

Java 基础 (8) : 深入解析常量池与装拆箱机制

引言

本文将介绍常量池与装箱拆箱机制，之所以将两者合在一起介绍，是因为网上不少文章在谈到常量池时，将包装类的缓存机制，java 常量池，不加区别地混在一起讨论，更有甚者完全将这两者视为一个整体，给初学者带来不少困扰。同时，也因为包装类的缓存与字符串常量池的思想是一样的，很容易混淆，但是实现方式是不一样的。

一、常量池

在介绍常量池前，先来介绍一下常量、字面常量、符号常量的定义。

常量可分为**字面常量**（也称为直接常量）和**符号常量**。

字面常量：是指在程序中无需预先定义就可使用的数字、字符、boolean 值、字符串等。简单的说，就是确定值的本身。如 10，2L，2.3f，3.5，“hello”，'a'，true、false、null 等等。

符号常量：是指在程序中用标识符预先定义的，其值在程序中不可改变的量。如 `final int a = 5;`

常量池

常量池引入的目的是为了避免频繁的创建和销毁对象而影响系统性能，其实现了对象的共享。这是一种享元模式的实现。

二、java 常量池

Java 的常量池可以细分为以下三类：

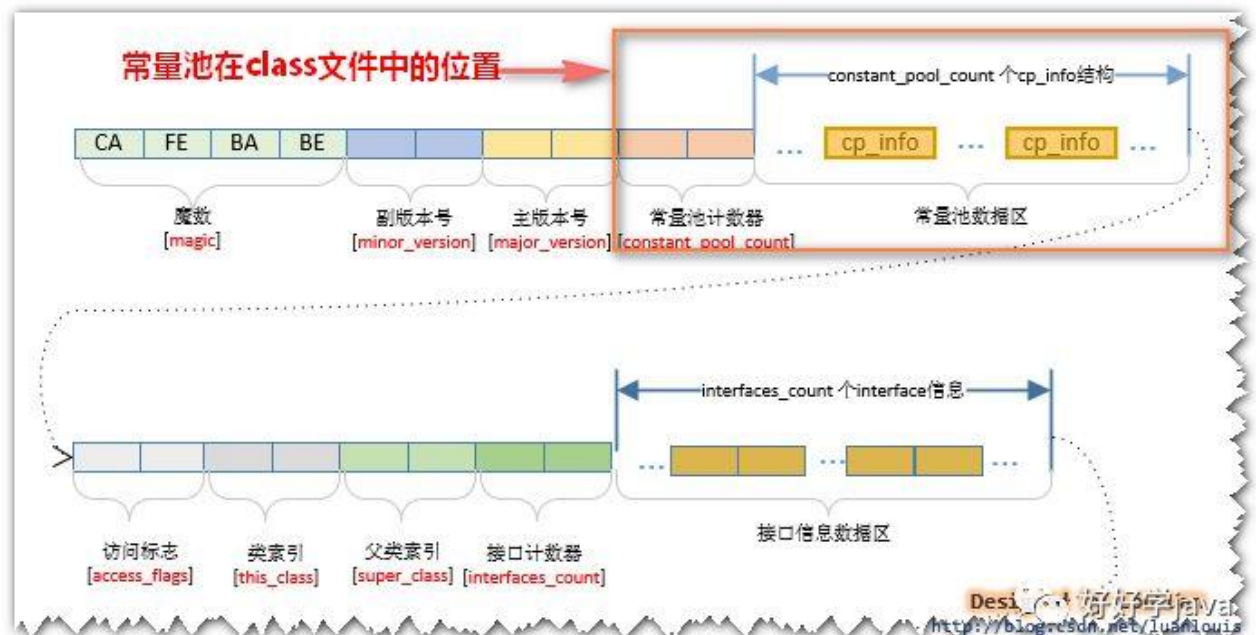
- 静态常量池（编译阶段）；
- 运行常量池（又称动态常量池，运行阶段）
- 字符串常量池（全局的常量池）

1. class 文件常量池

class 文件常量池，也被称为静态常量池，它是.class 文件所包含的一项信息。用于存放

编译器生成的各种字面量(Literal)和符号引用(Symbolic References)。

常量池在.class 文件的位置：

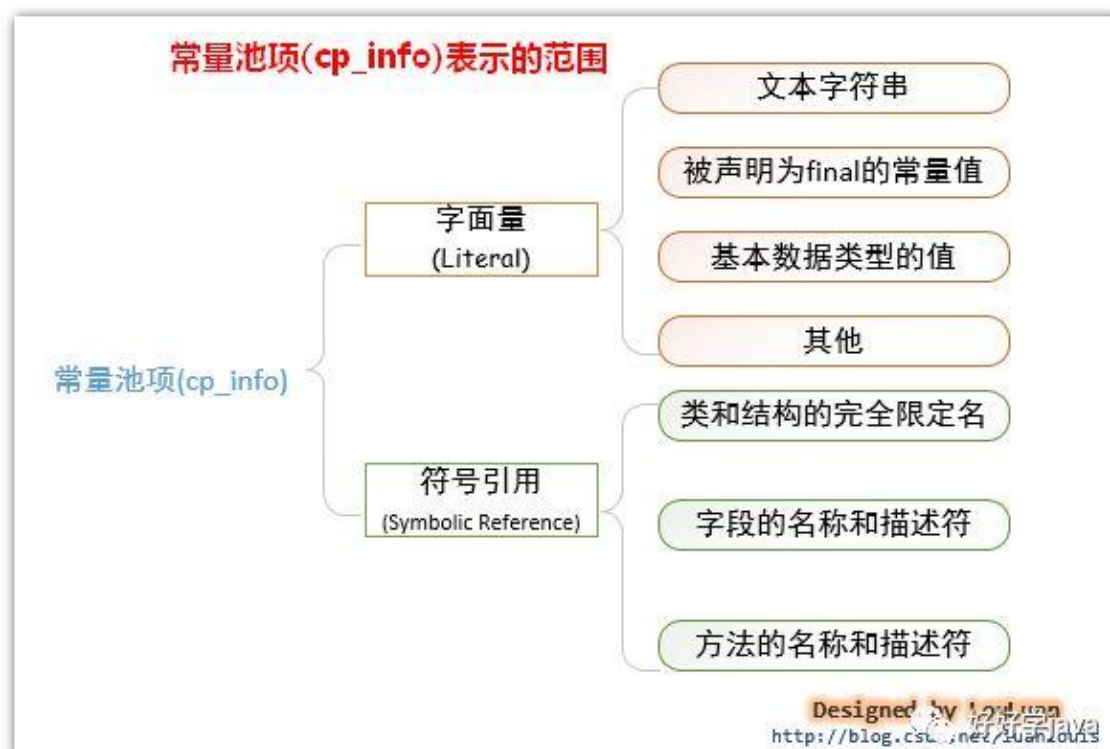


字面量： 就是上面所说的字面常量。

符号引用： 是一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可(它与直接引用区分一下，**直接引用**一般是指向方法区的本地指针，相对偏移量或是一个能间接定位到目标的句柄)。**符号引用可以看作是一个虚拟地址，只有在 JVM 加载完类，确认了字面量的地址，才会将符号引用换成直接引用。**一般包括下面三类常量：

- 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

常量池的信息



2. 运行时常量池

运行时常量池，又称为**动态常量池**，是 JVM 在完成加载类之后将 class 文件中常量池载入到内存中，并保存在方法区中。也就是说，运行时常量池中的常量，基本来源于各个 class 文件中的常量池。**运行时常量池相对于 Class 文件常量池的另外一个重要特征是具备动态性**，Java 语言并不要求常量一定只有编译期才能产生，也就是并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用比较多的就是 String 类的 intern()方法。

jvm 在执行某个类的时候，必须经过加载、连接、初始化，而连接又包括验证、准备、解析三个阶段。而当类加载到内存中后，jvm 就会将 class 常量池中的内容存放到运行时常量池中，也就是说，每个 class 对应运行时常量池中的一个独立空间，每个 class 文件存放的位置互不干扰。而在解析阶段，就会将符号引用替换成对应的直接引用。

不过，**String 类型**的字面常量要注意：并不是直接在堆上分配空间来创建对象的，JVM 为 String 字符串额外维护了一个常量池**字符串常量池**，所以遇到字符串常量是要先去字符串池中寻找是否有重复，如果有，则返回对应的引用。否则，才创建并添加到字符串常量池中。换句话说，**对于 String 类型的字面常量，必须要在字符串常量池中维护一个全局的引用。**

3. 字符串常量池 (string pool 也有叫做 string literal pool)

字符串常量池存储的就是字符串的字面常量。详细一点，字符串常量池里的内容是在类加载完成，经过验证，准备阶段之后在堆中生成字符串对象实例，然后将该字符串对象实例的引用值存到 string pool 中（记住：string pool 中存的是引用值而不是具体的实例对象，具体的实例对象是在堆中开辟的一块空间存放的）。

在 HotSpot VM 里实现的 string pool 功能的是一个 StringTable 类，它是一个哈希表，里面存的是驻留字符串(也就是我们常说的用双引号括起来的)的引用（而不是驻留字符串实例本身），也就是说在堆中的某些字符串实例被这个 StringTable 引用之后就等同被赋予了“驻留字符串”的身份。这个 StringTable 在每个 HotSpot VM 的实例只有一份，被所有的类共享。

运行时常量池与字符串常量池的区别

字符串常量池是位于运行时常量池中的。

网上有不少文章是将字符串常量池作为运行时常量池同等来说，我一开始也以为这两者就是同一个东西，其实不然。**运行时常量池与字符串常量池在 HotSpot 的 JDK1.6 以前，都是放在方法区的，JDK1.7 就将字符串常量池移到了堆外内存中去。**运行时常量池为每一个 Class 文件的常量池提供一个运行时的内存空间；而字符串常量池则为所有 Class 文件的 String 类型的字面常量维护一个公共的常量池，也就是 Class 文件的常量池加载进运行时常量池后，其 String 字面常量的引用指向要与字符串常量池的维护的要一致。

我们来几个例子理解一下常量池

@ Example 1 简单的例子

```
public class Test_6 {  
    public static void main(String[] args) {  
        String str = "Hello World!";  
    }  
}
```

我们使用 `javap -v MyTest.class` 查看 class 文件的字节码，经 `javap` 处理可以输出我们能看懂的信息。如下图：

```
public class Test_2018_1_14.Test_6  
  minor version: 0  
  major version: 52  
  flags: ACC_PUBLIC, ACC_SUPER  
Constant pool:  
  #1 = Class                #2          // Test_2018_1_14/Test_6  
  #2 = Utf8                 Test_2018_1_14/Test_6  
  #3 = Class                #4          // java/lang/Object  
  #4 = Utf8                 java/lang/Object  
  #5 = Utf8                 <init>  
  #6 = Utf8                 ()V  
  #7 = Utf8                 Code  
  #8 = Methodref            #3.#9      // java/lang/Object.<init>():()V  
  #9 = NameAndType          #5:#6      // "<init>":()V  
  #10 = Utf8                LineNumberTable  
  #11 = Utf8                LocalVariableTable  
  #12 = Utf8                this  
  #13 = Utf8                LTest_2018_1_14/Test_6;  
  #14 = Utf8                main  
  #15 = Utf8                ([Ljava/lang/String;)V  
  #16 = String              #17          // Hello World!  
  #17 = Utf8                Hello World!  
  #18 = Utf8                args  
  #19 = Utf8                [Ljava/lang/String;  
  #20 = Utf8                str  
  #21 = Utf8                Ljava/lang/String;  
  #22 = Utf8                SourceFile  
  #23 = Utf8                Test_6.java
```

 好好学java

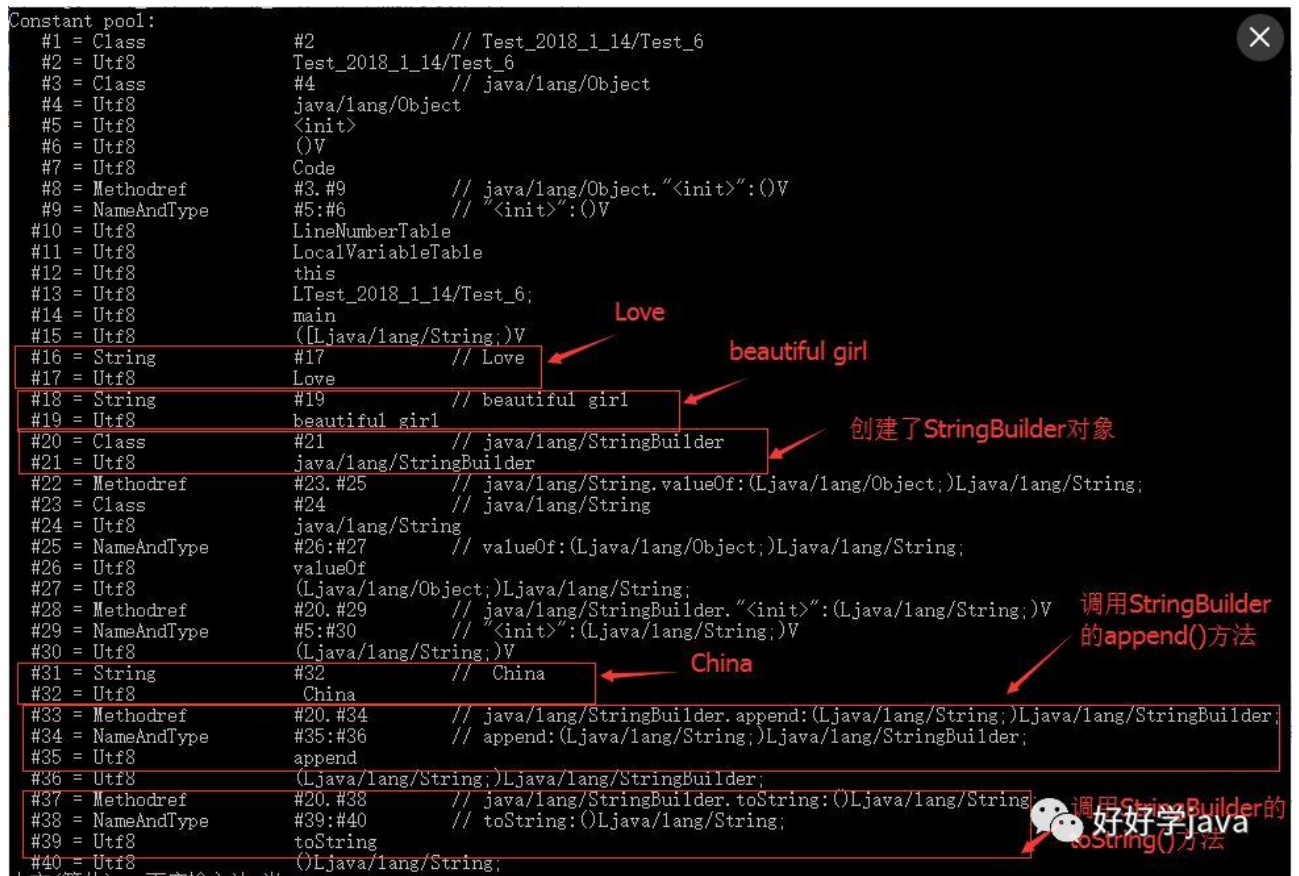
class 文件的索引#16 位置（第 16 个常量池项）存储的是一个描述了字符串字面常量信息（类型，以及内容索引）的数据结构体，这个结构体被称为 `CONSTANT_String_info`。这个结构体并没有存储字符串的内容，而是存储了一个指向字符串内容的索引--#17，即第 17 项存储的是 `Hello World` 的二进制码。

我们再来看一个比较复杂的例子

@ Example 2 String 的+运算例子

```
public class Test_6 {  
    public static void main(String[] args) {  
        String str_aa = "Love";  
        String str_bb = "beautiful" + " girl";  
        String str_cc = str_aa+" China";  
    }  
}
```

同样，查看 class 文件的字节码信息：



```
Constant pool:
#1 = Class                #2 // Test_2018_1_14/Test_6
#2 = Utf8                Test_2018_1_14/Test_6
#3 = Class                #4 // java/lang/Object
#4 = Utf8                java/lang/Object
#5 = Utf8                <init>
#6 = Utf8                ()V
#7 = Utf8                Code
#8 = Methodref            #3.#9 // java/lang/Object.<init>:()V
#9 = NameAndType          #5:#6 // <init>:()V
#10 = Utf8               LineNumberTable
#11 = Utf8               LocalVariableTable
#12 = Utf8               this
#13 = Utf8               LTest_2018_1_14/Test_6;
#14 = Utf8               main
#15 = Utf8               ([Ljava/lang/String;)V
#16 = String              #17 // Love
#17 = Utf8               Love
#18 = String              #19 // beautiful girl
#19 = Utf8               beautiful girl
#20 = Class                #21 // java/lang/StringBuilder
#21 = Utf8               java/lang/StringBuilder
#22 = Methodref            #23.#25 // java/lang/String.valueOf:(Ljava/lang/Object;)Ljava/lang/String;
#23 = Class                #24 // java/lang/String
#24 = Utf8               java/lang/String
#25 = NameAndType          #26:#27 // valueOf:(Ljava/lang/Object;)Ljava/lang/String;
#26 = Utf8               valueOf
#27 = Utf8               (Ljava/lang/Object;)Ljava/lang/String;
#28 = Methodref            #20.#29 // java/lang/StringBuilder.<init>:(Ljava/lang/String;)V
#29 = NameAndType          #5:#30 // <init>:(Ljava/lang/String;)V
#30 = Utf8               (Ljava/lang/String;)V
#31 = String              #32 // China
#32 = Utf8               China
#33 = Methodref            #20.#34 // java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
#34 = NameAndType          #35:#36 // append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
#35 = Utf8               append
#36 = Utf8               (Ljava/lang/String;)Ljava/lang/StringBuilder;
#37 = Methodref            #20.#38 // java/lang/StringBuilder.toString:()Ljava/lang/String;
#38 = NameAndType          #39:#40 // toString:()Ljava/lang/String;
#39 = Utf8               toString
#40 = Utf8               ()Ljava/lang/String;
```

class 文件的常量池保存了 Love、beautiful girl、China，但却没有 Love China。为什么 str_bb 与 str_cc 都是通过 + 链接得到的，为什么 str_cc 的值没有出现在常量池中，而 str_bb 的值却出现了。

这是因为 str_bb 的值是由两个常量计算得到的，这种只有常量的表达式计算在编译期间由编译器计算得到的，要记住，**能由编译器完成的计算，就不会拖到运行期间来计算。**

而 str_cc 的计算中包含了变量 str_aa，**涉及到变量的表达式计算都是在运行期间计算的**，因为变量是无法在编译期间确定它的值，特别是多线程下，同时得到结果是 CPU 动态分配空间存储的，也就是说地址也无法确定。我们再去细看，就会发现常量池中的包含了 StringBuilder 及其方法的描述信息，其实，这个 StringBuilder 是为了计算 str_aa+"China"表达式，先调用 append()方法，添加两个字符串，在调用 toString()方法，返回结果。也就是说，**在运行期间，String 字符串通过 + 来链接的表达式计算都是通过创建 StringBuilder 来完成的。**

@ Example 3 String 新建对象例子

下面的例子，str_bb 的值是直接通过 new 新建一个对象，观察静态常量池。

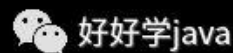
```
public class MyTest {
    public static void main(String[] args) {
        String str_bb = new String("Hello");
    }
}
```

查看对应 class 文件的字节码信息：


```

Constant pool:
#1 = Class                #2          // Test_2018_1_14/Test_6
#2 = Utf8                 Test_2018_1_14/Test_6
#3 = Class                #4          // java/lang/Object
#4 = Utf8                 java/lang/Object
#5 = Utf8                 <init>
#6 = Utf8                 ()V
#7 = Utf8                 Code
#8 = Methodref            #3.#9      // java/lang/Object.<init>():()V
#9 = NameAndType          #5:#6      // "<init>":()V
#10 = Utf8                LineNumberTable
#11 = Utf8                LocalVariableTable
#12 = Utf8                this
#13 = Utf8                LTest_2018_1_14/Test_6;
#14 = Utf8                main
#15 = Utf8                ([Ljava/lang/String;)V
#16 = Class                #17        // java/lang/String
#17 = Utf8                java/lang/String
#18 = String              #19        // Hello
#19 = Utf8                Hello
#20 = Methodref            #16.#21    // java/lang/String.<init>:(Ljava/lang/String;)V
#21 = NameAndType          #5:#22    // "<init>":(Ljava/lang/String;)V
#22 = Utf8                (Ljava/lang/String;)V
#23 = Utf8                args
#24 = Utf8                [Ljava/lang/String;
#25 = Utf8                str_bb
#26 = Utf8                Ljava/lang/String;
#27 = Utf8                SourceFile
#28 = Utf8                Test_6.java

```



通过 new 新建对象的操作是在运行期间才完成的，为什么这里仍旧在 class 文件的常量池中出现呢？这是因为"Hello"本身就是一个字面常量，这是很容易让人忽略的。有双引号包裹的都是字面常量。同时，new 创建一个 String 字符串对象，确实是在运行时完成的，但这个对象将不同于字符串常量池中所维护的常量。

三、自动装箱拆箱机制与缓存机制

1、自动装箱拆箱机制介绍

装箱： 可以自动将基本类型直接转换成对应的包装类型。

拆箱： 自动将包装类型转换成对应的基本类型值；

//普通的创建对象方式

```
Integer a = new Integer(5);
```

//装箱

```
Integer b = 5;
```

//拆箱

```
int c = b+5;
```

2. 自动装箱拆箱的原理

装箱拆箱究竟是怎么实现，感觉有点神奇，居然可以使基本类型与包装类型快速转换。我们再稍微简化上面的例子：

```
public class Test_6 {
```

```

public static void main(String[] args) {
    //装箱
    Integer b = 5;
    //拆箱
    int c = b+5;
}
}

```

依旧使用 `javap -v Test_6.class` 查看这个类的 class 文件的字节码信息，如下图：

```

Constant pool:
 #1 = Class                #2          // Test_2018_1_14/Test_6
 #2 = Utf8                  Test_2018_1_14/Test_6
 #3 = Class                #4          // java/lang/Object
 #4 = Utf8                  java/lang/Object
 #5 = Utf8                  <init>
 #6 = Utf8                  ()V
 #7 = Utf8                  Code
 #8 = Methodref             #3.#9      // java/lang/Object.<init>:()V
 #9 = NameAndType           #5:#6      // "<init>":()V
#10 = Utf8                  LineNumberTable
#11 = Utf8                  LocalVariableTable
#12 = Utf8                  this
#13 = Utf8                  LTest_2018_1_14/Test_6;
#14 = Utf8                  main
#15 = Utf8                  ([Ljava/lang/String;)V
#16 = Methodref             #17.#19     // java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
#17 = Class                 #18        // java/lang/Integer
#18 = Utf8                  java/lang/Integer
#19 = NameAndType           #20:#21     // valueOf:(I)Ljava/lang/Integer;
#20 = Utf8                  valueOf
#21 = Utf8                  (I)Ljava/lang/Integer;
#22 = Methodref             #17.#23     // java/lang/Integer.intValue:()I
#23 = NameAndType           #24:#25     // intValue:()I
#24 = Utf8                  intValue
#25 = Utf8                  ()I
#26 = Utf8                  args
#27 = Utf8                  [Ljava/lang/String;
#28 = Utf8                  b
#29 = Utf8                  Ljava/lang/Integer;
#30 = Utf8                  c
#31 = Utf8                  I
#32 = Utf8                  SourceFile
#33 = Utf8                  Test_6.java

```

调用Integer的
valueOf()方法

调用Integer的
intValue()方法

好好学java

可以从 class 的字节码发现 静态常量池中 由 `Integer.valueOf()` 和 `Integer.intValue()` 这两个方法的描述。这就有点奇怪，例子中的代码中并没有调用这两个方法，为什么编译后会出现呢？

感觉还是不够清晰，我们换另一种反编译工具来反编译一下，这次我们反编译回 java 代码，使用命令 `jad Test_6.class`，得到的反编译代码如下：

```

public class Test_6{
    public static void main(String args[]){
        Integer b = Integer.valueOf(5);
        int c = b.intValue() + 5;
    }
}

```

这回就非常直观明了了。所谓装箱拆箱并没有多厉害，还是要通过调用 `Integer.valueOf()`（装箱）和 `Integer.intValue()`（拆箱）来完成的。也就是说，**自动装箱拆箱机制是一种语**

法简写，为了方便程序员，省去了手动装箱拆箱的麻烦，变成了自动装箱拆箱。

判别是装箱还是拆箱

在下面的两个例子中，可能会让你很迷惑：不知道到底使用了装箱，还是使用了拆箱。

```
Integer x = 1;
Integer y = 2;
Integer z = x+y;
```

这种情况其实只要仔细想一下便可以知道：这是 先拆箱再装箱。因为 Integer 类型是引用类型，所以不能参与加法运算，必须拆箱成基本类型来求和，在装箱成 Integer。如果改造上面的例子，把 Integer 变成 Short，则正确代码如下：

```
Short a = 5;
Short b = 6;
Short c = (short) (a+b);
```

3. 包装类的缓存机制

我们先来看一个例子

```
public class MyTest {
    public static void main(String[] args) {
        Integer a = 5;
        Integer b = 5;
        Integer c = 129;
        Integer d = 129;
        System.out.println("a==b "+ (a == b));
        System.out.println("c==d "+ (c == d));
    }
}
```

运行结果：

```
a == b  true
c == d  false
```

咦，为什么是 a 和 b 所指向的是一个对象呢？难道 JVM 在类加载时也为包装类型维护了一个常量池？如果是这样，为什么变量 c、d 的地址不一样。事实上，JVM 确实没有为包装类维护一个常量池。变量 a、b、c、d 是由装箱得到的，根据前面所说的，装箱其实是编译器自动添加了 Integer.valueOf() 方法。秘密应该就在这个方法内，那么我们看一下 Integer.valueOf() 的源代码吧，如下：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```



```
}
```

代码很简单,判断装箱所使用的基本类型值是否在 [IntegerCache.low, IntegerCache.high] 的范围内,如果在,返回 IntegerCache.cache 数组中对应下标的元素。否则,才新建一个对象。我们继续深入查看 IntegerCache 的源码,如下:

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];
    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        //获取上限值
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;
        //创建数组
        cache = new Integer[(high - low) + 1];
        int j = low;
        //填充数组
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);
        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }
    private IntegerCache() {}
}
```

从源码中,可以知道,IntegerCache.cache 是一个 final 的 Integer 数组,这个数组存储的 Integer 对象元素的值范围是[-128, 127]。而且这个数组的初始化代码是包裹在 static

代码块中，也就是说 IntegerCache.cache 数组的初始化是在类加载时完成的。

再看回上面的例子，变量 a 和 b 的使用的基本类型值为 5，没有超出[-128,127]的范围，所以就使用缓存数组中的元素，所以 a、b 的地址是一样的。而 c、d 使用的基本类型值为 129，超出缓存范围，所以都是各自在堆上创建一个对，地址自然就不一样了。

- 包装类与 String 类很相似，都是非可变类，即一经创建后，便不可以修改。正因为这种特性，两者的对象实例在多线程下是安全的，不用担心异步修改的情况，这为他们实现共享提供了很好的保证，只需创建一个对象共享便可。
- 包装类的共享实现并不是由 JVM 来维护一个常量池，而是使用了缓存机制(数组)，而且这个缓存是在类加载时完成初始化，并且不可再修改。
- 包装类的数组缓存范围是有限，只缓存基本类型值在一个字节范围内，也就是说 -128 ~ 127。(Character 的范围是 0~127)
- 目前并不是所有包装类都提供缓存机制，只有 Byte、Character、Short、Integer 4 个包装类提供，Long、Float、Double 不提供。

1) 如下代码：

```
class Super {
    public Integer getLenght() { return new Integer(4); }
}

public class Sub extends Super {
    public Long getLenght() { return new Long(5); }
    public static void main(String[] args) {
        Super sooper = new Super();
        Sub sub = new Sub();
        System.out.println(sooper.getLenght().toString() + "," +
sub.getLenght().toString() );
    }
}
```

输出是什么？ A

- A. 4,4
- B. 4,5
- C. 5,4
- D. 5,5
- E. 编译失败。-----重定时不能改变返回类型

2) 说出数据连接池的工作机制是什么？

J2EE 服务器启动时会建立一定数量的池连接，并一直维持不少于此数目的池连接。客户端程序需要连接时，池驱动程序会返回一个未使用的池连接并将其标记为忙。如果当前没有空闲连接，池驱动程序就新建一定数量的连接，新建连接的数量由配置参数决定。当使用的池连接调用完成后，池驱动程序将此连接标记为空闲，其他调用就可以使用这个连接。