第二章　Category Codes and Internal States
类别码和内部状态

When characters are read, TEX assigns them category codes. The reading mechanism has three internal states, and transitions between these states are effected by category codes of characters in the input. This chapter describes how TEX reads its input and how the category codes of characters influence the reading behaviour. Spaces and line ends are discussed.

当字符被读取时，TEX 将为它们分配类别码。读取机制具有三个内部状态，并且这些状态之间的转换受输入中字符的类别码的影响。本章描述了 TEX 如何读取其输入以及字符的类别码如何影响读取行为。还讨论了空格和换行符。

\endlinechar The character code of the end-of-line character appended to input lines. IniTEX default: 13.

附加到输入行末尾的行结束符的字符码。IniTEX 默认值：13。

\par Command to close off a paragraph and go into vertical mode. Is generated by empty lines.

用于结束段落并进入垂直模式的命令。由空行生成。

\ignorespaces Command that reads and expands until something is encountered that is not a ⟨space token⟩.

读取并展开直到遇到非 ⟨space token⟩ 的命令。

\catcode Query or set category codes.

查询或设置类别码。

\ifcat Test whether two characters have the same category code.

检查两个字符是否具有相同的类别码。

\ Control space. Insert the same amount of space that a space token would when \spacefactor = 1000.

控制空格。插入与空格记号相同量的空格，当 \spacefactor = 1000 时。

\obeylines Macro in plain TEX to make line ends significant.

plain TEX 中使换行符具有意义的宏。

\obeyspaces Macro in plain TEX to make (most) spaces significant.

plain TEX 中使（大多数）空格具有意义的宏。

2.1 Introduction
引言

TEX's input processor scans input lines from a file or terminal, and makes tokens out of the characters. The input processor can be viewed as a simple finite state automaton with three internal states; depending on the state its scanning behaviour may differ. This automaton will be treated here both from the point of view of the internal states and of the category codes governing the transitions.

TEX 的输入处理器从文件或终端扫描输入行，并将字符转换为记号。输入处理器可以看作是一个具有三个内部状态的简单有限状态自动机；根据状态的不同，其扫描行为可能有所不同。在这里，将从内部状态和控制状态转换的类别码的角度对该自动机进行讨论。

2.2 Initial processing
初始处理

Input from a file (or from the user terminal, but this will not be mentioned specifically most of the time) is handled one line at a time. Here follows a discussion of what exactly is an input line for TEX.

从文件（或用户终端，但大部分时间不会具体提到）获取的输入按行处理。下面讨论的是对于 TEX 来说，什么是一个输入行。

Computer systems differ with respect to the exact definition of an input line. The carriage return/line feed sequence terminating a line is most common, but some systems use just a line feed, and some systems with fixed record length (block) storage do not have a line terminator at all. Therefore TEX has its own way of terminating an input line.

关于输入行的确切定义在不同的计算机系统中有所不同。以回车/换行序列作为行终止符最为常见，但有些系统仅使用换行符，而一些固定记录长度（块）存储的系统根本没有行终止符。因此，TEX 有自己的方式来终止一个输入行。

1. An input line is read from an input file (minus the line terminator, if any).
   从输入文件中读取一个输入行（如果有行终止符，则不包括终止符）。
2. Trailing spaces are removed (this is for the systems with block storage, and it prevents confusion because these spaces are hard to see in an editor).
   删除行尾的空格（这适用于使用块存储的系统，它可以避免在编辑器中难以看到这些空格时引起混淆）。
3. The , by default ⟨return⟩ (code 13) is appended. If the value of \endlinechar is negative or more than 255 (this was 127 in versions of TEX older than version 3; see page ?? for more differences), no character is appended. The effect then is the same as if the line were to end with a comment character.
   默认情况下，追加一个，即回车符（代码为 13）。如果 \endlinechar 的值为负数或大于 255（在早于版本 3 的 TEX 中为 127；有关更多差异，请参见第 ?? 页），则不追加任何字符。此时的效果与行以注释字符结束相同。

Computers may also differ in the character encoding (the most common schemes are ascii and ebcdic), so TEX converts the characters that are read from the file to its own character codes. These codes are then used exclusively, so that TEX will perform the same on any system. For more on this, see Chapter ??.

计算机在字符编码方面也可能存在差异（最常见的方案是 ascii 和 ebcdic），因此 TEX 将从文件中读取的字符转换为其自己的字符编码。然后，这些编码将被独占使用，以便 TEX 在任何系统上执行相同的操作。有关更多信息，请参阅第 ?? 章。

2.3 Category codes

类别码

Each of the 256 character codes (0–255) has an associated category code, though not necessarily always the same one. There are 16 categories, numbered 0–15. When scanning the input, TEX thus forms character-code–category-code pairs. The input processor sees only these pairs; from them are formed character tokens, control sequence tokens, and parameter tokens. These tokens are then passed to TEX's expansion and execution processes.

每个字符编码（0–255）都有一个相关的类别码，尽管不一定总是相同的。共有 16 个类别，编号为 0–15。在扫描输入时，TEX 形成了字符编码–类别码对。输入处理器仅看到这些对；从中形成字符记号、控制序列记号和参数记号。然后将这些记号传递给 TEX 的扩展和执行过程。

A character token is a character-code–category-code pair that is passed unchanged. A control sequence token consists of one or more characters preceded by an escape character; see below. Parameter tokens are also explained below.

字符记号是传递的字符编码–类别码对，不做任何改变。控制序列记号由一个或多个字符组成，前面带有转义字符；请参阅下文。参数记号也将在下文中进行解释。

This is the list of the categories, together with a brief description. More elaborate explanations follow in this and later chapters.

下面是类别码的列表，以及简要说明。在本章和后续章节中，将提供更详细的解释。

0. Escape character; this signals the start of a control sequence. IniTEX makes the backslash \ (code 92) an escape character.
   转义字符；表示控制序列的开始。IniTEX 将反斜杠 \（代码 92）作为转义字符。
1. Beginning of group; such a character causes TEX to enter a new level of grouping. The plain format makes the open brace { a beginningof-group character.
   组开始；此类字符导致 TEX 进入一个新的分组级别。Plain TEX 将左花括号 { 作为组开始字符。

2. End of group; TEX closes the current level of grouping. Plain TEX has the closing brace } as end-of-group character.
   组结束；TEX 关闭当前的分组级别。Plain TEX 将右花括号 } 作为组结束字符。
3. Math shift; this is the opening and closing delimiter for math formulas. Plain TEX uses the dollar sign $ for this.
   数学模式切换；用于数学公式的开启和关闭定界符。Plain TEX 使用美元符号 $ 作为数学模式切换符号。
4. Alignment tab; the column (row) separator in tables made with \halign (\valign). In plain TEX this is the ampersand &.
   对齐制表符；在使用 \halign（\valign）制作的表格中作为列（行）的分隔符。在 plain TEX 中，这是和号 &。
5. End of line; a character that TEX considers to signal the end of an input line. IniTEX assigns this code to the ⟨return⟩, that is, code 13. Not coincidentally, 13 is also the value that IniTEX assigns to the \endlinechar parameter; see above.
   行结束；TEX 认为此字符表示输入行的结束。IniTEX 将此代码分配给 ⟨return⟩，即代码 13。巧合的是，13 也是 IniTEX 分配给 \endlinechar 参数的值；参见上文。
6. Parameter character; this indicates parameters for macros. In plain TEX this is the hash sign #.
   参数字符；用于指示宏的参数。在 plain TEX 中，这是井号 #。
7. Superscript; this precedes superscript expressions in math mode. It is also used to denote character codes that cannot be entered in an input file; see below. In plain TEX this is the circumflex ^.
   上标；在数学模式中用于表示上标表达式。还用于表示无法在输入文件中输入的字符代码；参见下文。在 plain TEX 中，这是插入符号 ^。
8. Subscript; this precedes subscript expressions in math mode. In plain TEX the underscore _ is used for this.
   下标；在数学模式中用于表示下标表达式。在 plain TEX 中，下划线 _ 用于表示下标。
9. Ignored; characters of this category are removed from the input, and have therefore no influence on further TEX processing. In plain TEX this is the

⟨null⟩ character, that is, code 0.
忽略；此类别的字符从输入中被删除，因此对后续的 TEX 处理没有影响。在 plain TEX 中，这是空字符，即代码 0。

10. Space; space characters receive special treatment. IniTEX assigns this category to the ascii ⟨space⟩ character, code 32.
空格；空格字符接受特殊处理。IniTEX 将此类别分配给 ASCII 的空格字符，代码 32。

11. Letter; in IniTEX only the characters a..z, A..Z are in this category. Often, macro packages make some 'secret' character (for instance @) into a letter.
字母；在 IniTEX 中，只有字符 a..z 和 A..Z 属于此类别。通常，宏包会将某些“秘密”字符（例如 @）归类为字母。

12. Other; IniTEX puts everything that is not in the other categories into this category. Thus it includes, for instance, digits and punctuation.
其他；IniTEX 将不属于其他类别的所有内容归类为此类别。因此，它包括数字和标点符号等内容。

13. Active; active characters function as a TEX command, without being preceded by an escape character. In plain TEX this is only the tie character ~, which is defined to produce an unbreakable space; see page ??.
活动字符；活动字符充当 TEX 命令，而不需要在前面加上转义字符。在 plain TEX 中，只有波浪号 ~ 是活动字符，它被定义为生成不可分割的空格；参见第?? 页。

14. Comment character; from a comment character onwards, TEX considers the rest of an input line to be comment and ignores it. In IniTEX the per cent sign % is made a comment character.
注释字符；从注释字符开始，TEX 将输入行的剩余部分视为注释并忽略它。在 IniTEX 中，百分号符号 % 被设为注释字符。

15. Invalid character; this category is for characters that should not appear in the input. IniTEX assigns the ascii ⟨delete⟩ character, code 127, to this category.
无效字符；此类别适用于不应出现在输入中的字符。IniTEX 将 ASCII 的删除字符，即代码 127，归类为此类别。

The user can change the mapping of character codes to category codes with the command (see Chapter ?? for the explanation of concepts such as ⟨equals⟩):

用户可以使用命令（参见第 ?? 章中的 ⟨equals⟩ 解释）来更改字符代码到类别代码的映射：

\catcode⟨number⟩⟨equals⟩⟨number⟩.

In such a statement, the first number is often given in the form

在这样的语句中，第一个数字通常以以下形式给出：

`⟨character⟩ or `\⟨character⟩

both of which denote the character code of the character (see pages ?? and ??).

这两种形式都表示字符的字符代码（参见第??页和??页）。

Plain 格式定义了\active （活动字符） The plain format defines \active

\chardef\active=13

这样，我们可以编写如下语句： so that one can write statements such as

\catcode`\{=\active

The \chardef command is treated on pages ?? and ??.

关于\chardef 命令的处理可以参考第??页和??页。

The LaTeX format has the control sequences

LaTeX 格式提供了以下控制序列：

```
\def\makeatletter{\catcode`@=11 }
\def\makeatother{\catcode`@=12 }
```

in order to switch on and off the ‘secret’ character @ (see below).

以便打开和关闭“秘密”字符 @（见下文）。

The \catcode command can also be used to query category codes: in

\catcode 命令还可以用于查询类别代码：

\count255=\catcode`\{

it yields a number, which can be assigned.

它将返回一个数字，可以进行赋值。

Category codes can be tested by

测试类别代码可以通过:

\ifcat⟨token$_1$⟩⟨token$_2$⟩

TEX expands whatever is after \ifcat until two unexpandable tokens are found; these are then compared with respect to their category codes. Control sequence tokens are considered to have category code 16, which makes them all equal to each other, and unequal to all character tokens. Conditionals are treated further in Chapter ??.

TEX 会展开\ifcat 后面的内容，直到找到两个不可展开的记号；然后它们将根据它们的类别代码进行比较。控制序列记号被认为具有类别代码 16，这使得它们彼此相等，并且与所有字符记号都不相等。条件语句将在第?? 章进一步讨论。

2.4 From characters to tokens
从字符到记号

The input processor of TEX scans input lines from a file or from the user terminal, and converts the characters in the input to tokens. There are three types of tokens.

TEX 的输入处理器从文件或用户终端扫描输入行，并将输入中的字符转换为记号。记号有三种类型。

- Character tokens: any character that is passed on its own to TEX's further levels of processing with an appropriate category code attached.
  字符记号：任何以适当的类别码单独传递给 TEX 的进一步处理层级的字符。
- Control sequence tokens, of which there are two kinds: an escape character – that is, a character of category 0 – followed by a string of 'letters' is lumped together into a control word, which is a single token. An escape character followed by a single character that is not of category 11, letter, is made into a control symbol. If the distinction between control word and control symbol is irrelevant, both are called control sequences.
  The control symbol that results from an escape character followed \ by a space character is called control space.
  控制序列记号，有两种类型：

  - 以转义字符（类别码为 0 的字符）开头，后跟一串“字母”的字符串，被组合成一个控制词，成为单个记号。
  - 转义字符后跟一个非类别码为 11（字母）的单个字符，被组合成一个控制符号。

  如果控制词和控制符号之间的区别无关紧要，两者都被称为控制序列。
  转义字符后跟一个空格字符的控制符号被称为控制空格。

- Parameter tokens: a parameter character – that is, a character of category 6, by default # – followed by a digit 1..9 is replaced by a parameter token. Parameter tokens are allowed only in the context of macros (see Chapter ??). A macro parameter character followed by another macro parameter character (not necessarily with the same character code) is replaced by a single character token. This token has category 6 (macro parameter), and the character code of the second parameter character. The most common instance is of this is replacing ## by $\#_6$, where the subscript denotes the category code.
  参数记号：参数字符（默认为 #，类别码为 6 的字符）后跟一个数字 1..9，被替换为一个参数记号。参数记号只允许在宏的上下文中使用（参见第 ?? 章）。
  以宏参数字符开头，后跟另一个宏参数字符（不一定具有相同的字符码）的记号将被替换为单个字符记号。该记号的类别码为 6（宏参数），字符码为第二个参数字符的字符码。最常见的情况是将 ## 替换为 $\#_6$, 其中下标表示类别码。

2.5 The input processor as a finite state automaton 输入处理器作为有限状态自动机

TEX's input processor can be considered to be a finite state automaton with three internal states, that is, at any moment in time it is in one of three states, and after transition to another state there is no memory of the previous states.

TEX 的输入处理器可以看作是一个有三个内部状态的有限状态自动机，也就是说，在任何时刻它都处于三个状态之一，且在转换到另一个状态后，不记忆先前的状态。

2.5.1 State N: new line
状态 N：新行

State N is entered at the beginning of each new input line, and that is the only time TEX is in this state. In state N all space tokens (that is, characters of category 10) are ignored; an end-of-line character is converted into a \par token. All other tokens bring TEX into state M.

状态 N 在每个新输入行的开头进入，并且在这个状态下，TEX 只会在这个时刻出现。在状态 N 中，所有空格记号（即，类别码为 10 的字符）都被忽略；换行符被转换为一个 \par 记号。其他所有记号都将使 TEX 进入状态 M。

2.5.2 State S: skipping spaces
状态 S：跳过空格

State S is entered in any mode after a control word or control space (but after no other control symbol), or, when in state M, after a space. In this state all subsequent spaces or end-of-line characters in this input line are discarded.

状态 S 在任何模式下进入，紧跟在控制词或控制空格之后（但不跟在其他控制符号之后），或者在状态 M 下跟在一个空格之后。在这个状态下，当前输入行中的所有后续空格或换行符都被丢弃。

2.5.3 State M: middle of line
状态 M：行中部分

By far the most common state is M, 'middle of line'. It is entered after characters of categories 1–4, 6–8, and 11–13, and after control symbols other than control space. An end-of-line character encountered in this state results in a space token.

远远最常见的状态是 M，即“行中部分”。在类别码为 1–4、6–8 和 11–13 的字符之后，以及除控制空格之外的其他控制符号之后，进入此状态。在此状态下遇到换行符会生成一个空格记号。

2.6 Accessing the full character set 访问完整字符集

Strictly speaking, TEX's input processor is not a finite state automaton. This is because during the scanning of the input line all trios consisting of two equal superscript characters (category code 7) and a subsequent character (with character code < 128) are replaced by a single character with a character code in the range 0–127, differing by 64 from that of the original character.

严格来说，TEX 的输入处理器并不是一个有限状态自动机。这是因为在扫描输入行时，所有由两个相等的上标字符（类别码 7）和后续字符（字符代码 < 128）组成的三元组，都会被替换为一个字符，其字符代码在 0–127 范围内，与原始字符的字符代码相差 64。

This mechanism can be used, for instance, to access positions in a font corresponding to character codes that cannot be input, for instance because they are ascii control characters. The most obvious examples are the ascii ⟨return⟩ and ⟨delete⟩ characters; the corresponding positions 13 and 127 in a font are accessible as ^^M and ^^?. However, since the category of ^^? is 15, invalid, that has to be changed before character 127 can be accessed.

这个机制可以用来访问与无法输入的字符代码对应的字体位置，例如因为它们是 ascii 控制字符而无法输入。最明显的例子是 ascii 的回车符和删除符；字体中对应的位置 13 和 127 可以分别通过 ^^M 和 ^^? 访问。然而，由于 ^^? 的类别码是 15，即无效的，必须在访问字符 127 之前将其更改。

In TEX3 this mechanism has been modified and extended to access 256 characters: any quadruplet ^^xy where both x and y are lowercase hexadecimal digits 0–9, a–f, is replaced by a character in the range 0–255, namely the character the number of which is represented hexadecimally as xy. This imposes a slight restriction on the applicability of the earlier mechanism: if, for instance, ^^a is typed to produce character 33, then a following 0–9, a–f will be misunderstood.

在 TEX3 中，这个机制已经被修改和扩展，以访问 256 个字符：任何由小写十六进制数字 0–9、a–f 组成的四元组 ^^xy，都会被替换为范围在 0–255 的字符，即其十六进制表示的数字为 xy 的字符。这对于早期机制的适用性有一点限制：

例如，如果键入 ^^a 来生成字符 33，那么后面的 0–9、a–f 将会被误解。

While this process makes TEX's input processor somewhat more powerful than a true finite state automaton, it does not interfere with the rest of the scanning. Therefore it is conceptually simpler to pretend that such a replacement of triplets or quadruplets of characters, starting with ^^, is performed in advance. In actual practice this is not possible, because an input line may assign category code 7 to some character other than the circumflex, thereby influencing its further processing.

在这个过程中，TEX 的输入处理器比真正的有限状态自动机更强大一些，但它不会干扰其余的扫描过程。因此，在概念上更简单的做法是假装事先进行这种以^^开头的字符三元组或四元组的替换。实际上，这是不可能的，因为输入行可能会将类别码为 7 的字符分配给除了插入符号之外的某些字符，从而影响其进一步处理。

2.7　Transitions between internal states
内部状态之间的转换

Let us now discuss the effects on the internal state of TEX's input processor when certain category codes are encountered in the input.

现在让我们讨论当输入中出现某些类别码时，对 TEX 的输入处理器的内部状态产生的影响。

2.7.1　0: escape character
0: 转义字符

When an escape character is encountered, TEX starts forming a control sequence token. Three different types of control sequence can result, depending on the category code of the character that follows the escape character.

当遇到转义字符时，TEX 会开始形成一个控制序列记号。根据转义字符后面的字符的类别码，可以得到三种不同类型的控制序列。

- If the character following the escape is of category 11, letter, then TEX combines the escape, that character and all following characters of category 11, into a control word. After that TEX goes into state S, skipping spaces.
  如果转义字符后面的字符是类别码为 11 的字母，则 TEX 会将转义字符、该字符以及所有后续的类别码为 11 的字符组合成一个控制词。之后，TEX 进入状态 S，跳过空格。
- With a character of category 10, space, a control symbol called control space results, and TEX goes into state S.
  如果转义字符后面的字符是类别码为 10 的空格，则得到一个称为控制空格的控制符号，TEX 进入状态 S。
- With a character of any other category code a control symbol results, and TEX goes into state M, middle of line.
  如果转义字符后面的字符是其他任何类别码的字符，则得到一个控制符号，TEX 进入状态 M，即行中间状态。

The letters of a control sequence name have to be all on one line; a control sequence name is not continued on the next line if the current line ends with a comment sign, or if (by letting \endlinechar be outside the range 0–255) there is no terminating character.

控制序列名必须全部位于同一行；如果当前行以注释符结束，或者（通过使 \endlinechar 超出范围 0–255）没有终止字符，则控制序列名不会延续到下一行。

2.7.2 1–4, 7–8, 11–13: non-blank characters
1–4, 7–8, 11–13：非空格字符

Characters of category codes 1–4, 7–8, and 11–13 are made into tokens, and TEX goes into state M.

类别码为 1–4、7–8 和 11–13 的字符被转换为记号，TEX 进入状态 M。

2.7.3　5: end of line
行尾

Upon encountering an end-of-line character, TEX discards the rest of the line, and starts processing the next line, in state N. If the current state was N, that is, if the line so far contained at most spaces, a \par token is inserted; if the state was M, a space token is inserted, and in state S nothing is inserted.

遇到行终止符时，TEX 会丢弃当前行的剩余部分，并开始处理下一行，进入状态 N。如果当前状态为 N，也就是说，到目前为止的行最多只包含空格，那么会插入一个 \par 记号；如果状态为 M，则插入一个空格记号；而在状态 S 下则不插入任何记号。

Note that by 'end-of-line character' a character with category code 5 is meant. This is not necessarily the \endlinechar, nor need it appear at the end of the line. See below for further remarks on line ends.

请注意，“行终止符”指的是类别码为 5 的字符。它不一定是 \endlinechar，也不一定出现在行末。有关行尾的进一步说明，请参见下文。

2.7.4　6: parameter
6：参数

Parameter characters – usually # – can be followed by either a digit 1..9 in the context of macro definitions or by another parameter character. In the first case a 'parameter token' results, in the second case only a single parameter character is passed on as a character token for further processing. In either case TEX goes into state M.

参数字符（通常为 #）可以后跟宏定义上下文中的数字 1..9，也可以后跟另一个参数字符。在第一种情况下，会产生一个“参数记号”，在第二种情况下，只有一个单独的参数字符作为字符记号传递给进一步处理。在任一情况下，TEX 进入状态 M。

A parameter character can also appear on its own in an alignment preamble (see Chapter ??).

参数字符还可以在对齐前导中单独出现（参见第 ?? 章）。

2.7.5 7: superscript
7: 上标

A superscript character is handled like most non-blank characters, except in the case where it is followed by a superscript character of the same character code. The process that replaces these two characters plus the following character (possibly two characters in TEX3) by another character was described above.

处理上标字符与大多数非空白字符相同，除非它后面跟着一个相同字符编码的上标字符。这个过程会将这两个字符以及后面的字符（在 TEX3 中可能是两个字符）替换为另一个字符，这个过程在前面已经描述过。

2.7.6 9: ignored character
9: 忽略字符

Characters of category 9 are ignored; TEX remains in the same state.

类别码为 9 的字符会被忽略；TEX 保持在相同的状态中。

2.7.7 10: space
10: 空格

A token with category code 10 – this is called a ⟨space token⟩, irrespective of the character code – is ignored in states N and S (and the state does not change); in state M TEX goes into state S, inserting a token that has category 10 and character code 32 (ascii space), that is, the character code of the space token may change from the character that was actually input.

具有类别码为 10 的记号（称为 ⟨space token⟩）在状态 N 和 S 中被忽略（状态不会改变）；在状态 M 中，TEX 进入状态 S，插入一个类别码为 10 和字符编码为 32（ascii 空格）的记号

2.7.8　14: comment
14: 注释

A comment character causes TEX to discard the rest of the line, including the comment character. In particular, the end-of-line character is not seen, so even if the comment was encountered in state M, no space token is inserted.

注释字符会导致 TEX 丢弃行的其余部分，包括注释字符本身。特别地，换行符不可见，因此即使在状态 M 中遇到注释，也不会插入空格记号。

2.7.9　15: invalid
15: 无效字符

Invalid characters cause an error message. TEX remains in the state it was in. However, in the context of a control symbol an invalid character is acceptable. Thus \^^? does not cause any error messages.

无效字符会引发错误消息。TEX 保持在原来的状态中。然而，在控制符号的上下文中，无效字符是可接受的。因此，^^? 不会引发任何错误消息。

2.8　Letters and other characters
字母和其他字符

In most programming languages identifiers can consist of both letters and digits (and possibly some other character such as the underscore), but control sequences in TEX are only allowed to be formed out of characters of category 11, letter. Ordinarily, the digits and punctuation symbols have category 12, other character. However, there are contexts where TEX itself generates a string of characters, all of which have category code 12, even if that is not their usual category code.

在大多数编程语言中，标识符可以由字母和数字（以及可能的其他字符，如下划线）组成，但是在 TEX 中，控制序列只能由类别码为 11 的字母字符构成。通常，数字和标点符号的类别码为 12，表示其他字符。然而，在某些上下文中，

即使这些字符的通常类别码不是 12，TEX 本身也会生成一串类别码为 12 的字符。

This happens when the operations \string, \number, \romannumeral, \jobname, \fontname, \meaning, and \the are used to generate a stream of character tokens. If any of the characters delivered by such a command is a space character (that is, character code 32), it receives category code 10, space.

当使用操作符 \string、\number、\romannumeral、\jobname、\fontname、\meaning 和 \the 生成一串字符记号时，就会发生这种情况。如果由此类命令提供的任何字符是空格字符（即字符代码为 32），它将获得类别码为 10，表示空格。

For the extremely rare case where a hexadecimal digit has been hidden in a control sequence, TEX allows A_{12}–F_{12} to be hexadecimal digits, in addition to the ordinary A_{11}–F_{11} (here the subscripts denote the category codes).

对于极为罕见的情况，即十六进制数字被隐藏在控制序列中的情况，TEX 允许 A12–F12 作为十六进制数字，除了常规的 A11–F11（这里下标表示类别码）。

例如，　For example,

\string\end　生成四个字符记号 gives four character tokens　$\_{12}e_{12}n_{12}d_{12}$

请注意，　Note that

$\_{12}$ is used in the output only because the value of \escapechar is the character code for the backslash. Another value of \escapechar leads to another character in the output of \string. The \string command is treated further in Chapter ??.

$\_{12}$ 仅在输出中使用，因为 \escapechar 的值是反斜杠的字符代码。\escapechar 的另一个值会导致 \string 输出中的另一个字符。\string 命令将在第 ?? 章进一步讨论。

空格可能出现在控制序列中：　Spaces can wind up in control sequences:

```
\csname a b\endcsname
```

gives a control sequence token in which one of the three characters is a space. Turning this control sequence token into a string of characters

生成一个控制序列记号，其中三个字符之一是空格。将此控制序列记号转换为一串字符

```
\expandafter\string\csname a b\endcsname
```

得到 gives $\_{12}a_{12}\ {}_{10}b_{12}$.

As a more practical example, suppose there exists a sequence of input files file1.tex, file2.tex, and we want to write a macro that finds the number of the input file that is being processed. One approach would be to write

作为一个更实际的例子，假设存在一系列输入文件 file1.tex、file2.tex，我们想要编写一个宏来找出正在处理的输入文件的编号。一种方法是编写以下代码：

```
\newcount\filenumber  \def\getfilenumber file#1.{\filenumber=#1 }
\expandafter\getfilenumber\jobname.
```

where the letters file in the parameter text of the macro (see Section ??) absorb that part of the jobname, leaving the number as the sole parameter.

在宏的参数文本中，字母 file（参见第 ?? 节）吸收了作业名的那部分，将编号作为唯一参数。

However, this is slightly incorrect: the letters file resulting from the \jobname command have category code 12, instead of 11 for the ones in the definition of \getfilenumber. This can be repaired as follows:

然而，这有点不正确：\jobname 命令生成的字母 file 的类别码是 12，而不是 \getfilenumber 宏定义中的类别码 11。可以通过以下方式进行修复：

```
{\escapechar=-1
 \expandafter\gdef\expandafter\getfilenumber
      \string\file#1.{\filenumber=#1 }
}
```

Now the sequence \string\file gives the four letters $f_{12}i_{12}l_{12}e_{12}$; the \expandafter commands let this be executed prior to the macro definition; the backslash is omitted because we put \escapechar=-1. Confining this value to a group makes it necessary to use \gdef.

现在，序列 \string\file 得到的是四个字母 f12i12l12e12；\expandafter 命令使其在宏定义之前执行；由于我们设置了 \escapechar=-1，所以省略了反斜杠。将该值限定在一个组中，需要使用 \gdef。

2.9　The \par token

\par 记号

TEX inserts a token into the input after encountering a character with category code 5, end of line, in state N. It is good to realize when exactly this happens: since TEX leaves state N when it encounters any token but a space, a line giving a \par can only contain characters of category 10. In particular, it cannot end with a comment character. Quite often this fact is used the other way around: if an empty line is wanted for the layout of the input one can put a comment sign on that line.

在状态为 N 时，TEX 在遇到类别码为 5 的字符（行尾）后会将一个 记号插入输入中。了解这个发生的准确时机很重要：由于 TEX 在遇到除空格以外的任何记号时都会离开状态 N，包含 \par 的行只能包含类别码为 10 的字符。特别要注意的是，行不能以注释字符结尾。通常情况下，这个事实可以反过来使用：如果希望在输入的布局中使用空行，可以在该行上加上注释符号。

Two consecutive empty lines generate two \par tokens. For all practical purposes this is equivalent to one \par, because after the first one TEX enters vertical mode, and in vertical mode a \par only exercises the page builder, and clears the paragraph shape parameters.

连续两个空行会生成两个 \par 记号。在实际应用中，这与一个 \par 是等效的，因为在第一个 \par 后，TEX 进入垂直模式，而在垂直模式下，\par 只会触发页面生成器，并清除段落形状参数。

A \par is also inserted into the input when TEX sees a ⟨vertical command⟩ in unrestricted horizontal mode. After the \par has been read and expanded, the vertical command is examined anew (see Chapters ?? and ??).

当 TEX 在无限水平模式中看到 ⟨vertical command⟩ 时，也会将一个 \par 插入输入。在读取和展开 \par 后，会重新检查垂直命令（参见第 ??章和第?? 章）。

The \par token may also be inserted by the \end command that finishes off the run of TEX; see Chapter ??.

在 TEX 的运行结束时，通过 \end 命令插入 \par 记号；详见第 ?? 章。

It is important to realize that TEX does what it normally does when encountering an empty line (which is ending a paragraph) only because of the default definition

of the \par token. By redefining \par the behaviour caused by empty lines and vertical commands can be changed completely, and interesting special effects can be achieved. In order to continue to be able to cause the actions normally associated with \par, the synonym \endgraf is available in the plain format. See further Chapter ??.

重要的是要意识到，TEX 在遇到空行时所做的通常操作（即结束段落）仅因为 \par 记号的默认定义。通过重新定义 \par，可以完全改变空行和垂直命令引起的行为，并实现有趣的特殊效果。为了仍然能够引发通常与 \par 相关的操作，plain 格式中提供了同义词 \endgraf。更多信息请参见第 ?? 章。

The \par token is not allowed to be part of a macro argument, unless the macro has been declared to be \long. A \par in the argument of a non-\long macro prompts TEX to give a 'runaway argument' message. Control sequences that have been \let to \par (such as \endgraf) are allowed, however.

在非 \long 宏的参数中，不允许包含 \par 记号，否则 TEX 会给出 "runaway argument"（参数溢出）的错误消息。但是，允许使用 \let 给予 \par 的控制序列（例如 \endgraf）。

2.10　Spaces
空格

This section treats some of the aspects of space characters and space tokens in the initial processing stages of TEX. The topic of spacing in text typesetting is treated in Chapter ??.

本节介绍 TEX 在初始处理阶段中处理空格字符和空格记号的一些方面。关于文本排版中的间距问题会在第 ?? 章中详细讨论。

2.10.1　Skipped spaces
跳过的空格

From the discussion of the internal states of TEX's input processor it is clear that some spaces in the input never reach the output; in fact they never get

past the input processor. These are for instance the spaces at the beginning of an input line, and the spaces following the one that lets TEX switch to state S.

从 TEX 的输入处理器的内部状态的讨论中可以看出，输入中的某些空格永远不会到达输出；事实上，它们甚至无法通过输入处理器。例如，在输入行的开头的空格以及让 TEX 切换到状态 S 后的空格都属于此类空格。

On the other hand, line ends can generate spaces (which are not in the input) that may wind up in the output. There is a third kind of space: the spaces that get past the input processor, or are even generated there, but still do not wind up in the output. These are the ⟨optional spaces⟩ that the syntax of TEX allows in various places.

另一方面，换行符可能会生成（未在输入中的）空格，这些空格可能最终出现在输出中。还有第三种空格：那些通过输入处理器、甚至在其中生成的空格，但最终不会出现在输出中。这些是 TEX 语法在各个位置允许的 ⟨optional spaces⟩（可选空格）。

2.10.2 Optional spaces 可选空格

The syntax of TEX has the concepts of 'optional spaces' and 'one optional space':

TEX 的语法中有“可选空格”和“一个可选空格”的概念：

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

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

In general, ⟨one optional space⟩ is allowed after numbers and glue specifications, while ⟨optional spaces⟩ are allowed whenever a space can occur inside a number (for example, between a minus sign and the digits of the number) or glue specification (for example, between plus and 1fil). Also, the definition of ⟨equals⟩ allows ⟨optional spaces⟩ before the = sign.

一般来说，在数字和粘连规范之后可以有 ⟨one optional space⟩，而在数字内部（例如在负号和数字之间）或粘连规范内部（例如在 plus 和 1fil 之间）可以有 ⟨optional spaces⟩。此外，在 ⟨equals⟩ 的定义中允许在等号 = 之前有 ⟨optional spaces⟩。

Here are some examples of optional spaces.

下面是一些可选空格的示例。

- A number can be delimited by ⟨one optional space⟩. This prevents accidents (see Chapter ??), and it speeds up processing, as TEX can detect more easily where the ⟨number⟩ being read ends. Note, however, that not every 'number' is a ⟨number⟩: for instance the 2 in \magstep2 is not a number, but the single token that is the parameter of the \magstep macro. Thus a space or line end after this is significant. Another example is a parameter number, for example #1: since at most nine parameters are allowed, scanning one digit after the parameter character suffices.
  数字可以由 ⟨one optional space⟩ 来界定。这可以避免错误（参见第 ?? 章），并且加快处理速度，因为 TEX 可以更容易地检测出正在读取的 ⟨number⟩ 的结束位置。但是请注意，并不是每个“数字”都是 ⟨number⟩：例如，\magstep2 中的 2 不是一个数字，而是作为 \magstep 宏的参数的单个记号。因此，在此之后的空格或行结束符是有意义的。另一个例子是参数编号，例如 #1：由于最多只允许九个参数，因此在参数字符之后扫描一个数字即可。
- From the grammar of TEX it follows that the keywords fill and filll consist of fil and separate ls, each of which is a keyword (see page ?? for a more elaborate discussion), and hence can be followed by optional spaces. Therefore forms such as fil L l are also valid. This is a potential source of strange accidents. In most cases, appending a \relax token prevents such mishaps.
  根据 TEX 的语法，关键词 fill 和 filll 由 fil 和单独的 l,s 组成，每个都是一个关键词（有关更详细的讨论，请参见第 ?? 页），因此可以在后面跟随可选空格。因此，形如fil L l 的形式也是有效的。这可能会导致奇怪的错误。在大多数情况下，添加一个 \relax 记号可以防止这种意外发生。
- The primitive command may come in handy as the final command in a macro definition. As it gobbles up optional spaces, it can be used to prevent spaces following the closing brace of an argument from winding up in the output inadvertently. For example, in
  原始命令 可以作为宏定义的最后一个命令非常有用。由于它会吞掉可选空格，因此可以用于防止参数的右花括号后面的空格无意中出现在输出

中。例如，在以下代码中：

```
\def\item#1{\par\leavevmode
   \llap{#1\enspace}\ignorespaces}
\item{a/}one line \item{b/} another line \item{c/}
yet another
the \ignorespaces prevents spurious spaces in the second and third item.
An empty line after \ignorespaces will still insert a \par, however.
```

\ignorespaces 防止第二个和第三个条目中出现不必要的空格。但是，在 \ignorespaces 之后的空行仍然会插入一个 \par。

2.10.3 Ignored and obeyed spaces 忽略和保留的空格

After control words spaces are ignored. This is not an instance of optional spaces, but it is due to the fact that TEX goes into state S, skipping spaces, after control words. Similarly an end-of-line character is skipped after a control word.

在控制词之后，空格会被忽略。这不是可选空格的例子，而是因为在控制词之后，TEX 会进入状态 S，跳过空格。类似地，在控制词之后，换行符也会被跳过。

Numbers are delimited by only ⟨one optional space⟩, but still

数字由 ⟨ 一个可选空格 ⟩ 分隔，但仍然

```
a\count0=3 b
```

gives 得到 'ab',

because TEX goes into state S after the first space token. The second space is therefore skipped in the input processor of TEX; it never becomes a space token.

这是因为在第一个空格记号之后，TEX 会进入状态 S。因此，在 TEX 的输入处理器中，第二个空格被跳过；它永远不会变成一个空格记号。

Spaces are skipped furthermore when TEX is in state N, newline. When TEX is processing in vertical mode space tokens (that is, spaces that were not skipped) are ignored. For example, the space inserted (because of the line end) after the first box in

当 TEX 处于状态 N，即换行状态时，空格还会被跳过。当 TEX 在垂直模式下进行处理时，空格记号（即未被跳过的空格）会被忽略。例如，在以下代码中，在第一个盒子后插入的空格（由于行尾）

```
\par
\hbox{a}
\hbox{b}
```

has no effect.

不会产生效果。

Both plain TEX and LATEX define a command \obeyspaces that makes spaces significant: after one space other spaces are no longer ignored. In both cases the basis is

无论是 plain TEX 还是 LATEX 都定义了一个命令 \obeyspaces，使空格成为有效字符：在一个空格后，其他空格不再被忽略。在两种情况下，基础定义如下：

```
\catcode`\ =13 \def {\space}
```

However, there is a difference between the two cases: in plain TEX

然而，这两种情况之间存在一些差异：在 plain TEX 中，

```
\def\space{ }
```

while in LATEX

而在 LATEX 中，

```
\def\space{\leavevmode{} }
```

although the macros bear other names there.

尽管宏的名称不同。

The difference between the two macros becomes apparent in the context of \obeylines: each line end is then a \par command, implying that each next line is started in vertical mode. An active space is expanded by the plain macro to a space token, which is ignored in vertical mode. The active spaces in LATEX will immediately switch to horizontal mode, so that each space is significant.

这两个宏之间的差异在 \obeylines 的环境中变得明显：每个行尾都是一个 \par 命令，意味着下一行都是在垂直模式下开始的。plain 宏会将活动空格展开为一

个空格记号，在垂直模式下被忽略。而 LaTeX 中的活动空格会立即切换到水平模式，使得每个空格都是有效的。

2.10.4 More ignored spaces 更多被忽略的空格

There are three further places where TeX will ignore space tokens.

还有三个地方，TeX 会忽略空格记号。

1. When TeX is looking for an undelimited macro argument it will accept the first token (or group) that is not a space. This is treated in Chapter ??.
   当 TeX 查找未定界的宏参数时，它将接受第一个不是空格的记号（或组）。这将在第 ?? 章中介绍。
2. In math mode space tokens are ignored (see Chapter ??).
   在数学模式中，空格记号会被忽略（详见第 ?? 章）。
3. After an alignment tab character spaces are ignored (see Chapter ??).
   在对齐制表符后的空格会被忽略（详见第 ?? 章）。

2.10.5 ⟨space token⟩

Spaces are anomalous in TeX. For instance, the \string operation assigns category code 12 to all characters except spaces; they receive category 10. Also, as was said above, TeX's input processor converts (when in state M) all tokens with category code 10 into real spaces: they get character code 32. Any character token with category 10 is called ⟨space token⟩. Space tokens with character code not equal to 32 are called 'funny spaces' .

在 TeX 中，空格是异常的。例如，\string 操作会将除空格以外的所有字符的类别码设置为 12；空格的类别码为 10。此外，正如前面所述，TeX 的输入处理器（当处于状态 M 时）会将所有类别码为 10 的记号转换为真实的空格：它们的字符码为 32。任何类别码为 10 的字符记号被称为 ⟨space token⟩。字符码不等于 32 的空格记号被称为“有趣的空格”。

> After giving the character Q the category code of a space character, and using it in a definition

在给字符 Q 设置空格字符的类别码，并在定义中使用它的情况下，
\catcode`Q=10 \def\q{aQb}
we get 我们会得到
\show\q
macro:-> a b
because the input processor changes the character code of the funny space in the definition.
因为输入处理器会改变定义中有趣空格的字符码。

Space tokens with character codes other than 32 can be created using, for instance, \uppercase. However, 'since the various forms of space tokens are almost identical in behaviour, there's no point dwelling on the details'; see [?] p. 377.

可以使用 \uppercase 等命令创建字符码不是 32 的空格记号。然而，"由于各种形式的空格记号在行为上几乎相同，没有必要深入研究细节"；详见 [?] p. 377。

2.10.6 Control space
控制空格

The 'control space' command \ contributes the amount of space that a ⟨space token⟩ would when the \spacefactor is 1000. A control space is not treated like a space token, or like a macro expanding to one (which is how \space is defined in plain TEX). For instance, TEX ignores spaces at the beginning of an input line, but control space is a ⟨horizontal command⟩, so it makes TEX switch from vertical to horizontal mode (and insert an indentation box). See Chapter ?? for the space factor, and chapter ?? for horizontal and vertical modes.

'控制空格' 命令 \ 作为一个 ⟨space token⟩ 时，会在 \spacefactor 为 1000 时产生相同量的空格。控制空格不会像空格记号或扩展为一个空格的宏那样对待（这是 plain TEX 中 \space 的定义方式）。例如，TEX 忽略输入行开头的空格，但是控制空格是一个 ⟨horizontal command⟩，因此它会使 TEX 从垂直模式切换到水平模式（并插入一个缩进盒子）。有关空格因子，请参阅第 ?? 章；关于水平和垂直模式，请参阅第 ?? 章。

2.10.7 ‘ ’

The explicit symbol ‘ ’ for a space is character 32 in the Computer Modern typewriter typeface. However, switching to \tt is not sufficient to get spaces denoted this way, because spaces will still receive special treatment in the input processor.

显式符号‘ ’表示一个空格，它在计算机现代打字机字体中是字符 32。然而，仅切换到 \tt 是不足以得到以这种方式表示的空格的，因为空格仍然会在输入处理器中接受特殊处理。

One way to let spaces be typeset by ␣ is to set

让空格以 ␣ 的方式排版的一种方法是设置

\catcode`\ =12

TEX will then take a space as the instruction to typeset character number 32. Moreover, subsequent spaces are not skipped, but also typeset this way: state S is only entered after a character with category code 10. Similarly, spaces after a control sequence are made visible by changing the category code of the space character.

然后，TEX 将会将空格解释为排版字符编号为 32 的指令。此外，后续的空格也不会被跳过，而是以这种方式进行排版：只有在具有类别码为 10 的字符之后才会进入状态 S。类似地，更改空格字符的类别码可以使控制序列后面的空格可见。

2.11 More about line ends
更多关于行结束符

TEX accepts lines from an input file, excluding any line terminator that may be used. Because of this, TEX's behaviour here is not dependent on the operating system and the line terminator it uses (CR-LF, LF, or none at all for block storage). From the input line any trailing spaces are removed. The reason for this is historic; it has to do with the block storage mode on IBM mainframe

computers. For some computer-specific problems with end-of-line characters, see [?].

TEX 从输入文件中接受行，不包括可能使用的行终止符。因此，在这里，TEX 的行为不依赖于操作系统及其使用的行终止符（CR-LF、LF 或没有终止符用于块存储）。从输入行中删除任何尾随的空格。这样做的原因是历史原因；它与 IBM 大型计算机上的块存储模式有关。有关行结束字符的一些特定于计算机的问题，请参阅 [?]。

A terminator character is then appended with a character code of \endlinechar, unless this parameter has a value that is negative or more than 255. Note that this terminator character need not have category code 5, end of line.

然后，追加一个字符代码为 \endlinechar 的终止符字符，除非此参数的值为负数或大于 255。请注意，这个终止符字符不一定要具有类别码 5，即行结束符。

2.11.1　Obeylines

Every once in a while it is desirable that the line ends in the input correspond to those in the output. The following piece of code does the trick:

有时希望输入中的行结束符与输出中的行结束符相对应。以下代码片段可以实现这个目的：

```
\catcode`\^^M=13 %
\def^^M{\par}%
```

The \endlinechar character is here made active, and its meaning becomes \par. The comment signs prevent TEX from seeing the terminator of the lines of this definition, and expanding it since it is active.

这里使 \endlinechar 字符成为活动字符，并将其含义设置为 \par。注释符号防止 TEX 看到此定义的行终止符，并展开它，因为它是活动的。

However, it takes some care to embed this code in a macro. The definition

然而，将这段代码嵌入宏中需要一些小心。以下定义：

```
\def\obeylines{\catcode`\^^M=13 \def^^M{\par}}
```

will be misunderstood: TEX will discard everything after the second ^^M, because this has category code 5. Effectively, this line is then

会被误解：TEX 会丢弃第二个 ^^M 后面的所有内容，因为它的类别码是 5。实际上，这一行实际上是：

\def\obeylines{\catcode`\^^M=13 \def

To remedy this, the definition itself has to be performed in a context where ^^M is an active character:

为了解决这个问题，定义本身必须在^^M是活动字符的上下文中执行：

```
{\catcode`\^^M=13 %
 \gdef\obeylines{\catcode`\^^M=13 \def^^M{\par}}%
}
```

Empty lines in the input are not taken into account in this definition: these disappear, because two consecutive \par tokens are (in this case) equivalent to one. A slightly modified definition for the line end as

在这个定义中，输入中的空行不会被考虑在内：它们会消失，因为两个连续的 \par 记号（在此情况下）等效于一个。对于行尾的稍作修改的定义如下：

\def^^M{\par\leavevmode}

remedies this: now every line end forces TEX to start a paragraph. For empty lines this will then be an empty paragraph.

通过这样修改，现在每个行尾都会强制 TEX 开始一个段落。对于空行，这将变成一个空段落。

2.11.2 Changing the \endlinechar 改变 \endlinechar

Occasionally you may want to change the \endlinechar, or the \catcode of the ordinary line terminator ^^M, for instance to obtain special effects such as macros where the argument is terminated by the line end. See page ?? for a worked-out example.

偶尔您可能想要改变 \endlinechar 或普通行终止符 ^^M 的 \catcode，例如为了获得特殊效果，比如宏的参数以行尾为终止符。有关详细示例，请参见第 ?? 页。

There are a couple of traps. Consider the following:

这里有一些陷阱。考虑以下代码：

```
{\catcode`\^^M=12 \endlinechar=`\^^J \catcode`\^^J=5
...
... }
```

This causes unintended output of both character 13 (^^M) and 10 (^^J), caused by the line terminators of the first and last line.

它会导致意外输出字符 13（^^M）和字符 10（^^J），这是由于第一行和最后一行的行终止符造成的。

Terminating the first and last line with a comment works, but replacing the first line by the two lines

用注释来终止第一行和最后一行可以解决这个问题，但是将第一行替换为以下两行

```
{\endlinechar=`\^^J \catcode`\^^J=5
\catcode`\^^M=12
```

is also a solution.

也是一种解决方案。

Of course, in many cases it is not necessary to substitute another end-of-line character; a much simpler solution is then to put

当然，在许多情况下，不需要替换另一个行终止符；更简单的解决方案是使用：

```
\endlinechar=-1
```

which treats all lines as if they end with a comment.

这样会将所有行都视为以注释结尾。

2.11.3 More remarks about the end-of-line character 关于行终止符的更多说明

The character that TEX appends at the end of an input line is treated like any other character. Usually one is not aware of this, as its category code is special, but there are a few ways to let it be processed in an unusual way.

TEX 在输入行的末尾附加的字符被视为任何其他字符。通常情况下，我们对此并不感知，因为它的类别码是特殊的。但是有几种方法可以让它以不同寻常的方式进行处理。

> Terminating an input line with ^^ will (ordinarily, when \endlinechar is 13) give 'M' in the output, which is the ascii character with code 13+64.
> 当以 ^^ 结尾的输入行时（通常情况下，当 \endlinechar 为 13 时），输出中会得到 'M'，它是具有代码 13+64 的 ascii 字符。
>
> If \^^M has been defined, terminating an input line with a backslash will execute this command. The plain format defines
> 如果已经定义了 ^^M，以反斜杠结尾的输入行将执行此命令。plain 格式定义了
> \def\^^M{\ }
> which makes a 'control return' equivalent to a control space.
> 这将使“控制回车”等效于控制空格。

2.12 More about the input processor
关于输入处理器的更多内容

2.12.1 The input processor as a separate process
将输入处理器视为独立的过程

TEX's levels of processing are all working at the same time and incrementally, but conceptually they can often be considered to be separate processes that each accept the completed output of the previous stage. The juggling with spaces provides a nice illustration for this.

TEX 的处理层级是同时进行且逐步进行的，但在概念上，它们通常可以被视为独立的过程，每个过程接受前一阶段的完成输出。对于空格处理，这提供了一个很好的说明。

Consider the definition

考虑以下定义：

\def\DoAssign{\count42=800}

and the call 以及调用：

\DoAssign 0

The input processor, the part of TEX that builds tokens, in scanning this call skips the space before the zero, so the expansion of this call is

输入处理器，即构建记号的 TEX 的一部分，在扫描此调用时，会跳过零前面的空格，因此此调用的展开结果是：

\count42=8000

It would be incorrect to reason '\DoAssign is read, then expanded, the space delimits the number 800, so 800 is assigned and the zero is printed'. Note that the same would happen if the zero appeared on the next line.

推理“读取了 \DoAssign，然后展开，空格分隔了数字 800，因此分配了 800，并打印了零”是不正确的。请注意，如果零出现在下一行上，情况也是一样的。

Another illustration shows that optional spaces appear in a different stage of processing from that for skipped spaces:

另一个例子显示了可选空格和跳过空格出现在不同的处理阶段：

```
\def\c.{\relax}
a\c. b
```

expands to 展开为：

```
a\relax b
```

which gives as output

'a b'

because spaces after the \relax control sequence are only skipped when the line is first read, not when it is expanded. The fragment

输出结果为

a b'

，因为在 \relax 控制序列后的空格只有在首次读取该行时才会被跳过，而不是在展开时。另一方面，片段：

```
\def\c.{\ignorespaces}
a\c. b
```

on the other hand, expands to

a\ignorespaces b

Executing the \ignorespaces command removes the subsequent space token, so the output is

执行 \ignorespaces 命令会移除后续的空格记号，因此输出结果为

'ab'.

In both definitions the period after \c is a delimiting token; it is used here to prevent spaces from being skipped.

在这两个定义中，\c 后面的句点是一个定界记号；在这里，它用于防止空格被跳过。

2.12.2 The input processor not as a separate process 将输入处理器视为一个独立的过程

Considering the tokenizing of TEX to be a separate process is a convenient view, but sometimes it leads to confusion. The line

将 TEX 的记号化过程视为一个独立的过程是一种方便的观点，但有时会导致混淆。例如，下面的代码行：

```
\catcode`\^^M=13{}
```

makes the line end active, and subsequently gives an 'undefined control sequence' error for the line end of this line itself. Execution of the commands on the line thus influences the scanning process of that same line.

将行结束符设为活动字符，并随后在该行的行结束符处报告“未定义的控制序列”错误。因此，该行上的命令执行会影响该行的扫描过程。

By contrast,

相比之下，以下代码行：

```
\catcode`\^^M=13
```

does not give an error. The reason for this is that TEX reads the line end while it is still scanning the number 13; that is, at a time when the assignment has

not been performed yet. The line end is then converted to the optional space character delimiting the number to be assigned.

不会报错。原因是 TEX 在扫描数字 13 时读取行结束符；也就是说，在尚未执行赋值之前，TEX 就读取了行结束符。然后，行结束符被转换为可选空格字符，用于界定要赋值的数字。

2.12.3 Recursive invocation of the input processor 输入处理器的递归调用

Above, the activity of replacing a parameter character plus a digit by a parameter token was described as something similar to the lumping together of letters into a control sequence token. Reality is somewhat more complicated than this. TEX's token scanning mechanism is invoked both for input from file and for input from lists of tokens such as the macro definition. Only in the first case is the terminology of internal states applicable.

前面提到，将参数字符和数字组合替换为参数记号类似于将字母组合成控制序列记号。然而，现实情况比这更复杂。TEX 的记号扫描机制既适用于从文件中读取的输入，也适用于从记号列表（如宏定义）中读取的输入。只有在第一种情况下，内部状态的术语才适用。

Macro parameter characters are treated the same in both cases, however. If this were not the case it would not be possible to write things such as

然而，无论在哪种情况下，宏参数字符的处理方式都是相同的。如果情况不是这样的话，就无法编写以下代码：

```
\def\a{\def\b{\def\c####1{####1}}}
```

See page ?? for an explanation of such nested definitions.

有关此类嵌套定义的解释，请参见第 ?? 页。

2.13 The @ convention

@ 约定

Anyone who has ever browsed through either the plain format or the LaTeX format will have noticed that a lot of control sequences contain an 'at' sign: @. These are control sequences that are meant to be inaccessible to the ordinary user.

浏览过 plain 格式或 LaTeX 格式的人会注意到，许多控制序列包含一个“at”符号：@。这些控制序列是为普通用户不可访问的。

Near the beginning of the format files the instruction

在格式文件的开头附近，出现了以下指令：

```
\catcode`@=11
```

occurs, making the at sign into a letter, meaning that it can be used in control sequences. Somewhere near the end of the format definition the at sign is made 'other' again:

将 at 符号变为字母，这意味着它可以用于控制序列。在格式定义的末尾附近，at 符号被再次设为“其他”：

```
\catcode`@=12
```

Now why is it that users cannot call a control sequence with an at sign directly, although they can call macros that contain lots of those 'at-definitions'? The reason is that the control sequences containing an @ are internalized by TeX at definition time, after which they are a token, not a string of characters. Macro expansion then just inserts such tokens, and at that time the category codes of the constituent characters do not matter any more.

那么为什么用户不能直接调用带有 at 符号的控制序列，尽管他们可以调用包含许多这些“at-定义”的宏？原因是在定义时，带有 @ 的控制序列被 TeX 内部化为一个记号，而不是一个字符序列。宏展开只是插入这样的记号，此时组成字符的类别码不再重要。