第一次作业思路分析

by 22373337 张奕彤

目录

- 作业任务分析
- 代码架构分析
- 测评机搭建方法
- 深浅克隆介绍

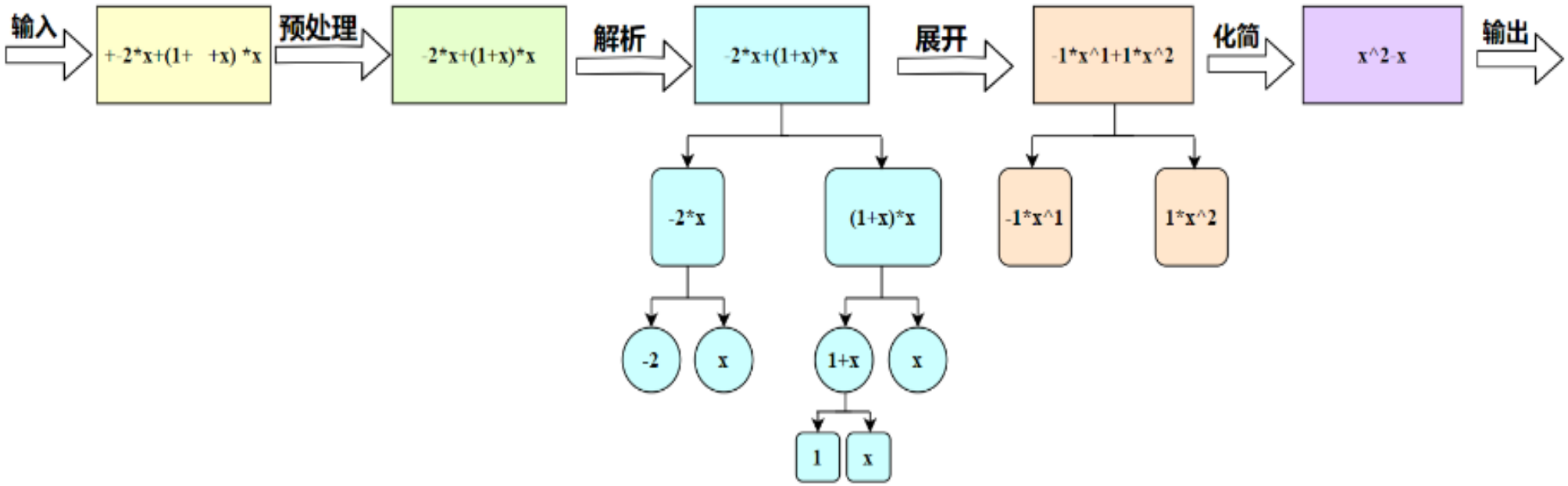
第一部分：作业任务简析

作业概括：**展开**一个最多**嵌套一层括号**的**单变量**多项式，多项式中项的因子只能是常数因子、幂函数因子、表达式因子，最后在正确展开的前提下再比较表达式的**长度**。

从中我们可以比较容易地意识到如果我们完成如下任务其实就能顺利完成本次作业：

- 分析题干数据结构并**建类**
- **预处理**输入的字符串，得到方便后续操作的字符串
- **解析**预处理后的字符串
- **展开**有括号的表达式，得到一个多项式
- **化简**多项式并输出

一个简单的例子：

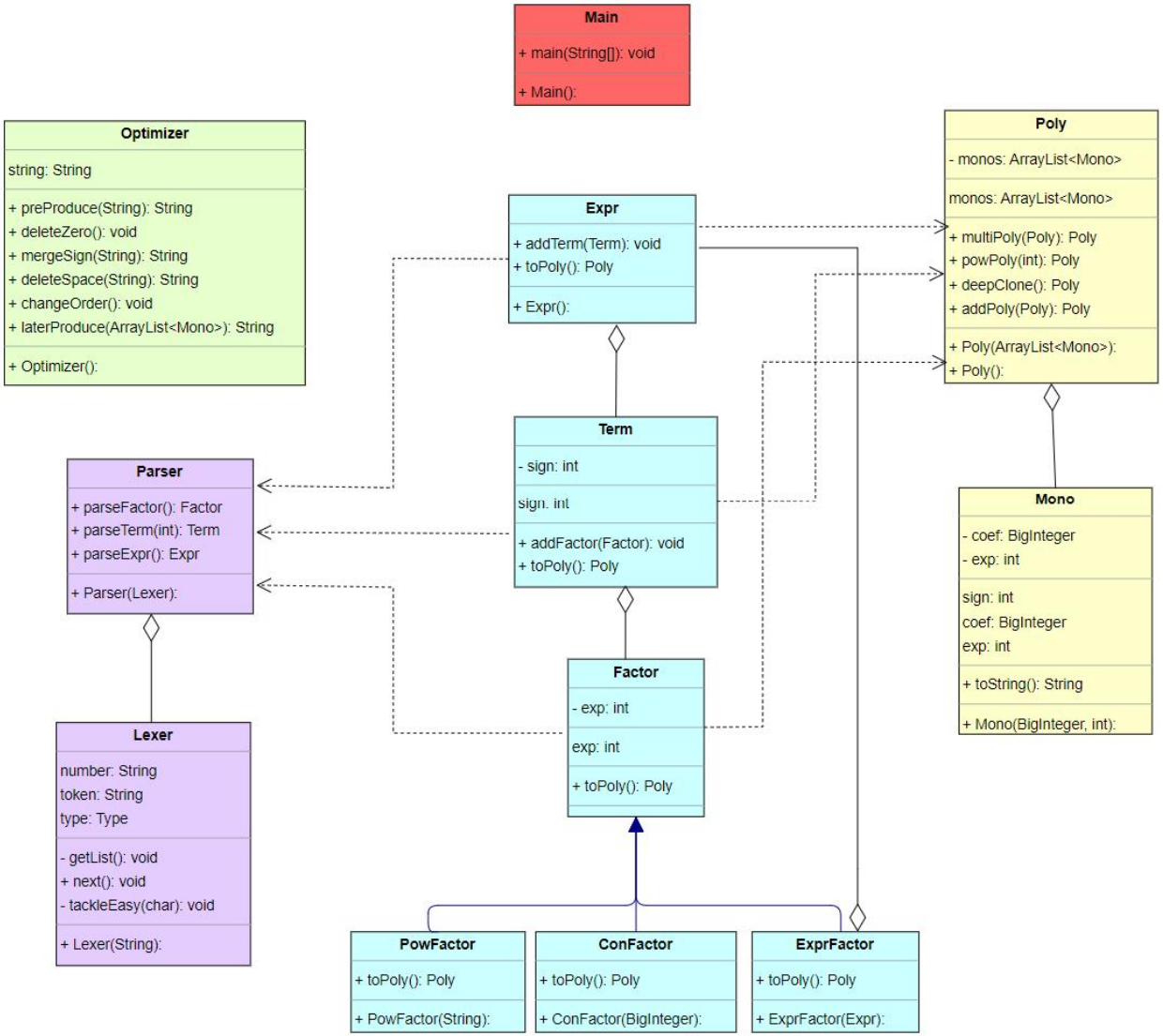


问题的重点：

解析字符串 + 展开

第二部分：代码架构分析

UML类图



建类

分析题目输入

题目的输入有**表达式-->项-->因子**的层次化结构

分析目标输出

题目要求我们输出一个去除括号的表达式，实际上就是若干形如a*x^b的项的和或者差

预处理

- 删除空白字符
- 合并连续符号

- **为什么要**预处理？
  - 方便后续操作，减少后续操作的逻辑复杂度；
  - 减少后续空间与时间的开销。
- **为什么能**预处理？
  - 题目中限制了输入一定合法，无需做出额外判断；
  - 存在一些操作不会损耗输入字符串的任何信息。

表达式解析——递归下降法

递归下降法为什么比正则表达式解析的方法好呢？

- 扩展性
- **debug**

一些特殊处理：

- 在词法分析中去除前导0
- 将+与-分成 加减运算符 和 常数因子符号 两个类别
- 由Term类中的sign去管理运算符

```
private String getNumber() {
    StringBuilder sb = new StringBuilder();
    while (iter < input.length() && Character.isDigit(input.charAt(iter))) {
        sb.append(input.charAt(iter));
        ++iter;
    }
    String str = sb.toString();
    str = str.replaceFirst( regex: "^0+(?!$)", replacement: "");
    return str;
}
```

```
while (lexer.getType() == Lexer.Type.ADD || lexer.getType() == Lexer.Type.SUB) {
    if (lexer.getType() == Lexer.Type.ADD) {
        lexer.next();
        expr.addTerm(parseTerm( sign: 1));
    }
    else {
        lexer.next();
        expr.addTerm(parseTerm( sign: -1));
    }
}
```

```
else if (ch == '+') {
    int size = typeList.size();
    if (size > 0) {
        Type topType = typeList.get(size - 1);
        if (topType.equals(Type.MULTI) || topType.equals(Type.EXP)) {
            //todo
        }
        else {
            //todo
        }
    }
    else {
            //todo
    }
}
else if (ch == '-') {
    int size = typeList.size();
    if (size > 0 && typeList.get(size - 1).equals(Type.MULTI)) {
            //todo
    }
    else {
            //todo
    }
}
```

展开并计算表达式

在所有管理数据的类（Expr、Term、ConFactor、PowFactor、ExprFactor）中均实现toPoly（）方法，将其转化为题目要求输出的表达式

- 对于`ConFactor`与`PowFactor`，转化为仅含有一个`Mono`对象的`Poly`；
- 对于`ExprFactor`,我们假设此时已经实现了`expr.toPoly()`，那么可以先调用`expr.toPoly()`得到一个标准的多项式，然后再求这个多项式的幂；
- 对于`Term`,仅需将`Factor`调用`toPoly()`的结果乘起来即可；
- 对于`Expr`,仅需将`Term`调用`toPoly()`的结果加起来即可。

```
public Poly toPoly() {
    Mono mono = new Mono(BigInteger.ONE,this.getExp());
    ArrayList<Mono> monos = new ArrayList<>();
    monos.add(mono);
    return new Poly(monos);
}
```

```
public Poly toPoly() {
    Poly poly = new Poly();
    for (Term term : terms) {
        poly = poly.addPoly(term.toPoly());
    }
    return poly;
```

新的问题：
应当如何合理地实现多项式Poly间的运算？

解决这个问题时需要特别注意什么？

- **合并同类项**，以免运行时TLE或者MLE
- **实现深克隆**，以防后续迭代时出现意想不到的bug

在Poly类中定义容器**HashMap** monos，存储多项式中的标准单项式。monos的key存储因子，value存储单项式的系数，可以充要地表示表达式的所有信息。

- 多项式与单项式 的 **加法**

  向一个多项式加入一个单项式，有且仅有两种情况：

  - poly中不含单项式的指数——直接put该单项式的<exp,coef>
  - poly中含有单项式的指数——将value更新为旧值与coef之和

- 多项式与多项式 的 **加法**
  - 返回一个新实例多项式，为传入的两个多项式之和；
  - 重复调用单项式加法即可实现。

```
public static Poly addPoly(Poly poly1, Poly poly2) {
    Poly poly = new Poly();

    for (Integer exp: poly1.getMonoMap().keySet()) {
        poly.addMono(new Mono(poly1.getMonoMap().get(exp).getCoe(), exp));
    }
```

- 多项式与多项式 的 **乘法**
  - 返回一个新实例多项式，为传入的两个多项式之积；
  - 嵌套for循环，计算Poly1中的任意单项式与Poly2中的任意单项式之积，并将其加入新的实例多项式。

- 多项式 的 **幂**
  - 返回一个新实例多项式，为传入的多项式的n次方；
  - 看作仅有一个系数为1指数为0的单项式组成的多项式与传入的多项式乘次，然后调用n次多项式与多项式的乘法。（可以应用快速幂）

化简得到的多项式

如果上述过程没有bug的话，那么我们现在会得到一个符合如下格式的多项式$\sum_{i=1}^{n}$coef$_i$*x*^exp$_i$，并且其中的指数不会相同（因为在之前就已经实现了**同类项合并**）

优化时我主要考虑了如下几点：

- 指数或系数取特殊数字的情况
  - 若coef$_i$为1，则化为x*^exp$_i$
  - 若coef$_i$为-1，则化为-x*^exp$_i$
  - 若exp$_i$为0，则化为coef$_i$
  - 若exp$_i$为1，则化为coef$_i$*x
- 删除coef$_i$为0的项，当`Poly`中`monos`被删空时（即表达式本身是0的情况），向其中加入一个0
- 由于会出现`-x+1`比`1-x`长的情况，因此这一步需要改变输出顺序使得第一项尽可能为正

```
public void changeOrder() {
    for (int i = 0; i < monos.size(); i++) {
        if (monos.get(i).getCoef().toString().charAt(0) != '-') {
            Collections.swap(monos, i: 0, i);
            break;
        }
    }
}
```

第三部分：评测机搭建方法介绍

基本思路：

- 通过翻译指导书上的形式化定义构建数据生成器
- 通过使用python的sympy包来进行正确性检验

- 将可嵌套层数deepNum作为参数传入了genDate(),每当调用一次getExprFactor()时传入deepNum-1
- 加入常量池，增加测试的强度

```
expPool = [0, 0, 0, 1, 1, 1, 2, 3, 4, 5, 6, 7, 7, 7, 8, 8, 8]
numPool = [0, 0, 1, 1, 2, 3, 114514, 2147483647, 2147483648, 2147483649, 4294967296]
```

- 根据最终生成字符串的长度来决定数据是否有效，以此控制生成数据的复杂度

```
if (len(string) > 1000):
    return genDate(deepNum)
```

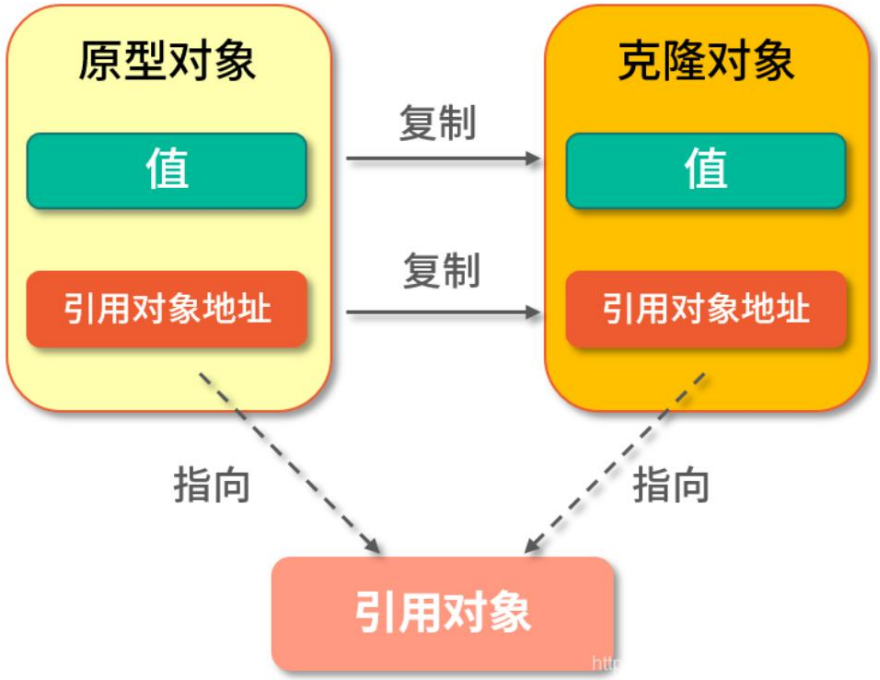
- 比较同时加入评测的多个人jar输出结果的长度，以辅助找到更好的优化方法

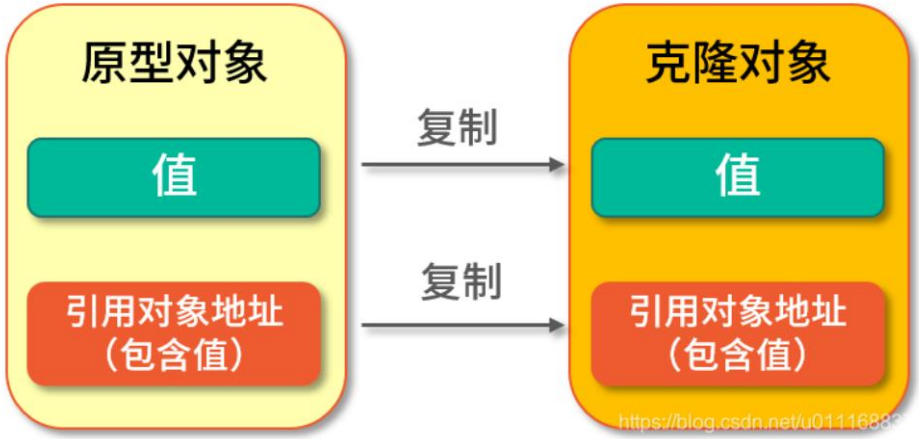
```
def genDate(deepNum):
    string = getExpr(deepNum)
    if (len(string) > 1000):
        return genDate(deepNum)
    return  string

def getExpr(deepNum):

def getTerm(deepNum):

def getFactor(deepNum):

def getConFactor():

def getPowFactor():

def getExprFactor(deepNum):

def getWhite():

def getZero():
```

第四部分：深浅克隆介绍

浅克隆：只能将基本类型的值进行复制和引用类型的地址进行复制，**无法将引用类型指向的真正对象进行复制。**

深克隆：基本数据类型变量和引用类型变量指向的对象都会被复制，即针对引用类型的成员变量真正地复制一份，重新开辟空间保存，**不会出现共享数据的问题。**





可以想一下是否会在第二次作业用到？

深克隆的方法介绍

1.通过构造方法实现深克隆

如果构造器的参数为基本数据类型或字符串类型则直接赋值，如果是对象类型，则需要重新 new 一个新的对象

```
People p1 = new People(1, "Java", address);

addressOfP2 = new Address(p1.getAddress().getId(), p1.getAddress().getCity());
People p2 = new People(p1.getId(), p1.getName(),addressOfP2);
```

2.通过重写clone()方法深克隆

```
// People.java
protected People clone() throws CloneNotSupportedException {
    People people = (People) super.clone();
    people.setAddress(this.address.clone()); // 引用类型克隆赋值
    return people;
    }
// Address.java
protected Address clone() throws CloneNotSupportedException {
    return (Address) super.clone();
    }
```

3.通过序列化和反序列化方法实现深克隆

- Java序列化就是指把Java对象转换为字节序列的过程
- Java反序列化就是指把字节序列恢复为Java对象的过程

我们用序列化反序列化实现深克隆的原理就是我们相当于是把对象保存到了一个文件中，然后立刻又从文件中把这个对象读了出来，此时这个新的对象就是一个深度 clone 的副本了。

```
// 用户类
public class People {
    private Integer id;
    private String name;
    // 包含 Address 引用对象
    private Address addres
    // ……
}

// 地质类
public class Address {
    private Integer id;
    private String city;
    // ……
}
```

Thanks