LaTeX 宏包开发入门

LaTeX Macro Package Development Guide

匿 名

目录

前言

LaTeX 作为建于 TeX 排版程序之上的最为成功的格式之一，其追求的目标是将内容和格式分离，或更确切地，将文档标记和其排版细节分离。并且 LaTeX 本身具有很强的扩展性，LaTeX 2ε 带来了完善的宏包和文档类架构，为扩展 LaTeX 功能的开发者提供了良好的界面。

我们从一些统计数字可以瞥见其成功的一斑：在作者计算机系统内安装的 TeX Live 2015，在所有宏包和工具完全安装的情况下，位于 TeX 安装目录（TDS）里 tex/latex 目录下的子目录有 2000 多个，共有超过 4000 个宏包文件（.sty），超过 600 个文档类（.cls）以及数以千计的各类辅助文件。LaTeX 的一大核心功能——新字体选择框架（New Font Selecting Scheme, NFSS）带来了良好的字体属性切换功能，约 3000 个字体描述文件（.fd）在各个字体宏包中为 LaTeX 的使用者提供字体支持。

关于 LaTeX 的文档也数不胜数，从 LaTeX 格式的创始人 Leslie Lamport 亲笔著作的 *LaTeX: A Document Preparing System*、上千页的砖头 *The LaTeX Companion*，到短小精悍的 *LaTeX 2ε: An unofficial reference manual*、*The Not So Short Introduction to LaTeX 2ε*[1]，再到中文书籍《LaTeX 入门》以及 lnotes 笔记，等等等。

但是，以上书籍面向的都是 LaTeX 的**使用者**，而非**开发者**。将 LaTeX 预定义的许多文档元素“改头换面”，修饰成自己需要的样式并非易事。LaTeX 并没有给用户预留很多修改样式的余地，这让我们不得不“黑”进 LaTeX 的本来面貌，仔细分析它是如何用 TeX 的基础命令构建我们所见到的样式，然后思考我们如何照猫画虎，或者做得更上一层楼。

文档结构

各章力图从易到难，从 LaTeX 封装完善、便于用户使用的命令，逐渐深入到 TeX 的一些底层的命令。

仿照 *The TeXbook*，有一些段落前面以两个 ‘dangerous sign’ 标记，字体设置为仿宋。这样的段落作为对正文的补充，涉及到的内容通常超出所在章节的范围。如果读者不能理解该段落的内容，可以跳过，不影响对正文的理解。

当然如果读者有志于做一个 TeXhacker，还是可以好好读一读的，但为了配合正文，补充内容往往只是蜻蜓点水，通常要配合一些经典的书籍深入而系统地理解。常见参考书如 *The TeXbook*，或者 *TeX by topic* 等。

然而要提醒各位读者：简单的往往是最好的。带标记的段落经常会告诉读者一些难以使用，或者用起来有一定风险的技巧。如果读者能够以自己所熟悉的技巧达到想要的目的，那么就应当避免使用一些黑客手段。

[1] 本书的副标题大名在外：*LaTeX 2ε in N minutes*（N 分钟学会 LaTeX 2ε），N 等于文档的页码数。

第一章 在 LaTeX 中定义命令和环境

1.1 定义 LaTeX 命令

LaTeX 为用户提供了一些相对方便的定义命令和环境的方式。用户可以用这些方式定义一些简便的、较为基本的命令。对于初学者来说，LaTeX 提供的这些方式相对安全一些。

1.1.1 定义新命令的 \newcommand

```
% 定义不带可选参数的命令
\newcommand\foo[2]{Test #1 and #2.}
% 中括号代表命令的参数个数，如果省略相当于没参数
% 范围从1到9，不能定义更多
% 大括号里的 #1 和 #2 代表具体的参数
% 用到的参数不能比前面定义的多，可以少，甚至一个都不用

% 定义带可选参数的命令
\newcommand\baz[2][text]{Test #1 and #2.}
% 第一个中括号代表命令的参数个数，如果省略相当于没参数
% 第二个中括号代表命令第一个参数的默认值
```

在这两句定义之后，我们就可以用这些命令了：

```
\foo{A}{B}      % 结果是：Test A and B.
\baz[A]{B}      % 结果是：Test A and B.
\baz{B}         % 结果是：Test text and B.
```

使用带参数的命令时，参数的形式可以是一个字符、一个命令，或者用花括号括起来的一组字符和/或命令。

\newcommand 在定义命令时会检查命令是否被定义过，如果定义过则报错。

确切地说，\newcommand 在检查被定义的命令时遇到以下情况会报错：

1. 命令已经定义；
2. 命令以 \end 开头；
3. 命令是 TeX 原始命令 \relax。

\newcommand 命令后可选择带一个星号，也就是 \newcommand*。用带星号的版本定义命令，以及使用其定义的命令时，内容里不能换行（不能有空行或者 \par），否则报类似于 Paragraph ended before \foo is complete 的错误。

1.1.2 重定义命令的 \renewcommand

如果想要改变一个命令原始的定义，\renewcommand 是一个较为方便和可靠的方法。\renewcommand 和 \newcommand 的方法一致：

```
\renewcommand\foo[4]{Test #1 and #2 and #3 and #4.}
```

\renewcommand 可以在方法规范内改变任何原来的定义——带星号与否，参数个数，带不带默认参数，默认参数是什么，都可以通过重定义来改变。

\renewcommand 所做的检查与 \newcommand 相反，它检查所要定义的命令是否存在，如果不存在，则报错。

1.1.3 折衷方案 \providecommand

读完上面两个小节，有人会问：我如果不清楚一个命令存在与否，用 \newcommand 和 \renewcommand 都有可能报错，怎样才好呢？\providecommand 提供了一个折衷的方案：如果没有定义则正常定义，否则不替换原来的定义。

```
\providecommand*\foo[2][text]{Test #1 and #2.}
\foo{more}       % 结果：Test text and more.

\def\baz#1#2#3{Test #1, #2 and #3.}
\providecommand\baz[4]{Test #1, #2, #3 and #4.}
\baz abc         % 结果：Test a, b and c.
\baz{a}{b}{c}{d} % 结果：Test a, b and c.d
% \baz{a}{b}{c} 起作用，{d}只是跟在其后输出
```

1.2 定义 LaTeX 环境

LaTeX 提供了 \begin{. . . } 和 \end{. . . } 这样的形式定义所谓的“环境”。如同前后括号 { 和 } 一样，LaTeX 定义的环境令一些命令在环境内部生效。

新定义环境的命令 \newenvironment 的用法为：

```
\newenvironment{foo}[3][text]{<begin>}{<end>}
% 第一个参数为环境名
% 后两个可选参数与 \newcommand 一致
% <begin> 和 <end> 分别为 \begin{foo} 和 \end{foo} 执行的代码
% 用于环境开头和结尾
```

\renewenvironment 的用法和 \newenvironment 一致。类似 \(re)newcommand，定义环境的命令会检查相应的环境定义情况。

\(re)newenvironment 也可带星号，它会对环境开头和结尾定义的代码同时生效。

\(re)newenvironment{foo} 实际在内部定义了一对命令 \foo 和 \endfoo，两个命令分别执行环境开头和结尾的内容。整个 \begin{foo} . . . \end{foo} 大致相当于以下代码：

```
<在此检查 \foo 是否定义>
\begingroup
  <如果 \foo 未定义，在此报错>
  <否则记录环境名称 foo>
  \foo
  ...
  % endfoo 的 foo 是 \end 命令的参数，可能误写成 bar
  \csname endfoo\endcsname
  <将 \end 命令的参数与记录比对，不一致则报错>
```

```
\endgroup
```

事实上一个环境里面的一对命令 \foo 和 \endfoo 中，后者不一定需要定义，即使没有也不报错。就上述例子中，如果 \endfoo 未定义，则 \csname foo \endcsname 的结果等同于 \relax。这允许我们将命令直接当做环境使用，如将 \itshape 直接用成 \begin{itshape} ... \end{itshape}。这种用法不甚合理，多数情况下，还是应该避免这种情况，尽量使用成对定义的环境。

1.3 底层的定义方式

\newcommand 和 \renewcommand 内部提供了完善的排查错误的机制，并将原始的定义封装起来。对于有经验的 LaTeX 用户，这样的封装有时候并不一定满足我们的需要，所以有必要了解一下更底层的定义方式。

1.3.1 \def 和 \gdef

TeX 最基本的定义命令为 \def，结构如下：

```
\def\foo#1#2#3{Test #1 and #2, and then #3.}
% \def 后一般是一个反斜杠开头的命令，如示例中的 \foo
% \foo 后面是参数列表
% 列表可以包括形如 #1-#9 的参数记号、普通字符和 \ 开头的命令组成
% 定义的内容，或称为“替换文本”(replace text)包裹在一对花括号内
```

\def 在定义命令时不会做任何检查，如果原来这个命令有定义，则新的 \def 直接覆盖之。

事实上 \newcommand* 定义的不带可选参数的命令完全可以用 \def 等效定义：

```
% 以下两种定义方式完全等效
\newcommand*\foo[3]{Test #1, #2 and #3}
\def\foo#1#2#3{Test #1, #2 and #3}
```

\def 命令之后的参数可以不仅仅是 #1~#n，也可以是字符和命令。这使得 \def 能够定义出复杂多变的命令：

```
\def\baz[#1|#2]{Test [#1] and [#2].}
% \baz 命令后的参数必须包含不在分组之内的 [、| 和 ] 三个符号
% [ 必须紧跟 \baz
% [ 和 | 之间的所有字符、命令和分组在展开时带入 #1
% | 和 ] 之间的内容带入 #2
\baz[some {\LaTeX | text}|more {\LaTeX | text}]
% 分组内的 [、| 和 ] 与 \baz 没有关系
```

\def 更加多变的用法在后文会详细阐述。

\gdef 用于定义全局命令，也就是说在分组内定义的命令可以在分组外生效：

```
\def\a{1}
{\def\a{2} \a}  % 2
\a              % 1
```

```
\def\b{1}
{\gdef\b{3} \b} % 3
\b              % 3
```

1.3.2 \long 和 \outer 前缀

\def 定义的命令，使用时是不能分段的。如果需要用命令排版大段的东西，在 \def 前方加上 \long 前缀命令即可办到：

```
\long\def\longtext#1{Test long text: #1}
\longtext{Long text here.

Long text at the next line.}
```

事实上用 \newcommand 定义的命令可以用 \long \def 等效地定义出来：

```
% 以下两种定义方式完全等效
\newcommand\foo[4]{Test #1, #2

Test #3, #4}
\long\def\foo#1#2#3#4{Test #1, #2

Test #3, #4}
```

另外一个前缀 \outer 不太常用，它定义的命令只能在“外部”使用，不允许用来定义其它命令：

```
\outer\def\baz#1#2{Test #1 and #2.}
\baz{A}{B}
% \def\foo#1#2{\baz{#1}{#2}} 报错
```

1.3.3 \edef 和 \xdef

\def 定义的命令称为“宏”(Macro)。计算机术语中的“宏”起到一种模式匹配和替换的作用。TeX 将匹配宏参数并替换成定义内容的过程称为“展开”(Expansion)。

我们看一个极端的例子：

```
\def\foooo{Foo!}
\def\foooo{Hello,\foooo}
% 编译中会报 TeX capacity exceeds 错误
```

这个例子的原意是想将 \foooo 定义的内容用到新定义的 \foooo 中去，但是适得其反，不但没能用上之前定义的“Foo!”，反而造成了 \foooo -> Hello, \foooo -> Hello, Hello, \foooo -> . . . 的死循环。

这时候就要用到 \edef ——完全展开的定义。\edef 将命令的内容不断展开，直到命令中只存在字符和 TeX 原始命令为止。

```
\def\foooo{Foo!}
```

```
\edef\foooo{Hello,\foooo}
% \foooo 正确得到 Hello, Foo!
```

因为彻底展开，\edef 定义的命令内容，除与参数有关的部分之外，都是固定的：

```
\def\a{A}
\def\b{\a B}
\def\foo{\b}
\edef\foooo{\b}
% 此时展开 \foo 和 \foooo 都得到 AB
\def\a{AA}
% 此时展开 \foo 得到的是 AAB，而 \foooo 仍然是 AB
```

1.3.4 \let 命令

\def 会覆盖之前已有的命令。\let 能够将一个命令以“别名”的形式保存下来，从而在重定义时加以利用。引用刘海洋《LaTeX 入门》一书上的一个示例：

```
\let\oldemph=\emph % 两个命令之间的等号可以省略
\def\emph#1{\textbf{\oldemph{#1}}}
An \emph{important} command
```

1.4 脆弱命令和健壮命令

脆弱命令和健壮命令是 LaTeX 格式才有的概念。LaTeX 的许多功能（交叉引用、目录、索引、参考文献等）要借助辅助文件完成相应的功能，将一些复杂命令写入文件成了自然而然的需求。然而就像常识告诉我们的，越复杂的东西越容易出错，也就越脆弱。

1.4.1 活动参数和脆弱命令

有一部分 LaTeX 命令的参数被称为“活动参数”。字面意义上来讲，这些参数可能会在别处用到，所以称为“活动的”。最简单的例子，\chapter、\section 等命令会向辅助文件‘.toc’写入信息，而 \tableofcontents 则读取这些信息生成目录。

LaTeX 中的活动参数包括（可能不仅包括）：

- 图表浮动体环境使用的 \caption 命令的参数；
- 索引命令 \index 和文献引用命令 \cite 的参数；
- 涉及终端输入输出的命令 \typeout \typein 的参数；
- 章节命令 \chapter、\section 等命令的参数；
  （包括它们用到的写页眉页脚的 \markboth 和 \markright 命令的参数）
- \bibitem 命令的可选参数；
- 标题页 \thanks 命令的参数；
- tabular/array 表格环境格式定义中 @ 格式后的参数。

脆弱命令在这些活动参数的位置出现时，很可能使 LaTeX 出错。LaTeX 定义的许多带可选参数的命令[1]（包括带星号的命令）是脆弱命令。 定义环境的 \begin 和 \end 也是脆弱命令。

[1] 有一些带可选参数的命令被定义为健壮的。详见1.4.3小节。

活动参数有一个共同特征：在处理活动参数时要用到 \edef 级别的展开。TeX 中除了 \edef 命令之外，还会在写文件、往屏幕输出结果，以及处理页眉页脚（用到 \mark）时会将参数完全展开。

脆弱命令展开时会牵扯较多命令，因而出错。后文会简要分析带可选参数的命令展开的过程，对展开过程中哪一步会引起错误不去深究。

脆弱命令可以在使用时加上 \protect 前缀加以保护，例如在 \section 中使用脚注：

```
% \footnote 属于脆弱命令
\section{节标题\protect\footnote{标题中的脚注}}
```

1.4.2 定义健壮命令

健壮命令可以直接在活动参数中使用而不出错。比如所有涉及字体的命令都是健壮命令，可以用 \textit 直接去改变标题中的一部分字体为斜体而不出错：

```
\section{Test section with \textit{italic part}}
```

LaTeX 提供了定义健壮命令的 \DeclareRobustCommand 命令。它的用法十分类似于 \(re)newcommand，包括星号、可选参数等。\DeclareRobustCommand 在命令本身未定义时，相当于 \newcommand，而如果命令已定义，则相当于 \renewcommand。

我们可以看到，\DeclareRobustCommand 定义的带可选参数的命令是健壮的：

```
\DeclareRobustCommand*\myrobustcmd[3][text]{Test #1, #2 and #3}
\section{\myrobustcmd[A]{B}{C}}
\section{\myrobustcmd{more}{more}}
```

LaTeX 还提供了 \MakeRobust 命令将脆弱命令变成健壮的[2]，比如将 \footnote 命令变成健壮命令：

```
\MakeRobust\footnote
\section{Test section\footnote{Robust footnote here.}}
```

值得一提的是，通过 \newcommand 或 \renewcommand 人为定义的带可选参数的命令是健壮的。但是对于不带可选参数的命令则并没有变得健壮，如果参数里用了脆弱命令，一样是脆弱的。

对于用户来说，如果需要的话，应该始终用 \DeclareRobustCommand 或者 \MakeRobust 来定义健壮命令。

1.4.3 LaTeX 格式定义的健壮命令

LaTeX 格式定义了相当一部分健壮命令，其中控制字体的命令占大多数。

- 输出 LaTeX 错误和警告的命令：

  \GenericError \GenericWarning \GenericInfo

- 控制断行断页的命令：

  \\[3] \newline \vspace \, \hspace

[2] 该命令在 2015 年的 LaTeX 格式更新时加入。老版本 TeX 系统在使用之前须检查是否更新。

[3] 在表格、列表等环境中使用时，\\ 是脆弱命令，不过这些环境本身就是脆弱的。

\nobreakdashes \nobreakspace

- LaTeX 标志：\LaTeX \LaTeXe
- 文本/数学模式通用符号：

  \$ \{ \} \P \S _

  \dag \ddag \copyright \pounds \dots
- NFSS 字体命令：

  \fontencoding \fontfamily \fontseries \fontshape

  \fontsize \selectfont \linespread \mathversion

  *family *series *shape \em \normalfont

  \text* \emph \textnormal
- 旧字体命令：

  \rm \sf \tt \bf \sl \it \sc \cal \mit
- 数学命令

  \([4] \)[4] \[[4] \][4] \ensuremath \sqrt
- 盒子命令[4]：

  \makebox \framebox \savebox \parbox \rule \raisebox
- 上标和下标命令 [用于脚注等]：\textsuperscript \textsubscript
- 引用参考文献的基本命令：\cite
- 杂类：\MakeUppercase \MakeLowercase \TextOrMath[5]

1.5 深层挖掘“可选参数”

本节首次涉及到 LaTeX 的内部命令，绝大多数为带 @ 的命令。初次见到这些命令的用户往往会感到难以理解。事实上，LaTeX 命令是由反斜杠和其后的字母组成的。@ 本来不是字母，为了在命令中带 @，需要将其赋予字母的属性。

LaTeX 提供了命令 \makeatletter，望文生义，这个命令起到的作用就是将 @ 赋予字母的属性 (Make-At(@)-Letter)。另一个命令 \makeatother 令 @ 失去字母的属性 (Make-At-Other)，从而带 @ 的就不能再正常使用。而在宏包、文档类中，@ 的属性已经得到正确处理，所以可以放心使用带 @ 的各种命令。

这两个命令实际上改变了字符 @ 的类型码 (Category code)。关于类型码，后文会有较详细的讲述，读者也可以参考前言中推荐的资料。

本节内容对于刚上手的用户来说比较困难，从下一段开始到本节末尾都将标记为特殊的段落。

事实上 TeX 本身并没有所谓的“可选参数”，\def 就定义不出来带“可选参数”的命令，因为它要求参数严格按照定义的方式给定。本节的内容将会阐明，“可选参数”实际上是个“伪命题”。

[4] 该（组）命令在 2015 年的 LaTeX 格式更新时被定义为健壮命令。老版本 TeX 系统在使用之前须检查是否更新。

[5] 该命令在 2015 年的 LaTeX 格式更新时加入。老版本 TeX 系统在使用之前须检查是否更新。

1.5.1 自定义带可选参数的命令

LaTeX 提供了一个内部命令 \@ifnextchar 来判断一个命令后边是否带有某个字符。按照习惯，可选参数用一对中括号包裹，那么 \@ifnextchar 检测的字符也自然是‘[’了：

```
\makeatletter
\def\foo{\@ifnextchar [{\baz}{\bazz}}
\def\baz[#1]#2{Test \string\foo\ with #2 (and #1).}
\def\bazz#1{Test \string\foo\ with #1.}
\foo{some text}
% \foo 后没有中括号，展开成 \bazz
% 和后面的 {some text} 一起进一步展开
% 结果是 Test \foo with some text.
\foo[extra text]{some text}
% \foo 后面有中括号，展开成 \baz
% 和后面的 [extra text]{some text} 一起进一步展开
% 结果是 Test \foo with some text (and extra text).
\makeatother
```

多数情况下，带与不带可选参数的命令都交给一个命令统一处理，于是在 \@ifnextchar 中，不带可选参数的处理方式是把括号加上，和可选参数一样交给同一个命令：

```
\makeatletter
\def\foo{\@ifnextchar [{\baz}{\baz[]}}
\def\baz[#1]#2{Test \string\foo\ with #2 (#1).}
\foo{some text}
% 展开成 \baz[]{some text}
% 结果是 Test \foo with some text().
\foo[extra text]{some text}
% 展开成 \baz[extra text]{some text}
% 结果是 Test \foo with some text (extra text).
\makeatother
```

如果 \foo 的定义里 \baz[] 的中括号里不是空的，而是带一些字符或者命令，这些就构成了 \foo 命令可选参数的**默认值**。

\(re)newcommand 等命令能够定义带一个可选参数的命令。当我们需要定义带多个可选参数的命令时，\(re)newcommand 就无法满足需求。而合理使用 \@ifnextchar 命令就能够定义出带多个可选参数的命令，而且能够自由控制可选参数的位置，甚至也不一定要用中括号包裹可选参数，可以用小括号、竖线等等（LaTeX 中的特殊字符如大括号、反斜杠等不能拿来用）。来看一个复杂的例子：

```
% \foo 定义为有两个必选参数和三个可选参数的命令，
% 前两个可选参数在两个必选参数之间，用 || 和 () 包裹；
% 后一个可选参数在第二个必选参数之后，用 // 包裹
\makeatletter
\def\foo#1{\@ifnextchar|{\fooi{#1}}{\fooi{#1}||}}
\def\fooi#1|#2|{%
  \@ifnextchar({\fooii{#1}|#2|}{\fooii{#1}|#2|()}}
```

```
\def\fooii#1|#2|(#3)#4{%
  \@ifnextchar/{\fooiii{#1}|#2|(#3){#4}}%
  {\fooiii{#1}|#2|(#3){#4}//}}
\long\def\fooiii#1|#2|(#3)#4/#5/{%
  Mandatory params: #1, #4\par
  Optional params: |#2|, (#3), /#5/\par
}
\makeatother
\foo{Param1}{Param2}
\foo{Param1}|Option1|{Param2}
\foo{Param1}(Option2){}
\foo{Param1}|Option1|(Option2){Param2}/Option3/
\foo{}(Option2){Param2}/Option3/
```

我们可以注意到，命令里用到了 3 个可选参数，所以用到了 3 个辅助命令来处理，原命令和前两个辅助命令顺次“补全”每一个可选参数，最后的辅助命令接受所有参数。

1.5.2 带星号的命令

LaTeX 定义了多个带星号的命令，包括 \newcommand、\hspace / \vspace，等等。星号事实上也类似一种可选参数，LaTeX 用内部命令 \@ifstar 识别一个命令（及其参数之后）是否有星号，根据判断执行不同的命令：

```
\def\baz#1#2{\@ifstar{\starbaz{#1}{#2}}{\nonstarbaz{#1}{#2}}}
\def\starbaz#1#2{Starred \string\baz with params `#1' and `#2'.}
\def\nonstarbaz#1#2{Non-starred \string\baz with params `#1' and `#2'.}
```

1.5.3 \newcommand 和 \DeclareRobustCommand 的秘密

前文说到，多个可选参数的命令最终通常交给一个命令处理所有参数。\newcommand 定义带可选参数的命令 \foo 时，处理所有参数的命令形式为 \\foo （用 \csname \string \foo \endcsname 构造）：

```
\newcommand*\foo[3][text]{Test #1, #2 and #3}
\expandafter\show\csname\string\foo\endcsname
% 程序将中断并显示以下信息，非报错，按回车可继续
% > \\foo=macro:
% [#1]#2#3->Test #1, #2 and #3.
```

而\DeclareRobustCommand 真正起作用的命令是 \foo␣，设法在命令名中添加了一个空格进去（可选命令模式下是 \\foo␣）。因为在正常环境下，命令后的空格都会被弃去，不起作用，于是将 \foo␣ 写入文件后再读出来的命令还是 \foo。

```
\DeclareRobustCommand*\foo[3][text]{Test #1, #2 and #3}
\expandafter\show\csname\string\foo\space\endcsname
% 注意 \\foo 后面的空格
% > \\foo =macro:
% [#1]#2#3->Test #1, #2 and #3.
```

不过有一个细节：如果是如同 \$ 这样的单个字符的命令，后面的空格是不被忽略的，在写入文件时要是不加以处理就会带多余的空格进去。这点在\DeclareRobustCommand 里做了处理。

第二章 宏包和文档类架构

LaTeX2.09 时代就已经有了宏包的概念，用 \documentstyle 设置选项并调用宏包：

```
\documentstyle[twoside,epsfig]{article}
% 设置文档属性为双面，并调用 'epsfig' 宏包
% 'epsfig' 是旧宏包，不推荐使用，在此仅作为举例，下同
```

LaTeX 2ε 更进一步，将宏包和文档类区分，并厘清了 \documentstyle 里混淆的宏包和选项，将调用宏包的功能交给 \usepackage，并且宏包和文档类一样也可以使用选项：

```
\documentstyle[twoside]{article}
\usepackage{epsfig}
\usepackage[showframe]{geometry}
% 设置文档属性为双面
% 调用 'epsfig' 宏包、'geometry' 宏包（带 'showframe' 选项）
```

有了上一章的 \(re)newcommand 和 \def 等命令，我们有了初步自定义命令的能力。本章的内容让我们能够将这些自定义命令很好地封装利用。

LaTeX 随系统发布的帮助文档中，名为 *LaTeX 2ε for class and package writers* 的文档讲述了所有关于宏包和文档类的命令。本章将在其基础上给出更多示例，并分析可能出现的问题。读者可以在 TeX 发行版内找到这篇文档，作为参考，方法是在 Windows 命令提示符或 *nix 终端下键入 texdoc clsguide。

2.1 宏包和文档类的声明

一个宏包或文档类的开头通常有规范的声明：

```
\ProvidesPackage{dummypkg}[2016/01/01 v0.1 dummy package]
% 宏包名称后面的参数可选，
% 如果存在，开头必须是 YYYY/MM/DD 格式的日期，否则报错，后同
\ProvidesClass{dummycls}[2016/01/01 v0.1 dummy class]
```

而任意的辅助文件也有规范的声明：

```
% 宏包和文档类都有固定的扩展名 'sty'和 'cls'
% 任意文件则需自行加扩展名
\ProvidesFile{dummycfg.cfg}[2016/01/01 v0.1 dummy config]
```

\ProvidesPackage 等命令的参数应与文件名保持一致，在使用 \documentclass 和 \usepackage 时，按照给定的名称寻找文件；但如果文件名与 \ProvidesPackage 等命令的参数不一致，则会报 Warning。

\ProvidesPackage 等命令本身会在命令行和日志文件输出信息，提示宏包和文档类被调用；并且命令本身具有防止重复调用宏包/文档类的功能。

\ProvidesPackage 等命令会生成一个特殊的形如 \ver@dummypkg.sty 的宏，在调用宏包前会先检查这个宏是否被定义，从而避免重复调用。宏本身被定义为 \ProvidesPackage 等的可选参数。

如果宏包开头未声明 \ProvidesPackage 等，这个宏不会得到定义，也就有检测不到冲突的风险。

宏包或文档类的开头事实上可以用 \ProvidesFile 代替 \ProvidesPackage 等命令声明，但这是一个不规范的做法，会带来以下问题：

1. \ProvidesFile 不会在命令行输出信息，在日志文件里输出的信息也不规范；
2. 如果发生了文件名与参数不一致的问题，\ProvidesFile 不具备排查的功能。

事实上，如果宏包文件 dummypkg.sty 用 \ProvidesFile 作声明，并且参数错写成 dummypk.stx，那么得到定义的宏将是 \ver@dummypk.stx 而不是 \ver@dummypkg.sty，这可能带来一些隐患。相反，\ProvidesPackage 等总是用文件名定义这个特殊宏。

2.2 在宏包内使用宏包

类似在文档中用 \documentclass 和 \usepackage，宏包和文档类内部也可以使用其它宏包/文档类，对应的命令为：

```
\RequirePackage[option1,option2]{dummypkg}
% 后面还可以带一个可选参数检测宏包的版本，形如[2000/01/01]，下同
\LoadClass[option1,option2]{dummycls}
% 只能在文档类中使用，宏包中使用会报错
```

2.3 定义和处理选项

宏包和文档类的选项为我们带来了控制排版行为的简便途径。比如标准文档类 article 有控制纸张大小的 a4paper/letterpaper 等选项、控制全局字号的 10pt/11pt/12pt 选项，等等。本节内容将让读者了解传给宏包的选项是如何得到处理的。

本节所指的“给定选项”，包括通过 \PassOptionsToClass 等命令传递的选项，以及通过 \LoadClass 等命令使用的选项。“全局选项”指文档的 \documentclass 命令使用的选项，与宏包的给定选项一起处理，并且优先。

2.3.1 定义选项

\DeclareOption 定义选项，带星号形式的 \DeclareOption* 定义所谓的“缺省选项”，也就是没有定义的选项在处理时执行 \DeclareOption* 的定义：

```
% File: dummypkg.sty
\ProvidesPackage{dummypkg}
\def\option{}
\DeclareOption{one}{\def\option{Option 1}}
\DeclareOption{two}{\def\option{Option 2}}
\DeclareOption*{\def\option{Undeclared option}}
\ProcessOptions\endinput
```

\DeclareOption(*) 定义中使用的 \CurrentOption 指代“当前选项”，也就是在后文所叙述的 \ProcessOptions 命令处理的每个选项本身。

宏包和文档类都有默认的 \DeclareOption* 定义：宏包报 Undeclared Option 错误；文档类用 \OptionNotUsed 命令存入未使用选项的列表。由于文档类的选项是全局选项，在宏包

中有可能得到处理而从未使用选项列表中移除。如果到 \begin{document} 还存在未使用的选项，将报 Unused option(s) 警告。

2.3.2 传递选项给宏包

用 \RequirePackage 或 \LoadClass 调用的宏包/文档类可以从调用者的给定选项中获得选项。这个功能由 \PassOptionsToPackage 和 \PassOptionsToClass 完成：

```
% 若当前宏包的给定选项中包含 testa，传递 testa 选项给 dummypkg2
\DeclareOption{testa}{\PassOptionsToPackage{\CurrentOption}{dummypkg2}}
% 若当前宏包的给定选项中包含 testb，传递 testc 选项给 dummypkg2
\DeclareOption{testb}{\PassOptionsToPackage{testc}{dummypkg2}}
% 将当前宏包的给定选项中所有未定义选项传递给 dummypkg2
\DeclareOption*{\PassOptionsToPackage{\CurrentOption}{dummypkg2}}
```

2.3.3 处理选项

在 \DeclareOption(*) 定义选项完毕后，须用 \ProcessOptions 处理选项。如果不用，除了使用的选项本身不能被正确处理而报错外，还会令其后的宏包调用出问题。事实上从第一个 \DeclareOption 开始，一直到 \ProcessOptions 命令为止，其间**不允许使用 \RequirePackage 或 \LoadClass 等命令**。缺失 \ProcessOptions 将令后面的宏包全部无法调用，甚至波及到文档中的 \usepackage。

\ProcessOptions 有带星号和不带星号两个版本，处理的策略不一样：

- 不带星号的 \ProcessOptions 首先顺次遍历由 \DeclareOption 定义的所有选项，如果在给定选项（宏包中包括全局选项）中出现则执行，之后遍历给定选项对未定义的选项执行 \DeclareOption* 的缺省定义；
- \ProcessOptions* 直接顺次遍历给定选项（宏包中包括全局选项），如果选项有定义则执行定义，否则执行缺省定义。

不带星号的 \ProcessOptions 在处理有可能冲突的选项时稍显劣势。以标准文档类 book 为例：book 文档类有一对选项 oneside/twoside 控制页面呈现单面/双面样式，两种效果不能共存，应当二选一。而 book 文档类是用不带星号的 \ProcessOptions 命令处理选项的，twoside 在 oneside 之后。如果两个选项同时存在，起作用的是 twoside，这意味着无法使用 oneside 选项覆盖 twoside 选项。例如：

```
% File:mybook.cls
\ProvidesClass{mybook}
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{book}}
\ProcessOptions
\LoadClass[twoside]{book}
\endinput

% Main TeX file
\documentclass[oneside]{mybook}
% 传给 book 的选项是 oneside,twoside
% oneside 不起效果
```

有鉴于此，若某个宏包的多个选项存在效果相互冲突的情况，并且处理选项时使用不带星号的 \ProcessOptions ，则尽可能避免在其它宏包中带选项调用这个宏包。如果一定要带选项调用，在相互冲突的选项里应当使用第一个选项。使用 \ProcessOptions* 处理选项的宏包则不会出现上述问题。

2.3.4 预处理选项

\ExecuteOptions 命令用来直接执行某些选项。一般将其放在 \ProcessOptions(*) 之前，相当于缺省选项，而 \ProcessOptions(*) 在处理选项时可以覆盖缺省选项的效果。

2.3.5 以当前的选项调用宏包或文档类

LaTeX 的 \RequirePackageWithOptions 和 \LoadClassWithOptions 提供了一种特殊的调用宏包和文档类的方式。它将当前的宏包或文档类的给定选项原样传给要调用的宏包或文档类，之前的 \PassOptionsToClass 等命令的作用被忽略：

```
% File:mybook.cls
\ProvidesClass{mybook}
\DeclareOption*{\relax} % 不传递任何选项
\ProcessOptions
\LoadClassWithOptions{book}
\endinput

% Main TeX file
\documentclass[a4paper,12pt]{mybook}
% 相当于：\documentclass[a4paper,12pt]{book}
```

宏包和文档类的选项由形如 \opt@dummypkg.sty 的宏以逗号列表的形式存储。\PassOptionsToPackage 和 \RequirePackage 在这个宏不存在的时候构造之，否则向已有列表追加新的选项。文档类相关的宏命令同理。

调用宏包时如果检测到重复调用（\ver@dummypkg.sty 已定义），会进一步检查给定选项是否已在 \opt@dummypkg.sty 中，如果有选项不在这个宏中，则报 Option clash 错误。有一些宏包避开了这种错误，例如 fontenc 宏包每次调用时都会将 \opt@fontenc.sty 和 \ver@fontenc.sty 清空，使这个宏包表现得像是未调用过一样。

2.4 用键值对形式定义选项

在相当多的编程语言中，键值对是一种常用的为数据赋值或定义对象属性的方式。LaTeX 宏包中也常常让用户可以使用键值对的方式使用选项，这大大扩展了宏包功能的可定制性。不过，LaTeX 并没有原生处理键值对的能力，键值对都是依赖一些宏包来处理的，如 keyval / xkeyval，Oberdiek 套件中专门用于为宏包处理选项的 kvoptions，PGF/TikZ 组件之一的 pgfkeys 等。

LaTeX3 原生支持键值对形式，功能更加复杂多变，且通过 l3keys2e 宏包支持处理 LaTeX 2ε 宏包/文档类的选项。附录的 B.6 节介绍了 LaTeX3 键值对的用法，在此不做赘述。

本节将简单介绍基础的 keyval 宏包。对其他宏包感兴趣的读者可查询各自的说明文档。

2.4.1 keyval 宏包

keyval 宏包引入了键值族的概念，键值对交给一个“族”处理，在这个“族”里处理各个键值：

```
% 定义 keyGroup 下各个键的处理方式
% 键值对的值作为参数 #1
\define@key{keyGroup}{keyfoo}{\def\foo{#1}}
% 带默认值的键
\define@key{keyGroup}{keybaz}[bazdefault]{\def\baz{#1}}

% 将键值对传给 keyGroup 族处理
% 多个键值对之间用逗号隔开
% 键值对的值本身用到逗号的，应用大括号包裹
\setkeys{keyGroup}{keyfoo=valuefoo,keybaz={valuebaz1,valuebaz2}}
\meaning\foo % macro:->valuefoo
\meaning\baz % macro:->valuebaz1,valuebaz2

\setkeys{keyGroup}{keybaz}
\meaning\baz % macro:->bazdefault
```

在宏包选项中使用的键值对，一般都通过 \DeclareOption* 传递给键值族：

```
% 宏包文件 dummypackage.sty
\ProvidesPackage{dummypackage}[2016/01/01]
% keyval 宏包必须在 \DeclareOption 之调用
\RequirePackage{keyval}
% 提供 \usepackage 之后修改选项的接口
\def\setpackage#1{\setkeys{dummykey}{#1}}
\define@key{dummykey}{testfoo}{\def\foo{#1}}
\define@key{dummykey}{testbaz}{\def\baz{#1}}
% testfoo 和 notestfoo 提供非键值对形式的选项
\DeclareOption{testfoo}{\setkeys{dummykey}{testfoo=true}}
\DeclareOption{notestfoo}{\setkeys{dummykey}{testfoo=false}}
\DeclareOption*{\setkeys{dummykey}{\CurrentOption}}
\ProcessOptions\relax
\endinput

% 主文件
\usepackage[notestfoo,testbaz=bazzz]{dummypackage}
% \foo 定义为 false，\baz 定义为 bazz
\setpackage{testfoo=true,testbaz=bazzzzz}
% \foo 定义为 true，  \baz 定义为 bazzzzz
```

2.5 自定义错误和警告

第三章 使用变量

3.1 LaTeX 计数器

LaTeX 封装了一种称作“计数器”(Counter) 的结构，并定义了相关命令。LaTeX 计数器有以下特点：

1. 计数器的值是全局的，在任何地方对计数器的改动都有效；
2. 计数器之间可以以上下级的方式关联，上级计数器步进可以使得下级计数器归零。

LaTeX 对计数器定义的命令有以下这些：

```
\newcounter{foo}          % 定义一个计数器 foo，初始值为零
                          % 定义前检查是否被定义过
\setcounter{foo}{3}       % 令 foo 的值为 3
\addtocounter{foo}{4}     % 另 foo 的值加 4，现在为 7
\newcounter{baz}[foo]     % 定义一个计数器 baz 为 foo 的下级计数器
\setcounter{baz}{15}      % 另 baz 的值为 15
\stepcounter{baz}         % 令 baz 的值加 1，现在为 16
\stepcounter{foo}         % 令 foo 的值加 1，现在为 8；将 baz 的值归零
```

对于一个计数器 foo，`\newcounter` 顺便定义了一个 `\thefoo` 以特定格式显示计数器的值，默认以数字显示。LaTeX 还定义了一系列命令用于设定计数器数值的显示格式，假如 foo 计数器的值为 4，各种格式的命令为：

- `\arabic{foo}` 显示为 4（`\thefoo` 默认格式）；
- `\roman{foo}` 显示为 iv；
- `\Roman{foo}` 显示为 IV；
- `\alph{foo}` 显示为 d；
- `\Alph{foo}` 显示为 D；
- `\fnsymbol{foo}` 显示为 §。

另外还有命令 `\value{foo}`，表示计数器内部使用的数值寄存器，不能直接用于显示（直接用 `\value{foo}` 会报 `Missing number, treated as zero` 错误）。数值寄存器的详情会在下一节阐述。

用户可以重定义 `\thefoo` 给计数器 foo 自定义格式，仍假如 foo 计数器的值为 4，另假定 baz 计数器的值为 8：

```
% 显示为 C4
\renewcommand*\thefoo{C\arabic{foo}}
% 将 foo 和 baz 计数器的值结合显示成 C4.8
\renewcommand*\thebaz{\thefoo.\arabic{baz}}
```

LaTeX 格式及标准文档类预定义了一些计数器，罗列如下，其中 foo ← baz 表示 baz 为 foo 的下级计数器，斜线前为 book/report 文档类的情况；斜线后为 article 文档类的情况：

- 标题计数器，与标题命令名称一致：

part, chapter ← section ← subsection ← subsubsection ← paragraph ← subparagraph

- 列表环境计数器：

  enumi, enumii, enumiii, enumiv
- 公式计数器：(chapter ←)equation/equation
- 脚注计数器：footnote, mpfootnote
- 页码计数器：page
- 控制章节编号和目录的计数器：secnumdepth, tocdepth
- 控制图表浮动体数量的计数器：

  topnumber, bottomnumber, totalnumber, dbltopnumber
- 图表计数器：

  (chapter/section ←)table, (chapter/section ←)figure

3.2 数值寄存器

上一节的计数器是一种封装结构，其数值是存储在寄存器中的。本节我们将接触到更底层一些的数值寄存器的操作。

3.2.1 TEX 数值的表示方式

数值的表示除使用十进制数字外，还可用八进制、十六进制或者 ASCII 字符方式：

十进制	123	八进制	'173
十六进制	"7B	ASCII	`\{

数值 123 的各种表示方式

上表最后一种方式为 ASCII 字符方式，数值为字符的 ASCII 码。在这种写法里，如果字符是普通的字母等（catcode 为 11 或者 12），字符前可以不加 \，否则一般要加。

两个定义字符的命令 \chardef 和 \mathchardef 也用来定义数值（常数）。用它们定义的命令既能用来输出字符，又能够用来当作数值使用。两个命令定义的数值范围分别是 0~255 和 0~32767。\mathchardef 的用法可在附录中的 A.2.3 小节一瞥。

```
\chardef\const=`A              % \const = 65
Once Upon \const\ Time         % Once Upon A Time
\newcounter{baz}
\setcounter{baz}{2}
\addtocounter{baz}{\const}
\arabic{baz}                   % 67
\mathchardef\mathconst="027A % \mathconst = 634
\setcounter{baz}{\mathconst}
\Roman{baz}                    % DCXXXIV
$\mathconst$                   % 符号 ‡
```

LATEX 以及 Plain TEX 定义了一些内部命令用作常数，有个别寄存器也被用作常数，参考 macros2e.pdf，这里不再赘述。

3.2.2 数值寄存器的赋值和计算

数值寄存器可以用编号访问，也可以定义成一个命令来访问，类似于“别名”：

```
% 数值寄存器 0 的值为 22
\count0=22
% \countfoo 定义为数值寄存器 0
\countdef\countfoo=0
% 数值寄存器 0 (\countfoo) 的值现在为 24
\countfoo=24
```

Plain TEX 和 LATEX 提供了 \newcount 命令，用于自动分配一个寄存器编号并让我们定义成命令：

```
\newcount\countbaz
% 假若分配的寄存器号为 36，相当于 \countdef\countbaz=36
% 还会在日志文件上记录 \countbaz=\count36
```

数值寄存器可以直接用代表数字的字符串赋值，也可以用其他寄存器为其赋值，在字符串或者其他寄存器之前可以加正负号。仍以之前定义的 \countfoo (\count0) 和 \countbaz (\count36) 为例：

```
\countbaz=3           % 寄存器 36 的值为 3
\countfoo=\countbaz   % 寄存器 0  的值为 3
\count0=22            % 寄存器 0  的值为 22
\countfoo=-\count36   % 寄存器 0  的值为 -3
```

值得一提的是，用代表数值的字符串赋值时，会将后边的宏命令展开，如果展开的结果有可以组成数字的字符，也一并用来组成数字，直到出现空格、不可展开的命令或者不能用来组成数字的字符为止：

```
\def\baz{A}
\countbaz="1\baz C % \countbaz 的值为 0x1AC(428)
\countbaz=1\baz C  % \countbaz 的值为 1
% A 和 C 不能组成十进制数，作为正常字符被排版
```

使用代表数值的字符串时，通常会在其后**加上空格或 \relax 以防止额外的展开**。

数值寄存器可以用 \advance、\multiply 和 \divide 命令进行运算，寄存器本身为被加数 (被减数)、被乘数或被除数，运算的结果重新存入寄存器。加数 (减数)、乘数或除数和赋值的时候类似，可以用代表数值的字符，也可以用其他寄存器：

```
% 以下命令中的 by 可加可不加
\countfoo=22 \countbaz=3
\divide\countfoo by 4   % 结果取整，\countfoo 的值为 5
\advance\countbaz by -4 % 减去数等于加上相反数
                        % \countbaz 的值为 -1
\multiply\countfoo\countbaz % \countfoo 的值为 -5
```

LATEX 计数器(仍然以 foo 为例)内部使用的寄存器被定义为 \c@foo。另有一个列表 \cl@foo 记录 foo 计数器的所有下级计数器。

\stepcounter{foo} 先使下级计数器的值为 -1，然后对下级计数器用 \stepcounter，将其归零的同时，令更下一级计数器按照同样的方式归零[1]。

数值寄存器的数值是 32 位带符号整数，范围为 -2147483648~2147483647。TEX 在赋值和乘法运算时会检查越界，用绝对值超过 2147483647 （包括 -2147483648）的整数给数值寄存器赋值，或乘法运算的结果超过 2147483647，都会导致 TEX 报错。

3.2.3 数值的输出

数值寄存器的值需要通过 \number 等命令为代表数值的字符串才能输出。\number 命令将数值转换为带正负号阿拉伯数字形式的字符串；\romannumeral 命令将数值转换为罗马数字形式的字符串，如果数值为零或负数，输出的字符串为空。\the 命令可用于多种变量的输出，对于数值寄存器和 \number 一致。

```
\countfoo=112
\number\countfoo       % 112
\romannumeral\countfoo % CXII
\the\countfoo          % 112
```

\number 和 \romannumeral 命令后面也可以跟代表数值的字符串，它们也遵循上一小节所述的展开规律：

```
\def\baz{A}
\number"1\baz C % 428
\romannumeral 1\baz C  % iAC
```

3.3 长度和弹性长度

3.3.1 LATEX 封装的长度命令

LATEX 也对长度变量的定义和计算进行了简单的封装，相比于计数器结构要简单得多：

```
\newlength\testlength          % 定义长度 \testlength
                               % 定义前检查是否已定义
\setlength\testlength{20pt}    % 令 \testlength 的值为 20pt
\addtolength\testlength{-3em} % 令 \testlength 的值减 3em
```

\newlength 封装的长度事实上是弹性长度。有关弹性长度的内容详见 3.3.4 小节。

3.3.2 TEX 长度的表示方式

长度变量用代表数值和单位的字符串表示，数值用阿拉伯数字和小数点（可以用逗号代表小数点）表示，单位是 TEX 基本单位 (pt / bp / cm / mm / in / pc / dd / cc / sp) 或是与当前字体有关的单位 (em / ex)。另外，长度也可以表示成其它长度寄存器的“倍数”。

TEX 提供了 \mag 命令，用于全局地放大或缩小所有的长度，如 \mag2000 会让 1cm 的长度实际代表 2cm。真实的长度可在单位前加前缀 true 表示，如 5 truecm 代表真实的 5cm。

[1] 最早 \stepcounter 只令下级计数器的值为 0，起不到归零更下一级计数器的作用。LATEX 在 2015 年的改动修复了这个问题。

3.3.3 长度寄存器

类似数值寄存器，长度寄存器的定义有 \dimendef 以及 \newdimen 命令：

```
\dimendef\dimenfoo=6
\newdimen\dimenbaz
\dimen6=4pt
\dimenbaz=2.2\baselineskip
```

长度寄存器的各类计算与数值寄存器类似，\advance 里的加数是长度，\multiply 和 \divide 里的乘/除数仍然是数值（整数）：

```
\dimenfoo=3pt
\advance\dimenfoo 2.5pt % \dimenfoo=5.5pt
\divide\dimenfoo by 2   % \dimenfoo=2.75pt
```

3.3.4 弹性长度

弹性长度带有“伸展”和“收缩”的部分，以调整环境尽量和所需的长度吻合。在下一章中会详细讨论弹性长度在盒子中的使用。

弹性长度的表示方式分为三部分——固定长度、伸展长度和收缩长度，其中后两者是可选的。每个部分的表示方式和一般的长度一致，而伸展长度和收缩长度还能用特殊的单位 fil, fill 和 filll 代表不同级别的“无限伸展”和“无限收缩”。

类似地，弹性长度的定义有 \skipdef 以及 \newskip 命令：

```
\skipdef\skipfoo=4 % \skipfoo=\skip4
\newskip\skipbaz
\skipfoo=12pt
\skipbaz=4.5\skip4
\skipfoo=12pt plus 2pt minus 2pt
\skipbaz=0pt plus 1fil
```

弹性长度的加法计算将三部分对应相加，伸展长度和收缩长度如果是普通长度和无限长度相加，或者不同级别的无限长度如 fil 和 fill 相加，相加结果是保留高级别的无穷长度，而舍弃低级别的无穷长度或普通长度。弹性长度的乘法和除法计算会对三部分同时相乘/相除。

sp 是最小的长度变量，65536sp=1pt。TEX 里所有长度变量都是 1sp 的倍数，使用其它单位定义的长度变量也会转化为使用 sp 为单位。事实上长度变量也是用 32 位无符号整数存储的，数值等于以 sp 为单位的长度。这意味着长度变量的值可以赋给数值变量，反之亦然；并且长度变量的值也可以用 \number 显示，或用于 \multiply / \divide 等。

```
\dimenfoo=0.5pt
\countfoo=\dimenfoo
\the\countfoo    % 32768
\number\dimenfoo % 32768

\dimenfoo=4.5\countfoo
\the\dimenfoo    % 2.25pt
```

```
% \p@=1pt，对应的数值是 65536
% 以下运算相当于以 pt 为单位对 \dimenfoo 取整
\divide\dimenfoo by \p@ \multiply\dimenfoo by \p@
\the\dimenfoo    % 2pt
```

弹性长度变量也可以赋值给长度变量或数值变量，用其固定长度部分赋值；反之长度变量和数值变量也可以赋值给弹性长度变量里的任意一个部分。

长度的范围是 ± 1073741823sp(16383.99999pt)。LaTeX 专门有长度寄存器 \maxdimen 储存允许的最长长度。用超过范围的长度给长度寄存器赋值，或乘法运算的结果超过范围，都会使 TeX 报错。

第四章 控制盒子

第五章 TEX 宏编程

5.1 TEX 中的记号

5.1.1 字符类别

5.2 宏定义和展开控制

5.2.1 再探宏定义

5.2.2 \expandafter 展开

\expandafter 是 TEX 宏语言里最为“黑科技”的一个命令，对于新手来说也是难以操控的命令。我们先以最简单的例子说明用途：前面我们知道用 \csname 以及 \endcsname 可以构造一个通常情况下写不出来的命令，但我们要怎么样用 \def 将这个命令定义成一个宏呢？读者可能会试图尝试：

```
\def\csname 123\endcsname{...}
```

但是这样写的话 \def 会将 \csname 定义为一个宏，参数有 1,2,3 和 \endcsname。不但我们想要的命令 \123 还是没能得到定义，还影响到了 \csname 命令以后的正确使用。

这时候 \expandafter 就派上用场了：

```
\expandafter\def\csname 123\endcsname{...}
```

\expandafter 会搁置 \def，将 \csname 123 \endcsname 展开成我们想要的命令 \123，然后再在展开结果前放上 \def。这样，我们构造出来的命令 \123 就能正确被 \def 定义。

我们在此归纳几个 \expandafter 常用的场景和使用技巧：

1. 将自定义的命令用于 \def 和 \let 等。

前文已经举例说明。另外更常见的用法是配合 \ifx 检验一个命令是否得到定义（关于条件判断的具体用法见下一节）。由于 \csname ... \endcsname 组成的命令如果本身不存在，则等同于 \relax，这样可以用 \ifx 来比对：

```
\expandafter\ifx\csname testmacro\endcsname\relax
  Not defined!
\else
  Defined!
\fi
```

当然如果某个命令存在，但被 \let 给了 \relax，这里的比对结果将会是"Not defined!"。不过这种极端的例子很罕见。

2. 在分组内定义命令并在分组外使用。

\global 前缀可以定义分组之外使用的全局命令。但如果我们希望这个命令的定义能用到分组外，而命令本身被局限在分组内不对外界产生影响，则需要在分组结束前展开这个命令：

```
\def\foo{Foo!}
\foo % Foo!
\begingroup
  \def\foo{Bar!}
\expandafter\endgroup\foo  % Bar!
\foo % Foo!
```

3. 延迟展开多个记号。

其中一个例子是展开命令定义式中的一部分。当命令定义式需要展开一部分，而不需要 \edef 时，如下的展开是有用的：

```
\def\foo{Foo!}
\def\baz{\foo}
\expandafter\def\expandafter\foofoo\expandafter{%
  \expandafter\foo\baz}
% 跳过 \def,\foofoo,{,\foo 将 \baz 展开，
% 完整的定义式为 \def\foofoo{\foo\foo}
% 注意定义式中的左括号也要跳过
```

4. 将一个记号之后的内容展开多次。

```
\def\aaa{AAA} \def\bbb{\aaa}
\def\ccc{\bbb} \def\ddd{\ccc}
\let\ea\expandafter

\ea\string\ddd                  % 展开一次，输出 \ccc
\ea\ea\ea\string\ddd            % 展开两次，输出 \bbb
% 首先按照上一条的规律展开成  \ea\string\ccc
% 然后进一步展开得到 \string\bbb
\ea\ea\ea\ea\ea\ea\ea\string\ddd % 展开三次，输出 \aaa
```

一个经验式是：**某个记号之前的 $2^n - 1$ 个 \expandafter 将这个记号其后的内容展开 n 次**。

由此甚至可以衍生出“逆序展开”：

```
\let\ea\expandafter
\ea\ea\ea\ccc\ea\bbb\aaa
% 先展开 \aaa 再展开 \bbb 最后展开 \ccc
```

5.3 条件判断

第六章 排版控制

6.1 垂直间距

6.1.1 基线间距与行间隙

在描述文字的行与行之间的距离时，很多人喜欢说“行距”。但这不严格，因为行的位置需要有个参照。在 TEX 中，这个参照就是“基线”。TEX 的 \baselineskip 就是各行基线的间距。本小节的“行”指代任何排版在正文中的水平盒子，包括段落生成的文字行、行间公式、图表浮动体等等。

基线间距在排版时需要转换为行与行之间的间隙。TEX 在处理行间隙时除了用到 \baselineskip，还有两个参数 \lineskiplimit 和 \lineskip：排版完一行后，TEX 先将最后一行的深度存储在 \prevdepth 中；在下一行生成后，TEX 计算两行盒子之间的间隙（\baselineskip - \prevdepth - 下一行高度），如果间隙正常，它就是最终的行间隙；如果某个盒子的高度或深度很大，使得计算出来的行间隙太小（<\lineskiplimit），则放弃这个间隙，改用 \lineskip 充当行间隙。

一个例外是：**如果在排版完一行后人为设定 \prevdepth=-1000pt，这一行和其后的一行之间不会插入任何行间隙。** \nointerlineskip 命令就利用这种方式去掉行间隙。

LATEX 的基线间距 \baselineskip 与字号的设定有关（详见附录中的 A.1.3 小节），如果文档类用默认的 10pt 属性，且 \baselinestretch 为默认值 1 ，那么 \baselineskip=12pt；\lineskiplimit 和 \lineskip 分别默认为 0pt 和 1pt。

另外关于行间隙还有若干要注意的地方：

1. 行间隙与行间插入的任何间距 (skip) 是相互独立的。

比方说，两行之间用 \baselineskip 计算得到的行间隙为 6pt，那么在两行之间插入一个 20pt 的 \vskip，最终的行间隙将为 26pt，且 6pt 的行间隙在后一行的上方，在 20pt 的 skip 下方。

2. 水平画线命令 \hrule 和前后的行（包括其他的 \hrule）之间不会插入任何行间隙。

\hrule 会紧贴着上一行或者上一个 \vskip 排版，并且将 \prevdepth 直接修改成 -1000pt，使得下一行不存在行间隙。如果 \hrule 前后直接是两行，没有额外间隙的话，效果就是横线和两行互相紧贴。要为 \hrule 前后增加间距，要么在其前后人为插入 \vskip，要么采取修改 \prevdepth 的方法：在 \hrule 之前将 \prevdepth 保存到一个变量中，在 \hrule 之后、下一行之前将 \prevdepth 修改为保存的值。

另外，每页第一行之前有一个特殊的行间隙，由 \topskip 减去第一行的高度得到。如果第一行高度超过 \topskip，这个行间隙则为 0。\topskip 的默认值为 10pt。

6.1.2 LATEX 中的垂直间距

除了行间隙，为了内容划分、排版美观等，LATEX 还引入了许多垂直间距。其中有些间距是弹性长度，在生成页面时能够伸展，以调整页面内容的高度。

有许多间距是与文档的字号设定有关的，在以下的列表项中用 † 标记。本文给出的间距是标准文档类 10pt 选项的设定。

段落间距 \parskip

每次新起一段时，在段落的第一行上方加入这个间距。LaTeX 默认的大小是 0.0pt plus 1.0pt。它是页面里最主要的弹性伸展长度。

† 行间公式与上下文的间距

行间公式和上下文之间用 \abovedisplayskip 和 \belowdisplayskip 的间距隔开；如果前一行的文字很短（短于公式左端到版心的距离），则改用 \abovedisplayshortskip 和 \belowdisplayshortskip。

\abovedisplayskip	10pt plus 2pt minus 5pt
\belowdisplayskip	10pt plus 2pt minus 5pt
\abovedisplayshortskip	0pt plus 3pt
\belowdisplayshortskip	6pt plus 3pt minus 3pt

图表浮动体之间的间距

\floatsep	12pt plus 2pt minus 2pt	t/b 位置浮动体之间的间距
\textfloatsep	20pt plus 2pt minus 4pt	t/b 位置浮动体和正文的间距
\intextsep	20pt plus 2pt minus 4pt	h 位置浮动体和上下文的间距
\dblfloatsep	12pt plus 2pt minus 2pt	t 位置跨栏浮动体之间的间距
\dbltextfloatsep	20pt plus 2pt minus 4pt	t 位置跨栏浮动体和正文的间距
\@fpsep	0pt plus 1fil	p 位置浮动体与页面顶部的间距
\@fptop	8pt plus 2fil	p 位置浮动提之间的间距
\@fpbot	0pt plus 1fil	p 位置浮动体与页面底部的间距

图表标题和上下文的间距

\caption 命令生成的图表标题前后会分别加 \abovecaptionskip 和 \belowcaptionskip。默认值分别为 10pt 和 0pt。也就是说，\caption 命令如果在图表下方，则有 10pt 的间距；在图表上方，则贴着图表。

† 列表环境内部、列表环境和上下文的间距

LaTeX 定义的 enumerate / itemize 列表环境默认的列表环境引入了大量的垂直间距，而且难以修改。下表为一级列表的一些参数：

\parsep	4pt plus 2pt minus 1pt	列表内的 \parskip
\topsep	8pt plus 2pt minus 4pt	列表和上下文的间距
\itemsep	4pt plus 2pt minus 1pt	\item 段落与之前列表段落的额外间距 （首个 \item 没有这个额外间距）
\partopsep	2pt plus 1pt minus 1pt	如果上文已分段，在 \topsep 上额外增加的间距

用户可直接使用 list 环境自定义一个列表，并修改上表的参数；用户也可以用 enumitem 宏包提供的修改 enumerate / itemize 列表环境各类参数的接口。

center / flushleft / flushright 环境和上下文的间距

这三个对齐的环境，内部实现是生成了一个简单的列表（trivlist 环境），所以也会和上下文之间产生间距（\topsep，\partopsep 等）。

LaTeX 定义的一些特殊环境也是用 trivlist 环境生成的，包括抄录环境 verbatim、制表位环境 tabbing 和用 \newtheorem 定义的定理环境等。这些环境和上下文的间距与 center 等环境类似。

章节标题与上下文的间距

LaTeX 的章节命令 \chapter、\section 等生成的章节标题和上下文环境通常有不小的间距。直接修改这些间距不太方便，需要重新定义生成章节的一些内部命令如 \@makechapterhead、\@startsection 等。ctex 宏包提供了一些修改上下文间距，以及修改标题格式的接口。更完善的接口由 titlesec 提供，它相当于重写了各个章节命令的实现。

章节命令	标题之前的间距	标题之后的间距
\chapter	50pt	40pt
\section	3.5ex plus 1ex minus 0.2ex	2.3ex plus 0.2ex
\subsection	3.25ex plus 1ex minus 0.2ex	1.5ex plus 0.2ex
\subsubsection	3.25ex plus 1ex minus 0.2ex	1.5ex plus 0.2ex
\paragraph	3.25ex plus 1ex minus 0.2ex	N/A
\subparagraph	3.25ex plus 1ex minus 0.2ex	N/A

注 1：\paragraph 和 \subparagraph 的标题是嵌入段落中的，标题后没有垂直间距，代之以水平间距 1em。

注 2：\chapter 的章号码和章名称分两行，之间有 20pt 的距离；\chapter* 没有章号码以及 20pt 的垂直间距。

注 3：\chapter 标题前的间距是用 \vspace* 实现的，它使得页面顶部的 \topskip 被保留，加上章标题自身的行间隙，实际上看起来比 50pt 要大得多。

† 脚注之间、脚注与正文的间距

6.2 水平距离和缩进

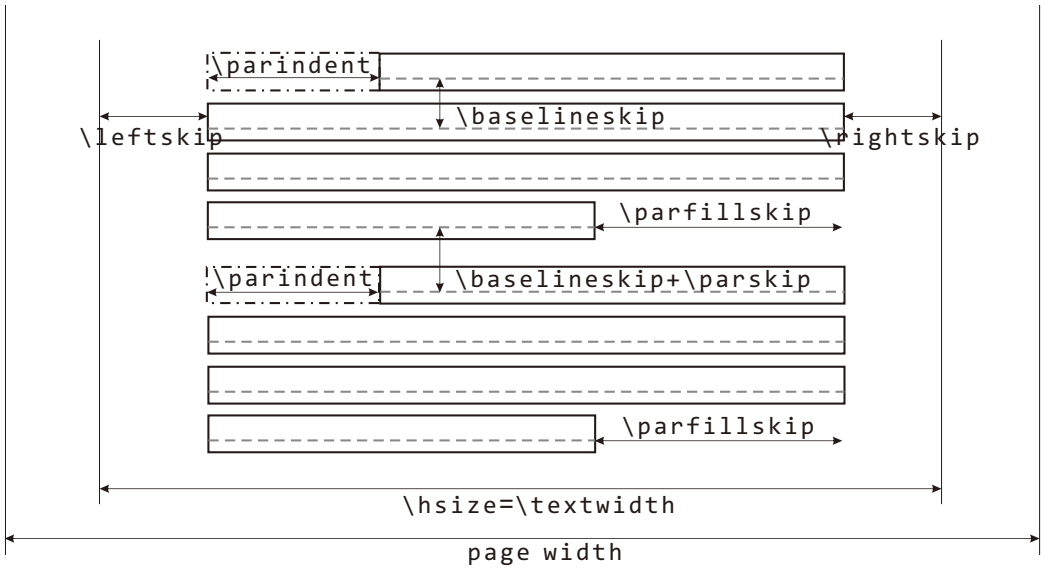

图 6.1: 段落的水平和垂直间距示意图

6.3 断行和断页控制

附录 A LaTeX 字体框架 (NFSS)

LaTeX 2.09 版本以及 Plain TeX 格式使用老式的字体切换方式，如 \bf，\it 等，字体之间没什么关联性，比如要得到粗斜体必须自行定义一个 \bfit 之类的命令。即使是修改字号，也是一件伤筋动骨的事情。

LaTeX 2_ε 引入了一套新的框架，称为新字体选择框架 (New Font Selection Scheme, NFSS)。这套框架将字体的属性分解成若干个正交的“坐标”，我们往往只需要切换其中一个或者几个“坐标”就可以切换字体。事实上 LaTeX 2_ε 格式的源代码里 NFSS 相关的代码占据了约三分之一，足见这套框架的重要性。

本章的内容主要用于字体宏包的开发，其他宏包较少涉及，它必须与具体的字体开发相结合。所以本章的内容作为附录放在最后，但不代表它是最不重要的。

A.1 文本字体框架

A.1.1 字体编码

TeX 的传统字体支持编码 256 个字符。除常规的 ASCII 可打印字符范围 (0x20–0x7E) 外，其他区域可以编码用于西欧语言的拉丁字母扩展、用于俄文的西里尔字母、希腊字母等等。甚至就像数学字体那样可以任意编码符号。这样的符号集合以及与 8 位二进制码的映射称为“字体编码”。

系统中的 encguide.pdf 文档举例介绍了常见的 LaTeX 字体编码，在此简单归类：

- 基本和扩展拉丁字母编码：OT1 T1 LY1
- LaTeX 符号编码：TS1
- 西里尔字母编码：OT2 T2A T2B T2C X2
- 数学字体编码：OML OMS OMX
- 国际音标字体编码：OT3 T3 TS3
- babel 宏包相关的用于各语言的字体编码：
  LAE LFE（阿拉伯和波斯语） LGR（希腊语） LCT（藏语） LHE（希伯来语）
- fontspec 宏包用于 Unicode 字体的编码：
  EU1（用于 XeTeX） EU2（用于 LuaTeX）

某种字体编码在使用之前必须由 \DeclareFontEncoding 定义。LaTeX 提供了 fontenc 宏包，用于载入与编码相关的 *enc.def 文件并执行其中的 \DeclareFontEncoding。

值得一提的是特殊编码 U，它在 LaTeX 格式里直接由 \DeclareFontEncoding 定义，没有自己的 def 格式文件。这个编码用于所有自定义的字体，在各种符号包中使用，包括 $\mathcal{AMS}$ 数学符号、METAFONT 的 logo 字体等等。

*enc.def 还用 \DeclareFontSubstitution 命令定义了该编码下处理字体回退时用到的属性 (family / series / shape)，并且用 \DeclareTextSymbol、\DeclareTextAccent 等命令定义了大量在该编码下使用的符号命令和重音等：

```
% 以 T1 字体编码为例（摘录于 t1enc.def 文件）
```

```
\DeclareTextSymbol{\i}{T1}{25}
% \i（不带点的 i）相当于 \char25
\DeclareTextAccent{\"}{T1}{4}
% \"（两点重音）相当于 \accent4，用于任意字符
\DeclareTextComposite{\"}{T1}{o}{246}
% \"{o} 定义为独立的 \char246 字符，而不用上述重音
\DeclareTextCommand{\k}{T1}[1]
   {\hmode@bgroup\ooalign{\null#1\crcr\hidewidth\char12}\egroup}
% \k （“小尾巴”重音）定义为利用 \char12 字符的一个宏
\DeclareTextCompositeCommand{\k}{T1}{o}{\textogonekcentered{o}}
% \k{o} 定义为独立的命令，而不用上述的 \k 宏命令
```

LaTeX 格式还用 \DeclareTextCommandDefault 等命令定义了一些缺省的符号命令。用 fontenc 宏包或者 \fontencoding 命令切换编码时，旧编码下有一些符号命令可能未在新编码下定义过。使用旧编码的符号命令时，会检查符号命令是否有缺省的定义，如果没有的话报错。

在此应区分“输入编码”和“字体编码”的区别。“输入编码”是 TeX 代码所使用的编码，也就是输入的文本符号在计算机中存储的 8-bit 二进制码，如 ISO 8859-1(Latin1)、Windows-1252、UTF-8 等等。“字体编码”是 TeX 字体中符号所对应的编码。对于绝大多数文本字体，在 0x20–0x7E 范围的字体编码和各种输入编码是一一对应的[1]，而在其范围之外大多数时候没有对应关系。

未经处理的情况下，位于 0x80–0xFF 范围的 8-bit 输入编码，catcode 均为 12，TeX 会按照其编码直接访问字体，但由于输入编码和字体编码不对应，通常得不到想要的输出。LaTeX 借助 inputenc 宏包对输入的编码进行正确的处理。对于 0x80–0xFF 范围的 8-bit 输入编码，inputenc 宏包调用相应的编码表将输入编码转为符号命令，再由 fontenc 宏包调用的字体编码定义文件 *enc.def 转为具体的字体编码或者宏命令。

A.1.2 字体族及样式属性

实际上字体属性的“坐标”并不是空穴来风的，每一组“坐标”都要对应一个具体的字体。这种对应关系由字体定义文件 (Font Definition, *.fd) 给出。

字体定义文件中主要的内容为 \DeclareFontFamily 以及 \DeclareFontShape 命令：

```
% 摘自 ot1cmr.fd
\DeclareFontFamily{OT1}{cmr}{\hyphenchar\font45 }
\DeclareFontShape{OT1}{cmr}{m}{n}%
     {<5><6><7><8><9><10><12>gen*cmr%
      <10.95>cmr10%
      <14.4>cmr12%
      <17.28><20.74><24.88>cmr17}{}
```

\DeclareFontShape 的倒数第二个参数用来根据字号找到对应的 tfm 文件名。具体用法参考 fntguide.pdf。

[1] OT1 编码的衬线字体较为特殊，占据了 0x20 （空格）这个码位输出形如 - 这样的符号（用于 ł 和 Ł 等），空格一般作为 skip 而不是字符处理。另外反斜线、大于号和小于号也被占为它用，通常需要借助数学模式输入。

\DeclareFontFamily 和 \DeclareFontShape 命令的最后一个参数是对本族/本组属性字体在选用时的一些初始化选项，在 \selectfont 执行时加载。如上述代码里定义了 cmr 族字体在行末断词时使用的字符为 \char45(-)；再如 CJK 宏包调用的 fd 文件在 \DeclareFontShape 命令中使用了 \CJKbold 和 \CJKnormal 命令开/关伪粗体效果。

A.1.3 字号和基线间距

LaTeX 定义了三个内部宏 \f@size、\f@baselineskip 以及 \f@linespread 分别表示字号大小、基础的基线间距和基线间距的扩大倍数。另外 LaTeX 还定义了一个外部宏 \baselinestretch 供用户修改基线间距。

需要注意的是 \f@size、\f@linespread 以及 \baselinestretch 存储的是代表数字的字符串，不是数值。所以重定义 \baselinestretch 用的是 \renewcommand 而不是 \setlength。

LaTeX 提供了诸多操作字号和基线间距的方式：

- \fontsize{<size>}{<skip>}

  令 \f@size = <size>，\f@baselineskip = <skip>，\f@linespread = \baselinestretch。

- \linespread{<spread>}

  \f@size 和 \f@baselineskip 不变，\f@linespread = <spread>，并令 \baselinestretch = \f@linespread。

- \renewcommand*\baselinestretch{<spread>}

  只改变 \baselinestretch = <spread>。

其中最后一种方式会导致 \baselinestretch 和 \f@linespread 不一致，\selectfont 检测到这种不一致时，会自动执行一次等效于 \linespread{\baselinestretch} 的命令。

\selectfont 命令最后令实际控制基线间距的原始命令 \baselineskip = \f@linespread × \f@baselineskip。

A.1.4 字体选择的过程

\fontencoding 和 \fontfamily 等命令只是缓存了字体属性的各个坐标，真正起到切换字体作用的命令是 \selectfont。形如 \bfseries 和 \textbf 等命令在内部都用了 \selectfont。

\selectfont 大致有以下步骤：

1. 首先检查和处理因 \baselinestretch 手动定义引起的不一致（见上一小节）。
2. 检查当前属性的字体是否被定义。如果已经定义，则直接用 \font 命令使用字体，否则：
   (a) 载入属性对应的 fd 文件，执行所有 \DeclareFontShape 的定义；
   (b) 检查字体属性是否被载入的 \DeclareFontShape 定义，如果否，则报缺字体的警告，并用当前字体编码下由 \DeclareFontSubstitution 给定的缺省属性按照 shape → series → family 的顺序回退；
   (c) 按照 \DeclareFontShape 定义的信息找寻当前字号对应的 tfm 字体名；
   (d) 执行 \DeclareFontFamily 和 \DeclareFontShape 定义的初始化命令；
   (e) 用 \font 原始命令定义并使用字体。
3. 根据字号的变化更新 \baselineskip。
4. 如果切换了字体编码，在此处理旧编码下的符号命令。

A.2 数学字体框架

LaTeX 强大的数学排版能力离不开数学字体的支持。与文本字体不同的是，数学字体难以(一般也没必要）自由设定和切换，通常都是成套设定的。当然，如果宏包开发者想把一些现成的符号用到数学模式中，本节的内容将会提供有用的帮助。

A.2.1 数学字体族

数学字体是按“族”(family) 设定的，排版数学公式的每个符号都有其特定的数学字体族。每个族由一个数值表示，并为不同大小的符号（\textstyle 等）定义字体。

LaTeX 将数学字体族封装为“组”(group)，每个符号组可以有自己的名字，并提供了为各个符号组定义字体的命令 \DeclareSymbolFont:

```
% LaTeX 提供的 CM 符号组，数学排版必需的 0/1/2/3 组
\DeclareSymbolFont{operators}   {OT1}{cmr} {m}{n}
\DeclareSymbolFont{letters}     {OML}{cmm} {m}{it}
\DeclareSymbolFont{symbols}     {OMS}{cmsy}{m}{n}
\DeclareSymbolFont{largesymbols}{OMX}{cmex}{m}{n}
% amsfonts 宏包提供的 CM 风格 AMS 符号组
\DeclareSymbolFont{AMSa}{U}{msa}{m}{n}
\DeclareSymbolFont{AMSb}{U}{msb}{m}{n}
```

数学公式中，包括字母、数字在内的一部分符号允许切换字体。LaTeX 为其提供了一种叫“字母表”(alphabet) 的命令，包括常见的 \mathbf 等。

```
% LaTeX 提供的字母表命令
% 以下三个命令用现成的符号组
\DeclareSymbolFontAlphabet{\mathrm}    {operators} % OT1/cmr/m/n
\DeclareSymbolFontAlphabet{\mathnormal}{letters}   % OML/cmm/m/it
\DeclareSymbolFontAlphabet{\mathcal}   {symbols}   % OMS/cmsy/m/n
% 以下四个命令用自定义的字体
% 它们在内部定义了新的符号组（无名称）并调用之
\DeclareMathAlphabet      {\mathbf}{OT1}{cmr}{bx}{n}
\DeclareMathAlphabet      {\mathsf}{OT1}{cmss}{m}{n}
\DeclareMathAlphabet      {\mathit}{OT1}{cmr}{m}{it}
\DeclareMathAlphabet      {\mathtt}{OT1}{cmtt}{m}{n}
% amsfonts 宏包定义的哥特体和黑板体
\DeclareMathAlphabet      {\mathfrak}{U}{euf}{m}{n}
\DeclareSymbolFontAlphabet{\mathbb}{AMSb}
% mathrsfs 宏包定义的花体字
\DeclareSymbolFont        {rsfs}{U}{rsfs}{m}{n}
\DeclareSymbolFontAlphabet{\mathscr}{rsfs}
```

A.2.2 数学字体版本

LaTeX 为数学字体引入了“版本”的概念，用以定义和改变全局的数学字体属性。LaTeX 提供了定义版本的 \DeclareMathVersion 命令并预定义了两个版本:

```
\DeclareMathVersion{normal}
```

```
\DeclareMathVersion{bold}
```

这两个版本对于多数数学字体宏包够用了。少数商业字体可能有 heavy 版本；最极端的一个例子是 kpfonts 宏包，这个大而全的字体宏包定义了 rm / boldrm / sf / boldsf 四个版本，还可根据宏包选项令原始的 normal / bold 版本使用衬线或无衬线体。

前一小节介绍的 \DeclareSymbolFont 以及 \DeclareMathAlphabet 会给**所有版本**定义字体。如果要修改某一个版本的字体，需要用以下形式的命令：

```
% 修改符号组的某个版本
\SetSymbolFont{operators}{bold}{OT1}{cmr} {bx}{n}
\SetSymbolFont{letters}  {bold}{OML}{cmm} {b}{it}
\SetSymbolFont{symbols}  {bold}{OMS}{cmsy}{b}{n}
% 修改字母表命令使用的符号组的某个版本
\SetMathAlphabet\mathsf{bold}{OT1}{cmss}{bx}{n}
\SetMathAlphabet\mathit{bold}{OT1}{cmr}{bx}{it}
```

给“所有版本”定义字体意味着，如果某些符号组没有重定义 bold 版本对应的粗体，则在 bold 版本中仍然沿用细体，如 Computer Modern 字体使用的巨算符和定界符。

A.2.3 数学符号的定义

TEX 将数学符号分为 8 个类别：普通符号（类别 0）、巨算符（类别 1）、二元算符（类别 2）、关系算符（类别 3）、左定界符（类别 4）、右定界符（类别 5）、标点（类别 6）、可变字体的符号（类别 7）。类别 7 就是上一小节所叙述的可用 \mathbf 这样的字母表命令改变字体的符号。不同类别的符号前后会根据上下文环境增加不同的间距，详见 *TEX by Topic* 第 23.6.1 小节。

LATEX 将定义数学符号的一些原始命令进行了封装：

```
% 将字符定义成数学符号
\DeclareMathSymbol{+}{\mathbin}{operators}{"2B}
% 原始命令为 \mathcode`\+="202B
% 第一位的 2 指二元算符的类别
% 第二位的 0 指 operators（第0组）
% 后两位指符号在 operators 使用的字体内的编码

% 将命令定义成数学符号
\DeclareMathSymbol{\gamma}{\mathord}{letters}{"0D}
% 原始命令为 \mathchardef\gamma="010D

% 将命令定义成数学重音
\DeclareMathAccent{\hat}{\mathalpha}{operators}{"5E}
% 原始命令为 \def\hat{\mathaccent"705E }
```

一些符号在行内和行间公式环境（更准确一点，是 \textstyle 和 \displaystyle）选用的字体有所区别，包括所有和 \left 和 \right 一起使用的定界符，以及根号。它们的定义都需要用到两个符号组：

```
% 将字符定义为定界符
% 不用作定界符时定义成一般的数学符号
\DeclareMathDelimiter{(}{\mathopen}
   {operators}{"28}{largesymbols}{"00}
% 原始命令为 \delcode`\(="028300 及 \mathcode`\(="4028

% 将命令定义为定界符
\DeclareMathDelimiter{\langle}
   {\mathopen}{symbols}{"68}{largesymbols}{"0A}
% 原始命令为 \def\langle{\delimiter"426830A }

% 将命令定义为根号，绝大多数字体包仅此一例
\DeclareMathRadical{\sqrtsign}{symbols}{"70}{largesymbols}{"70}
% 原始命令为 \def\sqrtsign{\radical"270370 }
```

A.2.4 数学字号

在 LaTeX 中，数学字体的基本字号 (\textstyle) 和正文字体的字号保持一致。每次进入数学模式都会检测字号，如果正文字号更改了，所有数学字体都会更新到与正文字号字号对应。

每个数学符号组都要为三种 style 定义对应的 tfm 字体。LaTeX 将字号作为一个单独的坐标处理，每次更新数学字体时，只需要将 \DeclareSymbolFont 定义的字体属性和字号结合即可。

LaTeX 提供了 \DeclareMathSizes 命令预定义与某个正文字号对应的三种 style 的数学字号。未经此命令定义的数学字号，按照 1 : \defaultscriptratio : \defaultscriptscriptratio 处理（默认为 1:0.7:0.5）。

```
% 摘自 fontmath.ltx
% 10pt 的 \normalsize
\DeclareMathSizes{\@xpt}{\@xpt}{7}{5}
% 11pt 的 \normalsize
\DeclareMathSizes{\@xipt}{\@xipt}{8}{6}
% 10pt 的 \large; 12pt 的 \normalsize
\DeclareMathSizes{\@xiipt}{\@xiipt}{8}{6}
% 10pt 的 \Large; 12pt 的 \large
\DeclareMathSizes{\@xivpt}{\@xivpt}{\@xpt}{7}
```

附录 B LaTeX3 初步

LaTeX3 的最终目标是真正完成用户界面、设计、编程三层分离的结构。

B.1 LaTeX3 格式规范

LaTeX3 修改了若干字符的 catcode 以构造格式规范，主要起作用的有：

- 字符 _ 和 : 的 catcode 为 11 (letter)。与 LaTeX 2ε 的 @ 类似，被用于 LaTeX3 的变量和宏命令中。
- 空格 ␣（包括制表符 `^^I`）的 catcode 为 9 (ignored)，被完全忽略。空格在此起到规范代码格式的作用。
- 波浪号 ~ 的 catcode 为 10 (space)，用作实际的“空格”。

LaTeX3 提供了一对命令 `\ExplSyntaxOn` `\ExplSyntaxOff`，类似 LaTeX 2ε 里的 `\makeatletter` 和 `\makeatother` 的作用，开启和关闭上述 catcode 设定。

在 LaTeX 2ε的宏包或文档类中使用时，首先引入 `expl3` 宏包，然后用其提供的 `\ProvidesExplPackage` 或 `\ProvidesExplClass` 命令：

```
\RequirePackage{expl3}

\endinput
```

在 LaTeX3 里，每个宏包/文档类为一个“模块”(module)，宏包/文档类的名称则为模块名。LaTeX3 引入了公共和私有的概念，公共的变量和命令可以被其他模块使用，而私有的变量和命令（应当）仅在模块内部使用。

B.1.1 LaTeX3 变量记号格式规范

LaTeX3 利用 _ 符号组织记号名称。一个变量记号分为四部分：

1. 作用域：c 为常量；g 为全局变量；l 为局部变量。

在使用 B.4 节中的命令操作时，常量应当用带 ‘const’ 的命令定义；全局变量应当用带 ‘g’ 的版本；局部变量用不带 ‘g’ 的版本。

2. 模块名称：宏包名、文档类名。
3. 自定义名称：以变量的用途命名，可以是一个或多个单词，以 _ 连接。
4. 变量类型：整数型 ‘int’，逗号列表类型 ‘clist’ 等等。

```
\c_dummypkg_var_name_tl   % 常量，记号列表类型，名字为 var_name
\g_dummypkg_counter_int   % 全局变量，整数类型，名字为 counter
\g__dummypkg_inner_int    % 私有全局变量，整数类型，名字为 inner
\l_dummypkg_sequence_seq  % 局部变量，队列类型，名字为 sequence
\l__dummypkg_lists_clist  % 私有局部变量，逗号列表类型，名字为 lists
```

B.1.2 LaTeX3 宏命令格式规范

所有 LaTeX3 底层命令（包括宏重命名的 TeX 原始命令）都需要带冒号。冒号后面跟着若干个字母（可以不跟），代表这个命令使用的各个参数的格式。

```
% 带两个参数的命令
% 两个参数各为一组记号
\dummypkg_macro:nn {<tokens>} {<tokens>}
% 带三个参数的私有命令
% 第一个参数为单个记号
% 第二个参数为表示命令名称的一组记号，展开成一个命令
% 第三个参数为一组记号
\__dummypkg_macro:Ncn <token> {<csname>} {<tokens>}
% 不带参数的命令
\dummypkg_macro_noarg:
```

基本格式

最基本的参数格式包括：

- **N 格式** 单个记号（字符或者命令）。
- **n 格式** 记号列表，以一对花括号（catcode 分别为 1 和 2）为界。
- **p 格式** 形如 #1 #2 #3 . . . 的参数模板。
- **w 格式** 记号形式不限。多见于 TeX 原始命令，以及具有复杂参数模板的宏包内部命令中。

条件判断格式

一些涉及条件判断的命令还用到了 T 格式和 F 格式，代表条件为真（为假）时执行的代码。详见 B.3 节。

展开格式

除了以上基本格式和条件判断格式，LaTeX3 还允许将参数先进行展开，展开的结果作为命令的参数。假设已经用 LaTeX 3 的规范定义了如下命令和变量（定义方式详见下一节）：

```
\cs_set_nopar:Npn \test_macro_a: { HAPPY~TEXING }
\cs_set_nopar:Npn \test_macro_b: { \test_macro_a: }
\cs_set_nopar:Npn \test_macro_c: { \test_macro_b: }
\cs_set_nopar:Npn \empty_macro_a: { }
\cs_set_nopar:Npn \empty_macro_b: { \empty_macro_a: }
\dim_new:N \test_dim
\dim_set:Nn \test_dim { 25 pt }
```

我们以以上命令和变量为例解释各种展开格式：

- **c 展开格式** (csname) 将若干记号展开成通过 `\csname` 和 `\endcsname` 得到的命令。
  如 `test_macro_a:` 经过 c 展开得到 `\test_macro_a:`。
- **o 展开格式** (one-level) 一个记号（或多个记号的第一个）展开一次。
  如 `\test_macro_b: \test_macro_c:` 经过 o 展开得到
  `-> \test_macro_a: \test_macro_c:`。
- **V 展开格式** (Value) 将一个记号代表的变量展开成变量的值。
  如 `\test_dim` 经过 V 展开得到 `25pt`。
- **v 展开格式** (value) 相当于先进行 c 展开，再进行 V 展开。

如 test_dim 经过 v 展开得到 25pt。

- **f 展开格式** (fully) 将一个或多个记号重复展开，直至出现字符、原始命令、变量等不可展开的记号。

如 \test_macro_b: \test_macro_c: 经过 f 展开得到

-> HAPPY TEXING\test_macro_c:;

再如 \empty_macro_b: \test_macro_b: \test_macro_c: ，首先展开 \empty_macro_b: ，其最终的展开结果为空，须继续展开 \test_macro_b:，最后也得到

-> HAPPY TEXING\test_macro_c:;

- **x 展开格式** (exhausted) 将一个或多个记号彻底展开（\edef 级别)。

如 \test_macro_b: \test_macro_c: 经过 x 展开得到

-> HAPPY TEXINGHAPPY TEXING。

\exp_not:N (\noexpand) 在这种展开方式下起保护作用，如 \exp_not:N \test_macro_b: \test_macro_c:，展开得到

-> \test_macro_b: HAPPY TEXING。

在某些情况下可能会出现 f 展开比 x 展开更彻底的结果。由于 x 展开事实上是将记号通过 \cs_set_nopar:Npx，也就是 \edef 定义到一个临时变量里，那么如果参数的健壮命令它们都不会得到展开。但在 f 展开中健壮命令的限制不起作用，记号将一直展开，直到出现不可展开的字符或命令为止。

用到 \exp_not:N(\noexpand) 的情形更为复杂。\noexpand 会将后面的记号展开成原样，但会令其意义暂时等于 \relax，也就是不可展记号，所以有可能令 f 展开中止。

如 \exp_not:N \test_macro_b: \test_macro_c: 经过 f 展开的实际结果是：

->\test_macro_b: \test_macro_c:。

几种展开的常用环境通常是：

- 需要用宏的参数组成命令时，使用 c 展开。这也是最常用的展开方式。
- 需要获得诸如 tl,str,clist 等类型变量的内容时，可使用 V 展开或 o 展开；如果变量名也用到宏的参数，则用 v 展开。
- 需要获得诸如 \int_eval:n 等表达式解析的结果，或者条件判断命令的结果时，由于这些命令通常需要多步展开，最终生成结果输出字符，所以使用 f 展开。
- x 展开一般用在要求某个参数完全由字符组成的情况。另外一种 x 展开的使用环境是，参数中的一部分需要完全展开成字符，而不需要展开的命令，除健壮命令外，都用 \exp_not:N 保护起来。

B.2 LaTeX3 宏定义

B.2.1 基本定义

LaTeX3 将各种宏定义的 TeX 原始命令封装成为 LaTeX3 的语法格式。具体情况见表 B.1：

基本定义还有 new 形式，如 \cs_new:Npn ，等同于全局定义 \cs_gset:Npn，但在定义之前加了检查，如果被定义的命令已有定义，则报错。

LaTeX3 还提供了 \cs_set_eq:NN 和 \cs_gset_eq:NN 将一个命令的意义赋予另一个命令，含义分别为 \let 和 \global \let。\cs_new_eq:NN 则相当于加了检查的 \cs_gset_eq:NN。

LaTeX3 格式	含义	LaTeX3 格式	含义
\cs_set_nopar:Npn	\def	\cs_set_protected_nopar:Npn	\protected\def
\cs_set_nopar:Npx	\edef	\cs_set_protected_nopar:Npx	\protected\edef
\cs_gset_nopar:Npn	\gdef	\cs_gset_protected_nopar:Npn	\protected\gdef
\cs_gset_nopar:Npx	\xdef	\cs_gset_protected_nopar:Npx	\protected\xdef
\cs_set:Npn	\long\def	\cs_set_protected:Npn	\protected\long\def
\cs_set:Npx	\long\edef	\cs_set_protected:Npx	\protected\long\edef
\cs_gset:Npn	\long\gdef	\cs_gset_protected:Npn	\protected\long\gdef
\cs_gset:Npx	\long\xdef	\cs_gset_protected:Npx	\protected\long\xdef

表 B.1: LaTeX3 封装的命令定义的语法格式

B.2.2 简便定义

我们注意到：第一，对于绝大多数基本宏命令（N 或者 n 格式参数），基本定义中的 p 格式参数，也就是宏的参数模板，有着 #1#2...#n 的简便形式；第二，每个宏按照规范都要包括参数格式。结合这两点，LaTeX3 还提供了一种简便的定义方式，通过统计参数格式的字母个数自动生成参数模板：

```
\cs_set_nopar:Nn \test_macro:nnn
  { Test ~ #1, ~ #2 ~ and ~ #3. }
% 等效于
% \cs_set_nopar:Npn \test_macro:nnn #1#2#3
%   { Test ~ #1, ~ #2 ~ and ~ #3. }
```

B.2.3 定义变体

LaTeX3 允许将基本的命令方便地扩展成各种变体，以利用各种展开格式。扩展变体的命令为 \cs_generate_variant:Nn，其第一个参数为基本命令，后一个参数是一个逗号列表，每一项代表将基本命令的前几个参数格式替换成变体格式。

```
\cs_new_nopar:Npn \test_macro:Nnn { ... }
\cs_generate_variant:Nn \test_macro:Nnn {c, Nvx, No}
% 定义了三种变体：
% \test_macro:cnn \testmacro:Nvx \testmacro:Non
```

关于扩展变体，有三点需要注意：

1. 变体的定义有严格的限制：基本命令的参数格式中，可用来扩展变体的部分或者只包括 N 或者 n，或者某个参数用任意格式，而变体不改动这个格式，如 \test_macro:on 扩展成 oc 变体等。

事实上，开发者应当总是用 \cs_new:Npn 这样的命令定义只含 N 或者 n 格式参数的命令作为基本命令，而通过 \cs_generate_variant:Nn 命令扩展出变体。

2. 对一个基本命令重复扩展相同的变体并不会引起冲突，\cs_generate_variant:Nn 总是会检查变体是否获得定义。

3. 有以下情形之一，变体命令是健壮的（用 \cs_new_protected_nopar:Npn 定义）：

(a) 基本命令是健壮的;

(b) 基本命令是不可展开的（TeX 原始命令等);

(c) 变体包含 x 展开格式。

B.3 LaTeX3 条件判断

LaTeX3 将复杂的条件判断封装成统一的格式:

- **p 形式** `\<csname>_p:<args>` {arg1} {arg2} ...
  根据真值展开为对应的布尔常量，可用于布尔表达式中。
- **T 形式** `\<csname>:<args>T` {arg1} {arg2} ... {<true code>}
  若判断为真，展开为 <true code>，反之展开结果为空。
- **F 形式** `\<csname>:<args>F` {arg1} {arg2} ... {<false code>}
  若判断为假，展开为 <true code>，反之展开结果为空。
- **TF 形式** `\<csname>:<args>TF` {arg1} {arg2} ... {<true code>}{<false code>}
  根据判断的真值展开为 {<true code>} 或 {<false code>}。
  我们以字符串变量 str 中用到的比较判断命令 `\str_if_eq:nnTF` 等举例:

```
\str_if_eq_p:nn { foo } { foo } % 展开为 \c_true_bool
\str_if_eq_p:nn { foo } { bar } % 展开为 \c_false_bool

% 当前判断为真，展开为 "Equivalent!"
\str_if_eq:nnTF { foo } { foo }
  { Equivalent! }{ Different! }
% 当前判断为真，展开为 "Equivalent!"
\str_if_eq:nnT { foo } { foo } { Equivalent! }
% 当前判断为真，展开为空
\str_if_eq:nnF { foo } { foo } { Different! }
% 当前判断为假，展开为 "Different!"
\str_if_eq:nnF { foo } { bar } { Different! }
```

LaTeX3 提供了自定义条件判断的命令 `\prg_set_conditional:Npnn`:

```
\prg_set_conditional:Npnn \test_if:n #1 { p, T, F, TF }
  {
    \cs_set_nopar:Npn \test_if_input: { #1 }
    \cs_set_nopar:Npn \test_if_foo: { foo }
    \cs_set_nopar:Npn \test_if_baz: { baz }
    \if_meaning:w \test_if_input: \test_if_foo:
      \prg_return_true:
    \else:
      \if_meaning:w \test_if_input: \test_if_baz:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
```

```
  }
```

其中 \prg_return_true: 和 \prg_return_false: 只用在条件判断结构定义的参数中，不得用于别处。它们用于正确生成各种形式，能够处理嵌套的 \if...\fi，如上述例子所示。

也可以用其它条件判断命令展开为 \prg_return_true: 和 \prg_return_false:，如仿照以上例子：

```
\prg_set_conditional:Npnn \test_if:n #1 { p, T, F, TF }
  {
    \str_if_eq:nnTF {#1} { foo }
      { \prg_return_true: }
      {
        \str_if_eq:nnTF {#1} { baz }
          { \prg_return_true: }
          { \prg_return_false: }
      }
  }
```

条件判断的定义也有 new 形式定义 \prg_new_conditional:Npnn、省略参数模板的简便定义 \prg_set_conditional:Nnn、健壮定义 \prg_set_protected_conditional:Npnn 以及以上几种定义方式的组合。需要注意的是**健壮定义无法定义 p 形式**，这是由于在布尔表达式中需要对命令进行展开。

可以用 \prg_set_eq_conditional:NNn 将一组条件判断命令的意义赋予另一组：

```
% 定义了 \test_eq_p:n \test_eq:nT \test_eq:nF \test_eq:nTF
\prg_set_eq_conditional:NNn \test_if_aux:n \test_if:n
  { p, T, F, TF }
  % 以上的形式必须在原条件判断命令都存在定义
```

条件判断命令也可以对除 T 和 F 之外的参数用 \cs_generate_variant:Nn 生成变体，一般要对所有的形式逐个生成：

```
\cs_generate_variant:Nn \test_if_p:n { c }
\cs_generate_variant:Nn \test_if:nT  { c }
\cs_generate_variant:Nn \test_if:nF  { c }
\cs_generate_variant:Nn \test_if:nTF { c }
```

在特别简单的情形，也可以用原始的 \if 和 \fi 手写条件判断命令：

```
\cs_new_nopar:Npn \manual_if_p:n #1
  { \if... \c_true_bool \else: \c_false_bool \fi: }
\cs_new_nopar:Npn \manual_if:nT #1
  { \if... \exp_after:wN \use_i:n \else: \exp_after:wN \use_none:n \fi: }
\cs_new_nopar:Npn \manual_if:nF #1
  { \if... \exp_after:wN \use_none:n \else: \exp_after:wN \use_i:n \fi: }
\cs_new_nopar:Npn \manual_if:nTF #1
  { \if... \exp_after:wN \use_i:nn \else: \exp_after:wN \use_ii:nn \fi: }
```

甚至可以在 \if 分支里定义条件判断命令，在这种用法中，条件的真值完全由 \if 决定，所以一般不需要额外的参数：

```
\if...
  \cs_set_eq:NN \manual_if_p: \c_true_bool
  \cs_set_eq:NN \manual_if:T  \use_i:n
  \cs_set_eq:NN \manual_if:F  \use_none:n
  \cs_set_eq:NN \manual_if:TF \use_i:nn
\else:
  \cs_set_eq:NN \manual_if_p: \c_false_bool
  \cs_set_eq:NN \manual_if:T  \use_none:n
  \cs_set_eq:NN \manual_if:F  \use_i:n
  \cs_set_eq:NN \manual_if:TF \use_ii:nn
\fi:
```

B.4 LaTeX3 变量的定义和使用

LaTeX3 将 TeX 定义变量的 \countdef, \newcount 等命令封装成较为一致的格式，并且在记号列表结构的基础上发展了栈、队列、键值对形式的复杂变量。

- 封装的 TeX 寄存器变量 int/dim/skip/muskip

  其中 int 就是 TeX 里的 count。它们通过 ε-TeX 支持表达式赋值。

- 布尔值/浮点数变量 bool/fp

  分别支持布尔表达式和浮点数表达式（包括初等函数）赋值。这两种变量的表达式解析完全是用宏实现的，LaTeX3 为表达式解析写了巨量代码，解析的效率很低。

- 记号列表变量 tl

  LaTeX3 的记号列表完全是用宏实现的，没有用到 TeX 的 \toks 寄存器。

- 字符串变量 str

  封装的记号列表变量之一，只包括字符记号。

- 逗号列表变量 clist

  封装的记号列表变量之一。

- 队列变量 seq

  封装的记号列表变量之一。

- 属性变量 prop

  封装的记号列表变量之一，具有键值对结构。

- 盒子变量 box

  对第 四 章所述的盒子变量的封装。

  本节的内容将要介绍变量常用的命令。许多变量的命令形式非常相近，将分小节归纳。

  注：条件判断命令中带下划线的 TF 代表同时存在 T,F,TF 三种形式。

B.4.1 变量的定义命令

下表归纳了各种变量的“共有”命令，涉及对变量名称的定义、操作和判断：

<var>=int,dim,skip,muskip,bool,fp,tl,str,clist,seq,prop,box	
\<var>_new:N	定义新变量
\<var>_set_eq:NN	令一个变量等同于已有变量
\<var>_gset_eq:NN	
\<var>_if_exist_p:N	判断变量是否存在
\<var>_if_exist:NTF	
\<var>_show:N	在终端显示变量的值

B.4.2 为变量直接赋值

TeX 寄存器变量、布尔值及浮点数变量、记号列表/字符串/逗号列表变量支持直接用各自支持的表达式（记号列表）赋值，或显示表达式的结果（记号列表的内容)。

下表归纳了与表达式赋值相关的命令：

<var>=int,dim,skip,muskip,bool,fp,tl,str,clist	
\<var>_set:Nn	为变量赋值
\<var>_gset:Nn	
\<var>_const:Nn	定义常量并赋值
\<var>_gconst:Nn	
\<var>_show:n	显示变量表达式的结果或记号列表的内容

B.4.3 数值变量的操作

TeX 寄存器变量 int, dim, skip 和 muskip 以及封装的浮点数变量 fp 可以直接进行加减数值的操作，以及显示变量值（相当于 \the)：

<var>=int,dim,skip,muskip,fp	
\<var>_zero:N	为变量赋零值
\<var>_gzero:N	
\<var>_zero_new:N	为变量赋零值，若变量未定义则定义之
\<var>_gzero_new:N	
\<var>_add:Nn	为变量加值
\<var>_gadd:Nn	
\<var>_sub:Nn	为变量减值
\<var>_gsub:Nn	
\<var>_use:N	显示变量值
\<var>_eval:n	显示变量表达式的结果

B.4.4 记号列表类型的变量

记号列表类型的变量，其值由若干个元素组成。tl 变量的每个元素是一个或一组记号；str 变量的每个元素是一个字符；clist 变量的每个元素是一组以逗号隔开的记号；seq 变量的每个元素是一个或一组记号；prop 变量的每个元素是一组键值对。

下表归纳了与记号列表内容相关的命令（注：盒子变量 box 也有对内容的类似操作，一并归纳至此。但盒子变量并不具备“元素”的概念)：

<var>=tl,str,clist,seq,prop,box	
\<var>_clear:N	清空变量的内容
\<var>_gclear:N	
\<var>_clear_new:N	清空变量的内容，若变量未定义则定义之
\<var>_gclear_new:N	
\<var>_if_empty_p:N	判断变量的内容是否为空
\<var>_if_empty:NTF	
<var>=tl,str,clist,seq,prop	
\<var>_concat:NNN	将后两个变量按顺序合并，赋给第一个变量
\<var>_gconcat:NNN	
\<var>_item:Nn	获得变量中的第 n 个元素，n 为数值表达式 对于 prop 变量，n 为键名
<var>=tl,str,clist	
\<var>_item:nn	获得记号列表中的第 n 个元素

记号列表的队列操作

除 prop 外的记号列表变量可以视为双端队列，可以向头尾追加元素。tl/str/clist 变量可以一次追加一个或多个元素；seq 变量每次只能追加一个元素。

<var>=tl,str,clist,seq	
\<var>_put_left:Nn	将元素追加到变量的左端
\<var>_gput_left:Nn	
\<var>_put_right:Nn	将元素追加到变量的右端
\<var>_gput_right:Nn	

记号列表变量的栈操作

clist 和 seq 变量也可以视为堆栈，进行操作，“栈顶”位于最左边。

<var>=clist,seq	
\<var>_get:NN	将栈顶元素赋予一个记号列表变量 若堆栈为空，变量的值为 \q_no_value
\<var>_get:NNTF	\<var>_get:NN 的带判断版本
\<var>_pop:NN	将栈顶元素出栈并赋予一个记号列表变量
\<var>_gpop:NN	若堆栈为空，变量的值为 \q_no_value
\<var>_pop:NNTF	\<var>_pop:NN 的带判断版本
\<var>_gpop:NNTF	
\<var>_push:Nn	将元素入栈
\<var>_gpush:Nn	

prop 变量的键值对操作

prop 变量没有左右或者栈顶的概念，元素由键名唯一定位。

<var>=prop	
\<var>_put:Nnn	将键值对加入变量
\<var>_gput:Nnn	如果键已存在，覆盖其值
\<var>_put_if_new:Nnn	将键值对加入变量
\<var>_gput_if_new:Nnn	如果键已存在，则无变化
\<var>_get:NnN	将给定键的值赋予一个记号列表变量 若给定键不存在，变量的值为 \q_no_value
\<var>_get:NnNTF	\<var>_get:NnN 的带判断版本
\<var>_pop:NnN	将给定键的值赋予一个记号列表变量，并清除键值对
\<var>_gpop:NnN	若给定键不存在，变量的值为 \q_no_value
\<var>_pop:NnNTF	\<var>_pop:NnN 的带判断版本
\<var>_gpop:NnNTF	

遍历记号列表变量的元素

除 str 类型外，记号列表的变量支持遍历，将每个元素用于赋值、定义命令等，类似 JavaScript 或 PHP 等动态语言中的 foreach 循环。

<var>=tl,clist,seq,prop	
\<var>_map_function:NN	将每个元素作为 function 宏的第一个参数 (prop 每个键值对作为头两个参数)
\<var>_map_inline:Nn	将每个元素作为 inline function 中的 #1 (prop 每个键值对作为 #1 和 #2)
\<var>_map_break:	中断对变量的遍历
\<var>_map_break:n	中断对变量的遍历并执行给定的代码
<var>=tl,clist,seq	
\<var>_map_variable:NNn	将每个元素赋给一个记号列表变量， 在之后的 function 中用到这个变量
<var>=tl,clist	
\<var>_map_function:nN	\<var>_map_function:NN 的表达式版本
\<var>_map_inline:nn	\<var>_map_inline:Nn 的表达式版本
\<var>_map_variable:nNn	\<var>_map_inline:NNn 的表达式版本

遍历命令允许嵌套使用。以下举一个生成九九乘法表的例子，利用了遍历命令的嵌套和中断。效果如图 B.1 所示。

```
% box 变量的用法参考 interface3.pdf，此处不专门归纳
% 定义乘法公式模板盒子，以此盒子的宽度为准生成所有公式
% 公式在模板盒子里居中对齐
\box_new:N \l__alphabet_box
\hbox_set:Nn \l__alphabet_box { ~ 9 $\times$ 9 = 81 ~ }
\clist_new:Nn \l__alphabet_clist
\clist_set:Nn \l__alphabet_clist
  { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }
```

1×1=1
1×2=2 2×2=4
1×3=3 2×3=6 3×3=9
1×4=4 2×4=8 3×4=12 4×4=16
1×5=5 2×5=10 3×5=15 4×5=20 5×5=25
1×6=6 2×6=12 3×6=18 4×6=24 5×6=30 6×6=36
1×7=7 2×7=14 3×7=21 4×7=28 5×7=35 6×7=42 7×7=49
1×8=8 2×8=16 3×8=24 4×8=32 5×8=40 6×8=48 7×8=56 8×8=64
1×9=9 2×9=18 3×9=27 4×9=36 5×9=45 6×9=54 7×9=63 8×9=72 9×9=81

图 B.1: 用 clist 遍历生成的九九乘法表

```
\clist_map_inline:Nn \l__alphabet_clist
  {
    \tex_noindent:D
    % 内层遍历的 inline function 参数为 ##1
    \clist_map_inline:Nn \l__alphabet_clist
      {
        \hbox_to_wd:nn { \box_wd:N \l__alphabet_box }
          {
            \tex_hfil:D
            ##1 $\times$ #1 = \int_eval:n { #1 * ##1 }
            \tex_hfil:D
          }
        \int_compare:nNnF {##1} < {#1} { \clist_map_break: }
      }
    \tex_par:D
  }
```

B.5 LaTeX3 的错误/警告功能

B.6 LaTeX3 的键值对功能

LaTeX3 原生引入了键值对功能，原来需要宏包才支持的键值对功能，现今是 LaTeX3 的基本组成部分之一。

```
\tl_new:N \l__dummykey_foo_tl
\fp_new:N \l__dummykey_baz_fp
\keys_define:nn { dummykey }
  {
    % 将 foo 和 baz 的值分别赋给对应的变量
    foo .tl_set:N = \l__dummykey_foo_tl ,
    baz .fp_set:N = \l__dummykey_baz_fp ,
    % 将 bar 的值用于定义一个宏
    bar .code:n = { \cs_set_nopar:Npn \dummykey_bar: {Test ~ #1} }
  }
```

```
\keys_set:nn { dummykey }
  { foo = thefoo , baz = 3.5 * 4.4 , bar = TeX }

\tl_to_str:N \l__dummykey_foo_tl          % 输出 thefoo
\fp_to_decimal:N  \l__dummykey_baz_fp     % 输出 15.4
\dummykey_bar:                            % 输出 Test TeX
```

B.6.1 用键值对为变量赋值

<var>=int,dim,skip,bool,fp,tl,clist	
.<var>_set:N	将键值赋予变量
.<var>_set:c	
.<var>_gset:N	
.<var>_gset:c	
特殊情况	
.bool_set_inverse:N	将键值的布尔表达式取反后赋给变量
.bool_set_inverse:c	
.bool_gset_inverse:N	
.bool_gset_inverse:c	
.tl_set_x:N	将键值作 \edef 展开后赋给变量
.tl_set_x:c	
.tl_gset_x:N	
.tl_gset_x:c	

B.6.2 执行特定代码的键值对

我们当然可以把键值都存为变量，在键值对外部处理。不过更为方便的是使用键值直接执行一些命令。本节开头所举的例子使用的 .code:n 就是最基础的执行代码的用法。

单选键值对

多选键值对

多选键值对允许用逗号列表向一个键赋多个值。操作命令除了名称由 .choice: 等相应变成 .multichoice: 之外，其它都是一致的。

B.6.3 子族键值对

B.6.4 元键值对

元键值对是形如 {metakey={foo=testfoo,baz=testbaz}} 的键值对，其值本身就是若干键值对。

B.6.5 键值对的属性

B.6.6 用 LaTeX3 键值对处理宏包/文档类选项

在当前的 LaTeX 2_ε 宏包和文档类架构基础上，LaTeX3 提供了 l3keys2e 宏包，令宏包和文档类的选项可以一律交给 LaTeX3 的键值对功能处理，不需要经过 LaTeX 2_ε 的\DeclareOption 和 \ProcessOptions 等步骤。

如果有需要传递给其他宏包或文档类的选项，仍然需要用 \PassOptionsToPackage 等命令操作，并且 \CurrentOption 仍然可用，详见 2.3.2 小节。

```
% 宏包 dummypkg.sty 代码
\keys_define:nn { key }
  {
    optiona .code:n = { ... },
    optionb .code:n =
      {
        \PassOptionsToPackage { \CurrentOption } { <other package> }
      }
  }
% 把文档类/宏包选项交给 key 键值族处理
% 宏包同时处理文档类定义的全局选项，与 \ProcessOptions 一致
\ProcessKeysOptions { key }
% 把文档类/宏包选项交给 key 键值族处理
% 宏包不处理文档类定义的全局选项
\ProcessKeysPackageOptions { key }
\RequirePackage { <other package> }

% 正文代码
\usepackage[optiona=aaa,optionb=bbb]{dummypkg}
% 自动载入了 \usepackage[optionb=bbb]{<other package>}
```

B.7 LaTeX3 定义用户命令