Homework2

JDK库中的不变类

String

- **String类**被声明为 `final`，意味着它不能被继承，从而避免了子类破坏 `String` 类的不变性。

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

- **String类**的所有成员变量都是私有的，并且用 `final` 修饰，确保一旦初始化就不能被改变。除了 `hash` 和 `hashIsZero`，在调用 `hashCode()` 方法设置成该字符串的hash值，否则默认值是0，一旦设置成字符串的 `hash` 值后，也不会改变，因为其由字符串本身决定。

```
    @Stable
    private final byte[] value;
    private final byte coder;
    private int hash; // Default to 0
    private boolean hashIsZero; // Default to false;
    @java.io.Serial
    private static final long serialVersionUID = -6849794470754667710L;
    static final boolean COMPACT_STRINGS;
```

- **String类**提供了一些方法如 `substring()`、`replace()`等，这些方法不会改变原有字符串的内容，而是调用 `StringLatin1` / `StringUTF16` 中的对应的方法，返回新建的字符串对象。

```
public String substring(int beginIndex, int endIndex) {
        int length = length();
        checkBoundsBeginEnd(beginIndex, endIndex, length);
        if (beginIndex == 0 && endIndex == length) {
            return this;
        }
        int subLen = endIndex - beginIndex;
        return isLatin1() ? StringLatin1.newString(value, beginIndex, subLen)
                          : StringUTF16.newString(value, beginIndex, subLen);
    }

public String replace(char oldChar, char newChar) {
        if (oldChar != newChar) {
            String ret = isLatin1() ? StringLatin1.replace(value, oldChar,
newChar)
                                    : StringUTF16.replace(value, oldChar,
newChar);
            if (ret != null) {
                return ret;
            }
        }
        return this;
    }
```

Boolean

- **Boolean类**被声明为 `final`，意味着它不能被继承，从而避免了子类破坏 `Boolean` 类的不变性。

```
public final class Boolean implements java.io.Serializable,
                                      Comparable<Boolean>, Constable {...}
```

- **Boolean类**定义了两个静态常量TRUE和FALSE，分别对应布尔值true和false。这两个常量在类加载时就已经被创建，并且在程序运行期间不会被改变。

```
    public static final Boolean TRUE = new Boolean(true);

    public static final Boolean FALSE = new Boolean(false);
```

- **Boolean类**的所有成员变量都用 `final` 修饰，确保一旦初始化就不能被改变。

```
    public static final Boolean TRUE = new Boolean(true);
    public static final Boolean FALSE = new Boolean(false);
    @SuppressWarnings("unchecked")
    public static final Class<Boolean> TYPE = (Class<Boolean>)
Class.getPrimitiveClass("boolean");
    private final boolean value;
    @java.io.Serial
    private static final long serialVersionUID = -3665804199014368530L;
```

共性

- 对于成员变量，通常用 final 修饰，使得即使外部能够访问，也不能修改。
- 对于类的声明，用 final 修饰，使得其不能被继承而使得其子类能修改成员变量的值。
- 对于成员方法，不提供修改状态的方法，如果有需要改变对象的内容的，就要重新创建一个新的对象返回，而不是改变原来的对象。

类MutableMatrix和InmutableMatrix

MutableMatrix

```
class MutableMatrix {
    // 成员变量
    private int row;
    private int col;
    private int[][] matrixValues;

    /**
     * 构造函数，初始化行、列和矩阵值
     * @param row 矩阵的行数
     * @param col 矩阵的列数
     */
    public MutableMatrix(int row, int col) {...}

    /**
```

```
     * 构造函数，从另一个矩阵复制行、列和矩阵值
     * @param row 矩阵的行数
     * @param col 矩阵的列数
     * @param otherMatrixValues 另一个矩阵的值
     */
    public MutableMatrix(int row, int col, int[][] otherMatrixValues) {...}

    /**
     * 构造函数，从另一个可变矩阵复制行、列和矩阵值
     * @param others 另一个可变矩阵
     */
    public MutableMatrix(MutableMatrix others) {...}

    /**
     * 构造函数，从不可变矩阵复制行、列和矩阵值
     * @param others 一个不可变矩阵
     */
    public MutableMatrix(InmutableMatrix others) {...}

    /**
     * 获取矩阵的行数
     * @return 矩阵的行数
     */
    public int row() {...}

    /**
     * 获取矩阵的列数
     * @return 矩阵的列数
     */
    public int col() {...}

    /**
     * 获取矩阵的值
     * @return 矩阵的值
     */
    public int[][] matrixValues() {...}

    /**
     * 将两个可变矩阵相加，并更新当前矩阵的值
     * @param others 另一个可变矩阵
     * @return 更新后的可变矩阵对象本身（this）
     * @throws Exception 如果两个矩阵的行数或列数不同，抛出异常
     */
    public MutableMatrix add(MutableMatrix others) throws Exception {...}

    /**
     * 将两个可变矩阵相减，并更新当前矩阵的值
     * @param others 另一个可变矩阵
     * @return 更新后的可变矩阵对象本身（this）
     * @throws Exception 如果两个矩阵的行数或列数不同，抛出异常
     */
    public MutableMatrix sub(MutableMatrix others) throws Exception {...}

    /**
     * 将可变矩阵的每个元素乘以一个标量值，并更新当前矩阵的值
     * @param number 标量值
```

```
     * @return 更新后的可变矩阵对象本身（this）
     */
    public MutableMatrix scalarMultiply(int number) {...}
    /**
     * 将两个可变矩阵相乘，并更新当前矩阵的值
     * @param others 另一个可变矩阵
     * @return 更新后的可变矩阵对象本身（this）
     * @throws Exception 如果第一个矩阵的列数不等于第二个矩阵的行数，抛出异常
     */
    public MutableMatrix multiply(MutableMatrix others) throws Exception {...}
    /**
     * 打印矩阵的值
     */
    public void print() {...}
}
```

- 可变矩阵的成员变量不用 `final` 修饰，它们原则上都是可修改的。
- 因为修改行和列会导致 `matrixValue` 的尺寸发生改变，进而可能会丢失部分信息，所以此处仅提供了修改 `matrixValues` 的接口：可以通过 `matrixValues()` 方法获得 `matrixValues` 的引用，进而修改其中的值。

```
    public int[][] matrixValues() {
        return matrixValues;
    }
```

- 对于可变矩阵的运算，最后返回的都是自身的引用 `this`，因此矩阵链式运算也是支持的。

```
    public MutableMatrix add(MutableMatrix others) throws Exception {
        // check row and col
        if (row != others.row || col != others.col) {
            throw new Exception("The size of both mutrix should be identical!");
        }
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                matrixValues[i][j] += others.matrixValues[i][j];
            }
        }
        return this;
    }
```

InmutableMatrix

```
final class InmutableMatrix {
    // 成员变量
    private final int row;
    private final int col;
    private final int[][] matrixValues;

    /**
     * 构造函数，初始化行、列和矩阵值
     * @param row 矩阵的行数
     * @param col 矩阵的列数
     */
```

```java
    public InmutableMatrix(int row, int col) {...}

    /**
     * 构造函数，从另一个矩阵复制行、列和矩阵值
     * @param row 矩阵的行数
     * @param col 矩阵的列数
     * @param otherMatrixValues 另一个矩阵的值
     */
    public InmutableMatrix(int row, int col, int[][] otherMatrixValues) {...}

    /**
     * 构造函数，从另一个不可变矩阵复制行、列和矩阵值
     * @param others 另一个不可变矩阵
     */
    public InmutableMatrix(InmutableMatrix others) {...}

    /**
     * 构造函数，从可变矩阵复制行、列和矩阵值
     * @param others 一个可变矩阵
     */
    public InmutableMatrix(MutableMatrix others) {...}

    /**
     * 获取矩阵的行数
     * @return 矩阵的行数
     */
    public int row() {...}

    /**
     * 获取矩阵的列数
     * @return 矩阵的列数
     */
    public int col() {...}

    /**
     * 获取矩阵的值
     * @return 矩阵的值
     */
    public int[][] matrixValues() {...}

    /**
     * 将两个不可变矩阵相加
     * @param others 另一个不可变矩阵
     * @return 相加后的不可变矩阵
     * @throws Exception 如果两个矩阵的行数或列数不同，抛出异常
     */
    public InmutableMatrix add(InmutableMatrix others) throws Exception {...}

    /**
     * 将两个不可变矩阵相减
     * @param others 另一个不可变矩阵
     * @return 相减后的不可变矩阵
     * @throws Exception 如果两个矩阵的行数或列数不同，抛出异常
     */
    public InmutableMatrix sub(InmutableMatrix others) throws Exception {...}
```

```
    /**
     * 将不可变矩阵的每个元素乘以一个标量值
     * @param number 标量值
     * @return 乘以标量值后的不可变矩阵
     */
    public InmutableMatrix scalarMultiply(int number) {...}

    /**
     * 将两个不可变矩阵相乘
     * @param others 另一个不可变矩阵
     * @return 相乘后的不可变矩阵
     * @throws Exception 如果第一个矩阵的列数不等于第二个矩阵的行数，抛出异常
     */
    public InmutableMatrix multiply(InmutableMatrix others) throws Exception
{...}

    /**
     * 打印矩阵的值
     */
    public void print() {...}
}
```

- 不可变矩阵的成员变量都用final修饰，它们都是不可修改的。
- 对于 matrixValues() 方法，不可变矩阵返回的是原对象的matrixValues的深拷贝，而不是引用，所以外部是不能修改的。

```
    public int[][] matrixValues() {
        int[][] otherMatrixValues = new int[row][col];
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                otherMatrixValues[i][j] = matrixValues[i][j];
            }
        }
        return otherMatrixValues;
    }
```

- 对于不可变矩阵的运算，最后返回的都是一个新创建的对象，而原对象的值是不变的。

```
    public InmutableMatrix add(InmutableMatrix others) throws Exception {
        // check row and col
        if (row != others.row || col != others.col) {
            throw new Exception("The size of both mutrix should be identical!");
        }
        int[][] retValues = new int[row][col];
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                retValues[i][j] = matrixValues[i][j] + others.matrixValues[i][j];
            }
        }
        return new InmutableMatrix(row, col, retValues);
    }
```

功能测试

初始化：

```
        // 功能测试
        // 初始化样例
        int[][] valuea = {{1, 2, 3}, {4, 5, 6}};
        int[][] valueb = {{7, 8, 9}, {10, 11, 12}};
        int[][] valuec = {{1, 2}, {3, 4}, {5, 6}};

        InmutableMatrix inmutableMatrixA;
        InmutableMatrix inmutableMatrixB;
        InmutableMatrix inmutableMatrixRes;

        MutableMatrix mutableMatrixA;
        MutableMatrix mutableMatrixB;
```

加法的测试

```
        // 测试矩阵加法
        inmutableMatrixA = new InmutableMatrix(2, 3, valuea);
        inmutableMatrixB = new InmutableMatrix(2, 3, valueb);
        inmutableMatrixRes = inmutableMatrixA.add(inmutableMatrixB);

        mutableMatrixA = new MutableMatrix(2, 3, valuea);
        mutableMatrixB = new MutableMatrix(2, 3, valueb);
        mutableMatrixA.add(mutableMatrixB);

        /*
         * 预期结果
         * 8 10 12
         * 14 16 18
         */
        System.out.println("InmutableMatrix Add Result:");
        inmutableMatrixRes.print();
        System.out.println("MutableMatrix Add Result:");
        mutableMatrixA.print();
```

输出：

```
InmutableMatrix Add Result:
8 10 12
14 16 18
MutableMatrix Add Result:
8 10 12
14 16 18
```

减法的测试

```
        // 测试矩阵减法
        inmutableMatrixA = new InmutableMatrix(2, 3, valuea);
        inmutableMatrixB = new InmutableMatrix(2, 3, valueb);
        inmutableMatrixRes = inmutableMatrixA.sub(inmutableMatrixB);

        mutableMatrixA = new MutableMatrix(2, 3, valuea);
        mutableMatrixB = new MutableMatrix(2, 3, valueb);
        mutableMatrixA.sub(mutableMatrixB);

        /*
         * 预期结果
         * -6 -6 -6
         * -6 -6 -6
         */
        System.out.println("InmutableMatrix Sub Result:");
        inmutableMatrixRes.print();
        System.out.println("MutableMatrix Add Result:");
        mutableMatrixA.print();
```

输出:

```
InmutableMatrix Sub Result:
-6 -6 -6
-6 -6 -6
MutableMatrix Sub Result:
-6 -6 -6
-6 -6 -6
```

数乘的测试

```
        //测试矩阵数乘
        inmutableMatrixA = new InmutableMatrix(2, 3, valuea);
        inmutableMatrixRes = inmutableMatrixA.scalarMultiply(2);

        mutableMatrixA = new MutableMatrix(2, 3, valuea);
        mutableMatrixA.scalarMultiply(2);

        /*
         * 预期结果
         * 2 4 6
         * 8 10 12
         */
        System.out.println("InmutableMatrix Scalar Multiply Result:");
        inmutableMatrixRes.print();
        System.out.println("MutableMatrix Scalar Multiply Result:");
        mutableMatrixA.print();
```

输出:

```
InmutableMatrix Scalar Multiply Result:
2 4 6
8 10 12
MutableMatrix Scalar Multiply Result:
2 4 6
8 10 12
```

乘法的测试

```
        //测试矩阵乘法
        inmutableMatrixA = new InmutableMatrix(2, 3, valuea);
        inmutableMatrixB = new InmutableMatrix(3, 2, valuec);
        inmutableMatrixRes = inmutableMatrixA.multiply(inmutableMatrixB);

        mutableMatrixA = new MutableMatrix(2, 3, valuea);
        mutableMatrixB = new MutableMatrix(3, 2, valuec);
        mutableMatrixA.multiply(mutableMatrixB);

        /*
         * 预期结果
         * 22 28
         * 49 64
         */
        System.out.println("InmutableMatrix Multiply Result:");
        inmutableMatrixRes.print();
        System.out.println("MutableMatrix Multiply Result:");
        mutableMatrixA.print();
```

输出:

```
InmutableMatrix Multiply Result:
22 28
49 64
MutableMatrix Multiply Result:
22 28
49 64
```

链式运算的测试

```
        inmutableMatrixA = new InmutableMatrix(2, 3, valuea);
        inmutableMatrixB = new InmutableMatrix(3, 2, valuec);

        inmutableMatrixRes = inmutableMatrixA.add(inmutableMatrixA).sub(new
InmutableMatrix(2, 3, valueb)).multiply(inmutableMatrixB);

        mutableMatrixA = new MutableMatrix(2, 3, valuea);
        mutableMatrixB = new MutableMatrix(3, 2, valuec);

        mutableMatrixA.add(mutableMatrixA).sub(new MutableMatrix(2, 3,
valueb)).multiply(mutableMatrixB);

        /*
         * 预期结果
```

```
         * -32 -44
         * -5 -8
        */
        System.out.println("InmutableMatrix Chained Operations Result:");
        inmutableMatrixRes.print();
        System.out.println("MutableMatrix Chained Operations Result:");
        mutableMatrixA.print();
```

输出：

```
InmutableMatrix Chained Operations Result:
-32 -44
-5 -8
MutableMatrix Chained Operations Result:
-32 -44
-5 -8
```

相互构造的测试

```
        // 相互构造测试
        inmutableMatrixA = new InmutableMatrix(new MutableMatrix(2, 3, valuea));
        mutableMatrixA = new MutableMatrix(new InmutableMatrix(2, 3, valuea));

        /**
         * 预期结果
         * 1 2 3
         * 4 5 6
         */
        System.out.println("InmutableMatrix create Result:");
        inmutableMatrixA.print();
        System.out.println("MutableMatrix create Result:");
        mutableMatrixA.print();
```

输出：

```
InmutableMatrix create Result:
1 2 3
4 5 6
MutableMatrix create Result:
1 2 3
4 5 6
```

抛出异常的测试

```
        // 不同大小的矩阵执行加法，应该抛出异常
        try {
            inmutableMatrixA = new InmutableMatrix(2, 3, valuea);
            inmutableMatrixB = new InmutableMatrix(3, 2, valuec);
            inmutableMatrixRes = inmutableMatrixA.add(inmutableMatrixB);
        } catch (Exception e) {
            System.out.println(e.getMessage());
```

```
        }

        try {
            mutableMatrixA = new MutableMatrix(2, 3, valuea);
            mutableMatrixB = new MutableMatrix(3, 2, valuec);
            mutableMatrixA.add(mutableMatrixB);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        // 不同大小的矩阵执行减法，应该抛出异常
        try {
            inmutableMatrixA = new InmutableMatrix(2, 3, valuea);
            inmutableMatrixB = new InmutableMatrix(3, 2, valuec);
            inmutableMatrixRes = inmutableMatrixA.sub(inmutableMatrixB);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        try {
            mutableMatrixA = new MutableMatrix(2, 3, valuea);
            mutableMatrixB = new MutableMatrix(3, 2, valuec);
            mutableMatrixA.sub(mutableMatrixB);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        // 不匹配的矩阵执行乘法，应该抛出异常
        try {
            inmutableMatrixA = new InmutableMatrix(2, 3, valuea);
            inmutableMatrixRes = inmutableMatrixA.multiply(inmutableMatrixA);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        try {
            mutableMatrixA = new MutableMatrix(2, 3, valuea);
            mutableMatrixA.multiply(mutableMatrixA);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
```

输出：

```
The size of both mutrix should be identical!
The size of both mutrix should be identical!
The size of both mutrix should be identical!
The size of both mutrix should be identical!
the column of the first matrix should be the same as the row of the second!
the column of the first matrix should be the same as the row of the second!
```

性能测试

初始化：

```
        // 性能测试
        // 初始化随机矩阵
        int[][] values1 = new int[size][size];
        int[][] values2 = new int[size][size];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                values1[i][j] = (int) (Math.random() * randomValue);
                values2[i][j] = (int) (Math.random() * randomValue);
            }
        }
        // 创建 MutableMatrix 对象
        MutableMatrix mMatrix1 = new MutableMatrix(size, size, values1);
        MutableMatrix mMatrix2 = new MutableMatrix(size, size, values2);

        // 创建 InmutableMatrix 对象
        InmutableMatrix imMatrix1 = new InmutableMatrix(size, size, values1);
        InmutableMatrix imMatrix2 = new InmutableMatrix(size, size, values2);
```

MutableMatrix的性能测试

```
        private static void testOperations(InmutableMatrix matrix1,
InmutableMatrix matrix2) throws Exception {
        long startTime = System.nanoTime();
        matrix1.add(matrix2);
        long addEndTime = System.nanoTime();
        System.out.println("InmutableMatrix 加法时间:  " + (addEndTime - startTime)
+ " 纳秒");

        startTime = System.nanoTime();
        matrix1.sub(matrix2);
        long subEndTime = System.nanoTime();
        System.out.println("InmutableMatrix 减法时间:  " + (subEndTime - startTime)
+ " 纳秒");

        startTime = System.nanoTime();
        matrix1.scalarMultiply(randomValue);
        long scalarMultiplicationEndTime = System.nanoTime();
        System.out.println("InmutableMatrix 标量乘法时间:  " +
(scalarMultiplicationEndTime - startTime) + " 纳秒");

        startTime = System.nanoTime();
        matrix1.multiply(matrix2);
        long multiplicationEndTime = System.nanoTime();
        System.out.println("InmutableMatrix 矩阵乘法时间:  " +
(multiplicationEndTime - startTime) + " 纳秒");
    }
```

输出:

```
MutableMatrix 加法时间:  8200 纳秒
MutableMatrix 减法时间:  7300 纳秒
MutableMatrix 标量乘法时间:  5000 纳秒
MutableMatrix 矩阵乘法时间:  190200 纳秒
```

InmutableMatrix的性能测试

```
    private static void testOperations(MutableMatrix matrix1, MutableMatrix
matrix2) throws Exception {
        long startTime = System.nanoTime();
        matrix1.add(matrix2);
        long addEndTime = System.nanoTime();
        System.out.println("MutableMatrix 加法时间:  " + (addEndTime - startTime) +
" 纳秒");

        startTime = System.nanoTime();
        matrix1.sub(matrix2);
        long subEndTime = System.nanoTime();
        System.out.println("MutableMatrix 减法时间:  " + (subEndTime - startTime) +
" 纳秒");

        startTime = System.nanoTime();
        matrix1.scalarMultiply(randomValue);
        long scalarMultiplicationEndTime = System.nanoTime();
        System.out.println("MutableMatrix 标量乘法时间:  " +
(scalarMultiplicationEndTime - startTime) + " 纳秒");

        startTime = System.nanoTime();
        matrix1.multiply(matrix2);
        long multiplicationEndTime = System.nanoTime();
        System.out.println("MutableMatrix 矩阵乘法时间:  " + (multiplicationEndTime
- startTime) + " 纳秒");
    }
```

输出:

```
InmutableMatrix 加法时间:  17300 纳秒
InmutableMatrix 减法时间:  21600 纳秒
InmutableMatrix 标量乘法时间:  30800 纳秒
InmutableMatrix 矩阵乘法时间:  205800 纳秒
```

总结

可变矩阵和不可变矩阵的根本区别就在于能否改变成员变量的值。为了实现不可变的特性，我们用到了 `final` 关键字修饰类和成员变量来避免其被继承和初始化后被改动。同时，还在运算方法里用新创建的对象保存结果并返回、在对外提供的接口里，通过返回成员变量的深拷贝来避免成员变量受到修改。

以上两个类都通过功能测试，满足要求的所有功能。而在性能测试中，不可变矩阵要慢于可变矩阵，这是因为不可变矩阵对比可变矩阵有额外的创建新对象的开销，故在上面的性能测试中都慢于可变矩阵。