l3regex 模块

TEX 中的正则表达式

LATEX 项目组* 2024 年 1 月 04 日 发布

张泓知 2024 年 1 月 18 日 【译】

目 录

*E-mail: latex-team@latex-project.org

l3regex 模块提供正则表达式测试、子匹配提取、分割和替换，全部作用于记号列表（token list，简写为 tl，后同不再注释）。正则表达式的语法主要是 PCRE 语法（Perl Compatible Regular Expressions，即 Perl 兼容正则表达式）的子集（并且非常接近 POSIX），但由于 TEX 操作的是记号（token，后同）而不是字符（character，后同），因此有一些附加的功能。由于性能原因，仅实现了有限的功能集。特别地，不支持反向引用。

下面我们给出一些示例：

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

在记号列表变量 \l_my_tl 中存储文本 “This cat.”，其中第一个 “at” 被替换为 “is”。一个更复杂的例子是用于强调每个单词并在其后添加逗号的模式：

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

\w 序列表示任何“word”字符，而 + 表示 \w 序列应重复尽可能多次（至少一次），因此匹配输入记号列表中的一个单词。在替换文本中，\0 表示完全匹配（这里是一个单词）。通过 \c{emph} 插入 \emph 命令，并将其参数 \0 放在括号 \cB\{ 和 \cE\} 之间。

如果要多次使用正则表达式，可以编译一次，并使用 \regex_set:Nn 将其存储在正则表达式变量中。例如，

```
\regex_new:N \l_foo_regex
\regex_set:Nn \l_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

在 \l_foo_regex 中存储一个正则表达式，该表达式匹配环境的起始标记：\begin，后跟一个起始组记号（\cB.），然后是任意数量的既不是起始组记号也不是结束组记号的字符记号（\c[^BE].*），最后是结束组记号（\cE.）。如下一节所述，圆括号 “捕获”（“capture”）了 \c[^BE].* 的结果，从而在进行替换时让我们能够访问环境的名称。

1 正则表达式的语法

1.1 正则表达式示例

我们从一些示例开始，并鼓励读者对这些正则表达式应用 \regex_show:n 。

- Cat 匹配以此方式大写的单词 “Cat”，但也匹配单词 “Cattle” 的开头：使用 \bCat\b 仅匹配完整的单词。

- `[abc]` 匹配字母 “a”、“b”、“c” 中的一个；模式 `(a|b|c)` 匹配相同的三个可能的字母（但请参阅下面的子匹配的讨论）。
- `[A-Za-z]*` 匹配任意数量（由于量词 `*`）的拉丁字母（没有重音）。
- `\c{[A-Za-z]*}` 匹配由拉丁字母组成的控制序列（control sequence，简写 cs，后同不再注释）。
- `\_[^\_]*\_` 匹配下划线，除下划线外的任意数量字符，和另一个下划线；这等效于 `\_.*?\_`，其中 `.` 匹配任意字符，懒惰量词 `*?` 表示尽可能匹配少的字符，从而避免匹配下划线。
- `[\+\-]?\d+` 匹配带有最多一个符号的显式整数。
- `[\+\-\␣]*\d+\␣*` 匹配带有任意数量 + 和 − 符号的显式整数，允许空格，除了在尾数内，且被空格包围。
- `[\+\-\␣]*(\d+|\d*\.\d+)\␣*` 匹配显式整数或小数；使用 `[.,]` 而不是 `\.` 允许逗号作为小数点。
- `[\+\-\␣]*(\d+|\d*\.\d+)\␣*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)\␣*` 匹配任何 TEX 知道的显式尺寸，其中 `(?i)` 表示对待小写和大写字母相同。
- `[\+\-\␣]*((?i)nan|inf|(\d+|\d*\.\d+)(\␣*e[\+\-\␣]*\d+)?)\␣*` 匹配显式浮点数或特殊值 `nan` 和 `inf`（允许符号和空格）。
- `[\+\-\␣]*(\d+|\cC.)\␣*` 匹配显式整数或控制序列（不检查是否是整数变量）。
- `\G.*?\K` 在正则表达式开头则匹配并丢弃（由于 `\K`）所有的在前一个匹配的结尾（`\G`）和由正则表达式的其余部分匹配的部分之间的内容；在 `\regex_replace_all:nnN` 中很有用，当目标是以比 `\regex_extract_all:nnN` 更精细的方式提取匹配项或子匹配项时。

尽管不可能让正则表达式仅匹配整数表达式，但是

```
[\+\-\(]*\d+\)*([\+\-*/][\+\-\(]*\d+\)*)*
```

匹配所有有效的整数表达式（仅由显式整数制成）。应该有进一步的测试。

1.2 正则表达式中的字符

大多数字符与它们自己完全匹配，具有任意类别码。一些字符是特殊的，必须用反斜杠转义（例如，`\*` 匹配星号字符）。一些反斜杠-字母形式的转义序列也具有特殊含义（例如 `\d` 匹配任何数字）。一般规则是，

- 每个字母数字（alphanumeric，后同不再注释）字符（A–Z、a–z、0–9）与其自身完全匹配，不应转义，因为 \A、\B 等具有特殊含义；
- 非字母数字可打印 ASCII 字符应始终（并且应该）转义：其中许多字符具有特殊含义（例如，使用 \(、\)、\?、\.、\^）；
- 空格应始终转义（即使在字符类中）；
- 其他任何字符可以转义，也可以不转义，都没有任何效果：两个版本都完全匹配该字符。

请注意，这些规则与许多非字母数字字符在正常类别码（category code，后同不再注释）下很难输入到 TEX 中的事实相吻合。例如，\\abc\% 匹配字符 \abc%（具有任意类别码），但不匹配控制序列 \abc 后跟一个百分号字符。可以使用 \c{⟨*regex*⟩} 语法（见下文）来匹配控制序列。

任何特殊字符出现在其特殊行为无法应用的地方时，将匹配自身（例如，在字符串开头出现的量词 [1]），并提出警告。

字符。

\x{hh…} 具有十六进制代码 hh… 的字符

\xhh 具有十六进制代码 hh 的字符

\a 警报（十六进制 07）

\e 转义（十六进制 1B）

\f 进纸换页（十六进制 0C）

\n 换行（十六进制 0A）

\r 回车（十六进制 0D）

\t 水平制表符（十六进制 09）

1.3 字符类

字符属性。

. 单个句点匹配任何记号。

\d 任何十进制数字。

\h 任何水平空白字符，等同于 [\ \^^I]：空格和制表符。

[1] 译者注：即 * + ? 等等字符

`\s` 任何空格字符，等同于 `[\ \^^I\^^J\^^L\^^M]`

`\v` 任何垂直空白字符，等同于 `[\^^J\^^K\^^L\^^M]`。注意 `\^^K` 是垂直空白，但不是空格，以兼容 Perl。

`\w` 任何单词字符，即字母数字和下划线，等同于明确的类[2] `[A-Za-z0-9\_]`

`\D` 任何非 `\d` 匹配的记号。

`\H` 任何非 `\h` 匹配的记号。

`\N` 任何非 `\n` 字符（十六进制 0A）的记号。

`\S` 任何非 `\s` 匹配的记号。

`\V` 任何非 `\v` 匹配的记号。

`\W` 任何非 `\w` 匹配的记号。

其中，`.`、`\D`、`\H`、`\N`、`\S`、`\V` 和 `\W` 匹配任意控制序列。
字符类精确匹配该类中的一个记号。

`[…]` 正向字符类。匹配指定的任何记号。

`[^…]` 负向字符类。匹配指定字符之外的任何记号。

`[x-y]` 在字符类中，表示一个范围（可以与转义字符一起使用）。

`[:`⟨*name*⟩`:]` 在字符类中（另一组括号），表示 POSIX 字符类 ⟨*name*⟩，可以是 `alnum`、`alpha`、`ascii`、`blank`、`cntrl`、`digit`、`graph`、`lower`、`print`、`punct`、`space`、`upper`、`word` 或 `xdigit`。

`[:^`⟨*name*⟩`:]` 负向 POSIX 字符类。

例如，`[a-oq-z\cC.]` 匹配任何小写拉丁字母，除了 `p`，以及控制序列（见下文对 `\c` 的描述）。

在字符类中，只有`[ ^ - ] \` 和空格是特殊的，应该被转义。其他非字母数字字符仍可被转义而不会受到影响。在字符类中支持匹配单个字符的任何转义序列（`\d`、`\D`、等）。如果第一个字符是 `^`，则字符类的含义被反转；在范围中的任何其他位置出现的 `^` 都不是特殊的。如果第一个字符（可能是在一个领头的 `^` 之后）是 `]`，则不需要转义，因为在那里结束范围将使其为空。字符的范围可以用 `-` 表示，例如，`[\D 0-5]` 和 `[^6-9]` 是等价的。

[2]译者注：这里简称的“类”在未明确所指时，都指的是字符类，即 Characters classes，后同不再注释。

1.4 结构：替代、分组、重复

量词（重复）。

`?` 0 或 1，贪婪模式。

`??` 0 或 1，懒惰模式。

`*` 0 或多次，贪婪模式。

`*?` 0 或多次，懒惰模式。

`+` 1 或多次，贪婪模式。

`+?` 1 或多次，懒惰模式。

`{`n`}` 恰好 n 次。

`{`n`,}` n 次或更多，贪婪模式。

`{`n`,}?` n 次或更多，懒惰模式。

`{`n,m`}` 至少 n 次，最多 m 次，贪婪模式。

`{`n,m`}?` 至少 n 次，最多 m 次，懒惰模式。

对于贪婪量词，正则表达式代码将首先尝试尽可能多的重复匹配，而对于懒惰量词，它将首先尝试尽可能少的重复匹配。

替代和捕获分组。

`A|B|C` A、B 或 C 中的任意一个，首先尝试匹配 A。

`(…)` 捕获分组。

`(?:…)` 非捕获分组。

`(?|…)` 非捕获分组，每个替代中都重置捕获分组编号。下一个分组将用第一个未使用的分组编号进行编号。

捕获分组是提取关于匹配的信息的一种方法。带括号的组按照其打开括号的顺序编号，从 1 开始。可以使用例如 `\regex_extract_once:nnNTF` 将这些组的内容提取并存储在一系列记号列表中。

`\K` 转义序列将匹配的开始位置重置为记号列表中的当前位置。这仅影响作为完整匹配报告的内容。例如，

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

的结果是 \l_foo_seq 包含项 {1} 和 {a}：真正的匹配是 {a1} 和 {aa}，但它们被使用 \K 截断。\K 命令不影响捕获分组，例如，

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

的结果是 \l_foo_seq 包含项 {c3} 和 {bc}：真正的匹配是 {acbc3}，其中第一个子匹配是 {bc}，但 \K 重置匹配的开始位置为它出现的最后位置。

1.5 匹配确切的记号

\c 转义序列允许测试记号的类别码，并匹配控制序列。每个字符类别由单个大写字母表示：

- C 表示控制序列；
- B 表示开始组记号；
- E 表示结束组记号；
- M 表示数学换位符；
- T 表示对齐制表符记号；
- P 表示宏参数记号；
- U 表示上标记号（上）；
- D 表示下标记号（下）；
- S 表示空格；
- L 表示字母；
- O 表示其他；以及
- A 表示活动字符。

\c 转义序列的使用如下。

\c{⟨*regex*⟩} 一个控制序列,其控制序列名称匹配 ⟨*regex*⟩,锚定在开头和结尾,因此 \c{begin} 精确匹配 \begin，而不匹配其他任何内容。

\cX 适用于下一个对象，可以是字符、转义字符序列（如 \x{0A}）、字符类或组，并强制该对象只匹配类别为 X 的记号（其中 X 为 CBEMTPUDSLOA 中的任意一个）。例如，\cL[A-Z\d] 匹配大写字母和类别为字母的数字，\cC. 匹配任何控制序列，\cO(abc) 匹配类别为其他的 abc。[3]

[3]最后一个示例还捕获了“abc”作为正则表达式组；要避免这种情况，请使用非捕获组 \cO(?:abc)。

`\c[XYZ]` 适用于下一个对象，并强制它只匹配类别为 X、Y 或 Z 的记号（其中每个都是 CBEMTPUDSLOA 中的任意一个）。例如，`\c[LSO](..)` 匹配两个类别为字母、空格或其他的记号。

`\c[^XYZ]` 适用于下一个对象，并阻止它匹配类别为 X、Y 或 Z 的任何记号（其中每个都是 CBEMTPUDSLOA 中的任意一个）。例如，`\c[^O]\d` 匹配类别为其他的数字。

类别码测试可用于类中；例如，`[\cO\d \c[LO][A-F]]` 匹配 TEX 认为的十六进制数字，即类别为其他的数字，或类别为字母或其他的大写字母。在受到类别码测试影响的组内，嵌套测试可以覆盖外部测试；例如，`\cL(ab\cO\*cd)` 匹配所有字符都是字母类别的 ab*cd，除了 * 的类别是其他。

`\u` 转义序列允许将记号列表的内容直接插入正则表达式或替换中，无需转义特殊字符。即，`\u{`⟨*var name*⟩`}` 精确匹配变量 `\`⟨*var name*⟩ 的内容（包括字符码和类别码），这是通过在编译正则表达式时应用 `\exp_not:v {`⟨*var name*⟩`}` 获得的。在 `\c{...}` 控制序列匹配中，`\u` 转义序列仅展开其参数一次，实际上执行 `\tl_to_str:v`。支持量词。

`\ur` 转义序列允许将 regex 变量的内容插入较大的正则表达式。例如，`A\ur{l_tmpa_regex}D` 匹配由匹配正则表达式 `\l_tmpa_regex` 的内容分隔的记号 A 和 D。这相当于将非捕获组包围在 `\l_tmpa_regex` 周围，并且 `\l_tmpa_regex` 中包含的任何组都会转换为非捕获组。支持量词。

例如，如果 `\l_tmpa_regex` 的值为 B|C，那么 `A\ur{l_tmpa_regex}D` 等效于 A(?:B|C)D（匹配 ABD 或 ACD），而不是 AB|CD（匹配 AB 或 CD）。要获得后者的效果，最简单的方法是直接使用 TEX 的展开机制：如果 `\l_mymodule_BC_tl` 包含 B|C，那么以下两行显示相同的结果：

```
\regex_show:n { A \u{l_mymodule_BC_tl} D }
\regex_show:n { A B | C D }
```

1.6 杂项

锚点和简单断言。

`\b` 单词边界：前一个记号由 `\w` 匹配，下一个由 `\W` 匹配；或者相反。为此，将记号列表的两端视为 `\W`。

`\B` 非单词边界：在两个 `\w` 记号或两个 `\W` 记号之间（包括边界）。

`^` 或 `\A` 所处的记号列表的开始。

`$`，`\Z` 或 `\z` 所处的记号列表的结尾。

`\G` 当前匹配的开始。仅在多次匹配的情况下与 `^` 不同。例如：

```
\regex_count:nnN { \G a } { aaba } \l_tmpa_int
```

得到 2，但将 `\G` 替换为 `^` 将导致 `\l_tmpa_int` 包含值 1。

选项 `(?i)` 使匹配不区分大小写（将 `A`–`Z` 和 `a`–`z` 视为等效，尚不支持 Unicode 大小写转换）。这适用于它所在的组直到结束，并可使用 `(?-i)` 还原。例如，在 `(?i)(a(?-i)b|c)d` 中，字母 `a` 和 `d` 受到 `i` 选项的影响。范围和类中的字符被单独影响：`(?i)[\?-B]` 等效于 `[\?@ABab]`（与较大的类 `[\?-b]` 不同），`(?i)[^aeiou]` 匹配任何不是元音的字符。`i` 选项对于 `\c{...}`、`\u{...}`、字符属性或字符类等没有任何影响，例如在 `(?i)\u{l_foo_tl}\d\d[[:lower:]]` 中根本不起作用。

2 替换文本的语法

正则表达式中描述的大多数功能在替换文本中没有意义。反斜杠引入各种特殊结构，下面将进一步描述：

- `\0` 表示整个匹配；
- `\1` 表示由第一个（捕获）组 `(...)` 匹配的子匹配；类似地，`\2`、…、`\9` 和 `\g{`⟨*number*⟩`}`
- `\␣` 插入一个空格（未转义的情况下忽略空格）；
- `\a`、`\e`、`\f`、`\n`、`\r`、`\t`、`\xhh`、`\x{hhh}` 对应于正则表达式中的单个字符；
- `\c{`⟨*cs name*⟩`}` 插入一个控制序列；
- `\c`⟨*category*⟩⟨*character*⟩（见下文）；
- `\u{`⟨*tl var name*⟩`}` 将变量 ⟨*tl var*⟩ 的内容直接插入替换文本，无需转义特殊字符。

除反斜杠和空格之外的字符直接插入结果中（但由于首先将替换文本转换为字符串，因此也应转义对于 TEX 而言是特殊的字符，例如使用 `\#`）。非字母数字字符始终可以安全地使用反斜杠进行转义。例如，

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?l|o) . } { (\0--\1) } \l_my_tl
```

导致 \l_my_tl 包含 H(ell--el)(o,--o) w(or--o)(ld--l)!

子匹配的编号按照捕获组的开放括号在要匹配的正则表达式中出现的顺序进行。如果捕获组少于 n 个，或者捕获组出现在未用于匹配的替代方案中，则第 n 个子匹配为空。如果捕获组在匹配期间多次匹配（由于量词），则在替换文本中仅使用最后一次匹配。子匹配始终保持与原始记号列表相同的类别码。

默认情况下，替换插入的字符的类别码由替换时的主要类别码制度确定，有两个例外：

- 通过 \␣、\x20 或 \x{20} 插入的空格字符（字符码为 32）无论当前的类别码制度如何，其类别码始终为 10；
- 如果类别码为 0（转义）、5（换行符）、9（忽略）、14（注释）或 15（无效），则替换时将其替换为 12（其他）。

转义序列 \c 允许插入具有任意类别码的字符，以及控制序列。

\cX(…) 产生类别为 X 的字符 "…"，其中 X 必须是正则表达式中的 CBEMTPUDSLOA 之一。括号对于单个字符（可能是转义序列）是可选的。嵌套时，应用最内层的类别码，例如 \cL(Hello\cS\ world)! 会产生标准类别码的此文本。

\c{⟨*text*⟩} 插入 csname 为 ⟨*text*⟩ 的控制序列。⟨*text*⟩ 可能包含对子匹配 \0、\1 等的引用，如下例所示。

转义序列 \u{⟨*var name*⟩} 允许将变量 ⟨*var name*⟩ 的内容直接插入替换文本，更容易控制类别码。在 \c{…} 和 \u{…} 结构中嵌套时，\u 和 \c 转义序列执行 \tl_to_str:v，即提取控制序列的值并将其转换为字符串。匹配还可在 \c 和 \u 的参数中使用。例如，

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{l_my_\0_tl} } \l_my_tl
```

结果为 \l_my_tl 包含 first,\emph{second},first,first。

正则表达式替换还是一个方便的方法，用于生成具有任意类别码的记号列表。例如

```
\tl_clear:N \l_tmpa_tl
\regex_replace_all:nnN { } { \cU\% \cA\~ } \l_tmpa_tl
```

导致 \l_tmpa_tl 包含类别码为 7（上标）的百分号字符和活动中划线字符。

3 预编译正则表达式

如果要多次使用正则表达式，最好是编译一次，而不是每次使用正则表达式时都编译。编译后的正则表达式存储在一个变量中。所有 l3regex 模块的函数都可以将其正则表达式参数作为显式字符串或编译后的正则表达式给出。

\regex_new:N
New: 2017-05-26

\regex_new:N ⟨*regex var*⟩

创建一个新的 ⟨*regex var*⟩，如果名称已被使用则引发错误。该声明是全局的。初始时，⟨*regex var*⟩ 被设置为永远不匹配。

\regex_set:Nn
\regex_gset:Nn
New: 2017-05-26

\regex_set:Nn ⟨*regex var*⟩ {⟨*regex*⟩}

在 ⟨*regex var*⟩ 中存储 ⟨*regular expression*⟩ 的编译版本。对于 \regex_set:Nn，赋值是局部的；对于 \regex_gset:Nn，是全局的。例如，此函数可以用作

```
\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ex|ular\ expression) }
```

\regex_const:Nn
New: 2017-05-26

\regex_const:Nn ⟨*regex var*⟩ {⟨*regex*⟩}

创建一个新的常量 ⟨*regex var*⟩，如果名称已被使用则引发错误。⟨*regex var*⟩ 的值被全局设置为 ⟨*regular expression*⟩ 的编译版本。

\regex_show:N
\regex_show:n
\regex_log:N
\regex_log:n
New: 2021-04-26
Updated: 2021-04-29

\regex_show:n {⟨*regex*⟩}
\regex_log:n {⟨*regex*⟩}

在终端显示或写入日志文件（分别）l3regex 如何解释 ⟨*regex*⟩。例如，\regex_show:n {\A X|Y} 显示

```
+-branch
  anchor at start (\A)
  char code 88 (X)
+-branch
  char code 89 (Y)
```

表明锚 \A 仅适用于第一分支：第二分支未锚定到匹配的开始。

4 匹配

所有正则表达式函数都有 :n 和 :N 两种变体。前者需要一个“标准”正则表达式，而后者需要由 \regex_set:Nn 生成的编译表达式。

\regex_match:nnTF
\regex_match:nVTF
\regex_match:NnTF
\regex_match:NVTF

New: 2017-05-26

\regex_match:nnTF {⟨*regex*⟩} {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

测试 ⟨*regular expression*⟩ 是否与 ⟨*token list*⟩ 的任何部分匹配。例如，

```
\regex_match:nnTF { b [cde]* } { abecdcx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

在输入流中留下 TRUE 然后 FALSE。

\regex_count:nnN
\regex_count:nVN
\regex_count:NnN
\regex_count:NVN

New: 2017-05-26

\regex_count:nnN {⟨*regex*⟩} {⟨*token list*⟩} ⟨*int var*⟩

在当前 TEX 组级别内，将 ⟨*int var*⟩ 设置为 ⟨*regular expression*⟩ 在 ⟨*token list*⟩ 中出现的次数。搜索从找到最左边最长的匹配开始，尊重贪婪和懒惰（非贪婪）运算符。然后，搜索从前一匹配的最后一个字符之后的字符开始，直到达到 token list 的末尾。在正则表达式可以匹配空 token list 的情况下，防止无限循环：在每对字符之间计数一次匹配。例如，

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcbb } \l_foo_int
```

的结果是 \l_foo_int 的值为 5。

\regex_match_case:nn
\regex_match_case:nnTF

New: 2022-01-10

\regex_match_case:nnTF
{
{⟨*regex*$_1$⟩} {⟨*code case*$_1$⟩}
{⟨*regex*$_2$⟩} {⟨*code case*$_2$⟩}
…
{⟨*regex*$_n$⟩} {⟨*code case*$_n$⟩}
} {⟨*token list*⟩}
{⟨*true code*⟩} {⟨*false code*⟩}

确定在 ⟨*token list*⟩ 中的最早位置哪个 ⟨*regular expression*⟩ 匹配，并在输入流中留下相应的 ⟨*code*$_i$⟩，后跟 ⟨*true code*⟩。如果多个 ⟨*regex*⟩ 在同一点开始匹配，则选择列表中的第一个，并丢弃其他的。如果没有任何 ⟨*regex*⟩ 匹配，则在输入流中留下 ⟨*false code*⟩。每个 ⟨*regex*⟩ 都可以是正则表达式变量或显式正则表达式。

具体而言，对于 ⟨*token list*⟩ 中的每个起始位置，依次搜索 ⟨*regex*⟩。如果其中一个匹配，则使用相应的 ⟨*code*⟩，并丢弃其他一切；如果在给定位置没有 ⟨*regex*⟩ 匹配，则尝试下一个起始位置。如果在 ⟨*token list*⟩ 的任何位置都没有任何 ⟨*regex*⟩ 匹配，则在输入流中什么都不留下。请注意，这与嵌套的 \regex_match:nnTF 语句不同，因为在每个位置尝试匹配所有 ⟨*regex*⟩，而不是在移动到 ⟨*regex*$_2$⟩ 之前尝试匹配 ⟨*regex*$_1$⟩。

5 子匹配提取

\regex_extract_once:nnN
\regex_extract_once:nVN
\regex_extract_once:nnNTF
\regex_extract_once:nVNTF
\regex_extract_once:NnN
\regex_extract_once:NVN
\regex_extract_once:NnNTF
\regex_extract_once:NVNTF

New: 2017-05-26

\regex_extract_once:nnN {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_extract_once:nnNTF {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

在 ⟨*token list*⟩ 中找到 ⟨*regular expression*⟩ 的第一个匹配项。如果存在匹配项，将匹配项存储为 ⟨*seq var*⟩ 的第一项，其余项是捕获组的内容，按其开括号的顺序。局部赋值给 ⟨*seq var*⟩。如果没有匹配项，则清除 ⟨*seq var*⟩。测试版本如果找到匹配项，则将 ⟨*true code*⟩ 插入输入流，否则插入 ⟨*false code*⟩。

例如，假设您键入

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
  { true } { false }
```

那么正则表达式（在开始处用 `\A` 锚定，在结束处用 `\Z` 锚定）必须匹配整个 token list。第一个捕获组，`(La)?`，匹配 `La`，第二个捕获组，`(!*)`，匹配 `!!!`。因此，`\l_foo_seq` 的结果包含项 `{LaTeX!!!}`，`{La}` 和 `{!!!}`，并在输入流中留下 `true` 分支。注意，`\l_foo_seq` 的第 n 项，使用 `\seq_item:Nn` 获取，对应于函数 `\regex_replace_once:nnN` 等中编号为 $(n-1)$ 的子匹配。

\regex_extract_all:nnN
\regex_extract_all:nVN
\regex_extract_all:nnNTF
\regex_extract_all:nVNTF
\regex_extract_all:NnN
\regex_extract_all:NVN
\regex_extract_all:NnNTF
\regex_extract_all:NVNTF

New: 2017-05-26

\regex_extract_all:nnN {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_extract_all:nnNTF {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

在 ⟨*token list*⟩ 中找到 ⟨*regular expression*⟩ 的所有匹配项，并将所有子匹配信息存储在一个序列中（连接多个 `\regex_extract_once:nnN` 调用的结果）。局部赋值给 ⟨*seq var*⟩。如果没有匹配项，则清除 ⟨*seq var*⟩。测试版本如果找到匹配项，则将 ⟨*true code*⟩ 插入输入流，否则插入 ⟨*false code*⟩。例如，假设您键入

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
  { true } { false }
```

那么正则表达式匹配两次，生成的序列包含两个项 `{Hello}` 和 `{world}`，并在输入流中留下 `true` 分支。

\regex_split:nnN
\regex_split:nVN
\regex_split:nnNTF
\regex_split:nVNTF
\regex_split:NnN
\regex_split:NVN
\regex_split:NnNTF
\regex_split:NVNTF

New: 2017-05-26

\regex_split:nnN {⟨*regular expression*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_split:nnNTF {⟨*regular expression*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

将 ⟨*token list*⟩ 拆分为一系列部分，由 ⟨*regular expression*⟩ 的匹配项分隔。如果 ⟨*regular expression*⟩ 具有捕获组，则它们匹配的 token lists 也存储为序列的项。局部赋值给 ⟨*seq var*⟩。如果找不到匹配项，生成的 ⟨*seq var*⟩ 以 ⟨*token list*⟩ 为其唯一项。如果 ⟨*regular expression*⟩ 匹配空 token list，则将 ⟨*token list*⟩ 拆分为单个 token。测试版本如果找到匹配项，则将 ⟨*true code*⟩ 插入输入流，否则插入 ⟨*false code*⟩。例如，在

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
  { true } { false }
```

之后，序列 `\l_path_seq` 包含项 `{the}`，`{path}`，`{for}`，`{this}` 和 `{file.tex}`，并在输入流中留下 `true` 分支。

6 替换

\regex_replace_once:nnN
\regex_replace_once:nVN
\regex_replace_once:nnNTF
\regex_replace_once:nVNTF
\regex_replace_once:NnN
\regex_replace_once:NVN
\regex_replace_once:NnNTF
\regex_replace_once:NVNTF

New: 2017-05-26

\regex_replace_once:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩
\regex_replace_once:nnNTF {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

在 ⟨*tl var*⟩ 的内容中搜索 ⟨*regular expression*⟩，并用 ⟨*replacement*⟩ 替换第一个匹配项。在 ⟨*replacement*⟩ 中，`\0` 代表完整匹配，`\1` 代表第一个捕获组的内容，`\2` 代表第二个，以此类推。结果局部赋值给 ⟨*tl var*⟩。

\regex_replace_all:nnN
\regex_replace_all:nVN
\regex_replace_all:nnNTF
\regex_replace_all:nVNTF
\regex_replace_all:NnN
\regex_replace_all:NVN
\regex_replace_all:NnNTF
\regex_replace_all:NVNTF

New: 2017-05-26

\regex_replace_all:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩
\regex_replace_all:nnNTF {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

将 ⟨*tl var*⟩ 的内容中的 ⟨*regular expression*⟩ 的所有匹配项都替换为 ⟨*replacement*⟩，其中 `\0` 代表完整匹配，`\1` 代表第一个捕获组的内容，`\2` 代表第二个，以此类推。每个匹配项都是独立处理的，匹配项之间不能重叠。结果局部赋值给 ⟨*tl var*⟩。

\regex_replace_case_once:nN
\regex_replace_case_once:nN*TF*

New: 2022-01-10

```
\regex_replace_case_once:nNTF
  {
    {⟨regex_1⟩} {⟨replacement_1⟩}
    {⟨regex_2⟩} {⟨replacement_2⟩}
    …
    {⟨regex_n⟩} {⟨replacement_n⟩}
  } ⟨tl var⟩
  {⟨true code⟩} {⟨false code⟩}
```

将 `(?|`⟨*regex*$_1$⟩`|…|`⟨*regex*$_n$⟩`)` 的最早匹配项，用与之匹配的 ⟨*replacement*⟩ 替换，在输入流中留下 ⟨*true code*⟩。如果没有 ⟨*regex*⟩ 匹配，则不修改 ⟨*tl var*⟩，并在输入流中留下 ⟨*false code*⟩。每个 ⟨*regex*⟩ 可以作为 regex 变量或显式正则表达式给出。

具体而言，对于 ⟨*token list*⟩ 中的每个起始位置，按顺序搜索每个 ⟨*regex*⟩。如果其中一个匹配，则将其替换为与之对应的 ⟨*replacement*⟩，并从紧随此匹配（和替换）的位置开始重新搜索。这相当于使用 `\regex_match_case:nn` 检查哪个 ⟨*regex*⟩ 匹配，然后用 `\regex_replace_once:nnN` 执行替换。

\regex_replace_case_all:nN
\regex_replace_case_all:nN*TF*

New: 2022-01-10

```
\regex_replace_case_all:nNTF
  {
    {⟨regex_1⟩} {⟨replacement_1⟩}
    {⟨regex_2⟩} {⟨replacement_2⟩}
    …
    {⟨regex_n⟩} {⟨replacement_n⟩}
  } ⟨tl var⟩
  {⟨true code⟩} {⟨false code⟩}
```

将 ⟨*token list*⟩ 中所有 ⟨*regex*⟩ 的所有匹配项都替换为相应的 ⟨*replacement*⟩。每个匹配项都是独立处理的，匹配项之间不能重叠。结果局部赋值给 ⟨*tl var*⟩，并根据是否进行了替换留下 ⟨*true code*⟩ 或 ⟨*false code*⟩。

具体而言，对于 ⟨*token list*⟩ 中的每个起始位置，按顺序搜索每个 ⟨*regex*⟩。如果其中一个匹配，则将其替换为与之对应的 ⟨*replacement*⟩，并从紧随此匹配（和替换）的位置开始重新搜索。例如

```
\tl_set:Nn \l_tmpa_tl { Hello,~world! }
\regex_replace_case_all:nN
  {
    { [A-Za-z]+ } { ``\0'' }
    { \b } { --- }
    { . } { [\0] }
  } \l_tmpa_tl
```

结果是 `\l_tmpa_tl` 包含内容 ``Hello''---[,][␣]``world''---[!]。请特别注意，单词边界断言 `\b` 在单词开头没有匹配，因为情况 `[A-Za-z]+` 在这些位置匹配。要更改这一点，可以简单地交换 `\regex_replace_case_all:nN` 参数中两个案例的顺序。

7 临时用的正则表达式

\l_tmpa_regex
\l_tmpb_regex

本地赋值的 Scratch 正则表达式。这些从未被内核代码使用，因此可安全用于任何 LaTeX3 定义的函数。但可能被其他非内核代码覆盖，因此仅用于短期存储。

New: 2017-12-11

\g_tmpa_regex
\g_tmpb_regex

全局赋值的 Scratch 正则表达式。这些从未被内核代码使用，因此可安全用于任何 LaTeX3 定义的函数。但可能被其他非内核代码覆盖，因此仅用于短期存储。

New: 2017-12-11

8 错误、缺陷、未来工作和其他可能性

现在需要完成以下任务。

- 以更有序的方式重写文档，或许添加一个 BNF?

更多的错误检查即将到来。

- 整理消息的使用。
- 在替换阶段进行更清晰的错误报告。
- 添加跟踪信息。
- 检测尝试使用反向引用和其他未实现语法的情况。
- 测试最大寄存器`\c_max_register_int` 的情况。
- 弄清楚`\W`和类似的匹配是否导致错误。可能更新`\__regex_item_reverse:n`。
- 空 cs 应该由`\c{}`匹配，而不是由`\c{csname.?endcsname\s?}`匹配。

代码改进即将到来。

- 将数组移动，使有用信息从位置 1 开始。
- 仅构建一次`\c{...}`。
- 在编译正则表达式时，使用左右状态堆栈的数组。
- `\__regex_action_free_group:n` 是否仅用于贪婪的`{n,}`量词?（我认为不是。）
- 对于`\u`和断言的量词。
- 在匹配时，跟踪 `curr_state` 和 `curr_submatches` 的显式堆栈。
- 如果可能的话，在同一线程中重复使用状态时，终止其他子线程。
- 使用数组而不是`\g__regex_balance_tl` 来构建函数`\__regex_replacement_balance_one_match:n`。
- 减少替代中的 转换数量。
- 优化简单的字符串:使用较少的状态(`abcade`应该给出两个状态,用于`abc`和`ade`)。[这真的有意义吗?]
- 优化没有替代的组。

- 优化只有一个__regex_action_free:n 的状态。
- 通过直接在状态 2 中插入__regex_action_success: 来优化其使用，而不是有额外的转换。
- 通过插入\int_step_... 函数的使用来优化。
- 组在 csnames 的正则表达式中不捕获；优化并记录。
- 更好的锚定、属性和类别码测试的“show”。
- \K是否真的需要一个新的状态?
- 在编译时，使用布尔变量 in_cs 和较少的魔术数字。
- 与其使用字符代码检查字符是否为特殊字符或字母数字，不如在正则表达式中使用\cs_if_exist 测试检查它是否为特殊字符。

以下功能可能在将来的某个时候实现。

- 一般的先行/后行断言。
- 在外部文件上进行正则表达式匹配。
- 具有先行/后行条件的条件子模式：“如果之后是 [...]，则 [...]”。
- (*..) 和 (?..) 序列以设置一些选项。
- pdfTEX 的 UTF-8 模式。
- 换行约定尚未完成。特别是，应该有一个选项使 . 不匹配换行符。此外，\A 应该与 ^ 不同，而 \Z、\z 和 $ 应该不同。
- Unicode 属性：\p{..} 和 \P{..}；\X 应该匹配任何“扩展”的 Unicode 序列。这需要操作大量数据，可能使用树状盒子。

下面的 PCRE 或 Perl 的功能可能会或可能不会被实现。

- 使用 (?C...) 或其他语法的调用：一些内部代码更改使这成为可能，在替换代码中找到标记时停止正则表达式替换可能很有用；这引发了潜在的 \regex_break: 问题，以及在正则表达式代码中从 \tl_map_break: 调用的良好处理问题。还提出了正则表达式机制内嵌套调用的问题，这是一个问题，因为\fontdimen是全局的。
- 条件子模式（不是先行或后行条件的）：这是非正则的，对吧?
- 命名子模式：TEX 程序员迄今为止无需命名宏参数。

下面的 PCRE 或 Perl 的功能肯定不会被实现。

- 反向引用：非正则特性，需要回溯，速度极慢。
- 递归：这是非正则特性。
- 原子分组、贪婪量词：这些工具主要用于修复灾难性回溯，在非回溯算法中是不必要的，并且难以实现。
- 子例程调用：这种语法糖难以包含在非回溯算法中，特别是因为相应的组应该被视为原子的。
- 回溯控制动词：与回溯密切相关。
- `\ddd`，匹配八进制代码为 `ddd` 的字符：我们已经有了 `\x{...}`，并且语法与我们可以用于反向引用（`\1`，`\2`，等）的语法相似，这使得产生有用的错误消息变得更加困难。
- `\cx`，类似于 TeX 的 `\^^x`。
- 注释：TeX 已经有了自己的注释系统。
- `\Q...\E` 转义：这需要逐字读取参数，不在此模块的范围内。
- 在 UTF-8 模式下的单字节 `\C`：XeTeX 和 LuaTeX 直接为我们提供字符，将其拆分为字节是棘手的，依赖编码，而且很可能并不实用。

9 l3regex 代码实现

1 ⟨*package⟩

2 ⟨@@=regex⟩

9.1 攻略计划

大多数正则表达式引擎使用回溯。这允许提供非常强大的功能（首先想到的是反向引用），但这是昂贵的，并且引发了灾难性的回溯问题。由于 TeX 首先不是一种编程语言，复杂的代码倾向于运行缓慢，我们必须使用更快但稍微更受限的技术，来自自动机理论。

给定一个长度为 n 的正则表达式，我们执行以下操作：

- （编译）分析正则表达式，找到无效输入，并将其转换为内部表示。
- （构建）将已编译的正则表达式转换为具有 $O(n)$ 状态的非确定有限自动机（NFA），该自动机精确接受与该正则表达式匹配的记号列表。

- （匹配）循环遍历查询记号列表的每个标记（每个“位置”），并在 NFA 中探索每个可能的路径（“活动线程”），按照量词的贪婪性确定的顺序考虑活动线程。

在代码注释（以及变量名称）中，我们使用以下术语。

- 组：捕获组的索引，对于非捕获组为 -1。
- 位置:查询中的每个标记都由整数$\langle position \rangle$标记,其中 $\texttt{min\_pos} - 1 \leq \langle position \rangle \leq \texttt{max\_pos}$。$\texttt{min\_pos} - 1$ 和 max_pos 对应于虚构的开始和结束标记（带有不存在的类别码和字符码）。max_pos 只在处理过程中相当晚地设置。
- 查询：我们应用正则表达式的记号列表。
- 状态：NFA 的每个状态由整数$\langle state \rangle$标记，满足 $\texttt{min\_state} \leq \langle state \rangle < \texttt{max\_state}$。
- 活动线程：在匹配过程中，通过读取查询记号列表达到 NFA 的状态。这些线程按照量词的贪婪性从最好到最不受欢迎的顺序排列。
- 步骤：在匹配时使用，从 0 开始，每次读取一个字符时递增，在搜索重复匹配时不重置。整数\l__regex_step_int是匹配算法所有步骤的唯一标识符。

我们使用 l3intarray 来操作整数数组。我们还滥用 TEX 的 \toks 寄存器，通过直接按编号访问它们，而不是使用\newtoks分配函数将它们绑定到控制序列。具体来说，这些数组和\toks用法如下。在构建时，\toks$\langle state \rangle$保存在 NFA 的$\langle state \rangle$中执行的测试和操作。在匹配时，

- \g__regex_state_active_intarray保存每个$\langle state \rangle$最后一次活跃的$\langle step \rangle$。
- \g__regex_thread_info_intarray由每个$\langle thread \rangle$的块组成(满足 $\texttt{min\_thread} \leq \langle thread \rangle < \texttt{max\_thread}$)。每个块都有 $1 + 2$\l__regex_capturing_group_int 个条目:$\langle thread \rangle$当前所处的$\langle state \rangle$,然后是所有子匹配的开始和结束。$\langle threads \rangle$按从最佳到最不受欢迎的顺序排序。
- \g__regex_submatch_prev_intarray、\g__regex_submatch_begin_intarray 和 \g__regex_submatch_end_intarray,对于每个子匹配(就像\regex_extract_all:nnN提取的那样），存储了寻找子匹配开始的地方以及子匹配的两个端点。由于历史原因, 最小索引是两倍的 max_state, 而已使用的寄存器最多到\l__regex_submatch_int。它们组织成\l__regex_capturing_group_int个条目的块，每个块对应于一个带有所有子匹配的匹配，这些子匹配存储在连续的条目中。

在实际构建结果时，

- \toks⟨*position*⟩保存⟨*tokens*⟩，o-和 e-展开为查询中的第⟨*position*⟩个标记。
- \g__regex_balance_intarray保存在记号列表中该点之前出现的开始组和结束组字符标记的平衡。

代码结构如下。变量在相关部分引入。首先介绍一些通用的辅助函数。然后是用于编译正则表达式和显示编译结果的代码。构建阶段将已编译的正则表达式转换为 NFA 状态，代码在以下部分运行自动机。唯一剩下的组件是解析替换文本并执行替换。然后，我们准备好所有用户函数。最后是消息和一点追踪代码。

9.2 辅助函数

__regex_int_eval:w 访问原始的\numexpr：性能至关重要，因此我们不使用通过\int_eval:n的较慢路线。

```
\cs_new_eq:NN \__regex_int_eval:w \tex_numexpr:D
```

(*__regex_int_eval:w 定义结束。*)

__regex_standard_escapechar: 将\escapechar设置为标准反斜杠。

```
\cs_new_protected:Npn \__regex_standard_escapechar:
  { \int_set:Nn \tex_escapechar:D { `\\ } }
```

(*__regex_standard_escapechar: 定义结束。*)

__regex_toks_use:w 根据其编号解包\toks。

```
\cs_new:Npn \__regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(*__regex_toks_use:w 定义结束。*)

__regex_toks_clear:N 清空\toks或将其设置为给定值，根据其编号。
__regex_toks_set:Nn
__regex_toks_set:No

```
\cs_new_protected:Npn \__regex_toks_clear:N #1
  { \__regex_toks_set:Nn #1 { } }
\cs_new_eq:NN \__regex_toks_set:Nn \tex_toks:D
\cs_new_protected:Npn \__regex_toks_set:No #1
  { \tex_toks:D #1 \exp_after:wN }
```

(*__regex_toks_clear:N 和 __regex_toks_set:Nn 定义结束。*)

__regex_toks_memcpy:NNn 复制从\toks寄存器 #2 开始的 #3 个寄存器到从 #1 开始的寄存器，类似于 C 的 memcpy 。

```
\cs_new_protected:Npn \__regex_toks_memcpy:NNn #1#2#3
  {
    \prg_replicate:nn {#3}
```

```
      {
        \tex_toks:D #1 = \tex_toks:D #2
        \int_incr:N #1
        \int_incr:N #2
      }
  }
```

(*__regex_toks_memcpy:NNn* 定义结束。)

__regex_toks_put_left:Ne
__regex_toks_put_right:Ne
__regex_toks_put_right:Nn

在构建阶段，我们希望将 e-扩展的材料添加到\toks，可以是左边也可以是右边。为了优化（这些操作相当频繁），我们手动进行扩展。提供__regex_toks_put_right:Ne的 Nn 版本，因为它比使用\exp_not:n进行 e-扩展更有效率。

```
\cs_new_protected:Npn \__regex_toks_put_left:Ne #1#2
  {
    \cs_set_nopar:Npe \__regex_tmp:w { #2 }
    \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
      { \exp_after:wN \__regex_tmp:w \tex_the:D \tex_toks:D #1 }
  }
\cs_new_protected:Npn \__regex_toks_put_right:Ne #1#2
  {
    \cs_set_nopar:Npe \__regex_tmp:w {#2}
    \tex_toks:D #1 \exp_after:wN
      { \tex_the:D \tex_toks:D \exp_after:wN #1 \__regex_tmp:w }
  }
\cs_new_protected:Npn \__regex_toks_put_right:Nn #1#2
  { \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }
```

(*__regex_toks_put_left:Ne* 和 *__regex_toks_put_right:Ne* 定义结束。)

__regex_curr_cs_to_str:

展开为当前位置\l__regex_curr_pos_int处的记号（已知是控制序列）的字符串表示。它应该仅在 e/x-扩展中使用，以避免丢失前导空格。

```
\cs_new:Npn \__regex_curr_cs_to_str:
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
    \l__regex_curr_token_tl
  }
```

(*__regex_curr_cs_to_str:* 定义结束。)

__regex_intarray_item:NnF

具有默认值的 intarray 的项。

__regex_intarray_item_aux:nNF

```
\cs_new:Npn \__regex_intarray_item:NnF #1#2
  { \exp_args:Nf \__regex_intarray_item_aux:nNF { \int_eval:n {#2} } #1 }
\cs_new:Npn \__regex_intarray_item_aux:nNF #1#2
```

```
  {
    \if_int_compare:w #1 > \c_zero_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { \__kernel_intarray_item:Nn #2 {#1} }
  }
```

(*__regex_intarray_item:NnF* 和 *__regex_intarray_item_aux:nNF* 定义结束。)

__regex_maplike_break: 与\tl_map_break:类似，这正确退出\tl_map_inline:nn和类似结构，并跳转到匹配的 \prg_break_point:Nn __regex_maplike_break: { }.

```
\cs_new:Npn \__regex_maplike_break:
  { \prg_map_break:Nn \__regex_maplike_break: { } }
```

(*__regex_maplike_break:* 定义结束。)

__regex_tl_odd_items:n
__regex_tl_even_items:n
__regex_tl_even_items_loop:nn

一次处理一个记号列表中的一对项目，留下奇数编号或偶数编号的项目（第一个项目编号为 1）。

```
\cs_new:Npn \__regex_tl_odd_items:n #1 { \__regex_tl_even_items:n { ? #1 } }
\cs_new:Npn \__regex_tl_even_items:n #1
  {
    \__regex_tl_even_items_loop:nn #1 \q__regex_nil \q__regex_nil
    \prg_break_point:
  }
\cs_new:Npn \__regex_tl_even_items_loop:nn #1#2
  {
    \__regex_use_none_delimit_by_q_nil:w #2 \prg_break: \q__regex_nil
    { \exp_not:n {#2} }
    \__regex_tl_even_items_loop:nn
  }
```

(*__regex_tl_odd_items:n*, *__regex_tl_even_items:n*, 和 *__regex_tl_even_items_loop:nn* 定义结束。)

9.2.1 常量和变量

__regex_tmp:w 用于各种短期目的的临时函数。

```
\cs_new:Npn \__regex_tmp:w { }
```

(*__regex_tmp:w* 定义结束。)

\l__regex_internal_a_tl
\l__regex_internal_b_tl
\l__regex_internal_a_int
\l__regex_internal_b_int
\l__regex_internal_c_int
\l__regex_internal_bool
\l__regex_internal_seq
\g__regex_internal_tl

用于各种目的的临时变量。

```
\tl_new:N   \l__regex_internal_a_tl
\tl_new:N   \l__regex_internal_b_tl
\int_new:N  \l__regex_internal_a_int
\int_new:N  \l__regex_internal_b_int
\int_new:N  \l__regex_internal_c_int
\bool_new:N \l__regex_internal_bool
\seq_new:N  \l__regex_internal_seq
\tl_new:N   \g__regex_internal_tl
```

(*\l__regex_internal_a_tl 以及其它的定义结束。*)

\l__regex_build_tl

此临时变量专门用于与tl_build机制一起使用。

```
\tl_new:N \l__regex_build_tl
```

(*\l__regex_build_tl 定义结束。*)

\c__regex_no_match_regex

此正则表达式匹配任何内容，但仍然是有效的正则表达式。我们可以使用失败的断言，但我选择了一个空类。它用作使用\regex_new:N声明的正则表达式的初始值。

```
\tl_const:Nn \c__regex_no_match_regex
  {
    \__regex_branch:n
      { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
  }
```

(*\c__regex_no_match_regex 定义结束。*)

\l__regex_balance_int

在此阶段，\l__regex_balance_int计算在记号列表中给定点之前出现的开始组和结束组字符标记的平衡。此变量也用于跟踪替换文本中的平衡。

```
\int_new:N \l__regex_balance_int
```

(*\l__regex_balance_int 定义结束。*)

9.2.2 测试字符

\c__regex_ascii_min_int
\c__regex_ascii_max_control_int
\c__regex_ascii_max_int

```
\int_const:Nn \c__regex_ascii_min_int { 0 }
\int_const:Nn \c__regex_ascii_max_control_int { 31 }
\int_const:Nn \c__regex_ascii_max_int { 127 }
```

(*\c__regex_ascii_min_int*, *\c__regex_ascii_max_control_int*, 和 *\c__regex_ascii_max_int* 定义结束。)

\c__regex_ascii_lower_int

```
\int_const:Nn \c__regex_ascii_lower_int { `a - `A }
```

(*\c__regex_ascii_lower_int 定义结束。*)

9.2.3 内部辅助函数

\q__regex_recursion_stop 内部递归 quark。

```
\quark_new:N \q__regex_recursion_stop
```

(*\q__regex_recursion_stop* 定义结束。)

\q__regex_nil 内部 quark。

```
\quark_new:N \q__regex_nil
```

(*\q__regex_nil* 定义结束。)

egex_use_none_delimit_by_q_recursion_stop:w
_regex_use_i_delimit_by_q_recursion_stop:nw
__regex_use_none_delimit_by_q_nil:w 用于吞掉 quark 的函数。

```
\cs_new:Npn \__regex_use_none_delimit_by_q_recursion_stop:w
  #1 \q__regex_recursion_stop { }
\cs_new:Npn \__regex_use_i_delimit_by_q_recursion_stop:nw
  #1 #2 \q__regex_recursion_stop {#1}
\cs_new:Npn \__regex_use_none_delimit_by_q_nil:w #1 \q__regex_nil { }
```

(*__regex_use_none_delimit_by_q_recursion_stop:w*, *__regex_use_i_delimit_by_q_recursion_stop:nw*, 和 *__regex_use_none_delimit_by_q_nil:w* 定义结束。)

__regex_quark_if_nil_p:n
__regex_quark_if_nil:n*TF* 分支 quark 条件。

```
\__kernel_quark_new_conditional:Nn \__regex_quark_if_nil:N { F }
```

(*__regex_quark_if_nil:nTF* 定义结束。)

__regex_break_point:TF
__regex_break_true:w 在测试查询记号列表中的字符是否与正则表达式中给定字符类匹配时，我们经常必须将其与几个字符范围进行比较，检查其中任何一个是否匹配。这通过以下结构完成:

⟨*test1*⟩ ... ⟨*test*$_n$⟩
__regex_break_point:TF {⟨*true code*⟩} {⟨*false code*⟩}

如果任何测试都成功,它调用 __regex_break_true:w,清理并在输入流中留下 ⟨*true code*⟩。否则，__regex_break_point:TF 在输入流中留下 ⟨*false code*⟩。

```
\cs_new_protected:Npn \__regex_break_true:w
  #1 \__regex_break_point:TF #2 #3 {#2}
\cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }
```

(*__regex_break_point:TF* 和 *__regex_break_true:w* 定义结束。)

__regex_item_reverse:n

此函数使正则表达式的显示更加容易，并允许我们以 \d 的形式定义 \D。有一个微妙之处：查询的末尾由 −2 标记，因此与 \D 和其他否定属性匹配；代码的另一部分捕获了这种情况。

```
\cs_new_protected:Npn \__regex_item_reverse:n #1
  {
    #1
    \__regex_break_point:TF { } \__regex_break_true:w
  }
```

(*__regex_item_reverse:n 定义结束。*)

__regex_item_caseful_equal:n

__regex_item_caseful_range:nn

触发 __regex_break_true:w 的简单比较。

```
\cs_new_protected:Npn \__regex_item_caseful_equal:n #1
  {
    \if_int_compare:w #1 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
  {
    \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
```

(*__regex_item_caseful_equal:n 和 __regex_item_caseful_range:nn 定义结束。*)

__regex_item_caseless_equal:n

__regex_item_caseless_range:nn

对于不区分大小写的匹配，我们对 curr_char 和 case_changed_char 都执行测试。在执行第二组测试之前，我们确保 case_changed_char 已经计算。

```
\cs_new_protected:Npn \__regex_item_caseless_equal:n #1
  {
    \if_int_compare:w #1 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
    \__regex_maybe_compute_ccc:
    \if_int_compare:w #1 = \l__regex_case_changed_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
```

```
{
  \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
    \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
      \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
    \fi:
  \fi:
  \__regex_maybe_compute_ccc:
  \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
    \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
      \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
    \fi:
  \fi:
}
```

(*__regex_item_caseless_equal:n* 和 *__regex_item_caseless_range:nn* 定义结束。)

__regex_compute_case_changed_char:

当尚未计算 `\l__regex_case_changed_char_int` 时调用此函数。如果当前字符代码在范围 $[65, 90]$（大写字母）中，则添加 32，使其变为小写。如果在小写字母范围 $[97, 122]$ 中，则减去 32。

```
\cs_new_protected:Npn \__regex_compute_case_changed_char:
  {
    \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
    \if_int_compare:w \l__regex_curr_char_int > `Z \exp_stop_f:
      \if_int_compare:w \l__regex_curr_char_int > `z \exp_stop_f: \else:
        \if_int_compare:w \l__regex_curr_char_int < `a \exp_stop_f: \else:
          \int_sub:Nn \l__regex_case_changed_char_int
            { \c__regex_ascii_lower_int }
        \fi:
      \fi:
    \else:
      \if_int_compare:w \l__regex_curr_char_int < `A \exp_stop_f: \else:
        \int_add:Nn \l__regex_case_changed_char_int
          { \c__regex_ascii_lower_int }
      \fi:
    \fi:
    \cs_set_eq:NN \__regex_maybe_compute_ccc: \prg_do_nothing:
  }
\cs_new_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
```

(*__regex_compute_case_changed_char:* 定义结束。)

__regex_item_equal:n
__regex_item_range:nn

这些必须始终定义为展开到 `caseful`（默认）或 `caseless` 版本，不能受保护：它们在编译时必须展开，以硬编码哪些测试是不区分大小写的或区分大小写的。

```
\cs_new_eq:NN \__regex_item_equal:n ?
\cs_new_eq:NN \__regex_item_range:nn ?
```

(*__regex_item_equal:n* 和 *__regex_item_range:nn* 定义结束。)

__regex_item_catcode:nT
__regex_item_catcode_reverse:nT
__regex_item_catcode:

参数是由允许的类别码（介于 0 和 13 之间）给出的 4 的幂的和。除以给定的 4 的幂，当且仅当允许该类别码时，结果为奇数。如果类别码不匹配，则跳过之后的字符代码测试。

```
\cs_new_protected:Npn \__regex_item_catcode:
  {
    "
    \if_case:w \l__regex_curr_catcode_int
         1       \or: 4       \or: 10      \or: 40
    \or: 100     \or:         \or: 1000    \or: 4000
    \or: 10000   \or:         \or: 100000  \or: 400000
    \or: 1000000 \or: 4000000 \else: 1*0
    \fi:
  }
\cs_new_protected:Npn \__regex_item_catcode:nT #1
  {
    \if_int_odd:w \int_eval:n { #1 / \__regex_item_catcode: } \exp_stop_f:
      \exp_after:wN \use:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
\cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
  { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }
```

(*__regex_item_catcode:nT*, *__regex_item_catcode_reverse:nT*, 和 *__regex_item_catcode:* 定义结束。)

__regex_item_exact:nn
__regex_item_exact_cs:n

这匹配一个精确的 ⟨*category*⟩-⟨*character code*⟩ 对，或者一个精确的控制序列，更准确地说，是由 \scan_stop: 分隔的若干可能的控制序列之一。

```
\cs_new_protected:Npn \__regex_item_exact:nn #1#2
  {
    \if_int_compare:w #1 = \l__regex_curr_catcode_int
      \if_int_compare:w #2 = \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
\cs_new_protected:Npn \__regex_item_exact_cs:n #1
  {
```

```
    \int_compare:nNnTF \l__regex_curr_catcode_int = 0
      {
        \__kernel_tl_set:Ne \l__regex_internal_a_tl
          { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
        \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
          \l__regex_internal_a_tl
          { \__regex_break_true:w } { }
      }
      { }
  }
```

(*__regex_item_exact:nn* 和 *__regex_item_exact_cs:n* 定义结束。)

__regex_item_cs:n 匹配一个控制序列（参数是已编译的正则表达式）。首先测试当前记号的类别码是否为零。然后执行匹配测试，并在 csname 确实匹配时中断。

```
\cs_new_protected:Npn \__regex_item_cs:n #1
  {
    \int_compare:nNnT \l__regex_curr_catcode_int = 0
      {
        \group_begin:
          \__regex_single_match:
          \__regex_disable_submatches:
          \__regex_build_for_cs:n {#1}
          \bool_set_eq:NN \l__regex_saved_success_bool
            \g__regex_success_bool
          \exp_args:Ne \__regex_match_cs:n { \__regex_curr_cs_to_str: }
          \if_meaning:w \c_true_bool \g__regex_success_bool
            \group_insert_after:N \__regex_break_true:w
          \fi:
          \bool_gset_eq:NN \g__regex_success_bool
            \l__regex_saved_success_bool
        \group_end:
      }
  }
```

(*__regex_item_cs:n* 定义结束。)

9.2.4 字符属性测试

__regex_prop_d:
__regex_prop_h:
__regex_prop_s:
__regex_prop_v:
__regex_prop_w:
__regex_prop_N:

\d、\W 等的字符属性测试。这些字符属性不受 (?i) 选项的影响。每个属性匹配的字符如下：\d=[0-9]，\w=[0-9A-Z_a-z]，\s=[\␣\^^I\^^J\^^L\^^M]，\h=[\␣\^^I]，\v=[\^^J-\^^M]，大写字符与小写字符的匹配相反。各个测试出现的顺序是为通常的大多数小写字母文本优化的。

```
\cs_new_protected:Npn \__regex_prop_d:
  { \__regex_item_caseful_range:nn { `0 } { `9 } }
\cs_new_protected:Npn \__regex_prop_h:
  {
    \__regex_item_caseful_equal:n { `\ }
    \__regex_item_caseful_equal:n { `\^^I }
  }
\cs_new_protected:Npn \__regex_prop_s:
  {
    \__regex_item_caseful_equal:n { `\ }
    \__regex_item_caseful_equal:n { `\^^I }
    \__regex_item_caseful_equal:n { `\^^J }
    \__regex_item_caseful_equal:n { `\^^L }
    \__regex_item_caseful_equal:n { `\^^M }
  }
\cs_new_protected:Npn \__regex_prop_v:
  { \__regex_item_caseful_range:nn { `\^^J } { `\^^M } } % lf, vtab, ff, cr
\cs_new_protected:Npn \__regex_prop_w:
  {
    \__regex_item_caseful_range:nn { `a } { `z }
    \__regex_item_caseful_range:nn { `A } { `Z }
    \__regex_item_caseful_range:nn { `0 } { `9 }
    \__regex_item_caseful_equal:n { `_ }
  }
\cs_new_protected:Npn \__regex_prop_N:
  {
    \__regex_item_reverse:n
      { \__regex_item_caseful_equal:n { `\^^J } }
  }
```

(*__regex_prop_d: 以及其它的定义结束。*)

__regex_posix_alnum:
__regex_posix_alpha:
__regex_posix_ascii:
__regex_posix_blank:
__regex_posix_cntrl:
__regex_posix_digit:
__regex_posix_graph:
__regex_posix_lower:
__regex_posix_print:
__regex_posix_punct:
__regex_posix_space:
__regex_posix_upper:
__regex_posix_word:
__regex_posix_xdigit:

POSIX 属性。不出意外。

```
\cs_new_protected:Npn \__regex_posix_alnum:
  { \__regex_posix_alpha: \__regex_posix_digit: }
\cs_new_protected:Npn \__regex_posix_alpha:
  { \__regex_posix_lower: \__regex_posix_upper: }
\cs_new_protected:Npn \__regex_posix_ascii:
  {
    \__regex_item_caseful_range:nn
      \c__regex_ascii_min_int
      \c__regex_ascii_max_int
  }
```

```
\cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\cs_new_protected:Npn \__regex_posix_cntrl:
  {
    \__regex_item_caseful_range:nn
      \c__regex_ascii_min_int
      \c__regex_ascii_max_control_int
    \__regex_item_caseful_equal:n \c__regex_ascii_max_int
  }
\cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
\cs_new_protected:Npn \__regex_posix_graph:
  { \__regex_item_caseful_range:nn { `! } { `\~ } }
\cs_new_protected:Npn \__regex_posix_lower:
  { \__regex_item_caseful_range:nn { `a } { `z } }
\cs_new_protected:Npn \__regex_posix_print:
  { \__regex_item_caseful_range:nn { `\  } { `\~ } }
\cs_new_protected:Npn \__regex_posix_punct:
  {
    \__regex_item_caseful_range:nn { `! } { `/ }
    \__regex_item_caseful_range:nn { `: } { `@ }
    \__regex_item_caseful_range:nn { `[ } { `` }
    \__regex_item_caseful_range:nn { `\{ } { `\~ }
  }
\cs_new_protected:Npn \__regex_posix_space:
  {
    \__regex_item_caseful_equal:n { `\  }
    \__regex_item_caseful_range:nn { `\^^I } { `\^^M }
  }
\cs_new_protected:Npn \__regex_posix_upper:
  { \__regex_item_caseful_range:nn { `A } { `Z } }
\cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
\cs_new_protected:Npn \__regex_posix_xdigit:
  {
    \__regex_posix_digit:
    \__regex_item_caseful_range:nn { `A } { `F }
    \__regex_item_caseful_range:nn { `a } { `f }
  }
```

（*__regex_posix_alnum: 以及其它的定义结束。*）

9.2.5 简单字符转义

在实际解析正则表达式或替换文本之前，我们首先通过它们一次，将 `\n` 转换为字符 10，等等。在此过程中，我们还将任何特殊字符（`*`、`?`、`{` 等）或转义的字母数

字字符转换为指示这是一个特殊序列的标记，并用指示这些字符为“原始”（“raw”）字符的标记替换转义的特殊字符和未转义的字母数字字符。然后，代码的其余部分可以避免关心转义问题（在与字符类范围结合使用时，这些问题可能变得相当复杂）。

用法：`\__regex_escape_use:nnnn` ⟨*inline 1*⟩ ⟨*inline 2*⟩ ⟨*inline 3*⟩ {⟨*token list*⟩} 将 ⟨*token list*⟩ 转换为字符串，然后从左到右阅读，将反斜杠解释为转义下一个字符。未转义的字符被传递给函数 ⟨*inline 1*⟩，而转义的字符则在 e-展开上下文中传递给函数 ⟨*inline 2*⟩（通常这些函数对其参数执行一些测试，以决定如何输出它们）。识别 `\a`、`\e`、`\f`、`\n`、`\r`、`\t` 和 `\x` 转义序列，并将它们替换为相应的字符，然后传递给 ⟨*inline 3*⟩。结果然后留在输入流中。空格除非转义，否则会被忽略。

转换是在 e-展开赋值中完成的。

`\__regex_escape_use:nnnn` 结果是在 `\l__regex_internal_a_tl` 中构建的，然后保留在输入流中。在此标记列表中添加了适当的跟踪代码。对 `#4` 进行一次处理，根据每个字符应用 `#1`、`#2` 或 `#3`。

```
\cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
  {
    \group_begin:
      \tl_clear:N \l__regex_internal_a_tl
      \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
      \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
      \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
      \__regex_standard_escapechar:
      \__kernel_tl_gset:Ne \g__regex_internal_tl
        { \__kernel_str_to_other_fast:n {#4} }
      \tl_put_right:Ne \l__regex_internal_a_tl
        {
          \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
          \scan_stop: \prg_break_point:
        }
      \exp_after:wN
    \group_end:
    \l__regex_internal_a_tl
  }
```

（*__regex_escape_use:nnnn* 定义结束。）

`\__regex_escape_loop:N`
`\__regex_escape_\:w`

`\__regex_escape_loop:N` 读取一个字符：如果它是特殊字符（空格、反斜杠或结束标记），则执行相关的操作，否则它只是一个未转义的字符。在反斜杠后，同样的操作，但未知字符被“转义”（“escaped”）。

```
\cs_new:Npn \__regex_escape_loop:N #1
  {
    \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
```

```
      { \__regex_escape_unescaped:N #1 }
    \__regex_escape_loop:N
  }
\cs_new:cpn { __regex_escape_ \c_backslash_str :w }
    \__regex_escape_loop:N #1
  {
    \cs_if_exist_use:cF { __regex_escape_/\token_to_str:N #1:w }
      { \__regex_escape_escaped:N #1 }
    \__regex_escape_loop:N
  }
```

(*__regex_escape_loop:N* 和 *__regex_escape_\:w* 定义结束。)

__regex_escape_unescaped:N
__regex_escape_escaped:N
__regex_escape_raw:N

这些函数在给定新含义之前从不被调用，因此这里的定义无关紧要。

```
\cs_new_eq:NN \__regex_escape_unescaped:N ?
\cs_new_eq:NN \__regex_escape_escaped:N   ?
\cs_new_eq:NN \__regex_escape_raw:N       ?
```

(*__regex_escape_unescaped:N*, *__regex_escape_escaped:N*, 和 *__regex_escape_raw:N* 定义结束。)

__regex_escape_\scan_stop::w
__regex_escape_/\scan_stop::w
__regex_escape_/a:w
__regex_escape_/e:w
__regex_escape_/f:w
__regex_escape_/n:w
__regex_escape_/r:w
__regex_escape_/t:w
__regex_escape_ :w

在看到结束标记 “break” 时结束循环，如果字符串以反斜杠结尾，则出现错误。忽略空格，\a、\e、\f、\n、\r、\t 在这里起到作用。

```
\cs_new_eq:cN { __regex_escape_ \iow_char:N\\scan_stop: :w } \prg_break:
\cs_new:cpn { __regex_escape_/ \iow_char:N\\scan_stop: :w }
  {
    \msg_expandable_error:nn { regex } { trailing-backslash }
    \prg_break:
  }
\cs_new:cpn { __regex_escape_~:w } { }
\cs_new:cpe { __regex_escape_/a:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^G }
\cs_new:cpe { __regex_escape_/t:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^I }
\cs_new:cpe { __regex_escape_/n:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^J }
\cs_new:cpe { __regex_escape_/f:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^L }
\cs_new:cpe { __regex_escape_/r:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^M }
\cs_new:cpe { __regex_escape_/e:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^[ }
```

(*__regex_escape_\scan_stop::w* 以及其它的定义结束。)

__regex_escape_/x:w
__regex_escape_x_end:w
__regex_escape_x_large:n

当遇到 \x 时，__regex_escape_x_test:N 负责获取一些十六进制数字，并将结果传递给 __regex_escape_x_end:w。如果数字太大，则中断赋值并生成错误，否则在相应的字符记号上调用 __regex_escape_raw:N。

```
\cs_new:cpn { __regex_escape_/x:w } \__regex_escape_loop:N
  {
    \exp_after:wN \__regex_escape_x_end:w
    \int_value:w "0 \__regex_escape_x_test:N
  }
\cs_new:Npn \__regex_escape_x_end:w #1 ;
  {
    \int_compare:nNnTF {#1} > \c_max_char_int
      {
        \msg_expandable_error:nnff { regex } { x-overflow }
          {#1} { \int_to_Hex:n {#1} }
      }
      {
        \exp_last_unbraced:Nf \__regex_escape_raw:N
          { \char_generate:nn {#1} { 12 } }
      }
  }
```

（*__regex_escape_/x:w*，*__regex_escape_x_end:w*，和 *__regex_escape_x_large:n* 定义结束。）

__regex_escape_x_test:N
__regex_escape_x_testii:N

查找第一个字符是否是左括号（允许任意数量的十六进制数字），或者不是（允许最多两个十六进制数字）。我们需要检查字符串的结束标记。最终，调用 __regex_escape_x_loop:N 或 __regex_escape_x:N。

```
\cs_new:Npn \__regex_escape_x_test:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_i:nnn \exp_after:wN ;
    \fi:
    \use:n
      {
        \if_charcode:w \c_space_token #1
          \exp_after:wN \__regex_escape_x_test:N
        \else:
          \exp_after:wN \__regex_escape_x_testii:N
          \exp_after:wN #1
        \fi:
      }
  }
\cs_new:Npn \__regex_escape_x_testii:N #1
```

```
  {
    \if_charcode:w \c_left_brace_str #1
      \exp_after:wN \__regex_escape_x_loop:N
    \else:
      \__regex_hexadecimal_use:NTF #1
        { \exp_after:wN \__regex_escape_x:N }
        { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
    \fi:
  }
```

(*__regex_escape_x_test:N* 和 *__regex_escape_x_testii:N* 定义结束。)

__regex_escape_x:N 在未括号的情况下查找第二个数字。

```
\cs_new:Npn \__regex_escape_x:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_i:nnn \exp_after:wN ;
    \fi:
    \use:n
      {
        \__regex_hexadecimal_use:NTF #1
          { ; \__regex_escape_loop:N }
          { ; \__regex_escape_loop:N #1 }
      }
  }
```

(*__regex_escape_x:N* 定义结束。)

__regex_escape_x_loop:N 抓取十六进制数字，跳过空格，最后检查是否有右括号，否则在赋值外部引发错误。

__regex_escape_x_loop_error:

```
\cs_new:Npn \__regex_escape_x_loop:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_ii:nnn
    \fi:
    \use_ii:nn
      { ; \__regex_escape_x_loop_error:n { } {#1} }
      {
        \__regex_hexadecimal_use:NTF #1
          { \__regex_escape_x_loop:N }
          {
            \token_if_eq_charcode:NNTF \c_space_token #1
              { \__regex_escape_x_loop:N }
              {
```

```
                ;
                \exp_after:wN
                \token_if_eq_charcode:NNTF \c_right_brace_str #1
                  { \__regex_escape_loop:N }
                  { \__regex_escape_x_loop_error:n {#1} }
              }
          }
      }
  }
\cs_new:Npn \__regex_escape_x_loop_error:n #1
  {
    \msg_expandable_error:nnn { regex } { x-missing-rbrace } {#1}
    \__regex_escape_loop:N #1
  }
```

(*__regex_escape_x_loop:N* 和 *__regex_escape_x_loop_error:* 定义结束。)

__regex_hexadecimal_use:NTF

TEX 会为我们检测大写的十六进制数字，但不会检测小写字母，我们需要检测并替换为它们的大写字母对应物。

```
\prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
  {
    \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
      #1 \prg_return_true:
    \else:
      \if_case:w
        \int_eval:n { \exp_after:wN ` \token_to_str:N #1 - `a }
           A
      \or: B
      \or: C
      \or: D
      \or: E
      \or: F
      \else:
        \prg_return_false:
        \exp_after:wN \use_none:n
      \fi:
      \prg_return_true:
    \fi:
  }
```

(*__regex_hexadecimal_use:NTF* 定义结束。)

__regex_char_if_alphanumeric:NTF
__regex_char_if_special:NTF

在解析正则表达式的第一遍中使用这两个测试。这个过程负责找到转义和非转义的字符，识别哪些字符具有特殊含义，哪些应被解释为“原始”（“raw”）字符。具体来说，

- 字母数字字符如果未被转义，则为“原始”（“raw”），当转义时可能具有特殊含义；
- 非字母数字可打印 ascii 字符如果被转义，则为“原始”（“raw”），当未转义时可能具有特殊含义；
- 可打印 ascii 之外的字符始终为“原始”（“raw”）。

代码很丑陋，高度依赖于魔术数字和字符的 ascii 码。出于性能原因，这在很大程度上是不可避免的。或许可以进一步优化这些测试。这里，“字母数字”（“alphanumeric”）表示 `0`–`9`，`A`–`Z`，`a`–`z`；“特殊”（“special”）字符表示非字母数字但可打印 ascii，从空格（十六进制 `20`）到 `del`（十六进制 `7E`）。

```
\prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
  {
    \if_int_compare:w `#1 > `Z \exp_stop_f:
      \if_int_compare:w `#1 > `z \exp_stop_f:
        \if_int_compare:w `#1 < \c__regex_ascii_max_int
          \prg_return_true: \else: \prg_return_false: \fi:
      \else:
        \if_int_compare:w `#1 < `a \exp_stop_f:
          \prg_return_true: \else: \prg_return_false: \fi:
      \fi:
    \else:
      \if_int_compare:w `#1 > `9 \exp_stop_f:
        \if_int_compare:w `#1 < `A \exp_stop_f:
          \prg_return_true: \else: \prg_return_false: \fi:
      \else:
        \if_int_compare:w `#1 < `0 \exp_stop_f:
          \if_int_compare:w `#1 < `\ \exp_stop_f:
            \prg_return_false: \else: \prg_return_true: \fi:
        \else: \prg_return_false: \fi:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
  {
    \if_int_compare:w `#1 > `Z \exp_stop_f:
      \if_int_compare:w `#1 > `z \exp_stop_f:
```

```
      \prg_return_false:
    \else:
      \if_int_compare:w `#1 < `a \exp_stop_f:
        \prg_return_false: \else: \prg_return_true: \fi:
    \fi:
  \else:
    \if_int_compare:w `#1 > `9 \exp_stop_f:
      \if_int_compare:w `#1 < `A \exp_stop_f:
        \prg_return_false: \else: \prg_return_true: \fi:
    \else:
      \if_int_compare:w `#1 < `0 \exp_stop_f:
        \prg_return_false: \else: \prg_return_true: \fi:
    \fi:
  \fi:
}
```

(*__regex_char_if_alphanumeric:NTF* 和 *__regex_char_if_special:NTF* 定义结束。)

9.3 编译

正则表达式最初是一串字符。在这一部分，我们将其转换为内部指令，得到一个“编译”（“compiled”）后的正则表达式。编译后的表达式在构建阶段转换为自动机的状态。编译后的正则表达式包括以下内容：

- `\__regex_class:NnnnN` ⟨*boolean*⟩ {⟨*tests*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*lazyness*⟩
- `\__regex_group:nnnN` {⟨*branches*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*lazyness*⟩,还有 `\__regex_group_no_capture:nnnN` 和 `\__regex_group_resetting:nnnN` 具有相同语法。
- `\__regex_branch:n` {⟨*contents*⟩}
- `\__regex_command_K:`
- `\__regex_assertion:Nn` ⟨*boolean*⟩ {⟨*assertion test*⟩}，其中 ⟨*assertion test*⟩ 是 `\__regex_b_test:` 或 `\__regex_Z_test:` 或 `\__regex_A_test:` 或 `\__regex_G_test:`

测试可以是以下类型：

- `\__regex_item_caseful_equal:n` {⟨*char code*⟩}
- `\__regex_item_caseless_equal:n` {⟨*char code*⟩}
- `\__regex_item_caseful_range:nn` {⟨*min*⟩} {⟨*max*⟩}

- `\__regex_item_caseless_range:nn {`⟨*min*⟩`} {`⟨*max*⟩`}`
- `\__regex_item_catcode:nT {`⟨*catcode bitmap*⟩`} {`⟨*tests*⟩`}`
- `\__regex_item_catcode_reverse:nT {`⟨*catcode bitmap*⟩`} {`⟨*tests*⟩`}`
- `\__regex_item_reverse:n {`⟨*tests*⟩`}`
- `\__regex_item_exact:nn {`⟨*catcode*⟩`} {`⟨*char code*⟩`}`
- `\__regex_item_exact_cs:n {`⟨*csnames*⟩`}`，更精确地给出为 ⟨*csname*⟩ `\scan_stop:` ⟨*csname*⟩ `\scan_stop:` ⟨*csname*⟩ 等等，放在括号组中。
- `\__regex_item_cs:n {`⟨*compiled regex*⟩`}`

9.3.1 编译时使用的变量

`\l__regex_group_level_int` 确保打开与关闭的组数量相同。

```
\int_new:N \l__regex_group_level_int
```

(*\l__regex_group_level_int 定义结束。*)

`\l__regex_mode_int` 在编译过程中，有十种模式，标记为 −63，−23，−6，−2，0，2，3，6，23，63。参见第 9.3.3 节。我们只定义其中的一些为常量。

`\c__regex_cs_in_class_mode_int`
`\c__regex_cs_mode_int`
`\c__regex_outer_mode_int`
`\c__regex_catcode_mode_int`
`\c__regex_class_mode_int`
`\c__regex_catcode_in_class_mode_int`

```
\int_new:N \l__regex_mode_int
\int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
\int_const:Nn \c__regex_cs_mode_int { -2 }
\int_const:Nn \c__regex_outer_mode_int { 0 }
\int_const:Nn \c__regex_catcode_mode_int { 2 }
\int_const:Nn \c__regex_class_mode_int { 3 }
\int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }
```

(*\l__regex_mode_int 以及其它的定义结束。*)

`\l__regex_catcodes_int` 我们希望允许像 `\c[^BE](..\cL[a-z]..)` 这样的构造，外部类别码测试适用于整个组，但会被内部类别码测试替代。为了使这个工作，我们需要跟踪允许的类别码列表：`\l__regex_catcodes_int` 和 `\l__regex_default_catcodes_int` 是位图，是所有允许的类别码 c 的 4^c 的和。后者是每个捕获组局部的，我们在每个字符或类中将 `\l__regex_catcodes_int` 重置为该值，仅在遇到 `\c` 转义时才更改它。布尔值记录了类别码测试的类别列表是否应被反转：比较 `\c[^BE]` 和 `\c[BE]`

`\l__regex_default_catcodes_int`
`\l__regex_catcodes_bool`

```
\int_new:N \l__regex_catcodes_int
\int_new:N \l__regex_default_catcodes_int
\bool_new:N \l__regex_catcodes_bool
```

(*\l__regex_catcodes_int*，*\l__regex_default_catcodes_int*，和 *\l__regex_catcodes_bool* 定义结束。)

\c__regex_catcode_C_int
\c__regex_catcode_B_int
\c__regex_catcode_E_int
\c__regex_catcode_M_int
\c__regex_catcode_T_int
\c__regex_catcode_P_int
\c__regex_catcode_U_int
\c__regex_catcode_D_int
\c__regex_catcode_S_int
\c__regex_catcode_L_int
\c__regex_catcode_O_int
\c__regex_catcode_A_int
\c__regex_all_catcodes_int

常量：每个类别的 4^c，以及所有 4 的幂的和。

```
\int_const:Nn \c__regex_catcode_C_int { "1 }
\int_const:Nn \c__regex_catcode_B_int { "4 }
\int_const:Nn \c__regex_catcode_E_int { "10 }
\int_const:Nn \c__regex_catcode_M_int { "40 }
\int_const:Nn \c__regex_catcode_T_int { "100 }
\int_const:Nn \c__regex_catcode_P_int { "1000 }
\int_const:Nn \c__regex_catcode_U_int { "4000 }
\int_const:Nn \c__regex_catcode_D_int { "10000 }
\int_const:Nn \c__regex_catcode_S_int { "100000 }
\int_const:Nn \c__regex_catcode_L_int { "400000 }
\int_const:Nn \c__regex_catcode_O_int { "1000000 }
\int_const:Nn \c__regex_catcode_A_int { "4000000 }
\int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(*\c__regex_catcode_C_int* 以及其它的定义结束。)

\l__regex_internal_regex

编译步骤将其结果存储在这个变量中。

```
\cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
```

(*\l__regex_internal_regex* 定义结束。)

\l__regex_show_prefix_seq

这个序列保存构成显示给用户的行的前缀。各种项目必须从右边移除，对于记号列表来说这是有技巧的，因此我们使用序列。

```
\seq_new:N \l__regex_show_prefix_seq
```

(*\l__regex_show_prefix_seq* 定义结束。)

\l__regex_show_lines_int

一个小技巧。为了知道给定类别是否在其中有一个单独的项目，我们在显示类别时计算行数。

```
\int_new:N \l__regex_show_lines_int
```

(*\l__regex_show_lines_int* 定义结束。)

9.3.2 编译时使用的通用助手

__regex_two_if_eq:NNNNTF

用于比较一对类似 `\__regex_compile_special:N ?` 的东西。获取要匹配字符的类别码通常是不方便的，因此我们只比较字符代码。此外，`\if:w` 的扩展行为非常有用，因为这意味着我们可以使用 `\c_left_brace_str` 等。

```
\prg_new_conditional:Npnn \__regex_two_if_eq:NNNN #1#2#3#4 { TF }
  {
```

```
    \if_meaning:w #1 #3
      \if:w #2 #4
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*__regex_two_if_eq:NNNNTF 定义结束。*)

__regex_get_digits:NTFw
__regex_get_digits_loop:w

如果后面有一些原始数字，则将它们逐个收集到整数变量 #1 中，并进入 true 分支。否则，进入 false 分支。

```
\cs_new_protected:Npn \__regex_get_digits:NTFw #1#2#3#4#5
  {
    \__regex_if_raw_digit:NNTF #4 #5
      { #1 = #5 \__regex_get_digits_loop:nw {#2} }
      { #3 #4 #5 }
  }
\cs_new:Npn \__regex_get_digits_loop:nw #1#2#3
  {
    \__regex_if_raw_digit:NNTF #2 #3
      { #3 \__regex_get_digits_loop:nw {#1} }
      { \scan_stop: #1 #2 #3 }
  }
```

(*__regex_get_digits:NTFw 和 __regex_get_digits_loop:w 定义结束。*)

__regex_if_raw_digit:NNTF

在抓取 {m,n} 量词的数字时使用的测试。它只接受非转义数字。

```
\prg_new_conditional:Npnn \__regex_if_raw_digit:NN #1#2 { TF }
  {
    \if_meaning:w \__regex_compile_raw:N #1
      \if_int_compare:w 1 < 1 #2 \exp_stop_f:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*__regex_if_raw_digit:NNTF* 定义结束。)

9.3.3 模式

在编译与给定正则表达式字符串对应的 NFA 时，我们可以处于十种不同的模式中，我们用一些魔法数字来标记这些模式：

-6 `[\c{...}]` 在类中的控制序列，

-2 `\c{...}` 控制序列，

0 `...` 外部，

2 `\c...` 类别码测试，

6 `[\c...]` 在类中的类别码测试，

-63 `[\c{[...]}]` 在模式 -6 中的类，

-23 `\c{[...]}` 在模式 -2 中的类，

3 `[...]` 在模式 0 中的类，

23 `\c[...]` 在模式 2 中的类，

63 `[\c[...]]` 在模式 6 中的类。

这个列表是详尽无遗的，因为 `\c` 转义序列不能被嵌套，并且字符类不能直接嵌套。选择这些数字是为了优化最有用的测试，并使从一个模式到另一个模式的转换尽可能简单。

- 偶数模式表示我们不直接在字符类中。在这种情况下，左括号将 3 附加到模式中。在字符类中，右括号将模式更改为 $m \to (m-15)/13$，截断。
- 在非正偶数模式（0，-2，-6）中允许分组，断言和锚点，并且不改变模式。否则，它们会触发错误。
- 在偶数模式中，左括号是特殊的，将 3 附加到模式中；在这些模式中，识别量词和点，并且右括号是正常的。在奇数模式中（在类中），左括号是正常的，但右括号结束类，将模式从 m 更改为 $(m-15)/13$，截断；此外，识别范围。
- 在非负模式中，左右括号是正常的。但是，在负模式中，左括号触发警告；右括号结束控制序列，从 -2 到 0 或 -6 到 3，对于奇数模式进行错误恢复。
- 属性（例如 `\d` 字符类）可以在任何模式中出现。

__regex_if_in_class:TF 测试是否直接在字符类中（在最内层嵌套）。在那里，许多转义序列不被识别，并且特殊字符是正常的。此外，对于每个原始字符，我们必须向前查找可能的原始短划线。

```
\cs_new:Npn \__regex_if_in_class:TF
  {
    \if_int_odd:w \l__regex_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

（*__regex_if_in_class:TF* 定义结束。）

__regex_if_in_cs:TF 右括号仅在直接位于控制序列内部时（在最内层嵌套中，不计算组）才是特殊的。

```
\cs_new:Npn \__regex_if_in_cs:TF
  {
    \if_int_odd:w \l__regex_mode_int
      \exp_after:wN \use_ii:nn
    \else:
      \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
      \fi:
    \fi:
  }
```

（*__regex_if_in_cs:TF* 定义结束。）

__regex_if_in_class_or_catcode:TF 断言仅允许在模式 0、-2 和 -6 中，即 偶数、非正模式中。

```
\cs_new:Npn \__regex_if_in_class_or_catcode:TF
  {
    \if_int_odd:w \l__regex_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
      \fi:
    \fi:
  }
```

(*__regex_if_in_class_or_catcode:TF* 定义结束。)

__regex_if_within_catcode:TF 如果我们在类别码测试中，要么紧随其后（模式 2 和 6），要么在它适用的类中（模式 23 和 63），则该测试将进入 true 分支。这用于调整模式 2 和 6 中左括号的行为。

```
\cs_new:Npn \__regex_if_within_catcode:TF
  {
    \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

(*__regex_if_within_catcode:TF* 定义结束。)

__regex_chk_c_allowed:T 仅在模式 0 和 3 中允许使用 \c 转义序列，即 不在任何其他 \c 转义序列中。

```
\cs_new_protected:Npn \__regex_chk_c_allowed:T
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
      \exp_after:wN \use:n
    \else:
      \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use:n
      \else:
        \msg_error:nn { regex } { c-bad-mode }
        \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
      \fi:
    \fi:
  }
```

(*__regex_chk_c_allowed:T* 定义结束。)

__regex_mode_quit_c: 此函数在 catcode 测试之后需要更改模式。

```
\cs_new_protected:Npn \__regex_mode_quit_c:
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
      \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
    \else:
      \if_int_compare:w \l__regex_mode_int =
        \c__regex_catcode_in_class_mode_int
        \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
      \fi:
    \fi:
  }
```

(*__regex_mode_quit_c:* 定义结束。)

9.3.4 框架

__regex_compile:w
__regex_compile_end:

用于编译用户正则表达式或在另一个正则表达式中的\c{...}转义序列中的正则表达式。开始在组内构建一个记号列表（在开始时进行 e-展开），设置一些变量（组级别、类别码），然后开始第一个分支。在结束时，确保没有悬空的类别码或组，关闭最后一个分支：我们完成了构建\l__regex_internal_regex。

```
\cs_new_protected:Npn \__regex_compile:w
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_group_level_int
      \int_set_eq:NN \l__regex_default_catcodes_int
        \c__regex_all_catcodes_int
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \cs_set:Npn \__regex_item_equal:n  { \__regex_item_caseful_equal:n }
      \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_branch:n { \if_false: } \fi: }
  }
\cs_new_protected:Npn \__regex_compile_end:
  {
      \__regex_if_in_class:TF
        {
          \msg_error:nn { regex } { missing-rbrack }
          \use:c { __regex_compile_]: }
          \prg_do_nothing: \prg_do_nothing:
        }
        { }
      \if_int_compare:w \l__regex_group_level_int > \c_zero_int
        \msg_error:nne { regex } { missing-rparen }
          { \int_use:N \l__regex_group_level_int }
        \prg_replicate:nn
          { \l__regex_group_level_int }
          {
              \tl_build_put_right:Nn \l__regex_build_tl
                {
                  \if_false: { \fi: }
                  \if_false: { \fi: } { 1 } { 0 } \c_true_bool
                }
              \tl_build_end:N \l__regex_build_tl
```

```
              \exp_args:NNNo
            \group_end:
            \tl_build_put_right:Nn \l__regex_build_tl
              { \l__regex_build_tl }
          }
      \fi:
      \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
      \tl_build_end:N \l__regex_build_tl
      \exp_args:NNNe
    \group_end:
    \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
  }
```

(*__regex_compile:w 和 __regex_compile_end: 定义结束。*)

__regex_compile:n

编译在__regex_compile:w和__regex_compile_end:之间进行，从模式 0 开始。然后__regex_escape_use:nnnn区分特殊字符、转义的字母数字字符和原始字符，解释\a、\x 和其他序列。最后的 4 个\prg_do_nothing:是必需的，因为后面定义的一些函数会查找 4 个记号。在结束之前，确保任何\c{...}都正确关闭。不需要检查括号是否正确关闭，因为__regex_compile_end:会处理。然而，捕获尾随的\cL构造的情况。

```
\cs_new_protected:Npn \__regex_compile:n #1
  {
    \__regex_compile:w
      \__regex_standard_escapechar:
      \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
      \__regex_escape_use:nnnn
        {
          \__regex_char_if_special:NTF ##1
            \__regex_compile_special:N \__regex_compile_raw:N ##1
        }
        {
          \__regex_char_if_alphanumeric:NTF ##1
            \__regex_compile_escaped:N \__regex_compile_raw:N ##1
        }
        { \__regex_compile_raw:N ##1 }
        { #1 }
      \prg_do_nothing: \prg_do_nothing:
      \prg_do_nothing: \prg_do_nothing:
      \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
        { \msg_error:nn { regex } { c-trailing } }
      \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
```

```
        {
          \msg_error:nn { regex } { c-missing-rbrace }
          \__regex_compile_end_cs:
          \prg_do_nothing: \prg_do_nothing:
          \prg_do_nothing: \prg_do_nothing:
        }
    \__regex_compile_end:
  }
```

(*__regex_compile:n* 定义结束。)

__regex_compile_use:n

使用正则表达式，无论是作为字符串给出（在这种情况下我们需要编译）还是作为正则表达式变量给出。这用于\regex_match_case:nn和相关函数，以允许显式正则表达式和正则表达式变量的混合使用。

```
\cs_new_protected:Npn \__regex_compile_use:n #1
  {
    \tl_if_single_token:nT {#1}
      {
        \exp_after:wN \__regex_compile_use_aux:w
        \token_to_meaning:N #1 ~ \q__regex_nil
      }
    \__regex_compile:n {#1} \l__regex_internal_regex
  }
\cs_new_protected:Npn \__regex_compile_use_aux:w #1 ~ #2 \q__regex_nil
  {
    \str_if_eq:nnT { #1 ~ } { macro:->\__regex_branch:n }
      { \use_ii:nnn }
  }
```

(*__regex_compile_use:n* 定义结束。)

__regex_compile_escaped:N
__regex_compile_special:N

如果特殊字符或转义的字母数字字符在正则表达式中有特定含义，则使用相应的函数。否则，将其解释为原始字符。我们区分特殊字符和转义的字母数字字符，因为当它们出现在范围的终点时，它们的行为不同。

```
\cs_new_protected:Npn \__regex_compile_special:N #1
  {
    \cs_if_exist_use:cF { __regex_compile_#1: }
      { \__regex_compile_raw:N #1 }
  }
\cs_new_protected:Npn \__regex_compile_escaped:N #1
  {
    \cs_if_exist_use:cF { __regex_compile_/#1: }
```

```
      { \__regex_compile_raw:N #1 }
  }
```

(*__regex_compile_escaped:N 和 __regex_compile_special:N 定义结束。*)

__regex_compile_one:n

在找到一个“测试”（“test”），比如 \d 或原始字符之后使用。如果后面跟着一个类别码测试（例如 \cL），则恢复模式。如果我们不在类别码中，则测试是 “standalone” 的，我们需要添加__regex_class:NnnnN并搜索限定符。在任何情况下，插入测试，可能还包括适当的类别码测试。

```
\cs_new_protected:Npn \__regex_compile_one:n #1
  {
    \__regex_mode_quit_c:
    \__regex_if_in_class:TF { }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
      }
    \tl_build_put_right:Ne \l__regex_build_tl
      {
        \if_int_compare:w \l__regex_catcodes_int <
          \c__regex_all_catcodes_int
          \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
            { \exp_not:N \exp_not:n {#1} }
        \else:
          \exp_not:N \exp_not:n {#1}
        \fi:
      }
    \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
    \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
  }
```

(*__regex_compile_one:n 定义结束。*)

__regex_compile_abort_tokens:n
__regex_compile_abort_tokens:e

此函数将收集的记号放回输入流，每个记号作为原始字符。空格不保留。

```
\cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
  {
    \use:e
      {
        \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
          \__regex_compile_raw:N
      }
  }
\cs_generate_variant:Nn \__regex_compile_abort_tokens:n { e }
```

(__regex_compile_abort_tokens:n 定义结束。)

9.3.5 限定符

__regex_compile_if_quantifier:TFw

这个函数向前查看并检查是否有任何限定符（特殊字符等于?+*{中的任何一个)。这对\u和\ur转义序列很有用。

```
\cs_new_protected:Npn \__regex_compile_if_quantifier:TFw #1#2#3#4
  {
    \token_if_eq_meaning:NNTF #3 \__regex_compile_special:N
      { \cs_if_exist:cTF { __regex_compile_quantifier_#4:w } }
      { \use_ii:nn }
    {#1} {#2} #3 #4
  }
```

(__regex_compile_if_quantifier:TFw 定义结束。)

__regex_compile_quantifier:w

这个函数向前查找并找到任何限定符（特殊字符等于?+*{中的任何一个)。

```
\cs_new_protected:Npn \__regex_compile_quantifier:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
      {
        \cs_if_exist_use:cF { __regex_compile_quantifier_#2:w }
          { \__regex_compile_quantifier_none: #1 #2 }
      }
      { \__regex_compile_quantifier_none: #1 #2 }
  }
```

(__regex_compile_quantifier:w 定义结束。)

__regex_compile_quantifier_none:
__regex_compile_quantifier_abort:eNN

当没有限定符，或者括号构造无效（等效于没有限定符，抓取的任何字符都保留为原始字符）时，调用这些函数。

```
\cs_new_protected:Npn \__regex_compile_quantifier_none:
  {
    \tl_build_put_right:Nn \l__regex_build_tl
      { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_abort:eNN #1#2#3
  {
    \__regex_compile_quantifier_none:
    \msg_warning:nnee { regex } { invalid-quantifier } {#1} {#3}
    \__regex_compile_abort_tokens:e {#1}
    #2 #3
  }
```

(*__regex_compile_quantifier_none:* 和 *__regex_compile_quantifier_abort:eNN* 定义结束。)

__regex_compile_quantifier_lazyness:nnNN

一旦找到“主要”(“main”)限定符(?、*、+ 或括号构造),我们检查它是否是懒惰的(后面跟着一个问号)。然后在编译的正则表达式中添加一个右括号(结束__regex_class:NnnnN等),范围的起点,终点和一个布尔值,对于懒惰操作符是 true,对于贪婪操作符是 false。

```
\cs_new_protected:Npn \__regex_compile_quantifier_lazyness:nnNN #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ?
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
      }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
        #3 #4
      }
  }
```

(*__regex_compile_quantifier_lazyness:nnNN* 定义结束。)

__regex_compile_quantifier_?:w
__regex_compile_quantifier_*:w
__regex_compile_quantifier_+:w

对于每个“基本”(“basic”)限定符,?、*、+,将正确的参数传递给__regex_compile_quantifier_lazyness:nnNN, -1 表示重复次数没有上限。

```
\cs_new_protected:cpn { __regex_compile_quantifier_?:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
\cs_new_protected:cpn { __regex_compile_quantifier_*:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
\cs_new_protected:cpn { __regex_compile_quantifier_+:w }
  { \__regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }
```

(*__regex_compile_quantifier_?:w*, *__regex_compile_quantifier_*:w*, 和 *__regex_compile_quantifier_+:w* 定义结束。)

__regex_compile_quantifier_{:w
__regex_compile_quantifier_braced_auxi:w
__regex_compile_quantifier_braced_auxii:w
__regex_compile_quantifier_braced_auxiii:w

三种可能的语法:{⟨*int*⟩}、{⟨*int*⟩,} 或 {⟨*int*⟩,⟨*int*⟩}。任何其他语法都会导致我们中止并将收集的任何内容放回输入流,作为 raw 字符,包括左括号。将一个数字抓取到\l__regex_internal_a_int中。如果数字后面跟着一个右括号,则范围是 $[a,a]$。如果后面跟着一个逗号,抓取另一个数字,并调用_ii 或_iii 辅助程序。这些辅助程序检查是否有右括号,导致范围 $[a,\infty]$ 或 $[a,b]$,编码为 $\{a\}\{-1\}$ 和 $\{a\}\{b-a\}$。

```
\cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
  {
    \__regex_get_digits:NTFw \l__regex_internal_a_int
```

```
      { \__regex_compile_quantifier_braced_auxi:w }
      { \__regex_compile_quantifier_abort:eNN { \c_left_brace_str } }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
  {
    \str_case_e:nnF { #1 #2 }
      {
        { \__regex_compile_special:N \c_right_brace_str }
          {
            \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
              { \int_use:N \l__regex_internal_a_int } { 0 }
          }
        { \__regex_compile_special:N , }
          {
            \__regex_get_digits:NTFw \l__regex_internal_b_int
              { \__regex_compile_quantifier_braced_auxiii:w }
              { \__regex_compile_quantifier_braced_auxii:w }
          }
      }
      {
        \__regex_compile_quantifier_abort:eNN
          { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxii:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
      {
        \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
          { \int_use:N \l__regex_internal_a_int } { -1 }
      }
      {
        \__regex_compile_quantifier_abort:eNN
          { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
      {
```

```
        \if_int_compare:w \l__regex_internal_a_int >
          \l__regex_internal_b_int
          \msg_error:nnee { regex } { backwards-quantifier }
            { \int_use:N \l__regex_internal_a_int }
            { \int_use:N \l__regex_internal_b_int }
          \int_zero:N \l__regex_internal_b_int
        \else:
          \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
        \fi:
        \exp_args:Noo \__regex_compile_quantifier_lazyness:nnNN
          { \int_use:N \l__regex_internal_a_int }
          { \int_use:N \l__regex_internal_b_int }
      }
      {
        \__regex_compile_quantifier_abort:eNN
          {
            \c_left_brace_str
            \int_use:N \l__regex_internal_a_int ,
            \int_use:N \l__regex_internal_b_int
          }
        #1 #2
      }
  }
```

(*__regex_compile_quantifier_{:w* 以及其它的定义结束。)

9.3.6 原始字符

__regex_compile_raw_error:N

在字符类中，并在类别码测试之后，一些转义的字母数字序列，如\b，没有任何含义。它们被替换为原始字符，然后输出错误。

```
\cs_new_protected:Npn \__regex_compile_raw_error:N #1
  {
    \msg_error:nne { regex } { bad-escape } {#1}
    \__regex_compile_raw:N #1
  }
```

(*__regex_compile_raw_error:N* 定义结束。)

__regex_compile_raw:N

如果我们在字符类中，下一个字符是未转义的破折号，这表示一个范围。否则，当前字符#1与其自身匹配。

```
\cs_new_protected:Npn \__regex_compile_raw:N #1#2#3
  {
    \__regex_if_in_class:TF
```

```
    {
      \__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N -
        { \__regex_compile_range:Nw #1 }
        {
          \__regex_compile_one:n
            { \__regex_item_equal:n { \int_value:w `#1 } }
          #2 #3
        }
    }
    {
      \__regex_compile_one:n
        { \__regex_item_equal:n { \int_value:w `#1 } }
      #2 #3
    }
}
```

(*__regex_compile_raw:N* 定义结束。)

__regex_compile_range:Nw
__regex_if_end_range:NNTF

我们刚刚读取了一个后跟破折号的原始字符；这应该后面跟着范围的端点。有效的端点包括：任何原始字符；除右括号之外的任何特殊字符。特别是，禁止使用转义字符。

```
\prg_new_protected_conditional:Npnn \__regex_if_end_range:NN #1#2 { TF }
  {
    \if_meaning:w \__regex_compile_raw:N #1
      \prg_return_true:
    \else:
      \if_meaning:w \__regex_compile_special:N #1
        \if_charcode:w ] #2
          \prg_return_false:
        \else:
          \prg_return_true:
        \fi:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
\cs_new_protected:Npn \__regex_compile_range:Nw #1#2#3
  {
    \__regex_if_end_range:NNTF #2 #3
      {
        \if_int_compare:w `#1 > `#3 \exp_stop_f:
```

```
          \msg_error:nnee { regex } { range-backwards } {#1} {#3}
        \else:
          \tl_build_put_right:Ne \l__regex_build_tl
            {
              \if_int_compare:w `#1 = `#3 \exp_stop_f:
                \__regex_item_equal:n
              \else:
                \__regex_item_range:nn { \int_value:w `#1 }
              \fi:
              { \int_value:w `#3 }
            }
        \fi:
      }
      {
        \msg_warning:nnee { regex } { range-missing-end }
          {#1} { \c_backslash_str #3 }
        \tl_build_put_right:Ne \l__regex_build_tl
          {
            \__regex_item_equal:n { \int_value:w `#1 \exp_stop_f: }
            \__regex_item_equal:n { \int_value:w `- \exp_stop_f: }
          }
        #2#3
      }
  }
```

(*__regex_compile_range:Nw* 和 *__regex_if_end_range:NNTF* 定义结束。)

9.3.7 字符属性

__regex_compile_.:
__regex_prop_.:

在字符类中，点没有特殊含义。在外部，插入__regex_prop_.:，它匹配任何字符或控制序列，并拒绝 −2（结束标记）。

```
\cs_new_protected:cpe { __regex_compile_.: }
  {
    \exp_not:N \__regex_if_in_class:TF
      { \__regex_compile_raw:N . }
      { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
  }
\cs_new_protected:cpn { __regex_prop_.: }
  {
    \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
```

(*__regex_compile_.: 和 __regex_prop_.: 定义结束。*)

__regex_compile_/d:
__regex_compile_/D:
__regex_compile_/h:
__regex_compile_/H:
__regex_compile_/s:
__regex_compile_/S:
__regex_compile_/v:
__regex_compile_/V:
__regex_compile_/w:
__regex_compile_/W:
__regex_compile_/N:

常量__regex_prop_d:,等包含与相应字符类匹配的一系列测试,并跳转到__regex_-break_point:TF标记。对于正常字符，我们检查限定符。

```
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpe { __regex_compile_/#1: }
      { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
    \cs_new_protected:cpe { __regex_compile_/#2: }
      {
        \__regex_compile_one:n
          { \__regex_item_reverse:n { \exp_not:c { __regex_prop_#1: } } }
      }
  }
\__regex_tmp:w d D
\__regex_tmp:w h H
\__regex_tmp:w s S
\__regex_tmp:w v V
\__regex_tmp:w w W
\cs_new_protected:cpn { __regex_compile_/N: }
  { \__regex_compile_one:n \__regex_prop_N: }
```

(*__regex_compile_/d: 以及其它的定义结束。*)

9.3.8 定位和简单断言

__regex_compile_anchor_letter:NNN
__regex_compile_/A:
__regex_compile_/G:
__regex_compile_/Z:
__regex_compile_/z:
__regex_compile_/b:
__regex_compile_/B:
__regex_compile_^:
__regex_compile_$:

在禁止断言的模式下，像 \A 这样的锚点会产生错误（\A 在类中无效）；否则，它们会根据需要添加 __regex_assertion:Nn 测试（唯一的负断言是 \B ）。测试函数将在后面定义。对于 $ 和 ^ 的实现与 \A 等不同，因为在类中它们是有效的。

```
\cs_new_protected:Npn \__regex_compile_anchor_letter:NNN #1#2#3
  {
    \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw_error:N #1 }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \__regex_assertion:Nn #2 {#3} }
      }
  }
\cs_new_protected:cpn { __regex_compile_/A: }
  { \__regex_compile_anchor_letter:NNN A \c_true_bool \__regex_A_test: }
\cs_new_protected:cpn { __regex_compile_/G: }
  { \__regex_compile_anchor_letter:NNN G \c_true_bool \__regex_G_test: }
\cs_new_protected:cpn { __regex_compile_/Z: }
```

```
  { \__regex_compile_anchor_letter:NNN Z \c_true_bool \__regex_Z_test: }
\cs_new_protected:cpn { __regex_compile_/z: }
  { \__regex_compile_anchor_letter:NNN z \c_true_bool \__regex_Z_test: }
\cs_new_protected:cpn { __regex_compile_/b: }
  { \__regex_compile_anchor_letter:NNN b \c_true_bool \__regex_b_test: }
\cs_new_protected:cpn { __regex_compile_/B: }
  { \__regex_compile_anchor_letter:NNN B \c_false_bool \__regex_b_test: }
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpn { __regex_compile_#1: }
      {
        \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw:N #1 }
          {
            \tl_build_put_right:Nn \l__regex_build_tl
              { \__regex_assertion:Nn \c_true_bool {#2} }
          }
      }
  }
\exp_args:Ne \__regex_tmp:w { \iow_char:N \^ } { \__regex_A_test: }
\exp_args:Ne \__regex_tmp:w { \iow_char:N \$ } { \__regex_Z_test: }
```

(*__regex_compile_anchor_letter:NNN 以及其它的定义结束。*)

9.3.9 字符类

__regex_compile_]: 在类外，右方括号没有意义。在类中，更改模式（$m \to (m-15)/13$，截断）以反映我们正在离开类的事实。查找限定符，除非我们在离开一个类后仍然在类中（即`[...\cL[...]...]`）。限定符。

```
\cs_new_protected:cpn { __regex_compile_]: }
  {
    \__regex_if_in_class:TF
      {
        \if_int_compare:w \l__regex_mode_int >
          \c__regex_catcode_in_class_mode_int
          \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
        \fi:
        \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
        \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
        \if_int_odd:w \l__regex_mode_int \else:
          \exp_after:wN \__regex_compile_quantifier:w
        \fi:
      }
```

```
      { \__regex_compile_raw:N ] }
  }
```

(*__regex_compile_]:* 定义结束。)

__regex_compile_[: 在类中，左方括号可能引入 POSIX 字符类，或者什么也不表示。紧跟在\c⟨*category*⟩之后，我们必须插入适当的类别码测试，然后解析类别；我们将类别码预先展开为优化。否则（模式 0，−2 和 −6），只需解析类别。模式稍后更新。

```
\cs_new_protected:cpn { __regex_compile_[: }
  {
    \__regex_if_in_class:TF
      { \__regex_compile_class_posix_test:w }
      {
        \__regex_if_within_catcode:TF
          {
            \exp_after:wN \__regex_compile_class_catcode:w
              \int_use:N \l__regex_catcodes_int ;
          }
          { \__regex_compile_class_normal:w }
      }
  }
```

(*__regex_compile_[:* 定义结束。)

__regex_compile_class_normal:w 在“正常”（enquotenormal）情况下，我们在编译代码中插入 __regex_class:NnnnN ⟨*boolean*⟩。对于正类，⟨*boolean*⟩为真，对于负类，其特征是前导^，为假。辅助函数__regex_compile_class:TFNN 还检查前导]，它有特殊含义。

```
\cs_new_protected:Npn \__regex_compile_class_normal:w
  {
    \__regex_compile_class:TFNN
      { \__regex_class:NnnnN \c_true_bool }
      { \__regex_class:NnnnN \c_false_bool }
  }
```

(*__regex_compile_class_normal:w* 定义结束。)

__regex_compile_class_catcode:w 对于模式 2 或 6 中的左方括号，调用此函数（类别码测试，在类中的类别码测试）。在模式 2 中，整个构造需要放在类中（比如单个字符）。然后确定类是正的还是负的，插入 __regex_item_catcode:nT 或逆变体，每个都带有当前类别码位图 #1 作为参数，并重置类别码。

```
\cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
  {
```

```
    \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
    \fi:
    \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
    \__regex_compile_class:TFNN
      { \__regex_item_catcode:nT {#1} }
      { \__regex_item_catcode_reverse:nT {#1} }
  }
```

(*__regex_compile_class_catcode:w* 定义结束。)

__regex_compile_class:TFNN
__regex_compile_class:NN

如果第一个字符是 ^，那么类是负的（使用 #2 ），否则是正的（使用 #1 ）。如果下一个字符是右方括号，那么它应该更改为原始字符。

```
\cs_new_protected:Npn \__regex_compile_class:TFNN #1#2#3#4
  {
    \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
    \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ^
      {
        \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
        \__regex_compile_class:NN
      }
      {
        \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
        \__regex_compile_class:NN #3 #4
      }
  }
\cs_new_protected:Npn \__regex_compile_class:NN #1#2
  {
    \token_if_eq_charcode:NNTF #2 ]
      { \__regex_compile_raw:N #2 }
      { #1 #2 }
  }
```

(*__regex_compile_class:TFNN* 和 *__regex_compile_class:NN* 定义结束。)

__regex_compile_class_posix_test:w
__regex_compile_class_posix:NNNNw
__regex_compile_class_posix_loop:w
__regex_compile_class_posix_end:w

在这里，我们检查类似于[:alpha:]的语法。我们还检测到 [= 和 [.，在 POSIX 正则表达式中具有意义，但在 l3regex 中没有实现。如果我们看到[:，则收集原始字符，直到有望到达:]。如果缺少这一部分，或者未知 POSIX 类，则中止。如果一切正确，将测试添加到当前类别，对于负类别，添加额外的 __regex_item_reverse:n （我们确保将其参数用括号括起来，否则 \regex_show:N将无法识别正则表达式为有效）。

```
\cs_new_protected:Npn \__regex_compile_class_posix_test:w #1#2
```

```
  {
    \token_if_eq_meaning:NNT \__regex_compile_special:N #1
      {
        \str_case:nn { #2 }
          {
            : { \__regex_compile_class_posix:NNNNw }
            = {
                \msg_warning:nne { regex }
                  { posix-unsupported } { = }
              }
            . {
                \msg_warning:nne { regex }
                  { posix-unsupported } { . }
              }
          }
      }
    \__regex_compile_raw:N [ #1 #2
  }
\cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
  {
    \__regex_two_if_eq:NNNNTF #5 #6 \__regex_compile_special:N ^
      {
        \bool_set_false:N \l__regex_internal_bool
        \__kernel_tl_set:Ne \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w
      }
      {
        \bool_set_true:N \l__regex_internal_bool
        \__kernel_tl_set:Ne \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w #5 #6
      }
  }
\cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
  {
    \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
      { #2 \__regex_compile_class_posix_loop:w }
      { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
  }
\cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N :
      { \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ] }
```

```
      { \use_ii:nn }
      {
        \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
          {
            \__regex_compile_one:n
              {
                \bool_if:NTF \l__regex_internal_bool \use:n \__regex_item_reverse:n
                { \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : } }
              }
          }
          {
            \msg_warning:nne { regex } { posix-unknown }
              { \l__regex_internal_a_tl }
            \__regex_compile_abort_tokens:e
              {
                [: \bool_if:NF \l__regex_internal_bool { ^ }
                \l__regex_internal_a_tl :]
              }
          }
      }
      {
        \msg_error:nnee { regex } { posix-missing-close }
          { [: \l__regex_internal_a_tl } { #2 #4 }
        \__regex_compile_abort_tokens:e { [: \l__regex_internal_a_tl }
        #1 #2 #3 #4
      }
  }
```

(*__regex_compile_class_posix_test:w* 以及其它的定义结束。)

9.3.10 分组和选择

__regex_compile_group_begin:N
__regex_compile_group_end:

正则表达式分组的内容在\l__regex_build_tl中被转换为编译后的代码，最终形式为 __regex_branch:n {⟨*concatenation*⟩}。这个构建过程使用 TEX 组内的 \tl_build_… 函数完成，自动确保选项（大小写敏感性和默认类别码）在组结束时被重置。参数 #1 是 __regex_group:nnnN 或其变体。一个小技巧用于支持 \cL(abc) 作为 (\cLa\cLb\cLc) 的缩写：退出任何挂起的类别码测试，将组开始时的类别码保存为该组的默认类别码，并确保在组外部将类别码恢复为默认值。

```
\cs_new_protected:Npn \__regex_compile_group_begin:N #1
  {
    \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
    \__regex_mode_quit_c:
```

```
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
      \int_incr:N \l__regex_group_level_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_branch:n { \if_false: } \fi: }
  }
\cs_new_protected:Npn \__regex_compile_group_end:
  {
    \if_int_compare:w \l__regex_group_level_int > \c_zero_int
        \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
        \tl_build_end:N \l__regex_build_tl
        \exp_args:NNNe
      \group_end:
      \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \exp_after:wN \__regex_compile_quantifier:w
    \else:
      \msg_warning:nn { regex } { extra-rparen }
      \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
    \fi:
  }
```

（*__regex_compile_group_begin:N* 和 *__regex_compile_group_end:* 定义结束。）

__regex_compile_(: 在字符类中，括号不是特殊字符。在字符类内的类别码测试中，左括号会引发错误，以捕捉 `[a\cL(bcd)e]`。否则，检查是否存在 `?`，表示特殊分组，并运行相应特殊分组的代码。

```
\cs_new_protected:cpn { __regex_compile_(: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
      {
        \if_int_compare:w \l__regex_mode_int =
          \c__regex_catcode_in_class_mode_int
          \msg_error:nn { regex } { c-lparen-in-class }
          \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
        \else:
          \exp_after:wN \__regex_compile_lparen:w
        \fi:
      }
  }
\cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
```

```
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N ?
      {
        \cs_if_exist_use:cF
          { __regex_compile_special_group_\token_to_str:N #4 :w }
          {
            \msg_warning:nne { regex } { special-group-unknown }
              { (? #4 }
            \__regex_compile_group_begin:N \__regex_group:nnnN
              \__regex_compile_raw:N ? #3 #4
          }
      }
      {
        \__regex_compile_group_begin:N \__regex_group:nnnN
          #1 #2 #3 #4
      }
  }
```

(*__regex_compile_(:* 定义结束。)

__regex_compile_|: 在字符类中，竖线不是特殊字符。否则，结束当前分支并开始另一个分支。

```
\cs_new_protected:cpn { __regex_compile_|: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N | }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
      }
  }
```

(*__regex_compile_|:* 定义结束。)

__regex_compile_): 在字符类中，括号不是特殊字符。在字符类外，关闭一个分组。

```
\cs_new_protected:cpn { __regex_compile_): }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
      { \__regex_compile_group_end: }
  }
```

(*__regex_compile_):* 定义结束。)

__regex_compile_special_group_::w
__regex_compile_special_group_|:w

非捕获和重置分组在编译过程中很容易处理；对于这些分组，更难的部分在构建时出现。

```
\cs_new_protected:cpn { __regex_compile_special_group_:w }
  { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }
\cs_new_protected:cpn { __regex_compile_special_group_|:w }
  { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }
```

（*__regex_compile_special_group_:w* 和 *__regex_compile_special_group_|:w* 定义结束。）

__regex_compile_special_group_i:w
__regex_compile_special_group_-:w

通过设置选项 `(?i)`，可以使匹配对大小写不敏感；通过 `(?-i)` 恢复原始行为。这是唯一支持的选项。

```
\cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N )
      {
        \cs_set:Npn \__regex_item_equal:n
          { \__regex_item_caseless_equal:n }
        \cs_set:Npn \__regex_item_range:nn
          { \__regex_item_caseless_range:nn }
      }
      {
        \msg_warning:nne { regex } { unknown-option } { (?i #2 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N i
        #1 #2
      }
  }
\cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_raw:N i
      { \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ) }
      { \use_ii:nn }
      {
        \cs_set:Npn \__regex_item_equal:n
          { \__regex_item_caseful_equal:n }
        \cs_set:Npn \__regex_item_range:nn
          { \__regex_item_caseful_range:nn }
      }
      {
        \msg_warning:nne { regex } { unknown-option } { (?-#2#4 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N -
        #1 #2 #3 #4
```

```
      }
  }
```

(*__regex_compile_special_group_i:w* 和 *__regex_compile_special_group_-:w* 定义结束。)

9.3.11 Catcode 和 csname

__regex_compile_/c:
__regex_compile_c_test:NN

由 \c 转义序列后面可以是表示字符类别的大写字母，左方括号（表示类别列表），或者括号组（包含控制序列名称的正则表达式）。否则，引发错误。

```
\cs_new_protected:cpn { __regex_compile_/c: }
  { \__regex_chk_c_allowed:T { \__regex_compile_c_test:NN } }
\cs_new_protected:Npn \__regex_compile_c_test:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      {
        \int_if_exist:cTF { c__regex_catcode_#2_int }
          {
            \int_set_eq:Nc \l__regex_catcodes_int
              { c__regex_catcode_#2_int }
            \l__regex_mode_int
              = \if_case:w \l__regex_mode_int
                  \c__regex_catcode_mode_int
                \else:
                  \c__regex_catcode_in_class_mode_int
                \fi:
            \token_if_eq_charcode:NNT C #2 { \__regex_compile_c_C:NN }
          }
      }
      { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
          {
            \msg_error:nne { regex } { c-missing-category } {#2}
            #1 #2
          }
  }
```

(*__regex_compile_/c:* 和 *__regex_compile_c_test:NN* 定义结束。)

__regex_compile_c_C:NN

如果 \cC 后面不是 . 或 (...)，则发出警告，因为该结构无法匹配任何内容，除非在类似 \cC[\c{...}] 的情况下，它不起作用。

```
\cs_new_protected:Npn \__regex_compile_c_C:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
      {
```

```
        \token_if_eq_charcode:NNTF #2 .
          { \use_none:n }
          { \token_if_eq_charcode:NNF #2 ( } % )
      }
      { \use:n }
    { \msg_error:nnn { regex } { c-C-invalid } {#2} }
    #1 #2
  }
```

(__regex_compile_c_C:NN 定义结束。)

__regex_compile_c_[:w
__regex_compile_c_lbrack_loop:NN
__regex_compile_c_lbrack_add:N
__regex_compile_c_lbrack_end:

当遇到 \c[时，任务是收集表示字符类别的大写字母。首先检查是否有 ^，它会否定类别代码列表。

```
\cs_new_protected:cpn { __regex_compile_c_[:w } #1#2
  {
    \l__regex_mode_int
      = \if_case:w \l__regex_mode_int
          \c__regex_catcode_mode_int
        \else:
          \c__regex_catcode_in_class_mode_int
        \fi:
    \int_zero:N \l__regex_catcodes_int
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N ^
      {
        \bool_set_false:N \l__regex_catcodes_bool
        \__regex_compile_c_lbrack_loop:NN
      }
      {
        \bool_set_true:N \l__regex_catcodes_bool
        \__regex_compile_c_lbrack_loop:NN
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      {
        \int_if_exist:cTF { c__regex_catcode_#2_int }
          {
            \exp_args:Nc \__regex_compile_c_lbrack_add:N
              { c__regex_catcode_#2_int }
            \__regex_compile_c_lbrack_loop:NN
          }
```

```
      }
      {
        \token_if_eq_charcode:NNTF #2 ]
          { \__regex_compile_c_lbrack_end: }
      }
          {
            \msg_error:nne { regex } { c-missing-rbrack } {#2}
            \__regex_compile_c_lbrack_end:
            #1 #2
          }
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
  {
    \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
    \else:
      \int_add:Nn \l__regex_catcodes_int {#1}
    \fi:
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_end:
  {
    \if_meaning:w \c_false_bool \l__regex_catcodes_bool
      \int_set:Nn \l__regex_catcodes_int
        { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
    \fi:
  }
```

(*__regex_compile_c_[:w 以及其它的定义结束。*)

__regex_compile_c_{:

对于左括号的情况，基于我们迄今为止所做的工作，很容易处理：在一个组内，编译正则表达式，同时将模式更改为禁止嵌套 \c。此外，禁用子匹配跟踪，因为组不会逃离 \c{...} 的作用域。

```
\cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
  {
    \__regex_compile:w
      \__regex_disable_submatches:
      \l__regex_mode_int
        = \if_case:w \l__regex_mode_int
            \c__regex_cs_mode_int
          \else:
            \c__regex_cs_in_class_mode_int
          \fi:
  }
```

(*__regex_compile_c_{:* 定义结束。)

__regex_compile_{:

我们禁止在 \c{...} 转义内部出现未转义的左括号，因为它们可能导致混淆的问题，即 \c{{}x} 中的第一个右括号应该结束 \c，还是应该匹配括号。

```
\cs_new_protected:cpn { __regex_compile_ \c_left_brace_str : }
  {
    \__regex_if_in_cs:TF
      { \msg_error:nnn { regex } { cu-lbrace } { c } }
      { \exp_after:wN \__regex_compile_raw:N \c_left_brace_str }
  }
```

(*__regex_compile_{:* 定义结束。)

__regex_cs
__regex_compile_}:
__regex_compile_end_cs:
__regex_compile_cs_aux:Nn
__regex_compile_cs_aux:NNnnnN

未转义的右括号只在编译 csname 的正则表达式时才是特殊的，但不在字符类内：\c{[{}]} 匹配控制序列 \{ 和 \}。因此，结束编译内部正则表达式（这会关闭任何悬空的字符类或组）。然后在外部正则表达式中插入相应的测试。作为优化，如果控制序列测试仅由多个显式可能性（分支）组成，则使用带有由 \scan_stop: 分隔的所有可能性组成的参数的 __regex_item_exact_cs:n。

```
\flag_new:n { __regex_cs }
\cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
  {
    \__regex_if_in_cs:TF
      { \__regex_compile_end_cs: }
      { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
  }
\cs_new_protected:Npn \__regex_compile_end_cs:
  {
    \__regex_compile_end:
    \flag_clear:n { __regex_cs }
    \__kernel_tl_set:Ne \l__regex_internal_a_tl
      {
        \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
        \q__regex_nil \q__regex_nil \q__regex_recursion_stop
      }
    \exp_args:Ne \__regex_compile_one:n
      {
        \flag_if_raised:nTF { __regex_cs }
          { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
          {
            \__regex_item_exact_cs:n
              { \tl_tail:N \l__regex_internal_a_tl }
          }
```

```
      }
  }
\cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
  {
    \cs_if_eq:NNTF #1 \__regex_branch:n
      {
        \scan_stop:
        \__regex_compile_cs_aux:NNnnnN #2
        \q__regex_nil \q__regex_nil \q__regex_nil
        \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
        \__regex_compile_cs_aux:Nn
      }
      {
        \__regex_quark_if_nil:NF #1 { \flag_ensure_raised:n { __regex_cs } }
        \__regex_use_none_delimit_by_q_recursion_stop:w
      }
  }
\cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
  {
    \bool_lazy_all:nTF
      {
        { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
        {#2}
        { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
        { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
        { \int_compare_p:nNn {#5} = { 0 } }
      }
      {
        \prg_replicate:nn {#4}
          { \char_generate:nn { \use_ii:nn #3 } {12} }
        \__regex_compile_cs_aux:NNnnnN
      }
      {
        \__regex_quark_if_nil:NF #1
          {
            \flag_ensure_raised:n { __regex_cs }
            \__regex_use_i_delimit_by_q_recursion_stop:nw
          }
        \__regex_use_none_delimit_by_q_recursion_stop:w
      }
  }
```

(*__regex_cs* 以及其它的定义结束。)

9.3.12 原始记号列表与 \u

__regex_compile_/u: 在字符类和直接跟在类别代码测试后面时，\u 转义无效。否则检查后面是否有 r（对应 \ur），并调用一个负责查找变量名称的辅助函数。

```
\cs_new_protected:cpn { __regex_compile_/u: } #1#2
  {
    \__regex_if_in_class_or_catcode:TF
      { \__regex_compile_raw_error:N u #1 #2 }
      {
        \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_raw:N r
          { \__regex_compile_u_brace:NNN \__regex_compile_ur_end: }
          { \__regex_compile_u_brace:NNN \__regex_compile_u_end: #1 #2 }
      }
  }
```

(*__regex_compile_/u: 定义结束。*)

__regex_compile_u_brace:NNN 这要求左括号的存在，然后启动一个循环来查找变量名。

```
\cs_new:Npn \__regex_compile_u_brace:NNN #1#2#3
  {
    \__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N \c_left_brace_str
      {
        \tl_set:Nn \l__regex_internal_b_tl {#1}
        \__kernel_tl_set:Ne \l__regex_internal_a_tl { \if_false: } \fi:
        \__regex_compile_u_loop:NN
      }
      {
        \msg_error:nn { regex } { u-missing-lbrace }
        \token_if_eq_meaning:NNTF #1 \__regex_compile_ur_end:
          { \__regex_compile_raw:N u \__regex_compile_raw:N r }
          { \__regex_compile_raw:N u }
        #2 #3
      }
  }
```

(*__regex_compile_u_brace:NNN 定义结束。*)

__regex_compile_u_loop:NN 我们使用 e-展开赋值来收集 \u 的参数中的字符。原则上，我们可以等待遇到右括号，但这是不安全的：如果右括号丢失，那么我们将达到正则表达式的结束标记，并继续，导致晦涩的致命错误。相反，我们只允许原始和特殊字符，并在遇到特殊右括号、任何转义字符或结束标记时停止。

```
\cs_new:Npn \__regex_compile_u_loop:NN #1#2
  {
```

```
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      { #2 \__regex_compile_u_loop:NN }
      {
        \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
          {
            \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
              { \if_false: { \fi: } \l__regex_internal_b_tl }
              {
                \if_charcode:w \c_left_brace_str #2
                  \msg_expandable_error:nnn { regex } { cu-lbrace } { u }
                \else:
                  #2
                \fi:
                \__regex_compile_u_loop:NN
              }
          }
          {
            \if_false: { \fi: }
            \msg_error:nne { regex } { u-missing-rbrace } {#2}
            \l__regex_internal_b_tl
            #1 #2
          }
      }
  }
```

(*__regex_compile_u_loop:NN* 定义结束。)

__regex_compile_ur_end:
__regex_compile_ur:n
__regex_compile_ur_aux:w

对于 \ur{...} 结构，一旦我们提取了变量的名称，我们就会在编译后的正则表达式中（作为 __regex_compile_ur:n 的参数传递）替换所有组为非捕获组。如果它只有一个分支（即 \tl_if_empty:oTF 为 false）并且没有量词，那么只需插入此分支的内容（由 \use_ii:nn 获得，稍后扩展）。在所有其他情况下，插入一个非捕获组，并查找量词以确定重复次数等。

```
\cs_new_protected:Npn \__regex_compile_ur_end:
  {
    \group_begin:
      \cs_set:Npn \__regex_group:nnnN { \__regex_group_no_capture:nnnN }
      \cs_set:Npn \__regex_group_resetting:nnnN { \__regex_group_no_capture:nnnN }
      \exp_args:NNe
    \group_end:
    \__regex_compile_ur:n { \use:c { \l__regex_internal_a_tl } }
  }
\cs_new_protected:Npn \__regex_compile_ur:n #1
```

```
  {
    \tl_if_empty:oTF { \__regex_compile_ur_aux:w #1 {} ? ? \q__regex_nil }
      { \__regex_compile_if_quantifier:TFw }
      { \use_i:nn }
        {
          \tl_build_put_right:Nn \l__regex_build_tl
            { \__regex_group_no_capture:nnnN { \if_false: } \fi: #1 }
          \__regex_compile_quantifier:w
        }
        { \tl_build_put_right:Nn \l__regex_build_tl { \use_ii:nn #1 } }
  }
\cs_new:Npn \__regex_compile_ur_aux:w \__regex_branch:n #1#2#3 \q__regex_nil {#2}
```

(*__regex_compile_ur_end:*, *__regex_compile_ur:n*, 和 *__regex_compile_ur_aux:w* 定义结束。)

__regex_compile_u_end:
__regex_compile_u_payload:

提取了变量的名称后，我们检查是否有量词，在这种情况下，我们设置了一个带有单个分支的非捕获组。在这个分支内（如果没有量词，我们将省略它和组），__regex_compile_u_payload: 放置了与变量内容相对应的正确测试，我们将其存储在 \l__regex_internal_a_tl 中。\u 的行为取决于我们是否在 \c{...} 转义内（在这种情况下，变量将转换为字符串）。

```
\cs_new_protected:Npn \__regex_compile_u_end:
  {
    \__regex_compile_if_quantifier:TFw
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          {
            \__regex_group_no_capture:nnnN { \if_false: } \fi:
            \__regex_branch:n { \if_false: } \fi:
          }
        \__regex_compile_u_payload:
        \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
        \__regex_compile_quantifier:w
      }
      { \__regex_compile_u_payload: }
  }
\cs_new_protected:Npn \__regex_compile_u_payload:
  {
    \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
    \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
      \__regex_compile_u_not_cs:
    \else:
      \__regex_compile_u_in_cs:
```

```
    \fi:
  }
```

(*__regex_compile_u_end:* 和 *__regex_compile_u_payload:* 定义结束。)

__regex_compile_u_in_cs:

当 \u 出现在控制序列内时，我们将变量转换为带有转义空格的字符串。然后对于每个字符，插入一个仅匹配该字符一次的类。

```
\cs_new_protected:Npn \__regex_compile_u_in_cs:
  {
    \__kernel_tl_gset:Ne \g__regex_internal_tl
      {
        \exp_args:No \__kernel_str_to_other_fast:n
          { \l__regex_internal_a_tl }
      }
    \tl_build_put_right:Ne \l__regex_build_tl
      {
        \tl_map_function:NN \g__regex_internal_tl
          \__regex_compile_u_in_cs_aux:n
      }
  }
\cs_new:Npn \__regex_compile_u_in_cs_aux:n #1
  {
    \__regex_class:NnnnN \c_true_bool
      { \__regex_item_caseful_equal:n { \int_value:w `#1 } }
      { 1 } { 0 } \c_false_bool
  }
```

(*__regex_compile_u_in_cs:* 定义结束。)

__regex_compile_u_not_cs:

在模式 0 中，\u 转义为 \l__regex_internal_a_tl 中的每个标记添加一个状态到 NFA。如果给定的 ⟨*token*⟩ 是一个控制序列，那么插入一个字符串比较测试，否则插入 __regex_item_exact:nn，它比较类别代码和字符代码。

```
\cs_new_protected:Npn \__regex_compile_u_not_cs:
  {
    \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
      {
        \tl_build_put_right:Ne \l__regex_build_tl
          {
            \__regex_class:NnnnN \c_true_bool
              {
                \if_int_compare:w "##3 = \c_zero_int
                  \__regex_item_exact_cs:n
                    { \exp_after:wN \cs_to_str:N ##1 }
```

```
                \else:
                  \__regex_item_exact:nn { \int_value:w "##3 } { ##2 }
                \fi:
              }
              { 1 } { 0 } \c_false_bool
          }
      }
  }
```

(*__regex_compile_u_not_cs:* 定义结束。)

9.3.13 其他

__regex_compile_/K:

控制序列 \K 目前是唯一一个执行某些操作而非匹配的“命令”（“command”）。允许在与 \b 相同的上下文中使用。在编译阶段，它被保留为一个单一的控制序列，稍后定义。

```
\cs_new_protected:cpn { __regex_compile_/K: }
  {
    \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
      { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
      { \__regex_compile_raw_error:N K }
  }
```

(*__regex_compile_/K:* 定义结束。)

9.3.14 显示正则表达式

__regex_clean_bool:n
__regex_clean_int:n
__regex_clean_int_aux:N
__regex_clean_regex:n
__regex_clean_regex_loop:w
__regex_clean_branch:n
__regex_clean_branch_loop:n
__regex_clean_assertion:Nn
__regex_clean_class:NnnnN
__regex_clean_group:nnnN
__regex_clean_class:n
__regex_clean_class_loop:nnn
__regex_clean_exact_cs:n
__regex_clean_exact_cs:w

在显示正则表达式之前，我们检查它是否在内部结构上是“干净”的。我们通过与同一正则表达式的经过清理的版本进行比较来实现这一点（在\regex_show:N和\regex_log:N的实现中）。同时，我们还需要为其他类型提供类似的函数：所有 __regex_clean_⟨*type*⟩:n 函数从任意输入产生有效的 ⟨*type*⟩ 标记（布尔值、显式整数等），且输出与输入在输入有效的情况下相符。

```
\cs_new:Npn \__regex_clean_bool:n #1
  {
    \tl_if_single:nTF {#1}
      { \bool_if:NTF #1 \c_true_bool \c_false_bool }
      { \c_true_bool }
  }
\cs_new:Npn \__regex_clean_int:n #1
  {
    \tl_if_head_eq_meaning:nNTF {#1} -
      { - \exp_args:No \__regex_clean_int:n { \use_none:n #1 } }
```

```
      { \int_eval:n { 0 \str_map_function:nN {#1} \__regex_clean_int_aux:N } }
  }
\cs_new:Npn \__regex_clean_int_aux:N #1
  {
    \if_int_compare:w 1 < 1 #1 ~
      #1
    \else:
      \exp_after:wN \str_map_break:
    \fi:
  }
\cs_new:Npn \__regex_clean_regex:n #1
  {
    \__regex_clean_regex_loop:w #1
    \__regex_branch:n { \q_recursion_tail } \q_recursion_stop
  }
\cs_new:Npn \__regex_clean_regex_loop:w #1 \__regex_branch:n #2
  {
    \quark_if_recursion_tail_stop:n {#2}
    \__regex_branch:n { \__regex_clean_branch:n {#2} }
    \__regex_clean_regex_loop:w
  }
\cs_new:Npn \__regex_clean_branch:n #1
  {
    \__regex_clean_branch_loop:n #1
    ? ? ? ? ? ? \prg_break_point:
  }
\cs_new:Npn \__regex_clean_branch_loop:n #1
  {
    \tl_if_single:nF {#1} { \prg_break: }
    \token_case_meaning:NnF #1
      {
        \__regex_command_K: { #1 \__regex_clean_branch_loop:n }
        \__regex_assertion:Nn { #1 \__regex_clean_assertion:Nn }
        \__regex_class:NnnnN { #1 \__regex_clean_class:NnnnN }
        \__regex_group:nnnN { #1 \__regex_clean_group:nnnN }
        \__regex_group_no_capture:nnnN { #1 \__regex_clean_group:nnnN }
        \__regex_group_resetting:nnnN { #1 \__regex_clean_group:nnnN }
      }
      { \prg_break: }
  }
\cs_new:Npn \__regex_clean_assertion:Nn #1#2
  {
```

```
    \__regex_clean_bool:n {#1}
    \tl_if_single:nF {#2} { { \__regex_A_test: } \prg_break: }
    \token_case_meaning:NnTF #2
      {
        \__regex_A_test: { }
        \__regex_G_test: { }
        \__regex_Z_test: { }
        \__regex_b_test: { }
      }
      { {#2} }
      { { \__regex_A_test: } \prg_break: }
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_class:NnnnN #1#2#3#4#5
  {
    \__regex_clean_bool:n {#1}
    { \__regex_clean_class:n {#2} }
    { \int_max:nn { 0 } { \__regex_clean_int:n {#3} } }
    { \int_max:nn { -1 } { \__regex_clean_int:n {#4} } }
    \__regex_clean_bool:n {#5}
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_group:nnnN #1#2#3#4
  {
    { \__regex_clean_regex:n {#1} }
    { \int_max:nn { 0 } { \__regex_clean_int:n {#2} } }
    { \int_max:nn { -1 } { \__regex_clean_int:n {#3} } }
    \__regex_clean_bool:n {#4}
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_class:n #1
  { \__regex_clean_class_loop:nnn #1 ????? \prg_break_point: }
```

清理类别时存在许多情况,其中包括十几个类似于 `\__regex_prop_d:` 或 `\__regex_posix_alpha:` 的情况。为了避免列举所有这些情况，我们允许任何以 13 个字符 `__regex_prop_` 或 `__regex_posix` 起始的命令（方便的是，除了末尾的下划线，它们的长度相同)。

```
\cs_new:Npn \__regex_clean_class_loop:nnn #1#2#3
  {
    \tl_if_single:nF {#1} { \prg_break: }
    \token_case_meaning:NnTF #1
      {
```

```
    \__regex_item_cs:n { #1 { \__regex_clean_regex:n {#2} } }
    \__regex_item_exact_cs:n { #1 { \__regex_clean_exact_cs:n {#2} } }
    \__regex_item_caseful_equal:n { #1 { \__regex_clean_int:n {#2} } }
    \__regex_item_caseless_equal:n { #1 { \__regex_clean_int:n {#2} } }
    \__regex_item_reverse:n { #1 { \__regex_clean_class:n {#2} } }
  }
  { \__regex_clean_class_loop:nnn {#3} }
  {
    \token_case_meaning:NnTF #1
      {
        \__regex_item_caseful_range:nn { }
        \__regex_item_caseless_range:nn { }
        \__regex_item_exact:nn { }
      }
      {
        #1 { \__regex_clean_int:n {#2} } { \__regex_clean_int:n {#3} }
        \__regex_clean_class_loop:nnn
      }
      {
        \token_case_meaning:NnTF #1
          {
            \__regex_item_catcode:nT { }
            \__regex_item_catcode_reverse:nT { }
          }
          {
            #1 { \__regex_clean_int:n {#2} } { \__regex_clean_class:n {#3} }
            \__regex_clean_class_loop:nnn
          }
          {
            \exp_args:Nf \str_case:nnTF
              {
                \exp_args:Nf \str_range:nnn
                  { \cs_to_str:N #1 } { 1 } { 13 }
              }
              {
                { __regex_prop_ } { }
                { __regex_posix } { }
              }
              {
                #1
                \__regex_clean_class_loop:nnn {#2} {#3}
              }
```

```
                  { \prg_break: }
              }
          }
      }
  }
\cs_new:Npn \__regex_clean_exact_cs:n #1
  {
    \exp_last_unbraced:Nf \use_none:n
      {
        \__regex_clean_exact_cs:w #1
        \scan_stop: \q_recursion_tail \scan_stop:
        \q_recursion_stop
      }
  }
\cs_new:Npn \__regex_clean_exact_cs:w #1 \scan_stop:
  {
    \quark_if_recursion_tail_stop:n {#1}
    \scan_stop: \tl_to_str:n {#1}
    \__regex_clean_exact_cs:w
  }
```

(*__regex_clean_bool:n 以及其它的定义结束。*)

__regex_show:N

在组内以及在 \tl_build_begin:N … \tl_build_end:N 内，我们重新定义所有可能出现在编译后的正则表达式中的函数，然后运行正则表达式。然后将结果存储在 \l__regex_internal_a_tl 中，然后可以显示该结果。

```
\cs_new_protected:Npn \__regex_show:N #1
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \cs_set_protected:Npn \__regex_branch:n
        {
          \seq_pop_right:NN \l__regex_show_prefix_seq
            \l__regex_internal_a_tl
          \__regex_show_one:n { +-branch }
          \seq_put_right:No \l__regex_show_prefix_seq
            \l__regex_internal_a_tl
          \use:n
        }
      \cs_set_protected:Npn \__regex_group:nnnN
        { \__regex_show_group_aux:nnnnN { } }
      \cs_set_protected:Npn \__regex_group_no_capture:nnnN
```

```
    { \__regex_show_group_aux:nnnnN { ~(no~capture) } }
  \cs_set_protected:Npn \__regex_group_resetting:nnnN
    { \__regex_show_group_aux:nnnnN { ~(resetting) } }
  \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
  \cs_set_protected:Npn \__regex_command_K:
    { \__regex_show_one:n { reset~match~start~(\iow_char:N\\K) } }
  \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
    {
      \__regex_show_one:n
        { \bool_if:NF ##1 { negative~ } assertion:~##2 }
    }
  \cs_set:Npn \__regex_b_test: { word~boundary }
  \cs_set:Npn \__regex_Z_test: { anchor~at~end~(\iow_char:N\\Z) }
  \cs_set:Npn \__regex_A_test: { anchor~at~start~(\iow_char:N\\A) }
  \cs_set:Npn \__regex_G_test: { anchor~at~start~of~match~(\iow_char:N\\G) }
  \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
    { \__regex_show_one:n { char~code~\__regex_show_char:n{##1} } }
  \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
    {
      \__regex_show_one:n
        { range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}] }
    }
  \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
    { \__regex_show_one:n { char~code~\__regex_show_char:n{##1}~(caseless) } }
  \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
    {
      \__regex_show_one:n
        { Range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}]~(caseless) }
    }
  \cs_set_protected:Npn \__regex_item_catcode:nT
    { \__regex_show_item_catcode:NnT \c_true_bool }
  \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
    { \__regex_show_item_catcode:NnT \c_false_bool }
  \cs_set_protected:Npn \__regex_item_reverse:n
    { \__regex_show_scope:nn { Reversed~match } }
  \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
    { \__regex_show_one:n { char~\__regex_show_char:n{##2},~catcode~##1 } }
  \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
  \cs_set_protected:Npn \__regex_item_cs:n
    { \__regex_show_scope:nn { control~sequence } }
  \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any~token } }
  \seq_clear:N \l__regex_show_prefix_seq
```

```
      \__regex_show_push:n { ~ }
      \cs_if_exist_use:N #1
      \tl_build_end:N \l__regex_build_tl
      \exp_args:NNNo
    \group_end:
    \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
  }
```

(__regex_show:N 定义结束。)

__regex_show_char:n

显示单个字符，同时显示其 ASCII 表示（如果可用）。这可以扩展到 ASCII 之外的字符。对于括号本身而言，这并不理想。

```
\cs_new:Npn \__regex_show_char:n #1
  {
    \int_eval:n {#1}
    \int_compare:nT { 32 <= #1 <= 126 }
      { ~ ( \char_generate:nn {#1} {12} ) }
  }
```

(__regex_show_char:n 定义结束。)

__regex_show_one:n

最终消息的每个部分都经过这个函数，它向输出中添加一行，带有适当的前缀。

```
\cs_new_protected:Npn \__regex_show_one:n #1
  {
    \int_incr:N \l__regex_show_lines_int
    \tl_build_put_right:Ne \l__regex_build_tl
      {
        \exp_not:N \iow_newline:
        \seq_map_function:NN \l__regex_show_prefix_seq \use:n
        #1
      }
  }
```

(__regex_show_one:n 定义结束。)

__regex_show_push:n
__regex_show_pop:
__regex_show_scope:nn

进入和退出嵌套级别。scope 函数将其第一个参数打印为“引言”（“introduction”），然后在更深层次的嵌套中执行其第二个参数。

```
\cs_new_protected:Npn \__regex_show_push:n #1
  { \seq_put_right:Ne \l__regex_show_prefix_seq { #1 ~ } }
\cs_new_protected:Npn \__regex_show_pop:
  { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
\cs_new_protected:Npn \__regex_show_scope:nn #1#2
  {
```

```
    \__regex_show_one:n {#1}
    \__regex_show_push:n { ~ }
    #2
    \__regex_show_pop:
  }
```

(*__regex_show_push:n*, *__regex_show_pop:*, 和 *__regex_show_scope:nn* 定义结束。)

__regex_show_group_aux:nnnnN

我们以相同的方式显示所有组，只需添加一条消息 (no capture) 或 (resetting) 给特殊组。奇怪的 \use_ii:nn 避免为第一个分支打印不必要的 +-branch。

```
\cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
  {
    \__regex_show_one:n { ,-group~begin #1 }
    \__regex_show_push:n { | }
    \use_ii:nn #2
    \__regex_show_pop:
    \__regex_show_one:n
      { `-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 }
  }
```

(*__regex_show_group_aux:nnnnN* 定义结束。)

__regex_show_class:NnnnN

我对这个函数完全不满意：我找不到测试类是否是单一测试的方法。相反，收集类中测试的表示。如果它有多行，单独写下 Match 或 Don't match，并带有重复的信息（如果有的话）。然后，各种测试在自己的行上，最后一行。否则，我们需要再次评估测试的表示（因为前缀不正确）。这有点笨拙，但不太昂贵，因为它只有一个测试。

```
\cs_set:Npn \__regex_show_class:NnnnN #1#2#3#4#5
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_show_lines_int
      \__regex_show_push:n {~}
      #2
    \int_compare:nTF { \l__regex_show_lines_int = 0 }
      {
        \group_end:
        \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
      }
      {
        \bool_if:nTF
          { #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
          {
```

```
            \group_end:
            #2
            \tl_build_put_right:Nn \l__regex_build_tl
              { \__regex_msg_repeated:nnN {#3} {#4} #5 }
          }
          {
            \tl_build_end:N \l__regex_build_tl
            \exp_args:NNNo
            \group_end:
            \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
            \__regex_show_one:n
              {
                \bool_if:NTF #1 { Match } { Don't~match }
                \__regex_msg_repeated:nnN {#3} {#4} #5
              }
            \tl_build_put_right:Ne \l__regex_build_tl
              { \exp_not:o \l__regex_internal_a_tl }
          }
      }
  }
```

(*__regex_show_class:NnnnN* 定义结束。)

__regex_show_item_catcode:NnT

生成包含 catcode 位图 #2 的类别的序列，并显示它，缩进适用于此 catcode 约束的测试。

```
\cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
  {
    \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
    \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
      { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
    \__regex_show_scope:nn
      {
        categories~
        \seq_map_function:NN \l__regex_internal_seq \use:n
        , ~
        \bool_if:NF #1 { negative~ } class
      }
  }
```

(*__regex_show_item_catcode:NnT* 定义结束。)

__regex_show_item_exact_cs:n

```
\cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
```

```
{
  \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
  \seq_set_map_e:NNn \l__regex_internal_seq
    \l__regex_internal_seq { \iow_char:N\\##1 }
  \__regex_show_one:n
    { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
}
```

(*__regex_show_item_exact_cs:n* 定义结束。)

9.4 构建

9.4.1 构建过程中使用的变量

\l__regex_min_state_int
\l__regex_max_state_int

最后分配的状态是 \l__regex_max_state_int − 1，因此 \l__regex_max_state_int 始终指向一个空闲状态。变量 min_state 起初是 1，但在匹配代码中的嵌套调用中进行了移动，即在 \c{...} 构造中。

```
\int_new:N  \l__regex_min_state_int
\int_set:Nn \l__regex_min_state_int { 1 }
\int_new:N  \l__regex_max_state_int
```

(*\l__regex_min_state_int* 和 *\l__regex_max_state_int* 定义结束。)

\l__regex_left_state_int
\l__regex_right_state_int
\l__regex_left_state_seq
\l__regex_right_state_seq

通过从 left 状态分支到不同的选择，然后将其合并到 right 状态来实现替代。我们在两个序列中存储关于这些状态的信息。这些状态还用于实现组量词。通常，左指针和右指针只相差 1。

```
\int_new:N  \l__regex_left_state_int
\int_new:N  \l__regex_right_state_int
\seq_new:N  \l__regex_left_state_seq
\seq_new:N  \l__regex_right_state_seq
```

(*\l__regex_left_state_int* 以及其它的定义结束。)

\l__regex_capturing_group_int

\l__regex_capturing_group_int 是要分配给捕获组的下一个 ID 号码。这从 0 开始，对于包含完整正则表达式的组，组的计数是按照其左括号的顺序进行的，除非遇到 resetting 组。

```
\int_new:N  \l__regex_capturing_group_int
```

(*\l__regex_capturing_group_int* 定义结束。)

9.4.2 框架

该阶段涉及从编译后的正则表达式到 NFA 的转换。NFA 的每个状态都存储在一个 \toks 中。可以出现在 \toks 中的操作是

- __regex_action_start_wildcard:N ⟨*boolean*⟩ 插入在正则表达式开始处，其中 true ⟨*boolean*⟩ 使其非锚定。
- __regex_action_success: 标记 NFA 的退出状态。
- __regex_action_cost:n {⟨*shift*⟩} 是从当前 ⟨*state*⟩ 到 ⟨*state*⟩ + ⟨*shift*⟩ 的转换，它消耗当前字符：目标状态被保存，在下一位置匹配时将再次考虑它。
- __regex_action_free:n {⟨*shift*⟩} 和 __regex_action_free_group:n {⟨*shift*⟩} 是自由转换，它们立即执行 NFA 的 ⟨*state*⟩ + ⟨*shift*⟩ 状态的操作。它们在检测和避免无限循环的方式上有所不同。目前，我们只需要知道 group 变体必须用于返回到组的开始的转换。
- __regex_action_submatch:nN {⟨*group*⟩} ⟨*key*⟩，其中 ⟨*key*⟩ 是 < 或 >，表示组 ⟨*group*⟩ 的开始或结束。这会将查询的当前位置存储为 ⟨*key*⟩ 子匹配边界。
- 在条件中的其中一种动作。

我们在构建过程中努力保持以下属性。

- 当前捕获组是 capturing_group − 1，如果现在打开一个组，它将被标记为 capturing_group。
- 最后分配的状态是 max_state − 1，因此 max_state 是一个空闲状态。
- left_state 指向当前组或最后一个类的左侧状态。
- right_state 指向一个新创建的、空的状态，其中一些转换导向它。
- left/right 序列保存嵌套组的相应端点的列表。

__regex_build:n
__regex_build_aux:Nn
__regex_build:N
__regex_build_aux:NN

n-type 的函数首先编译其参数。重置一些变量。分配两个状态，并在状态 0 中放置通配符（到状态 1 和 0 状态的转换）。然后在编号为 0（当前 capturing_group 的值）的（捕获）组内构建正则表达式。最后，如果匹配到最后的状态，它就成功了。辅助函数的参数 #1 中的 false 布尔值将禁止通配符，并使匹配锚定：用于 \peek_regex:nTF 等。

```
\cs_new_protected:Npn \__regex_build:n
  { \__regex_build_aux:Nn \c_true_bool }
\cs_new_protected:Npn \__regex_build:N
```

```
  { \__regex_build_aux:NN \c_true_bool }
\cs_new_protected:Npn \__regex_build_aux:Nn #1#2
  {
    \__regex_compile:n {#2}
    \__regex_build_aux:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \__regex_build_aux:NN #1#2
  {
    \__regex_standard_escapechar:
    \int_zero:N \l__regex_capturing_group_int
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_toks_put_right:Nn \l__regex_left_state_int
      { \__regex_action_start_wildcard:N #1 }
    \__regex_group:nnnN {#2} { 1 } { 0 } \c_false_bool
    \__regex_toks_put_right:Nn \l__regex_right_state_int
      { \__regex_action_success: }
  }
```

(*__regex_build:n 以及其它的定义结束。*)

\g__regex_case_int

在 \regex_match_case:nn 和相关函数中成功匹配的案例编号。

```
\int_new:N \g__regex_case_int
```

(*\g__regex_case_int 定义结束。*)

\l__regex_case_max_group_int

在 \regex_match_case:nn 和相关函数的参数中，⟨*regex*⟩ 中任何一个正则表达式中出现的最大组号。

```
\int_new:N \l__regex_case_max_group_int
```

(*\l__regex_case_max_group_int 定义结束。*)

__regex_case_build:n
__regex_case_build:e
__regex_case_build_aux:Nn
__regex_case_build_loop:n

参见 __regex_build:n，但带有循环。

```
\cs_new_protected:Npn \__regex_case_build:n #1
  {
    \__regex_case_build_aux:Nn \c_true_bool {#1}
    \int_gzero:N \g__regex_case_int
  }
\cs_generate_variant:Nn \__regex_case_build:n { e }
\cs_new_protected:Npn \__regex_case_build_aux:Nn #1#2
  {
    \__regex_standard_escapechar:
```

```
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_toks_put_right:Nn \l__regex_left_state_int
      { \__regex_action_start_wildcard:N #1 }
    %
    \__regex_build_new_state:
    \__regex_toks_put_left:Ne \l__regex_left_state_int
      { \__regex_action_submatch:nN { 0 } < }
    \__regex_push_lr_states:
    \int_zero:N \l__regex_case_max_group_int
    \int_gzero:N \g__regex_case_int
    \tl_map_inline:nn {#2}
      {
        \int_gincr:N \g__regex_case_int
        \__regex_case_build_loop:n {##1}
      }
    \int_set_eq:NN \l__regex_capturing_group_int \l__regex_case_max_group_int
    \__regex_pop_lr_states:
  }
\cs_new_protected:Npn \__regex_case_build_loop:n #1
  {
    \int_set:Nn \l__regex_capturing_group_int { 1 }
    \__regex_compile_use:n {#1}
    \int_set:Nn \l__regex_case_max_group_int
      {
        \int_max:nn { \l__regex_case_max_group_int }
          { \l__regex_capturing_group_int }
      }
    \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
    \__regex_toks_put_left:Ne \l__regex_right_state_int
      {
        \__regex_action_submatch:nN { 0 } >
        \int_gset:Nn \g__regex_case_int
          { \int_use:N \g__regex_case_int }
        \__regex_action_success:
      }
    \__regex_toks_clear:N \l__regex_max_state_int
    \seq_push:No \l__regex_right_state_seq
      { \int_use:N \l__regex_max_state_int }
    \int_incr:N \l__regex_max_state_int
```

```
1945   }
```

(*__regex_case_build:n*, *__regex_case_build_aux:Nn*, 和 *__regex_case_build_loop:n* 定义结束。)

__regex_build_for_cs:n

在匹配代码中，依赖于一些全局的 intarray 变量，但仅使用它们的一部分条目范围。具体来说，

- \g__regex_state_active_intarray 从 \l__regex_min_state_int 到 \l__regex_max_stat 1；

在这个对匹配代码的嵌套调用中，我们需要这个范围的新版本涉及完全新的 intarray 变量的条目，因此我们首先通过将（新的）\l__regex_min_state_int 设置为（旧的）\l__regex_max_state_int 来使用较高的条目。

当使用正则表达式匹配一个控制序列（cs）时，我们不插入通配符，我们在结尾处锚定，由于我们忽略子匹配，因此不需要用组括起表达式。然而，为了使分支在外层正常工作，我们需要在它们的序列中放入相应的 left 和 right 状态。

```
1946 \cs_new_protected:Npn \__regex_build_for_cs:n #1
1947   {
1948     \int_set_eq:NN \l__regex_min_state_int \l__regex_max_state_int
1949     \__regex_build_new_state:
1950     \__regex_build_new_state:
1951     \__regex_push_lr_states:
1952     #1
1953     \__regex_pop_lr_states:
1954     \__regex_toks_put_right:Nn \l__regex_right_state_int
1955       {
1956         \if_int_compare:w -2 = \l__regex_curr_char_int
1957           \exp_after:wN \__regex_action_success:
1958         \fi:
1959       }
1960   }
```

(*__regex_build_for_cs:n* 定义结束。)

9.4.3 构建 nfa 的辅助函数

__regex_push_lr_states:
__regex_pop_lr_states:

在构建正则表达式时，我们跟踪每个组的左端和右端的指针，而无需使用 TEX 的分组。

```
1961 \cs_new_protected:Npn \__regex_push_lr_states:
1962   {
1963     \seq_push:No \l__regex_left_state_seq
1964       { \int_use:N \l__regex_left_state_int }
```

```
    \seq_push:No \l__regex_right_state_seq
      { \int_use:N \l__regex_right_state_int }
  }
\cs_new_protected:Npn \__regex_pop_lr_states:
  {
    \seq_pop:NN \l__regex_left_state_seq  \l__regex_internal_a_tl
    \int_set:Nn \l__regex_left_state_int  \l__regex_internal_a_tl
    \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
  }
```

(*__regex_push_lr_states: 和 __regex_pop_lr_states: 定义结束。*)

__regex_build_transition_left:NNN
__regex_build_transition_right:nNn

使用函数 #1 从 #2 到 #3 添加一个转换。left 函数用于更高优先级的转换，而 right 函数用于更低优先级的转换（应稍后执行）。签名有所不同，以反映稍后的不同用法。两个函数都可以进行优化。

```
\cs_new_protected:Npn \__regex_build_transition_left:NNN #1#2#3
  { \__regex_toks_put_left:Ne  #2 { #1 { \int_eval:n { #3 - #2 } } } }
\cs_new_protected:Npn \__regex_build_transition_right:nNn #1#2#3
  { \__regex_toks_put_right:Ne #2 { #1 { \int_eval:n { #3 - #2 } } } }
```

(*__regex_build_transition_left:NNN 和 __regex_build_transition_right:nNn 定义结束。*)

__regex_build_new_state:

在 NFA 中添加一个新的空状态。然后更新 left、right 和 max 状态，以使 right 状态成为新的空状态，而 left 状态指向先前的 “current” 状态。

```
\cs_new_protected:Npn \__regex_build_new_state:
  {
    \__regex_toks_clear:N \l__regex_max_state_int
    \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
    \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
    \int_incr:N \l__regex_max_state_int
  }
```

(*__regex_build_new_state: 定义结束。*)

__regex_build_transitions_lazyness:NNNNN

该函数创建一个新状态，并在旧当前状态开始的地方放置两个转换。转换的顺序由 #1 控制，对于惰性量词为 true，对于贪婪量词为 false。

```
\cs_new_protected:Npn \__regex_build_transitions_lazyness:NNNNN #1#2#3#4#5
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
        \if_meaning:w \c_true_bool #1
```

```
          #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
          #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
        \else:
          #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
          #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
        \fi:
      }
  }
```

(*__regex_build_transitions_lazyness:NNNNN 定义结束。*)

9.4.4 构建类

__regex_class:NnnnN
__regex_tests_action_cost:n

参数是：⟨*boolean*⟩ {⟨*tests*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*lazyness*⟩。首先，在正类的 true 分支或负类的 false 分支中存储带有尾随 `\__regex_action_cost:n` 的测试。整数 ⟨*more*⟩ 对于固定重复次数是 0，对于无界重复是 −1，对于重复范围是 ⟨*max*⟩ − ⟨*min*⟩。

```
\cs_new_protected:Npn \__regex_class:NnnnN #1#2#3#4#5
  {
    \cs_set:Npe \__regex_tests_action_cost:n ##1
      {
        \exp_not:n { \exp_not:n {#2} }
        \bool_if:NTF #1
          { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
          { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }
      }
    \if_case:w - #4 \exp_stop_f:
          \__regex_class_repeat:n   {#3}
    \or:   \__regex_class_repeat:nN  {#3}      #5
    \else: \__regex_class_repeat:nnN {#3} {#4} #5
    \fi:
  }
\cs_new:Npn \__regex_tests_action_cost:n { \__regex_action_cost:n }
```

(*__regex_class:NnnnN 和 __regex_tests_action_cost:n 定义结束。*)

__regex_class_repeat:n

用于固定数量的重复。为每次重复构建一个状态，带有由我们收集的测试控制的转换。对于 `#1` = 0 重复，这完全没问题：什么都不会构建。

```
\cs_new_protected:Npn \__regex_class_repeat:n #1
  {
    \prg_replicate:nn {#1}
      {
        \__regex_build_new_state:
        \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
```

```
      \l__regex_left_state_int \l__regex_right_state_int
    }
  }
```

(*__regex_class_repeat:n 定义结束。*)

__regex_class_repeat:nN

这实现了单一类的无界重复（如 * 和 + 量词）。如果最小重复次数 #1 为 0，那么从当前状态到自身构建一个由测试控制的转换，并自由过渡到一个新状态（因此跳过测试）。否则，调用 __regex_class_repeat:n 以匹配 #1 次的代码，并添加自由过渡到前一个状态和到一个新状态。在两种情况下，转换的顺序由懒惰布尔 #2 控制。

```
\cs_new_protected:Npn \__regex_class_repeat:nN #1#2
  {
    \if_int_compare:w #1 = \c_zero_int
      \__regex_build_transitions_lazyness:NNNNN #2
        \__regex_action_free:n       \l__regex_right_state_int
        \__regex_tests_action_cost:n \l__regex_left_state_int
    \else:
      \__regex_class_repeat:n {#1}
      \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
      \__regex_build_transitions_lazyness:NNNNN #2
        \__regex_action_free:n \l__regex_right_state_int
        \__regex_action_free:n \l__regex_internal_a_int
    \fi:
  }
```

(*__regex_class_repeat:nN 定义结束。*)

__regex_class_repeat:nnN

我们要构建的代码是匹配从 #1 到 #1 + #2 次的重复。匹配 #1 次（可以为 0）。将下一次构造的最终状态计算为 a。构建 #2 > 0 个状态，每个状态都有一个由测试控制的转换到下一个状态，以及一个到最终状态 a 的转换。计算 a 的过程是安全的，因为状态是按顺序分配的，从 max_state 开始。

```
\cs_new_protected:Npn \__regex_class_repeat:nnN #1#2#3
  {
    \__regex_class_repeat:n {#1}
    \int_set:Nn \l__regex_internal_a_int
      { \l__regex_max_state_int + #2 - 1 }
    \prg_replicate:nn { #2 }
      {
        \__regex_build_transitions_lazyness:NNNNN #3
          \__regex_action_free:n       \l__regex_internal_a_int
          \__regex_tests_action_cost:n \l__regex_right_state_int
      }
  }
```

(*__regex_class_repeat:nnN* 定义结束。)

9.4.5 构建分组

__regex_group_aux:nnnnN

参数是：{⟨*label*⟩} {⟨*contents*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*lazyness*⟩。如果 ⟨*min*⟩ 为 0，我们需要在构建组之前添加一个状态，以便跳过组的线程不会同时设置子匹配的起点。在添加了一个状态之后，`left_state` 是组的左端，所有分支都起源于这里，`right_state` 是组的右端，所有分支都在这里结束。我们将这两个整数存储起来，以便为每个分支查询，构建组的内容 `#2` 的 NFA 状态，然后忘记这两个整数。完成这个步骤后，执行重复：精确地 `#3` 次，或者 `#3` 或更多次，或者在 `#3` 和 `#3` + `#4` 次之间，带有懒惰性 `#5`。子匹配跟踪使用 ⟨*label*⟩ `#1`。这三个辅助程序中的每一个都期望 `left_state` 和 `right_state` 被适当设置。

```
\cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
  {
    \if_int_compare:w #3 = \c_zero_int
      \__regex_build_new_state:
      \__regex_build_transition_right:nNn \__regex_action_free_group:n
        \l__regex_left_state_int \l__regex_right_state_int
    \fi:
    \__regex_build_new_state:
    \__regex_push_lr_states:
    #2
    \__regex_pop_lr_states:
    \if_case:w - #4 \exp_stop_f:
          \__regex_group_repeat:nn   {#1} {#3}
    \or:  \__regex_group_repeat:nnN  {#1} {#3}      #5
    \else: \__regex_group_repeat:nnnN {#1} {#3} {#4} #5
    \fi:
  }
```

(*__regex_group_aux:nnnnN* 定义结束。)

__regex_group:nnnN
__regex_group_no_capture:nnnN

将该组的标签（展开后）和该组本身一起传递给 `\__regex_group_aux:nnnnnN`，附带一些额外的命令执行。

```
\cs_new_protected:Npn \__regex_group:nnnN #1
  {
    \exp_args:No \__regex_group_aux:nnnnN
      { \int_use:N \l__regex_capturing_group_int }
      {
        \int_incr:N \l__regex_capturing_group_int
        #1
```

```
      }
  }
\cs_new_protected:Npn \__regex_group_no_capture:nnnN
  { \__regex_group_aux:nnnnN { -1 } }
```

(*__regex_group:nnnN* 和 *__regex_group_no_capture:nnnN* 定义结束。)

__regex_group_resetting:nnnN
__regex_group_resetting_loop:nnNn

再次将标签 -1 交给 `\__regex_group_aux:nnnnN`，但这次我们要更努力地跟踪任何分支末尾的最大组标签，并在每个分支处重置组号。这依赖于编译后的正则表达式始终是形式为 `\__regex_branch:n {`⟨*branch*⟩`}` 的项目序列的事实。

```
\cs_new_protected:Npn \__regex_group_resetting:nnnN #1
  {
    \__regex_group_aux:nnnnN { -1 }
      {
        \exp_args:Noo \__regex_group_resetting_loop:nnNn
          { \int_use:N \l__regex_capturing_group_int }
          { \int_use:N \l__regex_capturing_group_int }
          #1
          { ?? \prg_break:n } { }
        \prg_break_point:
      }
  }
\cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
  {
    \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
    \int_set:Nn \l__regex_capturing_group_int {#2}
    #3 {#4}
    \exp_args:Nf \__regex_group_resetting_loop:nnNn
      { \int_max:nn {#1} { \l__regex_capturing_group_int } }
      {#2}
  }
```

(*__regex_group_resetting:nnnN* 和 *__regex_group_resetting_loop:nnNn* 定义结束。)

__regex_branch:n

向当前组的左状态添加一个从新状态到右状态的自由转换，作为这个分支的起点。一旦分支构建完成，添加一个从其最后状态到组的右状态的转换。组的左右状态从相关的序列中提取。

```
\cs_new_protected:Npn \__regex_branch:n #1
  {
    \__regex_build_new_state:
    \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
```

```
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_left_state_int \l__regex_right_state_int
    #1
    \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_internal_a_tl
  }
```

(*__regex_branch:n* 定义结束。)

__regex_group_repeat:nn

调用此函数以重复一个固定次数的组 #2；如果这是 0，我们完全移除该组（但不重置 capturing_group 标签）。否则，辅助命令 __regex_group_repeat_aux:n 复制 #2 次组的 \toks，并将 internal_a 指向最后重复的左端。我们只在最后一次重复时记录子匹配信息。最后，在末尾添加一个状态（复制辅助已经处理了它的转换）。

```
\cs_new_protected:Npn \__regex_group_repeat:nn #1#2
  {
    \if_int_compare:w #2 = \c_zero_int
      \int_set:Nn \l__regex_max_state_int
        { \l__regex_left_state_int - 1 }
      \__regex_build_new_state:
    \else:
      \__regex_group_repeat_aux:n {#2}
      \__regex_group_submatches:nNN {#1}
        \l__regex_internal_a_int \l__regex_right_state_int
      \__regex_build_new_state:
    \fi:
  }
```

(*__regex_group_repeat:nn* 定义结束。)

__regex_group_submatches:nNN

将组 #1 的子匹配跟踪代码插入到状态 #2 和 #3 中，除非由标签 -1 抑制。

```
\cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
  {
    \if_int_compare:w #1 > - \c_one_int
      \__regex_toks_put_left:Ne #2 { \__regex_action_submatch:nN {#1} < }
      \__regex_toks_put_left:Ne #3 { \__regex_action_submatch:nN {#1} > }
    \fi:
  }
```

(*__regex_group_submatches:nNN* 定义结束。)

__regex_group_repeat_aux:n

在 left_state 到 max_state 范围内重复 \toks，#1 > 0 次。首先添加一个转换，以便复制“链”（“chain”）正确。计算原始复制和我们想要的最后复制之间的偏移 c。

将 right_state 和 max_state 移到它们的最终值。然后，我们想执行 c 次复制操作。最后，b 等于 max_state，a 指向组的最后副本的左侧。

```
\cs_new_protected:Npn \__regex_group_repeat_aux:n #1
  {
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_max_state_int
    \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
    \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
    \if_int_compare:w \int_eval:n {#1} > \c_one_int
      \int_set:Nn \l__regex_internal_c_int
        {
          ( #1 - 1 )
          * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
        }
      \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
      \int_add:Nn \l__regex_max_state_int   { \l__regex_internal_c_int }
      \__regex_toks_memcpy:NNn
        \l__regex_internal_b_int
        \l__regex_internal_a_int
        \l__regex_internal_c_int
    \fi:
  }
```

(*__regex_group_repeat_aux:n* 定义结束。)

__regex_group_repeat:nnN

此函数用于至少重复一个组 n 次；当 $n = 0$ 时，情况与 $n > 0$ 很不同。首先假设 $n = 0$。在组的开头和结尾插入子匹配跟踪信息，在右端到“真实”左状态 a 添加一个自由转换（记住：在这种情况下，我们在左状态之前添加了一个额外的状态）。这形成了循环，通过从 a 添加一个自由转换到一个新状态来中断循环。

现在考虑 $n > 0$ 的情况。重复组 n 次，通过自由转换链接各个副本。仅对最后一个副本添加子匹配跟踪，然后添加一个自由转换，从右端回到最后一个副本的左端，要么在移动到 NFA 的其余部分之前，要么在之后。这个转换最终可能会在子匹配跟踪之前结束，但这不重要，因为只有在再次经过组时才会这样做，记录新的匹配。最后，添加一个状态；我们已经有一条从 __regex_group_repeat_aux:n 指向它的转换。

```
\cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
  {
    \if_int_compare:w #2 = \c_zero_int
      \__regex_group_submatches:nNN {#1}
        \l__regex_left_state_int \l__regex_right_state_int
      \int_set:Nn \l__regex_internal_a_int
```

```
        { \l__regex_left_state_int - 1 }
      \__regex_build_transition_right:nNn \__regex_action_free:n
        \l__regex_right_state_int \l__regex_internal_a_int
      \__regex_build_new_state:
      \if_meaning:w \c_true_bool #3
        \__regex_build_transition_left:NNN \__regex_action_free:n
          \l__regex_internal_a_int \l__regex_right_state_int
      \else:
        \__regex_build_transition_right:nNn \__regex_action_free:n
          \l__regex_internal_a_int \l__regex_right_state_int
      \fi:
    \else:
      \__regex_group_repeat_aux:n {#2}
      \__regex_group_submatches:nNN {#1}
        \l__regex_internal_a_int \l__regex_right_state_int
      \if_meaning:w \c_true_bool #3
        \__regex_build_transition_right:nNn \__regex_action_free_group:n
          \l__regex_right_state_int \l__regex_internal_a_int
      \else:
        \__regex_build_transition_left:NNN \__regex_action_free_group:n
          \l__regex_right_state_int \l__regex_internal_a_int
      \fi:
      \__regex_build_new_state:
    \fi:
  }
```

(*__regex_group_repeat:nnN 定义结束。*)

__regex_group_repeat:nnnN

我们希望重复组在 #2 和 #2 + #3 次之间，由 #4 控制懒惰性。我们在前面插入子匹配跟踪：原则上，我们可以避免记录前 #2 个副本的子匹配，但这迫使我们特别处理 #2 = 0 的情况。用子匹配跟踪重复该组 #2 + #3 次（最大重复次数）。然后我们的目标是从第 #2 个组的末尾和每个随后的组添加 #3 个转换到末尾。对于懒惰量词，我们将这些转换添加到左状态之前，在子匹配跟踪之前。对于贪婪情况，我们在子匹配跟踪和转向更多重复的转换之后，将这些转换添加到右状态。在贪婪情况下，当 #2 = 0 时，跳过所有副本的转换必须单独添加，因为它的起始状态不遵循正常模式：我们不得不在之前“手动”（“by hand”）添加它。

```
\cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
  {
    \__regex_group_submatches:nNN {#1}
      \l__regex_left_state_int \l__regex_right_state_int
    \__regex_group_repeat_aux:n { #2 + #3 }
    \if_meaning:w \c_true_bool #4
```

```
      \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
      \prg_replicate:nn { #3 }
        {
          \int_sub:Nn \l__regex_left_state_int
            { \l__regex_internal_b_int - \l__regex_internal_a_int }
          \__regex_build_transition_left:NNN \__regex_action_free:n
            \l__regex_left_state_int \l__regex_max_state_int
        }
    \else:
      \prg_replicate:nn { #3 - 1 }
        {
          \int_sub:Nn \l__regex_right_state_int
            { \l__regex_internal_b_int - \l__regex_internal_a_int }
          \__regex_build_transition_right:nNn \__regex_action_free:n
            \l__regex_right_state_int \l__regex_max_state_int
        }
      \if_int_compare:w #2 = \c_zero_int
        \int_set:Nn \l__regex_right_state_int
          { \l__regex_left_state_int - 1 }
      \else:
        \int_sub:Nn \l__regex_right_state_int
          { \l__regex_internal_b_int - \l__regex_internal_a_int }
      \fi:
      \__regex_build_transition_right:nNn \__regex_action_free:n
        \l__regex_right_state_int \l__regex_max_state_int
    \fi:
    \__regex_build_new_state:
  }
```

(*__regex_group_repeat:nnnN 定义结束。*)

9.4.6 其他

__regex_assertion:Nn
__regex_b_test:
__regex_A_test:
__regex_G_test:
__regex_Z_test:

用法：`\__regex_assertion:Nn` ⟨*boolean*⟩ {⟨*test*⟩}，其中 ⟨*test*⟩ 是其他两个函数之一。根据断言测试条件向新状态添加自由转换。`\__regex_b_test:` 测试由 `\b` 和 `\B` 转义使用：检查最后一个字符是否是单词字符，然后检查当前字符。对于此目的，字符串的边界标记是非单词字符。

```
\cs_new_protected:Npn \__regex_assertion:Nn #1#2
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
```

```
        \exp_not:n {#2}
        \__regex_break_point:TF
          \bool_if:NF #1 { { } }
          {
            \__regex_action_free:n
              {
                \int_eval:n
                  { \l__regex_right_state_int - \l__regex_left_state_int }
              }
          }
          \bool_if:NT #1 { { } }
      }
  }
\cs_new_protected:Npn \__regex_b_test:
  {
    \group_begin:
      \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
      \__regex_prop_w:
      \__regex_break_point:TF
        { \group_end: \__regex_item_reverse:n { \__regex_prop_w: } }
        { \group_end: \__regex_prop_w: }
  }
\cs_new_protected:Npn \__regex_Z_test:
  {
    \if_int_compare:w -2 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_A_test:
  {
    \if_int_compare:w -2 = \l__regex_last_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_G_test:
  {
    \if_int_compare:w \l__regex_curr_pos_int = \l__regex_start_pos_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
```

(*__regex_assertion:Nn* 以及其它的定义结束。)

__regex_command_K: 修改第 0 个子匹配（完全匹配）的起始点，并过渡到一个新状态，假装这是一个新线程。

```
\cs_new_protected:Npn \__regex_command_K:
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
        \__regex_action_submatch:nN { 0 } <
        \bool_set_true:N \l__regex_fresh_thread_bool
        \__regex_action_free:n
          {
            \int_eval:n
              { \l__regex_right_state_int - \l__regex_left_state_int }
          }
        \bool_set_false:N \l__regex_fresh_thread_bool
      }
  }
```

(*__regex_command_K: 定义结束。*)

9.5 匹配

我们通过并行运行所有执行线程在 NFA 中搜索匹配，每步读取查询的一个记号。NFA 包含到其他状态的“free”转换和“consume”当前记号的转换。对于自由转换，NFA 的新状态上的指令立即执行。当一个转换消耗一个字符时，新状态被附加到`\g__regex_thread_info_intarray`中的“active states”列表（连同子匹配信息）：当下一个记号从查询中读取时，此线程再次变为活动状态。在每一步（对于查询中的每个记号），我们展开该活动状态列表和相应的子匹配属性，并清空它们。

如果 NFA 中的两个路径在读取给定记号后到达相同状态，则它们只在它们先前匹配的方式上有所不同，任何未来的执行对于两者都是相同的（注意，在存在反向引用时，这将是错误的）。因此，我们只需要保留两个线程中的一个：具有最高优先级的线程。我们的 NFA 是以这样一种方式构建的，以便较高优先级的动作总是在较低优先级的动作之前执行，这使得事情能够正常工作。

上一段的解释可能使我们认为我们只需要追踪在给定步骤中访问了哪些状态：毕竟，匹配 `(a?)*` 对 `a` 的循环生成已经被打破了，不是吗？不是的。该组首先匹配 `a`，正如它应该的那样，然后重复；它尝试再次匹配 `a`，但失败；它跳过 `a`，发现在查询的这个位置已经看到了此状态：匹配停止。捕获组是（错误的）`a`。出了什么问题是，一个线程与自身发生了碰撞，后来的版本，通过一个空匹配多次经过组，应该比不经过组的版本优先级更高。

我们通过区分“normal”自由转换__regex_action_free:n和转换__regex_action_free_group:n来解决这个问题，后者返回到组的开始。前者保留线程，除非它们被“completed”线程访问，后者类型的转换还阻止返回到当前线程访问的状态。

9.5.1 匹配时使用的变量

\l__regex_min_pos_int
\l__regex_max_pos_int
\l__regex_curr_pos_int
\l__regex_start_pos_int
\l__regex_success_pos_int

查询中的记号从第一个 min_pos 到最后一个 max_pos − 1 进行索引，并且它们的信息存储在几个带有这些数字的数组和\toks寄存器中。我们在没有回溯的情况下进行匹配，在查询的 curr_pos 位置保持所有线程同步。当前匹配尝试的起始点是 start_pos，并且在线程成功时，success_pos 用作下一个起始位置。

```
\int_new:N \l__regex_min_pos_int
\int_new:N \l__regex_max_pos_int
\int_new:N \l__regex_curr_pos_int
\int_new:N \l__regex_start_pos_int
\int_new:N \l__regex_success_pos_int
```

(*\l__regex_min_pos_int 以及其它的定义结束。*)

\l__regex_curr_char_int
\l__regex_curr_catcode_int
\l__regex_curr_token_tl
\l__regex_last_char_int
\l__regex_last_char_success_int
\l__regex_case_changed_char_int

当前位置的记号的字符和类别码以及扩展到该记号的记号列表；上一个位置记号的字符码；成功匹配之前记号的字符码；更改当前记号大小写（A-Z↔a-z）的结果的字符码。该整数仅在必要时计算，否则为\c_max_int。curr_char 变量在各个阶段也用于保存字符码。

```
\int_new:N \l__regex_curr_char_int
\int_new:N \l__regex_curr_catcode_int
\tl_new:N \l__regex_curr_token_tl
\int_new:N \l__regex_last_char_int
\int_new:N \l__regex_last_char_success_int
\int_new:N \l__regex_case_changed_char_int
```

(*\l__regex_curr_char_int 以及其它的定义结束。*)

\l__regex_curr_state_int

对于记号列表中的每个字符，依次考虑每个活动状态。变量\l__regex_curr_state_int保存当前考虑的 NFA 状态：转换随后以相对于当前状态的偏移给出。

```
\int_new:N \l__regex_curr_state_int
```

(*\l__regex_curr_state_int 定义结束。*)

\l__regex_curr_submatches_tl
\l__regex_success_submatches_tl

当前活动的线程的子匹配存储在 curr_submatches 列表中，它几乎是一个逗号列表，但以逗号结尾。这个列表由__regex_store_state:n存储到 intarray 变量中，在下一个位置匹配时将其检索出来。当一个线程成功时，此列表被复制到\l__regex_success_submatches_tl：只有最后成功的线程保留在那里。

```
\tl_new:N \l__regex_curr_submatches_tl
\tl_new:N \l__regex_success_submatches_tl
```

(*\l__regex_curr_submatches_tl* 和 *\l__regex_success_submatches_tl* 定义结束。)

\l__regex_step_int

这个整数总是偶数，每次读取查询中的一个字符时增加，且在进行多次匹配时不重置。我们在\g__regex_state_active_intarray中存储了 NFA 中每个⟨*state*⟩最后一次出现的⟨*step*⟩。这使我们能够通过在同一步骤中不访问相同的状态两次来打破无限循环。实际上，我们存储的⟨*step*⟩等于\toks⟨*state*⟩的操作已经开始执行的步骤，但尚未完成。但是，一旦我们完成，我们在\g__regex_state_active_intarray中存储 step + 1。这是为了正确跟踪子匹配信息（见构建阶段）。step 还用于将每组子匹配信息附加到给定迭代（并在它对应于过去的步骤时自动丢弃）。

```
\int_new:N \l__regex_step_int
```

(*\l__regex_step_int* 定义结束。)

\l__regex_min_thread_int
\l__regex_max_thread_int

所有当前活动的线程按照优先级的顺序保留在\g__regex_thread_info_intarray中，与相应的子匹配信息一起。这个 intarray 中的数据被组织为从 min_thread（包括）到 max_thread（不包括）的块。在每个步骤的开始，整个数组都被解包，以便空间可以立即被重用，并将 max_thread 重置为 min_thread，有效地清除数组。

```
\int_new:N \l__regex_min_thread_int
\int_new:N \l__regex_max_thread_int
```

(*\l__regex_min_thread_int* 和 *\l__regex_max_thread_int* 定义结束。)

\g__regex_state_active_intarray
\g__regex_thread_info_intarray

\g__regex_state_active_intarray存储每个⟨*state*⟩上一次活跃的⟨*step*⟩。\g__regex_thread_info_intarray存储要在下一步考虑的线程，更准确地说是这些线程所在的状态。

```
\intarray_new:Nn \g__regex_state_active_intarray { 65536 }
\intarray_new:Nn \g__regex_thread_info_intarray { 65536 }
```

(*\g__regex_state_active_intarray* 和 *\g__regex_thread_info_intarray* 定义结束。)

\l__regex_matched_analysis_tl
\l__regex_curr_analysis_tl

\l__regex_curr_analysis_tl列表由一个括号组成，其中包含三个与当前记号相对应的括号组，其语法与\tl_analysis_map_inline:nn相同。\l__regex_matched_analysis_tl（在 tl_build 机制下构建）为给定匹配尝试中到目前为止已处理的每个记号都有一个项：每个项都包含三个与\tl_analysis_map_inline:nn相同语法的括号组。

```
\tl_new:N \l__regex_matched_analysis_tl
\tl_new:N \l__regex_curr_analysis_tl
```

(*\l__regex_matched_analysis_tl* 和 *\l__regex_curr_analysis_tl* 定义结束。)

\l__regex_every_match_tl

每次找到匹配时都使用这个记号列表。对于单一匹配，记号列表为空。对于多次匹配，记号列表设置为重复匹配，在执行依赖于用户函数的某些操作后。参见__regex_single_match:和__regex_multi_match:n。

```
\tl_new:N \l__regex_every_match_tl
```

(*\l__regex_every_match_tl 定义结束。*)

\l__regex_fresh_thread_bool
\l__regex_empty_success_bool
__regex_if_two_empty_matches:F

在进行多次匹配时，我们需要避免无限循环，其中每次迭代都匹配相同的空记号列表。当匹配空记号列表时，抑制同一空记号列表的下一个成功匹配。我们通过将\l__regex_fresh_thread_bool设置为 true 来检测空匹配的方式：对于直接来自正则表达式开头或\K命令的线程，每当线程成功时测试该布尔值。函数__regex_if_two_empty_matches:F在每个匹配尝试时重新定义，具体取决于先前的匹配是否为空：如果是，则该函数必须在匹配为空并且与先前匹配的位置相同时取消所谓的成功；否则，我们绝对没有两个相同的空匹配，所以该函数是\use:n。

```
\bool_new:N \l__regex_fresh_thread_bool
\bool_new:N \l__regex_empty_success_bool
\cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(*\l__regex_fresh_thread_bool，\l__regex_empty_success_bool，和 __regex_if_two_empty_matches:F 定义结束。*)

\g__regex_success_bool
\l__regex_saved_success_bool
\l__regex_match_success_bool

如果当前匹配尝试成功，则布尔值\l__regex_match_success_bool为 true，如果至少有一次成功匹配，则\g__regex_success_bool为 true。这是整个模块中唯一的全局变量，但是当使用\c{...}匹配控制序列时，我们需要将其局部化。这通过将全局变量保存到\l__regex_saved_success_bool中完成，它是局部的，因此不受由于内部正则表达式函数引起的更改的影响。

```
\bool_new:N \g__regex_success_bool
\bool_new:N \l__regex_saved_success_bool
\bool_new:N \l__regex_match_success_bool
```

(*\g__regex_success_bool，\l__regex_saved_success_bool，和 \l__regex_match_success_bool 定义结束。*)

9.5.2 匹配：框架

__regex_match:n
__regex_match_cs:n
__regex_match_init:

初始化应该为每个用户函数设置一次的变量（即使对于多次匹配）。即总体匹配尚未成功；不应标记任何状态为已访问（\g__regex_state_active_intarray）；我们从步骤 0 开始；我们假装在查询的开头有一个先前的匹配，该匹配不是空的（以避免在开头扼杀一个空匹配）。一旦所有这些设置完成，我们准备好启动。找到第一个匹配。

```
\cs_new_protected:Npn \__regex_match:n #1
  {
    \__regex_match_init:
```

```
    \__regex_match_once_init:
    \tl_analysis_map_inline:nn {#1}
      { \__regex_match_one_token:nnN {##1} {##2} ##3 }
    \__regex_match_one_token:nnN { } { -2 } F
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_cs:n #1
  {
    \int_set_eq:NN \l__regex_min_thread_int \l__regex_max_thread_int
    \__regex_match_init:
    \__regex_match_once_init:
    \str_map_inline:nn {#1}
      {
        \tl_if_blank:nTF {##1}
          { \__regex_match_one_token:nnN {##1} {`##1} A }
          { \__regex_match_one_token:nnN {##1} {`##1} C }
      }
    \__regex_match_one_token:nnN { } { -2 } F
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_init:
  {
    \bool_gset_false:N \g__regex_success_bool
    \int_step_inline:nnn
      \l__regex_min_state_int { \l__regex_max_state_int - 1 }
      {
        \__kernel_intarray_gset:Nnn
          \g__regex_state_active_intarray {##1} { 1 }
      }
    \int_zero:N \l__regex_step_int
    \int_set:Nn \l__regex_min_pos_int { 2 }
    \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
    \int_set:Nn \l__regex_last_char_success_int { -2 }
    \tl_build_begin:N \l__regex_matched_analysis_tl
    \tl_clear:N \l__regex_curr_analysis_tl
    \int_set:Nn \l__regex_min_submatch_int { 1 }
    \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
    \bool_set_false:N \l__regex_empty_success_bool
  }
```

(*__regex_match:n*, *__regex_match_cs:n*, 和 *__regex_match_init:* 定义结束。)

__regex_match_once_init: 此函数重置在找到一个匹配时使用的各种变量。在遍历字符之前调用，并且每次找到

一个匹配之前调用（这由 every_match 记号列表控制）。

首先初始化一些变量：设置用于检测相同空匹配的条件；此匹配尝试从先前的 success_pos 开始，尚未成功，尚无子匹配；清除活动线程数组，并将起始状态 0 放入其中。然后，我们几乎准备好在查询中读取我们的第一个记号，但实际上我们从开始的位置开始一步，因为__regex_match_one_token:nnN会增加\l__regex_curr_pos_int，并将\l__regex_curr_char_int保存为 last_char，以便正确识别单词边界。

```
\cs_new_protected:Npn \__regex_match_once_init:
  {
    \if_meaning:w \c_true_bool \l__regex_empty_success_bool
      \cs_set:Npn \__regex_if_two_empty_matches:F
        {
          \int_compare:nNnF
            \l__regex_start_pos_int = \l__regex_curr_pos_int
        }
    \else:
      \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
    \fi:
    \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
    \bool_set_false:N \l__regex_match_success_bool
    \tl_set:Ne \l__regex_curr_submatches_tl
      { \prg_replicate:nn { 2 * \l__regex_capturing_group_int } { 0 , } }
    \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
    \__regex_store_state:n { \l__regex_min_state_int }
    \int_set:Nn \l__regex_curr_pos_int
      { \l__regex_start_pos_int - 1 }
    \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_success_int
    \tl_build_get_intermediate:NN \l__regex_matched_analysis_tl \l__regex_internal_a_tl
    \exp_args:NNf \__regex_match_once_init_aux:
    \tl_map_inline:nn
      { \exp_after:wN \l__regex_internal_a_tl \l__regex_curr_analysis_tl }
      { \__regex_match_one_token:nnN ##1 }
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_once_init_aux:
  {
    \tl_build_begin:N \l__regex_matched_analysis_tl
    \tl_clear:N \l__regex_curr_analysis_tl
  }
```

(*__regex_match_once_init:* 定义结束。)

__regex_single_match:
__regex_multi_match:n

对于单次匹配，整体成功由唯一的匹配尝试是否成功来确定。在进行多次匹配时，只要有任何匹配成功，整体匹配就成功。执行操作#1，然后找到下一个匹配。

```
\cs_new_protected:Npn \__regex_single_match:
  {
    \tl_set:Nn \l__regex_every_match_tl
      {
        \bool_gset_eq:NN
          \g__regex_success_bool
          \l__regex_match_success_bool
        \__regex_maplike_break:
      }
  }
\cs_new_protected:Npn \__regex_multi_match:n #1
  {
    \tl_set:Nn \l__regex_every_match_tl
      {
        \if_meaning:w \c_false_bool \l__regex_match_success_bool
          \exp_after:wN \__regex_maplike_break:
        \fi:
        \bool_gset_true:N \g__regex_success_bool
        #1
        \__regex_match_once_init:
      }
  }
```

(*__regex_single_match: 和 __regex_multi_match:n 定义结束。*)

__regex_match_one_token:nnN
__regex_match_one_active:n

在每个新位置，设置一些变量并从查询中获取新字符和类别。然后解包活动线程的数组，并通过重置其长度（max_thread）清除它。这将导致一系列__regex_use_state_and_submatches:w⟨*state*⟩,⟨*submatch-clist*⟩;，然后我们按顺序考虑这些状态。只要一个线程成功，就退出该步骤，如果下一个位置有要考虑的线程，并且我们尚未达到字符串的末尾，则重复循环。否则，最后一个成功的线程即为匹配。我们在描述__regex_action_wildcard:时解释了 fresh_thread 的业务。

```
\cs_new_protected:Npn \__regex_match_one_token:nnN #1#2#3
  {
    \int_add:Nn \l__regex_step_int { 2 }
    \int_incr:N \l__regex_curr_pos_int
    \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
    \cs_set_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
    \tl_set:Nn \l__regex_curr_token_tl {#1}
    \int_set:Nn \l__regex_curr_char_int {#2}
```

```
    \int_set:Nn \l__regex_curr_catcode_int { "#3 }
    \tl_build_put_right:Ne \l__regex_matched_analysis_tl
      { \exp_not:o \l__regex_curr_analysis_tl }
    \tl_set:Nn \l__regex_curr_analysis_tl { { {#1} {#2} #3 } }
    \use:e
      {
        \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
        \int_step_function:nnN
          { \l__regex_min_thread_int }
          { \l__regex_max_thread_int - 1 }
          \__regex_match_one_active:n
      }
    \prg_break_point:
    \bool_set_false:N \l__regex_fresh_thread_bool
    \if_int_compare:w \l__regex_max_thread_int > \l__regex_min_thread_int
      \if_int_compare:w -2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
      \fi:
    \fi:
    \l__regex_every_match_tl
  }
\cs_new:Npn \__regex_match_one_active:n #1
  {
    \__regex_use_state_and_submatches:w
    \__kernel_intarray_range_to_clist:Nnn
      \g__regex_thread_info_intarray
      { 1 + #1 * (\l__regex_capturing_group_int * 2 + 1) }
      { (1 + #1) * (\l__regex_capturing_group_int * 2 + 1) }
    ;
  }
```

(*__regex_match_one_token:nnN* 和 *__regex_match_one_active:n* 定义结束。)

9.5.3 使用 nfa 的状态

__regex_use_state:

使用当前的 NFA 指令。状态最初被标记为属于当前的 step：这允许正常的自由过渡重复，但是组重复过渡则不会。一旦我们完成了所有生成的分支的探索，该状态将被标记为 step + 1：在该点击中它的任何线程都将被终止。

```
\cs_new_protected:Npn \__regex_use_state:
  {
    \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
      { \l__regex_curr_state_int } { \l__regex_step_int }
```

```
    \__regex_toks_use:w \l__regex_curr_state_int
    \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
      { \l__regex_curr_state_int }
      { \int_eval:n { \l__regex_step_int + 1 } }
  }
```

(*__regex_use_state:* 定义结束。)

__regex_use_state_and_submatches:w

此函数在数组的活动线程被解包后的新步骤中作为一个项目调用。更新 curr_state 和 curr_submatches，并在此步骤尚未遇到该状态时使用该状态。

```
\cs_new_protected:Npn \__regex_use_state_and_submatches:w #1 , #2 ;
  {
    \int_set:Nn \l__regex_curr_state_int {#1}
    \if_int_compare:w
        \__kernel_intarray_item:Nn \g__regex_state_active_intarray
          { \l__regex_curr_state_int }
                      < \l__regex_step_int
      \tl_set:Nn \l__regex_curr_submatches_tl { #2 , }
      \exp_after:wN \__regex_use_state:
    \fi:
    \scan_stop:
  }
```

(*__regex_use_state_and_submatches:w* 定义结束。)

9.5.4 匹配时的动作

__regex_action_start_wildcard:N

对于未锚定的匹配，状态 0 具有自由过渡到下一个状态和昂贵过渡到自身的过渡，以在下一个位置重复。为了捕捉重复的相同空匹配，我们需要知道成功的线程是否对应于空匹配。重置\l__regex_fresh_thread_bool的指令可能会被成功的线程跳过，因此我们也必须将其添加到__regex_match_one_token:nnN中。

```
\cs_new_protected:Npn \__regex_action_start_wildcard:N #1
  {
    \bool_set_true:N \l__regex_fresh_thread_bool
    \__regex_action_free:n {1}
    \bool_set_false:N \l__regex_fresh_thread_bool
    \bool_if:NT #1 { \__regex_action_cost:n {0} }
  }
```

(*__regex_action_start_wildcard:N* 定义结束。)

__regex_action_free:n
__regex_action_free_group:n
__regex_action_free_aux:nn

在检查 NFA 状态在此位置是否已被使用之后，这些函数复制一个线程。如果尚未使用，则在新状态中存储子匹配，并将该状态的指令插入输入流中。然后恢复\l__regex_curr_state_int和当前子匹配的旧值。两种类型的自由过渡的不同之处在于它们如何测试该状态是否已经在当前线程中的较早位置被使用：group 版本更严格，如果在当前线程中先前已经使用了该状态，则不会使用该状态，从而强制中断循环，而“normal”版本甚至会在线程内重新访问状态。

```
\cs_new_protected:Npn \__regex_action_free:n
  { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
\cs_new_protected:Npn \__regex_action_free_group:n
  { \__regex_action_free_aux:nn { < \l__regex_step_int } }
\cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
  {
    \use:e
      {
        \int_add:Nn \l__regex_curr_state_int {#2}
        \exp_not:n
          {
            \if_int_compare:w
                \__kernel_intarray_item:Nn \g__regex_state_active_intarray
                  { \l__regex_curr_state_int }
                #1
              \exp_after:wN \__regex_use_state:
            \fi:
          }
        \int_set:Nn \l__regex_curr_state_int
          { \int_use:N \l__regex_curr_state_int }
        \tl_set:Nn \exp_not:N \l__regex_curr_submatches_tl
          { \exp_not:o \l__regex_curr_submatches_tl }
      }
  }
```

(__regex_action_free:n, __regex_action_free_group:n, 和 __regex_action_free_aux:nn 定义结束。)

__regex_action_cost:n

消耗当前字符并将状态按 #1 移动的过渡。将结果状态存储在适当的数组中，以供在下一个位置使用，并存储当前子匹配。

```
\cs_new_protected:Npn \__regex_action_cost:n #1
  {
    \exp_args:Ne \__regex_store_state:n
      { \int_eval:n { \l__regex_curr_state_int + #1 } }
  }
```

(__regex_action_cost:n 定义结束。)

__regex_store_state:n
__regex_store_submatches:

将给定的状态和当前子匹配信息放入 \g__regex_thread_info_intarray，并增加数组的长度。

```
\cs_new_protected:Npn \__regex_store_state:n #1
  {
    \exp_args:No \__regex_store_submatches:nn
      \l__regex_curr_submatches_tl {#1}
    \int_incr:N \l__regex_max_thread_int
  }
\cs_new_protected:Npn \__regex_store_submatches:nn #1#2
  {
    \__kernel_intarray_gset_range_from_clist:Nnn
      \g__regex_thread_info_intarray
      {
        \__regex_int_eval:w
        1 + \l__regex_max_thread_int *
        (\l__regex_capturing_group_int * 2 + 1)
      }
      { #2 , #1 }
  }
```

(*__regex_store_state:n* 和 *__regex_store_submatches:* 定义结束。)

__regex_disable_submatches:

一些用户函数不需要跟踪子匹配。通过简单地定义相关函数来删除它们的参数并对其不进行任何操作，我们可以获得性能提升。

```
\cs_new_protected:Npn \__regex_disable_submatches:
  {
    \cs_set_protected:Npn \__regex_store_submatches:n ##1 { }
    \cs_set_protected:Npn \__regex_action_submatch:nN ##1##2 { }
  }
```

(*__regex_disable_submatches:* 定义结束。)

__regex_action_submatch:nN
__regex_action_submatch_aux:w
__regex_action_submatch_auxii:w
__regex_action_submatch_auxiii:w
__regex_action_submatch_auxiv:w

使用当前位置的信息更新当前子匹配。可能是瓶颈。

```
\cs_new_protected:Npn \__regex_action_submatch:nN #1#2
  {
    \exp_after:wN \__regex_action_submatch_aux:w
    \l__regex_curr_submatches_tl ; {#1} #2
  }
\cs_new_protected:Npn \__regex_action_submatch_aux:w #1 ; #2#3
  {
    \tl_set:Ne \l__regex_curr_submatches_tl
      {
```

```
        \prg_replicate:nn
          { #2 \if_meaning:w > #3 + \l__regex_capturing_group_int \fi: }
          { \__regex_action_submatch_auxii:w }
        \__regex_action_submatch_auxiii:w
        #1
      }
  }
\cs_new:Npn \__regex_action_submatch_auxii:w
    #1 \__regex_action_submatch_auxiii:w #2 ,
  { #2 , #1 \__regex_action_submatch_auxiii:w }
\cs_new:Npn \__regex_action_submatch_auxiii:w #1 ,
  { \int_use:N \l__regex_curr_pos_int , }
```

(*__regex_action_submatch:nN 以及其它的定义结束。*)

__regex_action_success:

当执行路径到达 NFA 的最后状态时，表示有成功匹配，除非这标志着第二个相同的空匹配。如果成功匹配为空，我们标记它为“fresh”；并存储当前位置和子匹配。然后，通过 \prg_break: 中断当前步骤，只有具有更高优先级的路径才会进一步追求。这里存储的值可能会被后续具有更高优先级的路径的成功覆盖。

```
\cs_new_protected:Npn \__regex_action_success:
  {
    \__regex_if_two_empty_matches:F
      {
        \bool_set_true:N \l__regex_match_success_bool
        \bool_set_eq:NN \l__regex_empty_success_bool
          \l__regex_fresh_thread_bool
        \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
        \int_set_eq:NN \l__regex_last_char_success_int \l__regex_last_char_int
        \tl_build_begin:N \l__regex_matched_analysis_tl
        \tl_set_eq:NN \l__regex_success_submatches_tl
          \l__regex_curr_submatches_tl
        \prg_break:
      }
  }
```

(*__regex_action_success: 定义结束。*)

9.6 替换

9.6.1 用于替换的变量和辅助工具

\l__regex_replacement_csnames_int

替换文本中的闭合括号行为取决于是否遇到 \c{ 或 \u{ 序列。存储应该由 } 关闭的“打开”这类序列的数量，由 \l__regex_replacement_csnames_int 表示，并在每

个 } 减少 1。

```
\int_new:N \l__regex_replacement_csnames_int
```

(*\l__regex_replacement_csnames_int 定义结束。*)

\l__regex_replacement_category_tl
\l__regex_replacement_category_seq

这个字母序列用于正确还原嵌套结构中的类别，比如 \cL(abc\cD(_)d)。

```
\tl_new:N \l__regex_replacement_category_tl
\seq_new:N \l__regex_replacement_category_seq
```

(*\l__regex_replacement_category_tl 和 \l__regex_replacement_category_seq 定义结束。*)

\g__regex_balance_tl

这个记号列表保存了 __regex_replacement_balance_one_match:n 构建时的替换文本。

```
\tl_new:N \g__regex_balance_tl
```

(*\g__regex_balance_tl 定义结束。*)

__regex_replacement_balance_one_match:n

期望作为参数给出一组在 \g__regex_submatch_begin_intarray（以及相关数组）中保存给定匹配的子匹配信息的索引的第一个索引。它可以在整数表达式内使用，以获取执行该匹配替换所带来的括号平衡。这包括通过删除匹配丢失的括号，由替换中出现的所有子匹配添加的括号，以及替换中明确出现的括号。尽管在使用之前总是被重新定义，但我们将其初始化为一个空替换的情况。一个重要的特性是将该函数的多个调用连接起来必须得到一个有效的整数表达式（因此在实际定义中有一个前导的 +）。

```
\cs_new:Npn \__regex_replacement_balance_one_match:n #1
  { - \__regex_submatch_balance:n {#1} }
```

(*__regex_replacement_balance_one_match:n 定义结束。*)

__regex_replacement_do_one_match:n

输入与 __regex_replacement_balance_one_match:n 相同。此函数被重新定义为展开前一匹配的末尾到给定匹配的部分的记号列表，然后是替换文本。因此，将该函数的结果与所有可能的参数（每个匹配一个调用）以及从最后一个匹配的末尾到字符串末尾的范围连接起来，就产生了完全替换的记号列表。初始化不重要，但（作为示例）我们将其设置为空替换的情况。

```
\cs_new:Npn \__regex_replacement_do_one_match:n #1
  {
    \__regex_query_range:nn
      { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
      { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
  }
```

(*__regex_replacement_do_one_match:n 定义结束。*)

__regex_replacement_exp_not:N

这个函数让我们绕过原始的 \exp_not:n 需要带括号参数的事实。据我了解，只有当用户尝试在替换文本中包含设置为宏参数字符的控制序列时才需要，例如 \c_parameter_token。实际上，在 e/x-展开分配中，\exp_not:N # 的行为就像单个 #，而 \exp_not:n {#} 的行为就像双倍的 ##。

```
\cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(*__regex_replacement_exp_not:N 定义结束。*)

__regex_replacement_exp_not:V

用于实现 \u，并且为 \peek_regex_replace_once:nnTF 重新定义。

```
\cs_new_eq:NN \__regex_replacement_exp_not:V \exp_not:V
```

(*__regex_replacement_exp_not:V 定义结束。*)

9.6.2 查询和括号平衡

__regex_query_range:nn
__regex_query_range_loop:ww

当从记号列表中提取子匹配时，各种记号存储在从 \l__regex_min_pos_int 到 \l__regex_max_pos_int（不包括）的编号的 \toks 寄存器中。函数 __regex_query_range:nn {⟨*min*⟩} {⟨*max*⟩} 从位置 ⟨*min*⟩ 到位置 ⟨*max*⟩ − 1（包括）解包寄存器。一旦这被展开，第二个 e-展开会产生查询的实际记号。这第二次展开仅在用户函数在其操作的最后，首先检查（并纠正）括号平衡之后才执行。

```
\cs_new:Npn \__regex_query_range:nn #1#2
  {
    \exp_after:wN \__regex_query_range_loop:ww
    \int_value:w \__regex_int_eval:w #1 \exp_after:wN ;
    \int_value:w \__regex_int_eval:w #2 ;
    \prg_break_point:
  }
\cs_new:Npn \__regex_query_range_loop:ww #1 ; #2 ;
  {
    \if_int_compare:w #1 < #2 \exp_stop_f:
    \else:
      \exp_after:wN \prg_break:
    \fi:
    \__regex_toks_use:w #1 \exp_stop_f:
    \exp_after:wN \__regex_query_range_loop:ww
      \int_value:w \__regex_int_eval:w #1 + 1 ; #2 ;
  }
```

(*__regex_query_range:nn 和 __regex_query_range_loop:ww 定义结束。*)

__regex_query_submatch:n 寻找给定子匹配（属于给定匹配）的起始和结束位置。

```
\cs_new:Npn \__regex_query_submatch:n #1
  {
    \__regex_query_range:nn
      { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
      { \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
  }
```

(*__regex_query_submatch:n 定义结束。*)

__regex_submatch_balance:n 每个用户函数必须生成一个平衡的记号列表(TEX 不能存储不平衡的记号列表)。当我们解包查询时，我们跟踪括号平衡，因此给定范围的贡献是在 ⟨*max pos*⟩ 和 ⟨*min pos*⟩ 的括号平衡之间的差异。这两个位置可以在相应的“子匹配”（“submatch”）数组中找到。

```
\cs_new_protected:Npn \__regex_submatch_balance:n #1
  {
    \int_eval:n
      {
        \__regex_intarray_item:NnF \g__regex_balance_intarray
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_end_intarray {#1}
          }
          { 0 }
        -
        \__regex_intarray_item:NnF \g__regex_balance_intarray
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_begin_intarray {#1}
          }
          { 0 }
      }
  }
```

(*__regex_submatch_balance:n 定义结束。*)

9.6.3 框架

__regex_replacement:n
__regex_replacement:e
__regex_replacement_apply:Nn
__regex_replacement_set:n

替换文本是逐步构建的。我们在 \l__regex_balance_int 中跟踪显式起始和结束组记号的平衡，并在 \g__regex_balance_tl 中存储一些代码以从子匹配计算括号平衡（见其描述）。检测未转义的右括号和转义字符，尾随 \prg_do_nothing:，因为后

面的一些函数需要预先查看。一旦整个替换文本被解析，确保没有打开的控制序列名称。最后，定义 `balance_one_match` 和 `do_one_match` 函数。

```
\cs_new_protected:Npn \__regex_replacement:n
  { \__regex_replacement_apply:Nn \__regex_replacement_set:n }
\cs_new_protected:Npn \__regex_replacement_apply:Nn #1#2
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_balance_int
      \tl_gclear:N \g__regex_balance_tl
      \__regex_escape_use:nnnn
        {
          \if_charcode:w \c_right_brace_str ##1
            \__regex_replacement_rbrace:N
          \else:
            \if_charcode:w \c_left_brace_str ##1
              \__regex_replacement_lbrace:N
            \else:
              \__regex_replacement_normal:n
            \fi:
          \fi:
          ##1
        }
        { \__regex_replacement_escaped:N ##1 }
        { \__regex_replacement_normal:n ##1 }
        {#2}
      \prg_do_nothing: \prg_do_nothing:
      \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
        \msg_error:nne { regex } { replacement-missing-rbrace }
          { \int_use:N \l__regex_replacement_csnames_int }
        \tl_build_put_right:Ne \l__regex_build_tl
          { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
      \fi:
      \seq_if_empty:NF \l__regex_replacement_category_seq
        {
          \msg_error:nne { regex } { replacement-missing-rparen }
            { \seq_count:N \l__regex_replacement_category_seq }
          \seq_clear:N \l__regex_replacement_category_seq
        }
      \tl_gput_right:Ne \g__regex_balance_tl
        { + \int_use:N \l__regex_balance_int }
      \tl_build_end:N \l__regex_build_tl
```

```
      \exp_args:NNo
    \group_end:
    #1 \l__regex_build_tl
  }
\cs_generate_variant:Nn \__regex_replacement:n { e }
\cs_new_protected:Npn \__regex_replacement_set:n #1
  {
    \cs_set:Npn \__regex_replacement_do_one_match:n ##1
      {
        \__regex_query_range:nn
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_prev_intarray {##1}
          }
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_begin_intarray {##1}
          }
        #1
      }
    \exp_args:Nno \use:n
      { \cs_gset:Npn \__regex_replacement_balance_one_match:n ##1 }
      {
        \g__regex_balance_tl
        - \__regex_submatch_balance:n {##1}
      }
  }
```

(*__regex_replacement:n*, *__regex_replacement_apply:Nn*, 和 *__regex_replacement_set:n* 定义结束。)

__regex_case_replacement:n
__regex_case_replacement:e

```
\tl_new:N \g__regex_case_replacement_tl
\tl_new:N \g__regex_case_balance_tl
\cs_new_protected:Npn \__regex_case_replacement:n #1
  {
    \tl_gset:Nn \g__regex_case_balance_tl
      {
        \if_case:w
          \__kernel_intarray_item:Nn
            \g__regex_submatch_case_intarray {##1}
      }
    \tl_gset_eq:NN \g__regex_case_replacement_tl \g__regex_case_balance_tl
    \tl_map_tokens:nn {#1}
```

```
      { \__regex_replacement_apply:Nn \__regex_case_replacement_aux:n }
    \tl_gset:No \g__regex_balance_tl
      { \g__regex_case_balance_tl \fi: }
    \exp_args:No \__regex_replacement_set:n
      { \g__regex_case_replacement_tl \fi: }
  }
\cs_generate_variant:Nn \__regex_case_replacement:n { e }
\cs_new_protected:Npn \__regex_case_replacement_aux:n #1
  {
    \tl_gput_right:Nn \g__regex_case_replacement_tl { \or: #1 }
    \tl_gput_right:No \g__regex_case_balance_tl
      { \exp_after:wN \or: \g__regex_balance_tl }
  }
```

(*__regex_case_replacement:n* 定义结束。)

__regex_replacement_put:n

为 \peek_regex_replace_once:nnTF 重新定义。

```
\cs_new_protected:Npn \__regex_replacement_put:n
  { \tl_build_put_right:Nn \l__regex_build_tl }
```

(*__regex_replacement_put:n* 定义结束。)

__regex_replacement_normal:n
__regex_replacement_normal_aux:N

大多数字符只是通过 \tl_build_put_right:Nn 发送到输出，除非请求了特定的类别码：然后调用 __regex_replacement_c_A:w 或类似的辅助函数。一个例外是右括号，在组开始之前恢复类别码。请注意，这里的序列是非空的：它包含一个对应于 \l__regex_replacement_category_tl 初始值的空条目。参数 #1 是一个单个字符（包括类别码为其他的空格的情况）。如果没有请求特定的类别码，我们尽可能地考虑替换执行时的当前类别码制度，所有不可能的类别码（转义、换行等）都映射到“other”。

```
\cs_new_protected:Npn \__regex_replacement_normal:n #1
  {
    \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
      { \exp_args:No \__regex_replacement_put:n { \token_to_str:N #1 } }
      {
        \tl_if_empty:NTF \l__regex_replacement_category_tl
          { \__regex_replacement_normal_aux:N #1 }
          { % (
            \token_if_eq_charcode:NNTF #1 )
              {
                \seq_pop:NN \l__regex_replacement_category_seq
                  \l__regex_replacement_category_tl
              }
```

```
              {
                \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
                ? #1
              }
          }
      }
  }
\cs_new_protected:Npn \__regex_replacement_normal_aux:N #1
  {
    \token_if_eq_charcode:NNTF #1 \c_space_token
      { \__regex_replacement_c_S:w }
      {
        \exp_after:wN \exp_after:wN
        \if_case:w \tex_catcode:D `#1 \exp_stop_f:
             \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_B:w
        \or: \__regex_replacement_c_E:w
        \or: \__regex_replacement_c_M:w
        \or: \__regex_replacement_c_T:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_P:w
        \or: \__regex_replacement_c_U:w
        \or: \__regex_replacement_c_D:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_S:w
        \or: \__regex_replacement_c_L:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_A:w
        \else: \__regex_replacement_c_O:w
        \fi:
      }
    ? #1
  }
```

(*__regex_replacement_normal:n* 和 *__regex_replacement_normal_aux:N* 定义结束。)

__regex_replacement_escaped:N

类似于解析正则表达式，如果定义了从 #1 构建的辅助函数，我们将使用它。否则，检查转义的数字（从 0 到 9 的子匹配）：其他任何字符都是原始字符。

```
\cs_new_protected:Npn \__regex_replacement_escaped:N #1
  {
    \cs_if_exist_use:cF { __regex_replacement_#1:w }
      {
        \if_int_compare:w 1 < 1#1 \exp_stop_f:
```

```
          \__regex_replacement_put_submatch:n {#1}
        \else:
          \__regex_replacement_normal:n {#1}
        \fi:
      }
  }
```

(*__regex_replacement_escaped:N* 定义结束。)

9.6.4 子匹配

__regex_replacement_put_submatch:n
__regex_replacement_put_submatch_aux:n

在替换文本中插入一个子匹配。如果子匹配编号大于捕获组的数量，则丢弃它。除非子匹配出现在 \c{...} 或 \u{...} 结构内，否则必须考虑到它在括号平衡中的影响。后来，##1 将被替换为给定匹配的第 0 个子匹配的指针。

```
\cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
  {
    \if_int_compare:w #1 < \l__regex_capturing_group_int
      \__regex_replacement_put_submatch_aux:n {#1}
    \else:
      \msg_expandable_error:nnff { regex } { submatch-too-big }
        {#1} { \int_eval:n { \l__regex_capturing_group_int - 1 } }
    \fi:
  }
\cs_new_protected:Npn \__regex_replacement_put_submatch_aux:n #1
  {
    \tl_build_put_right:Nn \l__regex_build_tl
      { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
      \tl_gput_right:Nn \g__regex_balance_tl
        { + \__regex_submatch_balance:n { \int_eval:n { #1 + ##1 } } }
    \fi:
  }
```

(*__regex_replacement_put_submatch:n* 和 *__regex_replacement_put_submatch_aux:n* 定义结束。)

__regex_replacement_g:w
__regex_replacement_g_digits:NN

为 \g 转义序列获取数字，使用原始赋值给整数 \l__regex_internal_a_int。在数字运行结束时，检查它是否以右括号结尾。

```
\cs_new_protected:Npn \__regex_replacement_g:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
      { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
      { \__regex_replacement_error:NNN g #1 #2 }
  }
```

```
\cs_new:Npn \__regex_replacement_g_digits:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
      {
        \if_int_compare:w 1 < 1#2 \exp_stop_f:
          #2
          \exp_after:wN \use_i:nnn
          \exp_after:wN \__regex_replacement_g_digits:NN
        \else:
          \exp_stop_f:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
      {
        \exp_stop_f:
        \if_meaning:w \__regex_replacement_rbrace:N #1
          \exp_args:No \__regex_replacement_put_submatch:n
            { \int_use:N \l__regex_internal_a_int }
          \exp_after:wN \use_none:nn
        \else:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
    #1 #2
  }
```

(*__regex_replacement_g:w 和 __regex_replacement_g_digits:NN 定义结束。*)

9.6.5 替换中的 csname

__regex_replacement_c:w

\c 后只能跟着一个未转义的字符。如果后面是左括号，则通过调用与 \u 共用的一个辅助函数开始控制序列。否则测试类别码是否已知；如果不是，则发出警告。

```
\cs_new_protected:Npn \__regex_replacement_c:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
      {
        \cs_if_exist:cTF { __regex_replacement_c_#2:w }
          { \__regex_replacement_cat:NNN #2 }
          { \__regex_replacement_error:NNN c #1#2 }
      }
      {
```

```
        \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
          { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
          { \__regex_replacement_error:NNN c #1#2 }
      }
  }
```

(*__regex_replacement_c:w* 定义结束。)

__regex_replacement_cu_aux:Nw

使用 \cs:w 启动一个控制序列,由 #1(__regex_replacement_exp_not:N 或 \exp_not:V) 保护不受展开，或者在另一个控制序列构造 \c 或 \u 中转换为字符串。我们使用 \tl_to_str:V 而不是 \tl_to_str:N 处理整数和其他寄存器。

```
\cs_new_protected:Npn \__regex_replacement_cu_aux:Nw #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:n { \exp_after:wN #1 \cs:w } }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
    \fi:
    \int_incr:N \l__regex_replacement_csnames_int
  }
```

(*__regex_replacement_cu_aux:Nw* 定义结束。)

__regex_replacement_u:w

检查 \u 后是否跟着左括号。如果是，则使用 \cs:w 开始控制序列，然后根据当前上下文使用 \exp_not:V 或 \tl_to_str:V 展开。

```
\cs_new_protected:Npn \__regex_replacement_u:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
      { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:V }
      { \__regex_replacement_error:NNN u #1#2 }
  }
```

(*__regex_replacement_u:w* 定义结束。)

__regex_replacement_rbrace:N

在 \c{...} 或 \u{...} 结构中，结束控制序列，并减少括号计数。否则，这是一个原始的右括号。

```
\cs_new_protected:Npn \__regex_replacement_rbrace:N #1
  {
    \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
      \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
      \int_decr:N \l__regex_replacement_csnames_int
```

```
    \else:
      \__regex_replacement_normal:n {#1}
    \fi:
  }
```

(*__regex_replacement_rbrace:N* 定义结束。)

__regex_replacement_lbrace:N

在 \c{...} 或 \u{...} 结构中，这是被禁止的。否则，这是一个原始的左括号。

```
\cs_new_protected:Npn \__regex_replacement_lbrace:N #1
  {
    \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
      \msg_error:nnn { regex } { cu-lbrace } { u }
    \else:
      \__regex_replacement_normal:n {#1}
    \fi:
  }
```

(*__regex_replacement_lbrace:N* 定义结束。)

9.6.6 替换中的字符

__regex_replacement_cat:NNN

在这里，#1 是 BEMTPUDSLOA 中的一个字母，而 #2#3 表示下一个字符。如果到达替换的末尾或者在 \c{…} 或 \u{…} 结构内部出现，则发出警告，并检测到括号的情况。在这种情况下，将当前类别存储在一个序列中，并切换到新的类别。

```
\cs_new_protected:Npn \__regex_replacement_cat:NNN #1#2#3
  {
    \token_if_eq_meaning:NNTF \prg_do_nothing: #3
      { \msg_error:nn { regex } { replacement-catcode-end } }
      {
        \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
          {
            \msg_error:nnnn
              { regex } { replacement-catcode-in-cs } {#1} {#3}
            #2 #3
          }
          {
            \__regex_two_if_eq:NNNNTF #2 #3 \__regex_replacement_normal:n (
              {
                \seq_push:NV \l__regex_replacement_category_seq
                  \l__regex_replacement_category_tl
                \tl_set:Nn \l__regex_replacement_category_tl {#1}
              }
              {
```

```
                \token_if_eq_meaning:NNT #2 \__regex_replacement_escaped:N
                  {
                    \__regex_char_if_alphanumeric:NTF #3
                      {
                        \msg_error:nnnn
                          { regex } { replacement-catcode-escaped }
                          {#1} {#3}
                      }
                      { }
                  }
                \use:c { __regex_replacement_c_#1:w } #2 #3
              }
          }
      }
  }
```

(*__regex_replacement_cat:NNN* 定义结束。)

现在我们需要多次更改空字符的类别码，因此在一个组中工作。以下是按字母顺序定义的特定类别码的宏；如果您试图理解代码，请从字母表的末尾开始，因为那些类别码比活动字符或起始组简单。

```
\group_begin:
```

__regex_replacement_char:nNN

产生任意字符-类别码对的唯一方法是使用\lowercase或\uppercase原语。这是为我们目的而包装的。第一个参数是带有各种类别码的空字符。第二个和第三个参数从输入流中抓取：#3 是要复制的字符的字符代码。我们可以使用 \char_generate:nn，但仅适用于某些类别码（不支持活动字符和空格）。

```
\cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
  {
    \tex_lccode:D 0 = `#3 \scan_stop:
    \tex_lowercase:D { \__regex_replacement_put:n {#1} }
  }
```

(*__regex_replacement_char:nNN* 定义结束。)

__regex_replacement_c_A:w

对于活动字符，必须避免扩展两次，因为我们稍后进行两次 e-展开，以拆开\toks以进行查询，并将其内容扩展为查询的标记。

```
\char_set_catcode_active:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_A:w
  { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N ^^@ } } }
```

(*__regex_replacement_c_A:w* 定义结束。)

__regex_replacement_c_B:w 明确的开始组记号增加了平衡，除非在\c{...}或\u{...}构造内。使用标准的\if_-false:技巧添加所需的开始组字符。最终我们进行两次 e-展开。第一次必须产生一个平衡的记号列表，第二次产生裸的开始组记号。严格来说，\exp_after:wN 是不绝对需要的，但与 l3tl-analysis 更一致。

```
\char_set_catcode_group_begin:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_B:w
  {
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
      \int_incr:N \l__regex_balance_int
    \fi:
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_after:wN ^^@ \if_false: } \fi: } }
  }
```

(*__regex_replacement_c_B:w 定义结束。*)

__regex_replacement_c_C:w 这与类别码相关程度不太高：当用户请求类别为“控制序列”（“control sequence”）的字符时，返回单字符控制符。与活动字符一样，我们准备进行两次 e-展开。

```
\cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
  {
    \tl_build_put_right:Nn \l__regex_build_tl
      { \exp_not:N \__regex_replacement_exp_not:N \exp_not:c {#2} }
  }
```

(*__regex_replacement_c_C:w 定义结束。*)

__regex_replacement_c_D:w 下标符合模式：\lowercase空字节与正确的类别码。

```
\char_set_catcode_math_subscript:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_D:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*__regex_replacement_c_D:w 定义结束。*)

__regex_replacement_c_E:w 与开始组情况类似，第二次 e-展开产生裸的结束组记号。

```
\char_set_catcode_group_end:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_E:w
  {
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
      \int_decr:N \l__regex_balance_int
    \fi:
    \__regex_replacement_char:nNN
      { \exp_not:n { \if_false: { \fi:  ^^@ } } }
  }
```

(*__regex_replacement_c_E:w* 定义结束。)

__regex_replacement_c_L:w 简单地 \lowercase 一个字母的空字节以生成任意字母。

```
\char_set_catcode_letter:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_L:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*__regex_replacement_c_L:w* 定义结束。)

__regex_replacement_c_M:w 这里没有什么新鲜的，我们小写空数学切换（lowercase the null math toggle ）。

```
\char_set_catcode_math_toggle:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_M:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*__regex_replacement_c_M:w* 定义结束。)

__regex_replacement_c_O:w 小写（Lowercase ）一个其他的空字节。

```
\char_set_catcode_other:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_O:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*__regex_replacement_c_O:w* 定义结束。)

__regex_replacement_c_P:w 对于宏参数，扩展是一个棘手的问题。我们需要准备两次 e-展开并穿过各种宏定义。请注意，我们不能通过将宏参数字符翻倍来替换一个\exp_not:n，因为如果恶作剧的用户要求\c{\cP\#}，那么该宏参数字符将被翻倍。

```
\char_set_catcode_parameter:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_P:w
  {
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
  }
```

(*__regex_replacement_c_P:w* 定义结束。)

__regex_replacement_c_S:w 空格在输入时被 TEX 标准化为字符代码 32。事实上，不可能获得字符代码为 0 且类别码为 10 的标记。因此，我们使用 32 而不是 0 作为我们的基础字符。

```
\cs_new_protected:Npn \__regex_replacement_c_S:w #1#2
  {
    \if_int_compare:w `#2 = \c_zero_int
      \msg_error:nn { regex } { replacement-null-space }
    \fi:
    \tex_lccode:D `\ = `#2 \scan_stop:
    \tex_lowercase:D { \__regex_replacement_put:n {~} }
  }
```

(*__regex_replacement_c_S:w* 定义结束。)

__regex_replacement_c_T:w 对齐制表符在这里没有什么新鲜的。只要需要，这些制表符就会被适当的括号包围，因此它们不会在对齐设置中引起麻烦。

```
\char_set_catcode_alignment:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_T:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*__regex_replacement_c_T:w* 定义结束。)

__regex_replacement_c_U:w 对__regex_replacement_char:nNN的简单调用，它将数学上标^^@小写。

```
\char_set_catcode_math_superscript:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_U:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*__regex_replacement_c_U:w* 定义结束。)

恢复空字节的类别码。

```
\group_end:
```

9.6.7 一个错误

__regex_replacement_error:NNN 通过调用 replacement-c、replacement-g 或 replacement-u 中的一个消息来进行简单的错误报告。

```
\cs_new_protected:Npn \__regex_replacement_error:NNN #1#2#3
  {
    \msg_error:nne { regex } { replacement-#1 } {#3}
    #2 #3
  }
```

(*__regex_replacement_error:NNN* 定义结束。)

9.7 用户函数

\regex_new:N 在分配合理值之前，正则表达式变量匹配空。

```
\cs_new_protected:Npn \regex_new:N #1
  { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(*\regex_new:N* 定义结束。这个函数被记录在第12页。)

\l_tmpa_regex 常规的临时空间。
\l_tmpb_regex
\g_tmpa_regex
\g_tmpb_regex

```
\regex_new:N \l_tmpa_regex
\regex_new:N \l_tmpb_regex
\regex_new:N \g_tmpa_regex
\regex_new:N \g_tmpb_regex
```

(*\l_tmpa_regex* 以及其它的定义结束。这些变量被记录在第17页。)

\regex_set:Nn
\regex_gset:Nn
\regex_const:Nn

编译，然后使用适当的赋值函数将结果存储在用户变量中。

```
\cs_new_protected:Npn \regex_set:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_set_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \regex_gset:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_gset_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \regex_const:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_const:Ne #1 { \exp_not:o \l__regex_internal_regex }
  }
```

(*\regex_set:Nn*, *\regex_gset:Nn*, 和 *\regex_const:Nn* 定义结束。这些函数被记录在第12页。)

\regex_show:n
\regex_log:n
__regex_show:Nn
\regex_show:N
\regex_log:N
__regex_show:NN

用户函数：n 变体需要先进行编译。然后使用一些适当的文本显示变量。辅助的 __regex_show:N 在不同的部分中定义。

```
\cs_new_protected:Npn \regex_show:n { \__regex_show:Nn \msg_show:nneeee }
\cs_new_protected:Npn \regex_log:n { \__regex_show:Nn \msg_log:nneeee }
\cs_new_protected:Npn \__regex_show:Nn #1#2
  {
    \__regex_compile:n {#2}
    \__regex_show:N \l__regex_internal_regex
    #1 { regex } { show }
      { \tl_to_str:n {#2} } { }
      { \l__regex_internal_a_tl } { }
  }
\cs_new_protected:Npn \regex_show:N { \__regex_show:NN \msg_show:nneeee }
\cs_new_protected:Npn \regex_log:N { \__regex_show:NN \msg_log:nneeee }
\cs_new_protected:Npn \__regex_show:NN #1#2
  {
    \__kernel_chk_tl_type:NnnT #2 { regex }
      { \exp_args:No \__regex_clean_regex:n {#2} }
      {
        \__regex_show:N #2
        #1 { regex } { show }
          { } { \token_to_str:N #2 }
```

```
          { \l__regex_internal_a_tl } { }
      }
  }
```

(*\regex_show:n* 以及其它的定义结束。这些函数被记录在第12页。)

\regex_match:nnTF
\regex_match:nVTF
\regex_match:NnTF
\regex_match:NVTF

这些条件基于稍后定义的一个常见辅助工具。它的第一个参数构建与正则表达式对应的 NFA，第二个参数是查询记号列表。一旦完成匹配，将结果布尔值转换为 \prg_return_true: 或 \prg_return_false:。

```
\prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
  {
    \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
    \__regex_return:
  }
\prg_generate_conditional_variant:Nnn \regex_match:nn { nV } { T , F , TF }
\prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
  {
    \__regex_if_match:nn { \__regex_build:N #1 } {#2}
    \__regex_return:
  }
\prg_generate_conditional_variant:Nnn \regex_match:Nn { NV } { T , F , TF }
```

(*\regex_match:nnTF* 和 *\regex_match:NnTF* 定义结束。这些函数被记录在第13页。)

\regex_count:nnN
\regex_count:nVN
\regex_count:NnN
\regex_count:NVN

再次使用一个辅助工具，其第一个参数构建 NFA 。

```
\cs_new_protected:Npn \regex_count:nnN #1
  { \__regex_count:nnN { \__regex_build:n {#1} } }
\cs_new_protected:Npn \regex_count:NnN #1
  { \__regex_count:nnN { \__regex_build:N #1 } }
\cs_generate_variant:Nn \regex_count:nnN { nV }
\cs_generate_variant:Nn \regex_count:NnN { NV }
```

(*\regex_count:nnN* 和 *\regex_count:NnN* 定义结束。这些函数被记录在第13页。)

\regex_match_case:nn
\regex_match_case:nnTF

辅助工具如果 #1 具有奇数个项目，则会发生错误，否则根据找到的情况设置\g__regex_case_int（如果未找到，则为零）。true 分支将相应的代码留在输入流中。

```
\cs_new_protected:Npn \regex_match_case:nnTF #1#2#3
  {
    \__regex_match_case:nnTF {#1} {#2}
      {
        \tl_item:nn {#1} { 2 * \g__regex_case_int }
        #3
      }
```

```
  }
\cs_new_protected:Npn \regex_match_case:nn #1#2
  { \regex_match_case:nnTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_match_case:nnT #1#2#3
  { \regex_match_case:nnTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_match_case:nnF #1#2
  { \regex_match_case:nnTF {#1} {#2} { } }
```

(*\regex_match_case:nnTF* 定义结束。这个函数被记录在第13页。)

\regex_extract_once:nnN
\regex_extract_once:nVN
\regex_extract_once:nnNTF
\regex_extract_once:nVNTF
\regex_extract_once:NnN
\regex_extract_once:NVN
\regex_extract_once:NnNTF
\regex_extract_once:NVNTF
\regex_extract_all:nnN
\regex_extract_all:nVN
\regex_extract_all:nnNTF
\regex_extract_all:nVNTF
\regex_extract_all:NnN
\regex_extract_all:NVN
\regex_extract_all:NnNTF
\regex_extract_all:NVNTF
\regex_replace_once:nnN
\regex_replace_once:nVN
\regex_replace_once:nnNTF
\regex_replace_once:nVNTF
\regex_replace_once:NnN
\regex_replace_once:NVN
\regex_replace_once:NnNTF
\regex_replace_once:NVNTF
\regex_replace_all:nnN
\regex_replace_all:nVN
\regex_replace_all:nnNTF
\regex_replace_all:nVNTF
\regex_replace_all:NnN
\regex_replace_all:NVN
\regex_replace_all:NnNTF
\regex_replace_all:NVNTF
\regex_split:NnN
\regex_split:NVN
\regex_split:NnNTF
\regex_split:NVNTF

我们在这里定义了 40 个用户函数，遵循在:nnN 辅助工具中的一个常见模式，这些辅助工具在接下来的子部分中定义。该辅助工具使用__regex_build:n或 __regex_build:N作为适当的正则表达式参数，然后使用所有其他必要的参数（替换文本、记号列表等）。条件调用__regex_return:以在执行匹配后返回 true 或 false。

```
\cs_set_protected:Npn \__regex_tmp:w #1#2#3
  {
    \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
    \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N  ##1  } }
    \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
      { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
    \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
      { #1 { \__regex_build:N  ##1  } {##2} ##3 \__regex_return: }
    \cs_generate_variant:Nn #2 { nV }
    \prg_generate_conditional_variant:Nnn #2 { nV } { T , F , TF }
    \cs_generate_variant:Nn #3 { NV }
    \prg_generate_conditional_variant:Nnn #3 { NV } { T , F , TF }

  }
\__regex_tmp:w \__regex_extract_once:nnN
  \regex_extract_once:nnN \regex_extract_once:NnN
\__regex_tmp:w \__regex_extract_all:nnN
  \regex_extract_all:nnN \regex_extract_all:NnN
\__regex_tmp:w \__regex_replace_once:nnN
  \regex_replace_once:nnN \regex_replace_once:NnN
\__regex_tmp:w \__regex_replace_all:nnN
  \regex_replace_all:nnN \regex_replace_all:NnN
\__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN
```

(*\regex_extract_once:nnNTF* 以及其它的定义结束。这些函数被记录在第14页。)

如果输入不正确(项目数为奇数)，则执行 false 分支。否则，使用与 \regex_replace_once:nnN相同的辅助函数，但代码更复杂，用于构建自动机，并找到要使用的替换

文本。\tl_item:nn仅在我们知道\g__regex_case_int 的值时才会展开，即匹配的是哪个 case。

```
\cs_new_protected:Npn \regex_replace_case_once:nNTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_replace_case_once:nN(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
      {
        \__regex_replace_once_aux:nnN
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          { \__regex_replacement:e { \tl_item:nn {#1} { 2 * \g__regex_case_int } } }
          #2
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new_protected:Npn \regex_replace_case_once:nN #1#2
  { \regex_replace_case_once:nNTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_replace_case_once:nNT #1#2#3
  { \regex_replace_case_once:nNTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_replace_case_once:nNF #1#2
  { \regex_replace_case_once:nNTF {#1} {#2} { } }
```

(*\regex_replace_case_once:nNTF* 定义结束。这个函数被记录在第16页。)

\regex_replace_case_all:nN
\regex_replace_case_all:nN*TF*

如果输入不正确(项目数为奇数)，则执行 false 分支。否则，使用与 \regex_replace_all:nnN相同的辅助函数，但代码更复杂，用于构建自动机，并找到要使用的替换文本。

```
\cs_new_protected:Npn \regex_replace_case_all:nNTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_replace_case_all:nN(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
      {
        \__regex_replace_all_aux:nnN
```

```
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          { \__regex_case_replacement:e { \__regex_tl_even_items:n {#1} } }
          #2
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new_protected:Npn \regex_replace_case_all:nN #1#2
  { \regex_replace_case_all:nNTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_replace_case_all:nNT #1#2#3
  { \regex_replace_case_all:nNTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_replace_case_all:nNF #1#2
  { \regex_replace_case_all:nNTF {#1} {#2} { } }
```

(*\regex_replace_case_all:nNTF* 定义结束。这个函数被记录在第17页。)

9.7.1 用户函数的变量和辅助工具

\l__regex_match_count_int

到目前为止找到的匹配数量存储在\l__regex_match_count_int中。这仅在 \regex_count:nnN函数中使用。

```
\int_new:N \l__regex_match_count_int
```

(*\l__regex_match_count_int* 定义结束。)

__regex_begin
__regex_end

这些标志被设置为指示需要在提取子匹配时添加的开始组或结束组记号。

```
\flag_new:n { __regex_begin }
\flag_new:n { __regex_end }
```

(*__regex_begin* 和 *__regex_end* 定义结束。)

\l__regex_min_submatch_int
\l__regex_submatch_int
\l__regex_zeroth_submatch_int

每个子匹配的端点存储在两个数组中，其索引⟨*submatch*⟩的范围从 \l__regex_min_submatch_int（包括）到\l__regex_submatch_int（不包括）。每次成功匹配都带有一个 0-th 子匹配（完全匹配），以及每个捕获组的一个匹配：与上一次成功匹配对应的子匹配以 zeroth_submatch 开始标记。在\g__regex_submatch_prev_intarray中的条目\l__regex_zeroth_submatch_int中存储了该匹配尝试开始的位置：这用于拆分和替换。

```
\int_new:N \l__regex_min_submatch_int
\int_new:N \l__regex_submatch_int
\int_new:N \l__regex_zeroth_submatch_int
```

(*\l__regex_min_submatch_int*，*\l__regex_submatch_int*，和 *\l__regex_zeroth_submatch_int* 定义结束。)

\g__regex_submatch_prev_intarray
\g__regex_submatch_begin_intarray
\g__regex_submatch_end_intarray
\g__regex_submatch_case_intarray

分别保存匹配尝试开始的位置、每个子匹配的端点以及匹配对应的正则表达式案例。

```
\intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_case_intarray { 65536 }
```

(*\g__regex_submatch_prev_intarray 以及其它的定义结束。*)

\g__regex_balance_intarray

匹配时的第一步是将开始组/结束组字符的平衡存储在\g__regex_balance_intarray中。

```
\intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(*\g__regex_balance_intarray 定义结束。*)

\l__regex_added_begin_int
\l__regex_added_end_int

在执行正则表达式操作（如替换）时跟踪要添加的左/右括号的数量。

```
\int_new:N \l__regex_added_begin_int
\int_new:N \l__regex_added_end_int
```

(*\l__regex_added_begin_int 和 \l__regex_added_end_int 定义结束。*)

__regex_return:

该函数根据是否找到匹配触发\prg_return_false:或\prg_return_true:。它被所有用户条件使用。

```
\cs_new_protected:Npn \__regex_return:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*__regex_return: 定义结束。*)

__regex_query_set:n
__regex_query_set_aux:nN

一旦找到要切割的位置，为了轻松提取输入的子集，将输入标记一个接一个地存储到连续的\toks寄存器中。还在一个数组中存储括号平衡（用于检查总体括号平衡）。

```
\cs_new_protected:Npn \__regex_query_set:n #1
  {
    \int_zero:N \l__regex_balance_int
    \int_zero:N \l__regex_curr_pos_int
    \__regex_query_set_aux:nN { } F
    \tl_analysis_map_inline:nn {#1}
      { \__regex_query_set_aux:nN {##1} ##3 }
    \__regex_query_set_aux:nN { } F
    \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
```

```
  }
\cs_new_protected:Npn \__regex_query_set_aux:nN #1#2
  {
    \int_incr:N \l__regex_curr_pos_int
    \__regex_toks_set:Nn \l__regex_curr_pos_int {#1}
    \__kernel_intarray_gset:Nnn \g__regex_balance_intarray
      { \l__regex_curr_pos_int } { \l__regex_balance_int }
    \if_case:w "#2 \exp_stop_f:
    \or: \int_incr:N \l__regex_balance_int
    \or: \int_decr:N \l__regex_balance_int
    \fi:
  }
```

(*__regex_query_set:n* 和 *__regex_query_set_aux:nN* 定义结束。)

9.7.2 匹配

__regex_if_match:nn

我们不跟踪子匹配，并在找到单个匹配后停止。使用 #1 构建 NFA，并在查询 #2 上执行匹配。

```
\cs_new_protected:Npn \__regex_if_match:nn #1#2
  {
    \group_begin:
      \__regex_disable_submatches:
      \__regex_single_match:
      #1
      \__regex_match:n {#2}
    \group_end:
  }
```

(*__regex_if_match:nn* 定义结束。)

__regex_match_case:nnTF
__regex_match_case_aux:nn

如果 #1 中的项目数不是偶数，代码将被严重破坏，因此我们捕捉此情况，然后执行与 \regex_match:nnTF 相同的代码，但使用 __regex_case_build:n，并且不返回结果。

```
\cs_new_protected:Npn \__regex_match_case:nnTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_match_case:nn(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
```

```
      {
        \__regex_if_match:nn
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          {#2}
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new:Npn \__regex_match_case_aux:nn #1#2 { \exp_not:n { {#1} } }
```

(*__regex_match_case:nnTF* 和 *__regex_match_case_aux:nn* 定义结束。)

__regex_count:nnN

再次，我们不关心子匹配。与其在找到第一个“最长匹配”（“longest match”）后中止，我们搜索多个匹配，在每次匹配后增加 `\l__regex_match_count_int` 以记录匹配的数量。构建 NFA 并进行匹配。最后，将结果存储在用户变量中。

```
\cs_new_protected:Npn \__regex_count:nnN #1#2#3
  {
    \group_begin:
      \__regex_disable_submatches:
      \int_zero:N \l__regex_match_count_int
      \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
      #1
      \__regex_match:n {#2}
      \exp_args:NNNo
    \group_end:
    \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
  }
```

(*__regex_count:nnN* 定义结束。)

9.7.3 提取子匹配

__regex_extract_once:nnN
__regex_extract_all:nnN

匹配一次或多次。在每次匹配（或仅在匹配后）之后，使用 `\__regex_extract:` 提取子匹配。最后，在关闭组之后将包含所有子匹配的序列存储到用户变量 `#3` 中。

```
\cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
      #1
      \__regex_match:n {#2}
      \__regex_extract:
      \__regex_query_set:n {#2}
    \__regex_group_end_extract_seq:N #3
  }
```

```
\cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
      \__regex_match:n {#2}
      \__regex_query_set:n {#2}
    \__regex_group_end_extract_seq:N #3
  }
```

(*__regex_extract_once:nnN* 和 *__regex_extract_all:nnN* 定义结束。)

__regex_split:nnN

在子匹配处进行拆分有些棘手。对于每个匹配，提取所有子匹配，并将零子匹配的部分替换为匹配尝试开始和零子匹配开始之间的查询部分。如果定界符在此匹配尝试的开头匹配了一个空的记号列表，则会阻止此操作。在最后的匹配后，存储记号列表的最后部分，该部分范围从匹配尝试的开始到查询的结尾。如果最后的匹配为空且在最后，则会阻止此操作：减少的量 `\l__regex_submatch_int`——它控制将使用哪些匹配。

```
\cs_new_protected:Npn \__regex_split:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n
        {
          \if_int_compare:w
            \l__regex_start_pos_int < \l__regex_success_pos_int
            \__regex_extract:
            \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
              { \l__regex_zeroth_submatch_int } { 0 }
            \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
              { \l__regex_zeroth_submatch_int }
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
                  { \l__regex_zeroth_submatch_int }
              }
            \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
              { \l__regex_zeroth_submatch_int }
              { \l__regex_start_pos_int }
          \fi:
        }
      #1
      \__regex_match:n {#2}
      \__regex_query_set:n {#2}
```

```
      \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
        { \l__regex_submatch_int } { 0 }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
        { \l__regex_submatch_int }
        { \l__regex_max_pos_int }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
        { \l__regex_submatch_int }
        { \l__regex_start_pos_int }
      \int_incr:N \l__regex_submatch_int
      \if_meaning:w \c_true_bool \l__regex_empty_success_bool
        \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
          \int_decr:N \l__regex_submatch_int
        \fi:
      \fi:
    \__regex_group_end_extract_seq:N #3
  }
```

(__regex_split:nnN 定义结束。)

__regex_group_end_extract_seq:N
__regex_extract_seq:N
__regex_extract_seq:NNn
__regex_extract_seq_loop:Nw

子匹配的端点存储为两个数组的条目，从 \l__regex_min_submatch_int 到 \l__regex_submatch_int（不包括 \l__regex_submatch_int）。将相关范围提取到 \g__regex_internal_tl 中，用 __regex_tmp:w {} 分隔。我们在两个标志 __regex_begin 和 __regex_end 中跟踪添加到每个项目中的总体平衡的开始组或结束组记号数。在此步骤中，}{ 被视为平衡的（具有相同数量的开始组和结束组记号）。这个问题会被稍后解释的 __regex_extract_check:w 捕捉到。在抱怨了我们必须添加的任何开始组或结束组记号之后，我们准备在组外构造用户的序列。

```
\cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
  {
      \flag_clear:n { __regex_begin }
      \flag_clear:n { __regex_end }
      \cs_set_eq:NN \__regex_tmp:w \scan_stop:
      \__kernel_tl_gset:Ne \g__regex_internal_tl
        {
          \int_step_function:nnN { \l__regex_min_submatch_int }
            { \l__regex_submatch_int - 1 } \__regex_extract_seq_aux:n
          \__regex_tmp:w
        }
      \int_set:Nn \l__regex_added_begin_int
        { \flag_height:n { __regex_begin } }
      \int_set:Nn \l__regex_added_end_int
        { \flag_height:n { __regex_end } }
      \tex_afterassignment:D \__regex_extract_check:w
```

```
      \__kernel_tl_gset:Ne \g__regex_internal_tl
        { \g__regex_internal_tl \if_false: { \fi: } }
      \int_compare:nNnT
        { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
        {
          \msg_error:nneee { regex } { result-unbalanced }
            { splitting~or~extracting~submatches }
            { \int_use:N \l__regex_added_begin_int }
            { \int_use:N \l__regex_added_end_int }
        }
    \group_end:
    \__regex_extract_seq:N #1
  }
\cs_gset_protected:Npn \__regex_extract_seq:N #1
  {
    \seq_clear:N #1
    \cs_set_eq:NN \__regex_tmp:w  \__regex_extract_seq_loop:Nw
    \exp_after:wN \__regex_extract_seq:NNn
    \exp_after:wN #1
    \g__regex_internal_tl \use_none:nnn
  }
\cs_new_protected:Npn \__regex_extract_seq:NNn #1#2#3
  { #3 #2 #1 \prg_do_nothing: }
\cs_new_protected:Npn \__regex_extract_seq_loop:Nw #1#2 \__regex_tmp:w #3
  {
    \seq_put_right:No #1 {#2}
    #3 \__regex_extract_seq_loop:Nw #1 \prg_do_nothing:
  }
```

(*__regex_group_end_extract_seq:N* 以及其它的定义结束。)

__regex_extract_seq_aux:n
__regex_extract_seq_aux:ww

:n 辅助函数构建子匹配序列的一项。首先计算子匹配的括号平衡，然后从查询中提取子匹配，添加适当的括号，并在子匹配不平衡时引发一个标志。

```
\cs_new:Npn \__regex_extract_seq_aux:n #1
  {
    \__regex_tmp:w { }
    \exp_after:wN \__regex_extract_seq_aux:ww
    \int_value:w \__regex_submatch_balance:n {#1} ; #1;
  }
\cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
  {
    \if_int_compare:w #1 < \c_zero_int
      \prg_replicate:nn {-#1}
```

```
        {
          \flag_raise:n { __regex_begin }
          \exp_not:n { { \if_false: } \fi: }
        }
    \fi:
    \__regex_query_submatch:n {#2}
    \if_int_compare:w #1 > \c_zero_int
      \prg_replicate:nn {#1}
        {
          \flag_raise:n { __regex_end }
          \exp_not:n { \if_false: { \fi: } }
        }
    \fi:
  }
```

(*__regex_extract_seq_aux:n 和 __regex_extract_seq_aux:ww 定义结束。*)

__regex_extract_check:w
__regex_extract_check:n
__regex_extract_check_loop:w
__regex_extract_check_end:w

在 __regex_group_end_extract_seq:N 中，我们必须展开 \g__regex_internal_tl，将 \if_false: 结构转换为实际的开始和结束组标记。这是通过 __kernel_tl_gset:Ne 赋值完成的，并且由于 \afterassignment 原语，在此赋值结束后立即运行 __regex_extract_check:w。如果所有项目都平衡（足够的开始组标记在结束组标记之前，所以 }{ 不是），那么在 __kernel_tl_gset:Ne 的右括号之前（由于我们巧妙的 \if_false: { \fi: } 结构）调用 __regex_extract_check:w，并发现没有剩余的要展开的内容。如果任何项目不平衡，赋值会提前结束，由额外的结束组标记，而我们的检查会发现需要在新的 __kernel_tl_gset:Ne 赋值中展开更多标记。我们需要为不平衡的项目添加开始组和结束组标记，即为到目前为止找到的最后一个项目，我们通过循环到达该项目。

```
\cs_new_protected:Npn \__regex_extract_check:w
  {
    \exp_after:wN \__regex_extract_check:n
    \exp_after:wN { \if_false: } \fi:
  }
\cs_new_protected:Npn \__regex_extract_check:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \int_incr:N \l__regex_added_begin_int
        \int_incr:N \l__regex_added_end_int
        \tex_afterassignment:D \__regex_extract_check:w
        \__kernel_tl_gset:Ne \g__regex_internal_tl
          {
```

```
            \exp_after:wN \__regex_extract_check_loop:w
            \g__regex_internal_tl
            \__regex_tmp:w \__regex_extract_check_end:w
            #1
          }
      }
  }
\cs_new:Npn \__regex_extract_check_loop:w #1 \__regex_tmp:w #2
  {
    #2
    \exp_not:o {#1}
    \__regex_tmp:w { }
    \__regex_extract_check_loop:w \prg_do_nothing:
  }
```

`\__regex_extract_check_end:w` 的参数是：`#1` 是额外的结束组标记之前项目的一部分；`#2` 是废料；`#3` 是 `\prg_do_nothing:` 后跟尚未展开的项目的部分，它是额外的结束组标记之后。在替换文本中，第一个括号和 `\if_false: { \fi: }` 结构是添加的开始组和结束组标记（后者尚未展开，就像 `#3` 一样），而在 `\exp_not:o {#1}` 之后的关闭括号替换了提前结束赋值的额外结束组标记。特别是这意味着该结束组标记的字符代码丢失了。

```
\cs_new:Npn \__regex_extract_check_end:w
    \exp_not:o #1#2 \__regex_extract_check_loop:w #3 \__regex_tmp:w
  {
    { \exp_not:o {#1} }
    #3
    \if_false: { \fi: }
    \__regex_tmp:w
  }
```

(*__regex_extract_check:w 以及其它的定义结束。*)

`\__regex_extract:`
`\__regex_extract_aux:w`

我们的任务是存储子匹配的端点列表，并将它们存储在适当的数组条目中，从 `\l__regex_zeroth_submatch_int` 开始。首先，我们在 `\g__regex_submatch_prev_intarray` 中存储了匹配尝试开始的位置。我们从逗号列表 `\l__regex_success_submatches_tl` 中提取其余部分，该列表从存储在 `\g__regex_submatch_begin_intarray` 中的条目开始，然后是 `\g__regex_submatch_end_intarray` 的条目。

```
\cs_new_protected:Npn \__regex_extract:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
      \prg_replicate:nn \l__regex_capturing_group_int
```

```
        {
          \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
            { \l__regex_submatch_int } { 0 }
          \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
            { \l__regex_submatch_int } { 0 }
          \int_incr:N \l__regex_submatch_int
        }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
        { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
        { \l__regex_zeroth_submatch_int } { \g__regex_case_int }
      \int_zero:N \l__regex_internal_a_int
      \exp_after:wN \__regex_extract_aux:w \l__regex_success_submatches_tl
        \prg_break_point: \__regex_use_none_delimit_by_q_recursion_stop:w ,
        \q__regex_recursion_stop
    \fi:
  }
\cs_new_protected:Npn \__regex_extract_aux:w #1 ,
  {
    \prg_break: #1 \prg_break_point:
    \if_int_compare:w \l__regex_internal_a_int < \l__regex_capturing_group_int
      \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
        { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int } {#1
    \else:
      \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
        { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int - \l_
    \fi:
    \int_incr:N \l__regex_internal_a_int
    \__regex_extract_aux:w
  }
```

(*__regex_extract:* 和 *__regex_extract_aux:w* 定义结束。)

9.7.4 替换

__regex_replace_once:nnN
__regex_replace_once_aux:nnN

构建 NFA 和替换函数，然后找到单个匹配。如果匹配失败，就简单地退出组。否则，我们进行替换。提取子匹配。计算替换此匹配的替换的括号平衡（这取决于子匹配）。准备替换的记号列表：替换函数生成从查询的开始到匹配的开始和此匹配的替换文本的标记；我们需要添加从匹配的末尾到查询的末尾的标记。最后，在关闭组之后将结果存储在用户变量中：这一步涉及额外的 e-展开，并检查最终结果中的括号是否平衡。

```
\cs_new_protected:Npn \__regex_replace_once:nnN #1#2
```

```
  { \__regex_replace_once_aux:nnN {#1} { \__regex_replacement:n {#2} } }
\cs_new_protected:Npn \__regex_replace_once_aux:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
      #1
      \exp_args:No \__regex_match:n {#3}
    \bool_if:NTF \g__regex_success_bool
      {
        \__regex_extract:
        \exp_args:No \__regex_query_set:n {#3}
        #2
        \int_set:Nn \l__regex_balance_int
          {
            \__regex_replacement_balance_one_match:n
              { \l__regex_zeroth_submatch_int }
          }
        \__kernel_tl_set:Ne \l__regex_internal_a_tl
          {
            \__regex_replacement_do_one_match:n
              { \l__regex_zeroth_submatch_int }
            \__regex_query_range:nn
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
                  { \l__regex_zeroth_submatch_int }
              }
              { \l__regex_max_pos_int }
          }
        \__regex_group_end_replace:N #3
      }
      { \group_end: }
  }
```

(*__regex_replace_once:nnN 和 __regex_replace_once_aux:nnN 定义结束。*)

__regex_replace_all:nnN

多次匹配，对于每次匹配，提取子匹配并额外存储匹配尝试开始的位置。从 \l__regex_min_submatch_int 到 \l__regex_submatch_int 的条目按顺序保存有关每次匹配的子匹配的信息；每次匹配对应于 \l__regex_capturing_group_int 个连续条目。计算执行所有替换所对应的括号平衡：这是替换每个匹配的括号平衡的总和。将每个匹配的替换文本（包括匹配之前的查询部分）和查询的末尾连接在一起。

```
\cs_new_protected:Npn \__regex_replace_all:nnN #1#2
  { \__regex_replace_all_aux:nnN {#1} { \__regex_replacement:n {#2} } }
```

```
\cs_new_protected:Npn \__regex_replace_all_aux:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
      \exp_args:No \__regex_match:n {#3}
      \exp_args:No \__regex_query_set:n {#3}
      #2
      \int_set:Nn \l__regex_balance_int
        {
          0
          \int_step_function:nnnN
            { \l__regex_min_submatch_int }
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - 1 }
            \__regex_replacement_balance_one_match:n
        }
      \__kernel_tl_set:Ne \l__regex_internal_a_tl
        {
          \int_step_function:nnnN
            { \l__regex_min_submatch_int }
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - 1 }
            \__regex_replacement_do_one_match:n
          \__regex_query_range:nn
            \l__regex_start_pos_int \l__regex_max_pos_int
        }
    \__regex_group_end_replace:N #3
  }
```

(*__regex_replace_all:nnN* 定义结束。)

__regex_group_end_replace:N
__regex_group_end_replace_try:
__regex_group_end_replace_check:w
__regex_group_end_replace_check:n

在这个阶段，\l__regex_internal_a_tl（通过 e-展开为期望的结果）。根据 \l__regex_balance_int 猜测在结果之前或之后添加括号的数量，然后尝试展开。最简单的情况是，\l__regex_internal_a_tl 与通过 \prg_replicate:nn 插入的括号一起产生平衡的结果，并且赋值在 \if_false: { \fi: } 结构结束：然后 __regex_group_end_replace_check:w 看到没有剩余材料，我们成功地找到了结果。较难的情况是展开 \l__regex_internal_a_tl 可能会产生额外的闭合括号并提前结束赋值。然后，我们使用; 重要的是，后面的内容尚未展开，因此 __regex_group_end_replace_check:n 抓取直到 __regex_group_end_replace_try: 中的最后一个括号的所有内容，这样我们可以尝试再次使用额外的括号对结果进行环绕。

```
\cs_new_protected:Npn \__regex_group_end_replace:N #1
  {
    \int_set:Nn \l__regex_added_begin_int
      { \int_max:nn { - \l__regex_balance_int } { 0 } }
    \int_set:Nn \l__regex_added_end_int
      { \int_max:nn { \l__regex_balance_int } { 0 } }
    \__regex_group_end_replace_try:
    \int_compare:nNnT { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
      {
        \msg_error:nneee { regex } { result-unbalanced }
          { replacing } { \int_use:N \l__regex_added_begin_int }
          { \int_use:N \l__regex_added_end_int }
      }
    \group_end:
    \tl_set_eq:NN #1 \g__regex_internal_tl
  }
\cs_new_protected:Npn \__regex_group_end_replace_try:
  {
    \tex_afterassignment:D \__regex_group_end_replace_check:w
    \__kernel_tl_gset:Ne \g__regex_internal_tl
      {
        \prg_replicate:nn { \l__regex_added_begin_int } { { \if_false: } \fi: }
        \l__regex_internal_a_tl
        \prg_replicate:nn { \l__regex_added_end_int } { \if_false: { \fi: } }
        \if_false: { \fi: }
      }
  }
\cs_new_protected:Npn \__regex_group_end_replace_check:w
  {
    \exp_after:wN \__regex_group_end_replace_check:n
    \exp_after:wN { \if_false: } \fi:
  }
\cs_new_protected:Npn \__regex_group_end_replace_check:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \int_incr:N \l__regex_added_begin_int
        \int_incr:N \l__regex_added_end_int
        \__regex_group_end_replace_try:
      }
  }
```

(*__regex_group_end_replace:N* 以及其它的定义结束。)

9.7.5 预览

\l__regex_peek_true_tl
\l__regex_peek_false_tl

\peek_regex:nTF 或类似命令的真/假代码参数。

```
\tl_new:N \l__regex_peek_true_tl
\tl_new:N \l__regex_peek_false_tl
```

(*\l__regex_peek_true_tl 和 \l__regex_peek_false_tl 定义结束。*)

\l__regex_replacement_tl

在 \peek_regex_replace_once:nnTF 中预览时，我们需要存储替换文本。

```
\tl_new:N \l__regex_replacement_tl
```

(*\l__regex_replacement_tl 定义结束。*)

\l__regex_input_tl
__regex_input_item:n

将每个作为 __regex_input_item:n {⟨*tokens*⟩} 找到的记号存储在其中，其中 ⟨*tokens*⟩ o-展开为找到的记号，与 \tl_analysis_map_inline:nn 类似。

```
\tl_new:N \l__regex_input_tl
\cs_new_eq:NN \__regex_input_item:n ?
```

(*\l__regex_input_tl 和 __regex_input_item:n 定义结束。*)

\peek_regex:n*TF*
\peek_regex:N*TF*
\peek_regex_remove_once:n*TF*
\peek_regex_remove_once:N*TF*

T 和 F 函数只是调用相应的 TF 函数。四个 TF 函数在两个方面有所不同：是否删除记号，通过使用 __regex_peek_end: 或 __regex_peek_remove_end:n（后者需要一个参数，正如我们将看到的）来区分，以及正则表达式是否必须编译或已经在 N-type 变量中，通过调用 __regex_build_aux:Nn 或 __regex_build_aux:NN 来区分。这些函数的第一个参数是 \c_false_bool，以指示不应在模式的开头隐式插入通配符：否则代码将继续查找输入流，直到匹配正则表达式。

```
\cs_new_protected:Npn \peek_regex:nTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:Nn \c_false_bool {#1} }
      { \__regex_peek_end: }
  }
\cs_new_protected:Npn \peek_regex:nT #1#2
  { \peek_regex:nTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex:nF #1 { \peek_regex:nTF {#1} { } }
\cs_new_protected:Npn \peek_regex:NTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:NN \c_false_bool #1 }
      { \__regex_peek_end: }
  }
\cs_new_protected:Npn \peek_regex:NT #1#2
  { \peek_regex:NTF #1 {#2} { } }
```

```
\cs_new_protected:Npn \peek_regex:NF #1 { \peek_regex:NTF {#1} { } }
\cs_new_protected:Npn \peek_regex_remove_once:nTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:Nn \c_false_bool {#1} }
      { \__regex_peek_remove_end:n {##1} }
  }
\cs_new_protected:Npn \peek_regex_remove_once:nT #1#2
  { \peek_regex_remove_once:nTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex_remove_once:nF #1
  { \peek_regex_remove_once:nTF {#1} { } }
\cs_new_protected:Npn \peek_regex_remove_once:NTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:NN \c_false_bool #1 }
      { \__regex_peek_remove_end:n {##1} }
  }
\cs_new_protected:Npn \peek_regex_remove_once:NT #1#2
  { \peek_regex_remove_once:NTF #1 {#2} { } }
\cs_new_protected:Npn \peek_regex_remove_once:NF #1
  { \peek_regex_remove_once:NTF #1 { } }
```

(*\peek_regex:nTF* 以及其它的定义结束。这些函数被记录在第??页。)

__regex_peek:nnTF
__regex_peek_aux:nnTF

将用户的真/假代码(加上 \group_end:)存储到两个记号列表中。然后使用 #1 构建自动机, 不进行子匹配跟踪, 目标是单次匹配。然后开始匹配, 设置一些变量, 就像任何正则表达式匹配一样, 比如 \regex_match:nnTF, 另外加上 \l__regex_input_tl, 用于跟踪所见记号, 以在最后重新插入它们。我们不使用 \tl_analysis_map_inline:nn 处理输入, 而是调用 \peek_analysis_map_inline:n 遍历输入流中的记号。由于 __regex_match_one_token:nnN 调用了 __regex_maplike_break:, 我们需要捕捉它并中断 \peek_analysis_map_inline:n 循环。

```
\cs_new_protected:Npn \__regex_peek:nnTF #1
  {
    \__regex_peek_aux:nnTF
      {
        \__regex_disable_submatches:
        #1
      }
  }
\cs_new_protected:Npn \__regex_peek_aux:nnTF #1#2#3#4
  {
    \group_begin:
```

```
      \tl_set:Nn \l__regex_peek_true_tl { \group_end: #3 }
      \tl_set:Nn \l__regex_peek_false_tl { \group_end: #4 }
      \__regex_single_match:
      #1
      \__regex_match_init:
      \tl_build_begin:N \l__regex_input_tl
      \__regex_match_once_init:
      \peek_analysis_map_inline:n
        {
          \tl_build_put_right:Nn \l__regex_input_tl
            { \__regex_input_item:n {##1} }
          \__regex_match_one_token:nnN {##1} {##2} ##3
          \use_none:nnn
          \prg_break_point:Nn \__regex_maplike_break:
            { \peek_analysis_map_break:n {#2} }
        }
  }
```

(*__regex_peek:nnTF* 和 *__regex_peek_aux:nnTF* 定义结束。)

__regex_peek_end:
__regex_peek_remove_end:n

一旦正则表达式匹配（或永久无法匹配），我们调用 __regex_peek_end: 或带有最后看到的记号作为参数的 __regex_peek_remove_end:n。对于 \peek_regex:nTF，我们通过调用 __regex_peek_reinsert:N 重新插入看到的记号，无论匹配的结果如何。对于 \peek_regex_remove_once:nTF，仅当匹配失败时，我们才重新插入看到的记号；否则，我们只需用一个展开重新插入记号 #1。更确切地说，#1 包含那些 o-展开和 e-展开为最后看到的记号的记号，例如对于控制序列，它是 \exp_not:N ⟨*cs*⟩。这意味着仅执行 \exp_after:wN \l__regex_peek_true_tl #1 可能是不安全的，因为会抑制 ⟨*cs*⟩ 的展开。

```
\cs_new_protected:Npn \__regex_peek_end:
  {
    \bool_if:NTF \g__regex_success_bool
      { \__regex_peek_reinsert:N \l__regex_peek_true_tl }
      { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
  }
\cs_new_protected:Npn \__regex_peek_remove_end:n #1
  {
    \bool_if:NTF \g__regex_success_bool
      { \exp_args:NNo \use:nn \l__regex_peek_true_tl {#1} }
      { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
  }
```

(*__regex_peek_end:* 和 *__regex_peek_remove_end:n* 定义结束。)

__regex_peek_reinsert:N
__regex_reinsert_item:n

插入真/假代码 #1，然后是找到的记号，它们存储在 \l__regex_input_tl 中。为此，通过 __regex_reinsert_item:n 循环遍历该记号列表，该循环展开 #1 一次以获取单个记号，并跳过它以展开后面的内容，使用适当的 \exp:w 和 \exp_end:。我们不能只在整个记号列表上使用 \use:e，因为结果可能不平衡，这将导致原语提前停止，或者让它继续超过我们想要的位置。

```
\cs_new_protected:Npn \__regex_peek_reinsert:N #1
  {
    \tl_build_end:N \l__regex_input_tl
    \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
    \exp_after:wN #1 \exp:w \l__regex_input_tl \exp_end:
  }
\cs_new_protected:Npn \__regex_reinsert_item:n #1
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \exp_end:
    \exp_after:wN \exp_after:wN
    #1
    \exp:w
  }
```

(*__regex_peek_reinsert:N* 和 *__regex_reinsert_item:n* 定义结束。)

\peek_regex_replace_once:nn
\peek_regex_replace_once:nn*TF*
\peek_regex_replace_once:Nn
\peek_regex_replace_once:Nn*TF*

与上述的 \peek_regex:nTF 类似。

```
\cs_new_protected:Npn \peek_regex_replace_once:nnTF #1
  { \__regex_peek_replace:nnTF { \__regex_build_aux:Nn \c_false_bool {#1} } }
\cs_new_protected:Npn \peek_regex_replace_once:nnT #1#2#3
  { \peek_regex_replace_once:nnTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \peek_regex_replace_once:nnF #1#2
  { \peek_regex_replace_once:nnTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex_replace_once:nn #1#2
  { \peek_regex_replace_once:nnTF {#1} {#2} { } { } }
\cs_new_protected:Npn \peek_regex_replace_once:NnTF #1
  { \__regex_peek_replace:nnTF { \__regex_build_aux:NN \c_false_bool #1 } }
\cs_new_protected:Npn \peek_regex_replace_once:NnT #1#2#3
  { \peek_regex_replace_once:NnTF #1 {#2} {#3} { } }
\cs_new_protected:Npn \peek_regex_replace_once:NnF #1#2
  { \peek_regex_replace_once:NnTF #1 {#2} { } }
\cs_new_protected:Npn \peek_regex_replace_once:Nn #1#2
  { \peek_regex_replace_once:NnTF #1 {#2} { } { } }
```

(*\peek_regex_replace_once:nnTF* 和 *\peek_regex_replace_once:NnTF* 定义结束。这些函数被记录在第??页。)

__regex_peek_replace:nnTF

与上述的 __regex_peek:nnTF 相同（用于上面的 \peek_regex:nTF），但没有禁用子匹配，并带有不同的结束。替换文本 #2 被存储，以便以后分析。

```
\cs_new_protected:Npn \__regex_peek_replace:nnTF #1#2
  {
    \tl_set:Nn \l__regex_replacement_tl {#2}
    \__regex_peek_aux:nnTF {#1} { \__regex_peek_replace_end: }
  }
```

（*__regex_peek_replace:nnTF 定义结束。*）

__regex_peek_replace_end:

如果匹配失败，__regex_peek_reinsert:N 将重新插入找到的记号。否则，完成使用 __regex_extract: 子匹配信息的存储，并将输入存储到 \toks。重新定义一些辅助命令，稍微更改其展开行为，如下面所述。使用 __regex_replacement:n 分析替换文本，该命令通常定义 __regex_replacement_do_one_match:n，以插入匹配尝试开始到匹配开始之间的记号，然后是替换文本。例如，\use:e 展开到尾随的 __regex_query_range:nn，成为一系列 __regex_reinsert_item:n {⟨*tokens*⟩}，其中 ⟨*tokens*⟩ o-展开为我们要插入的单个记号。在 e-展开后，\use:e 执行 \use:n，因此我们有 \exp_after:wN \l__regex_peek_true_tl \exp:w … \exp_end:。这被设置为获取 \l__regex_peek_true_tl，后跟替换的记号（可能不平衡）在输入流中。

```
\cs_new_protected:Npn \__regex_peek_replace_end:
  {
    \bool_if:NTF \g__regex_success_bool
      {
        \__regex_extract:
        \__regex_query_set_from_input_tl:
        \cs_set_eq:NN \__regex_replacement_put:n \__regex_peek_replacement_put:n
        \cs_set_eq:NN \__regex_replacement_put_submatch_aux:n
          \__regex_peek_replacement_put_submatch_aux:n
        \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
        \cs_set_eq:NN \__regex_replacement_exp_not:N \__regex_peek_replacement_token:n
        \cs_set_eq:NN \__regex_replacement_exp_not:V \__regex_peek_replacement_var:N
        \exp_args:No \__regex_replacement:n { \l__regex_replacement_tl }
        \use:e
          {
            \exp_not:n { \exp_after:wN \l__regex_peek_true_tl \exp:w }
            \__regex_replacement_do_one_match:n
              { \l__regex_zeroth_submatch_int }
            \__regex_query_range:nn
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
                  { \l__regex_zeroth_submatch_int }
```

```
            }
            { \l__regex_max_pos_int }
          \exp_end:
        }
    }
    { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
}
```

(*__regex_peek_replace_end:* 定义结束。)

__regex_query_set_from_input_tl:
__regex_query_set_item:n

输入被存储到 \l__regex_input_tl 中，作为连续的项 __regex_input_item:n {⟨*tokens*⟩}。将其存储到连续的 \toks 中。在两者之前和之后的空条目是否都有用，目前不太清楚。

```
\cs_new_protected:Npn \__regex_query_set_from_input_tl:
  {
    \tl_build_end:N \l__regex_input_tl
    \int_zero:N \l__regex_curr_pos_int
    \cs_set_eq:NN \__regex_input_item:n \__regex_query_set_item:n
    \__regex_query_set_item:n { }
    \l__regex_input_tl
    \__regex_query_set_item:n { }
    \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
  }
\cs_new_protected:Npn \__regex_query_set_item:n #1
  {
    \int_incr:N \l__regex_curr_pos_int
    \__regex_toks_set:Nn \l__regex_curr_pos_int { \__regex_input_item:n {#1} }
  }
```

(*__regex_query_set_from_input_tl:* 和 *__regex_query_set_item:n* 定义结束。)

__regex_peek_replacement_put:n

在构建替换函数 __regex_replacement_do_one_match:n 时，我们经常想要放入简单的材料，给定为 #1，它的 e-展开被 o-展开为单个记号。通常我们可以将记号添加到 \l__regex_build_tl 中，但对于 \peek_regex_replace_once:nnTF，我们最终想要执行一些奇怪的展开，基本上是使用 \exp_after:wN 跳过众多记号（我们不能像对于 \regex_replace_once:nnNTF 那样使用 e-展开，因为结果可以是不平衡的，因为我们插入它而不是存储它）。在 csname 中时，我们不进行任何这样的花招，因为 \cs:w … \cs_end: 执行我们需要的所有展开。

```
\cs_new_protected:Npn \__regex_peek_replacement_put:n #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
```

```
        { \exp_not:N \__regex_reinsert_item:n {#1} }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl {#1}
    \fi:
  }
```

(*__regex_peek_replacement_put:n* 定义结束。)

__regex_peek_replacement_token:n

当遇到 \exp:w 时，__regex_peek_replacement_token:n {⟨*token*⟩} 停止 \exp_end: 并执行 \exp_after:wN ⟨*token*⟩ \exp:w 以继续展开。

```
\cs_new_protected:Npn \__regex_peek_replacement_token:n #1
  { \exp_after:wN \exp_end: \exp_after:wN #1 \exp:w }
```

(*__regex_peek_replacement_token:n* 定义结束。)

__regex_peek_replacement_put_submatch_aux:n

在分析替换时,我们还必须插入查询中找到的子匹配。由于查询项 __regex_input_item:n {⟨*tokens*⟩} 仅在由 \exp:w … \exp_end: 包围时正确展开，且由于在 csname 中不存在这些展开控制（因为 \cs:w … \cs_end: 使它们在大多数情况下变得不必要），因此我们必须在这里手动放入 \exp:w 和 \exp_end:。

```
\cs_new_protected:Npn \__regex_peek_replacement_put_submatch_aux:n #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp:w \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } \exp_end: }
    \fi:
  }
```

(*__regex_peek_replacement_put_submatch_aux:n* 定义结束。)

__regex_peek_replacement_var:N

这用于在 csname 外部的 \u。它确保在展开变量 #1 并停止先前的 \exp:w 之前，使用 \exp:w 继续展开。

```
\cs_new_protected:Npn \__regex_peek_replacement_var:N #1
  {
    \exp_after:wN \exp_last_unbraced:NV
    \exp_after:wN \exp_end:
    \exp_after:wN #1
    \exp:w
  }
```

(*__regex_peek_replacement_var:N* 定义结束。)

9.8 消息

预解析阶段的消息。

```
\use:e
  {
    \msg_new:nnn { regex } { trailing-backslash }
      { Trailing~'\iow_char:N\\'~in~regex~or~replacement. }
    \msg_new:nnn { regex } { x-missing-rbrace }
      {
        Missing~brace~'\iow_char:N\}'~in~regex~
        '...\iow_char:N\\x\iow_char:N\{...##1'.
      }
    \msg_new:nnn { regex } { x-overflow }
      {
        Character~code~##1~too~large~in~
        \iow_char:N\\x\iow_char:N\{##2\iow_char:N\}~regex.
      }
  }
```

无效的量词。

```
\msg_new:nnnn { regex } { invalid-quantifier }
  { Braced~quantifier~'#1'~may~not~be~followed~by~'#2'. }
  {
    The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
    The~only~valid~quantifiers~are~'*',~'?',~'+',~'{<int>}',~
    '{<min>,}'~and~'{<min>,<max>}',~optionally~followed~by~'?'.
  }
```

缺少或多余的闭括号和括号对的消息，对于括号对的情况，进行了一些繁琐的单数/复数处理。

```
\msg_new:nnnn { regex } { missing-rbrack }
  { Missing~right~bracket~inserted~in~regular~expression. }
  {
    LaTeX~was~given~a~regular~expression~where~a~character~class~
    was~started~with~'[',~but~the~matching~']'~is~missing.
  }
\msg_new:nnnn { regex } { missing-rparen }
  {
    Missing~right~
    \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
    inserted~in~regular~expression.
  }
  {
```

```
    LaTeX~was~given~a~regular~expression~with~\int_eval:n {#1} ~
    more~left~parentheses~than~right~parentheses.
  }
\msg_new:nnnn { regex } { extra-rparen }
  { Extra~right~parenthesis~ignored~in~regular~expression. }
  {
    LaTeX~came~across~a~closing~parenthesis~when~no~submatch~group~
    was~open.~The~parenthesis~will~be~ignored.
  }
```

一些转义的字母数字不能在所有地方使用。

```
\msg_new:nnnn { regex } { bad-escape }
  {
    Invalid~escape~'\iow_char:N\\#1'~
    \__regex_if_in_cs:TF { within~a~control~sequence. }
      {
        \__regex_if_in_class:TF
          { in~a~character~class. }
          { following~a~category~test. }
      }
  }
  {
    The~escape~sequence~'\iow_char:N\\#1'~may~not~appear~
    \__regex_if_in_cs:TF
      {
        within~a~control~sequence~test~introduced~by~
        '\iow_char:N\\c\iow_char:N\{'.
      }
      {
        \__regex_if_in_class:TF
          { within~a~character~class~ }
          { following~a~category~test~such~as~'\iow_char:N\\cL'~ }
        because~it~does~not~match~exactly~one~character.
      }
  }
```

范围错误。

```
\msg_new:nnnn { regex } { range-missing-end }
  { Invalid~end-point~for~range~'#1-#2'~in~character~class. }
  {
    The~end-point~'#2'~of~the~range~'#1-#2'~may~not~serve~as~an~
    end-point~for~a~range:~alphanumeric~characters~should~not~be~
    escaped,~and~non-alphanumeric~characters~should~be~escaped.
```

```
  }
\msg_new:nnnn { regex } { range-backwards }
  { Range~'[#1-#2]'~out~of~order~in~character~class. }
  {
    In~ranges~of~characters~'[x-y]'~appearing~in~character~classes,~
    the~first~character~code~must~not~be~larger~than~the~second.~
    Here,~'#1'~has~character~code~\int_eval:n {`#1},~while~
    '#2'~has~character~code~\int_eval:n {`#2}.
  }
```

与 \c 和 \u 有关的错误。

```
\msg_new:nnnn { regex } { c-bad-mode }
  { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
  {
    The~'\iow_char:N\\c'~escape~cannot~be~used~within~
    a~control~sequence~test~'\iow_char:N\\c{...}'~
    nor~another~category~test.~
    To~combine~several~category~tests,~use~'\iow_char:N\\c[...]'.
  }
\msg_new:nnnn { regex } { c-C-invalid }
  { '\iow_char:N\\cC'~should~be~followed~by~'.'~or~'(',~not~'#1'. }
  {
    The~'\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
    control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
    It~only~makes~sense~to~follow~it~by~'.'~or~by~a~group.
  }
\msg_new:nnnn { regex } { cu-lbrace }
  { Left~braces~must~be~escaped~in~'\iow_char:N\\#1{...}'. }
  {
    Constructions~such~as~'\iow_char:N\\#1{...\iow_char:N\{...}'~are~
    not~allowed~and~should~be~replaced~by~
    '\iow_char:N\\#1{...\token_to_str:N\{...}'.
  }
\msg_new:nnnn { regex } { c-lparen-in-class }
  { Catcode~test~cannot~apply~to~group~in~character~class }
  {
    Construction~such~as~'\iow_char:N\\cL(abc)'~are~not~allowed~inside~a~
    class~'[...]'~because~classes~do~not~match~multiple~characters~at~once.
  }
\msg_new:nnnn { regex } { c-missing-rbrace }
  { Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape. }
  {
    LaTeX~was~given~a~regular~expression~where~a~
```

```
    '\iow_char:N\\c\iow_char:N\{...'~construction~was~not~ended~
    with~a~closing~brace~'\iow_char:N\}'.
  }
\msg_new:nnnn { regex } { c-missing-rbrack }
  { Missing~right~bracket~inserted~for~'\iow_char:N\\c'~escape. }
  {
    A~construction~'\iow_char:N\\c[...'~appears~in~a~
    regular~expression,~but~the~closing~']'~is~not~present.
  }
\msg_new:nnnn { regex } { c-missing-category }
  { Invalid~character~'#1'~following~'\iow_char:N\\c'~escape. }
  {
    In~regular~expressions,~the~'\iow_char:N\\c'~escape~sequence~
    may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
    capital~letter~representing~a~character~category,~namely~
    one~of~'ABCDELMOPSTU'.
  }
\msg_new:nnnn { regex } { c-trailing }
  { Trailing~category~code~escape~'\iow_char:N\\c'... }
  {
    A~regular~expression~ends~with~'\iow_char:N\\c'~followed~
    by~a~letter.~It~will~be~ignored.
  }
\msg_new:nnnn { regex } { u-missing-lbrace }
  { Missing~left~brace~following~'\iow_char:N\\u'~escape. }
  {
    The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
    a~brace~group~with~the~name~of~the~variable~to~use.
  }
\msg_new:nnnn { regex } { u-missing-rbrace }
  { Missing~right~brace~inserted~for~'\iow_char:N\\u'~escape. }
  {
    LaTeX~
    \str_if_eq:eeTF { } {#2}
      { reached~the~end~of~the~string~ }
      { encountered~an~escaped~alphanumeric~character '\iow_char:N\\#2'~ }
    when~parsing~the~argument~of~an~
    '\iow_char:N\\u\iow_char:N\{...\}'~escape.
  }
```

当遇到 POSIX 语法 `[:...:]` 的错误。

```
\msg_new:nnnn { regex } { posix-unsupported }
  { POSIX~collating~element~'[#1 ~ #1]'~not~supported. }
```

```
  {
    The~'[.foo.]'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
    in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
    Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
  }
\msg_new:nnnn { regex } { posix-unknown }
  { POSIX~class~'[:#1:]'~unknown. }
  {
    '[:#1:]'~is~not~among~the~known~POSIX~classes~
    '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
    '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
    '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
    '[:word:]',~and~'[:xdigit:]'.
  }
\msg_new:nnnn { regex } { posix-missing-close }
  { Missing~closing~':]'~for~POSIX~class. }
  { The~POSIX~syntax~'#1'~must~be~followed~by~':]',~not~'#2'. }
```

在各种情况下，l3regex 操作的结果可能会导致我们得到一个不平衡的记号列表，我们必须通过添加开始组或结束组字符记号来重新平衡。

```
\msg_new:nnnn { regex } { result-unbalanced }
  { Missing~brace~inserted~when~#1. }
  {
    LaTeX~was~asked~to~do~some~regular~expression~operation,~
    and~the~resulting~token~list~would~not~have~the~same~number~
    of~begin-group~and~end-group~tokens.~Braces~were~inserted:~
    #2~left,~#3~right.
  }
```

未知选项的错误信息。

```
\msg_new:nnnn { regex } { unknown-option }
  { Unknown~option~'#1'~for~regular~expressions. }
  {
    The~only~available~option~is~'case-insensitive',~toggled~by~
    '(?i)'~and~'(?-i)'.
  }
\msg_new:nnnn { regex } { special-group-unknown }
  { Unknown~special~group~'#1~...'~in~a~regular~expression. }
  {
    The~only~valid~constructions~starting~with~'(?'~are~
    '(?:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
  }
```

替换文本中的错误。

```
\msg_new:nnnn { regex } { replacement-c }
  { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
    or~a~brace~group,~not~by~'#1'.
  }
\msg_new:nnnn { regex } { replacement-u }
  { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
  {
    In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
    must~be~~followed~by~a~brace~group~holding~the~name~of~the~
    variable~to~use.
  }
\msg_new:nnnn { regex } { replacement-g }
  {
    Missing~brace~for~the~'\iow_char:N\\g'~construction~
    in~a~replacement~text.
  }
  {
    In~the~replacement~text~for~a~regular~expression~search,~
    submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
    or~'\\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
  }
\msg_new:nnnn { regex } { replacement-catcode-end }
  {
    Missing~character~for~the~'\iow_char:N\\c<category><character>'~
    construction~in~a~replacement~text.
  }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
    the~character~category.~Then,~a~character~must~follow.~LaTeX~
    reached~the~end~of~the~replacement~when~looking~for~that.
  }
\msg_new:nnnn { regex } { replacement-catcode-escaped }
  {
    Escaped~letter~or~digit~after~category~code~in~replacement~text.
  }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
```

```
    the~character~category.~Then,~a~character~must~follow,~not~
    '\iow_char:N\\#2'.
  }
\msg_new:nnnn { regex } { replacement-catcode-in-cs }
  {
    Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
    '\iow_char:N\\c\{...\}'~in~a~replacement~text.
  }
  {
    In~a~replacement~text,~the~category~codes~of~the~argument~of~
    '\iow_char:N\\c\{...\}'~are~ignored~when~building~the~control~
    sequence~name.
  }
\msg_new:nnnn { regex } { replacement-null-space }
  { TeX~cannot~build~a~space~token~with~character~code~0. }
  {
    You~asked~for~a~character~token~with~category~space,~
    and~character~code~0,~for~instance~through~
    '\iow_char:N\\cS\iow_char:N\\x00'.~
    This~specific~case~is~impossible~and~will~be~replaced~
    by~a~normal~space.
  }
\msg_new:nnnn { regex } { replacement-missing-rbrace }
  { Missing~right~brace~inserted~in~replacement~text. }
  {
    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
    missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
  }
\msg_new:nnnn { regex } { replacement-missing-rparen }
  { Missing~right~parenthesis~inserted~in~replacement~text. }
  {
    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
    missing~right~
    \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
  }
\msg_new:nnn { regex } { submatch-too-big }
  { Submatch~#1~used~but~regex~only~has~#2~group(s) }
```

一些转义的字母数字不能在所有地方使用。

```
\msg_new:nnnn { regex } { backwards-quantifier }
  { Quantifer~"{#1,#2}"~is~backwards. }
  { The~values~given~in~a~quantifier~must~be~in~order. }
```

用于用户命令，并在显示正则表达式时使用。

```
\msg_new:nnnn { regex } { case-odd }
  { #1~with~odd~number~of~items }
  {
    There~must~be~a~#2~part~for~each~regex:~
    found~odd~number~of~items~(#3)~in\\
    \iow_indent:n {#4}
  }
\msg_new:nnn { regex } { show }
  {
    >~Compiled~regex~
    \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
    #3
  }
\prop_gput:Nnn \g_msg_module_name_prop { regex } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { regex } { }
```

__regex_msg_repeated:nnN

这在技术上不是一条消息，但似乎与此相关。参数是：#1 是最小重复次数；#2 是允许的额外重复次数（−1 表示无限次），而 #3 则告诉我们关于惰性的信息。

```
\cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
  {
    \str_if_eq:eeF { #1 #2 } { 1 0 }
      {
        , ~ repeated ~
        \int_case:nnF {#2}
          {
            { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
            {  0 } { #1~times }
          }
          {
            between~#1~and~\int_eval:n {#1+#2}~times,~
            \bool_if:NTF #3 { lazy } { greedy }
          }
      }
  }
```

（*__regex_msg_repeated:nnN* 定义结束。）

9.9 用于追踪的代码

在 l3trial 宏包 l3trace 中有一个更完整的追踪实现。函数名有点不同，但可以合并。

__regex_trace_push:nnN
__regex_trace_pop:nnN
__regex_trace:nne

这里 #1 是模块名 (regex)，#2 通常是 1。如果模块的当前追踪级别小于 #2，则不显示任何内容，否则将 #3 写入终端。

```
\cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
  { \__regex_trace:nne {#1} {#2} { entering~ \token_to_str:N #3 } }
\cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
  { \__regex_trace:nne {#1} {#2} { leaving~ \token_to_str:N #3 } }
\cs_new_protected:Npn \__regex_trace:nne #1#2#3
  {
    \int_compare:nNnF
      { \int_use:c { g__regex_trace_#1_int } } < {#2}
      { \iow_term:e { Trace:~#3 } }
  }
```

(*__regex_trace_push:nnN*, *__regex_trace_pop:nnN*, 和 *__regex_trace:nne* 定义结束。)

\g__regex_trace_regex_int

当此变量为零时，不进行追踪。

```
\int_new:N \g__regex_trace_regex_int
```

(*\g__regex_trace_regex_int* 定义结束。)

__regex_trace_states:n

此函数列出 NFA 的所有状态的内容，存储在从 0 到 \l__regex_max_state_int (不包括) 的 \toks 中。

```
\cs_new_protected:Npn \__regex_trace_states:n #1
  {
    \int_step_inline:nnn
      \l__regex_min_state_int
      { \l__regex_max_state_int - 1 }
      {
        \__regex_trace:nne { regex } {#1}
          { \iow_char:N \\toks ##1 = { \__regex_toks_use:w ##1 } }
      }
  }
```

(*__regex_trace_states:n* 定义结束。)

```
⟨/package⟩
```

索引

斜体数字指向相应条目描述的页面，下划线数字指向定义的代码行，其它的都指向使用条目的页面。

B

C

E

F

G

group 命令:

I

if 命令:

int 命令:

M

msg 命令:

O

or 命令:

P

peek 命令:

prg 命令:

Q

R

S

T

U