l3file 模块

文件与 I/O 操作

LaTeX 项目组* 2024 年 1 月 04 日 发布

张泓知 2024 年 1 月 19 日 【译】

目 录

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

该模块提供用于处理外部文件的函数。其中一些函数适用于整个文件，具有前缀 \file_…，而其他一些用于逐行处理文件，具有前缀 \ior_…（读取）或 \iow_…（写入）。

需要记住的是，在读取外部文件时，TEX 尝试使用操作系统路径和 TEX 文件数据库中的条目来定位它们（大多数 TEX 系统使用此类数据库）。因此，TEX 的“当前路径”比其他程序的要宽泛一些。

对于期望一个 ⟨*file name*⟩ 参数的函数，此参数可以包含字面项和可展开内容，在完全展开时应为所需的文件名。活动字符（如 \l_char_active_seq 中声明的）不会被展开，允许直接在文件名中使用它们。引号标记（"）在文件名中不允许，因为它们被一些 TEX 原语内部保留使用。

文件名的开头和结尾会被修剪空格：这反映了一些文件系统不允许或在这些位置与空格不可预测地交互的事实。当没有给出扩展名时，这将仅从名称开头修剪空格。

1 输入输出流管理

由于 TEX 引擎的输入和输出流数量有限，在 LATEX3 中不支持程序员直接使用这些流。相反，维护一个内部的流池，这些流会根据需要被分配和释放给其他模块使用。因此，程序员应该在不再需要流时关闭它们，以释放它们供其他进程使用。

注意，I/O 操作是全局的：流都应该以全局名称声明并相应地处理。

\ior_new:N
\ior_new:c
\iow_new:N
\iow_new:c

New: 2011-09-26
Updated: 2011-12-27

\ior_new:N ⟨*stream*⟩
\iow_new:N ⟨*stream*⟩

全局保留 ⟨*stream*⟩ 的名称，根据需要用于读取或写入。直到使用相应的 \…_open:Nn 函数打开 ⟨*stream*⟩ 之前，它不会被打开。尝试使用未打开的 ⟨*stream*⟩ 是一个错误，并且 ⟨*stream*⟩ 将表现得像相应的 \c_term_…。

\ior_open:Nn
\ior_open:cn

Updated: 2012-02-10

\ior_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

使用 ⟨*stream*⟩ 作为文件访问的控制序列，打开 ⟨*file name*⟩ 以供读取。如果 ⟨*stream*⟩ 已经打开，则在新操作开始之前关闭它。立即可以访问 ⟨*stream*⟩，并将保留给 ⟨*file name*⟩ 直到给出 \ior_close:N 指令或 TEX 运行结束。如果找不到文件，将引发错误。

\ior_open:NnTF \ior_open:NnTF ⟨*stream*⟩ {⟨*file name*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

\ior_open:cnTF 使用 ⟨*stream*⟩ 作为文件访问的控制序列，打开 ⟨*file name*⟩ 以供读取。如果 ⟨*stream*⟩ 已经打开，则在新操作开始之前关闭它。立即可以访问 ⟨*stream*⟩，并将保留给 ⟨*file name*⟩ 直到给出 \ior_close:N 指令或 TEX 运行结束。然后将 ⟨*true code*⟩ 插入到输入流中。如果找不到文件，不会引发错误，而是将 ⟨*false code*⟩ 插入到输入流中。

New: 2013-01-12

\iow_open:Nn \iow_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

\iow_open:(NV|cn|cV) 使用 ⟨*stream*⟩ 作为文件访问的控制序列，打开 ⟨*file name*⟩ 以供写入。如果 ⟨*stream*⟩ 已经打开，则在新操作开始之前关闭它。立即可以访问 ⟨*stream*⟩，并将保留给 ⟨*file name*⟩ 直到给出 \iow_close:N 指令或 TEX 运行结束。打开文件以写入会清除文件中的任何现有内容（即写入不是追加的）。

Updated: 2012-02-09

\ior_shell_open:Nn \ior_shell_open:Nn ⟨*stream*⟩ {⟨*shell command*⟩}

New: 2019-05-08 使用 ⟨*stream*⟩ 作为访问控制序列，打开由 ⟨*shell command*⟩ 输出创建的伪文件以供读取。如果 ⟨*stream*⟩ 已经打开，则在新操作开始之前关闭它。立即可以访问 ⟨*stream*⟩，并将保留给 ⟨*shell command*⟩ 直到给出 \ior_close:N 指令或 TEX 运行结束。如果禁用了管道系统调用，将引发错误。

有关处理 ⟨*shell command*⟩ 的详细信息，请参见 \sys_get_shell:nnNTF。

\iow_shell_open:Nn \iow_shell_open:Nn ⟨*stream*⟩ {⟨*shell command*⟩}

New: 2023-05-25 使用 ⟨*stream*⟩ 作为访问控制序列，打开由 ⟨*shell command*⟩ 输出创建的伪文件以供写入。如果 ⟨*stream*⟩ 已经打开，则在新操作开始之前关闭它。立即可以访问 ⟨*stream*⟩，并将保留给 ⟨*shell command*⟩ 直到给出 \iow_close:N 指令或 TEX 运行结束。如果禁用了管道系统调用，将引发错误。

有关处理 ⟨*shell command*⟩ 的详细信息，请参见 \sys_get_shell:nnNTF。

\ior_close:N \ior_close:N ⟨*stream*⟩

\ior_close:c \iow_close:N ⟨*stream*⟩

\iow_close:N 关闭 ⟨*stream*⟩。在完成对流的使用时，应始终关闭它们，以确保它们保持可用于其他程序员。

\iow_close:c

Updated: 2012-07-31

\ior_show:N \ior_show:N ⟨*stream*⟩
\ior_show:c \ior_log:N ⟨*stream*⟩
\ior_log:N \iow_show:N ⟨*stream*⟩
\ior_log:c \iow_log:N ⟨*stream*⟩
\iow_show:N 显示（在终端或日志文件中）与（读取或写入）⟨*stream*⟩ 关联的文件名。
\iow_show:c
\iow_log:N
\iow_log:c

New: 2021-05-11

\ior_show_list: \ior_show_list:
\ior_log_list: \ior_log_list:
\iow_show_list: \iow_show_list:
\iow_log_list: \iow_log_list:

New: 2017-06-27

显示（在终端或日志文件中）与每个已打开（读取或写入）流关联的文件名列表。这用于跟踪问题。

1.1 从文件读取

在 expl3 中，从文件读取和从终端读取是分开的过程。函数 \ior_get:NN 和 \ior_str_get:NN 及其相应的分支版本设计用于处理文件。

\ior_get:NN
\ior_get:NN*TF*

\ior_get:NN ⟨*stream*⟩ ⟨*token list variable*⟩
\ior_get:NNTF ⟨*stream*⟩ ⟨*token list variable*⟩ ⟨*true code*⟩ ⟨*false code*⟩

New: 2012-06-24
Updated: 2019-03-23

该函数从文件输入 ⟨*stream*⟩ 读取一行或多行（直到找到相等数量的左右大括号），并将结果存储在局部的 ⟨*token list*⟩ 变量中。从 ⟨*stream*⟩ 读取的内容由 TEX 根据函数使用时的类别码和 \endlinechar 进行记号化。在正常设置下，不以注释字符 % 结尾的任何行都将换行符转换为空格，因此例如输入

```
a b  c
```

将得到一个记号列表 a␣b␣c␣。任何空白行都会被转换为记号 \par。因此，可以使用类似以下的测试跳过空白行：

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

还要注意，如果读取多行以匹配大括号，那么结果的记号列表可能包含 \par 记号。在非分支版本中，如果 ⟨*stream*⟩ 未打开，则将 ⟨*tl var*⟩ 设置为 \q_no_value。

TEXhackers note: 该受保护的宏是对 TEX 原始命令 \read 的换行。无论设置如何，TEX 都会在将字符按照当前类别码转换为记号之前将每行中的尾随空格和制表符（字符代码为 32 和 9）替换为换行符（字符代码为 \endlinechar，如果 \endlinechar 为负或太大则被省略）。使用默认设置，行首的空格也会被忽略。

\ior_str_get:NN `\ior_str_get:NN` ⟨*stream*⟩ ⟨*token list variable*⟩

\ior_str_get:NNTF `\ior_str_get:NNTF` ⟨*stream*⟩ ⟨*token list variable*⟩ ⟨*true code*⟩ ⟨*false code*⟩

New: 2016-12-04

Updated: 2019-03-23

该函数从文件输入 ⟨*stream*⟩ 读取一行并将结果存储在局部的 ⟨*token list*⟩ 变量中。材料从 ⟨*stream*⟩ 中作为带有类别码 12（其他）的一系列记号读取，空格字符除外，空格字符的类别码为 10（空格）。此过程会保留多个空格字符。它总是只读取一行，输入中的任何空白行都导致 ⟨*token list variable*⟩ 为空。与 `\ior_get:NN` 不同，行尾不接受任何特殊处理。因此，输入

```
a b  c
```

将得到一个记号列表 `a b  c`，其中字母 `a`、`b` 和 `c` 的类别码为 12。在非分支版本中，如果 ⟨*stream*⟩ 未打开，则将 ⟨*tl var*⟩ 设置为 `\q_no_value`。

TEXhackers note: 该受保护的宏是对 ε-TEX 原始命令 `\readline` 的换行。无论设置如何，TEX 都会删除每行中的尾随空格和制表符（字符代码为 32 和 9）。但是，该原始命令通常添加的行尾字符不包括在 `\ior_str_get:NN` 的结果中。

所有映射都在当前组级别完成，即下面讨论的 ⟨*function*⟩ 或 ⟨*code*⟩ 中进行的任何局部赋值在循环后仍然有效。

\ior_map_inline:Nn `\ior_map_inline:Nn` ⟨*stream*⟩ {⟨*inline function*⟩}

New: 2012-02-11

对通过调用 `\ior_get:NN` 获得的每组 ⟨*lines*⟩ 应用 ⟨*inline function*⟩，直到达到文件结尾。TEX 忽略从读取的文件中的任何尾随换行标记。⟨*inline function*⟩ 应包含接收 ⟨*line*⟩ 为 `#1` 的代码。

\ior_str_map_inline:Nn `\ior_str_map_inline:Nn` ⟨*stream*⟩ {⟨*inline function*⟩}

New: 2012-02-11

将 ⟨*inline function*⟩ 应用于 ⟨*stream*⟩ 中的每个 ⟨*line*⟩。从 ⟨*stream*⟩ 读取的材料作为一系列类别码为 12（其他）的记号进行，空格字符除外，它们的类别码为 10（空格）。⟨*inline function*⟩ 应包含接收 ⟨*line*⟩ 为 `#1` 的代码。请注意，TEX 会从每行输入中删除尾随的空格和制表符（字符代码为 32 和 9）。TEX 还会忽略从读取的文件中的任何尾随换行标记。

\ior_map_variable:NNn `\ior_map_variable:NNn` ⟨*stream*⟩ ⟨*tl var*⟩ {⟨*code*⟩}

New: 2019-01-13

对通过调用 `\ior_get:NN` 获得的每组 ⟨*lines*⟩，将 ⟨*lines*⟩ 存储在 ⟨*tl var*⟩ 中，然后应用 ⟨*code*⟩。⟨*code*⟩ 通常会使用 ⟨*variable*⟩，但不强制要求。对 ⟨*variable*⟩ 的赋值是局部的。循环后，它的值为最后一组 ⟨*lines*⟩，如果 ⟨*stream*⟩ 为空，则为其原始值。TEX 忽略从读取的文件中的任何尾随换行标记。这个函数通常比 `\ior_map_inline:Nn` 更快。

\ior_str_map_variable:NNn `\ior_str_map_variable:NNn ⟨stream⟩ ⟨variable⟩ {⟨code⟩}`

New: 2019-01-13

对 ⟨*stream*⟩ 中的每个 ⟨*line*⟩，将 ⟨*line*⟩ 存储在 ⟨*variable*⟩ 中，然后应用 ⟨*code*⟩。从 ⟨*stream*⟩ 中读取的材料作为一系列类别码为 12（其他）的记号进行，空格字符除外，它们的类别码为 10（空格）。⟨*code*⟩ 通常会使用 ⟨*variable*⟩，但不强制要求。对 ⟨*variable*⟩ 的赋值是局部的。循环后，它的值为最后一个 ⟨*line*⟩，如果 ⟨*stream*⟩ 为空，则为其原始值。请注意，TEX 会从每行输入中删除尾随的空格和制表符（字符代码为 32 和 9）。TEX 还会忽略从读取的文件中的任何尾随换行标记。这个函数通常比 `\ior_str_map_inline:Nn` 更快。

\ior_map_break: `\ior_map_break:`

New: 2012-06-29

用于在处理完 ⟨*stream*⟩ 的所有行之前终止 `\ior_map_…` 函数。这通常发生在条件语句内部，例如

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break: }
      {
        % Do something useful
      }
  }
```

在 `\ior_map_…` 场景之外使用会导致低级别的 TEX 错误。

TEXhackers note: 当映射被中断时，可能会在从输入流中取出更多项目之前插入额外的记号。这取决于映射函数的设计。

\ior_map_break:n `\ior_map_break:n {⟨code⟩}`

New: 2012-06-29 用于在处理完 ⟨*stream*⟩ 的所有行之前终止 `\ior_map_…` 函数，在映射结束后插入 ⟨*code*⟩。这通常发生在条件语句内部，例如

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

在 `\ior_map_…` 场景之外使用会导致低级别的 TEX 错误。

TEXhackers note: 当映射被中断时，可能会在插入 ⟨*code*⟩ 之前插入额外的记号。这取决于映射函数的设计。

\ior_if_eof_p:N ⋆ `\ior_if_eof_p:N ⟨stream⟩`

\ior_if_eof:NTF ⋆ `\ior_if_eof:NTF ⟨stream⟩ {⟨true code⟩} {⟨false code⟩}`

Updated: 2012-02-10 测试在读取操作期间是否已经到达文件 ⟨*stream*⟩ 的末尾。如果 ⟨*stream*⟩ 未打开，测试还将返回 `true` 值。

1.2 从终端读取

\ior_get_term:nN `\ior_get_term:nN ⟨prompt⟩ ⟨token list variable⟩`

\ior_str_get_term:nN

New: 2019-03-23 该函数从终端读取一行或多行（直到找到相等数量的左右大括号），并将结果存储在 ⟨*token list*⟩ 变量中。记号化过程与 `\ior_get:NN` 或 `\ior_str_get:NN` 中描述的相同。当 ⟨*prompt*⟩ 为空时，TEX 将等待输入，不提供任何其他指示：通常，程序员将使用 `\iow_term:n` 等提供适当文本。如果给定了 ⟨*prompt*⟩，它将出现在终端上，后面跟着一个 `=`，例如

```
prompt=
```

1.3 写入文件

\iow_now:Nn
\iow_now:(NV|Ne|cn|cV|ce)
Updated: 2012-06-05

\iow_now:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

此函数立即将 ⟨*tokens*⟩ 写入指定的 ⟨*stream*⟩（即在 \iow_now:Nn 扩展时调用写操作）。

\iow_log:n
\iow_log:e

\iow_log:n {⟨*tokens*⟩}

该函数将给定的 ⟨*tokens*⟩ 立即写入日志（transcript）文件：这是 \iow_now:Nn 的专用版本。

\iow_term:n
\iow_term:e

\iow_term:n {⟨*tokens*⟩}

该函数将给定的 ⟨*tokens*⟩ 立即写入终端文件：这是 \iow_now:Nn 的专用版本。

\iow_shipout:Nn
\iow_shipout:(Ne|cn|ce)

\iow_shipout:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

此函数在当前页面最终确定时（即在排版输出时）将 ⟨*tokens*⟩ 写入指定的 ⟨*stream*⟩。e-type 变体在使用函数时扩展 ⟨*tokens*⟩，但在将生成的记号写入 ⟨*stream*⟩ 时不扩展（*cf.* \iow_shipout_e:Nn）。

TEXhackers note: 使用 expl3 与除 LATEX 之外的格式时，使用 \iow_newline: 插入的换行符或使用 \iow_wrap:nnnN 插入的换行代码在 \iow_shipout:Nn 的参数中不会被 TEX 识别。这可能导致插入额外的不需要的换行。

\iow_shipout_e:Nn
\iow_shipout_e:(Ne|cn|ce)
Updated: 2023-09-17

\iow_shipout_e:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

此函数在当前页面最终确定时（即在排版输出时）将 ⟨*tokens*⟩ 写入指定的 ⟨*stream*⟩。⟨*tokens*⟩ 在写入时扩展，以及在使用函数时的任何扩展。这使这些函数适用于包含在页面构建过程中最终确定的材料（例如页码整数）。

TEXhackers note: 这是对 TEX 原始命令 \write 的换行。使用 expl3 与除 LATEX 之外的格式时，使用 \iow_newline: 插入的换行符或使用 \iow_wrap:nnnN 插入的换行代码在 \iow_shipout:Nn 的参数中不会被 TEX 识别。这可能导致插入额外的不需要的换行。

\iow_char:N ⋆ `\iow_char:N \⟨char⟩`

将 ⟨*char*⟩ 插入输出流中。在消息中尝试写入难以处理的字符（例如 `%`，`{`，`}` 等）时非常有用，例如：

```
\iow_now:Ne \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

如果在不扩展的情况下进行写入（例如在 `\iow_now:Nn` 的第二个参数中），该函数将不起作用。

\iow_newline: ⋆ `\iow_newline:`

用于在写入文件时在 ⟨*tokens*⟩ 中添加新行的函数。如果在不扩展的情况下进行写入（例如在 `\iow_now:Nn` 的第二个参数中），该函数将不起作用。

TEXhackers note: 使用 expl3 与除 LaTeX 之外的格式时，由 `\iow_newline:` 插入的字符在 TEX 中不被识别，这可能导致插入额外的不需要的换行。这个问题只影响 `\iow_shipout:Nn`、`\iow_shipout_e:Nn` 和原始操作的直接使用。

1.4 让输出的文字换行

\iow_wrap:nnnN
\iow_wrap:nenN
New: 2012-06-28
Updated: 2017-12-04

\iow_wrap:nnnN {⟨*text*⟩} {⟨*run-on text*⟩} {⟨*set up*⟩} ⟨*function*⟩

该函数将 ⟨*text*⟩ 换行到每行固定的字符数中。在每行开始换行的地方，插入 ⟨*run-on text*⟩。所针对的行字符计数是 \l_iow_line_count_int 的值减去除第一行外所有行的 ⟨*run-on text*⟩ 字符数，对于第一行，目标字符数只是 \l_iow_line_count_int，因为没有 ⟨*run-on text*⟩。函数会详尽扩展 ⟨*text*⟩ 和 ⟨*run-on text*⟩，进行以下替换：

- \\ 或 \iow_newline：可以用于强制换行，
- \␣ 可以用于表示强制空格（例如在控制序列之后），
- \#、\%、\{、\}、\~ 可以用于表示相应的字符，
- \iow_wrap_allow_break：可以用于允许换行而不插入空格，
- \iow_indent:n 可以用于缩进 ⟨*text*⟩ 的一部分（而不是 ⟨*run-on text*⟩）。

通过使用 ⟨*set up*⟩，可以向换行添加其他功能，该函数在换行发生之前执行：这可能包括覆盖列出的替换。

⟨*text*⟩ 中不希望在换行时扩展的可扩展材料应使用 \token_to_str:N、\tl_to_str:n、\tl_to_str:N 等转换为字符串。

换行操作的结果作为带大括号的参数传递给 ⟨*function*⟩，通常是换行写操作的换行器。\iow_wrap:nnnN 的输出（即传递给 ⟨*function*⟩ 的参数）由类别代码为 “other”（类别代码为 12）的字符组成，空格除外，其类别代码为 “space”（类别代码为 10）。这意味着输出在写入文件时不会进一步扩展。

TEXhackers note: 在内部，\iow_wrap:nnnN 对 ⟨*text*⟩ 执行 e-type 扩展以扩展它。这是以 \exp_not:N 或 \exp_not:n 可以用于防止材料扩展的方式完成的。但是，与转换为字符串相比，这在概念上不够清晰，因此转换为字符串是处理 ⟨*text*⟩ 中可扩展材料的受支持方法。

\iow_wrap_allow_break:
New: 2023-04-25

\iow_wrap_allow_break:

在 \iow_wrap:nnnN 的第一个参数中（例如在消息中）插入一个允许换行的断点。如果没有发生换行，此函数对输出不添加任何内容。

\iow_indent:n
New: 2011-09-21

\iow_indent:n {⟨*text*⟩}

在 \iow_wrap:nnnN 的第一个参数中（例如在消息中），通过四个空格缩进 ⟨*text*⟩。此函数不会导致换行，并且仅影响从 ⟨*text*⟩ 范围内开始的行。如果希望缩进的 ⟨*text*⟩ 应出现在与周围文本不同的行上，请使用 \\ 强制换行。

\l_iow_line_count_int 由 \iow_wrap:nnnN 函数写入的行中的最大字符数。此值取决于正在使用的 TEX 系统：标准值为 78，通常适用于未修改的 TEX Live 和 MiKTEX 系统。

New: 2012-06-24

1.5 输入输出流常量和变量

\g_tmpa_ior \g_tmpb_ior 全局使用的临时输入流。这些从不被内核代码使用，因此对于任何由 LATEX3 定义的函数都是安全的。但是，它们可能会被其他非内核代码覆盖，因此只应用于短期存储。

New: 2017-12-11

\c_log_iow \c_term_iow 常量输出流，用于写入日志和终端（以及日志）。

\g_tmpa_iow \g_tmpb_iow 全局使用的临时输出流。这些从不被内核代码使用，因此对于任何由 LATEX3 定义的函数都是安全的。但是，它们可能会被其他非内核代码覆盖，因此只应用于短期存储。

New: 2017-12-11

1.6 原始条件语句

\if_eof:w ⋆ \if_eof:w ⟨*stream*⟩
⟨*true code*⟩
\else:
⟨*false code*⟩
\fi:
检查 ⟨*stream*⟩ 是否返回 “end of file”，对于不存在的文件为真。\else: 分支是可选的。

TEXhackers note: 这是 TEX 原语 \ifeof。

2 文件操作

2.1 基本文件操作

\g_file_curr_dir_str
\g_file_curr_name_str
\g_file_curr_ext_str

New: 2017-06-21

包含当前文件的目录、名称和扩展名。如果文件是在没有显式路径的情况下加载的（即它在 TEX 搜索路径中），则目录为空，并且除了它与根目录完全相等的情况外，不以 / 结尾。⟨*name*⟩ 和 ⟨*ext*⟩ 部分共同组成文件名，因此 ⟨*name*⟩ 部分可以被视为当前文件的“作业名称”。

请注意，TEX 不提供有关主文件（顶层文件）的 ⟨*dir*⟩ 和 ⟨*ext*⟩ 部分的信息，并且此文件的 ⟨*dir*⟩ 和 ⟨*ext*⟩ 组件始终为空。此外，这里的 ⟨*name*⟩ 将等于 `\c_sys_jobname_str`，这可能与实际文件名不同（如果使用 `--jobname` 设置）。

\l_file_search_path_seq

New: 2017-06-18

Updated: 2023-06-15

每个条目都是在寻找文件时应搜索的目录的路径。每个路径可以是相对的或绝对的，而且无需包括尾随的斜杠。空格无需引用。

TEXhackers note: 在 LATEX 2ε 中作为包工作时，expl3 将自动将当前 `\input@path` 添加到 `\l_file_search_path_seq` 的值集中。

\file_if_exist_p:n ⋆
\file_if_exist_p:V ⋆
\file_if_exist:nTF ⋆
\file_if_exist:VTF ⋆

Updated: 2023-09-18

`\file_if_exist_p:n {`⟨*file name*⟩`}`
`\file_if_exist:nTF {`⟨*file name*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

展开 `\file name` 的参数以得到字符串，然后使用当前 TEX 搜索路径和由 `\l_file_search_path_seq` 控制的其他路径搜索此字符串。

2.2 关于文件和文件内容的信息

本节中的函数返回关于文件的信息，作为 expl3 的 `str` 数据，但是，非可展开的函数如果请求的文件未找到，则将其返回值设置为 `\q_no_value`。因此，对文件名、哈希值、大小等的比较应使用 `\str_if_eq:nnTF` 而不是 `\tl_if_eq:nnTF` 等。

\file_hex_dump:n ☆
\file_hex_dump:V ☆
\file_hex_dump:nnn ☆
\file_hex_dump:Vnn ☆

New: 2019-11-19

\file_hex_dump:n {⟨*file name*⟩}
\file_hex_dump:nnn {⟨*file name*⟩} {⟨*start index*⟩} {⟨*end index*⟩}

使用当前 TEX 搜索路径和由 \l_file_search_path_seq 控制的其他路径搜索 ⟨*file name*⟩，然后扩展为输入流中文件内容的十六进制转储。文件以字节形式读取，这意味着与文本文件中使用的行结束符相比，将有不同的结果。同一文件将在不同引擎之间产生相同的结果：使用的算法在所有情况下都是相同的。当未找到文件时，扩展的结果为空。对于 {⟨*start index*⟩} 和 {⟨*end index*⟩} 的工作原理与 \str_range:nnn 中描述的相同。

\file_get_hex_dump:nN
\file_get_hex_dump:VN
\file_get_hex_dump:nNTF
\file_get_hex_dump:VNTF
\file_get_hex_dump:nnnN
\file_get_hex_dump:VnnN
\file_get_hex_dump:nnnNTF
\file_get_hex_dump:VnnNTF

New: 2019-11-19

\file_get_hex_dump:nN {⟨*file name*⟩} ⟨*tl var*⟩
\file_get_hex_dump:nnnN {⟨*file name*⟩} {⟨*start index*⟩} {⟨*end index*⟩} ⟨*tl var*⟩

将 ⟨*tl var*⟩ 设置为应用于 ⟨*file*⟩ 的 \file_hex_dump:n/\file_hex_dump:nnn 的结果。如果未找到文件，⟨*tl var*⟩ 将设置为 \q_no_value。

\file_mdfive_hash:n ☆
\file_mdfive_hash:V ☆

New: 2019-09-03

\file_mdfive_hash:n {⟨*file name*⟩}

使用当前 TEX 搜索路径和由 \l_file_search_path_seq 控制的其他路径搜索 ⟨*file name*⟩，然后扩展为输入流中从文件内容生成的 MD5 摘要。文件以字节形式读取，这意味着与文本文件中使用的行结束符相比，将有不同的结果。同一文件将在不同引擎之间产生相同的结果：使用的算法在所有情况下都是相同的。当未找到文件时，扩展的结果为空。

\file_get_mdfive_hash:nN
\file_get_mdfive_hash:VN
\file_get_mdfive_hash:nNTF
\file_get_mdfive_hash:VNTF

New: 2017-07-11
Updated: 2019-02-16

\file_get_mdfive_hash:nN {⟨*file name*⟩} ⟨*tl var*⟩

将 ⟨*tl var*⟩ 设置为应用于 ⟨*file*⟩ 的 \file_mdfive_hash:n 的结果。如果未找到文件，⟨*tl var*⟩ 将设置为 \q_no_value。

\file_size:n ☆
\file_size:V ☆

New: 2019-09-03

\file_size:n {⟨*file name*⟩}

使用当前 TEX 搜索路径和由 \l_file_search_path_seq 控制的其他路径搜索 ⟨*file name*⟩，然后扩展为输入流中文件的大小（以字节为单位）。当未找到文件时，扩展的结果为空。

\file_get_size:nN
\file_get_size:VN
\file_get_size:nNTF
\file_get_size:VNTF

New: 2017-07-09

Updated: 2019-02-16

\file_get_size:nN {⟨file name⟩} ⟨tl var⟩

将 ⟨tl var⟩ 设置为应用于 ⟨file⟩ 的 \file_size:n 的结果。如果未找到文件，⟨tl var⟩ 将设置为 \q_no_value。在较旧版本的 XeTeX 中不可用。

\file_timestamp:n ☆
\file_timestamp:V ☆

New: 2019-09-03

\file_timestamp:n {⟨file name⟩}

使用当前 TeX 搜索路径和由 \l_file_search_path_seq 控制的其他路径搜索 ⟨file name⟩,然后扩展为输入流中文件的修改时间戳。时间戳的格式为 D:⟨year⟩⟨month⟩⟨day⟩⟨hour⟩⟨minut
后者可以是 Z（UTC）或 ⟨plus-minus⟩⟨hours⟩'⟨minutes⟩'。当未找到文件时，扩展的结果为空。在较旧版本的 XeTeX 中不可用。

\file_get_timestamp:nN
\file_get_timestamp:VN
\file_get_timestamp:nNTF
\file_get_timestamp:VNTF

New: 2017-07-09

Updated: 2019-02-16

\file_get_timestamp:nN {⟨file name⟩} ⟨tl var⟩

将 ⟨tl var⟩ 设置为应用于 ⟨file⟩ 的 \file_timestamp:n 的结果。如果未找到文件，⟨tl var⟩ 将设置为 \q_no_value。在较旧版本的 XeTeX 中不可用。

\file_compare_timestamp_p:nNn ⋆
\file_compare_timestamp_p:(nNV|VNn|VNV) ⋆
\file_compare_timestamp:nNnTF ⋆
\file_compare_timestamp:(nNV|VNn|VNV)TF ⋆

New: 2019-05-13

Updated: 2019-09-20

\file_compare_timestamp_p:nNn {⟨file-1⟩} ⟨comparator⟩ {⟨file-2⟩}
\file_compare_timestamp:nNnTF {⟨file-1⟩} ⟨comparator⟩ {⟨file-2⟩} {⟨true code⟩} {⟨false code⟩}

比较两个 ⟨文件⟩ 的时间戳，如 ⟨comparator⟩ 所示，并根据需要插入 ⟨true code⟩ 或 ⟨false code⟩。未找到的文件将被视为任何找到的文件都更旧。这允许例如下面的构造：

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
  {
    % Code to regenerate derived file
  }
```

在派生文件完全不存在时工作。两个不存在文件的时间戳被视为不同。在较旧版本的 XeTeX 中不可用。

\file_get_full_name:nN
\file_get_full_name:VN
\file_get_full_name:nNTF
\file_get_full_name:VNTF

Updated: 2019-02-16

\file_get_full_name:nN {⟨*file name*⟩} ⟨*tl*⟩
\file_get_full_name:nNTF {⟨*file name*⟩} ⟨*tl*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

在详细说明了 \file_if_exist:nTF 的路径中搜索 ⟨*file name*⟩，如果找到，将 ⟨*tl var*⟩ 设置为文件的完全限定名，即路径和文件名。当给定的 ⟨*file name*⟩ 没有扩展名但找到的文件有扩展名 .tex 时，这包括扩展名 .tex。在非分支版本中，如果文件不存在，⟨*tl var*⟩ 将设置为 \q_no_value。

\file_full_name:n ☆
\file_full_name:V ☆

New: 2019-09-03

\file_full_name:n {⟨*file name*⟩}

在详细说明了 \file_if_exist:nTF 的路径中搜索 ⟨*file name*⟩，如果找到，将文件的完全限定名，即路径和文件名，留在输入流中。当在路径上未找到文件时，扩展为空。

\file_parse_full_name:nNNN
\file_parse_full_name:VNNN

New: 2017-06-23
Updated: 2020-06-24

\file_parse_full_name:nNNN {⟨*full name*⟩} ⟨*dir*⟩ ⟨*name*⟩ ⟨*ext*⟩

解析 ⟨*full name*⟩ 并将其拆分为三个部分，通过设置相应的本地字符串变量返回每个部分：

- ⟨*dir*⟩：⟨*file path*⟩ 中到最后一个 / （路径分隔符）的所有内容。与系统的 PATH 变量和相关函数一样，⟨*dir*⟩ 不包括末尾的 /，除非它指向根目录。如果没有路径（只有文件名），⟨*dir*⟩ 为空。
- ⟨*name*⟩：从最后一个 / 后的内容到最后一个 .，其中这两个字符都是可选的。如果 ⟨*full name*⟩ 仅包含目录名，则 ⟨*name*⟩ 为空。
- ⟨*ext*⟩：最后一个 . 后的所有内容（包括点）。如果在最后一个 / 后没有 .，⟨*ext*⟩ 为空。

在解析之前，⟨*full name*⟩ 会扩展，直到只剩下不可展开的记号，但活动字符也不会扩展。引号（"）在文件名中无效，并从输入中丢弃。

\file_parse_full_name:n ⋆
\file_parse_full_name:V ⋆

New: 2020-06-24

\file_parse_full_name:n {⟨*full name*⟩}

解析 ⟨*full name*⟩，如 \file_parse_full_name:nNNN 中所述，并在输入流中留下 ⟨*dir*⟩、⟨*name*⟩ 和 ⟨*ext*⟩，每个都在一对大括号中。

\file_parse_full_name_apply:nN ⋆
\file_parse_full_name_apply:VN ⋆

New: 2020-06-24

\file_parse_full_name_apply:nN {⟨*full name*⟩} ⟨*function*⟩

解析 ⟨*full name*⟩，如 \file_parse_full_name:nNNN 中所述，并将 ⟨*dir*⟩、⟨*name*⟩ 和 ⟨*ext*⟩ 作为参数传递给 ⟨*function*⟩，每个作为一个 n-类型的参数，按此顺序。

2.3 访问文件内容

\file_get:nnN
\file_get:VnN
\file_get:nnNTF
\file_get:VnNTF

New: 2019-01-16
Updated: 2019-02-16

```
\file_get:nnN {⟨file name⟩} {⟨setup⟩} ⟨tl⟩
\file_get:nnNTF {⟨file name⟩} {⟨setup⟩} ⟨tl⟩ {⟨true code⟩} {⟨false code⟩}
```

将 ⟨*tl*⟩ 定义为 ⟨*file name*⟩ 的内容。可以通过 ⟨*setup*⟩ 参数设置适当的类别码。如果找不到文件，非分支版本将 ⟨*tl*⟩ 设置为 `\q_no_value`。分支版本在分配给 ⟨*tl*⟩ 后，如果找到文件，运行 ⟨*true code*⟩，否则运行 ⟨*false code*⟩。文件内容将使用当前类别码制度进行记号化。

\file_input:n
\file_input:V

Updated: 2017-06-26

```
\file_input:n {⟨file name⟩}
```

在详细说明了 `\file_if_exist:nTF` 的路径中搜索 ⟨*file name*⟩，如果找到，将文件作为附加的 LaTeX 源读入。此函数记录所有已读取的文件用于信息，并更新文件名堆栈。如果找不到文件，将引发错误。

\file_input_raw:n ⋆
\file_input_raw:V ⋆

New: 2023-05-18

```
\file_input_raw:n {⟨file name⟩}
```

在详细说明了 `\file_if_exist:nTF` 的路径中搜索 ⟨*file name*⟩，如果找到，将文件作为附加的 TeX 源读入。不跟踪有关文件的任何数据。如果找不到文件，不采取任何措施。

TeXhackers note: 此函数仅用于必须纯展开读取文件的上下文，例如在 `\halign` 中的表格单元格的开头。

\file_if_exist_input:n
\file_if_exist_input:V
\file_if_exist_input:nF
\file_if_exist_input:VF

New: 2014-07-02

```
\file_if_exist_input:n {⟨file name⟩}
\file_if_exist_input:nF {⟨file name⟩} {⟨false code⟩}
```

在详细说明了当前 TeX 搜索路径和 `\l_file_search_path_seq` 中包含的其他路径中搜索 ⟨*file name*⟩。如果找到，则如 `\file_input:n` 中所述将文件作为附加的 LaTeX 源读入，否则插入 ⟨*false code*⟩。请注意，与 `\file_input:n` 不同，这些函数如果找不到文件不会引发错误。

\file_input_stop:

New: 2017-07-07

```
\file_input_stop:
```

结束由 `\file_input:n` 或类似函数开始的文件读取，直到达到文件末尾为止。如果由于错误而终止文件读取，应优先使用 `\msg_critical:nn(nn)`。

TeXhackers note: 此函数必须单独使用在一行上：TeX 逐行读取文件，因此仍将读取当前行中的任何其他记号。

如果该函数隐藏在另一个函数中（这将是正常情况），这也是真的，即在源文件中同一行上的所有记号仍将被处理。将其定义放在一个行上并不起作用，因为它是在使用它的行上计算的！

\file_show_list: \file_show_list:

\file_log_list: \file_log_list:

这些函数列出由 LaTeX 2ε 命令加载的所有文件，这些命令填充了 \@filelist，或者由 \file_input:n 加载的文件。虽然 \file_show_list: 在终端显示列表，\file_log_list: 仅将其输出到日志文件中。

3 l3file 代码实现

The following test files are used for this code: m3file001.

1 ⟨*package⟩

3.1 输入操作

2 ⟨@@=ior⟩

3.1.1 变量和常量

\l__ior_internal_tl 用作短期临时变量。

3 \tl_new:N \l__ior_internal_tl

(*\l__ior_internal_tl* 定义结束。)

\c__ior_term_ior 从终端（带有提示符）读取时使用正数但不存在的流号。与写入不同，没有从日志读取的概念。

4 \int_const:Nn \c__ior_term_ior { 16 }

(*\c__ior_term_ior* 定义结束。)

\g__ior_streams_seq 用作堆栈的当前可用输入流列表。

5 \seq_new:N \g__ior_streams_seq

(*\g__ior_streams_seq* 定义结束。)

\l__ior_stream_tl 用于从堆栈中恢复原始流号。

6 \tl_new:N \l__ior_stream_tl

(*\l__ior_stream_tl* 定义结束。)

\g__ior_streams_prop 跟踪与每个流关联的文件的名称的属性列表。要获取包模式下保留的流的正确数量，需要查询底层机制。对于 LaTeX 2ε 和 plain TeX，此数据存储在 \count16 中：如果加载了 etex 宏包，我们需要减去 1，因为该寄存器保存要使用的下一个流的编号。在 ConTeXt 中，我们需要查看 \count38，但没有减法：与原始的 plain TeX/LaTeX 2ε 机制一样，它保存了分配的最后一个流的值。

```
\prop_new:N \g__ior_streams_prop
\int_step_inline:nnn
  { 0 }
  {
    \cs_if_exist:NTF \contextversion
      { \tex_count:D 38 ~ }
      {
        \tex_count:D 16 ~ %
        \cs_if_exist:NT \loccount { - 1 }
      }
  }
  {
    \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved~by~format }
  }
```

(*\g__ior_streams_prop* 定义结束。)

3.1.2 流管理

\ior_new:N 通过将名称定义为使用终端来保留新流。

\ior_new:c

```
\cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c__ior_term_ior }
\cs_generate_variant:Nn \ior_new:N { c }
```

(*\ior_new:N* 定义结束。这个函数被记录在第3页。)

\g_tmpa_ior 常用的临时空间。

\g_tmpb_ior

```
\ior_new:N \g_tmpa_ior
\ior_new:N \g_tmpb_ior
```

(*\g_tmpa_ior* 和 *\g_tmpb_ior* 定义结束。这些变量被记录在第13页。)

\ior_open:Nn 使用条件版本，如果找不到文件，则引发错误。

\ior_open:cn

```
\cs_new_protected:Npn \ior_open:Nn #1#2
  { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
\cs_generate_variant:Nn \ior_open:Nn { c }
```

(*\ior_open:Nn* 定义结束。这个函数被记录在第3页。)

\l__ior_file_name_tl 数据存储。

```
\tl_new:N \l__ior_file_name_tl
```

(*\l__ior_file_name_tl 定义结束。*)

\ior_open:NnTF
\ior_open:cnTF 辅助宏在 TEX、LATEX 2ε和 LATEX3 路径中搜索文件，然后将找到的文件传递给处理流的底层函数。当文件未找到时，full_name为空。

```
\prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
  {
    \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
      {
        \__kernel_ior_open:No #1 \l__ior_file_name_tl
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }
```

(*\ior_open:NnTF 定义结束。这个函数被记录在第4页。*)

__ior_new:N 流需要在\newread之前进行预留才能由 ior 管理。为防止 ior 受到\newread重新定义的影响，此宏被保存在此处，使用私有名称。复杂的代码确保__ior_new:N不是\outer，尽管 plain TEX 的\newread是\outer的。对于ConTEXt，我们必须处理\newread的重新定义，它实际上检查定义之前的情况。

```
\exp_args:NNf \cs_new_protected:Npn \__ior_new:N
  { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
\cs_if_exist:NT \contextversion
  {
    \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
    \cs_gset_protected:Npn \__ior_new:N #1
      {
        \cs_undefine:N #1
        \__ior_new_aux:N #1
      }
  }
```

(*__ior_new:N 定义结束。*)

__kernel_ior_open:Nn
__kernel_ior_open:No
__ior_open_stream:Nn 流分配本身利用了所有可用流的列表。保持同步很重要，采用两部分方法实现：已被 ior 占用但现在可用的任何流都会被跟踪，因此首先尝试这些。如果失败，则向 plain TEX 或LATEX 2ε请求新的流并使用该数字（经过一些转换）。

```
\cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
```

```
  {
    \ior_close:N #1
    \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
      { \__ior_open_stream:Nn #1 {#2} }
      {
        \__ior_new:N #1
        \__kernel_tl_set:Ne \l__ior_stream_tl { \int_eval:n {#1} }
        \__ior_open_stream:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__kernel_ior_open:Nn { No }
```

在这里，我们采取防御性措施，以防使用LuaTEX和无扩展名的文件名。

```
\cs_new_protected:Npe \__ior_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
    \prop_gput:NVn \exp_not:N \g__ior_streams_prop #1 {#2}
    \tex_openin:D #1
      \sys_if_engine_luatex:TF
        { {#2} }
        {  \exp_not:N \__kernel_file_name_quote:n {#2} \scan_stop: }
  }
```

(*__kernel_ior_open:Nn* 和 *__ior_open_stream:Nn* 定义结束。)

\ior_shell_open:Nn
__ior_shell_open:nN
__ior_shell_open:oN

实际上比标准的打开或输入版本要容易得多！在调用__kernel_ior_open:Nn时，文件上添加了管道以表示 shell 命令，但尚未添加引号，稍后由__kernel_file_name_quote:n添加。

```
\cs_new_protected:Npn \ior_shell_open:Nn #1#2
  {
    \sys_if_shell:TF
      { \__ior_shell_open:oN { \tl_to_str:n {#2} } #1 }
      { \msg_error:nn { kernel } { pipe-failed } }
  }
\cs_new_protected:Npn \__ior_shell_open:nN #1#2
  {
    \tl_if_in:nnTF {#1} { " }
      {
        \msg_error:nne
          { kernel } { quote-in-shell } {#1}
      }
      { \__kernel_ior_open:Nn #2 { |#1 } }
  }
```

```
\cs_generate_variant:Nn \__ior_shell_open:nN { o }
\msg_new:nnnn { kernel } { pipe-failed }
  { Cannot~run~piped~system~commands. }
  {
    LaTeX~tried~to~call~a~system~process~but~this~was~not~possible.\\
    Try~the~"--shell-escape"~(or~"--enable-pipes")~option.
  }
```

(*\ior_shell_open:Nn* 和 *__ior_shell_open:nN* 定义结束。这个函数被记录在第4页。)

\ior_close:N
\ior_close:c

关闭流意味着在 TEX 级别摆脱它，并从各种数据结构中删除。除非传递的名称是无效的流编号（在范围 $[0,15]$ 之外），否则可以关闭它。另一方面，只有在尚未在堆栈中时才将其添加到堆栈中，以避免重复堆积。

```
\cs_new_protected:Npn \ior_close:N #1
  {
    \int_compare:nT { -1 < #1 < \c__ior_term_ior }
      {
        \tex_closein:D #1
        \prop_gremove:NV \g__ior_streams_prop #1
        \seq_if_in:NVF \g__ior_streams_seq #1
          { \seq_gpush:NV \g__ior_streams_seq #1 }
        \cs_gset_eq:NN #1 \c__ior_term_ior
      }
  }
\cs_generate_variant:Nn \ior_close:N { c }
```

(*\ior_close:N* 定义结束。这个函数被记录在第4页。)

\ior_show:N
\ior_log:N
__ior_show:NN

寻找流在\g__ior_streams_prop列表中，然后根据情况显示流是否打开或关闭。

```
\cs_new_protected:Npn \ior_show:N { \__ior_show:NN \tl_show:n }
\cs_generate_variant:Nn \ior_show:N { c }
\cs_new_protected:Npn \ior_log:N { \__ior_show:NN \tl_log:n }
\cs_generate_variant:Nn \ior_log:N { c }
\cs_new_protected:Npn \__ior_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \prop_get:NVNTF \g__ior_streams_prop #2 \l__ior_internal_tl
          {
            \exp_args:Ne #1
              { \token_to_str:N #2 ~ open: ~ \l__ior_internal_tl }
          }
          { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
```

```
      }
  }
```

(*\ior_show:N*, *\ior_log:N*, 和 *__ior_show:NN* 定义结束。这些函数被记录在第5页。)

\ior_show_list:
\ior_log_list:
__ior_list:N

显示属性列表, 但带有一些"漂亮的打印"。参见 l3msg 模块。消息的第一个参数是ior(与iow相对), 第二个参数为空如果没有读取流打开, 否则非空(使用\msg_show_item_unbraced:nn格式化的流列表)。消息 show-streams 的代码负责将ior/iow翻译为英语。

```
\cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nneeee }
\cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__ior_list:N #1
  {
    #1 { kernel } { show-streams }
      { ior }
      {
        \prop_map_function:NN \g__ior_streams_prop
          \msg_show_item_unbraced:nn
      }
      { } { }
  }
```

(*\ior_show_list:*, *\ior_log_list:*, 和 *__ior_list:N* 定义结束。这些函数被记录在第5页。)

3.1.3 读取输入

\if_eof:w

原始条件

```
\cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(*\if_eof:w* 定义结束。这个函数被记录在第13页。)

\ior_if_eof_p:N
\ior_if_eof:NTF

为了测试某个特定的输入流是否耗尽,提供了以下条件。原始测试只能处理范围 $[0, 15]$ 内的数字, 所以我们捕获异常情况(它们已经耗尽)。

```
\prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
  {
    \if_int_compare:w -1 < #1
      \if_int_compare:w #1 < \c__ior_term_ior
        \if_eof:w #1
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      \else:
```

```
        \prg_return_true:
      \fi:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*\ior_if_eof:NTF* 定义结束。这个函数被记录在第9页。)

\ior_get:NN
__ior_get:NN
\ior_get:NNTF

在这里我们从文件中读取。

```
\cs_new_protected:Npn \ior_get:NN #1#2
  { \ior_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_new_protected:Npn \__ior_get:NN #1#2
  { \tex_read:D #1 to #2 }
\prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
  {
    \ior_if_eof:NTF #1
      { \prg_return_false: }
      {
        \__ior_get:NN #1 #2
        \prg_return_true:
      }
  }
```

(*\ior_get:NN*, *__ior_get:NN*, 和 *\ior_get:NNTF* 定义结束。这些函数被记录在第6页。)

\ior_str_get:NN
__ior_str_get:NN
\ior_str_get:NNTF

作为字符串读取是一个更复杂的换行器，因为我们希望删除换行字符并在之后恢复它。

```
\cs_new_protected:Npn \ior_str_get:NN #1#2
  { \ior_str_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_new_protected:Npn \__ior_str_get:NN #1#2
  {
    \exp_args:Nno \use:n
      {
        \int_set:Nn \tex_endlinechar:D { -1 }
        \tex_readline:D #1 to #2
        \int_set:Nn \tex_endlinechar:D
      }   { \int_use:N \tex_endlinechar:D }
  }
\prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
  {
    \ior_if_eof:NTF #1
      { \prg_return_false: }
```

```
      {
        \__ior_str_get:NN #1 #2
        \prg_return_true:
      }
  }
```

(\ior_str_get:NN, __ior_str_get:NN, 和 \ior_str_get:NNTF 定义结束。这些函数被记录在第7页。)

\c__ior_term_noprompt_ior 用于无提示的读取。

```
\int_const:Nn \c__ior_term_noprompt_ior { -1 }
```

(\c__ior_term_noprompt_ior 定义结束。)

\ior_get_term:nN
\ior_str_get_term:nN
__ior_get_term:NnN 从终端获取最好使用漂亮的打印。

```
\cs_new_protected:Npn \ior_get_term:nN #1#2
  { \__ior_get_term:NnN \__ior_get:NN {#1} #2 }
\cs_new_protected:Npn \ior_str_get_term:nN #1#2
  { \__ior_get_term:NnN \__ior_str_get:NN {#1} #2 }
\cs_new_protected:Npn \__ior_get_term:NnN #1#2#3
  {
    \group_begin:
      \tex_escapechar:D = -1 \scan_stop:
      \tl_if_blank:nTF {#2}
        { \exp_args:NNc #1 \c__ior_term_noprompt_ior }
        { \exp_args:NNc #1 \c__ior_term_ior }
          {#2}
    \exp_args:NNNv \group_end:
    \tl_set:Nn #3 {#2}
  }
```

(\ior_get_term:nN, \ior_str_get_term:nN, 和 __ior_get_term:NnN 定义结束。这些函数被记录在第9页。)

\ior_map_break:
\ior_map_break:n 常规映射中断函数。

```
\cs_new:Npn \ior_map_break:
  { \prg_map_break:Nn \ior_map_break: { } }
\cs_new:Npn \ior_map_break:n
  { \prg_map_break:Nn \ior_map_break: }
```

(\ior_map_break: 和 \ior_map_break:n 定义结束。这些函数被记录在第8页。)

\ior_map_inline:Nn
\ior_str_map_inline:Nn
__ior_map_inline:NNn
__ior_map_inline:NNNn
__ior_map_inline_loop:NNN 在输入流上进行映射可以基于令牌或字符串进行，因此设置了两者。在其中，有一个检查，以避免读取文件末尾之外的内容，因此使用了两次 \ior_if_eof:N及其底层相似物\if_eof:w。该映射不能嵌套使用相同的流两次，因为流只有一个“当前行”。

```
\cs_new_protected:Npn \ior_map_inline:Nn
  { \__ior_map_inline:NNn \__ior_get:NN }
\cs_new_protected:Npn \ior_str_map_inline:Nn
  { \__ior_map_inline:NNn \__ior_str_get:NN }
\cs_new_protected:Npn \__ior_map_inline:NNn
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:Nc \__ior_map_inline:NNNn
      { __ior_map_ \int_use:N \g__kernel_prg_map_int :n }
  }
\cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
  {
    \cs_gset_protected:Npn #1 ##1 {#4}
    \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
    \prg_break_point:Nn \ior_map_break:
      { \int_gdecr:N \g__kernel_prg_map_int }
  }
\cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
  {
    #2 #3 \l__ior_internal_tl
    \if_eof:w #3
      \exp_after:wN \ior_map_break:
    \fi:
    \exp_args:No #1 \l__ior_internal_tl
    \__ior_map_inline_loop:NNN #1#2#3
  }
```

(*\ior_map_inline:Nn* 以及其它的定义结束。这些函数被记录在第7页。)

\ior_map_variable:NNn
\ior_str_map_variable:NNn
__ior_map_variable:NNNn
__ior_map_variable_loop:NNNn

由于 TEX 原语（\read或\readline）以与令牌列表分配相同的方式分配读取的令牌，我们只需调用相应的原语。为了提高速度，使用原语条件检查循环结束。

```
\cs_new_protected:Npn \ior_map_variable:NNn
  { \__ior_map_variable:NNNn \ior_get:NN }
\cs_new_protected:Npn \ior_str_map_variable:NNn
  { \__ior_map_variable:NNNn \ior_str_get:NN }
\cs_new_protected:Npn \__ior_map_variable:NNNn #1#2#3#4
  {
    \ior_if_eof:NF #2 { \__ior_map_variable_loop:NNNn #1#2#3 {#4} }
    \prg_break_point:Nn \ior_map_break: { }
  }
\cs_new_protected:Npn \__ior_map_variable_loop:NNNn #1#2#3#4
  {
    #1 #2 #3
```

```
    \if_eof:w #2
      \exp_after:wN \ior_map_break:
    \fi:
    #4
    \__ior_map_variable_loop:NNNn #1#2#3 {#4}
  }
```

(*\ior_map_variable:NNn* 以及其它的定义结束。这些函数被记录在第7页。)

3.2 输出操作

```
⟨@@=iow⟩
```

这里与输入操作有很多相似之处，至少对于许多基本操作来说是这样。因此，从前面的材料中复制了相当多的内容，进行了一些小的修改。

3.2.1 变量和常量

\l__iow_internal_tl 用作短期临时变量。

```
\tl_new:N  \l__iow_internal_tl
```

(*\l__iow_internal_tl* 定义结束。)

\c_log_iow
\c_term_iow 在这里，我们为写入到传输文件（\c_log_iow）和同时写入到终端和传输文件（\c_term_iow）分配了两个输出流。最近的LuaTEX提供 128 个写入流；我们还将\c_term_iow用作第一个不允许写入流，因此其值取决于引擎。

```
\int_const:Nn \c_log_iow  { -1 }
\int_const:Nn \c_term_iow
  {
    \bool_lazy_and:nnTF
      { \sys_if_engine_luatex_p: }
      { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
      { 128 }
      { 16 }
  }
```

(*\c_log_iow* 和 *\c_term_iow* 定义结束。这些变量被记录在第13页。)

\g__iow_streams_seq 一个当前可用的输出流列表，用作堆栈。

```
\seq_new:N \g__iow_streams_seq
```

(*\g__iow_streams_seq* 定义结束。)

\l__iow_stream_tl 用于从堆栈中恢复原始流编号。

```
\tl_new:N \l__iow_stream_tl
```

(*\l__iow_stream_tl* 定义结束。)

\g__iow_streams_prop 就像读取一样，根据要检查的寄存器编号进行适当的调整。

```
\prop_new:N \g__iow_streams_prop
\int_step_inline:nnn
  { 0 }
  {
    \cs_if_exist:NTF \contextversion
      { \tex_count:D 39 ~ }
      {
        \tex_count:D 17 ~
        \cs_if_exist:NT \loccount { - 1 }
      }
  }
  {
    \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved~by~format }
  }
```

(*\g__iow_streams_prop* 定义结束。)

3.2.2 内部辅助

\s__iow_mark 内部扫描标记。

\s__iow_stop

```
\scan_new:N \s__iow_mark
\scan_new:N \s__iow_stop
```

(*\s__iow_mark* 和 *\s__iow_stop* 定义结束。)

__iow_use_i_delimit_by_s_stop:nw 到扫描标记的函数。

```
\cs_new:Npn \__iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}
```

(*__iow_use_i_delimit_by_s_stop:nw* 定义结束。)

\q__iow_nil 内部 quark 。

```
\quark_new:N \q__iow_nil
```

(*\q__iow_nil* 定义结束。)

3.3 流管理

\iow_new:N
\iow_new:c

通过将名称定义为写入终端来保留新流：奇怪但至少一致。

```
\cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
\cs_generate_variant:Nn \iow_new:N { c }
```

(*\iow_new:N* 定义结束。这个函数被记录在第3页。)

\g_tmpa_iow
\g_tmpb_iow

常规的临时空间。

```
\iow_new:N \g_tmpa_iow
\iow_new:N \g_tmpb_iow
```

(*\g_tmpa_iow* 和 *\g_tmpb_iow* 定义结束。这些变量被记录在第13页。)

__iow_new:N

与读取流一样，复制\newwrite，确保它不是\outer。对于ConTEXt，我们必须处理\newwrite的工作方式与我们自己的相同的事实：在更改定义之前，它实际上会进行检查。

```
\exp_args:NNf \cs_new_protected:Npn \__iow_new:N
  { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
\cs_if_exist:NT \contextversion
  {
    \cs_new_eq:NN \__iow_new_aux:N \__iow_new:N
    \cs_gset_protected:Npn \__iow_new:N #1
      {
        \cs_undefine:N #1
        \__iow_new_aux:N #1
      }
  }
```

(*__iow_new:N* 定义结束。)

\l__iow_file_name_tl

数据存储。

```
\tl_new:N \l__iow_file_name_tl
```

(*\l__iow_file_name_tl* 定义结束。)

\iow_open:Nn
\iow_open:NV
\iow_open:cn
\iow_open:cV
__iow_open_stream:Nn
__iow_open_stream:NV

与读取相同的思路，但不包括路径，也不需要允许有条件版本。

```
\cs_new_protected:Npn \iow_open:Nn #1#2
  {
    \__kernel_tl_set:Ne \l__iow_file_name_tl
      { \__kernel_file_name_sanitize:n {#2} }
    \iow_close:N #1
    \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
      { \__iow_open_stream:NV #1 \l__iow_file_name_tl }
```

```
      {
        \__iow_new:N #1
        \__kernel_tl_set:Ne \l__iow_stream_tl { \int_eval:n {#1} }
        \__iow_open_stream:NV #1 \l__iow_file_name_tl
      }
  }
\cs_generate_variant:Nn \iow_open:Nn { NV , c , cV }
\cs_new_protected:Npn \__iow_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
    \prop_gput:NVn \g__iow_streams_prop #1 {#2}
    \tex_immediate:D \tex_openout:D
        #1 \__kernel_file_name_quote:n {#2} \scan_stop:
  }
\cs_generate_variant:Nn \__iow_open_stream:Nn { NV }
```

(*\iow_open:Nn* 和 *__iow_open_stream:Nn* 定义结束。这个函数被记录在第4页。)

\iow_shell_open:Nn
__iow_shell_open:nN
__iow_shell_open:oN

与 ior 版本非常相似。

```
\cs_new_protected:Npn \iow_shell_open:Nn #1#2
  {
    \sys_if_shell:TF
      { \__iow_shell_open:oN { \tl_to_str:n {#2} } #1 }
      { \msg_error:nn { kernel } { pipe-failed } }
  }
\cs_new_protected:Npn \__iow_shell_open:nN #1#2
  {
    \tl_if_in:nnTF {#1} { " }
      {
        \msg_error:nne
          { kernel } { quote-in-shell } {#1}
      }
      { \__kernel_iow_open:Nn #2 { |#1 } }
  }
\cs_generate_variant:Nn \__iow_shell_open:nN { o }
```

(*\iow_shell_open:Nn* 和 *__iow_shell_open:nN* 定义结束。这个函数被记录在第4页。)

\iow_close:N
\iow_close:c

关闭流不完全是打开的逆操作。首先，关闭操作比打开操作更容易，其次，由于流实际上是一个数字，我们可以直接使用它来显示已释放的槽。

```
\cs_new_protected:Npn \iow_close:N #1
  {
    \int_compare:nT { \c_log_iow < #1 < \c_term_iow }
```

```
      {
        \tex_immediate:D \tex_closeout:D #1
        \prop_gremove:NV \g__iow_streams_prop #1
        \seq_if_in:NVF \g__iow_streams_seq #1
          { \seq_gpush:NV \g__iow_streams_seq #1 }
        \cs_gset_eq:NN #1 \c_term_iow
      }
  }
\cs_generate_variant:Nn \iow_close:N { c }
```

(*\iow_close:N* 定义结束。这个函数被记录在第4页。)

\iow_show:N
\iow_log:N
__iow_show:NN

在\g__iow_streams_prop列表中查找流，然后根据情况显示流是否打开或关闭。

```
\cs_new_protected:Npn \iow_show:N { \__iow_show:NN \tl_show:n }
\cs_generate_variant:Nn \iow_show:N { c }
\cs_new_protected:Npn \iow_log:N { \__iow_show:NN \tl_log:n }
\cs_generate_variant:Nn \iow_log:N { c }
\cs_new_protected:Npn \__iow_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \prop_get:NVNTF \g__iow_streams_prop #2 \l__iow_internal_tl
          {
            \exp_args:Ne #1
              { \token_to_str:N #2 ~ open: ~ \l__iow_internal_tl }
          }
          { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
      }
  }
```

(*\iow_show:N*, *\iow_log:N*, 和 *__iow_show:NN* 定义结束。这些函数被记录在第5页。)

\iow_show_list:
\iow_log_list:
__iow_list:N

与输入一样，但是辅助函数的副本，以便名称是正确的。

```
\cs_new_protected:Npn \iow_show_list: { \__iow_list:N \msg_show:nneeee }
\cs_new_protected:Npn \iow_log_list: { \__iow_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__iow_list:N #1
  {
    #1 { kernel } { show-streams }
      { iow }
      {
        \prop_map_function:NN \g__iow_streams_prop
          \msg_show_item_unbraced:nn
      }
```

```
      { } { }
  }
```

(*\iow_show_list:*, *\iow_log_list:*, 和 *__iow_list:N* 定义结束。这些函数被记录在第5页。)

3.3.1 延迟写入

\iow_shipout_e:Nn
\iow_shipout_e:Ne
\iow_shipout_e:cn
\iow_shipout_e:ce

首先是简单部分，这是原语，期望其参数用大括号括起来。

```
\cs_new_protected:Npn \iow_shipout_e:Nn #1#2
  { \tex_write:D #1 {#2} }
\cs_generate_variant:Nn \iow_shipout_e:Nn { Ne , c, ce }
```

(*\iow_shipout_e:Nn* 定义结束。这个函数被记录在第10页。)

\iow_shipout:Nn
\iow_shipout:Ne
\iow_shipout:Nx
\iow_shipout:cn
\iow_shipout:ce
\iow_shipout:cx

有了ε-TEX，不带扩展的延迟写入很容易。

```
\cs_new_protected:Npn \iow_shipout:Nn #1#2
  { \tex_write:D #1 { \exp_not:n {#2} } }
\cs_generate_variant:Nn \iow_shipout:Nn { Ne , c, ce }
\cs_generate_variant:Nn \iow_shipout:Nn { Nx , cx }
```

(*\iow_shipout:Nn* 定义结束。这个函数被记录在第10页。)

3.3.2 立即写入

__kernel_iow_with:Nnn
__iow_with:nNnn
__iow_with:oNnn

如果整数 #1 等于 #2，则只需将 #3 保留在输入流中。否则，将旧值传递给辅助函数，该函数将整数设置为新值，运行代码，然后还原整数。

```
\cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
  {
    \int_compare:nNnTF {#1} = {#2}
      { \use:n }
      { \__iow_with:oNnn { \int_use:N #1 } #1 {#2} }
  }
\cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
  {
    \int_set:Nn #2 {#3}
    #4
    \int_set:Nn #2 {#1}
  }
\cs_generate_variant:Nn \__iow_with:nNnn { o }
```

(*__kernel_iow_with:Nnn* 和 *__iow_with:nNnn* 定义结束。)

\iow_now:Nn
\iow_now:NV
\iow_now:Ne
\iow_now:Nx
\iow_now:cn
\iow_now:cV
\iow_now:ce
\iow_now:cx

该例程将第二个参数写入输出流，不进行扩展。如果此流未打开，则输出转到终端。如果第一个参数根本不是输出流，我们会得到一个内部错误。我们不使用\write执行的扩展来获得Nx变体，因为它与 x-扩展在细微的方面有所不同，即，不需要将宏参数字符加倍。我们使用__kernel_iow_with:Nnn将\newlinechar设置为 10，以支持格式，例如 plain TEX：否则，\iow_newline: 将无法工作。我们对\iow_shipout:Nn或\iow_shipout_x:Nn不执行此操作，因为在这些情况下，TEX 在出航时查看\newlinechar的值。

```
\cs_new_protected:Npn \iow_now:Nn #1#2
  {
    \__kernel_iow_with:Nnn \tex_newlinechar:D { `\^^J }
      { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
  }
\cs_generate_variant:Nn \iow_now:Nn { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \iow_now:Nn { Nx , cx }
```

(*\iow_now:Nn* 定义结束。这个函数被记录在第10页。)

\iow_log:n
\iow_log:e
\iow_log:x
\iow_term:n
\iow_term:e
\iow_term:x

直接写入日志和终端相对较容易。由于我们需要两个 e-类型的变体进行引导，因此在这里进行重新定义。

```
\cs_new_protected:Npn \iow_log:n  { \iow_now:Nn \c_log_iow  }
\cs_set_protected:Npn \iow_log:e  { \iow_now:Ne \c_log_iow  }
\cs_generate_variant:Nn \iow_log:n { x }
\cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
\cs_set_protected:Npn \iow_term:e { \iow_now:Ne \c_term_iow }
\cs_generate_variant:Nn \iow_term:n { x }
```

(*\iow_log:n* 和 *\iow_term:n* 定义结束。这些函数被记录在第10页。)

3.3.3 写入的特殊字符

\iow_newline:

全局变量，保存强制将新行写入输出流时的字符。

```
\cs_new:Npn \iow_newline: { ^^J }
```

(*\iow_newline:* 定义结束。这个函数被记录在第11页。)

\iow_char:N

用于将任何转义字符写入输出流的函数。

```
\cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(*\iow_char:N* 定义结束。这个函数被记录在第11页。)

3.3.4 将行硬换行到字符计数

此处的代码实现了通用的硬换行函数。这用于消息系统，但设计为可用于其他用途。

\l_iow_line_count_int 这是可以写入终端的一行字符的“原始”数量。标准值是 TEX Live 和MiKTEX 通常使用的行长度。

```
\int_new:N  \l_iow_line_count_int
\int_set:Nn \l_iow_line_count_int { 78 }
```

(*\l_iow_line_count_int* 定义结束。这个变量被记录在第13页。)

\l__iow_newline_tl 插入产生新行的记号列表，带有⟨*run-on text*⟩。

```
\tl_new:N \l__iow_newline_tl
```

(*\l__iow_newline_tl* 定义结束。)

\l__iow_line_target_int 这存储目标行数：每行字符的完整数量，减去每行开头的引导部分。

```
\int_new:N \l__iow_line_target_int
```

(*\l__iow_line_target_int* 定义结束。)

__iow_set_indent:n
__iow_unindent:w
\l__iow_one_indent_tl
\l__iow_one_indent_int

one_indent 变量保存一个缩进标记及其长度。__iow_unindent:w辅助函数删除一个缩进。函数__iow_set_indent:n（可能是公共的）以一致的方式设置缩进。默认设置为四个空格。

```
\tl_new:N \l__iow_one_indent_tl
\int_new:N \l__iow_one_indent_int
\cs_new:Npn \__iow_unindent:w { }
\cs_new_protected:Npn \__iow_set_indent:n #1
  {
    \__kernel_tl_set:Ne \l__iow_one_indent_tl
      { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }
    \int_set:Nn \l__iow_one_indent_int
      { \str_count:N \l__iow_one_indent_tl }
    \exp_last_unbraced:NNo
      \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
  }
\exp_args:Ne \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }
```

(*__iow_set_indent:n* 以及其它的定义结束。)

\l__iow_indent_tl
\l__iow_indent_int

当前缩进（\l__iow_one_indent_tl的一些副本）及其字符数。

```
\tl_new:N \l__iow_indent_tl
\int_new:N \l__iow_indent_int
```

(*\l__iow_indent_tl* 和 *\l__iow_indent_int* 定义结束。)

\l__iow_line_tl
\l__iow_line_part_tl

这些分别保存当前文本行和要添加到它的部分行。

```
\tl_new:N \l__iow_line_tl
\tl_new:N \l__iow_line_part_tl
```

(*\l__iow_line_tl* 和 *\l__iow_line_part_tl* 定义结束。)

\l__iow_line_break_bool

指示行是否精确地在块边界处中断的布尔值。

```
\bool_new:N \l__iow_line_break_bool
```

(*\l__iow_line_break_bool* 定义结束。)

\l__iow_wrap_tl

用于解标记之前的扩展步骤，以及用于换行文本的输出：完全展开，行不会过长。

```
\tl_new:N \l__iow_wrap_tl
```

(*\l__iow_wrap_tl* 定义结束。)

\c__iow_wrap_marker_tl
\c__iow_wrap_end_marker_tl
\c__iow_wrap_newline_marker_tl
\c__iow_wrap_allow_break_marker_tl
\c__iow_wrap_indent_marker_tl
\c__iow_wrap_unindent_marker_tl

换行代码的每个特殊操作都以相同可识别的字符串\c__iow_wrap_marker_tl开头。看到该“word”后，换行代码读取一个空格分隔的参数，以知道要执行的操作。此处设置\escapechar并不是非常重要，但使\c__iow_wrap_marker_tl看起来略微更漂亮。

```
\group_begin:
  \int_set:Nn \tex_escapechar:D { -1 }
  \tl_const:Ne \c__iow_wrap_marker_tl
    { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
\group_end:
\tl_map_inline:nn
  { { end } { newline } { allow_break } { indent } { unindent } }
  {
    \tl_const:ce { c__iow_wrap_ #1 _marker_tl }
      {
        \c__iow_wrap_marker_tl
        #1
        \c_catcode_other_space_tl
      }
  }
```

(*\c__iow_wrap_marker_tl* 以及其它的定义结束。)

\iow_wrap_allow_break:
__iow_wrap_allow_break:
__iow_wrap_allow_break_error:

我们将\iow_wrap_allow_break:n设置为在消息外部产生错误。在换行消息中，如果有效，它设置为__iow_wrap_allow_break:，否则为__iow_wrap_allow_break_error:。第二个是可展开产生错误。

```
\cs_new_protected:Npn \iow_wrap_allow_break:
```

```
  {
    \msg_error:nnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
  }
\cs_new:Npe \__iow_wrap_allow_break: { \c__iow_wrap_allow_break_marker_tl }
\cs_new:Npn \__iow_wrap_allow_break_error:
  {
    \msg_expandable_error:nnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
  }
```

(*\iow_wrap_allow_break:*, *__iow_wrap_allow_break:*, 和 *__iow_wrap_allow_break_error:* 定义结束。这个函数被记录在第12页。)

\iow_indent:n
__iow_indent:n
__iow_indent_error:n

我们将\iow_indent:n设置为在消息外部产生错误。在换行消息中，如果有效，它设置为__iow_indent:n，否则为__iow_indent_error:n。第一个在其参数之前放置增加缩进的指令，以及之后放置减少缩进的指令。第二个是可展开产生错误。请注意，没有强制换行，因此缩进仅在下一行开始时更改。

```
\cs_new_protected:Npn \iow_indent:n #1
  {
    \msg_error:nnnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_indent:n } {#1}
    #1
  }
\cs_new:Npe \__iow_indent:n #1
  {
    \c__iow_wrap_indent_marker_tl
    #1
    \c__iow_wrap_unindent_marker_tl
  }
\cs_new:Npn \__iow_indent_error:n #1
  {
    \msg_expandable_error:nnnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_indent:n } {#1}
    #1
  }
```

(*\iow_indent:n*, *__iow_indent:n*, 和 *__iow_indent_error:n* 定义结束。这个函数被记录在第12页。)

\iow_wrap:nnnN
\iow_wrap:nenN

主要的换行函数工作如下。首先给出\\、\␣等格式化命令正确的定义以及执行给定的设置 #3。\␣的定义使用“other” space 而不是正常空格，因为后者可能被 TEX 吸收以结束数字或其他 f-type 扩展。如果定义了\conditionally@traceoff，则使用它；它由 trace 宏包引入，抑制换行代码的不相关跟踪。

```
\cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
  {
    \group_begin:
      \cs_if_exist_use:N \conditionally@traceoff
      \int_set:Nn \tex_escapechar:D { -1 }
      \cs_set:Npe \{ { \token_to_str:N \{ }
      \cs_set:Npe \# { \token_to_str:N \# }
      \cs_set:Npe \} { \token_to_str:N \} }
      \cs_set:Npe \% { \token_to_str:N \% }
      \cs_set:Npe \~ { \token_to_str:N \~ }
      \int_set:Nn \tex_escapechar:D { 92 }
      \cs_set_eq:NN \\ \iow_newline:
      \cs_set_eq:NN \  \c_catcode_other_space_tl
      \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break:
      \cs_set_eq:NN \iow_indent:n \__iow_indent:n
      #3
```

然后完全展开输入：在宏包模式中，扩展使用LaTeX 2_ε的`\protect`机制，与`\typeout`相同。在通用模式中，此设置无用但无害。一旦展开完成，将`\iow_indent:n`重置为其错误定义：它只在`\iow_wrap:nnnN`的第一个参数中起作用。

```
      \cs_set_eq:NN \protect \token_to_str:N
      \__kernel_tl_set:Ne \l__iow_wrap_tl {#1}
      \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break_error:
      \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n
```

然后，设置换行标记（两次赋值完全展开，然后转换为字符串），并初始化行的目标计数（第一行的目标计数为`\l_iow_line_count_int`）。

```
      \__kernel_tl_set:Ne \l__iow_newline_tl { \iow_newline: #2 }
      \__kernel_tl_set:Ne \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
      \int_set:Nn \l__iow_line_target_int
        { \l_iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }
```

完整性检查。

```
      \int_compare:nNnT { \l__iow_line_target_int } < 0
        {
          \tl_set:Nn \l__iow_newline_tl { \iow_newline: }
          \int_set:Nn \l__iow_line_target_int
            { \l_iow_line_count_int + 1 }
        }
```

然后是对输入的循环，它将换行的结果存储在`\l__iow_wrap_tl`中。循环后，将结果的文本传递给作为后处理器给定的函数。`\tl_to_str:N`步骤将“other”空格转换回普通空格。f-expansion 从`\l__iow_wrap_tl`中移除前导空格。

```
      \__iow_wrap_do:
    \exp_args:NNf \group_end:
    #4 { \tl_to_str:N \l__iow_wrap_tl }
  }
\cs_generate_variant:Nn \iow_wrap:nnnN { ne }
```

(*\iow_wrap:nnnN* 定义结束。这个函数被记录在第12页。)

__iow_wrap_do:
__iow_wrap_fix_newline:w
__iow_wrap_start:w

转义空格并将换行符更改为\c__iow_wrap_newline_marker_tl。设置一些变量，特别是\l__iow_wrap_tl的初始值：空格停止主换行函数的 f-扩展，\use_none:n去除后续代码插入的换行符。主循环包括反复调用 chunk 辅助程序，以换行由（换行符或缩进）标记界定的块。

```
\cs_new_protected:Npn \__iow_wrap_do:
  {
    \__kernel_tl_set:Ne \l__iow_wrap_tl
      {
        \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
        \c__iow_wrap_end_marker_tl
      }
    \__kernel_tl_set:Ne \l__iow_wrap_tl
      {
        \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
          ^^J \q__iow_nil ^^J \s__iow_stop
      }
    \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
  }
\cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
  {
    #1
    \if_meaning:w \q__iow_nil #2
      \__iow_use_i_delimit_by_s_stop:nw
    \fi:
    \c__iow_wrap_newline_marker_tl
    \__iow_wrap_fix_newline:w #2 ^^J
  }
\cs_new_protected:Npn \__iow_wrap_start:w
  {
    \bool_set_false:N \l__iow_line_break_bool
    \tl_clear:N \l__iow_line_tl
    \tl_clear:N \l__iow_line_part_tl
    \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
    \int_zero:N \l__iow_indent_int
```

```
    \tl_clear:N \l__iow_indent_tl
    \__iow_wrap_chunk:nw { \l_iow_line_count_int }
  }
```

(__iow_wrap_do:, __iow_wrap_fix_newline:w, 和 __iow_wrap_start:w 定义结束。)

__iow_wrap_chunk:nw
__iow_wrap_next:nw

辅助程序 chunk 和 next 间接定义，以获取\c_catcode_other_space_tl和\c__iow_wrap_marker_tl的扩展。next 辅助程序调用与标记类型相对应的函数（其##2），可以是 newline、indent、unindent 或 end。chunk 辅助程序的第一个参数是字符的目标数，第二个是要换行的字符串。如果块为空，只需调用 next。否则，设置调用__iow_wrap_line:nw的调用，包括如果当前行为空则包括缩进，并在__iow_wrap_end_chunk:w辅助程序之前包括一个尾随空格（#1）。

```
\cs_set_protected:Npn \__iow_tmp:w #1#2
  {
    \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
      {
        \tl_if_empty:nTF {##2}
          {
            \tl_clear:N \l__iow_line_part_tl
            \__iow_wrap_next:nw {##1}
          }
          {
            \tl_if_empty:NTF \l__iow_line_tl
              {
                \__iow_wrap_line:nw
                  { \l__iow_indent_tl }
                  ##1 - \l__iow_indent_int ;
              }
              { \__iow_wrap_line:nw { } ##1 ; }
            ##2 #1
            \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
          }
      }
    \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
      { \use:c { __iow_wrap_##2:n } {##1} }
  }
\exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl
```

(__iow_wrap_chunk:nw 和 __iow_wrap_next:nw 定义结束。)

__iow_wrap_line:nw
__iow_wrap_line_loop:w
__iow_wrap_line_aux:Nw
__iow_wrap_line_seven:nnnnnnn
__iow_wrap_line_end:NnnnnnnnN
__iow_wrap_line_end:nw
__iow_wrap_end_chunk:w

接下来是 {⟨*string*⟩} ⟨*int expr*⟩ ;。它将⟨*string*⟩和来自当前块的⟨*int expr*⟩字符存储到\l__iow_line_part_tl中。字符以 8 个为一组抓取，并由 line_loop 辅助程序

将其留在\l__iow_line_part_tl中。当还有 $k < 8$ 个要找到时，line_aux 辅助程序调用 line_end 辅助程序，后跟（单个数字）k，然后是 $7 - k$ 个空的花括号组，然后是块的剩余字符。line_end 辅助程序将来自块的 k 个字符留在行部分中，然后结束赋值。暂时忽略\use_none:nnnnn行。如果下一个字符是空格，则可以在此处换行：将找到的内容存储到结果中并获取下一行。否则，需要一些工作来找到断点。到目前为止，我们忽略了如果块短于请求的字符数会发生什么：这由 end_chunk 辅助程序处理，它被代码的其余部分视为字符处理。它最终被调用为 line_loop 辅助程序的参数之一#2–#9，或者作为 line_end 辅助程序的参数之一#2–#8。在这两种情况下，停止赋值并计算尚需多少字符。请注意，当我们有七个参数需要清理时，必须插入\exp_stop_f:来停止\exp:w。奇怪的\use_none:nnnnn确保所需的数据位于正确的位置。

```
\cs_new_protected:Npn \__iow_wrap_line:nw #1
  {
    \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
    #1
    \exp_after:wN \__iow_wrap_line_loop:w
    \int_value:w \int_eval:w
  }
\cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #1 < 8 \exp_stop_f:
      \__iow_wrap_line_aux:Nw #1
    \fi:
    #2 #3 #4 #5 #6 #7 #8 #9
    \exp_after:wN \__iow_wrap_line_loop:w
    \int_value:w \int_eval:w #1 - 8 ;
  }
\cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
  {
    #2
    \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \exp_after:wN \exp_after:wN
    \if_case:w #1 \exp_stop_f:
         \prg_do_nothing:
    \or: \use_none:n
    \or: \use_none:nn
    \or: \use_none:nnn
    \or: \use_none:nnnn
```

```
    \or: \use_none:nnnnn
    \or: \use_none:nnnnnn
    \or: \__iow_wrap_line_seven:nnnnnnn
    \fi:
    { } { } { } { } { } { } { } #3
  }
\cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
\cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    #2 #3 #4 #5 #6 #7 #8
    \use_none:nnnnn \int_eval:w 8 - ; #9
    \token_if_eq_charcode:NNTF \c_space_token #9
      { \__iow_wrap_line_end:nw { } }
      { \if_false: { \fi: } \__iow_wrap_break:w #9 }
  }
\cs_new:Npn \__iow_wrap_line_end:nw #1
  {
    \if_false: { \fi: }
    \__iow_wrap_store_do:n {#1}
    \__iow_wrap_next_line:w
  }
\cs_new:Npn \__iow_wrap_end_chunk:w
    #1 \int_eval:w #2 - #3 ; #4#5 \s__iow_stop
  {
    \if_false: { \fi: }
    \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
  }
```

(*__iow_wrap_line:nw 以及其它的定义结束。*)

__iow_wrap_break:w
__iow_wrap_break_first:w
__iow_wrap_break_none:w
__iow_wrap_break_loop:w
__iow_wrap_break_end:w

这里定义的函数是间接定义的：__iow_tmp:w 最终会以一个“其他”空格作为其参数调用。目标是从\l__iow_line_part_tl中移除最后一个空格之后的部分。在大多数情况下，这是通过反复调用 break_loop 辅助程序完成的，该程序在达到尾随空格之前留下“单词”（由空格分隔）：然后其参数 ##3 是 ? __iow_wrap_break_end:w 而不是单个标记，而 break_end 辅助程序在分配中留下直到最后一个空格的行，然后调用__iow_wrap_line_end:nw来完成行并继续下一个。如果\l__iow_line_part_tl中没有空格，那么 break_first 辅助程序调用 break_none 辅助程序。在这种情况下，如果当前行为空，则将完整的单词（包括 ##4，超出我们抓取的字符之外的字符）添加到行中，使其过长。否则，该单词用于下一行（并且由于存在标记，因此删除到目前为止的行的最后一个空格）。

```
\cs_set_protected:Npn \__iow_tmp:w #1
```

```
{
  \cs_new:Npn \__iow_wrap_break:w
    {
      \tex_edef:D \l__iow_line_part_tl
        { \if_false: } \fi:
          \exp_after:wN \__iow_wrap_break_first:w
          \l__iow_line_part_tl
          #1
          { ? \__iow_wrap_break_end:w }
          \s__iow_mark
    }
  \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
    {
      \use_none:nn ##2 \__iow_wrap_break_none:w
      \__iow_wrap_break_loop:w ##1 #1 ##2
    }
  \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
    {
      \tl_if_empty:NTF \l__iow_line_tl
        { ##2 ##4 \__iow_wrap_line_end:nw { } }
        { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
    }
  \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
    {
      \use_none:n ##3
      ##1 #1
      \__iow_wrap_break_loop:w ##2 #1 ##3
    }
  \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
    { ##1 \__iow_wrap_line_end:nw { } ##3 }
}
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
```

(*__iow_wrap_break:w 以及其它的定义结束。*)

__iow_wrap_next_line:w 在这里检测到特殊情况，即一行的结尾与块的结尾重合，以避免产生虚假的空行。否则，调用__iow_wrap_line:nw来查找下一行的字符（记得考虑缩进）。

```
\cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
  {
    \tl_clear:N \l__iow_line_tl
    \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
      {
        \tl_clear:N \l__iow_line_part_tl
```

```
      \bool_set_true:N \l__iow_line_break_bool
      \__iow_wrap_next:nw { \l__iow_line_target_int }
    }
    {
      \__iow_wrap_line:nw
        { \l__iow_indent_tl }
        \l__iow_line_target_int - \l__iow_indent_int ;
        #1 #2 \s__iow_stop
    }
  }
```

(*__iow_wrap_next_line:w* 定义结束。)

__iow_wrap_allow_break:n

在换行完块后调用这个函数。由于 allow_break 标记不应插入空格，因此通常\l__iow_line_part_tl以空格结尾（除非在行的开头?），我们移除它。然后继续下一个块，确保调整行的目标字符数，以防我们删除了一个空格。

```
\cs_new_protected:Npn \__iow_wrap_allow_break:n #1
  {
    \__kernel_tl_set:Ne \l__iow_line_tl
      { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \tl_if_empty:NTF \l__iow_line_part_tl
      { \__iow_wrap_chunk:nw {#1} }
      { \exp_args:Nf \__iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
  }
```

(*__iow_wrap_allow_break:n* 定义结束。)

__iow_wrap_indent:n
__iow_wrap_unindent:n

这些函数在换行完块后，在遇到 indent/unindent 标记时调用。将行部分（前一个块的最后一行部分）添加到行中，并重置表示存在换行的布尔值。最重要的是，从当前缩进（整数和记号列表）中添加或删除一个缩进。最后，继续换行。

```
\cs_new_protected:Npn \__iow_wrap_indent:n #1
  {
    \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
    \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
    \__iow_wrap_chunk:nw {#1}
  }
\cs_new_protected:Npn \__iow_wrap_unindent:n #1
  {
    \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
```

```
    \bool_set_false:N \l__iow_line_break_bool
    \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
    \__kernel_tl_set:Ne \l__iow_indent_tl
      { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
    \__iow_wrap_chunk:nw {#1}
  }
```

(*__iow_wrap_indent:n* 和 *__iow_wrap_unindent:n* 定义结束。)

__iow_wrap_newline:n
__iow_wrap_end:n

在遇到 newline/end 标记时，这些函数在一块文本被换行后被调用。除非刚刚发生过换行，否则将行部分和当前行存储到整个\l__iow_wrap_tl中，修剪尾随空格。在 newline 情况下，在新的块中查找长度为\l__iow_line_target_int的新行。

```
\cs_new_protected:Npn \__iow_wrap_newline:n #1
  {
    \bool_if:NF \l__iow_line_break_bool
      { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
    \bool_set_false:N \l__iow_line_break_bool
    \__iow_wrap_chunk:nw { \l__iow_line_target_int }
  }
\cs_new_protected:Npn \__iow_wrap_end:n #1
  {
    \bool_if:NF \l__iow_line_break_bool
      { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
    \bool_set_false:N \l__iow_line_break_bool
  }
```

(*__iow_wrap_newline:n* 和 *__iow_wrap_end:n* 定义结束。)

__iow_wrap_store_do:n

首先将最后一行部分添加到行中，然后将其附加到\l__iow_wrap_tl中，带有适当的新行（带有“run-on”文本），可能会去掉其最后的空格（#1为空或__iow_wrap_trim:N）。

```
\cs_new_protected:Npn \__iow_wrap_store_do:n #1
  {
    \__kernel_tl_set:Ne \l__iow_line_tl
      { \l__iow_line_tl \l__iow_line_part_tl }
    \__kernel_tl_set:Ne \l__iow_wrap_tl
      {
        \l__iow_wrap_tl
        \l__iow_newline_tl
        #1 \l__iow_line_tl
      }
    \tl_clear:N \l__iow_line_tl
  }
```

(*__iow_wrap_store_do:n* 定义结束。)

__iow_wrap_trim:N
__iow_wrap_trim:w
__iow_wrap_trim_aux:w

如果存在，从参数中删除一个尾随的“other”空格。

```
\cs_set_protected:Npn \__iow_tmp:w #1
  {
    \cs_new:Npn \__iow_wrap_trim:N ##1
      { \exp_after:wN \__iow_wrap_trim:w ##1 \s__iow_mark #1 \s__iow_mark \s__iow_stop }
    \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \s__iow_mark
      { \__iow_wrap_trim_aux:w ##1 \s__iow_mark }
    \cs_new:Npn \__iow_wrap_trim_aux:w ##1 \s__iow_mark ##2 \s__iow_stop {##1}
  }
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
```

(*__iow_wrap_trim:N*, *__iow_wrap_trim:w*, 和 *__iow_wrap_trim_aux:w* 定义结束。)

⟨@@=file⟩

3.4 文件操作

\l__file_internal_tl

用作短期临时变量。

```
\tl_new:N \l__file_internal_tl
```

(*\l__file_internal_tl* 定义结束。)

\g_file_curr_dir_str
\g_file_curr_ext_str
\g_file_curr_name_str

当前文件的名称应始终可用：名称本身是动态设置的。

```
\str_new:N \g_file_curr_dir_str
\str_new:N \g_file_curr_ext_str
\str_new:N \g_file_curr_name_str
```

(*\g_file_curr_dir_str*, *\g_file_curr_ext_str*, 和 *\g_file_curr_name_str* 定义结束。这些变量被记录在第14页。)

\g__file_stack_seq

输入文件列表被存储为序列栈。在包模式下，我们可以从LaTeX 2ε保存的详细信息中恢复信息(我们必须在导言中以\usepackage或\RequirePackage加载)。由于LaTeX 2ε不分别存储目录和名称，我们在这里坚持相同的约定。在预加载中，\@currnamestack为空，因此被跳过。

```
\seq_new:N \g__file_stack_seq
\group_begin:
  \cs_set_protected:Npn \__file_tmp:w #1#2#3
    {
      \tl_if_blank:nTF {#1}
        {
          \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \s__file_stop
```

```
            { { } {##2} {  } }
          \seq_gput_right:Ne \g__file_stack_seq
            {
              \exp_after:wN \__file_tmp:w \tex_jobname:D
                " \tex_jobname:D " \s__file_stop
            }
        }
        {
          \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
          \__file_tmp:w
        }
    }
  \cs_if_exist:NT \@currnamestack
    {
      \tl_if_empty:NF \@currnamestack
        { \exp_after:wN \__file_tmp:w \@currnamestack }
    }
\group_end:
```

(*\g__file_stack_seq* 定义结束。)

\g__file_record_seq 用于记录单独存储的文件使用列表，因为从此列表中永远不会弹出任何内容。当前文件名应包含在文件列表中！我们最终将复制\@filelist的内容。

```
\seq_new:N \g__file_record_seq
```

(*\g__file_record_seq* 定义结束。)

\l__file_base_name_tl 用于在内部传递数据时存储基本名称和完整路径。

\l__file_full_name_tl

```
\tl_new:N \l__file_base_name_tl
\tl_new:N \l__file_full_name_tl
```

(*\l__file_base_name_tl* 和 *\l__file_full_name_tl* 定义结束。)

\l__file_dir_str 用于解析路径的变量：与上述变量相比，这些变量从未在当前模块之外使用。

\l__file_ext_str

\l__file_name_str

```
\str_new:N \l__file_dir_str
\str_new:N \l__file_ext_str
\str_new:N \l__file_name_str
```

(*\l__file_dir_str*, *\l__file_ext_str*, 和 *\l__file_name_str* 定义结束。)

\l_file_search_path_seq 当前搜索路径。

```
\seq_new:N \l_file_search_path_seq
```

(*\l_file_search_path_seq* 定义结束。这个变量被记录在第14页。)

\l__file_tmp_seq 用于逗号列表转换的临时空间。

```
\seq_new:N \l__file_tmp_seq
```

(*\l__file_tmp_seq 定义结束。*)

3.4.1 内部辅助功能

\s__file_stop 内部扫描标记。

```
\scan_new:N \s__file_stop
```

(*\s__file_stop 定义结束。*)

\q__file_nil 内部 quark 。

```
\quark_new:N \q__file_nil
```

(*\q__file_nil 定义结束。*)

__file_quark_if_nil_p:n 分支 quark 条件。

__file_quark_if_nil:n*TF*

```
\__kernel_quark_new_conditional:Nn \__file_quark_if_nil:n { TF }
```

(*__file_quark_if_nil:nTF 定义结束。*)

\q__file_recursion_tail 内部递归 quarks。

\q__file_recursion_stop

```
\quark_new:N \q__file_recursion_tail
\quark_new:N \q__file_recursion_stop
```

(*\q__file_recursion_tail 和 \q__file_recursion_stop 定义结束。*)

__file_if_recursion_tail_break:NN 查询递归 quarks 的函数。

__file_if_recursion_tail_stop_do:Nn

```
\__kernel_quark_new_test:N \__file_if_recursion_tail_stop:N
\__kernel_quark_new_test:N \__file_if_recursion_tail_stop_do:nn
```

(*__file_if_recursion_tail_break:NN 和 __file_if_recursion_tail_stop_do:Nn 定义结束。*)

__kernel_file_name_sanitize:n 扩展文件名使用\csname为基础的方法，依赖于活动字符（例如 UTF-8 字符）的正确设置，以扩展为使用\ifcsname的扩展安全版本。这比以前使用的逐个标记的方法更不保守，但速度更快。

__file_name_expand:n

__file_name_expand_cleanup:Nw

__file_name_expand_cleanup:w

__file_name_expand_end:

__file_name_expand_error:Nw

__file_name_expand_error_aux:Nw

__file_name_strip_quotes:n

__file_name_strip_quotes:nnnw

__file_name_strip_quotes:nnn

__file_name_trim_spaces:n

__file_name_trim_spaces:nw

__file_name_trim_spaces_aux:n

__file_name_trim_spaces_aux:w

```
\cs_new:Npn \__kernel_file_name_sanitize:n #1
  {
    \exp_args:Ne \__file_name_trim_spaces:n
      {
        \exp_args:Ne \__file_name_strip_quotes:n
          { \__file_name_expand:n {#1} }
      }
  }
```

我们将使用\cs:w开始扩展文件名，为了避免创建与\relax相等的 csnames 并具有“common”名称，csname 前缀为__file_name=。结尾处还有一个守卫标记，以便我们可以检查在此过程中是否有错误，并（尽力）进行优雅清理。

```
\cs_new:Npn \__file_name_expand:n #1
  {
    \exp_after:wN \__file_name_expand_cleanup:Nw
      \cs:w __file_name = #1 \cs_end:
        \__file_name_expand_end:
  }
```

构建 csname 后，我们获取它，并获取由__file_name_expand_end:定界的剩余标记。如果还有剩余标记，那么出现了问题，因此我们将调用错误程序__file_name_expand_error:Nw。如果一切按计划进行，那么在 csname 上使用\token_to_str:N，并调用__file_name_expand_cleanup:w以删除我们之前添加的前缀。__file_name_expand_cleanup:w带有前导参数，因此我们不必担心\tex_escapechar:D的值。

```
\cs_new:Npn \__file_name_expand_cleanup:Nw #1 #2 \__file_name_expand_end:
  {
    \tl_if_empty:nF {#2}
      { \__file_name_expand_error:Nw #2 \__file_name_expand_end: }
    \exp_after:wN \__file_name_expand_cleanup:w \token_to_str:N #1
  }
\exp_last_unbraced:NNNNo
\cs_new:Npn \__file_name_expand_cleanup:w #1 \tl_to_str:n { __file_name = } { }
```

在非错误情况下,__file_name_expand_end:不应该扩展。只有在文件名中存在\csname太多的情况下，它才会这样做，因此它将引发错误（在扩展时），然后插入缺少的\cs_end:和另一个__file_name_expand_end:，它将由__file_name_expand_cleanup:Nw用作定界符（或者如果还缺少\endcsname，它将再次扩展）。

```
\cs_new:Npn \__file_name_expand_end:
  {
    \msg_expandable_error:nn
      { kernel } { filename-missing-endcsname }
    \cs_end: \__file_name_expand_end:
  }
```

现在到错误情况。__file_name_expand_error:Nw添加额外的\cs_end:，以便在文件名中有额外的\csname时，__file_name_expand_error_aux:Nw抛出错误。

```
\cs_new:Npn \__file_name_expand_error:Nw #1 #2 \__file_name_expand_end:
  { \__file_name_expand_error_aux:Nw #1 #2 \cs_end: \__file_name_expand_end: }
\cs_new:Npn \__file_name_expand_error_aux:Nw #1 #2 \cs_end: #3
    \__file_name_expand_end:
```

```
  {
    \msg_expandable_error:nnff
      { kernel } { filename-chars-lost }
        { \token_to_str:N #1 } { \exp_stop_f: #2 }
  }
```

引用文件名基本上使用与 `luaquotejobname` 相同的方法：计算"标记并删除它们。

```
\cs_new:Npn \__file_name_strip_quotes:n #1
  {
    \__file_name_strip_quotes:nw { 0 }
      #1 " \q__file_recursion_tail " \q__file_recursion_stop {#1}
  }
\cs_new:Npn \__file_name_strip_quotes:nw #1#2 "
  {
    \if_meaning:w \q__file_recursion_tail #2
      \__file_name_strip_quotes_end:wnwn
    \fi:
    #2
    \__file_name_strip_quotes:nw { #1 + 1 }
  }
\cs_new:Npn \__file_name_strip_quotes_end:wnwn \fi: #1
    \__file_name_strip_quotes:nw #2 \q__file_recursion_stop #3
  {
    \fi:
    \int_if_odd:nT {#2}
      {
        \msg_expandable_error:nnn
          { kernel } { unbalanced-quote-in-filename } {#3}
      }
  }
```

从名称的开始和任何扩展的结尾修剪空格。但是，我们传递的名称可能没有扩展：这意味着我们必须查找扩展名。如果没有扩展名，我们仍然使用标准修剪函数，但有意阻止删除结尾的任何空格。

```
\cs_new:Npn \__file_name_trim_spaces:n #1
  { \__file_name_trim_spaces:nw {#1} #1 . \q__file_nil . \s__file_stop }
\cs_new:Npn \__file_name_trim_spaces:nw #1#2 . #3 . #4 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#3}
      {
        \tl_trim_spaces_apply:nN { #1 \s__file_stop }
          \__file_name_trim_spaces_aux:n
      }
```

```
      { \tl_trim_spaces:n {#1} }
  }
\cs_new:Npn \__file_name_trim_spaces_aux:n #1
  { \__file_name_trim_spaces_aux:w #1 }
\cs_new:Npn \__file_name_trim_spaces_aux:w #1 \s__file_stop {#1}
```

(*__kernel_file_name_sanitize:n 以及其它的定义结束。*)

__kernel_file_name_quote:n
__file_name_quote:nw

```
\cs_new:Npn \__kernel_file_name_quote:n #1
  { \__file_name_quote:nw {#1} #1 ~ \q__file_nil \s__file_stop }
\cs_new:Npn \__file_name_quote:nw #1 #2 ~ #3 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#3}
      { #1 }
      { "#1" }
  }
```

(*__kernel_file_name_quote:n 和 __file_name_quote:nw 定义结束。*)

\c__file_marker_tl

与重新扫描标记 token 列表的标记相同：这一对 token 不能出现在被输入的文件中。

```
\tl_const:Ne \c__file_marker_tl { : \token_to_str:N : }
```

(*\c__file_marker_tl 定义结束。*)

\file_get:nnN*TF*
\file_get:VnN*TF*
\file_get:nnN
__file_get_aux:nnN
__file_get_do:Nw

这里的方法与\tl_set_rescan:Nnn类似。文件内容被抓取为由\c__file_marker_tl定界的参数。一些微妙之处：在\if_false: … \fi:中使用花括号以处理可能的对齐制表符，使用\tracingnesting避免有关在\scantokens内关闭组的警告，并且\prg_return_true:放置在文件末尾标记之后。

```
\cs_new_protected:Npn \file_get:nnN #1#2#3
  {
    \file_get:nnNF {#1} {#2} #3
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get:nnN { V }
\prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      {
        \exp_args:NV \__file_get_aux:nnN
          \l__file_full_name_tl
          {#2} #3
        \prg_return_true:
```

```
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \file_get:nnN { V } { T , F , TF }
\cs_new_protected:Npe \__file_get_aux:nnN #1#2#3
  {
    \exp_not:N \if_false: { \exp_not:N \fi:
    \group_begin:
      \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
      \exp_not:N \exp_args:No \tex_everyeof:D
        { \exp_not:N \c__file_marker_tl }
      #2 \scan_stop:
      \exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
      \exp_not:N \exp_after:wN #3
      \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
      \exp_not:N \tex_input:D
      \sys_if_engine_luatex:TF
        { {#1} }
        { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
    \exp_not:N \if_false: } \exp_not:N \fi:
  }
\exp_args:Nno \use:nn
  { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
  { \c__file_marker_tl }
  {
    \group_end:
    \tl_set:No #1 {#2}
  }
```

(*file_get:nnNTF* 以及其它的定义结束。这些函数被记录在第18页。)

__file_size:n

在原语可用的地方复制了原语。

```
\cs_new_eq:NN \__file_size:n \tex_filesize:D
```

(*__file_size:n* 定义结束。)

\file_full_name:n
__file_full_name:n
__file_full_name_aux:n
__file_full_name_auxi:nn
__file_full_name_auxii:nn
__file_full_name_aux:Nnn
__file_full_name_slash:n
__file_full_name_slash:w
__file_full_name_aux:nN
__file_full_name_aux:nnN
__file_name_cleanup:w
__file_name_end:
__file_name_ext_check:nn
__file_name_ext_check:nnw

如果\pdffilesize原语或等效物可用，可以执行文件搜索。当然，这意味着我们需要安排这里的一切都通过扩展来完成。我们首先通过清理名称并在需要时引用来准备一切：我们可能需要删除这些引号，因此原始名称也被传递。

```
\cs_new:Npn \file_full_name:n #1
  {
    \exp_args:Ne \__file_full_name:n
      { \__kernel_file_name_sanitize:n {#1} }
```

```
  }
\cs_generate_variant:Nn \file_full_name:n { V }
```

首先，我们检查文件是否就在这里：没有映射，因此我们不需要较广泛的辅助部分的中断。我们使用原语在文件完全不存在时返回空。为了避免不必要的文件系统查找，`\pdffilesize`的结果作为参数保持可用。对于包模式，`\input@path`是一个 token list 而不是一个 sequence。

```
\cs_new:Npn \__file_full_name:n #1
  {
    \tl_if_blank:nF {#1}
      { \exp_args:Nne \__file_full_name_auxii:nn {#1} { \__file_full_name_aux:n {#1} } }
  }
```

为了避免重复读取文件，我们需要缓存加载：这很重要，因为这里的代码被*所有*文件检查使用。在LaTeX 2_ε内核中使用相同的标记，这意味着我们在`\IfFileExists`等地方获得了双重节省。由于这一切都是关于性能的，我们对条件使用底层方法。对于已经看到的文件，如果标记已被设置，将大小报告为 -1，以便与任何非缓存的文件区分开。

```
\cs_new:Npn \__file_full_name_aux:n #1
  {
    \if_cs_exist:w __file_seen_ \tl_to_str:n {#1} : \cs_end:
      -1
    \else:
      \exp_args:Ne \__file_full_name_auxi:nn { \__file_size:n {#1} } {#1}
    \fi:
  }
```

我们以后需要文件的大小，而且我们必须避免`\scan_stop:`在我们提高标志时引起问题。因此，这里有一点奇怪的吞噬。

```
\cs_new:Npn \__file_full_name_auxi:nn #1#2
  {
    \if:w \scan_stop: #1 \scan_stop:
    \else:
      \exp_after:wN \use_none:n
        \cs:w __file_seen_ \tl_to_str:n {#2} : \cs_end:
      #1
    \fi:
  }
\cs_new:Npn \__file_full_name_auxii:nn #1 #2
  {
    \tl_if_blank:nTF {#2}
      {
```

```
        \seq_map_tokens:Nn \l_file_search_path_seq
          { \__file_full_name_aux:Nnn \seq_map_break:n {#1} }
        \cs_if_exist:NT \input@path
          {
            \tl_map_tokens:Nn \input@path
              { \__file_full_name_aux:Nnn \tl_map_break:n {#1} }
          }
        \__file_name_end:
      }
      { \__file_ext_check:nn {#1} {#2} }
  }
```

此处的辅助有两个部分，因此在我们找到正确的文件时可以避免两次引用。

```
\cs_new:Npn \__file_full_name_aux:Nnn #1#2#3
  {
    \exp_args:Ne \__file_full_name_aux:nN
      { \__file_full_name_slash:n {#3} #2 }
      #1
  }
\cs_new:Npn \__file_full_name_slash:n #1
  {
    \__file_full_name_slash:nw {#1} #1 \q_nil / \q_nil / \q_nil \q_stop
  }
\cs_new:Npn \__file_full_name_slash:nw #1#2 / \q_nil / #3 \q_stop
  {
    \quark_if_nil:nTF {#3}
      { #1 / }
      { #2 / }
  }
\cs_new:Npn \__file_full_name_aux:nN #1
  { \exp_args:Nne \__file_full_name_aux:nnN {#1} { \__file_full_name_aux:n {#1} } }
\cs_new:Npn \__file_full_name_aux:nnN #1 #2 #3
  {
    \tl_if_blank:nF {#2}
      {
        #3
          {
            \__file_ext_check:nn {#1} {#2}
            \__file_name_cleanup:w
          }
      }
  }
\cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
```

```
\cs_new:Npn \__file_name_end: { }
```

由于 TEX 在没有扩展时会自动添加.tex，这里有一些清理工作要做。首先，确保我们不在目录部分，保存该部分。然后检查是否有扩展。

```
\cs_new:Npn \__file_ext_check:nn #1 #2
{ \__file_ext_check:nnw {#2} { / } #1 / \q__file_nil / \s__file_stop }
\cs_new:Npn \__file_ext_check:nnw #1 #2 #3 / #4 / #5 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#4}
      {
        \exp_args:No \__file_ext_check:nnnw
          { \use_none:n #2 } {#1} {#3} #3 . \q__file_nil . \s__file_stop
      }
      { \__file_ext_check:nnw {#1} { #2 #3 / } #4 / #5 \s__file_stop }
  }
\cs_new:Npe \__file_ext_check:nnnw #1#2#3#4 . #5 . #6 \s__file_stop
  {
    \exp_not:N \__file_quark_if_nil:nTF {#5}
      {
        \exp_not:N \__file_ext_check:nnn
          { #1 #3 \tl_to_str:n { .tex } } { #1 #3 } {#2}
      }
      { #1 #3 }
  }
\cs_new:Npn \__file_ext_check:nnn #1
  { \exp_args:Nne \__file_ext_check:nnnn {#1} { \__file_full_name_aux:n {#1} } }
\cs_new:Npn \__file_ext_check:nnnn #1#2#3#4
  {
    \tl_if_blank:nTF {#2}
      {#3}
      {
        \bool_lazy_or:nnTF
          { \int_compare_p:nNn {#4} = {#2} }
          { \int_compare_p:nNn {#2} = { -1 } }
          {#1}
          {#3}
      }
  }
```

(\file_full_name:n 以及其它的定义结束。这个函数被记录在第17页。)

\file_get_full_name:nN
\file_get_full_name:VN
\file_get_full_name:nNTF
\file_get_full_name:VNTF
__file_get_full_name_search:nN

这些函数在使用\tex_filesize:D进行文件搜索之前就已存在，因此是带有保护的get函数。为了避免不同的搜索设置，它们只是上面代码的简单封装。

```
\cs_new_protected:Npn \file_get_full_name:nN #1#2
  {
    \file_get_full_name:nNF {#1} #2
      { \tl_set:Nn #2 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get_full_name:nN { V }
\prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
  {
    \__kernel_tl_set:Ne #2
      { \file_full_name:n {#1} }
    \tl_if_empty:NTF #2
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_get_full_name:nN
  { V } { T , F ,  TF }
```

(*\file_get_full_name:nN*, *\file_get_full_name:nNTF*, 和 *__file_get_full_name_search:nN* 定义结束。这些函数被记录在第17页。)

\g__file_internal_ior

用于测试是否打开 shell 的保留流。

```
\ior_new:N \g__file_internal_ior
```

(*\g__file_internal_ior* 定义结束。)

\file_mdfive_hash:n
\file_mdfive_hash:V
\file_size:n
\file_size:V
\file_timestamp:n
\file_timestamp:V
__file_details:nn
__file_details_aux:nn
__file_mdfive_hash:n

通过扩展获取文件详情相对容易，尽管有点重复。由于 MD5 函数的语法与其他命令略有不同，需要进行一些清理。

```
\cs_new:Npn \file_size:n #1
  { \__file_details:nn {#1} { size } }
\cs_generate_variant:Nn \file_size:n { V }
\cs_new:Npn \file_timestamp:n #1
  { \__file_details:nn {#1} { moddate } }
\cs_generate_variant:Nn \file_timestamp:n { V }
\cs_new:Npn \__file_details:nn #1#2
  {
    \exp_args:Ne \__file_details_aux:nn
      { \file_full_name:n {#1} } {#2}
  }
\cs_new:Npn \__file_details_aux:nn #1#2
  {
    \tl_if_blank:nF {#1}
      { \use:c { tex_file #2 :D } {#1} }
  }
```

```
\cs_new:Npn \file_mdfive_hash:n #1
  { \exp_args:Ne \__file_mdfive_hash:n { \file_full_name:n {#1} } }
\cs_generate_variant:Nn \file_mdfive_hash:n { V }
\cs_new:Npn \__file_mdfive_hash:n #1
  { \tex_mdfivesum:D file {#1} }
```

(*\file_mdfive_hash:n* 以及其它的定义结束。这些函数被记录在第15页。)

\file_hex_dump:nnn
\file_hex_dump:Vnn
__file_hex_dump_auxi:nnn
__file_hex_dump_auxii:nnnn
__file_hex_dump_auxiii:nnnn
__file_hex_dump_auxiiv:nnn
\file_hex_dump:n
\file_hex_dump:V
__file_hex_dump:n

这些函数需要多个参数或文件大小，因此单独处理。对于LuaTEX，模拟不需要文件大小，因此我们在扩展上稍微节省了一些。

```
\cs_new:Npn \file_hex_dump:nnn #1#2#3
  {
    \exp_args:Neee \__file_hex_dump_auxi:nnn
      { \file_full_name:n {#1} }
      { \int_eval:n {#2} }
      { \int_eval:n {#3} }
  }
\cs_generate_variant:Nn \file_hex_dump:nnn { V }
\cs_new:Npn \__file_hex_dump_auxi:nnn #1#2#3
  {
    \bool_lazy_any:nF
      {
        { \tl_if_blank_p:n {#1} }
        { \int_compare_p:nNn {#2} = 0 }
        { \int_compare_p:nNn {#3} = 0 }
      }
      {
        \exp_args:Ne \__file_hex_dump_auxii:nnnn
          { \__file_details_aux:nn {#1} { size } }
          {#1} {#2} {#3}
      }
  }
\cs_new:Npn \__file_hex_dump_auxii:nnnn #1#2#3#4
  {
    \int_compare:nNnTF {#3} > 0
      { \__file_hex_dump_auxiii:nnnn {#3} }
      {
        \exp_args:Ne \__file_hex_dump_auxiii:nnnn
          { \int_eval:n { #1 + #3 } }
      }
        {#1} {#2} {#4}
  }
\cs_new:Npn \__file_hex_dump_auxiii:nnnn #1#2#3#4
```

```
  {
    \int_compare:nNnTF {#4} > 0
      { \__file_hex_dump_auxiv:nnn {#4} }
      {
        \exp_args:Ne \__file_hex_dump_auxiv:nnn
          { \int_eval:n { #2 + #4 } }
      }
      {#1} {#3}
  }
\cs_new:Npn \__file_hex_dump_auxiv:nnn #1#2#3
  {
    \tex_filedump:D
      offset ~ \int_eval:n { #2 - 1 } ~
      length ~ \int_eval:n { #1 - #2 + 1 }
      {#3}
  }
\cs_new:Npn \file_hex_dump:n #1
  { \exp_args:Ne \__file_hex_dump:n { \file_full_name:n {#1} } }
\cs_generate_variant:Nn \file_hex_dump:n { V }
\sys_if_engine_luatex:TF
  {
    \cs_new:Npn \__file_hex_dump:n #1
      {
        \tl_if_blank:nF {#1}
          { \tex_filedump:D whole {#1} {#1} }
      }
  }
  {
    \cs_new:Npn \__file_hex_dump:n #1
      {
        \tl_if_blank:nF {#1}
          { \tex_filedump:D length \tex_filesize:D {#1} {#1} }
      }
  }
```

(*file_hex_dump:nnn* 以及其它的定义结束。这些函数被记录在第15页。)

\file_get_hex_dump:nN
\file_get_hex_dump:VN
\file_get_hex_dump:nNTF
\file_get_hex_dump:VNTF
\file_get_mdfive_hash:nN
\file_get_mdfive_hash:VN
\file_get_mdfive_hash:nNTF
\file_get_mdfive_hash:VNTF
\file_get_size:nN
\file_get_size:VN
\file_get_size:nNTF
\file_get_size:VNTF
\file_get_timestamp:nN

在适当的原始支持存在的情况下，对上述功能进行非可扩展封装。

```
\cs_new_protected:Npn \file_get_hex_dump:nN #1#2
  { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_hex_dump:nN { V }
\cs_new_protected:Npn \file_get_mdfive_hash:nN #1#2
  { \file_get_mdfive_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
```

```
\cs_generate_variant:Nn \file_get_mdfive_hash:nN { V }
\cs_new_protected:Npn \file_get_size:nN #1#2
  { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_size:nN { V }
\cs_new_protected:Npn \file_get_timestamp:nN #1#2
  { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_timestamp:nN { V }
\prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { hex_dump } #2 }
\prg_generate_conditional_variant:Nnn \file_get_hex_dump:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_mdfive_hash:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { mdfive_hash } #2 }
\prg_generate_conditional_variant:Nnn \file_get_mdfive_hash:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { size } #2 }
\prg_generate_conditional_variant:Nnn \file_get_size:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { timestamp } #2 }
\prg_generate_conditional_variant:Nnn \file_get_timestamp:nN
  { V } { T , F , TF }
\cs_new_protected:Npn \__file_get_details:nnN #1#2#3
  {
    \__kernel_tl_set:Ne #3
      { \use:c { file_ #2 :n } {#1} }
    \tl_if_empty:NTF #3
      { \prg_return_false: }
      { \prg_return_true: }
  }
```

(\file_get_hex_dump:nNTF 以及其它的定义结束。这些函数被记录在第15页。)

\file_get_hex_dump:nnnN
\file_get_hex_dump:VnnN
\file_get_hex_dump:nnnNTF
\file_get_hex_dump:VnnNTF

由于有额外的参数，所以是自定义代码。

```
\cs_new_protected:Npn \file_get_hex_dump:nnnN #1#2#3#4
  {
    \file_get_hex_dump:nnnNF {#1} {#2} {#3} #4
      { \tl_set:Nn #4 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get_hex_dump:nnnN { V }
\prg_new_protected_conditional:Npnn \file_get_hex_dump:nnnN #1#2#3#4
  { T , F , TF }
```

```
  {
    \__kernel_tl_set:Ne #4
      { \file_hex_dump:nnn {#1} {#2} {#3} }
    \tl_if_empty:NTF #4
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_get_hex_dump:nnnN
  { V } { T , F , TF }
```

(*file_get_hex_dump:nnnNTF* 定义结束。这个函数被记录在第15页。)

__file_str_cmp:nn

由于我们正在进行固定长度的“大”整数比较，最容易使用字符串比较的低级行为。

```
\cs_new_eq:NN \__file_str_cmp:nn \tex_strcmp:D
```

(*__file_str_cmp:nn* 定义结束。)

\file_compare_timestamp_p:nNn
\file_compare_timestamp_p:nNV
\file_compare_timestamp_p:VNn
\file_compare_timestamp_p:VNV
\file_compare_timestamp:nNnTF
\file_compare_timestamp:nNVTF
\file_compare_timestamp:VNnTF
\file_compare_timestamp:VNVTF
__file_compare_timestamp:nnN
__file_timestamp:n

通过使用字符串比较函数的低级性质，可以进行文件日期的比较。

```
\prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
  { p , T , F , TF }
  {
    \exp_args:Nee \__file_compare_timestamp:nnN
      { \file_full_name:n {#1} }
      { \file_full_name:n {#3} }
      #2
  }
\prg_generate_conditional_variant:Nnn \file_compare_timestamp:nNn
  { nNV , V , VNV } { p , T , F , TF }
\cs_new:Npn \__file_compare_timestamp:nnN #1#2#3
  {
    \tl_if_blank:nTF {#1}
      {
        \if_charcode:w #3 <
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
      {
        \tl_if_blank:nTF {#2}
          {
            \if_charcode:w #3 >
              \prg_return_true:
```

```
            \else:
              \prg_return_false:
            \fi:
          }
          {
            \if_int_compare:w
              \__file_str_cmp:nn
                { \__file_timestamp:n {#1} }
                { \__file_timestamp:n {#2} }
                #3 \c_zero_int
              \prg_return_true:
            \else:
              \prg_return_false:
            \fi:
          }
      }
  }
\cs_new_eq:NN \__file_timestamp:n \tex_filemoddate:D
```

(__file_compare_timestamp:nNnTF, __file_compare_timestamp:nnN, 和 __file_timestamp:n 定义结束。这个函数被记录在第16页。)

\file_if_exist_p:n
\file_if_exist_p:V
\file_if_exist:nTF
\file_if_exist:VTF

这个文件存在性检测是对添加路径到文件函数的封装。如果找到文件，路径就包含了一些内容；而如果未找到文件，返回值就是空的。

```
\prg_new_conditional:Npnn \file_if_exist:n #1 { p , T , F , TF }
  {
    \tl_if_blank:eTF { \file_full_name:n {#1} }
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_if_exist:n { V } { p , T , F , TF }
```

(__file_if_exist:nTF 定义结束。这个函数被记录在第14页。)

\file_if_exist_input:n
\file_if_exist_input:V
\file_if_exist_input:nF
\file_if_exist_input:VF

文件存在性测试的文件输入。我们没有定义T或TF变体，因为放置⟨*true code*⟩的最有用的地方在与其他条件不一致。

```
\cs_new_protected:Npn \file_if_exist_input:n #1
  {
    \file_get_full_name:nNT {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
  }
\cs_generate_variant:Nn \file_if_exist_input:n { V }
\cs_new_protected:Npn \file_if_exist_input:nF #1#2
```

```
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
      {#2}
  }
\cs_generate_variant:Nn \file_if_exist_input:nF { V }
```

(*file_if_exist_input:n* 和 *\file_if_exist_input:nF* 定义结束。这些函数被记录在第18页。)

\file_input_stop: 简单的重命名。

```
\cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }
```

(*\file_input_stop:* 定义结束。这个函数被记录在第18页。)

__kernel_file_missing:n 丢失文件的错误消息，也用于\ior_open:Nn。

```
\cs_new_protected:Npn \__kernel_file_missing:n #1
  {
    \msg_error:nne { kernel } { file-not-found }
      { \__kernel_file_name_sanitize:n {#1} }
  }
```

(*__kernel_file_missing:n* 定义结束。)

\file_input:n
\file_input:V
__file_input:n
__file_input:V
__file_input_push:n
__kernel_file_input_push:n
__file_input_pop:
__kernel_file_input_pop:
__file_input_pop:nnn

以安全的方式加载文件，首先检查文件是否存在，只有在存在时才加载。将文件名推送到\g__file_stack_seq，并将其添加到文件列表，可以是\g__file_record_seq，也可以是在包模式下的\@filelist。

```
\cs_new_protected:Npn \file_input:n #1
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
      { \__kernel_file_missing:n {#1} }
  }
\cs_generate_variant:Nn \file_input:n { V }
\cs_new_protected:Npe \__file_input:n #1
  {
    \exp_not:N \clist_if_exist:NTF \exp_not:N \@filelist
      { \exp_not:N \@addtofilelist {#1} }
      { \seq_gput_right:Nn \exp_not:N \g__file_record_seq {#1} }
    \exp_not:N \__file_input_push:n {#1}
    \exp_not:N \tex_input:D
    \sys_if_engine_luatex:TF
      { {#1} }
      { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
```

```
    \exp_not:N \__file_input_pop:
  }
\cs_generate_variant:Nn \__file_input:n { V }
```

跟踪文件数据很容易：我们存储了分离的部分，这样就不需要两次解析它们。

```
\cs_new_protected:Npn \__file_input_push:n #1
  {
    \seq_gpush:Ne \g__file_stack_seq
      {
        { \g_file_curr_dir_str }
        { \g_file_curr_name_str }
        { \g_file_curr_ext_str }
      }
    \file_parse_full_name:nNNN {#1}
      \l__file_dir_str \l__file_name_str \l__file_ext_str
    \str_gset_eq:NN \g_file_curr_dir_str  \l__file_dir_str
    \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
    \str_gset_eq:NN \g_file_curr_ext_str  \l__file_ext_str
  }
\cs_new_eq:NN \__kernel_file_input_push:n \__file_input_push:n
\cs_new_protected:Npn \__file_input_pop:
  {
    \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
    \exp_after:wN \__file_input_pop:nnn \l__file_internal_tl
  }
\cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
\cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
  {
    \str_gset:Nn \g_file_curr_dir_str  {#1}
    \str_gset:Nn \g_file_curr_name_str {#2}
    \str_gset:Nn \g_file_curr_ext_str  {#3}
  }
```

(*file_input:n* 以及其它的定义结束。这个函数被记录在第18页。)

\file_input_raw:n
\file_input_raw:V
__file_input_raw:nn

无错误检查，无跟踪。

```
\cs_new:Npn \file_input_raw:n #1
  { \exp_args:Ne \__file_input_raw:nn { \file_full_name:n {#1} } {#1} }
\cs_generate_variant:Nn \file_input_raw:n { V }
\cs_new:Npe \__file_input_raw:nn #1#2
  {
    \exp_not:N \tl_if_blank:nTF {#1}
      {
```

```
        \exp_not:N \exp_args:Nnne \exp_not:N \msg_expandable_error:nnn
          { kernel } { file-not-found }
          { \exp_not:N \__kernel_file_name_sanitize:n {#2} }
      }
      {
        \exp_not:N \tex_input:D
          \sys_if_engine_luatex:TF
            { {#1} }
            { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
        }
  }
\exp_args_generate:n { nne }
```

(*\file_input_raw:n* 和 *__file_input_raw:nn* 定义结束。这个函数被记录在第18页。)

\file_parse_full_name:n
\file_parse_full_name:V
\file_parse_full_name_apply:nN
\file_parse_full_name_apply:VN

主要解析宏 \file_parse_full_name_apply:nN 将文件名 #1 通过 __kernel_file_name_sanitize:n 处理，使我们在内部以单一规范化方式处理文件。\file_parse_full_name:n 使用前者，并使用 \prg_do_nothing: 使每个名字部分保留在一对大括号中。

```
\cs_new:Npn \file_parse_full_name:n #1
  {
    \file_parse_full_name_apply:nN {#1}
      \prg_do_nothing:
  }
\cs_generate_variant:Nn \file_parse_full_name:n { V }
\cs_new:Npn \file_parse_full_name_apply:nN #1
  {
    \exp_args:Ne \__file_parse_full_name_auxi:nN
      { \__kernel_file_name_sanitize:n {#1} }
  }
\cs_generate_variant:Nn \file_parse_full_name_apply:nN { V }
```

__file_parse_full_name_area:nw 将文件名分成由 / 分隔的块，直到达到最后一个块。最后一个块是文件名加上扩展名，而在那之前的所有内容都是路径。当 __file_parse_full_name_area:nw 完成时，它在扫描标记 \s__file_stop 之后将路径保留在大括号中，并继续解析实际的文件名。

__file_parse_full_name_auxi:nN
__file_parse_full_name_area:nw

```
\cs_new:Npn \__file_parse_full_name_auxi:nN #1
  {
    \__file_parse_full_name_area:nw { } #1
      / \s__file_stop
  }
\cs_new:Npn \__file_parse_full_name_area:nw #1 #2 / #3 \s__file_stop
```

```
  {
    \tl_if_empty:nTF {#3}
      { \__file_parse_full_name_base:nw { } #2 . \s__file_stop {#1} }
      { \__file_parse_full_name_area:nw { #1 / #2 } #3 \s__file_stop }
  }
```

`\__file_parse_full_name_base:nw` 做的事情与上面大致相同，但它在每个句点处分隔块。然而，这里有一些额外的复杂性：如果 `#1` 为空，则假定扩展名实际上为空，并且文件名为 `#2`。此外，必须在 `#2` 中添加一个额外的 `.`，因为它稍后将在 `\__file_parse_full_name_tidy:nnnN` 中删除。无论如何，如果有扩展名，它将以前导 `.` 返回。

```
\cs_new:Npn \__file_parse_full_name_base:nw #1 #2 . #3 \s__file_stop
  {
    \tl_if_empty:nTF {#3}
      {
        \tl_if_empty:nTF {#1}
          {
            \tl_if_empty:nTF {#2}
              { \__file_parse_full_name_tidy:nnnN { } { } }
              { \__file_parse_full_name_tidy:nnnN { .#2 } { } }
          }
          { \__file_parse_full_name_tidy:nnnN {#1} { .#2 } }
      }
      { \__file_parse_full_name_base:nw { #1 . #2 } #3 \s__file_stop }
  }
```

__file_parse_full_name_base:nw

现在我们只需要清理一些在前面放开的位。上述两个宏中使用的循环在文件路径和名称的开头都有一个前导 `/` 和 `.`，因此这里我们需要去除它们，除非它是单独的 `/`，在这种情况下它保留原样。全部完成后，传递给 `#4`。

```
\cs_new:Npn \__file_parse_full_name_tidy:nnnN #1 #2 #3 #4
  {
    \exp_args:Nee #4
      {
        \str_if_eq:nnF {#3} { / } { \use_none:n }
        #3 \prg_do_nothing:
      }
      { \use_none:n #1 \prg_do_nothing: }
      {#2}
  }
```

__file_parse_full_name_tidy:nnnN

(*\file_parse_full_name:n* 以及其它的定义结束。这些函数被记录在第17页。)

\file_parse_full_name:nNNN
\file_parse_full_name:VNNN

```
\cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
  {
    \file_parse_full_name_apply:nN {#1}
      \__file_full_name_assign:nnnNNN #2 #3 #4
  }
\cs_new_protected:Npn \__file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
  {
    \str_set:Nn #4 {#1}
    \str_set:Nn #5 {#2}
    \str_set:Nn #6 {#3}
  }
\cs_generate_variant:Nn \file_parse_full_name:nNNN { V }
```

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

\file_show_list:
\file_log_list:
__file_list:N
__file_list_aux:n

列出所有使用的文件到日志，去除重复。在包模式下，如果\@filelist仍然定义，需要将文件名列表\@filelist 考虑在内（我们在\AtBeginDocument中捕获它到\g__file_record_seq），转换为字符串（这不影响逗号列表的逗号）。

```
\cs_new_protected:Npn \file_show_list: { \__file_list:N \msg_show:nneeee }
\cs_new_protected:Npn \file_log_list: { \__file_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__file_list:N #1
  {
    \seq_clear:N \l__file_tmp_seq
    \clist_if_exist:NT \@filelist
      {
        \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
          { \tl_to_str:N \@filelist }
      }
    \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
    \seq_remove_duplicates:N \l__file_tmp_seq
    #1 { kernel } { file-list }
      { \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n }
        { } { } { }
  }
\cs_new:Npn \__file_list_aux:n #1 { \iow_newline: #1 }
```

(*\file_show_list:* 以及其它的定义结束。这些函数被记录在第19页。)

在作为包时，需要在这里保存标准文件列表和新列表。记录在 \@filelist中的文件名在添加到\g__file_record_seq之前必须转换为字符串。

```
\cs_if_exist:NT \@filelist
  {
```

```
    \AtBeginDocument
      {
        \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
          { \tl_to_str:N \@filelist }
        \seq_gconcat:NNN
          \g__file_record_seq
          \g__file_record_seq
          \l__file_tmp_seq
      }
  }
```

3.5 GetIdInfo

\GetIdInfo
__file_id_info_auxi:w
__file_id_info_auxii:w
__file_id_info_auxiii:w

此函数从 SVN Id 行中提取文件名等信息，如 expl3.dtx 中所述。这曾经是我们在所有模块中获取版本号等信息的方式，因此它必须在 l3bootstrap 中定义。现在在我们设置了很多工具之后，定义它更方便，l3file 似乎是最合理的地方。

这里的想法是从标准的 SVN Id 行中提取出所需的信息，但要避免在检入文件时会更改的行。因此，这些行中没有包含同时包含美元符号和 Id 关键字的行的原因！

```
\cs_new_protected:Npn \GetIdInfo
  {
    \tl_clear_new:N \ExplFileDescription
    \tl_clear_new:N \ExplFileDate
    \tl_clear_new:N \ExplFileName
    \tl_clear_new:N \ExplFileExtension
    \tl_clear_new:N \ExplFileVersion
    \group_begin:
    \char_set_catcode_space:n { 32 }
    \exp_after:wN
    \group_end:
    \__file_id_info_auxi:w
  }
```

首先检查 SVN 字段是否完全为空。如果不是这种情况，还有一种情况，即使用 svn cp 创建但尚未检入的文件。这会留下一个特殊的标记-1 版本，它没有进一步的数据。正确处理这一点是使用 __file_id_info_auxii:w的原因，在行中使用空格。

```
\cs_new_protected:Npn \__file_id_info_auxi:w $ #1 $ #2
  {
    \tl_set:Nn \ExplFileDescription {#2}
    \str_if_eq:nnTF {#1} { Id }
      {
        \tl_set:Nn \ExplFileDate { 0000/00/00 }
```

```
        \tl_set:Nn \ExplFileName { [unknown] }
        \tl_set:Nn \ExplFileExtension { [unknown~extension] }
        \tl_set:Nn \ExplFileVersion {-1}
      }
      { \__file_id_info_auxii:w #1 ~ \s__file_stop }
  }
```

在这里，#1是Id，#2是文件名，#3是扩展名，#4是版本，#5是检入日期，#6是检入时间和用户，加上一些尾随空格。如果#4是标记 -1值，那么#5和#6都是空的。

```
\cs_new_protected:Npn \__file_id_info_auxii:w
    #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
  {
    \tl_set:Nn \ExplFileName {#2}
    \tl_set:Nn \ExplFileExtension {#3}
    \tl_set:Nn \ExplFileVersion {#4}
    \str_if_eq:nnTF {#4} {-1}
      { \tl_set:Nn \ExplFileDate { 0000/00/00 } }
      { \__file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop }
  }
```

将 SVN 风格的日期转换为 LaTeX 风格的日期。

```
\cs_new_protected:Npn \__file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
  { \tl_set:Nn \ExplFileDate { #1/#2/#3 } }
```

(*\GetIdInfo* 以及其它的定义结束。这个函数被记录在第??页。)

3.6 检查内核依赖版本

__kernel_dependency_version_check:Nn
__kernel_dependency_version_check:nn
__file_kernel_dependency_compare:nnn
__file_parse_version:w

此函数负责检查 LaTeX3 内核的依赖关系是否与 LaTeX 2$_\varepsilon$内核中预加载的版本匹配。如果版本不匹配，该函数尝试通过搜索可能的格式文件来解释原因。

该函数首先检查内核日期是否定义，如果未定义，则使用零强制错误路线。然后将内核日期与请求的日期进行比较（通常是依赖项的打包日期）。如果内核日期小于所需日期，则是一个错误，加载应该中止。

```
\cs_new_protected:Npn \__kernel_dependency_version_check:Nn #1
  { \exp_args:NV \__kernel_dependency_version_check:nn #1 }
\cs_new_protected:Npn \__kernel_dependency_version_check:nn #1
  {
    \cs_if_exist:NTF \c__kernel_expl_date_tl
      {
        \exp_args:NV \__file_kernel_dependency_compare:nnn
          \c__kernel_expl_date_tl {#1}
      }
```

```
      { \__file_kernel_dependency_compare:nnn { 0000-00-00 } {#1} }
  }
\cs_new_protected:Npn \__file_kernel_dependency_compare:nnn #1 #2 #3
  {
    \int_compare:nNnT
        { \__file_parse_version:w #1 \s__file_stop } <
        { \__file_parse_version:w #2 \s__file_stop }
      { \__file_mismatched_dependency_error:nn {#2} {#3} }
  }
\cs_new:Npn \__file_parse_version:w #1 - #2 - #3 \s__file_stop {#1#2#3}
```

如果版本不同，则尝试为用户提供一些指导。此函数首先取得引擎名称`\c_sys_engine_str`，并将`tex`替换为`latex`，然后构建如下形式的命令：kpsewhich –all –engine=⟨*engine*⟩ ⟨*format*⟩[-dev].fmt 以查询可用的格式文件。打开一个 shell，并将每行读入一个序列。

__file_mismatched_dependency_error:nn

```
\cs_new_protected:Npn \__file_mismatched_dependency_error:nn #1 #2
  {
    \exp_args:NNe \ior_shell_open:Nn \g__file_internal_ior
      {
        kpsewhich ~ --all ~
          --engine = \c_sys_engine_exec_str
          \c_space_tl \c_sys_engine_format_str
            \bool_lazy_and:nnT
                { \tl_if_exist_p:N \development@branch@name }
                { ! \tl_if_empty_p:N \development@branch@name }
              { -dev } .fmt
      }
    \seq_clear:N \l__file_tmp_seq
    \ior_map_inline:Nn \g__file_internal_ior
      { \seq_put_right:Nn \l__file_tmp_seq {##1} }
    \ior_close:N \g__file_internal_ior
    \msg_error:nnnn { kernel } { mismatched-support-file }
      {#1} {#2}
```

最后，结束当前文件。

```
    \tex_endinput:D
  }
```

现在定义实际的错误消息：

```
\msg_new:nnnn { kernel } { mismatched-support-file }
  {
    Mismatched~LaTeX~support~files~detected. \\
    Loading~'#2'~aborted!
```

\c__kernel_expl_date_tl可能不存在，因为是较旧的格式，所以只有在存在标记的令牌列表时才打印日期：

```
    \tl_if_exist:NT \c__kernel_expl_date_tl
      {
        \\ \\
        The~L3~programming~layer~in~the~LaTeX~format \\
        is~dated~\c__kernel_expl_date_tl,~but~in~your~TeX~
        tree~the~files~require \\ at~least~#1.
      }
  }
  {
```

包含格式文件的序列应该恰好有一项：当前运行的格式文件。如果是这种情况，则错误的原因不是这个，因此打印一些可能原因的通用帮助。如果找到多个格式文件，则将列表打印给用户，指示系统中有什么，用户树中有什么。

```
    \int_compare:nNnTF { \seq_count:N \l__file_tmp_seq } > 1
      {
        The~cause~seems~to~be~an~old~format~file~in~the~user~tree. \\
        LaTeX~found~these~files:
        \seq_map_tokens:Nn \l__file_tmp_seq { \\~--~\use:n } \\
        Try~deleting~the~file~in~the~user~tree~then~run~LaTeX~again.
      }
      {
        The~most~likely~causes~are:
        \\~--~A~recent~format~generation~failed;
        \\~--~A~stray~format~file~in~the~user~tree~which~needs~
             to~be~removed~or~rebuilt;
        \\~--~You~are~running~a~manually~installed~version~of~#2 \\
        \ \ \ which~is~incompatible~with~the~version~in~LaTeX. \\
      }
    \\
    LaTeX~will~abort~loading~the~incompatible~support~files~
    but~this~may~lead~to \\ later~errors.~Please~ensure~that~
    your~LaTeX~format~is~correctly~regenerated.
  }
```

(*__kernel_dependency_version_check:Nn 以及其它的定义结束。*)

3.7 消息

```
\msg_new:nnnn { kernel } { file-not-found }
  { File~'#1'~not~found. }
  {
```

```
    The~requested~file~could~not~be~found~in~the~current~directory,~
    in~the~TeX~search~path~or~in~the~LaTeX~search~path.
  }
\msg_new:nnn { kernel } { file-list }
  {
    >~File~List~<
    #1 \\
    ..............
  }
\msg_new:nnnn { kernel } { filename-chars-lost }
  { #1~invalid~in~file~name.~Lost:~#2. }
  {
    There~was~an~invalid~token~in~the~file~name~that~caused~
    the~characters~following~it~to~be~lost.
  }
\msg_new:nnnn { kernel } { filename-missing-endcsname }
  { Missing~\iow_char:N\\endcsname~inserted~in~filename. }
  {
    The~file~name~had~more~\iow_char:N\\csname~commands~than~
    \iow_char:N\\endcsname~ones.~LaTeX~will~add~the~missing~
    \iow_char:N\\endcsname~and~try~to~continue~as~best~as~it~can.
  }
\msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
  { Unbalanced~quotes~in~file~name~'#1'. }
  {
    File~names~must~contain~balanced~numbers~of~quotes~(").
  }
\msg_new:nnnn { kernel } { iow-indent }
  { Only~#1 allows~#2 }
  {
    The~command~#2 can~only~be~used~in~messages~
    which~will~be~wrapped~using~#1.
    \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'#3'. }
  }
```

3.8 从先前模块延迟的函数

<@@=sys>

\c_sys_platform_str

在LuaTEX上检测平台很容易：对于其他引擎，我们使用两种常见情况的特殊空文件。虽然可以进一步探测（参见 platform 宏包），但这需要 shell escape，并且似乎不太可能有用。在这里设置，因为它需要文件搜索。

```
\sys_if_engine_luatex:TF
```

```
  {
    \str_const:Ne \c_sys_platform_str
      { \tex_directlua:D { tex.print(os.type) } }
  }
  {
    \file_if_exist:nTF { nul: }
      {
        \file_if_exist:nF { /dev/null }
          { \str_const:Nn \c_sys_platform_str { windows } }
      }
      {
        \file_if_exist:nT { /dev/null }
          { \str_const:Nn \c_sys_platform_str { unix } }
      }
  }
\cs_if_exist:NF \c_sys_platform_str
  { \str_const:Nn \c_sys_platform_str { unknown }  }
```

(*\c_sys_platform_str* 定义结束。这个变量被记录在第??页。)

\sys_if_platform_unix_p:
\sys_if_platform_unix:*TF*
\sys_if_platform_windows_p:
\sys_if_platform_windows:*TF*

现在我们可以设置测试。

```
\clist_map_inline:nn { unix , windows }
  {
    \__file_const:nn { sys_if_platform_ #1 }
      { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
  }
```

(*\sys_if_platform_unix:TF* 和 *\sys_if_platform_windows:TF* 定义结束。这些函数被记录在第??页。)

⟨/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。

F

file 命令:

G

group 命令:

I

if 命令:

int 命令:

ior 内部命令:

ior 命令:

iow 内部命令:

K

L

M

O

P

Q

S

T

U