浙江大学实验报告

课程名称：Java 应用技术 实验类型：无

实验项目名称：对 String、StringBuilder 和 StringBuffer 进行源码分析

学生姓名：杨樾人 专业：软件工程 学号：3160104875

同组学生姓名：无 指导老师：鲁伟明

实验地点： 实验日期：2018 年 11 月 18 日

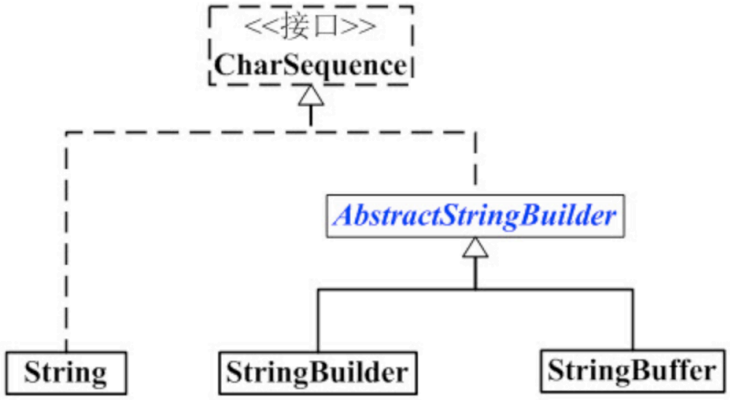
对 String、StringBuilder、StringBuffer 进行源代码分析

目录

一、分析其主要数据组织及功能实现，有什么区别?

- String 字符串常量
- StringBuffer 字符串变量（线程安全）
- StringBuilder 字符串变量（非线程安全）

三者性能：StringBuilder > StringBuffer > String。



图一 三者关系图

1.1 String 源码：

```
public final class String implements java.io.Serializable,
Comparable<String>, CharSequence {
    private final char value[];
    private int hash;
    public String() {
        this.value = new char[0];
    }

    public String(char value[]) {
        this.value = Arrays.copyOf(value, value.length);
    }

    public String(String original) {
        this.value = original.value;
        this.hash = original.hash;
    }
}
```

从 String 在 jdk 中的源代码可以看出，String 类是不可以被继承的，String 类的数据存放在一个以 final 类型的 char 数组，并且该数组是不可变的。并且类中有一个 int 型的变量 hash 用来存放计算后的哈希值。

因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象。

1.2 StringBuilder 源码：

```
public final class StringBuilder extends AbstractStringBuilder
implements java.io.Serializable, CharSequence {
    public StringBuilder() {
            super(16);
        }
    public StringBuilder(int capacity) {
        super(capacity);
    }
    public StringBuilder append(String str) {
        super.append(str);
        return this;
    }
```

从 StringBuilder 类中可以看出，StringBuilder 继承自 AbstractStringBuilder，并且自身不可以再被继承，StringBuilder 的初始化以及 appen 新的字符串的主要操作都在 AbstractStringBuilder 中。

下面是 AbstractStringBuilder 的源码。

```
abstract class AbstractStringBuilder implements Appendable,
CharSequence {
    char[] value;
    int count;
    AbstractStringBuilder(int capacity) {
        value = new char[capacity];
    }
    public AbstractStringBuilder append(String str) {
        if (str == null)
            return appendNull();
```

```
        int len = str.length();
        ensureCapacityInternal(count + len);
        str.getChars(0, len, value, count);
        count += len;
        return this;
    }
    public String toString() {
        // Create a copy, don't share the array
        return new String(value, 0, count);
    }
}
```

AbstractStringBuilder 存放数据时也是一个 char 型的数组，与 String 不同的是没有加 final 修饰符，所以是可以动态改变的。

该类在初始化的过程和 String 类似，但在 append 的时候有所不同，AbstractStringBuilder 是先扩容，再添加进去新的元素。所以 StringBuilder 在 append 字符串的时候直接拼接即可，不需要每次 new 一个新的 StringBuilder 对象。

StringBuilder 和 StringBuffer 通过 toString 方法可转为 String。

1.3 StringBuffer 源码：

```
public final class StringBuffer extends AbstractStringBuilder
implements java.io.Serializable, CharSequence {

    public StringBuffer() {
        super(16);
    }
    public StringBuffer(String str) {
        super(str.length() + 16);
        append(str);
    }

     public synchronized StringBuffer append(String str) {
        super.append(str);
        return this;
    }
    public synchronized StringBuffer append(StringBuffer sb) {
        super.append(sb);
        return this;
```

```
    }
    synchronized StringBuffer append(AbstractStringBuilder asb) {
        toStringCache = null;
        super.append(asb);
        return this;
    }
    public synchronized String toString() {
        if (toStringCache == null) {
            toStringCache = Arrays.copyOfRange(value, 0, count);
        }
        return new String(toStringCache, true);
    }
}
```

StringBuffer 也继承自 AbstractStringBuilder，其主要操作大都是调用 super()来操作实现的，唯一喝 StringBuilder 不同的是在 append 方法中添加了 synchronized 限制，所以是线程安全的。

StringBuffer 和 StringBuilder 的一些主要操作都在 AbstractStringBuilder 父类中完成的，StringBuilder 比 StringBuffer 的速度快的主要原因是 synchronized 造成的，所以对于只在单线程中使用的 string，选择用 StringBuilder 来操作。

1.4 方法比较

1.4.1 initialize：

```
public StringBuilder() {
        super(16);
    }
public StringBuilder(int capacity) {
        super(capacity);
    }
AbstractStringBuilder(int capacity) {
        value = new char[capacity];
    }

public String() {
        this.value = "".value;
    }
public String(String original) {
        this.value = original.value;
```

```
        this.hash = original.hash;
    }
```

三者的初始化方法类似，但是 String 建在了静态存储区，StringBuilder 和 StringBuffer 用 new 创建在堆上的对象。

1.4.2 append：

```
public AbstractStringBuilder append(Object obj) {
        return append(String.valueOf(obj));
    }
public AbstractStringBuilder append(String str) {
        if (str == null)
            return appendNull();
        int len = str.length();
        ensureCapacityInternal(count + len);
        str.getChars(0, len, value, count);
        count += len;
        return this;
    }
```

在 String 中，每次拼接新的字符串，都会 new 一个 StringBuilder 对象，也就是说如果拼接 N 次，就需要 new 出来 N 个 StringBuilder 对象，这样无疑上速度会慢很多。

多余的对象都将成为运行中的垃圾，首先会占用内存，然后 Java 虚拟机会请出垃圾回收线程来回收这些垃圾，这时又会出现 CPU 的损耗，同时这些垃圾对象生成的时候也会产生系统开销。如果在一个循环中使用字符串的加号，导致的系统开销就是不可忽略的了。

StringBuffer 和 StringBuilder 底层也是 char[],，在数组初始化的时候会有一个初始为 16 的生成器，如果不断的 append 一定有超过数组大小的时候。StringBuffer 和 StringBuilder 采用了动态扩展，就像 ArrayList 的实现一样，每次 append 首先检查容量，容量不够则将容量变为原来两倍，如果容量还不够，则直接扩容到需要容量的大小，然后复制原数组的内容到扩展以后的数组中。

String 拼接效率高于 StringBuilder 的特例：

每次 append 结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再

改变对象引用。所以在一般情况下使用 StringBuffer ，特别是字符串对象经常改变的情况下。

但是在我测试的过程中发现，以下情况中 String 的效率远高于 StringBuilder：

```
String S1 = “This is " + “ my " + “ test";
StringBuffer S2 = new StringBuilder(“This is ").append(“ my
").append(“ test");
```

这是因为 String 把 S1 直接看成了：

```
String S1 = “This is only a" + “ simple" + “test";
String S1 = “This is only a simple test";
```

所以 String 的效率会远高于 StringBuilder。

如果是以下情况：

```
String S2 = "This is only a";
String S3 = "simple";
String S4 = "test";
String S1 = S2 +S3 + S4;
```

String 类会按照原本的方法进行新建对象，所以 String 的效率低于 StringBuilder。

1.4.3 equal

```
public boolean equals(Object anObject) {
    //如果引用的是同一个对象，返回真
    if (this == anObject) {
        return true;
    }
    //如果不是 String 类型的数据，返回假
    if (anObject instanceof String) {
        String anotherString = (String) anObject;
        int n = value.length;
        //如果 char 数组长度不相等，返回假
```

```
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            //从后往前单个字符判断，如果有不相等，返回假
            while (n-- != 0) {
                if (v1[i] != v2[i])
                        return false;
                i++;
            }
            //每个字符都相等，返回真
            return true;
        }
    }
    return false;
}
```

String： String 的数据是存放在常量池中的，如果程序中有多个 String 对象，都包含相同的字符串序列，那么这些 String 对象都映射到同一块内存区域，所以如果两次 new String("hello")生成两个实例，虽然是相互独立的，但是对它们使用 hashCode()应该是同样的结果。

StringBuilder 和 StringBuffer：字符串从后往前，判断 String 类中 char 数组 value 的单个字符是否相等，有不相等则为返回 false。如果直到第一个数一直相同，则返回 true。

StringBuffer 和 StringBuilder 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。

1.4.4 concat

```
public String concat(String str) {
    int otherLen = str.length();
    //如果被添加的字符串为空，返回对象本身
    if (otherLen == 0) {
        return this;
    }
    int len = value.length;
```

```
    char buf[] = Arrays.copyOf(value, len + otherLen);
    str.getChars(buf, len);
    return new String(buf, true);
}
```

concat 方法先判断被添加字符串是否为空，决定要不要创建新的对象。

1.4.5 insert

```
public AbstractStringBuilder insert(int index, char[] str, int
offset,
                                    int len)
    {
        if ((index < 0) || (index > length()))
            throw new StringIndexOutOfBoundsException(index);
        if ((offset < 0) || (len < 0) || (offset > str.length - len))
            throw new StringIndexOutOfBoundsException(
                "offset " + offset + ", len " + len + ", str.length "
                + str.length);
        ensureCapacityInternal(count + len);
        System.arraycopy(value, index, value, index + len, count -
index);
        System.arraycopy(str, offset, value, index, len);
        count += len;
        return this;
    }

private void ensureCapacityInternal(int minimumCapacity) {
        // overflow-conscious code
        if (minimumCapacity - value.length > 0) {
            value = Arrays.copyOf(value,
                    newCapacity(minimumCapacity));
        }
    }
```

和 append 不一样，当调用 insert 时，StringBuilder 会尽量减少 string 中被移动的元素，所以每当 insert 一次，就只会增加相应的空间，而不会为下次操作预留空间。

二、说明为什么这样设计，这么设计对 String,StringBuilder 及 StringBuffer 的影响?

2.1 String 类被设计为不可变的原因

1.字符串常量池的需要

字符串常量池是 Java 方法区中一个特殊的存储区域， 当创建一个 String 对象时,假如此字符串值已经存在于常量池中,则不会创建一个新的对象,而是引用已经存在的对象。String a = “abcd”和 String b = “abcd”只创建了一个 String 对象。假若字符串对象允许改变,那么将会导致各种逻辑错误,比如改变一个对象会影响到另一个独立对象. 严格来说，这种常量池的思想,是一种优化手段.

2.允许 String 对象缓存 HashCode**

由于 Java 中 String 对象的哈希码经常被使用，字符串保持不变保证了 hash 码的唯一性，不用每次都去重新计算哈希码。

3.安全性

String 被许多的 Java 类和库用来当做参数,例如网络连接地址 URL,文件路径 path,还有反射机制所需要的 String 参数等， 假若 String 不是固定不变的,将会引起各种安全隐患。

4.线程安全

字符串不可变，同一字符串可以被多个线程共享，并且不需要在多线程时时安全的。

2.2 StringBuilder 和 StringBuffer 的设计理念

String 对象是不可改变的，所以每次使用 System.String 类中的方法之一时，都要在内存中创建一个新的字符串对象，这就需要为该新对象分配新的空间，当操作频繁时会带来昂贵的内存开销。如果要修改字符串而不创建新的对象，则可使用 System.Text.StringBuilder 和 StringBuffer 类，StringBuilder，StringBuffer 内部维护的是字符数组，每次的操作都是改变字符数组的状态，避免创建大量的 String 对象。例如，

当在一个循环中将许多字符串连接在一起时，使用 StringBuilder 类可以提升性能。

StringBuilder 和 StringBuffer 是一个可变的字符序列对象，主要操作是 insert 和 append，提高 String 的效率。

StringBuffer 与 StringBuilder 相比是线程安全的，有加锁开销，效率略低。StringBuilder 非线程安全，不用加锁，效率更高。

三、String,StringBuilder 及 StringBuffer 分别适合哪些场景?

使用 String 类的场景：在字符串不经常变化的场景中可以使用 String 类，例如常量的声明、少量的变量运算。

使用 StringBuffer 类的场景：在频繁进行字符串运算（如拼接、替换、删除等），并且运行在多线程环境中，则可以考虑使用 StringBuffer，例如 XML 解析、HTTP 参数解析和封装。

使用 StringBuilder 类的场景：在频繁进行字符串运算（如拼接、替换、和删除等），并且运行在单线程的环境中，则可以考虑使用 StringBuilder，如 SQL 语句的拼装、JSON 封装等。

四、问题

```
String s1 = "Welcome to Java";;
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

为什么 s1 == s2 返回 false，s1 == s3 返回 true？

答：

s1 和 s3 的内容"Welcome to Java"是放在字符串常量池的，字符串常量池是一个特殊的存储区域，当创建一个 String 对象时,假如此字符串值已经存在于常量池中,则不会创建一个新的对象,而是引用已经存在的对象。所以 s1 == s3 返回 true。

而 s2 是通过 new 的方法生成的，是存储在堆上面，与 s1 指向字符串常量池的地址不同，所以 s1 == s2 返回 false。