expl3 宏包与 LaTeX3 编程*

LaTeX3 项目组

发布于 2018-08-23

摘要

针对 LaTeX 等大型 TeX 宏编程项目对编程风格设计的需求，本文介绍一套属于 LaTeX3 的基础层面的全新程序设计约定。

该系统主要特性有：

- 将宏（在 LaTeX 术语中也称之为命令）分为 LaTeX 函数和 LaTeX 参数，并依照功能分为若干组模块；
- 基于这些分类提出了系统性的命名规则；
- 对于函数选项展开提出了简洁的控制机制。

该系统已经用于 LaTeX3 项目中的 TeX 程序设计基础。请注意，该语言不是为了用在文档标记或者样式规范中，而是为了构建其基础。

本文档简要介绍了 expl3 程序接口的主要思路。关于 LaTeX3 编程层面的完整叙述，请参考配套的 `interface3` 文档。

* 这份文档是英文文档 expl3.pdf 的中文翻译。该手册简要介绍了 LaTeX3 语法，是学习 LaTeX3 语法以及更多进阶材料的必读文档。

翻译过程中需要说明以下几点

convention 这里指 LaTeX3 提出的一整套新的代码格式（新语言），应当使用这种代码风格进行开发。翻译为“约定”。

interface 一般情况下翻译为“接口”。

primitive (e)TeX 以及各引擎提供的基本命令（控制序列），翻译为“原始命令”。

token 这是 TeX 中的一个基本概念，然而确实很难找到一个确切的中文词汇与之对应。这里遵从 zoho 在 *TeX for the Impatient* 等著作中的处理方式翻译为“记号”。进而“token list”翻译为“记号序列”。

register 翻译为“寄存器”。

quark LaTeX3 中的新概念，暂时按照物理学中的称呼翻译为“夸克”。

本文档的地址为：https://github.com/WenboSheng/LaTeX3-doc-cn/expl3-doc-cn

盛文博
<wbsheng88@foxmail.com>

2018 年 9 月 1 日

1 简介

在 LaTeX 2ε 之后，开发 LaTeX 内核的第一步就是如何设计底层系统编程。当前的情况是混合使用 LaTeX 和 TeX 宏。与此不同的是，LaTeX3 系统提供了自己的一套接口，用于控制 TeX 所需的所有函数。这项工作中关键的一部分就是确保所有的一切都是文档化的，这样即便 LaTeX 程序员和用户不熟悉内核机制或者 plainTeX，也可以进行高效开发。

expl3 宏集提供了新的 LaTeX 程序设计接口。为了使编程系统化，LaTeX3 使用了与 LaTeX 2ε 或 plain TeX 截然不同的程序风格约定。因此，进行 LaTeX3 开发需要熟悉这种全新的语法规则。

下一节将展示，在一个完整的 TeX 文档处理系统中，这项语言适用于哪些地方。之后将描述命令名称的语法结构的主要特性，比如函数选项的语法规范。然后将解释这种选项语法背后的实用思想，同时也包括展开控制机制和定义不同形式函数的接口。

我们将说明，使用结构化的命名规则和多样化的函数形式能够极大提高代码的可读性以及可靠性。此外，实践表明，新语法导致的长命令名不会明显加剧写代码开发的困难程度。

2 语言和接口

在基于 TeX 的文档处理系统中，我们可以罗列出涉及到不同层面接口的各种语言。本节将介绍在 LaTeX3 中最重要的一些部分。

文档标记 包括文档（`.tex` 文件）中的一些命令，通常称为标签（tag）。

一种广泛接受的观点是，这样的标记本质上应当是*声明式的*（*declarative*）。它可以是传统的 TeX 标记，比如 [3] 和 [2] 中描述的 LaTeX 2ε；或者是通过 HTML 或 XML 定义的标记语言。

传统 TeX 代码约定（如同 [1] 所描述的）的一大问题就是，TeX 确实精心而巧妙地设计了原始格式命令的名称和语法，这样文档作者就可以在标记或宏中很“自然”地直接使用这些原始命令。然而具有讽刺意味的是，无处不在的逻辑标记（这广泛认为是一大优势）反而导致几乎没有必要在文档或自定义宏中直接使用这些原始格式命令。几乎只有 TeX 开发人员定义高层次命令时才使用原始命令，而在社区中这种特殊的语法几乎是不流行的。另外，原始命令占用了很多方便的名称（例如 `\box` 或 `\special`），否则本可以像文档标记一样广泛使用的。

设计者接口 该接口将文档的排版设计规范与“格式化文档”的程序联系起来。理想状况下应当使用声明式的语言，这样可以更好地表达不同文档元素之间的布局关系和间距规则。

这种语言没有嵌入到文档文本中，所以其形式与文档标记语言有很大不同。对于 LaTeX，在 LaTeX2.09 中这一层次几乎完全是缺失的；LaTeX 2ε 在这一领域做了一些改进，然而在 LaTeX 中实现一套设计规范仍然需要很多“低层次”编程，而这是不可接受的。

开发者接口 这一语言构建在 TeX 原始命令（或者后继程序）之上，并实现了基本排版功能。它也用于“在 TeX 内”实现前两种语言，比如当前的 LaTeX 系统就是这样。

本文档描述的语法约定覆盖最后一个层面，这一系统旨在提供一个合适的 LaTeX3 编程基础。主要突出的特点总结如下：

- 为所有命令提供相容的命名机制，包括 TeX 原始命令。
- 将命令分为 LaTeX 函数或 LaTeX 参数，同时也根据功能分为若干模块。
- 为选项展开提出简洁的控制机制。
- 提供一组 LaTeX 核心函数，可以处理队列、集合、栈、属性列表等程序组件。
- TeX 编程环境。比如在其中所有的空格都会被忽略。

3 命名机制

LaTeX3 没有使用 @ 作为定义内部宏的“字母”，取而代之的是使用符号 _ 和 : 来表现结构。与 plainTeX 格式和 LaTeX 内核不同，这些额外的字母只能用在宏名称的各部分之间（没有奇怪的元音替换）。

虽然说 TeX 实际上是宏处理器，不过按照 expl3 程序语言的约定，我们还是区分*函数*和*变量*这两个概念。函数可以带有可展开或者立即执行的选项。变量可以被赋值，用于函数的选项；变量不会直接使用，而是通过函数进行操作（包括获取函数和设置函数）。具有相关功能的函数和变量（例如使用计数器、或者操作记号列表等）被归纳在一起组成一个*模块*。

3.1 例子

在给出命名机制的细节之前，这里列出一些典型例子来说明该机制的风格；首先是一些变量名

`\l_tmpa_box` 是对应于盒子寄存器的局部变量（这里的 `l_` 前缀代表局部）。

`\g_tmpa_int` 是对应于整型寄存器（即 TeX 计数寄存器）的全局变量（这里的 `g_` 前缀代表全局）。

`\c_empty_tl` 是常值（`c_`）记号列表变量，并且总是空的。

接下来是一个典型的函数名称例子。

\seq_push:Nn 是一个函数,将由第二个选项确定的记号列表放入由第一个选项确定的栈中。两个选项的不同之处由后缀 :Nn 指定。第一个选项必须是一个单记号,用以命名栈参数:这样的单记号选项记为 N。第二个选项是常规的 TEX "未限定选项",可以是单记号或者限定在配对大括号中的记号列表(这里我们称之为带括号的记号列表):后缀 n 说明是"常规(normal)"的选项形式。函数名说明它属于 seq 模块。

3.2 正式的命名语法

现在我们详细地看一下这些名称的语法。LATEX3 中的函数名包括三部分:

\⟨模块 *module*⟩_⟨描述 *description*⟩:⟨选项规范 *arg-spec*⟩

而变量名(至多)有四部分:

\⟨作用域 *scope*⟩_⟨模块 *module*⟩_⟨描述 *description*⟩_⟨类型 *type*⟩

所有的名称语法都包含

⟨模块 *module*⟩ 和 ⟨描述 *description*⟩

这两部分给出了命令的信息。

模块 *module* 是一组紧密相关的函数和变量的集合。典型的模块名包括整型参数(以及相关函数)模块 int,序列 seq 和盒子 box 等等。

提供新程序功能的宏包会按需要添加新的模块;开发者可以为模块选取任何未使用的名称,但只能包含字母。一般而言,模块名和模块前缀应当是关联起来的:例如,包含 box 函数的内核模块叫做 l3box。模块名和开发者的联系信息在文件 l3prefixes.csv 中。

描述 *description* 给出函数或参数的更多详细信息,并确定了唯一的名称。它应当包含字母和可能的 _ 字符。一般而言,描述部分应当使用 _ 来划分"单词"或者其它部分。例如,LATEX3 内核提供了 \if_cs_exist:N,并且正如所预料的那样用于检测一个命令是否存在。

进行变量操作的函数可以局部或者全局地执行。后者需要在函数名的第二部分中包含 g 标记。比如 \tl_set:Nn 是局部函数,而 \tl_gset:Nn 则是全局的。这两种类型的函数总是一起文档化,而作用域可以从函数名是否存在 g 来推断。关于变量作用域的信息详见下一小节。

3.2.1 区分私有和公共材料

TEX 语言的问题之一就是除了通过约定之外不支持名空间(name space)和封装。结果便是 LATEX 2ε 内核中的几乎所有内部命令最后都会被外部宏包使用并进行修改或扩展。造成的后果是现在几乎不可能改动 LATEX 2ε 内核中的任何东西而不造成破坏(即便很明显仅仅是个内部命令)。

在 expl3 中我们希望明确地区分公共接口(外部宏包可以使用或依赖)和私有函数/变量(不应当出现在模块之外),进而可以明显地改善目前这一状况。而这在没有严重的计算开销条件下几乎不可能实施,因此我们只是通过命名约定和一些支持机制来实现。不过,我们认为这种命名约定很容易理解和遵守,所以有信心认为该约定会被采用并提供想要的结果。

由模块创建的函数既可以是“公共”的(使用给定接口进行文档说明),也可以是“私有”的(只在该模块内部使用,因此没有正式文档)。很重要的一点是,只能使用带有文档的接口;同时有必要在函数名或变量名中表明是公共的还是私有的。

使用以下约定来明确地区分这两种情况。私有函数应当在模块名的开始加上 __。因此

```
\module_foo:nnn
```

是应当进行文档说明的公共函数,而

```
\__module_foo_nnn
```

是该模块私有的,在该模块外部不应当使用。

类似地,私有变量应当在模块名开始使用两个 _。因此

```
\l_module_foo_tl
```

是公共变量,而

```
\l__module_foo_tl
```

是私有的。

3.2.2 使用 @@ 和 l3docstrip 来标记私有代码

内部函数的正式语法可以清晰地区分公共和私有代码,但是同时也会包含冗余信息(每一内部函数或变量都包含 __⟨*module*⟩)。为了帮助开发者,l3docstrip 项目引入语法

```
%<@@=⟨module⟩>
```

这样允许在代码中使用 @@(对于变量的情况是 _@@)作为 __⟨*module*⟩ 的占位符。例如,l3docstrip 可以将如下代码

```
%<@@=foo>
%    \begin{macrocode}
\cs_new:Npn \@@_function:n #1
 ...
\tl_new:N \l_@@_my_tl
%    \end{macrocode}
```

转化并提取为

```
\cs_new:Npn \__foo_function:n #1
...
\tl_new:N \l__foo_function_my_tl
```

可以看出,`_@@` 和 `@@` 被映射到 `__`⟨*module*⟩。我们认为,在源文件中使用 `@@` 约定有助于区分函数和变量。

3.2.3 变量:作用域和类型

名称中的⟨作用域 *scope*⟩部分描述了变量是如何获取的。变量分为局部、全局和常值等类型。作用域类型出现在名称的开始处。使用的代码为:

c 常值(值不会变化的全局变量);

g 变量的值只应当全局设置;

l 变量的值只应当局部设置。

使用不同的函数来为局部和全局变量赋值;例如,`\tl_set:Nn` 和 `\tl_gset:Nn` 分别设置了局部和全局"记号列表"变量的值。请注意,给一个变量进行混合的局部和全局赋值是一种很不好的 TEX 做法;这可能会有耗尽保存的栈容量的风险。[1]

⟨类型 *type*⟩的内容位于可用的数据类型列表中;[2] 这包括 TEX 原始命令中的数据类型,比如各种寄存器。不过这些已经整合到由 LATEX 程序系统内部构建的数据类型中。

LATEX3 中的数据类型包括:

bool true 或者 false(LATEX3 不使用 `\iftrue` 或 `\iffalse`);

box 盒子寄存器;

clist 逗号分隔列表;

[1] 更多信息请参考 *The TEXbook*,301 页。

[2] 当然,例外是需要一个全新的数据类型。不过我们希望只有内核团队才需要创建新的数据类型。

coffin “带有句柄的盒子”——一种高层次的数据类型,可以执行 box 对齐操作;

dim “严格的”长度;

fp 浮点数值;

ior 输入流(用于文件读取);

iow 输出流(用于文件写入);

int 整型计数寄存器;

muskip 数学模式的“弹性”长度;

prop 属性列表;

seq 序列:用于实现列表(可以获取两端的值)和栈的数据类型;

skip “弹性”长度;

str TEX 字符串:一种特殊的 tl,其中所有的字符类型都是“其它”(类别码为 12),除了空格的类型是“空格”(类别码为 10);

tl “记号列表变量”:用于放置记号列表。

当 ⟨类型 *type*⟩ 和 ⟨模块 *module*⟩ 相同时(通常出现在基本模块中),⟨模块 *module*⟩ 部分出于美观原因通常会省略。

名称“记号列表”可能会引起困扰,因此介绍一些背景是有必要的。TEX 处理的是记号和记号列表,而不是字符,并且提供了两种方式来存储这些记号列表:在宏内部以及记号寄存器(toks)。在 LATEX3 中的实现意味着 toks 是不需要了,并且所有存储记号的操作都可以使用 tl 变量类型。

高级 TEX 开发者会注意到,一些变量类型是原生的 TEX 寄存器,而另一些则不是。一般来说,数据结构的底层实现会变动,而有文档说明的接口则是稳定的。例如,prop 数据类型一开始的实现是 toks,现在则构建在 tl 数据结构之上。

3.2.4 变量:说明指导

逗号列表和序列二者具有类似的特征。它们都使用特殊的定界符来标识各条目,并且都可以从两端读取。一般来说,“手动”创建逗号列表更容易一些,因为可以直接输入。用户输入通常采用逗号分隔列表的形式,因此许多情况中这是一种显然可用的数据类型。另一方面,序列会使用特殊的内部记号来分隔条目。这意味着它们可以用于逗号列表不能应用的场合(例如每一条目本身可能包含逗号)。一般而言,逗号列表主要用于创建项目中的固定列表以及处理逗号本身不会出现的用户输入。同时,序列应当用于存储任意数据列表。

expl3 使用序列数据结构实现栈。因此创建栈会首先创建一个序列，然后使用一些可以满足栈需求的序列函数(例如 `\seq_push:Nn` 等)。

由于底层 TEX 实现的固有性质，可以不首先声明就直接赋值给记号列表变量和逗号列表。然而，我们不支持这种行为。LATEX3 代码约定中，所有变量在使用之前都必须先声明。

expl3 宏包可以在导入时加载选项 `check-declarations`，用于确认所有的变量在使用前都已声明。不过这会带来性能的影响，因此主要用于开发中的测试而不是文档生产。

3.2.5 函数：选项规范

函数名称在冒号后以 ⟨选项规范 *arg-spec*⟩ 结尾。这给出函数使用的选项类型，并且提供了一种方便的手段用以给只有选项形式不同的类似函数命名（见下一节的例子)。⟨选项规范 *arg-spec*⟩ 包含一个（可能为空的）字母列表，对应于该函数的每一选项。字母(区分大小写)本身则带有所需选项类型的信息。

在所有函数都有的基本形式中，选项使用以下说明符之一：

n 不展开的记号或者带大括号的记号列表。
这是 TEX 的标准未定界宏选项。

N 单记号(与 n 不同，该选项一定不能在大括号内)。
带有 N 选项的函数中，一个典型的例子是 `\cs_set`，其中被定义的命令一定是不加大括号的。

p TEX 原始命令参数规范。
可以简单地取为 `#1#2#3`，但也可以使用任意定界选项语法，比如 `#1,#2\q_stop#3`。在定义函数时会使用。

T,F 这是选项 n 的特殊情形，用于条件命令中的真值判断。

另外还有两种意义更广泛的说明符：

D 意思是**不要使用**(**Do not use**)。这一特殊情形用于 TEX 原始命令。内核团队之外的开发者不要使用这些函数！

w 选项语法是“奇怪的(weird)”，不遵从任何标准规则。这用于带有非标准形式选项的函数：相关例子包括 TEX 层面的定界选项以及某些 `\if…` 原始命令之后需要的布尔测试。

在 n 选项的情况中，如果只含有单个记号，那么包裹的大括号在几乎所有场合都可以省略——那些即使是单记号选项也要加上大括号的函数会明确提及。然而，我们鼓励开发者总是为 n 选项加上大括号，这可以使得函数和选项的关系更清晰。

进一步的选项说明符是展开控制系统的一部分，在下一节中讨论。

4 展开控制

让我们看一些可能要进行的典型操作。假设我们维护一个文件打开的栈,并使用栈 \g_ior_file_name_seq 来保持对这些文件的跟踪(ior 是用于文件读取模块的前缀)。这里基本操作是将一个文件名推入栈内,可以通过以下操作完成:

```
\seq_gpush:Nn \g_ior_file_name_seq {#1}
```

其中 #1 是文件名。即,该操作会将文件名本身推入栈内。

然而,我们可能经常会面对的一种场合是,文件名存储在某一变量内,比如 \l_ior_curr_file_tl。此时我们想要检索该变量的值。如果简单地使用

```
\seq_gpush:Nn \g_ior_file_name_seq \l_ior_curr_file_tl
```

那么不会将变量的值推入栈内——推入栈内的仅仅是该变量本身的名称。取代的办法是,有必要使用一定合适数量的 \exp_after:wN (以及额外的大括号)来改变展开的顺序[3],即

```
\exp_after:wN
  \seq_gpush:Nn
\exp_after:wN
  \g_ior_file_name_seq
\exp_after:wN
  { \l_ior_curr_file_tl }
```

以上的例子也许是最简单的情形,但已经展示了代码是如何变得难以理解的。此外这里还有一个假设:存储的容器正好在一次展开后就完全展示其中的内容。依赖于这种机制就没法做任何合适的检查,为了得到正确的展开数量就必须完全搞清楚存储容器是怎样运行的。因此,LaTeX3 为开发者提供了一般性机制,使得代码紧凑并易于理解。

为了表达函数的选项需要特殊处理,只要在函数的选项规范部分中使用不同字母来标识需要的行为即可。在上面的例子中,使用命令

```
\seq_gpush:NV \g_ior_file_name_seq \l_ior_curr_file_tl
```

即可达到想要的效果。这里第二个选项 V 的意思是,在传递给基函数之前"先检索变量的值"。

以下字母可以用于表示选项传递给基函数之前的特殊处理:

[3] \exp_after:wN 是 TeX 原始命令 \expandafter 的 LaTeX3 名称

c 字符(character),作为命令名称使用。

该选项(记号或者带大括号的记号列表)是完全展开的;其结果必须是字符序列,并在之后用于构建命令名称(通过 `\csname …\endcsname` 方法)。该命令名是单记号,可以作为选项传递给函数。因此

```
\seq_gpush:cV { g_file_name_seq } \l_tmpa_tl
```

等价于

```
\seq_gpush:NV \g_file_name_seq \l_tmpa_tl
```

完全展开的含义是 (a) 整个选项必须是可展开的;(b) 任何变量都会转为该变量的内容。因此之前的例子也等价于

```
\tl_new:N \g_file_seq_name_tl
\tl_gset:Nn \g_file_seq_name_tl { g_file_name_seq }
\seq_gpush:cV { \tl_use:N \g_file_seq_name_tl } \l_tmpa_tl.
```

(记号列表变量是可展开的,此时我们可以省略访问函数 `\tl_use:N`。其它变量类型在这一环境下则需要使用合适的 `\<var>_use:N` 函数。)

V 变量的值(Value)。

此时,选项会使用寄存器的内容,可以是整型、长度类型的寄存器、记号列表变量等。该值作为带大括号的记号列表被传递给函数。可以用于带有 $\backslash\langle var\rangle$`_use:N` 函数(而不是浮点数或盒子),从而可以传递单"值"的变量。

v 寄存器的值(value),从命令名称的字符串构建而来。

该类型是 `c` 和 `V` 的结合,首先从选项中构造一个控制序列,然后将生成的寄存器的值传递给函数。可以用于带有 $\backslash\langle var\rangle$`_use:N` 函数(而不是浮点数或盒子),从而可以传递单值的变量。

x 完全展开(expand)的记号或者带大括号的记号列表。

该选项会像用 `\edef` 一样展开,然后展开值作为带大括号的记号列表传递给函数。展开会一直执行,直到遇到不可展开的记号。`x` 类型的选项不可以嵌套。

o 展开一层(one-level)的记号或带大括号的记号列表。

这说明该选项会像 `\expandafter` 那样展开一层,然后展开值作为带大括号的记号列表传递给函数。请注意,如果原始的参数是带大括号的记号列表,那么只有列表中的第一个记号会被展开。一般来说,对于简单的变量检索使用类型 `V` 比使用 `o` 更好一些。

f 在带有大括号的记号列表中递归地展开第一个(first)记号。

和 x 类型几乎相同,不同之处在于这里记号列表被完全展开,直到发现第一个不可展开的记号,而剩下的部分则保持不变。请注意,如果该函数在选项的一开始发现空格,那么会吞掉该空格,而不会展开接下来的记号。

4.1 越简单越好

如何安排函数的选项,使得选项在调用函数之前能够在合适的位置展开,这是每一个 TEX 开发者都经常面对的头疼问题。我们从 latex.ltx 中选取两个例子,进而说明展开控制机制能够对这一问题带来立竿见影的效果。

```
\global\expandafter\let
      \csname\cf@encoding \string#1\expandafter\endcsname
      \csname ?\string#1\endcsname
```

第一份代码片段本质上只是个全局的 \let 命令,它的两个选项必须在 \let 命令执行之前首先进行构造。#1 是一个控制序列,比如 \textcurrency。将当前字体编码字符(存储在\cf@encoding,需要完全展开)和符号名称连接起来,并获得第一个选项需要定义的记号。而第二个选项中的记号也是相同的过程,只不过使用默认编码 ?。而结果就是一团 \expandafter 和 \csname 交织在一起(当然 TEX 程序员都很喜欢用),导致代码本质上是不可读的。

使用这里提出的约定和功能,这项任务可以通过如下代码来实现:

```
\cs_gset_eq:cc
 { \cf@encoding \token_to_str:N  #1 } { ? \token_to_str:N #1 }
```

命令 \cs_gset_eq:cc 是全局的 \let,可以在定义之前先生成两个选项的命令名称。这样代码的可读性要好得多,出了问题也可以第一时间修改。(\token_to_str:N 是 \string 的 LATEX3 名称。)

这里是第二个例子:

```
\expandafter
    \in@
\csname sym#3%
    \expandafter
        \endcsname
    \expandafter
        {%
    \group@list}%
```

这份代码是另外一个函数定义的一部分。它首先做两件事情：通过展开一次 \group@list 获得一个记号列表，从 sym#3 获取一个记号的名称。然后调用函数 \in@ 并检测第一个选项是否在第二个选项代表的记号列表中。

同样地，我们可以极大改进这份代码。首先根据约定，我们要重命名函数 \in@，该函数的功能是检测第一个选项是否出现在第二个选项内。这样的函数有两个正常的“n”选项，并操作记号列表：因此很自然地可以命名为 \tl_test_in:nn。我们需要的函数变体需要定义合适的选项类型，因此名称为 \tl_test_in:cV。现在该代码片段可以简化为

```
\tl_test_in:cV { sym #3 } \group@list
```

可以使用序列 \l_group_seq 来代替裸记号列表 \group@list，从而进一步改进代码。请注意，除了不需要 \expandafter 之外，右大括号 } 之后的空格也会自动忽略，这是因为该程序环境中所有的空格都会被忽略。

4.2 函数推陈出新

对于很多常用函数，LaTeX3 内核提供了针对不同选项形式的一系列函数变种。类似地，我们也希望扩展宏包提供的新函数对于常见选项类型都是可用的。

然而构造新的函数变种形式有时候还是有必要的。为此，扩展模块提供了一种直接的机制，可以从带有“常规的”TeX 未定界选项的函数开始，创建带有任何选项类型的函数

为了阐述这一点，假设有一个带有三个常规选项的“基函数”\demo_cmd:Nnn。现在想要构造变种 \demo_cmd:cnx，其中第一个选项用于构造命令的“名称”，而第三个选项必须在传给 \demo_cmd:Nnn 之前完全展开。只要简单地使用如下代码就可以从基本形式生成变种形式：

```
\cs_generate_variant:Nn \demo_cmd:Nnn { cnx }
```

然后变种函数就可以使用了，比如：

```
\demo_cmd:cnx { abc } { pq } { \rst \xyz }
```

否则的话……就要使用如下可怕的代码！

```
\def \tempa {{pq}}%
\edef \tempb {\rst \xyz}%
\expandafter
    \demo@cmd:nnn
\csname abc%
    \expandafter
```

```
        \expandafter
    \expandafter
        \endcsname
    \expandafter
        \tempa
    \expandafter
        {%
    \tempb
        }%
```

另一个例子： 你想要根据已有函数 \demo_cmd_b:nnnnn 来声明变种函数 \demo_cmd_b:xcxcx，使得可以完全展开选项 1、3、5，然后使用 \csname 生成命令并传递选项 2、4。定义这样的函数仅需要：

```
\cs_generate_variant:Nn \demo_cmd_b:nnnnn { xcxcx }
```

扩展机制的目的在于，如果两个不同的扩展宏包都对某个已有命令进行新变种的实现，那么这两个定义是相同的，不会造成冲突。

5 发布

目前，expl3 模块被设计为在 LaTeX 2$_\varepsilon$ 之上导入。之后会基于这些代码生成 LaTeX3 格式。因此，目前可以在 LaTeX 2$_\varepsilon$ 宏包中使用这些代码，而独立的 LaTeX3 还在开发中。

虽然 expl3 宏集仍然处于实验性质阶段，不过可以认为已经相当稳定了。提供的语法约定和函数也可以广泛应用。当然某些函数可能还有有改动，但与整个 expl3 相比只是一些小改动。

新模块在成熟稳定后会加到 expl3 的发行版本中。目前 expl3 宏集已经包含了众多模块，在 LaTeX 2$_\varepsilon$ 宏包、文档类以及其它文件中包含如下一行代码即可导入绝大部分模块：

```
\RequirePackage{expl3}
```

expl3 中的模块可以认为是稳定的，可以适合在其基础上构建实际代码，具体如下：

l3basics 包含其它宏包使用的基本定义模块。

l3box 处理盒子的原始命令。

l3clist 操作逗号分隔记号列表的方法。

l3coffins 补充的盒子对齐操作。

l3expan 本文档之前讨论的选项展开模块。

l3int 实现整型数据类型 `int`。

l3keys 处理具有形式 `{ key1=val1 , key2=val2 }` 的列表，本模块的目的是 LaTeX3 版本的 xkeyval/kvoptions 宏包，不过输入的语法与 pgfkeys 更相似。

l3msg 用户交互：包括一些用于过滤信息的低层次的钩子（高层次的过滤信息接口还在开发）。

l3names 设置基本命名机制，并且重命名了所有的 TeX 原始命令。

l3prg 程序控制的一些结构，例如布尔变量类型 `bool`、一般的 do-while 循环以及条件流。

l3prop 实现了“属性列表”的数据类型，主要用于存储键值对。

l3quark “夸克（quark）”是定义为可以展开到自己的命令！因此绝对不可以被展开，否则会产生无穷递归。不过它们确实有很多应用，比如作为代码内特别的标识符或定界符。[4]

l3seq 实现了队列和栈等数据类型。

l3skip 实现了“弹性长度”数据类型 `skip`，“严格长度”数据类型 `dim`，以及数学模式“弹性长度”数据类型 `muskip`。

l3tl 实现了基本数据类型 *记号列表变量*（`tl var.`），用于存储命名记号列表：不带参数的 TeX 宏。

l3token 分析记号列表和记号流，包括查看接下来的记号以及检测相关类型。

6 从 LaTeX 2ε 到 LaTeX3

为了帮助开发者在现有的 LaTeX 2ε 宏包中使用 LaTeX3 代码，给出一些关于相关改变的简要注记是值得的。这里总的建议就是欢迎使用！以下的注记中一些是关于代码的，另一些则关于代码风格。

- expl3 主要关注于程序设计。所以某些领域仍然需要 LaTeX 2ε 的内部宏。例如你可能需要 `\@ifpackageloaded`，因为当前 LaTeX3 没有原生的宏包导入模块。

[4] 这个概念疑似从 quine 变形而来，后者指能够生成自己的程序代码，见 https://en.wikipedia.org/wiki/Quine_(computing) ——译注

- 用户水平的宏应当使用 xparse 宏包中的机制生成,这是 l3package 宏集的一部分,可以从 CTAN 或者 LaTeX3 SVN 仓库获得。
- 在内部层面上，大部分函数应当生成为 \long（使用 \cs_new:Npn）而不是“short”(使用 \cs_new_nopar:Npn)。
- 尽可能在使用之前声明所有变量和函数(使用 \cs_new:Npn, \tl_new:N 等)。
- 尽可能选择 “高层次” 函数而不是 “低层次” 函数。例如使用 \cs_if_exist:N(TF) 而不是 \if_cs_exist:N。
- 使用空格以增加代码的可读性。一般我们推荐以下的风格:

```
\cs_new:Npn \foo_bar:Nn #1#2
    {
        \cs_if_exist:NTF #1
            { \__foo_bar:n {#2} }
            { \__foo_bar:nn {#2} { literal } }
    }
```

  在 { 和 } 周围使用空格,除非是单独的 #1, #2 等。
- 将不同的代码项目分行放置:可读性比紧凑性有用的多。
- 在函数和变量中使用长的描述性的名称，而对于辅助函数则使用父函数加上 aux, auxi, auxii 等前后缀。
- 如果有任何疑问,通过 LaTeX-L 列表咨询团队:相关人员会很快回复!

7 宏包 expl3 的载入时选项

为了支持代码开发者,LaTeX 2$_\varepsilon$ 宏包 expl3 包含了少量导入时选项。这些选项都以键值形式给出,并可以识别 true 和 false 值。只给出选项名等价于该选项使用 true 值。

check-declarations 所有 LaTeX3 代码中的变量都必须声明。对于基于 TeX 寄存器的变量类型,这由 TeX 强制执行;而出于底层的存储原因,对于使用宏构造的变量则不保证。选项 check-declarations 则确保对于所有变量分配都开启检查,如果任何变量没有初始化则报错。另见 l3candidates 中的 \debug_on:n {check-declarations},可以进行更精细的控制。

log-functions 该选项用于在 .log 文件中开启每一新函数名的记录。这会有一份由导入模块

创建的全体函数列表（例外是 LaTeX3 的 bootstrap 代码需要的一小部分）。另见 l3candidates 中的\debug_on:n {log-functions}，可以进行更精细的控制。

enable-debug 除了 check-declarations 和 log-functions 之外，expl3 还提供了一些 \debug_… 函数（描述另见他处）在一个组内开启相应检查，这样可以允许更多的本地化检查和日志记录。只有当 expl3 导入时带有 enable-debug 选项时才可以使用这些函数。

driver 选择颜色、图像和其它驱动相关操作所需要的驱动类型。可用选项有：

auto 让 LaTeX3 自己确定正确的驱动。输出 DVI 时，会为 pdfTeX 和 LuaTeX 选择 dvips 后端，而对于 pTeX 和 upTeX 则会选择 dvipdfmx。这是标准设置。

latex2e 使用 graphics 宏包代替 LaTeX3 代码来选择驱动。

dvips 使用 dvips 驱动。

dvipdfmx 使用 dvipdfmx 驱动。

dvisvgm 使用 dvisvgm 驱动。

pdfmode 使用 pdfmode 驱动(pdfTeX 或 LuaTeX 的直接 PDF 输出)。

xdvipdfmx 使用 xdvipdfmx 驱动(仅用于 XeTeX)。

8 在 LaTeX 2ε 之外使用 expl3

在 LaTeX 2ε 宏包 expl3 之外，还有一种 “一般的” 方法可以导入该代码：expl3.tex。这可以在 plainTeX 语法中导入：

```
\input expl3-generic %
```

这可以在其它格式下进行程序设计层面的有关工作。使用这种方法时没有可用的选项，所以会自动选择“原生的”驱动。如果在 LaTeX 2ε 中使用这种“一般的”导入方法，那么代码会自动切换到合适的宏包路径上去。

使用一般接口导入程序设计层面后，可以使用命令 \ExplSyntaxOn、\ExplSyntaxOff、以及 interface3 中详细介绍的代码层面上的函数和变量。请注意不会导入其它使用 expl3 的 LaTeX 2ε 宏包：宏包导入取决于 LaTeX 2ε 的宏包管理机制。

9 引擎和原始命令需求

为了使用本团队提供的 expl3 和高层面宏包，目前原始命令需求的最小集合是

- TeX90 中的全部。

- ε-TeX 中的全部，除了 `\TeXXeTstate`, `\beginL`, `\beginR`, `\endL`, `\endR`（即，除了 TeX--XeT）。
- 等价于 pdfTeX 原始命令 `\pdfstrcmp` 的功能。

任何定义了 `\pdfoutput`（即，允许不经中间的 DVI 直接生成 PDF 文件）的引擎必须同时提供 `\pdfcolorstack`, `\pdfliteral`, `\pdfmatrix`, `\pdfrestore`, `\pdfsave` 或者等价的功能。Unicode 引擎必须提供以可展开方式生成字符记号的方法。

实际中，以下引擎可以满足这些需求

- pdfTeX v1.40 或之后版本。
- XeTeX v0.99992 或之后版本。
- LuaTeX v0.70 或之后版本。
- e-(u)pTeX 2012 中期或之后版本。

expl3 内核之外的其它模块可能需要额外的原始命令。特别要指出的是，第三方作者可能会明显地提高对原始命令覆盖的需求。

10 LaTeX3 项目组

LaTeX3 的开发由 LaTeX3 项目组实施。成员名单略。

参考文献

[1] Donald E Knuth *The TeXbook*. Addison-Wesley, Reading, Massachusetts, 1984.

[2] Goossens, Mittelbach and Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[3] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[4] Frank Mittelbach and Chris Rowley. "The LaTeX3 Project". *TUGboat*, Vol. 18, No. 3, pp. 195–198, 1997.