LaTeX 的钩子管理*

Frank Mittelbach† 【著】
张泓知 【译】

2023 年 12 月 29 日

目 录

1 介绍 **2**

2 包作者接口 **2**

2.1 LaTeX 2ε 接口 2
2.1.1 声明钩子 2
2.1.2 通用钩子的特殊声明 3
2.1.3 在代码中使用钩子 4
2.1.4 钩子名称和默认标签 7
2.1.5 top-level 标签 . 10
2.1.6 定义挂钩代码之间的关系 10
2.1.7 查询挂钩 12
2.1.8 显示挂钩代码 . . . 12
2.1.9 调试钩子代码 . . . 14
2.2 L3 层的编程 (expl3) 接口 14
2.3 关于钩子代码执行顺序 17
2.4 使用“反转”钩子 . . . 19
2.5 “普通”钩子与“一次性”钩子的区别 20
2.6 包提供的通用钩子 . . . 21
2.7 带参数的钩子 22
2.8 私有的 LaTeX 核心钩子 24
2.9 遗留的 LaTeX 2ε 接口 . 24

3 LaTeX 2ε 命令和由钩子增强的环境 **25**

3.1 通用钩子 25
3.1.1 所有环境的通用钩子 26
3.1.2 命令的通用钩子 . . 27
3.1.3 文件加载操作提供的通用钩子 28
3.2 \begin{document} 提供的钩子 28
3.3 \end{document} 提供的钩子 28
3.4 \shipout 操作提供的钩子 30
3.5 段落提供的钩子 30
3.6 NFSS 命令提供的钩子 . 30
3.7 标记机制提供的钩子 . . 31

索引 **31**

†Phelype Oleinik 做了代码改进以使速度更快以及其它好处。

1 介绍

钩子（Hooks）是命令或环境代码中的处理点，在这些点上可以添加处理代码到现有命令中。不同的包可以对同一命令进行处理，为了确保安全处理，需要将不同包添加的代码块按合适的顺序进行排序。

包通过 `\AddToHook` 添加代码块，并使用默认的包名作为标签对其进行标记。

在 `\begin{document}` 处，所有钩子的代码根据一些规则（由 `\DeclareHookRule` 给出）进行排序，以实现快速执行，避免额外的处理开销。如果后续修改了钩子代码（或更改了规则），将生成新的用于快速处理的版本。

一些钩子已在文档的导言部分使用。如果在此时已经使用了钩子，钩子将被准备（并排序）以便执行。

2 包作者接口

钩子管理系统提供了一组 CamelCase 命令，用于传统的 LaTeX 2_ε 包（以及必要时在文档导言部分使用），同时也提供了用于现代包的 `expl3` 命令，这些现代包使用了 LaTeX 的 L3 编程层。在幕后，访问的是一组单一的数据结构，使得来自两个世界的包可以共存并访问其他包中的钩子。

2.1 LaTeX 2_ε 接口

2.1.1 声明钩子

除了少数例外，钩子必须在使用前声明。这些例外包括命令和环境的通用钩子（在 `\begin` 和 `\end` 执行）以及加载文件时运行的钩子（参见第 3.1节）。

`\NewHook` `\NewHook {⟨hook⟩}`

创建一个新的 ⟨*hook*⟩。如果这个钩子在一个包内声明，建议其名称总是结构化的，形式为：⟨*package-name*⟩/⟨*hook-name*⟩。如果需要，您可以通过添加更多的 `/` 部分来进一步细分名称。如果钩子名称已经存在，将引发错误并且不会创建该钩子。

⟨*hook*⟩ 可以使用点语法指定为当前包的名称。请参见第 2.1.4节。

`\NewReversedHook` `\NewReversedHook {⟨hook⟩}`

类似于 `\NewHook` 声明一个新的 ⟨*hook*⟩。不同之处在于，该钩子的代码块默认按相反顺序排列（最后添加的先执行）。钩子的任何规则都将在默认排序之后应用。详细内容请参见第 2.3 和 2.4 节。

⟨*hook*⟩ 可以使用点语法指定为当前包的名称。请参见第 2.1.4 节。

\NewMirroredHookPair `\NewMirroredHookPair {⟨hook-1⟩} {⟨hook-2⟩}`

是 `\NewHook{⟨hook-1⟩}\NewReversedHook{⟨hook-2⟩}` 的简写。

⟨hook⟩ 可以使用点语法指定为当前包的名称。请参见第 2.1.4 节。

\NewHookWithArguments `\NewHookWithArguments {⟨hook⟩} {⟨number⟩}`

创建一个具有 ⟨number⟩ 个参数的新 ⟨hook⟩,在其他方面与 `\NewHook` 完全相同。第 2.7 节详细解释了带参数的钩子。

⟨hook⟩ 可以使用点语法指定为当前包的名称。请参见第 2.1.4 节。

\NewReversedHookWithArguments `\NewReversedHookWithArguments {⟨hook⟩} {⟨number⟩}`

类似于 `\NewReversedHook`，但创建的钩子的代码带有 ⟨number⟩ 个参数。第 2.7 节详细解释了带参数的钩子。

⟨hook⟩ 可以使用点语法指定为当前包的名称。请参见第 2.1.4 节。

\NewMirroredHookPairWithArguments `\NewMirroredHookPairWithArguments {⟨hook-1⟩} {⟨hook-2⟩} {⟨number⟩}`

是 `\NewHookWithArguments{⟨hook-1⟩}{⟨number⟩}`
`\NewReversedHookWithArguments{⟨hook-2⟩}{⟨number⟩}` 的简写。第 2.7 节详细解释了带参数的钩子。

⟨hook⟩ 可以使用点语法指定为当前包的名称。请参见第 2.1.4 节。

2.1.2 通用钩子的特殊声明

此处的声明通常不应该被使用。它们提供了对主要涉及通用命令钩子的特殊用例的支持。

\DisableGenericHook `\DisableGenericHook {⟨hook⟩}`

在此声明之后[1]，⟨hook⟩ 将不再可用：进一步尝试向其添加代码将导致错误，任何使用，例如 `\UseHook`，都将什么也不做。

这主要用于通用命令钩子（参见 `ltcmdhooks-doc`），因为根据命令的定义，这些通用钩子可能不可用。如果已知此情况，包开发人员可以提前禁用这些钩子。

⟨hook⟩ 可以使用点语法指定为当前包的名称。请参见第 2.1.4 节。

\ActivateGenericHook `\ActivateGenericHook {⟨hook⟩}`

此声明激活了包/类提供的通用钩子（例如，在使用 `\UseHook` 或 `\UseOneTimeHook` 代码中使用的钩子），而无需显式使用 `\NewHook` 进行声明）。此命令撤销了 `\DisableGenericHook` 的效果。如果钩子已经被激活，此命令将不做任何操作。

请参见第 2.6 节，了解何时使用此声明。

[1] 在 2020/06 版本中，此命令称为 `\DisableHook`，但该名称是误导性的，因为它不应用于禁用非通用钩子。

2.1.3 在代码中使用钩子

\UseHook `\UseHook {⟨hook⟩}`

执行存储在 ⟨*hook*⟩ 中的代码。

在 `\begin{document}` 之前，并未设置钩子的快速执行代码，因此在那里使用钩子时，需要显式地首先进行初始化。由于这涉及到赋值，在这些时刻使用钩子并非与在 `\begin{document}` 后完全相同。

无法使用点语法指定 ⟨*hook*⟩。其前面的 . 将被视为文字字符。

\UseHookWithArguments `\UseHookWithArguments {⟨hook⟩} {⟨number⟩} {⟨arg1⟩} … {⟨argn⟩}`

执行存储在 ⟨*hook*⟩ 中的代码，并将 {⟨arg_1⟩} 至 {⟨arg_n⟩} 参数传递给 ⟨*hook*⟩。否则，其行为与 `\UseHook` 完全相同。⟨*number*⟩ 应该是钩子声明的参数数量。如果钩子未声明，此命令将不执行任何操作，并将从输入中删除 ⟨*number*⟩ 个项目。第 2.7 节解释了带参数的钩子。

无法使用点语法指定 ⟨*hook*⟩。其前面的 . 将被视为文字字符。

\UseOneTimeHook `\UseOneTimeHook {⟨hook⟩}`

一些钩子仅在一个地方使用（并且只能在一个地方使用），例如，在 `\begin{document}` 或 `\end{document}` 中的钩子。从那时起，通过已定义的 `\⟨addto-cmd⟩` 命令（例如，`\AddToHook` 或 `\AtBeginDocument` 等）向钩子添加内容将不起作用（就像在钩子代码内部使用这样的命令一样）。因此，习惯上重新定义 `\⟨addto-cmd⟩` 以简单地处理其参数，即本质上使其行为类似于 `\@firstofone`。

`\UseOneTimeHook` 就是这样做的：它记录钩子已被消耗，任何进一步尝试向其添加内容都将导致立即执行要添加的代码。

多次使用 `\UseOneTimeHook` 对同一个 {⟨*hook*⟩} 意味着它只在第一次使用时执行。例如，如果它在可以被多次调用的命令中使用，则该钩子仅在该命令的 *第一次* 调用时执行；这允许其用作“初始化钩子”。

应避免混合使用 `\UseHook` 和 `\UseOneTimeHook` 用于同一个 {⟨*hook*⟩}，但如果这样做了，那么在第一次 `\UseOneTimeHook` 后，两者都不会再执行。

无法使用点语法指定 ⟨*hook*⟩。其前面的 . 将被视为文字字符。详见第 2.1.4 节。

\UseOneTimeHookWithArguments `\UseOneTimeHookWithArguments {⟨hook⟩} {⟨number⟩} {⟨arg$_1$⟩} … {⟨arg$_n$⟩}`

与 `\UseOneTimeHook` 完全相同，但将参数 {⟨arg$_1$⟩} 至 {⟨arg$_n$⟩} 传递给 ⟨hook⟩。⟨number⟩ 应该是钩子声明的参数数量。如果钩子未声明，此命令将不执行任何操作，并将从输入中删除 ⟨number⟩ 个项目。

应注意，一次性钩子使用后，将不再可能使用 `\AddToHookWithArguments` 或类似方法添加内容到该钩子。`\AddToHook` 仍然正常工作。第 2.7 节解释了带参数的钩子。

无法使用点语法指定 ⟨hook⟩。其前面的 . 将被视为文字字符。详见第 2.1.4 节。

\AddToHook `\AddToHook {⟨hook⟩}[⟨label⟩]{⟨code⟩}`

向标记为 ⟨label⟩ 的 ⟨hook⟩ 添加 ⟨code⟩。当不提供可选参数 ⟨label⟩ 时，将使用 ⟨默认标签⟩（参见第 2.1.4 节）。如果 `\AddToHook` 在包/类中使用，则 ⟨默认标签⟩ 为包/类名，否则为 `top-level`（`top-level` 标签处理方式不同：详见第 2.1.5 节）。

如果 ⟨label⟩ 下已存在代码，则新的 ⟨code⟩ 将附加到现有代码中（即使这是一个反向钩子）。如果要替换 ⟨label⟩ 下的现有代码，请先应用 `\RemoveFromHook`。

钩子不必存在即可向其添加代码。但是，如果未声明，则显然添加的 ⟨code⟩ 将永远不会执行。这使得钩子能够在不考虑包装载顺序的情况下工作，并使得包装可以从其他包装中向钩子添加内容，而无需担心它们实际上是否在当前文档中使用。详见第 2.1.7 节。

可以使用点语法指定 ⟨hook⟩ 和 ⟨label⟩。详见第 2.1.4 节。

\AddToHookWithArguments `\AddToHookWithArguments {⟨hook⟩}[⟨label⟩]{⟨code⟩}`

与 `\AddToHook` 完全相同，但 ⟨code⟩ 可以访问通过 `#1`、`#2`、…、`#n`（与钩子声明的参数数量相符）传递给钩子的参数。如果 ⟨code⟩ 中包含不希望被理解为钩子参数的参数符号（`#`），则应将这些符号加倍。例如，使用 `\AddToHook` 可以写成：

```
\AddToHook{myhook}{\def\foo#1{Hello, #1!}}
```

但是要使用 `\AddToHookWithArguments` 实现相同效果，应写成：

```
\AddToHookWithArguments{myhook}{\def\foo##1{Hello, ##1!}}
```

因为在后一种情况中，`#1` 指的是钩子 `myhook` 的第一个参数。第 2.7 节解释了带参数的钩子。

可以使用点语法指定 ⟨hook⟩ 和 ⟨label⟩。详见第 2.1.4 节。

\RemoveFromHook `\RemoveFromHook {⟨hook⟩}[⟨label⟩]`

从 ⟨*hook*⟩ 中删除由 ⟨*label*⟩ 标记的任何代码。当不提供可选参数 ⟨*label*⟩ 时，将使用 ⟨*default label*⟩ （参见第 2.1.4 节）。

如果在 ⟨*hook*⟩ 中不存在 ⟨*label*⟩ 下的代码，或者 ⟨*hook*⟩ 不存在，则在尝试 `\RemoveFromHook` 时发出警告，并忽略该命令。仅当您确切地了解钩子中有哪些标签时，才应使用 `\RemoveFromHook`。通常情况下，这将是当某个包将某些代码添加到钩子中时，然后同一个包稍后删除此代码时。如果您想阻止来自另一个包的代码执行，则应使用 `voids` 规则（参见第 2.1.6 节）。

如果可选的 ⟨*label*⟩ 参数是 `*`，则会删除所有代码块。这相当危险，因为它可能会删除其他包的代码（可能不为人所知）；因此，它不应在包中使用，而只应在文档导言中使用！

可以使用点符号语法指定 ⟨*hook*⟩ 和 ⟨*label*⟩，以表示当前包名称。参见第 2.1.4 节。

与 `\DeclareHookRule` 中两个标签之间的 `voids` 关系相比，这是一种破坏性的操作，因为标记的代码已从钩子数据结构中删除，而关系设置可以通过稍后提供不同的关系来撤消。

此声明在文档主体内的一个有用应用是当您想临时添加代码到钩子中，然后稍后再次删除它时，例如，

```
\AddToHook{env/quote/before}{\small}
\begin{quote}
  A quote set in a smaller typeface
\end{quote}
...
\RemoveFromHook{env/quote/before}
... now back to normal for further quotes
```

请注意，您无法通过以下方式取消设置：

```
\AddToHook{env/quote/before}{}
```

因为这只是“添加”了一个空的代码块到钩子中。添加 `\normalsize` 是可行的，但这意味着钩子中包含了 `\small\normalsize`，这意味着没有充分理由进行两次字体大小更改。

上述操作仅在想要以较小字体排版多个引用时才需要。如果钩子仅需要一次使用，那么 `\AddToHookNext` 更简单，因为它在使用一次后会重置自身。

\AddToHookNext `\AddToHookNext {⟨hook⟩}{⟨code⟩}`

向下一次 ⟨*hook*⟩ 调用中添加 ⟨*code*⟩。该代码在常规钩子代码执行完毕后执行，并且仅执行一次，即在使用后删除。

使用此声明是全局操作，即使声明在组内使用，并且钩子的下一次调用发生在该组结束之后，代码也不会丢失。如果在执行钩子之前多次使用声明，则所有代码将按照声明的顺序执行。[2]

如果此声明与一次性钩子一起使用，则仅当声明在钩子调用之前时才会使用代码。这是因为与 `\AddToHook` 相比，在钩子调用已经发生时，此声明中的代码不会立即执行——换句话说，此代码仅在下一次钩子调用时真正执行（对于一次性钩子，没有这样的“下一次调用”）。这给您一个选择：我的代码应该始终执行，还是仅在一次性钩子使用时执行（如果不可能则不执行）？对于这两种可能性，都存在使用情况。

可以使用相同钩子（或不同钩子）嵌套此声明，例如，

`\AddToHookNext{⟨hook⟩}{⟨code-1⟩\AddToHookNext{⟨hook⟩}{⟨code-2⟩}}`

将在下一次使用 ⟨*hook*⟩ 时执行 ⟨*code-1*⟩，并在那时将 ⟨*code-2*⟩ 放入 ⟨*hook*⟩ 中，以便在下次运行钩子时执行它。

钩子不一定存在才能向其添加代码。这使得钩子可以独立于包加载顺序工作。参见第 2.1.7 节。

可以使用点符号语法指定 ⟨*hook*⟩，以表示当前包名称。参见第 2.1.4 节。

\AddToHookNextWithArguments `\AddToHookNextWithArguments {⟨hook⟩}{⟨code⟩}`

功能与 `\AddToHookNext` 完全相同，但 ⟨*code*⟩ 可包含对 ⟨*hook*⟩ 参数的引用，正如上面对 `\AddToHookWithArguments` 的描述。第 2.7 节解释了带参数的钩子。

可以使用点符号语法指定 ⟨*hook*⟩，以表示当前包名称。参见第 2.1.4 节。

\ClearHookNext `\ClearHookNext{⟨hook⟩}`

通常，仅当您准确知道它将应用在何处以及为何需要一些额外代码时，才会使用 `\AddToHookNext`。然而，在某些情况下，需要取消这种声明，例如，使用 `\DiscardShipoutBox` 丢弃页面时（但甚至在这种情况下也不总是如此），在这种情况下可以使用 `\ClearHookNext`。

2.1.4 钩子名称和默认标签

在包或类中最好使用 `\AddToHook`，不指定 ⟨*label*⟩，因为这样可以自动使用包或类名称，如果需要规则，则会很有帮助，并避免了输入错误的 ⟨*label*⟩。

[2] 没有重新排序此类代码块的机制（或删除它们）。

仅在非常特定的情况下才需要使用显式的 ⟨*label*⟩，例如，如果要将多个代码块添加到单个钩子中，并希望将它们放置在钩子的不同部分（通过提供一些规则）。

另一个情况是当您开发具有多个子包的大型包时。在这种情况下，您可能希望在整个子包中使用相同的 ⟨*label*⟩，以避免在内部重新组织代码时标签发生变化。

除了 `\UseHook`、`\UseOneTimeHook` 和 `\IfHookEmptyTF`（及其 expl3 接口 `\hook_use:n`、`\hook_use_once:n` 和 `\hook_if_empty:nTF`）之外，所有 ⟨*hook*⟩ 和 ⟨*label*⟩ 参数的处理方式相同：首先，对参数周围的空格进行修剪，然后完全展开，直到只剩下字符记号。如果 ⟨*hook*⟩ 或 ⟨*label*⟩ 的完全展开包含一个不可展开的非字符记号，将引发低级 TEX 错误（即，使用 TEX 的 `\csname…\endcsname` 展开 ⟨*hook*⟩，因此 ⟨*hook*⟩ 和 ⟨*label*⟩ 参数中允许使用 Unicode 字符）。`\UseHook`、`\UseOneTimeHook` 和 `\IfHookEmptyTF` 的参数处理方式基本相同，只是不会修剪参数周围的空格，以获得更好的性能。

虽然不是强制要求，但强烈建议由包定义的钩子和用于向其他钩子添加代码的 ⟨*label*⟩，包含包名称，以便轻松识别代码块的来源并防止冲突。这应该是标准做法，因此此钩子管理代码提供了一个快捷方式，用于在 ⟨*hook*⟩ 名称和 ⟨*label*⟩ 中引用当前包。如果 ⟨*hook*⟩ 名称或 ⟨*label*⟩ 仅由一个单独的点（`.`）或以点开头，后跟斜杠（`./`），则该点表示 ⟨*default label*⟩（通常是当前包或类名称——参见 `\SetDefaultHookLabel`）。“`.`” 或 “`./`” 在 ⟨*hook*⟩ 或 ⟨*label*⟩ 的任何其他位置都会被按原样处理，不会被替换。

例如，在名为 `mypackage.sty` 的包中，默认标签是 `mypackage`，因此以下说明：

```
\NewHook   {./hook}
\AddToHook {./hook}[.]{code}     % Same as \AddToHook{./hook}{code}
\AddToHook {./hook}[./sub]{code}
\DeclareHookRule{begindocument}{.}{before}{babel}
\AddToHook {file/foo.tex/after}{code}
```

等价于：

```
\NewHook   {mypackage/hook}
\AddToHook {mypackage/hook}[mypackage]{code}
\AddToHook {mypackage/hook}[mypackage/sub]{code}
\DeclareHookRule{begindocument}{mypackage}{before}{babel}
\AddToHook {file/foo.tex/after}{code}                    % unchanged
```

⟨*default label*⟩ 在包加载时自动设置为当前包或类的名称。如果挂钩命令在包外使用，或者当前文件没有使用 `\usepackage` 或 `\documentclass` 加载，那么将使用 `top-level` 作为 ⟨*default label*⟩。这可能会有例外情况——参见 `\PushDefaultHookLabel`。

此语法适用于所有 ⟨*label*⟩ 参数和大多数 ⟨*hook*⟩ 参数，无论是在 LATEX 2_ε 接口中，还是在第 2.2 节描述的 LATEX3 接口中。

重要：
点语法在 \UseHook 和一些通常在代码中使用的其他命令中**不可用**！

注意，但要注意，当执行挂钩命令时，. 被 ⟨*default label*⟩ 替换，因此在包结束后某种程度上执行的操作，如果使用了点语法，将会有错误的 ⟨*default label*⟩。出于这个原因，这种语法在 \UseHook（和 \hook_use:n）中不可用，因为大多数情况下，挂钩在定义它的包文件之外使用。这种语法也不适用于挂钩条件语句 \IfHookEmptyTF（和 \hook_if_empty:nTF），因为这些条件语句在挂钩管理代码的一些性能关键部分中使用，并且通常用于引用其他包的挂钩，因此点语法并不太合适。

在某些情况下，例如在大型包中，可能希望将代码分离为逻辑部分，但仍然使用主包名称作为 ⟨*label*⟩，那么可以使用 \PushDefaultHookLabel{...} … \PopDefaultHookLabel 或 \SetDefaultHookLabel{...} 设置 ⟨*default label*⟩。

\PushDefaultHookLabel
\PopDefaultHookLabel

```
\PushDefaultHookLabel {⟨default label⟩}
  ⟨code⟩
\PopDefaultHookLabel
```

\PushDefaultHookLabel 设置当前 ⟨*default label*⟩ 以在 ⟨*label*⟩ 参数或替换前导的“.”时使用。\PopDefaultHookLabel 将 ⟨*default label*⟩ 恢复为其先前的值。

在包或类中，⟨*default label*⟩ 等于包或类名称，除非显式更改。在其他任何地方，⟨*default label*⟩ 是 top-level（参见第 2.1.5 节），除非显式更改。

\PushDefaultHookLabel 的效果持续到下一个 \PopDefaultHookLabel。\usepackage（以及 \RequirePackage 和 \documentclass）内部使用

```
\PushDefaultHookLabel{⟨package name⟩}
  ⟨package code⟩
\PopDefaultHookLabel
```

来设置包或类文件的 ⟨*default label*⟩。在 ⟨*package code*⟩ 中，也可以使用 \SetDefaultHookLabel 更改 ⟨*default label*⟩。\input 和其他从 LaTeX 核心中输入文件的命令不使用 \PushDefaultHookLabel，因此由这些命令加载的文件中的代码不会获得专用的 ⟨*label*⟩！（也就是说，⟨*default label*⟩ 是加载文件时的当前活动标签。）

提供自己类似包的接口的包（例如 TikZ 的 \usetikzlibrary）可以使用 \PushDefaultHookLabel 和 \PopDefaultHookLabel 设置专用标签，并在这些上下文中模拟类似 \usepackage 的挂钩行为。

top-level 标签处理方式不同，并保留给用户文档，因此不允许将 ⟨*default label*⟩ 更改为 top-level。

\SetDefaultHookLabel \SetDefaultHookLabel {⟨*default label*⟩}

\SetDefaultHookLabel 与 \PushDefaultHookLabel 类似，将当前 ⟨*default label*⟩ 设置为在 ⟨*label*⟩ 参数中使用,或替换前导的“.”时使用。其效果持续到标签再次更改或到下一个 \PopDefaultHookLabel。\PushDefaultHookLabel 和 \SetDefaultHookLabel 的区别在于后者不保存当前 ⟨*default label*⟩。

当一个大型包由几个较小的包组成，但所有这些包都应具有相同的 ⟨*label*⟩ 时，\SetDefaultHookLabel 可以在每个包文件的开头使用以设置正确的标签。

在主文档中不允许使用 \SetDefaultHookLabel,其中 ⟨*default label*⟩ 是 top-level，且没有 \PopDefaultHookLabel 来结束其效果。同样不允许将 ⟨*default label*⟩ 更改为 top-level。

2.1.5 top-level 标签

为从主文档中添加的代码分配的 top-level 标签与其他标签不同。添加到导言区挂钩（通常是 \AtBeginDocument）的代码几乎总是用于更改包定义的内容，因此应该放在挂钩的最末端。

因此，添加在 top-level 的代码始终在挂钩的末尾执行，无论它在何处声明。如果挂钩被反转（参见 \NewReversedHook），则 top-level 代码块将在最开始执行。

关于 top-level 的规则不起作用：如果用户想为代码块设置特定规则，应该为该代码块使用不同的标签，并为该标签提供规则。

top-level 标签专属于用户，因此试图从包中使用该标签添加代码将导致错误。

2.1.6 定义挂钩代码之间的关系

默认假设是由不同包添加到挂钩的代码是独立的，并且它们执行的顺序是不相关的。虽然在许多情况下这是正确的，但在其他情况下显然是错误的。

在引入挂钩管理系统之前，包必须采取复杂的预防措施来确定其他包是否也被加载（在前面或后面），并找到一些方法相应地更改其行为。此外，通常用户需要负责以正确的顺序加载包，以使添加到挂钩的代码以正确的顺序添加，有些情况即使更改加载顺序也无法解决冲突。

使用新的挂钩管理系统，现在可以定义（即关系）不同包添加的代码块之间的规则，并明确描述它们应该被处理的顺序。

\DeclareHookRule `\DeclareHookRule {⟨hook⟩}{⟨label1⟩}{⟨relation⟩}{⟨label2⟩}`

为给定的 ⟨*hook*⟩ 定义 ⟨*label1*⟩ 和 ⟨*label2*⟩ 之间的关系。如果 ⟨*hook*⟩ 是 `??`，则为使用这两个标签的所有挂钩定义了默认关系，即具有标记为 ⟨*label1*⟩ 和 ⟨*label2*⟩ 的代码块的挂钩。对于特定挂钩的规则优先于使用 `??` 作为 ⟨*hook*⟩ 的默认规则。

目前，支持的关系有以下几种：

`before` 或 `<` ⟨*label1*⟩ 的代码出现在 ⟨*label2*⟩ 的代码之前。

`after` 或 `>` ⟨*label1*⟩ 的代码出现在 ⟨*label2*⟩ 的代码之后。

`incompatible-warning` 只能出现 ⟨*label1*⟩ 或 ⟨*label2*⟩ 的代码（表示两个包或其部分不兼容）。如果两个标签同时出现在同一个挂钩中，会发出警告。

`incompatible-error` 类似于 `incompatible-warning`，但是不会发出警告，而是引发 LaTeX 错误，并在冲突解决前从该挂钩中删除两个标签的代码。

`voids` ⟨*label1*⟩ 的代码覆盖了 ⟨*label2*⟩ 的代码。更确切地说，在该挂钩中会删除 ⟨*label2*⟩ 的代码。例如，如果一个包在功能上是另一个包的超集，因此希望撤消某个挂钩中的代码并用自己的版本替换，则可以使用此选项。

`unrelated` ⟨*label1*⟩ 和 ⟨*label2*⟩ 的代码顺序无关紧要。此规则用于撤销之前指定的不正确规则。

对于给定挂钩的两个标签之间只能存在一个关系，即后续的 `\DeclareHookRule` 会覆盖任何先前的声明。

可以使用点语法指定 ⟨*hook*⟩ 和 ⟨*label*⟩，以表示当前包名称。请参阅第 2.1.4 节。

\ClearHookRule `\ClearHookRule{⟨hook⟩}{⟨label1⟩}{⟨label2⟩}`

这是一种简化的写法，表示给定的 ⟨*hook*⟩ 中 ⟨*label1*⟩ 和 ⟨*label2*⟩ 之间无关联。

\DeclareDefaultHookRule `\DeclareDefaultHookRule{⟨label1⟩}{⟨relation⟩}{⟨label2⟩}`

这为所有挂钩设置了 ⟨*label1*⟩ 和 ⟨*label2*⟩ 之间的关系，除非特定挂钩被另一个规则覆盖。适用于一个包与另一个包有特定关系的情况，例如，是 `incompatible` 或总是需要特殊顺序 `before` 或 `after`。（技术上，这只是使用 `\DeclareHookRule` 并将 `??` 作为挂钩名称的简写。）

声明默认规则仅在文档导言部分支持。[3]

可以使用点语法指定 ⟨*label*⟩，以表示当前包名称。请参阅第 2.1.4 节。

[3] 尝试这样做，例如通过使用 `??` 的 `\DeclareHookRule`，会产生不良的副作用，并且不受支持（尽管出于性能原因未显式捕获）。

2.1.7 查询挂钩

简单的数据类型，比如记号列表，有三种可能的状态：

- 存在但为空；
- 存在且非空；以及
- 不存在（此时不存在空的概念）。

挂钩稍微复杂一些：一个挂钩可以存在也可以不存在，独立于此，它可以是空的也可以是非空的。这意味着即使一个挂钩不存在，它也可能是非空的，而且它也可以被禁用。

这种看似奇怪的状态可能发生在这样的情况下，例如，包 A 定义了挂钩 `A/foo`，而包 B 向该挂钩添加了一些代码。然而，文档可能在加载包 A 之前加载了包 B，或者根本没有加载包 A。在这两种情况下，一些代码被添加到了挂钩 `A/foo` 中，但该挂钩尚未定义，因此该挂钩被认为是非空的，但实际上它并不存在。因此，查询挂钩的存在性并不意味着它的空值，反之亦然。

由于代码或规则可以添加到一个挂钩，即使它还不存在，所以查询其存在性没有实际用途（与其他变量不同，其他变量只有在已经声明的情况下才能更新）。因此，只有对空值的测试具有公共接口。

当没有代码添加到挂钩的永久代码池或其“next”记号列表时，挂钩被认为空。挂钩不需要被声明为具有代码池。当使用 `\NewHook` 或其变体声明挂钩时，该挂钩被认为存在。当向其添加代码时，通用挂钩如 `file` 和 `env` 会自动声明。

`\IfHookEmptyTF` ⋆ `\IfHookEmptyTF {`⟨*hook*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

检测 ⟨*hook*⟩ 是否为空（即没有使用 `\AddToHook` 或 `\AddToHookNext` 添加代码，或者通过 `\RemoveFromHook` 将代码移除），根据结果分别执行 ⟨*true code*⟩ 或 ⟨*false code*⟩。

无法使用点语法指定 ⟨*hook*⟩。前导的 . 会被视为字面量。

2.1.8 显示挂钩代码

如果需要使用挂钩规则调整挂钩中的代码执行顺序，了解挂钩相关信息、当前顺序和现有规则将会很有帮助。

\ShowHook \ShowHook {⟨*hook*⟩}
\LogHook \LogHook {⟨*hook*⟩}

显示关于 ⟨*hook*⟩ 的信息，例如：

- 挂钩中添加的代码块（及其标签），
- 任何用于排序的设置规则，
- 计算出的代码块执行顺序，
- 仅在下一次调用时执行的任何代码。

\LogHook 将信息打印到 .log 文件中，而 \ShowHook 将其打印到终端/命令窗口，并在 \errorstopmode 下启动 TEX 的提示，等待用户操作。

可以使用点语法指定 ⟨*hook*⟩，以表示当前包名称。请参阅第 2.1.4 节。

假设有一个名为 example-hook 的钩子，其 \ShowHook{example-hook} 的输出如下：

```
1   -> The hook 'example-hook':
2   > Code chunks:
3   >     foo -> [code from package 'foo']
4   >     bar -> [from package 'bar']
5   >     baz -> [package 'baz' is here]
6   > Document-level (top-level) code (executed last):
7   >     -> [code from 'top-level']
8   > Extra code for next invocation:
9   >     -> [one-time code]
10  > Rules:
11  >     foo|baz with relation >
12  >     baz|bar with default relation <
13  > Execution order (after applying rules):
14  >     baz, foo, bar.
```

在上面的列表中，第 3 到第 5 行展示了添加到钩子的三个代码片段及其相应的标签，格式如下：

⟨*label*⟩ -> ⟨*code*⟩

第 7 行展示了用户在主文档中添加的代码片段（标记为 top-level），格式如下：(labeled top-level) in the format

```
Document-level (top-level) code (executed ⟨first|last⟩):
  -> ⟨top-level code⟩
```

这段代码将是钩子执行的第一个或最后一个代码（如果钩子是正常的，则为 `last`，如果是反向的，则为 `first`）。这个代码块不受规则影响，也不参与排序。

第 9 行展示了下一次钩子执行时的代码片段格式，如下：

```
-> ⟨next-code⟩
```

这段代码将在下一次 `\UseHook{example-hook}` 时使用并消失，与之前提到的代码片段相反，这些代码片段只能通过 `\RemoveFromHook{⟨label⟩}[example-hook]` 从钩子中移除。

第 11 和第 12 行展示了影响该钩子的声明规则的格式，如下：

```
⟨label-1⟩|⟨label-2⟩ with ⟨default?⟩ relation ⟨relation⟩
```

这意味着 ⟨*relation*⟩ 应用于 ⟨*label-1*⟩ 和 ⟨*label-2*⟩，按照 `\DeclareHookRule` 中的详细说明顺序执行。如果关系是 `default`，则意味着此规则适用于所有钩子中的 ⟨*label-1*⟩ 和 ⟨*label-2*⟩（除非被非默认关系覆盖）。

最后，第 14 行按顺序列出了排序后钩子中的标签；即，在使用钩子时它们将被执行的顺序。

2.1.9 调试钩子代码

`\DebugHooksOn` `\DebugHooksOn`
`\DebugHooksOff`

打开或关闭钩子代码的调试。这会显示对钩子数据结构的大部分更改。输出相当粗糙，不适合正常使用。

2.2 L3 层的编程 (expl3) 接口

这是关于与 `expl3` 写的包一起使用的 LaTeX3 编程接口的快速摘要。与 LaTeX 2_ε 接口不同，它们始终仅使用必需的参数，例如，您总是必须为代码片段指定 ⟨*label*⟩。因此，我们建议即使在 `expl3` 包中也使用前面讨论过的声明，但选择权在您手中。

`\hook_new:n` `\hook_new:n {⟨hook⟩}`
`\hook_new_reversed:n` `\hook_new_reversed:n {⟨hook⟩}`
`\hook_new_pair:nn` `\hook_new_pair:nn {⟨hook-1⟩} {⟨hook-2⟩}`

创建一个具有正常或反向代码顺序的新 ⟨*hook*⟩。`\hook_new_pair:nn` 创建了一对此类钩子，其中 `{⟨hook-2⟩}` 是一个反向钩子。如果钩子名称已经被使用，将引发错误并且不会创建该钩子。

可以使用点号语法来指定 ⟨*hook*⟩，表示当前包的名称。参见第 2.1.4 节。

\hook_new_with_args:nn `\hook_new_with_args:nn {⟨hook⟩} {⟨number⟩}`
\hook_new_reversed_with_args:nn `\hook_new_reversed_with_args:nn {⟨hook⟩} {⟨number⟩}`
\hook_new_pair_with_args:nnn `\hook_new_pair_with_args:nnn {⟨hook-1⟩} {⟨hook-2⟩} {⟨number⟩}`

创建一个具有正常或反向代码顺序的新 ⟨*hook*⟩，在使用时从输入流中获取 ⟨*number*⟩ 个参数。`\hook_new_pair_with_args:nn` 创建了一对此类钩子，其中 {⟨*hook-2*⟩} 是一个反向钩子。如果钩子名称已经被使用，将引发错误并且不会创建该钩子。

可以使用点号语法来指定 ⟨*hook*⟩，表示当前包的名称。参见第 2.1.4 节。

\hook_disable_generic:n `\hook_disable_generic:n {⟨hook⟩}`

将 {⟨*hook*⟩} 标记为已禁用。任何进一步尝试向其添加代码或声明都将导致错误，并且任何对 `\hook_use:n` 的调用都将不起作用。

此声明旨在用于通用钩子，如果它们接收到代码，则已知它们无法正常工作（参见 `ltcmdhooks-doc`）。

可以使用点号语法来指定 ⟨*hook*⟩，表示当前包的名称。参见第 2.1.4 节。

\hook_activate_generic:n `\hook_activate_generic:n {⟨hook⟩}`

这类似于 `\hook_new:n`，但如果钩子之前使用 `\hook_new:n` 声明过，则不会执行任何操作。此声明应仅在特殊情况下使用，例如，当来自另一个包的命令需要更改，而不清楚是否已经先前显式声明了通用的 `cmd` 钩子（用于该命令）时。

通常情况下，应该使用 `\hook_new:n` 而不是这个声明。

\hook_use:n `\hook_use:n {⟨hook⟩}`
\hook_use:nnw `\hook_use:nnw {⟨hook⟩} {⟨number⟩} {⟨arg1⟩} … {⟨argn⟩}`

执行 {⟨*hook*⟩} 代码，然后执行（如果设置了）下一次调用的代码，随后清空该下一次调用的代码。对于使用参数声明的钩子，应使用 `\hook_use:nnw`，并且后面应跟着与声明的参数数量相同的大括号组。⟨*number*⟩ 应该是钩子声明的参数数量。如果钩子未声明，则此命令不起作用，并且将从输入中移除 ⟨*number*⟩ 个项目。

⟨*hook*⟩ 不能 使用点号语法指定。开头的 . 将被视为字面量处理。

\hook_use_once:n `\hook_use_once:n {⟨hook⟩}`
\hook_use_once:nnw `\hook_use_once:nnw {⟨hook⟩} {⟨number⟩} {⟨arg1⟩} … {⟨argn⟩}`

改变 {⟨*hook*⟩} 的状态，从现在开始，任何添加到钩子代码的操作都会立即执行。然后执行已设置的任何 {⟨*hook*⟩} 代码。对于使用参数声明的钩子，应使用 `\hook_use_once:nnw`，并且后面应跟着与声明的参数数量相同的大括号组。⟨*number*⟩ 应该是钩子声明的参数数量。如果钩子未声明，则此命令不起作用，并且将从输入中移除 ⟨*number*⟩ 个项目。

⟨*hook*⟩ 不能 使用点号语法指定。开头的 . 将被视为字面量处理。

\hook_gput_code:nnn `\hook_gput_code:nnn {⟨hook⟩} {⟨label⟩} {⟨code⟩}`
\hook_gput_code_with_args:nnn `\hook_gput_code_with_args:nnn {⟨hook⟩} {⟨label⟩} {⟨code⟩}`

将一段 ⟨*code*⟩ 添加到标记为 ⟨*label*⟩ 的 ⟨*hook*⟩ 中。如果标签已经存在，则将 ⟨*code*⟩ 追加到已有的代码后面。

如果使用了 `\hook_gput_code_with_args:nnn`，那么 ⟨*code*⟩ 可以访问传递给 `\hook_use:nnw`（或 `\hook_use_once:nnw`）的参数，使用 `#1`、`#2`、…、`#n`（最多为钩子声明的参数数量）。在这种情况下，如果要将实际参数标记添加到代码中，应该使用两个相同的参数标记。

如果要向外部的 ⟨*hook*⟩（例如内核或其他包）添加代码，那么约定是使用包名称作为 ⟨*label*⟩，而不是某个内部模块名称或其他任意字符串。

可以使用点号语法来指定 ⟨*hook*⟩ 和 ⟨*label*⟩，表示当前包的名称。参见第 2.1.4 节。

\hook_gput_next_code:nn `\hook_gput_next_code:nn {⟨hook⟩} {⟨code⟩}`
\hook_gput_next_code_with_args:nn

添加一段 ⟨*code*⟩，仅在下一次 ⟨*hook*⟩ 调用中使用。使用后即消失。

如果使用了 `\hook_gput_next_code_with_args:nn`，那么 ⟨*code*⟩ 可以访问传递给 `\hook_use:nnw`（或 `\hook_use_once:nnw`）的参数，使用 `#1`、`#2`、…、`#n`（最多为钩子声明的参数数量）。在这种情况下，如果要将实际参数标记添加到代码中，应该使用两个相同的参数标记。

这比 `\hook_gput_code:nnn` 更简单，代码将按照声明的顺序简单地附加到钩子末尾，即，在所有标准代码执行完毕后。因此，如果需要撤销标准操作，必须将其作为 ⟨*code*⟩ 的一部分处理。

可以使用点号语法来指定 ⟨*hook*⟩，表示当前包的名称。参见第 2.1.4 节。

\hook_gclear_next_code:n `\hook_gclear_next_code:n {⟨hook⟩}`

撤销任何之前的 `\hook_gput_next_code:nn`。

\hook_gremove_code:nn `\hook_gremove_code:nn {⟨hook⟩} {⟨label⟩}`

移除标记为 ⟨*label*⟩ 的 ⟨*hook*⟩ 中的任何代码。

如果在 ⟨*hook*⟩ 中没有 ⟨*label*⟩ 下的代码，或者 ⟨*hook*⟩ 不存在，尝试使用 `\hook_gremove_code:nn` 时将发出警告，并且命令将被忽略。

如果第二个参数是 `*`，则会移除所有代码块。这相当危险，因为会删除其他包中的代码，可能会影响到你不清楚的代码，请在使用之前三思！

可以使用点号语法来指定 ⟨*hook*⟩ 和 ⟨*label*⟩，表示当前包的名称。参见第 2.1.4 节。

\hook_gset_rule:nnnn `\hook_gset_rule:nnnn {⟨hook⟩} {⟨label1⟩} {⟨relation⟩} {⟨label2⟩}`

在 ⟨*hook*⟩ 中使用 ⟨*label1*⟩ 和 ⟨*label2*⟩ 进行关联。查看 `\DeclareHookRule` 获取允许的 ⟨*relation*⟩。如果 ⟨*hook*⟩ 是 `??`，则指定默认规则。

可以使用点号语法来指定 ⟨*hook*⟩ 和 ⟨*label*⟩，表示当前包的名称。参见第 2.1.4 节。点号语法在两个 ⟨*label*⟩ 参数中都进行解析，但通常只在其中一个参数中使用才有意义。

\hook_if_empty_p:n ⋆ `\hook_if_empty:nTF {⟨hook⟩} {⟨true code⟩} {⟨false code⟩}`

\hook_if_empty:nTF ⋆ 检测 ⟨*hook*⟩ 是否为空（即，未使用 `\AddToHook` 或 `\AddToHookNext` 添加代码），并根据结果分别执行 ⟨*true code*⟩ 或 ⟨*false code*⟩。

⟨*hook*⟩ 不能 使用点号语法指定。开头的 . 将被视为字面量处理。

\hook_show:n `\hook_show:n {⟨hook⟩}`

\hook_log:n `\hook_log:n {⟨hook⟩}`

显示关于 ⟨*hook*⟩ 的信息，例如

- 添加到其中的代码块（及其标签），
- 设定的任何用于排序的规则，
- 计算出的代码块执行顺序，
- 仅在下一次调用时执行的任何代码。

`\hook_log:n` 将信息打印到 `.log` 文件，而 `\hook_show:n` 将其打印到终端/命令窗口，并启动 TEX 的提示符（仅在 `\errorstopmode`）等待用户操作。

可以使用点号语法来指定 ⟨*hook*⟩，表示当前包的名称。参见第 2.1.4 节。

\hook_debug_on: `\hook_debug_on:`

\hook_debug_off: 打开或关闭钩子代码的调试。这会显示钩子数据的变化。

2.3 关于钩子代码执行顺序

如果在不设置特殊规则的情况下，⟨*hook*⟩ 下不同标签的代码块被视为独立的，这意味着你不能对执行顺序做出假设！

假设你有以下声明：

```
\NewHook{myhook}
\AddToHook{myhook}[packageA]{\typeout{A}}
\AddToHook{myhook}[packageB]{\typeout{B}}
\AddToHook{myhook}[packageC]{\typeout{C}}
```

使用 \UseHook 执行钩子将按顺序产生类型输出 A B C。换句话说，执行顺序计算为 packageA、packageB、packageC，可以使用 \ShowHook{myhook} 进行验证：

```
-> The hook 'myhook':
> Code chunks:
>     packageA -> \typeout {A}
>     packageB -> \typeout {B}
>     packageC -> \typeout {C}
> Document-level (top-level) code (executed last):
>     ---
> Extra code for next invocation:
>     ---
> Rules:
>     ---
> Execution order:
>     packageA, packageB, packageC.
```

原因在于代码块被内部保存在属性列表中，属性列表的初始顺序是添加键-值对的顺序。但是，这仅在除添加之外没有其他操作时才成立！

举个例子，假设你想替换 packageA 的代码块，比如说，

```
\RemoveFromHook{myhook}[packageA]
\AddToHook{myhook}[packageA]{\typeout{A alt}}
```

那么你的顺序变成了 packageB、packageC、packageA，因为标签从属性列表中移除，然后重新添加（放在末尾）。

虽然这可能不太令人惊讶，但如果添加了多余的规则，例如，如果指定了

```
\DeclareHookRule{myhook}{packageA}{before}{packageB}
```

而不是之前我们得到的那些行

```
-> The hook 'myhook':
> Code chunks:
>     packageA -> \typeout {A}
>     packageB -> \typeout {B}
>     packageC -> \typeout {C}
> Document-level (top-level) code (executed last):
>     ---
> Extra code for next invocation:
```

```
>     ---
> Rules:
>     packageB|packageA with relation >
> Execution order (after applying rules):
>     packageA, packageC, packageB.
```

当你看到代码块时，仍然是相同的顺序，但是在标签 packageB 和 packageC 的执行顺序已经交换了。原因是，根据规则，有两种满足条件的顺序，而排序算法恰好选择了与没有规则的情况不同的顺序（在没有规则的情况下，算法根本不会运行，因为没有需要解决的内容）。顺便说一下，如果我们改为指定多余的规则

```
\DeclareHookRule{myhook}{packageB}{before}{packageC}
```

执行顺序就不会改变了。

总结：除非存在部分或完全定义顺序的规则（你可以依赖它们被满足），否则无法依赖执行顺序。

2.4 使用“反转”钩子

也许您想知道为什么可以用 \NewReversedHook 声明一个“反转”钩子以及它到底是做什么的。

简而言之：一个没有任何规则的反转钩子的执行顺序与使用 \NewHook 声明的钩子顺序完全相反。

如果您有一对期望添加涉及分组的代码的钩子，比如在第一个钩子中开始一个环境，在第二个钩子中关闭该环境，这将非常有帮助。举个有些牵强的例子[4]，假设有一个包添加了以下内容：

```
\AddToHook{env/quote/before}[package-1]{\begin{itshape}}
\AddToHook{env/quote/after} [package-1]{\end{itshape}}
```

结果是，所有引用将呈现为斜体。现在再假设另一个 package-too 也使引用变为蓝色，因此添加了以下内容：

```
\usepackage{color}
\AddToHook{env/quote/before}[package-too]{\begin{color}{blue}}
\AddToHook{env/quote/after} [package-too]{\end{color}}
```

现在，如果 env/quote/after 钩子是一个普通的钩子，那么在两个钩子中我们将得到相同的执行顺序，即：

[4] 有更简单的方法实现相同的效果。

```
package-1, package-too
```

（或相反）结果将是：

```
\begin{itshape}\begin{color}{blue} ...
\end{itshape}\end{color}
```

并且会出现一个错误消息，指出 `\begin{color}` 被 `\end{itshape}` 结束了。如果将 `env/quote/after` 声明为反转钩子，执行顺序就会反转，因此所有环境都以正确的顺序关闭，`\ShowHook` 将给出以下输出：

```
-> The hook 'env/quote/after':
> Code chunks:
>     package-1 -> \end {itshape}
>     package-too -> \end {color}
> Document-level (top-level) code (executed first):
>     ---
> Extra code for next invocation:
>     ---
> Rules:
>     ---
> Execution order (after reversal):
>     package-too, package-1.
```

执行顺序的反转发生在应用任何规则之前，因此如果您更改顺序，则可能必须在两个钩子中都进行更改，而不仅仅是一个，但这取决于用例。

2.5 “普通”钩子与“一次性”钩子的区别

在执行钩子时，开发人员可以选择使用 `\UseHook` 或 `\UseOneTimeHook`（或它们的 expl3 等效命令 `\hook_use:n` 和 `\hook_use_once:n`）。这个选择影响了在钩子第一次执行后如何处理 `\AddToHook`。

对于普通钩子，通过 `\AddToHook` 添加代码意味着代码块被添加到钩子数据结构中，然后每次调用 `\UseHook` 时都会使用它。

对于一次性钩子，处理方式略有不同：在调用 `\UseOneTimeHook` 后，任何进一步尝试通过 `\AddToHook` 向钩子添加代码的操作都将立即执行 ⟨*code*⟩。

这有一些需要注意的后果：

- 如果在钩子执行后向普通钩子添加 ⟨*code*⟩，并且由于某种原因它再也不会执行，则新的 ⟨*code*⟩ 将永远不会被执行。

- 相比之下，如果这种情况发生在一次性钩子上，则 ⟨*code*⟩ 会立即执行。

具体来说，这意味着类似以下结构的构建：

```
\AddToHook{myhook}
          { ⟨code-1⟩ \AddToHook{myhook}{⟨code-2⟩} ⟨code-3⟩ }
```

对于一次性钩子来说是有效的[5]（三个代码块依次执行），但对于普通钩子来说则意义不大，因为对于普通钩子，第一次执行 `\UseHook{myhook}` 时将会：

- 执行 ⟨*code-1*⟩，
- 然后执行 `\AddToHook{myhook}{code-2}`，将代码块 ⟨*code-2*⟩ 添加到下一次调用时使用的钩子中，
- 最后执行 ⟨*code-3*⟩。

第二次调用 `\UseHook` 时，它将执行上述操作，并且额外执行 ⟨*code-2*⟩，因为此时它已被作为代码块添加到钩子中。因此，每次使用钩子时都会添加另一个副本的 ⟨*code-2*⟩，所以该代码块将被执行 $\langle \#\ of\ invocations \rangle - 1$ 次。

2.6 包提供的通用钩子

钩子管理系统还实现了一类称为“通用钩子”的钩子。通常，钩子在可以在代码中使用之前必须显式声明。这确保了不同的包不会为不相关的目的使用相同的钩子名称——这会导致绝对混乱。然而，有一些“标准”钩子，对于它们事先声明是不合理的，例如，每个命令（理论上）都有一个关联的 `before` 和 `after` 钩子。在这种情况下，即对于命令、环境或文件钩子，可以通过 `\AddToHook` 简单地向其中添加代码来使用它们。对于更专门的通用钩子，例如 babel 提供的那些，您需要使用下面解释的 `\ActivateGenericHook` 进行额外的启用。

LaTeX 提供的通用钩子包括 `cmd`、`env`、`file`、`include`、`package` 和 `class`，所有这些都可以直接使用：您只需使用 `\AddToHook` 来添加代码，但不必在您的代码中添加 `\UseHook` 或 `\UseOneTimeHook`，因为这已经为您完成了（或者在 `cmd` 钩子的情况下，在必要时会在 `\begin{document}` 处对命令代码进行修补）。

但是，如果您想在自己的代码中提供进一步的通用钩子，情况稍有不同。为此，您应该使用 `\UseHook` 或 `\UseOneTimeHook`，但是*不要使用 `\NewHook` 声明钩子*。如前所述，对未声明的钩子名称调用 `\UseHook` 不起任何作用。因此，作为额外的设置步骤，您需要显式激活您的通用钩子。请注意，以这种方式生成的通用钩子始终是普通钩子。

[5]这有时会用于 `\AtBeginDocument`，这就是为什么它被支持的原因。

对于真正的通用钩子，在钩子名称中包含可变部分的提前激活将是困难或不可能的，因为您通常不知道真实文档中可能出现的可变部分的类型。

例如，babel 提供了诸如 babel/⟨*language*⟩/afterextras 的钩子。然而，babel 中的语言支持通常是通过外部语言包完成的。因此，在核心 babel 代码中为所有语言执行激活并不可行。相反，需要由每个语言包执行（或者由希望使用特定钩子的用户执行）。

由于这些钩子没有使用 \NewHook 声明，因此它们的名称应谨慎选择，以确保它们（可能）是唯一的。最佳做法是包括包或命令名称，就像上面 babel 的示例中所做的那样。

通过这种方式定义的通用钩子始终是普通钩子（即，您不能以这种方式实现反转钩子）。这是一个故意的限制，因为它大大加快了处理速度。

2.7 带参数的钩子

有时需要向钩子传递上下文信息，并且由于某种原因，无法将此类信息存储在宏中。为了满足这个目的，可以声明带参数的钩子，以便程序员可以传递钩子中代码所需的数据。

带参数的钩子的工作原理基本上与常规钩子相同，大多数适用于常规钩子的命令也适用于带参数的钩子。不同之处在于钩子的声明（使用 \NewHookWithArguments 而不是 \NewHook），然后可以使用 \AddToHook 和 \AddToHookWithArguments 添加代码，以及钩子的使用（使用 \UseHookWithArguments 而不是 \UseHook）。

带参数的钩子必须像常规钩子一样在首次使用前（所有常规钩子一样）声明，使用 \NewHookWithArguments{⟨*hook*⟩}{⟨*number*⟩}。然后添加到该钩子的所有代码都可以使用 #1 访问第一个参数，#2 访问第二个参数，依此类推，直到声明的参数数量。但是，仍然可以添加带有对尚未声明的钩子参数的引用的代码（稍后我们将讨论这一点）。钩子本质上是宏，所以 TEX 的 9 个参数限制适用，并且如果尝试引用不存在的参数号码，则会引发低级 TEX 错误。

要使用带参数的钩子，只需写 \UseHookWithArguments{⟨*hook*⟩}{⟨*number*⟩}，然后接着是参数的大括号列表。例如，如果钩子 test 需要三个参数，写法如下：

```
\UseHookWithArguments{test}{3}{arg-1}{arg-2}{arg-3}
```

然后，在钩子的 ⟨*code*⟩ 中，所有的 #1 将被替换为 arg-1，#2 将被替换为 arg-2，以此类推。如果在使用时，程序员提供的参数多于钩子声明的参数，则超出的参数将被钩子简单地忽略。如果提供的参数过少，则行为是不可预测的[6]。如果钩子未被声明，⟨*number*⟩ 个参数将从输入流中移除。

[6]钩子 将采用声明的参数数量，发生了什么取决于被抓取的内容以及钩子代码对其参数的处理。

使用 `\AddToHookWithArguments` 可以像常规 `\AddToHook` 一样向带参数的钩子添加代码，以实现不同的结果。在这种情况下，向钩子添加代码的主要区别在于首先可以访问钩子的参数，当然还有参数标记（`#6`）的处理方式。

在带参数的钩子中使用 `\AddToHook` 将像对所有其他钩子一样工作。这允许包开发人员向本来没有参数的钩子添加参数，而无需担心兼容性问题。这意味着，例如：

```
\AddToHook{test}{\def\foo#1{Hello, #1!}}
```

无论钩子 `test` 是否带参数，都会定义相同的宏 `\foo`。

使用 `\AddToHookWithArguments` 允许向添加的 ⟨*code*⟩ 访问钩子的参数，如 `#1`、`#2` 等，直到钩子声明的参数数量。这意味着，如果想要在 ⟨*code*⟩ 中添加一个 `#6`，那个标记必须在输入中重复。上面的相同定义，使用 `\AddToHookWithArguments`，需要重写为：

```
\AddToHookWithArguments{test}{\def\foo##1{Hello, ##1!}}
```

将上述示例扩展为使用钩子参数，我们可以重写类似以下内容的代码（现在从声明到使用，以获得完整的画面）：

```
\NewHookWithArguments{test}{1}
\AddToHookWithArguments{test}{%
  \typeout{Defining foo with "#1"}
  \def\foo##1{Hello, ##1! Some text after: #1}%
}
\UseHook{test}{Howdy!}
\ShowCommand\foo
```

上述代码运行后会在终端打印：

```
Defining foo with "Howdy!"
> \foo=macro:
#1->Hello, #1! Some text after: Howdy!.
```

请注意，在对 `\AddToHookWithArguments` 的调用中，`##1` 变为了 `#1`，而 `#1` 被传递给钩子的参数。如果再次使用钩子并提供不同的参数，定义自然会发生变化。

在声明钩子和确定钩子数量固定之前，可以添加引用钩子参数的代码。但是，如果钩子中添加的某些代码引用的参数多于将为该钩子声明的参数数量，则在钩子声明时会出现低级 TEX 错误，指示“非法参数编号”，这将很难追踪，因为在这一点上 TEX 无法知道引起问题的代码来自何处。因此，包编写者明确记录每个钩子可以接受多少个参数（如果有的话）是非常重要的，以便使用这些包的用户知道可以引用多少个参数，同样重要的是，了解每个参数的含义。

2.8 私有的 LaTeX 核心钩子

有几个地方对于 LaTeX 正确运行而言绝对至关重要，需要按照精确定义的顺序执行代码。即使可以通过钩子管理实现这一点（通过添加各种规则来确保与包添加的其他代码的适当排序），但这会使每个文档变得不必要地缓慢，因为即使结果是预先确定的，也必须进行排序。此外，这会强迫包作者不必要地为钩子添加进一步的规则（或者破坏 LaTeX）。

出于这个原因，此类代码不使用钩子管理，而是直接在公共钩子之前或之后使用私有内核命令,命名约定如下:`\@kernel@before@`⟨*hook*⟩ 或 `\@kernel@after@`⟨*hook*⟩。例如，在 `\enddocument` 中你会找到:

```
\UseHook{enddocument}%
\@kernel@after@enddocument
```

这意味着首先执行用户/包可访问的 `enddocument` 钩子，然后执行内部核心钩子。正如它们的名称所示，这些内核命令不应由第三方包更改，请不要这样做，这样有利于稳定性，而是使用其旁边的公共钩子。[7]

2.9 遗留的 LaTeX 2_ε 接口

LaTeX 2_ε 提供了一小部分钩子以及用于向其添加代码的命令。它们在这里列出，并保留了向后兼容性。

使用新的钩子管理机制，LaTeX 添加了几个额外的钩子，未来还将添加更多。请参见下一节以了解已经可用的内容。

[7]与 TeX 中的所有内容一样，没有强制执行此规则，通过查看代码很容易发现内核如何向其添加内容。因此，这个部分的主要目的是说:“请不要这样做，这是不可配置的代码!”

\AtBeginDocument `\AtBeginDocument [⟨label⟩] {⟨code⟩}`

如果不使用可选参数 ⟨*label*⟩,它基本上与以前一样,即将 ⟨*code*⟩ 添加到 `begindocument` 钩子（在 `\begin{document}` 内执行）。但是，通过这种方式添加的所有代码都使用标签 `top-level` 进行标记（参见第 2.1.5 节），如果在包或类之外进行，或者使用包/类名称，如果在这样的文件内部调用（参见第 2.1.4 节）。

这样，使用 `\AddToHook` 或 `\AtBeginDocument` 使用不同的标签显式地按照需要排序代码块，例如，在另一个包的代码之前或之后运行某些代码。当使用可选参数时，该调用等效于运行 `\AddToHook {begindocument} [`⟨*label*⟩`] {`⟨*code*⟩`}`。

`\AtBeginDocument` 是 `begindocument` 钩子（参见第 3.2 节）的包装器，它是一个一次性钩子。因此，在 `begindocument` 钩子在 `\begin{document}` 处执行后，任何尝试使用 `\AtBeginDocument` 或 `\AddToHook` 向该钩子添加 ⟨*code*⟩ 的操作都将导致该 ⟨*code*⟩ 立即执行。有关一次性钩子的更多信息，请参见第 2.5 节。

对于具有已知顺序要求的重要包，我们可能会随着时间的推移向内核（或这些包）添加规则，以便它们不受文档加载顺序的影响而工作。

\AtEndDocument `\AtEndDocument [⟨label⟩] {⟨code⟩}`

Like `\AtBeginDocument` but for the `enddocument` hook.

在 LaTeX 2$_\varepsilon$ 中之前存在的少量钩子在内部使用诸如 `\@begindocumenthook` 之类的命令，有时包直接增强它们而不是通过 `\AtBeginDocument` 进行操作。出于这个原因，目前支持这样做，也就是说，如果系统检测到这样一个内部传统钩子命令包含代码，则将其添加到新的钩子系统中，并使用标签 `legacy` 进行标记，以防止丢失。

然而, 随着时间的推移, 剩余的直接使用情况需要更新, 因为在未来的某个 LaTeX 发布中，我们将关闭此传统支持，因为它会不必要地减慢处理速度。

3 LaTeX 2$_\varepsilon$ 命令和由钩子增强的环境

在本节中，我们描述了现在由 LaTeX 提供的标准钩子，或者提供了指向其他文档的指针，其中对它们进行了描述。本节将随时间而增长（并且可能最终会转移到 usrguide3）。

3.1 通用钩子

正如前面所述，除了通用钩子之外，所有钩子在使用之前都必须使用 `\NewHook` 声明。所有通用钩子的名称都采用以下形式：“⟨*type*⟩/⟨*name*⟩/⟨*position*⟩”, 其中 ⟨*type*⟩ 取自下面预定义的列表，⟨*name*⟩ 是其含义将取决于 ⟨*type*⟩ 的变量部分。最后一个组成部分 ⟨*position*⟩ 具有更复杂的可能性：它始终可以是 `before` 或 `after`；对于 `env`

钩子，还可以是 begin 或 end；对于 include 钩子，还可以是 end。每个特定的钩子在下面或 ltcmdhooks-doc.pdf 或 ltfilehook-doc.pdf 中有文档记录。

LaTeX 提供的通用钩子属于以下六种类型：

env 在环境之前和之后执行的钩子——⟨*name*⟩ 是环境的名称，⟨*position*⟩ 的可用值为 before、begin、end 和 after；

cmd 添加到命令之前和之后执行的钩子——⟨*name*⟩ 是命令的名称，⟨*position*⟩ 的可用值为 before 和 after；

file 在读取文件之前和之后执行的钩子——⟨*name*⟩ 是文件的名称（带有扩展名），⟨*position*⟩ 的可用值为 before 和 after；

package 在加载包之前和之后执行的钩子——⟨*name*⟩ 是包的名称，⟨*position*⟩ 的可用值为 before 和 after；

class 在加载类之前和之后执行的钩子——⟨*name*⟩ 是类的名称，⟨*position*⟩ 的可用值为 before 和 after；

include 在 \include 包含的文件之前和之后执行的钩子——⟨*name*⟩ 是包含的文件的名称（不包含 .tex 扩展名），⟨*position*⟩ 的可用值为 before、end 和 after。

下面详细介绍了上述每个钩子，并提供了链接的文档。

3.1.1 所有环境的通用钩子

每个环境 ⟨*env*⟩ 现在都有四个与之关联的钩子：

env/⟨*env*⟩/before 这个钩子作为 \begin 的一部分执行，特别是在开始环境组之前。因此，它的范围不受环境的限制。

env/⟨*env*⟩/begin 这个钩子作为 \begin 的一部分直接位于特定于环境开始的代码之前（例如，\newenvironment 的第二个参数）。它的范围是环境主体。

env/⟨*env*⟩/end 这个钩子作为 \end 的一部分直接位于特定于环境结束的代码之前（例如，\newenvironment 的第三个参数）。

env/⟨*env*⟩/after 这个钩子作为 \end 的一部分，在环境结束的代码和环境组结束之后执行。因此，它的范围不受环境的限制。

该钩子实现为一个反向钩子，因此，如果两个包向 env/⟨*env*⟩/before 和 env/⟨*env*⟩/after 添加代码，它们可以添加周围的环境，且关闭它们的顺序是正确的。

通用环境钩子即使对于只能在文档中出现一次的环境也不是一次性钩子。[8] 与其他钩子不同，也不需要使用 \NewHook 声明它们。

这些钩子只有在使用 \begin{⟨*env*⟩} 和 \end{⟨*env*⟩} 时才会执行。如果环境代码是通过对 \⟨*env*⟩ 和 \end⟨*env*⟩ 进行低级调用（例如，为了避免环境组），则它们不可用。如果要在使用此方法的代码中使用它们，您需要自己添加它们，即编写类似以下内容的代码：

```
\UseHook{env/quote/before}\quote
    ...
\endquote\UseHook{env/quote/after}
```

以添加外部钩子等。

为了与现有包的兼容性，还提供了以下四个命令来设置环境钩子；但对于新的包，我们建议直接使用钩子名称和 \AddToHook。

\BeforeBeginEnvironment `\BeforeBeginEnvironment [⟨*label*⟩] {⟨*env*⟩} {⟨*code*⟩}`

此声明使用 ⟨*label*⟩ 将代码添加到 env/⟨*env*⟩/before 钩子中。如果未给出 ⟨*label*⟩，则使用 ⟨*default label*⟩（参见第 2.1.4 节）。

\AtBeginEnvironment `\AtBeginEnvironment [⟨*label*⟩] {⟨*env*⟩} {⟨*code*⟩}`

这类似于 \BeforeBeginEnvironment，但它将代码添加到 env/⟨*env*⟩/begin 钩子中。

\AtEndEnvironment `\AtEndEnvironment [⟨*label*⟩] {⟨*env*⟩} {⟨*code*⟩}`

这类似于 \BeforeBeginEnvironment，但它将代码添加到 env/⟨*env*⟩/end 钩子中。

\AfterEndEnvironment `\AfterEndEnvironment [⟨*label*⟩] {⟨*env*⟩} {⟨*code*⟩}`

这类似于 \BeforeBeginEnvironment，但它将代码添加到 env/⟨*env*⟩/after 钩子中。

3.1.2 命令的通用钩子

与环境类似，现在（至少在理论上）对于任何 LaTeX 命令都有两个通用钩子可用。它们是：

cmd/⟨*name*⟩/before 此钩子在命令执行的开头执行。

cmd/⟨*name*⟩/after 此钩子在命令体的最后执行。它实现为一个反向钩子。

实际上有一些限制，尤其是 after 钩子仅适用于一部分命令。有关这些限制的详细信息可以在 ltcmdhooks-doc.pdf 中找到，或者在 ltcmdhooks-code.pdf 中查看代码。

[8] 因此，如果在处理环境之后添加代码，只有在环境再次出现且不会发生代码执行时，该代码才会被执行。

3.1.3 文件加载操作提供的通用钩子

在通过其高级接口加载文件(例如 \input、\include、\usepackage、\RequirePackage 等）时，LaTeX 添加了几个钩子。这些钩子在 ltfilehook-doc.pdf 中有文档说明，或者可以在 ltfilehook-code.pdf 中查看代码。

3.2 \begin{document} 提供的钩子

直到 2020 年，\begin{document} 仅提供了一个可通过 \AtBeginDocument 添加的钩子。多年的经验表明，在一个地方使用这个单一的钩子是不够的，因此，在添加通用钩子管理系统的过程中，在此处添加了许多其他的钩子。这些钩子的位置被选择为提供与外部包（例如 etoolbox 和其他增强 \document 以获得更好控制的包）所提供的支持相同。

现在支持以下钩子（它们都是一次性钩子）：

begindocument/before 此钩子在 \document 开始时执行，可以将其视为位于导言区末尾的代码的钩子，这就是 etoolbox 的 \AtEndPreamble 使用它的方式。

这是一个一次性钩子，因此在执行后，所有进一步尝试添加代码的操作都将立即执行该代码（参见第 2.5 节）。

begindocument 这个钩子是通过使用 \AddToHook{begindocument} 或使用 \AtBeginDocument 添加的，它在读取 .aux 文件和大多数初始化完成后执行，因此可以被钩子代码修改和检查。它后面紧跟一些不应该被更改的进一步初始化，因此稍后会出现。

该钩子不应该用于添加排版素材，因为我们仍然处于 LaTeX 的初始化阶段，而不是文档主体。如果需要将此类素材添加到文档主体中，请改用下一个钩子。

这是一个一次性钩子，因此在执行后，所有进一步尝试添加代码的操作都将立即执行该代码（参见第 2.5 节）。

begindocument/end 此钩子在 \document 代码结束时执行，换句话说，在文档主体开始时执行。其后唯一的命令是 \ignorespaces。

这是一个一次性钩子，因此在执行后，所有进一步尝试添加代码的操作都将立即执行该代码（参见第 2.5 节）。

\begin 执行的通用钩子也存在，即 env/document/before 和 env/document/begin，但对于此特殊环境，最好使用上述专用的一次性钩子。

3.3 \end{document} 提供的钩子

LaTeX 2_ε 一直提供 \AtEndDocument 来添加代码到 \end{document}，就在通常执行的代码前面。尽管这对于 LaTeX 2.09 的情况是一个很大的改进，但对于许多用

例来说并不够灵活，因此，诸如 etoolbox、atveryend 等包对 \enddocument 进行了补丁，以添加额外的代码挂载点。

使用包进行补丁总是有问题的，因为会导致冲突（代码可用性、补丁的顺序、不兼容的补丁等）。因此，在 \enddocument 代码中添加了一些额外的钩子，允许包以受控的方式在各个地方添加代码，而无需覆盖或补丁核心代码。

现在支持以下钩子（它们都是一次性钩子）：

enddocument 与 \AtEndDocument 相关联的钩子。它在 \enddocument 开始时立即调用。

当执行此钩子时，可能仍有未处理的素材（例如推迟列表上的浮动体），而钩子可能会添加进一步要排版的素材。之后，调用 \clearpage 来确保所有这样的素材都被排版。如果没有等待的素材，则 \clearpage 没有效果。

这是一个一次性钩子，因此在执行后，所有进一步尝试添加代码的操作都将立即执行该代码（参见第 2.5 节）。

enddocument/afterlastpage 如名称所示，此钩子不应该接收生成更多页面素材的代码。这是做一些最终的收尾工作的正确位置，可能要向 .aux 文件写一些信息（在此时，该文件仍然打开以接收数据，但由于不会再有页面，您需要使用 \immediate\write 来写入它）。这也是设置任何在下一步重新读取 .aux 文件时运行的测试代码的正确位置。

执行此钩子后，.aux 文件将关闭写入，并重新读取以进行一些测试（例如查找缺失引用或重复标签等）。

这是一个一次性钩子，因此在执行后，所有进一步尝试添加代码的操作都将立即执行该代码（参见第 2.5 节）。

enddocument/afteraux 此时，.aux 文件已经被重新处理，因此这是进行最终检查和向用户显示信息的可能位置。但是，对于后者，您可能更喜欢下一个钩子，这样您的信息会显示在（可能较长的）文件列表之后，如果通过 \listfiles 请求的话。

这是一个一次性钩子，因此在执行后，所有进一步尝试添加代码的操作都将立即执行该代码（参见第 2.5 节）。

enddocument/info 此钩子用于接收向终端写入最终信息消息的代码。它紧随上一个钩子之后执行（因此两者可以合并，但是然后添加更多代码的包始终需要提供显式规则来指定它应该放在何处）。

此钩子已经包含内核添加的一些代码（标签重复警告、缺失引用、字体替换等），即在使用 \listfiles 时列出的文件列表和警告信息。

这是一个一次性钩子，因此在执行后，所有进一步尝试添加代码的操作都将立即执行该代码（参见第 2.5 节）。

enddocument/end 最后，此钩子在最终调用 \@@end 前执行。

这是一个一次性钩子，因此在执行后，所有进一步尝试添加代码的操作都将立即执行该代码（参见第 2.5 节）。甚至在此之后添加代码可能吗?

还有一个名为 shipout/lastpage 的钩子。此钩子作为文档中最后一个 \shipout 的一部分执行，以允许包将最终的 \special 添加到该页面。此钩子相对于上述列表中的钩子的执行时间可以因文档而异。此外，要正确确定哪个 \shipout 是最后一个，需要多次运行 LaTeX，因此最初它可能在错误的页面上执行。有关详细信息，请参阅第 3.4 节。

还可以使用通用的 env/document/end 钩子, 它是由 \end 执行的, 即在上述的第一个钩子前执行。但是请注意,另一个通用的 \end 环境钩子,即 env/document/after，永远不会被执行，因为此时 LaTeX 已经完成了文档处理。

3.4 \shipout 操作提供的钩子

在 LaTeX 生成页面的过程中添加了几个钩子和机制。这些内容在 ltshipout-doc.pdf 中有详细记录，或者在 ltshipout-code.pdf 中有相关代码。

3.5 段落提供的钩子

段落处理已经增加了一些内部和公共钩子。这些内容在 ltpara-doc.pdf 中有详细记录，或者在 ltpara-code.pdf 中有相关代码。

3.6 NFSS 命令提供的钩子

对于需要同时支持多个脚本（因此有几套字体，例如支持拉丁字体和日文字体）的语言，NFSS 字体命令如 \sffamily 需要同时切换拉丁字体为“Sans Serif”，并且额外修改第二套字体。

为了支持这一点，几个 NFSS 命令都有钩子来添加这种支持。

rmfamily 在 \rmfamily 执行了其初始检查并准备字体系列更新后，此钩子在 \selectfont 之前执行。

sffamily 这类似于 rmfamily 钩子，但用于 \sffamily 命令。

ttfamily 这类似于 rmfamily 钩子，但用于 \ttfamily 命令。

normalfont `\normalfont` 命令将字体编码、系列和形状重置为文档默认值。然后执行此钩子，最后调用 `\selectfont`。

expand@font@defaults 内部命令 `\expand@font@defaults` 展开并保存当前的元系列（rm/sf/tt）和元系列（bf/md）的默认值。如果为了中文或日文等增加了 NFSS 机制，则可能需要在此时设置进一步的默认值。这可以在此钩子中完成，在此宏的末尾执行。

bfseries/defaults, bfseries 如果用户显式更改了 `\bfdefault` 的值，则在调用 `\bfseries` 时将其新值用于设置元系列（rm/sf/tt）的 bf 系列默认值。在这种情况下，`bfseries/defaults` 钩子允许进一步进行调整。如果检测到这样的更改，则仅执行此钩子。相反，`bfseries` 钩子总是在调用 `\selectfont` 以更改新系列之前执行。

mdseries/defaults, mdseries 这两个钩子与上面的类似，但是在 `\mdseries` 命令中。

selectfont 此钩子在 `\selectfont` 内执行，用于评估当前的编码、系列、形状和大小，并选择新的字体（如果必要则加载）。在此钩子执行后，NFSS 仍会执行任何必要的更新以适应新的大小（例如更改 `\strut` 的大小）和更改编码。

此钩子用于在主要字体更改的同时，处理其他字体的情况（例如在处理多种不同字母表的 CJK 处理中）。

3.7 标记机制提供的钩子

详细内容请参阅 `ltmarks-doc.pdf`。

insertmark 此钩子允许在 `\InsertMark` 插入标记时进行特殊设置。它在分组中执行，因此局部更改仅适用于被插入的标记。

索引

斜体数字指向相应条目描述的页面，下划线数字指向定义的代码行，其它的都指向使用条目的页面。

Symbols

\⟨*addto-cmd*⟩ *4*

A

\ActivateGenericHook *3*, *21*

\AddToHook *2*, *4*, *5*, *7*, *12*, *17*, *20–23*, *25*, *27*, *28*

\AddToHookNext *6*, *7*, *12*, *17*

\AddToHookNextWithArguments *7*

\AddToHookWithArguments 5, 7, 22, 23
\AfterEndEnvironment 27
\AtBeginDocument 4, 10, 21, 25, 28
\AtBeginEnvironment 27
\AtEndDocument 25, 28, 29
\AtEndEnvironment 27
\AtEndPreamble 28

B

\BeforeBeginEnvironment 27
\begin 2, 26–28
\bfdefault 31
\bfseries 31

C

\ClearHookNext 7
\ClearHookRule 11
\clearpage 29
\csname 8

D

\DebugHooksOff 14
\DebugHooksOn 14
\DeclareDefaultHookRule 11
\DeclareHookRule 2, 6, 11, 14, 17
\DisableGenericHook 3
\DisableHook 3
\DiscardShipoutBox 7
\document 28
\documentclass 8, 9

E

\end 2, 26–28, 30
\endcsname 8
\enddocument 24, 29
\errorstopmode 13, 17

H

hook commands:

\hook_activate_generic:n 15
\hook_debug_off: 17
\hook_debug_on: 17
\hook_disable_generic:n 15
\hook_gclear_next_code:n 16
\hook_gput_code:nnn 16
\hook_gput_code_with_args:nnn ... 16
\hook_gput_next_code:nn 16
\hook_gput_next_code_with_-
args:nn 16
\hook_gremove_code:nn 16
\hook_gset_rule:nnnn 17
\hook_if_empty:nTF 8, 9, 17
\hook_if_empty_p:n 17
\hook_log:n 17
\hook_new:n 14, 15
\hook_new_pair:nn 14
\hook_new_pair_with_args:nn 15
\hook_new_pair_with_args:nnn 15
\hook_new_reversed:n 14
\hook_new_reversed_with_args:nn . 15
\hook_new_with_args:nn 15
\hook_show:n 17
\hook_use:n 8, 9, 15, 20
\hook_use:nnw 15, 16
\hook_use_once:n 8, 15, 20
\hook_use_once:nnw 15, 16

I

\IfHookEmptyTF 8, 9, 12
\ignorespaces 28
\immediate 29
\include 26, 28
\input 9, 28
\InsertMark 31

L

\listfiles 29
\LogHook 13

M

\mdseries 31

N

\newenvironment 26
\NewHook 2, 3, 12, 19, 21, 22, 25, 27
\NewHookWithArguments 3, 22
\NewMirroredHookPair 3
\NewMirroredHookPairWithArguments ... 3

\NewReversedHook 2, 3, 10, 19
\NewReversedHookWithArguments 3
\normalfont 31
\normalsize 6

P

\PopDefaultHookLabel 9, 10
\PushDefaultHookLabel 8–10

R

\RemoveFromHook 5, 6, 12
\RequirePackage 9, 28
\rmfamily 30

S

\selectfont 30, 31
\SetDefaultHookLabel 8–10
\sffamily 30
\shipout 30
\ShowHook 13, 18, 20
\small 6
\special 30
\strut 31

T

TEX and LATEX 2ε commands:
\@begindocumenthook 25
\@firstofone 4
\@kernel@after@⟨*hook*⟩ 24
\@kernel@before@⟨*hook*⟩ 24
\@@end 30
\expand@font@defaults 31
\ttfamily 30

U

\UseHook 3, 4, 8, 9, 18, 20–22
\UseHookWithArguments 4, 22
\UseOneTimeHook 3–5, 8, 20, 21
\UseOneTimeHookWithArguments 5
\usepackage 8, 9, 28
\usetikzlibrary 9

W

\write 29