Java应用技术homework2

蔡佳伟 3220104519

一、寻找JDK库中的不变类（至少2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。

不变类是指一旦创建，其状态（成员变量的值）就无法被修改的类。不变类的优势在于它们天生具有线程安全性，可以在并发环境中使用，而不需要额外的同步措施。常见的不变类有：`java.lang.String`和`java.lang.Integer`。

1. `java.lang.String`

源码分析：

`String`类的源码在`java/lang/String.java`中，可以看到如下关键信息：

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

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

- **final修饰符**：`String`类被声明为`final`，这意味着不能继承它，因此无法通过子类来修改其行为。
- **final字段**：`value`和`hash`字段都被声明为`final`，意味着一旦通过构造函数初始化后，这些字段不能被修改。
- **只读设计**：字符串的每个字符都保存在`value`数组中，该数组也是`final`的，不允许修改。所有修改字符串的方法（如`substring`、`concat`等）都返回一个新的字符串对象，而不是修改原有字符串的内容。
- **哈希值缓存**：为了提高性能，`String`对象在第一次调用`hashCode()`时计算并缓存哈希值，并且不会随着对象状态的改变而变化。

2. `java.lang.Integer`

源码分析：

`Integer`类的源码在`java/lang/Integer.java`中，部分源码如下：

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

```
    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return value;
    }

    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
}
```

- **final修饰符**：`Integer`类也被声明为`final`，因此不能被继承。
- **final字段**：`value`字段被声明为`final`，意味着一旦对象创建后，`value`就不能被改变。
- **缓存机制**：`Integer`通过一个内部缓存（`IntegerCache`）来避免频繁创建新的对象。例如，对常见的小数值（如`-128`到`127`）会重用缓存中的对象，这也加强了不变性，因为缓存中的对象不能被修改。

不变类的共性如下：

1. **类被声明为`final`**：这样可以防止通过继承来破坏类的不可变性。子类不能修改父类的行为，也就保证了不变性。
2. **所有字段都使用`final`关键字**：关键的实例变量都被声明为`final`，保证了它们只能在构造函数中被赋值一次，且赋值后不能被修改。
3. **没有提供修改对象状态的方法**：不变类不会提供任何可以修改内部状态的方法，所有看似修改对象状态的方法，实际上都会返回一个新的对象。
4. **线程安全**：由于对象一旦创建，其内部状态就不会再改变，不变类天然具有线程安全性，可以在多个线程间共享，无需担心同步问题，从而在多线程环境下天然线程安全。
5. **缓存机制（可选）**：一些不变类（如`Integer`）使用了缓存机制来优化性能，避免不必要的对象创建，同时确保缓存中的对象是不可变的。

二、设计并实现类MutableMatrix和InmutableMatrix

- 体现可变类和不可变类的区别
- 具备矩阵的一般运算操作
- 提供MutableMatrix(InmutableMatrix)和 InmutableMatrix(MutableMatrix)构造函数
- 支持矩阵链式运算，如 – MutableMatrix m1, m2, m3; – ..... – m1.add(m2).add(m3)

(一)、代码实现

对于可变类与不可变类，不可变类我采用了`final`声明，类内的`data`变量也使用`final`修饰，这意味着不能继承，因而不能通过修改子类修改其数值。

在构造函数中，我使用`deepCopy()`构造可变矩阵和不可变矩阵，使用`deepCopy`构造函数的主要目的是确保矩阵数据的独立性，避免外部修改原始数据对矩阵对象内部状态产生影响。

此外，对于不可变矩阵，我不能允许任何外部代码修改矩阵的数据。如果矩阵对象直接引用外部传入的数据，那么外部代码可以直接修改原始数据，从而违反了不可变矩阵的设计原则。

通过在构造函数中使用 `deepCopy`，我将传入的数据复制到一个新的二维数组中，确保矩阵对象持有自己的数据副本，外部修改不会影响矩阵的内容。

```
private final int[][] data;

// Construct an immutable matrix using a 2D array
public InmutableMatrix(int[][] data) {
    this.data = deepCopy(data);
}

// Construct an immutable matrix from a mutable matrix
public InmutableMatrix(MutableMatrix mutableMatrix) {
    this(mutableMatrix.getData());
}
```

```
    // Construct a mutable matrix using a 2D array
    public MutableMatrix(int[][] data) {
        this.data = deepCopy(data);
    }

    // Construct a mutable matrix from an immutable matrix
    public MutableMatrix(InmutableMatrix inmutableMatrix) {
        this(inmutableMatrix.getData());
    }
```

为了实现矩阵的连加，连减，连乘操作，我重载了`add`等函数。链式调用的核心是每次方法调用都返回对象本身，这样可以继续调用后续的方法。如果 `add` 方法返回的是 `void` 类型，那么它就无法返回当前对象（`m1`），所以连加就无法实现。

为了支持链式调用，`add` 方法需要返回当前对象（例如返回 `this`）。通过重载 `add` 方法，允许我们不仅支持两个不可变矩阵的加法操作，还可以支持一个不可变矩阵与可变矩阵的加法操作。这样可以保证在使用不同类型的矩阵时，也能进行连加操作。

```
// Matrix addition, returns a new immutable matrix
public InmutableMatrix add(InmutableMatrix other) {
    int[][] result = new int[this.data.length][this.data[0].length];
    for (int i = 0; i < this.data.length; i++) {
        for (int j = 0; j < this.data[i].length; j++) {
            result[i][j] = this.data[i][j] + other.data[i][j];
        }
    }
    return new InmutableMatrix(result);  // Return a new immutable matrix
}
```

```
// Overload of add to support chain calls (supports mutable matrix as argument)
public InmutableMatrix add(MutableMatrix other) {
    return this.add(new InmutableMatrix(other));  // Call immutable matrix addition
}
```

(二)、测试

在代码测试中，我使用`public class MatrixTest`作为主类，构造了两个矩阵作为被操作的矩阵，然后使用构造函数构造了可变类和不可变类，分别进行操作。首先可以发现支持连加操作。由于要测试性能，我对比了两个类进行加法操作的时间，分别进行操作10000次。可以发现可变类所花费的时间远远小于不可变类，这可能是因为可变类不需要为每个操作创建新对象，减少了对象创建，内存分配和回收的开销。对于不可变类，每个操作都会创建一个新的矩阵实例。

```
public class MatrixTest {
    public static void main(String[] args) {
        // Create two mutable matrices
        int[][] data1 = {
            {1, 2},
            {3, 4}
        };
        int[][] data2 = {
            {5, 6},
            {7, 8}
        };

        MutableMatrix m1 = new MutableMatrix(data1);
        MutableMatrix m2 = new MutableMatrix(data2);

        // Create immutable matrices
        InmutableMatrix im1 = new InmutableMatrix(data1);
        InmutableMatrix im2 = new InmutableMatrix(data2);

        // Time the addition of mutable matrices
        int iterations = 10000;
        System.out.println("Testing MutableMatrix addition performance...");
        long startTime = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            m1.add(m2);  // Chain operation, modifies m1
        }
        long endTime = System.nanoTime();
        long durationMutable = endTime - startTime;
        System.out.println("Time taken for " + iterations + " mutable matrix additions: " + durationMutable  + " ms");
```

```
        // Time the addition of immutable matrices
        System.out.println("Testing InmutableMatrix addition performance...");
        startTime = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            im1.add(im2);  // Chain operation, returns a new matrix
        }
        endTime = System.nanoTime();
        long durationImmutable = endTime - startTime;
        System.out.println("Time taken for " + iterations + " immutable matrix additions: " + durationImmutable  + " ms");

        System.out.println("\nMutableMatrix m1 after additions:");
        m1.printMatrix();
        System.out.println("\nInmutableMatrix im3 after additions:");
        im1.add(im2).printMatrix();
    }
}
```

```
PS E:\vsjava> java MatrixTest
[11, 14]
[17, 20]
[11, 14]
[17, 20]
Testing MutableMatrix addition performance...
Time taken for 10000 mutable matrix additions: 502600 ms
Testing InmutableMatrix addition performance...
Time taken for 10000 immutable matrix additions: 4430500 ms

MutableMatrix m1 after additions:
[50011, 60014]
[70017, 80020]

InmutableMatrix im3 after additions:
[6, 8]
[10, 12]
```

在以下的测试操作中，我测试了矩阵的减法和乘法，结果也是正确的。

```
        MutableMatrix m1 = new MutableMatrix(data1);
        MutableMatrix m2 = new MutableMatrix(data2);
        m1.multiply(m2).printMatrix();
        m1.subtract(m2).printMatrix();
```

```
PS E:\vsjava> java MatrixTest
[19, 22]
[43, 50]
[14, 16]
[36, 42]
[11, 14]
[17, 20]
```