JSFormatter 文档

1 基本思想

格式化的过程主要分为两步：第一步是基于字符（**char**）的 LR(0)词法分析，通过连续的两个字符 charA 和 charB，将按字符读入的源代码分割成词（**token**）；第二步则根据词法分析得到的连续两个词 tokenA 和 tokenB 进行 LR(0)语法分析，根据规则输出 tokenA。

2 词法分析

词法分析通过 charB 对 charA 的类型进行判断，处理 charA，可能将 charA 加入 tokenB，也可能丢弃 charA，通过 charB 也能判断词的分割点。分割完成后的词已经存入 tokenB，给语法分析使用，主要的代码在 GetToken 函数中。

2.1 字符类型

- **一般字符**：'a'-'z', '0'-'9', 'A'-'Z', '_', '$'以及所有值大于 126 的字符（主要是非 ASCII 字符，如中文）。
- **空白字符**：' ', '\t', '\r'。这里注意的是'\n'也就是回车不被认为是空白字符，回车会被当作一个词提供给语法分析过程。我不想破坏那些有助于提高代码可阅读性的空白换行，后面会有策略来处理多个连续的换行。
- **单字符符号**：'.', '(', ')', '[', ']', '{', '}', ':', ',', ';', '~', '\n'。这些操作符长度一定是一个字符。
- **引号**：'\'', '\"'。
- **注释**：注释不得不同时判断 charA 和 charB，charA 是'/'并且 charB 是'/'或者'*'。

对于判断正则表达式和正负数（例如：-1），需要在语法层面进行分析，后面会讲到。

2.2 归约规则

CHAR=所有字符，NORCHAR=一般字符，BLACHAR=空白字符，SINCHAR=单字符符号，QUOCHAR=引号，COMMCHAR=注释，OTHCHAR=其它字符

CHAR=NORCHAR|BLACHAR|SINCHAR|QUOCHAR|COMMCHAR|OTHCHAR

正则表达式 REGULAR=/CHAR*/

注释 COMMENT=// CHAR* //|/* CHAR* */

字符串型 STRING=NORCHAR*|-NORCHAR*|+NORCHAR*

符号型 OPER=SINCHAR|OTHCHAR|OTHCHAR OTHCHAR|OTHCHAR=|===|!===

OPER=OTHCHAR|OTHCHAR OTHCHAR 的判断可能不完善，但是基于代码是正确的这个假设，这样做既快速又正确。

2.3 实现

准备：

对于一般情况，设 tokenBType 为字符串型（STRING_TYPE），**清空 tokenB**。对于正则表达式，则设 tokenBType 为正则型（REGULAR_TYPE），不要清空 tokenB，因为之前一定读出来的正则表达式的几个字符。对于正负数，tokenBType 仍然为字符串型，也不要清空 tokenB，正负号此时已经读出来了，之后就正常处理就可以了。

主循环：

将当前的 charB 移动到 charA，如果 charA 此时是 EOF，则停止整个格式化过程。读取下一个字符到 charB。

正则状态：（这是由语法分析判断并直接调用 GetToken 进入的状态）直接将 charA 连接到 tokenB 之后，并且循环直到 charA 是'/'。需要注意转义符号\/并非正则的停止，而是正则的一部分，也要处理\\。后面处理引号内容时也有类似规则。结束正则时，关掉正则状态（因为正则状态是成员变量），**返回**。

引号状态：（由循环的上一次进入）直接将 charA 连接到 tokenB 之后，并且循环直到 charA 是'\''或者'\"'（由引号开始符号决定）。依然要处理转义符号\\，\'和\"，它们是引号内容的一部分。结束引号后直接**返回**即可，引号状态是局部状态，下一次 GetToken 被调用时就重置了。还有就是并没有专门的引号型 tokenBType，因为引号内容被当作字符串型处理。

注释状态：直接将 charA 连接到 tokenB 之后。如果是//开始的注释（COMMENT_TYPE_1），循环直到换行；如果是/*开始的（COMMENT_TYPE_2），循环直到*/。需要注意的是跳过/t，因为缩进由算法控制。结束后也直接**返回**。

其余：如果能走到这里说明不是上面的各种情况，那么就按下面的逻辑判断。

charA 是一般字符：设 tokenBType 为字符串型，将 charA 接入 tokenB，执行下一次循环，直到 charB 不是**一般字符**（这里不能是 charA，如果进入了 charA 说明已经被处理了，下一次就会被抹掉。也保证了之后处理时不会出错）。结束后，关掉正负数状态（无论现在是否是开着的），之后返回。

charA 不是一般字符：首先跳过**空白字符**，接着判断 charA。如果 charA 是**引号**，先接入 charA，然后进入引号状态，设 tokenBType 为字符串型。如果是**注释**，接入 charA，进入注释状态。如果 charA 是**单字符符号**或者 charB 是**一般字符、空白字符、引号**，则 charA 是单字符符号，将 charA 放入 tokenB，设 tokenBType 为符号型（OPER_TYPE），返回即可。如果还能往下走，说明**有可能**是多字符符号。多字符符号只可能是 charB 为=或者 charA 和 charB 一样，如果不是前面两种，则还是单字符符号。如果的确是多字符符号，将 charA 和 charB 都接入 tokenB。并且需要在读入一个字符到 charB，如果此时 tokenB 是==或!=，而 charB 是=，则是三字符符号===，!==，<<=或者>>=，或者甚至有可能是>>>以及四字符符号>>>=。完成后设 tokenBType 为符号型（OPER_TYPE），返回。如果还能往下走，那么估计代码错了，停止。

到此，词法分析就完成了。再次说明，上面的算法基于代码是正确的。

3 语法分析

语法分析通过调用词法分析，持续的获得下一个词。词法分析将下一个词存入 tokenB 后，语法分析参考 tokenB 以及 tokenBType，判断应该如何输出 tokenA。完成后，将 tokenB 填入 tokenA，并且再次读取下一个词到 tokenB，如此循环直到结束。代码主要在 Go 函数中。

3.1 归约规则

缩进 INDENT=\t*

正则判断（下划线部分）return /CHAR*/|(/CHAR*/|,/CHAR*/|=/CHAR*/|:/CHAR*/|
[/CHAR*/|!/CHAR*/|&/CHAR*/| | /CHAR*/|?/CHAR*/|+/CHAR*/|
{/CHAR*/|}/CHAR*/|;/CHAR*/|\n/CHAR*/|

正负数判断!STRING&!COMMENT !++ !-- !] !) -|+NORCHAR*

循环和条件
IF=if(CODE)
ELSEIF=else if(CODE)

```
ELSE=else
FOR=for(CODE)
DO=do(CODE)
WHILE=while(CODE)
SWITCH=switch(CODE)
CASE=case STRING:|default:
TRY=try
CATCH=catch(CODE)

CODE=NULL|STRING|OPER|CODE*
BLOCK=BRACKBLOCK|IFBLOCK|FORBLOCK|DOBLOCK|SWITCHBLOCK|TRYBLOCK

BRACKBLOCK:
CODE{
INDENT CODE;*|BLOCK*
}

IFBLO:
IF
INDENT CODE;|BLOCK
IF{
INDENT CODE;*|BLOCK*
}

ELSEIFBLO:
ELSEIF
INDENT CODE;|BLOCK
ELSEIF{
INDENT CODE;*|BLOCK*
}

ELSEBLO:
ELSE
INDENT CODE;|BLOCK
ELSE{
INDENT CODE;*|BLOCK*
}

IFBLOCK:
IF
NULL|ELSEIF*
NULL|ELSE

FORBLOCK:
FOR
INDENT CODE;|BLOCK
FOR{
INDENT CODE;|BLOCK*
}

DOBLOCK:
DO
INDENT CODE;|BLOCK
DO{
INDENT CODE;|BLOCK*
}WHILE

CASEBLOCK:
CASE INDENT CODE;|BLOCK
```

```
CASE
INDENT CODE;|BLOCK
CASE{
INDENT CODE;|BLOCK*
}

SWITCHBLOCK:
SWITCH{
CASEBLOCK*
}

TRYBLO:
TRY
INDENT CODE;|BLOCK
TRY{
INDENT CODE;|BLOCK*
}

CATCHBLO:
CATCH
INDENT CODE;|BLOCK
CATCH{
INDENT CODE;*|BLOCK*
}

TRYBLOCK:
TRYBLO
CATCHBLO
```

通过归约规则可以看出最主要的问题在于循环条件块。注意的是例如这个规则：

```
IF
INDENT CODE;|BLOCK
```

if(...)如果不用{...}包裹代码，由;结束 IF，也可以由 BLOCK 结束 IF。而 BLOCK 可能是另一个 IFBLOCK 或 FORBLOCK 等。所以有可能出现;或{...}结束多个 IF 的问题。例如以下代码：

```
if(a == 1)
  while(a<2)
    a++;
```

此时 a++后面的;不仅仅结束了 while 循环，而且如果后面没有 else 的话，它也将结束 if。

3.2 格式规则

if, else if, else, for, do, while, try, catch：其中 if, else if, for, while, catch 之后必须有一个()出现，后面的代码才能被视为内部代码。内部代码格式如归约规则，可以由{}包裹，或者在单行的情况下（;作为结尾）不使用{}。被{}包裹时，格式如{}。如果没有，则内部代码另起一行，且增加一个缩进。

switch：switch 之后必须有一个()出现，后面的代码才能被视为内部代码，内部代码必须有{}包裹。格式类似{}，但是出现 case 和 default 时，缩进减一，直到:出现后换行，恢复正常的缩进，并且这个:前后不要空格。

{}：{符号可以位于上一行的结尾，或者新的一行。这个应该可以有参数控制。如果{位于新的一行或者上一行结尾有注释，使得{必须在新的一行，那么{应该与之前的缩进一样多。与{对应的}符号的缩进也应该是一样的。被{}包裹的代码则应该多一个缩进。{}内部的,会导致换行，但是不增加缩进，这是

为了使得 js 的类定义比较好看。

()：()内部不会自动加上换行，如果读入了换行，则新行增加一个缩进，之后同一个()里面的新行保持该缩进，)结束后恢复缩进。

;：;符号可以结束循环或者条件块。另外注意的是出现{};或者();时，;之前的换行会被取消。总之{,;之前的换行都不发生，除非前面有注释。

()[]!. ++ --：这几个符号输出时没有附加格式。

,：,之后加一个空格，如果是在{}中，换行不增加缩进。

=之后的换行应该增加一个缩进。

其余符号没有特别的处理，前后都加一个空格。

3.3 实现

先在 blockStack 中压入一个空格，防止出错，之后直接进入主循环。

开始：

判断 tokenB 是不是正则：这是唯一两个参考 tokenA 决定 tokenB 的步骤。判断方法和上面的归约规则一样：tokenB 不是注释，tokenA 不是字符串型，并且 tokenB 以/开始，tokenA 以(,=:[!&|?+{};\n 结尾，特殊情况是 tokenA 是 return，tokenB 以/开头。如果符合规则，调用 GetToken()取正则表达式。

判断是不是正负数：当然这和正则是不可能同时成立的，规则和归约规则也是一样的。tokenB 是-或者+，tokenA 不是字符串型也不是正则，并且 tokenA 不是++，--，]，)，最后 charB 也就是下一个字符是普通字符。符合规则，调用 GetToken()完成表达式。

之后按照下面的逻辑处理：

将 tokenB 填入 tokenA，之后准备下一个 tokenB。此时需要看是不是有排队的换行，如果有则直接从队列中得到 tokenB；没有就读取一个新的 tokenB，并判断是否跳过换行。

跳过换行的具体过程是一直读取下一个 token，直到不是\n。如果这个 token 是 else，while，catch，,或者;，那么跳过之前的换行；如果不是其中之一，将最多两个换行压入队列，并将该 token 也压入队列。这样做可以合并代码中的空白换行。之后就进入主要的处理逻辑部分。

主要逻辑：

tokenA 是正则表达式：直接输出，不要加任何的样式。因为有可能出现/.../g 之类的表达式，这种表达式会被拆成/.../和 g 两个，如果加空格就出错了。

tokenA 是注释：对于单行注释直接输出就可以了，因为一定会换行的。如果是多行注释，则应该补上一个换行（tokenB 是换行就不用了）。

tokenA 是操作符：逻辑和上面的格式规则一样。具体细节如下。

- **)]：**如果栈顶是对应的([，关掉赋值操作符状态（bAssign=false），弹栈，减小缩进。对于)还有特别处理的过程，如果栈顶是 if，while，for，switch，catch，并且在等待()结束（！brcNeedStack.top()），弹掉 brcNeedStack 的栈顶。之后准备进入块内代码。这里有一个问题需要处理，就是如果是 do...while()的情况，那么这个)应该结束 do...while()，将 while 和 do 都从栈中弹出，减小 if 块和 do 块计数。如果不是 do...while()就增加缩进。输出符号时后面加**空格**换行（tokenB 是换行就不用了）。如果不是前面的情况，但是 tokenB 是{或者换行，则加一个空格在)后面。其余的情况就直接输出，返回。
- **([：**入栈并增加缩进。

- **其余的无样式符号!，!!，~，.**：直接输出，返回。
- **;**：首先关掉赋值操作符状态。之后如果栈顶是循环、条件、try...catch，则认为是没有{}包裹的，结束这些块，弹栈、减小缩进（do 块只减小缩进，不减少块标记）、减少对应块标记数量（nIfLikeBlock，nDoLikeBlock）。之后判断是不是需要在**弹出多个类似的块**。完成后，准备输出;。如果当前在()中（认为是 for(;;)），则在;后面加一个空格，如果不在()中，则换行（tokenB 是换行就不用了）。之后返回。
- **,**：关掉赋值操作符状态。如果是在{}中，则换行（tokenB 是换行就不用了），这个是类定义时会出现的。不在就在,后面加一个空格。输出返回。
- **{**：关掉赋值操作符状态。如果栈顶是条件、循环或者 try...catch，则先减少一个缩进（因为后面还要加一个），但是不把栈顶的弹出来，也不调整块标记数量，而是等读到}时，和{一起弹栈，然后调整块标记数量。之后将{入栈，加一个缩进。在{后面加一个换行（一样 tokenB 是换行就不用了），返回。
- **}**：关掉赋值操作符状态。这里使用了一个比较激进的策略。就是对于}，会将栈一直弹到栈顶为{，途中弹出了循环、条件或者 try...catch 就减少缩进，减小块标记数量。这个策略有助于控制错误（如果有的还）的扩散。此时栈顶已经是{时，弹栈、减少缩进。如果{前面有 if 之类的也一起弹掉（do 不要弹掉，留给 while 处理），减小块标记数量（不用再减小缩进了）。到此缩进调整、栈调整完成。输出时，}前面需要一个换行（如果有正在等待的换行就不用了）。tokenB 如果不是,;) else，while，catch，则换行（tokenB 是换行就不用了），如果是 else，while，catch 加一个空格，,;)则直接输出}。对于栈顶的 function 和 HELPER 字符，则直接弹掉。之后在处理是否**弹出多个块**。完成后返回。
- **++，--，\n（\r\n）**：直接输出、返回。
- **:**：如果当前栈顶是 case，加空格换行（tokenB 是换行就不用换行），空格是为了后面有可能是{而考虑。之后返回。
- **=**：打开赋值操作符状态，在此状态中，换行时加一个缩进。
- **剩余的操作符**：输出时两边加空格。

对于**弹出多个块的处理**，如果刚刚弹出了 if，并且 tokenB 是 else（或者刚刚弹掉了 do，tokenB 是 while；以及刚刚弹掉 try，栈顶是 catch），则不再弹块。这个逻辑也是之后连续弹栈停止的逻辑。如果栈顶是循环、条件或 try...catch，则弹栈直到上面这个几个情况。弹块后，减小缩进、减少块标记（do 只减小缩进，块标记留给 while 处理）。

至此 tokenA 为操作符的情况就处理完成了。

如果 tokenA 是**字符串型**。

- **case 或 default**：输出时减少一个缩进，之后再恢复，并入栈、返回。
- **do，else（后面没跟着 if），try**：它们后面都准备进入块内代码（bBlockStmt=false），所以在这里输出并加换行，入栈，增加缩进，增加 nDoLikeBlock 计数，之后返回。

上面两个情况发生时，无论 tokenB 是什么都可以，下面得情况考虑 tokenB 的类型。

- **function**：关掉赋值操作符状态。将 function 入栈。
- **tokenB 是字符串型**：tokenA 输出后面加个空格。
- **tokenB 是符号**：如果 tokenA 是 if，for，while，catch，则输出，入栈，增加 nIfLikeBlock 计数，并且将 false 压入 bracNeedStack，表示等待()。这里不加缩进，缩进在处理)时加。如果 tokenA 是 switch，则增加 nSwitchBlock 计数，并且将 false 压入

bracNeedStack，表示等待()，之后 switch 入栈。由于 switch 之后一定会有{}，所以处理起来和 if 略有不同。

到此，token 处理结束。应该就能得到格式化过的源代码了。