xparse 宏包：进行文档命令解析*

LaTeX3 项目组

发布于 2018-05-12

xparse 宏包为生成文档命令提供了高层次接口，旨在代替 LaTeX 2_ε 的 \newcommand 宏。xparse 将为函数准备的接口（例如可选项、星号、必选项等）与内部具体实现相分离。xparse 将区分函数的内部形式与文档水平的选项组织，并为前者提供了标准化输入。

目前，xparse 中可以认为是“稳定”的函数有：

- \NewDocumentCommand
- \RenewDocumentCommand
- \ProvideDocumentCommand
- \DeclareDocumentCommand
- \NewDocumentEnvironment
- \RenewDocumentEnvironment
- \ProvideDocumentEnvironment

* 本文档是 xparse 宏包文档的中译本。xparse 宏包提供了 LaTeX3 编程中定义文档命令和环境的方法。为方便起见修改了一些章节结构。

一些专有词汇翻译如下：

argument specification 选项规范

embellishment 装饰器

argument processors 选项处理器

本文档的地址为：https://github.com/WenboSheng/LaTeX3-doc-cn/xparse-doc-cn

盛文博
<wbsheng88@foxmail.com>

2018 年 6 月 3 日

- \DeclareDocumentEnvironment
- \NewExpandableDocumentCommand
- \RenewExpandableDocumentCommand
- \ProvideExpandableDocumentCommand
- \DeclareExpandableDocumentCommand
- \IfNoValue(TF)
- \IfValue(TF)
- \IfBoolean(TF)

其它函数目前尚处于“实验”性质阶段。请尝试使用这里列出的所有命令。不过要当心的是，实验性质的函数可能会改变或取消。

1 选项指定

在介绍用于创建文档命令的函数之前，首先展示在 xparse 中如何指定选项。为了允许每一选项都能独立定义，xparse 不仅需要知道函数选项的数量，还需要知道每一选项的属性。这可以通过构造*选项规范*实现。选项规范定义了选项数量，每一选项的类型，以及任何 xparse 需要的额外信息——xparse 通过这些信息才可以正确地读取用户输入并将其传递给内部函数。

选项规范的基本形式是一组字母，其中每一字母定义了一种选项类型。如下所述，一些类型还需要额外信息（比如默认值）。选项类型可以分成两种，一些定义了必选项（如果未提供则会报错），另一些定义了可选项。必选项类型有：

`m`（mandatory）标准的必选项，可以是单记号或者是带大括号的多个记号。不过不管实际输入是怎样，传递到内部代码的选项总是会带大括号。xparse 据此指定常规的 TEX 选项类型。

`r`（right）读取“必须”定界的选项，其中定界符由 ⟨*char1*⟩ 和 ⟨*char2*⟩ 给出，即：`r`⟨*char1*⟩⟨*char2*⟩。如果 ⟨字符 *character*⟩ 缺失，则会产生相应的错误并插入默认标记 `-NoValue-`。

`R`（Right）与 `r` 类似，这是一个“必须”定界的选项，不同之处是缺省标记 ⟨*default*⟩ 由用户定义，即：`R`⟨*char1*⟩⟨*char2*⟩`{`⟨*default*⟩`}`。

v（verbatim）读取“抄录”选项，抄录内容位于接下来的字符和再次遇到该字符之间，这与 LaTeX 2_ε 命令 `\verb` 的选项类似。因此，在两个匹配记号之间的内容会作为 `v` 类型的选项读取，但匹配记号不可以是这些字符：`%`、`\`、`#`、`{`、`}` 或 ␣。抄录选项也可以放在大括号 `{` 和 `}` 内。带有抄录选项的命令不可以位于另一个函数的选项内。

可选项的类型有：

`o`（optional）标准的 LaTeX 可选项，位于方括号内，如果未给出则代之以特殊标记 `-NoValue-`（稍后介绍）。

`d`（delimited）由 ⟨*char1*⟩ 和 ⟨*char2*⟩ 定界的可选项，即：`d`⟨*char1*⟩⟨*char2*⟩。与 `o` 类似，如果未给出则返回特殊标记 `-NoValue-`。

`O`（Optional）与 `o` 类似，但是当未给出时返回 ⟨*default*⟩。因此形式为：`O{`⟨*default*⟩`}`。

`D`（Delimited）与 `d` 类似，但是当未给出时返回 ⟨*default*⟩，即：`D`⟨*char1*⟩⟨*char2*⟩`{`⟨*default*⟩`}`。实际上，`o`, `d` 和 `O` 的内部实现其实都是特殊的 `D` 选项简称。

`s`（star）可选的星号，当星号存在时会返回 `\BooleanTrue`，否则返回 `\BooleanFalse`（稍后介绍）。

`t`（token）可选的 ⟨字符 *char*⟩，当该 ⟨字符 *char*⟩ 存在时会返回 `\BooleanTrue`，否则返回 `\BooleanFalse`，即形式为 `t`⟨*char*⟩。

`e`（embellishment） 一组可选的“装饰器”，其中每一个都需要对应的值，即：`e{`⟨*chars*⟩`}`。如果某个装饰器未给出则返回 `-NoValue-`。每一个装饰器给出一个选项，并且按照选项规范中 ⟨*chars*⟩ 列表进行排序。所有的 ⟨*chars*⟩ 必须互不相同。这是一个实验性质的选项类型。

`E`（Embellishment） 与 `e` 类似，但是当未给出值时返回若干个指定的默认值 ⟨*defaults*⟩，即：`E{`⟨*chars*⟩`}{`⟨*defaults*⟩`}`。详见第 6 节。

使用这些选项规范就可以很容易地创建各种复杂的用户输入语法。例如，选项定义为“`s o o m O{default}`”，那么实际输入“`*[Foo]{Bar}`”会按照如下方式解析：

- `#1` = `\BooleanTrue`
- `#2` = `Foo`
- `#3` = `-NoValue-`
- `#4` = `Bar`
- `#5` = `default`

而“`[One][Two]{}[Three]`”的解析方式为：

- `#1` = `\BooleanFalse`

- #2 = One
- #3 = Two
- #4 =
- #5 = Three

定界选项类型（d, o, r）在确定选项时要求匹配的成对定界符。例如：

```
\NewDocumentCommand{\foo}{o}{#1}
\foo[[content]] % #1 = "[content]"
\foo[[]         % 报错信息为 Error: missing closing "]"
```

另外要注意的是，{ 和 } 不能用作定界符，因为在 TEX 中这些是组记号。在这些记号内部的选项必须以 m 或 g 类型进行创建。

在定界选项中，如果有不匹配的记号或者其它奇怪的情况，可以用大括号将整个选项保护起来再导入，例如

```
\NewDocumentCommand{\foobar}{o}{#1}
\foobar[{[}]          % 由于“[”被“隐藏”起来，这样使用是可以的
```

如果大括号将整个可选项内容全部包含，那么在处理时会被剥离：

```
\NewDocumentCommand{\foobaz}{o}{#1}
\foobaz[{abc}]          % => "abc"
\foobaz[ {abc}]          % => " {abc}"
```

另外还有两个字符在选项规范中有特殊意义。首先是 +，用来创建 long 类型选项（可以接受跨段落的记号）。与 LATEX 2ε 的 \newcommand 不同，该项功能是逐选项运行的。例如在“s o o +m O{default}”中，必选项是 \long 而可选项则不是。

其次是字符 >，用来声明所谓的“选项处理器”，该功能可以在传递给宏定义之前修改选项内容。某种程度上来说，选项处理器的使用已经属于高级专题，（或者至少是使用较少的特性），将在第 9 节中讨论。

如果一个可选项之后跟着一个带有相同定界符的必选项，那么 xparse 会产生一个警告。这是因为用户此时不能省略可选项，实际上该可选项是必须的。该情景包括 o, d, O, D, s, t, e 和 E 类型之后跟着 r 或 R 类型选项，以及 g 或 G 之后跟着 m 类型选项。

由于 xparse 也用于描述已经在 LATEX 2ε 生态系统中广泛使用的一些接口，因此也定义了另外一些选项类型（在第 13 节介绍）：必选类型 l 和 u，以及可选的大括号组类型 g 和 G。但是不推荐使用这些类型，因为如果所有宏包都使用相同的语法的话，那么对于用户来说过于简略。出于同样的原因，定界选项 r, R, d 和 D 应当使用自然配对的标准定界符，比如 [和]、(和)，或者二者是相同的，比如 " 和 "。此外，一种非常常见的语法是将一个可选项 o 当作键值列表（比如使用 l3keys），之后再跟着一些必选项 m（或 +m）。

2 空格和可选项

TEX 会在函数名之后寻找第一个选项，无论中间是否有空格，对于必选项和可选项都是如此。比如 `\foo[arg]` 和 `\foo␣␣␣[arg]` 就是等价的。不仅如此，实际上直到最后一个必选项之前（肯定有最后一个可选项），空格都会忽略。例如，使用如下定义后

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

用户输入 `\foo{arg1}[arg2]{arg3}` 和 `\foo{arg1}␣␣[arg2]␣␣␣{arg3}` 效果是相同的。

在必选项之后的可选项行为则是可以人为选择的。标准设置会允许有空格，比如

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

此时 `\foobar{arg1}[arg2]` 和 `\foobar{arg1}␣[arg2]` 都会寻找可选项。可以通过在选项规范中给出 `!` 来改变这一行为：

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

此时 `\foobar{arg1}␣[arg2]` 就不会寻找可选项。

要专门提到一点。TEX 对“控制符号”的处理会导致细微之处的差异，也就是说命令名称是单个字符，比如“`\\`”。此时 TEX 不会忽略空格，需要在该命令后直接跟着所需的可选项。最常见的例子就是 amsmath 环境中对 `\\` 的使用。在 xparse 术语标记为

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

3 必须定界的选项

定界选项（`D` 类型）和“必须定界”选项（`R` 类型）的区别是，后者缺失时会报错。例如

```
\NewDocumentCommand {\foobaz} {r()m} {}
\foobaz{oops}
```

会导致生成错误信息。其中会插入标记 `-NoValue-`（`r` 类型）或者用户指定的缺省值（对于 `R` 而言）以帮助纠错。

用户需要注意，对于必须定界选项的支持还处于一定的实验性质阶段。因此十分欢迎在 LaTeX-L 邮件列表上进行反馈。

4 抄录选项

类型 v 的选项读取时会进入抄录模式，因此选项内容会进行转义。其中空格的类代码为 10（空格符），而其它记号的类代码为 12（其它符号）或者 13（活动符）。选项的定界方式可以类似于 LaTeX 2$_\varepsilon$ 的 \verb 函数，也可以使用嵌套正确的成对大括号。

包含抄录选项的函数不能作为其它函数的选项内。指示符 v 背后的代码会为此进行检查。因此，如果 TeX 已经用不可逆的方式将选项内容读取为记号，那么则会报错。

默认情况下 v 类型的选项必须在一行内。如果选项加了前缀 + 那么可以换行。

用户需要注意，对于必须定界选项的支持还处于一定的实验性质阶段。因此十分欢迎在 LaTeX-L 邮件列表上进行反馈。

5 选项的默认值

大写的选项类型（O, D, …）可以为选项缺失的情形指定缺省值；而相对应的小写类型则使用特殊标记 -NoValue-。此外，也可以通过 #1、#2 这样的方式将选项的缺省值设为其它选项的值。例如

```
\NewDocumentCommand {\conjugate} { m O{#1ed} O{#2} } {(#1,#2,#3)}
\conjugate {walk}            % => (walk,walked,walked)
\conjugate {find} [found]    % => (find,found,found)
\conjugate {do} [did] [done] % => (do,did,done)
```

缺省值可以涉及到选项规范中之后出现的选项。例如，某个命令可以接受两个可选项，并且默认情况下是相同的：

```
\NewDocumentCommand {\margins} { O{#3} m O{#1} m } {(#1,#2,#3,#4)}
\margins {a} {b}               % => {(-NoValue-,a,-NoValue-,b)}
\margins [1cm] {a} {b}         % => {(1cm,a,1cm,b)}
\margins {a} [1cm] {b}         % => {(1cm,a,1cm,b)}
\margins [1cm] {a} [2cm] {b}   % => {(1cm,a,2cm,b)}
```

用户需要注意，对于以其它选项作为缺省值这一功能的支持还处于一定的实验性质阶段。因此十分欢迎在 LaTeX-L 邮件列表上进行反馈。

6 “装饰符”的默认值

E 类型选项可以允许每一测试字符都有默认值。具体方法是对于序列中的每一条目都给出默认值。例如：

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

如果默认值列表的长度比测试字符列表短，那么将会返回特殊标记 -NoValue-（与 e 类型选项相同）。例如

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

对于测试字符 ^ 会返回 UP，而对于 _ 则返回默认的 -NoValue- 标记。使用这种方式可以同时进行缺失值测试和显式的默认值指定。

7 声明命令和环境

定义了选项规范的概念，现在就可以描述使用 xparse 创建函数和环境的具体方法了。

接口命令是在 LaTeX3 中创建文档层面函数的首选方法。所有用这种方式生成的函数都自然是鲁棒的（使用了 ε-TeX 的 \protected 机制）。

\NewDocumentCommand
\RenewDocumentCommand
\ProvideDocumentCommand
\DeclareDocumentCommand

\NewDocumentCommand ⟨函数 *Function*⟩ {⟨选项规范 *arg spec*⟩} {⟨代码 *code*⟩}

这一组函数用于创建文档水平的⟨函数 *function*⟩。该函数的选项规范由 ⟨*arg spec*⟩ 给出，然后展开为 ⟨*code*⟩。

以下是一个例子：

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

这段代码定义了 \chapter 命令，本质上处理机制与当前 LaTeX 2_ε 的同名命令是类似的（不同之处仅在于即便有 * 也可以有可选项）。命令 \typesetnormalchapter 会检测第一个选项是否为 -NoValue- 以判断是否有可选项。

\New…, \Renew…, \Provide…, \Declare… 这些版本之间的区别在于 ⟨函数 *function*⟩ 是否已经有定义。

- 如果 ⟨*function*⟩ 已经有定义，那么 \NewDocumentCommand 会报错。
- 如果 ⟨*function*⟩ 尚未定义，那么 \ReNewDocumentCommand 会报错。
- 只有当 ⟨*function*⟩ 尚未定义时，\ProvideDocumentCommand 才会为其创建新的定义。

- `\DeclareDocumentCommand` 总是会创建新定义，不管当前 ⟨*function*⟩ 是否存在。该命令使用时应当注意克制。

TEXhackers note: 与 LaTeX 2ε 的 `\newcommand` 系列不同的是，`\NewDocumentCommand` 系列不会阻止创建以 `\end…` 作为开头的函数。

`\NewDocumentEnvironment`
`\RenewDocumentEnvironment`
`\ProvideDocumentEnvironment`
`\DeclareDocumentEnvironment`

`\NewDocumentEnvironment {`⟨环境 *environment*⟩`} {`⟨选项规范 *arg spec*⟩`}`
`{`⟨开始代码 *start code*⟩`} {`⟨结束代码 *end code*⟩`}`

这些命令创建环境（`\begin{`⟨*environment*⟩`}` … `\end{`⟨*environment*⟩`}`），其工作原理与 `\NewDocumentCommand` 系列相同。其中，⟨*start code*⟩ 和 ⟨*end code*⟩ 都可以接受由 ⟨*arg spec*⟩ 定义的选项。这些选项在 `\begin{`⟨*environment*⟩`}` 之后给出。

8 特殊值的测试

由 xparse 创建的可选项可以使用专门设计的变量，进而返回关于接收选项的属性信息。

\IfNoValueT ⋆ \IfNoValueTF {⟨选项 *argument*⟩} {⟨真值代码 *true code*⟩} {⟨假值代码 *false code*⟩}
\IfNoValueF ⋆ \IfNoValueT {⟨选项 *argument*⟩} {⟨真值代码 *true code*⟩}
\IfNoValueTF ⋆ \IfNoValueF {⟨选项 *argument*⟩} {⟨假值代码 *false code*⟩}

\IfNoValue(TF) 测试可以用于检查 ⟨*argument*⟩（#1、#2 等）是否为特殊标记 -NoValue-。例如：

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

要注意的是，根据判断分支结果的需求，这里提供了三个测试：\IfNoValueTF, \IfNoValueT 和 \IfNoValueF。

由于 \IfNoValue(TF) 测试是可展开的，因此对于值的测试可以延时进行，比如在排版时或者在展开的文本中进行。

另外要着重指出的是，这里 -NoValue- 的构建方式保证其不会匹配简单的文本输入 -NoValue-，即

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

判断结果为 false。

当有两个连续的可选项时（我们不鼓励这种语法），用户可以将第一个选项设为空而只给出第二个选项。此时可以分别检查选项是否为空以及是否为 -NoValue-，但更好的方法是使用选项类型 O 并将缺省值设为空，并直接使用 expl3 的条件命令 \tl_if_blank:nTF 或者 etoolbox 中的版本 \ifblank 检查是否为空。

\IfValueT ⋆ \IfValueTF {⟨选项 *argument*⟩} {⟨真值代码 *true code*⟩} {⟨假值代码 *false code*⟩}
\IfValueF ⋆
\IfValueTF ⋆

与 \IfNoValue(TF) 结果相反的测试命令也是有的，即 \IfValue(TF)。实际语境会决定在代码中哪一种逻辑形式是最有意义的。

\BooleanFalse
\BooleanTrue

当搜索可选字符时（使用 s 和 t⟨*char*⟩）设置的 true 和 false 标志，这样该变量名可以在代码块之外获取。

\IfBooleanT ⋆
\IfBooleanF ⋆
\IfBooleanTF ⋆

\IfBooleanTF {⟨选项 *argument*⟩} {⟨真值代码 *true code*⟩} {⟨假值代码 *false code*⟩}

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

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

该命令会检查第一个选项是否为星号，并基于此选择接下来的行为。

9 选项处理器

xparse 引入了选项处理器的概念，可以在底层系统收集选项之后但是尚未传递给 ⟨*code*⟩ 时作用在选项上。因此，选项处理器可以用于在早期调整输入，这样内部函数就可以完全独立于输入形式。选项处理器作用于用户输入以及可选项的缺省值上，但不会作用于特殊标记 -NoValue-。

每一个选项处理器都通过选项规范中的语法 >{⟨*processor*⟩} 来指定，然后从右到左依次执行，因此

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

会先将 \ProcessorA 作用在 m 选项收集的记号上，然后再将 \ProcessorB 作用上去。

\ProcessedArgument

xparse 只定义了少量处理器函数。当然，可以预见代码作者会希望创建自己的选项处理器。这需要将收集上来的（或者由之前的处理器函数返回的）记号作为选项。因此，处理器函数应当返回被处理的选项，这就是 \ProcessedArgument 变量。

\ReverseBoolean

\ReverseBoolean

该处理器将反转 \BooleanTrue 和 \BooleanFalse 的逻辑值，因此之前的例子会变成：

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

\SplitArgument

\SplitArgument {⟨数量 *number*⟩} {⟨记号 *token*⟩}

Updated: 2012-02-12

该处理器会将选项按照 ⟨*token*⟩ 的出现位置进行分割，并且最多有 ⟨*number*⟩ 个记号（因此输入的选项分成了 ⟨*number*⟩ + 1 个部分）。如果选项输入中有过多数量的 ⟨*token*⟩ 那么会报错。处理后的输入会放在 ⟨*number*⟩ + 1 个大括号集合中供进一步使用。如果选项中 {⟨*token*⟩} 的数量小于 {⟨*number*⟩}，那么会在处理后的选项末尾加上合适数量的 -NoValue-。

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

在分割进行之前任何类代码为 13（活动符）的 ⟨*token*⟩ 都会被代替。每一项两端的空格都会被删去。

\SplitList

\SplitList {⟨记号 *token(s)*⟩}

该处理器会将选项按照 ⟨*token(s)*⟩ 进行分割，分割的项目数量不限。然后每一项都会包裹在 #1 中的大括号内。处理后的选项可以进一步使用映射函数进行处理。

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

如果只用一个 ⟨*token*⟩ 进行分割，那么在分割进行之前任何类代码为 13（活动符）的 ⟨*token*⟩ 都会被代替。

\ProcessList ⋆ `\ProcessList {⟨列表 list⟩} {⟨函数 function⟩}`

为了支持 `\SplitList`，这里提供了函数 `\ProcessList` 。它可以将一个函数 ⟨*function*⟩ 作用到 ⟨*list*⟩ 的每一条目上。函数 ⟨*function*⟩ 应当吸收列表条目作为选项。例如

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

该函数是实验性质的。

\TrimSpaces `\TrimSpaces`

删除任何选项两端（开头和结尾）的空格（类代码为 32 和 10 的记号）。例如，声明如下函数

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

并在文档中使用如下命令

```
\foo{ hello world }
```

此时会在终端显示 `hello world`，两端的空格都会删除。当输入的内容已经导入时，就不能应用标准 TEX 将多个空格转成单个空格的机制，此时 `\TrimSpaces` 会删除多重空格。

该函数是实验性质的。

10 完全展开的文档命令

在极少数情况下，使用完全展开的选项收集器创建函数会更有效。除了常规的鲁棒函数，xparse 也提供了对创建可展开函数的支持。这对函数接受的选项属性以及实现的代码提出了许多限制。因此只有当绝对需要时才应该使用该功能。如果你不清楚是否必要，那么不要使用这些函数！

\NewExpandableDocumentCommand
\RenewExpandableDocumentCommand
\ProvideExpandableDocumentCommand
\DeclareExpandableDocumentCommand

\NewExpandableDocumentCommand
    ⟨函数 *function*⟩ {⟨选项规范 *arg spec*⟩} {⟨代码 *code*⟩}

这组函数创建的文档层面的 ⟨*function*⟩ 会按照完全展开的方式收集选项。函数的选项规范由 ⟨*arg spec*⟩ 给出，然后会执行 ⟨*code*⟩。一般而言，⟨*code*⟩ 也是完全展开的，尽管例外情况也是可以的（例如，表格中使用的函数展开时，可以将 `\omit` 作为第一个不展开的非空格记号）。

解析选项时就进行展开有很多限制条件，比如读取的选项类型和可用的错误检查：

- 最后一个选项（如果有的话）必须是必选项且类型为 `m` 或 `r`。
- 所有的段内（short）选项必须出现在段间（long）选项之前。
- 必选项类型 `l` 和 `u` 不可以用在可选项之后。
- 可选项类型 `g` 和 `G` 不可用。
- “抄录”选项类型 `v` 不可用。
- 选项处理器（使用 `>`）不可用。
- 不能区分一些情况，比如 `\foo[` 和 `\foo{[}`：在这两种情况下 `[` 都被解释为可选项的开始。因此，对于可选项的检查没有标准情形时那么鲁棒。

如果给出的选项指示符不符合前六项要求，xparse 就会报错。最后一项是关于函数何时使用的，所以已经超出了 xparse 本身的范畴。

11 获取选项规范

文档命令和环境的选项规范可以用于检查使用。

\GetDocumentCommandArgSpec
\GetDocumentEnvironmentArgSpec

\GetDocumentCommandArgSpec ⟨函数 *function*⟩
\GetDocumentEnvironmentArgSpec ⟨环境 *environment*⟩

这两个函数获取 ⟨*function*⟩ 函数或 ⟨*environment*⟩ 环境的当前选项规范，并将其转换为记号列表变量 `\ArgumentSpecification`。如果 ⟨*function*⟩ 或 ⟨*environment*⟩ 有未知选项规范则会报错。`\ArgumentSpecification` 的赋值是在当前 TEX 组内局部的。

\ShowDocumentCommandArgSpec	\ShowDocumentCommandArgSpec ⟨函数 *function*⟩
\ShowDocumentEnvironmentArgSpec	\ShowDocumentEnvironmentArgSpec ⟨环境 *environment*⟩

这两个函数会在终端显示 ⟨*function*⟩ 或 ⟨*environment*⟩ 的当前选项规范。如果 ⟨*function*⟩ 或 ⟨*environment*⟩ 有未知选项规范则会报错。

12 载入时选项

log-declarations

本宏包载入时可以带一个键值型选项 log-declarations，取值为 true 或 false。默认情况下是 true，即声明的每一个命令或环境都会在日志中记录。如果使用

```
\usepackage[log-declarations=false]{xparse}
```

导入宏包，那么不会生成相关信息。

13 向后兼容性

xparse 的作用之一便是描述已经有的 LaTeX 接口，包括定界选项等其中相当不常用的一些情况（与 plainTeX 格式相反）。正因为如此，本宏包定义了一些目前来说应当尽可能避免使用的选项规范，因为在宏包中使用会导致不相容的用户接口。

不再推荐使用的选项类型是：

- l（left）该必选项会依次读取，直到第一个组开始记号：在标准 LaTeX 中是左大括号。
- u（until）读取必选项，“直到”遇到记号 ⟨*tokens*⟩。其中，要求的 ⟨*tokens*⟩ 会传递给指示符，即 u{⟨*tokens*⟩}。
- g（group）可选项，其内容位于 TeX 组记号对内部（在标准 LaTeX 中为 { … }），未给出时返回 -NoValue-。
- G（Group） 与 g 类似， 但是当未给出时返回指定的默认值 ⟨*default*⟩， 即：G{⟨*default*⟩}。