DocStrip 程序 *

Frank Mittelbach　Denys Duchier　Johannes Braams
Marcin Woliński　Mark Wooding

张泓知 翻译

2023 年 12 月 12 日

摘要

本文档描述了 DocStrip 程序的实现。该程序最初由 Frank Mittelbach 开发，用于配合他的 `doc.sty`，使得在 LaTeX 中可以进行文学编程。Denys Duchier 对其进行了重写，使其可以在 TeX 或 LaTeX 中运行，并允许在条件 guard 中使用完整的布尔表达式，而不仅限于逗号分隔的列表。Johannes Braams 重新整合了这两个实现，对代码进行了文档化和调试。

在 1995 年 9 月，Marcin Woliński 改变了程序的许多部分，利用了 TeX 同时写入多个文件的能力，以避免重复读取源文件。版本 2.3 的性能提升付出了与一些更不常见的操作系统的兼容性代价，这些系统限制了进程可以保持打开的文件数量。这在 1996 年 9 月由 Mark Wooding 进行了修正，他的修改被 Marcin Woliński “创造性地合并”，同时在批处理文件处理、引导处理和引入“纯文本模式”方面进行了修改。之后，David Carlisle 将新版本合并到了 LaTeX 源文件中，并进行了一些其他更改，主要是使 DocStrip 在 initex 下工作，并删除了批处理文件中必须包含 `\def\batchfile{...}` 的需要。

*此文件版本号为 v2.6b，上次修订日期为 2022-09-03，文档日期为 2023-10-10。

目录

1 介绍

1.1 DocStrip 程序的由来

当 Frank Mittelbach 创建了 doc 包时，他发明了一种将 TEX 代码和其文档结合起来的方法。从那时起，基本上可以在 TEX 中进行文学编程。

这种编写 TEX 程序的方式显然有很大的优势，特别是当程序变得比几个宏还要大时。然而，有一个缺点，即此类程序运行时间可能比预期更长，因为 TEX 是一个解释器，必须针对程序文件的每一行决定如何处理它。因此，通过

删除所有注释，可以加快 TEX 程序的运行速度。

从 TEX 程序中删除注释会引入一个新问题。现在我们有两个版本的程序，它们都必须进行维护。因此，最好能够自动删除注释，而不是手工操作。因此，我们需要一个程序来从 TEX 程序中删除注释。这可以用任何高级语言来编程，但也许不是每个人都有合适的编译器来编译该程序。每个想要从 TEX 程序中删除注释的人都有 TEX。因此，DocStrip 程序完全是用 TEX 实现的。

1.2 DocStrip 程序的功能

创建 DocStrip 程序以从 TEX 程序中删除注释行 [1] 后，进行更多操作变得可行。

不仅可以删除注释，还可以根据某些条件包含代码的部分。另外，将 TEX 程序源代码分割成几个较小的文件并在后来组合成一个“可执行”文件，也是一个不错的选择。

所有这些愿望都已在 DocStrip 程序中实现。

2 如何使用 DocStrip 程序

有多种使用 DocStrip 程序的方法：

1. 使用 DocStrip 的常规方法是编写一个 *batch file*，使其可以直接由 TEX 处理。批处理文件应包含下面描述的控制 DocStrip 程序的命令。这使您可以设置一个发行版，其中用户只需运行

   TEX ⟨*batch file*⟩

   就可以从发行版源文件生成可执行文件。大多数 LATEX 发行版都是以此方式打包的。要生成此类批处理文件，请在“批处理文件”中包含一条指令，指示 TEX 读取 `docstrip.tex`。此类文件的开头将如下所示：

   ```
   \input docstrip
   ...
   ```

   按照惯例，批处理文件的扩展名应为 `.ins` 。但实际上，这些天 DocStrip 实际上可以使用任何扩展名。

2. 或者，您可以指示 TEX 读取文件 `docstrip.tex`，然后看看会发生什么。TEX 将向您询问一些有关您要处理的文件的问题。当您回答完这些问题后，它会执行它的任务并从您的 TEX 代码中删除注释。

[1] 请注意，仅删除注释行，即以单个 % 字符开头的行；所有其他注释保留在代码中。

3 配置 DocStrip

3.1 选择输出目录

由于希望简化 LaTeX 2ε 的重新安装并支持具有目录中文件数量上限的操作系统，DocStrip 现在允许安装脚本为其创建的文件指定输出目录。我们建议在此使用相对于 texmf 的 TDS（TeX 目录结构）目录名称。然而，这些名称应被视为标签，而不是目录的实际名称。它们将根据包含在名为 docstrip.cfg 的配置文件中的命令转换为实际的系统相关路径名。

配置文件在 DocStrip 开始处理任何批处理文件命令之前被读取。

如果此文件不存在，DocStrip 将使用一些默认设置，确保文件仅写入当前目录。但是通过使用此配置文件，站点维护者可以“启用”DocStrip 的功能，使文件被写入到备选目录。

\usedir 使用此宏包作者可以告知文件应安装到何处。在该声明范围内生成的所有 \file 都将写入由其参数指定的目录。例如，在 LaTeX 2ε 安装中，使用以下声明：

```
\usedir{tex/latex/base}
\usedir{makeindex}
```

而标准宏包使用

```
\usedir{tex/latex/tools}
\usedir{tex/latex/babel}
```

等等。

\showdirectory 用于在消息中显示目录名称。如果某个标签未定义，它会扩展为 UNDEFINED (label is ...)，否则为目录名称。对于每个安装脚本，在启动时显示将要使用的所有目录列表并要求用户确认可能是个不错的主意。

上述宏由宏包/安装脚本作者使用。以下宏由系统管理员在配置文件 docstrip.cfg 中使用，用于描述其本地目录结构。

\BaseDirectory 此宏是管理员表达“是的，我想要使用你的目录支持”的方式。除非你的配置文件调用了此宏，否则 DocStrip 只会写入当前目录。（这意味着除非你告诉它，否则 DocStrip 不会写入随机目录，这很好。）使用此宏，您可以为与 TeX 相关的内容指定一个基本目录。例如，对于许多 Unix 系统，这将是

```
\BaseDirectory{/usr/local/lib/texmf}
```

而对于标准 emTeX 安装，则为

```
\BaseDirectory{c:/emtex}
```

\DeclareDir 在指定了基本目录之后，您应该告诉 DocStrip 如何解释 \usedir 命令中使用的标签。这可以通过 \DeclareDir 实现，它有两个参数。第一个是标签，第二个是相对于基本目录的实际目录名称。例如，要让 DocStrip 使用标准 emTEX 目录，可以这样做：

```
\BaseDirectory{c:/emtex}
\DeclareDir{tex/latex/base}{texinput/latex2e}
\DeclareDir{tex/latex/tools}{texinput/tools}
\DeclareDir{makeindex}{idxstyle}
```

这将导致基本的 LaTeX 文件和字体描述被写入目录 c:\emtex\texinput\latex2e，tools 包的文件被写入 c:\emtex\texinput\tools，makeindex 文件被写入 c:\emtex\idxstyle 。

有时希望将某些文件放在基本目录之外。因此，\DeclareDir 有一种星号形式，用于指定绝对路径名。例如，可以这样说：

```
\DeclareDir*{makeindex}{d:/tools/texindex/styles}
```

\UseTDS 符合 TDS 标准的系统用户可能会问，“我真的需要在我的配置文件中放置一打行像下面这样的内容吗？

```
\DeclareDir{tex/latex/base}{tex/latex/base}
```

答案是 \UseTDS 。这个宏会使 DocStrip 对于您没有用 \DeclareDir 覆盖的任何目录都使用标签本身。默认行为是在未定义的标签上引发错误，因为一些用户可能希望准确知道文件的位置，而不允许 DocStrip 随意写入文件。然而，我（MW）觉得这非常酷，我的配置文件只有这么几行（我在 Linux 下运行 teTEX）：

```
\BaseDirectory{/usr/local/teTeX/texmf}
\UseTDS
```

重要的是要注意，无法在 TEX 中创建新目录。因此，无论您如何配置 DocStrip，都需要在运行安装程序之前创建所有必需的目录。作者可能希望在每次安装脚本开始时显示将要使用的目录列表，并询问用户是否确定所有这些目录都存在。

由于文件名语法是与操作系统相关的，DocStrip 试图从当前目录语法中猜测它。它应该能成功识别 Unix、MSDOS、Macintosh 和 VMS 的语法。但是，只有在与 LATEX 一起使用时，DocStrip 才会最初了解当前目录的语法[2]。如果您经常使用的格式不是 LATEX，您应该在文件 docstrip.cfg 中的开始部分定

[2]除了处理 LATEX 分发的主要 unpack.ins 批处理文件之外，该文件采取了特殊措施，以便 initex 可以了解目录语法。

义 \WriteToDir 。例如，在 MSDOS/Unix 上是 \def\WriteToDir{./}，在 Macintosh 上是 \def\WriteToDir{:}，在 VMS 上是 \def\WriteToDir{[]} 。

如果您的系统需要完全不同的配置，可以在 docstrip.cfg 中定义宏 \dirsep 和 \makepathname 。检查它们在实现部分的定义。如果希望使用完全不同的方案将 \usedir 标签转换为目录名称，请尝试重新定义宏 \usedir 。

3.2 设置最大流的数量

\maxfiles 为了支持一些更晦涩的操作系统，程序能打开的文件数量有一定限制。通过 \maxfiles 宏，可以向 DocStrip 表达这种限制。如果 DocStrip 允许打开的流的数量是 n，您的配置文件可以写成 \maxfiles{n}，这样 DocStrip 就不会尝试打开比这更多的文件。请注意，此限制不包括已经打开的文件。通常会有两个文件已经打开：您启动的安装脚本和它所包含的文件 docstrip.tex；您必须自己注意这一点。在命中某种最大限制之前，DocStrip 假设它可以打开至少四个文件：如果情况并非如此，那就是真正的问题所在。

\maxoutfiles 也许与其限制 TEX 可打开的文件数量，不如限制它可以写入的文件数量（例如，TEX 本身限制了一次性写入的文件数量为 16 个）。通过在配置文件中使用 \maxoutfiles{m} 可以表达这种情况。您必须能够同时打开至少一个输出文件；否则 DocStrip 就无法做任何事情。

这两种选项通常会放在 docstrip.cfg 文件中。

4 用户界面

4.1 主程序

\processbatchFile ‘主程序’ 以尝试处理批处理文件开始，通过调用宏 \processbatchFile 来完成此操作。它计算它处理的批处理文件数量，因此，在调用 \processbatchFile 后，如果处理的文件数量仍然为零，则可以采取适当的操作。

\interactive 当没有处理批处理文件时，调用宏 \interactive 。它提示用户提供信息。首先确定输入和输出文件的扩展名。然后询问有关可选代码的问题，最后用户可以提供要处理的文件列表。

\ReportTotals 当在 DocStrip-程序中包含 stats 选项时，它会记录处理的文件和行数。同时记录删除和传递的注释数，以及传递到输出的代码行数。宏 \ReportTotals 显示了这些信息的摘要。

4.2 批处理文件命令

本节描述的命令可用于构建 TEX 的批处理文件。

\input 所有 DocStrip 批处理文件应以以下行开始：`\input docstrip`

不要使用 LATEX 语法 `\input{docstrip}`，因为批处理文件可能与 plain TEX 或 iniTEX 一起使用。您可能会发现旧批处理文件始终在输入之前有一行 `\def\batchfile{`⟨*filename*⟩`}` 。虽然仍然支持这种用法，但现在已不鼓励使用，因为它会导致 TEX 重新输入相同的文件，消耗了它有限的输入流之一。

\endbatchfile 所有批处理文件应以此命令结束。文件中此命令后的任何行都将被忽略。在旧文件中，如果以 `\def\batchfile{`...开始，则此命令是可选的，但是建议始终使用。如果批处理文件中省略了此命令，则通常 TEX 将进入交互式 * 提示符，因此您可以通过在此提示符中键入 `\endbatchfile` 来停止 DocStrip。

\generate \file \from 构建 DocStrip 命令文件的主要原因是描述应该从哪些源文件生成文件，以及应该包括哪些可选（'guarded'）代码片段。宏 `\generate` 用于向 TEX 提供这些信息。其语法如下：

```
\generate{[\file{⟨output⟩}{[\from{⟨input⟩}{⟨optionlist⟩}]*}]*}
```

其中 ⟨*output*⟩ 和 ⟨*input*⟩ 是适合您计算机系统的普通文件规范。⟨*optionlist*⟩ 是一个用逗号分隔的 '选项' 列表，指定了应该在 ⟨*output*⟩ 中包括 ⟨*input*⟩ 中的哪些可选代码片段。`\generate` 的参数可以包含一些局部声明（例如下面描述的 `\use...` 命令），这些声明将适用于之后的所有 `\file` 。`\generate` 的参数在组内执行，因此当 `\generate` 结束时，所有局部声明都会被取消。

可以指定多个输入文件，每个文件都有自己的 ⟨*optionlist*⟩。这可以通过记号 [...]* 来表示。此外，在一个 `\generate` 子句中可以有多个 `\file` 规范。这意味着应该在读取每个 ⟨*input*⟩ 文件时生成所有这些 ⟨*output*⟩ 文件。输入文件按照在此子句中首次出现的顺序读取。例如，

```
\generate{\file{p1.sty}{\from{s1.dtx}{foo,bar}}
          \file{p2.sty}{\from{s2.dtx}{baz}
                        \from{s3.dtx}{baz}}
          \file{p3.sty}{\from{s1.dtx}{zip}
                        \from{s2.dtx}{zip}}
         }
```

将导致 DocStrip 读取文件 `s1.dtx`、`s2.dtx`、`s3.dtx`（按照该顺序）并生成文件 `p1.sty`、`p2.sty`、`p3.sty`。

限制同时打开最多 16 个输出流并不意味着你只能用一个 `\generate` 命令生成最多 16 个文件。在上面的例子中，只需要 2 个流，因为在处理 `s1.dtx`

时只生成了 p1.sty 和 p3.sty；在读取 s2.dtx 时只生成了 p2.sty 和 p3.sty；而在读取 s3.dtx 时只有 p2.sty 文件。然而，下面的例子需要 3 个流：

```
\generate{\file{p1.sty}{\from{s1.dtx}{foo,bar}}
          \file{p2.sty}{\from{s2.dtx}{baz}
                        \from{s3.dtx}{baz}}
          \file{p3.sty}{\from{s1.dtx}{zip}
                        \from{s3.dtx}{zip}}
         }
```

尽管在读取 s2.dtx 文件时并没有写入 p3.sty，但它必须保持打开状态，因为稍后 s3.dtx 的某些部分将会写入其中。

有时候通过一次读取所有源文件来创建文件是不可能的。考虑以下例子：

```
\generate{\file{p1.sty}{\from{s1.dtx}{head}
                        \from{s2.dtx}{foo}
                        \from{s1.dtx}{tail}}
          \file{s1.drv}{\from{s1.dtx}{driver}}
         }
```

要生成 p1.sty 文件，必须对 s1.dtx 进行两次读取：第一次读取时带有选项 head，然后读取文件 s2.dtx，然后再次读取 s1.dtx，这次带有选项 tail。DocStrip 可以正确处理这种情况：如果在一个 \file 声明内有多个相同输入文件的 \from，那么该文件将被多次读取。

如果一个 \file 声明中 \from 的顺序与之前 \file 确定的输入文件顺序不匹配，DocStrip 将会报错并中止。然后，你可以阅读下一节，或者放弃并将该文件放入单独的 \generate 中（但那样源文件将再次被读取）。

对于急切的人 尝试以下算法：找到从最多源文件生成的文件，以此文件和其源文件按正确顺序编写 \generate 子句。取出其他需要生成的文件，并检查它们是否不违反第一个文件源文件的顺序。如果这样不行，请阅读下一节。

对于数学家 “文件 A 必须在文件 B 之前读取”是所有源文件集合上的偏序关系。每个 \from 子句都向这个顺序添加了一个链。你需要做的是执行拓扑排序，即将偏序扩展为线性排序。完成后，只需按照首次出现在子句中的顺序列出你的源文件在 \generate 中，使其顺序与线性顺序相匹配。如果无法实现此目标，请阅读下一段。（也许未来的 DocStrip 版本将自动执行此排序，因此所有这些问题都将消失。）

对于那些必须了解所有内容的人 有一种特殊情况，无法实现源文件的正确读取顺序。假设你需要生成两个文件，第一个文件来自 s1.dtx 和 s3.dtx（按照那个顺序），第二个文件来自 s2.dtx 和 s3.dtx。无论如何指定，文件将会按照 s1 s3 s2 或 s2 s3 s1 的顺序读取。解决方法的关键是神奇的宏 \needed，它将一个文件标记为需要输入但不将任何输出从它导向当前的 \file 。在我们的例子中，正确的规范是：

```
\generate{\file{p1.sty}{\from{s1.dtx}{foo}
                        \needed{s2.dtx}
                        \from{s3.dtx}{bar}}
          \file{p2.sty}{\from{s2.dtx}{zip}
                        \from{s3.dtx}{zap}}
         }
```

\askforoverwritetrue
\askforoverwritefalse

这些宏指定了如果要生成的文件已经存在时应该发生什么。如果 \askforoverwritetrue 处于激活状态（默认情况下），会询问用户是否应该覆盖文件。但是如果已经使用了 \askforoverwritefalse，则现有文件将会被静默地覆盖。这些开关是局部的，可以在文件的任何地方发出，甚至在 \generate 子句内部（但在 \file 之间）。

\askonceonly

你可能不想在批处理文件中设置 \askforoverwritefalse，因为这意味着始终可以悄悄地覆盖其他人的文件。然而对于大型安装（如基础 LaTeX 发行版），单独询问用户数百个文件是否覆盖并不是很有帮助。因此批处理文件可以指定 \askonceonly 。这意味着在批处理文件第一次询问用户问题后，用户可以选择更改行为，以便将“是”自动应用于所有未来的问题。这适用于 DocStrip 命令 \Ask 的任何使用，包括但不限于由 \askforoverwritetrue 控制的文件覆盖问题。

\preamble
\endpreamble
\postamble
\endpostamble

可以向 DocStrip 程序的输出添加多行信息。要添加到输出文件开头的信息应列在 \preamble 和 \endpreamble 命令之间；要添加到输出文件末尾的行应列在 \postamble 和 \endpostamble 命令之间。对于 DocStrip 发现的所有前言和后言内容，都会写入输出文件，但前面会加上 \MetaPrefix 的值（默认是两个% 字符）。如果在这些行中包含 ^^J 字符，那么与其在同一行上的所有内容都会被写入输出文件的新行中。这个“特性”可以用来向剥离后的文件中添加 \typeout 或 \message 。

\declarepreamble
\declarepostamble
\usepreamble
\usepostamble
\nopreamble
\nopostamble

有时候，对于一个更大的包中的不同文件，希望拥有不同的前言是很理想的（例如，因为其中一些是可定制的配置文件，需要标记为这样）。在这种情况下，可以使用 \declarepreamble\somename，然后输入你的前言，用 \endpreamble 结束，稍后使用 \usepreamble\somename 切换到这个前言。如果不想使用任何前言，可以使用 \nopreamble 命令。此命令相当于说 \usepreamble\empty

。同样的机制也适用于后言，\use... 声明是局部的，可以出现在 \generate 内部。

命令 \preamble 和 \postamble 定义并激活名为 \defaultpreamble 和 \defaultpostamble 的前言（后文）。

\batchinput 批处理文件命令可以放入多个批处理文件中，然后从主批处理文件中执行这些文件。例如，如果一个分发包含几个不同的部分，这是很有用的。你可以为每个部分编写单独的批处理文件，并且还有一个主文件，简单地调用部分的批处理文件。为此，可以使用命令 \batchinput{⟨*file*⟩} 调用主文件中的单独批处理文件。不要使用 \input 来实现这个目的，这个命令只应该用于调用前面解释过的 DocStrip 程序，并且在用于其他目的时会被忽略。

\ifToplevel 当批处理文件被嵌套时，你可能希望在较低级别的批处理文件中抑制某些命令，比如终端消息。为此，可以使用 \ifToplevel 命令，它仅在当前批处理文件是最外层文件时执行其参数。确保将参数的左花括号放在与命令本身相同的行上，否则 DocStrip 程序会感到困惑。

\showprogress 当在 DocStrip 中包含 stats 选项时，它可以在处理输入文件的每一行时
\keepsilent 向终端写入消息。这条消息由一个单个字符组成，表示特定行的类型。我们使用以下字符：

- % 每当输入行是注释时，在终端上写入 %-字符。
- . 每当遇到一个代码行时，在终端上写入一个 .-字符。
- / 当输入文件中出现一系列空行时，至多保留其中的一行。DocStrip 程序使用 /-字符表示删除的空行。
- < 当在输入中发现一个 'guard line' 并且它开始一个可选包含的代码块时，在终端上通过显示 <-字符来表示，并伴随着 guard 的布尔表达式。
- > 条件包含的代码块结束时，通过显示 >-字符来表示。

当包含 stats 选项时，默认情况下会打开此功能，否则会关闭。可以使用命令 \showprogress 和 \keepsilent 来切换此功能。

4.2.1 支持旧接口

\generateFile 以下是指定要生成的文件的旧语法。它只允许指定一个输出文件。

\generateFile{⟨*output*⟩}{⟨*ask*⟩}{[\from{⟨*input*⟩}{⟨*optionlist*⟩}]*}

其中 ⟨*output*⟩、⟨*input*⟩ 和 ⟨*optionlist*⟩ 的含义与 \generate 相同。通过 ⟨*ask*⟩，你可以指示 TEX 是静默地覆盖先前存在的文件（f），还是发出警告并询问是否应该覆盖现有文件（t）（它会覆盖 \askforoverwrite 设置）。

\include \processFile

早期版本的 DocStrip 程序支持一种不同类型的命令告诉 TEX 要做什么。这个命令比 \generateFile 功能较弱；当 ⟨*output*⟩ 是由一个 ⟨*input*⟩ 创建时可以使用。语法如下：

```
\include{⟨optionlist⟩}
\processFile{⟨name⟩}{⟨inext⟩}{⟨outext⟩}{⟨ask⟩}
```

这个命令基于文件名由两部分构成的环境，即名称和扩展名，用点分隔。此命令的语法假定 ⟨*input*⟩ 和 ⟨*output*⟩ 共享相同的名称，只在扩展名上有所不同。为了向后兼容旧版 DocStrip，保留了此命令，但不鼓励使用。

5 代码的条件包含

当你使用 DocStrip 程序剥离 TEX 宏文件中的注释时，你有可能从一个文档文件中生成多个剥离后的宏文件。这是通过对可选代码的支持实现的。在文档文件中，可选代码是通过一个 'guard' 来标记的。

guard 是一个布尔表达式，它被包含在 < 和 > 中。它还必须紧跟在行首的 % 后面。例如：

```
...
%<bool>\TeX code
...
```

在这个例子中，如果在 \generateFile 命令的 ⟨*optionlist*⟩ 中存在选项 bool，那么这行代码将被包含在 ⟨*output*⟩ 中。

布尔表达式的语法是：

⟨*Expression*⟩ ::= ⟨*Secondary*⟩ [{|, ,} ⟨*Secondary*⟩]*

⟨*Secondary*⟩ ::= ⟨*Primary*⟩ [& ⟨*Primary*⟩]*

⟨*Primary*⟩ ::= ⟨*Terminal*⟩ | !⟨*Primary*⟩ | (⟨*Expression*⟩)

| 代表析取，& 代表合取，! 代表否定。而 ⟨*Terminal*⟩ 是任意的字母序列当且仅当在必须包含的选项列表中评估为 ⟨*true*⟩ （[3]）。

两种类型的可选代码受到支持：一种是可以放在一行文本上的可选代码，就像上面的例子一样；另一种是可以有可选代码块。

为了区分这两种可选代码，引入了 'guard 修饰符'。它是紧跟在 guard 的 < 后面的一个字符。它可以是 * 表示代码块的开始，也可以是 / 表示代码块的结束[4]。代码块的开始和结束 guard 必须单独占据一行。

当一个代码块不被包含时，该块内出现的任何 guard 都不会被评估。

[3] iff 代表 "当且仅当"

[4] 为了与早期版本的 DocStrip 兼容，也支持 + 和 - 作为 'guard 修饰符'。但是，与先前行为相比，与一个 + 修饰的 guard 对应的行不会包含在求值为假的 guard 块内，这里存在不兼容。

6 内部函数和变量

LaTeX 开发的一个重要考虑因素是分离公共函数和内部函数。对于一个模块私有的函数和变量不应该被任何其他模块使用或修改。由于 TeX 没有任何正式的命名空间系统，这需要一种约定来指示哪些函数在代码级别的模块中是公共的，哪些是私有的。

使用 DocStrip 可以使用“双部分”系统来指示内部函数。在 `.dtx` 文件中，内部函数可以用 `@@` 代替模块名称，例如：

```
\cs_new_protected:Npn \@@_some_function:nn #1#2
  {
    % Some code here
  }
\tl_new:N \l_@@_internal_tl
```

要使用 DocStrip 提取代码，原始的 ‘guard’ 机制被扩展，引入了语法 `%<@@=`⟨*module*⟩`>`。当提取代码时，⟨*module*⟩ 名称会替换 `@@`，这样

```
%<*package>
%<@@=foo>
\cs_new_protected:Npn \@@_some_function:nn #1#2
  {
    % Some code here
  }
\tl_new:N \l_@@_internal_tl
%</package>
```

这段代码提取出来的形式如下：

```
\cs_new_protected:Npn \__foo_some_function:nn #1#2
  {
    % Some code here
  }
\tl_new:N \l__foo_internal_tl
```

这里的 `__` 表示这些函数和变量是 `foo` 模块内部的。

使用 `@@@@` 可以在输出中得到 `@@`（使用 `@@@@@` 可得到 `@@@`）。对于较长的代码片段，可以通过给出一个空的模块名称来完全禁止替换，即使用语法 `%<@@=>`。

这个替换算法的具体步骤如下：

1. 首先，将 `@@@@` 视为一个特殊情况（通过使用一个临时伪装）。

2. 然后将所有的 `__@@` 更改为 `__`⟨*module*⟩。

3. 然后将所有剩余的 `_@@` 更改为 `__`⟨*module*⟩。

4. 然后将所有剩余的 `@@` 更改为 `__`⟨*module*⟩。

5. 最后，通过将每个“伪装的 `@@@@`”更改为 `@@` 来进行整理。

因此，替换意味着 `@@` 被 ⟨*module*⟩ 名称替换，并且在 `@@` 前的 0、1 或 2 个下划线被替换为精确的 2 个下划线（同时保留任何更多的下划线）。

7 其他语言

由于 TEX 是一个开放的系统，一些 TEX 包含有非 TEX 文件。一些作者使用 DocStrip 生成 PostScript 头文件、shell 脚本或其他语言中的程序。对于他们来说，DocStrip 的注释去除可能会引起一些问题。本节描述了如何有效地使用 DocStrip 生成非 TEX 文件。

7.1 添加到每个文件中的内容

在生成“其他”语言的文件时的第一个问题是 DocStrip 会向每个生成的文件的开头和结尾添加一些内容，这些内容可能与该语言的语法不匹配。所以我们会仔细研究到底具体添加了什么内容。

放在文件开头的整个文本都保存在由 `\declarepreamble` 定义的宏中。每一行输入到 `\declarepreamble` 的内容都会以 `\MetaPrefix` 的当前值作为前缀。标准的 DocStrip 头部会被插入到你的文本之前，并且宏 `\inFileName`、`\outFileName` 和 `\ReferenceLines` 被用作稍后填充信息的占位符（具体用于每个输出文件）。不要尝试重新定义这些宏。例如：

```
\declarepreamble\foo

----------------------------
Package FOO for use with TeX
\endpreamble
```

宏 `\foo` 就被定义为：

```
%%^^J
%% This is file `\outFileName ',^^J
%% generated with the docstrip utility.^^J
\ReferenceLines^^J
%% ----------------------------^^J
%% Package FOO for use with TeX.
```

你可以自由地对其进行操作甚至从头开始定义。要将前文嵌入到 Adobe 结构化注释中，只需使用 \edef：

```
\edef\foo{\perCent!PS-Adobe-3.0^^J%
          \DoubleperCent\space Title: \outFileName^^J%
          \foo^^J%
          \DoubleperCent\space EndComments}
```

然后使用 \usepreamble\foo 来选择你的新前文。关于后文的内容也适用相同的方法。

你也可以阻止 DocStrip 向文件中添加任何内容，并直接在代码中添加任何特定于语言的调用：

```
\generate{\usepreamble\empty
          \usepostamble\empty
          \file{foo.ps}{\from{mypackage.dtx}{ps}}}
```

或者使用 \nopreamble 和 \nopostamble 。

7.2 Meta 注释（元注释）

你可以通过重新定义 \MetaPrefix 来更改用于将元注释放入输出文件中的前缀。它的默认值是 \DoubleperCent 。前文使用了 \MetaPrefix 在 \declarepreamble 时的当前值，而源文件中的元注释使用了在 \generate 时的当前值。请注意，这意味着你不能同时使用不同的 \MetaPrefix 生成两个文件。

7.3 Verbatim（抄录）模式

如果你的编程语言使用了某种结构可能会与 DocStrip 产生严重干扰（例如第一列中的百分号），你可能需要一种方式来阻止它被剥离。为此，DocStrip 提供了“抄录模式”。

形式为 %<< ⟨*END-TAG*⟩ 的“guard 表达式”标记了一个将逐字复制的部分，直到包含只有一个百分号且位于第一列，后跟 ⟨*END-TAG*⟩ 的行。你可以选择任何你想要的 ⟨*END-TAG*⟩，但请注意这里计算空格。例如：

```
%<*myblock>
some stupid()
   #computer<program>
%<<COMMENT
% These two lines are copied verbatim (including percents
%% even if \MetaPrefix is something different than %%).
```

```
%COMMENT
   using*strange@programming<language>
%</myblock>
```

输出为（当使用定义了 myblock 时）：

```
some stupid()
   #computer<program>
% These two lines are copied verbatim (including percents
%% even if \MetaPrefix is something different than %%).
   using*strange@programming<language>
```

8 生成文档

我们提供了一个简短的驱动文件，可以通过 DocStrip 程序使用条件 'driver' 进行提取。为了允许在 IniTEX 时使用 docstrip.dtx 作为程序（例如，去除自己的注释），我们需要添加一些原始代码。通过这种额外的检查，仍然可以使用 LaTeX 2ε 处理此文件以排版文档。

```
1 ⟨*driver⟩
```

如果 \documentclass 未定义，例如在 IniTEX 或 plain TEX 中进行格式化时，我们会绕过驱动文件。

我们使用一些技巧来避免在 \ifx 结构未完成时发出 \end{document} 。如果下面的条件为真，则会实时构建一个 \fi，完成了 \ifx，真正的 \fi 将永远不会被看到，因为它位于 \end{document} 之后。另一方面，如果条件为假，TEX 将跳过 \csname fi\endcsname，不知道它可能代表 \fi，驱动文件将被跳过，然后才完成条件。

额外的保护 gobble 防止 DocStrip 将这些技巧提取到真实的驱动文件中。

```
2 ⟨*gobble⟩
3 \ifx\jobname\relax\let\documentclass\undefined\fi
4 \ifx\documentclass\undefined
5 \else \csname fi\endcsname
6 ⟨/gobble⟩
```

否则，我们会处理以下行，从而生成文档。

```
7 \documentclass{ltxdoc}
8 \usepackage[fontset=source]{ctex}
9 \usepackage{xcolor}
10 \EnableCrossrefs
11 % \DisableCrossrefs
```

```
% use \DisableCrossrefs if the
% index is ready
\RecordChanges
% \OnlyDescription
\typeout{Expect some Under- and overfull boxes}
\begin{document}
  \DocInput{docstrip-zh-cn.dtx}
\end{document}
⟨*gobble⟩
\fi
⟨/gobble⟩
⟨/driver⟩
```

9 代码实现

9.1 Initex 初始化

允许此程序在 `initex` 下运行。在 plain TEX 中，`Z` 的技巧避免了担心 `\outer` 方面的问题。

```
⟨*initex⟩
\catcode`\Z=\catcode`\%
\ifnum13=\catcode`\~{\egroup\else
  \catcode`\Z=9
Z
Z  \catcode`\{=1  \catcode`\}=2
Z  \catcode`\#=6  \catcode`\^=7
Z  \catcode`\@=11 \catcode`\^^L=13
Z  \let\bgroup={  \let\egroup=}
Z
Z  \dimendef\z@=10 \z@=0pt \chardef\@ne=1 \countdef\m@ne=22 \m@ne=-1
Z  \countdef\count@=255
Z
Z  \def\wlog{\immediate\write\m@ne} \def\space{ }
Z
Z  \count10=22 % allocates \count registers 23, 24, ...
Z  \count15=9 % allocates \toks registers 10, 11, ...
Z  \count16=-1 % allocates input streams 0, 1, ...
Z  \count17=-1 % allocates output streams 0, 1, ...
Z
Z  \def\alloc@#1#2#3{\advance\count1#1\@ne#2#3\count1#1\relax}
Z
```

```
  \def\newcount{\alloc@0\countdef} \def\newtoks{\alloc@5\toksdef}
  \def\newread{\alloc@6\chardef}   \def\newwrite{\alloc@7\chardef}

\def\newif#1{%
  \count@\escapechar \escapechar\m@ne
    \let#1\iffalse
    \@if#1\iftrue
    \@if#1\iffalse
  \escapechar\count@}
\def\@if#1#2{%
  \expandafter\def\csname\expandafter\@gobbletwo\string#1%
                     \expandafter\@gobbletwo\string#2\endcsname
                        {\let#1#2}}

  \def\@gobbletwo#1#2{}
  \def\@gobblethree#1#2#3{}

  \def\loop#1\repeat{\def\body{#1}\iterate}
  \def\iterate{\body \let\next\iterate \else\let\next\relax\fi \next}
  \let\repeat\fi

  \def\empty{}

  \def\tracingall{\tracingcommands2 \tracingstats2
    \tracingpages1 \tracingoutput1 \tracinglostchars1
    \tracingmacros2 \tracingparagraphs1 \tracingrestores1
    \showboxbreadth 10000 \showboxdepth 10000 \errorstopmode
    \errorcontextlines 10000 \tracingonline1 }

\bgroup}\fi\catcode`\Z=11
\let\bgroup={  \let\egroup=}
⟨/initex⟩
```

9.2 声明和初始化

为了能够在控制序列中包含 @ 符号，其分类码被更改为 ⟨*letter*⟩。此处的 'program' 保护语句允许在提取驱动文件时排除大部分代码。

```
⟨*program⟩
\catcode`\@=11
```

当我们想要用一条语句向终端写入多行内容时，我们需要一个字符告诉 TEX 换行。我们使用 ^^J 来实现这个目的。

```
\newlinechar=`\^^J
```

重置 8 位字符的分类码，以便在 plain TEX 或 LATEX 中处理 .ins 文件。

```
\count@=128\relax
\loop
  \catcode\count@ 12\relax
\ifnum\count@ <255\relax
    \advance\count@\@ne
\repeat
```

9.2.1 开关

\ifGenerate 程序将检查是否存在与将要创建的文件同名的文件。开关 \ifGenerate 用于指示是否需要生成剥离后的文件。

```
\newif\ifGenerate
```

\ifContinue 开关 \ifContinue 在程序的各个地方用于指示 \loop 是否需要结束。

```
\newif\ifContinue
```

\ifForlist 程序包含一个基于 plain TEX 的 \loop 宏实现的 for 循环。该实现需要一个开关来终止循环。

```
\newif\ifForlist
```

\ifDefault 开关 \ifDefault 用于指示是否使用默认批处理文件。

```
\newif\ifDefault
```

\ifMoreFiles 开关 \ifMoreFiles 用于决定是否处理更多文件。它仅在交互模式下使用；最初评估为 ⟨*true*⟩。

```
\newif\ifMoreFiles \MoreFilestrue
```

\ifaskforoverwrite 开关 \askforoverwrite 用于决定当文件要被覆盖时是否询问用户。

```
\newif\ifaskforoverwrite \askforoverwritetrue
```

9.2.2 计数寄存器

\blockLevel 可选包含的代码块可以嵌套。计数器 \blockLevel 将用于跟踪嵌套的级别。它的初始值为零。

```
\newcount\blockLevel \blockLevel\z@
```

\emptyLines 计数寄存器 \emptyLines 用于计算连续空输入行的数量。只有第一个会被复制到输出文件中。

```
\newcount\emptyLines \emptyLines \z@
```

\processedLines \commentsRemoved \commentsPassed \codeLinesPassed 为了能够向用户提供有关剥离过程的一些统计信息，如果在 DocStrip 该程序时包含了统计信息，则分配了四个计数器。处理的行数存储在计数器 \processedLines 中。包含注释但不写入输出文件的行数存储在计数器 \commentsRemoved 中；复制到输出文件的注释数量存储在计数器 \commentsPassed 中。复制到输出文件的包含宏代码的行数存储在计数器 \codeLinesPassed 中。

```
⟨*stats⟩
\newcount\processedLines   \processedLines  \z@
\newcount\commentsRemoved  \commentsRemoved \z@
\newcount\commentsPassed   \commentsPassed  \z@
\newcount\codeLinesPassed  \codeLinesPassed \z@
```

\TotalprocessedLines \TotalcommentsRemoved \TotalcommentsPassed \TotalcodeLinesPassed 当处理多个文件且包含统计信息时，我们还向用户提供有关已处理的总行数的信息。为此，在此处分配了另外四个计数寄存器。

```
\newcount\TotalprocessedLines   \TotalprocessedLines  \z@
\newcount\TotalcommentsRemoved  \TotalcommentsRemoved \z@
\newcount\TotalcommentsPassed   \TotalcommentsPassed  \z@
\newcount\TotalcodeLinesPassed  \TotalcodeLinesPassed \z@
⟨/stats⟩
```

\NumberOfFiles 当处理多个文件时，文件的数量存储在计数器 \NumberOfFiles 中。

```
\newcount\NumberOfFiles \NumberOfFiles\z@
```

9.2.3 I/O 流

\inFile 用于读取文档化的 TEX 代码文件，分配了一个输入流 \inFile 。

```
\newread\inFile
```

\ttyin \ttyout 与用户的通信通过（不存在的）16 号流进行。

```
\chardef\ttyin16
\chardef\ttyout16
```

\inputcheck 此流仅用于检查文件是否存在。

```
\newread\inputcheck
```

\ifToplevel 如果当前批处理文件是最外层的文件，则执行参数。否则将其抑制。

```
\newif\iftopbatchfile \topbatchfiletrue
\def\ifToplevel{\relax\iftopbatchfile
   \expandafter\iden \else \expandafter\@gobble\fi}
```

\batchinput 当因批处理文件中的 \input 语句而读取文件 docstrip.tex 时，我们必须防止一个无限循环（由 TEX 的堆栈限制）。因此，我们保存原始的原语 \input 并定义一个新的带有用 ␣（即空格）界定的参数的宏，它只是吞掉参数。由于换行符被 TEX 转换为一个空格。这意味着在批处理文件中 \input 不可用作命令。

\@@input 因此，我们将原始命令保存为 \@@input 以供内部使用。如果 DocStrip 在 LATEX 下运行，则此命令已经定义，因此我们进行快速测试。

```
\ifx\undefined\@@input \let\@@input\input\fi
```

为了允许批处理文件的嵌套，提供了 \batchinput 命令，它接受一个参数，即要切换到的批处理文件的名称。

```
\def\batchinput#1{%
```

我们开始一个新组，并在局部重新定义 \batchFile 以保存新的批处理文件名。我们切换了 \iftopbatchfile 开关，因为这绝对不是顶层批处理文件。

```
  \begingroup
    \def\batchfile{#1}%
    \topbatchfilefalse
    \Defaultfalse
    \usepreamble\org@preamble
    \usepostamble\org@postamble
    \let\destdir\WriteToDir
```

然后我们可以简单地调用 \processbatchFile，它将打开新的批处理文件并读取其中的内容直到耗尽。请注意，如果批处理文件不存在或拼写错误，此例程将产生警告并返回。

```
    \processbatchFile
```

在这个结束的 \endgroup 中，\batchfile 的值以及预导言、目录等的局部定义将被恢复，以便进一步处理在调用的批处理文件中继续进行。

```
  \endgroup
}
```

\skip@input 接下来是对 \input 的承诺重新定义：

```
\def\skip@input#1 {}
\let\input\skip@input
```

9.2.4 空宏和展开为字符串的宏

\guardStack 因为将有条件地包含在输出中的代码块可以被嵌套，需要维护一个堆栈来跟踪这些块。主要原因在于我们想要能够检查这些块是否正确地嵌套。堆栈本身存储在 \guardStack 中。

```
\def\guardStack{}
```

\blockHead 宏 \blockHead 用于存储和检索开始一个块的布尔表达式。

```
\def\blockHead{}
```

\yes 当用户被问及一个他必须用 ⟨*yes*⟩ 或 ⟨*no*⟩ 回答的问题时，他的回应需要被评
\y 估。因此定义了宏 \yes 和 \y 。

```
\def\yes{yes}
\def\y{y}
```

\n 我们也定义了 \n 用于 DocStrip 命令文件中的使用。

```
\def\n{n}
```

\Defaultbatchile 当 DocStrip 程序需要处理一个批处理文件时，它可以寻找一个具有默认名称的批处理文件。这个名称存储在 \DefaultbatchFile 中。

```
\def\DefaultbatchFile{docstrip.cmd}
```

\perCent 为了能在终端显示百分号，一个带有类别码 12 的 % 被存储在 \perCent 和
\DoubleperCent \DoubleperCent 中。宏 \MetaPrefix 被放置在每个元注释行的开头。它是以
\MetaPrefix 间接方式定义的，因为某些应用需要重新定义它。

```
{\catcode`\%=12
 \gdef\perCent{%}
 \gdef\DoubleperCent{%%}
}
\let\MetaPrefix\DoubleperCent
```

为了允许输入中的换页符，我们定义了一个单字符控制序列 ^^L 。

```
\def^^L{ }
```

使用 \Name 的唯一结果是减慢执行速度，因为它的典型用法（例如，\Name\def{foo bar}...）与其展开具有完全相同数量的标记。然而，我认为这样更容易阅读。\Name 作为一个黑盒的含义是：“从第二个参数构造一个名称，然后将其作为参数传递给你的第一个参数”。

\@stripstring 用于获取构建宏名称的标记，但不包括导言反斜杠。

```
\def\Name#1#2{\expandafter#1\csname#2\endcsname}
\def\@stripstring{\expandafter\@gobble\string}
```

9.2.5 杂项变量

\sourceFileName 宏 \sourceFileName 用于存储当前输入文件的名称。

\batchfile 宏 \batchfile 用于存储批处理文件的名称。

\inLine 宏 \inLine 用于在进一步处理之前存储从输入文件读取的行。

\answer 当需要与用户交互时，宏 \answer 用于存储用户的响应。

\tmp 有时需要临时将某些内容存储在控制序列中。为此，使用控制序列 \tmp 。

9.3 支持宏

9.3.1 堆栈机制

可以有“嵌套保护”。这意味着在一个可选包含的代码块内部，只有在指定了额外选项时才包含一个子组。为了跟踪保护的嵌套，当前“打开”的保护可以被推到堆栈 \guardStack 中，稍后再从堆栈中弹出。实现这个堆栈机制的宏 loosly 基于在 LaTeX3 项目中开发的代码。

为了能够实现堆栈机制，我们需要一些支持宏。

\eltStart 宏 \eltStart 和 \eltEnd 用于界定堆栈元素。它们都是空的。

\eltEnd

```
\def\eltStart{}
\def\eltEnd{}
```

\qStop 宏 \qStop 是所谓的 ‘quark’ ，一个展开为自身的宏[5]。

```
\def\qStop{\qStop}
```

\pop 宏 \pop⟨*stack*⟩⟨*cs*⟩ 从堆栈中“弹出”顶部元素。它将顶部元素的值赋给 ⟨*cs*⟩ 并从 ⟨*stack*⟩ 中删除它。当 ⟨*stack*⟩ 为空时，发出警告并将 ⟨*cs*⟩ 赋值为空。

```
\def\pop#1#2{%
  \ifx#1\empty
    \Msg{Warning: Found end guard without matching begin}%
    \let#2\empty
  \else
```

为了能够“剥离”第一个保护符，我们使用了额外的宏 \popX，它在其参数中接收扩展和未扩展的堆栈。扩展的堆栈用 quark \qStop 分隔。

```
    \def\tmp{\expandafter\popX #1\qStop #1#2}%
    \expandafter\tmp\fi}
```

[5] ‘quark’ 的概念是为了 LaTeX3 项目开发的。

\popX 当堆栈被展开时，元素被 \eltStart 和 \eltEnd 包围。堆栈的第一个元素被赋值给 #4 。

```
\def\popX\eltStart #1\eltEnd #2\qStop #3#4{\def#3{#2}\def#4{#1}}
```

\push 可以使用宏 \push⟨*stack*⟩⟨*guard*⟩ 将保护符推送到堆栈上。同样，我们需要一个辅助宏（\pushX），它的参数包括扩展和未扩展的堆栈。

```
\def\push#1#2{\expandafter\pushX #1\qStop #1{\eltStart #2\eltEnd}}
```

\pushX 宏 \pushX 接收堆栈的完整展开作为其第一个参数，并将保护符放在 #3 的“顶部”。

```
\def\pushX #1\qStop #2#3{\def #2{#3#1}}
```

9.3.2 编程结构

\forlist 当程序在交互模式下使用时，用户可以提供要处理的文件列表。为了处理此列表，需要使用 for 循环。此编程结构的实现基于在 plain TEX 中定义的 \loop{⟨*body*⟩}\repeat 宏的使用。该循环的语法如下：

\for⟨*control sequence*⟩ := ⟨*list*⟩ \do
⟨*body*⟩
\od

⟨*list*⟩ 应该是逗号分隔的列表。

首先要执行的操作是将开关 \ifForlist 设置为 ⟨*true*⟩，并将循环条件存储在宏 \ListCondition 中。这是通过使用 \edef 完成的，以允许包含 ⟨*list*⟩ 的控制序列。

```
\def\forlist#1:=#2\do#3\od{%
    \edef\ListCondition{#2}%
    \Forlisttrue
```

然后开始循环。我们将 \ListCondition 中的第一个元素存储在作为 \forlist 第一个参数提供的宏中，再将此元素从 \ListCondition 中移除。

```
    \loop
      \edef#1{\expandafter\FirstElt\ListCondition,\empty.}%
      \edef\ListCondition{\expandafter\OtherElts\ListCondition,\empty.}%
```

当从 ⟨*list*⟩ 中获取的第一个元素为空时，表示处理结束，因此我们将 \ifForlist 切换为 ⟨*false*⟩。当不为空时，我们执行第三个参数，应包含要执行的 TEX 命令。

```
      \ifx#1\empty \Forlistfalse \else#3\fi
```

最后，我们测试 \ifForlist 开关，决定是否继续循环。

```
      \ifForlist
    \repeat}
```

\FirstElt 宏 \FirstElt 用于从逗号分隔的列表中获取第一个元素。

```
\def\FirstElt#1,#2.{#1}
```

\OtherElts 宏 \OtherElts 用于获取逗号分隔列表中的除第一个元素之外的所有元素。

```
\def\OtherElts#1,#2.{#2}
```

\whileswitch 当程序在交互模式下使用时，用户可能希望使用不同的选项或扩展名处理多个文件。可以通过多次运行程序来实现这个目标，但更加用户友好的方式是在处理完最后一个请求后询问用户是否想处理更多文件。为了实现这一目标，我们需要实现一个 while 循环。同样，使用 plain TEX 的 \loop{⟨*body*⟩}\repeat 来实现这种编程结构。

此循环的语法如下：

\whileswitch⟨*switch*⟩ \fi ⟨*list*⟩ {⟨*body*⟩}

此宏的第一个参数必须是一个开关，使用 \newif 定义；第二个参数包含在开关评估为 ⟨*true*⟩ 时要执行的语句。

```
\def\whileswitch#1\fi#2{#1\loop#2#1\repeat\fi}
```

9.3.3 输出流分配器

对于每一个可用的十六个输出流，我们有一个名为 \s@0 到 \s@15 的宏，表示流是否分配给文件 (1) 或未分配 (0)。最初，所有流都未分配。

我们还声明了 16 个计数器，这些计数器将被条件代码包含算法所需。

```
\ifx\@tempcnta\undefined \newcount\@tempcnta \fi
\@tempcnta=0
\loop
\Name\chardef{s@\number\@tempcnta}=0
\csname newcount\expandafter\endcsname%
  \csname off@\number\@tempcnta\endcsname
\advance\@tempcnta1
\ifnum\@tempcnta<16\repeat
```

我们将使用《The TEXbook》风格的列表来搜索流。

```
\let\s@do\relax
\edef\@outputstreams{%
```

```
  \s@do\Name\noexpand{s@0}\s@do\Name\noexpand{s@1}%
  \s@do\Name\noexpand{s@2}\s@do\Name\noexpand{s@3}%
  \s@do\Name\noexpand{s@4}\s@do\Name\noexpand{s@5}%
  \s@do\Name\noexpand{s@6}\s@do\Name\noexpand{s@7}%
  \s@do\Name\noexpand{s@8}\s@do\Name\noexpand{s@9}%
  \s@do\Name\noexpand{s@10}\s@do\Name\noexpand{s@11}%
  \s@do\Name\noexpand{s@12}\s@do\Name\noexpand{s@13}%
  \s@do\Name\noexpand{s@14}\s@do\Name\noexpand{s@15}%
  \noexpand\@nostreamerror
  }
```

\@nostreamerror
\@streamfound

当 \@outputstreams 被执行时，\s@do 被定义为在某个测试条件下执行某些操作。如果条件总是失败，则列表末尾的宏 \@nostreamerror 会引发错误。当条件成功时，将调用 \@streamfound，它会吞掉列表的其余部分，包括结束的 \@nostreamerror 。它还会吞掉结束条件的 \fi，因此会重新插入 \fi 。

```
\def\@nostreamerror{\errmessage{No more output streams!}}
\def\@streamfound#1\@nostreamerror{\fi}
```

\@stripstr 是一个辅助宏，吞掉字符 \s@（反斜杠、s、@）。由于 \s@ 必须具有全是类别码 12 （其他字符）的特殊定义方式，因此这个宏用于从流名称中提取流号。

```
\bgroup\edef\x{\egroup
 \def\noexpand\@stripstr\string\s@{}}
\x
```

\quote@name

从 ltfiles.dtx 复制的一个宏，以便允许文件名中包含空格。

```
\def\quote@name#1{"\quote@@name#1\@gobble""}
\def\quote@@name#1"{#1\quote@@name}
```

\StreamOpen
\StreamPut
\StreamClose

这是流打开操作符。它的参数应该是一个与要打开的外部文件同名的宏。例如，要写入文件 foo.tex，使用 \StreamOpen\foo，然后使用 \StreamPut\foo 和 \StreamClose\foo 。

```
\chardef\stream@closed=16
\def\StreamOpen#1{%
  \chardef#1=\stream@closed
  \def\s@do##1{\ifnum##1=0
    \chardef#1=\expandafter\@stripstr\string##1 %
    \global\chardef##1=1 %
    \edef\q@curr@file{%
      \expandafter\expandafter\expandafter\quote@name
```

```
      \expandafter\expandafter\expandafter{\csname pth@\@stripstring#1\endcsname}}
    \immediate\openout#1=\q@curr@file\relax
    \@streamfound
    \fi}
  \@outputstreams
  }
\def\StreamClose#1{%
  \immediate\closeout#1%
  \def\s@do##1{\ifnum#1=\expandafter\@stripstr\string##1 %
    \global\chardef##1=0 %
    \@streamfound
    \fi}
  \@outputstreams
  \chardef#1=\stream@closed
  }
\def\StreamPut{\immediate\write}
```

9.3.4 输入和输出

\maybeMsg
\showprogress
\keepsilent

当使用该程序时，可以选择在终端显示其进度。在这种情况下，它会为每个输入行向终端（和记录文件）写入一个特殊字符。当在 docstrip.tex 中包含统计信息时，默认情况下启用此选项。当排除统计信息时，此选项关闭。命令 \showprogress 和 \keepsilent 可以用于选择其他方式。

```
\def\showprogress{\let\maybeMsg\message}
\def\keepsilent{\let\maybeMsg\@gobble}
⟨*stats⟩
\showprogress
⟨/stats⟩
⟨-stats⟩\keepsilent
```

\Msg

为了在终端显示消息，宏 \Msg 被定义为立即写入到 \ttyout 。

```
\def\Msg{\immediate\write\ttyout}
```

\Ask

宏 \Ask{⟨*cs*⟩}{⟨*string*⟩} 是对 LaTeX 宏 \typein 稍作修改的副本。它用于向用户提问。⟨*string*⟩ 将显示在用户的终端上，而响应将存储在 ⟨*cs*⟩ 中。由回车产生的尾随空格会被宏 \strip 剥离。如果用户只输入了回车，结果将是一个空的宏。

```
\def\iden#1{#1}
\def\strip#1#2 \@gobble{\def #1{#2}}
\def\@defpar{\par}
\def\Ask#1#2{%
```

```
    \message{#2}\read\ttyin to #1\ifx#1\@defpar\def#1{}\else
    \iden{\expandafter\strip
       \expandafter#1#1\@gobble\@gobble} \@gobble\fi}
```

\OriginalAsk

```
\let\OriginalAsk=\Ask
```

\askonceonly

```
\def\askonceonly{%
  \def\Ask##1##2{%
    \OriginalAsk{##1}{##2}%
    \global\let\Ask\OriginalAsk
    \Ask\noprompt{%
      By default you will be asked this question for every file.^^J%
      If you enter `y' now,^^J%
      I will assume `y' for all future questions^^J%
      without prompting.}%
    \ifx\y\noprompt\let\noprompt\yes\fi
    \ifx\yes\noprompt\gdef\Ask####1####2{\def####1{y}}\fi}}
```

9.3.5 杂项

\@gobble 接受一个参数并将其丢弃的宏。

```
\def\@gobble#1{}
```

\Endinput 当 doc 文件包含单独一行的 \endinput 时，通常意味着该文件中以下的内容都应被忽略。因此，我们需要一个宏，其中包含 \endinput 作为其替换文本，以便与 \inLine（当前输入文件的当前行）进行比较。当然，反斜杠必须具有正确的 \catcode 。一种做法是将 \\ 提供给 \string 操作，然后移除一个 \ 字符。

```
\edef\Endinput{\expandafter\@gobble\string\\endinput}
```

\makeOther 在读取包含 TEX 代码的文件过程中，所有特殊字符的类别码必须更改为 ⟨*other*⟩。宏 \makeOther 就是为此目的而设的。

```
\def\makeOther#1{\catcode`#1=12\relax}
```

\end 目前我们希望 DocStrip 程序与 plain TEX 和 LATEX 都兼容。LATEX 隐藏了 plain TEX 的 \end 命令，并将其称为 \@@end 。我们在这里取消隐藏。

```
\ifx\undefined\@@end\else\let\end\@@end\fi
```

\@addto 一个扩展宏定义的宏。使用 \csname 的技巧是为了解决在 plain TEX 和 LATEX 版本 2.09 中 \newtoks 是外部命令的问题。

```
\ifx\@temptokena\undefined \csname newtoks\endcsname\@temptokena\fi

\def\@addto#1#2{%
  \@temptokena\expandafter{#1}%
  \edef#1{\the\@temptokena#2}}
```

\@ifpresent 这个宏检查它的第一个参数是否存在于作为第二个参数传递的列表中。根据结果，它执行第三个或第四个参数。

```
\def\@ifpresent#1#2#3#4{%
  \def\tmp##1#1#1##2\qStop{\ifx!##2!}%
  \expandafter\tmp#2#1\qStop #4\else #3\fi
  }
```

\tospaces 这个宏将用 \secapsot 分隔的参数转换为适当数量的空格。我们在屏幕上智能显示消息时需要这个。

当需要连续多个空格时，使用 \@spaces 。

```
\def\tospaces#1{%
  \ifx#1\secapsot\secapsot\fi\space\tospaces}
\def\secapsot\fi\space\tospaces{\fi}
\def\@spaces{\space\space\space\space\space}
```

\uptospace 这个宏从其用 \qStop 分隔的参数中提取直到第一个空格的部分。

```
\def\uptospace#1 #2\qStop{#1}
```

\afterfi 这个宏可用于条件语句，在条件完成后（即读取匹配的 \fi 后）执行某些操作（它的第一个参数）。第二个参数用于吞掉剩余的 \if ... \fi 结构（可能是一些 \else）。请注意，这在嵌套的 \if 中不起作用！

```
\def\afterfi#1#2\fi{\fi#1}
```

\@ifnextchar 这是 LATEX 中的一个宏，不是在 plain TEX 中定义的。我的诡计般的定义与标准定义不同，但功能是相同的。

```
\def\@ifnextchar#1#2#3{\bgroup
  \def\reserved@a{\ifx\reserved@c #1 \aftergroup\@firstoftwo
    \else \aftergroup\@secondoftwo\fi\egroup
    {#2}{#3}}%
  \futurelet\reserved@c\@ifnch
  }
\def\@ifnch{\ifx \reserved@c \@sptoken \expandafter\@xifnch
```

```
      \else \expandafter\reserved@a
      \fi}
\def\@firstoftwo#1#2{#1}
\def\@secondoftwo#1#2{#2}
\iden{\let\@sptoken= } %
\iden{\def\@xifnch} {\futurelet\reserved@c\@ifnch}
```

\kernel@ifnextchar

LaTeX 的 2003/12/01 版本引入了此宏以避免与 amsmath 的问题，但这也意味着当人们在包含 \ProvidesFile 的安装文件上使用 LaTeX 时，我们必须在这里执行相同的技巧。

```
\let\kernel@ifnextchar\@ifnextchar
```

9.4 布尔表达式的运算

为了清晰起见，我们在此重复布尔表达式的语法，稍作调整但是等效的方式：

$$
\begin{aligned}
\langle Expression\rangle &::= \langle Secondary\rangle \mid \langle Secondary\rangle\ \{\texttt{|},\texttt{,}\}\ \langle Expression\rangle\\
\langle Secondary\rangle &::= \langle Primary\rangle \mid \langle Primary\rangle\ \texttt{\&}\ \langle Secondary\rangle\\
\langle Primary\rangle &::= \langle Terminal\rangle \mid \texttt{!}\langle Primary\rangle \mid \texttt{(}\langle Expression\rangle\texttt{)}
\end{aligned}
$$

| 代表析取，& 代表合取，! 代表否定。⟨*Terminal*⟩ 是任意的字母序列，当它出现在必须被包含的选项列表中时，会评估为 ⟨*true*⟩。

由于我们可以从一个输入生成多个输出文件，同样的 guard 表达式可能会使用不同的选项计算多次。因此，我们首先将表达式“编译”为一个参数宏 \Expr 的形式，展开为嵌套的 \if，当给定当前的选项列表时，会产生 1 或 0 作为结果。这个想法是对所有输出文件说 \if1\Expr{⟨*current set of options*⟩}...\fi 。

下面是递归定义右侧语法的翻译表。$\tau(X)$ 表示 X 的翻译。

$$
\begin{aligned}
\tau(\langle Terminal\rangle) &= \texttt{\t@<Terminal>,\#1,<Terminal>,\qStop}\\
\tau(\texttt{!}\langle Primary\rangle) &= \texttt{\if1}\,\tau(\langle Primary\rangle)\,\texttt{0\else1\fi}\\
\tau(\texttt{(}\langle Expression\rangle\texttt{)}) &= \tau(\langle Expression\rangle)\\
\tau(\langle Primary\rangle\texttt{\&}\langle Secondary\rangle) &= \texttt{\if0}\,\tau(\langle Primary\rangle)\,\texttt{0\else}\,\tau(\langle Secondary\rangle)\,\texttt{\fi}\\
\tau(\langle Secondary\rangle\texttt{|}\langle Expression\rangle) &= \texttt{\if1}\,\tau(\langle Secondary\rangle)\,\texttt{1\else}\,\tau(\langle Expression\rangle)\,\texttt{\fi}
\end{aligned}
$$

\t@<Terminal> 表示宏的名称由 t@ 构造，并附加了终端的标记。例如，对于终端 foo，翻译将是

```
\t@foo,#1,foo,\qStop
```

这将最终定义为 \Expr，所以 #1 在这里将被当前选项列表替换，当调用 \Expr 时。宏 \t@foo 将被定义为

```
\def\t@foo#1,foo,#2\qStop{\ifx>#2>0\else1\fi}
```

当像上面这样调用时，如果 foo 存在于当前选项列表中，这将扩展为 1，否则为 0 。

下面的宏以“几乎只扩展”的方式工作，即表达式仅通过展开进行分析，但是我们必须在途中定义一些宏（例如，\Expr 和 \t@foo）。

每个这些宏的第一个参数都是“续延”（类似于 Scheme 语言的含义）。续延是至少有一个参数（参数）的函数，它是先前计算步骤的值。例如，宏 \Primary 构造了 ⟨*Primary*⟩ 表达式的翻译。当它决定表达式已完成时，它调用其续延（它的第一个参数），传递给它整个构造的翻译。如果续延希望查看输入的下一个内容，则可能期望更多的参数。

我们将执行递归下降解析，但是定义将以自底向上的顺序呈现。

\Terminal ⟨*Terminal*⟩ 被宏 \Terminal 识别。正确的调用方式是 \Terminal{⟨*current continuation*⟩}{} 。参数包括：续行、到目前为止的 ⟨*Terminal*⟩ 和输入的下一个字符。该宏检查 #3 是否是终止字符之一，然后采取相应的操作。由于有 7 个结束字符，可能一个 \csname 的成本比 7 个嵌套的 \if 更低，我们构造一个名称并检查其是否定义。

在执行下一步操作之前，我们必须完全展开 \ifx，因此我们使用 \afterfi 。

```
\def\Terminal#1#2#3{%
  \expandafter\ifx\csname eT@#3\endcsname\relax
```

如果条件为真，#3 属于当前 ⟨*Terminal*⟩，因此我们将其附加到迄今为止的 ⟨*Terminal*⟩，然后再次调用 \Terminal 。

```
    \afterfi{\Terminal{#1}{#2#3}}\else
```

当条件为假时，是结束当前 ⟨*Terminal*⟩ 的时候，所以我们调用宏 \TerminalX 。下一个字符被重新插入输入。

在两种情况下，续行都保持不变。

```
    \afterfi{\TerminalX{#1}{#2}#3}\fi
  }
```

\eT@ 这里我们定义标记不能出现在终端内的字符的宏。值并不重要，只要与 \relax 不同即可。

```
\Name\let{eT@>}=1
\Name\let{eT@&}=1 \Name\let{eT@!}=1
```

```
\Name\let{eT@|}=1 \Name\let{eT@,}=1
\Name\let{eT@(}=1 \Name\let{eT@)}=1
```

\TerminalX 这个宏应该结束对 ⟨*Terminal*⟩ 的扫描。参数包括续行和 ⟨*Terminal*⟩ 的收集的记号。

如果 ⟨*Terminal*⟩ 为空，宏首先发出错误消息。

```
\def\TerminalX#1#2{%
  \ifx>#2> \errmessage{表达式错误：终端为空}\fi
```

然后构造一个用于检查选项列表中是否存在 ⟨*Terminal*⟩ 的宏。

```
  \Name\def{t@#2}##1,#2,##2\qStop{\ifx>##2>0\else1\fi}%
```

然后按照公式调用当前的续行，根据公式

$$\tau(\langle Terminal\rangle) = \texttt{\textbackslash t@<Terminal>,\#1,<Terminal>,\textbackslash qStop}$$

进行翻译 ⟨*Terminal*⟩。

```
  #1{\Name\noexpand{t@#2},##1,#2,\noexpand\qStop}%
  }
```

\Primary 参数包括续行和输入的下一个字符。

根据语法，⟨*Primari*⟩es 可以有三种形式。这使我们使用比通常更多的巧妙技巧。注意在一系列 `\ifx` 之后的 `\space` 。这个系列产生一个一位数字的情况来执行。数字传递给 `\ifcase`，而 `\space` 停止 TEX 扫描一个 ⟨*number*⟩。使用 `\ifcase` 可以在不嵌套 `\if` 的情况下选择三个操作之一，并使用 `\afterfi` 。

```
\def\Primary#1#2{%
  \ifcase \ifx!#20\else\ifx(#21\else2\fi\fi\space
```

第一个情况是 `!`，即否定的 ⟨*Primary*⟩。在这种情况下，我们递归调用 `\Primary`，但创建一个新的续行：宏 `\NPrimary`，它将否定由 `\Primary` 传递的结果，并将其传递给当前续行（`#1`）。

```
  \afterfi{\Primary{\NPrimary{#1}}}\or
```

当下一个字符是 `(` 时，我们调用 `\Expression`，将其作为续行，它将只是将结果传递上去，但首先确保下一个是 `)`。

```
  \afterfi{\Expression{\PExpression{#1}}}\or
```

否则，我们开始一个 ⟨*Terminal*⟩。`#2` 不作为 ⟨*Terminal*⟩-so-far 传递，而是重新插入输入，因为我们没有检查它是否可以出现在 ⟨*Terminal*⟩ 中。

```
  \afterfi{\Terminal{#1}{}#2}\fi
  }
```

\NPrimary 参数包括续行和先前计算的 ⟨*Primary*⟩。

此宏根据规则否定先前计算的结果：

$$\tau(!\langle Primary\rangle) = \texttt{\textbackslash if1}\,\tau(\langle Primary\rangle)\,\texttt{0\textbackslash else1\textbackslash fi}$$

```
\def\NPrimary#1#2{%
  #1{\noexpand\if1#20\noexpand\else1\noexpand\fi}%
  }
```

\PExpression 参数包括：续行、⟨*Expression*⟩ 和输入的下一个字符。我们检查字符是否为)，然后将未更改的结果传递给我们的续行。

```
\def\PExpression#1#2#3{%
  \ifx)#3\else
    \errmessage{表达式错误：期望右括号}\fi
  #1{#2}}
```

\Secondary 每个 ⟨*Secondary*⟩ 表达式都以 ⟨*Primary*⟩ 开始。接下来的检查将由 \SecondaryX 进行。

```
\def\Secondary#1{%
  \Primary{\SecondaryX{#1}}}
```

\SecondaryX 参数包括：续行、⟨*Primary*⟩ 的翻译、下一个字符。我们首先检查下一个字符是否是 & 。

```
\bgroup\catcode`\&=12
\gdef\SecondaryX#1#2#3{%
  \ifx&#3%
```

如果是，我们应该解析下一个 ⟨*Secondary*⟩，然后将其与到目前为止的结果合并。注意 \SecondaryXX 将有 3 个参数。

```
  \afterfi{\Secondary{\SecondaryXX{#1}{#2}}}\else
```

否则，⟨*Secondary*⟩ 实际上就是 ⟨*Primary*⟩。我们调用续行，传递该 ⟨*Primary*⟩ 的翻译，不要忘记将 #3 重新插入输入，因为它不属于这里。

```
  \afterfi{#1{#2}#3}\fi
  }
\egroup
```

\SecondaryXX 参数包括：续行、⟨*Primary*⟩ 的翻译、⟨*Secondary*⟩ 的翻译。根据规则构造整个结构的翻译：

$$\tau(\langle Primary\rangle\&\langle Secondary\rangle) = \texttt{\textbackslash if0}\,\tau(\langle Primary\rangle)\,\texttt{0\textbackslash else}\,\tau(\langle Secondary\rangle)\,\texttt{\textbackslash fi}$$

并将其传递给我们的续行。

```
\def\SecondaryXX#1#2#3{%
  #1{\noexpand\if0#20\noexpand\else#3\noexpand\fi}}
```

\Expression 每个 ⟨*Expression*⟩ 都以 ⟨*Secondary*⟩ 开始。我们构造一个新的续行并将其传递给 \Secondary 。

```
\def\Expression#1{%
  \Secondary{\ExpressionX{#1}}}
```

\ExpressionX 参数包括：续行、⟨*Secondary*⟩ 的翻译、下一个字符。我们检查字符是否为 | 或 , 。

```
\def\ExpressionX#1#2#3{%
  \if0\ifx|#31\else\ifx,#31\fi\fi0
```

如果不是，⟨*Expression*⟩ 就是一个 ⟨*Secondary*⟩。我们将其翻译传递给续行，并重新插入 #3 。

```
  \afterfi{#1{#2}#3}\else
```

如果我们正在处理复杂的 ⟨*Expression*⟩,我们现在应该解析另一个 \Expression 。

```
  \afterfi{\Expression{\ExpressionXX{#1}{#2}}}\fi
  }
```

\ExpressionXX 参数包括：续行、⟨*Secondary*⟩ 的翻译、⟨*Expression*⟩ 的翻译。根据公式完成 ⟨*Expression*⟩ 的翻译：

$$\tau(\langle Secondary\rangle\,|\,\langle Expression\rangle) = \texttt{\textbackslash if1}\,\tau(\langle Secondary\rangle)\,\texttt{1\textbackslash else}\,\tau(\langle Expression\rangle)\,\texttt{\textbackslash fi}$$

```
\def\ExpressionXX#1#2#3{%
  #1{\noexpand\if1#21\noexpand\else#3\noexpand\fi}}
```

\StopParse 这里是整个解析过程的初始续行。它将被 \Evaluate 使用。请注意，我们假设表达式以 > 结尾。这个宏最终定义了 \Expr 。参数包括：整个 ⟨*Expression*⟩ 的翻译和输入的下一个字符。

```
\def\StopParse#1#2{%
  \ifx>#2 \else\errmessage{表达式错误：多余的 #2}\fi
  \edef\Expr##1{#1}}
```

\Evaluate 这个宏用于开始解析。我们使用上面定义的续行调用 \Expression 。在表达式结束时，我们附加一个 > 。

```
\def\Evaluate#1{%
  \Expression\StopParse#1>}
```

9.5 处理输入行

\normalLine 宏 \normalLine 将其参数（必须以 \endLine 分隔）写入所有活动输出文件，即计数器为零的文件。它使用了名为 \replaceModuleInLine 的搜索和替换宏，用于将任何出现的 @@ 替换为当前模块名称。如果包含了统计信息，它会增加计数器 \codeLinesPassed 的值。

```
\def\normalLine#1\endLine{%
⟨*stats⟩
  \advance\codeLinesPassed\@ne
⟨/stats⟩
  \maybeMsg{.}%
  \def\inLine{#1}%
  \replaceModuleInLine
  \let\do\putline@do
  \activefiles
  }
```

\putline@do 将行复制到输出文件时 \do 的一个取值。

```
\def\putline@do#1#2#3{%
  \StreamPut#1{\inLine}}
```

\removeComment 宏 \removeComment 丢弃其参数（必须以 \endLine 分隔）。当程序包含统计信息时，移除的注释会计入统计数。

```
%
\def\removeComment#1\endLine{%
⟨*stats⟩
    \advance\commentsRemoved\@ne
⟨/stats⟩
    \maybeMsg{\perCent}}
```

\putMetaComment 如果一行以两个连续的百分号开头，则被视为 *MetaComment*。这样的注释行会不经修改地传递到输出文件。

```
\bgroup\catcode`\%=12
\iden{\egroup
\def\putMetaComment%}#1\endLine{%
```

如果包含统计信息，该行会被计入统计数。

```
⟨*stats⟩
  \advance\commentsPassed\@ne
⟨/stats⟩
```

宏 \putMetaComment 有一个参数，以 \endLine 作为分界。它带有源代码行，其中 %% 被剥离。我们在其前面加上 \MetaPrefix（可能与 %% 不同），然后将该行发送到所有活动文件中。

```
\edef\inLine{\MetaPrefix#1}%
\let\do\putline@do
\activefiles
}
```

\processLine 从输入流中读取的每一行都需要经过处理，以确定是否需要将其写入输出流。这项任务通过调用宏 \processLine 来完成。为了实现这一点，它需要检查行是否以 '%' 开头。因此，该宏在一个组内进行全局定义。在该组内，'%' 的类别码被更改为 12（其他）。因为需要一个注释字符，'*' 的类别码被更改为 14（注释字符）。

如果包含统计信息，该宏会将计数器 \processedLines 增加 1。我们不能使用 %<*stats> 将这行包含进去，因为 '%' 的类别码被修改了，并且文件必须能够在未剥离的情况下加载。因此，整个定义被嵌入到了保护区域内。

从输入流中传递的下一个记号是 #1 。如果它是 '%'，则需要由 \processLineX 进行进一步处理；否则，这是一行正常的（未被注释的）行。

无论哪种情况，读取的字符都会重新插入到输入流中，因为可能需要将其写出。

```
⟨*!stats⟩
\begingroup
\catcode`\%=12 \catcode`\*=14
\gdef\processLine#1{*
  \ifx%#1
    \expandafter\processLineX
  \else
    \expandafter\normalLine
  \fi
  #1}
\endgroup
⟨/!stats⟩
⟨*stats⟩
\begingroup
\catcode`\%=12 \catcode`\*=14
\gdef\processLine#1{*
  \advance\processedLines\@ne
  \ifx%#1
    \expandafter\processLineX
  \else
```

```
    \expandafter\normalLine
  \fi
  #1}
\endgroup
⟨/stats⟩
```

\processLineX 这个宏也在一个组内定义，因为它还必须检查输入流中的下一个记号是否为‘%’字符。

如果当前行的第二个记号恰好是‘%’，则找到了一个⟨*MetaComment*⟩。这必须完整地复制到输出中。另一个可能的第二个字符是‘<’，它引入了一个 guard 表达式。通过调用 \checkOption 来启动这种表达式的处理。

当记号既不是‘%’也不是‘<’时，该行包含一个需要移除的普通注释。

我们以这样的方式表达条件，使得所有操作都出现在 \if 的第一层嵌套中。在这样的条件下，只有一个 expandafter 将我们推到整个结构的外面。这里需要注意的是 \relax 。它停止了对数字常量的搜索。如果没有它，在知道值之前，TEX 会在 \ifcase 的第一个情况下展开。

```
\begingroup
\catcode`\%=12 \catcode`\*=14
\gdef\processLineX%#1{*
  \ifcase\ifx%#10\else
         \ifx<#11\else 2\fi\fi\relax
    \expandafter\putMetaComment\or
    \expandafter\checkOption\or
    \expandafter\removeComment\fi
  #1}
\endgroup
```

9.6 选项处理

\checkOption 当处理一行的宏发现该行以‘%<’开头时，就会遇到一个 guard 行。guard 的第一个字符可以是星号 (*)、斜杠 (/)、加号 (+)、减号 (-)、尖括号 (<)（以原样模式开始）、商用符号 (@) 或其他任何选项名称中的字符。这意味着我们需要窥探下一个记号，并决定我们遇到了什么类型的 guard 。

我们重新插入 #1，因为它可能被 \doOption 使用。

```
\def\checkOption<#1{%
  \ifcase
    \ifx*#10\else \ifx/#11\else
    \ifx+#12\else \ifx-#13\else
    \ifx<#14\else \ifx @#15\else 6\fi\fi\fi\fi\fi\fi\relax
```

```
  \expandafter\starOption\or
  \expandafter\slashOption\or
  \expandafter\plusOption\or
  \expandafter\minusOption\or
  \expandafter\verbOption\or
  \expandafter\moduleOption\or
  \expandafter\doOption\fi
  #1}
```

\doOption 当 \checkOptions 没有找到 guard 修饰符时，会调用宏 \doOption 。它评估一个布尔表达式。该评估的结果存储在 \Expr 中。guard 只影响当前行，因此 \do 的定义方式取决于测试 \if1\Expr{⟨*options*⟩}，当前行要么复制到输出流中，要么被移除。然后该测试计算所有活动输出文件的结果。

```
\def\doOption#1>#2\endLine{%
  \maybeMsg{<#1 . >}%
  \Evaluate{#1}%
  \def\do##1##2##3{%
    \if1\Expr{##2}%
      \def\inLine{#2}%
      \replaceModuleInLine
      \StreamPut##1{\inLine}\fi
    }%
  \activefiles
  }
```

\plusOption 当找到 '+' 作为 guard 修饰符时，会调用 \plusOption 。这个宏与 \doOption 非常相似，唯一的区别是显示的消息现在包含 '+'。

```
\def\plusOption+#1>#2\endLine{%
  \maybeMsg{<+#1 . >}%
  \Evaluate{#1}%
  \def\do##1##2##3{%
    \if1\Expr{##2}%
      \def\inLine{#2}\replaceModuleInLine
      \StreamPut##1{\inLine}\fi
    }%
  \activefiles
  }
```

\minusOption 当找到 '-' 作为 guard 修饰符时,会调用 \minusOption 。这个宏与 \plusOption 非常相似，不同之处在于条件被否定了。

```
\def\minusOption-#1>#2\endLine{%
```

```
\maybeMsg{<-#1 . >}%
\Evaluate{#1}%
\def\do##1##2##3{%
  \if1\Expr{##2}\else
    \def\inLine{#2}\replaceModuleInLine
    \StreamPut##1{\inLine}\fi
  }%
\activefiles
}
```

\starOption 当作为 guard 修饰符找到 '*' 时，会调用 \starOption 。在这种情况下，当 guard 表达式评估为 ⟨*true*⟩ 时，将包含一段代码块到输出中。

当前行作为 #2 被吞掉，因为它只包含 guard 和可能的注释。

```
\def\starOption*#1>#2\endLine{%
```

首先，我们可能向终端写入消息，指示新选项从这里开始。

```
\maybeMsg{<*#1}%
```

然后，我们将 \blockHead 的当前内容推入块的堆栈 \guardStack 中，并递增计数器 \blockLevel，表示我们现在进入了更深一级的嵌套。

```
\expandafter\push\expandafter\guardStack\expandafter{\blockHead}%
\advance\blockLevel\@ne
```

此代码块的 guard 现在存储在 \blockHead 中。

```
\def\blockHead{#1}%
```

现在我们为所有输出文件评估 guard 表达式，更新 off-counter。然后我们创建新的活动输出文件列表。现在只有外部块中活动的文件可以保持活动状态。

```
\Evaluate{#1}%
\let\do\checkguard@do
\outputfiles
\let\do\findactive@do
\edef\activefiles{\activefiles}
}
```

\checkguard@do 这种形式的 \do 根据守卫表达式的值更新 off-count。

```
\def\checkguard@do#1#2#3{%
```

如果此代码块位于不包括在输出中的另一个代码块中，我们增加 off-counter。在这种情况下，不会评估守卫表达式，因为一个不包括在输出中的代码块内部也会被排除，不管其守卫的评估如何。

```
\ifnum#3>0
  \advance#3\@ne
```

当 off-counter 的值为 0 时，我们必须评估守卫表达式。如果结果是 ⟨*false*⟩，则增加 off-counter。

```
\else
  \if1\Expr{#2}\else
    \advance#3\@ne\fi
\fi}
```

\findactive@do 这种形式的 \do 挑选输出文件列表中 off-counter 为零的元素。

```
\def\findactive@do#1#2#3{%
  \ifnum#3=0
    \noexpand\do#1{#2}#3\fi}
```

\slashOption 宏 \slashOption 是 \starOption 的对应物。它表示有条件地包含的代码块的结束。我们将参数存储在临时控制序列 \tmp 中。

```
\def\slashOption/#1>#2\endLine{%
  \def\tmp{#1}%
```

当计数器 \blockLevel 的值小于 1 时，此“结束块”行没有相应的“开始块”。因此我们发出错误信号并忽略此块的结束。

```
  \ifnum\blockLevel<\@ne
    \errmessage{Spurious end block </\tmp> ignored}%
```

接下来，我们将 \tmp 的内容与 \blockHead 的内容进行比较。后者包含遇到的代码块的最后一个守卫。如果内容匹配，我们就从堆栈中弹出先前的守卫。

```
  \else
    \ifx\tmp\blockHead
      \pop\guardStack\blockHead
```

这是期望的行为吗??

当两个宏的内容不匹配时，说明出现了问题。我们向用户发出信号，但接受了“结束块”。

```
    \else
      \errmessage{Found </\tmp> instead of </\blockHead>}%
    \fi
```

当遇到可选包含代码块的结束时，我们可以选择在终端上发出信号，并递减计数器 \blockLevel 。

```
    \maybeMsg{>}%
    \advance\blockLevel\m@ne
```

最后要做的是递减 off-counter，并创建新的活动文件列表。现在必须搜索整个输出文件列表，因为一些不活跃的文件可能已被重新激活。

```
  \let\do\closeguard@do
  \outputfiles
```

```
  \let\do\findactive@do
  \edef\activefiles{\outputfiles}
  \fi
  }
```

\closeguard@do 此宏递减非零 off-counter。

```
\def\closeguard@do#1#2#3{%
  \ifnum#3>0
    \advance#3\m@ne
  \fi}
```

\verbOption 当一行以 %<< 开头时，调用此宏。它以抄录模式读取一堆行：这些行将无需检查以 % 开头就直接传递到输出。这种处理方式在发现仅包含百分号和在 %<< 行上给出的停止标记的行时结束。

```
\def\verbOption<#1\endLine{{%
  \edef\verbStop{\perCent#1}\maybeMsg{<<<}%
  \let\do\putline@do
  \loop
    \ifeof\inFile\errmessage{Source file ended while in verbatim
                             mode!}\fi
    \read\inFile to \inLine
  \if 1\ifx\inLine\verbStop 0\fi 1% if not inLine==verbStop
    \activefiles
    \maybeMsg{.}%
  \repeat
  \maybeMsg{>}%
  }}
```

\moduleOption 如果行以 %<@ 开始：定义的语法要求其继续到 %<@@= 。目前我们假定语法正确，并且这里的 #1 是模块名称，用于替换提取材料中的任何内部函数。

```
\def\moduleOption @@=#1>#2\endLine{%
  \maybeMsg{<@@=#1>}%
  \prepareActiveModule{#1}%
}
```

\prepareActiveModule \replaceModuleInLine 在这里，我们设置进行搜索和替换的过程。参数（#1）是要使用的替换文本，或者如果为空，则表示不应进行任何替换。搜索材料是 __@@、_@@ 或 @@，以便使得所有三者在输出中最终相同。字符串 @@@@ 通过暂时将其转换为带有不同类别代码的一对字母来隐藏这些替换，不受 DocStrip 产生的影响；这允许在输出中得到 @@ 。替换函数初始化为对于从未看到 %<@@= 的情况不执行任何操作。

```
\begingroup
  \catcode`\_ = 12 %
  \long\gdef\prepareActiveModule#1{%
    \ifx\relax#1\relax
       \let\replaceModuleInLine\empty
    \else
      \edef\replaceModuleInLine{%
        \noexpand\replaceAllIn\noexpand\inLine{@@@@}{\string aa}%
        \noexpand\replaceAllIn\noexpand\inLine{__@@}{__#1}%
        \noexpand\replaceAllIn\noexpand\inLine{_@@}{__#1}%
        \noexpand\replaceAllIn\noexpand\inLine{@@}{__#1}%
        \noexpand\replaceAllIn\noexpand\inLine{\string aa}{@@}%
      }%
    \fi
  }
\endgroup
\let\replaceModuleInLine\empty
```

\replaceAllIn
\replaceAllInAuxI
\replaceAllInAuxII
\replaceAllInAuxIII

这里的代码是一个简单的搜索和替换例程，用于将宏 #1 中的 #2 替换为 #3 。在这里设置的假设是，没有任何可展开的内容，这是合理的，因为 DocStrip 处理的是“字符串”材料。

```
\long\def\replaceAllIn#1#2#3{%
  \long\def\tempa##1##2#2{%
    ##2\qMark\replaceAllInAuxIII#3##1%
  }%
  \edef#1{\expandafter\replaceAllInAuxI#1\qMark#2\qStop}%
}
\def\replaceAllInAuxI{%
  \expandafter\replaceAllInAuxII\tempa\replaceAllInAuxI\empty
}
\long\def\replaceAllInAuxII#1\qMark#2{#1}
\long\def\replaceAllInAuxIII#1\qStop{}
```

9.7 批处理文件命令

DocStrip 保存控制多个列表结构中源文件包含的信息。这些列表是宏，展开为对宏 \do 的一系列调用，具有两个或三个参数。随后 \do 以各种方式重新定义，并且有时列表宏会执行以对每个元素执行某些操作，有时会用于 \edef 内部，以生成具有某些属性的新元素列表。对于每个输入文件 ⟨*infile*⟩，保存以下列表：

\b@⟨*infile*⟩ “开放列表”——所有输出文件的名称，其生成应始于读取 ⟨*infile*⟩，

`\o@`⟨*infile*⟩ “输出列表”——从该源文件生成的所有输出文件的名称，以及适当的选项集（守卫），

`\e@`⟨*infile*⟩ “关闭列表”——当读取此源文件时应关闭的所有输出文件的名称。

对于每个输出文件名 ⟨*outfile*⟩，DocStrip 保存以下信息：

`\pth@`⟨*outfile*⟩ 完整路径名（包括文件名），

`\ref@`⟨*outfile*⟩ 文件的参考行，

`\in@`⟨*outfile*⟩ 与空格分隔的所有源文件的名称（由 `\InFileName` 需要），

`\pre@`⟨*outfile*⟩ 导言模板（由 `\declarepreamble` 定义），

`\post@`⟨*outfile*⟩ 后记模板。

`\generate` 此宏在组内执行其参数。`\inputfiles` 是要读取的文件列表，`\filestogenerate` 是输出文件的名称列表（下面的消息所需）。包含在 `#1` 中的 `\file` 定义了 `\inputfiles`，使得当执行参数时，唯一要做的就是调用此宏。直到没有需要推迟的输出文件为止，将反复调用 `\inputfiles` 命令。

```
\def\generate#1{\begingroup
  \let\inputfiles\empty \let\filestogenerate\empty
  \let\file\@file
  #1
  \ifx\filestogenerate\empty\else
  \Msg{^^JGenerating file(s) \filestogenerate}\fi
  \def\inFileName{\csname in@\outFileName\endcsname}%
  \def\ReferenceLines{\csname ref@\outFileName\endcsname}%
  \processinputfiles
  \endgroup}
```

`\processinputfiles` 这是一个循环函数，处理输入文件直到它们全部处理完毕。

```
\def\processinputfiles{%
  \let\newinputfiles\empty
  \inputfiles
  \let\inputfiles\newinputfiles
  \ifx\inputfiles\empty\else
    \expandafter\processinputfiles
  \fi
  }
```

`\file` 第一个参数是要生成的文件，第二个参数包含输入文件列表。每个条目应具有以下格式：`\from{`⟨*filename.ext*⟩`}{`⟨*options*⟩`}`。

开关 \ifGenerate 最初设为 ⟨*true*⟩。

```
\def\file#1#2{\errmessage{Command `\string\file' only allowed in
                          argument to `\string\generate'}}
\def\@file#1{%
  \Generatetrue
```

接下来，我们构建输出文件的完整路径名，并检查是否需要小心地覆盖现有文件。如果用户指定 \askforoverwritetrue，我们会询问用户是否要覆盖现有文件。否则，我们直接进行操作。

```
  \makepathname{#1}%
  \ifaskforoverwrite
```

我们尝试打开具有输出文件名的文件以进行读取。如果成功，表示文件已存在，我们会询问用户是否要覆盖该文件。

```
    \immediate\openin\inFile\@pathname\relax
    \ifeof\inFile\else
      \Ask\answer{File \@pathname\space already exists
                  \ifx\empty\destdir somewhere \fi
                  on the system.^^J%
                  Overwrite it%
                  \ifx\empty\destdir\space if necessary\fi
                  ? [y/n]}%
```

根据用户的回答设置开关 \ifGenerate 。我们允许“y”和“yes”两种回答。

```
      \ifx\y  \answer \else
      \ifx\yes\answer \else
                      \Generatefalse\fi\fi\fi
```

不要忘记关闭刚打开的文件，因为我们希望写入它。

```
    \closein\inFile
  \fi
```

如果要生成文件，我们保存其目标路径名，并将控制传递给宏 \@fileX 。注意，在调用 \@fileX 之前，文件名被转换为控制序列，并跳过了 \else 分支。

```
  \ifGenerate
  \Name\let{pth@#1}\@pathname
  \@addto\filestogenerate{\@pathname\space}%
  \Name\@fileX{#1\expandafter}%
  \else
```

如果我们不被允许覆盖现有文件，我们告知用户我们不会生成他的文件，并吞掉 \from 规范。

```
    \Msg{Not generating file \@pathname^^J}%
  \expandafter\@gobble
```

```
  \fi
  }
```

\@fileX 我们将当前输出文件的名称放入 \curout，并将 \curinfiles（此输出文件的源文件列表）初始化为空——这些将被 \from 使用。然后我们开始定义当前文件的导言。

```
\def\@fileX#1#2{%
  \chardef#1=\stream@closed
  \def\curout{#1}%
  \let\curinfiles\empty
  \let\curinnames\empty
  \def\curref{\MetaPrefix ^^J%
              \MetaPrefix\space The original source files were:^^J%
              \MetaPrefix ^^J}%
```

接下来执行第二个参数。\from 将向导言添加参考行。

```
  \let\from\@from \let\needed\@needed
  #2%
  \let\from\err@from \let\needed\err@needed
```

我们检查输入文件的顺序。

```
  \checkorder
```

每个 \from 子句将其第一个参数定义为 \curin 。现在 \curin 包含当前输出文件的最后一个输入文件的名称。这意味着在处理 \curin 后应关闭当前输出文件。我们将 #1 添加到适当的“关闭列表”中。

```
  \Name\@addto{e@\curin}{\noexpand\closeoutput{#1}}%
```

最后，我们保存关于当前文件的所有有趣信息。

```
  \Name\let{pre@\@stripstring#1\expandafter}\currentpreamble
  \Name\let{post@\@stripstring#1\expandafter}\currentpostamble
  \Name\edef{in@\@stripstring#1}{\expandafter\iden\curinnames}
  \Name\edef{ref@\@stripstring#1}{\curref}
  }
```

\checkorder 此宏检查 \curinfiles 中文件的顺序是否与 \inputfiles 中的顺序一致。编码有些笨拙。

```
\def\checkorder{%
  \expandafter\expandafter\expandafter
  \checkorderX\expandafter\curinfiles
  \expandafter\qStop\inputfiles\qStop
  }
\def\checkorderX(#1)#2\qStop#3\qStop{%
```

```
  \def\tmp##1\readsource(#1)##2\qStop{%
    \ifx!##2! \order@error
    \else\ifx!#2!\else
       \checkorderXX##2%
    \fi\fi}%
  \def\checkorderXX##1\readsource(#1)\fi\fi{\fi\fi
    \checkorderX#2\qStop##1\qStop}%
  \tmp#3\readsource(#1)\qStop
  }
\def\order@error#1\fi\fi{\fi
  \errmessage{DOCSTRIP error: Incompatible order of input
              files specified for file
              `\iden{\expandafter\uptospace\curin} \qStop'.^^J
              Read DOCSTRIP documentation for explanation.^^J
              This is a serious problem, I'm exiting}\end
  }
```

\needed
\@needed

这个宏将传递的输入文件的名称变为唯一，并标记为需要输入。它在 \from 内部使用，但也可以作为 \file 的参数使用，以影响读取文件的顺序。

```
\def\needed#1{\errmessage{\string\needed\space can only be used in
                argument to \string\file}}
\let\err@needed\needed
\def\@needed#1{%
  \edef\reserved@a{#1}%
  \expandafter\@need@d\expandafter{\reserved@a}}
\def\@need@d#1{%
  \@ifpresent{(#1)}\curinfiles
```

如果 #1 出现在当前输出文件的输入文件列表中，我们在其名称末尾添加一个空格并重试。这个想法是构造一个在 TEX 中看起来不同但在操作系统中看起来相同的名称。

```
    {\@need@d{#1 }}%
```

如果不是，我们检查 #1 是否出现在要处理的文件列表中。如果没有，我们将其添加，并初始化该输入文件的输出文件列表和应在该文件关闭时关闭的输出文件列表。我们还将构造的名称添加到 \curinfiles 中，并定义 \curin，以便从 \@from 中访问构造的名称。

```
    {\@ifpresent{\readsource(#1)}\inputfiles
       {}{\@addto\inputfiles{\noexpand\readsource(#1)}%
       \Name\let{b@#1}\empty
       \Name\let{o@#1}\empty
       \Name\let{e@#1}\empty}%
```

```
    \@addto\curinfiles{(#1)}%
    \def\curin{#1}}%
  }
```

\from `\from` 开始时向输出文件的导言添加一行。

```
\def\from#1#2{\errmessage{Command `\string\from' only allowed in
                          argument to `\string\file'}}
\let\err@from\from
\def\@from#1#2{%
  \@addto\curref{\MetaPrefix\space #1 \if>#2>\else
                                \space (with options: `#2')\fi^^J}%
```

然后我们将文件标记为需要输入文件。

```
  \needed{#1}%
```

如果这是当前 `\file` 中的第一个 `\from`（即到目前为止 `\curinnames` 为空），则文件名将添加到当前输入文件的“开放列表”中。并将 `\do`⟨*current output*⟩`{`⟨*options*⟩`}` 添加到当前输入文件的输出文件列表中。

```
  \ifx\curinnames\empty
     \Name\@addto{b@\curin}{\noexpand\openoutput\curout}%
  \fi
  \@addto\curinnames{ #1}%
  \Name\@addto{o@\curin}{\noexpand\do\curout{#2}}%
  }
```

\readsource 对每个要处理的输入文件调用此宏。

```
\def\readsource(#1){%
```

我们尝试打开输入文件。如果失败，我们告诉用户并且什么也不会发生。

```
  \immediate\openin\inFile\uptospace#1 \qStop\relax
  \ifeof\inFile
    \errmessage{Cannot find file \uptospace#1 \qStop}%
  \else
```

如果包含统计信息，我们将行计数器归零

```
⟨*stats⟩
    \processedLines\z@
    \commentsRemoved\z@
    \commentsPassed\z@
    \codeLinesPassed\z@
⟨/stats⟩
```

当成功打开输入文件时,我们尝试执行“开放列表”以打开所有需要的输出文件。如果任何文件由于流限制而无法打开，则它们的名称被放入 \refusedfiles 列表中。此列表随后成为下一次传递的开放列表。

```
\let\refusedfiles\empty
\csname b@#1\endcsname
\Name\let{b@#1}\refusedfiles
```

现在所有可以打开的输出文件都已打开。所以我们遍历“输出列表”，对于每个打开的文件，我们显示一条消息并将离线计数器归零，而关闭的文件则被附加到 \refusedfiles 。

```
\Msg{} \def\@msg{Processing file \uptospace#1 \qStop}
\def\change@msg{%
  \edef\@msg{\@spaces\@spaces\@spaces\space
    \expandafter\tospaces\uptospace#1 \qStop\secapsot}
  \let\change@msg\relax}
\let\do\showfiles@do
\let\refusedfiles\empty
\csname o@#1\endcsname
```

如果 \refusedfiles 不为空，则当前源文件需要重新读取，所以我们将其附加到 \newinputfiles 。

```
\ifx\refusedfiles\empty\else
  \@addto\newinputfiles{\noexpand\readsource(#1)}
\fi
```

最后，我们定义 \outputfiles 并构造离线计数器名称。现在 \do 将有 3 个参数！所有输出文件都变为活动状态。

```
\let\do\makeoutlist@do
\edef\outputfiles{\csname o@#1\endcsname}%
\let\activefiles\outputfiles
\Name\let{o@#1}\refusedfiles
```

现在我们将许多字符的类别码更改为 $\langle other \rangle$，并通过将 \endlinechar 设置为 -1，确保读取的行中没有额外的空格出现。

```
\makeOther\ \makeOther\\\makeOther\$%
\makeOther\#\makeOther\^\makeOther\^^K%
\makeOther\_\makeOther\^^A\makeOther\%%
\makeOther\~\makeOther\{\makeOther\}\makeOther\&%
\endlinechar-1\relax
```

为了避免对任何 UTF-8 字符的处理，我们将代码点 128–255 设置为其他类别码。

```
\@tempcnta=128\relax
```

```
    \loop
    \catcode\@tempcnta 12\relax
    \ifnum\@tempcnta <255\relax
    \advance\@tempcnta\@ne
    \repeat
```

然后我们开始一个循环，逐行处理文件中的行。

```
    \loop
      \read\inFile to\inLine
```

我们首先检查的是当前行是否包含 `\endinput` 。为了允许源文件中的真实 `\endinput` 命令，只有当它直接出现在行的开头时才会识别 `\endinput` 。

```
      \ifx\inLine\Endinput
```

在这种情况下，我们输出一条消息通知程序员（如果这是一个错误），并立即通过将 `Continue` 设置为 $\langle false \rangle$ 结束循环。注意，`\endinput` 不会放入输出文件中。这在输出文件是从多个 `doc` 文件生成的情况下很重要。

```
        \Msg{File #1 ended by \string\endinput.}%
        \Continuefalse
      \else
```

当发现文件结束时，我们必须中断循环。

```
       \ifeof\inFile
         \Continuefalse
```

如果文件没有结束，我们检查输入行是否为空。如果是，计数器 `\emptyLines` 就会增加。

```
       \else
         \Continuetrue
         \ifx\inLine\empty
            \advance\emptyLines\@ne
          \else
            \emptyLines\z@
          \fi
```

当迄今为空行的数量超过 1 时，我们跳过它们。如果没有，`\inLine` 的展开被传递给 `\processLine`，并附加 `\endLine` 表示行的结尾。

```
          \ifnum \emptyLines<2
            \expandafter\processLine\inLine\endLine
          \else
            \maybeMsg{/}%
          \fi
       \fi
      \fi
```

处理完该行后，我们检查是否还有更多工作要做，如果是，则重复循环。

```
    \ifContinue
    \repeat
```

输入文件被关闭。

```
    \closein\inFile
```

我们关闭了这个文件是最后一个输入文件的输出文件。

```
  \csname e@#1\endcsname
```

如果用户对统计数据感兴趣，我们会告知他处理的行数，已删除或传递的注释数，以及写入输出文件的代码行数。还会更新总数。

```
⟨*stats⟩
    \Msg{Lines \space processed: \the\processedLines^^J%
         Comments removed: \the\commentsRemoved^^J%
         Comments \space passed: \the\commentsPassed^^J%
         Codelines passed: \the\codeLinesPassed^^J}%
      \global\advance\TotalprocessedLines  by \processedLines
      \global\advance\TotalcommentsRemoved by \commentsRemoved
      \global\advance\TotalcommentsPassed  by \commentsPassed
      \global\advance\TotalcodeLinesPassed by \codeLinesPassed
⟨/stats⟩
```

即使没有收集统计信息，也需要知道 `\NumberOfFiles`，所以我们总是更新它。

```
      \global\advance\NumberOfFiles by \@ne
    \fi}
```

`\showfiles@do` 在终端显示一条消息，告诉用户我们即将要做什么。对于每个打开的输出文件，我们显示一行内容，说明它是用哪些选项生成的，并将与文件关联的离线计数器归零。第一行还包含输入文件的名称。关闭的输出文件的名称将附加到 `\refusedfiles` 。

```
\def\showfiles@do#1#2{%
  \ifnum#1=\stream@closed
    \@addto\refusedfiles{\noexpand\do#1{#2}}%
  \else
    \Msg{\@msg
         \ifx>#2>\else\space(#2)\fi
         \space -> \@stripstring#1}
    \change@msg
  \csname off@\number#1\endcsname=\z@
  \fi
}
```

\makeoutlist@do 此宏仅选择已打开的输出文件，并构建离线计数器的名称。

```
\def\makeoutlist@do#1#2{%
  \ifnum#1=\stream@closed\else
    \noexpand\do#1{#2}\csname off@\number#1\endcsname
  \fi}
```

\openoutput 此宏在可能的情况下打开输出流。

```
\def\openoutput#1{%
```

如果两个 maxfile 计数器都不为零...

```
  \if 1\ifnum\@maxfiles=\z@ 0\fi
        \ifnum\@maxoutfiles=\z@ 0\fi1%
```

...可以打开流并递减计数器。但如果不能...

```
    \advance\@maxfiles\m@ne
    \advance\@maxoutfiles\m@ne
    \StreamOpen#1%
    \WritePreamble#1%
  \else
```

...文件将被添加到“拒绝列表”。

```
     \@addto\refusedfiles{\noexpand\openoutput#1}%
  \fi
  }
```

\closeoutput 此宏在不再需要时关闭打开的输出流，并增加 maxfiles 计数器。

```
\def\closeoutput#1{%
  \ifnum#1=\stream@closed\else
    \WritePostamble#1%
    \StreamClose#1%
    \advance\@maxfiles\@ne
    \advance\@maxoutfiles\@ne
  \fi}
```

9.7.1 导言与后文

\ds@heading 这是几行文字，说明正在写入的文件以及它是如何生成的。

```
\def\ds@heading{%
  \MetaPrefix ^^J%
  \MetaPrefix\space This is file `\outFileName',^^J%
  \MetaPrefix\space  generated with the docstrip utility.^^J%
  }
```

\AddGenerationDate DocStrip 的旧版本会添加任何文件生成的日期和 DocStrip 的版本号。这使一些人感到困惑，因为他们误以为这是正在写入的文件的版本/日期。因此，现在通常不会写入这些信息，但是批处理文件可以调用此命令以获得旧格式的标题。

```
\def\AddGenerationDate{%
  \def\ds@heading{%
    \MetaPrefix ^^J%
    \MetaPrefix\space This is file `\outFileName', generated %
          on <\the\year/\the\month/\the\day> ^^J%
    \MetaPrefix\space with the docstrip utility (\fileversion).^^J%
 }}
```

\declarepreamble 当使用批处理文件时，用户可以指定自己的导言内容，这些内容将写入每个创建的文件中。这对于在文件的剥离版本中包含额外的版权声明可能很有用。同时，可以通过在这样的导言中写入一个警告，告知始终应一起分发文件的两个版本。

写入 \outFile 的属于导言的每一行都以两个百分号字符开头。这将阻止 DocStrip 剥离文件中的这些行。

导言应以宏 \declarepreamble 开始；通过 \endpreamble 结束。为了能够局部更改一些值，所有处理都在一个组内完成。

\ReferenceLines 被设为 \relax，以使其不可展开。

```
\let\inFileName\relax
\let\outFileName\relax
\let\ReferenceLines\relax
\def\declarepreamble{\begingroup
\catcode`\^^M=13 \catcode`\ =12 %
\declarepreambleX}
{\catcode`\^^M=13 %
\gdef\declarepreambleX#1#2
\endpreamble{\endgroup%
  \def^^M{^^J\MetaPrefix\space}%
  \edef#1{\ds@heading%
          \ReferenceLines%
          \MetaPrefix\space\checkeoln#2\empty}}%
\gdef\checkeoln#1{\ifx^^M#1\else\expandafter#1\fi}%
}
```

\declarepostamble 就像可以在批处理文件中指定导言一样，也可以为 后文 指定相同的内容。

\declarepostamble 的定义非常类似于上面的 \declarepreamble 的定义。

```
\def\declarepostamble{\begingroup
\catcode`\ =12 \catcode`\^^M=13
\declarepostambleX}
{\catcode`\^^M=13 %
\gdef\declarepostambleX#1#2
\endpostamble{\endgroup%
  \def^^M{^^J\MetaPrefix\space}%
  \edef#1{\MetaPrefix\space\checkeoln#2\empty^^J%
          \MetaPrefix ^^J%
          \MetaPrefix\space End of file `\outFileName'.%
  }}%
}
```

\usepreamble
\usepostamble

选择要使用的导言与后文的宏。

```
\def\usepreamble#1{\def\currentpreamble{#1}}
\def\usepostamble#1{\def\currentpostamble{#1}}
```

\nopreamble
\nopostamble

用于禁用写入 [pre/post]ambles 的快捷方式。这不是通过禁用 `\WritePreamble` 或 `\WritePostamble` 实现的, 因为那样无法撤销。相反, 空的 [pre/post]ambles 在这些宏中被特殊处理。

```
\def\nopreamble{\usepreamble\empty}
\def\nopostamble{\usepostamble\empty}
```

\preamble
\postamble

为了向后兼容，我们提供了这些宏，用于定义默认的导言与后文。

```
\def\preamble{\usepreamble\defaultpreamble
  \declarepreamble\defaultpreamble}
\def\postamble{\usepostamble\defaultpostamble
  \declarepostamble\defaultpostamble}
```

\org@preamble
\org@postamble

如果没有提供不同的值，则使用的默认值。

```
\declarepreamble\org@preamble

IMPORTANT NOTICE:

For the copyright see the source file.

Any modified versions of this file must be renamed
with new filenames distinct from \outFileName.

For distribution of the original source see the terms
for copying and modification in the file \inFileName.
```

```
807
808 This generated file may be distributed as long as the
809 original source files, as listed above, are part of the
810 same distribution. (The sources need not necessarily be
811 in the same archive or directory.)
812 \endpreamble

813 \edef\org@postamble{\string\endinput^^J%
814   \MetaPrefix ^^J%
815   \MetaPrefix\space End of file `\outFileName'.%
816   }

817 \let\defaultpreamble\org@preamble
818 \let\defaultpostamble\org@postamble

819 \usepreamble\defaultpreamble
820 \usepostamble\defaultpostamble
```

\originaldefault 默认的导言区头在 v2.5 中更改，以允许分发生成的文件，只要源文件也被分发。如果你需要原始的默认设置，不允许分发生成的文件，在你的.ins 文件中添加 `\usepreamble\originaldefault` 。然后需要注意的是，你的文件不能被包含在大多数预装有所有 LaTeX 文件并移至合适目录的 TeX 搜索路径中的 CD 中的大多数 TeX 发行版中。

```
821 \declarepreamble\originaldefault
822
823 IMPORTANT NOTICE:
824
825 For the copyright see the source file.
826
827 You are *not* allowed to modify this file.
828
829 You are *not* allowed to distribute this file.
830 For distribution of the original source see the terms
831 for copying and modification in the file \inFileName.
832
833 \endpreamble
```

\WritePreamble

```
834 \def\WritePreamble#1{%
```

我们只写出非空的导言区。

```
835   \expandafter\ifx\csname pre@\@stripstring#1\endcsname\empty
836   \else
837     \edef\outFileName{\@stripstring#1}%
```

接着是引用行，告知从哪些源文件创建了被剥离文件以及用户提供的导言区。

```
    \StreamPut#1{\csname pre@\@stripstring#1\endcsname}%
  \fi}
```

\WritePostamble 后文归属于#1被写出。最后一行标识了文件的身份。

```
\def\WritePostamble#1{%
```

我们只写出非空的后文。

```
  \expandafter\ifx\csname post@\@stripstring#1\endcsname\empty
  \else
    \edef\outFileName{\@stripstring#1}%
    \StreamPut#1{\csname post@\@stripstring#1\endcsname}%
  \fi}
```

9.8 支持写入指定目录

正如我们之前所见，每个输出文件都写入由 \destdir 的值所指定的目录，该值在此文件的 \file 声明时是当前的。

\usedir 当调用此宏时，它应该将其一个参数转换为一个目录名称，并将 \destdir 定义为该值。\usedir 的默认行为是忽略其参数并返回当前目录的名称（如果已知）。这可以通过 docstrip.cfg 文件中的命令进行更改。

\showdirectory 仅用于显示目录名称供用户参考。

```
\def\usedir#1{\edef\destdir{\WriteToDir}}
\def\showdirectory#1{\WriteToDir}
```

\BaseDirectory 这是用于指定 TEX 层次结构根目录的配置文件命令。它通过重新定义 \usedir 来启用整个目录选择机制。首先确保目录语法命令已经被设置，通过调用 \@setwritedir，以便在 \edef 使用的 \dirsep 的值（希望是）正确的。

```
\def\BaseDirectory#1{%
  \@setwritetodir
  \let\usedir\alt@usedir
  \let\showdirectory\showalt@directory
  \edef\basedir{#1\dirsep}}
```

\convsep 此宏循环遍历其参数中的斜杠，并用当前的 \dirsep 替换它们。应该调用 \convsep some/directory/name/\qStop（以斜杠结尾）。

```
\def\convsep#1/#2\qStop{%
  #1\ifx\qStop#2\qStop \pesvnoc\fi\convsep\dirsep#2\qStop}
\def\pesvnoc#1\qStop{\fi}
```

\alt@usedir 目录名称构建宏，使写入到不同的目录成为可能。

```
\def\alt@usedir#1{%
  \Name\ifx{dir@#1}\relax
    \undefined@directory{#1}%
  \else
    \edef\destdir{\csname dir@#1\endcsname}%
  \fi}
\def\showalt@directory#1{%
  \Name\ifx{dir@#1}\relax
    \showundef@directory{#1}%
  \else\csname dir@#1\endcsname\fi}
```

\undefined@directory 当发现未定义的标签时，此宏开始工作。其作用是引发错误，并将 \destdir 定义为指向当前目录。

```
\def\undefined@directory#1{%
  \errhelp{docstrip.cfg should specify a target directory for^^J%
   #1 using \DeclareDir or \UseTDS.}%
  \errmessage{You haven't defined the output directory for `#1'.^^J%
            Subsequent files will be written to the current directory}%
  \let\destdir\WriteToDir
  }
\def\showundef@directory#1{UNDEFINED (label is #1)}
```

\undefined@TDSdirectory 当在使用 TDS 时标签未定义时会发生这种情况。标签将被转换为使用正确的分隔符，然后附加到基本目录名称后面。

```
\def\undefined@TDSdirectory#1{%
  \edef\destdir{%
    \basedir\convsep#1/\qStop
  }}
\def\showundef@TDSdirectory#1{\basedir\convsep#1/\qStop}
```

\UseTDS 未定义标签的行为变更非常简单：

```
\def\UseTDS{%
  \@setwritetodir
  \let\undefined@directory\undefined@TDSdirectory
  \let\showundef@directory\showundef@TDSdirectory
  }
```

\DeclareDir 此宏将某个目录名称重新映射为另一个目录。

```
\def\DeclareDir{\@ifnextchar*{\DeclareDirX}{\DeclareDirX\basedir*}}
\def\DeclareDirX#1*#2#3{%
```

```
  \@setwritetodir
  \Name\edef{dir@#2}{#1#3}}
```

9.8.1 与旧版本的兼容性

\generateFile 旧版本 DocStrip 的主要宏。

```
\def\generateFile#1#2#3{{%
  \ifx t#2\askforoverwritetrue
  \else\askforoverwritefalse\fi
  \generate{\file{#1}{#3}}%
  }}
```

为了支持为第一个版本的 DocStrip 编写的命令文件,这里定义了 \include 和 \processFile 命令。不建议使用此接口，因为它可能在将来的 DocStrip 版本中被移除。

\include 为了向 DocStrip 程序提供应该包含在输出中的选项列表，可以使用命令 \include{⟨*Options*⟩} 。这个宏应该与 \processFile 命令一起使用。

```
\def\include#1{\def\Options{#1}}
```

\processFile 宏 \processFile{⟨*filename*⟩}{⟨*inext*⟩}{⟨*outext*⟩}{⟨*t*|*f*⟩} 用于当使用单个输入文件产生单个输出文件时。该宏也用于 DocStrip 程序的交互模式中。

参数 ⟨*inext*⟩ 和 ⟨*outext*⟩ 分别表示输入和输出文件的扩展名。第四个参数可用于指定是否应该在不询问的情况下覆盖现有文件。如果指定了 ⟨*t*⟩，程序将在覆盖现有文件之前询问权限。

此宏是使用更通用的宏 \generateFile 定义的。

```
\def\processFile#1#2#3#4{%
  \generateFile{#1.#3}{#4}{\from{#1.#2}{\Options}}}
```

\processfile 早期版本的 DocStrip 定义了 \processfile 和 \generatefile，而不是此版
\generatefile 本中定义的命令。为了保持向上兼容，我们仍然提供这些命令，但在使用它们时会发出警告。

```
\def\processfile{\Msg{%
    ^^Jplease use \string\processFile\space instead of
      \string\processfile!^^J}%
    \processFile}
\def\generatefile{\Msg{%
    ^^Jplease use \string\generateFile\space instead of
      \string\generatefile!^^J}%
    \generateFile}
```

9.9 限制打开的文件流

（本节由 Mark Wooding 撰写）

\maxfiles 一些具有糟糕库或其他限制的操作系统无法处理 DocStrip 一次性尝试输出的所有文件。配置文件可以使用 \maxfiles{⟨*number*⟩} 描述环境的最大限制。

我需要一个计数器来保存这个值，所以最好分配一个。

```
\newcount\@maxfiles
```

配置命令 \maxfiles 稍微比赋值漂亮一些，对于 LaTeX 用户来说。它也给了我一个机会来检查限制是否合理。我至少需要 4 个流：

1. 一个批处理文件。
2. LaTeX 安装工具使用的子批处理文件。
3. 用于读取未剥离文件的输入流。
4. 用于写入剥离文件的输出流。

```
\def\maxfiles#1{%
  \@maxfiles#1\relax
  \ifnum\@maxfiles<4
    \errhelp{I'm not a magician.  I need at least four^^J%
             streams to be able to work properly, but^^J%
             you've only let me use \the\@maxfiles.}%
    \errmessage{\noexpand\maxfiles limit is too strict.}%
    \@maxfiles4
  \fi
}
```

由于批处理文件现在是 \input，所以这里不应该有默认限制。我将使用一些抽象的大数字。

```
\maxfiles{1972} % year of my birth (MW)
```

\maxoutfiles 或许只有输出流有限制。（嗯，确实有限制：我知道，因为 TeX 只允许 16 个。）我可以让配置设置这个限制。

再次，我需要一个计数器。

```
\newcount\@maxoutfiles
```

现在是宏的内容。我认为至少需要一个输出流是合理的。

```
\def\maxoutfiles#1{%
  \@maxoutfiles=#1\relax
```

```
  \ifnum\@maxoutfiles<1
    \@maxoutfiles1
    \errhelp{I'm not a magician.  I need at least one output^^J%
             stream to be able to do anything useful at all.^^J%
             Please be reasonable.}%
    \errmessage{\noexpand\maxoutfiles limit is insane}%
  \fi
}
```

默认限制是 16，因为这是 TEX 所规定的。

```
\maxoutfiles{16}
```

\checkfilelimit 当启动新的批处理文件时，这会检查文件限制。如果这里剩余的文件少于两个，我们将无法剥离任何文件。文件限制计数器是局部于围绕 \batchinput 设置的组内的，所以这一切都相当不错。

```
\def\checkfilelimit{%
  \advance\@maxfiles\m@ne
  \ifnum\@maxfiles<2 %
    \errhelp{There aren't enough streams left to do any unpacking.^^J%
             I can't do anything about this, so complain at the^^J%
             person who made such a complicated installation.}%
    \errmessage{Too few streams left.}%
    \end
  \fi
}
```

9.10 与用户的交互

\strip@meaning 丢弃 \meaning 输出的第一部分。

```
\def\strip@meaning#1>{}
```

\processbatchFile 当运行 DocStrip 时，它总是尝试使用一个批处理文件。

为此，它调用了宏 \processbatchFile 。

首先要做的是检查是否还有输入流。

```
\def\processbatchFile{%
  \checkfilelimit
  \let\next\relax
```

现在我们尝试打开批处理文件进行读取。

```
  \openin\inputcheck \batchfile\relax
  \ifeof\inputcheck
```

如果我们没有成功打开文件，则假设它不存在。如果我们尝试了默认文件名，我们会默默地继续；在这种情况下，DocStrip 程序将切换到交互模式。

```
    \ifDefault
    \else
```

如果我们无法打开用户提供的文件，表示出现了问题，我们会警告用户。这也将导致切换到交互模式。

```
      \errhelp
        {A batchfile specified in \batchinput could not be found.}%
      \errmessage{^^J%
           **************************************************^^J%
           * Could not find your \string\batchfile=\batchfile.^^J%
           **************************************************}%
    \fi
  \else
```

当我们成功打开一个文件时，我们需要再次检查它是否是默认文件。在这种情况下，我们告诉用户我们找到了那个文件，并询问他是否想要使用它。

```
    \ifDefault
      \Msg{**************************************************^^J%
           * Batchfile \DefaultbatchFile\space found Use it? (y/n)?}%
      \Ask\answer{%
            **************************************************}%
    \else
```

如果它是用户提供的文件，我们可以安全地假设他想要使用它，所以我们将 `\answer` 设置为 `y`。

```
      \let\answer\y
    \fi
```

如果宏 `\answer` 包含 `y`，我们可以读取批处理文件。我们以一种间接的方式进行—在完成 `\if` 后。

```
    \ifx\answer\y
      \closein\inputcheck
      \def\next{\@@input\batchfile\relax}%
    \fi
  \fi
  \next}
```

`\ReportTotals` 宏 `\ReportTotals` 用于报告所有处理过的文件的总体统计信息。只有在包含选项 `stats` 的情况下，此代码才会包含在程序中。

```
⟨*stats⟩
\def\ReportTotals{%
```

```
  \ifnum\NumberOfFiles>\@ne
    \Msg{Overall statistics:^^J%
          Files \space processed: \the\NumberOfFiles^^J%
          Lines \space processed: \the\TotalprocessedLines^^J%
          Comments removed: \the\TotalcommentsRemoved^^J%
          Comments \space passed: \the\TotalcommentsPassed^^J%
          Codelines passed: \the\TotalcodeLinesPassed}%
  \fi}
⟨/stats⟩
```

\SetFileNames 当程序以交互模式运行并且要求用户提供扩展名和文件名列表时，会使用宏 \SetFileNames 。

```
\def\SetFileNames{%
    \edef\sourceFileName{\MainFileName.\infileext}%
    \edef\destFileName{\MainFileName.\outfileext}}
```

\CheckFileNames 在交互模式下，用户被要求提供输入和输出文件的扩展名。还需要输入文件的名称或名称（不带扩展名）。然后，通过 \SetFileNames 根据这些信息构建输入和输出文件的名称。这假设输入文件的名称与输出文件的名称相同。但我们不应该将内容写入我们正在读取的文件，因此扩展名应该不同。

宏 \CheckFileNames 确保输出到一个与输入不同的文件。

```
\def\CheckFileNames{%
    \ifx\sourceFileName\destFileName
```

如果输入和输出文件相同，我们会发出错误信号并停止处理。

```
      \Msg{^^J%
     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^J%
     ! It is not possible to read from and write to the same file !^^J%
     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^J}%
      \Continuefalse
    \else
```

如果它们不同，我们检查输入文件是否存在，通过尝试打开它进行读取。

```
      \Continuetrue
      \immediate\openin\inFile \sourceFileName\relax
      \ifeof\inFile
```

如果找到文件结束符，则无法打开文件，因此我们会发出错误信号并停止处理。

```
        \Msg{^^J%
              !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^J%
              ! Your input file `\sourceFileName' was not found !^^J%
```

```
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^J}%
      \Continuefalse
    \else
```

我们需要进行的最后一项检查是输出文件是否已经存在。因此，我们尝试以读取模式打开它。作为预防措施，我们首先关闭输入流。

```
      \immediate\closein\inFile
      \immediate\openin\inFile\destdir \destFileName\relax
      \ifeof\inFile
```

如果打开输出文件进行读取失败，表示该文件不存在，一切都很好。

```
        \Continuetrue
      \else
```

如果成功打开输出文件进行读取，我们需要询问用户是否要覆盖它。我们假设他不想覆盖它，因此开关 `\ifContinue` 最初设置为 ⟨*false*⟩。只有当他肯定地回答问题，即使用 'y' 或 'yes'，我们才将开关设置回 ⟨*true*⟩。

```
        \Continuefalse
        \Ask\answer{File \destdir\destFileName\space already
                    exists
                    \ifx\empty\destdir somewhere \fi
                    on the system.^^J%
                    Overwrite it%
                    \ifx\empty\destdir\space if necessary\fi
                    ? [y/n]}%
        \ifx\y  \answer \Continuetrue \else
        \ifx\yes\answer \Continuetrue \else
        \fi\fi
      \fi
```

所有检查现在都已完成，因此我们可以关闭仅为此目的而打开的任何文件。

```
    \fi
  \fi
  \closein\inFile}
```

`\interactive` 宏 `\interactive` 实现了 DocStrip 程序的交互模式。该宏使用 ⟨*while*⟩ 结构实现。只要开关 `\ifMoreFiles` 保持为真，我们就会继续处理。

```
\def\interactive{%
 \whileswitch\ifMoreFiles\fi%
```

为了保持宏重定义的局部性，我们开始一个组并询问用户一些关于他希望我们做什么的问题。

```
  {\begingroup
    \AskQuestions
```

待处理的文件名称以逗号分隔的列表的形式存储在宏 \filelist 中，由 \AskQuestions 生成。我们使用一个 ⟨*for*⟩ 循环逐个处理这些文件。

```
      \forlist\MainFileName:=\filelist
      \do
```

首先构造输入和输出文件的名称，并检查所有的文件名信息是否正确。

```
        \SetFileNames
        \CheckFileNames
        \ifContinue
```

如果一切正常，生成输出文件。

```
        \generateFile{\destFileName}{f}%
                    {\from{\sourceFileName}{\Options}}
        \fi%
```

这个过程重复进行，直到 \filelist 被处理完。

```
      \od
    \endgroup
```

也许用户希望处理更多文件，可能是使用另一组选项，因此我们给他这个机会。

```
    \Ask\answer{More files to process (y/n)?}%
    \ifx\y  \answer\MoreFilestrue \else
    \ifx\yes\answer\MoreFilestrue \else
```

如果他不想处理更多文件，为了中断 ⟨*while*⟩ 循环，将开关 \ifMoreFiles 设置为 ⟨*false*⟩。

```
                    \MoreFilesfalse\fi\fi
  }}
```

\AskQuestions 宏 \AskQuestions 被 \interactive 调用，以获取用户关于需要处理的文件的一些信息。

```
\def\AskQuestions{%
    \Msg{^^J%
        **************************************************}%
```

我们想要知道输入文件的扩展名，

```
    \Ask\infileext{%
        * First type the extension of your input file(s): \space  *}%
    \Msg{**************************************************^^J^^J%
        **************************************************}%
```

输出文件的扩展名，

```
    \Ask\outfileext{%
```

```
        * Now type the extension of your output file(s) \space: *}%
    \Msg{**************************************************^^J^^J%
        **************************************************}%
```

是否要包含选项，

```
    \Ask\Options{%
       * Now type the name(s) of option(s) to include \space\space: *}%
    \Msg{**************************************************^^J^^J%
        **************************************************^^J%
       * Finally give the list of input file(s) without \space\space*}%
```

输入文件的名称或名称列表，用逗号分隔。

```
    \Ask\filelist{%
        * extension separated by commas if necessary %
                                  \space\space\space\space: *}%
    \Msg{**************************************************^^J}}%
```

9.11 主程序

当 TEX 处理 DocStrip 程序时，它会在终端上显示关于程序版本及其功能的消息。

```
\Msg{Utility: `docstrip' \fileversion\space <\filedate>^^J%
     English documentation \space\space\space <\docdate>}%
\Msg{^^J%
     ************************************************************^^J%
     * This program converts documented macro-files into fast *^^J%
     * loadable files by stripping off (nearly) all comments! *^^J%
     ************************************************************^^J}%
```

\WriteToDir 宏 \WriteToDir 要么为空，要么保存从当前目录读取文件所需的前缀。在 UNIX 下，这个前缀是 ./ ，但很多其他系统也采用了这个概念。这个宏是 \destdir 的默认值。

此宏的定义现在被延迟到 \@setwritedir 被调用时。

\makepathname 此宏应该将 \@pathname 定义为由当前 \destdir 值与其参数（文件名）组合而成的完整路径名。在此定义的默认值适用于 UNIX，MS-DOS 和 Macintosh，但对于某些系统可能需要在 docstrip.cfg 文件中重新定义此值。我们在此提供了 VMS 系统的重新定义。

宏 \dirsep 包含特定于系统的目录分隔符。适用于 UNIX 和 DOS 的默认值是斜杠。当直接使用 \usedir 标签时，它生效。

此宏的定义现在被延迟到 \@setwritedir 被调用时。

\@setwritedir 以下测试尝试自动设置宏 \WriteToDir、\dirname 和 \makepathname，以 Unix、Mac 或 VMS 样式为准。这些测试没有在顶层运行，而是被保存在此宏中，以便配置文件有机会定义 \WriteToDir ，从而允许其他两个宏自动设置。这些测试可以更简单地在读取配置文件后运行，但是配置命令（例如 \BaseDirectory）需要（至少目前需要）正确定义 \dirsep 。它不会定义已经定义的任何命令，因此通过定义这些命令，配置文件可以针对特殊需求产生不同的效果。因此，此命令由 BaseDirectory、\UseTDS、\DeclareDir 并最终在 cfg 运行后的顶层调用。它首先重新定义自身为无操作，因此它实际上只被调用一次。

```
\def\@setwritetodir{%
  \let\setwritetodir\relax
  \ifx\WriteToDir\@undefined
    \ifx\@currdir\@undefined
      \def\WriteToDir{}%
    \else
      \let\WriteToDir\@currdir
    \fi
  \fi
  \let\destdir\WriteToDir
```

VMS 风格。

```
  \def\tmp{[]}%
  \ifx\tmp\WriteToDir
    \ifx\dirsep\@undefined
      \def\dirsep{.}%
    \fi
    \ifx\makepathname\@undefined
      \def\makepathname##1{%
        \edef\@pathname{\ifx\WriteToDir\destdir
          \WriteToDir\else[\destdir]\fi##1}}%
    \fi
  \fi
```

Unix 和 Mac 风格。

```
  \ifx\dirsep\@undefined
    \def\dirsep{/}%
    \def\tmp{:}%
    \ifx\tmp\WriteToDir
      \def\dirsep{:}%
    \fi
  \fi
```

```
  \ifx\makepathname\@undefined
    \def\makepathname##1{%
      \edef\@pathname{\destdir\ifx\empty\destdir\else
           \ifx\WriteToDir\destdir\else\dirsep\fi\fi##1}}%
  \fi}
```

如果用户有一个 docstrip.cfg 文件，现在使用它。此宏尝试读取 docstrip.cfg 文件。如果成功，则执行其第一个参数；否则执行第二个参数。

```
\immediate\openin\inputcheck=docstrip.cfg\relax
\ifeof\inputcheck
  \Msg{%
     ********************************************************^^J%
     * No Configuration file found, using default settings. *^^J%
     ********************************************************^^J}%
\else
  \Msg{%
     ****************************************^^J%
     * Using Configuration file docstrip.cfg. *^^J%
     ****************************************^^J}%
  \closein\inputcheck
  \afterfi{\@@input docstrip.cfg\relax}
\fi
```

现在运行 \@setwritedir，以防尚未通过配置文件中的命令运行过它。

```
\@setwritetodir
```

\process@first@batchfile 处理批处理文件，然后正常终止。对于不以 \def\batchfile{…开头的“新风格”批处理文件，可以将其设置为 \relax 。

```
\def\process@first@batchfile{%
  \processbatchFile
  \ifnum\NumberOfFiles=\z@
    \interactive
  \fi
  \endbatchfile}
```

\endbatchfile 用户级别命令，用于结束批处理文件处理。在顶层，返回总计然后停止 TEX。在嵌套级别，只执行 \endinput 。

```
\def\endbatchfile{%
  \iftopbatchfile
⟨*stats⟩
    \ReportTotals
```

```
⟨/stats⟩
    \expandafter\end
  \else
    \endinput
  \fi}
```

现在我们来看是否要处理批处理文件。

\@jobname 作业名称（catcode 12）

```
\edef\@jobname{\lowercase{\def\noexpand\@jobname{\jobname}}}%
\@jobname
```

\@docstrip docstrip（catcode 12）

```
\def\@docstrip{docstrip}%
\edef\@docstrip{\expandafter\strip@meaning\meaning\@docstrip}
```

首先检查用户是否定义了控制序列 \batchfile。如果定义了，它应该包含要处理的文件名。如果没有定义，尝试当前文件，除非当前文件是 docstrip.tex，在这种情况下会尝试默认名称。是否使用默认批处理文件由将开关 \ifDefault 设为 ⟨*true*⟩ 或 ⟨*false*⟩ 来记住。

```
\Defaultfalse
\ifx\undefined\batchfile
```

\@jobname 是小写的作业名称（字符编码 12）
\@docstrip 是 docstrip（字符编码 12）

```
  \ifx\@jobname\@docstrip
```

设置批处理文件为默认值

```
    \let\batchfile\DefaultbatchFile
    \Defaulttrue
```

否则不处理新的批处理文件，直接继续超出本文件的部分。在这种情况下，处理将移动到初始批处理文件，该文件 *必须* 通过 \endbatchfile 终止，否则 TEX 将会陷入星号提示符。

```
  \else
    \let\process@first@batchfile\relax
  \fi
\fi
\process@first@batchfile
⟨/program⟩
```

Change History