，如果用户自己编写了一个java.lang.Object，并放在CLASSPATH下，那系统中将会出现多个不同的Object类，Java体系中最基础的行为也将无法保证，应用程序也将会变得一片混乱。

## 类加载机制

我们知道我们写的程序经过编译后成为了.class文件，.class文件中描述了类的各种信息，最终都需要加载到虚拟机之后才能运行和使用。而虚拟机如何加载这些.class文件？.class文件的信息进入到虚拟机后会发生什么变化？类加载的每个阶段虚拟机需要做什么？

类从被加载到虚拟机内存中开始，到卸载出内存，它的整个生命周期包括：

**加载**（Loading）、**验证**（Verification）、**准备**（Preparation）、**解析**（Resolution）、**初始化**（Initiallization）、**使用**（Using）和**卸载**（Unloading）这7个阶段。其中验证、准备、解析3个部分统称为连接（Linking）

前五个步骤完整组成了类加载的过程

#### 加载Loading

加载是类加载的第一个阶段。有两种时机会触发类加载：

1. 预加载。虚拟机启动时加载，加载的是JAVA\_HOME/lib/下的rt.jar下的.class文件，这个jar包里面的内容是程序运行时经常用到的，像java.lang.\*、java.util.\*、java.io.\*等等，因此随着虚拟机一起加载。要证明这一点很简单，写一个空的main函数，设置虚拟机参数为"-XX:+TraceClassLoading"来获取类加载信息，运行一下。

2、运行时加载。虚拟机在用到一个.class文件的时候，会先去内存中查看一下这个.class文件有没有被加载，如果没有就会按照类的全限定名来加载这个类。

那么，加载阶段做了什么，其实加载阶段做了有三件事情：

1、获取.class文件的二进制流

2、将类信息、静态变量、字节码、常量这些.class文件中的内容放入方法区中

3、在内存中生成一个代表这个.class文件的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。一般这个Class是在堆里的，不过HotSpot虚拟机比较特殊，这个Class对象是放在方法区中的

虚拟机规范对这三点的要求并不具体，因此虚拟机实现与具体应用的灵活度都是相当大的。例如第一条，根本没有指明二进制字节流要从哪里来、怎么来，因此单单就这一条，就能变出许多花样来：

· 从zip包中获取，这就是以后jar、ear、war格式的基础

· 从网络中获取，典型应用就是Applet

· 运行时计算生成，典型应用就是动态代理技术

· 由其他文件生成，典型应用就是JSP，即由JSP生成对应的.class文件

· 从数据库中读取，这种场景比较少见

总而言之，在类加载整个过程中，这部分是对于开发者来说可控性最强的一个阶段。

**验证**

连接阶段的第一步，这一阶段的目的是为了确保.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。

Java语言本身是相对安全的语言（相对C/C++来说），但是前面说过，.class文件未必要从Java源码编译而来，可以使用任何途径产生，甚至包括用十六进制编辑器直接编写来产生.class文件。在字节码语言层面上，Java代码至少从语义上是可以表达出来的。虚拟机如果不检查输入的字节流，对其完全信任的话，很可能会因为载入了有害的字节流而导致系统崩溃，所以验证是虚拟机对自身保护的一项重要工作。

验证阶段将做一下几个工作，具体就不细讲了，这是虚拟机实现层面的问题：

1、文件格式验证

这个地方要说一点和开发者相关的。.class文件的第5~第8个字节表示的是该.class文件的主次版本号，验证的时候会对这4个字节做一个验证，高版本的JDK能向下兼容以前版本的.class文件，但不能运行以后的class文件，即使文件格式未发生任何变化，虚拟机也必须拒绝执行超过其版本号的.class文件。举个具体的例子，如果一段.java代码是在JDK1.6下编译的，那么JDK1.6、JDK1.7的环境能运行这个.java代码生成的.class文件，但是JDK1.5、JDK1.4乃更低的JDK版本是无法运行这个.java代码生成的.class文件的。如果运行，会抛出java.lang.UnsupportedClassVersionError，这个小细节，务必注意。

2、元数据验证3、字节码验证4、符号引用验证

**准备**

准备阶段是正式为类变量分配内存并设置其初始值的阶段，这些变量所使用的内存都将在方法区中分配。关于这点，有两个地方注意一下：

1、这时候进行内存分配的仅仅是类变量（被static修饰的变量），而不是实例变量，实例变量将会在对象实例化的时候随着对象一起分配在Java堆中

2、这个阶段赋初始值的变量指的是那些不被final修饰的static变量，比如"public static int value = 123;"，value在准备阶段过后是0而不是123，给value赋值为123的动作将在初始化阶段才进行；比如"public static final int value = 123;"就不一样了，在准备阶段，虚拟机就会给value赋值为123。

**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。来了解一下符号引用和直接引用有什么区别：

1、符号引用。

这个其实是属于编译原理方面的概念，符号引用包括了下面三类常量：

· 类和接口的全限定名

· 字段的名称和描述符

· 方法的名称和描述符

那其实总而言之，符号引用和我们上面讲的是一样的，是对于类、变量、方法的描述。符号引用和虚拟机的内存布局是没有关系的，引用的目标未必已经加载到内存中了。

2、直接引用

直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同的虚拟机示例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经存在在内存中了。

**初始化**

初始化阶段是类加载过程的最后一步，初始化阶段是真正执行类中定义的Java程序代码（或者说是字节码）的过程。初始化过程是一个执行类构造器<clinit>()方法的过程，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。把这句话说白一点，其实**初始化阶段做的事就是给static变量赋予用户指定的值以及执行静态代码块**。

注意一下，**虚拟机会保证类的初始化在多线程环境中被正确地加锁、同步**，即如果多个线程同时去初始化一个类，那么只会有一个类去执行这个类的<clinit>()方法，其他线程都要阻塞等待，直至活动线程执行<clinit>()方法完毕。因此如果在一个类的<clinit>()方法中有耗时很长的操作，就可能造成多个进程阻塞。不过其他线程虽然会阻塞，但是执行<clinit>()方法的那条线程退出<clinit>()方法后，其他线程不会再次进入<clinit>()方法了，因为同一个类加载器下，一个类只会初始化一次。实际应用中这种阻塞往往是比较隐蔽的，要小心。

Java虚拟机规范严格规定了有且只有5种场景必须立即对类进行初始化，这4种场景也称为对一个类进行**主动引用**（其实还有一种场景，不过暂时我还没弄明白这种场景的意思，就先不写了）：

1、使用new关键字实例化对象、读取或者设置一个类的静态字段（被final修饰的静态字段除外）、调用一个类的静态方法的时候

2、使用java.lang.reflect包中的方法对类进行反射调用的时候

3、初始化一个类，发现其父类还没有初始化过的时候

4、虚拟机启动的时候，虚拟机会先初始化用户指定的包含main()方法的那个类

除了上面4种场景外，所有引用类的方式都不会触发类的初始化，称为被动引用，接下来看下被动引用的几个例子：

1、子类引用父类静态字段，不会导致子类初始化。至于子类是否被加载、验证了，前者可以通过"-XX:+TraceClassLoading"来查看

3、引用静态常量时，常量在编译阶段会存入类的常量池中，本质上并没有直接引用到定义常量的类

在编译阶段通过常量传播优化，常量HELLOWORLD的值"Hello World"实际上已经存储到了NotInitialization类的常量池中，以后NotInitialization对常量ConstClass.HELLOWORLD的引用实际上都被转化为NotInitialization类对自身常量池的引用了。也就是说，实际上的NotInitialization的Class文件中并没有ConstClass类的符号引用入口，这两个类在编译成Class之后就不存在任何联系了。

**Comparable和Comparator的区别（有机会再学习 ）**

## Java虚拟机

为什么Java可以实现所谓的“一次编写，到处运行”，主要是因为虚拟机的存在。Java虚拟机负责Java程序设计语言的安全特性和平台无关性。Java虚拟机屏蔽了与具体操作系统平台相关的信息，使得Java语言编译器只需要生成在Java虚拟机上运行的字节码，就可以在多种平台上不加修改地运行。Java虚拟机使得Java摆脱了具体机器的束缚，使跨越不同平台编写程序成为了可能。

Java虚拟机基本上都是JDK自带的虚拟机HotSpot，这款虚拟机也是目前商用虚拟中市场份额最大的一款虚拟机，可以通过在命令行程序中输入“java -version”

计算机存储单位：

从小到大依次为位Bit、字节Byte、千字节KB、兆M、千兆GB、TB，相邻单位之间都是1024倍，1024为2的10次方

1Byte = 8bit

1K = 1024Byte

1M = 1024K

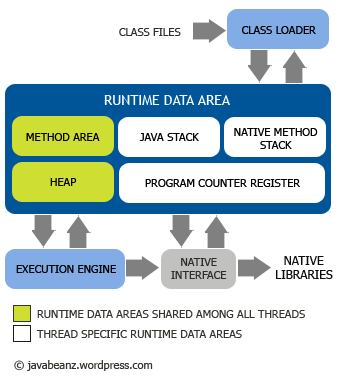
1G = 1024M

1T = 1024G

前言：

对于Java程序员来说，在虚拟机自动内存管理机制的帮助下，不再需要为每一个new操作去配对delete/free代码，不容易出现内存泄露和内存溢出问题。但是，也正是因为Java把内存控制权交给了虚拟机，一旦出现内存泄露和内存溢出的问题，就难以排查，因此一个好的Java程序员应该去了解虚拟机的内存区域以及会引起内存泄露和内存溢出的场景。

**Jvm的内存区域：**



**1、线程独有的内存区域**

（1）PROGRAM COUNTER REGISTER，程序计数器

这块内存区域很小，它是当前线程所执行的字节码的行号指示器，字节码解释器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。Java方法这个计数器才有值，如果执行的是一个Native方法，那这个计数器是空的。

（2）JAVA STACK，虚拟机栈

生命周期和线程相同。每个方法执行的同时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息，每一个方法从调用直至执行完毕的过程，就对应着一个栈帧在虚拟机中入栈到出栈的过程。栈的大小和具体JVM的实现有关，通常在256K~756K之间。

（3）NATIVE METHOD STACK，方法栈

和虚拟机栈起的作用一样，只不过方法栈为虚拟机使用到的Native方法服务。虚拟机规范并没有对这个区域有什么强制规定，因此我们使用的HotSpot虚拟机，就干脆没有这块区域了，它和虚拟机栈是一起的。

**2、线程间共享的内存区域**

（1）HEAP，堆

大多数应用，堆都是Java虚拟机所管理的内存中最大的一块，它在虚拟机启动时创建，此内存唯一的目的就是存放对象实例。由于现在垃圾收集器采用的基本都是分代收集算法，所以堆还可以细分为新生代和老年代，再细致一点还有Eden区、From Survivior区、To Survivor区，这个后面都会讲到的。

（2）METHOD AREA，方法区

这块区域用于存储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，虚拟机规范是把这块区域描述为堆的一个逻辑部分的，但实际它应该是要和堆区分开的。从上面提到的分代收集算法的角度看，HotSpot中，方法区≈永久代。不过JDK 7之后，我们使用的HotSpot应该就没有永久代这个概念了，会采用Native Memory来实现方法区的规划了。

（3）RUNTIME CONSTANT POOL，运行时常量池

上面的图中没有画出来，因为它是方法区的一部分。Class文件中除了有类的版本信息、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译期间生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中，另外翻译出来的直接引用也会存储在这个区域中。这个区域另外一个特点就是动态性，Java并不要求常量就一定要在编译期间才能产生，运行期间也可以在这个区域放入新的内容，String.intern()方法就是这个特性的应用。

**3、直接内存**

想想还是把这块加上。直接内存并不是虚拟机运行时数据区的一部分，也不是Java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，而且也可能导致内存溢出问题。JDK1.4中新增加了NIO，引入了一种基于通道与缓冲区的I/O方式，它可以使用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。这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回复制数据。显然，本机直接内存的分配不会受到Java堆大小的限制，但是，既然是内存，肯定还是会受到本机总内存（包括RAM、SWAP区）大小以及处理器寻址空间的限制。

**对象创建**

1、虚拟机遇到一条new指令，首先去检查这个指令的参数能否在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化。如果没有，那么必须先执行类的初始化过程。

2、类加载检查通过后，虚拟机为新生对象分配内存。对象所需内存大小在类加载完成后便可以完全确定，为对象分配空间无非就是从Java堆中划分出一块确定大小的内存而已。这个地方会有两个问题：

（1）如果内存是规整的，那么虚拟机将采用的是指针碰撞法来为对象分配内存。意思是所有用过的内存在一边，空闲的内存在另外一边，中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针向空闲那边挪动一段与对象大小相等的距离罢了。如果垃圾收集器选择的是Serial、ParNew这种基于压缩算法的，虚拟机采用这种分配方式。

（2）如果内存不是规整的，已使用的内存和未使用的内存相互交错，那么虚拟机将采用的是空闲列表法来为对象分配内存。意思是虚拟机维护了一个列表，记录上哪些内存块是可用的，再分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的内容。如果垃圾收集器选择的是CMS这种基于标记-清除算法的，虚拟机采用这种分配方式。

另外一个问题及时保证new对象时候的线程安全性。因为可能出现虚拟机正在给对象A分配内存，指针还没有来得及修改，对象B又同时使用了原来的指针来分配内存的情况。虚拟机采用了CAS配上失败重试的方式保证更新更新操作的原子性和TLAB两种方式来解决这个问题。

3、内存分配结束，虚拟机将分配到的内存空间都初始化为零值（不包括对象头）。这一步保证了对象的实例字段在Java代码中可以不用赋初始值就可以直接使用，程序能访问到这些字段的数据类型所对应的零值。

4、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的对象头中。

5、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。

**对象的定位方式：**

建立对象是为了使用对象，Java程序需要通过栈上的reference（引用）数据来操作堆上的具体对象。

Object obj = new Object()

而new Object()之后其实有两部分内容，一部分是类数据（比如代表类的Class对象）、一部分是实例数据

由于reference在Java虚拟机规范中只是一个指向对象new Object()的引用obj，并没有规定obj应该通过何种方式去定位、访问堆中对象的具体位置，所以对象访问方式也是取决于虚拟机而定的。主流方式有两种：

1、句柄访问。Java堆中划分出一块句柄池，obj指向的是对象的句柄地址，句柄中则包含了类数据的地址和实例数据的地址 访问对象需要2次定位

2、指针（直接）访问。对象中存储所有的实例数据和类数据的地址，obj指向的是这个对象HotSpot虚拟机采用的是后者，不过前者的对象访问方式也是十分常见的。对象的访问是非常频繁的，因此这种访问方式效率更高。

访问对象 只需要 一次定位

内存泄露：是指程序中己动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

内存溢出：

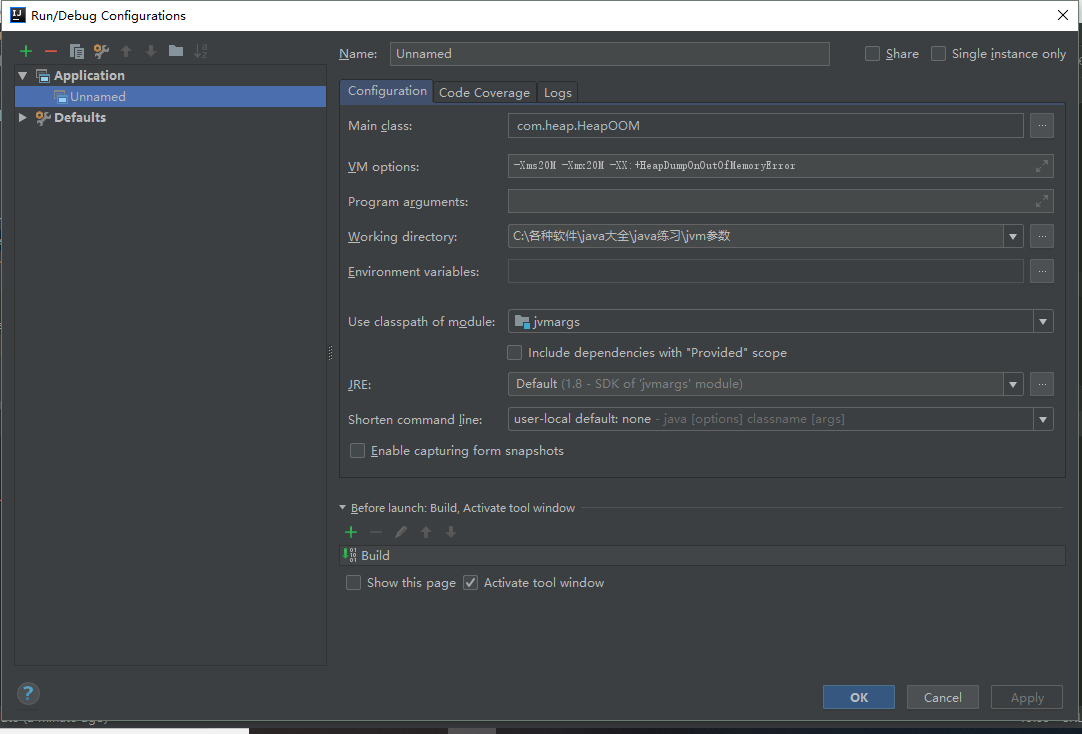
OutOfMemoryError异常（OOM）：

目的：1：通过代码验证jvm中描述的各个运行时区域储存的内容。

2：在遇到内存溢出异常时，能根据异常信息快速判断是哪个区域的内存溢出。

**常用的jvm参数**

Idea中可以在 这里设置jvm参数

****

**测试Java堆溢出：（由于堆内存中对象过多 没被回收）**

堆的最小值，-Xms 参数 和堆的最大值，-Xmx参数

参数 -XX:+HeapDumpOnOutOfMemoryError 可以让虚拟机在出现内存溢出异常时dump出当前内存堆转储快照，以分析遍

**此区域异常的解决：**

一段通过内存映像分析工具（Idea 中的 JProfile），对dump出来的堆转储快照进行内存分析。

虚拟机栈和本地方法栈溢出：

（如果线程的请求深度太深可以理解为方法调用深度太深，就会产生栈溢出）

由于HotSpot 不区分这2个栈，所以，-Xoss参数(设置本地方法栈的大小)是无效的。 栈容量只由 -Xss参数设定。

Jvm描述了两种异常：

1 ：线程请求的栈深度大于jvm所允许的最大深度，抛出StackOverflowError

2：jvm在扩展栈时无法申请到足够的内存空间，抛出 OutOfMemoryError

在单线程环境下，无论是栈帧太大，还是jvm栈容量太小，当内存无法分配的时候，抛出的都是StackOverflowError

在多线程环境下，每个线程分配的栈容量越大（反而越容易产生内存溢出），可以建立的线程数就越少，建立线程时就越容易把剩下的内存耗尽

StackOverFlowError这个异常，有错误堆栈可以阅读，比较好定位。而且如果使用虚拟机默认参数，栈深度在大多数情况下，达到1000~2000完全没有问题，正常方法的调用这个深度应该是完全够了。

线程占用的是虚拟机栈和本地方法栈。但是如果建立过多线程导致的OutOfMemoryError，在不能减少线程数或者更换64位虚拟机的情况下，就只能通过减小最大堆容量和减小栈容量来换取更多的线程了。

## 垃圾回收机制

1：哪些 “垃圾需要回收”

可达性分析法

这个算法的基本思想是通过一系列称为“GC Roots”的对象作为起始点，从这些节点向下搜索，搜索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（即GC Roots到对象不可达）时，则证明此对象是不可用的。在Java语言中可以作为GC Roots的对象包括：

· 虚拟机栈中引用的对象

· 方法区中静态属性引用的对象

· 方法区中常量引用的对象

· 本地方法栈中JNI（即Native方法）引用的对

2：四种引用状态

强引用

代码中普遍存在的类似"Object obj = new Object()"这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象

软引用

描述有些还有用但并非必需的对象。在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围进行二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。Java中的类SoftReference表示软引用

弱引用

描述非必需对象。被弱引用关联的对象只能生存到下一次垃圾回收之前，垃圾收集器工作之后，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。Java中的类WeakReference表示弱引用

虚引用

这个引用存在的唯一目的就是在这个对象被收集器回收时收到一个系统通知，被虚引用关联的对象，和其生存时间完全没关系。Java中的类PhantomReference表示虚引用

3：方法区回收

虚拟机规范中不要求方法区一定要实现垃圾回收，而且方法区中进行垃圾回收的效率也确实比较低，但是HotSpot对方法区也是进行回收的，主要回收的是废弃常量和无用的类两部分。判断一个常量是否“废弃常量”比较简单，只要当前系统中没有任何一处引用该常量就好了，但是要判定一个类是否“无用的类”条件就要苛刻很多，类需要同时满足以下三个条件：

（1）该类所有实例都已经被回收，也就是说Java堆中不存在该类的任何实例

（2）加载该类的ClassLoader已经被回收

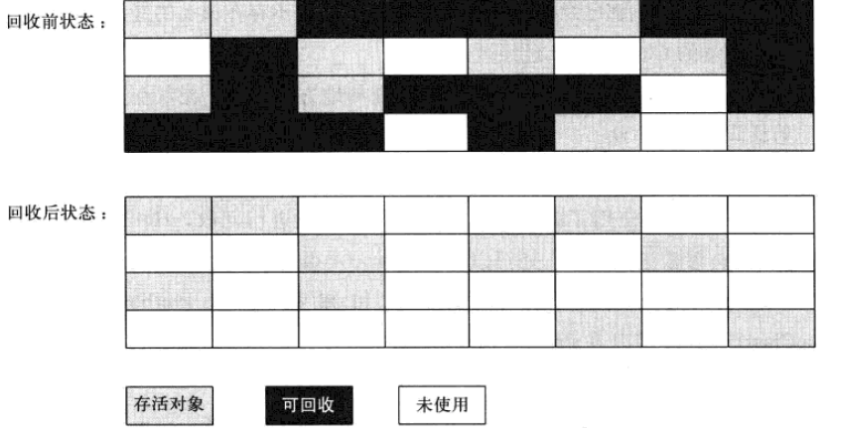
（3）该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法

在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP以及OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载功能，以保证方法区不会溢出。

4：怎么回收（回收算法）

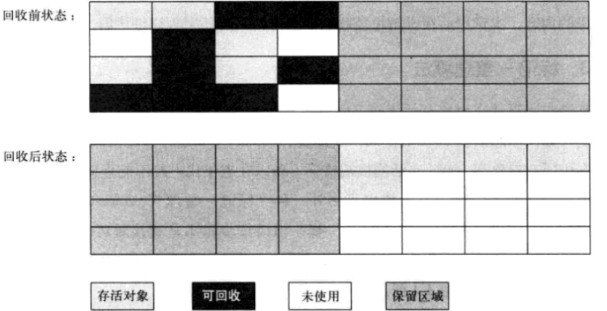
1. 标记-清除（Mark-Sweep）算法

这是最基础的算法，标记-清除算法就如同它的名字样，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。这种算法的不足主要体现在效率和空间，从效率的角度讲，标记和清除两个过程的效率都不高；从空间的角度讲，标记清除后会产生大量不连续的内存碎片， 内存碎片太多可能会导致以后程序运行过程中在需要分配较大对象时，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作。标记-清除算法执行过程如图：



2、复制（Copying）算法

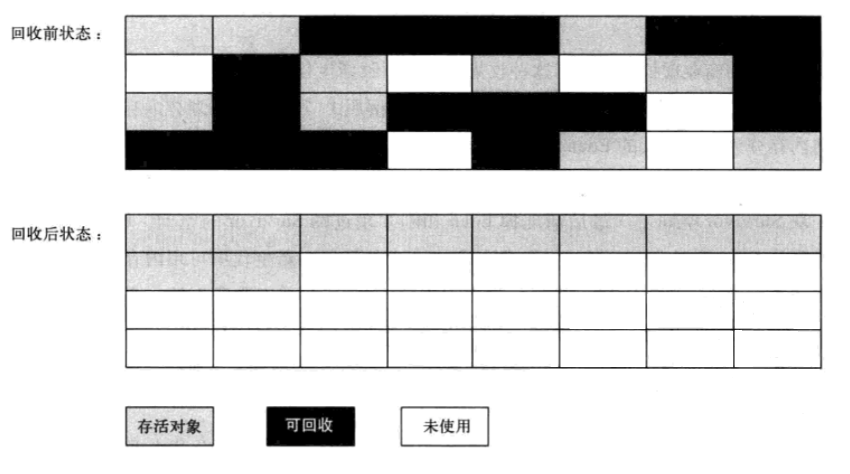
复制算法是为了解决效率问题而出现的，**它将可用的内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉**。这样每次只需要对整个半区进行内存回收，内存分配时也不需要考虑内存碎片等复杂情况，只需要移动指针，按照顺序分配即可。复制算法的执行过程如图：



不过这种算法有个缺点，内存缩小为了原来的一半，这样代价太高了。现在的商用虚拟机都采用这种算法来**回收新生代**，不过研究表明1:1的比例非常不科学，因此**新生代的内存被划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。**每次回收时，将Eden和Survivor中还存活着的对象一次性复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden区和Survivor区的比例为8:1，意思是每次新生代中可用内存空间为整个新生代容量的90%。当然，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖老年代进行分配担保（Handle Promotion）。

3、标记-整理（Mark-Compact）算法

复制算法在对象存活率较高的场景下要进行大量的复制操作，效率很低。万一对象100%存活，那么需要有额外的空间进行分配担保。老年代都是不易被回收的对象，对象存活率高，因此一般不能直接选用复制算法。根据老年代的特点，有人提出了另外一种标记-整理算法，过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。标记-整理算法的工作过程如图：



内存分配原则：

**对象优先分配在Eden区上**

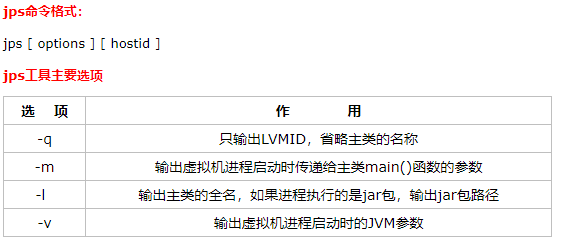
**大对象直接进入老年代**

**1、长期存活的对象将进入老年代。Eden区中的对象在一次Minor GC后没有被回收，则对象年龄+1，当对象年龄达到“-XX:MaxTenuringThreshold”设置的值的时候，对象就会被晋升到老年代中**

1. **Survivor空间中相同年龄的所有对象大小总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到“-XX:MaxTenuringThreshold”设置要求的年龄**

****虚拟机性能监控与故障处理工具****

**JDK的很多小工具的名字都参考了UNIX命令的命名方式，jps（JVM Process Status）是其中的典型。除了名字像UNIX的ps命令外，它的功能也和ps命令类似：可以列出正在运行的虚拟机进程，并显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一ID（Local Virtual Machine Identifier,LVMID）。虽然功能比较单一，但它是使用最高的JDK命令行工具，因为其他的JDK工具大多需要输入它查询到的LVMID来确定要监控的是哪一个虚拟机进程。**

****

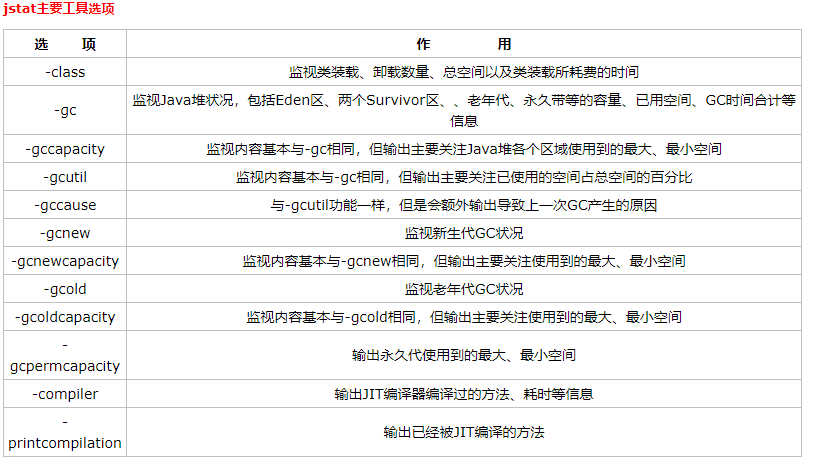
**jstat：虚拟机统计信息监控工具**

**jstat（JVM Statistics Monitoring Tool）使用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程（需要远程主机提供RMI支持）虚拟机进程中的类信息、内存、垃圾收集、JIT编译等运行数据，在没有GUI，只提供了纯文本控制台环境的服务器上，它将是运行期间定位虚拟机性能问题的首选工具。jstat命令格式**

**jstat [ option vmid [ interval [ s | ms ] [ count ] ] ]**

**这个VMID，对于本地虚拟机进程而言，VMID和LVMID是一致的。参数interval和count分别表示查询间隔和次数，如果省略这两个参数，说明只查询一次，假设需要每250毫秒查询一次进程2764的垃圾收集情况，一共查询20次，那命令应当是：**

**jstat -gc 2764 250 20**

****

**可以利用可视化工具对指定PID(进程id)的虚拟机运行时信息进行监控**

1、JConsole

在Java\_HOME/bin目录下，有一个jconsole.exe，双击运行一下就可以了。

2、Visual VM 等等等

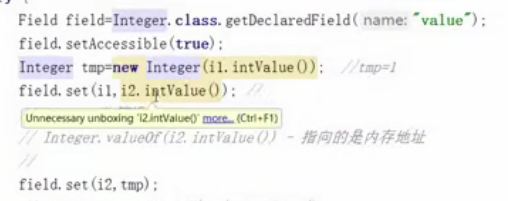
## Java中基本类型和引用类型

值传递和 引用传递

值传递，传递的是 实参的副本，引用类型传递的是 它的引用

值传递，也可以 有办法通过形参改变实参。

利用 反射 可以实现



## 数据类型

### Integer

装箱，拆箱 引起的 缓存问题

IntegerCache ：会对 -128-127之间的数字做出缓存处理

Integer a=1,b=1 a==b 结果为 true，比较的是内存地址，a和b在同一个缓存内，地址相同

Integer c=300 ,d=300 c==d 结果 为false

装箱操作 Integer.valueOf()

拆箱操作 Integer.intValue()

## 锁