1. 不变类

1. `String`类

源码分析：

Java中的`String`类是最常见的不可变类。我们可以通过阅读`String`类的源码来理解它为什么是不可变的。

```java
public final class String implements java.io.Serializable, Comparable<String>,
CharSequence {
    // 内部使用 final 关键字修饰的 char 数组
    private final char value[];

    private int hash; // 缓存hash值

    public String(String original) {
        this.value = original.value;
        this.hash = 0;
    }

    // 不提供任何修改value[]的方法
    public char charAt(int index) {
        if (index < 0 || index >= value.length) {
            throw new StringIndexOutOfBoundsException(index);
        }
        return value[index];
    }

    public String concat(String str) {
        int otherLen = str.length();
        if (otherLen == 0) {
            return this;
        }
        char buf[] = Arrays.copyOf(value, value.length + otherLen);
        str.getChars(buf, value.length);
        return new String(buf, true);
    }
}
```

不变性的原因：

1. **`final`关键字**：`String`类本身被声明为`final`，这意味着它不能被继承，从而防止了子类破坏它的不可变性。
2. **`final`字段**：`String`类内部的`value`数组是一个`final`修饰的`char[]`，表示该引用一旦被初始化就不能被指向其他数组，确保了字符串内容的不可变性。

3. **没有提供修改状态的方法**：`String`类没有任何方法可以修改`value[]`中的字符内容。即使我们调用`concat()`方法来拼接字符串，它实际上会返回一个新的`String`对象，而不是在原来的`String`对象上修改。
4. **缓存`hash`值**：`hash`值也被缓存起来，并且不会改变。因此，`String`的哈希值在创建后也不会变，确保在集合中使用时的稳定性。

2. `Integer`类

源码分析：

`Integer`类是基本类型`int`的包装类，它也是不可变的。我们来看一下`Integer`类的实现。

```java
public final class Integer extends Number implements Comparable<Integer> {
    // 使用 final 关键字修饰的 int 值
    private final int value;

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

    public int intValue() {
        return value;
    }

    // 提供的转换方法，返回新的对象
    public Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high) {
            return IntegerCache.cache[i + (-IntegerCache.low)];
        }
        return new Integer(i);
    }

    public boolean equals(Object obj) {
        if (obj instanceof Integer) {
            return value == ((Integer) obj).intValue();
        }
        return false;
    }
}
```

不变性的原因：

1. **`final`关键字**：与`String`类似，`Integer`类也被声明为`final`，使得它不能被继承，从而避免了子类破坏它的不可变性。
2. **`final`字段**：`Integer`类内部的`value`字段是`final`的`int`类型变量，表示该值在对象创建后就不能再被改变。
3. **没有提供修改方法**：`Integer`类没有提供任何能够修改`value`字段的方法。所有对整数值的操作（如`valueOf()`方法）都返回一个新的`Integer`对象，而不会改变现有的对象。

4. **缓存机制**：`Integer`类中提供了`IntegerCache`，通过缓存常用的`Integer`对象（默认是`-128`到`127`），避免频繁创建相同值的对象，从而提高了性能和节省了内存。

结果

不变类的共性：

通过分析`String`和`Integer`类的源码，可以总结出不变类的几个共性：

1. **类声明为`final`**：不变类通常被声明为`final`，从而防止子类继承和修改它们的行为。
2. **字段声明为`final`**：类中的所有关键字段（特别是保存状态的字段）通常都被声明为`final`，以确保它们在对象创建后不能被修改。
3. **没有提供修改状态的方法**：不变类不会提供任何可以直接或间接修改对象状态的方法。如需要更改状态，通常会创建一个新的对象而不是修改现有对象。
4. **线程安全**：由于不变类的状态一旦创建就无法改变，因此它们天然是线程安全的，多个线程可以安全地共享相同的不可变对象。
5. **可重用性**：不变类的对象可以在不同上下文中安全地复用，例如`Integer`类使用缓存来复用常见的整数对象。

2. 设计并实现类MutableMatrix和InmutableMatrix

我们将设计两个类：`MutableMatrix`（可变类）和`InmutableMatrix`（不可变类）。它们将支持矩阵的基本操作，并体现可变类和不可变类的区别。设计的重点在于：

1. **可变类**和**不可变类**的设计：
   - `MutableMatrix`允许在现有对象上进行操作并修改其状态。
   - `InmutableMatrix`每次操作都会返回一个新的矩阵对象，而不会改变现有对象的状态。
2. **矩阵基本操作**：加法、减法、乘法、转置等。
3. **支持链式操作**：支持类似`m1.add(m2).add(m3)`的链式操作。
4. **相互转换**：`MutableMatrix`可以通过构造函数转换为`InmutableMatrix`，反之亦然。

代码实现

1. `Matrix`基础类

首先，定义一个基础类`Matrix`，用于存储矩阵的数据及一些辅助方法。

JAVA

```java
public abstract class Matrix {
    protected final int rows;
    protected final int cols;
    protected final double[][] data;

    // 构造函数
    public Matrix(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        this.data = new double[rows][cols];
    }

    public Matrix(double[][] data) {
        this.rows = data.length;
        this.cols = data[0].length;
        this.data = new double[rows][cols];
        for (int i = 0; i < rows; i++) {
            System.arraycopy(data[i], 0, this.data[i], 0, cols);
        }
    }

    // 获取矩阵的行数
    public int getRows() {
        return rows;
    }

    // 获取矩阵的列数
    public int getCols() {
        return cols;
    }

    // 打印矩阵
    public void print() {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(data[i][j] + " ");
            }
            System.out.println();
        }
    }

    // 校验两个矩阵是否具有相同的尺寸
    protected void checkSameDimension(Matrix other) {
        if (this.rows != other.rows || this.cols != other.cols) {
            throw new IllegalArgumentException("Matrices dimensions do not match.");
        }
```

```
        }
    }
```

2. `MutableMatrix` 类

`MutableMatrix` 是可变类，可以直接修改矩阵的值。

JAVA

```java
public class MutableMatrix extends Matrix {

    // 构造函数
    public MutableMatrix(int rows, int cols) {
        super(rows, cols);
    }

    public MutableMatrix(double[][] data) {
        super(data);
    }

    public MutableMatrix(InmutableMatrix immutableMatrix) {
        super(immutableMatrix.data);
    }

    // 矩阵加法，返回自身以支持链式调用
    public MutableMatrix add(MutableMatrix other) {
        checkSameDimension(other);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                this.data[i][j] += other.data[i][j];
            }
        }
        return this;
    }

    // 矩阵减法，返回自身以支持链式调用
    public MutableMatrix subtract(MutableMatrix other) {
        checkSameDimension(other);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                this.data[i][j] -= other.data[i][j];
            }
        }
        return this;
    }

    // 矩阵乘法，返回自身以支持链式调用
    public MutableMatrix multiply(double scalar) {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                this.data[i][j] *= scalar;
            }
        }
        return this;
    }
```

```
    // 转置矩阵，修改自身并返回自身
    public MutableMatrix transpose() {
        double[][] transposed = new double[cols][rows];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                transposed[j][i] = this.data[i][j];
            }
        }
        this.data = transposed;
        return this;
    }

    // 将 MutableMatrix 转为不可变类 InmutableMatrix
    public InmutableMatrix toInmutableMatrix() {
        return new InmutableMatrix(this);
    }
}
```

3. `InmutableMatrix` 类

`InmutableMatrix`是不可变类，所有操作都会返回一个新的矩阵对象，而不会修改当前矩阵的状态。

JAVA

```java
public final class InmutableMatrix extends Matrix {

    // 构造函数
    public InmutableMatrix(int rows, int cols) {
        super(rows, cols);
    }

    public InmutableMatrix(double[][] data) {
        super(data);
    }

    public InmutableMatrix(MutableMatrix mutableMatrix) {
        super(mutableMatrix.data);
    }

    // 矩阵加法，返回新的不可变矩阵
    public InmutableMatrix add(InmutableMatrix other) {
        checkSameDimension(other);
        double[][] result = new double[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = this.data[i][j] + other.data[i][j];
            }
        }
        return new InmutableMatrix(result);
    }

    // 矩阵减法，返回新的不可变矩阵
    public InmutableMatrix subtract(InmutableMatrix other) {
        checkSameDimension(other);
        double[][] result = new double[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = this.data[i][j] - other.data[i][j];
            }
        }
        return new InmutableMatrix(result);
    }

    // 矩阵乘法，返回新的不可变矩阵
    public InmutableMatrix multiply(double scalar) {
        double[][] result = new double[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = this.data[i][j] * scalar;
            }
```

```
        }
        return new InmutableMatrix(result);
    }

    // 矩阵转置，返回新的不可变矩阵
    public InmutableMatrix transpose() {
        double[][] transposed = new double[cols][rows];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                transposed[j][i] = this.data[i][j];
            }
        }
        return new InmutableMatrix(transposed);
    }

    // 将 InmutableMatrix 转为可变类 MutableMatrix
    public MutableMatrix toMutableMatrix() {
        return new MutableMatrix(this);
    }
}
```

4. 测试代码

JAVA

```java
import java.util.Random;

public class MatrixTest {

    public static void main(String[] args) {
        // 增大运算量，使用 1000 x 1000 的矩阵
        int size = 1000;
        double[][] data1 = generateRandomMatrix(size, size);
        double[][] data2 = generateRandomMatrix(size, size);

        MutableMatrix m1 = new MutableMatrix(data1);
        MutableMatrix m2 = new MutableMatrix(data2);
        InmutableMatrix im1 = new InmutableMatrix(data1);
        InmutableMatrix im2 = new InmutableMatrix(data2);

        // 测试 MutableMatrix
        long startMutable = System.nanoTime();
        m1.add(m2).multiply(1.5).subtract(m2).transpose();
        long endMutable = System.nanoTime();
        System.out.println("MutableMatrix 运算时间: " + (endMutable - startMutable) /
1_000_000.0 + " 毫秒");

        // 测试 InmutableMatrix
        long startInmutable = System.nanoTime();
        InmutableMatrix result =
im1.add(im2).multiply(1.5).subtract(im2).transpose();
        long endInmutable = System.nanoTime();
        System.out.println("InmutableMatrix 运算时间: " + (endInmutable -
startInmutable) / 1_000_000.0 + " 毫秒");
    }

    // 随机矩阵生成器
    private static double[][] generateRandomMatrix(int rows, int cols) {
        Random rand = new Random();
        double[][] matrix = new double[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = rand.nextDouble() * 100;
            }
        }
        return matrix;
    }
```

```
    }
```

- `generateRandomMatrix` **方法**：生成一个随机的 `size x size` 矩阵，用于测试。
- **计时功能**：通过 `System.nanoTime()` 来记录`MutableMatrix`和`InmutableMatrix`的运算时间。
- **测试矩阵运算**：使用 1000x1000 的矩阵进行加法、乘法、减法和转置运算，并记录运行时间。

性能分析和说明

运行测试类，得到结果如下：

```
C:\Users\Frey\.jdks\openjdk-21\bin\java.exe
MutableMatrix 运算时间: 44.2623 毫秒
InmutableMatrix 运算时间: 88.5472 毫秒
```

1. **可变类 (`MutableMatrix`)**：对于需要频繁修改的矩阵运算，`MutableMatrix`更高效，因为它直接在原有对象上进行操作，减少了对象的创建和内存的分配。但由于对象状态可能被修改，因此需要在并发场景中注意线程安全性问题。

2. **不可变类 (`InmutableMatrix`)**：每次运算都会创建新的矩阵对象，因而在高频率操作下性能会相对较低。然而，不可变类天然是线程安全的，适合并发场景中使用，并且能够避免由于修改共享对象导致的不可预见错误。