链滴

程序员的算法趣题系列 -Q02- 数列的四则运算

作者：DattyRabbit

原文链接：https://ld246.com/article/1599279804640

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

前言

此篇为《程序员的算法趣题》中的入门篇第二题"数列的四则运算"的相关解题分析博文。

关于该系列的介绍请看：

《程序员的算法趣题》-开坑记录

这篇拖了一个周，之前自己看牙拔牙加上带我妈去医院看病耽误了不少时间，本来应该是上周就要完的，真是之前才立的一周一篇的flag马上就被打脸了，现在脸都还是生疼（拔牙之后的那种疼）。

题目

Q2 数列的四则运算

大家小时候可能也玩过“组合车牌号里的 4 个数字最终得到 10”的游戏。

组合的方法是在各个数字之间插入四则运算的运算符组成算式，然后计算算式的结果（某些数位之间以没有运算符，但最少要插入 1 个运算符）。

例）

1234 → 1 ＋ 2×3 - 4 ＝ 3

9876 → 9×87 ＋ 6 ＝ 789

假设这里的条件是，组合算式的计算结果为“将原数字各个数位上的数逆序排列得到的数”，并且算的运算按照四则运算的顺序进行（先乘除，后加减）。

那么位于 100~999，符合条件的有以下几种情况。

351-→-3×51 ＝ 153

621-→-6×21 ＝ 126

886-→-8×86 ＝ 688

问题

求位于 1000~9999，满足上述条件的数。

作者思路及代码实现

解决这个问题时，“计算算式的方法”会影响实现方法。如果要实现的是计算器，那么通常会用到逆兰表示法，而本题则是使用编程语言内置的功能来实现更为简单。

很多脚本语言都提供了类似 eval 这样的标准函数。譬如用 JavaScript 实现时，可以用代码清单 02.01 解决问题。

（以下为代码清单02.01）

```
var op = ["+", "-", "*", "/", ""];
for (i = 1000; i < 10000; i++) {
    var c = String(i);
    for (j = 0; j < op.length; j++) {
        for (k = 0; k < op.length; k++) {
            for (l = 0; l < op.length; l++) {
                val = c.charAt(3) + op[j] + c.charAt(2) + op[k] + c.charAt(1) + op[l] + c.charAt(0);
                if (val.length > 4) { /* 一定要插入1 个运算符 */
                    if (i == eval(val)) {
                        console.log(val + " = " + i);
                    }
                }
            }
        }
    }
}
```

第 10 行中的 eval 就是本题的关键点，接下来只是选择和设置运算符了。虽然有比较深的循环嵌套，只要确定了位数就没有问题。

的确，如果只是对比和评估字符串的算式，这样实现就足够了。

我发现一旦用了“*”以外的任意运算符，最终的结果就凑不够4位数了。

说得很对。用“+”时，最大的值只有999+9=1008。逆序排列不可能得到原始值。当然，用“-”也不可能。

基于这样的考虑，如果把代码第 1 行的 op 变量设置成以下值，可以进一步提高程序执行效率。

```
var op = ["*", ""];
```

Point

如果用其他语言实现同样逻辑，需要对 0 进行特别处理。例如在 Ruby 中，“以 0 开头的数”会被作八进制数来处理，因此必须排除以 0 开头的数。此外，也需要排除除数为 0 的情况。

我打算用C语言来实现一遍，但发现没有eval这样的函数。

很多脚本语言都提供eval这样的函数，但C语言里没有这类功能。这种情况下，可以使用“逆波兰表示法”等实现算式计算。

“逆波兰表示法”也常常出现在初学者的编程练习题中呢。

答案

5931（5 * 9 * 31 = 1395）

Column

本书以 Ruby 为主要语言编写源代码，但也有像本题一样用 JavaScript 实现的情况。凡用 JavaScript 实现时，结果都用 console.log 来输出，这个结果可以用浏览器来确认。用 Mozilla Firefox 浏览器开加载了 JavaScript 源代码的 HTML 文件，然后打开“开发者”→“Web 控制台”（如果用 Googl Chrome 浏览器，则是打开“更多工具”→“开发者工具”），就可以确认代码的执行结果了。

eval 函数的危险性

本题使用了 eval 函数。这个函数在计算算式等场景下非常方便，但 eval 可以做到的事情不止于此。如，eval 还可以用来执行指令。

如果在 Web 应用中直接用 eval 执行用户输入的内容，那么用户可能会输入并让程序执行任意指令，括不恰当的指令。举个例子，假设存在下面这样的用 PHP 编写的 Web 页面（代码清单 02.02）。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title> 计算器</title></head>
  <body>
    <form method="post" action="<?php echo $_SERVER['SCRIPT_NAME'];?>">
      <input type="text" name="exp" size="30">
      <input type="submit" value=" 计算">
    </form>
    <div>
      <?php
      if($_SERVER["REQUEST_METHOD"] == "POST"){
        $exp = $_POST["exp"];
        eval("echo $exp;");
      }
      ?>
    </div>
  </body>
```

```
</html>
```

这个页面的功能就是计算表单中输入的类似“1 + 2*3”这样的算式，并且显示结果。正常输入算式情况当然没有问题，但根据输入内容不同，我们还可以执行 PHP 脚本。

举个例子，如果输入 phpinfo( )，那么 PHP 的版本等信息会被打印出来。从安全的角度来看，这是常危险的。

自己做的思路及实现

刚看到这个题目，其实一开始自己能想到的也是用js中的eval来实现，但是因为现在是要用java来实，而java中没有类似eval功能的函数，所以马上想到的就是用动态编译加动态加载的方式来实现。（好前段时间学spring源码课的时候有看到相关的内容）

然后是解决遍历数字，加入操作符拼接成计算表达式的操作。然后自己动手写代码前先试了下几个数里面加运算符，也很容易就可以知道，除了加*之外的运算符，根本不可能保持原有数字的位数。所操作符只需要加入乘法操作符即可，另外还需考虑数字之间不加操作符，将多个数字进行整体化处理符号。

上面那一段得出只能加*的道理很简单。就假设你是一个10-99的十位数，中间只能加一个操作符。除乘法和加法，不可能进行运算之后的数字还能保持是十位数。然后考虑如果是加法的情况，将一个十数拆分成两个个位数，通过加法计算，也不可能满足题目的要求。

因为这两个位数相加之后的取值范围是1-18，能满足是十位数的数字只有10-18，当中任何一个数，不可能满足题目的要求。所以对于其他高位的数字也可以套用同样的规律，操作符只需要考虑乘法操符的情况即可。

下面是动态编译/加载完成的一个类似Eval函数的方法，用于计算数学表达式。

```
/**
 * 自定义的实现类似JavaScript中Eval()函数的工具类，但是因为需要返回计算式的运行结果，
 * 所以这个函数中传入的str不能是语句运行是void类型的。
 *  （例如:"System.out.println(\"Hello\")"）
 *
 * @version V1.0
 * @Package: cn.dattyrabbit.programerArithmeticTopic.util
 * @author: 丁奕
 * @date: 2020-08-26 09:55
 **/
public class Eval {
    public static Object eval(String str) throws Exception {
        StringBuffer sb = new StringBuffer();
        sb.append("public class Temp");
        sb.append("{");
        sb.append("    public Object getObject()");
        sb.append("    {");
        sb.append("        Object obj = " + str + "return obj;");
        sb.append("    }");
        sb.append("}");
        // 调用自定义类加载器加载编译在内存中class文件
        // 说明：这种方式也需要些数据落地写磁盘的
        // 为毛一定要落地呢，直接内存里加载不就完了嘛
        // 应该也是可以的，它从磁盘读了也是进内存
        // 只不过java不允许直接操作内存
```

```
        // 写jni估计是可以
        Class clazz = new MyClassLoader().findClass(sb.toString());
        Method method = clazz.getMethod("getObject");
        // 通过反射调用方法
        return method.invoke(clazz.newInstance());
    }

    public static void main(String[] args) throws Exception {
        Object rval = eval("1+2+3+4;");
        System.out.println(rval);
    }
}

/**
 * 自定义ClassLoader。用于加载动态编译的类
 *
 * @version V1.0
 * @Package: cn.dattyrabbit.programerArithmeticTopic.util
 * @author: 丁奕
 * @date: 2020-08-28 10:18
 **/
public class MyClassLoader extends ClassLoader {
    public static final String classPath = System.getProperty("user.dir") + "\\bin\\";

    /**
     * 自定义的一个加载方法,这样的做法似乎破坏了Java的双亲委派
     * @param classFullName
     * @return
     */
    protected Class customLoadClass(String classFullName){
        String filePath = classFullName2FileName(classFullName, classPath);
        byte[] data = loadClassFromFS(filePath );
        //调用defineClass将一个字节数据转换成一个类并进行初始化工作.
        return defineClass(classFullName, data, 0, data.length);
    }

    /**
     * 将一个完整类名转换为一个当前工程classpath为基础的文件路径.
     * @param classFullName     一个完整类名
     * @param classPath         当前工程类路径
     * @return
     */
    public String classFullName2FileName(String classFullName, String classPath){
        classFullName = classFullName.replaceAll("[.]", "\\\\" );
        return classPath  + classFullName + ".class";
    }

    /**
     * 从文件系统中加载类文件,生成一个byte[].
     * @param filePath   文件路径
     * @return   类文件的字节码数组
     */
    private byte[] loadClassFromFS(String filePath) {
        FileInputStream fis = null;
```

```
        byte[] byteSource = null;
        try {
            fis = new FileInputStream(new File(filePath ) );
            ByteArrayOutputStream tempSource = new ByteArrayOutputStream();
            int readChar = 0;
            while ((readChar = fis.read()) != -1) {
                tempSource.write(readChar );
            }
            byteSource = tempSource.toByteArray();
        } catch (IOException e) {
            //IO出错
            e.printStackTrace();
        }
        return byteSource;
    }

    @Override
    public Class<?> findClass(String str) throws ClassNotFoundException{
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        // 内存中的源代码保存在一个从JavaFileObject继承的类中
        JavaFileObject file = new JavaSourceFromString("Temp", str);
        // System.out.println(file);
        Iterable compilationUnits = Arrays.asList(file);
        // 需要为compiler.getTask方法指定编译路径:
        // 执行过程如下:
        // 1、定义类的字符串表示。
        // 2、编译类
        // 3、加载编译后的类
        // 4、实例化并进行调用。
        String flag = "-d";
        String outDir = System.getProperty("user.dir") + "\\" + "bin";
        Iterable<String> stringdir = Arrays.asList(flag, outDir); // 指定-d dir 参数
        // 建立一个编译任务
        JavaCompiler.CompilationTask task = compiler.getTask(null, null, null,
                stringdir, null, compilationUnits);
        // 编译源程序
        boolean result = task.call();
        if (result) {
            try {
                return customLoadClass( "Temp");
            } catch (Exception e) {
                throw new ClassNotFoundException("Temp", e);
            }
        }
        return null;
    }

}

class JavaSourceFromString extends SimpleJavaFileObject {
    private String code;

    public JavaSourceFromString(String name, String code) {
        super(URI.create("string:///" + name.replace('.', '/')
```

```
            + Kind.SOURCE.extension), Kind.SOURCE);
        this.code = code;
    }

    public CharSequence getCharContent(boolean ignoreEncodingErrors) {
        return code;
    }
}
```

这样就有一个类似Eval()的功能可以使用。然后是解题代码的部分

```
/**
 * 《程序员的算法趣题》 - 入门篇 - Q02 - 数列的四则运算
 *
 * 题目：大家小时候可能也玩过“组合车牌号里的 4 个数字最终得到 10”的游戏。
 *
 * 组合的方法是在各个数字之间插入四则运算的运算符组成算式，
 * 然后计算算式的结果（某些数位之间可以没有运算符，但最少要插入 1 个运算符）。
 *
 * 例)
 *
 * 1234 → 1 ＋ 2×3 - 4 ＝ 3
 * 9876 → 9×87 ＋ 6 ＝ 789
 *
 * 假设这里的条件是，组合算式的计算结果为“将原数字各个数位上的数逆序排列得到的数”，
 * 并且算式的运算按照四则运算的顺序进行（先乘除，后加减）。
 *
 * 那么位于 100~999，符合条件的有以下几种情况。
 *
 * 351-→-3×51 ＝ 153
 * 621-→-6×21 ＝ 126
 * 886-→-8×86 ＝ 688
 *
 * 问题
 *
 * 求位于 1000~9999，满足上述条件的数。
 *
 * @description 数列的四则运算实现
 *              思路：看到这个题目的时候，最初能想到的就是使用JavaScript中的eval()函数来解决是便捷的。但是因为JAVA没有类似
 *              的功能，所以就需要自己来实现一个类似的功能。
 *
 *              这里我目前能想到的实现eval()函数功能的方式就是使用动态编译和动态加载来完成。所得先完成实现该功能的相关工具
 *              类，然后再在这个类当中去使用工具类的方法，实现类似JavaScript的eval()函数来完成题。
 *
 *              然后在已有实现类似eval()函数功能之后，再去遍历1000~9999，然后分别把四则运算操作数也遍历加入到1000~9999的字
 *              符串之中，然后再使用实现的eval()方法来及时运算，获得结果再判断即可。当然这里四运算的操作书除了基本的加减乘
 *              除外还需要多定义一个空符号""，表示不加入操作数，让前后的单个数字连起来作为一个数计算。
 *
```

```
 *              -------------------------
 *              使用最初想到动态编译和动态加载的方式有大量的I/O操作，所以性能较低，耗时巨长，
来找了一种使用avaScriptEngine处理的方法，
 *              耗时减少很多，但是还是觉得耗时挺长的。最后才另寻方法，换为将计算表达式转化为
缀表达式再计算的方法。时间开销终于
 *              降到一个可以接受的范围。
 *
 * @version V1.0
 * @Package: cn.dattyrabbit.programerArithmeticTopic.primer.q2.sequenceArithmetic
 * @author: 丁奕
 * @date: 2020-08-26 09:37
 **/
public class SequenceArithmetic {
    //初始化ScriptEngine
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine engine = manager.getEngineByName("JavaScript");
    /**
     * 包装方法，用于调用的时候不传type，给type赋予0的默认处理方式。
     * @param min
     * @param max
     * @return
     */
    public ArrayList<Integer> searchTargetNum(int min, int max){
        return searchTargetNum(min, max, 0);
    }

    /**
     * 根据传入的最大和最小值来查找其中符合要求的数字
     * @param min 最小值，大于等于
     * @param max 最大值，小于
     * @param type 处理方式: 0|其他 -> 转换为后缀表达式计算;
     *                  1 -> 使用JavaScriptEngine 中的eval处理
     *                  2 -> 使用态编译和动态加载处理
     * @return
     */
    public ArrayList<Integer> searchTargetNum(int min, int max, int type){
        //参数验证
        if(min >= max){
            return (ArrayList)Arrays.asList(-1);
        }
        //初始化满足的数的list
        ArrayList<Integer> targetNum = new ArrayList<>();
        //设置遍历的初始值
        int num = min;
        //定义操作符的数组,空字符暂用_代替，传入eval时替换为空
        char options[] = {'_','*'};
        //循环遍历
        for(; num < max ; num++){
            //打印当前验证的数字
            System.out.println("正在处理:--- "+ num +" ---数字");
            //遍历每个数的时候根据数字转换为字符串，并再遍历操作符，在数字之间插入符号
            char operand[] = String.valueOf(num).toCharArray();
            //根据当前num的值，计算需要插入的操作符的个数，从而求出操作符的组合数量。遍历改
合数量，化为二进制。
```

```
            //需要插入的操作符的组合数量就是 (操作符的长度)^(位数 - 1)。因为操作符的长度为2,所以
果是2的N次幂
            int operatorNums = (int)Math.pow((double)options.length,(operand.length - 1));
            //判断拼接后的字符串，至少插入了一个操作符所以遍历操作符应该从1开始，到operatorN
ms-1结束。
            //0开始则没有添加任何操作符，遍历到operatorNums则操作符会加在数字的外面而不是数
的中间
            for(int i = 1; i < operatorNums; i++){
                //转换成二进制,且根据需要的位数进行左边补零的操作
                String operatorBinary = zeorFill(Integer.toBinaryString(i),operatorNums);
                //开始拼接放入eval函数中的语句
                String evalStatement = "";
                for(int j = operand.length - 1; j >= 0; j--){
                    evalStatement += operand[j];
                    //判断是否是最后一个数
                    if(j != 0){
                        evalStatement += options[Integer.parseInt(String.valueOf(operatorBinary.char
t(j-1)))];
                    }
                }
                //替换_符号。
                evalStatement = evalStatement.replaceAll("_","");
                //处理evalStatement语句中的各个操作数，使其符合java语法规范。因为出现08*1这样
数字是编译无法通过的。所以要对各操作数进行去除左边的0的操作。
                evalStatement = handleStatement(evalStatement);
                //加分号
                evalStatement += ";";
                try{
                    int check;
                    if(type == 1){
                        check = (int) engine.eval(evalStatement);
                    }else if(type == 2){
                        check = (int)Eval.eval(evalStatement);
                    }else{
                        check = ((Double)Calculator.calculate(evalStatement)).intValue();
                    }
                    //若满足条件，放入targetNum中
                    if(num == check){
                        targetNum.add(num);
                    }
                }catch (Exception e){
                    e.printStackTrace();
                }
            }
        }
        return targetNum;
    }

    /**
     * 根据传入的待计算语句进行处理，对各个操作数进行去除左边多余的0的操作，使其符合语法规范
     * @param evalStatement
     * @return
     */
    private String handleStatement(String evalStatement) {
```

```
        String[] operatorNums = evalStatement.split("\\*");
        List<String> array = new ArrayList<>();
        for (String operatorNum : operatorNums) {
            //用字符串转Int再转字符串的操作来去左边的0
            array.add(String.valueOf(Integer.parseInt(operatorNum)));
        }
        String resultStatement = StringUtils.join(array, "*");
        return resultStatement;
    }

    /**
     * 根据传入的二进制数（Integer.toBinaryString直接转换的数）和允许的最大的数字进行左边补
的操作，
     * @param originalBinary
     * @param operatorNums
     * @return
     */
    private String zeorFill(String originalBinary, int operatorNums) {
        String binaryStr = originalBinary;
        String maxOperatorNums = Integer.toBinaryString(operatorNums - 1);
        while(binaryStr.length() < maxOperatorNums.length()){
            binaryStr = "0"+binaryStr;
        }
        return binaryStr;
    }
}
```

代码里面我已经在写代码前把思路用注释写过一遍，所以为啥写这些方法，每个方法是干啥的。都体在代码里了。需要多说的就是通过自己的方式实现一遍之后，运行下来非常耗时，因为自己的实现方有大量的I/O操作，所以性能比较不堪，也是正常，所以才有了后面两种其他的处理方式。也就是通给方法的type参数传值来控制具体的处理方式。

然后是写一个测试类，测试类代码如下

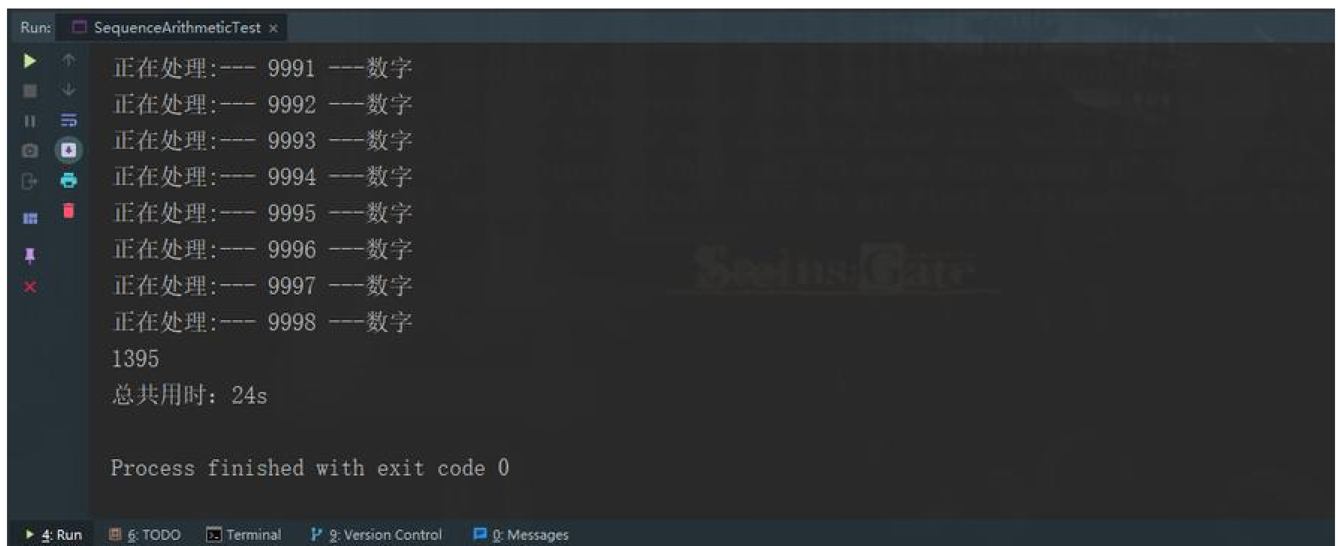
```
/**
 * 数列的四则运算测试
 *
 * @version V1.0
 * @Package: primer.q2.sequenceArithmetic
 * @author: 丁奕
 * @date: 2020-09-01 11:00
 **/
public class SequenceArithmeticTest {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        SequenceArithmetic sequenceArithmetic = new SequenceArithmetic();
        ArrayList<Integer> integers = sequenceArithmetic.searchTargetNum(1000, 9999);
        for (Integer integer : integers) {
            System.out.println(integer);
        }
        long end = System.currentTimeMillis();
        System.out.println("总共用时： " + (end-start) +"ms");
    }
}
```

这里直接给出三种模式的运行结果图作为对比吧

动态编译/加载的方式实现

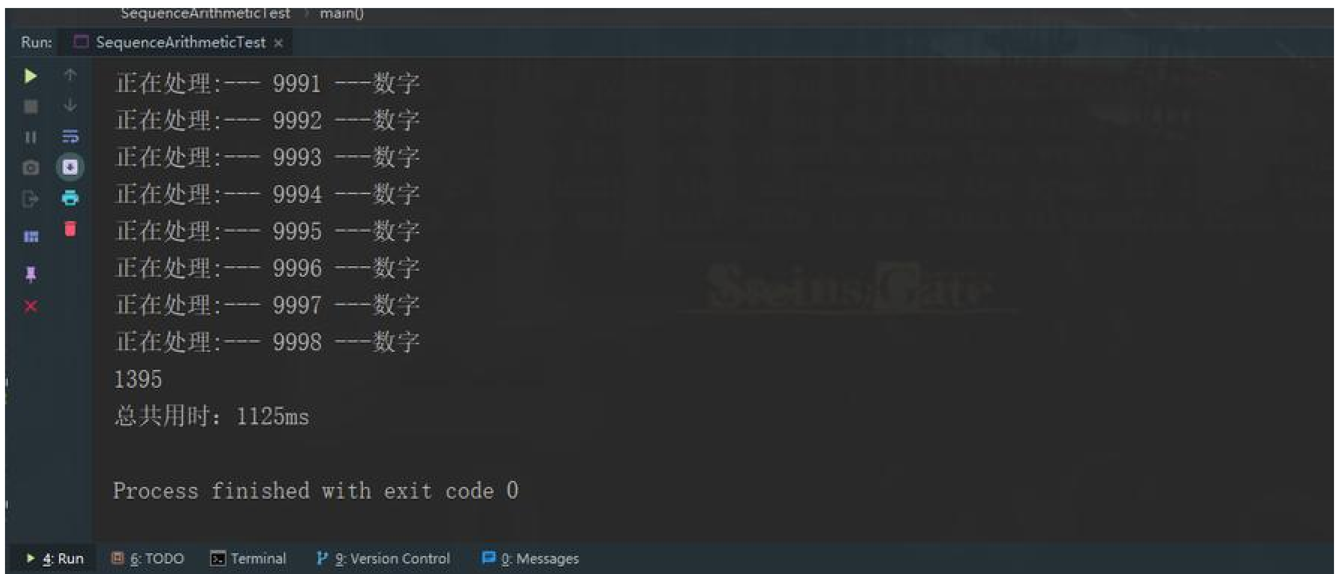
这种真的太久了，我都下去取了个快递回来都还没运行完，肯定不能这样做。

使用ScriptEngine调用js的eval实现



这个勉勉强强能接受，但是肯定不够好

使用后缀表达式计算的方式实现

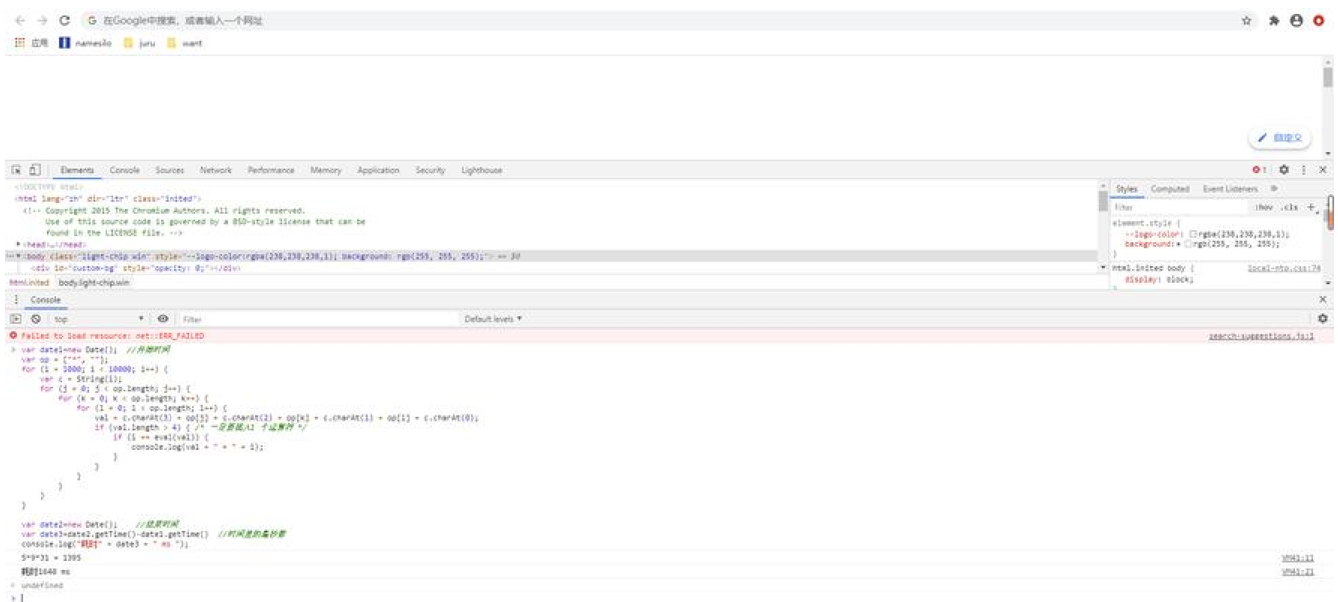


重要找到了一个好的方案，谢天谢地。

这部分使用到的数学表达式转后缀表达式进行计算的代码没贴出来，可以在git上查看源码。git地址前言的开坑记录中有。这部分实现的参考文章在这里：(参考文章:java实现运算字符计算)

不同思路的对比

要对比跟作者的不同，这一节可以运行下作者提供的JS代码。下图为运行结果，我是直接开了个浏览，然后再console中粘入代码运行的。如下图



看计算时间的话，跟我最后使用的后缀表达的方案没差太多。这一回自己这边尝试了三种实现，说到还是搞了蛮久的，不过我觉得我在对遍历数字插入计算符的操作上，有不少创新思路，而且把问题更的泛化了，如果使用作者的方法，那么求100-99999中的符合要求的数，就要改代码。而且循环的层也会有增加。而我自己实现的代码，直接改变参数即可。计算结果如下图

```
13    * @date: 2020-09-01 11:00
14    **/
15  public class SequenceArithmeticTest {
16      public static void main(String[] args) {
17          long start = System.currentTimeMillis();
18          SequenceArithmetic sequenceArithmetic = new SequenceArithmetic();
19          ArrayList<Integer> integers = sequenceArithmetic.searchTargetNum(100, 99999);
20          for (Integer integer : integers) {
21              System.out.println(integer);
22          }
23          long end = System.currentTimeMillis();
24          System.out.println("总共用时: " + (end-start) +"ms");
```

改变参数即可，然后注释了方法中检验每个数字时的打印输出，然后遇到符合的数字，再把计算式输出则得到该运行结果。

```
5*9*31;
9*7*533;
86*6*73;
126
153
688
1395
33579
37668
总共用时: 9680ms

Process finished with exit code 0
```

Compilation completed successfully in 793 ms (a minute ago)

总结

这次的题目，个人觉得实现还是比较满意的，虽然第一时间想到的方案并不是最佳的，但是也尝试实了出来，并且在不满意结果的同时，又尝试了不同的解决方案，最后一步一步得到了一个相对满意的果。

当然最后的代码也是不完善的，也还有很多地方可以继续改进优化，不过实现的时候有考虑去泛化问，使得在后期去比对作者的实现时更具有优势这一点来说，还算是比较满意的。所以平时就算是写业代码，也尽可能的多考虑考虑之如果有业务变更，怎样做才能在今后做最少的改动。

因为这一篇拖了一周，所以下一篇会尽力尽快在最近两天肝出来先把整体的进度补上，peace ray 。

原文链接：程序员的算法趣题系列 -Q02- 数列的四则运算