第三章

词法结构

本章说明了 Java 编程语言的词法结构。程序是用 Unicode(§3.1)编写的，但提供了词法翻译(§3.2)，因此 Unicode 转义(§3.3)可以用来包含任何只使用 ASCII 字符的 Unicode 字符。行终止符(§3.4)被定义用来支持现有主机系统的不同约定，同时保持一致的行号。

由词法翻译产生的 Unicode 字符被简化为一系列输入元素(§3.5)，其中包括空格(§3.6)、注释(§3.7)和标记。标记是句法语法中的标识符(§3.8)、关键字(§3.9)、字面量(§3.10)、分隔符(§3.11)和操作符(§3.12)。

3.1 Unicode（统一字符编码标准）

程序使用 Unicode 字符集(§1.7)编写。关于这个字符集及其相关字符编码的信息可以在 https:// www.unicode.org/找到。Java SE 平台跟踪 Unicode 标准的发展。

给定版本所使用的 Unicode 字符集的精确版本在类 Character 的文档中说明。

> JDK 1.1 之前的 Java 编程语言版本使用 Unicode 1.1.5。对 Unicode 标准的更新版本包括:JDK 1.1(至 Unicode 2.0)、JDK 1.1.7(至 Unicode 2.1)、Java SE 1.4(至 Unicode 3.0)、Java SE 5.0(至 Unicode 4.0)、Java SE 7(至 Unicode 6.0)、Java SE 8(至 Unicode 6.2)、Java SE 9(至 Unicode 8.0)、Java SE 11(至 Unicode 10.0)、Java SE 12(至 Unicode 11.0)、Java SE 13(至 Unicode 12.1)和 Java SE 15(至 Unicode 13.0)。

Unicode 标准最初设计为固定宽度的 16 位字符编码。此后，它被修改为允许表示需要超过 16 位的字符。合法代码点的范围现在是 U+0000 到 U+10FFFF，使用十六进制的 U+n 表示法。码点值大于 U+FFFF 的字符称为补充字符。为了仅使用 16 位单元表示所有字符，Unicode 标准定义了一种称为 UTF-16 的编码。在这种编码中，补充字符表示为 16 位代码单元对，第一个来自高代理范围（U+D800 到 U+DBFF），第二个来自低代理范围（U+DC00 到 U+DFFF）。对于 U+0000 ~ U+FFFF 范围内的字符，码点值与 UTF-16 编码单元值相同。

Java 编程语言使用 UTF-16 编码，以 16 位代码单元序列表示文本。

> Java SE 平台的一些 api(主要在 Character 类中)使用 32 位整数将代码点表示为单个实体。 Java SE 平台提供了在 16 位和 32 位表示之间转换的方法。

该规范在表示相关的地方使用术语码点和 UTF-16 代码单元，在表示与讨论无关的地方使用通用术语字符。除了注释(§3.7)、标识符(§3.8)、字符字面量、字符串字面量和文本块(§3.10.4、§3.10.5、§3.10.6)的内容外，程序中所有的输入元素(§3.5)都只能由 ASCII 字符(或 Unicode 转义(§3.3)得到 ASCII 字符)组成。

> ASCII (ANSI X3.4)是美国信息交换标准代码。Unicode UTF-16 编码的前 128 个字符是 ASCII 字符。

3.2 词法转换

使用以下三个词法转换步骤，将原始 Unicode 字符流转换为一系列标记，这三个步骤依次应用:

1. 将原始 Unicode 字符流中的 Unicode 转义(§3.3)转换为相应的 Unicode 字符。Unicode 转义形式\uxxxx，其中 xxxx 是一个十六进制值，表示编码为 xxxx 的 UTF-16 代码单元。这个转换步骤允许只使用 ASCII 字符来表示任何程序。

2. 将步骤 1 产生的 Unicode 流转换为输入字符和行终止符流(§3.4)。

3. 将第 2 步产生的输入字符流和行终止符转换为一系列输入元素(§3.5)，这些元素在空格(§3.6)和注释(§3.7)被丢弃之后组成标记，它们是语法的终止符号(§2.3)。

每一步都使用尽可能长的转换，即使结果最终不能生成正确的程序，而另一个词法转换可以。对于需要更细粒度翻译的情况，有两个例外:在步骤 1 中，用于处理连续的\字符(§3.3)，在步骤 3 中，用于处理上下文关键字和相邻的>字符(§3.5)。

> 输入字符 a--b 被标记为 a，--，b，它不是任何语法正确程序的一部分，即使标记 a， -， -， b 可能是语法正确程序的一部分。
>
> 可以假设原始输入\\u1234 被翻译成\字符和(遵循"最长可能"规则)一个 Unicode 转义形式的\u1234。事实上，前导\字符导致这个原始输入被翻译成七个不同的字符:\ \ u 1 2 3 4。

3.3 Unicode 转义

Java 编程语言的编译器("Java 编译器")首先识别原始输入中的 Unicode 转义，将 ASCII 字符 u 后面的四个十六进制数字转换为表示所指示的十六进制值的 UTF-16 代码单元(§3.1)的原始输入字符。一种 Unicode 转义可以表示 U+0000 到 U+FFFF 范围

内的字符;表示 U+010000 到 U+10FFFF 范围内的补充字符需要连续两次 Unicode 转义。 编译器原始输入中的所有其他字符都被识别为原始输入字符，并不加更改地传递。

这个转换步骤产生一个 Unicode 输入字符序列，所有这些字符都是原始输入字符(任何 Unicode 转义都被简化为原始输入字符)。

UnicodeInputCharacter:
  UnicodeEscape
  RawInputCharacter

UnicodeEscape:
  \ *UnicodeMarker HexDigit HexDigit HexDigit HexDigit*

UnicodeMarker:
  u *{u}*

HexDigit:
  (one of)
  0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

RawInputCharacter:
  any Unicode character

这里的\、u 和十六进制数字是所有的 ASCII 字符。

UnicodeInputCharacter 产生是不明确的，因为编译器原始输入中的 ASCII \字符可以被简化为 RawInputCharacter 或 UnicodeEscape 的\(后面要跟一个 ASCII u)。为了避免歧义，对于编译器原始输入中的每个 ASCII \字符，输入处理必须考虑这个转换步骤产生的最近的原始输入字符:

- 如果结果中最近的原始输入字符本身是从编译器原始输入中的 Unicode 转义转换而来的，那么 ASCII \字符就有资格开始 Unicode 转义。

  例如，如果结果中最近的原始输入字符是由原始输入中的 Unicode 转义\u005c 产生的反斜杠，那么原始输入中下一个出现的 ASCII \字符就有资格开始另一个 Unicode 转义。

- 否则，考虑有多少反斜杠连续地作为原始输入字符出现在结果中，返回到非反斜杠字符或结果的开始。(这样的反斜杠是来自于编译器原始输入中的 ASCII \字符，还是来自于编译器原始输入中的 Unicode 转义\u005c，都无关紧要。) 如果这个数字是偶数，那么 ASCII \字符可以开始 Unicode 转义; 如果数字是奇数，那么 ASCII \字符就不适合开始 Unicode 转义。

例如，原始输入“\\u2122=\u2122”会产生 11 个字符“\\u2122=™”，因为原始输入中的第二个 ASCII \字符不适合开始 Unicode 转义，而第三个 ASCII \字符是合适的，而\u2122 是字符™的 Unicode 编码。

如果一个符合条件的\后面没有 u，那么它将被视为一个 RawInputCharacter，并且仍然是转义的 Unicode 流的一部分。

如果一个符合条件的\后跟 u，或者一个以上的 u，并且最后一个 u 后面没有四个十六进制数字，那么就会发生编译错误。

由 Unicode 转义产生的字符不参与进一步的 Unicode 转义。

例如，原始输入\u005cu005a 会产生六个字符\u005 a，因为 005c 是反斜杠的 Unicode 值。 它不会产生字符 Z，即 Unicode 值 005a，因为处理 Unicode 转义\u005c 所产生的反斜杠不会被解释为进一步 Unicode 转义的开始。

请注意，\u005cu005a 不能用字符串字面量来表示六个字符\u 0 0 5 a。这是因为翻译产生的前两个字符，\和 u 在字符串字面量中被解释为非法转义序列（§3.10.7）。

幸运的是，关于连续反斜杠字符的规则有助于程序员设计原始输入，以字符串字面量表示 Unicode 转义。 在字符串字面量中表示六个字符\u005a 只需要将另一个\放置在现有\的旁边，例如“\\u005a is Z”。这是可行的，因为原始输入\\u005a 中的第二个\没有资格开始 Unicode 转义，所以第一个\和第二个\被保留为原始输入字符，接下来的五个字符 u005a 也是如此。两个\字符随后在字符串字面量中解释为反斜杠的转义序列，产生一个包含所需的六个字符\ u 0 0 5 a 的字符串。如果没有该规则，原始输入\\u005a 将被处理为原始输入字符\，后面跟着一个 Unicode 转义\u005a，它将成为原始输入字符 Z; 这将是没有用的，因为\Z 是字符串字面量中的非法转义序列。(注意，该规则将\u005c\u005c 转换为\\，因为将第一个 Unicode 转义转换为原始输入字符\并不阻止将第二个 Unicode 转义转换为另一个原始输入字符\。)

该规则还允许程序员制作原始输入，在字符串字面量中表示转义序列。 例如，原始输入\\\u006e 会产生三个字符\\ n，因为第一个\和第二个\被保留为原始输入字符，而第三个\可以开始 Unicode 转义，因此\u006e 被转换为原始输入字符 n。三个字符\ \ n 随后在字符串字面量中解释为\ n，它表示换行的转义序列。 (注意\\\u006e 可以被写为\u005c\ u005c\u006e，因为每个 Unicode 转义\u005c 都被转换为原始输入字符\，因此剩余的原始输入\u006e 前面有偶数个反斜杠，并被处理为 n 的 Unicode 转义。)

Java 编程语言指定了一种将以 Unicode 编写的程序转换为 ASCII 的标准方法，该方法将程序转换为一种可由基于 ASCII 的工具处理的形式。 转换包括通过添加一个额外的 u(例如\uxxxx 变成\uxxxx)将程序源文本中的任何 Unicode 转义转换为 ASCII，同时将源文本中的非 ASCII 字符转换为每个包含单个 u 的 Unicode 转义。

这个转换后的版本同样可以被 Java 编译器接受，并且表示完全相同的程序。通过将每个出现多个 u 的转义序列转换为一个 u 更少的 Unicode 字符序列，同时将每个带有单个 u 的转义序列转换为相应的单个 Unicode 字符，可以稍后从这种 ASCII 形式

恢复确切的 Unicode 源。

当没有合适的字体时，Java 编译器应该使用\uxxxx 表示法作为输出格式来显示 Unicode 字符。

3.4 行终止符

然后，Java 编译器通过识别行终止符将 Unicode 输入字符序列分成行。

LineTerminator:
ASCII 的 LF 字符，也称为“换行符”。
ASCII CR 字符，也称为“回车符”
ASCII CR 字符后面跟着 ASCII LF 字符

InputCharacter:
UnicodeInputCharacter 但不是 **CR** 或 **LF**

行以 ASCII 字符 CR 或 LF 或 CR LF 结束。紧跟在 LF 后面的两个字符 CR 被视为一行终止符，而不是两行。

行终止符指定//形式的注释终止 (§3.7).

由行终止符定义的行可以确定行号，这是由 Java 编译器产生的。

结果是一系列行终止符和输入字符，它们是标记化过程中的第三步的结束符号。

3.5 输入元素和标记

由 Unicode 转义处理(§3.3)和输入行识别(§3.4)产生的输入字符和行终止符被简化为一个输入元素序列。

Input:
{InputElement} [Sub]

InputElement:
WhiteSpace
Comment
Token

Token:
Identifier
Keyword

Literal
Separator
Operator

Sub:
ASCII SUB 字符，也称为“control-Z”

那些不是空格或注释的输入元素是标记。标记是句法语法的终结符号(§2.3)。空白(§3.6)和注释(§3.7)可以用来分隔标记，如果相邻，可以用另一种方式标记。

例如，只有在没有插入空格或注释的情况下，输入字符-和=才能形成操作符标记-=(§3.12)。另一个例子是，10 个输入字符 staticvoid 形成一个标识符标记，而 11 个输入字符 static void(在 c 和 v 之间有一个 ASCII SP 字符)形成一对关键字标记，static 和 void，用空格分隔。

为了与某些操作系统兼容，如果 ASCII SUB 字符(\u001a 或 control-Z)是转义输入流中的最后一个字符，那么它将被忽略。

输入结果是不明确的，这意味着对于某些输入字符序列，有多种方法可以将输入字符简化为输入元素（即，标记输入字符）。歧义的解决方法如下：

- 可以缩减为标识符标记或文字标记的输入字符序列始终缩减为文字标记。
- 可以缩减为标识符标记或保留关键字标记（§3.9）的输入字符序列始终缩减为保留关键字标记。
- 可缩减为上下文关键字标记或其他（非关键字）标记的输入字符序列根据上下文缩减，如§3.9 所述。
- 如果输入字符>出现在类型上下文中（§4.11），即作为句法语法中的类型或非关联类型的一部分（§4.1，§8.3），则它始终被简化为数值比较运算符>，即使它可以与相邻的>字符组合以形成不同的运算符。

  如果没有对>字符的此规则，则类型 List<List<String>>中的两个连续>括号将标记为有符号右移运算符>>，而类型（如 List<List<List<String>>>）中的三个连续>括号将标记为无符号右移运算符>>>。更糟糕的是，类型中四个或更多连续的>括号的标记化，例如 List<List<List<List<String>>>>将是不明确的，因为>、>>和>>>标记的各种组合可能表示>>>>字符。

考虑结果输入流中的两个标记 x 和 y。如果 x 在 y 之前，那么我们说 x 在 y 的左边或者 y 在 x 的右边。

例如，在这段简单的代码中：

```
class Empty { }
```

我们说}标记在{标记的右边，尽管在这个二维表示中，它出现在{标记的左边和下面。使用“左”和

“右”这两个词的约定允许我们谈论，例如二元运算符的右操作数或赋值运算符的左操作数。

3.6 空格

空格被定义为 ASCII 空格字符、水平制表符、换行符和行结束符(§3.4)。

WhiteSpace:

ASCII 的 SP 字符，也称为“空格”。
ASCII HT 字符，也称为“水平制表符”。
ASCII FF 字符，也被称为“换行符”。

3.7 注释

有两种类型的注释：

- `/*` *text* `*/`

  传统注释: 从 ASCII 字符/*到 ASCII 字符*/的所有文字被忽略 (和 C 和 C++一样).

- `//` *text*

  行末尾注释: 从 ASCII 字符//到行末尾的所有文字被忽略 (和 C++一样).

Comment:
TraditionalComment
EndOfLineComment

TraditionalComment:
`/ *` *CommentTail*

CommentTail:
`*` *CommentTailStar*
NotStar CommentTail

CommentTailStar:
`/`
`*` *CommentTailStar*
NotStarNotSlash CommentTail

NotStar:
InputCharacter but not `*`
LineTerminator

NotStarNotSlash:

InputCharacter but not * or /
LineTerminator

EndOfLineComment:
* / *{InputCharacter}*

这些产品包含以下所有属性:

- 注释不支持嵌套。
- 在以//开头的注释里，/*和*/没有特别含义。
- 在以/*或者/**开头的注释里，//没有特别含义。

因此，以下文本是一条完整的注释:

```
/* this comment /* // /** ends here: */
```

词法语法暗示注释不会出现在字符字面量、字符串字面量或文本块中(§3.10.4、§3.10.5、§3.10.6)。

3.8 标识符

标识符是 Java 字母和 Java 数字的无限长序列，其中第一个字符必须是 Java 字母。

Identifier:
IdentifierChars but not a *ReservedKeyword* or *BooleanLiteral* or *NullLiteral*

IdentifierChars:
JavaLetter {JavaLetterOrDigit}

JavaLetter:
任何 Unicode 字符

JavaLetterOrDigit:
任何 Unicode 字符

一个“Java 字母”字符调用方法 Character.isJavaIdentifierStart (int)返回 true。

一个“Java 字母或数字”字符调用方法 Character.isJavaIdentifierPart(int)返回 true。

"Java 字母" 包括大写和小写 ASCII 拉丁字母 A-Z(\u0041- \u005a), 和 a-z (\u0061-\u007a), 因为历史原因，还包括 ASCII 美元符($, or \u0024) 和下划线 (_, or \u005f). 美元符号只应该在机器生成的源代码中使用，或者用于访问遗留系统上已存在的名称。下划线可以用于由两个或多个字符组成

的标识符，但由于是关键字，它不能用作一个字符的标识符。

"Java 数字" 包括 ASCII 数字 0-9 (\u0030-\u0039)。

字母和数字可以从整个 Unicode 字符集中提取，该字符集支持当今世界上使用的大多数书写脚本，包括中文、日语和韩语的大型字符集。 这允许程序员在他们的程序中使用用其本地语言编写的标识符。

两个标识符只有在忽略可忽略的字符后，对每个字母或数字具有相同的 Unicode 字符时才相同。可忽略字符调用方法 Character.isIdentifierIgnorable(int)返回 true. 具有相同外观的标识符可能是不同的。

例如，由单个字母组成的标识符拉丁大写字母 A (A， \u0041)，拉丁小写字母 A (A， \u0061)，希腊大写字母 ALPHA (A， \u0391)，西里尔字母小写字母 A (A， \u0430)和数学粗体斜体小写字母 A (A， \ud835\udc82)都是不同的。

Unicode 复合字符不同于它们的标准等效分解字符。例如，一个拉丁大写字母 A 重音符(区，\u00ci) 和拉丁大写字母 A (A, \u0041) 加上重音符(', \u0301) 不同（非间隔）. 请参阅 Unicode 标准，第 3.11 节“规范化形式”。

以下是一些标识符的例子:

- `String`
- `i3`
- αρετη
- `MAX_VALUE`
- `isLetterOrDigit`

由于标记化规则（§3.5），标识符的拼写（Unicode 字符序列）与保留关键字（§3.9）、布尔字面量（§3.10.3）或空字面量（§4.10.8）不同。 然而，标识符可能与上下文关键字具有相同的拼写，因为作为标识符或上下文关键字的输入字符序列的标记化取决于序列在程序中出现的位置。

为了方便上下文关键字的识别，句法语法(§2.3)有时通过定义一个产品只接受标识符的子集来禁止某些标识符。子集如下:

TypeIdentifier:
Identifier but not `permits`, `record`, `sealed`, `var`, or `yield`

UnqualifiedMethodIdentifier:

Identifier but not `yield`

TypeIdentifier 用于类、接口和类型参数的声明(§8.1、§9.1、§4.4)，以及引用类型(§6.5)。例如，类的名称必须是 TypeIdentifier，因此声明名为 permit、record、sealed、var 或 yield 的类是非法的。

UnqualifiedMethodIdentifier 在方法调用表达式通过简单名称引用方法时使用(§6.5.7.1)。由于术语 yield 被排除在 UnqualifiedMethodIdentifier 之外，任何名为 yield 的方法的调用都必须是限定的，从而将调用与 yield 语句区分开来(§14.21)。

3.9 关键字

由 ASCII 字符组成的 51 个字符序列被保留作为关键字使用，不能用作标识符(§3.8)。另外 16 个字符序列也由 ASCII 字符组成，可以被解释为关键字或其他标记，这取决于它们出现的上下文。

Keyword:
ReservedKeyword
ContextualKeyword

ReservedKeyword:
(one of)

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

_ (underscore)

ContextualKeyword:
(one of)

exports	opens	requires	uses
module	permits	sealed	var
non-sealed	provides	to	with
open	record	transitive	yield

关键字 const 和 goto 是保留的，即使它们目前没有被使用。如果这些 C++关键字不正确地出现在程序中，这可以让 Java 编译器产生更好的错误消息。

关键字 strictfp 已经过时，不应该在新代码中使用。

关键字_(下划线)被保留以供将来在参数声明中使用。

true 和 false 不是关键字，而是布尔字面量(§3.10.3)。

null 不是关键字，而是空字面量(§3.10.8)。

在将输入字符还原为输入元素(§3.5)的过程中，当且仅当以下两个条件同时满足时，在概念上与上下文关键字匹配的输入字符序列将被还原为上下文关键字:

1. 这个序列被认为是一个在适当的句法语法(§2.3)中指定的终端，如下所示:
   - 对于 module 和 open，当被识别为 ModuleDeclaration 中的一个终端时(§7.7)。
   - 对于 exports, opens, provides, requires, to, uses 和 with, 当被识别为 ModuleDirective 的一个终端时。
   - 对于 transitive, 当被识别为 *RequiresModifier* 的一个终端时。

     例如，识别序列 requires transitive; 并不使用 RequiresModifier, 所以 transitive 这个词在这里被简化为一个标识符，而不是上下文关键字。
   - 对于 var, 当被识别为 LocalVariableType (§14.4)或 LambdaParameterType (§15.27.1) 的一个终端时。

     在其他情况下，试图使用 var 作为标识符会导致错误，因为 var 不是 TypeIdentifier (§3.8)。
   - 对于 yield, 当被识别为 YieldStatement (§14.21) 的一个终端时。

     在其他情况下，试图使用 yield 作为标识符会导致错误，因为 yield 不是 TypeIdentifier 或 UnqualifiedMethodIdentifier。
   - 对于 record, 当被识别为 RecordDeclaration (§8.10) 的一个终端时。
   - 对于 non-sealed, permits 和 sealed, 当被识别为 NormalClassDeclaration (§8.1) 或 NormalInterfaceDeclaration (§9.1) 的一个终端时。
2. 序列的前面或后面没有匹配 JavaLetterOrDigit 的输入字符。

通常，由于“最长可能转换”规则(§3.2)，在源代码中意外省略空白将导致输入字符序列被标记为标识符。例如，十二个输入字符序列总是被标记为标识符 publicstatic，而不是保留关键字 public 和 static。如果要使用两个标记，它们必须用空格或注释分隔。

上述规则与“最长转换可能”规则一起工作，在可能出现上下文关键字的上下文中产生直观的结果。例如，11 个输入字符序列 varfilename 通常被标记为标识符 varfilename，但在局部变量声明中，前面三个输入字符被上述规则的第一个条件暂时识别为上下文关键字 var。但是，如果将接下来

的 8 个输入字符识别为标识符 filename，就会忽略序列中缺乏空白，这将是令人困惑的。(这意味着序列在不同的上下文中会经历不同的标记化:在大多数上下文中是一个标识符，但在局部变量声明中是一个上下文关键字和一个标识符。) 因此，第二个条件阻止识别上下文关键字 var，因为紧跟着的输入字符 f 是 JavaLetterOrDigit。因此，序列 v a r f i l e n a m e 在局部变量声明中被标记为标识符 varfilename。

作为仔细识别上下文关键字的另一个例子，考虑 15 个输入字符的序列 n o n - s e a l e d c l a s s. 这个序列通常被转换为三个标记—标识符 non、操作符-和标识符 sealedclass—但是在通常的类声明中，第一个条件保持不变，前十个输入字符暂时被识别为上下文关键字 non-sealed。为了避免将序列转换为两个关键字标记(non-sealed 和 class)而不是三个非关键字标记，并避免奖励程序员在 class 之前省略空白，第二个条件阻止识别上下文关键字。因此，序列 n o n - s e a l e d c l a s s 在一个类声明中被识别为三个标记。

在上面的规则中，第一个条件取决于语法的细节，但是 Java 编程语言的编译器可以在不完全解析输入程序的情况下实现该规则。 例如，启发式可以用于跟踪标记器的上下文状态，只要启发式保证上下文关键字的有效使用被标记为关键字，而标识符的有效使用被标记为标识符。另外，编译器也可以总是将上下文关键字标记为标识符，留待稍后阶段来识别这些标识符的特殊用途。

3.10 字面量

字面量是原生类型(§4.2)、String 类型(§4.3.3)或 null 类型(§4.1)值的源代码表示。

Literal:
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
TextBlock
NullLiteral

3.10.1 整数字面量

整数字面量可以用十进制(以 10 为基数)、十六进制(以 16 为基数)、八进制(以 8 为基数)或二进制(以 2 为基数)表示。

IntegerLiteral:
DecimalIntegerLiteral
HexIntegerLiteral
OctalIntegerLiteral
BinaryIntegerLiteral

DecimalIntegerLiteral:
DecimalNumeral [IntegerTypeSuffix]

HexIntegerLiteral:
HexNumeral [IntegerTypeSuffix]

OctalIntegerLiteral:
OctalNumeral [IntegerTypeSuffix]

BinaryIntegerLiteral:
BinaryNumeral [IntegerTypeSuffix]

IntegerTypeSuffix:
(one of)
`l L`

如果以 ASCII 字母 L 或 L (ell)作为后缀，整型字面量为 long 类型；否则为 int 类型(§4.2.1)。

推荐使用后缀 L，因为字母 L (ell)通常很难与数字 1 (one)区分开来。

允许使用下划线作为表示整数的数字之间的分隔符。

在十六进制或二进制字面量中，整数仅由 0x 或 0b 字符之后和任何类型后缀之前的数字表示。因此，下划线不能直接出现在 0x 或 0b 之后，也不能出现在字面量的最后一个数字之后。

在十进制或八进制字面值中，整数由字面量中任何类型后缀前的所有数字表示。因此，下划线不能出现在字面量的第一个数字之前或最后一个数字之后。下划线可以出现在八进制数字中的初始 0 之后(因为 0 是整型数的一部分)，也可以出现在非零十进制字面量中的初始非零数字之后。

十进制数字可以是单个 ASCII 数字 0，表示整数 0；也可以是一个 1 ~ 9 的 ASCII 数字后跟一个或多个 0 ~ 9 中间有下划线的 ASCII 数字，表示一个正整数。

DecimalNumeral:
`0`
NonZeroDigit [Digits]
NonZeroDigit Underscores Digits

NonZeroDigit:
(one of)
`1 2 3 4 5 6 7 8 9`

Digits:
Digit

Digit [DigitsAndUnderscores] Digit

Digit:
0
NonZeroDigit

DigitsAndUnderscores:
DigitOrUnderscore {DigitOrUnderscore}

DigitOrUnderscore:
Digit

Underscores:
_ *{_}*

十六进制数字是 ASCII 码前导字符 0x 或 0X 后面跟着一个或多个 ASCII 码十六进制数字，中间用下划线隔开，可以表示正整数、0 或负整数。

10 到 15 的十六进制数字分别用 ASCII 字母 a 到 f 或 A 到 F 表示；作为十六进制数字使用的每个字母可以是大写或小写。

HexNumeral:
0 x *HexDigits*
0 X *HexDigits*

HexDigits:
HexDigit
HexDigit [HexDigitsAndUnderscores] HexDigit

HexDigit:
(one of)
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

HexDigitsAndUnderscores:
HexDigitOrUnderscore {HexDigitOrUnderscore}

HexDigitOrUnderscore:
HexDigit

以上 HexDigit 产品来自§3.3。

八进制数字由一个 ASCII 数字 0 后面跟一个或多个 ASCII 数字 0 到 7 中间加上下划线组成，可以表示正整数、0 或负整数。

OctalNumeral:
`0` *OctalDigits*
`0` *Underscores OctalDigits*

OctalDigits:
OctalDigit
OctalDigit [OctalDigitsAndUnderscores] OctalDigit

OctalDigit:
(one of)
`0 1 2 3 4 5 6 7`

OctalDigitsAndUnderscores:
OctalDigitOrUnderscore {OctalDigitOrUnderscore}

OctalDigitOrUnderscore:
OctalDigit

注意，八进制数字总是由两个或两个以上的数字组成，因为 0 本身总是被认为是一个十进制数字——这在实践中并不重要，因为数字 0、00 和 0x0 都表示完全相同的整数值。

二进制数字由开头的 ASCII 字符 0b 或 0B 后面跟着一个或多个 ASCII 数字 0 或 1，中间用下划线隔开，可以表示正整数、0 或负整数。

BinaryNumeral:
`0 b` *BinaryDigits*
`0 B` *BinaryDigits*

BinaryDigits:
BinaryDigit
BinaryDigit [BinaryDigitsAndUnderscores] BinaryDigit

BinaryDigit:
(one of) `0 1`

BinaryDigitsAndUnderscores:
BinaryDigitOrUnderscore {BinaryDigitOrUnderscore}

BinaryDigitOrUnderscore:
BinaryDigit

int 类型的最大十进制值是 2147483648 (2^{31}).

从 0 到 2147483647 的所有十进制值都可以出现在整型字面量出现的任何地方。十进制值 2147483648 只能作为一元减法操作符-的操作数(§15.15.4)。

如果十进制值 2147483648 出现在除了一元减法操作符的操作数以外的任何位置，或者 int 类型的十进制值大于 2147483648(2^{31})，则会出现编译错误。

int 类型的最大的正整数的十六进制、八进制和二进制字面量——每一个都表示十进制值 2147483647(2^{31}-1)——分别是：

- `0x7fff_ffff`,
- `0177_7777_7777`, 和
- `0b0111_1111_1111_1111_1111_1111_1111_1111`

int 类型的最小的负整数的十六进制、八进制和二进制字面量——每一个都代表十进制值-2147483648(-2^{31})——分别是：

- `0x8000_0000`,
- `0200_0000_0000`, 和
- `0b1000_0000_0000_0000_0000_0000_0000_0000`

下面的十六进制、八进制和二进制表示十进制值-1：

- `0xffff_ffff`,
- `0377_7777_7777`, 和
- `0b1111_1111_1111_1111_1111_1111_1111_1111`

如果十六进制、八进制或二进制整型字面量不能用 32 位表示，则产生编译时错误。

long 类型的最大十进制字面量是 9223372036854775808L (2^{63})。

从 0L 到 9223372036854775807L 的所有十进制字面值都可以出现在长整型出现的任何地方。十进制字面量 9223372036854775808L 只能作为一元减号操作符-的操作数(§15.15.4)。

如果十进制值 `9223372036854775808L` 出现在除了一元减法操作符的操作数以外的任何位置，或者 long 类型的十进制值大于 `9223372036854775808L` (2^{63})，则会出现编译错误。

long 类型的最大的正整数的十六进制、八进制和二进制字面量——每一个都表示十

进制值 9223372036854775807L (2^{63}-1)——分别是:

- 0x7fff_ffff_ffff_ffffL,
- 07_7777_7777_7777_7777_7777L, and
- 0b0111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111L

long 类型的最小的负整数的十六进制、八进制和二进制字面量——每一个都代表十进制值-9223372036854775808L (-2^{63})——分别是:

- 0x8000_0000_0000_0000L, and
- 010_0000_0000_0000_0000_0000L, and
- 0b1000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000L

下面的十六进制、八进制和二进制表示十进制值-1L:

- 0xffff_ffff_ffff_ffffL,
- 017_7777_7777_7777_7777_7777L, and
- 0b1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111L

如果十六进制、八进制或二进制长整型字面量不能用 64 位表示，则产生编译时错误。

int 字面量的例子:

```
0 2 0372 0xDada_Cafe 1996 0x00_FF__00_FF
```

long 字面量的例子:

```
0l 0777L 0x100000000L 2_147_483_648L 0xC0B0L
```


3.10.2 浮点型字面量

浮点型字面量由以下部分组成：整数部分、十进制或十六进制点(由 ASCII 句点字符表示)、分数部分、指数和类型后缀。

浮点字面值可以表示为十进制(以 10 为基数)或十六进制(以 16 为基数)。

对于十进制浮点字面值，至少需要一个数字(整数部分或小数部分)和一个小数点、一个指数或一个浮点类型后缀。所有其他部件都是可选的。指数部分(如果存在)由 ASCII 字母 e 或 E 后跟一个可选的带符号整数表示。

对于十六进制浮点型字面量，至少需要一个数字(整数部分或小数部分)，指数部分是必须的，float 类型后缀是可选的。指数由 ASCII 字母 p 或 P 后跟一个可选的带符号整数表示。

在表示整数部分的数字之间、表示小数部分的数字之间以及表示指数的数字之间，

可以使用下划线作为分隔符。

FloatingPointLiteral:
  DecimalFloatingPointLiteral
  HexadecimalFloatingPointLiteral

DecimalFloatingPointLiteral:
  Digits `.` *[Digits] [ExponentPart] [FloatTypeSuffix]*
  `.` *Digits [ExponentPart] [FloatTypeSuffix]*
  Digits ExponentPart [FloatTypeSuffix]
  Digits [ExponentPart] FloatTypeSuffix

ExponentPart:
  ExponentIndicator SignedInteger

ExponentIndicator:
  (one of)
  `e E`

SignedInteger:
  [Sign] Digits

Sign:
  (one of)
  `+ -`

FloatTypeSuffix:
  (one of)
  `f F d D`

HexadecimalFloatingPointLiteral:
  HexSignificand BinaryExponent [FloatTypeSuffix]

HexSignificand:
  HexNumeral [.]
  `0 x` *[HexDigits]* `.` *HexDigits*
  `0 X` *[HexDigits]* `.` *HexDigits*

BinaryExponent:
  BinaryExponentIndicator SignedInteger

BinaryExponentIndicator:

(one of) `p P`

如果浮点型字面量以 ASCII 字母 F 或 f 作为后缀，则其类型为 float；否则，它的类型是 double，并且可以选择用 ASCII 字母 D 或 d 作为后缀。

float 和 double 类型的元素是分别可以使用 IEEE 754 binary32 和 IEEE 754 binary64 浮点格式表示的值(§4.2.3)。

> 从浮点数的 Unicode 字符串表示到内部的 IEEE 754 二进制浮点表示的正确输入转换的详细信息由包 java.lang 的 Float 类和 Double 类的 valueOf 方法描述。

float 类型的最大和最小正字面量如下:

- 最大的正有限浮点值在数值上等于 $(2 - 2^{-23}) \cdot 2^{127}$。

  舍入到这个值的最短的十进制字面值是 3.4028235e38f。

  这个值的十六进制字面量是 0x1.fffffeP+127f。

- 最小的正有限非零浮点值在数值上等于 2^{-149}。

  舍入到这个值的最短的十进制字面量是 1.4e-45f。

  这个值的两个十六进制字面量是 0x0.000002P-126f 和 0x1.0P-149f。

double 类型的最大和最小正字面量如下:

- 最大的正有限 double 值为$(2 - 2^{-52}) \cdot 2^{1023}$。

  舍入到这个值的最短的十进制字面量是 1.7976931348623157e308。

  这个值的十六进制字面量是 `0x1.f_ffff_ffff_ffffP+1023`。

- 最小的正有限非零 double 值为 2^{-1074}。

  舍入到这个值的最短的十进制字面量是 4.9e-324。

  这个值的两个十六进制字面量是 0x0.0_0000_0000_0001P-1022 和 0x1.0P-1074。

如果非零浮点字面值太大，则会导致编译时错误，因此在四舍五入转换为其内部表示时，它会变成 IEEE 754 无穷大。

程序可以通过使用常量表达式(如 1f/0f 或-1d/0d)或通过使用 Float 和 Double 类的预定义常量 POSITIVE_INFINITY 和 NEGATIVE_INFINITY 来表示无穷，而不会产生编译时错误。

如果非零浮点字面量太小，以致在四舍五入转换为其内部表示时变成 0，则为编译时错误。

如果非零浮点字面值具有一个小值，该值在四舍五入转换为其内部表示时变成非零次正常数，则不会发生编译时错误。

预定义的表示非数字值的常量定义在 Float 类和 Double 类中，它们是 Float.NaN 和 Double.NaN。

float 类型字面量的例子：

```
1e1f 2.f       .3f 0f     3.14f   6.022137e+23f
```

double 类型字面量的例子：

```
1e1     2.      .3      0.0       3.14  1e-9d    1e137
```

3.10.3 布尔值字面量

布尔类型有两个值，分别是由 ASCII 字母组成的布尔值 true 和 false。

BooleanLiteral:
(one of)
`true false`

布尔字面量总是布尔类型(§4.2.5)。

3.10.4 字符型字面量

字符字面量表示为字符或转义序列(§3.10.7)，用 ASCII 单引号括起来。(单引号或撇号字符是\u0027。)

CharacterLiteral:
' *SingleCharacter* '
' *EscapeSequence* '

SingleCharacter:
InputCharacter but not ' or \

字符字面量总是 char 类型的(§4.2.1)。

字符字面量的内容是开头'之后的 SingleCharacter 或 EscapeSequence。

如果内容后面的字符不是'，则产生编译时错误。

如果行结束符(§3.4)出现在开头'之后和结尾'之前，则产生编译时错误。

字符 CR 和 LF 从来不是 InputCharacter;每一个都被认为构成一个 LineTerminator，因此可能不会出现在字符字面量中，即使是在转义序列\ LineTerminator 中。

表示字符字面量的字符是解释了任何转义序列的字符字面量的内容，就像通过在内容上执行 String.translateEscapes 一样。

字符字面值只能表示 UTF-16 代码单元(§3.1)，也就是说，它们的值被限制在\u0000 到\uffff 之间。补充字符必须表示为字符序列中的代理项对，或表示为整数，这取决于使用它们的 API。

下面是字符字面量的例子：

- `'a'`
- `'%'`
- `'\t'`
- `'\\'`
- `'\''`
- `'\u03a9'`
- `'\uFFFF'`
- `'\177'`
- `'™'`

因为 Unicode 转义处理的非常早，对于一个值为换行（LF）的字符字面量来说，写入'\u000a'是不正确的；Unicode 转义\u000a 在转换步骤 1(§3.3)中被转换为实际的换行符，而换行符在步骤 2(§3.4)中成为 LineTerminator，因此字符字面量在步骤 3 中是无效的。相反，应该使用转义序列'\n'。类似地，对其值为回车(CR)的字符字面量写入'\u000d'也是不正确的。相反，应该使用'\r'。最后，不能为包含撇号(')的字符字面量写入'\u0027'。

在 C 和 C++中，一个字符字面量可以包含多个字符，但是这样一个字符字面量的值是实现定义的。在 Java 编程语言中，字符字面量总是恰好代表一个字符。

3.10.5 字符串字面量

字符串字面值由 0 个或多个用双引号括起来的字符组成。像换行这样的字符可以用转义序列来表示(§3.10.7)。

StringLiteral:
" *{StringCharacter}* "

StringCharacter:
InputCharacter but not " or \
EscapeSequence

字符串字面量始终是 String 类型(§4.3.3)。

字符串字面量的内容是字符序列，紧接在开头的"之后，紧接在匹配的结尾的"之前结束。

如果行结束符(§3.4)出现在开头的“和匹配的结尾符”之前，这是一个编译时错误。

字符 CR 和 LF 从来不是 InputCharacter;每一个都被认为构成一个 LineTerminator，因此可能不会出现在字符串字面量中，即使是转义序列\ LineTerminator 中。

字符串字面量表示的字符串是字符串字面量的内容，每个转义序列都被解释，就像在内容上执行 String.translateEscapes 一样

下面是字符串字面量的例子：

```
""        // the empty string
"\""      // a string containing " alone
"This is a string" // a string containing 16 characters
"This is a " + // actually a string-valued constant expression,
       "two-line string" // formed from two string literals
```

因为 Unicode 转义处理的非常早，对于一个值为换行（LF）的字符串字面量来说，写入'\u000a'是不正确的；Unicode 转义\u000a 在转换步骤 1(§3.3)中被转换为实际的换行符，而换行符在步骤 2(§3.4)中成为 LineTerminator，因此字符串字面量在步骤 3 中是无效的。相反，应该使用转义序列'\n'(§3.10.6)。类似地，对其值为回车(CR)的字符串字面量写入'\u000d'也是不正确的。相反，应该使用'\r'。最后，不能为包含双引号(")的字符串字面量写入'\u0027'。

一个较长的字符串字面量总是可以被分解成较短的部分，并使用字符串连接操作符+(§15.18.1)写成一个(可能带圆括号)表达式。

在运行时，字符串字面量是对 String 类(§4.3.3)实例的引用，该类表示由字符串字面量表示的字符串。

而且，字符串字面量总是指向 string 类的同一个实例。这是因为字符串字面量-或者更一般地说，是常量表达式的值(§15.29)——被“保留”了，以便共享唯一实例，就像通过执行 String.intern 方法(§12.5)一样。

例子 3.10.5-1. 字符串字面量

程序由编译单元(§7.3)组成：

```
package testPackage;
class Test {
   public static void main(String[] args) {
     String hello = "Hello", lo = "lo";
     System.out.println(hello == "Hello");
     System.out.println(Other.hello == hello);
     System.out.println(other.Other.hello == hello);
     System.out.println(hello == ("Hel"+"lo"));
     System.out.println(hello == ("Hel"+lo));
```

```
        System.out.println(hello == ("Hel"+lo).intern());
    }
}

class Other { static String hello = "Hello"; }
```

和编译单元：

```
package other;
public class Other {
public static String hello = "Hello"; }
```

产生如下输出：

```
true
true
true
true
false
true
```

这个例子说明了六个要点：

- 同一个类和包中的字符串字面量表示对同一个 String 对象的引用(§4.3.1)。
- 同一个包中不同类中的字符串字面量表示对同一个 String 对象的引用。
- 不同包中不同类中的字符串字面量同样表示对同一个 String 对象的引用。
- 由常量表达式(§15.29)连接而来的字符串在编译时被计算，然后被当作字面量来处理。
- 通过连接在运行时计算的字符串是新创建的，因此是不同的。
- 显式地保留一个计算字符串的结果与任何具有相同内容的预先存在的字符串字面量是相同的 string 对象。

- **3.10.6 文本块**

文本块由零个或多个字符组成，这些字符由开始和结束分隔符括起来。字符可以用转义序列表示(§3.10.7)，但必须用转义序列表示的换行符和双引号字符(§3.10.5)可以直接在文本块中表示。

TextBlock:
  " " " *{TextBlockWhiteSpace}* *LineTerminator* *{TextBlockCharacter}* " " "

TextBlockWhiteSpace:
  WhiteSpace but not *LineTerminator*

TextBlockCharacter:
  InputCharacter but not \

EscapeSequence LineTerminator

以下是§3.3、§3.4、§3.6 的作品，为方便起见,展示在此:

WhiteSpace:
ASCII 的 SP 字符，也称为“空格”。
ASCII HT 字符，也称为“水平制表符”。
ASCII 码的 FF 字符，也被称为“跳页”
LineTerminator

LineTerminator:
ASCII 的 LF 字符，也称为“换行符”。
ASCII CR 字符，也称为“回车符”
ASCII CR 字符后面跟着 ASCII LF 字符

InputCharacter:
UnicodeInputCharacter 但不是 CR 或 LF

UnicodeInputCharacter:
UnicodeEscape
RawInputCharacter

UnicodeEscape:
\ *UnicodeMarker HexDigit HexDigit HexDigit HexDigit*

RawInputCharacter: 任何一个 Unicode 字符

文本块总是 String 类型的(§4.3.3)。

开始分隔符是一个序列，它以三个双引号字符(""")开始，以零个或多个空格、制表符和换页符继续，并以行结束符结束。

结束分隔符是由三个双引号字符组成的序列。

文本块的内容是紧接在开始分隔符的行结束符之后开始，紧接在结束分隔符的第一个双引号之前结束的字符序列。

与字符串文字不同(§3.10.5)，在文本块的内容中出现行结束符并不是编译时错误。

例子 3.10.6-1. 文本块

当需要多行字符串时，文本块通常比串接的字符串字面值更容易读懂。例如，比较这些 HTML 片段的替代表示:

```
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, world</p>\n" +
              "    </body>\n" +
              "</html>\n";
```

```
String html ="""
              <html>
                  <body>
                                <p>Hello, world</p>
                  </body>
              </html>
              """
```

下面是一些文本块的例子:

```
   class Test {
     public static void main(String[] args) {
              // The six characters w i n t e r
              String season = """
                              winter""";
              // The seven characters w i n t e r LF
              String period = """
                              winter
                             """
              // The ten characters H i , SP " B o b " LF
              String greeting = """
                               Hi, "Bob"
                               """
              // The eleven characters H i , LF SP " B o b " LF
              String salutation = """
                                  Hi,
                                           "Bob"
                                           """
             // The empty string (zero length)
             String empty = """
                            """
             // The two characters " LF
             String quote = """
                            "
                           """
             // The two characters \ LF
             String backslash = """
                                \\
```

```
                """
        }
    }
```

在文本块中允许使用转义序列\n 和\"分别表示换行字符和双引号字符，尽管通常没有必要。例外情况是出现了三个连续的双引号字符，它们不是结束分隔符"""-在这种情况下，必须转义至少一个双引号字符，以避免模仿结束分隔符。

例子 3.10.6-2.文本块中的转义序列

在下面的程序中，如果单个双引号字符被转义，那么 story 变量的值将变得不那么易读:

```
class Story1 {
 public static void main(String[] args) {
    String story = """
         "When I use a word," Humpty Dumpty said, in rather a scornful
         tone, "it means just what I choose it to mean - neither more
         nor less." "The question is," said Alice, "whether you can make
         words mean so many different things." "The question is," said
         Humpty Dumpty, "which is to be master - that's all."
        ;""";
      }
    }
```

如果将程序修改为将结束分隔符放在内容的最后一行，则会发生错误，因为最后一行的前三个连续双引号字符被翻译(§3.2)为结束分隔符“”，因此会留下一个游离的双引号字符:

```
class Story2 {
 public static void main(String[] args) {
    String story = """
         "When I use a word," Humpty Dumpty said, in rather a scornful
         tone, "it means just what I choose it to mean - neither more
         nor less." "The question is," said Alice, "whether you can make
         words mean so many different things." "The question is," said
         Humpty Dumpty, "which is to be master - that's all.""";   //
         error
      }
 }
```

可以通过转义内容中的最后一个双引号字符来避免此错误:

```
class Story3 {
 public static void main(String[] args) {
    String story = """
         "When I use a word," Humpty Dumpty said, in rather a scornful
         tone, "it means just what I choose it to mean - neither more
         nor less." "The question is," said Alice, "whether you can make
         words mean so many different things." "The question is," said
         Humpty Dumpty,
         "which is to be master - that's all.\""";  // OK
         }
      }
```

如果一个文本块用来表示另一个文本块，那么建议转义嵌入的开、闭分隔符的第一个双引号字符:

```
class Code {
 public static void main(String[] args) {
      String text = """
        The quick brown fox jumps over the lazy dog
       """;

      String code =
           """
        String text = \"""
        The quick brown fox jumps over the lazy dog
        \""";
        """;
       }
      }
```

文本块表示的字符串不是内容中字符的字面量序列。相反，文本块表示的字符串是对内容按顺序应用以下转换的结果:

1. 行终止符被规范化为 ASCII 的 LF 字符，如下所示:
   - ASCII CR 字符后跟 ASCII LF 字符被转换为 ASCII LF 字符。
   - ASCII 的 CR 字符转换为 ASCII 的 LF 字符。
2. 附带的空白被删除，就像执行 String.stripIndent 对步骤 1 产生的字符进行缩进一样。
3. 转义序列被解释，就像对第 2 步产生的字符执行 String.translateEscapes 一样。

当这个规范说一个文本块包含一个特定的字符或字符序列，或者一个特定的字符或字符序列在一个文本块中，它意味着文本块所表示的字符串(与内容中字符的字面序列相反)包含字符或字符序列。

例子 3.10.6-3. 文本块内容的转换顺序

解释转义序列最后允许程序员使用\n、\f 和\r 对字符串进行垂直格式化而不影响行终止符的规范化，使用\b 和\t 对字符串进行水平格式化而不影响删除附带的空白。例如，考虑下面提到转义序列\r (CR)的文本块:

```
String html = """
   <html>\r
        <body>\r
                <p>Hello, world</p>\r
        </body>\r
   </html>\r
   """;
```

只有在行终止符规范化为 LF 之后，才解释\r 转义序列。 使用 Unicode 转义来可视

化 LF (\u000A)和 CR (\u000D)，使用|来可视化左距，文本块表示的字符串是:

```
|<html>\u000D\u000A
|  <body>\u000D\u000A
|  <p>Hello, world</p>\u000D\u000A
|  </body>\u000D\u000A
|</html>\u000D\u000A
```

在运行时，文本块是对类 String 实例的引用，该类表示文本块所表示的字符串。

此外，文本块总是引用 String 类的同一个实例。这是因为文本块所表示的字符串-或者更普遍地说，字符串是常量表达式的值(§15.29)-是被“保留”的，以便共享唯一实例，就像通过执行 String.intern 方法(§12.5)一样。

例子 3.10.6-4. 文本块计算为 String

文本块可以在任何允许 String 类型表达式的地方使用，例如在字符串连接(§15.18.1)中，在 String 实例的方法调用中，以及在带有 String 元素的注解中:

```
System.out.println("ab" + """
                         cde
                         """);

String cde = """
             abcde""".substring(2);

               String math = """
              1+1 equals \
              """ + String.valueOf(2);

               @Preconditions("""
    rate > 0 &&
    rate <= MAX_REFRESH_RATE
               """)
               public void setRefreshRate(int rate) {…}
```


3. 10. 7 转义序列

在字符字面量、字符串字面量和文本块(§3.10.4、§3.10.5、§3.10.6)中，转义序列允许不使用 Unicode 转义(§3.3)来表示一些非图形字符，以及单引号、双引号和反斜杠字符。

EscapeSequence:

- \ b (backspace BS, Unicode \u0008)
- \ s (space SP, Unicode \u0020)
- \ t (horizontal tab HT, Unicode \u0009)
- \ n (linefeed LF, Unicode \u000a)

\ `f` (form feed FF, Unicode `\u000c`)

\ `r` (carriage return CR, Unicode `\u000d`)

\ *LineTerminator* (line continuation, no Unicode representation)

\ `"` (double quote ", Unicode `\u0022`)

\ `'` (single quote ', Unicode `\u0027`)

\ `\` (backslash \, Unicode `\u005c`)

OctalEscape (octal value, Unicode `\u0000` to `\u00ff`)

OctalEscape:

\ *OctalDigit*

\ *OctalDigit OctalDigit*

\ *ZeroToThree OctalDigit OctalDigit*

OctalDigit:

(one of)

`0 1 2 3 4 5 6 7`

ZeroToThree:

(one of) `0 1 2 3`

> 上面的 OctalDigit 产品来自§3.10.1。八进制转义是为了与 C 兼容而提供的，但是只能表示从\u0000 到\u00FF 的 Unicode 值，所以 Unicode 转义通常是首选的。

如果转义序列中反斜杠后面的字符不是 LineTerminator 或 ASCII b、s、t、n、f、r、"、'、\、0、1、2、3、4、5、6 或 7，则为编译时错误。

字符字面量、字符串字面量或文本块内容中的转义序列是通过用 EscapeSequence 语法中的 Unicode 转义表示的单个字符替换其\和尾随字符来解释的。延续行转义序列没有对应的 Unicode 转义，因此解释为不替换它。

行延续转义序列可以出现在文本块中，但不能出现在字符字面量或字符串字面量中，因为它们都不允许使用 LineTerminator。

3.10.8 空字面量

null 类型有一个值，即空引用，由空字面量 null 表示，该值由 ASCII 字符组成。

NullLiteral:

`null`

null 字面量总是 null 类型 (§4.1).

3.11 分隔符

ASCII 字符组成的 12 个标记是分隔符(标点符号)。

Separator:
(one of)

( ) { } [ ] ; , . … @ ::

3.12 操作符

由 ASCII 字符组成的 38 个标记就是操作符。

Operator:
(one of)

=	>	<	!	~	?	:	->				
==	>=	<=	!=	&&	||	++	--				
+	-	*	/	&	|	^	%	<<	>>	>>>	
+=	-=	*=	/=	&=	|=	^=	%=	<<=	>>=	>>>=	