🛸 时空通信矩阵破译

📜 目录

📜 梳理题目要求

题目要求的本质其实是 **实现矩阵乘法计算器**，只不过这里的矩阵每一个位置**有两个元素**（**极化值** 和 **相位标识**）

题目规定了 **极化值** 和 **相位标识** 的计算方法，并且需要 **面向对象编程** 实现

📜 思路

1. 既然是一个 **类矩阵乘法** 计算，那么计算逻辑和线性代数领域的矩阵乘法是相同的，只需要模拟手工计算的逻辑进行编程即可

2. 题目要求 **面向对象** 编程（什么是面向对象编程？）

3. 首先拿到两个矩阵，应当首先判断能否进行矩阵乘法，如果不行则 **抛出异常** （如何抛出异常？）

4. 按照矩阵乘法的规则进行模拟运算，结果保存在一定的数据结构里

📜相关知识学习

1. 面向对象编程

面向过程（Procedure Oriented 简称PO：如C语言）：

从名字可以看出它是注重过程的。当解决一个问题的时候，面向过程会把事情拆分成：一个个函数和数据（用于方法的参数）。然后按照一定的顺序，执行完这些**方法**（每个方法看作一个过程），等方法执行完了，事情就搞定了。

面向对象（Object Oriented简称OO：如C++，JAVA等语言）：

看名字它是注重对象的。当解决一个问题的时候，面向对象会把事物抽象成对象的概念，就是说这个问题里面有哪些对象，然后给对象赋一些属性和方法，然后让每个对象去执行自己的方法，问题得到解决。

三：面向过程与面向对象的优缺点：

小编在看了@十四期_李光✦ 博客上解释的非常通俗易懂，分享给大家

用面向过程的方法写出来的程序是一份蛋炒饭，而用面向对象写出来的程序是一份盖浇饭。

蛋炒饭制作的细节不说了，最后的一道工序肯定是把米饭和鸡蛋混在一起炒匀。盖浇饭呢，则是把米饭和盖菜分别做好，你如果要一份红烧肉盖饭呢，就给你浇一份红烧肉；如果要一份青椒土豆盖浇饭，就给浇一份青椒土豆丝。

蛋炒饭的好处就是入味均匀，吃起来香。如果恰巧你不爱吃鸡蛋，只爱吃青菜的话，那么唯一的办法就是全部倒掉，重新做一份青菜炒饭了。盖浇饭就没这么多麻烦，你只需要把上面的盖菜拨掉，更换一份盖菜就可以了。盖浇饭的缺点是入味不均，可能没有蛋炒饭那么香。

到底是蛋炒饭好还是盖浇饭好呢？其实这类问题都很难回答，非要比个上下高低的话，就必须设定一个场景，否则只能说是各有所长。如果大家都不是美食家，没那么多讲究，那么从饭馆角度来讲的话，做盖浇饭显然比蛋炒饭更有优势，他可以组合出来任意多的组合，而且不会浪费。

盖浇饭的好处就是 ‘’ 菜” “饭”分离，从而提高了制作盖浇饭的灵活性。饭不满意就换饭，菜不满意换菜。用软件工程的专业术语就是”可维护性“比较好，” 饭” 和”菜”的耦合度比较低。蛋炒饭将”蛋” “饭”搅和在一起，想换”蛋” “饭”中任何一种都很困难，耦合度很高，以至于”可维护性”比较差。软件工程追求的目标之一就是可维护性✦，可维护性主要表现在3个方面：可理解性、可测试性和可修改性。面向对象的好处之一就是显著的改善了软件系统的可维护性。

四、面向对象编程✦的特性

三大基本特性：封装，继承，多态

封装

封装，就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

继承

继承，指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。 通过继承创建的新类称为“子类”或“派生类✦”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过 “继承”（Inheritance）和“组合” （Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用父类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力。

多态

多态，是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

面向对象编程 是一种编程方式，它把 **数据** 和 **操作数据的方法** 组织在一起，形成“对象”，就像现实世界中的事物一样。

可以把“对象”想象成一个“东西”，它既有“属性”（特征），也能执行“动作”（行为）。

举个例子:
现实世界中的对象：一只小狗

```
属性（特征）：
品种：柯基
颜色：棕色
年龄：2 岁
行为（动作）：
吃饭
跑步
摇尾巴
```

于是可以用“类（Class）”来创建这样的对象：

```python
class Dog:  # 定义一个“狗”的类
    def __init__(self, breed, color, age):  # 初始化狗的属性
        self.breed = breed  # 品种
        self.color = color  # 颜色
        self.age = age  # 年龄

    def bark(self):  # 定义狗的行为
        print("汪汪！")  # 狗会叫

# 创建一只小狗（对象）
my_dog = Dog("柯基", "棕色", 2)

# 访问属性
print(my_dog.breed)  # 输出：柯基

# 调用行为
my_dog.bark()  # 输出：汪汪！
```

2. python 实现异常抛出处理

异常 就是程序运行过程中发生的错误，如果不处理，程序就会崩溃。
抛出异常 就是当程序发现问题时，主动报错，并提供错误信息。

Python 提供 `raise` 语句，让我们可以主动抛出错误，终止程序，并显示错误信息：

```python
raise 异常类型("错误信息")
```

示例：

```
raise ValueError("这里有个错误！")
```

输出：

```
ValueError: 这里有个错误！
```

📜 流程记录

1. 定义异常类 `QuantumDimensionError` ，维度不匹配时抛出

在本问题中，矩阵维度不匹配时候，想报 `QuantumDimensionError` 而不是普通的 `ValueError` ，又无法在 python **内置的** 异常找到合适的，这样能提供更清晰的错误信息

在 Python 中，自定义异常只需要创建一个新的异常类，继承 Python 的 `Exception` ， 在需要报错的地方用 `raise` 抛出异常即可

```
class QuantumDimensionError(Exception):
    # 自定义异常类：维度不匹配时抛出
    pass
```

2. 定义 `QuantumMatrix` 类，用于数据的保存以及矩阵乘法运算

```
class QuantumMatrix:
    def __init__(self, matrix):
        """
        初始化量子矩阵，格式应为：
        [[(数值，字符)，(数值，字符)]，
         [(数值，字符)，(数值，字符)]]
        """
        self.matrix = matrix
        self.rows = len(matrix)
        self.cols = len(matrix[0])

    def multiply(self, other: "QuantumMatrix") -> "QuantumMatrix":
        """执行矩阵乘法"""
        if self.cols != other.rows:
            raise QuantumDimensionError("矩阵维度不匹配，无法相乘")

        # 计算结果矩阵的大小 (self.rows x other.cols)
        result = [[(0, ' ') for _ in range(other.cols)] for _ in range(self.rows)]

        for i in range(self.rows):  # 遍历 A 的行
            for j in range(other.cols):  # 遍历 B 的列
                # 保存二元组的数据信息
                sum_value = 0
                max_char = 0
                for k in range(self.cols):  # 遍历 A 的列 / B 的行
                    a_value, a_phase = self.matrix[i][k]
                    b_value, b_phase = other.matrix[k][j]

                    # 极化值和相位值计算
                    sum_value += a_value * b_value
                    max_char = max(max(ord(a_phase), ord(b_phase)), max_char)

                result[i][j] = (sum_value, chr(max_char))

        return QuantumMatrix(result)

    def __str__(self):
        # 格式化输出矩阵
        return "\n".join(str(row) for row in self.matrix)
```

3. 主函数进行测试

首先，构造两个 `QuantumMatrix` 实例，分别表示两个量子矩阵。然后，调用 `matrix_a.multiply(matrix_b)` 方法进行矩阵乘法运算，并将计算结果保存在 `result` 中，最后以格式化方式输出。

```
if __name__ == '__main__':
    # 测试用例
    matrix_a = QuantumMatrix([
        [(1, 'a'), (2, 'b'), (3, 'b')],
        [(4, 'y'), (5, '6'), (6, 'b')]
    ])

    matrix_b = QuantumMatrix([
        [(-1, 'z'), (0, 'x')],
        [(0, 'm'), (1, 'n')],
        [(1, 'm'), (2, 'n')]
    ])

    result = matrix_a.multiply(matrix_b)
    print("相乘结果: ")
    print(result)
```

📜运行成功截图

输入用例1：

```
    matrix_a = QuantumMatrix([
        [(1, 'a'), (2, 'b'), (3, 'b')],
        [(4, 'y'), (5, '6'), (6, 'b')]
    ])

    matrix_b = QuantumMatrix([
        [(-1, 'z'), (0, 'x')],
        [(0, 'm'), (1, 'n')],
        [(1, 'm'), (2, 'n')]
    ])
```

输出结果1：

```
C:\Users\zhang\miniconda3\python.exe D:\lianshidaiRecurit\02\main.py
相乘结果：
[(2, 'z'), (8, 'x')]
[(2, 'z'), (17, 'y')]

进程已结束，退出代码为 0
```

通过纸笔演算，程序输出结果无误，说明在正确的输入格式下能正常运行计算

输入用例2：

```
matrix_a = QuantumMatrix([
    [(1, 'a'), (2, 'b')],
    [(4, 'y'), (5, '6')]
])

matrix_b = QuantumMatrix([
    [(-1, 'z'), (0, 'x')],
    [(0, 'm'), (1, 'n')],
    [(1, 'm'), (2, 'n')]
])
```

输出结果2：

```
C:\Users\zhang\miniconda3\python.exe D:\lianshidaiRecurit\02\main.py
Traceback (most recent call last):  使用 AI 解释
  File "D:\lianshidaiRecurit\02\main.py", line 57, in <module>
    result = matrix_a.multiply(matrix_b)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "D:\lianshidaiRecurit\02\main.py", line 19, in multiply
    raise QuantumDimensionError("矩阵维度不匹配，无法相乘")
QuantumDimensionError: 矩阵维度不匹配，无法相乘

进程已结束，退出代码为 1
```

结果说明，在错误的输入格式下，程序成功抛出了预期的错误