Java应用技术-HW3

一、实验内容

1. 寻找 JDK 库中的不变类（至少 3 类），并进行 源码分析，分析其为什么是不变的？文档说明其共性。
2. 对String、StringBuilder以及StringBuffer进行源代码分析，
    1. 分析其主要数据组织及功能实现，有什么区别？
    2. 说明为什么这样设计，这么设计对String, StringBuilder及SringBuffer的影响？
    3. String, StringBuilder及StringBuffer分别适合哪些场景？
3. 设计不变类
    1. 实现 Vector, Matrix 类，可以进行向量、矩阵的基本运算、可以得到（修改） Vector 和 Matrix 中的元素，如 Vector 的第 k 维， Matrix 的第 i,j 位的值。
    2. 实现 UnmodifiableVector , UnmodifiableMatrix 不可变类
    3. 实现 MathUtils ，含有静态方法
        - UnmodifiableVector getUnmodifiableVector (Vector v)
        - UnmodifiableMatrix getUnmodifiableMatrix (Matrix m)

二、实验环境

- 操作系统：Windows 10 发行版
- JDK版本：1.8
- IDE：IntelliJ IDEA 2020版

三、实验过程和结果

3.1 第一步：寻找JDK中的不变类

- Java中常见的不变类包括String类型，基本的包装类和大数类型BigDemical，BigInteger等等，可以在JDK中找到他们的源代码并进行阅读和分析：
    - 对于String，我们发现JDK源代码中用了一个数组`private final char value[];`来存放字符串中的各个字符，并且类本身的定义如下，说明这个类自身的定义也是final类型的，不能被其他类继承，因此String是个不可变类

```
public final class String
    implements java.io.Serializable, Comparable<String>,
CharSequence {
    private final char value[];
}
```

    - 对于Integer类型，我们发现JDK源代码中类定义和关键变量如下

```
public final class Integer extends Number implements
Comparable<Integer> {
    private final int value;
}
```

    - value是用来存储Integer所包装的整型值的，而被声明为了private final类型，因此一旦初始化之后就不能修改其值
    - 而类的定义中也声明了final类型，不能被其他类继承，因此也不会因为继承而导致value被修改，因此Integer类是不可变类
  - 对于BigInteger类型，其类定义和关键变量声明如下

```
public class BigInteger extends Number implements
Comparable<BigInteger> {
    final int signum;
    final int[] mag;
}
```

    - 其中signum表示符号位，-1表示负数，0表示0，1表示正数，mag将保存的大整数按位存储，而二者都是final类型的，因此一旦初始化之后就不可以再改变
    - 值得注意的是类本身不是final类型，因此是可以被继承的
- 总结，不变类的共性——用一定的数据结构(基本的变量类型、数组等)封装需要包装的内容，并将这些数据结构声明为private final类型，即一经初始化之后就不能改变，又因为是private，不能在类外直接访问，保证了封装性，这样子定义的类都可以作为不变类来使用

3.2 第二步：分析String，StringBuilder和StringBuffer的源代码

3.2.1 分析其主要数据组织

- String的基本数据结构如下所示

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -6849794470754667710L;

    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;
    }
}
```

    - 这表明String类型调用了`java.io.Serializable, Comparable<String>, CharSequence`三个接口，并且有这样一些核心的成员变量
    - 其中hash用来存储其**哈希值**，目的是暂存字符串的哈希值，可以减少后面反复调用hashcode的时候所需要消耗的时间
        - 相当于空间换时间
        - serialVersionUID主要是在序列化里使用的，这里暂时不关注
    - String类的特点是因为value声明的是final类型，因此一旦初始化之后，char数组value中的值是不能改变的，是不可变类
        - 每次需要对string进行改变的时候，都会产生一个新的string对象
    - 从构造函数来看，String的构造函数主要有三种类型：
        - 默认的构造函数是初始化一个长度为1的字符串
        - 可以用一个char类型的数组来构造一个string，此时string中的value直接copy参数中的数组
        - 也可以用另一个string来进行拷贝构造，此时直接使得value和hashcode拷贝参数中string的对应值

- StringBuilder和StringBuffer的类定义声明如下

```
public final class StringBuilder
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
{
    //some thing
    static final long serialVersionUID = 4383685877147921099L;
}

public final class StringBuffer
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
{
    //some thing
    private transient char[] toStringCache;
    static final long serialVersionUID = 3388685877147921107L
}
```

    - 我们观察到这两个类都是**继承了抽象类AbstractStringBuilder**
    - 而抽象类AbstractStringBuilder的定义是这样子的：

```
abstract class AbstractStringBuilder implements Appendable, CharSequence
{
    char[] value;
    int count;
}
```

    - 因此两个类都从基类中得到了数组value和长度count的属性，而这两个因为没有声明为final，因此是**可以在派生类中改变**的
    - 而StringBuffer相比于Builder多了一个toStringCache的数组，并且有一个表示被序列化的transient的关键字，可以判断这个数组是用来做缓冲区的。
- 总结：三个类主要的数据组织都是类似的，用一个char数组来存储字符串中的每个字符
    - 区别是String的数组是final类型不可变，而另外两个是可以改变的
    - 另外三个类本身都是final类型的类，不能被其他类继承

3.2.2 功能实现上的区别

- 初始化的时候，String建立在**静态存储区**，而另外两个类是动态分配的内存，因此在堆上分配内存
  - Builder和Buffer因为会调用父类的构造函数来初始化value数组
  - String直接在自己的构造函数中对value进行初始化

```
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;
}
```

- String类型没有提供改变value数据内容的方法，事实上因为是final类型的，就算提供了也过不了编译，而StringBuilder和StringBuffer则提供了一些改变value数组内容的方法，比如

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

  - 当然这里还有一系列重载的append，就不一一复制粘贴了，这些append方法主要是覆写了父类中的append方法，将一个对象添加到数组末尾，这就可以改变value数组中的内容
  - StringBuffer中也有append方法，并且实现方式和Builder基本完全一样，区别是多了`synchronized`关键字的声明，目的是**保证线程的安全**，这也是StringBuffer和StringBuilder的主要区别

```
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);
}
```

- 另外StringBuilder还提供了`toString`方法，可以返回一个对象转化成的String类型

```
public String toString() {
    // Create a copy, don't share the array
    return new String(value, 0, count);
}
```

  - StringBuffer中也提供了类型的方法，并且多了线程安全的声明`synchronized`

- 字符串的拼接：
  - String类提供了concat的方法用于实现两个字符串的拼接，但是因为是不可变类所以生成了一个新的对象
  - 但是concat会先判断被添加的字符串是不是空的，如果被添加的字符串是空的就直接返回原本对象的引用this
  - StringBuilder和StringBuffer提供了append方法来修改字符串，直接在数组value中修改，并且效率往往比concat要高
- 字符串的相等：
  - 三个类都可以用equal方法来判断字符串是否相等
  - 判断的标准：StringBuffer和StringBuilder都是通过value数组的逐个比较来判断整个字符串是否相等的
  - String需要比较两个字符串是否映射到了相同的内存区域，这是因为JVM会给String建立常量池，内容相同的字符串会指向常量池中的同一个区域

3.2.3 说明为什么这样设计，这么设计对String, StringBuilder及SringBuffer的影响？

- String为什么设计成不可变类
  - 因为Java中有字符常量池的机制，在创建一个String对象的时候会现在常量池中寻找是否已经有相同内容的对象，如果找到了就不再创建新的对象，而**常量池需要存放的都是常量**，因此需要将String设计为不可变的类型
  - 因为String的hashcode使用频率比较高，设计成不可变类可以缓存hashcode，提高运行的效率，不需要每次发生改变就重新计算一次hashcode
  - 安全性：用来存储URL，文件路径等信息的时候，String的不可变性使其更加安全，另外因为不可变，所以一个String就算在线程之间共享，也不需要担心线程安全
- StringBuilder和Buffer的设计理念
  - String的不可变性也带来了很多问题，比如使用String类中的方法经常需要创建新的对象，分配新的内存空间，经常使用会导致内存开销非常大
  - Builder和Buffer就是为了解决这个问题，提供了可以修改内容的字符串，通过insert和append等操作来提高面对需要大量处理的字符串时候的运行效率
  - Buffer比起Builder而言是线程安全的，但是需要加锁，效率不如Builder
  - 我认为这三者的总体设计理念就是，为不同的使用场景提供专门的高效率类库，通过给开发者更多的选择来提高其开发效率，不用兼顾一些琐碎的细节，这也是Java语言的初衷

3.2.4 String, StringBuilder及StringBuffer分别适合哪些场景

- String类适用于不需要改变字符串内容的场景，比如构建常量，和变量运算比较少的时候
- StringBuffer的使用场景是：对字符串运算使用较多，并且程序是多线程的时候，用于确保效率和线程安全的trade-off，比如XML的解析和HTTP的解析
- StringBuilder的使用场景是：需要频繁修改字符串中内容的单线程程序，比如在Leetcode算法题中非常常见

3.3 类的设计和实现

3.3.1 普通类型Vector和Matrix

- 对于vector和matrix类，我分别实现了如下成员变量和方法：

```
public class Vector {
    private double[] elements;
    private int length;

    public Vector(double[] nums)
    public Vector(Vector m)
    public int getLength()
    public double elementAt(int k)
    public void showVector()
    public void changeVector(int k, double val)
    public void addVector(Vector that)
    // 以下定义了向量的若干种基本运算，包含加减法，内积和数量积
    public void subVector(Vector that)
    public void dotProduct(double dot)
    public double innerProduct(Vector that)
}

public class Matrix {
    private double[][] elements;
    private int row;
    private int col;

    public Matrix(double[][] nums)
    public Matrix(Matrix that)
    public Matrix()
    public void showMatrix()
    public int getRow() { return row; }
    public int getCol() { return col; }
    public double elementAt(int i, int j)
    public void changMatrix(int i, int j, double val)
    public boolean equalSize(Matrix that)
    public void addMatrix(Matrix that)
    public void subMatrix(Matrix that)
    public Matrix mulMatrix(Matrix that)
    public boolean canMultiply(Matrix that)
}
```

- 我编写了两个测试函数testVector和testMatrix，测试了每种基本运算的正确性，在main函数中测试这两个类，测试结果如下，具体的测试程序可以看提交的源代码

```
------------------Vector----------------------
1.0 2.0 3.0
1.0 2.0 3.0
2.0 4.0 6.0
a * b = 28.0
7.0 14.0 9.0
6.0 12.0 6.0
6.0 100.0 6.0
------------------Matrix----------------------
1.0 2.0 3.0
4.0 5.0 6.0
1.0 2.0 3.0
4.0 5.0 6.0
2.0 4.0 6.0
8.0 10.0 12.0
44.0 56.0
98.0 128.0

Process finished with exit code 0
```

3.3.2 不变类UnmodifiedVector和UnmodifiedMatrix

- 和上面的普通类的区别就是，我将存放向量/矩阵中元素的数组声明为了final类型，并且由于是不可变类，因此不能直接在原数组上修改向量和矩阵存储的值
- 基本运算的结果需要new一个新的对象作为返回值，因此返回值类型也从void改为了UnmodifiedVector和UnmodifiedMatrix，类的定义如下

```
public class UnmodifiableVector {
    private final double[] elements;
    private final int length;

    public UnmodifiableVector(double[] nums)
    public UnmodifiableVector(UnmodifiableVector that)
    public int getLength()
    public double elementAt(int k)
    public void showUnmodifiableVector()
    public UnmodifiableVector addUnmodifiableVector(UnmodifiableVector
that)
    public UnmodifiableVector subUnmodifiableVector(UnmodifiableVector
that)
    public UnmodifiableVector dotProduct(double val)
    public double innerProduct(UnmodifiableVector that)
}

public class UnmodifiableMatrix {
    private final double[][] elements;
```

```
    private final int row;
    private final int col;

    public UnmodifiableMatrix(double[][] nums)
    public void showMatrix()
    public int getRow()
    public int getCol()
    public double elementAt(int i, int j)
    public boolean equalSize(UnmodifiableMatrix that)
    public UnmodifiableMatrix addMatrix(UnmodifiableMatrix that)
    public UnmodifiableMatrix subMatrix(UnmodifiableMatrix that)
    public UnmodifiableMatrix mulMatrix(UnmodifiableMatrix that)
    public boolean canMultiply(UnmodifiableMatrix that)
}

```

- 我也编写了对应的测试程序来测试两个类是否能够完成这些基本运算，具体的源代码可以看提交的代码文件，测试结果如下：

```
------------------Unmodified Vector----------------------
1.0 2.0 3.0
2.0 4.0 6.0
3.0 6.0 9.0
-1.0 -2.0 -3.0
84.0
------------------Unmodified Matrix----------------------
1.0 2.0 3.0
4.0 5.0 6.0
1.0 2.0
3.0 4.0
5.0 6.0
2.0 4.0 6.0
8.0 10.0 12.0
1.0 2.0 3.0
4.0 5.0 6.0
22.0 28.0
49.0 64.0

Process finished with exit code 0
```

3.3.3 MathUtil

- 我实现了两个转换函数用来将可变类转化为对应的不可变类，测试代码如下

```java
public static void testMathUtils() {
        double[] nums = {1, 2, 3};
        Vector a = new Vector(nums);
        UnmodifiableVector unmodified_a = MathUtils.getUnmodifiableVector(a);
        a.showVector();
        unmodified_a.showUnmodifiableVector();

        Matrix b = new Matrix(new double[][] {{1, 2, 3}, {4, 5, 6}});
        UnmodifiableMatrix unmodified_b = MathUtils.getUnmodifiableMatrix(b);
        b.showMatrix();
        unmodified_b.showMatrix();
}
```

- 测试结果如下，表明我们生成了对应的不可变类

```
1.0 2.0 3.0
1.0 2.0 3.0
1.0 2.0 3.0
4.0 5.0 6.0
1.0 2.0 3.0
4.0 5.0 6.0

Process finished with exit code 0
```