拓扑图方案

目录

[目标 1](#_Toc850259330)

[NetDSL语言设计 1](#_Toc52507924)

[接口 3](#_Toc517470088)

[实现方法 4](#_Toc2023381234)

### 目标

通过自定义语言实现网络拓扑图的描述，前端解释后转换为GoJS的模型，并支持在页面上预览。这个自定义语言命名为NetDSL，不包含排版的描述，而是让GoJS通过层级排版算法自动排版。

### NetDSL语言设计

* 关键字是node edge group endgroup，关键字检测的时候部分大小写，其他字符区分大小写
* 以//开头的行认为是注释，正行会给忽略
* 节点和边的定义都是一行完成：

|  |
| --- |
| node 节点ID 节点描述  edge 边ID 源节点ID 目标节点ID 边描述 |

节点描述和边描述可以不写

边的源节点和目标节点必须在所在行之前定义

* 分组则以group开头的行开始，直到遇到endgroup开头的行结束：

|  |
| --- |
| group 分组ID 分组描述  节点ID ...  节点ID ...  ...  endGroup |

* 每一行检测定义后多余的字符将被丢弃。
* 解析器将检查出以下错误

|  |  |
| --- | --- |
| 错误类型 | 错误代码 |
| 没有以关键字node/edge/group/endgroup开头 | 1 |
| 节点ID与分组ID重复 | 2 |
| 边的源节点找不到,边的目标节点找不到,分组中的节点找不到 | 3 |
| 节点参数不完整,边参数不完整,分组参数不完整 | 4 |
| 节点ID重复,边ID重复,分组ID重复 | 5 |
| 分组中的节点已经指派到另外一个分组 | 6 |

* 解析器将发出以下警告：

|  |  |
| --- | --- |
| 警告类型 | 错误代码 |
| 多余的字符被丢弃 | 7 |
| 分组包含零个节点，丢弃 | 8 |
| 另一条边包含相同的源和目标节点 | 9 |

拓扑图定义的例子(这个例子覆盖了以上所有可能的错误)：

|  |
| --- |
| node node1 label1 nodeIgnores  node node2  node node3 label3  edge edge1 node1 node2 label2 edgeIgnores  edge edge2 node1 node3  edge edge3 node1 node3 edgeLabel3  group group1 label4 groupIgnores  node1 node2  node3 node4  endGroup endGroupIgnores  //invalid lines followed  node  node node1  node group1  edge  edge edge1  edge edge4 node1  edge edge4 node node  group  group node1  node1  endGroup  group group1  node1  endGroup  group group2  node1  endGroup  group group3  endGroup  group group4  node3 |

解析器将返回以下错误消息（格式为：错误等级 | 错误代号 | 错误位置 错误消息）：

|  |
| --- |
| Warning | 7 | (1,19) : discard token nodeIgnores  Warning | 7 | (4,31) : discard token edgeIgnores  Warning | 9 | (6,1) : edge with same source and target already exist edge3  Warning | 7 | (7,21) : discard token groupIgnores  Error | 3 | (9,7) : group node node4 not found  Warning | 7 | (10,10) : discard token endGroupIgnores  Error | 4 | (13,1) : insufficient node parameter  Error | 5 | (14,6) : duplicated node node1  Error | 2 | (15,6) : group id conflicts with node id: group1  Error | 4 | (16,1) : insufficient edge parameter  Error | 5 | (17,6) : duplicated edge edge1  Error | 4 | (17,1) : insufficient edge parameter  Error | 4 | (18,1) : insufficient edge parameter  Error | 3 | (19,12) : source node not found  Error | 3 | (19,17) : target node not found  Error | 4 | (20,1) : insufficient group parameter  Error | 6 | (22,1) : node node1 already belong to another group1  Warning | 8 | (21,1) : group node1 contains no nodes and discarded  Error | 5 | (24,7) : duplicated group group1  Error | 2 | (24,7) : group id conflicts with node id: group1  Error | 1 | (25,1) : expect node/edge/group..endgroup  Error | 1 | (26,1) : expect node/edge/group..endgroup  Error | 6 | (28,1) : node node1 already belong to another group1  Warning | 8 | (27,1) : group group2 contains no nodes and discarded  Warning | 8 | (30,1) : group group3 contains no nodes and discarded  Error | 6 | (33,1) : node node3 already belong to another group1  Warning | 8 | (32,1) : group group4 contains no nodes and discarded |

### 接口

实现的核心代码在NetDslParser.ts，关键接口如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 函数 | 参数 | 返回值 | 说明 |
| NetDslParser.Parse | 字符串：NetDSL语言定义的拓扑图，可以从网页输入，也可以从文件读取。 | 拓扑图和解析过程中检测到的错误 | 拓扑图Net包含三个数据：节点数组，边数组和分组数组。  节点Node的Group属性可能为空，其余属性均非空。Label可能是空字符串。  边Edge的所有属性均非空，Label可能是空字符串。  分组Group的所有属性非空，包含的节点数组NodeList包含节点数量大于零（节点数为零的分组解析时会被抛弃）。 |
| NetDslParser.GenerateGoJsModel | 拓扑图：从NetDslParser.Parse方法调用的结果（当然也可以人工造一个出来） | GoJS模型（NodeDataArray和LinkDataArray） | GoJS节点类型定义为：{key:string,isGroup?:boolean,group?:string};  GoJS边类型定义为：{from:string,to:string}; |

NetDslParser.ts通过Nodejs或者typescript.org/play网站转换为js后引入到项目使用。

NetDslParser.ts包含测试代码：Main函数和netDslStr变量，请使用前删除。

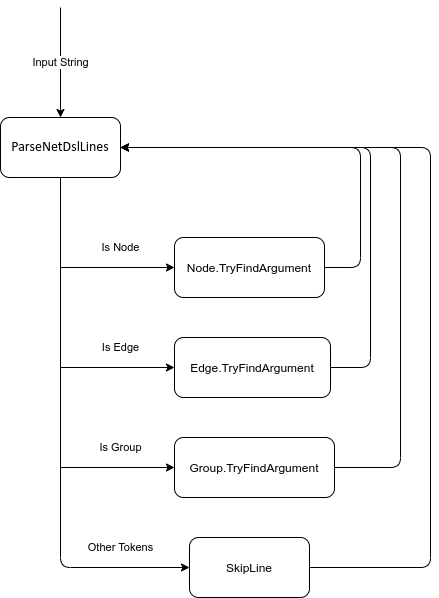
### 实现方法

源码在

<https://github.com/DragonLi/exercise/tree/master/ProjAsia/NetDSL/netdsl>/NetDslParser.ts

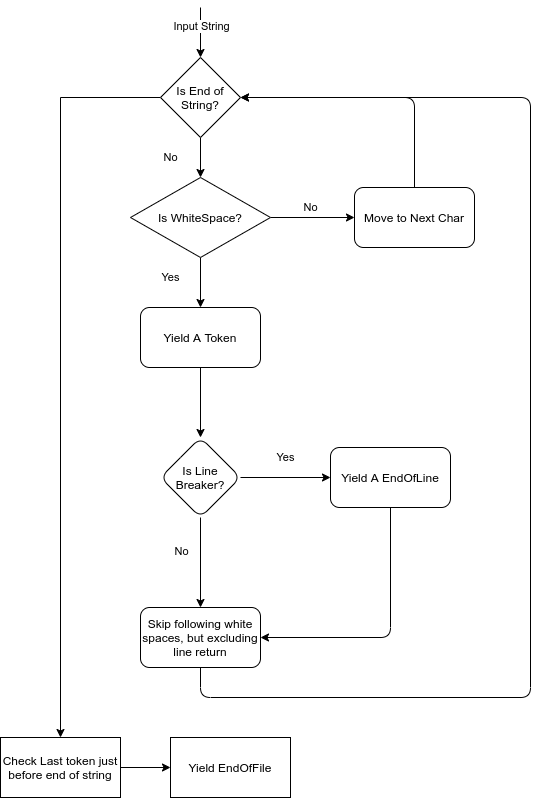
本次实现方案使用的是隐式状态机。后面会对可选的实现方案进行对比分析。

实现的主流程大致如下：



函数ParseNetDslLines根据分隔符（空格或者换行或者字符串读取结束）把输入分解为词（命名为token）；函数Parse调用ParseNetDslLines得到一个词组成的数据流，然后按照语法规则，对每行的第一个词进行比对（node/edge/group），并根据三种可能分派到节点/边/分组各自的解析函数Node.TryFindArgument/Edge.TryFindArgument/Group.TryFindArgument；如果不是期望的词，则将整行跳过并记录错误。

ParseNetDslLines的流程如下：



要特别注意两点：1 在发现一个词后需要执行跳过后续空格的操作，否则会产生大量无效的“空词”，降低效率（执行判空并调整内部状态的操作）2 当扫描字符串到达末尾的时候，可能会有一个词被“剩下”，因为没有遇到空格/换行符等就结束了，这个时候通过检查起始索引是否相等就知道是否有“剩下”的词；最后才发出”EndOfFile”。

Node.TryFindArgument/Edge.TryFindArgument/Group.TryFindArgument这三个方法比较简单，都是根据语法规则搜索后续的参数，如果参数有问题就记录错误/警告，没有错误就添加到对应的节点集合/边集合/分组集合。流程就不再画了。

|  |
| --- |
| function \* ParseNetDslLines(netDslStr:string):Generator<ParsedToken,ParsedToken>  function Parse(netDslStr:string):[Net, Info[]] |

下面进行一些关键的技术分析对比：

* 方案对比：隐式状态机/显式状态机/BNF语法+Parser Generator/正则表达式

隐式状态机的意思是，分析过程的状态没有通过定义枚举类型和一个专用的状态记录变量来控制整个过程。

显式状态机就是指定义了以上状态枚举类型以及对应的状态变量。例如可以这样定义枚举：Init/NodeStart/NodeId/NodeLabel/NodeEnd/.... 。然后再定义这些状态之间的变换关系，例如Init遇到”node”跳转到NodeStart状态，遇到”edge”跳转到EdgeStart状态，NodeLabel遇到换行符跳转到NodeEnd状态，等。

BNF语法+ParserGenerator的意思是，使用编译器工具链给出的辅助功能，首先使用BNF这种语言来定义NetDSL语言的语法，例如

|  |
| --- |
| Node := “node” nodeid nodelabel;  Nodeid, nodelabel :: token  .... |

然后通过ParserGenerator这类工具按照输入的BNF语法规则，生成一个解析器分析NetDSL的语法。

这次选用显式状态机，原因如下：

NetDSL的规则比较简单，但是状态却不少，使用显式状态机比较麻烦（需要手写大量状态变换规则），扩展的时候工作量较大。

BNF工具的错误定制消息不好控制，对换行符的控制也不太容易（然而ParserGenerator如果使用LL(n)语法分析生成器，解析效率是最快的）。

正则表达式效率较低（必须先按行拆分，第二次再进行每行的词拆分，一行多个词拆分后，再遍历第三次才可以执行具体规则的解析）。

隐式状态机在规则较少且较简单的前提下，是比较容易维护和扩展的。

例如对节点的规则调整，就在Node.TryFindArgument里面调整，不会影响其他规则。

* Js中的字符判断

Js没有字符类型，只有字符串这个类型。然而使用CharCodeAt方法可以得到字符对应的整数编码，利用这个方法可以较方便的判断是否是WhiteSpace：

|  |
| --- |
| function IsWhiteSpace(char: string):boolean {  const ch = char.charCodeAt(0);  return ch===32 || (ch>=9 && ch<=13) || ch===133 || ch===160;  } |

* Js/ts Generator(coroutine)

词法分析器(tokenizer)使用generator来编写，原因是想做出一个词的数据流。这样就不需要全部词构成一个数组才做下一步工作，大量节省内存。

另外，Generator支持yield的返回值进行处理（这里没有用到这个功能特征），然而接口使用不够方便，因此按照C#数据流的接口IEnumerator再进行了一层封装

|  |
| --- |
| class IEnumerator<T> {  private \_holder: Generator<T, T>;  private \_current: IteratorResult<T, T> | undefined;  public constructor(holder: Generator<T, T>) {  this.\_holder = holder;  }  public MoveNext(): boolean {  this.\_current = this.\_holder.next();  return !(this.\_current.done??false);  }  public get Current(): T {  if (this.\_current == undefined)  throw new Error("MoveNext first!");  return this.\_current.value;  }  } |

* Symbol.iterator

由于代码从C#转过来，js/ts并没有List这个数据结构，自己实现了一个（简单用array做，效率不高），为了能够用for of循环遍历自定义的数据结构，定义以下形式的方法：

|  |
| --- |
| [Symbol.iterator]():Iterator<T>{  return this.\_array.values();  } |

* Set判断二元组问题

Js/ts没有C#里面的HashSet，改用Set类型。然而当存储一个tuple的时候(类型为[Node,Node])，发现无法正常的判断相等，也没有开放接口可以重定义equals方法。

解决办法，定义TupleSet<T>类型，内部封装Map<T,Set<T>>类型作为状态，先按照二元组的第一个对象进行索引，把第一个对象相同的所有第二个对象作为集合存放。这样就可以准确判断是否重复了。

|  |
| --- |
| class TupleSet<T> {  private readonly \_set:Map<T,Set<T>>;  constructor() {  this.\_set = new Map<T, Set<T>>();  }  Add(k1:T,k2:T):void{  const v = this.\_set.get(k1)??new Set<T>();  if (v.size == 0){  this.\_set.set(k1,v);  }  v.add(k2);  }  Contains(k1:T,k2:T):boolean{  const v = this.\_set.get(k1);  return v?.has(k2)??false;  }  } |

* 解析器工程规范

错误需要有明确的定义；

错误有对应的位置；

需要把类似的错误进行归类；

需要定义错误编码；

除了解析的结果（拓扑图的数据结构）以及错误信息，其他类型不应该暴露，以免以后更新优化内部实现的时候影响了外部使用

* Error Recover 分析

Group定义有错的时候，有两种情况是有可能继续纠正并继续解析：

|  |
| --- |
| 1  Group  XXX  endGroup  2 Group  后面没有endGroup  Node ..... |

然而，是否有endgroup需要不断向后扫描才知道，这样可能会导致大量的重复扫描并产生大量的临时对象，降低效率。因此使用了最简单的处理方法，group如果后面没有找到endgroup，则整段都会作为错误。用户只要修正写好endgroup重新解析就可以了。