# Homework 3

**刘轩铭 3180106071 软件工程**

## 寻找不变类（至少3类）

### String

对String类的源码摘要如下：

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {

    private final byte[] value;

    private final byte coder;

    private int hash;

    ...

    public String replaceAll(String regex, String replacement) {
        return
Pattern.compile(regex).matcher(this).replaceAll(replacement);
    }
}
```

如上所示，我们可以发现：

- String类本身是final类型，意味着他不可以被其他类所继承
- String类的成员变量，都是 private final 类型的，这意味着成员变量不能被外界修改
- String类没有对外开放的，能够修改成员变量的方法。以replaceAll为例，它实现功能的方式并不是直接在字符串内进行替换，而是新建了一个对象，以该对象作为结果返回
- String类甚至缓存了String的hashcode，来确保它不可变

综上，它是一个不可变类

### Boolean

对Boolean类的源码摘要如下：

```
public final class Boolean implements java.io.Serializable,
                                      Comparable<Boolean>
{
    public static final Class<Boolean> TYPE = (Class<Boolean>)
Class.getPrimitiveClass("boolean");

    private final boolean value;
```

```
    ...

    public Boolean(boolean value) {
        this.value = value;
    }

    public Boolean(String s) {
        this(parseBoolean(s));
    }

    ...
}
```

从上面可以看出，Boolean类是不可变的，因为：

- 该类本身是final的，也就是不可以被继承的
- 成员变量都是final类型，确保无法被修改
- 在类的内部没有能够修改成员变量的方法，只有使用变量的值进行操作的方法，如parseBoolean方法
- Boolean类型的变量实例，只能被初始化为true和false两者。因为Boolean只提供了两个构造函数，其返回结果都是boolean类型值

综上，Boolean类是不可变的。

## LocalDate

对LocalDate类进行源码摘要如下：

```
public final class LocalDate
        implements Temporal, TemporalAdjuster, ChronoLocalDate,
Serializable {
    private final int year;

    private final short month;

    private final short day;

    ...
}
```

从上面可以看出，LocalDate类是不可变的，因为：

- 该类本身是final的，也就是不可以被继承的
- 成员变量都是private final类型，确保无法被修改
- 该类没有显式构造函数，只能通过调用内部方法得到LocalDate类的实例对象
- 该类的方法都没有对成员变量或者实例本身进行修改

综上，LocalDate类是不可变的。

## 共同特点：

- 都没有提供任何可以修改对象属性的方法，也没有为属性提供set类型的方法

- 都保证类不会被扩展，用final修饰类
- 都使所有的属性都是final的，初始化之后不能修改属性的值
- 都使所有的可用属性都是private的，防止别的对象访问不可变类的属性
- 都确保对任何可变组件的互斥访问，如果有指向可变对象的属性，一定确保了，这个可变对象不被其他类访问和修改。也没有在类中添加指向其他可变类的属性
- 都是访问安全和线程安全的
- 都有着一些缺点，那就是如果需要新的实例，不能对原对象进行修改，而是一定要创造一个新的对象，这样有着较高的代价

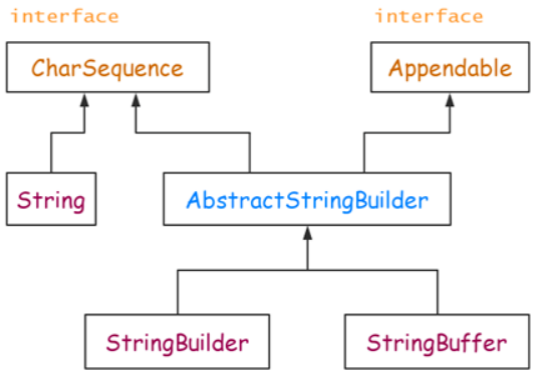
# 对String、StringBuilder以及StringBuffer进行

## 分析其主要数据组织及功能实现，有什么区别?

### 三者的联系

- 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）

三者的关系示意图如下：




### String的源码分析

以下为String类的源码摘要：

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {
    private final byte[] value;
    private final byte coder;
    private int hash;

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

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

```
18
19      public String(char value[]) {
20          this(value, 0, value.length, null);
21      }
22
23      public String(char value[], int offset, int count) {
24          this(value, offset, count, rangeCheck(value, offset, count));
25      }
26
27      ...
28
29  }
```

通过阅读源码可以发现，String类主要实现了其构造函数，边界检测等基础方法，以及对字符串的基本操作方法。String 类不可以被继承，String 类的数据存放在一个以 final 类型的 byte 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。

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。以replace方法为例，该方法判断是否oldchar等于newChar，如果是，会直接返回原对象；如果不是，那么新生成一个对象ret，然后返回ret。String中的所有类似方法都是新生成字符串对象然后返回的。

```
1      public String replace(char oldChar, char newChar) {
2          if (oldChar != newChar) {
3              String ret = isLatin1() ? StringLatin1.replace(value, oldChar,
   newChar)
4                                      : StringUTF16.replace(value, oldChar,
   newChar);
5              if (ret != null) {
6                  return ret;
7              }
8          }
9          return this;
10     }
```

## StringBuffer源码分析

以下为StringBuffer的源码摘要：

```
1   abstract class AbstractStringBuilder implements Appendable, CharSequence {
2
3       byte[] value;
4
5       byte coder;
6
7       int count;
8
9       ...
10  }
```

在结构上他继承了AbstractStringBuilder类，在这个类中定义了字符串的储存形式，是类似于String类的。与 String 不同的是它没有加 final 修饰符，所以是可以动态改变的。

```
```

```java
public final class StringBuffer
    extends AbstractStringBuilder
    implements java.io.Serializable, Comparable<StringBuffer>, CharSequence
{
    public StringBuffer() {
        super(16);
    }

    public StringBuffer(int capacity) {
        super(capacity);
    }

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

    ...

    @Override
    public synchronized int compareTo(StringBuffer another) {
        return super.compareTo(another);
    }

    ...

    @Override
    public synchronized StringBuffer append(Object obj) {
        toStringCache = null;
        super.append(String.valueOf(obj));
        return this;
    }
}
```

在StringBuffer中调用super方法，来进行对象的实例化，从而得到新的StringBuffer对象。

由于在方法中添加了synchronized关键字，即是一种线程锁机制，所以是线程安全的。

在对于字符串的修改上，可以看出，其源码是直接调用了父类的方法进行修改操作，原理是直接在原对象上进行修改，然后返回的也是原有的对象。以append方法为例，他直接调用了父类的append方法，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。返回的是原字符串对象，结果是对原字符串进行了修改。

## StringBuilder源码分析

以下为StringBuilder的源码摘要：

```java
public final class StringBuilder
    extends AbstractStringBuilder
    implements java.io.Serializable, Comparable<StringBuilder>,
CharSequence
{

    public StringBuilder() {
        super(16);
```

```
    }

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

    ...

    public StringBuilder append(String str) {
        super.append(str);
        return this;
    }
}
```

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。但是在多线程程序中，为了保证线程的安全，还是应该选择StringBuffer。

# 说明为什么这样设计，这么设计对三者的影响?

## String不可变的原因

- **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。

- **允许 String 对象一些数据**
  String中有大量的缓存数据，是为了加快速度，减少多余的访问。例如，由于 Java 中 String 对象的哈希码经常被使用，且字符串保持不变保证了 hash 码的唯一性，所以为了不用每次都去重新计算哈希码，类缓存了String的HashCode。

- **为了安全考虑**

  String 被许多的 Java 类和库用来当做参数，假若 String 不是固定不变的,将会引起各种安全隐患。同样这样也保证了线程的安全。

## StringBuilder和StringBuffer设计原因

- 由于String 对象是不可改变的，所以每次使用 String 类中的方法时，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 。那么为了较少开销，直接在字符串上进行操作当然是更加简便和快速的。所以有了StringBuilder和StringBuffer两个类。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。
- 但是，由于StringBuilder是一个可变的字符对象，它们在提高 String 的效率也存在线程不安全的特点。由于加入了sychronized关键字来控制线程，所以StringBuffer 与 StringBuilder 相比是线程安全的。然而由于有加锁开销，其效率却要略低。
- 相比而言，StringBuilder非线程安全，但不用加锁，效率更高。

## 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。

## 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，而s1==s3返回true

- 首先，对象的 '==' 符号判断的是两个引用是否指向同一片内存区域。
- 其次，String是有着常量池技术的。当一个普通字符串常量被创建以后，会被存在常量池中。之后创建字符串常量，会先在pool中寻找是否有相同的内容已经被创建。如果是，则新的引用会指向同一片区域。
- 对于 's1 == s2'， 由于new实际上是在栈上分配了新的内存给对象，所以s2指向的是不同区域，所以是false的。
- 对于 's1 == s3'， 由于常量池技术的影响，s3实际和s1指向的是同一片区域，所以是true的。

# 设计不变类

## 实现Vector, Matrix类，可以进行基本运算，可以修改元素

### Vector实现

源码摘录如下：

```
1  public class Vector {
2
3      private double[] elementData;
4
5      private int capacity;
6
7      private int size;
8
9      public Vector() {
10         this(10);
11     }
12
13     public Vector(int initialCapacity) {
14         if (initialCapacity < 0)
```

```
            throw new IllegalArgumentException("Illegal Capacity: "+
                    initialCapacity);
        this.capacity = initialCapacity;
        this.size = 0;
        this.elementData = new double[initialCapacity];
    }

    ...
}
```

该模块内含elementData，capacity和size这三个成员变量。实现了两个构造函数如上。

在方法部分，可以对Vector进行单个位置的设置（setElementAt）和访问（elementAt）操作，可以获取Vector的大小和容量，对特定位置进行访问，并实现了Vector同维下的加减操作。

当capacity和size相同时，会调用私有方法resize，来重新分配空间。在代码中采用的方法是让capacity扩充到原来的两倍。

## Matrix

源码摘录如下：

```
public class Matrix {

    private double[][] elementData;

    private int rows;

    private int cols;

    public Matrix() { this(5,5); }

    public Matrix(int rows, int cols) {
        if(rows <= 0 || cols <= 0)
            throw new IllegalArgumentException("Cannot set rows with " +
                    rows + " or cols with " + cols);
        this.rows = rows;
        this.cols = cols;
        this.elementData = new double[rows][cols];
    }

    public Matrix(double[][] m) {
        this.rows = m.length;
        this.cols = m[0].length;
        this.elementData = new double[rows][cols];
        for(int i = 0; i < m.length; i++)
            for(int j = 0; j < m[0].length; j++)
                this.elementData[i][j] = m[i][j];
    }

    ...
}
```

该模块内含rows, cols，elementData三个成员变量，提供了多种构造方法。对外提供对特定位置元素的方位和修改，对特定行和列的访问。以及对矩阵进行加减操作。

# 实现UnmodifiableVector, UnmodifiableMatrix不可变类

## UnmodifiableVector

源码摘录如下：

```
public final class UnmodifiableVector {

    private final double[] elementData;

    private final int capacity;

    private final int size;

    public UnmodifiableVector() {
        this(10);
    }

    public UnmodifiableVector(int initialCapacity) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                    initialCapacity);
        this.capacity = initialCapacity;
        this.size = 0;
        this.elementData = new double[initialCapacity];
    }

    public UnmodifiableVector(Vector v) {
        this.capacity = v.capacity();
        this.size = v.size();
        this.elementData = new double[this.capacity];
        for(int i = 0; i < this.size; i++)
            this.elementData[i] = v.elementAt(i);
    }

    public UnmodifiableVector(UnmodifiableVector v) {
        this.capacity = v.capacity();
        this.size = v.size();
        this.elementData = new double[this.capacity];
        for(int i = 0; i < this.size; i++)
            this.elementData[i] = v.elementData[i];
    }

    public UnmodifiableVector(int initialCapacity, int initialSize,
double[] elementData) {
        if(initialCapacity < initialSize)
            throw new IllegalArgumentException("Cannot set a initial size
bigger than initial capacity.");
        if(initialSize > elementData.length)
            throw new IllegalArgumentException("Cannot set a initial size
less than array length.");
        this.capacity = initialCapacity;
        this.size = initialSize;
        this.elementData = new double[this.capacity];
        for(int i = 0; i < initialSize; i++)
```

```
            this.elementData[i] = elementData[i];
    }

    ...
}
```

该模块的功能和Vector模块类似。为了让模块变为不可变的，对类进行了一些限制：将Class设置为final类型，使它不能被继承；将成员变量都设计为private fina类型，使它们不能被修改；对外开放的接口方法都不对内置成员变量进行修改。

以setElementAt方法为例：

```
    public UnmodifiableVector setElementAt(double elem, int index) {
        if (index >= size || index < 0) {
            throw new IndexOutOfBoundsException(index + " >= " + size);
        }

        double[] newElementData = new double[this.size];
        for(int i = 0; i < this.size; i++)
            newElementData[i] = this.elementAt(i);
        newElementData[index] = elem;

        UnmodifiableVector ret = new UnmodifiableVector(this.capacity,
this.size, newElementData);
        return ret;
    }
```

该方法对特定位置的元素进行修改。首先判断该index是否在范围内，若不是则会抛出异常。之后会先复制出原对象的elementData成员，对特定位置进行修改。新建一个对象并返回。

### UnmodifiableMatrix

该类与上述UnmodifiableVector类的设置类似，也是在原可变类基础上进行了相应的限制，使其变为不可变类。

详细说明可以见源代码。

## 实现MathUtils类

源码如下：

```java
package com.MathUtils;

public class MathUtils {
    public static UnmodifiableMatrix getUnmodifiableMatrix(Matrix m) {
        UnmodifiableMatrix ret = new UnmodifiableMatrix(m);
        return ret;
    }

    public static UnmodifiableVector getUnmodifiableVector(Vector v) {
        UnmodifiableVector ret = new UnmodifiableVector(v);
        return ret;
    }
}
```

这一部分主要实现的是从可变类向不可变类的转化。

通过调用相应不可变类的构造函数即可实现。

## 测试说明

### 对Vector进行测试：

```java
...
    // VECTOR TEST.
        System.out.println("Test vector.");
        Vector v = new Vector(5);
        for(int i = 0; i < 30; i++)
            v.push_back((double)i);
        for(int i = 0; i < v.size(); i++)
            v.setElementAt(30-v.elementAt(i), i);
        for(int i = 0; i < v.size(); i++)
            System.out.print(v.elementAt(i) + " ");
        System.out.println();
        System.out.println(v.size());
        System.out.println(v.capacity());
        System.out.println();
...
```

测试结果如下：

```
Test unmodifiable vector.
100.0
200.0
2.0
30.0 29.0 28.0 27.0 26.0 25.0 24.0 23.0 22.0 21.0 20.0 19.0 18.0 17.0 16.0 15.0 14.0 13.0 12.0 11.0 10.0 9.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0
```

可以看出，上面所测试的各个方法的功能均正常。

### 对Matrix进行测试

```java
...
// MATRIX TEST.
        System.out.println("Test matrix.");
        double[][] dArray = new double[5][6];
        for(int i = 0; i < dArray.length; i++)
```

```
            for(int j = 0; j < dArray[0].length; j++)
                dArray[i][j] = i + j;
        Matrix m = new Matrix(dArray);
        m.setElementAt(3, 4, 20);
        for(int i = 0; i < m.rows(); i++) {
            for (int j = 0; j < m.cols(); j++)
                System.out.print(m.elementAt(i, j) + " ");
            System.out.println();
        }
        double[] arr = m.elementsAtRow(3);
        for(int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
        arr = m.elementsAtCol(1);
        for(int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
...
```

测试结果如下：

```
Test matrix.
0.0 1.0 2.0 3.0 4.0 5.0
1.0 2.0 3.0 4.0 5.0 6.0
2.0 3.0 4.0 5.0 6.0 7.0
3.0 4.0 5.0 6.0 20.0 8.0
4.0 5.0 6.0 7.0 8.0 9.0
3.0 4.0 5.0 6.0 20.0 8.0
1.0 2.0 3.0 4.0 5.0
```

可以看出，上面测试的方法功能均正常。其中前面显示了对Matrix修改后的结果，后面两行是对特定行列的获取结果。

## UnmodifiableVector

```
...
// UNMODIFIABLE VECTOR TEST.
        System.out.println("Test unmodifiable vector.");
        UnmodifiableVector uv = MathUtils.getUnmodifiableVector(v);
        UnmodifiableVector uvv = uv.setElementAt(100, 5);
        System.out.println(uvv.elementAt(5));
        uvv = uv.push_back(200);
        System.out.println(uvv.back());
        uvv = uv.erase_back();
        System.out.println(uvv.back());
        for(int i = 0; i < uv.size(); i++)
            System.out.print(uv.elementAt(i) + " ");
        System.out.println();
...
```

测试部分首先调用MathUtils.getUnmodifiableVector方法，从可变类实例构造了uv这个不可变类的实例。然后对各自的功能进行测试。可以发现，对uv进行了“修改”后，其本身其实没有发生变化，只是返回了一个新的对象，该对象和原来相比被改变了。

```
Test unmodifiable vector.
100.0
200.0
2.0
30.0 29.0 28.0 27.0 26.0 25.0 24.0 23.0 22.0 21.0 20.0 19.0 18.0 17.0 16.0 15.0 14.0 13.0 12.0 11.0 10.0 9.0 8.0 7.0 6.0 5.0 4.0 3.0 2.0 1.0
```

由测试结果来看，各方法的功能也是正常的。

## UnmodifiableMatrix

```
...
        System.out.println("Test Unmodifiable Matrix");
        UnmodifiableMatrix um = MathUtils.getUnmodifiableMatrix(m);
        UnmodifiableMatrix umm = um.setElementAt(3, 4, 40);
        System.out.println(um.elementAt(3,4));
        System.out.println(umm.elementAt(3,4));
        UnmodifiableMatrix um2 = MathUtils.getUnmodifiableMatrix(m);
        umm = um.add(um2);
        for(int i = 0; i < umm.rows(); i++) {
            for (int j = 0; j < umm.cols(); j++)
                System.out.print(umm.elementAt(i, j) + " ");
            System.out.println();
        }
...
```

测试部分首先调用MathUtils.getUnmodifiableMatrix方法，从可变类实例构造了um这个不可变类的实例。然后对各自的功能进行测试。可以发现，对um进行了“修改”后，其本身其实没有发生变化，只是返回了一个新的对象，该对象和原来相比被改变了。此外，umm作为加法的结果，其运算结果正常为um和um2的和，而um本身没有发生变化。

```
Test Unmodifiable Matrix
20.0
40.0
0.0 2.0 4.0 6.0 8.0 10.0
2.0 4.0 6.0 8.0 10.0 12.0
4.0 6.0 8.0 10.0 12.0 14.0
6.0 8.0 10.0 12.0 40.0 16.0
8.0 10.0 12.0 14.0 16.0 18.0
```

经测试，功能都是正常的。

## MathUtils

由上面的测试过程看出，调用两个方法的结果也是正常的。