定义 *ekeys-cmd*

雾月　Longaster

2024 年 4 月 25 日

§ 1 前言 2
§ 2 复制和显示命令 2
§ 3 通用命令钩子 2
§ 4 控制序列获取其需要的参数 3
§ 5 lt3ekeyscmd 和 lt3ekeysext 简介 4
§ 6 常用的参数类型——m、R、D、t 5
§ 7 以纯文本的形式读取参数——v 6
§ 8 获取某些记号之前的内容——l、u、U 6
§ 9 判断记号是否出现——t、p、P、k、T 7
§ 10 可以以任意顺序给出定界符的参数类型——W 8
§ 11 将参数保存到寄存器的类型——c 和 C 8
§ 12 可自定义参数获取方式的类型——K . 10
§ 13 自定义参数扫描方式 11
§ 14 参数预处理器 12
§ 15 *ekeys-cmd* 诸例 14
§ 16 用法说明 17

§ 1 前言

lt3ekeyscmd 和 lt3ekeysext 这两个宏包提供了一个定义命令的新接口。它的使用方式类似于 ltcmd 的 \DeclareDocumentCommand，功能更加丰富的同时也更加复杂。本文介绍详细介绍一下这两个宏包以及它们的前置宏包——collectn，这个宏包在定义获取特殊的参数的命令时是非常有用的。

以下称由 \NewDocumentCommand 系列命令定义的命令为 *document-cmd*，包括它们的完整复制[①]。

首先来看看 \DeclareDocumentCommand 的用法。

\DeclareDocumentCommand ⟨*cmd*⟩ {⟨*arg spec*⟩} {⟨*code*⟩}

这 ⟨*cmd*⟩ 就是要定义的命令，⟨*arg spec*⟩ 是参数说明符列表，⟨*code*⟩ 是命令的定义。

参数说明符（argument specification）是用来标识参数性质的单个字母，有的参数说明符还可能需要一些额外的信息。“参数性质”就是告诉 LaTeX 以什么方式获取这个参数，以及传递给实参什么内容。例如有的参数是通过某个符号是否出现来传递给实参不同的内容，有的参数包裹在一对 [ ] 里，有的参数传递给实参是一些键值对，有的还需对获取的参数经过一些处理再作为实参，等等。根据这些性质的不同，就有了各样的参数类型，不同的参数类型就是根据参数说明符（以及可能的额外信息）来标识的。

参数可以分为必须给出的（即如果没有则会出错），和可选的（即不给出也能正常执行）。定义 *document-cmd* 可用的说明符如下：

- 必须的：m、r、R、v 以及 b（仅能在定义环境时使用）；
- 可选的：o、d、O、D、s、t、e、E。

这些参数类型在 LaTeX 2_ε 内核中定义，可以直接使用。此外，还有标记为过时的：

- 必须的：l、u；
- 可选的：g、G；

它们仅能在加载 xparse 宏包后使用，且定义 *document-cmd* 时一般不推荐使用。

除了参数说明符还有几种参数修饰符，它们可以放在说明符的前面，用于修饰这些参数类型：+、!、=、>。

本文在此并不准备对这些参数类型和修饰符进行说明。不过，*ekeys-cmd* 和 *document-cmd* 的相同参数类型其用法基本相同，且本文会对前者进行详细介绍，因此，读者若不了解上述参数说明符也可继续阅读下去。如若想了解上述参数类型和修饰符的用法，可参考 usrguide.pdf。

§ 2 复制和显示命令

document-cmd 除了可以使用 \NewDocumentCommand 系列命令定义外，还可通过复制另一个 *document-cmd* 得到。不过必须是“完整复制”。

\DeclareCommandCopy ⟨*new cmd*⟩ ⟨*old cmd*⟩

\DeclareCommandCopy 系列命令不仅可以完整复制由 \newcommand、\DeclareRobustCommand 定义的命令，还可以完整复制 *document-cmd* 和 *ekeys-cmd*。

在终端中显示命令的定义也很简单，使用 \ShowCommand{⟨*cmd*⟩} 即可，同样支持由 \newcommand、\DeclareRobustCommand 定义的命令，以及 *document-cmd* 和 *ekeys-cmd*。

§ 3 通用命令钩子

通用命令钩子就是形如 cmd/⟨*cmd name*⟩/before 或 cmd/⟨*cmd name*⟩/after 的钩子。通用命令钩子无需声明即可使用。前者在命令最开始时执行，后者在命令最末尾执行。

\AddToHook、\AddToHookWithArguments、\AddToHookNext、\AddToHookNextWithArguments 这四个命令用于给通用命令钩子添加代码。对于没有声明的命令钩子，可以自动为命令添加使用钩子的代码 \UseHookWithArguments（本文称为自动“修补”），同样支持由 \newcommand、\DeclareRobustCommand 定义的命令，以及 *document-cmd* 和 *ekeys-cmd*。

不过自动添加 \UseHookWithArguments 并非总会执行。

- 对于已经声明的命令钩子，不会自动添加 \UseHookWithArguments;

① 即由 \NewCommandCopy、\RenewCommandCopy、\DeclareDocumentCopy 复制的。

- 对于在导言区使用的 \AddToHook 系列命令，自动修补在 begindocument 钩子中才会执行；
- 对于在正文区使用的 \AddToHook 系列命令，自动修补仅执行一次，命令重定义以后不会再执行自动修补，因为已经自动声明了这个钩子。

自动修补并不总是有效，有时甚至会导致严重的错误。例如：

```
\newcommand\fancybox{\@ifnextchar({\@fancybox}{\@fancybox(5cm)}}
\def\@fancybox(#1)#2{\fbox{\parbox{#1}{#2}}}
```

倘若想在 \fancybox 后面添加一些代码，使用

```
\AddToHook{cmd/fancybox/after}{<code>}
```

结果是 \fancybox 变成了：

```
\newcommand\fancybox{\@ifnextchar({\@fancybox}{\@fancybox(5cm)}%
  \UseHookWithArguments{cmd/fancybox/after}{0}}
```

会导致严重的错误。

因此若要使用命令钩子，最佳实践是在定义命令时就加上 \UseHookWithArguments：

```
\newcommand\fancybox{\@ifnextchar({\@fancybox}{\@fancybox(5cm)}}
\def\@fancybox(#1)#2{\fbox{%
  \UseHookWithArguments{cmd/fancybox/before}{2}{#1}{#2}%
  \parbox{#1}{#2}%
  \UseHookWithArguments{cmd/fancybox/after}{2}{#1}{#2}}}
```

当然，使用了 \UseHookWithArguments 后需要更长的执行时间，具体如何做就需要自行斟酌了。

关于钩子和命令钩子的具体用法，见 lthooks-doc.pdf 和 ltcmdhooks-doc.pdf

§ 4 控制序列获取其需要的参数

根据 TEX 中控制序列获取其需要的参数的方式总体可分为两类：一类是 TEX 语法规定的，另一类是定义宏时设置的。前者基本固定，只需阅读 TEX 引擎的文档即可知晓；而后者则可以十分灵活。

在 LATEX 格式下，命令获取参数的规则有许多一致的地方，例如命令后面跟着一个可选的 *，或者是跟着可选的一个或多个由 [ ] 包裹的实参，最常见的就是由一对 { } 包裹的实参。

而由一对 { } 包裹的“实参”从可使用的 { } 上看又有多种情况：

1. 是除了显式的左括号（记为 ⟨left brace⟩）、右括号（记为 ⟨right brace⟩）和空格[②]之一的单个记号（token），此时无需加上 ⟨left brace⟩ 和 ⟨right brace⟩；

1′. 由 ⟨left brace⟩ 和 ⟨right brace⟩ 包裹，且它们之间的 ⟨left brace⟩ 和 ⟨right brace⟩ 数量保持平衡[③]；

2. 由 { （表示显式的或隐式的类别码[④]为 1 的字符）以及 ⟨right brace⟩ 包裹，且它们之间的 ⟨left brace⟩ 和 ⟨right brace⟩ 数量保持平衡；

3. 由某一特定的类别码为 1 的显式字符和 ⟨right brace⟩ 包裹，且它们之间的 ⟨left brace⟩ 和 ⟨right brace⟩ 数量保持平衡；

4. 由某一特定的隐式左、右括号包裹，且它们之间的 ⟨left brace⟩ 和 ⟨right brace⟩ 数量保持平衡；

5. 由 { 和 } 包裹。

其中第 1 和 1′ 可认为是一类。LATEX 格式中，大部分命令都使用第 1 类（含 1′，如无特别说明，以后同），除了那些暴露的 TEX 原语。

LATEX3 则区分 1 和 1′，前者为 N 类型，后者为 n 类型。实际使用时，尽管在某些情况下它们可以混用，但仍然有例外[⑤]，因此最好严格按照类型指定的用法使用，至少这样不会出错。除非你了解这些命令的定义并且

② 显式字符就是有唯一类别码的普通字符，隐式字符就是被 \let 为显式字符的控制序列或活动字符。显式的左、右括号和空格就是类别码分别为 1、2、10 的字符。

③ 所谓“保持平衡”即数量一致。

④ “类别码”是一个字符记号拥有的属性，在 XETEX 和 LuaTEX中，每一个字符都有唯一的字符码（就是它的 Unicode 编码）和类别码。可以修改字符的类别码为任何有效的值。pdfTEX、XETEX、LuaTEX中类别码的有效值和它们的含义见表 1。

⑤ 如 \token_to_str:N、\token_to_meaning:N 的参数可以是任意记号，\tl_to_str:n、\exp_not:n 可以是第 1′ 或第 2 类，但不能是第 1 类。

确信它们不会更改。

§ 5 lt3ekeyscmd 和 lt3ekeysext 简介

lt3ekeyscmd 提供定义 *ekeys-cmd* 的主要接口，lt3ekeysext 扩展了前者的功能，提供了几个参数修饰符和额外的参数说明符，并且支持自定义参数获取方式。加载 lt3ekeysext 对 *ekeys-cmd* 的使用上没有任何影响，且功能更强大，推荐直接加载 lt3ekeysext。本文示例代码也建立在加载 lt3ekeysext 的基础上。

可通过 `\DeclareEKeysCommand` 来定义 *ekeys-cmd*：

```
\DeclareEKeysCommand   ⟨cmd⟩ {⟨arg spec⟩} {⟨code⟩}
\DeclareEKeysCommand * ⟨cmd⟩ {⟨arg spec⟩} {⟨code⟩}
```

`\DeclareEKeysCommand` 和 `\DeclareDocumentCommand` 使用上几乎没有区别，只是前者还支持星号可选参数，它的功能后面再说明。`\DeclareEKeysCommand` 也可以写为 `\ekeysdeclarecmd`。

以下列出 *ekeys-cmd* 可用的参数说明符，其中在 lt3ekeyscmd 中定义的有：

- 必须的：`m`、`r`、`R`、`l`、`u`、`U`、`v`；
- 可选的：`o`、`O`、`d`、`D`、`s`、`t`、`p`、`P`、`k`、`t`、`T`、`e`、`E`、`w`、`W`、`g`、`G`。

在 lt3ekeysext 中定义的有：

- 必须的：`c`、`C`；
- 自定义的：`K`；
- 预处理指示符：`?`；
- 参数修饰符：`&`、`#`、`@`。

其中，*document-cmd* 标记为过时的 `g`、`G`、`l`、`u` 参数类型，*ekeys-cmd* 是直接支持的，不过其行为略有不同。在此，我们不讨论“应该做什么、不应该做什么”，而注重讨论“能做什么”和“怎么做”。这也是 lt3ekeyscmd 和 lt3ekeysext 的作者编写这两个宏包所遵循的原则。至于要不要这么做，则交由读者自行考量。

这其中有两个参数类型使用了同一个参数说明符，但使用时会有区别，因此即使如此也不会混淆。

ekeys-cmd 的参数说明符与 *document-cmd* 相同的部分，其用法和作用基本相同，不过也有例外的情况，在此先做概述，待后文详细说明：

- 所有参数都是 `\long`，即都可以包含 `par`[6]；
- 向后寻找可选参数时，忽略空格。但这未来可能会更改；
- 带有定界符的参数类型，如 `r`、`d`，默认情况下是不会处理嵌套的定界符的，但可以在该参数说明符的前面加上 `@`，这样就会处理嵌套的定界符了。
- 有默认值的参数类型，默认情况下是不会处理用 `#1` 等引用其它实参的，但可以使用带星号的 `\DeclareEKeysCommand`，这样就和 *document-cmd* 一样了。不过默认值之间不能循环引用。[7]
- `s` 和 `t` 参数类型及带定界符的参数类型，还可以通过加上 `&` 修饰符，使得其实参为等价的原始输入，而不是布尔值或移除掉定界符的结果。
- `e` 和 `E` 参数类型，在 *document-cmd* 其中的 ⟨*tokens*⟩ 是不能重复的，但 *ekeys-cmd* 却可以。
- `g` 和 `G` 参数类型，在寻找其参数时，左括号既可以是显式的，又可以是隐式的；且若没有发现左括号，而发现的是 `\relax`（或 `\ifx` 判断与之相等），则会移除这个 `\relax` 后再继续寻找之后的参数。这样在实际使用时，可以用 `\relax` 阻止 `G` 参数获取实参，又不会干扰其它参数获取实参；
- `v` 参数类型，对于 *document-cmd*，其实参的类别码为 10、12、13 之一，而 *ekeys-cmd* 其实参所含字符的类别码的可能值为 10、12、13（在 (u)pTEX 引擎下，仅 Unicode 编码小于等于 255 的字符有此特性）；且 `^^M` 的类别码为 13；
- “处理器” vs. “预处理器”。与 *document-cmd* 对偶的，*ekeys-cmd* 使用的是预处理器机制。且 *document-cmd* 的处理器不能是 *document-cmd*，换句话说，*document-cmd* 不能嵌套。*ekeys-cmd* 则没有这个限制。

除了直接定义命令外，lt3ekeyscmd 还支持不使用 *ekeys-cmd* 而直接收集参数：

---

[6] `par` 表示一个控制序列，它的名字为 `par`。

[7] 关于默认值，另外还有一种情况与 *document-cmd* 不同：当默认值引用了其它实参，且此参数的值为特殊的 -NoValue- 标记时，*ekeys-cmd* 不会进行引用替换，而 *document-cmd* 会。由于该特殊的标记是内部值，用户一般不能输入，但若放在另一个 *document-cmd* 或 *ekeys-cmd* 则是可能的。

```
\DeclareEKeysCollector   ⟨collector cmd⟩ {⟨arg spec⟩}
\DeclareEKeysCollector * ⟨collector cmd⟩ {⟨arg spec⟩} ⟨collect to cmd⟩ {⟨do code⟩}
\ekeyscollectargs    ⟨collect to cmd⟩ ⟨ekeys-cmd⟩ {⟨do code⟩} ⟨args⟩
\ekeyscollectargs * ⟨collect to cmd⟩ {⟨arg spec⟩} {⟨do code⟩} ⟨args⟩
```

这个命令根据 ⟨*ekeys-cmd*⟩ 获取参数的方式，或指定的 ⟨*arg spec*⟩，从 ⟨*args*⟩ 收集参数并保存到 ⟨*collect to cmd*⟩ 里，然后执行 ⟨*do code*⟩。使用 `\DeclareEKeysCollector` 定义的命令称为 *ekeys-collector*。

其中，使用 `\DeclareEKeysCollector` 和 `\ekeyscollectargs*` 时，⟨*arg spec*⟩ 的数量不限，可以获取超过 9 个参数。

从执行同一作用的命令的执行时间来看，*document-cmd* 长于 *ekeys-cmd* 长于 *ekeys-collector* 约等于使用不带星号的 `\ekeyscollectargs` 直接收集参数。

```
\ekeyscollectorarg {⟨arg number⟩}
```

获取第 ⟨*arg number*⟩ 个参数。可以用于上面四个命令的 ⟨*do code*⟩ 中。会在值不存在时返回 `\q_no_value`，可使用 `\IfQuarkNoValueTF` 判断是否为该值。`\IfNoValueTF` 用于判断是否为特殊的 -NoValue- 标记，它是用来判断可选参数是否有值的。

§ 6 常用的参数类型——m、R、D、t

本节介绍最为常用的参数类型：`m`、`r`、`R`、`o`、`O`、`d`、`D`、`s`、`t`。

对于一个完整的参数类型 ⟨*full spec*⟩，构成为 ⟨*spec*⟩ 或 ⟨*spec*⟩{⟨*default*⟩}，其中 ⟨*spec*⟩ 为 ⟨*spec name*⟩⟨*args*⟩。⟨*spec name*⟩ 就是标识参数类型的字母，⟨*args*⟩ 是参数类型需要的参数，⟨*default*⟩ 也可算作是参数，不过这里单独区分它们。就是说，⟨*full spec*⟩ 为 ⟨*spec name*⟩⟨*args or none*⟩{⟨*default or none*⟩}。

一般情况下，如未特别说明，显式或隐式的空格、左括号、右括号、宏变量字符（一般为 `#`）都不能包含在说明符的 ⟨*args*⟩ 里。

`m` 类型的参数获取的实参是第 1 类带 `{ }` 的。

`r`、`R`、`o`、`O`、`d`、`D` 通过是否给出指定的定界符来判断是否给出此实参。对于前两个，如果没有给出实参就会出错。大小写的区别是，大写的说明符用法为 ⟨*spec*⟩{⟨*default*⟩}，即还需给说明符一个参数，表示如果没有给出实参，则用此作为实参。而小写的说明符设置 ⟨*default*⟩ 为一个特殊的标记：-NoValue-，可以用 `\IfNoValue(TF)` 和 `\IfValue(TF)` 判断实参是否为此标记。

对于 `R` 和 `D`（以及对应的小写。若大小写的功能类似，只有带与不带 ⟨*default*⟩ 的区别，则只写大写，以后同）；⟨*spec*⟩ 为 ⟨*spec name*⟩⟨*token*$_1$⟩⟨*token*$_2$⟩。对于 `O`，⟨*spec*⟩ 为 ⟨*spec name*⟩，相当于 `D[]`。⟨*spec name*⟩ 就是这几个说明符之一，⟨*token*$_1$⟩ 和 ⟨*token*$_2$⟩ 是单个字符或控制序列。如 `R(){NaN}`、`o`、`O{NaN}` 等都是正确的。

这几类带定界符的参数，默认情况下是不支持嵌套的，不过可以在它们前面加上 `@` 修饰符，来标记此参数需要处理嵌套的情形。并且判断定界符是否出现只会检查此定界符对的左定界符是否出现，若左定界符出现了而右定界符没有出现，则会出错。

对于 `s` 和 `t` 通过是否给出指定的字符或控制序列来给出 `\BooleanTrue` 或 `\BooleanFalse`，可通过 `\IfBoolean(TF)` 判断为何者。⟨*spec*⟩ 为 `t`⟨*token*⟩，`s` 相当于 `t*`。

对于 ⟨*token*⟩ 的判断，是很严格的。如果是一个字符，则字符码和类别码都必须一致；如果是一个控制序列，则是控制序列的名字一致。在通过是否给出指定的记号来判断是否给出实参时，所有参数类型都遵循这个规则。

上面这些参数类型，除了 `m` 外，支持加上 `&` 修饰符，这样实参为等价的原始输入。参数的默认值不会自动加上定界符。如果一个命令有带定界符的参数，而它的定义使用了也有带定界符参数的命令，使用此前缀能减少大量的 `\IfValue(TF)` 判断。如：

```
\DeclareEKeysCommand \mycaption { & s &@ O{} m } {<some code>\caption#1#2{#3}}
```

如果出现了 `*` 或 `[]` 可选参数，那么会 `#1` 为 `*`，而 `#2` 为 `[{args}]`，这样省略了大量重复的 `\If..` 判断。

由于 *ekeys-cmd* 不支持 `=` 修饰符，且若设置了 `&` 修饰符，那么会自动在定界符中间加上一对 `{}`，这会干扰 *document-cmd* 判断其参数是否为键值对。读者需注意这种情况。

代码 1

```
\DeclareEKeysCommand \faa { &s O{} &d() m } {\detokenize{[#1|#2|#3|#4]}}
\DeclareEKeysCommand \fbb { s @O{} @d() m } {\detokenize{[#1|#2|#3|#4]}}
\DeclareEKeysCommand \fcc { s @o @d() m }
  {\IfBooleanTF{#1}{starred}{non-starred}, %
    \IfValueTF{#2}{b valued}{b no-valued}, %
    \IfNoValueTF{#3}{p no-valued}{p valued}, %
    \{#4\}}
\ttfamily\obeylines
\faa * ({paren(p)}) m; \faa [{bracket[b]}] {mandatory};
\fbb * (paren(p)) m;   \fbb [bracket[b]] {mandatory};
\fcc * (paren(p)) m;   \fcc [bracket[b]] {mandatory};
```

```
[*||({paren(p)})|m]; [|bracket[b]|-NoValue-|mandatory];
[\BooleanTrue ||paren(p)|m]; [\BooleanFalse |bracket[b]|-NoValue-|mandatory];
starred, b no-valued, p valued, {m}; non-starred, b valued, p no-valued, {mandatory};
```

使用 \ekeyscollectorarg 或 \tl_item:Nn 即可获取 *ekeys-collector* 保存的参数的值。

代码 2

```
\DeclareEKeysCollector \faa { m m @o @d() m m m m m m m }
\faa\tmp{\meaning\tmp.} 12 [[B]](b(b)) 56789ABCD.
\faa\tmp{\edef\tmp{\ekeyscollectorarg{10}}\meaning\tmp.} % 把第10个值保存到 \tmp
  12 [[a]](b(b)) 56789ABCD.

\DeclareEKeysCollector*\fbb { m m @o @d() m m m m m m m } \tmp {\meaning\tmp.}
\fbb 12 [[B]](b(b)) 56789ABCD.
```

```
macro:->{1}{2}{[B]}{b(b)}{5}{6}{7}{8}{9}{A}{B}.CD. macro:->A.CD.
macro:->{1}{2}{[B]}{b(b)}{5}{6}{7}{8}{9}{A}{B}.CD.
```

§ 7 以纯文本的形式读取参数——v

v 以纯文本的形式读取参数，它要读取的参数不能做为另一个命令的参数（不能是 tokenized）。它的实参所含字符的类别码的可能值为 10、12、13 （在 (u)pTEX 引擎下，仅 Unicode 编码小于等于 255 的字符有此特性）。且 ^^M 的类别码为 13，默认情况下，它导致换行。

代码 3

```
\DeclareEKeysCommand \faa { v } {开始#1结束}
\faa {@!#$ab`";}
\faa |@!#$ab`";|
```

```
开始 @!#$ab`"; 结束 开始 @!#$ab`"; 结束
```

§ 8 获取某些记号之前的内容——l、u、U

本节介绍的是可获取某些（某个）记号之前的所有内容的参数类型：l、u、U。

l 的 ⟨*full spec*⟩ 为 l。它获取的是字符码为 123，类别码为 1 的记号（也就是通常的左括号 {）之前的所有内容。如果不存在这样的左括号，那么将会出错。

u 的 ⟨*full spec*⟩ 为 u{⟨*tokens*$_1$⟩}，U 的 ⟨*full spec*⟩ 为 U{⟨*tokens*$_1$⟩}{⟨*tokens*$_2$⟩}。它们获取的是 ⟨*tokens*$_1$⟩ 之前的所有内容，并移除 ⟨*tokens*$_1$⟩。U 还会在移除 ⟨*tokens*$_1$⟩ 之后留下 ⟨*tokens*$_2$⟩。

u 和 U 支持使用 # 修饰符来加快执行速度。

代码 4

```
\DeclareEKeysCommand \faa { u{\relax\empty} l } {\detokenize{[#1|#2]}}
\DeclareEKeysCommand \fbb { # U{jk}{lm} l } {\detokenize{[#1|#2]}}
\ttfamily\obeylines
\faa a list tokens \relax end \relax\empty {?};
\fbb a b c i j k jk{?};
```

```
[a list tokens \relax end |]?;
[a b c i j k |lm]?;
```

§ 9 判断记号是否出现——t、p、P、k、T

本节介绍的是用来判断记号是否出现的参数类型：s、t、p、P、k、t、T。其中 s 和 t 已经在第 § 6 节介绍过了。

p 的 ⟨*full spec*⟩ 为 p{⟨*tokens*⟩}，P 的 ⟨*full spec*⟩ 为 P{⟨*tokens*⟩}{⟨*indexed*⟩}。它们用来判断 ⟨*tokens*⟩ 中的某一个记号是否出现，如果出现了，对于 p，其实参为该记号在 ⟨*tokens*⟩ 中的位置，若没有出现则实参为 0；对于 P，其实参为用该记号在 ⟨*tokens*⟩ 中的位置索引 ⟨*indexed*⟩，⟨*indexed*⟩ 的长度为 ⟨*tokens*⟩ 的长度加一，即 P 的完整 ⟨*full spec*⟩ 为：P {⟨$token_1$⟩⟨$token_2$⟩...⟨$token_n$⟩} {{⟨$entry_0$⟩}{⟨$entry_1$⟩}{⟨$entry_2$⟩}...{⟨$entry_n$⟩}}。它们支持 # 修饰符。p 支持 & 修饰符。

k 的 ⟨*full spec*⟩ 为 k{⟨*keyword*⟩}，用来判断关键字 ⟨*keyword*⟩ 是否出现。为了判断 ⟨*keyword*⟩ 是否存在，它会自动展开后面的内容。⟨*keyword*⟩ 被转化为普通字符，且忽略大小写和类别码[⑧]。实参为 \BooleanTrue 和 \BooleanFalse 之一。它支持 & 修饰符。

t 和 T 的 ⟨*spec*⟩ 为 ⟨*spec name*⟩{⟨*paired tokens*⟩}，用于判断多个定界符对 ⟨*paired tokens*⟩ 中的某一对是否出现，T 还可设置默认值。它们占用 2 个参数，前一个为该字符对在 ⟨*paired tokens*⟩ 中的位置，若没有出现则为 0。后一个实参为定界符中间的内容，若没有则为设置的默认值。这些定界符每对的第二个还可以为空或空格。每对第一个必须不同，第二个可以相同。它们支持 #、@、& 修饰符，且使用 & 时会自动使用 #。

注意到 t 说明符有两个用法，一个为 t⟨*token*⟩，另一个为 t{⟨*paired tokens*⟩}，它们是完全不同的类型。⟨*token*⟩ 只能有一个记号，而 ⟨*paried tokens*⟩ 的内容是成对存在的，至少有 2 项。

代码 5

```
\DeclareEKeysCommand \faa { p{*+.} k{shipout} } {\detokenize{[#1|#2]}}
\DeclareEKeysCommand \fbb { P{*+.}{{}*+.} &k{shipout} }{\detokenize{[#1|#2]}}
\ttfamily\obeylines
\def\ipout{ipout}
\faa + shipout; \faa * SHIPOUT; \faa sh\ipout; \faa ;
\fbb + shipout; \fbb * SHIPOUT; \fbb sh\ipout; \fbb ;
```

```
[2|\BooleanTrue ]; [1|\BooleanTrue ]; [0|\BooleanTrue ]; [0|\BooleanFalse ];
[+|shipout]; [*|shipout]; [|shipout]; [|];
```

当 T 类型使用 # 前缀且有控制序列作为定界符时，定义 *ekeys-cmd* 时 \escapechar 的值必须和使用这个 *ekeys-cmd* 命令时的值一致。一般情况下总是一致的。

代码 6

```
\DeclareEKeysCommand \faa { t{ () [] *{} } } {\detokenize{{#1|#2}}}
\DeclareEKeysCommand \fbb { @& T{ () [] *{} }{NaN} } {\detokenize{{#1|#2}}}
\DeclareEKeysCommand \fcc { @& t{ [] \a\b *{ } } } {\detokenize{{#1|#2}}}
\ttfamily\obeylines
\faa [{bracket[b]}]; \faa * m; \faa *{at}; \faa ;
\fbb [bracket[b]];   \fbb * m; \fbb *{at}; \fbb ;
\fcc [bracket[b]];   \fcc \a a\a r\b\b \fcc *aa*a b ; \fcc ;
```

```
{2|bracket[b]}; {3|m}; {3|at}; {0|-NoValue-};
{2|[{bracket[b]}]}; {3|*{m}}; {3|*{at}}; {0|NaN};
{1|[{bracket[b]}]}; {2|\a {a\a r\b }\b }{3|*{aa*a b} }; {0|-NoValue-};
```

⑧ 严格地来说，“转化为普通字符”是先对 ⟨*keyword*⟩ 执行 \detokenize（\tl_to_str:n），然后把字符码为 32、类别码为 10 的字符替换为字符码为 32、类别码为 12 的字符。“忽略类别码”不会忽略类别码为 1、2、10、13 的字符，它们会终止展开和判断。

§ 10 可以以任意顺序给出定界符的参数类型——W

本节介绍的参数类型和 D、T 类似，但不同的是它们可以以任意顺序匹配多个定界符对：e、E、w、W。

e 和 E 的 ⟨*spec*⟩ 为 ⟨*spec name*⟩{⟨*tokens*⟩}。E 可以为 ⟨*tokens*⟩ 中的每一个都设置默认值。E 类型的完整形式为 E {⟨$token_1$⟩⟨$token_2$⟩...⟨$token_n$⟩} {{⟨$default_1$⟩}...{⟨$default_2$⟩}...{⟨$default_n$⟩}}。它们占用 *n* 个参数。在获取参数时 ⟨$token_i$⟩ 可以以任意顺序出现，但作为实参时仍然按照 ⟨*tokens*⟩ 所给的顺序。例如 \foo 的参数为 e{_^}，那么 \foo _a^b，\foo ^b_a 皆可。

仍然要提及的是，⟨*tokens*⟩ 的匹配是严格的，即字符码和类别码都必须相同。

⟨$token_i$⟩ 和 ⟨$token_j$⟩ ($i \neq j$) 可以相同，匹配参数时，已经用过的 ⟨*token*⟩ 不会由于后面的出现了相同 ⟨*token*⟩ 而更新其实参。

w 和 W 的 ⟨*spec*⟩ 为 ⟨*spec name*⟩{⟨*paired tokens*⟩}。W 可以为 ⟨*paired tokens*⟩ 中的每一对都设置默认值。W 类型的完整形式为 W {⟨$token_{11}$⟩⟨$token_{12}$⟩...⟨$token_{n1}$⟩⟨$token_{n2}$⟩} {{⟨$default_1$⟩}...{⟨$default_2$⟩}...{⟨$default_n$⟩}}。它们占用 *n* 个参数。在获取参数时每对定界符可以以任意顺序出现。但作为实参时仍然按照 ⟨*paired tokens*⟩ 中定界符对所给的顺序。相同的定界符对可以重复出现。定界符的左边相同时，右边也必须相同。定界符的右边可以为空或空格，但左边不可以。

e、E、w 都是 W 的特例。它们都支持 #、@、& 修饰符，且使用 & 时会自动使用 #。当使用 # 前缀且有控制序列作为定界符时，定义 *ekeys-cmd* 时 \escapechar 的值必须和使用这个 *ekeys-cmd* 命令时的值一致。一般情况下总是一致的。

代码 7

```
\catcode`\^=7 \catcode`\_=8
\DeclareEKeysCommand \faa { e{_^_} } {\detokenize{[#1|#2|#3]}}
\DeclareEKeysCommand \fbb { &e{_^_} } {\detokenize{[#1|#2|#3]}}
\DeclareEKeysCommand \fcc { @w{ [] () [] } } {\detokenize{{#1|#2|#3}}}
\DeclareEKeysCommand \fdd { @&w{ [] *{ } [] } } {\detokenize{{#1|#2|#3}}}
\DeclareEKeysCommand \fee { @&w{ [] *{ } \a\b } } {\detokenize{{#1|#2|#3}}}
\ttfamily\small\obeylines
\faa _a _{abc} ^e; \faa ^{eee} _a; \faa ;
\fbb _a _{abc} ^e; \fbb ^{eee} _a; \fbb ;
\fcc (paren(p)) [bracket[b]] [B[B]]; \fcc [bracket[b]]; \fcc ;
\fdd *hhh*h j [bracket[b]] [{bracket[}]; \fdd *jj ;
\fee \a zmg\a ?\b\b [[?]];
```

```
[a|e|abc]; [a|eee|-NoValue-]; [-NoValue-|-NoValue-|-NoValue-];
[_{a}|^{e}|_{abc}]; [_{a}|^{eee}|-NoValue-]; [-NoValue-|-NoValue-|-NoValue-];
{bracket[b]|paren(p)|B[B]}; {bracket[b]|-NoValue-|-NoValue-}; {-NoValue-|-NoValue-|-NoValue-};
{[{bracket[b]}]|*{hhh*h j} |[{bracket[}]}; {-NoValue-|*{jj} |-NoValue-};
{[{[?]}]|-NoValue-|\a {zmg\a ?\b }\b };
```

上述几节提到的参数说明符的用法都比较简单，读者能很快掌握。而接下来的几个小节会介绍更加“高级”同时也更为复杂的参数类型：c、C、K 以及预处理指示符 ?。

§ 11 将参数保存到寄存器的类型——c 和 C

从本小节开始就有一定的难度了，读者需要有 TEX 和 LATEX3 宏编程的基础知识。

c 的用法为 c{⟨*collect spec*⟩}。

C 的用法为 C{⟨*allocated collect spec*⟩}，⟨*allocated collect spec*⟩ 为 ⟨*register*⟩⟨*collect spec*⟩。⟨*register*⟩ 是一个已经用诸如 \newcount、\int_new:N 等分配了的寄存器。

⟨*collect spec*⟩ 有 8 类用法：

i 为 count（int）寄存器赋值；

d 为 dimen（dim）寄存器赋值；

s 为 skip 寄存器赋值；

m 为 muskip 寄存器赋值；

`t` 为 `toks` 寄存器赋值；

以上 5 种，其实参为所分配的寄存器，而不是它们所保存的值。实参作为 `\the` 的参数，或作为 LaTeX3 的 `V` 变体的参数都能获得它们保存的值。[9]

`b` `b`⟨*spec*⟩，保存到水平盒子中。⟨*spec*⟩ 可以为：

- `*`，表示在获取参数时可以动态调整水平盒子的宽度；
- `[`⟨*width*⟩`]` `[`⟨*pos*⟩`]`，方括号定界的为可选参数，正如 `\makebox` 的前两个可选参数，分别设置盒子的宽度和水平对齐方式。

`w` `w`⟨*spec*⟩，为一个固定宽度的盒子赋值，类似于把盒子放在 `minipage` 里。⟨*spec*⟩ 可以为：

- `{`⟨*width*⟩`}`，盒子的宽度。盒子是一个垂直盒子，可以用 `\vsplit` 或 `\vbox_set_split_to_ht:NNn` 分割；
- `[`⟨*vpos*⟩`]` `[`⟨*height*⟩`]` `[`⟨*inner pos*⟩`]``{`⟨*width*⟩`}`，方括号定界的为可选参数，正如 `minipage` 的参数。盒子是一个水平盒子。若没有可选参数，则和上一个用法一样。

`v` `v`⟨*spec*⟩，为一个设置了最长宽度的盒子赋值，类似于把盒子放在 `varwidth` 里。⟨*spec*⟩ 的用法和作用与 `w` 一样。

以上这些用法的参数（可选、必须参数）之间不能有任何空格。且需注意 ⟨*spec*⟩ 自身是不能有 `{}` 的，花括号的添加与否由具体用法决定。它们的实参为所分配的盒子。可作为 `\box`，`\box_use:N` 等命令的参数。[10]

代码 8

```
\DeclareEKeysCommand \faa { c{d} ci } {Length: \the#1; Number: \the#2;}
\newdimen\fbbtempdim \newcount\fbbtempint
\DeclareEKeysCommand \fbb { C{\fbbtempdim} C\fbbtempint }
  {Length: \the#1; Number: \the#2;}
\ttfamily\obeylines
\faa 12pt 19 |\faa\dimexpr 12pt+8pt-10pt\relax \inteval{19+3-12} |
\fbb 12pt 19 |\fbb\dimexpr 12pt+8pt-10pt\relax \inteval{19+3-12} |
```

```
Length: 12.0pt; Number: 19;|Length: 10.0pt; Number: 10;|
Length: 12.0pt; Number: 19;|Length: 10.0pt; Number: 10;|
```

代码 9

```
\DeclareEKeysCommand \faa { cb c{b*} } {\fbox{\box#1} \fbox{\box#2}}
\newbox\fbbtempboxa \newbox\fbbtempboxb
\DeclareEKeysCommand\fbb{C\fbbtempboxa C{\fbbtempboxb b*}}{\fbox{\box#1} \fbox{\box#2}}
\faa {好的} to 3cm{好\hfil 的}\quad \fbb {好的} to 3cm{好\hfil 的}
```

好的 好 的 好的 好 的

代码 10

```
\DeclareEKeysCommand \faa { c{w{3em}} c{v{3em}} } {\fbox{\box#1} \fbox{\box#2}}
\DeclareEKeysCommand \fbb { c{w[c]{3em}} c{v[c]{3em}} } {\fbox{\box#1} \fbox{\box#2}}
\DeclareEKeysCommand \fcc { c{w[c][9ex]{3em}} c{v[c][9ex]{3em}} }
  {\fbox{\box#1} \fbox{\box#2}}
\DeclareEKeysCommand \fdd { c{w[c][9ex][t]{3em}} c{v[c][9ex][t]{3em}} }
  {\fbox{\box#1} \fbox{\box#2}}
基准。
\faa {好的\par 吗？} {不好\par 吗？}
\fbb {好的\par 吗？} {不好\par 吗？}
\fcc {好的\par 吗？} {不好\par 吗？}
\fdd {好的\par 吗？} {不好\par 吗？}
```

[9] 这五类寄存器，都有相应的 `\`⟨*type*⟩`def` 原语。因此在 ⟨*allocated collect spec*⟩ 中可根据 ⟨*register*⟩ 自动判断保存的类型。对于这 5 种寄存器，`C` 参数类型只需给出 ⟨*register*⟩ 即可。如 `C\@tempdima` 或 `C\l_tmpa_dim`。

[10] 对于盒子来说，并没有 `\boxdef` 原语，分配新的盒子寄存器是用的 `\chardef` 或 `\mathchardef`。因此，在 ⟨*allocated collect spec*⟩ 是根据 ⟨*register*⟩ 是否为 `\chardef` 或 `\mathchardef` token 来判断是否是保存到盒子的。

基准。 好的吗? 不好吗? 好的吗? 不好吗? 好的吗? 不好吗? 好的吗? 不好吗?

§ 12 可自定义参数获取方式的类型——K

如果上面的小节介绍的参数获取方式还不能满足你的需要，*ekeys-cmd* 还支持自定义参数获取方式——就是使用 K 类型的参数。K 的用法为 K{⟨*scanner-and-args*⟩}，其中 ⟨*scanner-and-args*⟩ 为

- {⟨*scanner*⟩}⟨*args*⟩，⟨*scanner*⟩ 带有括号，则它的参数可以不带有括号；
- ⟨*scanner*⟩{⟨arg_1⟩}⟨*extra args*⟩，带有一个或多个参数，若 ⟨*scanner*⟩ 不带括号，则第一个参数必须有括号；
- ⟨*scanner*⟩，不带任何参数的扫描器（“扫描器”就是自定义的参数获取方式）；

以纯文字描述就是：移除 ⟨*scanner-and-args*⟩ 的首尾空格后，若它以一对 { } 开始，则这个花括号里的内容就是 scanner，之后的内容移除掉两端的空格后就是额外的参数；否则，就是 scanner 就是第一个花括号之前的内容（移除掉两端的空格），额外的参数就是第一个花括号及其之后的记号；否则，没有任何花括号，就只有 scanner，没有参数。如若 ⟨*scanner-and-args*⟩ 为 ␣{a␣b␣}␣[v]␣j␣，则 scanner 为 a␣b␣，参数为 [v]␣j；若 ⟨*scanner-and-args*⟩ 为 ␣fo␣o␣{da}␣[b␣]␣，则 scanner 为 fo␣o，参数为 {da}␣[b␣]。

有几个预定义的扫描器：

- norelax，移除一个 \relax，在向后寻找这个 \relax 时忽略空格；
- norelax!，移除一个 \relax，在向后寻找这个 \relax 时不忽略空格；
- ?⟨*preprocessor spec*⟩，参数预处理器。⟨*preprocessor spec*⟩ 用法为：
  - {⟨*ekeys-cmd arg spec*⟩}，使用 ⟨*ekeys-cmd arg spec*⟩ 获取后面的参数，转化为带花括号的标准参数；
  - {⟨*ekeys-cmd arg spec*⟩}{⟨*scanner code*⟩}，使用 ⟨*ekeys-cmd arg spec*⟩ 获取后面的参数，并用 ⟨*scanner code*⟩ 替换这些参数。⟨*scanner code*⟩ 必须包含 \ekeys_cmd_scanner_end:，且在它后面的内容会被 *ekeys-cmd* 重新读取；
  - {⟨*ekeys-cmd arg spec*⟩}{⟨*scanner code*⟩}⟨*text*⟩，同上，⟨*text*⟩ 会放在要读取的参数之前。
- u{⟨*alias*⟩}，使用由 \ekeysnewscanneralias 设置的别名。
- define⟨*define spec*⟩，它用来在定义 *ekeys-cmd* 时，进一步的自定义参数获取方式。⟨*define spec*⟩ 用法为：
  - {⟨*definition*⟩}，它不占用任何参数。主要用作向后检查，或展开，也可以向输入中添加额外的内容。需要在 ⟨*definition*⟩ 的合适位置添加 \ekeys_cmd_add_args:n 和 \ekeys_cmd_scanner_end:；
  - {⟨*numbers*⟩}{⟨*parameters*⟩}，它占用 ⟨*numbers*⟩ 个参数。根据 ⟨*parameters*⟩ 向后获取参数。⟨*parameters*⟩ 的参数数量不能少于 ⟨*numbers*⟩。这是 \def 的 *ekeys-cmd* 接口；
  - {⟨*numbers*⟩}{⟨*parameters*⟩}{⟨*definition*⟩}，同上。注意：当 *numbers* ≥ 0 时，会在 ⟨*definition*⟩ 后面自动加上 \ekeys_cmd_add_args:n 和 \ekeys_cmd_scanner_end:；
  - {*⟨*numbers*⟩}{⟨*parameters*⟩}{⟨*definition*⟩}，⟨*numbers*⟩ 与 ⟨*parameters*⟩ 没有关系。但需要在 ⟨*definition*⟩ 的合适位置添加 \ekeys_cmd_add_args:n 和 \ekeys_cmd_scanner_end:，且添加的参数必须和 ⟨*numbers*⟩ 一致。
- lohi⟨*defaults*⟩，它获取一组数学上下标，占用 2 个参数，第一个为下标，第二个为上标。如果不存在，则为使用默认值。设置默认值是通过给扫描器的额外的参数来设置。其中 ⟨*defaults*⟩ 为：
  - {⟨$default_{lo}$⟩}{⟨$default_{hi}$⟩}，设置下标和上标的默认值；
  - {⟨*default*⟩}，设置上下标的默认值；
  - 空，表示上下标使用特殊的标记：-NoValue-。

  它支持 & 修饰符，即可以自动添加 _ 和 ^，但默认值不会自动添加这两个符号。

本节的这些都归类为 *ekeys-cmd* 的参数扫描器，预处理器也是一类特殊的参数扫描器。参数扫描器是完全可自定义的，所有自定义的参数处理器都需要执行 \ekeys_cmd_scanner_end: 或与之等价的 \EKeysEndPreprocessor，例如在 ⟨*preprocessor spec*⟩ 或 ⟨*definition*⟩ 中。

接下来的两个小节会详细介绍参数扫描器。

§ 13 自定义参数扫描方式

先来介绍 lohi 这个预定义的 scanner。它用来获取一组数学上下标。在找寻上下标时，会自动展开和移除空格和 \relax （即自动移除 ⟨filler⟩），它并不需要用于数学模式，但是严格匹配的，只会匹配类别码为 7 或 8 的字符（显式或隐式的）。可以为它设置默认值。

代码 11

```
\catcode`\^=7 \catcode`\_=8
\def\temp{\relax _ \space\relax{ij}}
\DeclareEKeysCommand \faa { K{lohi} } {\detokenize{[#1|#2]}}
\DeclareEKeysCommand \fbb { K{lohi{n}} } {\detokenize{[#1|#2]}}
\DeclareEKeysCommand \fcc { K{lohi{m}{n}} } {\detokenize{[#1|#2]}}
\DeclareEKeysCommand \fmm { &p{\limits\nolimits} &K{lohi{_m}{^n}} } {\sum#1#2#3}
\obeylines
\faa ; \faa _a^b; \faa _a; \faa ^b; \faa\temp;
\fbb ; \fbb _a^b; \faa _a; \fbb ^b; \fbb\temp;
\fcc ; \fcc _a^b; \fcc _a; \fcc ^b; \fcc\temp;
$ \fmm ; \fmm _a^b; \fmm _a; \fmm ^b; \fmm\temp; \fmm\limits\temp $
```

[-NoValue-|-NoValue-]; [a|b]; [a|-NoValue-]; [-NoValue-|b]; [ij|-NoValue-];
[n|n]; [a|b]; [a|-NoValue-]; [n|b]; [ij|n];
[m|n]; [a|b]; [a|n]; [m|b]; [ij|n];
$\sum_{m}^{n}$; $\sum_{a}^{b}$; $\sum_{a}^{n}$; $\sum_{m}^{b}$; $\sum_{ij}^{n}$; $\sum\limits_{ij}^{n}$

\ekeysnewscanneralias 可以为扫描器全局地设置别名，在 u 中使用。不带中间的可选参数时，可以为 scanner 设置别名；带中间的可选参数时，可以为任意参数类型设置别名，但需写出这个类型。

代码 12

```
\catcode`\^=7 \catcode`\_=8
\ekeysnewscanneralias{lohi-xy}{lohi{x}{y}} % scanner
\ekeysnewscanneralias{lohi-K-xy}[K]{ & K{lohi{_x}{^y}} } % 任意说明符/修饰符
\DeclareEKeysCommand \fcc { K{u{lohi-xy}} } {\detokenize{[#1|#2]}}
\DeclareEKeysCommand \fmm { K{u{lohi-K-xy}} } {\sum#1#2}
\fcc ; \fcc _a^b; \fcc _a; \fcc ^b;
$ \fmm ; \fmm _a^b; \fmm _a; \fmm ^b; $
```

[x|y]; [a|b]; [a|y]; [x|b]; $\sum_{x}^{y}$; $\sum_{a}^{b}$; $\sum_{a}^{y}$; $\sum_{x}^{b}$;

```
\ekeys_cmd_new_scanner:nnnpn {⟨scanner⟩} {⟨argument number⟩} {⟨initial action⟩} ⟨parameter list⟩ {⟨scanner action⟩}
\ekeys_cmd_new_scanner:nnpn {⟨scanner⟩} {⟨argument number⟩} ⟨parameter list⟩ {⟨scanner action⟩}
\ekeys_cmd_add_args:n { {⟨arg_1⟩} {⟨arg_2⟩} ... {⟨arg_n⟩} }
\ekeys_cmd_add_arg:n {⟨arg⟩}
\ekeys_cmd_scanner_end:
```

\ekeys_cmd_new_scanner:nnnpn 用于定义一个参数扫描器。⟨*scanner action*⟩ 中必须执行 \ekeys_cmd_scanner_end:，在它后面的内容会被此扫描器之后的参数（或扫描器）读取。在 ⟨*scanner action*⟩ 中用 \ekeys_cmd_add_args:n 和 \ekeys_cmd_add_arg:n 来给 *ekeys-cmd* 添加参数。还可在 ⟨*initial action*⟩ 中设置 \l_ekeys_cmd_scanner_args_int 来修改 ⟨*parameter number*⟩。添加的参数数目必须和该整数值一致。

代码 13

```
\ExplSyntaxOn
\ekeys_cmd_new_scanner:nnpn { my/scan-bra-ket } { 2 } <#1|#2>
  {
    \ekeys_cmd_add_args:n { {#1} {#2} }
    \ekeys_cmd_scanner_end:
  }
```

```
\ExplSyntaxOff
\DeclareEKeysCommand \foo { K{my/scan-bra-ket} } {\left<#1\middle|#2\right>}
$ \foo<a|b> $\quad $ \foo<\sum|\prod> $
```

$\langle a|b\rangle$ $\langle \sum|\prod\rangle$

针对上面这种特别简单的情况，可以直接使用 `define` scanner：

代码 14

```
\ExplSyntaxOn
\DeclareEKeysCommand \foo { K{ define {2} {<#1|#2>} { } } } {\left<#1\middle|#2\right>}
\ExplSyntaxOff
$ \foo<a|b> $\quad $ \foo<\sum|\prod> $
```

甚至，由于 ⟨*definition*⟩ 为空，还可以直接不写：

代码 15

```
\ExplSyntaxOn
\DeclareEKeysCommand \foo { K{ define {2} {<#1|#2>} } } {\left<#1\middle|#2\right>}
\ExplSyntaxOff
$ \foo<a|b> $\quad $ \foo<\sum|\prod> $
```

这是由于，对于上面这两种情况，会在 ⟨*definition*⟩ 后面自动附加合适的 `\ekeys_cmd_add_args:n` 和 `\ekeys_cmd_scanner_end:`。如果某些情况下，不需要自动添加的 `\ekeys_cmd_add_args:n` 和 `\ekeys_cmd_scanner_end:` 该怎么办呢？答案是在参数数字前面加上 `*`：

代码 16

```
\ExplSyntaxOn
\DeclareEKeysCommand \foo {
  K{ define {*2} {<#1|#2>} { \ekeys_cmd_add_args:n {{#1}{#2}}\ekeys_cmd_scanner_end: } }
} {\left<#1\middle|#2\right>}
\ExplSyntaxOff
$ \foo<a|b> $\quad $ \foo<\sum|\prod> $
```

当参数数目设置为 -1 时，也不会自动添加 `\ekeys_cmd_add_args:n` 和 `\ekeys_cmd_scanner_end:`。

以上是一些简单的情形，实际上参数扫描器还可以定义得更加复杂。更多的例子见第 § 15 节。

参数扫描器除了可以添加参数，还可以用作检查后面的内容并进行必要的处理。针对这一类特殊的扫描器，称之为参数预处理器，这是下一节要介绍的。

§ 14 参数预处理器

document-cmd 有“参数处理器”，而 *ekeys-cmd* 有“参数预处理器”。“处理器”和“预处理器”的区别是，前者先按参数说明符的规则获取参数后再对其进行处理，把结果作为实参；而后者是对未读取的内容先一步进行处理，然后再根据参数说明符的规则获取参数作为实参。

假设我们想查看后面的那个记号是不是 `\relax`，借助 `\peek_meaning_remove:NTF` 可以这么做：

代码 17

```
\ExplSyntaxOn
\DeclareEKeysCommand \foo {
  K{ define {
      \peek_meaning_remove:NTF \scan_stop:
        { \ekeys_cmd_scanner_end: { is-relax } }
        { \ekeys_cmd_scanner_end: { is-not-relax } }
    }
  } m u;
} { #1. \tl_to_str:n {#2}. }
\ExplSyntaxOff \ttfamily
\foo \relax; \foo a; \foo {\relax};
```

```
is-relax.. is-not-relax.a. is-not-relax.\relax .
```

可以看到，在获取第一个参数之前，我们根据后面是不是跟着 \relax 来让它获得不同的值。

这是预处理器不需要获取参数的情况，倘若需要获取参数，仍然可以用 define scanner：

代码 18

```
\ExplSyntaxOn
\DeclareEKeysCommand \foo {
  K{ define {-1} { #1#2 } {
      \tl_if_eq:nnTF {#1} {#2}
        { \ekeys_cmd_scanner_end: \BooleanTrue }
        { \ekeys_cmd_scanner_end: \BooleanFalse }
    }
  } m
} { \IfBooleanTF {#1} { eq } { neq } }
\ExplSyntaxOff
\foo {a}{b}; \foo {a}{a};
```

```
neq; eq;
```

define{*0}{#1#2}... 也是可行的。也可直接使用 \ekeys_cmd_new_scanner:nnnpn。

使用上面这种方式定义的预处理器通常比较简短，获取参数的方式也比较有限，若要获取更加复杂参数，则可以使用 ? scanner 和 ? 预处理器指示符。它们的区别是，预处理器指示符和参数类型的地位等同，而 ? 参数扫描器则是和 define 扫描器等同，且它们的具体用法也不一致。

? 预处理器指示符的用法 ?{⟨*preprocessor action*⟩}。⟨*preprocessor action*⟩ 通常是一个命令，它的定义中有一个 \ekeys_cmd_scanner_end:。

代码 19

```
\ExplSyntaxOn
% 它用来合并 () 和 [] 的键值选项
\DeclareEKeysCommand \foopreprocessor { m @w{ () [] } }
  {
    \tl_if_novalue:nTF {#2}
      {
        \tl_if_novalue:nTF {#3}
          { \ekeys_cmd_scanner_end: { } }
          { \ekeys_cmd_scanner_end: {#3} }
      }
      {
        \tl_if_novalue:nTF {#3}
          { \ekeys_cmd_scanner_end: {#2} }
          { \ekeys_cmd_scanner_end: { #2 #1 #3 } }
      }
  }
\ExplSyntaxOff
\DeclareEKeysCommand \foo { ?{\foopreprocessor{,}} m } {\detokenize{#1}}
\foo [a=b,b=k] (c=d,e=f); \foo ;
```

```
c=d,e=f,a=b,b=k; ;
```

对于上面预处理器命令中的 \ekeys_cmd_scanner_end: 可以替换为 \EKeysEndPreprocessor，它们的区别是，前者必须用在参数扫描器的定义中，而后者若没有用在参数扫描器中，不会报错且 f-展开为空。

上例可以转换为 ? scanner：

代码 20

```
\def\mergenovalue#1#2#3{%
  \IfNoValueTF{#2}{\IfNoValueTF{#3}{#1{}}{#1{#3}}}
    {\IfNoValueTF{#3}{#1{#2}}{#1{#2,#3}}}}
```

```
\DeclareEKeysCommand \foo {
  K{ ?{ @w{ () [] } }{\mergenovalue\EKeysEndPreprocessor{#1}{#2}} } m
} {\detokenize{#1}}
\foo [a=b,b=k] (c=d,e=f); \foo ;
```

c=d,e=f,a=b,b=k; ;

若 ? scanner 没有给出 ⟨*preprocessor spec*⟩ 没有给出 ⟨*scanner code*⟩，则会按照给定的参数说明符获取参数，并将其转化为带花括号的标准参数。

代码 21

```
\DeclareEKeysCommand \foo { K{?{ @w{[] ()} }} m m }{\detokenize{{#1}{#2}}}
\foo (brace(b)) [bracket[B]]; \foo ;
```

{bracket[B]}{brace(b)}; {-NoValue-}{-NoValue-};

当然，参数扫描器和预处理器能做的还远不止于此，例如，使用使用 LaTeX3 的正则表达式库，可以对参数进行更加细致的处理，使用 \peek_regex:NTF、\peek_regex_replace_once:NnTF 等可以对参数进行更加细致的检测。

下一小节是一些例子。它们使用了 collectn 宏包，它的用法可以参考文末附上的用法说明。

§ 15 *ekeys-cmd* 诸例

尽管 c 和 C 不能引用其它参数，但可以间接做到这一点。并且，由于 K 和预处理器可以嵌套使用，在处理参数时是非常灵活的。

代码 22

```
\ekeysdeclarecmd \faa {
  K{ ? {G{{3cm}}} {\def\footmpa{#1}\EKeysEndPreprocessor} }
  c{ w\footmpa }
} {\fbox{\box#1}}
\faa \relax {你好\\ 吗？ } ! % 这个 \relax 用来阻止 G 参数
%
\ekeysdeclarecmd \fbb {
  K{ ?{
       K{ ?{O{t} O{9ex} G{2cm}}{\EKeysEndPreprocessor{[#1][#2]{#3}}} } G{{3cm}}
     }{\def\footmpa{#1}\EKeysEndPreprocessor}
  }
  c { w\footmpa }
}{\fbox{\box#1}}
\fbb [c]\relax {你好\\ 吗？}!
% 这个 \relax 用来阻止内层的 G 参数，而外层的 G 参数已经由内层的预处理器给出了
```

你好
吗?
!
你好
吗?
!

嵌套使用 K 参数时，需要注意使用的 \ekeys_cmd_scanner_end: 次数要和需要的一致。

使用 K 类型的参数可以做到和 \def 定义宏一样的效果。

代码 23

```
\ekeysnewscanneralias{math-style}[p]
  {p{\displaystyle\textstyle\scriptstyle\scriptscriptstyle}}
\ekeysnewscanneralias{scan-bra-ket}{define{2}{<#1|#2>}}

\DeclareEKeysCommand \foo { &K{u{math-style}} K{u{scan-bra-ket}} }
  {{#1\left<#2\middle|#3\right>}}
$ \foo<a|b> $\quad $ \foo\displaystyle<\sum|\prod> $.
```

$\langle a|b\rangle$ $\langle\Sigma|\prod\rangle$.

等价的方法是：

代码 24

```
\ExplSyntaxOn
\ekeys_cmd_new_scanner:nnpn { my/scan-bra-ket } { 2 } <#1|#2>
  {
    \ekeys_cmd_add_args:n { {#1} {#2} }
    \ekeys_cmd_scanner_end:
  }
\ExplSyntaxOff
\DeclareEKeysCommand \foo
  { &p{\displaystyle\textstyle\scriptstyle\scriptscriptstyle} K{my/scan-bra-ket} }
  {{#1\left<#2\middle|#3\right>}}
$ \foo<a|b> $\quad $ \foo\displaystyle<\sum|\prod> $.
```

有一类原语，它们可以通过关键字的出现与否来实现不同的行为，如 \special：

\special [shipout] ⟨general text⟩

⟨general text⟩ ⟶ ⟨filler⟩{⟨balanced text⟩⟨right brace⟩

⟨filler⟩ ⟶ ⟨optional spaces⟩|⟨filler⟩\relax⟨filler⟩

⟨optional spaces⟩ ⟶ ⟨empty⟩|⟨space token⟩⟨optional spaces⟩

其中，这 shipout 是忽略大小写和类别码的，而且在检测关键字是否出现时会自动展开后面的记号。\collectn_scan_keyword:NTF 为我们提供了此功能。

代码 25

```
\ExplSyntaxOn
\newtoks \l__my_scan_nos_toks
\collectn_set_keyword:Nn \c__my_scan_nos_tl { shipout }
\ekeys_cmd_new_scanner:nnpn { my/scan-nos } { 2 }
  {
    \collectn_scan_keyword:NTF \c__my_scan_nos_tl
      {
        \ekeys_cmd_add_arg:n { \BooleanTrue }
        \__my_scan_nos_next:
      }
      {
        \ekeys_cmd_add_arg:n { \BooleanFalse }
        \__my_scan_nos_next:
      }
    % 不可将 \__my_scan_nos_next: 放到这里，
    % 否则，\collectn_scan_keyword:NTF 将会扫描到 \__my_scan_nos_next:
  }
\cs_new:Npn \__my_scan_nos_next:
  {
    \collectn_value:Nnw \l__my_scan_nos_toks
      {
        \ekeys_cmd_add_arg:V \l__my_scan_nos_toks
        \ekeys_cmd_scanner_end: % <- 扫描器完成时必须使用它
      } = % 加上一个等号，这样后面就不能再出现等号了
  }
\ExplSyntaxOff

\DeclareEKeysCommand \foo { K{my/scan-nos} }
  {Is\IfBooleanF{#1}{ not} shipout, text: [#2]}
\foo {special code} \quad
```

```
\foo ShipouT {special code} \quad
\foo shipout\relax\bgroup special code}
```

Is not shipout, text: [special code] Is shipout, text: [special code] Is shipout, text: [special code]

不过，当字符的类别码为 1、2、10 之一时，即使它的字符码匹配，`\collectn_scan_keyword:NTF` 系列命令也无法判定它为匹配的关键字，但 TEX 原语却能识别和匹配。同时，类别码为 13 的字符无法构成关键字。

倘若我们想要模拟 `\vrule` 和 `\hrule` 这样的命令，它们的语法规则是这样的：

⟨vertical rule⟩ ⟶ `\vrule`⟨rule specification⟩

⟨horizontal rule⟩ ⟶ `\hrule`⟨rule specification⟩

⟨rule specification⟩ ⟶ ⟨optional spaces⟩|⟨rule dimension⟩⟨rule specification⟩

⟨rule dimension⟩ ⟶ `width`⟨dimen⟩|`height`⟨dimen⟩|`depth`⟨dimen⟩

像 `width` 这个关键词，它是忽略大小写和类别码的。而且 ⟨rule dimension⟩ 需要匹配多个无序的关键词。`\collectn_scan_keywords:NTF` 正好为我们提供了对应功能。至于 ⟨dimen⟩ 只需使用 `\collectn_value:Nnw` 即可获取。

下例定义了一个参数扫描器 `my/scan-whd`，用于捕获以关键字 `width`、`height`、`depth` 引导的长度。

代码 26

```
\ExplSyntaxOn
\dim_new:N \l__my_scan_whd_dim % 临时保存长度
\seq_new:N \l__my_scan_whd_seq % 保存三个值，width,height,depth
\cs_generate_variant:Nn \seq_set_item:Nnn { NnV }
\collectn_set_keywords:Nn \c__my_scan_whd_tl { width, height, depth }
% 3 个参数，为 width, height, depth
\ekeys_cmd_new_scanner:nnpn { my/scan-whd } { 3 }
  {
    % initial seq
    \seq_clear:N \l__my_seq_whd_seq
    \seq_put_right:NV \l__my_seq_whd_seq \c_novalue_tl
    \seq_put_right:NV \l__my_seq_whd_seq \c_novalue_tl
    \seq_put_right:NV \l__my_seq_whd_seq \c_novalue_tl
    \__my_scan_whd_next:
  }
\cs_new:Npn \__my_scan_whd_next:
  {
    \collectn_scan_keywords:NTF \c__my_scan_whd_tl
      {
        \collectn_value:Nnw \l__my_scan_whd_dim
          {
            \seq_set_item:NnV \l__my_seq_whd_seq
              { \l_collectn_keywords_int } \l__my_scan_whd_dim
            \__my_scan_whd_next:
          } = % 加上一个等号，这样后面就不能再出现等号了
      }
      {
        \ekeys_cmd_add_args:e
          { \seq_map_function:NN \l__my_seq_whd_seq \ekeys_exp_not_braced:n }
        \ekeys_cmd_scanner_end:
      }
  }
\ExplSyntaxOff

\DeclareEKeysCommand \foo { K{my/scan-whd} }
  {[ W: \IfValueTF{#1}{#1}{-}; H: \IfValueTF{#2}{#2}{-}; D: \IfValueTF{#3}{#3}{-} ]}
\ttfamily
```

```
\foo height 3pt depth \dimeval{3pt+2pt-10pt} height \dimexpr 3pt-2pt wide
```

```
[ W: -; H: 1.0pt; D: -5.0pt ]wide
```

表 1: pdfTEX、XETEX、LuaTEX可用的类别码及其含义

类别	含义	Meaning	常见值	类别	含义	Meaning	常见值
0	转义字符	Escape character	\	8	下标	Subscript	_
1	组开始	Beginning of group	{	9	忽略的字符	Ignored character	⟨null⟩
2	组结束	End of group	}	10	空格	Space	␣
3	数学切换	Math shift	$	11	字母	Letter	26 个英文字母
4	表对齐	Alignment tab	\&	12	其它字符	Other character	其它
5	行结束	End of line	⟨return⟩	13	活动字符	Active character	~
6	变量	Parameter	#	14	注释字符	Comment character	%
7	上标	Superscript	^	15	无效字符	Invalid character	⟨delete⟩

注：在 XELATEX 和 LUALATEX下，使用 xeCJK 或 luatexja 宏包，中日韩字符的类别码也是 11。

§ 16 用法说明

\collectn_verb:Nnw
\l_collectn_long_verbatim_bool

```
\collectn_verbatim:Nnw ⟨tl⟩ {⟨code⟩} ⟨token⟩ ⟨tokens⟩ ⟨token⟩
\collectn_verbatim:Nnw ⟨tl⟩ {⟨code⟩} {⟨balanced tokens⟩}
```

向后以 verbatim 的形式收集内容，将其保存至 ⟨*tl*⟩ 中，再执行 ⟨*code*⟩。⟨*token*⟩ 或 ⟨{*balanced tokens*}⟩ 不能在命令的参数里。

\l_collectn_long_verbatim_bool P.87 控制是否接受长的 verbatim 文本（包含 \par 的）。

在 ⟨*tokens*⟩ 或 ⟨*balanced tokens*⟩ 中，字母的类别码不能为 1、2、10。

结果 ⟨*tl*⟩ 包含的字符其类别码有三种，为 10、12、13（在 (u)pTEX 引擎下，仅 Unicode 编码小于等于 255 的字符有此特性），且 ^^M 的类别码为 13。

\collectn_varwidth:nnw
\collectn_varwidth_end:

```
\collectn_varwidth:nnw {⟨vertical pos⟩} {⟨maximum width⟩}
    ⟨contents⟩ \collectn_varwidth_end:
```

⟨*contents*⟩ 是可变宽的，最大宽度为 ⟨*maximum width*⟩；基线的位置为 ⟨*vertical pos*⟩，可选值为 b、c、t，分别表示对齐底部基线、居中对齐、对齐顶部基线。

该命令是对 varwidth 环境的封装。

\collectn_set_varwidth:Nnnw
\collectn_set_varwidth_end:
\collectn_gset_varwidth:Nnnw
\collectn_gset_varwidth_end:

```
\collectn_set_varwidth:Nnnw ⟨box⟩ {⟨vertical pos⟩} {⟨maximum width⟩}
    ⟨contents⟩ \collectn_set_varwidth_end:
```

设置 ⟨*box*⟩ 为一个包含 ⟨*contents*⟩ 的最大宽度为 ⟨*maximum width*⟩ 的水平盒子。

\collectn_set_vbox_width:Nnw
\collectn_set_vbox_width_end:
\collectn_set_vbox_varwidth:Nnw
\collectn_set_vbox_varwidth_end:
\collectn_gset_vbox_width:Nnw
\collectn_gset_vbox_width_end:
\collectn_gset_vbox_varwidth:Nnw
\collectn_gset_vbox_varwidth_end:

```
\collectn_set_vbox_width:Nnw ⟨box⟩ {⟨width⟩}
    ⟨content⟩ \collectn_set_vbox_width_end:
```

\collectn_set_vbox_width:Nnw P.87 与 \vbox_set:Nw 类似，把 ⟨*content*⟩ 保存到 vbox ⟨*box*⟩ 中，该 ⟨*box*⟩ 的宽度为 ⟨*width*⟩。它和 minipage 环境相似。

\collectn_set_vbox_varwidth:Nnw P.87 则是设置 ⟨*content*⟩ 的最大宽度为 ⟨*width*⟩。它和 varwidth 环境相似。

这 ⟨*box*⟩ 可以使用 \vsplit 或 \vbox_set_split_to_ht:NNn 来分割。

\collectn_box:Nnw

```
\collectn_box:NNnw ⟨box⟩ ⟨primitive box cs⟩ {⟨code⟩} ⟨material⟩
```

先将 ⟨*material*⟩ 保存到 ⟨*box*⟩ 中，此盒子为 ⟨*primitive box cs*⟩，可为 \hbox、\vbox、\vtop 之一。之后再使用 ⟨*code*⟩ 处理。

⟨*material*⟩ 可以是 {⟨horizontal/vertical mode material⟩}，也可以有 ⟨box specification⟩。左右括号可以是隐式的。

这种方式与 \peek_charcode_remove:NTF 类似。另见 collectbox 宏包。

⟨*code*⟩ 与 \collectn_box:Nnw P.87 在同一个组中执行。

注意：\hbox:n、\vbox:n、\vbox_top:n 并不分别等于 \hbox、\vbox、\vtop。

例 72

```
\ExplSyntaxOn
\cs_set:Npn \myfbox
  {
    \collectn_box:NNnw \l_tmpa_box \tex_hbox:D
      { \fbox { \box_use_drop:N \l_tmpa_box } }
  }
\ExplSyntaxOff
\myfbox{\verb|\myfbox| 与 \verb|\fbox|}
\myfbox{\parbox{3cm}{可分段的\par fbox}}
```

\myfbox 与 \fbox | 可分段的 fbox

另见例 86。

\collectn_value:Nnw

\collectn_value:Nnw ⟨*register*⟩ {⟨*code*⟩} ⟨*value*⟩

将 ⟨*value*⟩ 保存到 ⟨*register*⟩ 中，再使用 ⟨*code*⟩ 处理。⟨*value*⟩ 必须确实可以保存到 ⟨*register*⟩ 中。

⟨*code*⟩ 与此命令在同一个组中执行。

例 73

```
\ExplSyntaxOn
\cs_set:Npn \showintval
  { \collectn_value:Nnw \l_tmpa_int { 值为 $ \int_use:N \l_tmpa_int $ } }
\ExplSyntaxOff
\showintval 3
\showintval -310
\showintval "3FA
\showintval \shellescape
```

值为 3 值为 −310 值为 1018 值为 2

\collectn_width:Nnnw
\collectn_minipage:Nnnw
\collectn_varwidth:Nnnw

\collectn_width:Nnnw ⟨*box*⟩ {⟨*code*⟩} {⟨*width*⟩} ⟨*material*⟩

先使用类似于 minipage（varwidth）的处理方式处理 ⟨*material*⟩，然后将它保存到 vbox ⟨*box*⟩ 中，再使用 ⟨*code*⟩ 处理。这 ⟨*box*⟩ 的宽度（或最大宽度）为 ⟨*width*⟩。

⟨*material*⟩ 可以是 {⟨vertical mode material⟩}，正如 \vbox {⟨vertical mode material⟩} 那样。左右括号可以是隐式的。

⟨*code*⟩ 与这些命令在同一个组中执行。

\collectn_minipage:Nnnnw P.88 是 \collectn_width:Nnnnw P.88 的另一个名字。

保存的垂直盒子可以用 \vsplit 或 \vbox_set_split_to_ht:NNn 来分割。

\collectn_width:Nnnnw
\collectn_minipage:Nnnnw
\collectn_varwidth:Nnnnw

\collectn_width:Nnnnw ⟨*box*⟩ {⟨*code*⟩} {⟨*vpos*⟩} {⟨*width*⟩} ⟨*material*⟩

先使用类似于 minipage（varwidth）的处理方式处理 ⟨*material*⟩，然后将它保存到 hbox ⟨*box*⟩ 中，再使用 ⟨*code*⟩ 处理。这 ⟨*box*⟩ 的宽度为 ⟨*width*⟩，基线的位置为 ⟨*vpos*⟩，可选值为 b、c、t，分别表示底部基线、居中、顶部基线，默认为底部基线。

⟨*material*⟩ 可以是 {⟨vertical mode material⟩}，正如 \vbox {⟨vertical mode material⟩} 那样。左右括号可以是隐式的。

⟨*code*⟩ 与这些命令在同一个组中执行。

\collectn_minipage:Nnnnw P.88 是 \collectn_width:Nnnnw P.88 的另一个名字。

例 74

```
\ExplSyntaxOn
\cs_set:Npn \myparfbox #1#2
  {
    \collectn_width:Nnnnw \l_tmpa_box
      { \fbox { \box_use_drop:N \l_tmpa_box } } {#1} {#2}
  }
```

```
\cs_set:Npn \myvarfbox #1#2
  {
    \collectn_varwidth:Nnnnw \l_tmpa_box
      { \fbox { \box_use_drop:N \l_tmpa_box } } {#1} {#2}
  }
\ExplSyntaxOff
\myparfbox{t}{5cm}{一个可以包含 \verb|\verb| 的\par
  定宽 \verb|\fbox|}
\myvarfbox{b}{5cm}{一个可以包含 \verb|\verb| 的\par
  变宽 \verb|\fbox|}
```

一个可以包含 \verb 的
定宽 \fbox

一个可以包含 \verb 的
变宽 \fbox

\collectn_width:Nnnnnnw ⟨*box*⟩ {⟨*code*⟩}
{⟨*vpos*⟩} {⟨*height*⟩} {⟨*inner pos*⟩} {⟨*width*⟩} ⟨*material*⟩

\collectn_width:Nnnnnnw
\collectn_minipage:Nnnnnnw
\collectn_varwidth:Nnnnnnw

前述命令的完整形式。

- ⟨*vpos*⟩ 表示垂直位置，可选值为 b、c、t，分别表示对齐底部基线、居中对齐、对齐顶部基线，默认为对齐底部基线；
- ⟨*height*⟩ 表示盒子的高度，如果不设置，则为盒子的自然高度，
- ⟨*inner pos*⟩ 表示如果设置的 ⟨*height*⟩ 过大时，盒子的内容在盒子内的垂直位置，可选值为 b、c、t、s，分别表示置于盒子底部、居中、置于顶部、垂直分散对齐，默认为垂直分散对齐。

⟨*material*⟩ 可以是 {⟨vertical mode material⟩}，正如 \vbox {⟨vertical mode material⟩} 那样。左右括号可以是隐式的。

⟨*code*⟩ 与这些命令在同一个组中执行。

\collectn_hbox_auto:Nnnw ⟨*box*⟩ {⟨*code*⟩} {⟨*spec*⟩} {⟨*content*⟩}

\collectn_hbox_auto:Nnnw
\collectn_width_auto:Nnnw
\collectn_minipage_auto:Nnnw
\collectn_varwidth_auto:Nnnw

根据 ⟨*spec*⟩ 自动使用合适的命令。

\collectn_hbox_auto:Nnnw P89 的 ⟨*spec*⟩ 为 \makebox 的前两个参数：[⟨*width*⟩][⟨*pos*⟩]。

其它三个命令的 ⟨*spec*⟩ 为 \parbox 的前四个参数：[⟨*vpos*⟩][⟨*height*⟩][⟨*inner pos*⟩]{⟨*width*⟩}。仅当只有 ⟨*width*⟩ 给出时，所保存的盒子为垂直盒子，且可以使用 \vsplit 或 \vbox_set_split_to_ht:NNn 来分割该盒子。

\collectn_if_in:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

\collectn_if_in:n*TF*

判断下一个记号是否在给定的 ⟨*token list*⟩ 中。

无法判断显式或隐式的左、右括号和空格。

\collectn_delimited_inline:NNn ⟨$token_1$⟩ ⟨$token_2$⟩ {⟨*inline code*⟩}
\collectn_delimited_variable:NNNn ⟨$token_1$⟩ ⟨$token_2$⟩ ⟨*tl*⟩ {⟨*code*⟩}

\collectn_delimited_inline:NNn
\collectn_delimited_variable:NNNn

获取接下来的以 ⟨$token_1$⟩ 和 ⟨$token_2$⟩ 定界的参数。若不存在，则为 \q_no_value。⟨*inline code*⟩ 可以用 #1 获取这个参数，⟨*tl*⟩ 用于保存这个参数。

\collectn_delimited_any_inline:nn

\collectn_delimited_any_inline:nn {⟨paired tokens⟩} {⟨inline code⟩}

判断接下来的定界的参数的定界符是否在 ⟨paried tokens⟩ 之中。⟨inline code⟩ 可使用 2 个参数，第一个为此定界符在 ⟨paired tokens⟩ 中的位置，第二个为它的值。若不存在，则第一个值为 0，第二个值为 \q_no_value。

\collectn_delimited_all_inline:nn

\collectn_delimited_all_inline:nn {⟨paried tokens⟩} {⟨inline code⟩}

对于 ⟨paried tokens⟩ 给出的定界符对，逐一考察是否接下来的参数以此定界符对定界。

\l_collectn_delimited_strip_bool

用于控制上面四个命令在获取定界的参数时，若其参数整个用 {} 包裹，是否移除此 {}。

例 75

```
\ExplSyntaxOn \ttfamily
\bool_set_false:N \l_collectn_delimited_strip_bool
\collectn_delimited_inline:NNn [ ] { \tl_to_str:n { |#1| } } [{bracket}] ;
\collectn_delimited_inline:NNn [ ] { \tl_to_str:n { |#1| } } \scan_stop: ;
\par
\collectn_delimited_any_inline:nn { [](){} } { \tl_to_str:n { |#1,#2| } }
  [[bracket]] ;
\collectn_delimited_any_inline:nn { [](){} } { \tl_to_str:n { |#1,#2| } }
  ({brace}) ;
\par
\collectn_delimited_all_inline:nn { [](){} } { \tl_to_str:n { |#1| } }
  (brace) [[bracket]] ;
\collectn_delimited_all_inline:nn { [](){} } { \tl_to_str:n { |#1| } }
  ({brace}) ;
\ExplSyntaxOff
```

```
|{bracket}|;|\q_no_value |;
|1,[bracket]|;|2,{brace}|;
|{[bracket]}{brace}|;|{\q_no_value }{{brace}}|;
```

\collectn_lohi:Nnw
\collectn_lohis:Nnw

\collectn_lohi:Nnw ⟨tl⟩ {⟨code⟩} ⟨content⟩

判断接下来的一些字符是否是数学上下标。\collectn_lohi:Nnw P.90 获取一组数学上下标（即数学下标和上标每个最多获取一个）；\collectn_lohis:Nnw P.90 可获取任意多个上下标。把这些上下标存储到 ⟨tl⟩ 里，然后执行 ⟨code⟩。

⟨tl⟩ 由形如 {^{⟨text⟩}} 或 {_{⟨text⟩}} 的项组成。其中 ^ 和 _ 的类别码分别为 7 和 8。

它们会自动展开后面的内容，并且自动移除空格和 \relax。

它们只会匹配类别码为 7 和 8 的显式或隐式字符。若加载了 underscore 宏包，则 _ 的类别码会修改为 13，在文本模式下无法匹配，但在数学模式下可以。

例 76

```
\ExplSyntaxOn
\collectn_lohi:Nnw \l_tmpa_tl { \tl_to_str:N \l_tmpa_tl }
  \c_math_subscript_token \scan_stop: { a } ^ { b }
\par
\collectn_lohis:Nnw \l_tmpa_tl { \tl_to_str:N \l_tmpa_tl }
  \scan_stop: \c_math_superscript_token \scan_stop: { a }
  \scan_stop: \sb { b } \sp { c } ^{ d }
\ExplSyntaxOff
```

```
{_{a}}{^{b}}
{^{a}}{_{b}}{^{c}}{^{d}}
```

\collectn_scan_keyword:nTF {⟨keyword⟩} {⟨true⟩} {⟨false⟩}

\collectn_scan_keyword:nTF
\collectn_scan_keywordcs:nTF

判断接下来的一些字符是否匹配 ⟨keyword⟩，若匹配则移除它们。为了检测后面的文字是否匹配 ⟨keyword⟩，它会自动展开后面的命令。

⟨keyword⟩ 被转为类别码为 12 的记号列（包括空格），两端的空格不会被移除。\collectn_scan_keyword:n 忽略大小写，\collectn_scan_keywordcs:n 不忽略大小写。

\collectn_set_keyword:Nn ⟨str⟩ {⟨keyword⟩}

\collectn_set_keyword:Nn
\collectn_set_keyword:(No|NV|Nv|Ne|cn|co|cV|cv|ce)

把 ⟨keyword⟩ 处理成可供使用的记号列，然后将其保存至 ⟨str⟩。

先对 ⟨keyword⟩ 执行 \tl_to_str:n，然后把所有空格替换为类别码为 12 的空格。

\collectn_scan_keyword:NTF ⟨keyword str⟩ {⟨true⟩} {⟨false⟩}

\collectn_scan_keyword:NTF
\collectn_scan_keyword:cTF
\collectn_scan_keywordcs:NTF
\collectn_scan_keywordcs:cTF

判断接下来的一些字符是否匹配 ⟨keyword⟩，若匹配则移除它们。为了检测后面的文字是否匹配 ⟨keyword⟩，它会自动展开后面的命令。

⟨keyword str⟩ 需用 \collectn_set_keyword:Nn P.91 设置。

\collectn_scan_keywords:nTF {⟨keyword clist⟩} {⟨true⟩} {⟨false⟩}

\collectn_scan_keywords:nTF
\collectn_scan_keywordscs:nTF

判断接下来的一些字符是否匹配 ⟨keywords clist⟩ 中的某一项，若匹配则移除它们。忽略大小写。为了检测后面的文字是否匹配 ⟨keywords clist⟩ 中的项，它会自动展开后面的命令。

对 ⟨keyword clist⟩ 中的每一项，先移除两端的空格和空项，然后转为类别码为 12 的记号列（包括空格）。

它不会移除 ⟨keywords clist⟩ 中重复的项，若有重复的项，则以最后出现的那个为准。

在 ⟨true⟩ 和 ⟨false⟩ 中可使用 \l_collectn_keywords_int P.92 来获取匹配的 ⟨keyword⟩ 的位置，\l_collectn_keywords_str P.92 获取匹配的 ⟨keyword⟩。

\collectn_set_keywords:NTF ⟨tl⟩ {⟨keyword clist⟩}

\collectn_set_keywords:Nn
\collectn_set_keywords:(No|NV|Nv|Ne|cn|co|cV|cv|ce)

对 ⟨keyword clist⟩ 中的每一项先移除两端的空格，再应用 \collectn_set_keyword:Nn P.91，然后将这些项保存至 ⟨tl⟩。

它不会移除 ⟨keywords clist⟩ 中重复的项，若有重复的项，则以最后出现的那个为准。

\collectn_set_keywords_from_seq:NN ⟨tl⟩ ⟨seq⟩

\collectn_set_keywords_from_seq:NN
\collectn_set_keywords_from_seq:(Nc|cN|cc)

对 ⟨seq⟩ 中的每一项应用 \collectn_set_keyword:Nn P.91，然后将这些项保存至 ⟨tl⟩。

它不会移除重复的项，若有重复的项，则以最后出现的那个为准。

\collectn_scan_keywords:NTF
\collectn_scan_keywords:cTF
\collectn_scan_keywordscs:NTF
\collectn_scan_keywordscs:cTF

\collectn_scan_keywords:NTF ⟨*keywords tl*⟩ {⟨*true*⟩} {⟨*false*⟩}

判断接下来的一些字符是否匹配 ⟨*keywords tl*⟩ 中的某一项，若匹配则移除它们。为了检测后面的文字是否匹配 ⟨*keywords tl*⟩ 中的项，它会自动展开后面的命令。

⟨*keywords tl*⟩ 需用 \collectn_set_keywords:NnP91 或 \collectn_set_keywords_from_seq:NNP91 设置。

\l_collectn_keywords_int
\l_collectn_keywords_str

⟨*keywords clist/tl*⟩ 中匹配的 ⟨*keyword*⟩ 的位置和值。若不匹配则位置为 0，值为空。

§ 2 lt3ekeys

本宏包扩展了 l3keys 的部分功能。

6.2.1 定义键

6.2.2 设置键

6.2.3 lt3ekeys-elkernel

本宏包为 LaTeX 和 LaTeX3 的一些有用的内部命令提供公共接口。

\elkernel_arg_to_keyvalue:Nnn

\elkernel_arg_to_keyvalue:Nnn ⟨*result*⟩ {⟨*key*⟩} {⟨*arg*⟩}

检查 ⟨*arg*⟩ 是否为键值对，若不是，则把 ⟨*key*⟩ 作为键，⟨*arg*⟩ 作为值。将结果保存到 ⟨*result*⟩ 中。

检查是否为键值对的方法和 ltcmd 的 = spec 一致。

\elkernel_if_date_at_least:nnTF

\elkernel_if_date_at_least:nnTF {⟨*date*$_1$⟩} {⟨*date*$_2$⟩} {⟨*true*⟩} {⟨*false*⟩}

测试 ⟨*date*$_1$⟩ 是否等于或晚于 ⟨*date*$_2$⟩。年月日可用 \ 或 - 分隔，但不可混用。

例 77

```
\ExplSyntaxOn
\tl_to_str:N \fmtversion;
\elkernel_if_date_at_least:nnTF { \fmtversion } { 2024/06/01 } { t } { f }
\ExplSyntaxOff
```

```
2023-11-01;f
```

6.2.4 定义命令——lt3ekeyscmd

本宏包实现了一个与 \DeclareDocumentCommand 类似的命令，提供了更多常用的功能。

以下称使用 \DeclareDocumentCommand 定义的命令为 *document-cmd*，称使用 \ekeysdeclarecmdP93 定义的命令为 *ekeys-cmd*。

\DeclareEKeysCommand
\ekeysdeclarecmd
\ekeys_declare_cmd:Nnn
\ekeys_declare_cmd_x:Nnn
\l_ekeys_cmd_defaults_bool

```
\ekeysdeclarecmd    ⟨cmd⟩ {⟨arg spec⟩} {⟨code⟩}
\ekeysdeclarecmd * ⟨cmd⟩ {⟨arg spec⟩} {⟨code⟩}
\ekeys_declare_cmd:Nnn    ⟨cmd⟩ {⟨arg spec⟩} {⟨code⟩}
\ekeys_declare_cmd_x:Nnn ⟨cmd⟩ {⟨arg spec⟩} {⟨code⟩}
```

类似于 \DeclareDocumentCommand，部分 ⟨*arg spec*⟩ 未实现，增加了一些 ⟨*arg spec*⟩。

\ekeysdeclarecmd[P93] 等于 \ekeys_declare_cmd:Nnn[P93]。\ekeys_declare_cmd_x:Nnn[P93] 完全展开 ⟨*code*⟩。

\DeclareEKeysCommand[P93] 是 \ekeysdeclarecmd[P93] 的另一个名字。

⟨*arg spec*⟩ 可以是数字，此时相当于 \cs_generate_from_arg_count:NNnn ⟨*cmd*⟩ \cs_set_protected:Npn {⟨*arg spec*⟩} {⟨*code*⟩}。

⟨*arg spec*⟩ 不为数字（或空）时定义的命令称为 *ekeys-cmd*。

最大的参数数量为 9。

此命令定义的命令均为使用长参数。即它们定义的命令总是 \protected\long。

目前，在向后寻找可选参数时，会忽略空格，但未来可能会修改为不忽略空格，因此不宜在可选参数前加上空格（但必需参数 m、r、R 总忽略空格）。如 \foo␣{a}␣[o] 应写为 \foo {a}[o]，假定 *arg spec* 为 mo。

在定义 *ekeys-cmd* 时，若设置 \l_ekeys_cmd_defaults_bool[P93] 为真，则可选参数的默认值可以引用其它实参（称为“执行可选参数默认值引用替换”），否则 ⟨*arg spec*⟩ 中的参数不能引用其它实参，如 r[] D(){#1} 中的 #1 只是普通文本。\ekeysdeclarecmd[P93] 不带星号的版本设置 \l_ekeys_cmd_defaults_bool[P93] 为假，\ekeysdeclarecmd[P93] 带星号的版本设置 \l_ekeys_cmd_defaults_bool[P93] 为真。\l_ekeys_cmd_defaults_bool[P93] 只有在使用上述三个命令时才会生效。注意：若默认参数之间互相循环引用，则在执行时会出错，但 *document-cmd* 则不会出错。如命令的 spec 为 D(){#2} D<>{#1}，则命令为 *ekeys-cmd* 会出错，而为 *document-cmd* 不会出错。

ekeys-cmd 可以用于 \ShowCommand、\NewCommandCopy 中，也支持 cmd 钩子，限制条件和 *document-cmd* 一致。

\DeclareEKeysCollector
\ekeysdeclarecollector

```
\DeclareEKeysCollector    ⟨collector⟩ {⟨arg spec⟩}
\DeclareEKeysCollector * ⟨collector⟩ {⟨arg spec⟩} ⟨tl⟩ {⟨do code⟩}
```

定义和 *ekeys-cmd* 类似的命令，⟨*arg spec*⟩ 的参数数量可以超过 9 个。这种命令称之为 *ekeys-collector*。

\ekeyscollectorarg ☆

```
\ekeyscollectorarg {⟨arg number⟩}
```

完全展开为第 ⟨*arg number*⟩ 个参数。如果没有该值，则为 \q_no_value，可以使用 \IfQuarkNoValueTF[P96] 检测。非正整数都无效。它只能用于 *ekeys-collector* 的 ⟨*do code*⟩ 中。

注意：不能使用 f 展开，因为值可能为 \q_no_value，它会造成无限递归。

例 78

```
\DeclareEKeysCollector \faa { m m @w{ () [] } m m m m m m m }
\faa\tmp{\meaning\tmp.} 12 [[a]](b(b)) 56789ABCD.
```

```
\faa\tmp{\edef\tmp{\ekeyscollectorarg{10}}\meaning\tmp.}
  12 [[a]](b(b)) 56789ABCD.

\DeclareEKeysCollector*\fbb { m m @w{ () [] } m m m m m m m } \tmp
↪ {\meaning\tmp.}
\fbb 12 [[a]](b(b)) 56789ABCD.
```

```
macro:->{1}{2}{b(b)}{[a]}{5}{6}{7}{8}{9}{A}{B}.CD.
macro:->A.CD.
macro:->{1}{2}{b(b)}{[a]}{5}{6}{7}{8}{9}{A}{B}.CD.
```

\ekeysgenerategrabber

\ekeysgenerategrabber ⟨*grabber tl*⟩ {⟨*arg spec*⟩}

从 ⟨*arg spec*⟩ 生成 grabber。生成的 grabber 可以用于 \ekeyscollectargsP94 命令中。

预先生成 grabber 可以避免重复生成以节省时间。

默认最大可用的参数个数为 \e@alloc@top，它远远超过 9。在 pdfLaTeX和 XeLaTeX 中为 $2^{15}-1=32767$，在 LuaLaTeX和 (u)pLaTeX 中为 $2^{16}-1=65535$。可以通过 \l_ekeys_cmd_max_args_intP94 修改。

\ekeys_grabber_args_do:NNn

\ekeys_grabber_args_do:NNn ⟨*tl*⟩ ⟨*grabber tl*⟩ {⟨*do code*⟩}

首先收集 ⟨*grabber tl*⟩ 所需的参数，每个参数都用 {} 括起来，将其保存到 ⟨*tl*⟩ 中，然后执行 ⟨*do code*⟩。

此命令不会执行可选参数默认值引用替换。

\ekeys_collect_args_do:NNn
\ekeyscollectargs

\ekeys_collect_args_do:NNn ⟨*tl*⟩ ⟨*ekeys-cmd OR grabber tl*⟩ {⟨*do code*⟩}

首先收集 ⟨*ekeys-cmd*⟩ 所需的参数或 ⟨*grabber tl*⟩ 所需的参数，每个参数都用 {} 括起来，将其保存到 ⟨*tl*⟩ 中，然后执行 ⟨*do code*⟩。

只有当第二个参数为 *ekeys-cmd* 时，才会此执行可选参数默认值引用替换。

\ekeys_collect_args_do:Nnn
\ekeyscollectargs*

\ekeys_collect_args_do:Nnn ⟨*tl*⟩ {⟨*arg spec*⟩} {⟨*do code*⟩}

首先根据 ⟨*arg spec*⟩ 收集所需的参数，每个参数都用 {} 括起来，将其保存到 ⟨*tl*⟩ 中，然后执行 ⟨*do code*⟩。

最大可用的参数个数为 \e@alloc@top，它远远超过 9。在 pdfLaTeX和 XeLaTeX 中为 $2^{15}-1=32767$，在 LuaLaTeX和 (u)pLaTeX 中为 $2^{16}-1=65535$。可以通过 \l_ekeys_cmd_max_args_intP94 修改。

此命令相当于先用 \ekeysgenerategrabberP94 生成 grabber，然后再使用 \ekeyscollectargsP94。

此命令不会执行可选参数默认值引用替换。

\l_ekeys_cmd_max_args_int

最大的可收集的参数数目。

\ekeys_cmd_undefine:N

\ekeys_cmd_undefine:N ⟨*ekeys-cmd*⟩

将 ⟨*ekeys-cmd*⟩ （局部地）设置为未定义。如果不是 *ekeys-cmd* 则保持不变。对 *ekeys-cmd* 和 *ekeys-collector* 都有效。

\ekeysmakeglobal ⟨*ekeys-cmd*⟩

\ekeysmakeglobal
\ekeys_cmd_global:N

把局部定义的 ⟨*ekeys-cmd*⟩ 改为全局定义的。\ekeysdeclarecmdP.93 总是局部地定义命令。如果不是 *ekeys-cmd* 则保持不变。

__ekeys_if_cmd:NTF ⟨*cs*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

__ekeys_if_cmd:NTF ⋆

简单判断一个控制序列是否为 *ekeys-cmd*。

可用的参数类型为：

m 标准的必须的参数；

r r⟨*token*$_1$⟩⟨*token*$_2$⟩；

R R⟨*token*$_1$⟩⟨*token*$_2$⟩{⟨*default*⟩}；

o 可选参数；

O O{⟨*default*⟩}；

d d⟨*token*$_1$⟩⟨*token*$_2$⟩；

D D⟨*token*$_1$⟩⟨*token*$_2$⟩{⟨*default*⟩}；

s 可选的星号，如果给出则为 \BooleanTrue，否则为 \BooleanFalse；

t t⟨*token*⟩，如果 ⟨*token*⟩ 给出，则为 \BooleanTrue，否则为 \BooleanFalse；

k k{⟨*keyword*⟩}，如果 ⟨*keyword*⟩ 给出，则为 \BooleanTrue，否则为 \BooleanFalse；定义命令时，忽略 ⟨*keyword*⟩ 的类别码，执行时，仅比较字符，忽略类别码（活动字符除外，它将导致失配）和大小写的区别；另见 \collectn_scan_keyword:nTFP.91；

t t{⟨*token pairs*⟩}，它占用两个参数，用于确定多个定界符对是否有一个给出了，第一个为定界符对所在的位置（如果未出现则为 0），第二个为它的值；

T T{⟨*token pairs*⟩}{⟨*default*⟩}，它占用两个参数，用于确定多个定界符对是否有一个给出了，第一个为定界符对所在的位置（如果未出现则为 0），第二个为它的值（如果未出现则为 ⟨*default*⟩）；

p p{⟨*tokens*⟩}，如果出现在 ⟨*tokens*⟩ 中，则给出所在的位置，否则给出 0；

P P{⟨*tokens*⟩}{⟨*defaults*⟩}，⟨*defaults*⟩ 的长度必须为 ⟨*tokens*⟩ 的长度加一，用出现在 ⟨*tokens*⟩ 中的位置索引 ⟨*defaults*⟩（⟨*defaults*⟩ 的索引从 0 开始），p{*-+} 相当于 P{*-+}{0123}；

e e{⟨*tokens*⟩}，使用 w 参数实现，每个符号可以重复使用；

E E{⟨*tokens*⟩}{⟨*defaults*⟩}，使用 W 参数实现，每个符号可以重复使用；

w w{⟨*token pairs*⟩}，智能匹配多个 d 类型的参数，从而可以以任意顺序给出参数的值；

W W{⟨*token pairs*⟩}{⟨*defaults*⟩}，智能匹配多个 D 类型的参数，从而可以以任意顺序给出参数的值；

g 可选的以 { 开始的参数，{ 可以是显式或隐式的；若后面跟着的不是 { 而是 \relax，则移除这个 \relax；

G G{⟨*default*⟩}，可选的以 { 开始的参数，{ 可以是显式或隐式的；若后面跟着的不是 { 而是 \relax，则移除这个 \relax；

l { 前的内容，如没有 { 则出错；

u u{⟨*tokens*⟩}，⟨*tokens*⟩ 前的内容，⟨*tokens*⟩ 会被移除，如没有 ⟨*tokens*⟩ 则出错；

U U{⟨*tokens*$_1$⟩}{⟨*tokens*$_2$⟩}，⟨*tokens*$_1$⟩ 前的内容，移除 ⟨*tokens*$_1$⟩ 并留下 ⟨*tokens*$_2$⟩，

如没有 ⟨*tokens*$_1$⟩ 则出错；

`v` 以 verbatim 的形式读取文字，该参数所含字符的类别码的可能值为 10、12、13（在 (u)pTEX 引擎下，仅 Unicode 编码小于等于 255 的字符有此特性）；且 `^^M` 的类别码为 13；

未实现的参数类型为 `b`，`+`，`!`，`>`，`=`，使用它们仅给出警告，不会报错。

除非另有说明否则上述参数类型的参数中均不能出现显式或隐式的空格（␣）、宏变量字符（`#`）、左右括号（`{ }`），即 `d<#`、`t\bgroup` 等均为错误用法。

`u` 和 `U` 的参数可以有空格，如可以 `u{ }`，必须有括号。

带有左右定界符的参数在使用时不能嵌套，如 ⟨*arg spec*⟩ 为 `d<>`，则 `\foo<a<b>>` 的参数为 `a<b`，要得到 `a<b>`，必须写 `\foo<{a<b>}>`。

lt3ekeysext 宏包对其进行了进一步扩展，例如支持定义可嵌套的定界符以及更高的执行效率。见第 6.2.5 小节。

`\ekeys_cmd_name:` ⋆

完全展开为 *ekeys-cmd* 的名称。仅能用在 *ekeys-cmd* 执行的 ⟨*code*⟩ 中。

`\IfQuarkNoValueTF` ⋆
`\IfQuarkNoValueT` ⋆
`\IfQuarkNoValueF` ⋆

`\IfQuarkNoValueTF {`⟨*arg*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

判断 ⟨*arg*⟩ 是否为 `\q_no_value`。

`\NumberCase` ⋆
`\@numbercase` ⋆

`\NumberCase {`⟨*integer*⟩`} { {`⟨*case*$_0$⟩`} {`⟨*case*$_1$⟩`} ... {`⟨*case*$_n$⟩`} } {`⟨*else*⟩`}`
`\@numbercase` ⟨*integer*⟩ `; {`⟨*case*$_0$⟩`} {`⟨*case*$_1$⟩`} ... {`⟨*case*$_n$⟩`} ; {`⟨*else*⟩`}`

根据 ⟨*integer*⟩ 的值选择对应的 ⟨*case*$_i$⟩，如果 *integer* $>$ *n* 或 *integer* $<$ 0 则使用 ⟨*else*⟩。

`\BooleanFalse` 相当于 0，`\BooleanTrue` 相当于 1。

展开两次即可得到结果。

例 79

```
\ExplSyntaxOn
\ekeys_declare_cmd:Nnn \foo { p{!@*-+} } { 位置为: #1. }
\ekeys_declare_cmd:Nnn \fee { p{!@*-+} }
  { \NumberCase {#1} { {错误! } } { 给出 \tl_item:nn {!@*-+} {#1}. } }
\ekeys_declare_cmd:Nnn \fii { P{!@*-+}{ {错误}{!}{@}{*}{-}{+} } } {
↪ 给出: #1. }
\ExplSyntaxOff

\foo @ \par
\foo ? \par
\fee @ \par
\fee ? \par
\fii @ \par
\fii ?
```

位置为：2.
位置为：0.?
给出 @.
错误！？
给出：@.
给出：错误.?

例 80

```
\ekeysdeclarecmd \foo { t{ () [] <> } } {位置为：#1，值为：#2。}
\foo <c> \foo (p) \foo\relax
```

```
位置为：3，值为：c。位置为：1，值为：p。位置为：0，值为：-NoValue-。
```

w 和 W 中（包括 e 和 E）的定界符对可以重复使用，但在第一个相同的情况下第二个也必须相同。如支持 w{ [] () [] }、w{ (] [] <] }，但不支持 w{ [) [] }。

例 81

```
\ekeysdeclarecmd \foo { w{ [] () <> } } {\{#1;#2;#3\}}
\ekeysdeclarecmd \fee { w{ [] [] () } } {\{#1;#2;#3\}}

\foo <a> (b) [c] , \foo (a) [b] <c> ,
\fee [a] (b) [c] , \fee [a] [b] (c) , \fee (a) [b] [c] ,
```

```
{c;b;a} , {b;a;c} , {a;c;b} , {a;b;c} , {b;c;a} ,
```

t、T、w、W 的每对的第二个还能为空或空格。例如 w{ _{} ^{} } 相当于 e{_^}，例如支持 w{ [] () _{} ^{} :{ } }。但第一个不能为空或空格，如不支持 w{ {}_ }。

t 和 T 用于仅需要一个参数的情况，这些定界符对只会检测一次，而 w 和 W 则支持多个，可以以任意顺序给出。

\ekeys_exp_not_braced:n {⟨*text*⟩}

\ekeys_exp_not_braced:n ⋆
\ekeys_exp_not_braced:(o|V|v|f|e) ⋆

展开为 {\exp_not:n{⟨*text*⟩}}。

\ekeys_cmd_after_declare:n {⟨*code*⟩}

\ekeys_cmd_after_declare:n
\ekeys_cmd_after_declare:e

定义完本命令后执行 ⟨*code*⟩。在定义命令的外面使用是无效的。

6.2.5 定义命令扩展——lt3ekeysext

lt3ekeysext 进一步扩展了 lt3ekeyscmd 的功能。

为 \ekeysdeclarecmd[p93] 增加了几个前缀：

- # 将其后的那个参数类型标记为需要特殊处理（如果可以的话），使用此前缀能（大大）提高 *ekeys-cmd* 的执行速度，但有轻微的副作用（一般很难发生，详见后文）；支持 T、p、P、u、U、e、E、w、W；
- @ 将其后的那个参数类型标记为需要嵌套（如果可以的话）；
- & 将其后的那个参数类型标记为保留等价的原始输入；支持 r、R、o、O、d、D，s、t、p，k，t、T，g、G，e、E、w、E；

在使用 # 前缀时，参数在其定界符为控制序列的时候，定义和使用 *ekeys-cmd* 时 \escapechar 的值必须为相同的正数。如，当定义 \foo 时 \escapechar 为 \，则使用 \foo 时也必须为 \。受影响的参数类型为 t、T、p、P、e、E、w、W。

使用 t、T、e、E、w、W 时，设置 & 将自动设置 #。

对于 s、t 和 k 参数，使用 & 前缀，在检测到存在此 ⟨*token*⟩（或 ⟨*keyword*⟩）时，会把这个参数设置为该 ⟨*token*⟩（或 ⟨*keyword*⟩），否则这个参数为空。使用其它几个支持 & 前缀的参数时，若检测到有此定界符，则该参数会包含此定界符，否则该参数为 ⟨*default*⟩。

例 82

```
\newcommand{\abdel}[5]{#1#3#4#2#5}
\ekeysdeclarecmd\ab{ p{\big\Big\bigg\Bigg*} &t{ \{\} [] () || } }
```

```
  {\IfNoValueTF{#3}{NAN}% 如果没有所列的括号，输出 NAN
    {\NumberCase{#1} {
        {\abdel\left\right}
        {\abdel\bigl\bigr} {\abdel\Bigl\Bigr}
        {\abdel\biggl\biggr} {\abdel\Biggl\Biggr} {\abdel{}{}}
      }{}
     #3}
  }

$ \ab[\dfrac12] $, $ \ab\Big(\dfrac12) $, $ \ab*|\dfrac12| $, $ \ab? $
```

$$\left[\frac{1}{2}\right], \left(\frac{1}{2}\right), \left|\frac{1}{2}\right|, NAN?$$

例 83

```
\ekeysdeclarecmd\foo{ &s &t{ () [] } &w{ !{} +{ } } &k{p_t} }
  { 1.\{#1\}, 2.\{#2|#3\}, 4.\{#4\}\{#5\} 6.\{#6\} }
\ttfamily
{\catcode`\_=13 \foo *[b]+plus !{mu} p_t} , % 此时 _ 为活动字符
{\catcode`\_=8  \foo *[b]+plus !{mu} p_t} , % 此时 _ 为下标
\foo (b)!mu ,
```

```
    1.{*}, 2.{2|[b]}, 4.{!mu}{+plus } 6.{} p_t ,  1.{*}, 2.{2|[b]},
4.{!mu}{+plus } 6.{p_t}  ,  1.{}, 2.{1|(b)}, 4.{!m}{-NoValue-} 6.{}
u ,
```

执行效率从慢到块大致如下：无 # 且为 t、T、w、W 的 *ekeys-cmd* < 无 # 且有 @ 的 *ekeys-cmd* < *document command* < 带 # 和 @ 的 *ekeys-cmd* < 带 # 的 *ekeys-cmd* < \ekeys_grabber_args_do:NNn[p.94] < __ekeys_grabber_args_do:Nnn。

为 \ekeysdeclarecmd[p.93] 增加了几个参数类型：

c c{⟨*collect spec*⟩}。特殊的保存方式；必须的参数；

C C{⟨*allocated collect spec*⟩}。特殊的保存方式；必须的参数；

K K{⟨*scanner-and-args*⟩}。自定义的参数扫描器。

c 和 C 用 TEX 寄存器来保存各种值，设置这些值的方式和 TEX 相似。前者会自动分配一个新的寄存器，而 C 使用已经分配好的寄存器，这是它们的唯一区别。这两个参数类型的实参为所分配的寄存器，一般情况下不应直接修改这个寄存器。

如

例 84

```
\ekeysdeclarecmd\foo{ c{i} m }{[\the#1,\detokenize{#2}]}
\ttfamily
\foo -12 {a}; \quad \foo -12 \relax; \quad \foo -12\relax;
\foo \inteval{12+8*(-2)}\relax; \foo \numexpr 12+8*(-2)\relax;
```

```
    [-12,a];   [-12,\relax ];   [-12,\relax ]; [-4,\relax ]; [-4,;]
```

定义了一个命令 \foo，第一个参数存储一个整数，第二个参数为通常的参数。注意到给出第一个参数时与通常的使用方式很不相同，这是 TEX 中为寄存器赋值的语法，每个类型都遵循各自的语法规则，详细用法见 *The TEXbook*。

⟨*collect spec*⟩ 可用的值如下：

i 为 int（count）寄存器赋值；
d 为 dim（dimen）寄存器赋值；
s 为 skip 寄存器赋值；
m 为 muskip 寄存器赋值；
t 为 toks 寄存器赋值；
b ⟨*extra spec*⟩，为一个水平盒子赋值；{⟨*extra spec*⟩} 可以为“*”表示可自由设置盒子宽度；也可为 [⟨*width*⟩][⟨*pos*⟩]，正如 \makebox 的前两个参数；
w ⟨*extra spec*⟩，为一个固定宽度的盒子赋值；类似于把参数放在一个 minipage 里；⟨*extra spec*⟩ 可以为 [⟨*vpos*⟩][⟨*height*⟩][⟨*inner pos*⟩]{⟨*width*⟩}，正如 \parbox 的前四个参数一样。若只给出 ⟨*width*⟩ 则为垂直盒子，且可以对其使用 \vsplit，否则为水平盒子；
v ⟨*extra spec*⟩，为一个有最大宽度的盒子赋值；类似于把参数放在一个 varwidth 里；⟨*extra spec*⟩ 可以为 [⟨*vpos*⟩][⟨*height*⟩][⟨*inner pos*⟩]{⟨*width*⟩}，正如 \parbox 的前四个参数一样。若只给出 ⟨*width*⟩ 则为垂直盒子，且可以对其使用 \vsplit，否则为水平盒子。

注意 ⟨*extra spec*⟩ 的外侧不能有花括号，除非是只有 ⟨*width*⟩ 这个必须参数。并且可选参数之间不能有额外的空格。如 b{*}、b[3cm]、w{3cm}、w[c]{3cm} 可以，b{[3cm]}、b{[3cm][l]}、w{[c]{3cm}} 不可以。

前 5 个类型，即 i、d、s、m、t，的实参可以作为 LaTeX3 函数的 V 变体的参数。

⟨*allocated collect spec*⟩ 是一个已经分配的寄存器加上 ⟨*collect spec*⟩，若 ⟨*collect spec*⟩ 的类型没有必须参数，则可以省略不写，只写寄存器，但寄存器必须和类型一致。

⟨*collect spec*⟩ 和 ⟨*allocated collect spec*⟩ 中不能引用其它实参，因为它们并非默认值。

例 85

```
\newdimen\tmpadim
\ekeysdeclarecmd\foo { C{\tmpadim} c{i} c{b[2cm][c]} }
  { [\the#1, \the#2, \fbox{\box#3}] }
\foo 12pt -1 {a a}
\foo \dimexpr 5cm+12pt\relax \numexpr 3+4*(1-2)\relax {a a}
```

[12.0pt, -1, a a] [154.26378pt, -1, a a]

例 86

```
\ekeysdeclarecmd \myfbox { c{b} } {\fbox{\box#1}}
\myfbox{\verb|\myfbox| 与 \verb|\fbox|}
\myfbox{\parbox{3cm}{可分段的\par fbox}}
```

\myfbox 与 \fbox

可分段的
fbox

\ekeys_cmd_new_scanner:nnnpn {⟨*scanner*⟩} {⟨*argument numbers*⟩}
{⟨*initial action*⟩} ⟨*parameter list*⟩ {⟨*scanner action*⟩}
\ekeys_cmd_new_scanner:nnpn {⟨*scanner*⟩} {⟨*argument numbers*⟩}
⟨*parameter list*⟩ {⟨*scanner action*⟩}

\ekeys_cmd_new_scanner:nnnpn
\ekeys_cmd_new_scanner:nnpn
\ekeys_cmd_new_scanner:nnnpx
\ekeys_cmd_new_scanner:nnpx

为参数类型 K （局部地）定义新的扫描器 ⟨*scanner*⟩。该扫描器为 *ekeys-cmd* 增加 ⟨*argument numbers*⟩ 个参数。

⟨*initial action*⟩ 为定义 *ekeys-cmd* 时要执行的操作。可以使用参数，分别为 *ekeys-cmd* 的名称，给该扫描器的额外的参数，此 *ekeys-cmd* 已有的参数个数，是否为 raw 参数（使用了 &），以及是否为 nested 参数（使用了 @）。

注意，直接在 ⟨*initial action*⟩ 中定义 *ekeys-cmd* 和 *ekeys-collector* 是不行的。但可以在其中使用 \ekeys_cmd_after_declare:n p97，先定义完本命令后再定义新的命令。

给扫描器的额外的参数指的是：移除 ⟨*scanner-and-args*⟩ 的首尾空格后，若它以一对 { } 开始，则这个花括号里的内容就是 scanner，之后的内容移除掉两端的空格后就是额外的参数；否则，就是 scanner 就是第一个花括号之前的内容（移除掉两端的空格），额外的参数就是第一个花括号及其之后的记号。如若 ⟨*scanner-and-args*⟩ 为 ␣{a␣b␣}␣[v]␣j␣，则 scanner 为 a␣b␣，参数为 [v]␣j；若 ⟨*scanner-and-args*⟩ 为 ␣fo␣o␣{da}␣[b␣]␣，则 scanner 为 fo␣o，参数为 {da}␣[b␣]。

⟨*scanner action*⟩ 为 *ekeys-cmd* 执行到该扫描器时需要执行的操作。⟨*parameter list*⟩ 为需要的形参列表。⟨*scanner action*⟩ 中，使用 \ekeys_cmd_add_args:n P.102 为 *ekeys-cmd* 添加实参。⟨*scanner action*⟩ 完成时，必须执行 \ekeys_cmd_scanner_end: P.101。

注意 \ekeys_cmd_name: P.96 不能在 ⟨*initial action*⟩ 中使用（可以用 #1 替代），但可以在 ⟨*scanner action*⟩ 中使用。另见 \ekeys_cmd_args: P.102。

\ekeys_cmd_name: P.96 与 ⟨*initial action*⟩ 的第一个可用的参数相同，\ekeys_cmd_args: P.102 与 ⟨*scanner action*⟩ 的第三个可用参数相同。

\ekeysnewscanneralias
\ekeys_cmd_new_scanner_alias:nn
\ekeys_cmd_new_scanner_alias:nnn

```
\ekeysnewscanneralias {⟨alias⟩} {⟨scanner definition⟩}
\ekeysnewscanneralias {⟨alias⟩} [⟨spec name⟩] {⟨part spec⟩}
\ekeys_new_scanner_alias:nn  {⟨alias⟩} {⟨scanner definition⟩}
\ekeys_new_scanner_alias:nnn {⟨alias⟩} {⟨spec name⟩} {⟨part spec⟩}
```

（局部地）定义扫描器的别名。

⟨*scanner definition*⟩ 是用在 K 的 ⟨*scanner-and-args*⟩ 里的内容。⟨*spec name*⟩ 是标识参数类型的字母，⟨*part spec*⟩ 是此类型具体的一个。见例 88。

当使用 K{u⟨*args*⟩} 时，只使用 ⟨*args*⟩ 的第一项，多余的部分会放到 ⟨*scanner definition*⟩ 或 ⟨*part spec*⟩ 后面。

有几个预定义的扫描器：

norelax，移除一个 \relax，在向后寻找这个 \relax 时忽略空格；

norelax!，移除一个 \relax，在向后寻找这个 \relax 时不忽略空格；

?⟨*preprocessor spec*⟩，参数预处理器。⟨*preprocessor spec*⟩ 用法为：

- {⟨*ekeys-cmd arg spec*⟩}，使用 ⟨*ekeys-cmd arg spec*⟩ 获取后面的参数，转化为带花括号的标准参数；
- {⟨*ekeys-cmd arg spec*⟩}{⟨*scanner code*⟩}，使用 ⟨*ekeys-cmd arg spec*⟩ 获取后面的参数，并用 ⟨*scanner code*⟩ 替换这些参数。⟨*scanner code*⟩ 必须包含 \ekeys_cmd_scanner_end: P.101，且在它后面的内容会被 *ekeys-cmd* 重新读取；
- {⟨*ekeys-cmd arg spec*⟩}{⟨*scanner code*⟩}⟨*text*⟩，同上，⟨*text*⟩ 会放在要读取的参数之前。

u u{⟨*alias*⟩}，使用由 \ekeysnewscanneralias P.100 设置的别名。

define 可以用它来在定义 *ekeys-cmd* 时，进一步的自定义参数获取方式。用法为 K{define⟨*define spec*⟩}。⟨*define spec*⟩ 可以为：

- {⟨*definition*⟩}，它不占用任何参数。主要用作向后检查，或展开，也可以向输入中添加额外的内容。需要在 ⟨*definition*⟩ 的合适位置添加 \ekeys_cmd_add_args:n P.102 和 \ekeys_cmd_scanner_end: P.101；
- {⟨*numbers*⟩}{⟨*parameters*⟩}，它占用 ⟨*numbers*⟩ 个参数。根据 ⟨*parameters*⟩ 向后获取参数。⟨*parameters*⟩ 的参数数量不能少于 ⟨*numbers*⟩。这是 \def 的 *ekeys-cmd* 接口；
- {⟨*numbers*⟩}{⟨*parameters*⟩}{⟨*definition*⟩}，同上。注意：当 *numbers* ≥ 0 时，会在 ⟨*definition*⟩ 后面自

动加上 \ekeys_cmd_add_args:n P.102 和 \ekeys_cmd_scanner_end: P.101；

- {*⟨numbers⟩}{⟨parameters⟩}{⟨definition⟩}，⟨numbers⟩ 与 ⟨parameters⟩ 没有关系。但需要在 ⟨definition⟩ 的合适位置添加 \ekeys_cmd_add_args:n P.102 和 \ekeys_cmd_scanner_end: P.101，且添加的参数必须和 ⟨numbers⟩ 一致。

lohi 它获取一组数学上下标，占用 2 个参数，第一个为下标，第二个为上标。如果不存在，则为使用默认值。

设置默认值是通过给扫描器的额外的参数来设置：K{lohi⟨defaults⟩}，其中 ⟨defaults⟩ 为：

- {⟨$default_{lo}$⟩}{⟨$default_{hi}$⟩}，设置下标和上标的默认值；
- {⟨default⟩}，设置上下标的默认值；
- 空，表示上下标使用特殊的标记：-NoValue-。

它支持 raw 修饰符，即可以自动添加 _ 和 ^，但默认值不会自动添加这两个符号。

例 87

```
\ekeysdeclarecmd\faa{ K{lohi} }{[#1.#2]}
\ekeysdeclarecmd\fbb{ K{lohi{}} }{[#1.#2]}
\ekeysdeclarecmd\fcc{ &K{ lohi {_x}{^y} } } {[$\int #1#2$]}

\faa ^b_a; \faa \relax _a ^\relax b; \faa ^b; \faa ?; \par
\fbb ^b; \fbb _a^b; \fbb ?; \par
\fcc _a^b; \fcc _a; \fcc ?;
```

[a.b]; [a.b]; [-NoValue-.b]; [-NoValue-.-NoValue-]?;

[.b]; [a.b]; [.]?;

$[\int_a^b]$; $[\int_a^y]$; $[\int_x^y]$?;

例 88

```
\ekeysnewscanneralias{math-style}[p]
  {p{\displaystyle\textstyle\scriptstyle\scriptscriptstyle}}
\ekeysnewscanneralias{scan-bra-ket}{define{2}{<#1|#2>}}

\DeclareEKeysCommand \foo { &K{u{math-style}} K{u{scan-bra-ket}} }
  {{#1\left<#2\middle|#3\right>}}
$ \foo<a|b> $\quad $ \foo\displaystyle<\sum|\prod> $.
```

$\langle a|b\rangle$ $\left\langle \sum \middle| \prod \right\rangle$.

除了 ? 参数扫描器外，还有一个预处理指示符 ?，用法为 ?{⟨preprocessor action⟩}。预处理器指示符可以看成是 ? 参数扫描器的预先构建的版本。

\l_ekeys_cmd_scanner_args_int

实际上，scanner 的参数数量不仅可以通过 ⟨argument numbers⟩ 设置，还可以在 ⟨initial action⟩ 中直接修改 \l_ekeys_cmd_scanner_args_int P.101 来设置参数数量。

\ekeys_cmd_scanner_end:
\EKeysEndPreprocessor

参数扫描器完成时，必须执行该命令。放在此命令之后的内容会添加到输入中，可供后面的参数解析。

如果不是用在扫描器或预处理器内，\ekeys_cmd_scanner_end: P.101 会出错，而 \EKeysEndPreprocessor P.101 则什么也不做。预处理器和自定义的参数扫描器必须包含它们之一。

例 89

```
\ExplSyntaxOn
\ekeys_cmd_new_scanner_alias:nn { replace-keyword-to-num }
  {
    define
      {
```

```
        \collectn_scan_keywords:nTF { true, false, on, off, yes, no }
          {
            \int_if_odd:nTF \l_collectn_keywords_int
              { \ekeys_cmd_scanner_end: 1~ }
              { \ekeys_cmd_scanner_end: 0~ }
          }
          { \ekeys_cmd_scanner_end: -1~ }
      }
  }
\DeclareEKeysCommand \foo { K{u{replace-keyword-to-num}} u{~} } {[#1]}
\ExplSyntaxOff
\foo TRUE; \foo on; \foo false; \foo ;
```

[1]; [1]; [0]; [-1];

\ekeys_cmd_add_arg:n
\ekeys_cmd_add_arg:(o|V|v|f|e)

\ekeys_cmd_add_arg:n {⟨*argument*⟩}

为 *ekeys-cmd* 添加 1 个实参。⟨*scanner action*⟩ 中添加的实参总数量必须和定义扫描器时设置的数目一致。

\ekeys_cmd_add_args:n
\ekeys_cmd_add_args:(o|V|v|f|e)

\ekeys_cmd_add_args:n { {⟨*argument*$_1$⟩} ... {⟨*argument*$_n$⟩} }

为 *ekeys-cmd* 添加 *n* 个实参。⟨*scanner action*⟩ 中添加的实参总数量必须和定义扫描器时设置的数目一致。

\ekeys_cmd_register_cs:N
\ekeys_cmd_register_cs:c

\ekeys_cmd_register_cs:N ⟨*auxiliary function*⟩

仅当 ⟨*auxiliary function*⟩ 不能在不同命令之间共享时才需注册，否则可以直接在外层定义，不必置于 ⟨*initial action*⟩ 之内。

注册命令的主要作用是使用 \ekeys_cmd_undefine:N P94 时，把这些注册的命令也设置为未定义。

\ekeys_cmd_scanner_undefine:n
\ekeys_cmd_scanner_alias_undefine:n

\ekeys_cmd_scanner_undefine:n {⟨*scanner*⟩}
\ekeys_cmd_scanner_alias_undefine:n {⟨*scanner alias*⟩}

将 ⟨*scanner*⟩ 或 ⟨*scanner alias*⟩ 局部地设置为未定义。

\ekeys_cmd_args: ⋆

完全展开为到目前为止此 *ekeys-cmd* 已经使用了的参数数目。仅能用于 ⟨*scanner action*⟩。可与 \ekeys_cmd_name: P96 配合使用。

请看下面两个例子了解它们的用法。

例 90

```
\ExplSyntaxOn
\ekeys_cmd_new_scanner:nnnpn { my/if-pt } { 1 }
  {
    \ekeys_cmd_register_cs:c { #1-#3-my/if-pt/is-raw }
    \bool_new:c { #1-#3-my/if-pt/is-raw }
    \bool_set:cn { #1-#3-my/if-pt/is-raw } {#4}
  }
  {
    \bool_set_eq:Nc \l_tmpa_bool
      { \ekeys_cmd_name: - \ekeys_cmd_args: - my/if-pt/is-raw }
    \collectn_scan_keyword:nTF { pt }
      {
        \bool_if:NTF \l_tmpa_bool
          { \ekeys_cmd_add_args:e { { \tl_to_str:n { pt } } } }
```

```
          { \ekeys_cmd_add_args:n { { \BooleanTrue } } }
        \ekeys_cmd_scanner_end:
      }
      {
        \bool_if:NTF \l_tmpa_bool
          { \ekeys_cmd_add_args:n { {} } }
          { \ekeys_cmd_add_args:n { { \BooleanFalse } } }
        \ekeys_cmd_scanner_end:
      }
  }
\ExplSyntaxOff
\ttfamily
\ekeysdeclarecmd \foo { &K{my/if-pt} m } {[#1](#2)}
\DeclareCommandCopy{\foocopied}{\foo}
\AddToHookWithArguments{cmd/foocopied/before}{\{#1|#2\}}
\foocopied pt; \foocopied PT; \foocopied p @; \quad
\foo pt; \foo PT; \foo p @;
```

```
    {pt|;}[pt](;)  {pt|;}[pt](;)  {|p}[](p)  @;    [pt](;)  [pt](;)
[](p) @;
```

例 91

```
\ExplSyntaxOn
\dim_new:N \l__my_scan_whd_dim % 临时保存长度
\seq_new:N \l__my_scan_whd_seq % 保存三个值, width,height,depth
\cs_generate_variant:Nn \seq_set_item:Nnn { NnV }
\collectn_set_keywords:Nn \c__my_scan_whd_tl { width, height, depth }
% 3 个参数, 为 width, height, depth
\ekeys_cmd_new_scanner:nnpn { my/scan-whd } { 3 }
  {
    % initial seq
    \seq_clear:N \l__my_seq_whd_seq
    \seq_put_right:NV \l__my_seq_whd_seq \c_novalue_tl
    \seq_put_right:NV \l__my_seq_whd_seq \c_novalue_tl
    \seq_put_right:NV \l__my_seq_whd_seq \c_novalue_tl
    \__my_scan_whd_next:
  }
\cs_new:Npn \__my_scan_whd_next:
  {
    \collectn_scan_keywords:NTF \c__my_scan_whd_tl
      {
        \collectn_value:Nnw \l__my_scan_whd_dim
          {
            \seq_set_item:NnV \l__my_seq_whd_seq
              { \l_collectn_keywords_int } \l__my_scan_whd_dim
            \__my_scan_whd_next:
          } =
      }
      {
        \ekeys_cmd_add_args:e
          { \seq_map_function:NN \l__my_seq_whd_seq \ekeys_exp_not_braced:n
↪ }
        \ekeys_cmd_scanner_end:
      }
  }
\ExplSyntaxOff

\ekeysdeclarecmd \foo { K{my/scan-whd} }
  {[ W: \IfValueTF{#1}{#1}{--}; H: \IfValueTF{#2}{#2}{--}; D: #3 ]}
```

```
\foo height 3pt depth \dimeval{3pt+2pt-10pt} height \dimexpr 3pt-2pt wide
```

[W: –; H: 1.0pt; D: -5.0pt]wide