

16 | NFA和DFA：如何自己实现一个正则表达式工具？

2019-09-18 宫文学 来自北京

《编译原理之美》



回顾之前讲的内容，原理篇重在建立直观理解，帮你建立信心，这是第一轮的认识迭代。应用篇帮你涉足应用领域，在解决领域问题时发挥编译技术的威力，积累运用编译技术的一手经验，也启发你用编译技术去解决更多的领域问题，这是第二轮的认识迭代。而为时三节课的算法篇将你是第三轮的认识迭代。

在第三轮的认识迭代中，我会带你掌握前端技术中的核心算法。而本节课，我就借“怎样实现正则表达式工具？”这个问题，探讨第一组算法：**与正则表达式处理有关的算法。**

在词法分析阶段，我们可以手工构造有限自动机（FSA，或 FSM）实现词法解析，过程比较简单。现在我们不再手工构造词法分析器，而是直接用正则表达式解析词法。

你会发现，我们只要写一些规则，就能基于这些规则分析和处理文本。这种能够理解正则表达式的功能，除了能生成词法分析器，还有很多用途。比如 Linux 的三个超级命令，又称三剑客（grep、awk 和 sed），都是因为能够直接支持正则表达式，功能才变得强大的。

接下来，我就带你完成编写正则表达式工具的任务，与此同时，你就能用正则文法生成词法分析器了：

首先，把正则表达式翻译成非确定的有限自动机（Nondeterministic Finite Automaton, NFA）。

其次，基于 NFA 处理字符串，看看它有什么特点。

然后，把非确定的有限自动机转换成确定的有限自动机（Deterministic Finite Automaton, DFA）

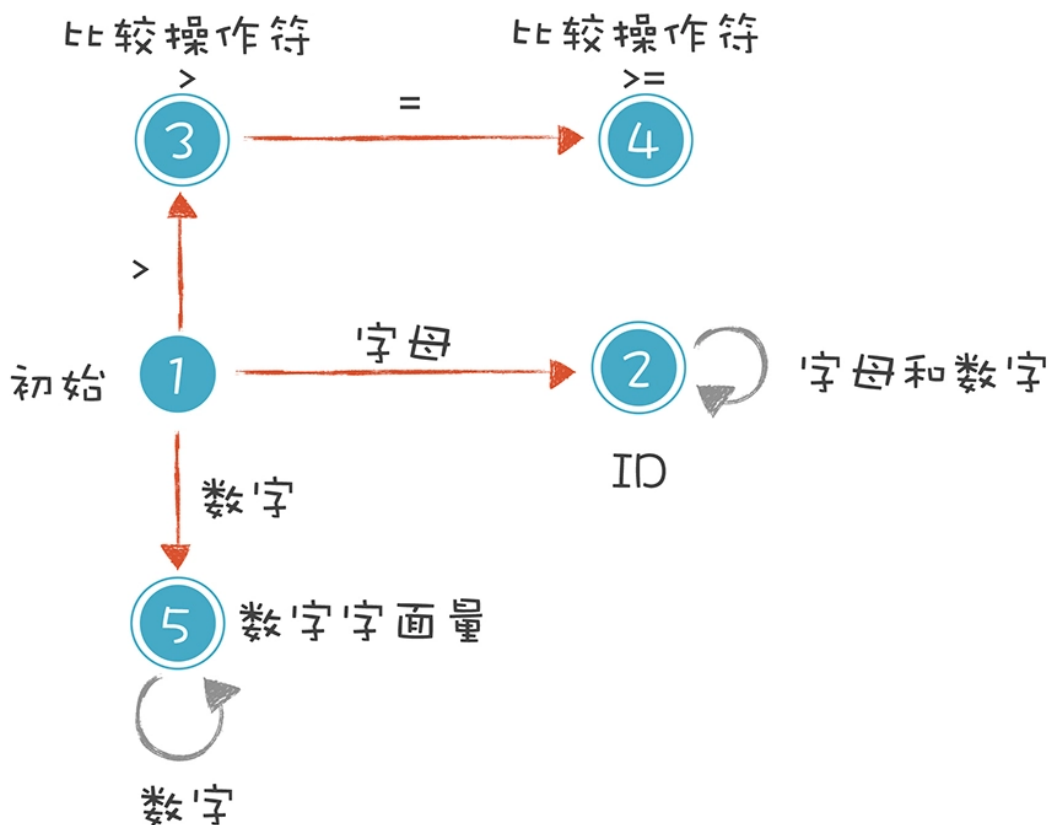
最后，运行 DFA，看看它有什么特点。

强调一下，不要被非确定的有限自动机、确定的有限自动机这些概念吓倒，我肯定让你学明白。

认识 DFA 和 NFA

在讲词法分析时，我提到有限自动机（FSA）有有限个状态。识别 Token 的过程，就是 FSA 状态迁移的过程。其中，FSA 分为**确定的有限自动机（DFA）和非确定的有限自动机（NFA）**。

DFA 的特点是，在任何一个状态，我们基于输入的字符串，都能做一个确定的转换，比如：



NFA 的特点是，它存在某些状态，针对某些输入，不能做一个确定的转换，这又细分成两种情况：

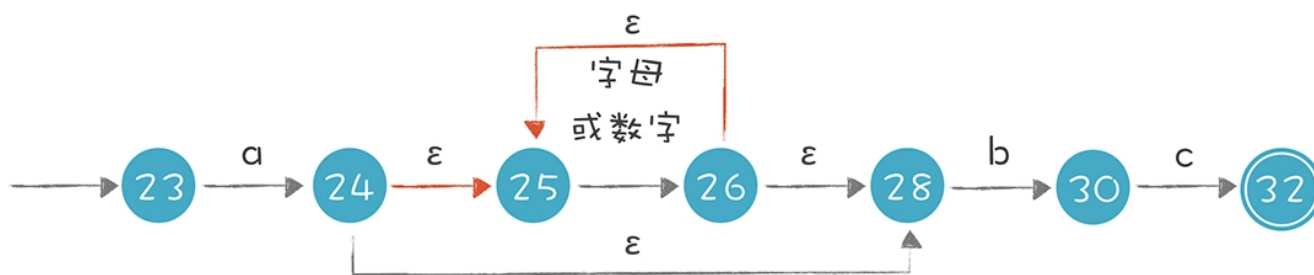
对于一个输入，它有两个状态可以转换。

存在 ϵ 转换。也就是没有任何输入的情况下，也可以从一个状态迁移到另一个状态。

比如，“ $a[a-zA-Z0-9]^*bc$ ”这个正则表达式对字符串的要求是以 a 开头，以 bc 结尾， a 和 bc 之间可以有任意多个字母或数字。在图中状态 1 的节点输入 b 时，这个状态是有两条路径可以选择的，所以这个有限自动机是一个 NFA。



这个 NFA 还有引入 ϵ 转换的画法，它们是等价的。实际上，第二个 NFA 可以用我们今天讲的算法，通过正则表达式自动生成出来。



需要注意的是，无论是 NFA 还是 DFA，都等价于正则表达式。也就是，所有的正则表达式都能转换成 NFA 或 DFA，所有的 NFA 或 DFA，也都能转换成正则表达式。

理解了 NFA 和 DFA 之后，来看看我们如何从正则表达式生成 NFA。

从正则表达式生成 NFA

我们需要把它分为两个子任务：

第一个子任务，是把正则表达式解析成一个内部的数据结构，便于后续的程序使用。因为正则表达式也是个字符串，所以要先做一个小的编译器，去理解代表正则表达式的字符串。我们可以偷个懒，直接针对示例的正则表达式生成相应的数据结构，不需要做出这个编译器。

用来测试的正则表达式可以是 int 关键字、标识符，或者数字字面量：

[复制代码](#)

```
1 int | [a-zA-Z][a-zA-Z0-9]* | [0-9]+
```

我用下面这段代码创建了一个树状的数据结构，来代表用来测试的正则表达式：

[复制代码](#)

```
1 private static GrammarNode sampleGrammar1() {
2     GrammarNode node = new GrammarNode("regex1", GrammarNodeType.Or);
3
4     //int关键字
5     GrammarNode intNode = node.createChild(GrammarNodeType.And);
6     intNode.createChild(new CharSet('i'));
7     intNode.createChild(new CharSet('n'));
8     intNode.createChild(new CharSet('t'));
9
10    //标识符
11    GrammarNode idNode = node.createChild(GrammarNodeType.And);
12    GrammarNode firstLetter = idNode.createChild(CharSet.letter);
13
14    GrammarNode letterOrDigit = idNode.createChild(CharSet.letterOrDigit);
15    letterOrDigit.setRepeatTimes(0, -1);
16
17
18    //数字字面量
19    GrammarNode literalNode = node.createChild(CharSet.digit);
20    literalNode.setRepeatTimes(1, -1);
21
22    return node;
23 }
```

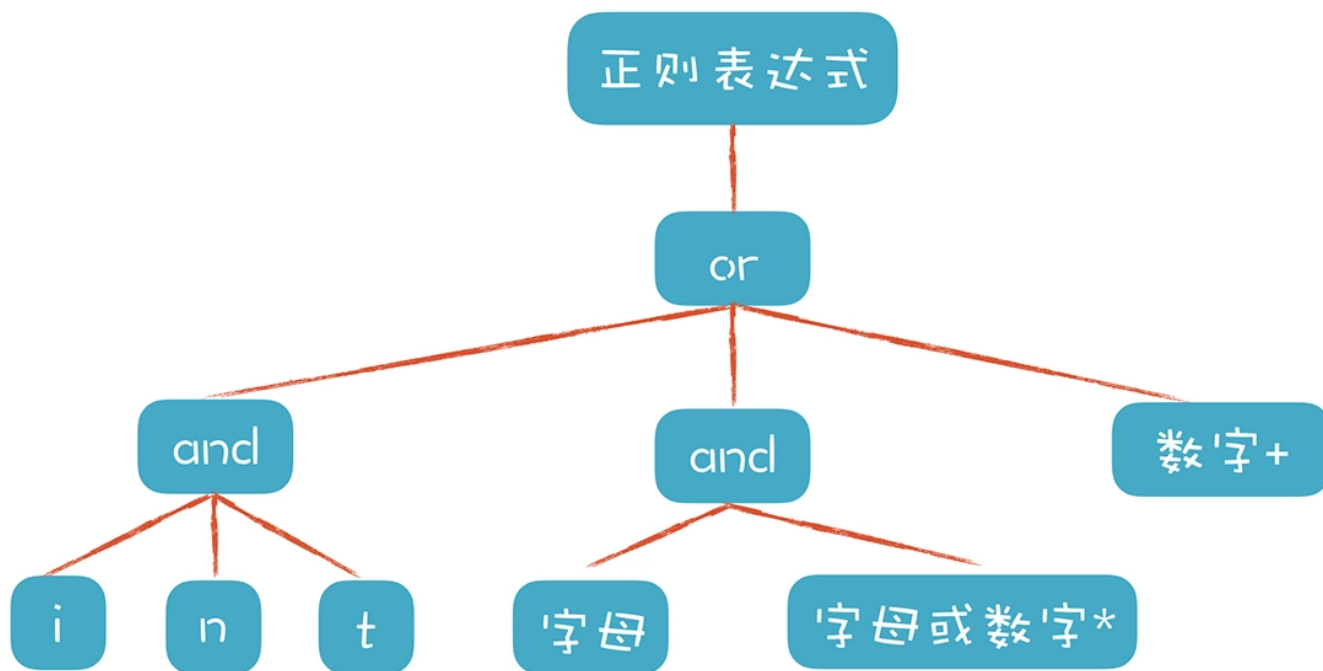
打印输出的结果如下：

[复制代码](#)

```
1 RegExpression
2   Or
3     Union
4       i
5       n
6       t
7     Union
8       [a-z] | [A-Z]
```

```
9      [0-9] | [a-z] | [A-Z] *
10     [0-9] +
```

画成图会更直观一些：



测试数据生成之后，**第二个子任务**就是把表示正则表达式的数据结构，转换成一个 NFA。这个过程比较简单，因为针对正则表达式中的每一个结构，我们都可以按照一个固定的规则做转换。

识别 ϵ 的 NFA：

不接受任何输入，也能从一个状态迁移到另一个状态，状态图的边上标注 ϵ 。



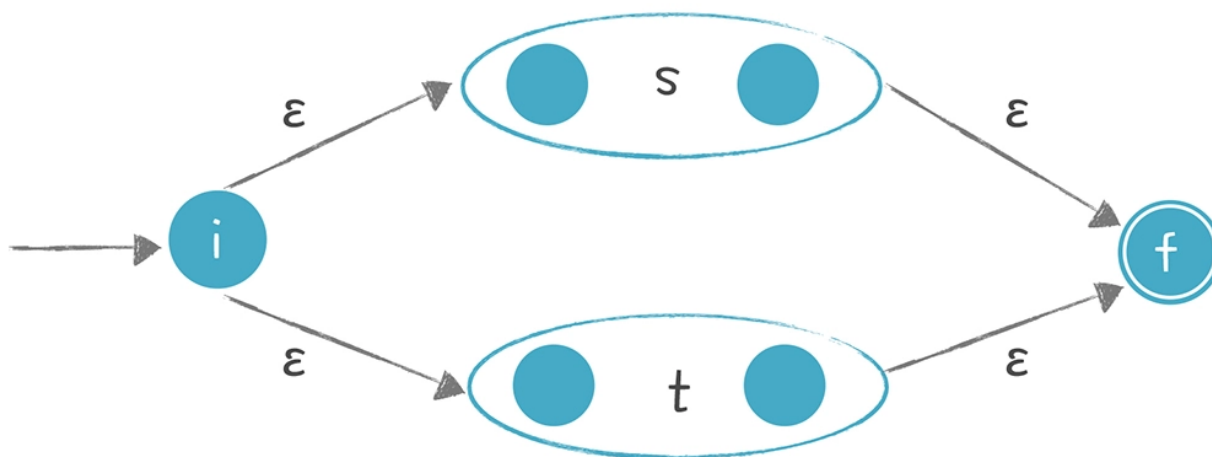
识别 i 的 NFA:

当接受字符 i 的时候, 引发一个转换, 状态图的边上标注 i 。



转换 " $s|t$ " 这样的正则表达式:

它的意思是或者 s , 或者 t , 二者选一。 s 和 t 本身是两个子表达式, 我们可以增加两个新的状态: **开始状态和接受状态 (最终状态)** 也就是图中带双线的状态, 它意味着被检验的字符串此时是符合正则表达式的。然后用 ϵ 转换分别连接代表 s 和 t 的子图。它的含义也比较直观, 要么走上面这条路径, 那就是 s , 要么走下面这条路径, 那就是 t 。



转换 " st " 这样的正则表达式:

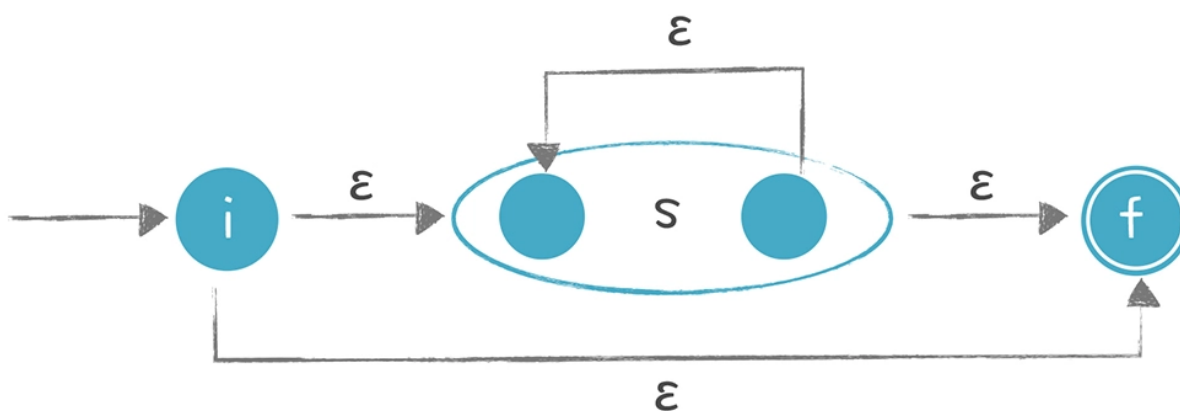
s 之后接着出现 t , 转换规则是把 s 的开始状态变成 st 整体的开始状态, 把 t 的结束状态变成 st 整体的结束状态, 并且把 s 的结束状态和 t 的开始状态合二为一。这样就把两个子图接了起来, 走完 s 接着走 t 。



对于 “?” “*” 和 “+” 这样的操作：

意思是重复 0 次、0 到多次、1 到多次，转换时要增加额外的状态和边。

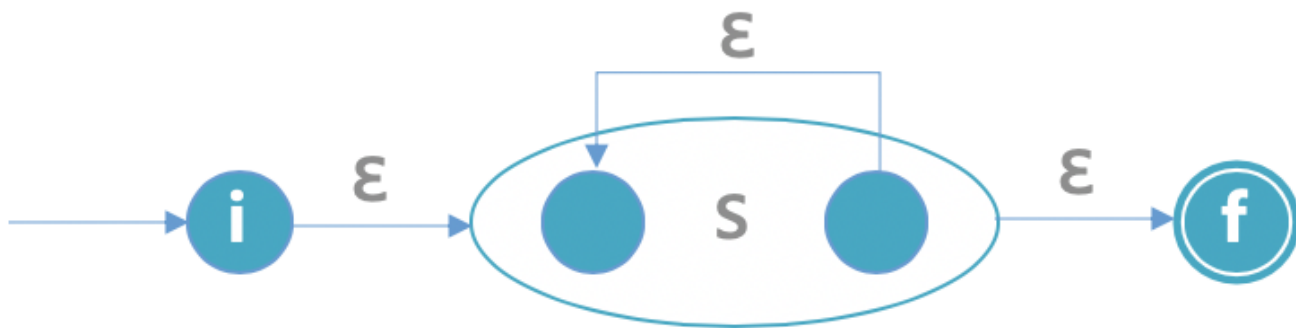
以 “s*” 为例，做下面的转换：



你能看出，它可以从 i 直接到 f，也就是对 s 匹配零次，也可以在 s 的起止节点上循环多次。

“s+”：

没有办法跳过 s，s 至少经过一次。



按照这些规则，我们可以编写程序进行转换。你可以参考示例代码 [Regex.java](#) 中的 `regexToNFA` 方法。转换完毕以后，将生成的 NFA 打印输出，列出了所有的状态，以及每个状态到其他状态的转换，比如 “0 $\epsilon \rightarrow 2$ ” 的意思是从状态 0 通过 ϵ 转换，到达状态 2：

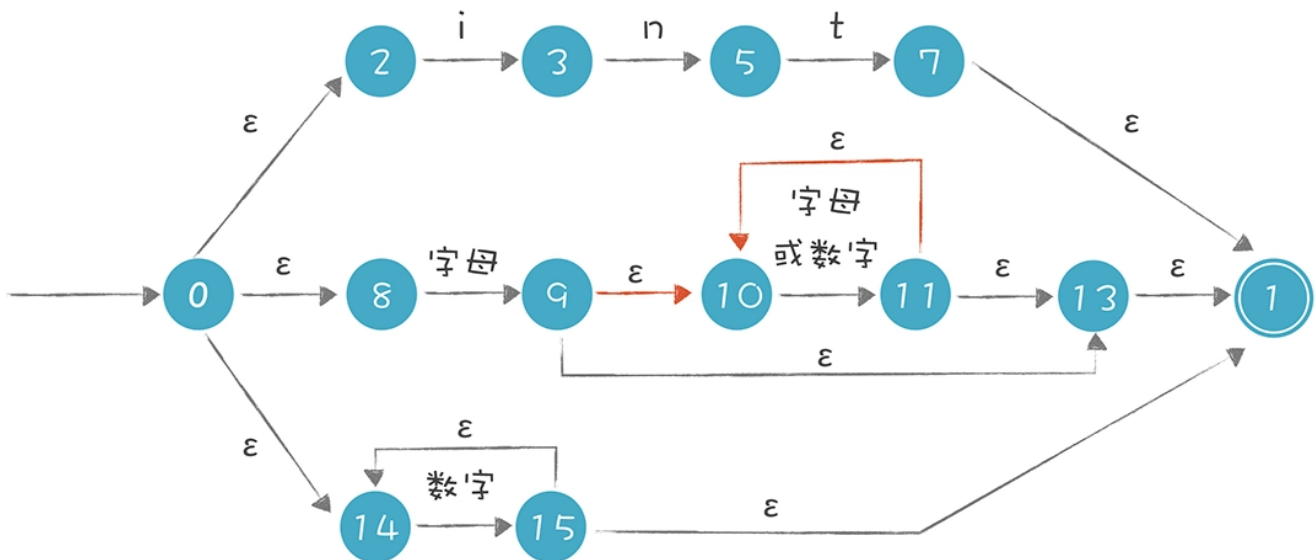
 复制代码

```

1 NFA states:
2 0   $\epsilon \rightarrow$  2
3    $\epsilon \rightarrow$  8
4    $\epsilon \rightarrow$  14
5 2  i  $\rightarrow$  3
6 3  n  $\rightarrow$  5
7 5  t  $\rightarrow$  7
8 7   $\epsilon \rightarrow$  1
9 1  (end)
10 acceptable
11 8  [a-z] | [A-Z]  $\rightarrow$  9
12 9   $\epsilon \rightarrow$  10
13    $\epsilon \rightarrow$  13
14 10 [0-9] | [a-z] | [A-Z]  $\rightarrow$  11
15 11  $\epsilon \rightarrow$  10
16    $\epsilon \rightarrow$  13
17 13  $\epsilon \rightarrow$  1
18 14 [0-9]  $\rightarrow$  15
19 15  $\epsilon \rightarrow$  14
20    $\epsilon \rightarrow$  1

```

我用图片直观地展示了输出结果，图中分为上中下三条路径，你能清晰地看出解析 `int` 关键字、标识符和数字字面量的过程：



生成 NFA 之后，如何利用它识别某个字符串是否符合这个 NFA 代表的正则表达式呢？

以上图为例，当我们解析 intA 这个字符串时，首先选择最上面的路径去匹配，匹配完 int 这三个字符以后，来到状态 7，若后面没有其他字符，就可以到达接受状态 1，返回匹配成功的信息。可实际上，int 后面是有 A 的，所以第一条路径匹配失败。

失败之后不能直接返回“匹配失败”的结果，因为还有其他路径，所以我们要回溯到状态 0，去尝试第二条路径，在第二条路径中，尝试成功了。

运行 Regex.java 中的 matchWithNFA() 方法，你可以用 NFA 来做正则表达式的匹配：

[复制代码](#)

```

1  /**
2   * 用NFA来匹配字符串
3   * @param state 当前所在的状态
4   * @param chars 要匹配的字符串，用数组表示
5   * @param index1 当前匹配字符开始的位置。
6   * @return 匹配后，新index的位置。指向匹配成功的字符的下一个字符。
7   */
8  private static int matchWithNFA(State state, char[] chars, int index1){
9      System.out.println("trying state : " + state.name + ", index =" + index1);
10
11      int index2 = index1;
12      for (Transition transition : state.transitions()){
13          State nextState = state.getState(transition);


```

```

14     //epsilon转换
15     if (transition.isEpsilon()){
16         index2 = matchWithNFA(nextState, chars, index1);
17         if (index2 == chars.length){
18             break;
19         }
20     }
21     //消化掉一个字符, 指针前移
22     else if (transition.match(chars[index1])){
23         index2 ++; //消耗掉一个字符
24
25         if (index2 < chars.length) {
26             index2 = matchWithNFA(nextState, chars, index1 + 1);
27         }
28         //如果已经扫描完所有字符
29         //检查当前状态是否是接受状态, 或者可以通过epsilon到达接受状态
30         //如果状态机还没有到达接受状态, 本次匹配失败
31         else {
32             if (acceptable(nextState)) {
33                 break;
34             }
35             else{
36                 index2 = -1;
37             }
38         }
39     }
40 }
41
42 return index2;
43 }

```

其中, 在匹配 “intA” 时, 你会看到它的回溯过程:

 复制代码

```

1  NFA matching: 'intA'
2  trying state : 0, index =0
3  trying state : 2, index =0    //先走第一条路径, 即int关键字这个路径
4  trying state : 3, index =1
5  trying state : 5, index =2
6  trying state : 7, index =3
7  trying state : 1, index =3    //到了末尾了, 发现还有字符'A'没有匹配上
8  trying state : 8, index =0    //回溯, 尝试第二条路径, 即标识符
9  trying state : 9, index =1
10 trying state : 10, index =1   //在10和11这里循环多次
11 trying state : 11, index =2

```

```
12 trying state : 10, index =2
13 trying state : 11, index =3
14 trying state : 10, index =3
15 true
```

从中可以看到用 NFA 算法的特点：因为存在多条可能的路径，所以需要试探和回溯，在比较极端的情况下，回溯次数会非常多，性能会变得非常慢。特别是当处理类似 s^* 这样的语句时，因为 s 可以重复 0 到无穷次，所以在匹配字符串时，可能需要尝试很多次。

注意，在我们生成的 NFA 中，如果一个状态有两条路径到其他状态，算法会依据一定的顺序来尝试不同的路径。

9 和 11 两个状态都有两条向外走的线，其中红色的线是更优先的路径，也就是尝试让 $*$ 号匹配尽量多的字符。这种算法策略叫做“贪婪 (greedy)”策略。

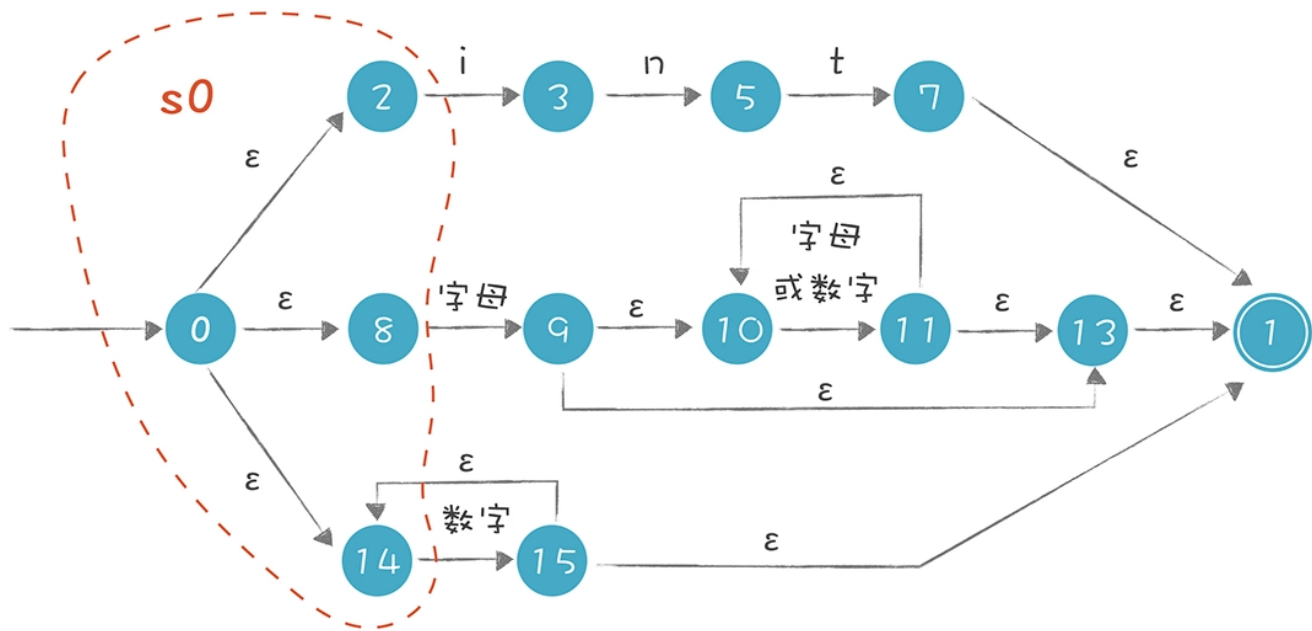
在有的情况下，我们会希望让算法采用非贪婪策略，或者叫“忽略优先”策略，以便让效率更高。有的正则表达式工具会支持多加一个 $?$ ，比如 $??$ 、 $*?$ 、 $+?$ ，来表示非贪婪策略。

NFA 的运行可能导致大量的回溯，所以能否将 NFA 转换成 DFA，让字符串的匹配过程更简单呢？如果能的话，那整个过程都可以自动化，从正则表达式到 NFA，再从 NFA 到 DFA。

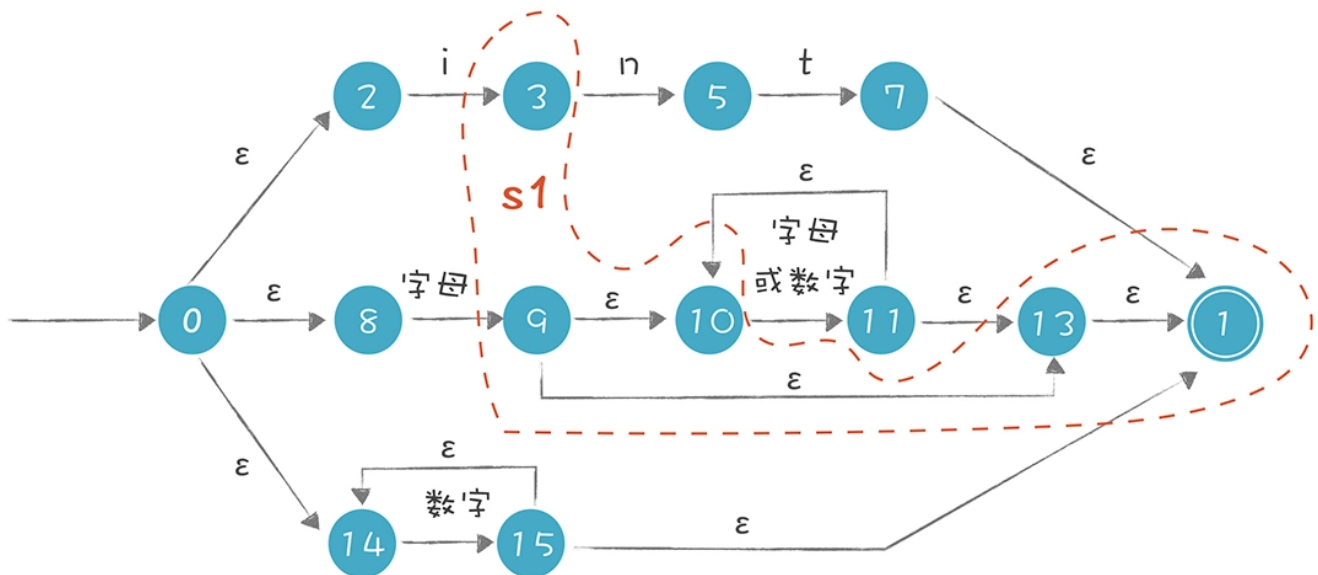
把 NFA 转换成 DFA

的确有这样的算法，那就是**子集构造法**，它的思路如下。

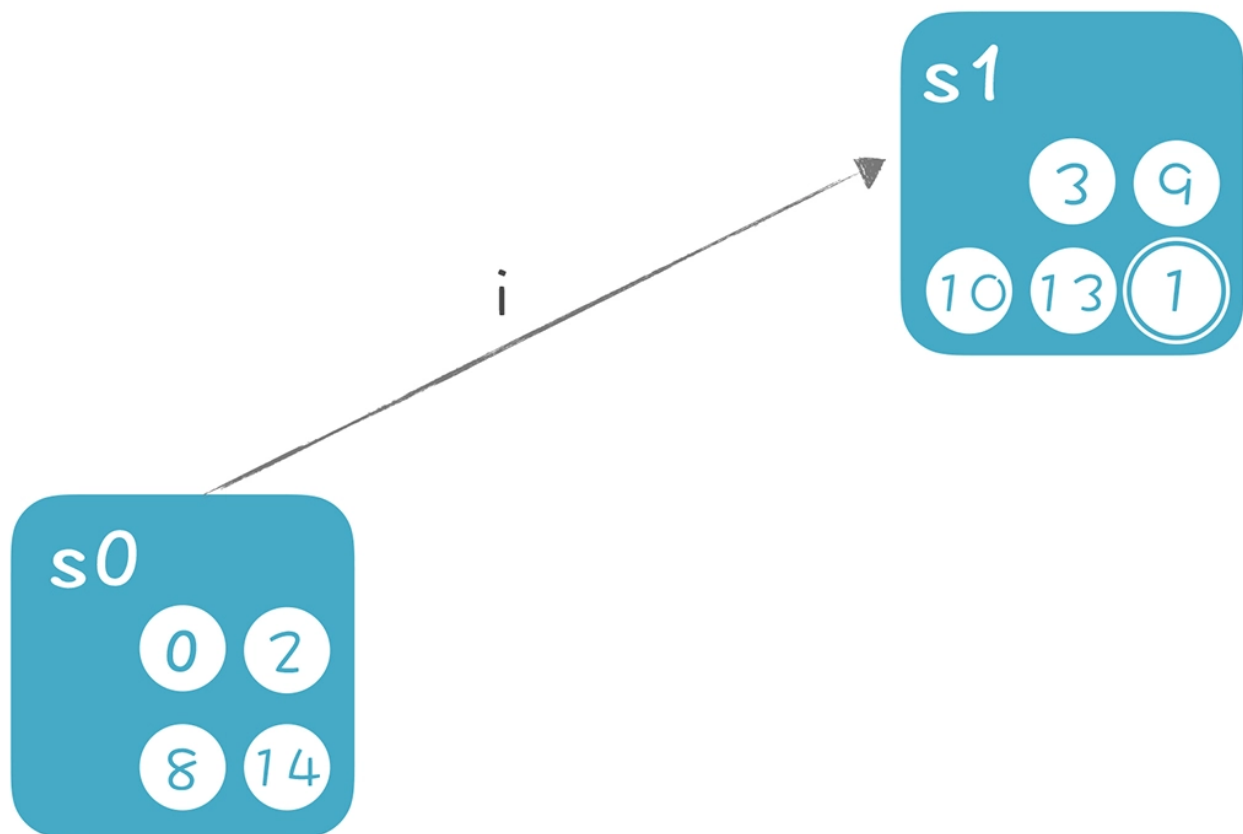
首先 NFA 有一个初始状态（从状态 0 通过 ϵ 转换可以到达的所有状态，也就是说，在不接受任何输入的情况下，从状态 0 也可以到达的状态）。这个状态的集合叫做“状态 0 的 ϵ 闭包”，简单一点儿，我们称之为 s_0 ， s_0 包含 0、2、8、14 这几个状态。



将字母 i 给到 s_0 中的每一个状态，看它们能转换成什么状态，再把这些状态通过 ϵ 转换就能到达的状态也加入进来，形成一个包含 “3、9、10、13、1” 5 个状态的集合 s_1 。其中 3 和 9 是接受了字母 i 所迁移到的状态，10、13、1 是在状态 9 的 ϵ 闭包中。



在 s_0 和 s_1 中间画条迁移线，标注上 i ，意思是 s_0 接收到 i 的情况下，转换到 s_1 ：



在这里，我们把 s_0 和 s_1 分别看成一个状态。也就是说，要生成的 DFA，它的每个状态，是原来的 NFA 的某些状态的集合。

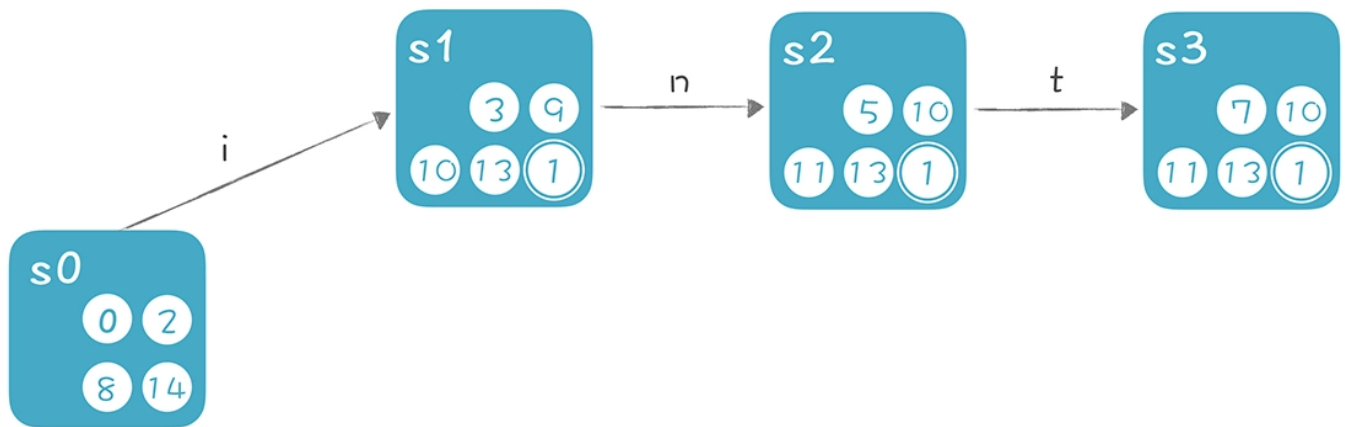
在上面的推导过程中，我们有两个主要的计算：

1. ϵ -closure(s)，即集合 s 的 ϵ 闭包。也就是从集合 s 中的每个节点，加上从这个节点出发通过 ϵ 转换所能到达的所有状态。

2. $\text{move}(s, 'i')$ ，即从集合 s 接收一个字符 i ，所能到达的新状态的集合。

所以， $s_1 = \epsilon\text{-closure}(\text{move}(s_0, 'i'))$

按照上面的思路继续推导，识别 `int` 关键字的识别路径也就推导出来了：



我们把上面这种推导的思路写成算法，参见 [Regex.java](#) 中的 `NFA2DFA()` 方法。我写了一段伪代码，方便你阅读：

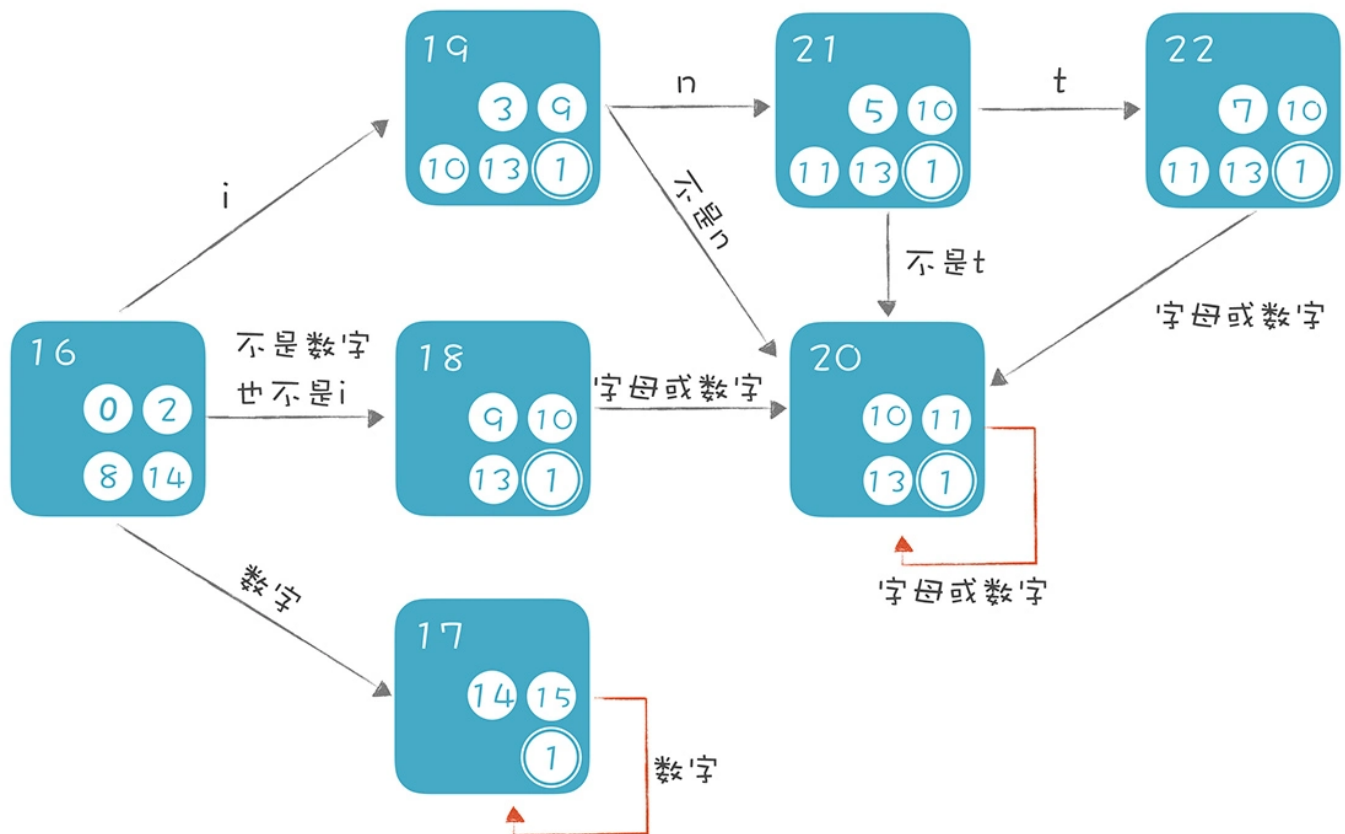
[复制代码](#)

```

1  计算s0，即状态0的ε闭包
2  把s0压入待处理栈
3  把s0加入所有状态集的集合S
4  循环：待处理栈内还有未处理的状态集
5      循环：针对字母表中的每个字符c
6          循环：针对栈里的每个状态集合s(i)（未处理的状态集）
7              计算s(m) = move(s(i), c)（就是从s(i)出发，接收字符c能够
8                  迁移到的新状态的集合）
9              计算s(m)的ε闭包，叫做s(j)
10             看看s(j)是不是个新的状态集，如果已经有这个状态集了，把它找出来
11                 否则，把s(j)加入全集S和待处理栈
12             建立s(i)到s(j)的连线，转换条件是c

```

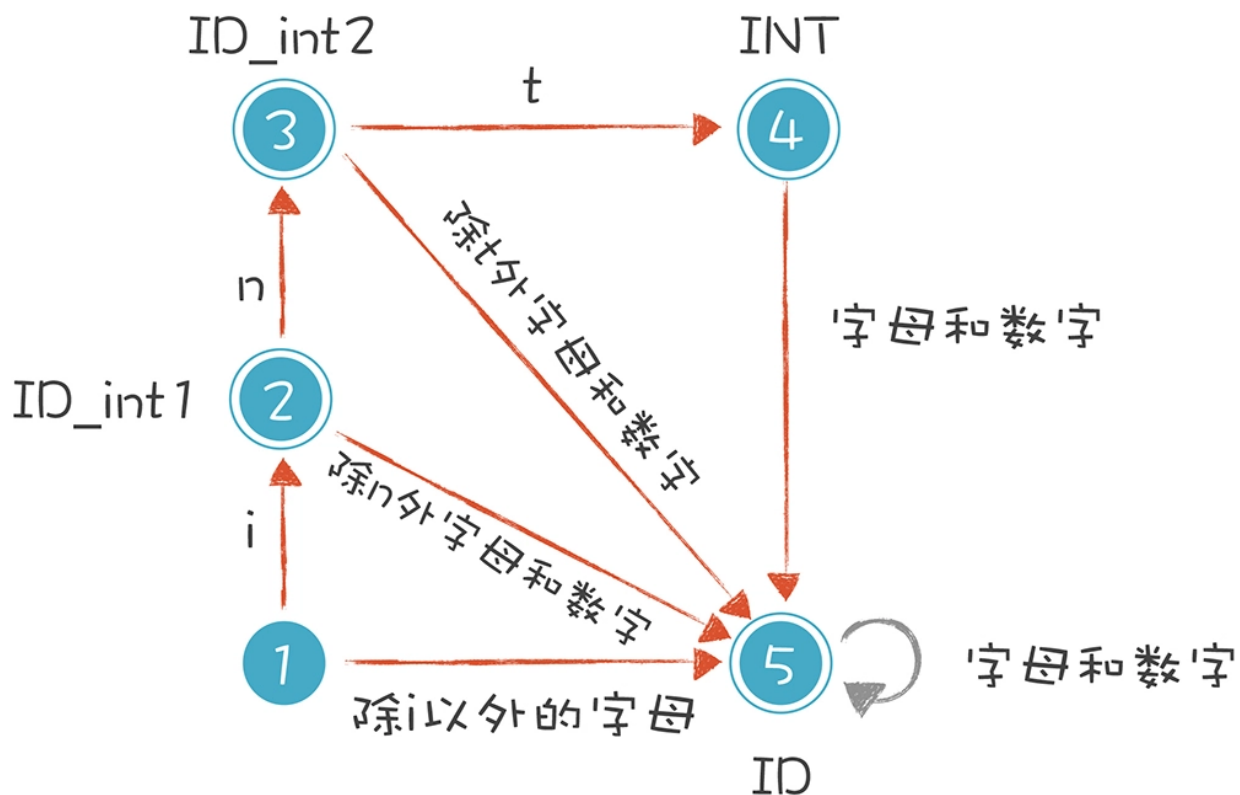
运行 `NFA2DFA()` 方法，然后打印输出生成的 DFA。画成图，你就能很直观地看出迁移的路径了：



从初始状态开始，如果输入是 i，那就走 int 识别这条线，也就是按照 19、21、22 这条线依次迁移，如果中间发现不符合 int 模式，就跳转到 20，也就是标识符状态。

注意，在上面的 DFA 中，只要包含接受状态 1 的，都是 DFA 的接受状态。进一步区分的话，22 是 int 关键字的接受状态，因为它包含了 int 关键字原来的接受状态 7。同理，17 是数字字面量的接受状态，18、19、20、21 都是标识符的接受状态。

而且，你会发现，算法生成的 DFA 跟手工构造 DFA 是很接近的！我们在第二讲手工构造了 DFA 识别 int 关键字和标识符，比本节课少识别一个数字字面量：



不过，光看对 int 关键字和标识符的识别，我们算法生成的 DFA 和手工构造的 DFA，非常相似！手工构造的相当于把 18 和 20 两个状态合并了，所以，这个算法是非常有效的！你可以运行一下示例程序 Regex.java 中的 matchWithDFA() 的方法，看看效果：

[复制代码](#)

```

1 private static boolean matchWithDFA(DFAState state, char[] chars, int index){
2     System.out.println("trying DFAState : " + state.name + ", index =" + index);
3     //根据字符，找到下一个状态
4     DFAState nextState = null;
5     for (Transition transition : state.transitions()){
6         if (transition.match(chars[index])){
7             nextState = (DFAState)state.getState(transition);
8             break;
9         }
10    }
11
12    if (nextState != null){
13        //继续匹配字符串
14        if (index < chars.length-1){
15            return matchWithDFA(nextState,chars, index + 1);
16        }
17        else{
18            //字符串已经匹配完毕
19            //看看是否到达了接受状态

```

```
20         if(state.isAcceptable()){
21             return true;
22         }
23         else{
24             return false;
25         }
26     }
27 }
28 else{
29     return false;
30 }
31 }
```

运行时会打印输出匹配过程，而执行过程中不产生任何回溯。

现在，我们可以自动生成 DFA 了，可以根据 DFA 做更高效的计算。不过，有利就有弊，DFA 也存在一些缺点。比如，DFA 可能有很多个状态。

假设原来 NFA 的状态有 n 个，那么把它们组合成不同的集合，可能的集合总数是 2 的 n 次方个。针对我们示例的 NFA，它有 13 个状态，所以最坏的情况下，形成的 DFA 可能有 2 的 13 次方，也就是 8192 个状态，会占据更多的内存空间。而且生成这个 DFA 本身也需要消耗一定的计算时间。

当然了，这种最坏的状态很少发生，我们示例的 NFA 生成 DFA 后，只有 7 个状态。

课程小结

本节课，我带你实现了一个正则表达式工具，或者说根据正则表达式自动做了词法分析，它们的主要原理是相同的。

首先，我们需要解析正则表达式，形成计算机内部的数据结构，然后要把这个正则表达式生成 NFA。我们可以基于 NFA 进行字符串的匹配，或者把 NFA 转换成 DFA，再进行字符串匹配。

NFA 和 DFA 有各自的优缺点：NFA 通常状态数量比较少，可以直接用来进行计算，但可能会涉及回溯，从而性能低下；DFA 的状态数量可能很大，占用更多的空间，并且生成 DFA 本身

也需要消耗计算资源。所以，我们根据实际需求选择采用 NFA 还是 DFA 就可以了。

不过，一般来说，正则表达式工具可以直接基于 NFA。而词法分析器（如 Lex），则是基于 DFA。原因很简单，因为在生成词法分析工具时，只需要计算一次 DFA，就可以基于这个 DFA 做很多次词法分析。

一课一思

本节课我们实现了一个简单的正则表达式工具。在你的日常编程任务中，有哪些需要进行正则处理的需求？用传统的正则表达式工具有没有性能问题？你有没有办法用本节课讲到的原理来优化这些工作？欢迎在留言区分享你的发现。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

本节课的示例代码我放在了文末，供你参考。

lab/16-18（算法篇的示例代码）：[码云](#) [GitHub](#)

Regex.java（正则表达式有关的算法）：[码云](#) [GitHub](#)

Lexer.java（基于正则文法自动做词法解析）：[码云](#) [GitHub](#)

GrammarNode.java（用于表达正则文法）：[码云](#) [GitHub](#)

State.java（自动机的状态）：[码云](#) [GitHub](#)

DFAState.java（DFA 的状态）：[码云](#) [GitHub](#)

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (16)



刘桢

2020-05-14

尤雨溪:会编译原理真的可以为所欲为

作者回复: 我最近鼓励公司里的一个vue活跃分子学习编译原理, 增加成长的后劲。



👍 9



峰

2019-09-22

老师: 为什么NFA要加空转换这样的操作呢, 感觉对表达能力并没有扩展。

作者回复: 加空转换不是为了扩展表达能力, 而是为了能够通过一个简单标准的方法, 把正则文法转换成NFA。

共 2 条评论 >

👍 7



xindoo

2020-05-19

<https://github.com/xindoo/regex> 我用java写了个正则引擎, 包含了老师这节讲的内容, readme中附了博客, 欢迎各位查阅。

作者回复: 看到了, 很不错! 期待你的博客!



👍 4



漠北

2021-03-30

感觉很像递归转成动态规划

作者回复: 你离散数学学得不错!



👍 3



VictorLee

2020-09-16

这里的柯林闭包和js中的闭包有什么关系吗? mdn中的定义是函数及其环境的混合, 我理解的是js中的闭包是对理算数学中柯林闭包的扩展, 推到极致是可以用集合去解释的, 不知道我理解的对不对

作者回复: 你是指kleene closure?

这两个闭包不是一回事。

前者是一个集合运算, 也就是我们在正则表达式等场合下常用的*的来源。

后者仅仅指引用了本作用域之外的自由变量。至于是否可以用集合计算来解释, 我相信可以。因为整个现代数学的公理化过程(至少是一个流派), 都是基于集合理论的。



👍 1



芒果

2020-01-04

讲的深入原理, 受益匪浅, NFA转DFA可以用子集法

作者回复: 对。

这些算法有兴趣的话, 还可以往细里去追究一下, 比如深度优先vs宽度优先, 时间复杂度, 等等。



👍 1



bucher

2019-09-23

如果在dfa中加上通配符点号有什么好方法吗, 我是在move里进行修改的, 但是这样的话如果有大量正则表达式的时候, nfa转dfa很慢。



👍 1



醉雪飘痕

2019-09-22

请问老师, 您的图是用什么工具做得呀?

编辑回复: 是用Mac自带的Keynote呐~😁



👍 1



沉淀的梦想

2019-09-22

感觉NFA的匹配很适合并行啊, 如果对于每个转换条件, 开个线程并行匹配, 这样就不需要回溯了, 是不是能提升不少效率, 虽然浪费了一些算力

作者回复: 嗯, 如果计算机有多余的算力的情况下。

共 2 条评论 >

👍 1



if...else... 

2022-01-06

理解了一点



风

2021-11-02

用Python实现了一版RE引擎:

<https://github.com/killua-killua/RE-Engine>

包含了这节课的内容 + 一个手写的 re parser



ano

2021-10-09

老师，这个 playScript 的前端，我想用 Go 把它实现出来，就当是把代码都练习一遍，你觉得会有什么问题么？

共 1 条评论 >



飞翔

2019-09-29

老师NFA2DFA这个函数的这一行`dfaState = findDFAState(dfaStates, nextStateSet);`中的`nextStateSet`是不是应该是`calculatedClosures`这个？还有，这一节的代码怎么运行啊，一直编不过

作者回复: 我明天抽空检查一下代码库是否存在编译问题，再给你回复。最近有点忙，回复大家迟了点。



余晓飞

2019-09-28

文中代码块

`int | [a-zA-Z][a-zA-Z0-9]* | [0-9]*`

最后一个字符*应该是+

作者回复: get!是小编的问题，谢谢提醒，已更新。



飞翔

2019-09-27

老师，这一节的代码怎么运行，GrammarNodeType没有找到定义的地方

作者回复: 你再试一下。

我可能之前提交代码的时候，有些没有提交。现在已经把我开发环境的全都提交了。

共 2 条评论 >



Geek_dba6ea

2019-09-21

第一次从这个层面理解了贪心正则匹配

作者回复: 你是指，“知其所以然”了吗？：-)

