类加载过程

<https://blog.csdn.net/lijingjingchn/article/details/111605394>

Java创建对象的过程



**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**

类加载完成以后，虚拟机就开始为对象分配内存，此时所需内存的大小就已经确定了。只需要在堆上分配所需要的内存即可。

1. 具体的分配内存有两种情况：第一种情况是内存空间绝对规整，第二种情况是内存空间是不连续的。

* 对于内存绝对规整的情况相对简单一些，虚拟机只需要在被占用的内存和可用空间之间移动指针即可，这种方式被称为**指针碰撞**。
* 对于内存不规整的情况稍微复杂一点，这时候虚拟机需要维护一个列表，来记录哪些内存是可用的。分配内存的时候需要找到一个可用的内存空间，然后在列表上记录下已被分配，这种方式成为**空闲列表**。

1. 分配内存的时候也需要考虑线程安全问题，有两种解决方案：

* 第一种是采用同步的办法，使用CAS+失败重试来保证操作的原子性。
* 第二种一种是每个线程分配内存都在自己的空间内进行，即是每个线程都在堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB），分配内存的时候再TLAB上分配，互不干扰

**Step3:初始化零值**

对象的内存分配完成后，还需要将对象的内存空间都初始化为零值，这样能保证对象即使没有赋初值，也可以直接使用

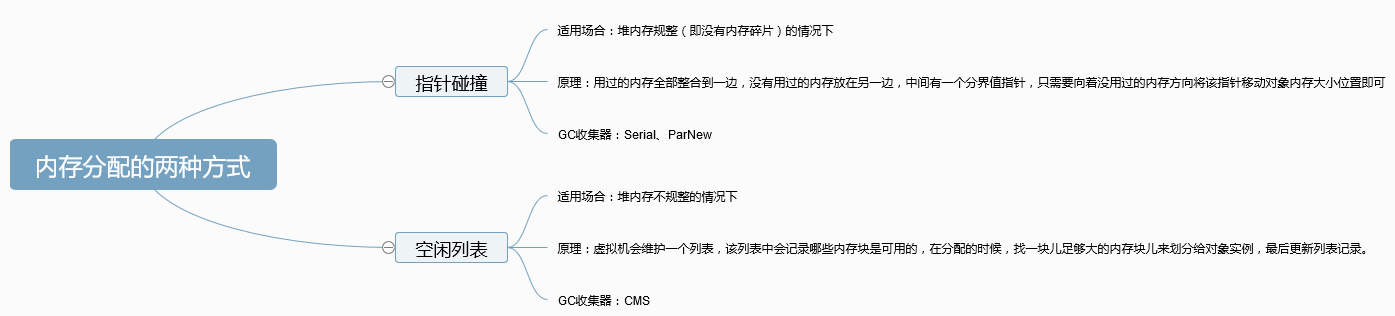
**Step4:设置对象头**

初始化零值完成之后，虚拟机要对对象进行必要的设置，包括这个对象所属的类，类的元数据信息，对象的hashcode，GC分代年龄等信息。 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。

内存分配方式



其实创建对象的方式有四种：

* 用new关键字创建
* 调用对象的clone方法
* 利用反射，调用Class类的或者是Constructor类的newInstance（）方法
* 用反序列化，调用ObjectInputStream类的readObject（）方法

CAS

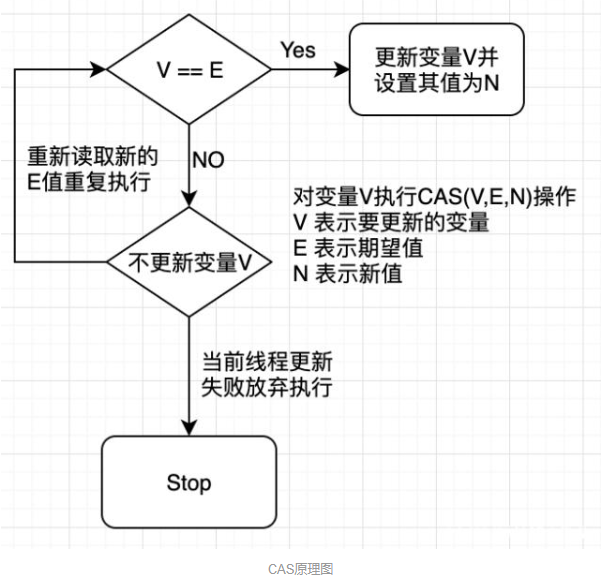
CAS(CompareAndSwap)比较并交换，是一种实现并发算法时常用到的技术，Java并发包中的很多类都使用了CAS技术。CAS具体包括三个参数：当前内存值V、旧的预期值A、即将更新的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做，并返回false。

函数公式：CAS(V,E,N)

V：表示要更新的变量

E：表示预期值

N：表示新值



原子变量类（例如java并发包原子操作类 [公式] 中的 AtomicXXX）就使用了这种高效的CAS操作。

CAS的缺点：

【1】循环时间长、开销很大。

当某一方法比如：getAndAddInt执行时，如果CAS失败，会一直进行尝试。如果CAS长时间尝试但是一直不成功，可能会给CPU带来很大的开销。

【2】只能保证一个共享变量的原子操作。

当操作1个共享变量时，我们可以使用循环CAS的方式来保证原子操作，但是操作多个共享变量时，循环CAS就无法保证操作的原子性，这个时候就需要用锁来保证原子性。

【3】存在ABA问题

CAS 会导致“ABA 问题”。CAS 算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。

部分乐观锁的实现是通过版本号(version)的方式来解决 ABA 问题，乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行+1 操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现 ABA 问题，因为版本号只会增加不会减少

### 1.String 类和常量池

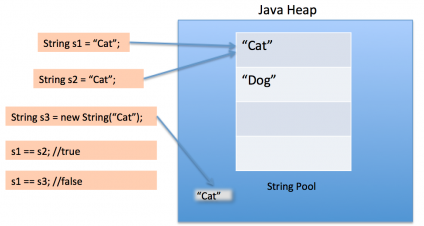
**String 对象的两种创建方式：**

* String str1 = "abcd";//先检查字符串常量池中有没有"abcd"，如果字符串常量池中没有，则创建一个，然后 str1 指向字符串常量池中的对象，如果有，则直接将 str1 指向"abcd""；
* String str2 = new String("abcd");//堆中创建一个新的对象
* String str3 = new String("abcd");//堆中创建一个新的对象
* System.out.println(str1==str2);//false
* System.out.println(str2==str3);//false

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。

记住一点：**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，JDK1.7之前（不包含1.7）的处理方式是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，JDK1.7以及之后的处理方式是在常量池中记录此字符串的引用，并返回该引用。

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

**字符串拼接:**

String str1 = "str";

String str2 = "ing";

String str3 = "str" + "ing";//常量池中的对象

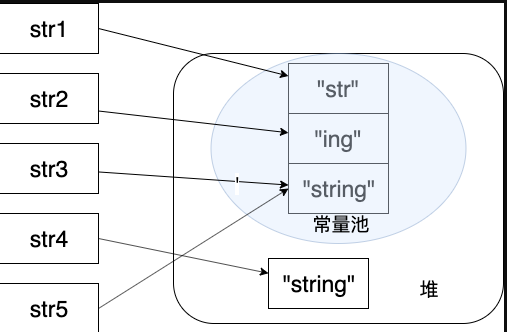
String str4 = str1 + str2; //在堆上创建的新的对象

String str5 = "string";//常量池中的对象

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

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

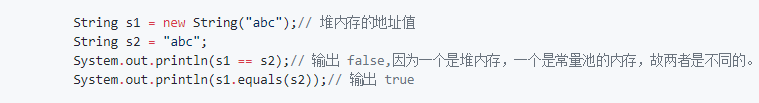
System.out.println(str4 == str5);//false



### 2.String s1=new String("abc");这句话创建了几个字符串对象

将创建 1 或 2 个字符串。如果池中已存在字符串常量“abc”，则只会在堆空间创建一个字符串常量“abc”。如果池中没有字符串常量“abc”，那么它将首先在池中创建，然后在堆空间中创建，因此将创建总共 2 个字符串对象。

**验证：**



**结果：**false true

### 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；

前面 4 种包装类默认创建了数值[-128，127] 的相应类型的缓存数据，Character创建了数值在[0,127]范围的缓存数据，Boolean 直接返回True Or False。如果超出对应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

为啥把缓存设置为[-128，127]区间？性能和资源之间的权衡

public static Boolean valueOf(boolean b) {

return (b ? TRUE : FALSE);

}

private static class CharacterCache {

private CharacterCache(){}

static final Character cache[] = new Character[127 + 1];

static {

for (int i = 0; i < cache.length; i++)

cache[i] = new Character((char)i);

}

}

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

Integer 缓存源代码：

/\*\*

\*此方法将始终缓存-128 到 127（包括端点）范围内的值，并可以缓存此范围之外的其他值。

\*/

public static Integer valueOf(int i) {

if (i >= IntegerCache.low && i <= IntegerCache.high)

return IntegerCache.cache[i + (-IntegerCache.low)];

return new Integer(i);

}

**应用场景：**

1. 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。
2. 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

解释：

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。