expl3 宏包和 LaTeX3 编程

LaTeX 项目 *

黄旭华 译

2022-08-23 发行

摘 要

本文档介绍了一组新的编程约定 (programming conventions)，这些约定旨在满足实施大规模 TeX 宏编程项目 (如 LaTeX) 的要求。这些编程约定是 LaTeX3 的底层。该系统的主要特点如下:

- 宏 (macros)(或在 LaTeX 术语中的命令) 分为 LaTeX 函数 (functions) 和 LaTeX 参数 (parameters)，以及包含相关命令的模块 (modules)；
- 基于这些分类的系统命名方案 (systematic naming scheme)；
- 控制函数参数展开 (expansion) 的简单机制。

这个系统正在被用作 LaTeX 项目中 TeX 编程的基础。请注意，该语言 (language) 既不适用于文档标记 (document mark-up)，也不适用于样式规范 (style specification)。相反，这些特性 (features) 将构建在这里描述的约定之上。

文档介绍了 expl3 编程接口 (programming interface) 背后的思想。有关 LaTeX 项目提供的编程层 (programming layer) 的完整文档，请参阅附带的 `interface3` 文档。

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

目　录

1 介绍

开发一个 LaTeX 2$_\varepsilon$ 以外的 LaTeX 内核的第一步是解决底层系统 (underlying system) 如何编程的问题。与目前混合使用 LaTeX 和 TeX 宏不同的是，LaTeX3 系统提供了自己的一致接口 (consistent interface)，可以控制 TeX 所需的所有函数 (functions)。这项工作的一个关键部分是确保所有内容都是文档化的 (documented)，这样 LaTeX 程序员和用户就可以高效地工作，而无需熟悉内核 (kernel) 的内部特性或 plain TeX。

expl3 宏包为 LaTeX 提供了这个新的编程接口 (programming interface)。为了使编程系统化，LaTeX3 使用了一些与 LaTeX 2$_\varepsilon$ 或 plain TeX 完全不同的约定。因此，以 LaTeX3 开始的程序员需要熟悉新语言的语法。

下一节将介绍这种语言在基于 TeX 的完整文档处理系统 (document processing system) 中的应用。然后，我们描述了命令名 (command names) 的语法结构的主要特征，包括函数名 (function names) 中使用的参数规范语法 (argument specification syntax)。

本文将解释这种参数语法 (argument syntax) 背后的实际思想 (practical ideas)，以及展开的控制机制 (expansion control mechanism) 和用于定义函数变体形式 (variant forms of functions) 的接口。

正如我们将要演示的那样，使用结构化命名方案 (structured naming scheme) 和函数变体形式 (variant forms for functions) 极大地提高了代码的可读性 (readability)，因此也提高了其可靠性 (reliability)。此外，经验表明，新语法产生的较长的命令名 (longer command names) 不会使编写 (*writing*) 代码的过程变得非常困难。

2 语言和接口

可以识别与基于 TEX 的文档处理系统 (document processing system) 中所需的各种接口相关的几种不同语言 (languages)。本节介绍我们认为对 LATEX3 系统最重要的功能。

文档标记 (document mark-up) 这包括那些要嵌入到文档 (`.tex` 文件) 中的命令 (通常称为标记 [tags])。

人们普遍认为，这种标记 (mark-up) 本质上应该是*声明性的* (*declarative*)。它可以是传统的基于 TEX 的标记，如 [3] 和 [2] 中所述的 LATEX 2$_\varepsilon$，或者是通过 HTML 或 XML 定义的标记语言 (mark-up language)。

更传统的 TEX 编码约定 (coding conventions)(如 [1] 中所述) 的一个问题是，TEX 原语格式命令 (primitive formatting commands) 的名称和语法被巧妙地设计为当作者直接用作文档标记或宏时是很“自然的 (natural)”。具有讽刺意味的是，逻辑标记 (logical mark-up) 的普遍性 (以及公认的优越性) 意味着在文档或作者定义的宏中几乎不需要这种显式格式化命令 (explicit formatting commands)。因此，它们几乎只被 TEX 程序员用来定义更高级别的命令，而且它们的独特的语法在这个社区中一点也不受欢迎。此外，如果它们不是作为原语 (例如 `\box` 或 `\special`) 被抢占的话，它们中的许多名称可以作为文档标记是非常有用的。

设计师接口 (designer interface) 这将 (人类) 排版设计师对文档的规范 (specification) 和“格式化文档”的程序 (program) 联系起来。理想情况下，它应该使用声明性语言 (declarative language)，这种语言可以方便地表达为各种文档元素 (document elements) 的布局 (layout) 指定的关系 (relationship) 和间距 (spacing) 规则。

这种语言不嵌入在文档文本 (document text) 中，它在形式上与文档标记语言 (document mark-up language) 有很大不同。对于 LATEX2.09 来说，这个级别几乎完全不存在，LATEX 2$_\varepsilon$ 在这个领域做了一些改进，但是仍然是这种情况，在 LATEX 中实现一个设计规范 (design specification) 需要比可接受的更多的“底层 (low-level)”编码。

程序员接口 (programmer interface) 该语言是在 TEX(或后续程序) 原语 (primitives) 的基础上实现基本排版功能的实现语言 (implementation language)。它还可以用于在 TEX(如当前的 LATEX 系统) 中实现两种语言。

最后一层 (last layer) 由本文档中描述的约定 (conventions) 所涵盖，本文档描述了一种旨在为 LATEX3 编码提供适当基础的系统。其主要特性概述如下:

- 所有命令的一致命名方案 (naming scheme)，包括 TEX 原语。
- 将命令分类为 LATEX 函数 (functions) 或 LATEX 参数 (parameters)，并根据其功能 (functionality) 将其划分为模块 (modules)。
- 一种控制参数展开 (controlling argument expansion) 的简单机制。
- 提供一组核心 LATEX 函数，足以处理队列 (queues)、集合 (stacks)、堆栈 (stacks) 和属性列表 (property lists) 等编程结构 (programming constructs)。
- 一种 TEX 编程环境 (programming environment)，例如，在该环境中忽略所有空格 (space)。

3 命名方案

LaTeX3 不使用 `@` 作为定义内部宏 (internal macros) 的“字母 (letter)”。相反，在内部宏名称中使用符号 `_` 和 `:` 来提供结构 (structure)。与 plain TeX 格式和 LaTeX 2_ε 内核不同，这些额外的字母 (extra letters) 仅用于宏名称 (macro name) 的部分 (parts) 之间 (没有奇怪的元音替换 [vowel replacement])。

虽然 TeX 实际上是一个宏处理器 (macro processor)，但是通过对 expl3 编程语言的约定，我们区分了*函数* (*functions*) 和*变量* (*variables*)。函数可以有参数，它们可以被展开或被执行。变量 (variables) 可以被赋值 (assigned values)，并在函数的参数中使用；变量不是直接使用的，而是由函数 (包括获取 [getting] 和设置函数 [setting functions]) 操纵的。具有相关功能的函数和变量 (例如，访问计数器或操纵令牌列表 [token lists] 等) 一起收集到一个*模块* (*module*) 中。

3.1 示例

在给出命名方案 (naming scheme) 的细节之前，这里有几个典型的例子来说明方案的特点：首先是一些变量名 (variable names)。

> **`\l_tmpa_box`** 是对应于 box 寄存器 (box register) 的局部变量 (local variable)(因为 `l_` 前缀)。
>
> **`\g_tmpa_int`** 是对应于整数寄存器 (integer register)(如一个 TeX 计数寄存器) 的全局变量 (global variable)(因为 `g_` 前缀)。
>
> **`\c_empty_tl`** 是始终为空 (empty) 的常量 (`c_`) 令牌列表变量 (token list variable)。

下面是一个典型函数名 (typical function name) 的示例：

`\seq_push:Nn` 是一个函数 (function)，它将第二个参数所指定的令牌列表 (token list) 放在第一个参数指定的堆栈 (stack) 上。两个参数的不同性质 (natures) 由 `:Nn` 后缀表示。第一个参数必须是“names”堆栈参数的单个令牌 (single token)：这样的单令牌参数 (single-token arguments) 表示为 `N`。第二个参数是一个普通的 TeX“未限制参数 (undelimited argument)”，它可以是单令牌的或对称的 (balanced)、用大括号分隔 (brace-delimited) 的令牌列表 (这里我们将其称为*大括号令牌列表 [braced token list]*)：`n` 表示这种“普通 (normal)”参数形式。函数名表明它属于 `seq` 模块 (module)。

3.2 正式的命名语法

现在我们将更详细地研究这些名称的语法。LaTeX3 中的函数名 (function name) 由三部分组成:

\⟨*module*⟩_⟨*description*⟩:⟨*arg-spec*⟩

而变量的名称有 (多达) 四个不同的部分:

\⟨*scope*⟩_⟨*module*⟩_⟨*description*⟩_⟨*type*⟩

所有名称 (names) 的语法包含:

⟨*module*⟩ 和 ⟨*description*⟩

它们都提供了关于命令的信息。

module，即模块，是密切相关的函数 (functions) 和变量 (variables) 的集合。典型的模块名 (module names) 包括整数参数 (integer parameters) 和相关函数的 `int`、序列 (sequences) 的 `seq` 和盒子 (box) 的 `box`。

提供新编程功能的宏包将根据需要添加新模块；程序员可以为模块选择任何未使用的名称，仅由字母组成。一般来说，模块名 (module name) 和模块前缀 (module prefix) 应该是相关的：例如，包含 `box` 函数的内核模块 (kernel module) 称为 `l3box`。l3prefixes.csv 中列出了模块名称和程序员的联系方式。

description，即描述，提供了有关函数 (function) 或参数 (parameter) 的更详细信息，并为其提供了唯一名称 (unique name)。它应该由字母 (letters) 和 _ 字符 (characters) 组成。一般来说，*description* 应使用 _ 分隔“单词 (words)”或名称中其他易于跟踪 (follow) 的部分。例如，LaTeX3 内核提供了 `\if_cs_exist:N`，正如预期的那样，它测试命令名 (command name) 是否存在。

当用于变量操作 (variable manipulation) 的函数可以局部或全局执行赋值时，后一种情况通过在函数名 (function name) 的第二部分包含 `g` 来表示。因此 `\tl_set:Nn` 是一个局部函数 (local function)，但 `\tl_gset:Nn` 起全局作用。这种类型的函数总是记录在一起 (documented together)，因此可以从 `g` 的存在与否在来推断作用范围 (scope of action)。有关变量范围 (variable scope) 的更多详细信息，请参见下一小节。

3.2.1 区分私有材料和公共材料

TeX 语言的一个问题是，除了约定 (convention) 之外，它不支持名称空格 (name spaces) 和封装 (encapsulation)。因此，LaTeX 2_ε 内核中的几乎所有内部命令最终都被扩

展包 (extension packages) 用作修改 (modifications) 或扩展 (extensions) 的入口点 (entry point)。其结果是，现在几乎不可能在不破坏某些东西的情况下更改 LaTeX 2_ε 内核中的任何内容 (即使它显然只是一个内部命令)。

在 expl3 中，我们希望通过明确区分公共接口 (public interfaces)(扩展包可以使用和依赖) 和私有 (private) 函数和变量 (不应出现在其模块之外)，来大幅改善上述情况。在没有严重的计算开销 (computing overhead) 的情况下，(几乎) 无法实现这一点，因此我们只能通过命名约定 (naming convention) 和一些支持机制 (support mechanisms) 来实现。然而，我们认为这种命名约定很容易理解 (understand) 和遵循 (follow)，因此我们相信这将被采用并能提供预期的结果。

由模块 (module) 创建的函数可以是“公共 (public)”(用已定义的接口记录) 或“私有 (private)”(仅在该模块中使用，因此未正式记录)。重要的是仅使用记录的接口 (documented interfaces)；同时，有必要在函数或变量的名称中显示它是公共的 (public) 还是私有的 (private)。

为了明确区分这两种情况，使用以下约定 (convention)。应在模块名 (module name) 的开头添加 `__` 来定义私有函数 (private functions)。因此

```
\module_foo:nnn
```

是一个公共函数 (public function) 应该被记录，而

```
\__module_foo:nnn
```

是该模块私有的，不应在该模块之外使用。

同样，私有变量 (private variables) 应该在模块名的开头使用`__`，这样

```
\l_module_foo_tl
```

是公共变量 (public variable)，而

```
\l__module_foo_tl
```

是私有变量。

3.2.2 使用 @@ 和 l3docstrip 标记私有代码

内部函数 (internal functions) 的正式语法 (formal syntax) 允许明确区分公共 (public) 和私有 (private) 代码，但包含冗余信息 (redundant information)(每个内部函数或变量包含 `__`⟨*module*⟩)。为了帮助程序员，l3docstrip 程序引入了以下语法

```
%<@@=⟨module⟩>
```

然后，在代码中它允许 @@ (和 _@@ 在变量情况下) 用作 __⟨*module*⟩ 的占位符 (place holder)。举个例子

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

提取时由 l3docstrip 转换为

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

正如您可以看到的，_@@ 和 @@ 都映射到 __⟨*module*⟩，因为我们认为这有助于在使用 @@ 约定时区分源代码 (source) 中的变量 (variables) 和函数 (functions)。

请注意，您必须使用 l3docstrip，而不是 .ins 文件中的 docstrip 程序才能使此工作正常 — 原始 LaTeX 2_ε docstrip 不理解 @@，只会将其复制到您的代码中而不做任何修改!

3.2.3 变量：声明

在格式良好的 expl3 代码中，应该始终在尝试赋值之前声明变量。即使对于基础 TeX 实现 (underlying TeX implementation) 允许直接赋值 (assignment) 的变量类型 (variable types) 也是如此。这既适用于直接设置 (setting directly)(\tl_set:Nn 等)，也适用于设置相等 (setting equal)(\tl_set_eq:NN 等)。

为了帮助程序员坚持这种方法，可能会给出调试选项 check-declarations。

```
\debug_on:n { check-declarations }
```

并且每当对未声明变量进行赋值时，将发出错误。这会影响性能，因此此选项只能用于测试。

3.2.4 变量：作用域和类型

这个 ⟨*scope*⟩ 即作用域的名称的一部分描述了如何访问 (accessed) 变量。变量分为局部 (local) 变量、全局 (global) 变量或常量 (constant) 变量。此 *scope* 类型 (type) 在名称开头显示为代码，使用的代码为:

c 常量 (constants)(值不应更改的全局变量);

g 其值只能全局设置的变量;

l 其值只能在局部设置的变量。

提供单独的函数 (functions) 来将数据赋值 (assign) 给局部和全局变量，例如，`\tl_set:Nn` 和 `\tl_gset:Nn` 分别设置了局部或全局“令牌列表 (token list)”变量的值。请注意，混合变量的局部和全局赋值是一种糟糕的 TEX 实践，这可能会耗尽存储堆栈 (save stack)。[1]

这个 ⟨*type*⟩ 即*类型*在可用 *data-types* 即*数据类型* [2] 列表中；这些包括原始 TEX 数据类型，如各种寄存器 (registers)，但在 LATEX 编程系统中添加了数据类型。

LATEX3 的数据类型有:

bool true 或 false (LATEX3 的实现不使用 `\iftrue` 或 `\iffalse`);

box 盒子寄存器 (box register);

cctab 类别代码表 (category code table);

clist 逗号分隔列表 (comma separated list);

coffin“带手柄的盒子 (box with handles)” — 进行 `box` (盒子) 对齐操作 (alignment operations) 的较高级别的数据类型;

dim“刚性 (rigid)”长度;

fp 浮点值 (floating-point values);

ior 输入流 (input stream)(用于从文件读取);

iow 输出流 (output stream)(用于写入文件);

int 整数值计数寄存器 (integer-valued count register);

muskip“橡皮 (rubber)”长度的数学模型 (math mode);

prop 属性列表 (property list);

---

[1] 更详细的信息请参考 *The TEXbook*, p. 301

[2] 当然，如果需要一种全新的数据类型 (data type)，则情况并非如此。但是，希望只有内核团队 (kernel team) 才能创建新的数据类型。

seq 序列 (sequence)：用于实现列表 (两端都有访问权限) 和堆栈的数据类型；

skip“橡皮 (rubber)”长度；

str TEX 字符串：`tl` 的一种特殊情况，其中所有字符都有类别 (category)“other”(类别 12)，但类别“space”(类别 10) 的空格 (spaces) 除外；

tl“令牌列表变量 (token list variables)”：令牌列表的占位符。

当 ⟨*type*⟩ 和 ⟨*module*⟩ 相同时 (在更基本的模块中经常发生)，出于美观，⟨*module*⟩ 部分经常被省略。

“token list (令牌列表)”这个名称可能会引起混淆，因此一些背景知识是有用的。TEX 处理令牌 (tokens) 和令牌列表，而不是字符 (characters)。它提供了两种存储这些令牌列表的方法：在宏 (macros) 中和作为令牌寄存器 (token registers)(`toks`)。LATEX3 中的实现 (implementation) 意味着不需要 `toks`，并且存储令牌 (storing tokens) 的所有操作都可以使用 `tl` 变量类型。

有经验的 TEX 程序员会注意到，列出的一些变量类型 (variable types) 是原生的 (native) TEX 寄存器，而其他的则不是。一般来说，数据结构 (data structure) 的底层 TEX 实现 (implementation) 可能有所不同，但 *documented interface* 即记录的接口是稳定的 (stable)。例如，`prop` 数据类型最初实现为 `toks`，但目前构建在 `tl` 数据结构之上。

3.2.5 变量：指导

逗号列表 (comma lists) 和序列 (sequences) 都具有相似的特征。它们都使用特殊的分隔符 (special delimiters) 将一个条目 (entry) 从下一个条目标记出来 (mark out)，并且两端都可以访问。一般来说，“手工 (by hand)”创建逗号列表更容易，因为它们可以直接键入。用户输入 (user input) 通常采用逗号分隔的列表形式，因此在许多情况下，这是可以使用的显而易见的数据类型。另一方面，序列 (sequences) 使用特殊的内部令牌 (internal tokens) 来分隔条目。这意味着它们可以用于包含逗号列表无法包含的材料 (例如，项目 [items] 本身可能包含逗号！)。一般来说，在程序内部创建固定列表 (fixed lists) 以及在不出现逗号的情况下处理用户输入时，逗号列表应该是首选的。另一方面，序列 (sequences) 应该用来存储任意的数据列表。

expl3 使用序列数据结构 (sequence data structure) 实现堆栈 (stacks)。因此，创建堆栈首先需要创建一个序列，然后使用以堆栈方式 (stack manner) 工作的序列函数 (sequence functions) (`\seq_push:Nn`，等等)。

由于底层 (underlying) TEX 实现的性质，可以在不首先声明的情况下为令牌列表 (token list) 变量和逗号列表 (comma lists) 赋值。但是，这是不受支持的行为。LATEX3 编码约定 (coding convention) 是所有变量必须在使用前声明。

expl3 包可以加载 `check-declarations` 选项，以验证在使用之前是否声明了所有变量。这对性能有影响，因此用于开发期间的测试，而不是用于生成文档 (production documents)。

3.2.6 函数：参数规范

函数名在冒号 (colon) 后面以 ⟨*arg-spec*⟩ 结尾。这表明了函数采用的参数类型，并提供了一种方便的方法来命名仅在参数形式上不同的类似函数 (参见下一节的示例)。

这个 ⟨*arg-spec*⟩ 由字母 (可能为空) 列表组成，每个字母表示函数的一个参数。字母 (letter)，包括其大小写 (case)，传达了所需参数类型的信息。

所有函数都有一个带参数的基形式 (base form)，使用一个下面的参数规范：

n 未展开的令牌或大括号令牌列表 (braced token list)。
这是一个标准的 TEX 未限制 (undelimited) 宏参数 (macro argument)。

N 单个令牌 (与 `n` 不同，参数不能被大括号包围)。
采用 `N` 参数的命令的典型示例是 `\cs_set`，其中定义的命令必须没有括号 (unbraced)。

p 原始 TEX 参数规范 (parameter specification)。
这可以很简单，比如 `#1#2#3`，但可以使用任意分隔的参数语法，比如：`#1,#2\q_stop#3`。这在定义函数时使用。

T,F 这些是 `n` 个参数的特殊情况，用于条件命令中的 true 和 false 代码。

还有另外两个更具普遍含义的规范 (specifiers)：

D 代表不要使用 **(do not use)**。这个特例用于 TEX 原语 (primitives)。这些函数没有标准的语法 (standardized syntax)，它们依赖于引擎 (engine)，而且它们的名称可以在没有警告的情况下更改，因此在宏包代码中强烈建议不要使用它们：程序员应该使用 interface3.pdf [3] 中记录的接口。

[3] 如果原语 (primitive) 提供了内核中尚未提供的功能，则鼓励程序员和用户向 `LaTeX-L` 邮件列表 (mailto:LATEX-L@listserv.uni-heidelberg.de) 描述它们的用例 (use-case) 和预期行为 (intended behaviour)，以便讨论可能的接口。目前，尽管未提供接口，程序员可以使用 l3styleguide.pdf 中描述的过程 (procedure)。

w 这意味着参数语法 (argument syntax) 是“怪异的 (weird)”，因为它不遵循任何标准规则 (standard rule)。它用于具有非标准形式参数的函数：例如 TEX 级别的分隔参数 (delimited arguments)，以及在某些原始 `\if…` 命令之后所需的布尔测试 (boolean tests)。

如果 `n` 参数由单个令牌 (single token) 组成，则几乎在所有情况下都可以省略周围的大括号—明确提到了强制使用大括号的函数，即使对于单个令牌参数 (token arguments) 也是如此。然而，鼓励程序员总是在 `n` 参数周围使用大括号，因为这使函数和参数之间的关系更清晰。

作为展开控制系统 (expansion control system) 的一部分，可以使用其他参数规范 (argument specifiers)。这些将在下一节中讨论。

4 展开控制

让我们看一下您可能希望执行的一些典型操作 (typical operations)。假设我们维护一个打开文件的堆栈 (stack)，并使用堆栈 `\g_ior_file_name_seq` 来跟踪它们 (`ior` 是用于文件读取模块的前缀)。这里的基本操作 (basic operation) 是将名称推送到这个堆栈上，这个堆栈可以由下面的操作 (operation) 完成：

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

其中 `#1` 是文件名 (filename)。换句话说，此操作将文件名按原样 (as is) 推送到堆栈上。

但是，我们可能会遇到文件名存储在某种变量中的情况，例如 `\l_ior_curr_file_tl`。在本例中，我们希望检索 (retrieve) 变量的值。如果我们简单地使用

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

不获取推送到堆栈上的变量的值，只获取变量名本身。相反，需要适当数量的 `\exp_after:wN` (还有额外的大括号) 来改变展开顺序 (order of expansion) [4]。例如：

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

上面的示例可能是最简单的情况，但是已经显示了代码如何更改为难以理解的内容。此外，这里还有一个假设：存储容器 (storage bin) 在完成一次展开后会显示其内容。这意味着您不能进行正确的检查 (proper checking)，而且您必须确切了解存储容器的工作方式，以便获得正确的展开数 (number of expansions)。因此，LaTeX3 为程序员提供了一个通用的方案 (general scheme)，使代码保持紧凑并易于理解。

为了表示函数的某些参数需要特殊处理，只需在函数的 arg-spec 部分使用不同的字母来标记想要的行为 (desired behavior)。在上面的例子中，人们会编写

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

[4] `\exp_after:wN` 是 TeX `\expandafter` 原语的 LaTeX3 名称。

以达到预期效果。这里的 V (第二个参数) 表示“检索变量的值”，然后将其传递给基函数 (base function)。

以下字母可用于表示在将参数传递给基函数之前对参数的特殊处理:

c 用作命令名的字符串 (character string)。

参数 (令牌或括号令牌列表) 被完全展开；结果必须是一个字符序列 (sequence of characters)，然后该字符序列用于构造命令名 (通过 \csname …\endcsname)。此命令名是作为参数传递给函数的单个令牌 (single token)。因此

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

相当于

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

完全展开 (full expansion) 意味着：(a) 全部参数必须可展开，(b) 任何变量都转换为其内容。所以前面的例子也相当于

```
\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.
```

令牌列表变量 (token list variables) 是可展开的, 我们可以省略存取器函数 (accessor function) \tl_use:N。其他变量类型需要适当的 \⟨*var*⟩_use:N 函数以在此上下文 (context) 中使用。

V 变量 (variable) 的值。

这意味着所讨论的寄存器 (register) 的内容被用作参数，可以是整数 (integer)、长度类型寄存器 (length-type register)、令牌列表变量 (token list variable) 或类似变量。该值作为大括号令牌列表 (braced token list) 传递给函数。可以应用于具有 \⟨*var*⟩_use:N 函数 (浮点 [floating points] 和盒子 [boxes] 除外) 的变量，因此只传递 (deliver) 单个“值”。

v 寄存器 (register) 的值，由用作命令名的字符串构造。

这是 c 和 V 的组合，它首先从参数构造 (constructs) 控制序列 (control sequence)，然后将结果寄存器 (resulting register) 的值传递给函数。可以应用于具有 \⟨*var*⟩_use:N 函数 (浮点 [floating points] 和盒子 [boxes] 除外) 的变量，因此只传递 (deliver) 单个“值”。

x 完全展开的令牌或大括号令牌列表 (braced token list)。

这意味着参数将像 `\edef` 的替换文本 (replacement text) 一样被展开，被展开后作为大括号的令牌列表 (braced token list) 传递给函数。展开将一直进行，直到只剩下不可展开的令牌。`x` 类型参数不能被嵌套 (nested)。

e 完全扩展的令牌或大括号令牌列表 (braced token list)，不需要双重 `#` 令牌。此扩展非常类似于 `x` 类型，但可能是嵌套的，不需要双重 `#` 令牌。

f 在大括号令牌列表 (braced token list) 中递归 (recursively) 展开第一个令牌。

与 `x` 类型几乎相同，但这里的令牌列表被完全展开，直到找到第一个不可展开的令牌，其余的保持不变。请注意，如果此函数在参数的开头找到一个空格 (space)，它会将其吞并，而不会展开下一个令牌。

o 一级扩展令牌 (One-level-expanded token) 或大括号令牌列表 (braced token list)。

这意味着将参数展开一级，就象 `\expandafter` 一样，并将展开为大括号令牌列表 (braced token list) 传递给函数。请注意，如果原始参数 (original argument) 是一个大括号令牌列表，则只会展开该列表中的第一个令牌。一般来说，对于简单的变量检索 (simple variable retrieval)，应首选使用 `V` 而不是使用 `o`。

4.1 简单意味着更好

任何用 TEX 编程的人都非常熟悉这样一个问题，即在调用函数之前，函数的参数要适当地扩展。为了说明展开控制 (expansion control) 如何立即缓解这个问题，我们将考虑从 `latex.ltx` 复制的两个示例。

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

第一段代码本质上只是一个全局 `\let`，在执行 `\let` 之前，必须首先构造它的两个参数。`#1` 是一个控制序列名称 (control sequence name)，如 `\textcurrency`。要定义的令牌是通过连接存储在 `\cf@encoding` (必须完全展开) 中的当前字体编码的 (current font encoding) 字符和符号名称来获得的。第二个令牌相同，只是它使用默认编码 `?`。结果是所有 TEX 程序员都喜爱的 `\expandafter` 和 `\csname` 交织在一起，代码基本上是不可读的 (unreadable)。

使用此处概述的约定 (conventions) 和功能 (functionality)，可以通过以下代码来完成任务：

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

命令 `\cs_gset_eq:cc` 是一个全局 `\let`，该全局 `\let` 在定义之前从两个参数中生成命令名。这样生成的代码更具可读性，而且第一次更有可能是正确的。(`\token_to_str:N` 是 `\string` 的 LaTeX3 名称。)

这是第二个例子:

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

这段代码是另一个函数定义的一部分。它首先生成两个东西：一个令牌列表，通过展开 `\group@list` 一次以及一个其名称来自 “`sym#3`”的令牌。然后调用 `\in@` 函数，并测试其第一个参数是否出现在第二个参数的令牌列表中。

同样，我们可以极大地改进代码。首先，我们将重命名函数 `\in@`，根据我们的约定，它测试它的第一个参数是否出现在第二个参数中。这样的函数接受两个普通的 (normal) ““`n`”” 参数，并对令牌列表进行操作：它可能被合理地命名为 `\tl_test_in:nn`。因此，我们需要的变量函数 (variant function) 将用适当的参数类型 (argument types) 定义，其名称为 `\tl_test_in:cV`。现在，这个代码片段很简单:

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

通过使用 `\l_group_seq` 序列而不是 `\group@list` 裸令牌列表 (bare token list)，可以进一步改进此代码。请注意，除了缺少 `\expandafter` 之外，`}` 后面的空格 (space) 也会自动忽略，因为在此编程环境中会忽略所有空格。

4.2 旧功能中的新功能

对于许多常见函数 (common functions)，LaTeX3 内核提供了具有一系列参数形式的变体 (variants)，类似地，提供新函数的扩展包 (extension packages) 将使它们以所有常见需要的形式 (forms) 可用。

然而，在某些情况下，有必要构建新的此类变体形式；因此，扩展模块 (expansion module) 提供了一种直接的机制 (straightforward mechanism) 来创建具有任何所需参数类型的函数，从一个带有“普通 (normal)”TEX 未分隔参数 (undelimited arguments) 的函数开始。

为了说明这一点，让我们假设您有一个“基函数 (base function)”`\demo_cmd:Nnn`，它接受三个普通参数，并且您需要构造变量 `\demo_cmd:cnx`，该变量的第一个参数用于构造命令的名称，而第三个参数必须在传递给 `\demo_cmd:Nnn` 之前被完全展开。要从基本形式生成变体形式 (variant form)，只需使用以下命令:

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

它定义了变体形式，这样你就可以编写，例如:

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

而不是 … ，嗯，是这样的!

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

另一个示例: 您可能希望声明一个函数 `\demo_cmd_b:xcxcx`，它是现有函数 `\demo_cmd_b:nnnnn` 的一个变体，它完全展开了参数 1、3 和 5，并使用 `\csname` 生成要作为参数 2 和 4 传递的命令。您需要的定义很简单

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

编写这种展开机制 (extension mechanism) 的目的是，如果某些现有命令的相同新形式由两个扩展包 (extension packages) 实现，那么这两个定义是相同的，因此不会发生冲突。

5 分发

expl3 模块 (modules) 被设计为在 LaTeX 2_ε 上加载。

核心 expl3 语言非常稳定，因此提供的语法约定 (*syntax conventions*) 和函数现在可以广泛使用。某些函数可能仍有更改，但与 expl3 的范围 (*scope*) 相比，这些更改将是微不足道的。对于此类弃用 (*deprecations*)，有一个健壮的机制 (*robust mechanism*)。

expl3 在 CTAN 上的分发分为三个包：l3kernel、l3packages 和 l3experimental。出于历史原因，

```
\RequirePackage{expl3}
```

加载现在作为 l3kernel 分发的代码。这个整体封装包 (monolithic package) 包含团队认为稳定的所有模块 (modules)，并且此代码中的任何更改都非常有限。因此，这些材料 (material) 适用于第三方宏包 (third-party packages)，而无需担心不受支持。所有这些代码都记录在 `interface3.pdf` 中。使用最新的 LaTeX 2_ε 内核，此代码内置于格式文件 (format file) 中，因此无需任何进一步步骤即可使用。

l3packages 包中的材料 (material) 也很稳定，但并不总是处于编程级别 (programming level)：最值得注意的是，xparse 是稳定的，适合更广泛的使用。

最后，l3experimental 包含可供公共使用 (public use) 但尚未集成到 l3kernel 中的模块。这些模块必须显式加载 (loaded explicitly)。团队预计，随着时间的推移，所有这些模块都将进入稳定状态 (stable status)，但在接口 (interface) 和功能细节 (functionality detail) 方面可能会更加灵活。关于这些模块的反馈非常宝贵。

6 从 LaTeX 2ε 迁移到 expl3

为了帮助程序员在现有的 LaTeX 2ε 包中使用 expl3 代码，可能需要一些关于进行更改的简短注释。欢迎在此提出建议！以下部分与代码有关，有些与编码风格 (coding style) 有关。

- expl3 主要关注编程 (programming)。这意味着某些区域仍然需要使用 LaTeX 2ε 内部宏。例如，您可能需要 `\@ifpackageloaded`，因为目前没有本地 expl3 包加载模块 (package loading module)。
- 用户级宏 (user level macros) 应该使用 xparse 包中的机制生成，xparse 是 `l3package` 包的一部分。
- 在内部级别 (internal level)，大多数函数都应该生成 `\long`(使用 `\cs_new:Npn`)，而不是“short”(使用`\cs_new_nopar:Npn`)。
- 如果可能，请在使用之前声明所有变量和函数 (使用 `\cs_new:Npn`, `\tl_new:N` 等)。
- 在可能的情况下，优先选择“higher-level(高级)”功能而不是“lower-level(低级)”功能。例如，使用 `\cs_if_exist:NTF` 而不是 `\if_cs_exist:N`。
- 使用空格 (space) 使代码可读。通常，我们建议采用如下布局 (layout)：

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

  其中 `{` 和 `}` 周围使用空格，但孤立的 (isolated)`#1`、`#2` 等除外。
- 将不同的代码项 (code items) 放在不同的行上：可读性比紧凑性更重要。
- 函数和变量使用长的描述性名称，辅助函数 (auxiliary functions) 使用父函数 (parent function) 名称加上 `aux`、`auxi`、`auxii` 等。
- 如果有疑问，请通过 LaTeX-L 列表询问团队：很快就会有人给您回复！

7 expl3 的加载时间选项

为了支持代码作者 (code authors)，LaTeX 2$_\varepsilon$ 的 expl3 包包含少量加载时间选项 (load-time options)。这些都以键值的方式工作，识别 `true` 和 `false` 值。仅提供选项名称 (option name) 等同于使用具有 `true` 值的选项。

`check-declarations` (*env.*) 应声明 expl3 代码中使用的所有变量。对于基于 TeX 寄存器 (registers) 的变量类型，TeX 会强制执行此操作，但对于那些使用宏作为底层存储系统 (underlying storage system) 构建的变量类型则不会强制执行。`check-declarations` 选项允许检查所有变量赋值，如果未初始化任何变量，则会发出错误消息。另请参阅 l3candidates 中的 `\debug_on:n {check-declarations}` 以获得更精细的控制 (finer control)。

`log-functions` (*env.*) `log-functions` 选项用于在 `.log` 文件中记录每个新函数名。这对于调试非常有用，因为这意味着有一个由每个加载的模块创建的所有函数的完整列表 (引导代码 [bootstrap code] 所需的非常少的函数除外)。另请参阅 l3candidates 中的`\debug_on:n {log-functions}` 以获得更精细的控制 (finer control)。

`enable-debug` (*env.*) 为了允许比 `check-declarations` 和 `log-functions` 提供的更多的本地化检查 (localized checking) 和日志记录 (logging)，expl3 提供了几个 `\debug_…` 函数 (在别处描述)，这些函数可以打开组内 (within a group) 的相应检查 (checks)。只有在使用 `enable-debug` 选项加载 expl3 时，才能使用这些函数。

`backend` (*env.*) 选择要用于颜色 (color)、图形 (graphics) 和相关操作 (related operations) 的后端 (backend)，这些操作依赖于后端。可用选项包括:

`dvips` 使用 `dvips` 驱动程序。

`dvipdfmx` 使用 `dvipdfmx` 驱动程序。

`dvisvgm` 使用 `dvisvgm` 驱动程序。

`luatex` 使用 LuaTeX 的直接 PDF 输出模式。

`pdftex` 使用 pdfTeX 的直接 PDF 输出模式。

`xetex` 使用 `dvipdfmx` 驱动程序的 XeTeX 版本

出于历史原因，也有 `pdfmode` 等效于 `luatex` 或 `pdftex`，`xdvipdfmx` 等效于 `xetex`，但这些都不推荐使用。

`suppress-backend-headers` (*env.*) `suppress-backend-headers` 选项禁止加载后端特定的头文件 (backend-specific header files)；目前，这只影响 `dvips`。此选项可用于支持基于 DVI 的路径 (DVI-based routes)，该路径不支持 `dvips` 使用的标题行 (`header` line)。

调试选项 (debugging options) 也可以使用 `\keys_set:nn { sys } { ... }`；只有在尚未加载后端 (backend) 的情况下，才能以这种方式提供 `backend` 选项。当 expl3 由 LaTeX 2_ε 格式预加载时，这种设置选项的方法很有用。

8 使用 expl3 和 LaTeX 2ε 以外的格式

除了 LaTeX 2ε 包 expl3 之外，还有一个代码的“generic(通用)”加载程序(loader)`expl3-generic.tex`。这可以使用 plain TeX 语法加载

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

这使编程层 (programming layer) 能够使用其他格式。由于没有选项可以通过这种方式加载，因此会自动使用“native(本地)”驱动程序。如果此“generic(通用)”加载程序与 LaTeX 2ε 一起使用，代码会自动切换到适当的包路径 (package route)。

使用通用接口 (generic interface) 加载编程层 (programming layer) 后，命令 `\ExplSyntaxOn` 和 `\ExplSyntaxOff` 以及 interface3 详述的代码级 (code-level) 函数和变量可用。请注意，使用 expl3 的其他 LaTeX 2ε 包不可加载：包加载 (package loading) 取决于 LaTeX 2ε 包管理机制 (package-management mechanism)。

9 引擎/基本要求

为了使用团队提供的 expl3 和更高级别的包，目前只有最小的基本需求集 (minimal set of primitive requirements)

- 所有这些都来自 TEX90。
- 来自 ε-TEX 的所有数据，不包括\TeXXeTstate、\beginL、\beginR、\endL 和 \endR(即不包括 TEX--XETEX)。
- 功能等同于 pdfTEX 原语 \pdfstrcmp。

任何定义 \pdfoutput(即允许直接生成 PDF 文件而无需 DVI 中间文件) 的引擎还必须提供\pdfcolorstack、\pdfliteral、\pdfmatrix、\pdfrestore 和 \pdfsave 或等效功能 (equivalent functionality)。完全 Unicode 引擎必须提供一种以可扩展方式 (expandable manner) 生成字符令牌 (character tokens) 的方法。

实际上，这些要求是由引擎 (engines) 来满足的

- pdfTEX v1.40 或更高版本。
- XETEX v0.99992 或更高版本。
- LuaTEX v0.95 或更高版本。
- e-(u)pTEX 2012 年中或更高版本。

expl3 核心之外的其他模块可能需要额外的原语 (primitives)。特别是，第三方作者 (third-party authors) 可能会大大扩展原始覆盖要求 (primitive coverage requirements)。

10 LaTeX 项目

LaTeX3 的开发由 LaTeX 项目 (https://www.latex-project.org/latex3/) 执行。

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 LaTeX Project". *TUGboat*, Vol. 18, No. 3, pp. 195–198, 1997.

11 expl3 实现

这里的实现 (implementation) 包括几个方面。有两个“加载程序 (loaders)”需要定义：代码中特定于 LaTeX 2_ε 或非 LaTeX 2_ε 格式的部分。它们必须以完全不同的方式涵盖相同的概念：因此，大部分代码是以单独的块 (separate blocks) 的形式给出的。还有一小段代码用于启动“有效负载 (payload)”：这是为了确保加载始终以正确的方式进行。

11.1 加载程序联锁

一个简短的设置，用于检查加载程序 (loader) 和“有效负载 (payload)”版本是否匹配。

\ExplLoaderFileDate

由于 DocStrip 用于为来自同一源的所有文件生成 \ExplFileDate，因此它必须匹配。故加载程序只需使用新名称 (new name) 保存此信息。

```
⟨*loader⟩
\let\ExplLoaderFileDate\ExplFileDate
⟨/loader⟩
```

(*End definition for* `\ExplLoaderFileDate`. *This function is documented on page* ??.)

\c__kernel_expl_date_tl

LaTeX 2_ε 加载程序 (loader) 存储 \ExplFileDate 的私有副本 (private copy)，该副本在构建格式时是固定的，以后不能更改。而 \ExplFileDate 确保加载程序 (2ekernel 或 package) 版本与 expl3-code.tex 的版本匹配。这可以确保最终的 \usepackage{expl3} 版本与格式中的版本匹配。在用户树 ((user tree)) 中常常会有杂散的格式文件 (stray format files)，这些文件会因版本不匹配 (例如https://github.com/latex3/latex3/issues/781) 而引发错误。

这个代码块也发送给通用加载程序 (generic loader)，因为这里没有预加载 expl3，所以内核日期 (kernel date) 应该等于加载程序日期 (loader date)。

```
⟨*2ekernel | generic⟩
\begingroup
  \catcode`\_=11
  \expandafter
  \ifx\csname c__kernel_expl_date_tl\endcsname\relax
    \global\let\c__kernel_expl_date_tl\ExplFileDate
  \fi
\endgroup
⟨/2ekernel | generic⟩
```

(*End definition for* `\c__kernel_expl_date_tl`.)

联锁测试 (interlock test) 本身很简单：必须定义 `\ExplLoaderFileDate` 并与 `\ExplFileDate` 相同。因为这必须同时适用于 LaTeX 2_ε 和其他格式，所以需要进行一些自动检测，这样做可以避免 LaTeX 2_ε 和其他格式有两个非常相似的块 (blocks)。

```
⟨*!loader⟩
\begingroup
  \def\next{\endgroup}%
  \expandafter\ifx\csname PackageError\endcsname\relax
    \begingroup
      \def\next{\endgroup\endgroup}%
      \def\PackageError#1#2#3%
        {%
          \endgroup
          \errhelp{#3}%
          \errmessage{#1 Error: #2!}%
        }%
  \fi
  \expandafter\ifx\csname ExplLoaderFileDate\endcsname\relax
    \def\next
      {%
        \PackageError{expl3}{No expl3 loader detected}
          {%
            您试图直接使用 expl3 代码，而不是使用正确的加载程序。expl3 加载将中止。
          }%
        \endgroup
        \endinput
      }
  \else
    \ifx\ExplLoaderFileDate\ExplFileDate
    \else
      \def\next
        {%
          \PackageError{expl3}{Mismatched expl3 files detected}
            {%
              您试图用不匹配的文件加载 expl3：可能有一个或多个“本地安装”的文件
              存在冲突。expl3 加载将中止。
```

```
            }%
          \endgroup
          \endinput
        }%
    \fi
\fi \next
```
⟨/!loader⟩

有效负载 (payload) 的重新加载测试 (reload test)，以防万一。

```
⟨*!loader⟩
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname ver@expl3-code.tex\endcsname\relax
  \expandafter\edef\csname ver@expl3-code.tex\endcsname
    {%
      \ExplFileDate\space
      L3 编程层
    }%
\else
  \expandafter\endinput
\fi
⟨/!loader⟩
```

所有好处：记录所用代码的版本 (为了记录完整性 [log completeness])。由于这是或多或少的 `\ProvidesPackage` 没有一个单独的文件 (separate file)，并且这也需要在没有 LaTeX 2_ε 的情况下工作，只需将信息直接写入日志 (log) 即可。

```
⟨*!loader⟩
\immediate\write-1 %
  {%
    宏包：expl3
      \ExplFileDate\space
      L3 编程层（代码）%
  }%
⟨/!loader⟩
```

11.2 LaTeX 2_ε 加载程序

使用 LaTeX 2_ε 加载 (loading) 可以作为格式的一部分 (预加载) 或作为一个包。我们必须考虑两种可能的路径 (paths)，当然，要将包加载到预加载的 (pre-load)。这意味着

这里的代码必须能够安全地防止重新加载 (re-loading)。

```
⟨*package & loader | 2ekernel⟩
```

标识 (identify) 包或添加到格式化消息 (format message) 中。

```
⟨*2ekernel⟩
\everyjob\expandafter{\the\everyjob
  \message{L3 programming layer <\ExplFileDate>}%
}
⟨/2ekernel⟩
⟨*!2ekernel⟩
\ProvidesPackage{expl3}
  [%
    \ExplFileDate\space
    L3 编程层 (加载程序)
  ]%
⟨/!2ekernel⟩
```

\ProvidesExplPackage
\ProvidesExplClass
\ProvidesExplFile

对于在此基础上构建的其他包 (packages) 和类 (classes)，不需要每次都使用 \ExplSyntaxOn 是很方便的。所有宏都使用相同的内部宏和适当的 LaTeX 2_ε 命令。

```
\protected\def\ProvidesExplPackage
  {\@expl@provides@file@@Nnnnnn\ProvidesPackage{Package}}
\protected\def\ProvidesExplClass
  {\@expl@provides@file@@Nnnnnn\ProvidesClass{Document Class}}
\protected\def\ProvidesExplFile
  {\@expl@provides@file@@Nnnnnn\ProvidesFile{File}}
```

(*End definition for* \ProvidesExplPackage*,* \ProvidesExplClass*, and* \ProvidesExplFile*. These functions are documented on page* ??.)

\@expl@provides@file@@Nnnnnn
\@expl@provides@generic@@wnnw

我们需要检查 \Provides⟨*thing*⟩ 是否存在，因为我们需要在 LaTeX 2_ε 内核中尽早加载。

```
\protected\long\def\@expl@provides@file@@Nnnnnn#1#2#3#4#5#6%
  {%
    \ifnum0%
        \ifdefined#11\fi
        \ifx\relax#1\else1\fi
        =11
      \expandafter#1%
    \else
```

```
      \@expl@provides@generic@@wnnw{#2}%
    \fi
      {#3}[{#4 \ifx\relax#5\relax\else v#5\space\fi #6}]%
    \ExplSyntaxOn
  }
\protected\long\def\@expl@provides@generic@@wnnw#1\fi#2[#3]%
  {%
    \immediate\write-1{#1: #2 #3}%
  }
```

(*End definition for* `\@expl@provides@file@@Nnnnnn` *and* `\@expl@provides@generic@@umnw`*. These functions are documented on page* **??**.)

加载业务结束：这将保留 `\expl3` 语法。测试确保我们只加载一次，而不需要知道是否有预加载步骤 (preloading step)。

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname tex\string _let:D\endcsname\relax
  \expandafter\@firstofone
\else
  \expandafter\@gobble
\fi
  {\input expl3-code.tex }%
```

检查引导程序代码 (bootstrap code) 是否中止了加载：如果中止了加载，则在这里静默 (silently) 退出。

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname tex\string _let:D\endcsname\relax
  \expandafter\endinput
\fi
```

⟨`@@=expl`⟩

此时, 如果定义了 `\c__kernel_expl_date_tl`, 只需调用 `\__kernel_dependency_version_check:Nn`, 检查它是否与 `\ExplLoaderFileDate` 匹配。这里只在 `\c__kernel_expl_date_tl` 存在的情况下执行测试，因为此文件可以以 LaTeX 2_ε 格式加载，而不预加载 expl3，其中令牌列表 (token list) 不存在。

这都是在 package docstrip 中完成的，因为它不适用于 expl3.ltx。

⟨*package⟩

```
\ifcsname\detokenize{c__kernel_expl_date_tl}\endcsname
```

```
  \expandafter\@firstofone
\else
```

如果 \c__kernel_expl_date_tl 不存在，我们可能会以未预加载 expl3 的格式加载，或者以出现上述错误的较早版本 (尽管仍然兼容) 加载。如果作为包加载，expl3-code.tex 已被读取，此时 expl3 语法已开启。否则，它已加载到稍旧的内核中，因此我们将触发不兼容错误消息并中止加载。

```
  \ifodd\csname\detokenize{l__kernel_expl_bool}\endcsname
```

在包模式 (package mode) 下，所有文件都会一次性加载，因此版本会匹配。我们只需设置 \c__kernel_expl_date_tl，以便进一步的依赖关系不会被中断:

```
    \global\expandafter\let\csname\detokenize
      {c__kernel_expl_date_tl}\endcsname\ExplLoaderFileDate
    \expandafter\expandafter
    \expandafter\@gobble
  \else
```

在不兼容的版本中重新加载是一个错误:

```
    \expandafter\expandafter
    \expandafter\@firstofone
  \fi
\fi
  {\csname\detokenize{__kernel_dependency_version_check:Nn}\endcsname
     \ExplLoaderFileDate{expl3.sty}}%
⟨/package⟩
```

在这里，我们还可以检测是否正在重新加载 (reloading)。这段代码进入 expl3.ltx 和 expl3.sty，前者加载为 LaTeX 2ε 格式。第一次加载此代码时，\g__expl_reload_bool 布尔值 (boolean) 不存在 (下面 \ifcsname 的 \else 分支)，因此我们创建了它。如果 \ifcsname 为 true，则执行 \ExplSyntaxOn(因为在重新加载时，不会再次读取 expl3-code.tex)，并将 \g__expl_reload_bool 设置为 true。

```
\ifcsname\detokenize{g__expl_reload_bool}\endcsname
  \ExplSyntaxOn
  \bool_gset_true:N \g__expl_reload_bool
\else
  \bool_new:N \g__expl_reload_bool
\fi
```

\c__expl_def_ext_tl

LaTeX 2_ε 所需，并避免重新加载问题。最好显式检查变量，而不是使用 \g__expl_reload_bool，因为有些变量只出现在一个代码文件中，所以 \g__expl_reload_bool 并不一定意味着变量已经声明。

```
\tl_if_exist:NF \c__expl_def_ext_tl
  { \tl_const:Nn \c__expl_def_ext_tl { def } }
```

(*End definition for* \c__expl_def_ext_tl.)

__kernel_sys_configuration_load:n

__kernel_sys_configuration_load_std:n

为了加载配置，我们有以下几种情况：

- expl3 expl3 是预加载的 (pre-loading)：在加载配置 (configuration) 时，我们已经有了完整的文件加载堆栈 (loading stack)，这里只需要标准版本的代码。
- 这个包是预加载的 (pre-loading)：我们再次使用标准版本，但我们还没有测试。
- 该包在没有预加载代码 (pre-loaded code) 的情况下使用：我们需要手动管理 expl3 语法。

```
\cs_gset_protected:Npn \__kernel_sys_configuration_load:n #1
⟨*!2ekernel⟩
  {
    \ExplSyntaxOff
    \cs_undefine:c { ver@ #1 .def }
    \@onefilewithoptions {#1} [ ] [ ]
      \c__expl_def_ext_tl
    \ExplSyntaxOn
  }
\cs_gset_protected:Npn \__kernel_sys_configuration_load_std:n #1
⟨/!2ekernel⟩
  {
    \cs_undefine:c { ver@ #1 .def }
    \@onefilewithoptions {#1} [ ] [ ]
      \c__expl_def_ext_tl
  }
```

(*End definition for* __kernel_sys_configuration_load:n *and* __kernel_sys_configuration_load_std:n.)

\l__expl_options_clist

```
⟨*!2ekernel⟩
\clist_if_exist:NF \l__expl_options_clist
  { \clist_new:N \l__expl_options_clist }
\DeclareOption*
  { \clist_put_right:NV \l__expl_options_clist \CurrentOption }
\ProcessOptions \relax
⟨/!2ekernel⟩
```

(*End definition for* `\l__expl_options_clist`.)

相当标准的设置创建 (setting creation)。

```
\keys_define:nn { sys }
  {
    backend .choices:nn =
      { dvipdfmx , dvips , dvisvgm , luatex , pdftex , pdfmode , xdvipdfmx , xetex }
      { \sys_load_backend:n {#1} } ,
    check-declarations .code:n =
      {
        \sys_load_debug:
        \debug_on:n { check-declarations }
      } ,
    driver .meta:n = { backend = #1 } ,
    enable-debug .code:n =
      \sys_load_debug: ,
    log-functions .code:n =
      {
        \sys_load_debug:
        \debug_on:n { log-functions }
      } ,
    suppress-backend-headers .bool_gset_inverse:N
      = \g__kernel_backend_header_bool ,
    suppress-backend-headers .initial:n = false ,
    undo-recent-deprecations .code:n = {} % A stub
  }
```

`\@expl@sys@load@backend@@` 后端 (backend) 必须在文档开始时就位 (place)：必须在检查全局选项 (global options) 以使用之前就位。

在本包中定义的 `\@expl@...@@` 宏是 LaTeX 2_ε 的接口。目前 (这将随着下一版本的 LaTeX 2_ε 而改变) 在代码的这一点上有两种可能的情况：`\@expl@sys@load@backend@@` (以及其他) 已经存在，因为它们是在 `ltexpl.ltx` 中定义的 (在 `2ekernel` 模式中) 或在 `expl3.ltx`(在 `package` 模式中) 中定义的。

在 `2ekernel` 模式中，如果它们存在，我们将使用未来 (2020-10-01) 发布的 LaTeX 2_ε，我们不需要 (也不能) 修补 LaTeX 2_ε 的内部，因为这些命令已经存在。虽然它们不存在于 `2ekernel` 模式中，但我们使用的是较旧版本的内核，所以我们必须进行修补。

在 `package` 模式中，如果存在这些命令，那么我们使用的是预加载了 `expl3` 的 LaTeX 2_ε 版本 (任何版本)，而且在任何情况下，补丁 (patching) 已经完成，或者宏本身就是格式，因此无需执行任何操作。但如果在 `package` 模式中这些宏不存在，我们有一个更旧版本的 LaTeX 2_ε，它甚至没有预加载 `expl3`，因此补丁是必要的。

这意味着在 `2ekernel` 和 `package` 模式中,我们必须检查 `\@expl@sys@load@backend@@` 是否存在，如果不存在，则修补一些 LaTeX 2_ε 内部构件。

在更新的 LaTeX 2_ε 中，这些宏在 `ltexpl.dtx` 中有一个空定义 (empty definition)，以防加载此文件 (`expl3.ltx`) 时发生错误，因此可以在 LaTeX 2_ε 内核中安全使用它们。

`\@expl@sys@load@backend@@` 在 `\document` 的开头插入，但在关闭后，由 `\begin` 开始的组 (group)。当使用 `\tl_put_left:Nn` 修补 `\document` 中的后端加载 (backend loading) 时，我们需要确保它发生在组级别 (group level)0，因此出现了奇怪的 `\endgroup...\begingroup` 问题。

这段代码只应在未预加载 `expl3` 的情况下，在 LaTeX 2_ε 中加载 `expl3.sty` 时执行，因此我们检查 `\@expl@sys@load@backend@@` 是否存在。

```
\cs_if_exist:NF \@expl@sys@load@backend@@
  {
    \tl_put_left:Nn \document
      {
        \endgroup
        \@expl@sys@load@backend@@
        \begingroup
      }
  }
```

现在我们无论如何都要定义它。

```
\cs_gset_protected:Npn \@expl@sys@load@backend@@
  {
    \str_if_exist:NF \c_sys_backend_str
```

```
      { \sys_load_backend:n { } }
  }
```

(*End definition for* `\@expl@sys@load@backend@@`. *This function is documented on page* **??**.)

处理宏包选项 (package options)。

```
⟨*!2ekernel⟩
\keys_set:nV { sys } \l__expl_options_clist \str_if_exist:NF \c_sys_backend_str
  { \sys_load_backend:n { } }
⟨/!2ekernel⟩
```

```
⟨*!2ekernel⟩
\bool_if:NT \g__expl_reload_bool
  {
    \cs_gset_eq:NN \__kernel_sys_configuration_load:n
      \__kernel_sys_configuration_load_std:n
    \ExplSyntaxOff
    \file_input_stop:
  }
⟨/!2ekernel⟩
```

现在或在下一次运行期间加载代码的动态部分 (dynamic part)。

```
\cs_if_free:cTF { ver@expl3.sty }
  {
    \tex_everyjob:D \exp_after:wN
      {
        \tex_the:D \tex_everyjob:D
        \sys_everyjob:
      }
  }
  { \sys_everyjob: }
```

`\s__expl_stop` 内部扫描标记。此代码必须是重载安全的 (reload-safe)，因此必须使用`\if_cs_exist:N` (`\cs_if_exist:NF`) 保护此代码，因为它对于等于 `\scan_stop:` 的控制序列 (control sequences) 返回 false。

```
\reverse_if:N \if_cs_exist:N \s__expl_stop
  \scan_new:N \s__expl_stop
\fi:
```

(*End definition for* `\s__expl_stop`.)

\@pushfilename
\@popfilename
__expl_status_pop:w
\@expl@push@filename@@
\@expl@push@filename@aux@@
\@expl@pop@filename@@

这里的想法是使用 LaTeX 2_ε 的 \@pushfilename 和 \@popfilename 来跟踪当前语法状态 (syntax status)。这可以通过在每次推送到堆栈 (stack) 时保存当前状态标志 (status flag)，然后在弹出阶段 (pop stage) 恢复它，并检查代码环境 (code environment) 是否仍应处于活动状态来实现。

对于 \@expl@sys@load@backend@@，这里的代码遵循与上面相同的修补逻辑。

```
\cs_if_exist:NF \@expl@push@filename@@
  {
    \tl_put_left:Nn  \@pushfilename { \@expl@push@filename@@ }
    \tl_put_right:Nn \@pushfilename { \@expl@push@filename@aux@@ }
  }
\cs_gset_protected:Npn \@expl@push@filename@@
  {
    \exp_args:Nx \__kernel_file_input_push:n
      {
        \tl_to_str:N \@currname .
        \tl_to_str:N \@currext
      }
    \tl_put_left:Nx \l__expl_status_stack_tl
      {
        \bool_if:NTF \l__kernel_expl_bool
          { 1 }
          { 0 }
      }
    \ExplSyntaxOff
  }
```

需要使用这个小技巧来获取正在加载的文件的名称，以便我们可以记录 (record) 它。

```
\cs_gset_protected:Npn \@expl@push@filename@aux@@ #1#2#3
  {
    \str_gset:Nn \g_file_curr_name_str {#3}
    #1 #2 {#3}
  }
\cs_if_exist:NF \@expl@pop@filename@@
  {
    \tl_put_right:Nn \@popfilename
      { \@expl@pop@filename@@ }
```

```
  }
\cs_gset_protected:Npn \@expl@pop@filename@@
  {
    \__kernel_file_input_pop:
    \tl_if_empty:NTF \l__expl_status_stack_tl
      { \ExplSyntaxOff }
      { \exp_after:wN \__expl_status_pop:w \l__expl_status_stack_tl \s__expl_stop }
  }
```

弹出辅助函数 (pop auxiliary function) 从堆栈 (stack) 中删除第一项 (first item)，保存堆栈的其余部分，然后执行测试 (test)。此处的标志 (flag) 不是正确的 `bool`，因此使用了低级测试 (low-level test)。

```
\cs_gset_protected:Npn \__expl_status_pop:w #1#2 \s__expl_stop
  {
    \tl_set:Nn \l__expl_status_stack_tl {#2}
    \int_if_odd:nTF {#1}
      { \ExplSyntaxOn }
      { \ExplSyntaxOff }
  }
```

(*End definition for* `\@pushfilename` *and others. These functions are documented on page* ??.)

`\l__expl_status_stack_tl`

由于 expl3 本身无法加载已经处于活动状态的代码环境 (code environment)，因此可以安全地调用包末尾的 `\ExplSyntaxOff`。

```
\tl_if_exist:NF \l__expl_status_stack_tl
  {
    \tl_new:N \l__expl_status_stack_tl
    \tl_set:Nn \l__expl_status_stack_tl { 0 }
  }
```

(*End definition for* `\l__expl_status_stack_tl`.)

按照承诺 (promised) 整理配置加载 (configuration loading)。

```
⟨*!2ekernel⟩
\cs_gset_eq:NN \__kernel_sys_configuration_load:n
  \__kernel_sys_configuration_load_std:n
⟨/!2ekernel⟩
```

对于预加载，我们必须手动禁用语法 (syntax)。

```
⟨*2ekernel⟩
\ExplSyntaxOff
⟨/2ekernel⟩
⟨/package & loader | 2ekernel⟩
```

11.3 通用加载程序

```
⟨*generic⟩
```

通用加载程序 (generic loader) 从测试开始，以确保当前格式不是 LaTeX 2_ε！

```
\begingroup
  \def\tempa{LaTeX2e}%
  \def\next{}%
  \ifx\fmtname\tempa
    \def\next
      {%
        \PackageInfo{expl3}{Switching from generic to LaTeX2e loader}%
```

`\relax` 停止 `\RequirePackage` 扫描日期参数 (date argument)。加载包之后放入 `\endinput` 至关重要，否则 `\endinput` 将在文件第一行末尾关闭文件 `expl3.sty`：实际上，只要 `expl3.sty` 处于打开状态，就不可能关闭文件 `expl3-generic.tex`。

```
        \RequirePackage{expl3}\relax \endinput
      }%
  \fi
\expandafter\endgroup \next
```

重新加载检查 (reload check) 并识别宏包 (identify the package)：没有 LaTeX 2_ε 机制，所以这一切非常基本 (pretty basic)。

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname ver@expl3-generic.tex\endcsname\relax \else
  \immediate\write-1
    {%
      expl3 包信息：包已加载。%
    }%
  \expandafter\endinput
\fi \immediate\write-1
  {%
```

```
    Package: expl3
    \ExplFileDate\space
    L3 编程层 (加载程序)%
  }%
\expandafter\edef\csname ver@expl3-generic.tex\endcsname
  {\ExplFileDate\space L3 programming layer}%
```

\l@expl@tidy@tl 保存 @ 的类别代码 (category code)，然后将其设置为“letter”。

```
\expandafter\edef\csname l@expl@tidy@tl\endcsname
  {%
    \catcode64=\the\catcode64\relax
    \let\expandafter\noexpand\csname l@expl@tidy@tl\endcsname
      \noexpand\undefined
  }%
\catcode64=11 %
```

(*End definition for* `\l@expl@tidy@tl`.)

\AtBeginDocument 包代码中有一些 \AtBeginDocument 的用法：最简单的方法是“现在”编写代码。由于
\expl@AtBeginDocument 诸如 miniltx 之类的捆绑包 (bundles) 可能已经定义了 \AtBeginDocument，因此任何现有定义都会保存起来，以便在有效负载 (payload) 之后进行恢复。

```
\let\expl@AtBeginDocument\AtBeginDocument
\def\AtBeginDocument#1{#1}%
\expandafter\def\expandafter\l@expl@tidy@tl\expandafter
  {%
    \l@expl@tidy@tl
    \let\AtBeginDocument\expl@AtBeginDocument
    \let\expl@AtBeginDocument\undefined
  }%
```

(*End definition for* `\AtBeginDocument` *and* `\expl@AtBeginDocument`. *These functions are documented on page* ??.)

加载业务结束：这将保留 \expl3 语法。

```
\input expl3-code.tex %
```

检查引导程序代码 (bootstrap code) 是否中止了加载：如果中止了加载，则在这里静默 (silently) 退出。

```
\begingroup\expandafter\expandafter\expandafter\endgroup
```

```
\expandafter\ifx\csname tex\string _let:D\endcsname\relax
  \expandafter\endinput
\fi
```

__kernel_sys_configuration_load:n 非常基本。

```
\cs_gset_protected:Npn \__kernel_sys_configuration_load:n #1
  {
    \group_begin:
    \cs_set_protected:Npn \ProvidesExplFile
      {
        \char_set_catcode_space:n { `\  }
        \ProvidesExplFileAux
      }
    \cs_set_protected:Npn \ProvidesExplFileAux ##1##2##3##4
      {
        \group_end:
        \iow_log:x { File:~##1~##2~v##3~##4 }
      }
    \tex_input:D #1 .def \scan_stop:
  }
```

(*End definition for* __kernel_sys_configuration_load:n.)

\g__kernel_backend_header_bool 加载动态代码 (dynamic code) 和标准后端 (standard back-end)。

```
\sys_everyjob: \bool_new:N \g__kernel_backend_header_bool \bool_gset_true:N
\g__kernel_backend_header_bool \sys_load_backend:n { }
```

(*End definition for* \g__kernel_backend_header_bool.)

对于通用加载程序 (generic loader)，需要执行一些最后的步骤。转换 `\expl3` 语法并整理少量的临时更改 (temporary changes)。

```
\ExplSyntaxOff \l@expl@tidy@tl
```

⟨/generic⟩

索 引

斜体数字 (italicnumbers) 表示描述相应条目所在的页面 (pages)，带有下划线的数字 (numbersunderlined) 表示定义 (definition)，所有其他数字表示使用的位置 (places)。

符 号

A

B

C

D

E

U

W