南京大學

本 科 毕 业 论 文（设计）

院　系 软件学院

题　目 基于 JavaScript 的词法分析器自动生成工具的设计与实现

学生姓名 葛羽航　学　号 081251041

年　级 2012 级　专　业 软件工程

指导教师 葛季栋　职　称 讲师

论文提交日期

摘　要

词法分析作为编译原理的重要组成部分，在生活生产中有着广泛的应用。词法分析器自动生成工具（英语：Lexical Analyzer Generator，LEX）可以直接从词法规则动作文本转换成对应的词法分析器源代码，简化了词法分析器的开发，方便了词法分析的实现，在实际中也有着重要的作用。在传统的语言平台上，包括 C/C++，JAVA 等，都有对应的 LEX 程序。

随着 Web2.0 的不断进步和发展，尤其是在 html5 的推动下，JavaScript 语言的应用越来越广泛。值得注意的是，在以移动互联网为主体的操作系统 Windows 8 上，使用 html5 开发应用(App)是和使用 C#开发并列的开发方式。所以可以预见，词法分析和 LEX 程序将会在 JavaScript 语言上得到广泛应用。但是，目前没有一款在 JavaScript 语言上开发并且生成 JavaScript 源代码的词法分析器自动生成工具。

因此，开发一个在 JavaScript 语言上的 LEX 工具，简化在 JavaScript 语言上的词法分析任务，显得越发必要。本毕业设计的目的在于开发一个在 JavaScript 语言上的词法分析器自动生成工具。主要工作包括三部分：其一，实现一个 LEX 工具，具有传统词法分析器自动生成工具所具有的功能；其二，针对 JavaScript 语言平台的特殊性，对该 LEX 工具进行优化和扩展，使其可以执行在浏览器前端和 NodeJS 上；其三，将该工具进行包装，使之成为一个开源项目，并对其进行推广。目前上述工作的前两部分已经全部完成，第三个开源工作正在进一步的进行之中。

关键词：LEX，词法分析，编译原理，JavaScript

Abstract

The lexical analysis as an important part of the compiler theory has been widely applied in the production of life. The Lexical Analyzer Generator, known as LEX, can directly convert the lexical rule action text to the corresponding lexical analyzer source code. By this way, The Lexical Analyzer Generator can simplify lexer development, facilitate the realization of lexical analysis, plays an important role in practice. In the traditional language platforms, including C / C + +, JAVA, etc., has a corresponding LEX program.

With the continuously progress and development of the Web2.0, and the promoting of the HTML5, the JavaScript language has been more and more widely used in practice. It is worth noting that there are two parallel development modes available when it comes to Windows 8, a mobile-internet-based operating system, i.e., applying HTML5 or applying C#. So it can be expected, the theory of lexical analysis and the Lexical Analyzer Generator will be widely used in the JavaScript language. However, there is no a lexical analyzer Generator which is developed in the JavaScript language and generates JavaScript source code lexer.

Therefore, the development of a Lexical Analyzer Generator on the JavaScript language to simplify the lexical analysis tasks is more necessary. The purpose of this paper is to develop an LEX tool on the JavaScript language. The main work consists of three parts: First, a Lexical Analyzer Generator on JavaScript, which has functions as the traditional LEX tools; Second, for the particularity of the JavaScript language platform, optimize and expand this LEX tool so that it can performed on the browser front-end and NodeJS platform; Third, package this LEX tool, making it an open source project. The first two parts of the above work has been completed, and the third open-source work is further underway.

Keywords：Lexical Analyzer Generator, lexical analysis, compiler theory, JavaScript

目 录

图目录

表目录

第一章 引言

1.1 项目背景

词法分析作为编译原理中的理论之一，在软件应用中有着广泛的应用：编译器中的 scanner，网络游戏中的脚本引擎，文本编辑器中的自动纠错等等，都会用到词法分析作为语法分析或者词素检测的前趋工作。

词法分析器自动生成工具（传统称之为 lex），则是为了满足不用应用场景下的词法分析需求而设计的用于生成特定语言平台的词法分析器源代码的工具。Lex 工具通过解析词法规则说明书的要求来生成词法分析程序，从而简化了编写词法分析器的难度。

当前 lex 程序已经在得到广泛的使用，在各个语言平台中都有对应的 lex 程序。常见的 lex 程序包括：Flex[1]、JFlex[2]、PLY[3]、CSFlex[4]、RAA[5]等，涵盖了主流的程序语言平台。但是目前在 JavaScript 语言上，却没有对应的 lex 程序。

随着 web2.0 的不断发展和进步，网络应用在日常生活和工业生产中的作用逐渐凸显，基于前端 JavaScript 语言的应用越来越多也越来越复杂。此外，随着 Html5 的普及和推进，包括微软 windows 8 在内的系统平台逐步以 JavaScript 脚本为核心的 App 开发模式做为主流，JavaScript 语言的使用场景将会越来越丰富。可以预见，在以下 web 应用场景中都会使用到词法分析的相关技术：

1. Html5 游戏（特别是 WebGL 推广后的大型网页 3D 游戏）中的游戏脚本引擎
2. 在线文档编辑工具（如 Google Doc, 微软的在线 Office）中的拼写检查
3. Window 8 系统中大量会使用到词法分析器的 App
4. 云计算平台的在线 IDE 中的代码高亮
5. 其它各种会使用到编译原理的应用场景

所以，开发一个基于 JavaScript 的词法分析器自动生成工具越来越有着潜在的必要性的作用。基于这些背景和考虑，本项目完成了基于 JavaScript 的词法分析器自动生成工具。

---

[1] Flex 是 C/C++平台的 lex 工具，参见：http://flex.org

[2] JFlex 是 Java 平台的 lex 工具，参见：http://jflex.de

[3] PLY 是 Python 平台的 lex 工具，参见：http://www.dabeaz.com/ply/

[4] CSFlex 是 c#(.net)平台的 lex 工具，参见：http://ostatic.com/csflex

[5] RAA 是 Ruby 平台的 lex 工具，参见：http://raa.ruby-lang.org/project/ruby-lex/

1.2 国内(外)lex 技术研究现状

词法分析器和词法分析器生成工具（lex）作为编译原理的重要应用，在生活生产中有着广泛的应用。对于 lex 相关技术，国内外也有着深入而成熟的研究。目前对于 lex 技术的研究主要集中在正则表达式、NFA 和 DFA、lex 程序等方面。

1.2.1 正则表达式

正则表达式[9]出现于理论计算机科学的自动控制理论和形式化语言理论中。在这些领域中有对计算（自动控制）的模型和对形式化语言描述与分类的研究。1940 年代，Warren McCulloch 与 Walter Pitts 将神经系统中的神经元描述成小而简单的自动控制元。在 1950 年代，数学家斯蒂芬·科尔·克莱尼利用称之为“正则集合”的数学符号来描述此模型。肯·汤普逊将此符号系统引入编辑器 QED，然后是 Unix 上的编辑器 ed，并最终引入 grep。自此，正则表达式被广泛地使用于各种 Unix 或者类似 Unix 的工具，例如 Perl。

1.2.2 NFA 和 DFA

有限状态机（英语：finite-state machine, FSM），又称有限状态自动机，简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。非确定有限自动机(英语：Nondeterministic finite automaton, NFA)[7]是对每个状态和输入符号对可以有多个可能的下一个状态的有限状态自动机；确定有限自动机(英语：deterministic finite automaton, DFA)[8]是一个能实现状态转移的自动机,它的下一个可能状态是唯一确定的。

1.2.3 Lex 程序

在计算机科学里面，Lex[10]是一个产生词法分析器(lexical analyzer) ("扫描器"(scanners)或者"lexers")的程式。Lex 常常与 yacc 语法分析器产生程式(parser generator)一起使用。Lex(最早是埃里克·施密特和 Mike Lesk 制作)是许多 UNIX 系统的标准词法分析器(lexical analyzer)产生程式，而且这个工具所作的行为被详列为 POSIX 标准的一部分。

传统的 Lex 程序读进一个代表词法分析器规则的输入字串流，然后输出以 C 语言实做的词法分析器源代码。

虽然传统上是商业软件，但是有些根据原本 AT&T 程式码这些版本的 Lex 可以以公开源代码的形式获得，并被视为某些系统的一部份，例如说 OpenSolaris

和贝尔实验室九号计划。 另一个有名的 Lex 公开源代码版本是 flex，代表"快速的词法分析器"(fast lexical analyzer)。

此外在国际上，除了 C 语言平台之外，在包括 JAVA、.NET、Ruby、Python 等语言平台上都有对应的 lex 程序。

1.3 论文的主要工作和组织结构

本文介绍了词法分析和词法分析自动生成工具的相关理论知识，以及一个基于 JavaScript 的 lex 工具——AliceLex 的设计和实现。主要是对词法分析相关的编译原理进行描述，同时描述一个基于 JavaScript 的 Lex 的设计和实现过程以及这个过程中出现的一些问题，论文主要工作包括：

1. 介绍了 Web2.0 的发展现状，阐述了编译原理，尤其是词法分析在 JavaScript 上面的应用前景。

2. 介绍了编译原理的相关理论知识，词法、文法的相关概念。包括正则表达式，NFA，DFA，以及相关的算法理论。

3. 介绍了词法分析器自动生成工具和其实现的难点要点，展现当前 Lex 工具的现状。

5. 详细介绍了基于 JavaScript 的词法分析自动生成工具 AliceLex 的设计与实现，展示了该工具的使用典示例。

本篇论文的组织结构如下：

第一章：概述和前言部分，主要介绍了项目背景，当前词法分析和词法分析自动生成工具的研究及现状 ，并描述了该论文的主要工作。

第二章：主要介绍基于 JavaScript 的词法分析器自动生成工具的开发和实现过程中用到的相关理论知识和理论研究，以及基于 JavaScript 的 Lex 工具与传统 Lex 工具相比较的特点和实现难点。

第三章：从需求分析和总体设计两个方面，描述基于 JavaScript 的 Lex 工具的提出背景和开发者信息，同时分析和总结出功能性和非功能性需求。

第四章：对 AliceLex 工具的详细设计和实现进行描述，重点包括 Lex 工具相关算法的设计和具休实现。

第五章：AliceLex 工具的使用说明和相关示例，包括展示经典 Lex 示例在 AliceLex 上的实现，以及介绍了基于 AliceLex 核心的在线代码编辑器。

第六章：总结该项目已实现的功能，探讨项目的缺点和不足，并指出该项目未来的扩展和发展方向

第二章 AliceLex 工具相关技术概述

2.1 词法分析相关理论

词法分析是编译的第一阶段，其主要任务是读入源程序的输入字符、将它们组成词素，生成并输出一个词法单元序列，每个词法单元对应于一个词素。这个词法单元序列可以被输出到词法处理单元进行下一步的分析处理（典型的就是输出到语法分析器中进行语法分析）。

2.1.1 词法单元、模式和词素

词法分析中的三个关键术语[1]：

- **词法单元**由一个单元名和一个可选的属性值组成。词法单元名是一个表示某种词法单位的抽象符号，比如一个特定的关键字，或者代表一个标识符的输入字符序列。
- **模式**描述了一个词法单元的词素可能具有的形式。当词法单元是一个关键字时，它的模式就是组成这个关键字的字符序列。对于标识符和其他词法单元，模式是一个更加复杂的结构，它可以很多符号串匹配。
- **词素**是源程序中的一个字符序列，它和某个词法单元的模式匹配，并被词法分析器识别为该词法单元的一个实例。

2.1.2 正则表达式

正则表达式是一种用来描述词素模式的重要表示方法，它可以高效地描述在处理词法单元时要用到的模式类型。Lex 程序中对于词法规则的定义部分，正是使用正则表达式及其扩展来描述。下面给出相关的定义[2]。

定义 2.1 正则表达式

设Δ是字母表。Δ上的正则表达式（regular expressions）和正则表达式表示的语言递归定义如下：

1. $\emptyset$ 是一个正则表达式，表示空集；
2. ε 是一个正则表达式，表示$\{\varepsilon\}$；
3. α 是一个正则表达式，表示$\{\}$，这里$\alpha \in \Delta$；
4. 如果 r 和 s 分别表示语言 R 和 S 的正则表达式，那么：
   $(r|s)$ 是表示$R \cup S$的正则表达式；
   (rs) 是表示 RS 的正则表达式；
   (r^*) 是表示 R^* 的正则表达式。

定义 2.2 正则表达式扩展

在 lex 的词法规则定义块中使用的正则表达式扩展表示法在词法分析的规约中有着普遍的使用，其典型的扩展表示定义[1]如下：

1. 一个或多个实例。单目后缀运算符+表示一个正则表达式及其语言的正闭包。即如果 r 是一个正则表达式，那么$(r)^+$ 就表示语言$(L(r))^+$，$r^+ = r^*r = rr^*$，$r^* = r+|\varepsilon$
2. 零个或一个实例。单目后缀运算符 ？ 表示零个或一个出现，r? = r|e，即 L(r) = L(r) U {e}
3. 字符类。正则表达式 a1|a2|…|an（其中 ai 是字母表中的各个符号）可以缩写成[a1a2…an]。当 a1,a2,…,an 形成一个逻辑上的连续的序列时，比如连续的大写字母，连续的小写字母时，可以表示成[a1-an]。如果在[]内第一个字符是^，表示排除[]内之后字符范围内的所有字符。
4. 重复。后续花括弧{}运算符表示重复之前的正则表达式。r{m,n}表示最少 m 个，最多 n 个 r 的重复出现。如果省略 m，表示最少 0 个，省略 n，表示最多不限制。
5. 预定义表示。转义符 \ 用来表示预定义的正则表达式。比如\d 表示数字，相当于[0-9]。更多的预定符包括 \D, \s, \S, \w, \W[6]

2.1.2 有穷自动机

词法分析器的工作核心是有穷自动机的状态转移变化，而 lex 程序的工作原理则是将词法规则分析转换成自动机并对自动机进行处理和优化，包括正则表达式转到不确定有穷自动机（NFA）、不确定有穷自动机（NFA）转到确定有穷自动机（DFA）、确定有穷自动机的压缩（DFA compression）等等。最后将自动机转换成数组表的表示形式。下面给出相关的定义[1][2]：

定义 2.3 不确定有穷自动机（NFA）

一个不确定的有穷自动机（NFA）由以下几个部分组成：

1. 一个有穷的状态集合 S。
2. 一个输入符号集合 E，即输入字母表（input alphabet）。
3. 一个转换函数（transition function），它为每个状态和 EU{e}中的每个符号给出了相应的后继状态（next state）的集合。
4. S 中的一个状态 s0 被指定为开始状态或叫初始状态。
5. S 中的一个子集 F 被指定为接受状态（或者叫终止状态）集合。

表 2.1 NFA 定义

[6] 更多的预定义符号及其含义，参见 5.1.3 节

定义 2.4 确定有穷自动机（DFA）

设$M=(E,R)$是一个不含ε-规则的有穷自动机。如果对每个$q\epsilon Q$和$a\epsilon\Delta$，有$card(\{\boldsymbol{rhs}(r)\|r\epsilon R,\boldsymbol{rhs}(r)=qa\})\leq 1$，则$M$是一个确定有穷自动机（deterministic finite automaton）。

表 2.2 DFA 定义

即确定有穷自动机（DFA）是不确定有穷自动机（NFA）的一个特例，没有输入ε之上的转换动作，对于每个状态 s 和每个输入符号 a，有且只有一条标号为 a 的边离开。

2.2 词法分析器

词法分析器包括三大组件：

1. 表示自动机的一个转换表。
2. 由 Lex 编译器从 Lex 程序中直接拷贝到输出文件的函数。
3. 输入程序定义的动作。这些动作是一些代码片段，将在适当的时候由自动机模拟器调用。

下图概括了由 Lex 生成的词法分析器的体系结构：

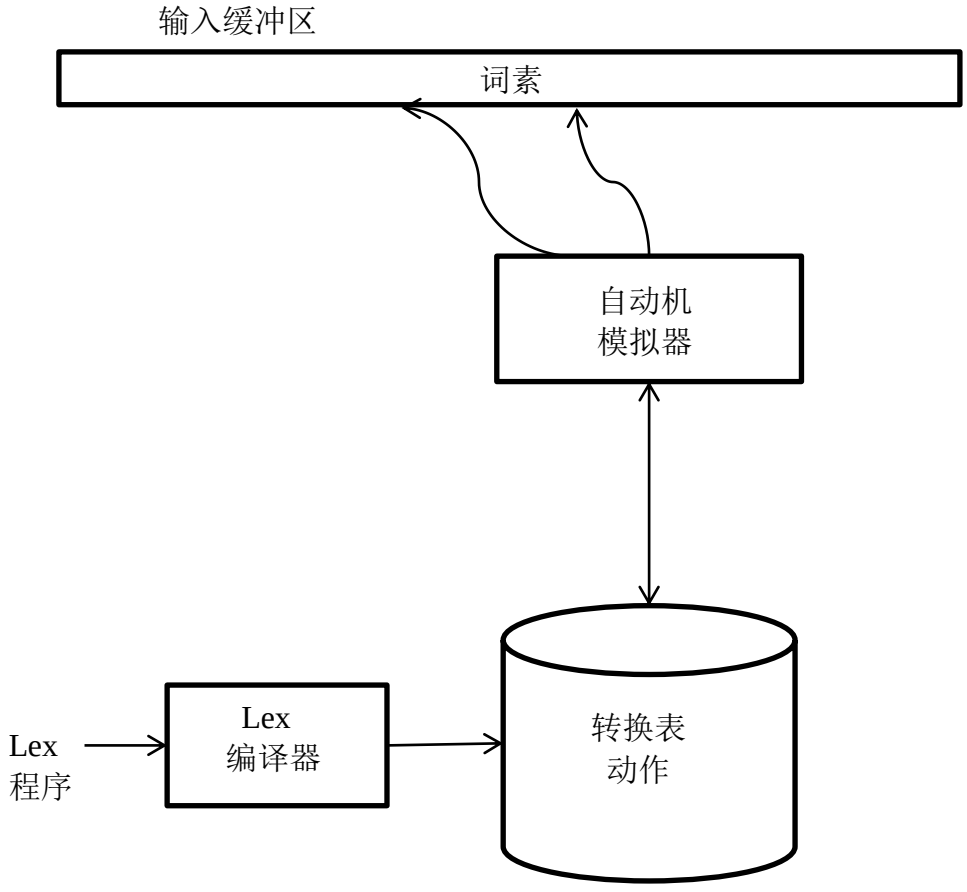

图 2.1 词法分析器体系结构

第三章 AliceLex 工具需求分析与概要设计

AliceLex 是 JavaScript 语言平台上的词法分析器自动生成工具，其本身使用 JavaScript 脚本编写，通过词法规则动作本文，生成 JavaScript 脚本的词法分析器源代码。

3.1 项目需求分析

3.1.1 功能需求

作为一个词法分析器自动生成工具，AliceLex 工具拥有传统 Lex 程序的功能，包括以下主要方面：

1. 接受 Lex 词法规则文本，将其转换成规则对应的词法分析器的 JavaScript 源码。
2. 接受带状态的词法规则，规则中可以定义不同的状态，在不同状态下有规则动作；不同状态间通过内置函数 yygoto 进行跳转。
3. 生成的词法分析器是面向对象的，该对象名称可以由用户在词法规则文本中通过参数指定。
4. 支持通过参数指定大小写忽略，指定是否支持 Unicode 编码。
5. 支持词法分析器源码模板。

同时，AliceLex 作为 JavaScript 语言平台上的 Lex 工具，还具有如下功能需求：

1. 可以在浏览器前端和服务器端 nodejs 两个平台上使用。
2. nodejs 端包装成 nodejs 的扩展，可以通过 npm 安装。

该工具的用例相对简单，用例图如下：

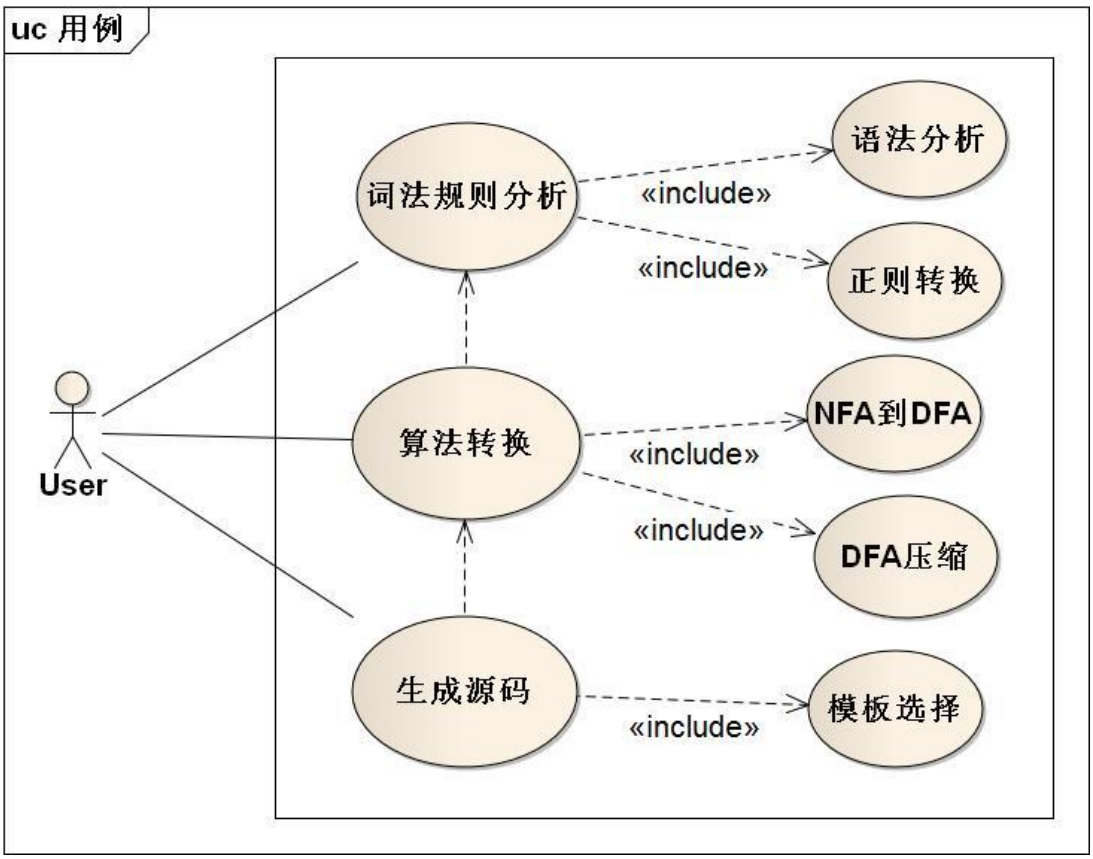

图 3.1 AliceLex 用例图

3.1.2 非功能需求

目前，AliceLex 工具需要符合以下约定：

1. 工具使用的语言和生成的源码都是 JavaScript。
2. 工具可以在浏览器前端和 nodejs 两种平台上使用。
3. 工具接收的 lex 规则文本符合传统 lex 程序使用的规则文本的定义。

在此基础上，AliceLex 工具的性能需求如下：

时间特性要求	工具解析词法规则文件的功能，要求解析 1kb 文本时间不超过 0.5 秒
	工具对词法分析器的算法，要求运行时间不超过 1.5 秒
灵活性要求	本系统需要满足开源项目的要求，支持跨平台使用
	本系统需要预留开发和扩展接口，以便后续开发
输入输出要求	输入文件格式为纯文本：txt 文件或者直接的字符串
	输出文件为 JavaScript 源代码
故障处理要求	编写异常类，提供异常处理功能，工具出现错误能友好的提醒
	对于开发人员，提示错误发生的位置和原因，以便于修改和维护。

表 3.1 AliceLex 性能需求

同时，作为一个开源项目，本项目使用如下开发平台和工具：

开发语言	JavaScript
开发工具	Aptana 3.0
源代码管理	Git
操作系统	支持浏览器的所有操作系统平台
开源地址	https://github.com/YuhangGe/alicelex

表 3.2 AliceLex 的开发平台

3.2 项目概要设计

3.2.1 体系结构设计

AliceLex 是 JavaScript 语言上的 lex 工具，跟传统 lex 工具一样，其核心流程包括：读取词法规则(lex)文本—>解析词法规则文本，构造规则文本对应的 NFA—>将 NFA 转换成 DFA—>DFA 压缩（最小化）—>DFA 转换成线性表—>将线性表生成词法分析器程序源码。

综合以上分析，AliceLex 项目的体系结构使用管道过滤器风格，其体系结构图如下：

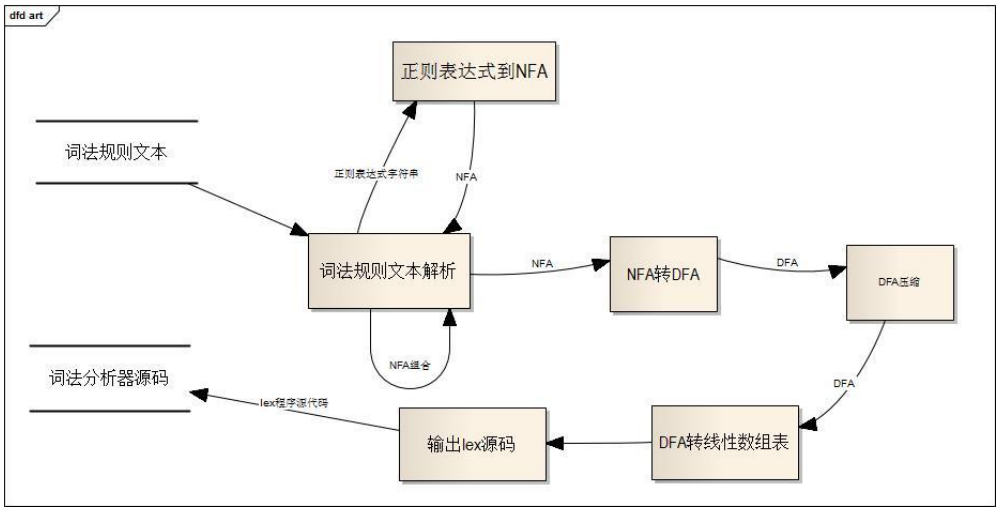

图 3.2 AliceLex 的体系结构图

其整个过程是从词法规则文本，通过对文本进行语法分析，将其中的正则表达式规则转换成 NFA，进一步转换成 DFA，然后对 DFA 进行状态最小化（压缩），进行线性表生成，最后根据线性表生成规则文本对应的词法分析器源代码。整个过程的输入输出程流式结构，符合管道过滤器风格的特点[13]。

3.2.2 模块划分及功能

AliceLex 项目的模块划分及其功能的描述如下：

1. 公共核心模块：包括了输入符类，有限状态机状态基类等，提供公共的核心数据结构。同时提供对外的接口。
2. Nfa 处理模块：包括 NFA 状态类和 NFA 类，处理 NFA 及其相关算法，包括将正则表达式转换成 NFA，将 NFA 转换成 DFA 等。
3. Dfa 处理模块：包括 DFA 状态类和 DFA 类，处理 DFA 及其相关算法，包括对 DFA 进行状态最小化等。
4. 线性表和等价类管理模块：包括了将 DFA 生成线性表的类和等价类管理类，处理线性表的生成、压缩以及输入符的等价类管理。
5. 实用类模块：包括了一个实用类，封装了对数组操作，字符串处理和字符类型判断等常用的函数操作。
6. 源码模版模块：提供生成的词法分析器源代码的不同模板，以适应不同场合下的词法分析器的生成需求。

第四章 AliceLex 工具的详细设计与实现

4.1 模块概述

AliceLex 工具共由 6 个模块组成，分别是 Core、Nfa、Dfa、Table、Utility、Template。模块之前的依赖关系图如下：

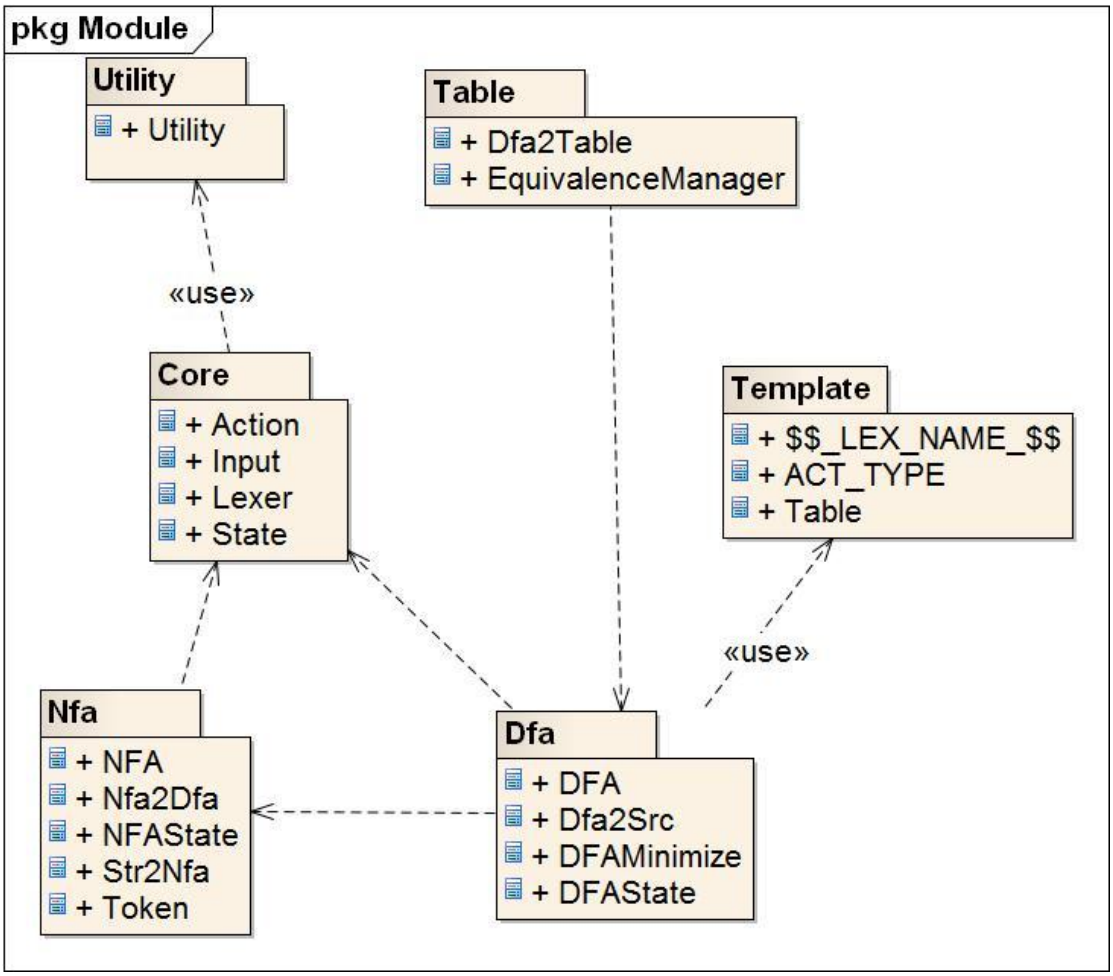

图 4.1 体系模块依赖关系图

各模块的概述如下：

- Core 公共核心模块，包含了核心的类——输入类 Input，状态父类 State，动作类 Action，以及全局唯一的对外公开类 Lexer。
- Nfa 模块，包含了与 NFA 相关的处理类——NFA 状态类 NFAState，NFA 类，实现 NFA 到 DFA 转换的类 Nfa2Dfa，将正则表达式字符串转换成 NFA 的类 Str2Nfa。

- Dfa 模块，包含了与 DFA 相关的处理类——DFA 状态类 DFAState，DFA 类，实现 DFA 状态最小化的类 DFAMinimize，将 DFA 类最终输出成记法分析器源代码的类 Dfa2Src
- Table 模块，包含和表驱动相关的类——为 DFA 生成相应的线性数组表（即 default, base, check, next, action 表）的类 Dfa2Table，在整个词法分析过程中对输入符进行等价类管理的类 EquivalenceManager。
- Utility 模块，包含了一个静态类——Utility。其作用是提供静态公共辅助函数。
- Template 模块。该模块严格意义上不是工具的代码构成，只是需要生成的词法分析器的源代码的模版。

4.2 模块的详细设计

4.2.1 公共核心模块—Core

模块类图

公共核心模块 Core 提供了工具各个模块都需要依赖的全局类 Input，Action，State，以及工具全局唯一的对外接口类 Lexer，模块的类图如下：

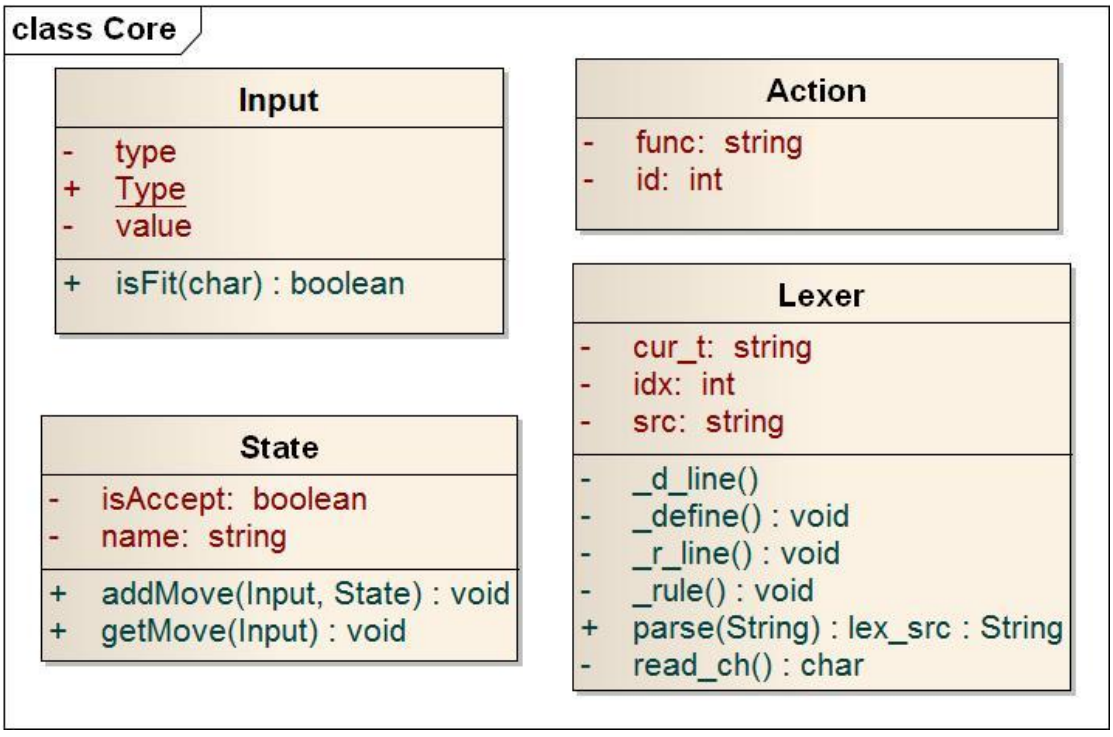

图 4.2 公共核心模块类图

模块类详细描述

- Input 类，输入类，用于包装 NFAState 和 DFAState 的状态转移的输入条件。Type 是枚举类型，包括 SINGLE, RANGE, EXCEPT, EMPTY，分别代表单字符的输入，某个范围的输入，排除某个范围的输入和空输入（特指状态转移中的 ε 输入）。函数 isFit 接收一个 char，判断该 char 是否满足该输入。
- Action 类，动作类，用于包装在词法规则代码中规则动作定义部分里定义的动作。其中 id 是其标识，func 储存用户定义的动作代码。
- State 类，状态基类。定义了有限状态机的状态，属性 isAccept 表明该状态是否是接受状态；name 是该状态的名字（这只是个辅助属性，并没有实际作用）。函数 addMove 和 getMove 是抽像函数，声明了添加一个状态转移和得到某输入对应的转移状态的接口。State 类的子类包括 Nfa 模块中的 NFAState 和 Dfa 模块中的 DFAState。
- Lexer 是 AliceLex 工具全局的唯一对外公开接口。其公开函数 parse 接收词法规则的文本，将其最终转换成该规则对应的词法分析器的 JavaScript 源代码。整个转换过程使用自顶向下的递归实现，类中定义的属性（src, cur_t, idx）和私有函数（read_ch, _define, _d_line, _rule, _r_line 等）都是实现词法和语法分析的成员。

4.2.2 NFA 模块—Nfa

模块类图

Nfa 模块包括 NFA 的相关实现的类 NFAState, NFA, Nfa2Dfa 以及 Str2Nfa，其类图如下：

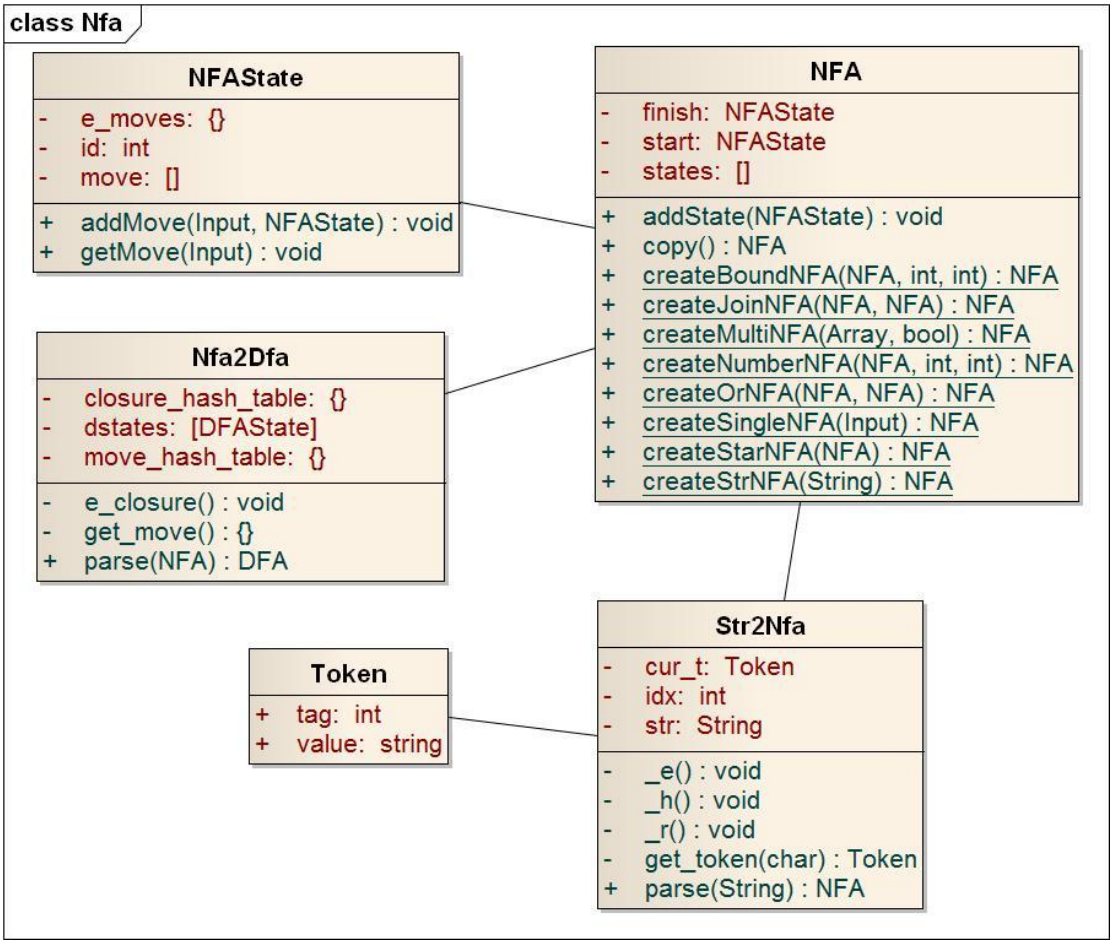

图 4.3 NFA 模块类图

模块类详细描述

- NFAState 类，继承自状态机状态基类 State，实现了 NFA 状态的定义和相关函数成员。e_moves 专门为 ε 转移保存了转移规则，moves 为其它输入保存了转移规则，addMove 函数添加一个输入 input 对应的转移状态，getMove 得到一个输入 Input 对应转移到的 NFAState
- NFA 是不确定有限状态机的抽像数据类型，其属性 start 和 finish 分别指向该 NFA 的开始和结束状态，states 保存了状态机的所有状态。函数 addState 为 NFA 添加一个 NFAState，copy 会返回一个当前 NFA 的拷贝。除此之外，还有以 create 开头的静态公开成员函数，这些函数实现了 NFA 的相关计算，包括以一个输入生成基本 NFA 的 createSingleNFA，将两个 NFA 进行连接运算的 createJoinNFA，将两个 NFA 进行选择运算的 createOrNFA，对 NFA 进行闭包运算的 createStarNFA，从一个连续字符串生成 NFA 的 createStrNFA，对 NFA 进行重复的 createNumberNFA。
- Nfa2Dfa 类，将 NFA 转换成 DFA。唯一的公开函数 parse 接收一个 NFA，将其转换成等价的 DFA。
- Str2Nfa 类，将正则表达式字符串转换成 NFA。唯一的分开函数 parse 接收一

个正则表达格式的字符串，使用自顶向下的递归进行语法分析，调用 NFA 类的静态计算函数，最终生成描述该正则表达式的 NFA。

4.2.3 DFA 模块—Dfa

模块类图

Dfa 模块包括 DFA 相关的实现类 DFAState, DFA, Dfa2Src, DfaMinimize，其类图如下：

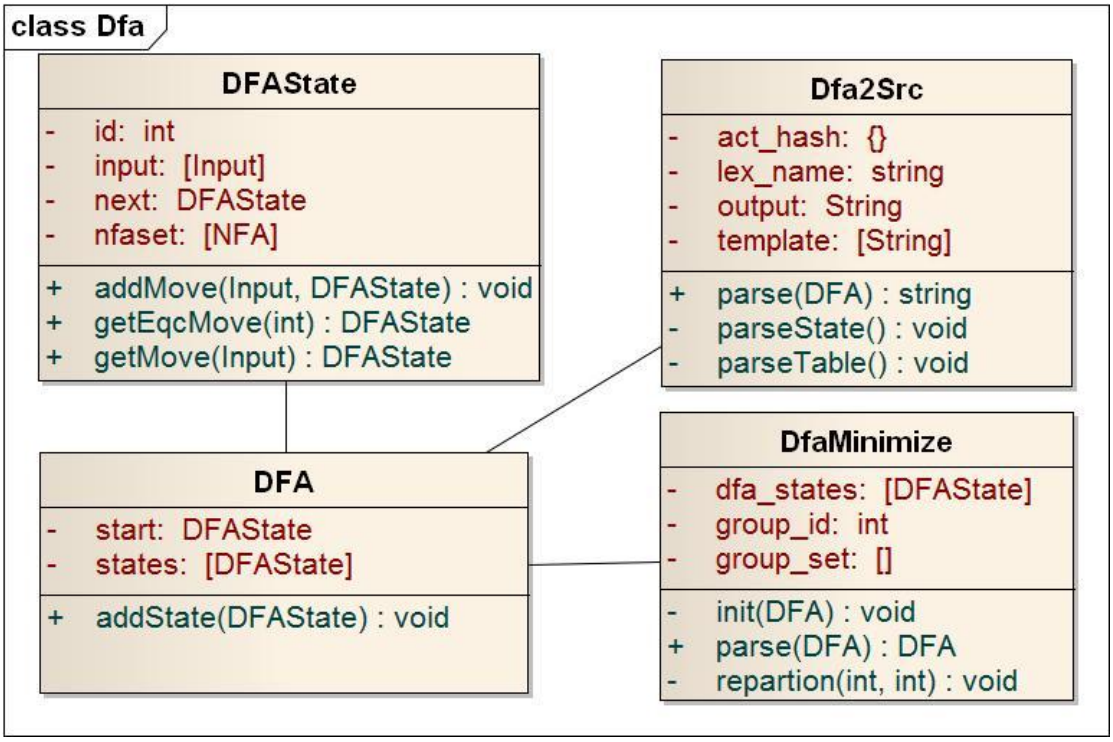

图 4.4 DFA 模块类图

模块类详细描述

- DFAState 类，继承自状态机状态基类 State，实现了 DFA 的状态的定义和相关函数。input 和 next 两个数组分别储存状态的输入和该输入对应的转移状态，nfaset 保存在 NFA 转换成 DFA 过程中该 DFA 对应的 NFA 状态集。addMove 函数添加一个输入 input 对应的转移状态，getMove 得到一个输入 Input 对应转移到的 NFAState，getEqcMove 是 DFAState 特有的函数，通过一个输入字符的等价类值直接返回一个转移状态。
- DFA 类，对确定的有限状态自动机的抽像数据结构，start 指向该 DFA 的开始状态，states 保存该 DFA 的所有状态集合。addState 函数添加一个 DFAState。
- DfaMinimize 类，对 DFA 进行状态最小化[1]。其唯一的公开函数接受一个 DFA，将其状态数进行压缩以达到最小化，最后返回一个最小化状态后的新的 DFA。
- Dfa2Src 类，从 DFA（已经经过压缩和转换成线性表）生成词法分析器源码。

其唯一公开函数是 parse，接受一个 DFA，生成对应词法分析器源码的字符串。

4.2.4 线性表和等价类管理模块—Table 模块

模块类图

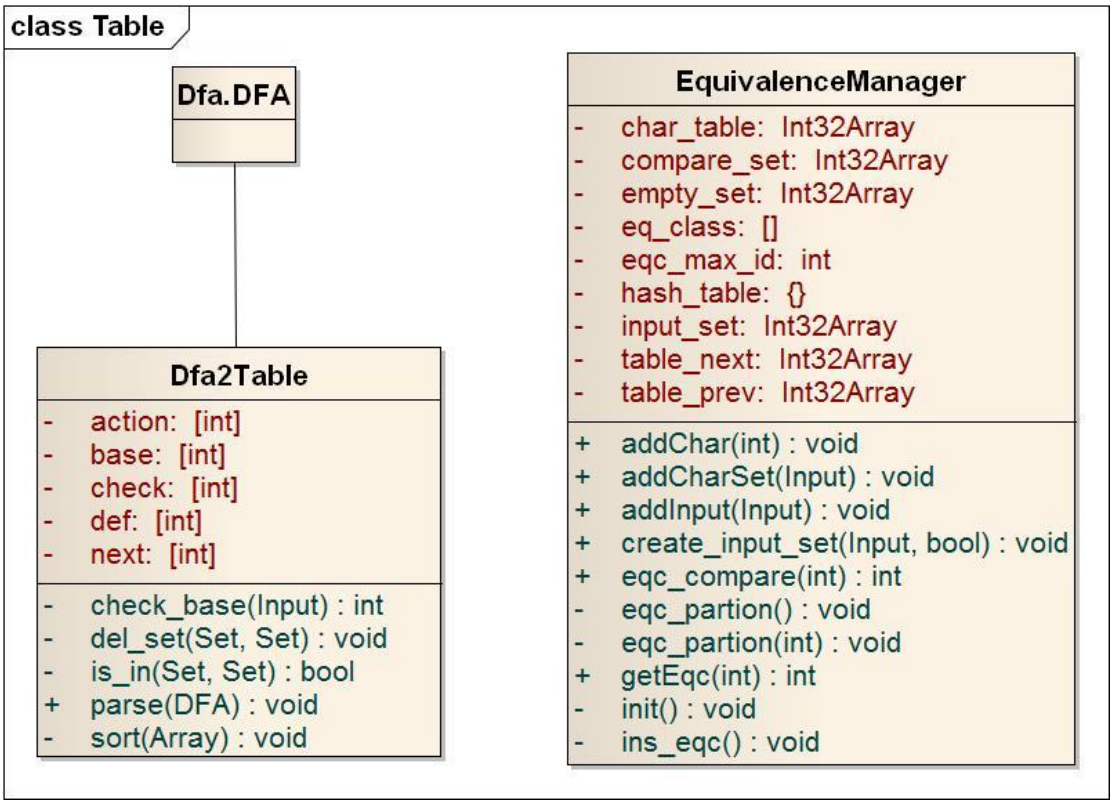

图 4.5 Table 模块类图

模块类详细描述

- Dfa2Table 类，将 DFA 转换为线性表，即五个数组 default, base, check, next, action[1]。转换的过程参考了引文[4]、引文[6]以及 Flex[14]的源码。这四个表会传给 DFA 模块的 Dfa2Src 类转换成源代码。
- Equivalence Manager 类，对输入字符进行等价类[11]管理。实现的过程参考了 Flex[14]的源码。

4.2.5 实用集模块—Utility

模块类图

实用类模块包括一个辅助的实用类 Utility，该类只包含公开的静态函数，全部是实现的函数，其类图如下：

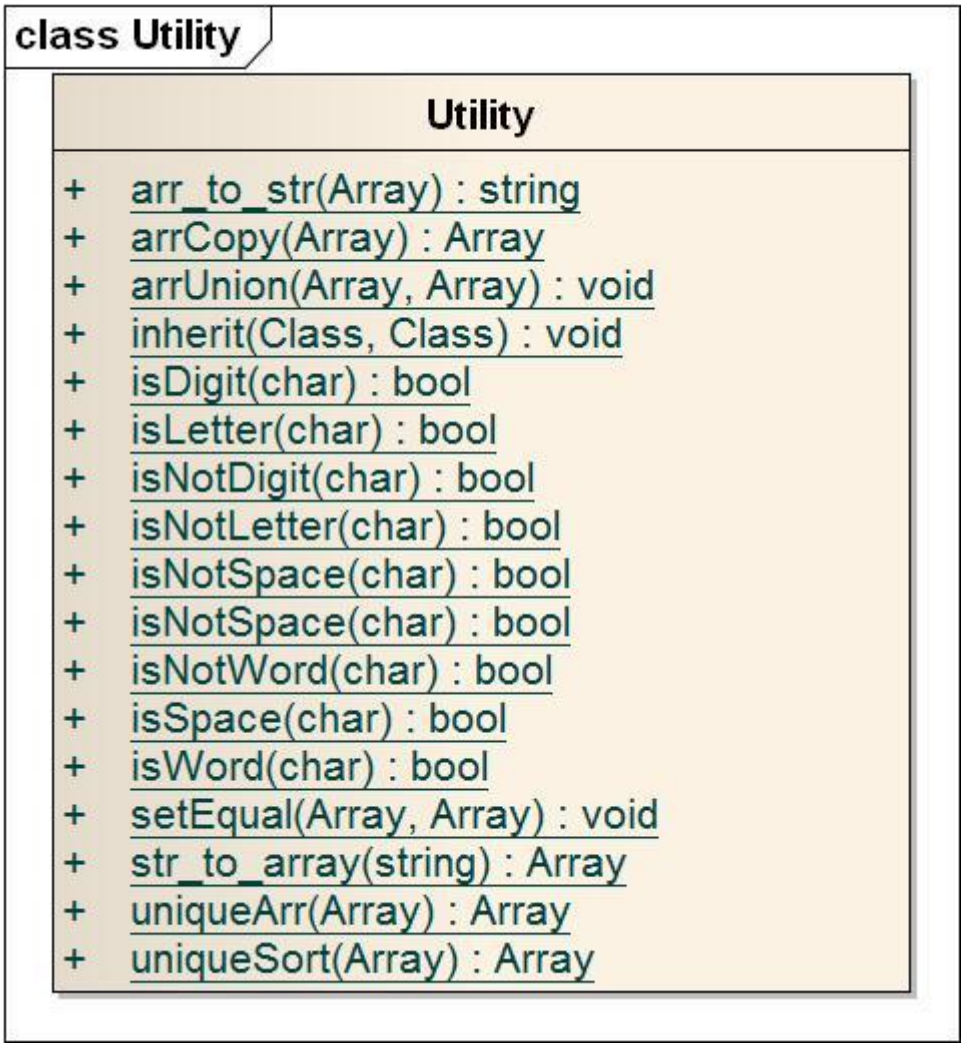

图 4.6 Utility 模块类图

模块类详细描述

类 Utility 是辅助类，其只包含用于辅助作用的静态函数。包括集合相关的判断处理函数：setEqual（判断两个集合是否相等），arrUnion（对集合求并集），uniqueArr（将数组元素唯一化，即转换成集合），uniqueSort（对数组进行排序并移除重复元素）；数组常用函数：arrCopy（复制数组），arr_to_str（将整数数组压缩成字符串），str_to_array（将压缩的字符串还原成整娄数组）；以及常用的字符判断函数 isDigit（是否是数字），isLetter（是否是字母）等等。

4.2.6 词法分析器源码模板模块—Template

模块类图

该模块的代码不是 AliceLex 的组成部分，而是 AliceLex 跟据词法规则生成的词法分析器源代码的模板。模块下的各个类，是使用在不同场景下的词法分析器的源代码的模板。其类图如下：

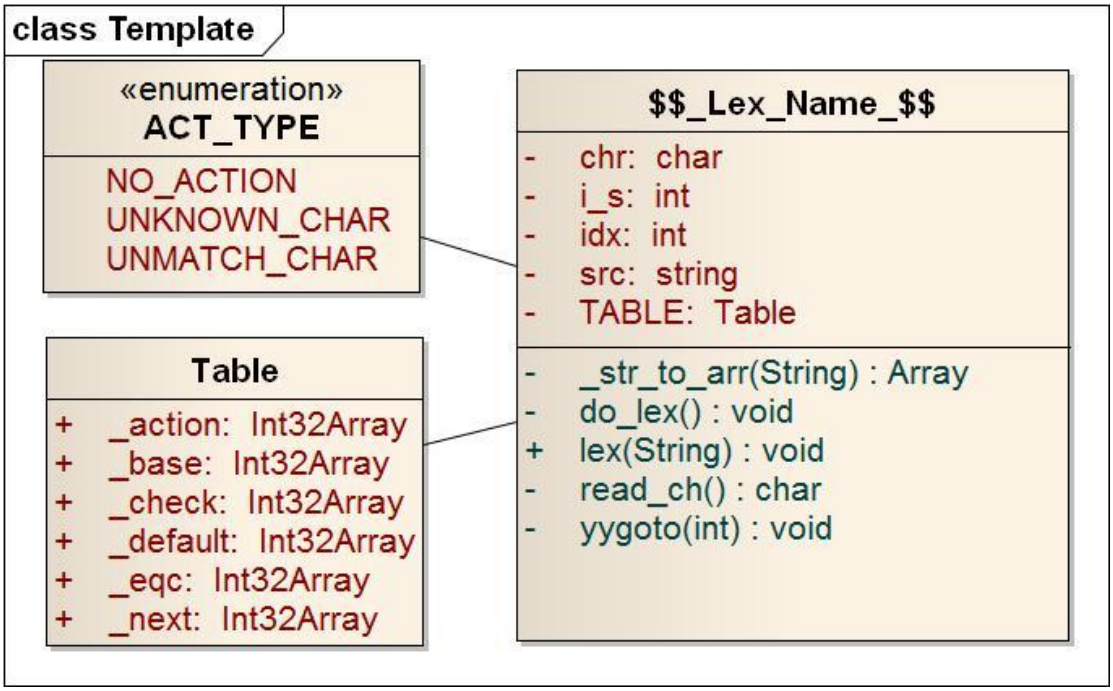

图 4.7 Template 模块类图

模块类详细描述

- ACT_TYPE 是枚举，指明当前词法分析到的位置的 Action 类型。NO_ACTION 表明当前没有定义动作，UNKNOWN_CHAR 表明遇到未知的字符，UNMATH_CHAR 表明当前字符不满足词法规则。如果当前 Action 是非三个枚举的其它值，则表明当前到达用户定义的动作，执行该值在 action 表中对应的动作。
- Table 是一个辅助类，保存当前词法分析器的六张表。分别是 default, base, check, next, action 和 eqc。其中 eqc 是保存的输入符对应的等价类。
- $$_LEX_NAME_$$ 是由 AliceLex 生成的词法分析器类。该 $$_LEX_NAME_$$的实际值由用户在词法规则文件中定义，最终模板中的该符号会被替换。其唯一的公开接口是 lex，接受一个待分析的文本，对其进行词法分析。yygoto 是在当前状态中进行跳转，借此函数可以实现带状态的词法分析。

4.3 算法的详细设计和实现

AliceLex 工具中使用了比较多的编译原理中词法分析相关的算法，并在 JavaScript 中得到完整实现，包括正则表达式字符串转换成 NFA、NFA 到 DFA 的转换、DFA 状态最小化、DFA 转换成线性表等。接下来会对这些关键算法进行描述，包括伪代码描述以及关键实现代码的展现。

4.3.1 正则表达式构造 NFA

伪码描述[1]

算法主逻辑：使用自顶向下递归对表达式字符串 reg 进行语法分析，分解出组成它的子表达式，在这个过程中使用两个规则将语法分析的子表达式构造成 NFA。两个规则包括基本规则和归纳规则，基本规则处理不包含运算符的子表达式，而归纳规则根据一个给定表达式的直接子表达式的 NFA 构造出这个子表达式的 NFA。

基本规则：对于表达式 ε，构造下面的 NFA。

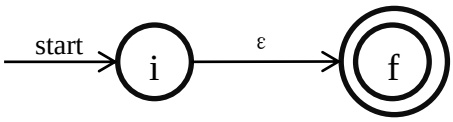

图 4.8 构造 ε 的 NFA

这里，i 是一个新的状态，也是这个 NFA 的开始状态；f 是另一个新的状态，也是这个 NFA 的接受状态。

对于字母表 E 中的子表达式 a，构造下面的 NFA。

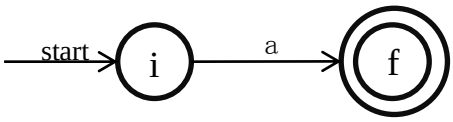

图 4.9 构造 a 的 NFA

同样，i 和 f 是新状态，分别是这个 NFA 的开始和接受状态。在基本规则中，对于 ε 或某个 a 的作为 reg 的子表达式的每次出现，都使用新状态分别构造出一个独立的 NFA。

归纳规则：

- 选择关系的运算，包括|运算符和[]运算符，比如 $r=s_1|s_2|\ldots|s_n$ 或 $r=[s_1\text{-}s_n]$，使用如下图示进行构造：

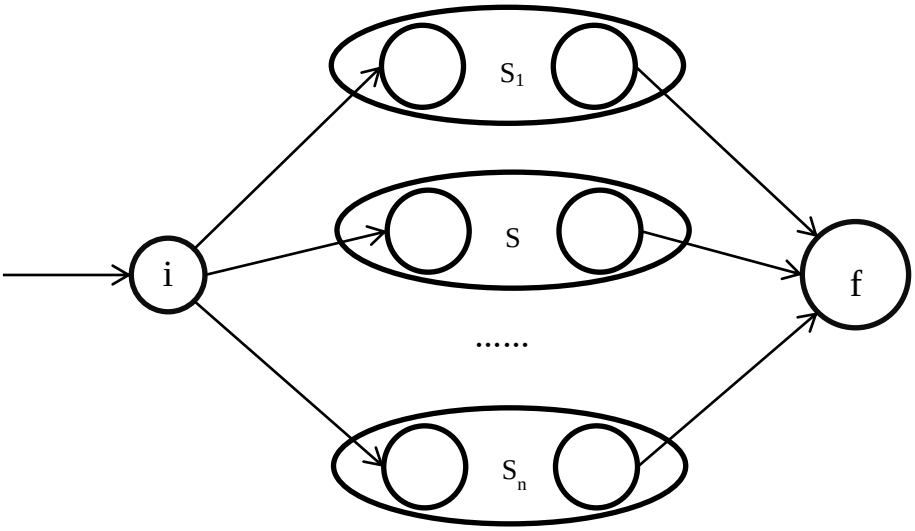

图 4.10 构造选择关系 NFA

其中 i 和 f 是新状态，分别是这个 NFA 的开始和接受状态。从 i 到 s_k(k=1…n) 的开始状态各有一个 ε 转换，从 s_k 的接受状态到 f 各有一个 ε 转换。

- 连接关系的运算，比如 $r=s_1s_2…s_n$，使用如下图示进行构造：

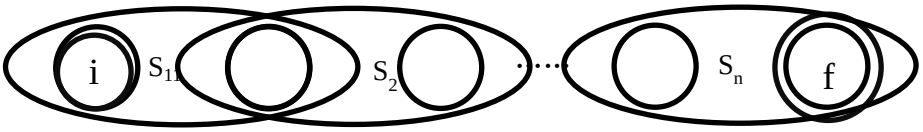

图 4.11 构造连接关系的 NFA

其中 i 是第一个 NFA 即 s_1 的开始状态，f 是最后一个 NFA 即 s_n 的接受状态。第 k 个 NFA 即 s_k 的开始状态和第 k+1 个 NFA 即 s_{k+1} 的接受状态合并为一个状态，合并后的状态拥有原来进入和离开合并前两个状态的全部转换。

- 闭包关系运算，即 $r=s^*$，使用如下图示进行构造：

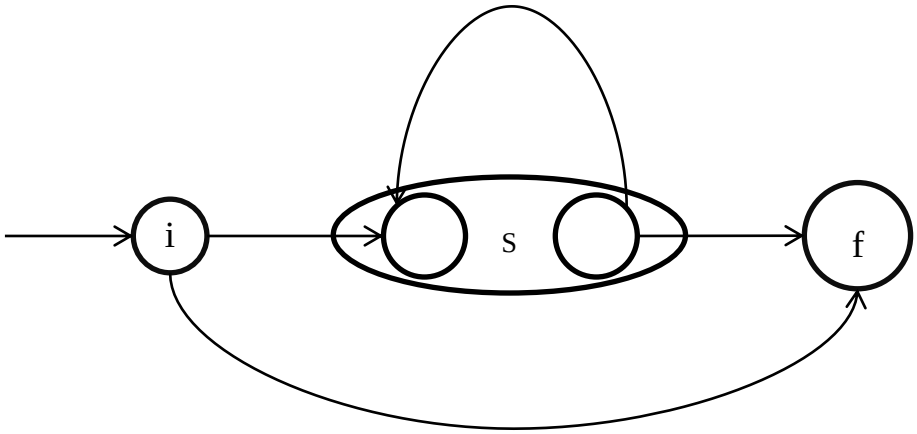

图 4.12 构造闭包关系的 NFA

其中 i 和 f 是两个新状态，分别是新 NFA 的开始和接受状态。

- 对于其它扩展关系运算，包括+，？，{}等都转换成上面的 3 种运算构造。

关键实现代码

对正则表达式字符串的语法分析由 Nfa 模块中的 Str2Nfa 类完成，以下是其中的自顶向下递归中的部分关键代码：

```
//处理或运算，即 r=s|f
_r : function() {
                var nfa1, nfa2;
                nfa1 = this._e();
                while(true) {
                    if(this.cur_t.tag === N.Tag['|']) {
                        //$.dprint('|')
                        this.read_token();
                        nfa2 = this._e();
                        nfa1 = N.NFA.createOrNFA(nfa1, nfa2);
                        //$.dprint(nfa1);
                    } else
                        break;
                }
                //$.dprint(nfa1);

                return nfa1;
            },
//处理连接运算，即 r = sf
            _e : function() {
                var nfa1 = this._t();
                var nf2;
```

```
                while(true) {
                    if(this.cur_t.tag !== N.Tag['|'] && this.cur_t.tag !== N.Tag[')'] &&
this.cur_t !== N.Token.EOF) {
                        nfa2 = this._t();
                        nfa1 = N.NFA.createJoinNFA(nfa1, nfa2);
                        //$.dprint(nfa1);
                    } else
                        break;
                }
                //$.dprint(nfa1);
                return nfa1;
            },
```

表 4.1 正则表达式语法解析关键代码

在以上代码中，只对正则字符串进行了语法分析和构造，更具体的生成 NFA 实例的职责交由 Nfa 模块中 NFA 类的静态函数实现，以下是其中两个函数的示例：

```
/**
             * 将两个 nfa 进行并运算，返回一个新的 nfa。
             * r=s|t
             */
            createOrNFA : function(nfa1, nfa2) {

                var s = new N.NFAState();
                var f = new N.NFAState(true);
                s.addMove(C.Input.e, nfa1.start);
                s.addMove(C.Input.e, nfa2.start);
                nfa1.finish.isAccept = false;
                nfa1.finish.addMove(C.Input.e, f);
                nfa2.finish.isAccept = false;
                nfa2.finish.addMove(C.Input.e, f);
                var rtn = new N.NFA(s, f);
                rtn.addState(nfa1.states);
                rtn.addState(nfa2.states);
                rtn.addState(s, f);
                return rtn;

            },
            /**
             * 将两个 nfa 进行连接运算，返回一个新的 nfa。
             * r=st
             */
            createJoinNFA : function(nfa1, nfa2) {
                // $.dprint('*********');
```

```
            // $.dprint(nfa1);
            // $.dprint(nfa2);
            var rtn = new N.NFA(nfa1.start, nfa2.finish);
            nfa1.finish.isAccept = false;
            //合并 nfa1 的接受状态和 nfa2 的开始状态为同一个状态
            if(nfa2.start.move)
                nfa1.finish.addMove(nfa2.start.move[0], nfa2.start.move[1]);
            for(var i = 0; i < nfa2.start.e_moves.length; i++)
                nfa1.finish.addMove(C.Input.e, nfa2.start.e_moves[i]);

            //将 nfa1 的状态和 nfa2 状态增加到新的 nfa 中，因为 nfa1 的开始态和 nfa2 开始态已经合并，
            //不需要将 nfa2 的开始态添加。
            rtn.addState(nfa1.states);
            for(var i = 0; i < nfa2.states.length; i++) {
                if(nfa2.states[i] !== nfa2.start)
                    rtn.addState(nfa2.states[i]);
            }
            // $.dprint(rtn);
            // $.dprint('**********');
            return rtn;
        },
```

表 4.2 构造 NFA 的关键代码

4.3.2 NFA 转换为 DFA

伪码描述[1]

算法主逻辑

```
定义 DFAStates， DFATran
一开始，e-closure(s0)是 DFAStates 中唯一状态，且它未加标记；
while(在 DFAStates 中有一个未标记的状态 T){
    给 T 加上标记；
    for(每个输入符号 a){
        U=e-closure(move(T, a));
        if(U 不在 DFAStates 中){
            将 U 加入到 DFAStates 中，且不加标记；
        }
        DFATran[T, a]=U;
    }
}
返回状态集 DFAStates 和其上的转换函数 DFATran，则是构造而成的
```

DFA

图 4.13 NFA 子集转换算法伪代码

主逻辑中使用到以下函数：

函数	描述
e-closure(s)	能够从 NFA 的状态 s 开始只通过 ε 转换到达的 NFA 状态集合
e-closure(T)	能够从状态集 T 中某个状态 s 开始只通过 ε 转换到达的 NFA 状态集合，即$\cup_{s\epsilon T}\ \varepsilon - \mathrm{closure}(s)$
Move(T, a)	能够从 T 中某个状态 s 出发通过标号为 a 的转换到达的 NFA 状态的集合

图 4.14 子集转换算法关键函数

其中 e-closure(s)和 move(T, a)只需要使用简单的线性遍历可以得到，e-closure(T)使用如下描述算法得到：

```
将 T 的所有状态压入栈 stack 中；
将 e-closure(T)初始化为 T；
while( stack )非空 {
    将栈顶元素 t 弹出栈；
    for (每个满足如下条件的 u：从 t 出发有一个标号为 ε 的转换到达状态 u)
    {
        if(u 不在 e-closure(T)中)
        {
            将 u 加入到 e-closure(T)中;
            将 u 压入到栈中；
        }
    }
}
```

图 4.15 e-closure 函数伪代码

关键实现代码

```
while( S = this.get_untag_state()) {
        S.tag = true;

        for(var i = 0; i < eqc.length; i++) {

            var mv = this.get_move(S.nfaset, eqc[i]);

            if(!mv)
                continue;
            //$.dprint(mv.move);
```

```
                    /*
                     * 计算 move 集合的 e_closure 集合
                     */
                    var ec = this.e_closure(mv);

                    /**
                     * 从已经存在的 e_closure 集中查找该 e_closure 对应的 dfa 状态，
                     * 如果没有打到，说明该 dfa 状态还示存在，需要新建
                     */
                    var dfa_state = this.closure_hash_table[ec.hash_key];
                                    if(!dfa_state) {
                        dfa_state = new D.DFAState(ec.is_accept, U._n.get());
                        dfa_state.nfaset = ec.closure;

                        this.add_dfa_state(dfa_state, ec.hash_key);
                    }

                    var eqc_index = Alice.CharTable.getEqc(eqc[i]);
                    S.addMove(eqc_index, dfa_state);
                }
            }
```

表 4.3 NFA 子集转换算法关键代码

e-closure 函数实现如下：

```
            if(this.move_hash_table[mv.hash_key])
                return this.move_hash_table[mv.hash_key];
            var T = mv.move, is_accept = false, stack = [], e_c = [], e_id = [], new_closure =
null, hash_key = "";

            for(var i = 0; i < T.length; i++) {
                stack.push(T[i]);
                e_c.push(T[i]);
                e_id.push(T[i].id);
                /*if(T[i].isAccept === true){
                  is_accept = true;
                  }*/
                is_accept |= T[i].isAccept;
            }
            while(stack.length > 0) {
                var t = stack.pop();
                //$.dprint(t);
                var u = t.e_moves;
                //$.dprint(u);
                for(var i = 0; i < u.length; i++) {
```

```
                    if(e_id.indexOf(u[i].id) === -1) {
                        e_c.push(u[i]);
                        stack.push(u[i]);
                        e_id.push(u[i].id);
                        is_accept |= u[i].isAccept;
                    }
                }
            }
            /**
             * 计算哈希值，同样是对集合中每个状态的 id 进行排序后连接成字符串
             */
            hash_key = e_id.sort(function(a, b) {
                return a > b;
            }).join(",");
            is_accept = is_accept ? true : false;
            new_closure = {
                hash_key : hash_key,
                closure : e_c,
                is_accept : is_accept
            };
            return new_closure;
        },
```

表 4.4 e-closure 函数实现代码

4.3.3DFA 状态最小化

伪码描述[1]

DFA 状态最小化算法的目的在于将任意 DFA 转化为等价的状态最少的 DFA，其作用是使得最终生成的线性表尽可能小，从而提升生成的词法分析器程序的空间效率。对于输入的 DFA（接下来用 D 表示），使用如下方法实现对其最小化：

1. 首先构造初始划分∏，构造原则是所有非接受状态为一组，其它所有接受状态各自为一组。
2. 应用如下过程来构造一个新的划分$\prod_{new}$。

```
最初，令∏new=∏；
for(∏中的每个组 G){
    将 G 分划为更小的组，使得两个状态 s 和 t 在同一组中当且仅当对于所有的输入符号 a，状态 s 和 t 在 a 上的转换都到达∏中的同一组；
    在∏new 中将 G 替换为对 G 进行分划得到的那些小组；
}
```

3. 如果Π_{new}=Π，令Π_{final}=Π并接着执行步骤 4；否则用Π_{new}替换Π并重复步骤 2。
4. 在分划Π_{final}的每个组中选取一个状态作为该组的代表。这些代表构成了状态最小 DFA 的状态。该 DFA（接下来用 D'表示）的其它部分按如下步骤构建：
   a) D'的开始状态是包含了 D 的开始状态的组的代表；
   b) D'的接受状态是那些包含了 D 的接受状态的组的代表。
   c) 令 s 是Π_{final}中某个组 G 的代表，并令 D 中在输入 a 上离开 s 的转换到达状态 t。令 r 为 t 所在组 H 的代表，那么在 D'中存在一个从 s 到 r 在输入 a 上的转换。

关键实现代码

```
repartion : function(f, t) {
        for(var i = t; i > f; i--) {
            for(var j = f; j < i; j++) {
                if(this.group_id_tmp[j] < this.group_id_tmp[j + 1]) {
                    this.swap_index_and_id(j, j + 1);
                }
            }
            //$.aprint(arr);
        }
        this.group_id[f] = this.max_group_id;
        for(var i = f + 1; i <= t; i++) {
            if(this.group_id_tmp[i] < this.group_id_tmp[i - 1]) {
                this.group_set_new[this.max_group_id] = i - 1;
                this.group_set_new[this.size]++;
                this.max_group_id++;
            }
            this.group_id[i] = this.max_group_id;
        }
        this.group_set_new[this.max_group_id] = t;
        this.group_set_new[this.size]++;
        this.max_group_id++;
    },
parse : function(dfa) {
        var eqc = Alice.CharTable.eq_class;
        var debug = 0;
        this.init(dfa);
        outer:
```

```
while(true && debug < 10000000) {
    for(var i = 0; i < eqc.length; i++) {
        if(debug++ > 10000000) {
            break outer;
        }
        this.get_group_id_tmp(Alice.CharTable.getEqc(eqc[i]));
        this.get_group_set_new();
        if(!this.is_group_set_same()) {
            this.swap_group_set();
            //this.output();
            continue outer;
        }
    }
    if(this.is_group_set_same())
        break;          }
D.DFAState.__auto_id__ = 0;
var new_size = this.group_set[this.size];
var new_states = new Array(new_size);
for(var i = 0; i < new_size; i++) {
    new_states[i] = new D.DFAState(i.toString());
}
var new_start_index = this.group_id[dfa.start.__minimize_id__];
var new_start = new_states[new_start_index];
for(var i = 0; i < new_size; i++) {
    var old_s = this.dfa_states[this.group_set[i]];
    for(var j = 0; j < old_s.input.length; j++) {
        var new_next = this.group_id[old_s.next[j].__minimize_id__];
        //$.dprint("new next %d",new_next);
        new_states[i].addMove(old_s.input[j], new_states[new_next]);
    }
}
for(var i = 0; i < this.accept_states.length; i++) {
    var gid = this.group_id[this.accept_states[i].__minimize_id__];
    new_states[gid].isAccept = true;
    if(!this.accept_states[i].action)
        continue;
    if(!new_states[gid].action)
        new_states[gid].action = this.accept_states[i].action;
```

```
            else if(new_states[gid].action !== this.accept_states[i].action){
                $.log("最小化 DFA 时出现问题，Action 丢失。");
            }
        }
        $.log("dfa minimized. %d states to %d states.", this.size, new_size);
        var new_dfa = new D.DFA(new_start, new_states);
        //new_dfa.startIndex = new_start_index;
        return new_dfa;
    }
```

表 4.5 DFA 状态最小化算法关键代码

第五章 AliceLex 的使用说明和示例

本章详细介绍了基于 JavaScript 的词法分析器自动生成工具 AliceLex 的使用说明，包括 AliceLex 项目的构建，引入和使用方法。同时会给出 lex 程序的经典示例在 JavaScript 上的实现，最后介绍基于 AliceLex 为核心的在线代码编辑器。

5.1 使用说明

5.1.1 项目配置

AliceLex 项目作为开源项目，托管在 GitHub[7]上，使用如下方法配置 AliceLex 的最新代码：

安装和配置 nodejs[8]

1. 安装和配置 github
2. 新建目录 alicelex，在此目录上执行命令：
3. 执行 git clone https://github.com/YuhangGe/alicelex
4. 执行 node make.js

在 alicelex 目录的根下会生成两个文件，alicelex.js 和 alicelex-node.js，其中 alicelex.js 是浏览器使用的前端代码，alicelex-node.js 是后端用于 nodejs 的代码。

5.1.2 使用方法

对于 alicelex.js，在浏览器端使用如下方法使用：

1. 通过<script>标签引入 alicelex.js
2. 这时候全局会有唯一的公开对象 Alice，通过调用其唯一的公开接口 doLex 函数将词法规则文本转换成词法分析器的源代码。具体如下：

```
var js_src = Alice.doLex(lex_text)
```

3. 其中 lex_text 是需要转换的源始词法规则文本，更详细的规则说明参见 5.1.3。返回值是 string 类型的字符串，是该规则对应生成的词法分析器的 JavaScript 源代码。
4. 使用 eval 函数执行 js_src，这样就会产生一个词法分析器可以直接使用。具体来说，如果 lex_text 文本中通过参数$name 定义的词法分析器的名字是 JSLexer，则可以使用如下代码执行分析过程：

```
var text = “这里是你要进行词法分析的文本”;
```

[7] Github 是目前使用广泛的开源托管平台，详情和配置方法参见：http://github.com

[8] nodejs 是本地的（而非浏览器）JavaScript 脚本解析引擎，详情和配置方法参见：http://nodejs.org

```
var lexer = new window.JSLexer();
lexer.lex(text);
```

对于 alicelex-node.js，相当于是本地的一个命令行软件，使用方法为：

1. 安装和配置 nodejs
2. 使用命令 node alicelex-node.js lex_file output_file

其中 lex_file 是词法规则文件名，output_file 是需要输出的词法分析器源代码文件。然后你可以在 html 文件中引入该 output_file，使用生成的词法分析器。

5.1.3 词法规则

基本框架

AliceLex 使用传统 lex 程序相似的基本框架来定义词法规则动作文本[3]：

```
{参数声明}
{规则声明}
$$
{规则动作定义}
$$
{全局函数}
```

图 5.1 AliceLex 词法规则动作文本结构

在当前版本中有两点与传统 lex 软件不一样，一是使用美元符代替了百分号，二是规则文本中暂时还没有支持注释。

参数声明

AliceLex 通过使用 ${参数名} {参数值} … 这样的格式来定义参数。目前已经实现的参数如下：

- $caseignore true/false
  生成的词法分析器是否忽略大小写，默认为 false
- $lexname name
  生成的词法分析器的名称，默认是 Daisy。该名称是一个对像实例的名称，通过 window.name 可以得到该实例。
- $template normal/editor/…
  生成的词法分析器的源代码模板，默认为 normal，生成可以在浏览器出使用的源代码。指定 editor 参数则会为在线代码编辑器 Daisy Editor[9]的代码高亮部分生成核心代码。
- $unicode true/false
  是否是 unicode 模式。Unicode 模式可以处理包括中文在内的所有字符，

[9] 关于 Daisy Editor 的详情参见 5.2.2 节

但会占用更大的内存空间。默认是 false，会把所有非 ascii 码字符交给错误处理函数处理。

规则声明

规则声明部分规则如下：

rule_name　expression

其中 rule_name 是定义的规则名称，expression 是该规则对应的正则表达式。正则表达式的规则跟 JavaScript 语言中的正则表达式[12]含义一致，支持+*[]?^等操作符。但有一些不同说明如下：

- 预定义的表达式：
  \d　数字（0-9）
  \D　非数字
  \w　字符（a-z，A-Z 和字符_）
  \W　非字符
  \s　空白符（\n\t\v）
  \S　非空白符
  \a　字母（a-z，A-Z）
  \A　非字母
  \u　大写字母（A-Z）
  \U　非大写字母
  \l　小写字母（a-z）
  \L　非小写字母
- 转义符：
  除了上面指定的预定义表达式，其它“\”符号都理解成跟 javascript 语言中一致的转义　　符，比如\n，\t 等等。同时，" \ [ ] ^ - ? . * + | ( ) / { } % < > 这些字符在正则表达式中　专门含义，如果需要作为字符串使用必须使用转义符。
- 字符串：
  使用引号包含的部分，全部当作普通字符串处理。比如 \d“\d”这个正则规则代表一个数字后面紧跟着\d 这个字符串。注意在引号包含的字符串不转义。
- 使用已定义的规则：
  如果需要使用已经定义的规则，则在正则表达式中使用大括号｛｝将其包含。如果不使用｛｝包含，则会理解成字符串而不是已经定义的规则。比如如下两个规则声明：

  DIGIT　　\d

  NUMBER　{DIGIT}+

  其中的 NUMBER 会被理解成 \d+，即一个以上的数字。但如果写成如下：

DIGIT \d

NUMBER DIGIT+

则其中的 NUMBER 会被理解成“DIGI”T+，即一个字符串 DIGI 后面跟一个以上的字符 T

- 不包含的规则：
  在 AliceLex 的当前版本中，正则表达式不支持^和$符作为起始和结束的规则，不支持/字符回退规则。

规则动作定义

规则声明如下：

{rule_name} {rule_action}

其中 rule_name 必须是在词法声明块定义过的规则名，rule_action 是对应的动作，即相应匹配的处理代码。Rule_action 的代码，如果是单行代码可以不用{ }包含，如果是多行代码，必须使用 { } 包含。

在规则动作定义部分，动作处理代码中可以直接使用的变量包括：yylen 和 yytxt。这两个变量的名称借用了经典 lex 程序的变量名，其含义也跟经典 lex 程序一致，分别代表当前匹配的字符串长度和匹配的字符串。

全局函数

全局函数部分定义词法分析器的全局函数，包括词法分析开始，词法分析错误处理，词法分析结束。格式如下：

${func_name} {function}

其中{func_name}是函数名，目前支持以下函数：

- $construct
  词法分析器的构造函数执行完成后调用此处代码。用来执行额外的构造函数任务，比如添加额外的成员变量。
- $start
  词法分析器会在用户调用 lex 函数执行词法分析之前调用此处代码。
- $finish
  词法分析器会在 lex 函数执行完，即词法分析结束之后调用。
- $error
  当词法分析器分析到错误时会调用

{function}是对应的代码，和规则动作定义部分的 action 一致，如果是单行代码可以不用 { } 包含，如果是多行代码需要使用 { } 包含。

歧义处理

AliceLex 对于歧义的处理同传统 lex 程序一致，即：

1. 首选最长的匹配。
2. 在匹配同样多字符的情况下，选择第一个给出的规则。

5.2 使用示例

5.2.1 lex 程序经典示例[10]

行数统计

```
$name Example1

NEW_LINE \n
OTHER     .

$$

NEW_LINE { line_number++; }
OTHER {}

$$

$start { line_number = 0; }
$finish { alert(“line count is:” + line_number);}
```

表 5.1 行数统计 lex 程序

带状态转移的词法分析

```
$name Example2

STR_QUOTE \”
STR_CHAR    [^\”]*
OTHER .
$$

STR_QUOTE { yygoto(STR);}
<STR> STR_CHAR { alert(“string is: “ + yytxt);}
<STR> STR_QUOTE {yygoto(DEFAULT);}

OTHER {/* do nothing */}

$$
```

[10] 在线演示的 demo 参见：http://lex.xiaoge.me

表 5.2 带状态转移的 lex 程序

5.2.2 Daisy Editor[11]

Daisy Editor 是一款基于 HTML5 的在线代码编辑工具，其使用 Canvas 绘制而不是像传统代码编辑器使用 DOM 元素构造，从而实现了海量代码文本时的流畅处理。同时，使用 AliceLex 作为其代码高亮部分词法分析器的生成工具，可以更灵活地支持更多语言的高亮，并且比传统在线代码编辑器具有更高的高亮渲染效率。以下是程序的示例截图：

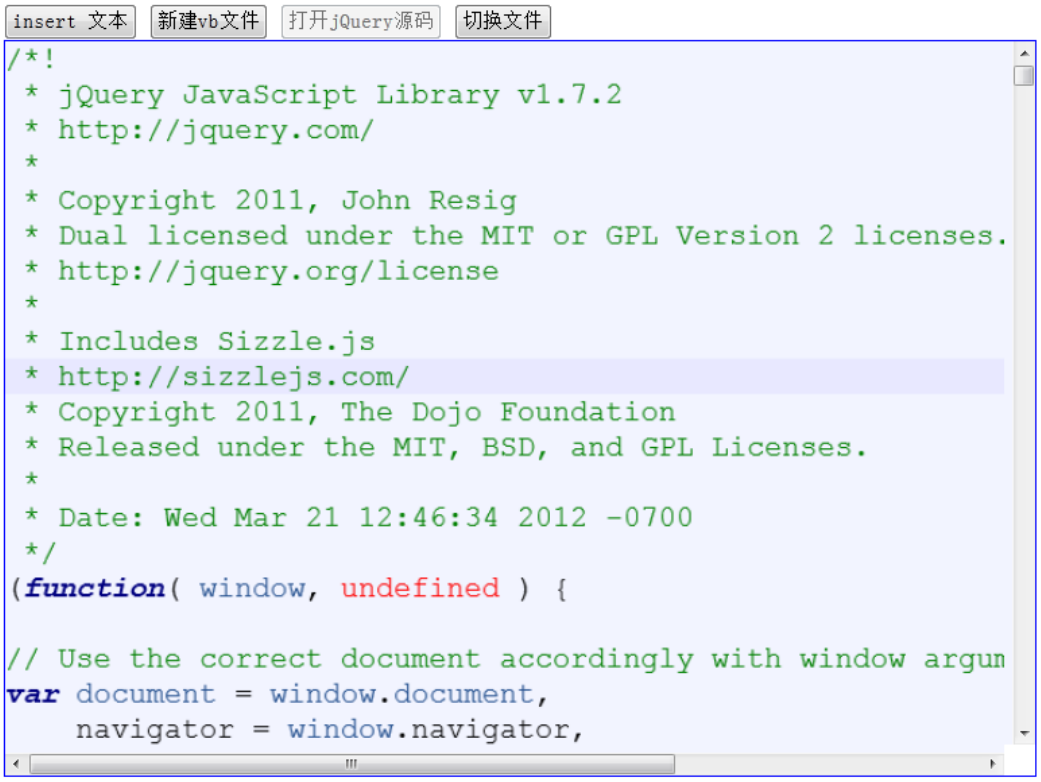

图 5.2 使用 Daisy Editor 编辑 jQuery 源代码截图

Daisy Editor 目前还处于原型阶段，还在进一步开发中。当前在网上还没有同类产品的出现，其使用场景主要在云计算平台中用作在线代码查看和编辑。这个项目的目的之一也是为了说明 JavaScript 语言平台上的词法分析器自动生成工具的使用场景和潜力。

为了说明 AliceLex 是如何提供高亮分析，展示如下 JavaScript 语言高亮的规则文本（限于篇幅，只处是部分代码）：

```
INT \d+
FLOAT \d*\.\d+
```

[11] 在线的演示参见：http://editor.xiaoge.me

```
NUM \-?({INT}|{FLOAT})
VALUE {NUM}|true|false|null|undefined
KEYWORD function|if|else|do|while|break|var|for|in|break|switch|case|return
ID [\w$][\w$\d]*
THREE_OPE "==="|">>>"
ONE_OPE "+"|"-"|"*"|"/"|"!"|"&"|"|"|"^"|"%"|"="
OPERATOR {THREE_OPE}|{TWO_OPE}|{ONE_OPE}
LINE_CMT_B "//"
REG "/"[^*/]([^\n/]|"\/")*"/"g?i?m?
STR_A_B \"
OTHER [.\n]

$$

VALUE {this.yystyle="value";}
KEYWORD {this.yystyle="keyword";}
PARAM {this.yystyle="param";}
OBJECT {this.yystyle="object";}
ID {this.yystyle="id";}
OPERATOR {this.yystyle="operator";}
LINE_CMT_B
{this.yystyle="comment";this.yydefault="comment";this.yygoto(LINE_COMMENT);}
<LINE_COMMENT> LINE_CMT_E
{this.yystyle="comment";this.yydefault="default";this.yygoto(DEFAULT);}
<LINE_COMMENT> LINE_CMT_C {this.yystyle="comment";}
REG {this.yystyle="regexp";}
STR_A_B {this.yystyle="string";this.yydefault="string";this.yygoto(STRING_A);}
<STRING_A> STR_A_C {this.yystyle="string";}
<STRING_A> STR_A_D {this.yystyle="string";}
<STRING_A> STR_A_E
{this.yystyle="string";this.yydefault="default";this.yygoto(DEFAULT);}
STR_B_B {this.yystyle="string";this.yydefault="string";this.yygoto(STRING_B);}
<STRING_B> STR_B_C {this.yystyle="string";}
<STRING_B> STR_B_D {this.yystyle="string";}
<STRING_B> STR_B_E
{this.yystyle="string";this.yydefault="default";this.yygoto(DEFAULT);}
OTHER {this.yystyle="default";}

$$
```

表 5.3 Daisy Editor 使用的词法高亮分析 lex 示例

第六章 总结与展望

5.1 总结

词法分析器生成工具（LEX）在生活生产中有着广泛的应用，极大简化了词法分析的工作。随着 web2.0，Html5 的发展，特别是 windows 8 的 html5 开发模式的推动，以及 NodeJS 的日益健壮，使用 JavaScript 语言进行词法分析的场景越来越多，开发一个基于 JavaScript 语言的词法分析器生成工具愈发必要。本文设计并实现了 JavaScript 版本的 LEX 工具，并将其作为托管在 GitHub 上的开源项目，以期望项目能够不段得到发展和完善。

本文的主要工作：

1. 介绍了 Web2.0 的发展现状，阐述了编译原理，尤其是词法分析在 JavaScript 上面的应用前景。

2. 介绍了编译原理的相关理论知识，词法、文法的相关概念。包括正则表达式，NFA，DFA，以及相关的算法理论。

3. 介绍了词法分析器自动生成工具和其实现的难点要点，展现当前 Lex 工具的现状。

5. 详细介绍了基于 JavaScript 的词法分析自动生成工具 AliceLex 的设计与实现，展示了该工具的使用典示例。

5.2 展望

AliceLex 工具作为 JavaScript 语言平台上的词法分析器生成工具，基本实现了传统 LEX 程序的主要功能：从词法规则动作文本生成对应词法分析器源代码，同时支持包括大小写敏感等主要参数配置，支持带状态的词法分析，支持浏览器前端和 NodeJS 两个平台。但是，目前 AliceLex 工具还不够完善，主要包括以下一些方面：

1. 规则文本不支持注释。
2. 规则不支持向前看运算符（即正则表达式中的 r_1/r_2 模式）。
3. 对规则文本和正则表达式字符串进行解析的过程使用的是手工编写的递归实现，代码组织比较乱。
4. 作为开源项目，缺乏相应的推广页面和文档。

在以后的工作中，对于本工具，还需要进行进一步的开发和完善，包括将功能进一步完善，优化代码结构，完善开源项目的推广和文档。

同时，就像传统 Lex[3]和 Yacc[5]的关系，开发与 AliceLex 配合使用的 JavaScript 语言上的语法分析器生成工具是本项目更为长远的一个展望。

参考文献

[1] Alfred V.Aho 等著,赵建华、郑滔等译,编译原理(第 2 版),北京.机械工业出版社,2009/1,页 28-120.

[2] Alexander Meduna 著,杨萍、王生源等译,编译器设计基础,北京.清华大学出版社,2009/4,页 19-68

[3] M. E. Lesk and E. Schmidt, Lex − A Lexical Analyzer Generator, Bell Laboratories, Murray Hill, New Jersey 07974

[4] Jun-Ichi Aoe, Katsushi Morimoto, Takashi Sato, An efficient implementation of trie structures, 30 OCT 2006

[5] Johnson, S. C. YACC-Yet another compiler-compiler. Bell Lab. NJ. Computing Science Technical Report 32. pp.1-34. 1975.

[6] An Implementation of Double-Array Trie, http://linux.thai.net/~thep/datrie/datrie.html

[7] Nondeterministic finite automaton, http://en.wikipedia.org/wiki/Nondeterministic_finite_automaton

[8] Deterministic finite automaton, http://en.wikipedia.org/wiki/Deterministic_finite_automaton

[9] Regular Expression, http://en.wikipedia.org/wiki/Regular_expression

[10] Lex(Software), http://en.wikipedia.org/wiki/Lex_(software)

[11] 等价类, http://zh.wikipedia.org/wiki/等价类

[12] 揭开正则表达式的神秘面纱, http://www.regexlab.com/zh/regref.htm

[13] 李小龙, 毛文林, 管道-过滤器模式的软件体系结构及其设计 Pipe-Filter Software Architecture and its Design 学术期刊 计算机工程与应用 COMPUTER ENGINEERING AND APPLICATIONS 2003 年第 39 卷第 35 期

[14] The Fast Lexical Analyzer, http://flex.sourceforge.net/

致谢

时光如白驹过隙，转眼间本科的时光已尽。在过去的四年里，感谢南京大学和南京大学软件学院的培养。四年的本科生涯，让我对软件工程有了系统的认识也具有了足够的实践经验。感谢软件学院给我提供的良好的软硬件条件，感谢软件学院勤劳善良的老师们的谆谆教导。

特别要感谢我的论文导师，南京大学软件学院的葛季栋老师，在论文开题和论文撰写的过程中对我的关心和指导。由于他的耐心提醒，让我很好的控制了项目实践和论文撰写的进度，并在理论研究上给了我很大帮助。感谢他在我修改论文的过程中，一次又一次细心认真的审阅。同时也感谢他在生活和学习中对我的帮助和指导。

在此要感谢郑涛老师在《软件构造》课程中讲述的编译原理对我起到的启蒙作用，使得我对编译原理产生了较浓兴趣，也才能自己去学习和体悟编译原理中的各种算法，并将其在毕业设计中一一实现。

感谢 209 宿舍的三个哥们，以及后来的 211 的众多舍友，是你们的关心支持和帮助，让我在大学四年里生活愉快而幸福，才能更好地投入到学习之中，才能专注地去研究和学习自己感兴趣的内容。

另外，衷心感谢一直以来默默支持关心我的家人和朋友们。