String的常量池与intern方法

# String对象的不变性

## final关键字

String类利用final修饰，表示该类不可被继承；String类中有个value属性(字符char数组)，也被final修饰，表示该变量不可被修改，String 使用 **private final char value[]**来实现字符串存储。

对应源码：

**/\*\* The value is used for character storage. \*/**

**private final char value[];**

所以String对象创建之后就不能再修改此对象中存储的字符串内容，所以说String本质是**字符数组char[]**，且其类型是不可变的！

如果实现字符串的拼接，建议使用StringBuilder和StringBuffer。

## 对String对象的增减、替换等操作，都是生成新的String对象

示例：replace替换某个字符char的源码：

/\*\*

\* Returns a string resulting from replacing all occurrences of

\* {@code oldChar} in this string with {@code newChar}.

\* <p>

\* If the character {@code oldChar} does not occur in the

\* character sequence represented by this {@code String} object,

\* then a reference to this {@code String} object is returned.

\* Otherwise, a {@code String} object is returned that

\* represents a character sequence identical to the character sequence

\* represented by this {@code String} object, except that every

\* occurrence of {@code oldChar} is replaced by an occurrence

\* of {@code newChar}.

\* <p>

\* Examples:

\* <blockquote><pre>

\* "mesquite in your cellar".replace('e', 'o')

\* returns "mosquito in your collar"

\* "the war of baronets".replace('r', 'y')

\* returns "the way of bayonets"

\* "sparring with a purple porpoise".replace('p', 't')

\* returns "starring with a turtle tortoise"

\* "JonL".replace('q', 'x') returns "JonL" (no change)

\* </pre></blockquote>

\*

\* @param oldChar the old character.

\* @param newChar the new character.

\* @return a string derived from this string by replacing every

\* occurrence of {@code oldChar} with {@code newChar}.

\*/

public String replace(char oldChar, char newChar) {

if (oldChar != newChar) {

int len = value.length;

int i = -1;

char[] val = value; /\* avoid getfield opcode \*/

while (++i < len) {

if (val[i] == oldChar) {

break;

}

}

if (i < len) {

char buf[] = new char[len];

for (int j = 0; j < i; j++) {

buf[j] = val[j];

}

while (i < len) {

char c = val[i];

buf[i] = (c == oldChar) ? newChar : c;

i++;

}

**return new String(buf, true);//生成一个新的字符串**

}

}

return this;

}

替换字符串：

/\*\*

\* Replaces each substring of this string that matches the literal target

\* sequence with the specified literal replacement sequence. The

\* replacement proceeds from the beginning of the string to the end, for

\* example, replacing "aa" with "b" in the string "aaa" will result in

\* "ba" rather than "ab".

\*

\* @param target The sequence of char values to be replaced

\* @param replacement The replacement sequence of char values

\* @return The resulting string

\* @since 1.5

\*/

public String replace(CharSequence target, CharSequence replacement) {

return Pattern.compile(target.toString(), Pattern.LITERAL).matcher(

this).replaceAll(Matcher.quoteReplacement(replacement.toString()));

}

# String对象的创建机制

## 定义方法(常用的主要是两种)

### **直接使用" "双引号创建； (String s1 = "BUPT";)**

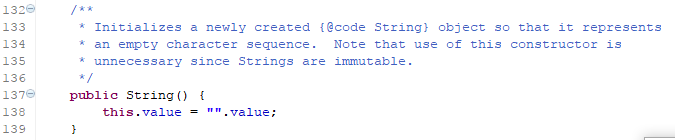
采用**字符串连接符+**创建新的字符串对象; (String s3 = "first" + "second";)

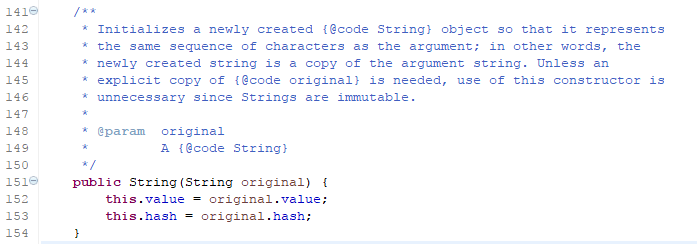
### 使用构造方法创建：new String()或new String("BUPT")

**使用new String("BUPT")创建；(String s2 = new String("BUPT");)**

使用**new String()**创建一个空字符串。

源码：

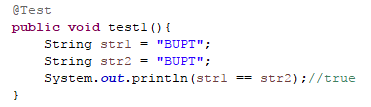
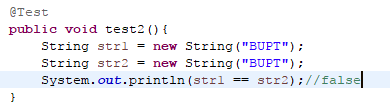




## 常量池

**在Java编译好的字节码文件.class文件中，有个区域被称为Constant Pool，**是一个由数组组成的表，用来存储程序中的各种常量，包括Class、String、Integer等各种Java基本数据类型；String Pool字符串常量池是Constant Pool中存储String常量的区域。

## 用双引号创建字符串机制

### 编译期：**"BUPT"是编译期常量，编译期就能确认它的值，在编译好的.class字节码文件中，" BUPT "就已经存在String Pool中了**；

### 运行期：JVM仅仅会查找维护常量池，拿着" BUPT"在String Pool中查找是否存在内容相同的字符串（**用equals()方法确认**），如果存在，返回String Pool中相应内存单元的引用，赋值给str1（str1即是String Pool中存放" BUPT "内存单元的地址）；如果不存在，则创建一个" BUPT "放在String Pool中，返回引用，赋值给str1；str2同理；

**这个过程实际是调用intern()方法实现的**；在此过程中，JVM绝不会在堆区（heap）创建String对象；所以，上述代码，str1与str2指向String Pool中同一块内存区域，是同一个对象，故返回true。

**利用双引号创建字符串，其实是通过调用intern方法是实现的。**

## String.intern()方法介绍

/\*\*

\* **Returns a canonical representation for the string object**.

\* <p>

\* A pool of strings, initially empty, is maintained privately by the

\* class {@code String}.

\* <p>

**\* When the intern method is invoked, if the pool already contains a**

**\* string equal to this {@code String} object as determined by**

**\* the {@link #equals(Object)} method, then the string from the pool is**

**\* returned. Otherwise, this {@code String} object is added to the**

**\* pool and a reference to this {@code String} object is returned.**

\* <p>

\* It follows that for any two strings {@code s} and {@code t},

\* {@code s.intern() == t.intern()} is {@code true}

\* if and only if {@code s.equals(t)} is {@code true}.

\* <p>

\* All literal strings and string-valued constant expressions are

\* interned. String literals are defined in section 3.10.5 of the

\* <cite>The Java&trade; Language Specification</cite>.

\*

**\* @return a string that has the same contents as this string, but is**

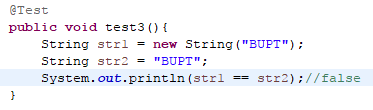
**\* guaranteed to be from a pool of unique strings.**

\*/

**public native String intern();**

## new String()创建对象

相当于**创建两次String对象**，**一次在String Pool中，一次在堆区（heap）中**。



在Java中，使用new关键字创建一个新对象，不管在String Pool中是否有值相同的对象，**总会创建一个新的String对象存储在堆区（heap）中，**然后返回**堆区（heap）中相应内存单元的引用**，赋值给str1，而str2还是指向String Pool中相应内存单元；故str1与str2肯定不是同一个对象，只是存储字符串值相同，故返回false。

## 注意：

在默认的JVM中，Constant Pool中已经存在了一些字符串，如

# 双引号"BUPT"创建与new String("BUPT")的区别

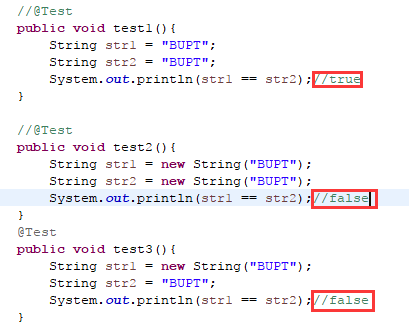
### 创建对象的次数

**"BUPT"仅仅在Constant Pool中创建一次；**new String(**"BUPT"**)创建2次，在Constant Pool和Heap中分别创建一次，当Constant Pool中已经存在，则不创建，但是在Heap中是肯定创建一个新的对象。

### 返回对象地址的内存区域不同

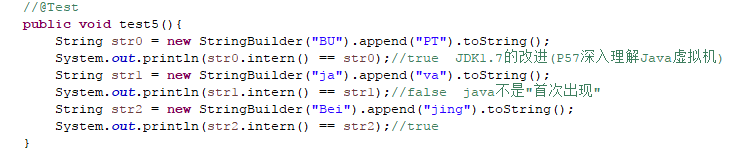
**String s = "BUPT";返回的是Constant Pool中的地址；String s2 =** new String(**"BUPT"**);返回的是Java Heap中的地址。

**在Constant Pool中，只能存在字符串value值相同的唯一的一个对象；而在Heap中可以创建多个value相同的String对象。**



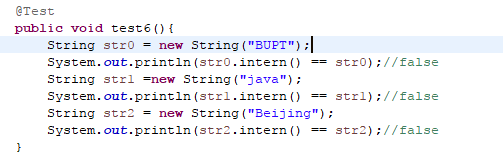
# JDK1.7对String.intern()的改进

运行时常量池存在Java内存模型的永久代(Permanent Generattion,方法区)中。

在JDK1.6及之前，String.intern()方法是复制String实例，而JDK1.7及之后，intern()方法的实现不再复制实例，**只是在常量池中记录首次出现的实例引用**。  


BUPT是首次出现的，因此intern()返回的就是堆中该实例首次出现的地址，故为true；

而java不是首次出现的，所以返回false。Beijing同BUPT。



注意：通过**new StringBuilder().toString()创建对象与new String()创建对象的方式**是不一样的。需要进一步理解。

# Java的内存模型

### 堆Heap

JVM具有一个堆（heap），堆是**运行时数据区**，为**所有类实例与数组动态分配内存（堆的优势）**，数据的生命周期不必告诉编译器，而内存的释放由GC（垃圾处理机制）自动回收，但**由于动态分配内存，堆的存取效率较低（堆的缺点）**；

### 栈Stack

相比于堆，**栈存取速度很快（栈的优势），**仅次于寄存器，一般用来存放**基本类型变量数据如（int、short、long、byte、float、double、boolean、char）与对象的引用**，栈中数据可以共享；但栈中数据大小与生命周期必须是确定的，**缺乏灵活性（栈的缺点）**。