# 面向作者的 LaTeX— 当前版本

© 版权 2020-2023, LaTeX Project Team.
版权所有*

张泓知　　翻译

2023 年 05 月 23 日

## 目　录



*本文件可根据 LaTeX 项目公共许可证的条件进行分发和/或修改，可以选择本许可证的 1.3c 版本或（自选）以后的版本 (LPPL v1.3c)。请参阅源文件 `usrguide.tex` 以获取完整详情。

# 1 介绍

LaTeX 2$_\varepsilon$于 1994 年发布，并为 LaTeX 增添了许多当时的新概念。这些概念在 usrguide-historic 中有描述，并且在很大程度上保持不变。自那时起，LaTeX 团队已经致力于许多想法，首先是 LaTeX 的编程语言（expl3），然后是一系列建立在该语言基础上的文档作者工具。在这里，我们描述了从这项工作中产生的*稳定*且*广泛可用*的概念。这些 '新' 概念已从开发包转移到LaTeX 2$_\varepsilon$内核中。因此，它们现在对*所有*的 LaTeX 用户都可用，并且具有与内核的任何其他部分一样的*稳定性*。它们 '幕后' 建立在 expl3 上的事实对于开发团队很有用，但对用户来说并不直接重要。

# 2 创建文档命令和环境

## 2.1 概述

使用 LaTeX3 工具集创建文档命令和环境的基本思想是，可以使用一组通用描述来涵盖几乎所有实际文档中使用的参数类型。因此，解析被简化为对命令接受哪些参数的简单描述：这个描述提供了文档语法与命令实现之间的 '粘合剂'。

首先，我们将描述参数类型，然后介绍如何使用这些参数类型来创建文档命令和环境。随后描述了各种更专门化的特性，这些特性允许更丰富地应用简单的接口设置。

这里的细节旨在帮助用户总体上创建文档命令。适合 TeX 程序员的更多技术细节在 interface3 中有介绍。

## 2.2 描述参数类型

为了允许每个参数独立定义，解析器不仅需要知道函数的参数数量，还需要了解每个参数的性质。这是通过构建一个 *参数规范*来实现的，它定义了参数的数量、每个参数的类型以及解析器读取用户输入并正确传递给内部函数所需的任何附加信息。

参数规范的基本形式是一个字母列表，其中每个字母定义了一个参数类型。正如下面将描述的那样，某些类型需要额外的信息，比如默认值。参数类型可以分为两类，一类定义了必须的参数（如果未找到可能会引发错误），另一类定义了可选参数。必须的类型有：

`m` 标准必须参数，可以是单个独立的标记，也可以是用花括号 `{}` 包围的多个标记。无论输入如何，参数都将被传递到内部代码中而不带有外部花括号。这是用于普通 TEX 参数的类型说明符。

`r` 给定为 `r`⟨*token1*⟩⟨*token2*⟩，表示 '必需' 定界参数，其中分界符为 ⟨*token1*⟩ 和 ⟨*token2*⟩。如果缺少起始分界符 ⟨*token1*⟩，在合适的错误后将插入默认标记 `-NoValue-`。

`R` 给定为 `R`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`，这是与 `r` 类似的 '必需' 定界参数，但具有用户可定义的恢复 ⟨*default*⟩，而不是 `-NoValue-`。

`v` 以 'verbatim' 方式读取参数，在以下字符及其下一个出现之间，类似于LATEX 2ε命令`\verb` 的参数。因此，`v`-类型参数在两个相同字符之间读取，这些字符不能是 `%`、`\`、`#`、`{`、`}` 或 ␣。verbatim 参数也可以用大括号 `{` 和 `}` 括起来。带有 verbatim 参数的命令在另一个命令的参数中出现时会产生错误。

`b` 仅适用于环境的参数规范，表示环境的主体，在 `\begin{`⟨*environment*⟩`}` 和 `\end{`⟨*environment*⟩`}` 之间。详见第 2.11 节。

定义可选参数的类型包括：

`o` 标准的 LATEX 可选参数，用方括号括起来，如果未提供参数值，则提供特殊的 `-NoValue-` 标记（后面会描述）。

`d` 给定为 `d`⟨*token1*⟩⟨*token2*⟩，表示由 ⟨*token1*⟩ 和 ⟨*token2*⟩ 定界的可选参数。与 `o` 类似，如果未给出值，则返回特殊标记 `-NoValue-`。

`O` 给定为 `O{`⟨*default*⟩`}`，类似于 `o`，但如果未提供值，则返回 ⟨*default*⟩。

`D` 给定为 `D`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`，类似于 `d`，但如果未提供值，则返回 ⟨*default*⟩。在内部，`o`、`d` 和 `O` 类型是适当构造的 `D` 类型参数的简化方式。

`s` 一个可选星号，如果存在星号则返回值为 `\BooleanTrue`，否则返回值为 `\BooleanFalse`（后面会描述）。

`t` 一个可选的 ⟨*token*⟩，如果存在 ⟨*token*⟩ 则返回值为 `\BooleanTrue`，否则返回值为 `\BooleanFalse`。给定为 `t`⟨*token*⟩。

`e` 给定为 `e{`⟨*tokens*⟩`}`，一组可选的修饰符，每个修饰符需要一个值。如果某个修饰符不存在，则返回 `-NoValue-`。每个修饰符都给出一个参数，按照参数规范中 ⟨*tokens*⟩ 列表的顺序排列。所有 ⟨*tokens*⟩ 必须是不同的。

E 类似于 e，但如果未提供值，则返回一个或多个 ⟨*defaults*⟩:E{⟨*tokens*⟩} {⟨*defaults*⟩}。更多细节参见第 2.7 节。

## 2.3 修改参数描述

除了上面讨论的参数类型外，参数描述还赋予另外三个字符特殊含义。

首先，+ 用于将参数设置为长参数（接受段落标记）。与 \newcommand 不同，这适用于每个参数。因此，将示例修改为 's o o +m O{default}' 表示必需参数现在是 \long，而可选参数不是。

其次，! 用于控制在可选参数之前是否允许空格。这涉及到一些微妙之处，因为 TEX 本身对空格的 '检测' 位置有一些限制：更多细节参见第 2.6 节。

第三，= 用于声明后续参数应被解释为一系列关键键值。更多细节参见第 2.9 节。

最后，字符 > 用于声明所谓的 '参数处理器'，它们可用于修改传递到宏定义之前的参数内容。使用参数处理器是一个相对高级的主题（或者至少是一个较少使用的特性），详见第 2.10 节。

## 2.4 创建文档命令和环境

\NewDocumentCommand {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
\RenewDocumentCommand {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
\ProvideDocumentCommand {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}
\DeclareDocumentCommand {⟨*cmd*⟩} {⟨*arg spec*⟩} {⟨*code*⟩}

这组命令用于创建一个 ⟨*cmd*⟩。函数的参数规范由 ⟨*arg spec*⟩ 给出，命令使用 ⟨*code*⟩，其中 #1、#2 等被解析器找到的参数替换。

一个示例：

```
\NewDocumentCommand\chapter{s o m}
  {%
    \IfBooleanTF{#1}%
      {\typesetstarchapter{#3}}%
      {\typesetnormalchapter{#2}{#3}}%
  }
```

这将是定义一个 \chapter 命令的方法，其基本行为与当前的 LaTeX $2_\varepsilon$ 命令相同（除了在解析到 * 时也能接受可选参数）。命令 \typesetnormalchapter 可以测试其第一个参数是否为 -NoValue-，以确定可选参数是否存在（详见第 2.8 节中的 \IfBooleanTF 和 -NoValue- 的测试细节）。

\New...、\Renew...、\Provide... 和 \Declare... 版本之间的区别在于如果 ⟨*cmd*⟩ 已经被定义的行为。

- \NewDocumentCommand 如果 ⟨*cmd*⟩ 已经被定义，将会报错。
- \RenewDocumentCommand 如果 ⟨*cmd*⟩ 之前未被定义，将会报错。
- \ProvideDocumentCommand 仅在未给出定义时为 ⟨*cmd*⟩ 创建新定义。
- \DeclareDocumentCommand 将始终创建新的定义，而不考虑是否已存在同名的 ⟨*cmd*⟩。这应该谨慎使用。

如果 ⟨*cmd*⟩ 不能作为单个标记提供而需要“构建”,你可以使用 \ExpandArgs，如第 4 节中所解释的那样。该节还提供了一个需要这种方法的示例。

```
\NewDocumentEnvironment {⟨env⟩} {⟨arg spec⟩} {⟨beg-code⟩} {⟨end-code⟩}
\RenewDocumentEnvironment {⟨env⟩} {⟨arg spec⟩} {⟨beg-code⟩} {⟨end-code⟩}
\ProvideDocumentEnvironment {⟨env⟩} {⟨arg spec⟩} {⟨beg-code⟩} {⟨end-code⟩}
\DeclareDocumentEnvironment {⟨env⟩} {⟨arg spec⟩} {⟨beg-code⟩} {⟨end-code⟩}
```

这些命令的工作方式与 \NewDocumentCommand 等相同，但创建环境（\begin{⟨*env*⟩} … \end{⟨*env*⟩}）。⟨*beg-code*⟩ 和 ⟨*end-code*⟩ 都可以访问由 ⟨*arg spec*⟩ 定义的参数。这些参数将在 \begin{⟨*env*⟩} 之后给出。

## 2.5 可选参数

与使用 LaTeX $2_\varepsilon$ 的 \newcommand 创建的命令不同,使用 \NewDocumentCommand 创建的可选参数可以安全地嵌套。因此，例如，在以下示例之后，

```
\NewDocumentCommand\foo{om}{I grabbed `#1' and `#2'}
\NewDocumentCommand\baz{o}{#1-#1}
```

使用如下命令：

```
\foo[\baz[stuff]]{more stuff}
```

会显示

```
I grabbed `stuff-stuff' and `more stuff'
```

这在将一个带有可选参数的命令嵌套在第二个命令的可选参数中时特别有用。

当一个可选参数后面紧跟着使用相同定界符的必选参数时，解析器会发出警告，因为用户无法省略可选参数，实际上变成了必选参数。这适用于 o、d、O、D、s、t、e 和 E 类型的参数后面跟着 r 或 R 类型的必选参数。

对于 O、D 和 E 类型参数的默认值可以是获取另一个参数的结果。因此，例如

```
\NewDocumentCommand\foo{O{#2} m}
```

会将必选参数作为前导可选参数的默认值。

## 2.6 间距和可选参数

TEX 会在函数名称后找到第一个参数，无论中间是否有空格。这适用于必选参数和可选参数。因此，\foo[arg] 和 \foo␣[arg] 是等价的。当收集参数直到最后一个必选参数时（因为必选参数必须存在），空格也会被忽略。因此，在以下示例之后

```
\NewDocumentCommand\foo{m o m}{ ... }
```

用户输入 \foo{arg1}[arg2]{arg3} 和 \foo{arg1}␣[arg2]␣{arg3} 将被解析为相同的方式。

可选参数在任何必选参数之后的行为是可选择的。标准设置允许这里的空格，因此在以下示例之后

```
\NewDocumentCommand\foobar{m o}{ ... }
```

\foobar{arg1}[arg2] 和 \foobar{arg1}␣[arg2] 都将找到一个可选参数。可以通过在参数规范中使用修改后的 ! 来更改这种行为：

```
\NewDocumentCommand\foobar{m !o}{ ... }
```

在这里，`\foobar{arg1}␣[arg2]` 将不会找到可选参数。

这里有一个微妙之处，因为 TEX 对 '控制符号' 的处理不同，其中命令名由单个字符组成，比如 '\\'。在这种情况下，TEX 不会忽略空格，因此可以要求可选参数直接跟随这样的命令。最常见的例子是在 amsmath 环境中使用 `\\`，按照这种方式的定义将是

```
\NewDocumentCommand\\{!s !o}{ ... }
```

还需要注意的是，当在最后位置使用可选参数时，TEX 必然会预先查找参数的开启标记。这意味着如果这样的尾随可选参数*不存在*并且命令结束于一行时，`\inputlineno` 的值将 '有所偏差'；它将比包含最后一个必选参数的行号大一。

## 2.7 '修饰符'

`E` 类型参数允许每个测试标记一个默认值。这可以通过为列表中的每个条目提供默认值列表来实现，例如：

```
E{^_}{{UP}{DOWN}}
```

如果默认值列表比测试标记列表要*短*，则将返回特殊的 `-NoValue-` 标记（与 `e` 类型参数一样）。因此，例如

```
E{^_}{{UP}}
```

对于 `^` 测试字符有默认值 `UP`，但对于 `_` 则会返回 `-NoValue-` 标记作为默认值。这允许混合使用显式默认值和测试缺失值。

## 2.8 测试特殊值

可选参数利用专用变量返回有关接收到的参数性质的信息。

`\IfNoValueTF` {⟨*arg*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
`\IfNoValueT` {⟨*arg*⟩} {⟨*true code*⟩}
`\IfNoValueF` {⟨*arg*⟩} {⟨*false code*⟩}

`\IfNoValue(TF)` 测试用于检查 ⟨*argument*⟩（`#1`、`#2` 等）是否为特殊的 `-NoValue-` 标记。例如

```
\NewDocumentCommand\foo{o m}
  {%
    \IfNoValueTF {#1}%
      {\DoSomethingJustWithMandatoryArgument{#2}}%
      {\DoSomethingWithBothArguments{#1}{#2}}%
  }
```

将在可选参数存在与否时使用不同的内部函数。

需要注意的是，根据所需的结果分支，有三种测试可用：`\IfNoValueTF`、`\IfNoValueT` 和 `\IfNoValueF`。

由于 `\IfNoValue(TF)` 测试是可展开的，因此可以在稍后的时候进行测试，例如在排版的时候或扩展上下文中。

需要重要注意的是，`-NoValue-` 被构造成不会与简单的文本输入 `-NoValue-` 匹配，即

```
\IfNoValueTF{-NoValue-}
```

逻辑上将是 `false`。当两个可选参数连续出现时（我们通常不鼓励这种语法），允许命令的使用者只提供第二个参数可能是有意义的，这时可以提供一个空的第一个参数。与单独测试空值和 `-NoValue-` 相比，最好使用带有空默认值的参数类型 `O`，然后使用条件 `\IfBlankTF`（下面描述）来测试空值。

新的说明
2022/06/01

`\IfValueTF {`⟨*arg*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfValueT {`⟨*arg*⟩`} {`⟨*true code*⟩`}`
`\IfValueF {`⟨*arg*⟩`} {`⟨*false code*⟩`}`

`\IfNoValue(TF)` 测试的反向形式也可用作 `\IfValue(TF)`。根据给定代码情境，上下文将决定哪种逻辑形式对于特定的代码情况最有意义。

`\IfBlankTF {`⟨*arg*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfBlankT {`⟨*arg*⟩`} {`⟨*true code*⟩`}`
`\IfBlankF {`⟨*arg*⟩`} {`⟨*false code*⟩`}`

新的特色
2022/06/01

`\IfNoValueTF` 命令在可选参数根本没有被使用（并返回特殊的 `-NoValue-` 标记）时选择 ⟨*true code*⟩，但如果给出了空值，则不会选择。相比之下，`\IfBlankTF` 会在其参数真正为空或仅包含一个或多个普通空格时返回 true。例如

```
\NewDocumentCommand\foo{m!o}{\par #1:
  \IfNoValueTF{#2}
    {No optional}%
    {%
      \IfBlankTF{#2}
        {Blanks in or empty}%
        {Real content in}%
    }%
  \space argument!}
\foo{1}[bar] \foo{2}[  ] \foo{3}[] \foo{4}[\space] \foo{5} [x]
```

结果如下输出：

```
1: Real content in argument!
2: Blanks in or empty argument!
3: Blanks in or empty argument!
4: Real content in argument!
5: No optional argument! [x]
```

请注意，(4) 中的 \space 被视为真正的内容——因为它是一个命令而不是一个“空格”字符——即使它产生一个空格。你还可以在 (5) 中观察到 ! 修饰符的效果，它阻止最后一个 \foo 将 [x] 解释为其可选参数。

\BooleanFalse
\BooleanTrue

在搜索可选字符时设置的 true 和 false 标志（使用 s 或 t⟨*char*⟩）具有在代码块外部可访问的名称。

\IfBooleanTF {⟨*arg*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\IfBooleanT {⟨*arg*⟩} {⟨*true code*⟩}
\IfBooleanF {⟨*arg*⟩} {⟨*false code*⟩}

用于测试 ⟨*argument*⟩（#1、#2 等）是否为 \BooleanTrue 或 \BooleanFalse。例如

```
\NewDocumentCommand\foo{sm}
  {%
```

```
    \IfBooleanTF {#1}%
      {\DoSomethingWithStar{#2}}%
      {\DoSomethingWithoutStar{#2}}%
  }
```

检查第一个参数是否为星号，然后根据此信息选择要采取的动作。

## 2.9 自动转换为键-值格式

一些文档命令长期以来接受“自由文本”可选参数，例如\caption 和章节命令\section 等。因此，要向这些命令引入更复杂的（keyval）选项，需要一种方法将可选参数同时解释为自由文本或一系列键值对。这需要在参数抓取期间进行，因为需要仔细处理大括号以获得正确的结果。

= 修饰符可用于允许 ltcmd 正确实现此过程。该修饰符保证参数将作为一系列键值对传递给后续代码。为了做到这一点，= 后面应该跟一个包含默认键名称的参数。如果“原始”参数形式不正确以被解释为一组键值对，那么此默认键名称将用作键名，形成键-值对。

以\caption 为例，演示实现如下：

```
\DeclareDocumentCommand
  \caption
  {s ={short-text} +O{#3} +m}
  {%
    \showtokens{Grabbed arguments:^^J(#2)^^Jand^^J(#3)}%
  }
```

默认键名称为 short-text。当使用命令 \caption 时，如果可选参数是自由文本，例如

```
\caption[Some short text]{A much longer and more detailed text for
  demonstration purposes}
```

则输出将为

```
Grabbed arguments:
(short-text={Some short text})
and
(A much longer and more detailed text for demonstration purposes)
```

另一方面，如果使用键值形式的标题

```
\caption[label = cap:demo]%
  {A much longer and more detailed text for demonstration purposes}
```

则会被正确处理为

```
Grabbed arguments:
(label = cap:demo)
and
(A much longer and more detailed text for demonstration purposes)
```

是否解释为键值形式取决于参数中是否存在 `=` 字符。在行内数学模式中（在 `$...$` 或 `\(...\)` 中），这些字符将被忽略。可以通过在开头包含一个空条目来强制将参数读取为键值对

```
\caption[=,This is now a keyval]%
% ...
\caption[This is not $=$ keyval]%
```

这个空条目不会传递给底层代码，因此不会导致不允许空键名的键值解析器出现问题。任何文本模式的 `=` 符号都需要加上大括号以避免被错误解释：最方便的做法可能是将整个参数放入大括号中

```
\caption[{Not = to a keyval!}]%
```

这将被正确传递为

```
Grabbed arguments:
(short-text = {Not = to a keyval!})
```

## 2.10 参数处理器

参数处理器在底层系统抓取参数后但在传递给 ⟨*code*⟩ 前应用于参数。因此，参数处理器可用于在早期阶段规范化输入，使内部函数完全独立于输入形式。处理器应用于用户输入和可选参数的默认值，但不应用于特殊的 `-NoValue-` 标记。

每个参数处理器由语法 `>{`⟨*processor*⟩`}` 在参数规范中指定。处理器从右向左应用，因此

```
>{\ProcessorB} >{\ProcessorA} m
```

将 \ProcessorA 后跟 \ProcessorB 应用于由 m 参数抓取的标记。

\SplitArgument {⟨*number*⟩} {⟨*token(s)*⟩}

此处理器在给定的参数中每次出现 ⟨*tokens*⟩ 时拆分参数，最多拆分为 ⟨*number*⟩ 个标记（因此将输入分成 ⟨*number*⟩ + 1 部分）。如果输入中出现了太多的 ⟨*tokens*⟩，则会报错。处理后的输入将放置在 ⟨*number*⟩ + 1 组大括号中供进一步使用。如果参数中的 {⟨*tokens*⟩} 少于 {⟨*number*⟩}，则在处理后的参数末尾添加 -NoValue- 标记。

```
\NewDocumentCommand \foo {>{\SplitArgument{2}{;}} m}
  {\InternalFunctionOfThreeArguments#1}
```

如果拆分仅使用了单个字符 ⟨*token*⟩，则在进行拆分之前，任何与 ⟨*token*⟩ 匹配的类别码 13 （活动）字符将被替换。每个项解析时两端的空格都被修剪。

E 参数类型有点特殊，因为在命令声明中只有一个 E 时，您可能会得到多个参数（每个修饰符标记一个形式参数）。因此，当参数处理器应用于 e/E-类型的参数时，所有参数在被传递给 ⟨*code*⟩ 前都通过该处理器。例如，这个命令

```
\NewDocumentCommand \foo { >{\TrimSpaces} e{_^} }
  { [#1](#2) }
```

将 \TrimSpaces 应用于两个参数。

\SplitList {⟨*token(s)*⟩}

这个处理器将给定的参数在每次出现 ⟨*token(s)*⟩ 处分割，项目数量不固定。然后将每个项目包裹在 #1 中的大括号中。处理后的参数可使用映射函数进一步处理（见下文）。

```
\NewDocumentCommand \foo {>{\SplitList{;}} m}
  {\MappingFunction#1}
```

如果拆分仅使用了单个字符 ⟨*token*⟩，它将考虑到 ⟨*token*⟩ 可能已被设为活动状态（类别码为 13），并在这些标记处进行拆分。每个解析的项目两端的空格都被修剪。如果一个完整项目被大括号包围，则将剥去一组大括号，即以下输入和输出结果（每个单独项目都是一个大括号组）。

```
a       ==> {a}
{a}     ==> {a}
{a}b    ==> {{a}b}
a,b     ==> {a}{b}
{a},b   ==> {a}{b}
a,{b}   ==> {a}{b}
a,{b}c  ==> {a}{{b}c}
```

### \ProcessList {⟨*list*⟩} {⟨*cmd*⟩}

为了支持 \SplitList，函数 \ProcessList 可应用于 ⟨*list*⟩ 中的每个条目，对每个条目执行一次 ⟨*cmd*⟩。⟨*cmd*⟩ 应吸收一个参数：列表条目。例如

```
\NewDocumentCommand \foo {>{\SplitList{;}} m}
  {\ProcessList{#1}{\SomeDocumentCommand}}
```

### \ReverseBoolean

该处理器颠倒了 \BooleanTrue 和 \BooleanFalse 的逻辑，因此前面的示例将变为

```
\NewDocumentCommand\foo{>{\ReverseBoolean} s m}
  {%
    \IfBooleanTF#1%
      {\DoSomethingWithoutStar{#2}}%
      {\DoSomethingWithStar{#2}}%
  }
```

### \TrimSpaces

删除参数两端的任何前导和尾随空格（字符编码为 32、类别码为 10 的标记）。例如，声明一个函数

```
\NewDocumentCommand\foo {>{\TrimSpaces} m}
  {\showtokens{#1}}
```

并在文档中使用它如下

```
\foo{␣hello␣world␣}
```

将在终端显示 'hello␣world'，两端的空格已被移除。在输入中，\TrimSpaces 将删除多个空格，这些空格被包含以使得标准 TEX 的多个空格转换为单个空格不适用的情况下。

## 2.11 环境的主体

尽管环境 \begin{⟨*environment*⟩} … \end{⟨*environment*⟩} 通常用于实现 ⟨*environment*⟩ 的代码不需要访问环境的内容（它的“主体”），但有时将主体作为标准参数是很有用的。

通过以 b 结尾的参数规范来实现这一点，b 是专门用于这种情况的参数类型。例如

```
\NewDocumentEnvironment{twice} {O{\ttfamily} +b}
  {#2#1#2} {}
\begin{twice}[\itshape]
  Hello world!
\end{twice}
```

排版为 'Hello world!*Hello world!*'。

前缀 + 用于允许环境主体中包含多个段落。参数处理器也可以应用于 b 参数。默认情况下，在主体两端修剪空格：在示例中，否则会有来自 [\itshape] 和 world! 后行末的空格。在 b 前放置前缀 ! 可以取消空格修剪。

当参数规范中使用 b 时,环境声明的最后一个参数(例如,\NewDocumentEnvironment 中的一个 ⟨*end code*⟩，要插入到 \end{⟨*environment*⟩} 处）是多余的，因为可以简单地将该代码放在 ⟨*start code*⟩ 的末尾。尽管如此，这个（空的）⟨*end code*⟩ 必须提供。

使用此功能的环境可以被嵌套。

## 2.12 完全展开的文档命令

使用 \NewDocumentCommand 等创建的文档命令通常被创建为不会意外展开。这是通过引擎特性实现的，因此比 LATEX 2ε 的 \protect 机制更强大。只有在非常罕见的情况下，才可能有必要使用仅展开的抓取器创建函数。这对函数接受的参数类型以及其实现的代码施加了许多限制。此功能只应在必要时使用。

```
\NewExpandableDocumentCommand {⟨cmd⟩} {⟨arg spec⟩} {⟨code⟩}
\RenewExpandableDocumentCommand {⟨cmd⟩} {⟨arg spec⟩} {⟨code⟩}
\ProvideExpandableDocumentCommand {⟨cmd⟩} {⟨arg spec⟩} {⟨code⟩}
\DeclareExpandableDocumentCommand {⟨cmd⟩} {⟨arg spec⟩} {⟨code⟩}
```

此命令族用于创建一个文档级别的 ⟨*cmd*⟩，该命令将以完全展开的方式抓取其参数。函数的参数规范由 ⟨*arg spec*⟩ 给出，而 ⟨*cmd*⟩ 将执行 ⟨*code*⟩。通常，⟨*code*⟩ 也应该是完全展开的，尽管有可能不是这种情况（例如，用于表格的函数可能展开到第一个非展开且非空格的标记为 `\omit`）。

纯展开方式解析参数施加了许多限制，既有关于可以读取的参数类型，也有关于错误检查的限制：

- 最后一个参数（如果有的话）必须是强制类型 `m`、`r` 或 `R` 中的一个。
- “verbatim”参数类型 `v` 不可用。
- 参数处理器（使用 `>`）不可用。
- 不可能区分，例如，`\foo[` 和 `\foo{[}`：在这两种情况下，`[` 都会被解释为可选参数的开始。因此，可选参数的检查比标准版本不够健壮。

## 2.13 表格单元起始处的命令

在表格单元起始处使用命令会对底层实现施加一些限制。标准的 LaTeX 表格环境（`tabular` 等）使用一种机制，要求任何包装 `\multicolumn` 或类似命令的命令必须是“可展开”的。但是使用 `\NewDocumentCommand` 等创建的命令不符合这种情况，因为正如在第 2.12 节中详细说明的那样，它们使用了一个阻止这种“展开”的引擎特性。因此，要创建在表格单元起始处使用的这种包装器，你必须使用 `\NewExpandableDocumentCommand`，例如

```
\NewExpandableDocumentCommand\MyMultiCol{m}{\multicolumn{3}{c}{#1}}
\begin{tabular}{lcr}
a & b & c \\
\MyMultiCol{stuff} \\
\end{tabular}
```

## 2.14 关于参数定界符的细节

在普通（非可展开）命令中，定界类型通过向前查找初始定界符（使用 expl3 的 \peek_... 函数）来查找定界符标记。标记必须具有与定义为定界符的标记相同的含义和“形状”。定界符有三种可能的情况：字符标记、控制序列标记和活动字符标记。在此描述的所有实际情况中，活动字符标记的行为将完全与控制序列标记相同。

### 2.14.1 字符标记

字符标记的特征是其字符编码，其含义是其类别码（\catcode）。当定义命令时，字符标记的含义被固定到命令的定义中，不可更改。如果在定义时开放定界符具有与定义时相同的字符和类别码，则命令将正确看到参数定界符。例如：

```
\NewDocumentCommand { \foobar } { D<>{default} } {(#1)}
\foobar <hello> \par
\char_set_catcode_letter:N <
\foobar <hello>
```

输出将是：

```
(hello)
(default)<hello>
```

因为在两次调用 \foobar 之间，开放定界符 < 的含义发生了变化，所以第二次调用不会将 < 视为有效的定界符。命令假设如果找到了有效的开放定界符，那么匹配的闭合定界符也会存在。如果不是这样（要么省略了，要么含义发生了变化），就会引发低级 TEX 错误，并中止命令调用。

### 2.14.2 控制序列标记

控制序列（或控制字符）标记的特征是其名称，其含义是其定义。一个标记不能同时具有两种不同的含义。当将控制序列定义为命令的定界符时，在文档中找到控制序列名称时，无论其当前定义如何，都会将其检测为定界符。例如：

```
\cs_set:Npn \x { abc }
\NewDocumentCommand { \foobar } { D\x\y{default} } {(#1)}
\foobar \x hello\y \par
\cs_set:Npn \x { def }
\foobar \x hello\y
```

输出将是：

```
(hello)
(hello)
```

两次命令调用都将看到定界符 \x。

### 2.15 创建新的参数处理器

\ProcessedArgument

参数处理器允许在传递到底层代码之前操纵抓取的参数。新的处理器实现可以作为函数创建，它们接受一个尾随参数，并将结果留在 \ProcessedArgument 变量中。例如，\ReverseBoolean 被定义为

```
\ExplSyntaxOn
\cs_new_protected:Npn \ReverseBoolean #1
  {
    \bool_if:NTF #1
      { \tl_set:Nn \ProcessedArgument { \c_false_bool } }
      { \tl_set:Nn \ProcessedArgument { \c_true_bool } }
  }
\ExplSyntaxOff
```

[顺便说一句：代码是用 expl3 编写的，所以我们不必担心空格渗入定义。]

## 3 复制和显示（健壮的）命令和环境

如果你想（稍微）修改现有命令，你可能希望将当前定义保存为新名称，然后在新定义中使用它。如果现有命令是健壮的，那么使用低级的 \let 技巧将无法实现复制，因为它只复制了顶层定义，而未复制实际执行工作的部分。由于现在大多数的 LaTeX 命令都是健壮的，LaTeX 现在提供了一些高级声明来完成这个任务。

然而，请注意，通常最好利用可用的钩子（例如，通用命令或环境钩子），而不是复制当前定义，从而使其固定化；有关详细信息，请参阅钩子管理文档 lthooks-doc.pdf。

`\NewCommandCopy {⟨cmd⟩} {⟨existing-cmd⟩}`
`\RenewCommandCopy {⟨cmd⟩} {⟨existing-cmd⟩}`
`\DeclareCommandCopy {⟨cmd⟩} {⟨existing-cmd⟩}`

这将现有命令 ⟨*existing-cmd*⟩ 的定义复制到 ⟨*cmd*⟩ 中。在此之后，⟨*existing-cmd*⟩ 可以重新定义，而 ⟨*cmd*⟩ 仍然有效！这使你可以为 ⟨*existing-cmd*⟩ 提供一个新的定义，该定义可以使用 ⟨*cmd*⟩（即其旧的定义）。例如，在执行了以下代码之后：

```
\NewCommandCopy\LaTeXorig\LaTeX
\RenewDocumentCommand\LaTeX{}{\textcolor{blue}{\LaTeXorig}}
```

所有使用 `\LaTeX` 生成的 LaTeX 标志将以蓝色显示（假设你已加载了颜色包）。

`\New...` 和 `\Renew...` 之间的区别与其他地方一样：即，根据 ⟨*cmd*⟩ 是否已存在来决定是否产生错误，或者在 `\Declare...` 的情况下，无论如何都会复制。请注意，没有 `\Provide...` 声明，因为这将具有有限的价值。

如果 ⟨*cmd*⟩ 或 ⟨*existing-cmd*⟩ 无法作为单个标记提供而需要“构造”，你可以使用 `\ExpandArgs`，如第 4 节中所解释的那样。

`\ShowCommand {⟨cmd⟩}`

这会在终端上显示 ⟨*cmd*⟩ 的含义，然后停止（就像原始命令 `\show`）。不同之处在于，它会正确显示更复杂命令的含义，例如，对于健壮的命令，它不仅显示顶层定义，还显示实际的 payload 代码；对于使用 `\NewDocumentCommand` 等定义的命令，它还会提供有关参数签名的详细信息。

`\NewEnvironmentCopy {⟨env⟩} {⟨existing-env⟩}`
`\RenewEnvironmentCopy {⟨env⟩} {⟨existing-env⟩}`
`\DeclareEnvironmentCopy {⟨env⟩} {⟨existing-env⟩}`

这将环境 ⟨*existing-env*⟩ 的定义（开始和结束代码）复制到 ⟨*env*⟩ 中，即对定义环境的内部命令应用了两次 `\NewCommandCopy`，即 `\`⟨*env*⟩ 和 `\end`⟨*env*⟩。`\New...`、`\Renew...` 和 `\Declare...` 之间的区别与往常一样。

`\ShowEnvironment {⟨env⟩}`

这会显示环境 ⟨*env*⟩ 的开始和结束代码的含义。

# 4 预构造命令名称（或展开参数）

在使用 \NewDocumentCommand、\NewCommandCopy 或类似命令声明新命令时，有时需要“构造”csname。L3 编程层面通常有 \exp_args:N... 作为一种通用机制，但如果没有激活 \ExplSyntaxOn，则没有这样的机制（而且混合编程和用户界面级别的命令也不是一个好的方法）。因此，我们提供了一种使用 CamelCase 命名访问此功能的机制。

\UseName {⟨*string*⟩}
\ExpandArgs {⟨*spec*⟩} {⟨*cmd*⟩} {⟨*arg1*⟩} ...

\UseName 直接将 ⟨*string*⟩ 转换为 csname，然后执行它：这相当于长期存在的 LaTeX $2_\varepsilon$ 内部命令 \@nameuse 或 L3 编程层面的 \use:c。\ExpandArgs 接受一个 ⟨*spec*⟩，描述如何展开 ⟨*arguments*⟩，执行这些操作，然后执行 ⟨*cmd*⟩。⟨*spec*⟩ 使用 L3 编程层面提供的描述，相关的 \exp_args:N... 函数必须存在。常见情况下，⟨*spec*⟩ 可以是 c、cc 或 Nc：见下文。

例如，以下声明提供了一种生成校对命令的方法：

```
\NewDocumentCommand\newcopyedit{mO{red}}
  {%
    \newcounter{todo#1}%
    \ExpandArgs{c}\NewDocumentCommand{#1}{s m}%
      {%
        \stepcounter{todo#1}%
        \IfBooleanTF {##1}%
          {\todo[color=#2!10]{\UseName{thetodo#1}: ##2}}%
          {\todo[inline,color=#2!10]{\UseName{thetodo#1}: ##2}}%
      }%
  }
```

有了这个声明，你就可以写 \newcopyedit{note}[blue]，定义了一个名为 \note 的命令和相应的计数器。

第二个例子是通过字符串名称复制命令使用 \NewCommandCopy：在这里，我们可能需要构造两个命令名称。

```
\NewDocumentCommand\savebyname{m}
  {\ExpandArgs{cc}\NewCommandCopy{saved#1}{#1}}
```

在 ⟨*spec*⟩ 中，每个 `c` 代表一个被转换为 ‘c’ommand 的参数。一个 `n` 表示不改变的 ‘n’ormal 参数，`N` 表示不变的 ‘N’ormal 参数，但是只包含一个单个标记（通常不带大括号）。因此，要构造命令名称，只需使用 `\NewCommandCopy` 的第二个参数：

```
\ExpandArgs{Nc}\NewCommandCopy\mysectionctr{c@section}
```

在 L3 编程层面还支持其他几个单个字母，可以在 ⟨*spec*⟩ 中使用以其他方式操作参数。如果感兴趣，可以查看 `interface3.pdf` 中 L3 编程层面文档的“Argument expansion”部分。

# 5 可扩展的浮点数（及其他）计算

LaTeX3 编程层是格式的一部分，提供了丰富的接口来操作浮点变量和值。为了让（更简单的）应用能够在文档级别或者在否则不使用 L3 编程层的包中使用这些功能，我们提供了一些接口命令。

`\fpeval {`⟨*floating point expression*⟩`}`

可扩展命令 `\fpeval` 接受一个浮点表达式作为参数，并根据数学的常规规则生成结果。由于这个命令是可扩展的，它可以用在 TeX 需要数字的地方，例如在低级别的 `\edef` 操作中，给出纯粹的数值结果。

简而言之，浮点表达式可能包括：

- 基本算术运算：加法 $x$ `+` $y$，减法 $x$ `-` $y$，乘法 $x$ `*` $y$，除法 $x$ `/` $y$，平方根 $\sqrt{x}$，以及括号。
- 比较运算符：$x$ `<` $y$，$x$ `<=` $y$，$x$ `>?` $y$，$x$ `!=` $y$ 等等。
- 布尔逻辑：符号 `sign` $x$，否定 `!` $x$，合取 $x$ `&&` $y$，析取 $x$ `||` $y$，三元运算符 $x$ `?` $y$ `:` $z$。
- 指数运算：`exp` $x$，`ln` $x$，$x$`^`$y$。
- 整数阶乘：`fact` $x$。
- 三角函数：`sin` $x$，`cos` $x$，`tan` $x$，`cot` $x$，`sec` $x$，`csc` $x$ 期望其参数为弧度，并且 `sind` $x$，`cosd` $x$，`tand` $x$，`cotd` $x$，`secd` $x$，`cscd` $x$ 期望其参数为角度。

- 反三角函数：`asin` $x$，`acos` $x$，`atan` $x$，`acot` $x$，`asec` $x$，`acsc` $x$ 返回弧度单位的结果，并且 `asind` $x$，`acosd` $x$，`atand` $x$，`acotd` $x$，`asecd` $x$，`acscd` $x$ 返回角度单位的结果。
- 极值：`max`$(x_1, x_2, \ldots)$，`min`$(x_1, x_2, \ldots)$，`abs`$(x)$。
- 四舍五入函数，由两个可选值控制，$n$（小数位数，默认为 0）和 $t$（在遇到“平局”时的行为，默认为 `NaN`）：
  - `trunc`$(x, n)$ 向零舍入，
  - `floor`$(x, n)$ 向 $-\infty$ 舍入，
  - `ceil`$(x, n)$ 向 $+\infty$ 舍入，
  - `round`$(x, n, t)$ 四舍五入到最接近的值，如果 $t = 0$，则平局舍入为偶数值，如果 $t > 0$，则向 $+\infty$ 舍入，如果 $t < 0$，则向 $-\infty$ 舍入。
- 随机数：`rand`()，`randint`$(m, n)$。
- 常数：`pi`，`deg`（一度的弧度值）。
- 尺寸，自动转换为点数表示，例如，`pc` 为 12。
- 整数、尺寸和间距变量的自动转换（无需 `\number`）为浮点数，用点数表示尺寸，并忽略间距的拉伸和收缩部分。
- 元组：$(x_1, \ldots, x_n)$，可以相加、乘以或除以浮点数，并且可以嵌套。

一个使用的示例可能如下所示：

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
  = \fpeval{sin(3.5)/2 + 2e-3} $.
```

它产生以下输出：

LaTeX can now compute: $\frac{\sin(3.5)}{2} + 2 \cdot 10^{-3} = -0.1733916138448099$.

`\inteval {`⟨*integer expression*⟩`}`

# 6 可扩展的整数（和其他）计算

可扩展命令 `\inteval` 接受一个整数表达式作为参数，并使用数学的常规规则生成结果，但有一些限制，见下文。它识别的操作符有 `+`、`-`、`*` 和 `/`，还有括号。由于这个命令是可扩展的，它可以用在 TEX 需要数字的地方，例如在低级别的 `\edef` 操作中，给出纯粹的数值结果。

这基本上是对原始命令 `\numexpr` 的一个薄包装，因此有一些语法限制。这些限制包括：

- `/` 表示最接近整数的四舍五入除法，遇到平局时远离零方向舍入；
- 当任何中间结果的绝对值超过 $2^{31}-1$ 时（除了缩放操作 $a*b/c$，其中 $a*b$ 可能会任意大），会产生错误，整个表达式的结果将为零；
- 括号不能出现在一元 `+` 或 `-` 的后面，即在表达式的开头或者在 `+`、`-`、`*`、`/` 或 `(` 的后面放置 `+(` 或 `-(` 会导致错误。

一个使用的示例可能如下所示。

```
\LaTeX{} can now compute: The sum of the numbers is $\inteval{1 + 2 + 3}$.
```

产生的结果为 “LATEX 现在可以计算：这些数的和为 6。”

`\dimeval {`⟨*dimen expression*⟩`}` `\skipeval {`⟨*skip expression*⟩`}`

类似于 `\inteval`，但计算长度（`dimen`）或者伸缩长度（`skip`）的值。两者都是对应引擎原始命令的薄包装，这使它们速度很快，但因此显示了上述讨论的相同语法特点。不过，在实践中它们通常是足够的。例如

```
\newcommand\calculateheight[1]{%
  \setlength\textheight{\dimeval{\topskip+\baselineskip*\inteval{#1-1}}}}
```

如果一个页面应该容纳特定数量的文本行，则将 `\textheight` 设置为相应值。因此，在 `\calculateheight{40}` 之后，它被设置为 591.25806pt，给出当前文档中 `\topskip` (10.0pt) 和 `\baselineskip` (14.90405pt) 的值。

# 7 大小写转换

TEX 提供了两个原始命令 \uppercase 和 \lowercase 用于改变文本的大小写。然而，它们有许多限制：它们只改变显式字符的大小写，不考虑周围的上下文，不支持使用 8 位引擎的 UTF-8 输入等。为了解决这个问题，LATEX 提供了命令 \MakeUppercase、\MakeLowercase 和 \MakeTitlecase：这些命令在 TEX 原始命令的基础上有了显著的增强。这些命令是引擎稳定的（\protected），因此可以在移动参数中使用。

大小写转换在一般对话中是众所周知的。标题大写遵循 Unicode 联盟的定义：输入的第一个字符将转换为大写，其余字符转换为小写。可以支持 Unicode UTF-8 输入的全范围。

```
\MakeUppercase{hello WORLD ßüé}   HELLO WORLD SSÜÉ
\MakeLowercase{hello WORLD ßüé}   hello world ßüé
\MakeTitlecase{hello WORLD ßüé}   Hello world ßüé
```

大小写转换命令接受一个可选参数，用于定制输出。此可选参数接受键 locale，也可以使用别名 lang，可以使用 BCP-47 格式给出语言标识符。然后，它被应用于在大小写转换过程中选择语言特定功能。

在对这些命令给定的输入进行大小写转换之前，该输入将被“展开”。这意味着输入中的任何命令，如果转换为纯文本，都将被大小写转换。数学内容会自动排除在外，以及命令 \label、\ref、\cite、\begin 和 \end 的参数。可以使用命令 \AddToNoCaseChangeList 添加额外的排除项。还可以使用命令 \NoCaseChange 将输入排除在大小写转换之外。

```
\MakeUppercase{Some text $y = mx + c$}   SOME TEXT y = mx + c
 \MakeUppercase{\NoCaseChange{iPhone}}   iPhone
```

为了允许在大小写转换中使用稳健的命令并产生预期的输出，还提供了两个额外的控制命令。\CaseSwitch 允许用户指定四种可能情况的结果：

- 不进行大小写转换
- 转为大写
- 转为小写
- 标题大小写（仅适用于输入的开头）

命令 \DeclareCaseChangeEquivalent 提供了一种在大小写转换环境中替换命令为另一个版本的方法。有三个命令用于自定义代码点的大小写转换：

```
\DeclareLowercaseMapping [⟨locale⟩] {⟨codepoint⟩} {⟨output⟩}
\DeclareTitlecaseMapping [⟨locale⟩] {⟨codepoint⟩} {⟨output⟩}
\DeclareUppercaseMapping [⟨locale⟩] {⟨codepoint⟩} {⟨output⟩}
```

这三个命令接受一个 ⟨*codepoint*⟩（作为整数表达式），并会在相应的大小写转换操作下产生 ⟨*output*⟩。可选的 ⟨*locale*⟩ 参数用于指定映射仅适用于特定语言：它使用 BCP-47 格式给出（https://en.wikipedia.org/wiki/IETF_language_tag）。例如，内核在 8 位引擎中针对 U+01F0 (ǰ) 的大写映射进行了自定义：

```
\DeclareUppercaseMapping{"01F0}{\v{J}}
```

因为没有预组合的 J 字符，如果引擎不原生支持 Unicode，这会导致问题。类似地，要将一个 xx 区域设置为像土耳其语一样行为，并保留带点和无点 i 的区别，可以使用以下示例：

```
  \DeclareLowercaseMapping[xx]{"0049}{\i}
\DeclareLowercaseMapping[xx]{"0130}{i}
\DeclareUppercaseMapping[xx]{"0069}{\.{I}}
\DeclareUppercaseMapping[xx]{"0131}{I}
```