**深入研究String**

String是一个非可变类（immutable class），其实现采用Copy On Write技术。简单说来，非可变类的实例是不能被修改的，每个实例中包含的信息都必须在该实例创建的时候就提供出来，并且在对象的整个生存周期内固定不变。非可变类有着自身的优势，如状态单一，对象简单，便于维护；其次，该类的对象本质上是线程安全的，不要求同步。此外用户可以共享非可变对象，甚至可以共享它们的内部信息。

**String对象的创建**

创建一个String 对象，主要就有以下两种方式：

第一种 String str1 = new String("abc");

第一种  String str2 = "abc";

对于第一种，JVM会在heap中创建一个String对象，然后将该对象的引用返回给用户。

对于第二种，JVM首先会在内部维护的strings pool中通过String的 equals 方法查找是对象池中是否存放有该String对象，如果有，则返回已有的String对象给用户，而不会在heap中重新创建一个新的String对象；如果对象池中没有该String对象，JVM则在heap中创建新的String对象，将其引用返回给用户，同时将该引用添加至strings pool中。

注意：使用第一种方法创建对象时，JVM是不会主动把该对象放到strings pool里面的，除非程序调用 String的intern方法。看下面的例子：

//JVM 在堆上创建一个String对象

String str1 = **new** String("abc");

    //      jvm 在strings pool中找不到值为“abc”的字符串，因此

    //      在堆上创建一个String对象，并将该对象的引用加入至strings pool中

    //      此时堆上有两个String对象

    String str2 = "abc";

**if**(str1 == str2)

    {

        System.out.println("str1 == str2");

    }

**else**

    {

        System.out.println("str1 != str2");

    }

    //打印结果是 str1 != str2,因为它们是堆上两个不同的对象

    String str3 = "abc";

    //      此时，jvm发现strings pool中已有“abc”对象了，因为“abc”equels “abc”

    //      因此直接返回str2指向的对象给str3，也就是说str2和str3是指向同一个对象的引用

**if**(str2 == str3)

    {

        System.out.println("str2 == str3");

    }

**else**

    {

        System.out.println("str2 != str3");

    }

//      打印结果为 str2 == str3

再看下面的例子：

String str1 = **new** String("abc"); //JVM 在堆上创建一个String对象

str1 = str1.intern();

// 程序显式将str1放到strings pool中，intern运行过程是这样的：首先查看strings pool

// 有没“abc”对象的引用，没有，则在堆中新建一个对象，然后将新对象的引用加入至

// strings pool中。执行完该语句后，str1原来指向的String对象已经成为垃圾对象了。

// 此时，JVM发现strings pool中已有“abc”对象了，因为“abc”equals “abc”

// 因此直接返回str1指向的对象给str2，也就是说str2和str1引用着同一个对象，

// 此时，堆上的有效对象只有一个。

String str2 = "abc";

**if**(str1 == str2)

{

    System.out.println("str1 == str2");

}

**else**

{

    System.out.println("str1 != str2");

}

//打印结果是 str1 == str2

为什么JVM可以这样处理String对象呢？就是因为String的非可变性。既然所引用的对象一旦创建就永不更改，那么多个引用共用一个对象时互不影响。

**JVM对String常量的处理和优化**

**JVM对String的处理**

**class** Test

{

**public** **static** **void** main(String[] args)

    {

        /\*

         \* 1.字面上的 "Hi" 字符串将被自动 intern 到虚拟机的字符串池中.

         \* 2.不是字面上的String对象，通过调用intern()方法,被intern到虚拟机的字符串池中.

         \*

         \* 虚拟机的字符串池由类 String 私有地维护。

         \*/

        String a = "Hi";

        String b = "Hi";

        String c = **new** String("Hi");

        System.out.println(a == b);// true

        System.out.println(a == c);// false

        c = c.intern();

        System.out.println(a == c);// true

        /\*

         \* 创建了两个对象

         \* 1.字面"AA" ——被intern到虚拟机字符串池中了。

         \* 2.new 操作符创建的 对象

         \*

         \* 由AA != AA.intern()可以看出：String AA = new String("AA");创建了两个对象。

         \*/

        String AA = **new** String("AA");

        System.out.println(AA == "AA");// false

        System.out.println(AA == AA.intern());// false

        System.out.println("AA" == AA.intern());// true

        /\*

         \* JAVA 规范中的例子：

         \*/

        String hello = "Hello", lo = "lo";

        System.out.println((hello == "Hello") + " "); //true

        System.out.println((hello == ("Hel" + "lo")) + " "); //true

        System.out.println((hello == ("Hel" + lo)) + " "); //false--lo为非literal的

        System.out.println(hello == ("Hel" + lo).intern());//true

    }

}

**编译器对String常量表达式的优化**

**问题代码**

String a = "ab";

String b = "a" + "b";

System.out.println((a == b));

打印结果会是什么？一般答案会是以下几种：

（1）true

"a" + "b"的结果就是"ab"，这样a,b都是"ab"了，内容一样所以"相等"，结果true

一般java新人如是答。

（2）false

"a" + "b"会生成新的对象"ab"，但是这个对象和String a = "ab";不同，(a == b)是比较对象引用，因此不相等，结果false 对java的String有一定了解的通常这样回答。

（3）true

String a = "ab";创建了新的对象"ab"； String b = "a" + "b";没有创建新的对象，而是从JVM字符串常量池中获取之前已经存在的"ab"对象。因此a,b具有对同一个string对象的引用，两个引用相等，结果true 能回答出这个答案的，基本已经是高手了，对java中的string机制比较了解。很遗憾,这个答案,是错误的，或者说，压根没有这么回事.

（4）.true

String b = "a" + "b";编译器将这个"a" + "b"作为常量表达式，在编译时进行优化，直接取结果"ab"，这样这个问题退化

String a = "ab";

String b = "ab";

System.out.println((a == b));

String b = "ab";没有创建新的对象，而是从JVM字符串常量池中获取之前已经存在的"ab"对象。因此a,b具有对同一个string对象的引用，两个引用相等，结果true

**编译器优化证明过程**

这里有一个疑问就是String不是基本类型，像 int secondsOfDay = 24 \* 60 \* 60; 这样的表达式是常量表达式,编译器在编译时直接计算容易理解，而"a" + "b" 这样的表达式,string是对象不是基本类型,编译器会把它当成常量表达式来优化吗?

下面简单证明我的推断，首先编译这个类:

public class Test {

private String a = "aa";

}

复制class文件备用,然后修改为

public class Test {

private String a = "a" + "a";

}

再次编译,用ue之类的文本编辑器打开,察看二进制内容,可以发现,两个class文件完全一致,连一个字节都不差.

真相大白了.根本不存在运行期的处理String b = "a" + "b";这样的代码的问题,编译时就直接优化掉了。

**扩展应用**

下面进一步探讨，什么样的String + 表达式会被编译器当成常量表达式？

（1）       String + String被正式是ok的

（2）       String + int    被正式也是ok的

        String a = "a1";

        String b = "a" + 1;

        System.*out*.println((a == b)); //result = true

（3）       String + boolean被正式也是ok的

        String a = "atrue";

        String b = "a" + **true**;

        System.*out*.println((a == b)); //result = true

（4）       （3）    String + double被正式也是ok的

        String a = "a3.4";

        String b = "a" + 3.4;

        System.*out*.println((a == b)); //result = true

可见编译器对string + 基本类型是当成常量表达式直接求值来优化的。

（5）     再注意看这里的String都是"\*\*"这样的，我们换成变量来试试：

        String a = "ab";

        String bb = "b";

        String b = "a" + bb;

        System.*out*.println((a == b)); //result = false

这个好理解，"a" + bb中的bb是变量，不能进行优化。

（6）     再修改一下,把bb作为常量变量：

        String a = "ab";

**final** String bb = "b";

        String b = "a" + bb;

        System.*out*.println((a == b)); //result = true

竟然又是true，编译器的优化好厉害啊！

（7）     考虑下面这种情况：

**private** **static** String getBB()

{

**return**"b";

}

**public** **static** **void** main(String[] args)

{

    String a = "ab";

**final** String bb = getBB();

    String b = "a" + bb;

    System.out.println((a == b)); //result = false

}

看来java(包括编译器和jvm)对String的优化，真的是到了极点了，String这个所谓的"对象"，完全不可以看成一般的对象，java对String的处理近乎于基本类型，最大限度的优化了几乎能优化的地方。

**String串接（Concatenation）**

滥用String的串接操作符是会影响程序的性能的。归根结底就是String类的非可变性。既然String对象都是非可变的，但是串接操作明显是要增长字符串的，也就是要改变String的内部状态，两者出现了矛盾。所以要维护String的非可变性，只好在串接完成后新建一个String 对象来表示新产生的字符串了。也就是说，每一次执行串接操作都会导致新对象的产生，如果串接操作执行很频繁，就会导致大量对象的创建，性能问题也就随之而来了。

为了解决这个问题，JDK为String类提供了一个可变的配套类，StringBuffer。

使用StringBuffer对象，由于该类是可变的，串接时仅仅时改变了内部数据结构，而不会创建新的对象，因此性能上有很大的提高。

针对单线程，JDK5.0还提供了StringBuilder类，在单线程环境下，由于不用考虑同步问题，使用该类使性能得到进一步的提高。

**String的长度** 查看String的源代码我们可以得知类String中是使用int域 count 来记录对象字符的数量，因此，我们可以推测最长的长度为 2^32，也就是4G。

不过，我们在编写源代码的时候，如果使用 Sting str = "aaaa";的形式定义一个字符串，那么双引号里面的ASCII字符最多只能有 65534 个。原因是在class文件的规范中， CONSTANT\_Utf8\_info表中使用一个16位的无符号整数来记录字符串的长度的，最多能表示 65536个字节，而java class 文件是使用一种变体UTF-8格式来存放字符的，null值使用两个字节来表示，因此只剩下 65536－ 2 ＝ 65534个字节。也正是变体UTF-8的原因，如果字符串中含有中文等非ASCII字符，那么双引号中字符的数量会更少（一个中文字符占用三个字节）。如果超出这个数量，在编译的时候编译器会报错。

**String参数传递问题**

**public** **class** StringTest

{

**static** **void** func(String s)

    {

        s += "tail";

    }

**static** **void** test()

    {

        String a = "abc";

        func(a);

        System.out.println(a);

    }

**public** **static** **void** main(String[] args)

    {

        test();

    }

}

程序运行结果：

abc