算法project2 3180104584 毛雨帆

Part1：Leetcode 题目

验证给定的字符串是否可以解释为十进制数字。(Hard)

例如:

"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
" -90e3   " => true
" 1e" => false
"e3" => false
" 6e-1" => true
" 99e2.5 " => false
"53.5e93" => true
" --6 " => false
"-+3" => false
"95a54e53" => false

说明: 我们有意将问题陈述地比较模糊。在实现代码之前，你应当事先思考所有可能的情况。这里给出一份可能存在于有效十进制数字中的字符列表：

数字 0-9
指数 - "e"
正/负号 - "+"/"-"
小数点 - "."
当然，在输入中，这些字符的上下文也很重要。

为方便阅读，将算法思路的描述与代码实现放在一个part

Part2：算法设计与代码实现（C++）

方法一：直接法

开头中间结尾三个位置分情况讨论。

下面我们开始正式以开头、中间、结尾三个不同位置来分情况进行讨论：

1. 在讨论开头、中间、结尾三个不同位置之前做预处理，去掉字符串首尾的空格，可以采用两个指针分别指向开头和结尾，遇到空格则跳过，分别指向开头结尾非空格的字符。
2. 对首字符进行处理，首字符只能为数字或者正负号 '+/-"，我们需要定义三个flag以标志我们之前检是否测到过小数点，自然数和正负号。首字符如为数字或正负号，则标记对应的flag，若不是，直接返回false。
3. 对中间字符的处理，中间字符会出现五种情况，数字，小数点，自然数，正负号和其他字符。

若是数字，标记flag并通过。

若是自然数，则必须是第一次出现自然数，并且前一个字符不能是正负号，而且之前一定要出现过数字，才能标记flag通过。

若是正负号，则之前的字符必须是自然数e，才能标记flag通过。

若是小数点，则必须是第一次出现小数点并且自然数没有出现过，才能标记flag通过。

若是其他，返回false。

4. 对尾字符处理，最后一个字符只能是数字或小数点，其他字符都返回false。

若是数字，返回true。

若是小数点，则必须是第一次出现小数点并且自然数没有出现过，还有前面必须是数字，才能返回true。

```
class Solution {
public:
    bool isNumber(string s) {
        int len = s.size();
        int left = 0, right = len - 1;
        bool eExisted = false;
        bool dotExisted = false;
        bool digitExisited = false;
        //做预处理，去掉字符串首尾的空格，采用两个指针分别指向开头和结尾，遇到空格则跳过，分别指向开头结尾非空格的字符。
        while (s[left] == ' ') ++left;
        while (s[right] == ' ') --right;
        //特殊情况，当且仅当只有一个字符，且不是数字的情况下，返回false
        if (left >= right && (s[left] < '0' || s[left] > '9')) return false;
        //开始处理开头
        if (s[left] == '.') dotExisted = true;
        else if (s[left] >= '0' && s[left] <= '9') digitExisited = true;
        else if (s[left] != '+' && s[left] != '-') return false;
        // 处理中间
        for (int i = left + 1; i <= right - 1; ++i) {
            if (s[i] >= '0' && s[i] <= '9') digitExisited = true;
            else if (s[i] == 'e' || s[i] == 'E') { // e/E cannot follow +/-, must follow a digit
                if (!eExisted && s[i - 1] != '+' && s[i - 1] != '-' && digitExisited) eExisted = true;
                else return false;
            } else if (s[i] == '+' || s[i] == '-') { // +/- can only follow e/E
                if (s[i - 1] != 'e' && s[i - 1] != 'E') return false;
            } else if (s[i] == '.') { // dot can only occur once and cannot occur after e/E
                if (!dotExisted && !eExisted) dotExisted = true;
                else return false;
            } else return false;
        }
        // 处理结尾。结尾只能是数字或者小数点，当它是小数点，它前面不能是小数点和e/E，且前面必须是数字
        if (s[right] >= '0' && s[right] <= '9') return true;
        else if (s[right] == '.' && !dotExisted && !eExisted && s[right-1]<='9' &&'0'<=s[right-1])
            return true;
        //由于中间字符已经处理过了，倘若能开始处理结尾，说明中间已符合，由中间的判断依据知，此时结尾字符前出现过数字等价于结尾字符之前是一个数字，所以上面可以改为else if (s[right] == '.' && !dotExisted && !eExisted && digitExisited)
```

```
        else return false;
    }
};
```

对各种情况进行归纳，可以获得更为简洁的直接法

valid number需满足以下四条规律：

（1）空格只能出现在开头或末尾
（2）正负号只能在数字最前（或指数最前）出现，并且不能超过一个
（3）.的前、后只要有数字就认为可以接受，如果前后都无数字则拒绝
（4）e后只能出现正负号、纯数字和空格，并且至少有一位纯数字

```
class Solution {
    bool isfloat=false;
    //allblank函数，判断从s[start]开始是否全为空格
    bool allblank(const string&s,int start){
        for(int i=start;i<s.length();++i){
            if(s[i]!=' ')return false;
        }
        return true;
    }
    //afterE函数，判断e之后的字符是否符合规范，即需要满足规律（4）
    bool afterE(const string&s,int start){
        if(start>=s.length())return false;
        if(s[start]=='+'||s[start]=='-')return afterE(s,start+1);
        if(s[start]<'0'||s[start]>'9')return false;
        for(int i=start;i<s.length();++i){
            if(s[i]==' ')return allblank(s,i);
            if(s[i]<'0'||s[i]>'9')return false;
        }
        return true;
    }
     //afterdot函数，判断dot之后的字符是否符合规范，即需要满足规律（3）
    bool afterdot(const string&s,int start){
        for(int i=start;i<s.length();++i){
            if(s[i]==' '){
                if(!isfloat)return false;
                return allblank(s,i);
            }
            if(s[i]=='e'){
                if(!isfloat)return false;
                return afterE(s,i+1);
            }
            if(s[i]<'0'||s[i]>'9')return false;
            isfloat=true;
        }
        return isfloat;
    }
    //Float函数，
    bool Float(const string&s,int start){
        if(start>=s.length())return false;
        for(int i=start;i<s.length();++i){
            if(s[i]==' ')return allblank(s,i);
            if(s[i]=='e')return afterE(s,i+1);
            if(s[i]=='.')return afterdot(s,i+1);
```

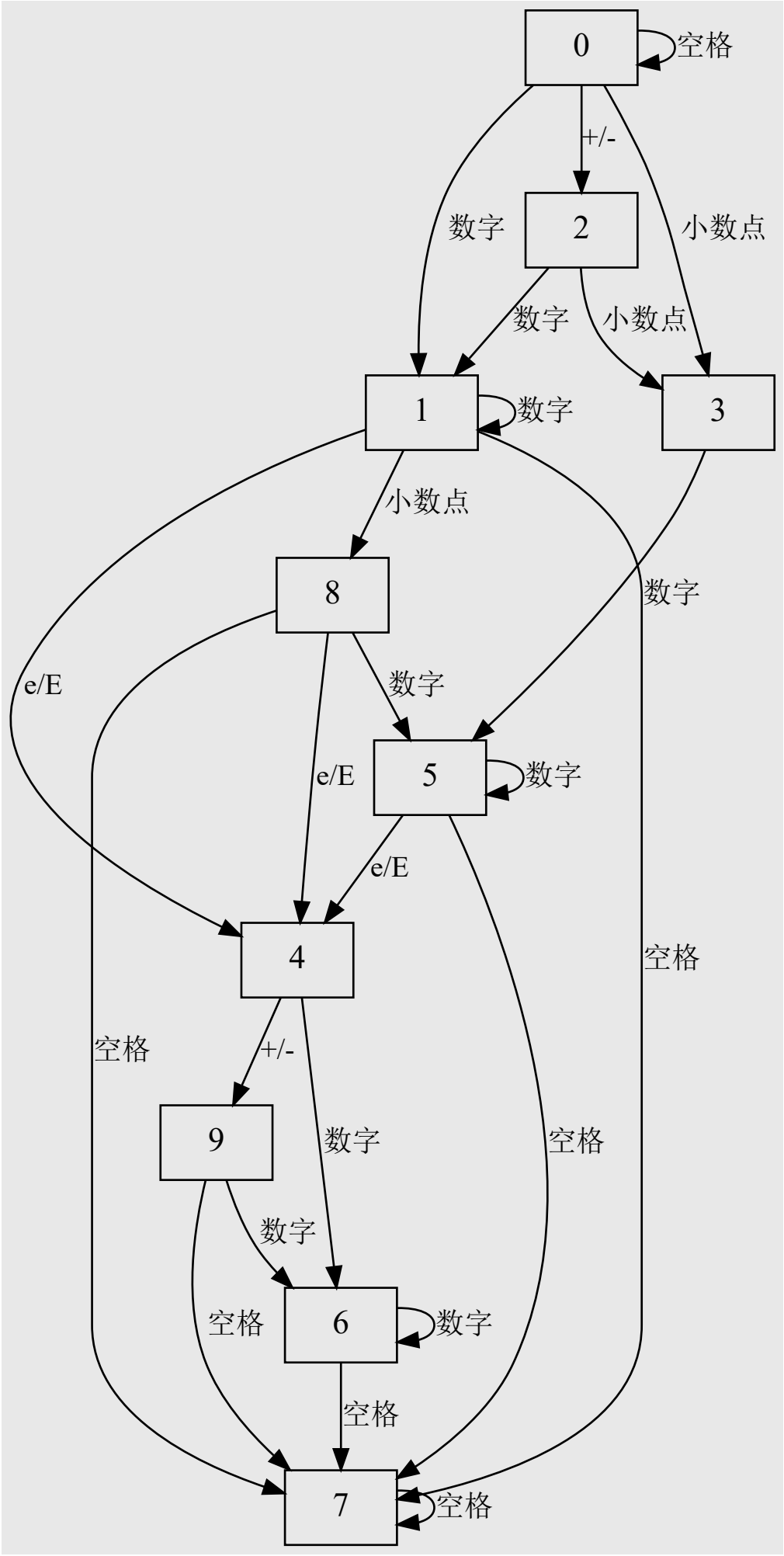
```
            if(s[i]<'0'||s[i]>'9')return false;
            isfloat=true;
        }
        return true;
    }
    bool checksign(const string&s){
        if(s.length()==0)return false;//零字符，返回false
        char c=s[0];
        if(c=='+'||c=='-'){
            if(s.length()>=1&&s[1]>='0'&&s[1]<='9'){
                isfloat=true;
                return Float(s,1);
            }
            if(s.length()>=1&&s[1]=='.')return afterdot(s,2);
            return false;
        }
        if(c>='0'&&c<='9'){
            isfloat=true;
            return Float(s,0);
        }
        if(c=='.')return afterdot(s,1);
        return false;
    }
public:
    bool isNumber(string s) {
        int i=0;
        while(s[i]==' '){
            ++i;
        }
        //当然这里也可以优化为先作预处理，利用双指针消除首尾的空格，这样前面的函数也能够简化
        s=s.erase(0,i);
        return checksign(s);
    }
};
```

方法二：有限自动机（Finite Automata Machine）

本题可以采用确定的有限状态机（DFA）解决。构造一个DFA并实现，构造方法可以先写正则表达式，然后转为 DFA。DFA 作为确定的有限状态机，比 NFA 更加实用，因为对于每一个状态接收的下一个字符，DFA 能确定唯一一条转换路径，所以使用简单的表驱动的一些方法就可以实现，并且只需要读一遍输入流，比起 NFA 需要回读在速度上会有所提升。

构建出来的状态机如下图所示。

状态转换图：


0
空格
+/-
数字
2
小数点
数字
小数点
1
数字
3
小数点
8
数字
e/E
数字
e/E
5
数字
e/E
4
空格
空格
+/-
9
数字
空格
数字
空格
6
数字
空格
7
空格

根据《编译原理》的解释，DFA 从状态 0 接受串 s 作为输入。当s耗尽的时候如果当前状态处于中间状态，则拒绝；如果到达终止状态，则接受。然后，根据 DFA 列出如下的状态跳转表，之后我们就可以采用 表驱动法 进行编程实现了。需要注意的是，这里面多了一个状态 9，是用于处理串后面的若干个多余空格的。所以，所有的终止态都要跟上一个状态 9。其中，有一些状态标识为-1，是表示遇到了一些意外的字符，可以直接停止后续的计算。状态跳转表如下：

状态	空格	数字	字符 e	+/-	小数点	其他
0	0	1	-1	2	3	-1
1	7	1	4	-1	8	-1
2	-1	1	-1	-1	3	-1
3	-1	5	-1	-1	-1	-1
4	-1	6	-1	9	-1	-1
5	7	5	4	-1	-1	-1
6	7	6	-1	-1	-1	-1
7	7	-1	-1	-1	-1	-1
8	7	5	4	-1	-1	-1
9	7	6	-1	-1	-1	-1

```
class Solution {
public:
    bool isNumber(string s) {
        vector<vector<int>> trans={
            {0,1,-1,2,3,-1},
            {7,1,4,-1,8,-1},
            {-1,1,-1,-1,3,-1},
            {-1,5,-1,-1,-1,-1},
            {-1,6,-1,9,-1,-1},
            {7,5,4,-1,-1,-1},
            {7,6,-1,-1,-1,-1},
            {7,-1,-1,-1,-1,-1},
            {7,5,4,-1,-1,-1},
            {-1,6,-1,-1,-1,-1}
        };
        unordered_map<char,int> hash;
        hash[' ']=0;
        for(char i='0';i<='9';i++)hash[i]=1;
        hash['e']=2;
        hash['+']=3;
        hash['-']=3;
        hash['.']=4;
        int start=0;
```

```
        for(int i=0;i<s[i];i++){
            if(hash.count(s[i])==0)return false;
            start=trans[start][hash[s[i]]];
            if(start==-1)return false;
        }
        vector<int> ans={0,1,0,0,0,1,1,1,1,0};
        if(ans[start])return true;
        return false;
    }
};
```

方法三：正则表达式

凡是可以用有限状态机实现的，必然也可以采用正则表达式匹配来实现。

```
class Solution {
public:
    bool isNumber(string &s) {
        // regex r("\\s*[+-]?(\\d+\\.?\\d*|\\.\\d+)(e[+-]?\\d+)?\\s*$");
        int i=s.find_first_not_of(' ');
        int d1=0, dot=0, d2=0, e=0, d3=0;
        if(s[i]=='+' || s[i]=='-') ++i;
        for(; i<s.length() && isdigit(s[i]); d1=++i);
        if(i<s.length() && s[i]=='.') dot=++i;
        for(; i<s.length() && isdigit(s[i]); d2=++i);
        if(dot && !d1 && !d2) return false;
        if(i<s.length() && (d1||d2) && s[i]=='e') e=++i;
        if(i<s.length() && e && (s[i]=='+'|s[i]=='-')) ++i;
        for(; i<s.length() && isdigit(s[i]); d3=++i);
        if(e && (!(d1||d2) || !d3)) return false;
        for(; i<s.length() && s[i]==' '; ++i);
        return i==s.length();
    }
};
```

Part3:实验结果与分析

（1）测试样例（仅给出对直接法的测试，其余方法均通过验证，篇幅限制在此不表）

样例集1：

test(1, "123", true);
test(2, " 123 ", true);
test(3, "0", true);
test(4, "0123", true);
test(6, "-10", true);
test(7, "-0", true);
test(8, "123.5", true);
test(9, "123.000000", true);
test(10, "-500.777", true);
test(11, "0.0000001", true);
test(12, "0.00000", true);
test(13, "0.", true);

```
test(14, "00.5", true);
test(15, "123e1", true);
test(16, "1.23e10", true);
test(17, "0.5e-10", true);
test(18, "1.0e4.5", false);
test(19, "0.5e04", true);
test(20, "12 3", false);
test(21, "1a3", false);
test(22, "", false);
test(23, "     ", false);
test(24, null, false);
test(25, ".1", true); //Ok, if you say so
test(26, ".", false);
test(27, "2e0", true);  //Really?!
test(28, "+.8", true);
test(29, " 005047e+6", true);  //Damn = =| | |
```

样例集2：

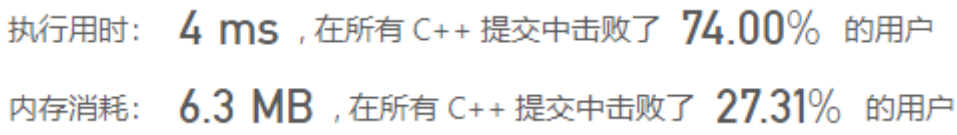
```
string s1 = "0"; // True
string s2 = " 0.1 "; // True
string s3 = "abc"; // False
string s4 = "1 a"; // False
string s5 = "2e10"; // True

string s6 = "-e10"; // False
string s7 = " 2e-9 "; // True
string s8 = "+e1"; // False
string s9 = "1+e"; // False
string s10 = " "; // False

string s11 = "e9"; // False
string s12 = "4e+"; // False
string s13 = " -."; // False
string s14 = "+.8"; // True
string s15 = " 005047e+6"; // True

string s16 = ".e1"; // False
string s17 = "3.e"; // False
string s18 = "3.e1"; // True
string s19 = "+1.e+5"; // True
string s20 = " -54.53061"; // True

string s21 = ". 1"; // False
```

（2）方案的时间与空间效率

方法一：直接法

时间复杂度：O（n）。

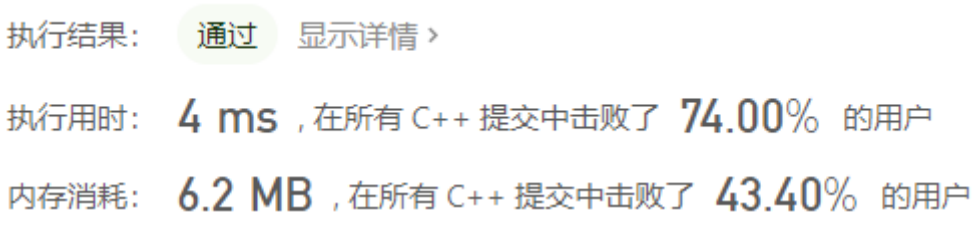
空间复杂度：O（1）。



直接法优化（归纳各自情况分类）

时间复杂度：O（n）。

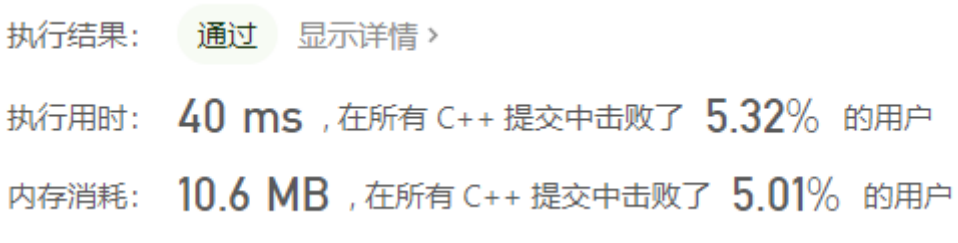
空间复杂度：O（1）。



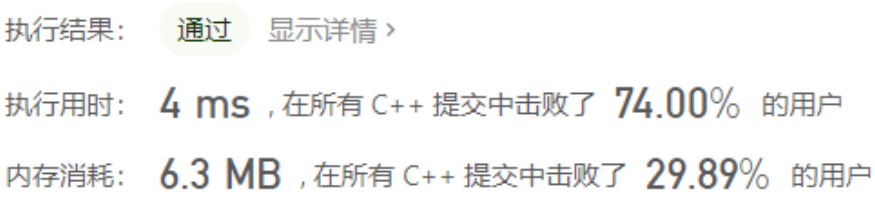
方法二：有限自动机

时间复杂度：O（n）。

空间复杂度：O（1）。



方法三：正则表达式



Part4：拓展——责任链设计模式

有限状态机法看起来已经很清晰明了，只需要把状态图画出来，然后表驱动实现代码就可以了。但缺点是，如果状态图少考虑了特殊情况，修改起来就会很麻烦。

参考网上的技术博客，提出利用责任链的设计模式，会使得写出的算法扩展性以及维护性更高。这里用到的思想就是，每个类只判断一种类型。比如判断是否是正数的类，判断是否是小数的类，判断是否是科学计数法的类，这样每个类只关心自己的部分，出了问题很好排查，而且互不影响。

实际上直接法中的优化方案已经采取了这种思想，只不过是每一个函数判断一种类型，没有集成到类，且不够精准。这里给出参考的责任链设计模式代码，采用java实现。

```
//每个类都实现这个接口
interface NumberValidate {
    boolean validate(String s);
}
```

```
//定义一个抽象类，用来检查一些基础的操作，是否为空，去掉首尾空格，去掉 +/-
//doValidate 交给子类自己去实现
abstract class  NumberValidateTemplate implements NumberValidate{

    public boolean validate(String s)
    {
        if (checkStringEmpty(s))
        {
            return false;
        }

        s = checkAndProcessHeader(s);

        if (s.length() == 0)
        {
            return false;
        }

        return doValidate(s);
    }

    private boolean checkStringEmpty(String s)
    {
        if (s.equals(""))
        {
            return true;
        }

        return false;
    }

    private String checkAndProcessHeader(String value)
    {
        value = value.trim();

        if (value.startsWith("+") || value.startsWith("-"))
        {
            value = value.substring(1);
        }


        return value;
    }



    protected abstract boolean doValidate(String s);
}

//实现 doValidate 判断是否是整数
class IntegerValidate extends NumberValidateTemplate{

    protected boolean doValidate(String integer)
    {
        for (int i = 0; i < integer.length(); i++)
        {
            if(Character.isDigit(integer.charAt(i)) == false)
            {
```

```
                return false;
            }
        }

        return true;
    }
}

//实现 doValidate 判断是否是科学计数法
class SienceFormatValidate extends NumberValidateTemplate{

    protected boolean doValidate(String s)
    {
        s = s.toLowerCase();
        int pos = s.indexOf("e");
        if (pos == -1)
        {
            return false;
        }

        if (s.length() == 1)
        {
            return false;
        }

        String first = s.substring(0, pos);
        String second = s.substring(pos+1, s.length());

        if (validatePartBeforeE(first) == false || validatePartAfterE(second) ==
false)
        {
            return false;
        }


        return true;
    }

    private boolean validatePartBeforeE(String first)
    {
        if (first.equals("") == true)
        {
            return false;
        }

        if (checkHeadAndEndForSpace(first) == false)
        {
            return false;
        }

        NumberValidate integerValidate = new IntegerValidate();
        NumberValidate floatValidate = new FloatValidate();
        if (integerValidate.validate(first) == false &&
floatValidate.validate(first) == false)
        {
            return false;
        }
```

```
        return true;
    }

    private boolean checkHeadAndEndForSpace(String part)
    {

        if (part.startsWith(" ") ||
            part.endsWith(" "))
        {
            return false;
        }

        return true;
    }

    private boolean validatePartAfterE(String second)
    {
        if (second.equals("") == true)
        {
            return false;
        }

        if (checkHeadAndEndForSpace(second) == false)
        {
            return false;
        }

        NumberValidate integerValidate = new IntegerValidate();
        if (integerValidate.validate(second) == false)
        {
            return false;
        }

        return true;
    }
}
//实现 doValidate 判断是否是小数
class FloatValidate extends NumberValidateTemplate{

    protected boolean doValidate(String floatVal)
    {
        int pos = floatVal.indexOf(".");
        if (pos == -1)
        {
            return false;
        }

        if (floatVal.length() == 1)
        {
            return false;
        }

        NumberValidate nv = new IntegerValidate();
        String first = floatVal.substring(0, pos);
        String second = floatVal.substring(pos + 1, floatVal.length());

        if (checkFirstPart(first) == true && checkFirstPart(second) == true)
        {
```

```
            return true;
        }

        return false;
    }

    private boolean checkFirstPart(String first)
    {
        if (first.equals("") == false && checkPart(first) == false)
        {
            return false;
        }

        return true;
    }

    private boolean checkPart(String part)
    {
        if (Character.isDigit(part.charAt(0)) == false ||
            Character.isDigit(part.charAt(part.length() - 1)) == false)
        {
            return false;
        }

        NumberValidate nv = new IntegerValidate();
        if (nv.validate(part) == false)
        {
            return false;
        }

        return true;
    }
}
//定义一个执行者，把之前实现的各个类加到一个数组里，然后依次调用
class NumberValidator implements NumberValidate {

    private ArrayList<NumberValidate> validators = new ArrayList<NumberValidate>
();

    public NumberValidator()
    {
        addValidators();
    }

    private  void addValidators()
    {
        NumberValidate nv = new IntegerValidate();
        validators.add(nv);

        nv = new FloatValidate();
        validators.add(nv);

        nv = new HexValidate();
        validators.add(nv);

        nv = new SienceFormatValidate();
        validators.add(nv);
    }
```

```
    @Override
    public boolean validate(String s)
    {
        for (NumberValidate nv : validators)
        {
            if (nv.validate(s) == true)
            {
                return true;
            }
        }

        return false;
    }


}
public boolean isNumber(String s) {
    NumberValidate nv = new NumberValidator();
    return nv.validate(s);
}
```

Part5：分析与小结

直接法在这题应用起来相对麻烦，因为没有给出清晰的有效数字的构造形式，需自己总结归纳，且要考虑的特殊情况很多，容易遗漏，但时间和空间效率都比较高。

自动机的应用，会使得自己的思路更清晰。一旦列出状态转换表，根据表驱动构建代码非常容易且清晰，但此处时间和空间效率差一些。

而对责任链这一设计模式的应用，使自己眼前一亮，虽然代码变多了，但是维护性，扩展性变的很强了。比如，题目新增了一种情况，"0x123" 16 进制也算是合法数字。这样的话，直接法和NFD就没什么用了，完全得重新设计。但对于责任链，我们只需要新增一个类，专门判断这种情况，然后加到执行者的数组里就够了，非常方便改动。