xparse 宏包
文档命令解析

The LaTeX Project*

由 张泓知 翻译

原文版本 2023-10-10

xparse 包提供了一个用于生成文档级命令的高级接口。因此，它旨在替代 LaTeX 2ε 的 \newcommand 宏。然而，xparse 的工作方式是，函数的接口（例如可选参数、星号和必选参数）与内部实现分离。xparse 为函数的内部形式提供了一个规范化的输入，独立于文档级别的参数排列。

目前，xparse 中被认为是“稳定的（stable）”有：

- \NewDocumentCommand
  \RenewDocumentCommand
  \ProvideDocumentCommand
  \DeclareDocumentCommand
- \NewDocumentEnvironment
  \RenewDocumentEnvironment
  \ProvideDocumentEnvironment
  \DeclareDocumentEnvironment
- \NewExpandableDocumentCommand
  \RenewExpandableDocumentCommand
  \ProvideExpandableDocumentCommand
  \DeclareExpandableDocumentCommand
- \IfNoValue(TF)
- \IfValue(TF)

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

- `\IfBoolean(TF)`

除了目前被视为“实验性”的其他功能。请尝试这里提供的所有命令，但请注意，实验性的功能可能会更改或消失。

1 指定参数

在介绍用于创建文档命令的功能之前，将展示使用 xparse 指定参数的方法。为了允许每个参数独立定义，xparse 不仅需要知道函数的参数数量，还需要知道每个参数的性质。这是通过构建一个*参数规范*来完成的，该规范定义了参数的数量、每个参数的类型和 xparse 读取用户输入并正确传递到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。正如下面将要描述的那样，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种：定义必须参数的类型（如果未找到可能会引发错误），以及定义可选参数的类型。必须类型包括：

`m` 标准的必选参数，可以是单个标记，也可以是由花括号 `{}` 包围的多个标记。无论输入是什么，参数都将被传递到内部代码，但不带有外部花括号。这是 xparse 中用于普通 TEX 参数的类型说明符。

`r` 给出为 `r`⟨*token1*⟩⟨*token2*⟩，表示一个“必需”的分隔参数，其分隔符为 ⟨*token1*⟩ 和 ⟨*token2*⟩。如果缺少开始分隔符 ⟨*t oken1*⟩，将在适当的错误后插入默认标记 `-NoValue-`。

`R` 给出为 `R`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`，这是一个“必需”的分隔参数，与 `r` 类似，但具有用户可定义的恢复值 ⟨*default*⟩ ，而不是 `-NoValue-`。

`v` 以“verbatim”方式读取参数，介于以下字符和其下一个出现之间，类似于 LaTeXe 命令 `\verb` 的参数。因此，`v` 类型的参数在两个相同的字符之间读取，但这些字符不能是 `%`、`\`、`#`、`{`、`}` 或 ␣。verbati m 参数也可以被括在花括号 `{` 和 `}` 中。拥有 verbatim 参数的命令在另一个函数的参数中出现时会产生错误。

`b` 只适用于环境的参数规范，表示环境的主体部分，位于 `\begin{`⟨ *environment*⟩`}` 和 `\end{`⟨*environment*⟩`}` 之间。详见第 1.6 节。

定义可选参数的类型有：

`o` 一个标准的 LATEX 可选参数，用方括号括起来，如果未给出参数，将提供特殊标记 `-NoValue-`（如后文所述）。

`d` 给定为 `d`⟨*token1*⟩⟨*token2*⟩，一个由 ⟨*token1*⟩ 和 ⟨*token2*⟩ 分隔的可选参数。与 `o` 类似，如果未给出值，将返回特殊标记 `-NoValue-`。

`O` 给定为 `O{`⟨*default*⟩`}`，与 `o` 类似，但如果未给出值，则返回 ⟨*default*⟩。

`D` 给定为 `D`⟨*token1*⟩⟨*token2*⟩`{`⟨*default*⟩`}`，与 `d` 类似，但如果未给出值，则返回 ⟨*default*⟩。在内部，`o`、`d` 和 `O` 类型是构造 `D` 类型参数的快捷方式。

`s` 一个可选的星号,如果存在星号则结果为 `\BooleanTrue`,否则为 `\BooleanFalse`（如后文所述）。

`t` 一个可选的 ⟨*token*⟩，如果 ⟨*token*⟩ 存在则结果为 `\BooleanTrue`，否则为 `\BooleanFalse`。给定为 `t`⟨*token*⟩。

`e` 给定为 `e{`⟨*tokens*⟩`}`，一组可选的修饰，每个修饰都需要一个值。如果修饰不存在，则返回 `-NoValue-`。每个修饰给出一个参数，顺序与参数规范中 ⟨*tokens*⟩ 列表的顺序相同。所有 ⟨*tokens*⟩ 必须是不同的。这是一个实验性质的类型。

`E` 与 `e` 相同,但如果没有给出值,则返回一个或多个 ⟨*defaults*⟩`:E{`⟨*tokens*⟩`}{`⟨*defaults*⟩`}`。详见第 1.5 节。

利用这些说明符，可以非常容易地创建复杂的输入语法。例如，给定参数定义 '`s␣o␣o␣m␣O{default}`'，输入 '`*[Foo]{Bar}`' 将被解析为：

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

而 '`[One][Two]{}[Three]`' 将被解析为：

- `#1` = `\BooleanFalse`
- `#2` = `One`
- `#3` = `Two`
- `#4` =
- `#5` = `Three`

分隔参数类型（`d`、`o` 和 `r`）被定义为在收集参数时需要匹配的分隔符对。例如：

```
\NewDocumentCommand{\foo}{o}{#1}
\foo[[content]] % #1 = "[content]"
\foo[[]         % 错误：缺少闭合的 "]"
```

还要注意，`{` 和 `}` 不能用作分隔符，因为它们被 TEX 用作分组标记。隐式的开始或结束组标记（比如 `\bgroup` 和 `\egroup`）不允许用于分隔参数类型。在这些标记内部抓取的参数必须创建为 `m` 或 `g` 类型参数。

在分隔参数内部，非平衡或其他尴尬的标记可以通过用花括号保护整个参数来包含：

```
\NewDocumentCommand{\foobar}{o}{#1}
\foobar[{[}]          % 允许，因为 "[" 被 ' 隐藏'
```

这些花括号只有在围绕可选参数的整个内容时才会被剥离：

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

当创建参数说明符时，另外两个字符具有特殊含义。首先，+ 用于将参数设为长参数（接受段落标记）。与 LaTeX 2_ε 的 \newcommand 不同的是，这是基于每个参数的设定。因此，将示例修改为 'sooo+mO{default}' 意味着强制参数现在是 \long，而可选参数则不是。

第二，字符 > 用于声明所谓的"参数处理器"，可用于修改参数内容，然后将其传递给宏定义。参数处理器的使用是一个稍微高级的主题（或者至少是一个不太常用的功能），在第 3.2 节进行了介绍。

当可选参数后跟着具有相同分隔符的强制参数时，xparse 会发出警告，因为用户无法省略可选参数，实际上变成了强制参数。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，还适用于 g 或 G 类型的参数，后面跟着 m 类型参数。

由于 xparse 还用于描述出现在更广泛的 LaTeX 2_ε 生态系统中的接口，它还定义了额外的参数类型，在第 1.8 节描述：强制类型 l 和 u，以及可选大括号组类型 g 和 G。不建议使用它们，因为如果所有包都使用类似的语法，对用户来说会更简单。出于同样的原因，分隔参数 r、R、d 和 D 通常应使用自然配对的分隔符，如 [和] 或 (和)，或者使用相同的分隔符，比如 " 和 "。一个非常常见的语法是，一个可选参数 o 被视为键值对列表（例如使用 l3keys），后面跟着一些强制参数 m（或 +m）。

1.1 间距和可选参数

TeX 将会找到函数名后的第一个参数，而不管中间有多少空格。这对于强制参数和可选参数都适用。因此，\foo[arg] 和 \foo␣␣␣[arg] 是等价的。当收集参数到最后一个必须收集的强制参数时（因为它必须存在），在收集参数期间，空格也会被忽略。

因此，在

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

用户输入 `\foo{arg1}[arg2]{arg3}` 和 `\foo{arg1}␣␣[arg2]␣␣␣{arg3}` 将以相同的方式解析。

可选参数在任何必选参数之后的行为是可选择的。标准设置允许在此处使用空格，因此在

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

`\foobar{arg1}[arg2]` 和 `\foobar{arg1}␣[arg2]` 都会找到一个可选参数。

这可以通过在参数规范中加入修改后的 `!` 来改变：

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

在这种情况下，`\foobar{arg1}␣[arg2]` 将不会找到一个可选参数。

这里有一个微妙之处，是由于 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，因此可以要求一个可选参数直接跟在这样的命令后面。最常见的例子是在 amsmath 环境中使用 `\\`。在 xparse 中，其签名为

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

1.2 必须分隔的参数

分隔参数（`D` 类型）和“必需分隔”参数（`R` 类型）之间的区别在于，如果缺少后者，将引发错误。因此，例如

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

将导致发出错误消息。插入标记 `-NoValue-`（`r` 类型）或用户指定的默认值（对于 `R` 类型）以允许错误恢复。

1.3 Verbatim 参数

类型为 `v` 的参数以 verbatim 模式读取，这将导致所抓取的参数由类别码为 12（“其它”）和 13（“活动”）的记号组成，除了空格外，空格被赋予类别码 10（“空格”）。参数的定界方式类似于 LATEX 2_ε 的 `\verb` 函数，或者由（正确嵌套的）大括号对定界。

包含 verbatim 参数的函数不能出现在其他函数的参数中。`v` 参数说明符包含代码来检查这一点，如果所抓取的参数已被 TEX 以不可逆的方式记号化，它将引发错误。

默认情况下，`v` 类型的参数最多只能有一行。在前面加上 `+` 可以允许参数内换行。

用户应注意，对于 verbatim 参数的支持在某种程度上是实验性的。因此，在 LaTeX-L 邮件列表上非常欢迎反馈意见。

1.4 参数的默认值

大写的参数类型（O，D，…）允许指定参数缺失时使用的默认值；其小写对应项使用特殊标记 -NoValue-。默认值可以用 #1，#2 等参数的值来表达。

```
\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)
```

默认值可以引用参数规范中后面出现的参数。例如，一个命令可以接受两个默认相等的可选参数：

```
\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)}
```

用户应注意，对于默认参数引用其他参数的支持在某种程度上是实验性的。因此，在 LaTeX-L 邮件列表上非常欢迎反馈意见。

1.5 对于“修饰符”的默认值

E 类型的参数允许每个测试记号有一个默认值。这是通过为列表中的每个条目提供一个默认值列表来实现的，例如：

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

如果默认值列表比测试记号列表短，则会返回特殊的 -NoValue- 标记（与 e 类型参数一样）。因此，例如

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

对于 ^ 测试字符具有默认值 UP，但对于 _ 将返回 -NoValue- 标记作为默认值。这允许混合显式默认值与检测缺失值。

1.6 环境的主体

虽然 `\begin{`⟨*environment*⟩`}` … `\end{`⟨*environment*⟩`}` 环境通常用于实现 ⟨*environment*⟩ 的代码不需要访问环境内容（其“主体”）的情况，但有时将主体作为标准参数会很有用。

在 xparse 中，通过在参数规范末尾加上 `b` 来实现这一点。在 xparse 中采取的方法与早期的包 environ 或 newenviron 不同：环境的主体作为一个普通的参数 `#1`、`#2` 等传递给代码部分，而不是存储在诸如 `\BODY` 这样的宏中。

例如

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

排版出 “Hello world!*Hello world!*”。

前缀 `+` 用于允许环境主体中有多个段落。也可以对 `b` 参数应用参数处理器。

默认情况下，在主体两端修剪空格：否则在 `[\itshape]` 和 `world!` 后面的行尾会有空格。在 `b` 前放置前缀 `!` 可以抑制修剪空格。

当参数规范中使用 `b` 时，`\NewDocumentEnvironment` 的最后一个参数，即在 `\end{`⟨*environment*⟩`}` 处插入的⟨*end code*⟩，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嵌套。

用户应注意，此功能在某种程度上是实验性的。因此，在 `LaTeX-L` 邮件列表上非常欢迎反馈意见。

1.7 带星号的环境

许多宏包用于定义带有和不带有 `*` 的环境，例如 `tabular` 和 `tabular*`。目前，xparse 并没有提供专门的工具来定义这些带星号的环境：应该简单地分别定义这两种环境，例如：

```
\NewDocumentEnvironment { tabular } { o +m } {...} {...}
\NewDocumentEnvironment { tabular* } { m o +m } {...} {...}
```

当然，在这个例子中标为 “`...`” 的这两种环境的实现可以依赖于相同的内部命令。

请注意，这种情况与 `s` 类型的参数不同：如果环境的签名以 `s` 开头，那么在 `\begin` 的参数之后会搜索星号。例如，以下示例会排版出 `star`。

```
\NewDocumentEnvironment { envstar } { s }
  {\IfBooleanTF {#1} {star} {no star}} {}
\begin{envstar}*
\end{envstar}
```

1.8 向前兼容性

xparse 的一个作用是描述现有的 LaTeX 接口，其中包括一些在 LaTeX 中相当不寻常（与 plain TeX 等格式相比）的接口，比如分界符参数。因此，该宏包定义了一些参数说明符，目前在编写宏包时应尽量避免使用，因为使用它们会导致用户接口不一致。最简单的语法通常是最好的，使用参数说明符如 `mmmm` 或 `ommmm`，即一个可选参数后跟一些标准的必选参数。可以使用 l3keys 提供的工具使可选参数支持键-值语法。

不再推荐使用的参数类型包括：

`l` 读取直到第一个开始组标记之前的所有内容的必选参数：在标准 LaTeX 中，这是左花括号。

`u` 读取必选参数“直到”遇到 ⟨*tokens*⟩，其中期望的 ⟨*tokens*⟩ 作为该说明符的参数给出：`u{`⟨*tokens*⟩`}`。

`g` 给定在一对 TeX 组标记内部的可选参数（在标准 LaTeX 中为 `{ …}`），如果不存在则返回 `-NoValue-`。

`G` 类似于 `g`，但如果没有给定值则返回 ⟨*default*⟩：`G{`⟨*default*⟩`}`。

1.9 关于参数分界符的细节

在普通（不可展开）命令中，分界符类型会通过向前查看（使用 expl3 的 `\peek_...` 函数）来寻找初始分界符。分界符标记必须具有与定义为分界符的标记相同的含义和“形状”。分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。在本描述的所有实际情况中，活动字符标记将与控制序列标记的行为完全相同。

1.9.1 字符标记

字符标记由其字符代码和其含义（类别码 `\catcode`）所特征化。当定义命令时，字符标记的含义将固定在命令的定义中，不可更改。如果在定义时打开分界符具有与定义时相同的字符和类别码，则命令将正确地将其视为参数分界符。例如，在以下代码中：

```
\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，打开分界符 `<` 的含义发生了变化，所以第二次调用不会将 `<` 视为有效分界符。命令假设如果找到了有效的打开分界符，就会有一个匹配的关闭分界符。如果没有（要么是被省略了，要么是发生了含义变化），就会引发低级 TEX 错误并中止命令调用。

1.9.2 控制序列标记

控制序列（或控制字符）标记由其名称所特征化，其含义是其定义。一个标记不能同时具有两个不同的含义。当控制序列被定义为命令中的分界符时，无论其当前定义如何，只要在文档中找到该控制序列名称，就会将其检测为分界符。例如，在以下代码中：

```
\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)
```

在两次调用中，都会看到 `\x` 作为分界符。

2 声明命令和环境

有了参数规范的概念，现在可以描述使用 xparse 创建函数和环境的方法。

接口构建命令是在 LATEX3 中创建文档级函数的首选方法。通过这种方式生成的所有函数都是天然健壮的 (robust)（使用 ε-TEX 的 `\protected` 机制）。

\NewDocumentCommand
\RenewDocumentCommand
\ProvideDocumentCommand
\DeclareDocumentCommand

\NewDocumentCommand ⟨*function*⟩ {⟨*arg spec*⟩} {⟨*code*⟩}

这组命令用于创建文档级别的 ⟨*function*⟩。函数的参数规范由 ⟨*arg spec*⟩ 给出，并且函数通过 xparse 找到的参数用 ⟨*code*⟩ 展开，其中 #1、#2 等被替换为参数。

例如：

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

这是定义一个 \chapter 命令的方式，它基本上的行为类似于当前的 LaTeX 2_ε 命令（除了它在解析 * 时也接受可选参数）。\typesetnormalchapter 可以测试它的第一个参数是否为 -NoValue-，以确定是否存在可选参数。

\New…、\Renew…、\Provide… 和 \Declare… 版本的区别在于 ⟨*function*⟩ 是否已经定义了行为。

- \NewDocumentCommand 如果 ⟨*function*⟩ 已经定义，会发出错误。
- \RenewDocumentCommand 如果 ⟨*function*⟩ 没有之前的定义，会发出错误。
- \ProvideDocumentCommand 只有在尚未给出 ⟨*function*⟩ 的定义时才创建新定义。
- \DeclareDocumentCommand 将始终创建新定义，无论同名的 ⟨*function*⟩ 是否已存在。应谨慎使用。

TEXhackers note: 与 LaTeX 2_ε 的 \newcommand 等不同，\NewDocumentCommand 函数系列不会阻止创建以 \end… 开头的函数。

\NewDocumentEnvironment
\RenewDocumentEnvironment
\ProvideDocumentEnvironment
\DeclareDocumentEnvironment

\NewDocumentEnvironment {⟨*environment*⟩} {⟨*arg spec*⟩}
{⟨*start code*⟩} {⟨*end code*⟩}

这些命令与 \NewDocumentCommand 等类似，但创建环境（\begin{⟨*environment*⟩} … \end{⟨*environment*⟩}）。⟨*start code*⟩ 和 ⟨*end code*⟩ 都可以访问由 ⟨*arg spec*⟩ 定义的参数。参数将在 \begin{⟨*environment*⟩} 之后给出。

3 其他 xparse 命令

3.1 测试特殊值

使用 xparse 创建的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。

\IfNoValueT ⋆ \IfNoValueTF {⟨*argument*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\IfNoValueF ⋆ \IfNoValueT {⟨*argument*⟩} {⟨*true code*⟩}
\IfNoValueTF ⋆ \IfNoValueF {⟨*argument*⟩} {⟨*false code*⟩}

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

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

如果给定了可选参数，将使用不同的内部函数，而不是当它不存在时。

请注意，有三个测试可用，取决于所需的结果分支：\IfNoValueTF、\IfNoValueT 和 \IfNoValueF。由于 \IfNoValue(TF) 测试是可展开的，因此可以稍后测试这些值，例如在排版或扩展上下文中。

需要注意的是 -NoValue- 被构造成不会与简单文本输入 -NoValue- 匹配，也就是说

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

在逻辑上是 false 的。当两个可选参数依次跟随（这是我们通常不鼓励的语法）时，允许命令的使用者仅提供第二个参数并提供空的第一个参数是有意义的。然而，与其分别测试空值和 -NoValue-，最好使用参数类型为 O 并使用空默认值，并仅使用 expl3 条件 \tl_if_blank:nTF 或其 etoolbox 类似物 \ifblank 测试是否为空。

\IfValueT ⋆ \IfValueTF {⟨*argument*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\IfValueF ⋆
\IfValueTF ⋆

\IfNoValue(TF) 的相反形式也作为 \IfValue(TF) 提供。根据特定代码场景，上下文将决定哪种逻辑形式更合理。

\BooleanFalse
\BooleanTrue

在搜索可选字符时（使用 s 或 t⟨*char*⟩），设置的 true 和 false 标志具有可以在代码块外部访问的名称。

\IfBooleanT ⋆
\IfBooleanF ⋆
\IfBooleanTF ⋆

\IfBooleanTF {⟨*argument*⟩} {⟨*true code*⟩} {⟨*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} }
  }
```

检查第一个参数是否为星号，然后根据这一信息选择要采取的操作。

3.2 Argument processors

xparse 引入了参数处理器的概念，它在底层系统抓取参数之后但在传递给 ⟨*code*⟩之前应用。因此，参数处理器可用于在早期规范输入，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 适用于特殊的 -NoValue- 标记。

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

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

将对 m 参数抓取的标记应用 \ProcessorA，然后是 \ProcessorB。

有时使用另一个参数的值作为处理器的一个参数可能会有用。例如，使用以下定义的 \SplitList 处理器，

```
\NewDocumentCommand \foo { O{,} >{\SplitList{#1}} m } { \foobar{#2} }
\foo{a,b;c,d}
```

结果是 \foobar 接收参数 {a}{b;c}{d}，因为 \SplitList 接收两个参数，可选参数（这里的默认值是逗号）和必选参数。总结一下，首先在输入中搜索参数，然后确定任何默认参数，就像第1.4节中解释的那样，然后将这些默认参数传递给任何处理器。在处理器中引用参数（通过 #1、#2等）时，使用的始终是任何处理器之前的参数。

\ProcessedArgument

xparse 定义了一组非常小的处理器函数。主要预计代码编写者将想要创建自己的处理器。这些处理器需要接受一个参数，即抓取的标记（或前一个处理器函数返回的标记）。处理器函数应将处理后的参数作为变量 \ProcessedArgument返回。

\ReverseBoolean \ReverseBoolean

此处理器颠倒了 \BooleanTrue 和 \BooleanFalse 的逻辑，因此之前的示例将变成

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

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

Updated: 2012-02-12

此处理器将给定的参数在每个 ⟨*tokens*⟩ 出现的地方分割，最多分割 ⟨*number*⟩ 个标记（因此将输入分成 ⟨*number*⟩ + 1 部分）。如果输入中存在过多的 ⟨*tokens*⟩，将报错。处理后的输入放置在 ⟨*number*⟩ + 1 组大括号中供进一步使用。如果参数中 {⟨*tokens*⟩} 的数量少于 {⟨*number*⟩}，则在处理后的参数末尾添加 -NoValue- 标记。

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

如果分割使用了单个字符 ⟨*token*⟩，则在分割之前将替换任何匹配 ⟨*token*⟩ 的类别码为 13（活动状态）的字符。每个解析的项目的两端都会去除空格。

\SplitList \SplitList {⟨*token(s)*⟩}

此处理器将给定的参数在每个 ⟨*token(s)*⟩ 出现的地方分割，其中项目数目不是固定的。然后在 #1 中将每个项目放置在大括号中。结果是，处理后的参数可以使用映射函数进一步处理。

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

如果分割使用了单个字符 ⟨*token*⟩，则在分割之前将替换任何匹配 ⟨*token*⟩ 的类别码为 13（活动状态）的字符。每个解析的项目的两端都会去除空格。

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

为了支持 `\SplitList`，函数 `\ProcessList` 可用于对 ⟨*list*⟩ 中的每个条目应用 ⟨*function*⟩。⟨*function*⟩ 应吸收一个参数：列表条目。例如

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

此函数为实验性质。

\TrimSpaces `\TrimSpaces`

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

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

并在文档中使用它，如

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

将在终端显示 `hello world`，两端的空格已被移除。`\TrimSpaces` 将删除多个空格，若它们被包含进来，以至于标准 TEX 将多个空格转换为单个空格的规则不适用。

此函数为实验性质。

3.3 可完全展开的文档命令

在极为罕见的情况下，可能有用的是使用可完全展开的参数抓取器来创建函数。为了支持这一点，xparse 可以创建可展开函数以及通常的健壮函数。这给函数所接受的参数的性质以及它所实现的代码施加了许多限制。这个功能只应在绝对必要时使用；如果你不了解何时可能需要使用它们，请不要使用这些函数！

\NewExpandableDocumentCommand
\RenewExpandableDocumentCommand
\ProvideExpandableDocumentCommand
\DeclareExpandableDocumentCommand

\NewExpandableDocumentCommand ⟨*function*⟩ {⟨*arg spec*⟩} {⟨*code*⟩}

这组命令用于创建一个文档级别的 ⟨*function*⟩，它将以可完全展开的方式抓取其参数。该函数的参数规范由 ⟨*arg spec*⟩ 给出，并且函数将执行 ⟨*code*⟩。通常，⟨*code*⟩ 也将是可完全展开的，尽管也可能不是这种情况（例如，用于表格的函数可能会展开，以便 `\omit` 是第一个非可展开的非空格标记）。

可展开地解析参数施加了许多限制，涉及到可以读取的参数类型以及可用的错误检查：

- 如果存在最后一个参数，则必须是 `m`、`r`、`R`、`l` 或 `u` 中的一个。
- 所有短参数都出现在长参数之前。
- 可选参数后面不能使用必需参数类型 `l` 和 `u`。
- 不可用可选参数类型 `g` 和 `G`。
- 不可用的 “verbatim” 参数类型 `v`。
- 不可用参数处理器（使用 `>`）。
- 不可能区分，例如 `\foo[` 和 `\foo{[}`：在这两种情况下，`[` 都将被解释为可选参数的开始。因此，对可选参数的检查不够健壮，比标准版本差。

如果给出的参数说明符不符合前六个要求，xparse 将会发出错误。最后一条是在函数使用时出现的问题，因此超出了 xparse 本身的范围。

3.4 获取参数规范

文档命令和环境的参数规范可供检查和使用。

\GetDocumentCommandArgSpec
\GetDocumentEnvironmentArgSpec

\GetDocumentCommandArgSpec ⟨*function*⟩
\GetDocumentEnvironmentArgSpec {⟨*environment*⟩}

这些函数将请求的 ⟨*function*⟩ 或 ⟨*environment*⟩ 的当前参数规范转移到记号列表变量 `\ArgumentSpecification` 中。如果 ⟨*function*⟩ 或 ⟨*environment*⟩ 没有已知的参数规范，则会发出错误。对 `\ArgumentSpecification` 的赋值局限于当前的 TEX 组。

\ShowDocumentCommandArgSpec	\ShowDocumentCommandArgSpec ⟨*function*⟩
\ShowDocumentEnvironmentArgSpec	\ShowDocumentEnvironmentArgSpec {⟨*environment*⟩}

这些函数在终端显示请求的 ⟨*function*⟩ 或 ⟨*environment*⟩ 的当前参数规范。如果 ⟨*function*⟩ 或 ⟨*environment*⟩ 没有已知的参数规范，则会发出错误。

4 加载时选项

log-declarations

该包识别加载时选项 log-declarations，这是一个键-值选项，接受值 true 和 false。默认情况下，该选项设置为 false，意味着不记录任何声明的命令或环境。通过使用以下方式加载 xparse：

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

每个新声明的命令或环境都将被记录。

5 xparse 实现

1 ⟨*2ekernel | package⟩

2 ⟨@@=cmd⟩

该包文件与 xparse-2018-04-12、xparse-2020-10-01 和 xparse-generic.tex 中的冻结版本旨在跨不同的 LaTeX 发布版本中工作，符合 xparse 所需的最小 expl3 版本。

我们无法在这里使用 \DeclareRelease 的 latexrelease 机制，因为前进或后退的情况不同，由于从 xparse 到 cmd 的前缀更改，所以我们自行检查以确保加载了正确的版本。

所有这些加载假设，如果应加载 latexrelease，那么它应该在 xparse 之前加载。反之则完全没有进行测试，很可能会导致故障。

在 2020-10-01 之前的发布版中，\NewDocumentCommand 未定义，因此未加载 xparse，因此我们可以从 xparse-2018-04-12 加载完整版本。否则，我们将仅从 xparse-2020-10-01 加载已弃用的参数类型。如果我们处于 2ekernel 模式，则意味着向前滚动，因此仅加载来自 xparse-generic.tex 的代码代码。

在 2ekernel 模式中，我们预期定义了两个宏来解析日期，以便我们可以正确比较 \fmtdate。

3 ⟨*2ekernel⟩

```
4 \def\@parse@version#1/#2/#3#4#5\@nil{%
5   \@parse@version@dash#1-#2-#3#4\@nil}
6 \def\@parse@version@dash#1-#2-#3#4#5\@nil{%
7   \if\relax#2\relax\else#1\fi#2#3#4 }
```

8 ⟨/2ekernel⟩

```
\ExplSyntaxOn
\cs_set_protected:Npn \__cmd_tmp:w #1
  {
    \DeclareOption* { \PassOptionsToPackage { \CurrentOption } {#1} }
    \ProcessOptions \relax
    \RequirePackage {#1}
  }
\cs_if_free:NTF \NewDocumentCommand
  {
    \ExplSyntaxOff
    \ifnum\expandafter
        \@parse@version\fmtversion//00\@nil <
        \@parse@version 2020-10-01//00\@nil
      \__cmd_tmp:w { xparse-2018-04-12 }
    \else
⟨2ekernel⟩      \@@input xparse-generic.tex ~
⟨package⟩       \__cmd_tmp:w { xparse-2020-10-01 }
    \fi
    \file_input_stop:
  }
%    \begin{macrocode}
%
% 如果 \cs{NewDocumentCommand} 已经定义，我们要么处于 \LaTeX{} 2020-10-01
% 或更新版本。在前一种情况下，内核中加载的代码具有 |__xparse| 前缀，因此我们
% 将加载 \pkg{xparse-2020-10-01}；否则，我们将继续使用 |xparse.sty|，其中
% 包含带有 |__cmd| 前缀的 \pkg{xparse} 的最后残余部分。要检查这一点，我们只需
% 查看一个带有 |__cmd| 前缀的内部命令。

\_if_exist:NF @__start:nNNnnn @__tmp:w xparse-2020-10-01 _input_stop:
⟨/2ekernel | package⟩
```

在旧版本中，前缀是 xparse，但自 LaTeX 2$_\varepsilon$ 2021 春季版本起，核心代码已包含在内核中，而此文件仅保存了不推荐使用的参数说明符 G、l 和 u。为了使 xparse.sty 加载时不推荐使用的类型能够正常工作，匹配了 LaTeX 2$_\varepsilon$ 内核中的前缀，因此前缀已更改为 cmd。

```
⟨*package⟩

\ProvidesExplPackage{xparse}{2023-10-10}{}
  {L3 Experimental document command parser}
```

5.1 包选项

\l__cmd_options_clist
\l__cmd_log_bool

用键-值选项手动记录信息：手动操作以降低依赖性。

```
\clist_new:N \l__cmd_options_clist
```

```
\DeclareOption* { \clist_put_right:NV \l__cmd_options_clist \CurrentOption }
\ProcessOptions \relax
\cs_set_protected:Npn \__cmd_tmp:w #1
  {
    \keys_define:nn {#1}
      {
        log-declarations .bool_set:N = \l__cmd_log_bool ,
        log-declarations .initial:n  = false
      }
    \keys_set:nV {#1} \l__cmd_options_clist
    \bool_if:NTF \l__cmd_log_bool
      { \msg_redirect_module:nnn {#1} { info } { } }
      { \msg_redirect_module:nnn {#1} { info } { none } }
    \cs_new_protected:Npn \__cmd_unknown_argument_type_error:n ##1
      {
        \msg_error:nnee {#1} { unknown-argument-type }
          { \__cmd_environment_or_command: } { \tl_to_str:n {##1} }
      }
  }
\msg_if_exist:nnTF { cmd } { define-command }
  { \__cmd_tmp:w { cmd } }
  { \__cmd_tmp:w { ltcmd } }
```

(*End of definition for* `\l__cmd_options_clist` *and* `\l__cmd_log_bool`.)

5.2 规范化参数规范

`\__cmd_normalize_arg_spec_loop:n` 遍历参数规范，调用针对每种参数类型的辅助程序。如果任何参数未知，则停止定义。

```
\cs_gset_protected:Npn \__cmd_normalize_arg_spec_loop:n #1
  {
    \quark_if_recursion_tail_stop:n {#1}
    \int_incr:N \l__cmd_current_arg_int
    \cs_if_exist_use:cF { __cmd_normalize_type_ \tl_to_str:n {#1} :w }
      {
        \__cmd_unknown_argument_type_error:n {#1}
        \__cmd_bad_def:wn
      }
  }
```

(*End of definition for* `\__cmd_normalize_arg_spec_loop:n`.)

`\__cmd_normalize_type_g:w` 这些参数类型是更一般类型的别名，例如具有默认参数 `-NoValue-`。

```
\cs_new_protected:Npe \__cmd_normalize_type_g:w
  { \exp_not:N \__cmd_normalize_type_G:w { \exp_not:V \c_novalue_tl } }
```

(*End of definition for* `\__cmd_normalize_type_g:w`.)

`\__cmd_normalize_type_G:w` 可选参数类型。检查所有必需的数据是否存在（如果适用，是否由单个字符组成），并检查可展开命令的禁用类型。然后，在每种情况下将数据存储在 `\l__cmd_arg_spec_tl` 中，以供后续检查，在 `\l__cmd_last_delimiters_tl` 中存储决定是否有可选参数的标记（对于大括号，存储为 `{}`，稍后将其视为空分隔符）。

```
\cs_new_protected:Npn \__cmd_normalize_type_G:w #1
  {
    \quark_if_recursion_tail_stop_do:nn {#1} { \__cmd_bad_arg_spec:wn }
    \__cmd_normalize_check_gv:N G
    \__cmd_add_arg_spec:n { G {#1} }
    \tl_put_right:Nn \l__cmd_last_delimiters_tl { { } }
    \__cmd_normalize_arg_spec_loop:n
  }
```

(*End of definition for* `\__cmd_normalize_type_G:w`.)

`\__cmd_normalize_type_l:w` `\__cmd_normalize_type_u:w` 必选参数。首先检查所需的数据是否存在，是否由单个字符组成（如果适用），并且如有必要，检查参数类型是否允许用于可展开命令。然后将数据保存在 `\l__cmd_arg_spec_tl` 中，计算必选参数的数量，并清空最后分隔符的列表。

```
\cs_new_protected:Npn \__cmd_normalize_type_l:w
  {
    \__cmd_normalize_check_lu:N l
    \__cmd_add_arg_spec_mandatory:n { l }
    \__cmd_normalize_arg_spec_loop:n
  }
\cs_new_protected:Npn \__cmd_normalize_type_u:w #1
  {
    \quark_if_recursion_tail_stop_do:nn {#1} { \__cmd_bad_arg_spec:wn }
    \__cmd_normalize_check_lu:N u
    \__cmd_add_arg_spec_mandatory:n { u {#1} }
    \__cmd_normalize_arg_spec_loop:n
  }
```

(*End of definition for* `\__cmd_normalize_type_l:w` *and* `\__cmd_normalize_type_u:w`.)

5.3 设置标准签名

`\__cmd_add_type_G:w` 对于 `G` 类型，抓取器和默认值被添加到签名中。

```
\cs_new_protected:Npn \__cmd_add_type_G:w #1
```

```
  {
    \__cmd_flush_m_args:
    \__cmd_add_default:n {#1}
    \__cmd_add_grabber:N G
    \__cmd_prepare_signature:N
  }
```

(*End of definition for* `\__cmd_add_type_G:w`.)

`\__cmd_add_type_l:w` 寻找 `l` 参数非常简单：除了添加抓取器外，没有其他要做的事情。

```
\cs_new_protected:Npn \__cmd_add_type_l:w
  {
    \__cmd_flush_m_args:
    \__cmd_add_default:
    \__cmd_add_grabber:N l
    \__cmd_prepare_signature:N
  }
```

(*End of definition for* `\__cmd_add_type_l:w`.)

`\__cmd_add_type_u:w` 在设置阶段，`u` 类型参数与 `G` 类型相同，除了抓取器函数的名称不同。

```
\cs_new_protected:Npn \__cmd_add_type_u:w #1
  {
    \__cmd_flush_m_args:
    \__cmd_add_default:
    \__cmd_add_grabber:N u
    \tl_put_right:Nn \l__cmd_signature_tl { {#1} }
    \__cmd_prepare_signature:N
  }
```

(*End of definition for* `\__cmd_add_type_u:w`.)

5.4 设置可扩展类型

`\__cmd_add_expandable_type_l:w` 通过重复使用类型 `u`，因为在 TEX 宏的参数文本以 `#` 结束，实际上就会以开放花括号为定界符。

```
\cs_new_protected:Npn \__cmd_add_expandable_type_l:w
  { \__cmd_add_expandable_type_u:w ## }
```

(*End of definition for* `\__cmd_add_expandable_type_l:w`.)

`\__cmd_add_expandable_type_u:w` 定义一个辅助函数，它将直接用在签名中。它抓取一个由 `#1` 定界的参数，并将其放置在 `\q__cmd` 之前。

```
\cs_new_protected:Npn \__cmd_add_expandable_type_u:w #1
```

```
{
  \__cmd_add_default:
  \bool_if:NTF \l__cmd_long_bool
    { \cs_set:cpn }
    { \cs_set_nopar:cpn }
    { \l__cmd_expandable_aux_name_tl } ##1 \q__cmd ##2 ##3 ##4 #1
    { ##1 {##4} \q__cmd ##2 ##3 }
  \__cmd_add_expandable_grabber:nn { u }
    { \exp_not:c  { \l__cmd_expandable_aux_name_tl } }
  \__cmd_prepare_signature:N
}
```

(*End of definition for* `\__cmd_add_expandable_type_u:w`.)

5.5 复制命令及其内部结构

复制命令的装置几乎完全位于 `ltcmd.dtx` 中，预先加载在 LaTeX 2_ε 内核中。关于复制已弃用的参数类型 `G`、`l` 和 `u` 的缺失部分归结为复制 `u` 的可展开抓取器。

`\__cmd_copy_grabber_u:w` 可展开的 `u` 类型使用专用抓取器，就像 `D` 类型一样，只是它的两个分隔符标记被省略，因此为了复制它，我们只需复制一个 `D` 类型，并将最后两个参数留空：

```
\cs_new_protected:Npn \__cmd_copy_grabber_u:w #1 #2 #3
  { \__cmd_copy_grabber_D:w {#1} {#2} {#3} { } { } }
```

(*End of definition for* `\__cmd_copy_grabber_u:w`.)

5.5.1 显示命令的定义

`\c__cmd_show_type_u_tl` `\c__cmd_show_type_G_tl` 与复制相同，此处仅缺少一些小部分。具体来说，有两个令牌列表告诉 ltcmd 机制如何处理这里定义的参数类型。`G` 和 `u` 都被分类为 `3`：带有默认值的命令。对于 `u` 来说实际并非如此，但这足够接近以获得我们想要的输出。

```
\tl_const:Nn \c__cmd_show_type_u_tl { 3 }
\tl_const:Nn \c__cmd_show_type_G_tl { 3 }
```

(*End of definition for* `\c__cmd_show_type_u_tl` *and* `\c__cmd_show_type_G_tl`.)

5.6 抓取参数

`\__cmd_grab_G:w` `\__cmd_grab_G_long:w` `\__cmd_grab_G_obey_spaces:w` `\__cmd_grab_G_long_obey_spaces:w` `\__cmd_grab_G_aux:nNN` 可选组通过含义进行检查，以便相同的代码适用于例如类似 ConTeXt 的输入方式。

```
\cs_new_protected:Npn \__cmd_grab_G:w #1 \__cmd_run_code:
  {
    \__cmd_grab_G_aux:nNN {#1} \cs_set_protected_nopar:Npn
      \__cmd_peek_nonspace:NTF
```

```
  }
\cs_new_protected:Npn \__cmd_grab_G_long:w #1 \__cmd_run_code:
  {
    \__cmd_grab_G_aux:nNN {#1} \cs_set_protected:Npn
      \__cmd_peek_nonspace:NTF
  }
\cs_new_protected:Npn \__cmd_grab_G_obey_spaces:w #1 \__cmd_run_code:
  {
    \__cmd_grab_G_aux:nNN {#1} \cs_set_protected_nopar:Npn
      \peek_meaning:NTF
  }
\cs_new_protected:Npn \__cmd_grab_G_long_obey_spaces:w #1 \__cmd_run_code:
  {
    \__cmd_grab_G_aux:nNN {#1} \cs_set_protected:Npn
      \peek_meaning:NTF
  }
\cs_new_protected:Npn \__cmd_grab_G_aux:nNN #1#2#3
  {
    \tl_set:Nn \l__cmd_signature_tl {#1}
    \exp_after:wN #2 \l__cmd_fn_tl ##1
      { \__cmd_add_arg:n {##1} }
    #3 \c_group_begin_token
      { \l__cmd_fn_tl }
      { \__cmd_add_arg:o \c_novalue_tl }
  }
```

(*End of definition for* `\__cmd_grab_G:w` *and others.*)

`\__cmd_grab_l:w`
`\__cmd_grab_l_long:w`
`\__cmd_grab_l_aux:nN`

参数抓取器用于必选的 TEX 参数非常简单。

```
\cs_new_protected:Npn \__cmd_grab_l:w #1 \__cmd_run_code:
  { \__cmd_grab_l_aux:nN {#1} \cs_set_protected_nopar:Npn }
\cs_new_protected:Npn \__cmd_grab_l_long:w #1 \__cmd_run_code:
  { \__cmd_grab_l_aux:nN {#1} \cs_set_protected:Npn }
\cs_new_protected:Npn \__cmd_grab_l_aux:nN #1#2
  {
    \tl_set:Nn \l__cmd_signature_tl {#1}
    \exp_after:wN #2 \l__cmd_fn_tl ##1##
      { \__cmd_add_arg:n {##1} }
    \l__cmd_fn_tl
  }
```

(*End of definition for* `\__cmd_grab_l:w`, `\__cmd_grab_l_long:w`, *and* `\__cmd_grab_l_aux:nN`.)

`\__cmd_grab_u:w`
`\__cmd_grab_u_long:w`
`\__cmd_grab_u_aux:nnN`

抓取到一组记号的过程相当简单：定义抓取器，然后收集。

```
\cs_new_protected:Npn \__cmd_grab_u:w #1#2 \__cmd_run_code:
  { \__cmd_grab_u_aux:nnN {#1} {#2} \cs_set_protected_nopar:Npn }
\cs_new_protected:Npn \__cmd_grab_u_long:w #1#2 \__cmd_run_code:
  { \__cmd_grab_u_aux:nnN {#1} {#2} \cs_set_protected:Npn }
\cs_new_protected:Npn \__cmd_grab_u_aux:nnN #1#2#3
  {
    \tl_set:Nn \l__cmd_signature_tl {#2}
    \exp_after:wN #3 \l__cmd_fn_tl ##1 #1
      { \__cmd_add_arg:n {##1} }
    \l__cmd_fn_tl
  }
```

(*End of definition for* `\__cmd_grab_u:w`, `\__cmd_grab_u_long:w`, *and* `\__cmd_grab_u_aux:nnN`.)

`\__cmd_expandable_grab_u:w` 原来什么都不用做：接着是一个以函数命名的辅助功能，它完成了一切。

```
\cs_new_eq:NN \__cmd_expandable_grab_u:w \prg_do_nothing:
```

(*End of definition for* `\__cmd_expandable_grab_u:w`.)

5.7 访问参数规范

有一段时间这是包含在内核中的，因此在本小节中我们必须始终使用 gset（或类似的）。

`\__cmd_get_arg_spec_error:N` 当尝试获取非 xparse 命令或环境的参数规范时提供信息性错误。

`\__cmd_get_arg_spec_error:n`

`\__cmd_get_arg_spec_error_aux:n`

```
\cs_gset_protected:Npn \__cmd_get_arg_spec_error:N #1
  {
    \bool_set_false:N \l__cmd_environment_bool
    \tl_set:Nn \l__cmd_fn_tl {#1}
    \__cmd_get_arg_spec_error_aux:n { \cs_if_exist:NTF #1 }
  }
\cs_gset_protected:Npn \__cmd_get_arg_spec_error:n #1
  {
    \bool_set_true:N \l__cmd_environment_bool
    \str_set:Ne \l__cmd_environment_str {#1}
    \__cmd_get_arg_spec_error_aux:n
      { \cs_if_exist:cTF { \l__cmd_environment_str } }
  }
\cs_gset_protected:Npn \__cmd_get_arg_spec_error_aux:n #1
  {
    #1
      {
        \msg_error:nne { cmd } { non-xparse }
```

```
          { \__cmd_environment_or_command: }
      }
      {
        \msg_error:nne { cmd } { unknown }
          { \__cmd_environment_or_command: }
      }
  }
```

(*End of definition for* `\__cmd_get_arg_spec_error:N`, `\__cmd_get_arg_spec_error:n`, *and* `\__cmd_get_arg_spec_error_aux:n`.)

`\__cmd_get_arg_spec:NTF` 如果命令不是 xparse 命令，则报错。如果是，它的第二个“item”就是参数规范。

```
\cs_gset_protected:Npn \__cmd_get_arg_spec:NTF #1#2#3
  {
    \__kernel_cmd_if_xparse:NTF #1
      {
        \tl_set:Ne \ArgumentSpecification { \tl_item:Nn #1 { 2 } }
        #2
      }
      {#3}
  }
```

(*End of definition for* `\__cmd_get_arg_spec:NTF`.)

`\ArgumentSpecification`

```
\tl_clear_new:N \ArgumentSpecification
```

(*End of definition for* `\ArgumentSpecification`. *This variable is documented on page* ??.)

`\__cmd_get_arg_spec:N` 现在恢复参数规范就很简单了。

`\__cmd_get_arg_spec:n`

```
\cs_gset_protected:Npn \__cmd_get_arg_spec:N #1
  {
    \__cmd_get_arg_spec:NTF #1 { }
      { \__cmd_get_arg_spec_error:N #1 }
  }
\cs_gset_protected:Npn \__cmd_get_arg_spec:n #1
  {
    \exp_args:Nc \__cmd_get_arg_spec:NTF
      { environment~ \tl_to_str:n {#1} }
      { }
      { \__cmd_get_arg_spec_error:n {#1} }
  }
```

(*End of definition for* `\__cmd_get_arg_spec:N` *and* `\__cmd_get_arg_spec:n`.)

__cmd_show_arg_spec:N
__cmd_show_arg_spec:n

显示参数规范简单来说就是找到它然后调用 \tl_show:N 函数。

```
\cs_gset_protected:Npn \__cmd_show_arg_spec:N #1
  {
    \__cmd_get_arg_spec:NTF #1
      { \tl_show:N \ArgumentSpecification }
      { \__cmd_get_arg_spec_error:N #1 }
  }
\cs_gset_protected:Npn \__cmd_show_arg_spec:n #1
  {
    \exp_args:Nc \__cmd_get_arg_spec:NTF
      { environment~ \tl_to_str:n {#1} }
      { \tl_show:N \ArgumentSpecification }
      { \__cmd_get_arg_spec_error:n {#1} }
  }
```

(*End of definition for* __cmd_show_arg_spec:N *and* __cmd_show_arg_spec:n.)

\GetDocumentCommandArgSpec
\GetDocumentEnvironmentArgSpec
\ShowDocumentCommandArgSpec
\ShowDocumentEnvironmentArgSpec

简单映射，需要检查参数是否为单个控制序列或活动字符。

```
\cs_gset_protected:Npn \GetDocumentCommandArgSpec #1
  {
    \__cmd_check_definable:nNT {#1} \GetDocumentCommandArgSpec
      { \__cmd_get_arg_spec:N #1 }
  }
\cs_gset_eq:NN \GetDocumentEnvironmentArgSpec \__cmd_get_arg_spec:n
\cs_gset_protected:Npn \ShowDocumentCommandArgSpec #1
  {
    \__cmd_check_definable:nNT {#1} \ShowDocumentCommandArgSpec
      { \__cmd_show_arg_spec:N #1 }
  }
\cs_gset_eq:NN \ShowDocumentEnvironmentArgSpec \__cmd_show_arg_spec:n
```

(*End of definition for* \GetDocumentCommandArgSpec *and others. These functions are documented on page* 15.)

```
\msg_set:nnnn { cmd } { non-xparse }
  { \str_uppercase:n #1~not~defined~using~xparse. }
  {
    You~have~asked~for~the~argument~specification~for~the~#1,~
    but~this~was~not~defined~using~xparse.
  }
\msg_set:nnnn { cmd } { unknown }
  { Unknown~document~#1. }
  {
    You~have~asked~for~the~argument~specification~for~the~#1,~
    but~it~is~not~defined.
```

```
271     }
272 ⟨/package⟩
```