文档编号： 0.1
日期： 2024-12-12

编程语言 X

目录

表格列表

图片列表

1 词法约定 [lex]

Token:
Identifier
Keyword
Punctuator
Literal
LambdaParameter
MacroInvocation

TokenDelimited:
`(` *TokenList?* `)`
`[` *TokenList?* `]`
`{` *TokenList?* `}`

TokenList:
TokenListItem+

TokenListItem:
Token 但不是 `(` `)` `[` `]` `{` `}`
TokenDelimited

1 程序文本指将被翻译为 *X* 程序的文本的整体或者一部分。它存储在源文件中，并以 UTF-8 编码读取。

2 程序文本将被分割为标记的序列。标记是程序文本中的最小单元，包括标识符、关键字、标点符号、字面量、lambda 参数。除了少数地方，标记之间包含的空白字符或注释会被忽略。它们不影响程序含义。

3 宏调用是特殊的标记，它将在编译时展开为标记序列。

1.1 注释 [lex.comment]

1 有两种形式的注释：以`/*` 开始，`*/`结束的块注释和以`//` 开始，到行末结束的行注释。注释可以嵌套。在将程序文本分割为标记以后，注释和空白一起被删除。

1.2 标识符 [lex.identifier]

Identifier:
NormalIdentifier
RawIdentifier

NormalIdentifier:
*IdentifierHead IdentifierTail**

IdentifierHead:
Unicode(XID_Start)
`_`

IdentifierTail:
IdentifierHead
Unicode(XID_Continue)

RawIdentifier:
` *RIchar*+ `

RIchar:
除了 `以外的非空白可打印字符
空格

1 标识符以具有 Unicode XID_Start 属性的字符或_开始，后跟零个或数个具有 Unicode XID_Continue 属性的字符，但不能与关键字相同。标识符区分大小写。

2 使用反引号包围的字符序列称为*原始标识符*。原始标识符可以用于将无法作为普通标识符的字符序列作为标识符使用。

3 如果原始标识符中的字符序列是普通标识符的字符系列，则它与不带反引号的形式完全相同，但它不会被认为是关键字。

4 原始标识符中不能包含空格以外的空白字符以及 `。原始标识符不能以 $ 开头。

1.3 关键字 [lex.keyword]

1

表 1 — 关键字

_	any	as	assert	async
await	auto	bool	borrow	break
catch	char	class	cmp	const
continue	defer	do	dyn	else
enum	extern	false	float	for
func	if	impl	import	in
infer	init	int	internal	is
let	macro	match	module	mut
never	nil	operator	partial	private
public	ref	return	self	shl
shl_eq	shr	shr_eq	some	static
string	this	throw	trait	true
try	type	typeof	uint	unsafe
void	while			

2 表 2 中的标识符称为*上下文关键字*。在特定的语法结构中它将被解析为关键字，在其他位置可以当作一般标识符使用。

1.4 标点符号 [lex.punc]

Punctuator:
PunctuatorPart+

表 2 — 上下文关键字

`didSet`	`get`	`infix`	`lazy`	`once`
`prefix`	`root`	`set`	`suffix`	`super`
`then`	`willSet`			

PunctuatorPart: 以下之一

`~ ! @ # $ % ^ & * ( ) - + = [ ] { } | ; : ' < > , . ? /`

1 标点符号由一个或数个符号组成，其中的一部分称为运算符，参见 11。

2 除了多字符标点符号外，标点符号都由单个字符组成。表 3 列出了内建的多字符标点，但不包含运算符。解析标点符号时，应尽可能长地匹配多字符标点符号。用户也可以自定义多字符运算符。

表 3 — 多字符标点符号

`'(`	→	⇒	`!!`
`::`	`...`		

1.5 字面量 [lex.literal]

Literal:

IntegerLiteral
FloatingLiteral
StringLiteral
CharacterLiteral
SymbolLiteral
BooleanLiteral

1.5.1 整数字面量 [literal.integer]

IntegerLiteral:

DecimalLiteral Suffix?
BinaryLiteral Suffix?
HexadecimalLiteral Suffix?

DecimalLiteral:

Digits

Digits:

Digit
Digits `'?` *Digit*

Digit: 以下之一

`0 1 2 3 4 5 6 7 8 9`

BinaryLiteral:

`0b` *BinaryDigit* `('?` *BinaryDigit*`)*`

BinaryDigit:

`0`

`1`

HexadecimalLiteral:

`0x` *HexadecimalDigits*

HexadecimalDigits:

HexadecimalDigit

HexadecimalDigits `'*` *HexadecimalDigit*

HexadecimalDigit: 以下之一

`0 1 2 3 4 5 6 7 8 9`

`A B C D E F`

`a b c d e f`

1 整数字面量由一系列数字构成。可以使用单引号作分隔并且不影响字面量的值。字面量的前缀用于指示它的进制。十进制字面量由若干十进制数字构成；十六进制字面量由前缀 `0x` 后跟若干十六进制数字构成；二进制字面量前缀 `0b` 后跟若干二进制数字构成。*X* 不支持八进制字面量。

2 整数字面量的值为其数字序列表示的值，依不同前缀分别为十进制、十六进制或二进制。最左侧的数字为最高位。字面量的类型参见表格 4，其中 i 为其字面值：

表 4 — 整数字面量后缀

后缀	对应的类型	后缀	对应的类型
无	`int`$_i$	`u`	`uint`$_i$
`i8`	`int<8>`$_i$	`u8`	`uint<8>`$_i$
`i16`	`int<16>`$_i$	`u16`	`uint<16>`$_i$
`i32`	`int<32>`$_i$	`u32`	`uint<32>`$_i$
`i64`	`int<64>`$_i$	`u64`	`uint<64>`$_i$
`i128`	`int<128>`$_i$	`u128`	`uint<128>`$_i$
`f16`	`float<16>`	`f32`	`float<32>`
`f` 或 `f64`	`float<64>`	`f128`	`float<128>`

如果字面量的字面值超出了其类型的约束范围，则这是一个编译错误。

1.5.2 浮点字面量 [literal.floating]

FloatingLiteral:

DecimalFloatingLiteral Suffix?

HexadecimalFloatingLiteral Suffix?

DecimalFloatingLiteral:

Digits `.` *Digits ExponentPart*?

Digits ExponentPart

HexadecimalFloatingLiteral:

HexadecimalPrefix HexadecimalDigits `.` *HexadecimalDigits BinaryExponentPart*`?`

HexadecimalPrefix HexadecimalDigits BinaryExponentPart

ExponentPart:

`e` *Sign*`?` *Digit*`+`

`E` *Sign*`?` *Digit*`+`

BinaryExponentPart:

`p` *Sign*`?` *Digit*`+`

`P` *Sign*`?` *Digit*`+`

Sign: 以下之一

`+` `-`

[1] 浮点字面量用于表示浮点数, 其中的单引号用作分隔并且不影响字面量的值。浮点字面量的小数点前后不允许省略数字。

[2] 浮点字面量的类型按表 5 确定。其值依如下方式确定：如果它包含指数部分，则命 e 为指数部分按十进制数字解析得到的数；否则，e 为 0。对十进制浮点字面量而言，命 s 为除去指数的部分按十进制数字解析得到的数，则令 $f = s \times 10^e$。对十六进制浮点字面量而言，命 s 为除去指数的部分按十六进制数字解析得到的数，则令 $f = s \times 2^e$。浮点字面量的值为其类型中最接近 f 的值。如果 f 太大，则值为对应的正无限大；如果 f 太小，则值为 0。

表 5 — 浮点字面量后缀

后缀	对应的类型
`f16`	`float<16>`
`f32`	`float<32>`
无或 `f64`	`float<64>`
`f128`	`float<128>`

1.5.3 字符串字面量 [literal.string]

StringLiteral:

`"` *Schar*`*` `"` *Suffix*`?`

`@+` `"` *Rchar*`*` `"` `@+` *Suffix*`?`

Schar:

除了`\`和`"` 以外的可打印字符

EscapeSeq

TextInterpolation

EscapeSeq:

`\` *SimpleEscape*

`\u{` *HexadecimalDigit*`+` `}`

TextInterpolation:
`\(` *Expression* `)`

SimpleEscape: 以下之一
`0 ' " \ a b f n r t v`

Rchar:
可打印字符
RawTextInterpolation

RawTextInterpolation:
`\ @+ (` *Expression* `)`

1 字符串字面量表示 UTF-8 字符串，其类型为 `string`。普通字符串字面量被双引号包围。其中可以使用反斜杠开始的转义序列表示其他字符。普通字符串字面量也可以前后加上等量的 `@`，此时它被称为*原始字符串字面量*。原始字符串字面量中的反斜杠不会被解释为转义序列，而是字符本身。

2 字符串字面量可以横跨多行，但其开头引号必须为其所在行的最后一个字符，且结尾的引号必须为其所在行的第一个非空白字符。结尾引号之前的所有空白字符将作为这个字符串字面量的*行前缀*。开头引号到结尾引号之间的所有字符（包括换行符）按顺序成为该字符串字面量的内容，但除了以下字符：

(2.1) — 开头引号之后紧邻的换行；

(2.2) — 每一行的行前缀；

(2.3) — 结尾引号前一行的换行符；

(2.4) — 如果一行末尾有反斜杠，这个反斜杠和其后的换行符。

换行符会变成`'\n'`。如果开始引号后不是换行符，或者结束引号前有非空白字符，这是一个编译错误。如果行前缀不是某一行的前缀，这是一个编译错误。

[例：表 6 是一些多行字符串字面量及其等价的单行表示：

表 6 — 多行字符串字面量示例

`"`
`abc`
`"`
	`"abc"`

`"`
`abc\`
`def`
`"`
	`"abcdef"`

`"`
`abc`
`␣␣␣"`
	编译错误，空格不是公共前缀

`"`
 	
`abc`
 	
`"`
	`"\nabc\n"`

]

3 字符串字面量可以包含字符串插值，其形式为`\(`e`)`，其中 e 为表达式。字符串插值会被求值后转换为字符串插入当前位置。对原始字符串字面量而言，反斜杠和括号之间需要插入等量的 `@`，否则仍然会被解释为字面符号。

1.5.4 字符字面量 [literal.char]

CharacterLiteral:
`'` *Character* `'`

Character:
除了`\`和`'` 以外的非换行可打印字符
EscapeSeq

1 字符字面量表示单个字符，其类型为 `char`。字符字面量由单引号包围，其中的字符可以是除了单引号和反斜杠以外的任意字符，或者转义序列。

1.5.5 符号字面量 [literal.symbol]

SymbolLiteral:
`'` *Identifier*

1 符号字面量用于标识成员名称，其类型和其值为其标识符的值。

1.5.6 布尔字面量 [literal.boolean]

BooleanLiteral:
`true`
`false`

$$\texttt{true} \coloneqq \langle \text{true}, \texttt{bool} \rangle$$
$$\texttt{false} \coloneqq \langle \text{false}, \texttt{bool} \rangle$$

1 布尔字面量的类型为 `bool`。`true` 和 `false` 分别对应其真值与假值。

1.5.7 字面量后缀 [literal.suffix]

Suffix:
`_?` *SuffixIdentifier*

SuffixIdentifier:
SuffixIdentifierHead SuffixIdentifierTail`*`

SuffixIdentifierHead:
Unicode(XID_Start)

SuffixIdentifierTail:
SuffixIdentifierHead
Unicode(XID_Continue)

1 整数、浮点数与字符串能带有内建或用户自定义的后缀。后缀由字母开始，后跟任意数量的字母或数字，字面量与后缀之间可以添加_分隔。用户自定义的后缀不能与内建后缀相同，否则这是一个编译错误。

2 自定义后缀的规则与标识符相同，但会自动去除前导的_。如果前导的下划线多于一个，这是一个编译错误。自定义后缀不能与内建后缀相同。

[例：

```
IntegerLiteral<'s>; // 后缀为 s
IntegerLiteral<'_s>; // 错误，后缀不能包含前导下划线

0x0123ABC; // 没有后缀
0x0123_ABC; // 后缀为 ABC，下划线用作区分
```

]

3 特征 `IntegerLiteral`、`FloatingLiteral`、`StringLiteral`、`CharacterLiteral` 用于实现具有特定后缀的字面量。如果有多于一个类型实现了相同的后缀或者提供了非法的后缀，这是一个编译错误。

1.6 Lambda 参数 [lex.lambda-param]

LambdaParameter:
    $ *Digit*+
    $ *Identifier*

1 Lambda 参数只能在 lambda 作用域（2.1.3）中使用，用于引用匿名参数。其类型是待推导的。不在 lambda 作用域中使用 lambda 参数，或在显式指定参数的 lambda 表达式中使用 lambda 参数，是一个编译错误。

2 基本概念 [basic]

1 实体包括对象、函数、类型、模块、运算符、扩展。

2.1 作用域 [scope]

1 *作用域*是一段程序文本。特定的语言功能可能只能在特定的作用域中生效。不同的作用域具有不同的类型，分别被不同的语言功能所引用。作用域可以互相包含。

2 *全局作用域*是整个程序文本代表的作用域，包含所有其他作用域。

2.1.1 声明作用域 [scope.decl]

1 声明作用域限制声明的范围。一个声明或绑定将会被插入到最近的声明作用域中，并且在该作用域内可以使用该名称引用被声明的实体。在离开该作用域之后，被声明的实体将不能被使用该方式引用。

2 所有的语句都具有声明作用域。全局作用域也是声明作用域。

2.1.2 函数作用域 [scope.func]

1 函数作用域限制 `return` 语句、`throw` 语句的使用。参见 5.6。

2 函数作用域也限制 `await` 运算符的使用。参见 4.3.4。

3 函数定义的块、lambda 表达式的块或表达式、`do` 表达式的块和函数调用表达式的 lambda 块具有函数作用域。

2.1.3 Lambda 作用域 [scope.lambda]

1 Lambda 作用域限制 lambda 参数的使用。参见 1.6。

2 lambda 表达式的块或表达式以及函数调用表达式的 lambda 块具有 lambda 作用域。

3 属性的访问器也具有 lambda 作用域。参见13.1。

2.1.4 序列作用域 [scope.sequence]

1 序列作用域限制 `$` 的使用，参见 4.2.5。序列作用域有一个关联的当前序列值，无论它是否实现 `Sequence`。

2 下标运算符 *s*`[...]` 的两个方括号之间具有序列作用域。其当前序列为 *s*。

3 函数调用表达式 *s*`(...)` 的括号之间具有序列作用域。如果 *s* 形如 *o*`.`*f*，且 *f* 是一个方法，则其当前序列为 *o*，否则当前序列为 *s*。[注：此处只能进行一次拆分，即 *a*`.`*b*`.`*c* 的当前序列不可能为 *a*。]

4 如果函数调用表达式带有一个 lambda 块，则这整个块也具有序列作用域，其当前序列确定方法同上。

[例：

```
let a = [1, 2, 3, 4, 5];
```

```
let o = (a: a);

impl int[] : Functor<() → int[]> {
    func call(&this, index: usize) ⇒ 0;
}

impl int[] {
    func v(&this, index: usize) ⇒ 1;
}

a[$ - 1] // 当前序列为 a
a($ - 1) // 当前序列为 a
o.a[$ - 1] // 成员访问，当前序列为 o.a
a.v($ - 1) // 方法调用，当前序列为 a
```

]

2.2 名称 [name]

UnqualID:

Identifier

`init`

FullID:

UnqualID

UnqualID `::` *FullID*

TypeName `::` *FullID*

1 名称用于引用程序实体。一个未限定名称可能是一个标识符或关键字 `init`。

2 限定名称由未限定名称通过`::` 连接而成，用于指定嵌套在其他实体内部的名称。除最后一个名称之外，其他未限定名称也可以是类型名称。[例：例如 `int::someMethod`。]

2.2.1 名称查找 [name.lookup]

1 名称查找用于解析一个 *IDExpr* 具体指代的实体。

3 类型系统 [typesystem]

Type:
NormalType
ResultType

NormalType:
`(` *Type* `)`
FundaType
SpecialType
CompType
OpaqueType
SomeType
AnyType
`typeof (` *Expression* `)`
Type TypeQualifier

TypeQualifier:
`mut`
`const`

$$\mathscr{V} = \{\langle v, T, Q\rangle \mid T \in \mathscr{T}, v \in T, Q \subset \mathbb{Q}\}$$

[1] 类型是一个集合。值是类型、类型的成员和修饰符集合的元组。T 称为值 v 的类型。

3.1 基本类型 [type.funda]

FundaType:
`void`
`never`
`bool`
`int`
`uint`
`int <` *Expression* `>`
`uint <` *Expression* `>`
`float`
`float <` *Expression* `>`
`char`
`string`
SymbolType

$$\texttt{void} \coloneqq \{\text{void}\}$$

1 `void` 标识只有唯一一个值的类型。

$$\texttt{never} \coloneqq \{\}$$

2 `never` 标识没有值的类型。

$$\texttt{bool} \coloneqq \{\text{false}, \text{true}\}$$

3 `bool` 标识具有假（`false`）或真（`true`）两个值的类型。

4 `true` 的二进制表示为 `0x01`，`false` 为 `0x00`。持有其他二进制表示的 `bool` 类型的值是未定义的。

5 `bool` 是有序的，其中 `false` 小于 `true`。

$$\texttt{int}_{l,h} \coloneqq \{x \in \mathfrak{Z} \mid l \le x \le h\}$$

6 $\texttt{int}_{l,h}$ 称作*整数类型*，其中 l 和 h 为待推导常数。在本规范中，如果 $l = h$，则记作 $\texttt{int}_l$。存在实现定义的常数 m 和 M。l 和 h 须满足

$$l \ge m$$

$$h \le M$$

$$0 \le h - l \le M$$

`uint` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l \ge 0$。

7 `int<`w`>` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l \ge -2^{w-1}$ 且 $h \le 2^{w-1} - 1$。`uint<`w`>` 是 $\texttt{int}_{l,h}$ 的别名，但满足 $l \ge 0$ 且 $h \le 2^w - 1$。其中 w 可以取 8、16、32、64 或 128。它们称作*定长整数类型*，表示长度固定的整数。只能在定长整数类型上进行位运算。

$$\texttt{float<}s\texttt{>}^* \subset \mathfrak{R}$$

$$\texttt{float<}s\texttt{>}^\dagger \subset \{+\infty, -\infty, \text{NaN}s\}$$

$$\texttt{float<}s\texttt{>} \coloneqq \texttt{float<}s\texttt{>}^* \cup \texttt{float<}s\texttt{>}^\dagger$$

8 `float<`s`>` 称作*浮点类型*，其中 s 可以取实现定义的值（典型值为 16、32、64 或 128）。`float` 为 `float<64>` 的别名。

9 整数类型、定长整数类型和浮点类型称为*算术类型*。

$$\texttt{char} \coloneqq \{\text{Any Unicode scalar value}\}$$

10 `char` 表示一个 Unicode 标量值。值不在 Unicode 标量值范围内的 `char`（例如，其值为代理项）是未定义的。

$$\texttt{string} \coloneqq \{\text{Any UTF-8 String}\}$$

11 `string` 表示任意长度的 UTF-8 字符串，包括空字符串。

3.1.1 符号类型 [type.symbol]

SymbolType:
SymbolLiteral
`'(` *UnqualID* `)`

$$\texttt{'}s \coloneqq \{\langle \texttt{'}s, \texttt{'}s\rangle\}$$

$$\texttt{Symbol} \coloneqq \bigcup_{\texttt{'}s} \{\texttt{'}s\}$$

1 符号字面量的类型与其值相同。`Symbol` 是符号字面量的公共类型。参见 20.1。

3.1.2 特殊类型 [type.special]

SpecialType:
`self`

1 `self` 在特征或实现中代表当前类型。在此之外使用 `self` 是一个编译错误。

3.2 复合类型 [type.comp]

CompType:
Type `?`
Type `[ ]`
Type `[` *Type* `]`
StructType
FuncType
UnionType
Type `&`
FuncType
OpaqueType

3.2.1 可空类型 [type.optional]

$$T\texttt{?} \coloneqq \{\langle t\rangle \mid t \in T\} \cup \{\text{nil}\}$$

1 T`?` 为可空类型。T`?` 要么包含一个 T 的值（使用 `some` 标识），要么不包含任何值（使用 `nil` 标识）。

2 可空类型在核心库中对应 `core::Optional`。参见 20.1。

3.2.2 数组类型 [type.array]

$$T\texttt{[]} \coloneqq \bigcup_{i=1}^{\infty} T^i$$

1 $T\texttt{[]}$ 为数组类型，代表有限个类型 T 的值的序列。

2 数组类型在核心库中对应 `core::Array`。参见 20.1。

3.2.3 字典类型 [type.dict]

$$T\texttt{[}K\texttt{]} \coloneqq T^K$$

1 $T\texttt{[}K\texttt{]}$ 为字典类型，代表键类型 K 到值类型 T 的映射。

2 字典类型在核心库中对应 `core::Dictionary`。参见 20.1。

3.2.4 结构类型 [type.struct]

StructType:
`(` `)`
`(` *StructTypeList* `,?` `)`

StructTypeList:
UnnamedStructType
UnnamedStructType `,` *NamedStructType*
NamedStructType

UnnamedStructType:
StructTypeQualifier`*` *Type*
UnnamedStructType `,` *StructTypeQualifier*`*` *Type*

NamedStructType:
StructTypeQualifier`*` *Identifier* `:` *Type InitialValue*`?`
NamedStructType `,` *StructTypeQualifier*`*` *Identifier* `:` *Type InitialValue*`?`

StructTypeQualifier:
`mut`

TypeNotation:
`:` *Type*

InitialValue:
`=` *Expression*

$$\texttt{(}T_1\texttt{,}\ldots\texttt{,}T_m\texttt{,}K_1\texttt{:}U_1\texttt{,}\ldots\texttt{,}K_n\texttt{:}U_n\texttt{)} \coloneqq \prod_{i=1}^{m} T_i \times \prod_{j=1}^{n} U_j$$

$$\texttt{()} \coloneqq \texttt{void}$$

1 (*T*₁, …, *T*ₘ, *K*₁:*U*₁, …, *K*ₙ:*Uₙ*) 称作结构类型。结构类型包含顺序成员和命名成员，其中顺序成员可以使用索引访问，命名成员可以使用标识符访问。如果结构类型不包含任何命名成员，则它称为元组类型。

2 特别地，只有一个元素的元组需表示为 `(`*T*`,)`。没有元素的元组与 `void` 等价。

3 命名成员可以具有默认值。具有默认值的成员在初始化的时候可以省略而使用默认值。

3.2.5 引用类型 [type.ref]

1 引用类型是另一个值的引用。

3.2.6 函数类型 [type.func]

FuncType:
ParameterInType FuncTypeQual`*` *ReturnType*
Type FuncTypeQual`*` *ReturnType*

ParameterInType:
`(` *ParamListInType*`?` `)`

ParamListInType:
ThisParamDeclInType
ThisParamDeclInType `,` *NamedParamListInType*
ThisParamDeclInType `,` *UnnamedParamListInType*
ThisParamDeclInType `,` *UnnamedParamListInType* `,` *NamedParamListInType*
UnnamedParamListInType
UnnamedParamListInType `,` *NamedParamListInType*
NamedParamListInType

UnnamedParamListInType:
UnnamedParamDeclInType
UnnamedParamListInType `,` *UnnamedParamDeclInType*

NamedParamListInType:
NamedParamDeclInType
NamedParamListInType `,` *NamedParamDeclInType*

UnnamedParamDeclInType:
ParamQual`?` *Type*
ParamQual`?` *Identifier* `:` *Type*`?`

NamedParamDeclInType:
ParamQual`?` `(` *Identifier* `)` *Type*`?`
ParamQual`?` `(` *Identifier* `)` *Identifier* `:` *Type*`?`

ThisParamDeclInType:
`this` `:` *Type*

FuncTypeQual:
ThrowQual
`async`
`unsafe`
`mut`
`once`

1 (*T*₁, ..., *Tₘ*, *K*₁:*U*₁, ..., *Kₙ*:*Uₙ*) → *R* 称作*函数类型*。函数类型标识能以函数方式调用的值。如果函数只有一个顺序参数，则可以省略括号。

2 函数类型可以包含显式指定类型的 `this` 参数。在这种情况下，函数可以在动态成员访问表达式中作为方法被调用。

3 如果函数类型包含 `once` 修饰符,则它只能调用一次,随后值会被消耗,它实现了 `core::ops::FunctorOnce`。如果函数类型包含 `mut` 修饰符，则对它的调用将会更改其值，它实现了 `core::ops::FunctorMut`。如果函数类型不包含这两种修饰符，则对它的调用不会更改其值，它实现了 `core::ops::Functor`。函数均具有此类函数类型。

3.3 不透明类型 [type.opaque]

OpaqueType:
`class` *Type*

1 不透明类型用于从现有的类型出发构造一个相同但不能混用的类型。类型 *T* 与其构造的不透明类型 *U* 之间不具有隐式转换，但可以显式转换。同一个类型构造的复数个不透明类型之间也不能混用。

2 不透明类型会继承原类型的

(2.1) — 成员访问（顺序成员和命名成员），但这些成员的默认访问级别会更改为 `private`；

(2.2) — 固有实现；

(2.3) — 所有特征实现（以及所有运算符）。

[例：

```
type T = class int;
type U = class int;

let i = 0;
let j: T = i; // 错误，int 不能隐式转换为 class int
let k: T = i as T; // 可以，显式转换
let m: U = j; // 错误，T 和 U 之间没有隐式转换
```

]

3.4 `impl` 类型 [type.impl]

SomeType:
    `impl` *Type*
    `impl _`

1 `impl` T 标识一个静态待推导类型，但保证该类型为 T 的子类型。它可以出现在具有初始化值的绑定的类型中或函数返回值处。`impl _`代表一个基础类型待推导的 `impl` 类型。

2 `impl` 还能用于简化泛型函数声明。参见 14.2。

[例：

```
trait T { }
type A { }
type B { }

impl A : T { }
impl B : T { }

let x: impl T = A(); // x 的类型是 A
let mut y: impl T = B(); // y 的类型是 B

y = x; // 错误，B 不能赋给 A
```

]

3.5 `dyn` 类型 [type.dyn]

AnyType:
    `dyn` *Type*
    `dyn _`
    `dyn`

1 `dyn` T 对 T 的子类型进行包装，保证在运行时可以接受任何为 T 的子类型的值。`dyn _`代表一个基础类型待推导的 `dyn` 类型。`dyn` 表示对任意类型的包装。

[例：

```
trait T { }
type A { }
type B { }

impl A : T { }
impl B : T { }

let x: dyn T = A { }; // x 的类型是 dyn T
```

```
let mut y: dyn T = B { }; // y 的类型是 dyn T

y = x; // 正确，dyn T 之间可以互相赋值
```

]

3.6 结果类型 [type.result]

ResultType:
    NormalType `!!` *NormalType*

1 结果类型 T `!!` E 标识一个可能产生错误的值，其中 T 是正常的返回值类型，E 是错误类型，且必须实现 `ErrorCode`。

3.7 具名类型 [type.named]

TypeName:
    `(` *Type* `)`
    EntityID
    FundaType
    SpecialType
    `typeof (` *Expression* `)`

1 类型名称在特定的语法位置表示类型，以避免潜在的语法歧义。

3.8 子类型 [subtype]

1 类型 A 可能是类型 B 的子类型，记作 $A \preceq B$。A 可以在需要 B 的上下文中隐式转换到 B。

2 子类型关系具有自反性和传递性，即对任意类型 A、B 和 C 有 $A \preceq A$ 和 $A \preceq B \land B \preceq C \implies A \preceq C$ 成立。

$$T \preceq \texttt{void}, T \in \mathcal{T}$$
$$\texttt{never} \preceq T, T \in \mathcal{T}$$

3 任何类型都是 `void` 的子类型。`never` 是任何类型的子类型。

$$I_{l_1,h_1} \preceq J_{l_2,h_2} \text{ if } l_1 \geq l_2 \lor h_1 \leq h_2$$
$$\texttt{float<}s_1\texttt{>} \preceq \texttt{float<}s_2\texttt{>} \text{ if } s_1 \leq s_2$$
$$I_{l,h} \preceq \texttt{float<}s\texttt{>}$$
$$I, J \in \{\texttt{int}, \texttt{uint}, \texttt{int<}w\texttt{>}, \texttt{uint<}w\texttt{>}\}$$

4 范围更小的整数类型是范围更大的整数类型的子类型。长度更小的浮点类型是长度更大的浮点类型的子类型。

5 整数类型是浮点类型的子类型。当整数被隐式转换为浮点数时，其值将被转换为最接近的浮点数。[注：虽然整数转换到浮点数可能无法保持值不变，但出于习惯仍然保持这个隐式转换。] [注：浮点类型不能隐式转换到整数类型，但可以显式转换。]

$$\texttt{'}s \preceq \texttt{Symbol}$$

6 所有符号字面量类型都是 `Symbol` 的子类型。

7 `bool` 没有语言内建的子类型约束。但是，其他类型的值可以在 `if` 表达式中当作条件使用，这通过实现 `Condition` 完成。

$$T \preceq T\texttt{?}$$
$$T\texttt{[]} \preceq U\texttt{[]} \text{ if } T \preceq U$$
$$T\texttt{[}K\texttt{]} \preceq U\texttt{[}L\texttt{]} \text{ if } T \preceq U \text{ and } K \preceq L$$

8 任意类型是其对应可空类型的子类型。如果 T_i 是 U_i 的子类型，则 $T_0\texttt{[]}$、$T_0\texttt{[}T_1\texttt{]}$、$\texttt{(}T_1\texttt{, ..., }T_n\texttt{)}$ 分别是 $U_0\texttt{[]}$、$U_0\texttt{[}U_1\texttt{]}$、$\texttt{(}U_1\texttt{, ..., }U_n\texttt{)}$ 的子类型。[注：这意味着数组和字典的元素类型是协变的。]

9 对两个结构类型 T 和 U 而言，如果

(9.1) — T 的顺序成员数量小于或等于 U 的；

(9.2) — T 的顺序成员是对应 U 顺序成员的子类型；

(9.3) — T 的命名成员是对应 U 命名成员的子类型，或具有默认值；

则 U 是 T 的子类型。

10 对两个函数类型 T 和 U 而言，如果 U 的每个参数类型都是对应 T 的参数的子类型，且 T 的返回类型是 U 的返回类型的子类型，则 T 是 U 的子类型。[注：这意味着函数类型的参数类型是逆变的，返回类型是协变的。] T 可以拥有比 U 更多的顺序参数，或者 U 不包含的命名参数。

11 类类型可以定义类型转换函数。每个这样的函数定义了一个子类型关系。

12 对类型表达式而言，$T\texttt{ \& }U$ 是 T 和 U 的子类型；T 和 U 是 $T\texttt{ | }U$ 的子类型。

3.9 公共类型 [type.common]

1 对两个类型 A 和 B 而言，存在一个唯一的类型 C 称为 A 和 B 的*公共类型*。C 满足：

$$\begin{gathered} A \preceq C \\ B \preceq C \\ \forall D \in \mathcal{T}, A \preceq D \land B \preceq D \Rightarrow C \preceq D \end{gathered} \tag{3.1}$$

记作 $C = A \otimes B$。公共类型满足交换律。

2 如果 $A \preceq B$，则 $A \otimes B = B$。

3 如果 A 和 B 之间没有子类型关系，则 $A \otimes B = A\texttt{ | }B$。

3.10 修饰符 [qualifier]

1 值除了总是具有类型之外，还可能带有一个或数个修饰符。修饰符指示了值的其他属性。

2 类型可以带有修饰符，指定该值需有特定的修饰符约束。

3.10.1 `mut` [qual.mut]

1 `mut` 表示该值是可变的。具有 `mut` 的值才能成为赋值操作符的左操作数。参见 4.14。

3.10.2 `const` [qual.const]

1 `const` 表示该值是一个常量值，于编译期间确定且不可变。部分语言功能只允许常量值。

2 `const` 绑定创建一个常量并插入当前作用域。该绑定的初始值必须是常量表达式。

4 表达式 [expr]

Expression:
PrimaryExpr
Operator Expression
Expression Operator
Expression Operator Expression

1 表达式由运算符与操作数按照顺序组合在一起，它表示一个计算过程。运算符是若干标点符号的组合、标识符或是语言规定的特殊结构。运算符可以是前缀、后缀或者是中缀的，这决定了运算符与其操作数的结合方式。运算符组是运算符的集合，每一个运算符都属于某个运算符组。运算符组之间有一个弱偏序关系，决定了它们的优先级。完全由中缀运算符构成的运算符组有结合性：左结合的组每个运算符从左到右选择操作数；右结合的组从右到左；无结合的组两个运算符不能选择同一个操作数。

2 4.3到4.14各节按照优先级从高到低依次对运算符组进行描述。若无特别说明，每一节描述一个运算符组。用户也能定义新的运算符组或在当前的组中添加新的运算符，参见11.2。

3 对含有子表达式的表达式求值时，总是先对其子表达式按出现次序从左到右求值。

4 丢弃表达式 e 的结果指，在决定 e 的类型时，直接将它确定为 `never` 而跳过所有步骤；在对 e 求值时，进行所有步骤，但是如果求值正常结束，丢弃最后的值。

5 完整表达式指下列情况之一：

(5.1) — 表达式语句中的表达式；

(5.2) — 一个语句表达式；

(5.3) — 绑定语句中的初始化表达式；

(5.4) — lambda 表达式的函数体表达式；

(5.5) — `if`、`match`、`while` 的条件表达式；

(5.6) — `for` 语句的迭代表达式；

(5.7) — `return`、`throw`、`break` 表达式的子表达式；

(5.8) — `assert` 表达式的子表达式；

(5.9) — 函数参数的默认值表达式；

(5.10) — 函数体与属性体表达式；

(5.11) — 枚举的初始化表达式；

(5.12) — 模式匹配与泛型的约束表达式；

(5.13) — `typeof` 的参数表达式；

(5.14) — 泛型参数的非类型表达式。

4.1 表达式属性 [expr.props]

1 表达式具有若干属性。这些属性决定了表达式能用在什么样的位置。

2 括号表达式 `(`*e*`)` 的属性总是与 *e* 完全相同。

3 表达式总是具有类型。类型决定了从表达式读取值会得到什么值以及向表达式写入值需要什么值。

4.1.1 可读性 [expr.props.readable]

1 *可读*的表达式可以获取它的值，对可读表达式进行*求值*会得到具有该表达式类型的值。除了下述表达式，任何表达式都是可读的。除非指明，需要表达式的地方总是需要一个可读表达式。

2 表达式 `_` 是不可读的。含有项 `...` 的数组、结构或字典字面量是不可读的。

3 如果一个 ID 表达式引用一个不具有 `get` 访问器的属性，则它是不可读的。

4 除非指明，具有不可读子表达式的表达式也是不可读的。

4.1.2 可写性 [expr.props.writable]

1 *可写*的表达式可以向其中写入内容，能作为赋值表达式的左操作数。除了下述表达式，其他表达式都是不可写的。

2 如果可寻址的表达式类型具有 `mut` 修饰符，则它是可写的。如果一个 ID 表达式引用一个具有 `set` 访问器的属性，则它是可写的。

3 表达式 `_` 是可写的。

4 如果一个数组或结构字面量，其成员要么是可写的，要么为 `...` 或 `...`*e*（其中 *e* 是可写的），则它是可写的。

4.1.3 可寻址性 [expr.props.addressable]

1 *可寻址*的表达式可以获取其内存地址。除了下述表达式，其他表达式都是不可寻址的。

2 如果一个 ID 表达式引用一个变量或函数，则它是可寻址的。这包括局部变量（及函数参数）和任何非局部变量。

3 前缀 `*` 表达式总是可寻址的。后缀 `[]` 表达式也总是可寻址的。

4 如果成员访问表达式 `.` 引用的是结构成员（无论顺序的或命名的），则它是可寻址的。

5 如果操作数是可寻址的，则内建的后缀 `?` 及后缀 `!` 表达式是可寻址的。

4.2 基本表达式 [expr.primary]

PrimaryExpr:
- `(` *Expression* `)`
- `_`
- *Statement*
- *LiteralExpr*
- *TypeLiteral*
- *Identifier*
- *DeductedEnumerator*
- `this`
- `$`
- *LambdaParameter*
- *LambdaExpr*
- *DoExpr*
- *TryExpr*

1 表达式 `(`*e*`)` 等价于 *e*，括号只作分组用途。[注：注意括号表达式不是一元元组。]

2 表达式`_`代表赋值中的空位，它不能被读取值，且不参与类型推导。它的类型是 `void`。

3 语句都是基本表达式。但由于某些语句可以在末尾包含一个表达式，这些语句实际上只能在另一侧当作基本表达式。

[例：

```
return 1 + 2 // 等价于 return (1 + 2)
1 + return 2 // 等价于 1 + (return 2)
```

]

4.2.1 字面量表达式 [expr.lit]

LiteralExpr:
- *IntegerLiteral*
- *FloatingLiteral*
- *StringLiteral*
- *CharacterLiteral*
- *SymbolLiteral*
- `'(` *UnqualID* `)`
- *BooleanLiteral*
- `nil`
- *ArrayLiteral*
- *StructLiteral*
- *DictLiteral*

ArrayLiteral:
`[` *ExprList*`?` `]`

ExprList:
ExprListNoComma `,`*?*

ExprListNoComma:
ExprItem
ExprListNoComma `,` *ExprItem*

ExprItem:
Expression
`...` *Expression*`?`

StructLiteral:
`(` *StructItems*`?` `)`

StructItems:
StructItemsNoComma `,`*?*

StructItemsNoComma:
StructItem
StructItemsNoComma `,` *StructItem*

StructItem:
Identifier `:` *Expression*
Identifier
`...` *Expression*`?`

DictLiteral:
`[` *DictItems* `]`
`[` `:` `]`

DictItems:
DictItemsNoComma `,`*?*

DictItemsNoComma:
DictItem
DictItemsNoComma `,` *DictItem*

DictItem:
Expression `:` *Expression*
`...` *Expression*

1 字面量本身是基本表达式。整数字面量、浮点字面量、字符串字面量和布尔字面量的值分别参见1.5.1、1.5.2、1.5.3和1.5.6节的定义。

2 `'(`*id*`)` 表示与 *id* 对应的符号表达式。*id* 本身不能是符号字面量。

3 `nil` 的类型为 *T*`?`，其中 *T* 为待推导的类型参数。

4 数组、结构和字典字面量中的 `...` 项是不可读的，只能用于赋值运算符的左侧。它的类型是 `void`，但不参

与数组、结构或字典的类型推导。

5 数组字面量 `[`e_1`,` ..., e_n`]` 表示一个显式写出其各元素的数组值。其类型为 T`[]`，其中 T 为各表达式的公共类型。如果其中包含形如 `...`e 的项，则视同将 e 的各元素显式插入在该位置。e 必须可迭代。如果数组字面量不包含任何成员，则 T 是一个待推导的类型。

6 结构字面量 `(`e_1`,` ..., e_m`,` k_1`:`f_1`,` ..., x_n`:`f_n`)` 表示一个显式写出其各元素的结构值。其类型为 `(`T_1`,` ..., T_m`,` k_1`:`U_1`,` ..., x_n`:`U_n`)`，其中 T_i 为各 e_i 的类型，U_i 为各 f_i 的类型。特别的，`(`e`,)` 表示一元元组，此时逗号不能省略。`()` 的类型为 `void`，是 `void` 的唯一值。

7 结构字面量必须保持顺序成员在前、命名成员在后的顺序。如果结构字面量中包含形如 `...`e 的项，则视同将 e 的各成员插入在该位置。e 必须是结构类型。如果该项后面有顺序成员，则该项的类型必须不包含任何命名成员；如果该项前面有命名成员，则该项的类型必须不包含任何顺序成员。

8 字典字面量 `[`k_1`:`v_1`,` k_2`:`v_2`,` ..., k_n`:`v_n`]` 表示一个显式写出其各元素的字典值。其类型为 T`[`K`]`，其中 T 为各 v_i 的公共类型，K 为各 k_i 的公共类型。如果其中包含形如 `...`e 的项，则视同将 e 的各元素对显式插入在该位置，e 必须是字典类型。如果字典字面量不包含任何表达式，则需写作 `[:]`。此时 T 和 K 是待推导的类型。

9 如果以方括号界定的字面量只包含元素展开，则其总是被识别为数组字面量。[注：如果需要创建一个只包含元素展开的字典字面量，请将其中一个元素改写为键值对的展开。]

4.2.2 初始化字面量 [expr.lit.init]

TypeLiteral:
TypeParenLiteral
TypeArrayLiteral
TypeDictLiteral

TypeParenLiteral:
TypeName `(` *Arguments?* `)` *Block?*
TypeName Block

TypeArrayLiteral:
TypeName `[` *ExprList?* `]`

TypeDictLiteral:
TypeName DictLiteral

1 可以使用与字面量类似的语法来创建指定类型的值。被创建的类型必须是一个类型名称。

[例：

```
let a = int[] [1, 2, 3]; // 错误，不能用 [] 初始化 int

type IntArray = int[];

let a = IntArray[1, 2, 3]; // 可以
```

```
let a = (int[])[1, 2, 3]; // 可以
```

]

2 T`(`a_1`, ..., `a_n`)` 以给定的参数创建 T 的对象。创建对象时，会以这些参数调用给定的初始化器。以这种方式初始化的对象也可以带有 lambda 块。如果这个对象的初始化器只接受这一个参数，也可以省略括号。如果不存在对应的初始化器，且该参数符合结构字面量的语法（并不包含 lambda 块），则会依次初始化对应的顺序成员和命名成员。

3 T`[`v_1`, ..., `v_n`]` 以对应的数组字面量为参数创建 T 的对象。T`[`k_1`:`v_1`, `k_2`:`v_2`, ..., `k_n`:`v_n`]` 以对应的字典字面量创建 T 的对象。

4 T`[...`e`]` 的语法歧义将按与普通字面量类似的规则解决（见上）。

5 如果使用了内建的初始化方式或者采用了不抛出异常的初始化器，则结果类型为 T；否则，结果类型为 T `!!` E，其中 E 为抛出的异常类型。

4.2.3 匿名静态成员表达式 [expr.enum]

DeductedEnumerator:
- `.` *Identifier*
- `.` *Identifier* `(` *Arguments?* `)` *Block**
- `.` *Identifier Block*
- `.` *Identifier* `[` *ExprList?* `]`

1 静态成员和枚举符可以在适当的上下文中省略类型名称而使用自动推导。参见 15.1。

4.2.4 `this` [expr.this]

1 表达式 `this` 在类方法或扩展方法中表示当前方法的调用者。如果没有在参数中显式指定 `this` 的类型，则其类型为 `self`。

4.2.5 `$` [expr.dollar]

1 表达式 `$` 只能在序列作用域（2.1.4）中使用。如果当前序列为 s，则 `$` 等价于 s`.opeartor$()`。如果 s 实现了 `Sequence`（例如内建数组 T`[]`），其值为 s`.size`。在其他位置使用 `$` 是一个编译错误。

4.2.6 Lambda 表达式 [expr.lambda]

LambdaExpr:
LambdaParameter LambdaQual ReturnType?* ⇒ *LambdaBody*

LambdaQual:
`async`
ThrowQual

LambdaParameter:
ParamDecl

LambdaBody:
Expression

1 `$` 后跟数字 i 指代第 i 个参数。`$` 后跟标识符 n 指代具名参数 n。

4.2.6.1 `do` 表达式 [expr.do]

DoExpr:
`do` *LambdaQual** *Block*
`do!` *LambdaQual** *Block*

1 `do` 后跟一个块创建一个没有显式指定参数的 lambda 表达式。

2 `do!` 后跟一个块创建一个无参的 lambda 表达式并立即调用它，将其值作为整个表达式的值。[注：`do` 和后面的`!`之间必须没有空白，否则会被识别成两个分开的运算符。]

[例：

```
let arr = [1, 2, 3];

let first1 = arr.first(do { $0 > 2 }); // 获取第一个满足条件的元素

let firstFinder = do {
    for let v : $0 {
        if v > 2 { return v; }
    }
    nil
};

let first2 = firstFinder(arr); // 与上面等价

let first3 = do! {
    for let v : arr {
        if v > 2 { return v; }
    }
    nil
}; // 与上面等价
```

]

4.3 后缀运算符 [expr.suffix]

SuffixExpr:
PrimaryExpr
IndexExpr
FuncCallExpr
MemberAccessExpr
AwaitExpr
NullCheckExpr
PrevNextExpr
IncDecExpr

4.3.1 下标运算符 [expr.index]

IndexExpr:
SuffixExpr `[` *ExprList*`?` `]`

1 下标运算符用于对数组或字典进行访问。用户自定义的下标运算符可以接受多于一个参数。

2 对数组 T`[]` 而言，a`[`i`]` 表示数组 a 的第 i 个元素。如果 i 超出可索引的范围，则会 `panic`。结果类型是 T。

3 对字典 T`[`K`]` 而言，d`[`k`]` 表示字典 d 的键 k 对应的值。如果字典中没有这个键，则会返回 `nil`。结果类型是 T`?`。此外，在赋值语境下，d`[`k`] = `e 表示将键 k 对应的值设为 e。如果字典中没有这个键，则会插入一个新的键值对。e 的类型需要为 T。

4 下标运算可以在可变和不可变的情况下具有不同的重载语义。参见 11.3.2。

4.3.2 函数调用运算符 [expr.call]

FuncCallExpr:
SuffixExpr `(` *Arguments*`?` `)` *Block*`*`
SuffixExpr Block

Arguments:
UnnamedArgs
NamedArgs
UnnamedArgs `,` *NamedArgs*

UnnamedArgs:
Argument
UnnamedArgs `,` *Argument*

NamedArgs:
Identifier `:` *Argument*
NamedArgs `,` *Identifier* `:` *Argument*

Argument:
ParamQual`?` *Expression*

1 函数调用运算符用于调用函数。括号内的项作为参数传递给函数。

2 如果函数调用运算符的左操作数形如 $o.f$，其中 f 是一个名称且不是 o 的成员名称，则这称作*方法调用*。此时，o 将作为 f 的 `this` 参数传递给函数 f。

3 函数调用运算符可以后跟一个块。这个块将作为一个匿名 lambda 块，创建一个 lambda 表达式并作为函数的最后一个顺序参数传递给函数。若此时函数没有任何其他参数，则函数调用的小括号可以省略。

4 如果函数参数指定了修饰符，则传递实参时必须显式传递相同的修饰符。

4.3.3 成员访问运算符 [expr.member]

MemberAccessExpr:
SuffixExpr `.` *UnqualID*
SuffixExpr `.` *IntegerLiteral*
SuffixExpr `.` `(` *Expression* `)`

1 成员访问运算符用于访问对象的成员。

2 $o.m$ 表示对象 o 的命名成员 m。如果 o 没有命名成员 m，若它后跟一个函数调用运算符，则作为方法调用处理；否则这是一个编译错误。

3 $o.i$ 用于结构顺序成员的访问，表示 o 的第 i 个顺序成员。i 必须是一个不包含前缀或后缀的十进制字面量。如果 i 大于等于结构顺序成员的数量，这是一个编译错误。

4 $o.(e)$ 首先对 e 求值。如果得到一个 `symbol` 类型的值，则对 o 进行命名成员访问。如果得到一个整数类型的值，则对 o 进行顺序成员访问。此时 o 必须是一个常量表达式。否则，o 必须是一个带有 `this` 参数的函数类型，且其必须后跟一个函数调用运算符，此时将视为对 e 调用方法 o。

[例：

```
let o = (a: 1, b: 2);
let s = 'a;

o.(s) // 等价于 o.a

let t = (1, 2);
let k = 0;

t.(k) // 等价于 t.0

impl int {
    func test(this) ⇒ this;
}

let f: (this: int) → int = int::test;

0.(f)() // 等价于 0.test()
```

]

4.3.4 `await` 运算符 [expr.await]

AwaitExpr:
SuffixExpr `.` `await`

1 `await` 运算符用于等待一个异步表达式的结果。e`.await` 挂起当前计算，直到 e 的值可用。如果 e 的类型是 `Future<`T`>`，则 e`.await` 的类型是 T。只能在具有 `async` 修饰的函数作用域中使用 `await` 运算符。

4.3.5 空值检测运算符 [expr.null]

NullCheckExpr:
SuffixExpr `?`
SuffixExpr `!`

1 e`?` 对 e 进行空值检测。如果 e 的值不为 `nil`，则 e`?` 的值为 e；否则，该表达式直到空值检测运算符为止的整个表达式的值为 `nil`。e 的类型必须是 T`?`。即使空值检测运算符检测为空，表达式的其它部分仍然会被求值。

[例：

```
let a: int? = nil;

a + 1 // 编译错误，不存在接受 int? 和 int 的加法运算符
a? + 1 // nil
(a? + 1)? // nil，多余的运算符
a? + 1 ?? 1 // 1，? 不会越过?? 运算符
(a? + 1) ?? 1 // 同上，但是更清晰
a? + 1 == b // 等号也会停止传播，等价于 (a? + 1) == b

func f(i: int) => i + 1;

f(a?) // 整个表达式的类型为 int?，值为 nil
f(a ?? 2) // 被空值检测运算符截断，整个表达式的类型为 int，值为 3

let mut b: int? = 1;

b = a?; // 被赋值运算符截断，b 成为 nil

match a? { some let t -> true, nil -> false } // 被 match 的条件截断，不会传播到整个 match 表达式级别
```

]

2 后缀`!` 与后缀`?` 类似，但是它将空值直接返回而不是传播给完整表达式。

3 如果空值检测运算符的操作数类型是 T`?`，则整个表达式的类型为 T。

4 如果 e 是结果类型 T `!!` E，则 e`?` 在正确值的情况下使用该值，为错误值的情况下将错误传播到完整表达式。

4.3.6 前驱后继运算符 [expr.prev-next]

PrevNextExpr:
SuffixExpr `+!`
SuffixExpr `-!`

1 e`+!` 和 e`-!` 分别表示 e 的后继和前驱。如果 e 的类型是算术类型，e`+!` 等价于 $e+1$，e`-!` 等价于 $e-1$。

4.3.7 自增自减运算符 [expr.inc-dec]

IncDecExpr:
SuffixExpr `++`
SuffixExpr `--`

1 自增运算符 e`++` 等价于 e `=` e`+!`，自减运算符 e`--`等价于 e `=` e`-!`，但 e 只被求值一次。

4.4 前缀运算符 [expr.prefix]

PrefixExpr:
SuffixExpr
`+` *PrefixExpr*
`-` *PrefixExpr*
`!` *PrefixExpr*
`'~` *PrefixExpr*
`&` `mut?` *PrefixExpr*
`*` *PrefixExpr*
SomeExpr

1 前缀运算符 `+` 和 `-` 分别表示正号和负号。其中 `+` 的值为其操作数的值，而 `-` 的值为其相反数。操作数类型必须为算术类型。

2 逻辑否运算符`!` 用于对布尔值取反。如果操作数为 `true`，则结果为 `false`；如果操作数为 `false`，则结果为 `true`。

3 位取反运算符`'~`进行按位取反。操作数的类型必须是定长整数类型。

4 引用运算符 `&` 获得操作数的引用。`&mut` 获得操作数的可变引用。

5 解引用运算符 `*` 获得一个引用指向的值。

4.4.1 `some` 运算符 [expr.some]

SomeExpr:
`some` *PrefixExpr*

1 `some` 运算符用于将一个值转换为可空值。如果操作数的类型为 T，则结果的类型为 T`?`。

4.5 乘法运算符 [expr.mul]

MulExpr:
PrefixExpr
MulExpr `*` *PrefixExpr*
MulExpr `/` *PrefixExpr*
MulExpr `%` *PrefixExpr*

1 运算符 `*`、`/`和`%` 分别表示乘法、除法和余数。乘除法只对整数类型进行溢出检查，而不对定长整数类型和浮点类型进行。除零检测对整数类型和定长整数类型都生效。

4.6 加法运算符 [expr.add]

AddExpr:
MulExpr
AddExpr `+` *MulExpr*
AddExpr `-` *MulExpr*

1 运算符 `+` 和 `-` 分别表示加法和减法。其操作必须为算术类型。加减法只对整数类型进行溢出检查，而不对定长整数类型和浮点类型进行。

4.7 移位运算符 [expr.shift]

ShiftExpr:
AddExpr
AddExpr `shl` *AddExpr*
AddExpr `shr` *AddExpr*

1 运算符 `shl` 和 `shr` 表示按位左移和右移。其操作数必须为定长整数类型。在同一个表达式中混合使用 `shl` 和 `shr` 是一个编译错误。

4.8 位运算符 [expr.bit]

BitwiseExpr:
ShiftExpr
BitwiseExpr `'&` *ShiftExpr*
BitwiseExpr `'^` *ShiftExpr*
BitwiseExpr `'|` *ShiftExpr*

1 运算符`'&`、`'^` 和`'|` 分别表示按位与、按位异或和按位或。其操作数必须为定长整数类型。在同一个表达式中混合使用`'&`、`'^` 和`'|` 是一个编译错误。

4.9 区间运算符 [expr.range]

RangeExpr:
NullCoalExpr
NullCoalExpr `..` *NullCoalExpr*
NullCoalExpr `..=` *NullCoalExpr*

1 运算符 `..` 生成左闭右开区间,结果类型是 `Range`。运算符 `..=` 生成左闭右闭区间,结果类型是 `ClosedRange`。参数类型必须是整数类型。参见 20.4。

4.10 连接运算符 [expr.connect]

ConnectExpr:
RangeExpr
ConnectExpr `~` *RangeExpr*

1 运算符`~`用于连接字符串或集合。其操作数的类型必须满足以下条件之一：

(1.1) — 两个操作数都是 `string`；

(1.2) — 一个操作数满足 `Sequence<`T`>`，另一个是 T；

(1.3) — 两个操作数都满足 `Sequence<`T`>`。

对第一种情况，结果等于将两个字符串左右连接得到的结果；对第二种情况，x `~` y 等于 `[`x`, ...`y`]` 或 `[...`x`,` y`]`，取决于哪个操作数是序列；对第三种情况，x `~` y 等于 `[...`x`, ...`y`]`。

4.11 空值合并运算符 [expr.null-coal]

NullCoalExpr:
BitwiseExpr
BitwiseExpr `??` *NullCoalExpr*

1 a `??` b 首先对 a 求值，如果其结果是 `some` e，则表达式的值为 e，且 b 不会被求值；否则表达式的值为 b。如果 a 的类型为 A`?`，b 的类型为 B，则表达式的类型为 A 和 B 的公共类型。

2 `??` 是右结合的。

4.12 比较运算符、包含运算符 [expr.cmp-in]

BooleanExpr:
RangeExpr
CompareExpr
IncludeExpr
CastExpr
MatchExpr

1 本节中的运算符的结果都是 `bool`。

4.12.1 比较运算符 [expr.compare]

CompareExpr:
RangeExpr `==` *RangeExpr*
RangeExpr `cmp` *RangeExpr*
LessChainExpr
GreaterChainExpr

LessChainExpr:

RangeExpr LessChainOperator RangeExpr

LessChainExpr LessChainOperator RangeExpr

LessChainOperator: 以下之一

`<` `==` `⩽`

GreaterChainExpr:

RangeExpr GreaterChainOperator RangeExpr

GreaterChainExpr GreaterChainOperator RangeExpr

GreaterChainOperator: 以下之一

`>` `==` `⩾`

$$
\begin{aligned}
a \texttt{ == } b &\iff a \texttt{ cmp } b = \texttt{.equal} \\
a \texttt{ != } b &\iff a \texttt{ cmp } b \neq \texttt{.equal} \\
a \texttt{ < } b &\iff a \texttt{ cmp } b = \texttt{.less} \\
a \texttt{ > } b &\iff a \texttt{ cmp } b = \texttt{.greater} \\
a \texttt{ ⩽ } b &\iff a \texttt{ cmp } b = \texttt{.less} \text{ or } \texttt{.equal} \\
a \texttt{ ⩾ } b &\iff a \texttt{ cmp } b = \texttt{.greater} \text{ or } \texttt{.equal}
\end{aligned}
$$

[1] a `cmp` b 比较两个表达式，其结果类型为 `Order`。其余比较运算符的结果类型为 `bool`。

[2] `<`、`⩽` 和 `==` 可以连续使用。a `<` b `⩽` c 等价于 a `<` b `&` b `⩽` c。`>`、`⩾` 和 `==` 也可以用类似方式混合。以其他方式在一个表达式中使用超过一个比较运算符是一个编译错误。

4.12.2 包含运算符 [expr.include]

IncludeExpr:

RangeExpr `in` *RangeExpr*

RangeExpr `!in` *RangeExpr*

[1] a `in` b 检测 a 是否在 b 中。a `!in` b 等价于`!(`a `in` b`)`。表达式的类型为 `bool`。

[2] [注：`!in` 中`!` 和 `in` 之间必须没有空白，否则会被识别成两个分开的运算符。]

4.12.3 类型转换运算符 [expr.cast]

CastExpr:

SuffixExpr `as` *Type*

SuffixExpr `as?` *Type*

SuffixExpr `as!` *Type*

[1] e `as` T 运算符用于进行显式的类型转换，结果的类型为 `T`。

2 e `as?` T 用于进行可能失败的类型转换，结果类型为可空类型或结果类型。e `as!` T 用于进行强制类型转换，其等价于 `(`e `as?` T`)!`。

3 [注：`as?` 和 `as!` 中 `as` 和后面的符号之间必须没有空白，否则会被识别为两个分开的运算符。]

4.12.4 匹配运算符 [expr.match]

MatchExpr:
RangeExpr `is` *Pattern*
RangeExpr `!is` *Pattern*

1 a `is` p 检测 a 是否匹配模式 p。a `!is` p 检测 a 是否不匹配模式 p。表达式的类型为 `bool`。模式中不能包含绑定模式。

2 [注：`!is` 中`!` 和 `is` 之间必须没有空白，否则会被识别成两个分开的运算符。]

4.13 逻辑运算符 [expr.logic]

LogicExpr:
BooleanExpr
LogicExpr `&` *BooleanExpr*
LogicExpr `|` *BooleanExpr*

1 `&` 和 `|` 是逻辑运算符，它们都使用短路求值。在同一个表达式中混合使用两个运算符是一个编译错误。

4.14 赋值运算符 [expr.assign]

AssignExpr:
LogicExpr
SuffixExpr `=` *LogicExpr*
SuffixExpr `+=` *LogicExpr*
SuffixExpr `-=` *LogicExpr*
SuffixExpr `*=` *LogicExpr*
SuffixExpr `/=` *LogicExpr*
SuffixExpr `%=` *LogicExpr*
SuffixExpr `shl_eq` *LogicExpr*
SuffixExpr `shr_eq` *LogicExpr*
SuffixExpr `'&=` *LogicExpr*
SuffixExpr `'^=` *LogicExpr*
SuffixExpr `'|=` *LogicExpr*
SuffixExpr `??=` *LogicExpr*
SuffixExpr `~=` *LogicExpr*

1 赋值表达式的结果类型是 `void`。

2 `=` 将左操作数的值更新为右操作数的值。左操作数必须是可写的，但可以是不可读的。如果左操作数并非单个位置表达式，则使用类似模式匹配的规则进行赋值。

[3] 复合赋值运算符 `+=`、`-=`、`*=`、`/=`、`%=`、`shl_eq`、`shr_eq`、`'&=`、`'^=`、`'|=`、`~=` 和`??=` 是对应二元运算符的赋值简化。对这些运算符而言，a op`=` b 或 a op`_eq` b 等价于 a `=` a op b，但 a 只被求值一次。复合运算符的左操作数必须是可读可写的。

5 语句 [stmt]

Statement:
Block
IfStatement
MatchStatement
WhileStatement
ForStatement
BreakStatement
ContinueStatement
ReturnStatement
ThrowStatement
AssertStatement

1 语句是一类特殊的表达式。

2 语句是块的构成部分。如果语句包含子块，则这个块将优先作为语句的构成部分而不是其中的表达式的一部分。[例：

```
// 错误：{ true } 被认为是 if 语句的第一个子块，而不是它的条件表达式的一部分
if x.filter{ true }
```

]

5.1 块 [stmt.block]

Block:
`{` *BlockItems* `}`

BlockItems:
BlockItemsNoExpr Expression`*`

BlockItemsNoExpr:
BlockItem
BlockItemsNoExpr BlockItem

BlockItem:
Expression `;`
BlockDecl
Statement

1 块是由大括号包裹的一系列声明和表达式的序列。块定义了一个块作用域。块的求值按顺序进行。如果最后一项为不带分号的表达式，整个块的值为最后一项的值；否则块的类型为 `void`。

5.2 `if` 语句 [stmt.if]

IfStatement:
`if` *Condition IfClause*
`if` *Condition IfClause* `else` *Block*
`if` *Condition IfClause* `else` *IfStatement*

Condition:
Expression
Binding

IfClause:
Block
`then` *Expression*

1 *`if` 表达式*进行条件分支。条件后必须跟一个块，除非使用上下文关键字 `then` 进行分隔。如果 `if` 语句的条件成立，对第一个表达式（或子块）求值。否则，若有 `else` 子句，执行 `else` 子块。`else` 子块允许直接的 `if` 语句串联。

2 如果条件为表达式 e[1]，那么这个表达式的类型必须实现了 `Condition`。确定条件真假时，连续调用 `cond()` 直到得到一个 `bool` 值。条件成立当且仅当该值为真。

3 如果条件是绑定，那条件成立当且仅当绑定成功。该绑定必须可以失败。

4 `if` 表达式的类型是其两个分支表达式的公共类型。如果 `else` 分支被省略，则视为 `void`。

5.3 `match` 语句 [stmt.match]

MatchStatement:
`match` *Expression MatchBlock*

MatchBlock:
`{` *BlockItem*`*` *MatchItem*`+` `}`

MatchItem:
Pattern → *Statement*

1 *`match` 语句*对其后跟的表达式进行模式匹配。

2 `match` 语句的各项中的模式必须覆盖被匹配表达式的所有可能值，否则这是一个编译错误。如果该表达式的某可能值包含类型为 `never` 的项，则可以省略该项。

[例：

```
enum E {
    A(int),
    B(never),
}
```

1) 因为赋值表达式的类型是 `void`，形如 `p = e` 的程序文本将始终被看做一个模式匹配而不是赋值表达式。

```
func foo(e: E) {
    match e {
        .A(_) ⇒ { ... },
        .B(_) ⇒ { ... }, // 可以省略，不是编译错误
                         // 包含该项同样也不是编译错误，但可能导致不可达代码警告
    }
}
```

]

3 对 `match` 语句的求值将按如下顺序进行：

(3.1) — 如果语句匹配块之前有项，执行这些项。他们的作用域是整个块。

(3.2) — 按出现顺序对每个项进行匹配。如果某个项的模式匹配成功，则执行其后的语句。所有其他项都不会进行求值。

`match` 表达式的类型为其所有分支的公共类型。

5.4 `while` 语句 [stmt.while]

WhileStatement:
`while` *SymbolLiteral*`?` *Condition*`?` *Block*
`while` *SymbolLiteral*`?` *Condition*`?` *Block* `else` *Block*

1 *`while` 语句*处理循环。`while` 语句每次循环都会对控制表达式进行求值。如果求值为真，则继续循环，否则终止循环。如果表达式被省略，则等价于表达式为 `true`。

2 如果 `while` 语句包含一个 `else` 块，则在循环结束时执行该块，并将该块的值作为整个语句的值。

3 `while` 语句的类型为语句主体块包含的所有终止该循环的 `break` 语句值的类型以及 `else` 块的类型的公共类型。如果语句不含 `else` 块，则主体块的类型为 `void`。如果语句的条件被省略，则主体块的类型为 `never`。`else` 块的类型将被忽略。

4 `while` 语句可以带有一个符号字面量作为标签，用于在嵌套循环中终止指定的循环。[注：`while` 之后紧跟的符号字面量总是被解析为一个标签。如果你需要在循环条件中使用符号字面量，请使用括号将其包裹起来。]

5.5 `for` 语句 [stmt.for]

ForStatement:
`for` *SymbolLiteral*`?` `await`*?* *Pattern* `:` *Expression Block*
`for` *SymbolLiteral*`?` `await`*?* *Pattern* `:` *Expression Block* `else` *Block*

1 *`for` 语句*进行明确的范围循环。形如 `for` p e `:` B 的 `for` 语句需满足：e 实现了 `Sequence` 且 `typeof(`e`)::Item` 匹配 p 不会失败，否则这是一个编译错误。p 中注入的变量在整个 `for` 语句的范围内生效。

2 如果 `for` 语句包含一个 `else` 块，则在循环结束时执行该块，并将该块的值作为整个语句的值。

3 `for` 语句的类型为语句主体块包含的所有终止该循环的 `break` 语句值的类型以及 `else` 块的类型的公共

类型。如果语句不含 `else` 块，则视为其类型为 `void`。

4 `for` 语句将被展开为如下形式：

```
{
    let mut i = e.iter;

    while let v = i.next(); v =/= nil {
        p = v;
        B
    } /* else E */
}
```

其中 i 和 v 是内部使用的匿名变量名称。参见 21。

5 与 `while` 语句相同，`for` 语句也可以带有标签。

6 `for` 语句可以带有一个 `await` 标识，代表该语句进行异步迭代。

5.6 控制语句 [stmt.control]

BreakStatement:
    `break` *SymbolLiteral*? *Expression*?

ContinueStatement:
    `continue` *SymbolLiteral*?

ReturnStatement:
    `return` *Expression*?

ThrowStatement:
    `throw` *Expression*?

1 控制语句包括 `break` 语句、`continue` 语句、`return` 语句和 `throw` 语句。

2 `break` 语句只能在 `while` 或 `for` 语句中使用。它终止最内层的循环语句，将控制流移动到该语句之后。`break` 语句可以带有一个表达式，此时该表达式将作为整个循环语句的值被返回。如果它带有一个符号字面量，则它会终止具有对应符号字面量的循环语句。如果没有对应的符号字面量，则这是一个编译错误。[注：如果希望将符号字面量作为返回表达式，则需要使用括号将其包裹起来。]

3 `continue` 语句只能在 `while` 或 `for` 语句中使用。它终止最内侧循环语句的本次循环。将控制流移动到该语句的下一次循环开始。如果它带有一个符号字面量，则它会终止具有对应符号字面量的循环语句。如果没有对应的符号字面量，则这是一个编译错误。

4 `return` 语句只能在函数作用域（2.1.2）中使用。它中止当前函数的执行，并将后跟的表达式作为整个函数的返回值。如果表达式被省略，则等价于 `()`。

5 `throw` 语句只能在函数作用域中使用。它终止当前函数的执行，并将后跟的表达式作为异常抛出。如果表达式被省略，除非语句当前处于 `catch` 块中，此时将会重新抛出原异常；否则这是一个编译错误。

6 这些语句的类型都是 `never`。

5.7 `assert` 语句 [stmt.assert]

AssertStatement:
`assert` *Expression*`?`
`assert` *Expression*`?` `:` *Expression*

1 *`assert` 语句*用于在运行时检查表达式是否为真。如果表达式为假，则调用 `panic`。

2 `assert` 语句可以带有一个`:` 分隔的表达式，此时如果表达式为假，则将该表达式的值作为 `panic` 的参数。

3 `assert` 语句的类型为 `void`。

6 模式匹配 [pattern]

Pattern:
    *PatternBody PatternAssertion**

PatternBody:
    NullPattern
    ExprPattern
    BindPattern
    SomePattern
    EnumPattern
    ArrayPattern
    StructPattern
    AltPattern

PatternAssertion:
    TypeAssertion
    IncludeAssertion
    CondAssertion

1 模式匹配用于检验一个值是否符合特定的模式，以及在符合特定的模式时从中提取某些成分。本节中，v 表示值，p 表示模式。值符合特定的模式称为这个值匹配这个模式，记作 $v \parallel p$。

2 模式 p 由模式主体和模式断言构成。模式主体规定匹配的结构与操作，模式断言则对值的特征进行断言。一个主体可以带有任意数量的断言。

6.1 空模式 [pattern.null]

NullPattern:
    `_`

1 空模式能够匹配任意值。匹配成功后，v 的值将被丢弃。

6.2 表达式模式 [pattern.expr]

ExprPattern:
    RangeExpr

1 v 和 e 必须可比较。$v \parallel e$ 当且仅当 $v == e$。

2 [注：表达式模式的语法可能与其他模式产生歧义。此时总是优先解析为其他模式。]

6.3 `some` 模式 [pattern.some]

SomePattern:
    `some` *Pattern*

1 `some` 模式匹配可空类型。如果该值为非空，则匹配成功。

6.4 枚举模式 [pattern.enum]

EnumPattern:
    EntityID EnumPatternPayload?
    `.` *Identifier EnumPatternPayload?*

EnumPatternPayload:
    ArrayPattern
    StructPattern

1 枚举模式匹配枚举项。如果 v 是 p 中的枚举项，且其载荷匹配 p 中的载荷模式（如果有），则匹配成功。

2 枚举模式中的枚举符可以使用匿名枚举符，此时会进行静态推导。

6.5 数组模式 [pattern.array]

ArrayPattern:
    `[` *AnyPattern* `(,` *AnyPattern*`)*` `]`

AnyPattern:
    Pattern
    `...`
    BindKeyword `...` *Identifier*

1 数组模式匹配序列中的元素。其中 `...` 项（称作*任意项模式*）只能出现至多一次，否则这是一个编译错误。v 必须实现 `Sequence`，否则这是一个编译错误。

1. 如果模式不包含任意项，且 v`.size` 与模式中项的数量不相等，则匹配失败。
2. 如果模式包含任意项，且 v`.size` 小于模式中非任意项的数量，则匹配失败。

2 在那之后，将按如下规则依次对 v 的元素进行匹配。如果每个匹配都成功，则整个模式 p 匹配 v。

1. 对任意项模式之前的模式（如果不存在任意项则对每个子模式），p_i 匹配 v`[`i`]`，其中 i 是子模式的索引（从 0 开始）。
2. 对任意项模式之后的模式，p_r 匹配 v`[$-`r`]`，其中 r 是子模式从后向前数的索引（从 0 开始）。

3 如果任意项包含一个绑定，则该任意项匹配到的所有元素将被绑定到相应的标识符上。该标识符将具有与 v 相同的类型。

6.6 结构模式 [pattern.struct]

StructPattern:
    `(` `)`
    `(` *StructPatternBody* `,?` `)`

StructPatternBody:
    StructPatternItem
    StructPatternBody `,` *StructPattern*

StructPatternItem:
Pattern
Identifier `:` *Pattern*
`...` *Pattern?*

1 结构模式对结构进行匹配。如果对于每个对 (k, p_k) 而言，v`.`k 匹配 p_k 都成立，则整个模式匹配成功。

2 如果结构模式中存在没有匹配的字段，且不包含任意项模式，则这是一个编译错误。

3 结构模式的任意项匹配所有未被其他项匹配的字段。如果该模式被绑定到一个标识符，则其对应的值为这些剩余字段的集合。

6.7 绑定模式 [pattern.bind]

BindPattern:
BindKeyword PatternBind

BindKeyword:
`let`
`let mut`
`let const`

PatternBind:
Identifier PatternAssertion
SomePatternBind
ArrayPatternBind
StructPatternBind

SomePatternBind:
`some` *AnyPatternBind*

ArrayPatternBind:
`[` *AnyPatternBind* `(,` *AnyPatternBind*`)*` `]`

AnyPatternBind:
PatternBind
`...` *Identifier?*
NullPattern
ExprPattern

StructPatternBind:
`(` `)`
`(` *StructPatternBodyBind?* `)`

StructPatternBodyBind:
StructItemBind
StructPatternBodyBind `,` *StructItemBind*

StructItemBind:
Identifier `:` *AnyPatternBind*
AnyPatternBind

1 绑定模式可以匹配任意值。匹配成功后，该标识符将作为一个变量插入到当前作用域中。

2 如果绑定使用关键字 `const`，则这是一个常量绑定。参见 3.10.2。

3 绑定模式可以使用简写，表 7 列出了一些常见的简写形式。

表 7 — 绑定简写与其完整形式

`let [a, b]`	`[let a, let b]`
`let (v, _)`	`(let v, _)`
`let [x, ...y]`	`[let x, let... y]`

6.7.1 选择模式 [pattern.alt]

AltPattern:
Pattern `|` *Pattern*
AltPattern `|` *Pattern*

1 选择模式同时匹配多个模式。如果其中有模式匹配成功，则整个模式匹配成功。匹配将从左到右进行。

2 选择模式各分支可以包含绑定模式，但绑定的变量必须在每个分支中都出现且类型相同，否则这是一个编译错误。

6.8 类型断言 [pattern.type]

TypeAssertion:
`is` *Type*
`:` *Type*
`as` *Type*

1 类型断言对值的类型进行约束。它包括以下类型：

(1.1) — `is T` 要求值的类型与 `T` 完全一致。

(1.2) — `: T` 要求值的类型是 `T` 的子类型。

(1.3) — `as T` 要求值的类型能够转换到 `T`，无论显式或隐式。

6.9 包含断言 [pattern.include]

IncludeAssertion:
`in` *Expression*

1 包含断言要求值包含在某个集合 e 中。如果 v `!in` e，则匹配失败。

6.10 条件断言 [pattern.cond]

CondAssertion:
`if` *Expression*

1 条件断言要求值满足某个条件。

7 声明 [decl]

Declaration:
BlockDecl

BlockDecl:
Binding
FuncDecl
TypeDecl
ClassDecl
EnumDecl
TraitDecl
ImplDecl
OperatorDecl
MacroDecl
StaticAssertion
QualifierDirective

1 声明将名称插入当前作用域，或者使用特定的程序结构实现程序功能。声明在插入名称时也可能定义了该名称对应的程序项。

7.1 绑定 [decl.binding]

Binding:
BindKeyword PatternBind TypeNotation? `=` *Expression* `;`
BindKeyword PatternBind TypeNotation? `=` *Expression* `else` *Block*

TypeNotation:
`:` *Type*

1 绑定形如 `let` p `=` e，其中 p 是绑定模式。

2 绑定将一个绑定插入当前作用域中。该绑定必须不能失败。绑定可以显式指定类型。

3 绑定可以带有一个 `else` 块，此时绑定可以失败，且失败时会执行该 `else` 块。该块的类型必须是 `never`。只能在块作用域中使用带 `else` 块的绑定。

7.2 类型声明 [decl.type]

TypeDecl:
TypeQual`*` `type` *Identifier TypeBody*

TypeDeclName:
Identifier
`self`

TypeQual:
    `const`

TypeBody:
    StructType
    `{ }`
    `{` *StructTypeList* `,?` `}`
    `=` *Type*

1 类型声明用于为类型创建别名。在类型声明之后，可以使用该名称代表所关联的类型。在类型声明中，可以使用大括号代替结构类型的小括号。

7.3 类声明 [decl.class]

ClassDecl:
    ClassQual`*` `class` *Identifier ClassBody*

ClassQual:
    `const`

ClassBody:
    StructType
    `{ }`
    `{` *StructTypeList* `,?` `}`
    `=` *Type*

1 类声明创建不透明类型。`class` *C* *B* 等价于 `type` *C* `=` `class` *B*。参见 3.3。在类声明中，可以使用大括号代替结构类型的小括号。

7.4 静态断言 [decl.static-assert]

StaticAssertion:
    `static assert` *Expression* `;`
    `static assert` *Expression* `:` *Expression* `;`

1 静态断言用于在编译时对条件进行检查。如果条件为假，则会产生一个编译错误。静态断言可以在函数、模块、枚举定义、特征定义、实现定义中出现。

2 静态断言可以带有一个可选的错误消息。如果提供了错误消息，当条件为假时，编译器会输出该错误消息。

3 静态断言的条件和错误消息都必须是常量。

7.5 限定符指令 [qual.dir]

QualifierDirective:
    DirectiveQualifiers `::`

DirectiveQualifiers:
    DirectiveQualifier`+`

DirectiveQualifier:
AccessQualifier

1 可以使用限定符后跟 `::` 的方式为多个声明指定限定符。直到下一个限定符指令或该声明作用域结束为止，所有出现的声明都会受所指定的修饰符修饰。忽略无效的限定符。

[例：

```
public:: // 后面所有声明都是 public 的
type A = int;
const let size = 0;
```

8 函数 [func]

FuncDecl:
    *FuncQual** `func` *FuncName Parameter ThrowQual*? *ReturnType*? *Block*
    *FuncQual** `func` *FuncName Parameter ThrowQual*? `⇒` *Expression* `;`

FuncName:
    UnqualID

FuncQual:
    `async`
    `const`
    `extern`
    `unsafe`

ReturnType:
    `→` *Type*

1 函数声明将函数、方法或闭包引入当前作用域。函数声明由关键字 `func` 标记，后跟函数名、参数列表、异常修饰符、返回类型和函数体。

2 在同一个作用域中可以定义两个名称相同的函数，但是这些函数必须具有不同名称的命名参数（忽略顺序）。如果两个函数在省略某些具有默认值的命名参数之后，剩余的命名参数名称完全相同（忽略顺序），则这也是一个编译错误。

3 函数的返回类型可以省略，此时函数的返回类型为函数体的类型与函数中 `return` 语句参数类型的公共类型。如果函数包含异常修饰符，则返回类型还会继续受到异常修饰符的影响变为错误类型。参见 9.4。

4 函数体为单个块或者 `⇒` 后跟任意表达式和分号。如果采用后一种形式，则不能显式指定返回类型。

5 实现中的函数可以为*方法*。参见 13.2。

6 函数中可以有另一个函数声明。此时如果内部函数使用了外部函数作用域中的值，该函数为*闭包*，否则为普通函数。

8.1 函数参数 [param]

Parameter:
    `(` *ParamList*? `)`

ParamList:
ThisParamDecl
ThisParamDecl `,` *NamedParamList*
ThisParamDecl `,` *UnnamedParamList*
ThisParamDecl `,` *UnnamedParamList* `,` *NamedParamList*
UnnamedParamList
UnnamedParamList `,` *NamedParamList*
NamedParamList

UnnamedParamList:
UnnamedParamDecl
UnnamedParamList `,` *UnnamedParamDecl*

NamedParamList:
NamedParamDecl
NamedParamList `,` *NamedParamDecl*

UnnamedParamDecl:
Attribute`*` *ParamQual*`?` *Pattern* *TypeNotation*`?`

NamedParamDecl:
Attribute`*` *ParamQual*`*` `(` *Identifier* `)` *Pattern* *TypeNotation*`?` *DefaultValue*`?`

ThisParamDecl:
Attribute`*` `mut?` `this`
Attribute`*` `&` `mut?` `this`
Attribute`*` `this` *TypeNotation*

ParamQual:
`mut`
`ref`
`lazy`
`borrow`

TypeNotation:
`:` *Type*

DefaultValue:
`=` *Expression*

[1] 参数为调用函数时传入其内的值。如果函数不接受任何参数，则必须使用 `()` 标识而不能省略括号。函数参数默认是不可变的，除非由 `mut` 修饰。

[2] 参数可以为 `this` 参数、顺序参数或具名参数，且必须按照此顺序排列，并且 `this` 参数不能出现超过一次。顺序参数和具名参数必须显式指定其类型，不能省略类型。具名参数可以指定默认值。如果函数调用时没有提供对应的值，则视为传入该默认值。

[3] 函数参数可以是一个模式，但视为处于绑定模式中，其中所有的标识符都视为一个绑定。这个模式不能是可失败的。

[例：

```
func foo((a, b): (int, bool)) {
    // a is int, b is bool
}
```

]

8.2 参数修饰符 [param.qual]

1 参数修饰符修饰函数参数，指定参数的某些属性。

8.2.1 `mut` [param.qual.mut]

1 `mut` 修饰的参数是可变的，可以在函数内部被修改或重新赋值。[注：由于参数总是移动传递，所以除非使用 `ref` 修饰，否则调用处实参的值不会受影响。]

8.2.2 `ref` [param.qual.ref]

1 `ref` 修饰的参数以引用方式传递，调用处实参的值会受影响。除非使用 `mut` 修饰，否则参数是只读的。

2 类型为 T 的 `ref` 参数实际上是类型为 T`&`（或 T `mut&`）的引用参数的语法糖。在函数内部使用该参数时，将会自动对参数进行解引用。

3 `ref` 参数将会按如下方式脱糖：

```
func foo(ref a: int) {
    a = 1;
}

let mut a = 0;

foo(ref a);
```

变为

```
func foo(r_a: int mut&) {
    *r_a = 1;
}

let mut a = 0;

foo(&mut a);
```

8.2.3 `lazy` [param.qual.lazy]

1 `lazy` 修饰的参数是惰性求值的，只有在函数内部使用时才会计算实参的值。`lazy` 不能与 `mut` 修饰符一起使用。

2 类型为 T 的 `lazy` 参数实际上是类型为 `() once` $\to$ T 的闭包参数的语法糖。在函数内部首次读取该参数时，将会调用该闭包并将结果缓存，之后再次读取时直接使用缓存的值。

3 `lazy` 参数将会按如下方式脱糖：

```
func foo(lazy a: int) {
    bar(a);
}

let a = 0;

foo(a);
```

变为

```
func foo(c_a: () once → int) {
    let s_a: int? = nil;
    let g_a = do {
        if let some v_a = s_a {
            v_a
        } else {
            let v_a = c_a();
            s_a = some v_a;
            v_a
        }
    };

    bar(g_a());
}

let a = 0;

foo(do { a });
```

8.2.4 `borrow` [param.qual.borrow]

1 `borrow` 修饰的参数为借用参数，它会将实参移入到函数中，然后在函数返回时将其返回并覆盖原实参。`borrow` 不能与 `lazy` 和 `ref` 修饰符一起使用。

2 `borrow` 参数将会按如下方式脱糖：

```
func foo(borrow a: int) {
    let a = 1;
}

let a = 0;
```

```
foo(borrow a);
```

变为

```
func foo(a: int) → (void, int) {
    let a = 1;
    ((), a)
}

let a = 0;

let (_, a) = foo(a);
```

8.3 异步函数 [func.async]

1 使用 `async` 修饰的函数称为异步函数。

9 异常处理 [except]

9.1 概述 [except.intro]

1 X 将程序中出现的问题分为一般错误与致命错误。一般错误能被上层代码处理，例如打开文件时文件不存在；致命错误则无法被处理，例如内存耗尽。代码中的一般错误可以使用异常机制来处理。致命错误应该调用 `panic` 直接终止程序的执行。参见 9.5。

9.2 抛出异常 [except.throw]

1 `throw` 表达式（参见 5.6）抛出异常。这等价于函数以结果类型的错误值返回。在没有 `throw` 修饰符的函数中使用 `throw` 表达式是一个编译错误。

2 `throw` 表达式的参数必须实现了 `ErrorCode`。

3 不带参数的 `throw` 表达式只能在 `catch` 块中使用。它重新抛出当前捕获的异常。

9.3 处理异常 [except.catch]

TryExpr:
    `try` *Expression* *CatchBlock*`+` *TryElseBlock*`?`
    `try` *Expression*

CatchBlock:
    `catch` *Pattern*`?` *Block*

TryElseBlock:
    `else` *Pattern*`?` *Block*

1 `try` 表达式可以解构结果类型 T `!!` E。如果其子表达式为正确值 e，则整个表达式的值就为 e。否则，将错误值与每个 `catch` 子句的模式进行匹配。如果找到了匹配的块，则整个表达式的值就为该块的值。如果没有找到匹配的块，则异常会继续向上抛出，等价于一个隐含的 `catch { throw }`。

2 `try` 表达式也可以包含一个 `else` 块。在这种情况下，`else` 块的值将取代子表达式成为正确时整个表达式的值。匹配 `else` 块模式的是子表达式的正确值。

3 `try` 表达式的类型 T 与所有 `catch` 块的类型的公共类型。

4 如果一个异常传播到了最顶层函数的最外层，将会调用 `core::terminate`。

9.4 函数 `throw` 修饰符 [except.func.throw]

ThrowQual:
    `throw (` *TypeList*`*` `)`
    `throw`

TypeList:
Type
TypeList `,` *Type*

1 `throw` 修饰符显式指定了函数会产生什么类型的异常。`throw` 后面的括号标记了可能抛出的异常类型。如果括号内部为空，则代表该函数不抛出任何异常，其异常类型为 `never`。如果括号被省略，将会自动推导函数抛出的异常类型。这些类型必须实现了 `ErrorCode`。

2 如果函数指定了 `throw` 修饰符，且修饰符指定或推导的类型为 E，其显式指定或推导的返回值为 R，则该函数实际的返回值为 R `!!` E。

9.5 `panic` [except.panic]

1 `core::panic` 在程序遇到无法处理的错误时终止程序的执行。参见 20.5.3。

10 枚举 [enum]

EnumDecl:
    `enum` *Identifier EnumBaseType*`?` `{` *StaticAssertion*`*` *Enumerators* `}`

EnumBaseType:
    `:` *Type*

Enumerators:
    Enumerator `(` `,` *Enumerator* `)*` `,?`

Enumerator:
    Attribute`?` *Identifier EnumeratorTail*`?`

EnumeratorTail:
    `=` *Expression*
    `[` *Type* `]`
    StructType

1 枚举类型用来表示一组孤立值，在其定义中使用枚举符表示。枚举符还可以带有载荷，以表示同一枚举符下的一系列值。

[例：

```
enum E {
    A,
    B(int),
    C[int],
    D(name: string)
}
```

上述代码定义了一个枚举类型 `E`，它包含四个枚举符，可以以如下方式访问：`E.A`、`E.B(0)`、`E.C[1, 2, 3]` 及 `E.D(name: "Hello")`。]

2 枚举类型定义中可以在所有枚举符之前包含任意数量的静态断言。

10.1 传统枚举类型 [enum.trad]

1 只包含单独枚举符的枚举类型称作*传统枚举类型*。传统枚举类型可以指定基底类型 B，也可以为其枚举符指定值。如果一个传统枚举类型没有显式指定基底类型，则 B 为 `int`。B 必须实现 `Equtable`。

2 在传统枚举类型中，每个枚举符都有对应的值。其确定如下：

(2.1) — 如果该枚举符被指定值，则其值为被指定的表达式隐式转换到 B 的结果；

(2.2) — 否则，如果该枚举符是第一个值，且基底类型实现了 `core::Default`，则其值为 B`::default()`；

(2.3) — 否则，假设该枚举符的前一个值为 v，则其值为 v**+!**。

每个枚举符都可以显式转换为对应的基底类型。

11 运算符 [op]

11.1 运算符名称 [op.name]

OperatorName:
Operator
Identifier
OperatorKeyword
`( )`
`[ ]`

OperatorKeyword: 以下之一
`shl shr cmp in`

OperatorType: 以下之一
`infix prefix suffix`

Operator:
CustomOperator
`;`

CustomOperator:
`'?` *OperatorSymbol*+

OperatorSymbol: 以下之一
`~ ! % ^ & * - | + = / ? < > .`

1 表 8 列出了全部内建运算符，其中上半部分为可以重载的运算符，下半部分为不能重载的运算符。

表 8 — 内建运算符

`+!`	`-!`	前缀 `+`	前缀`-`	前缀`!`
`'~`	前缀 `*`	`?`	后缀`!`	`*`
`/`	`%`	二元 `+`	二元`-`	`shl`
`shr`	`'&`	`'^`	`'|`	`..`
`..=`	`~`	`=`	`≠`	`<`
`⩽`	`>`	`⩾`	`cmp`	`in`
`&`	`|`			
`.`	`await`	前缀 `&`	`!in`	`is`
`!is`	`=`	`+=`	`-=`	`*=`
`≠`	`%=`	`shl_eq`	`shr_eq`	`'&=`
`'^=`	`'⊨`	`??=`	`++`	`--`
`~>`	`<~`	`;`		

11.2 自定义运算符 [op.user]

OpeartorDecl:

`operator` *CustomOperator OperatorSpecifier OperatorTraitBinder* `;`

`operator` *Identifier OperatorSpecifier OperatorTraitBinder* `;`

OperatorSpecifier:

`prefix`

`suffix`

`infix`

`infix (` *OperatorName* `)`

`infix (` *OperatorPrecedence* `,` *OperatorPrecedence* `)`

OperatorPrecedence:

OperatorName

`_`

OperatorTraitBinder:

`=` *PathList*

PathList:

Path

PathList `,` *Path*

1 用户可以声明自己的运算符。自定义运算符需要指定一个或多个特征方法，实现此特征的类型可以使用这个运算符，且等价于调用该方法。

2 自定义的运算符不能与内建的运算符相同，也不能是→、⇒ 或 `...`。自定义运算符不能包含`/*`、`*/`、`//`序列作为其一部分。如果自定义的运算符是一个标识符，则它不能在任何作用域中与另一个标识符相同，否则这是一个编译错误。自定义的运算符也不能是 `prefix`、`suffix` 或 `infix`。

3 自定义的运算符可以是前缀、后缀或者中缀的，分别使用 `prefix`、`suffix` 和 `infix` 指定。后缀与前缀运算符不能指定优先级。后缀运算符的优先级总是高于前缀运算符，并高于所有类型的中缀运算符。中缀运算符可以指定其优先级。表 9 展示了可选择的中缀运算符优先级。

表 9 — 中缀运算符优先级

运算符	结合性
`*` `/` `%`	左结合
`+` `-`	左结合
`shl` `shr`	不结合
`'&` `'^` `'|`	左结合，不能混用
`..` `..=`	不结合
`~`	左结合，不能混用
`??`	右结合，不能混用
`==` 等	特殊结合性
`&` `|`	左结合，不能混用
`=` 等	不结合

4 中缀运算符可以指定自己的优先级与某个内建的运算符组相同，或指定两个相邻的运算符组，创建一个自定义的优先级。不能在 `==` 到 `=` 之间创建新的优先级。此外，还可以使用 `(_, *)` 或 `(=, _)` 指定比表中运算符更低或更高的优先级。[注：自定义的优先级无法指定结合性。然而，可以通过自定义运算符的重载函数自行处理优先级。参见11.3.8。] 中缀运算符也可以不指定优先级。这类运算符视为比 `==` 具有更高的优先级。除了未指定优先级的中缀运算符，运算符的优先级关系是线性的。

5 如果中缀运算符的优先级组处于 `==` 组，则其不能与任何内建运算符混用，且其返回值必须为布尔类型。如果中缀运算符处于 `&` 组，则其不能与任何内建运算符混用，且其参数和返回值必须为布尔类型。如果中缀运算符处于 `=` 组，则其不能与任何内建运算符混用，且其返回值必须是 `void` 或 `never`。如果两个自定义运算符具有相同等级的优先级，且不是内建的优先级，则在一个表达式中将其混用是一个编译错误。如果一个自定义运算符未指定优先级，则它与任何优先级高于 `==` 的中缀运算符混用是一个编译错误。

[例：

```
operator >> infix(+) = A::a;
operator << infix(+) = B::b;
operator !! infix(*, +) = C::c;
operator ~~ infix(*, +) = D::d;

a >> b + c; // OK，等价于 (a >> b) + c
a >> b << c; // OK，等价于 (a >> b) << c
a !! b * c; // OK，等价于 a !! (b * c)
a !! b ~~ c; // 错误，不能在同一个表达式中混用具有新优先级的运算符

operator ** infix = E::e;
operator && infix = F::f;

a ** b * c; // 错误，未指定优先级的运算符不能与 * 混用
a = a ** b; // 可以，允许赋值运算符
a ** b && c; // 错误，相互之间也不能混用
```

]

6 一个运算符可以同时定义为前缀、后缀和中缀，但不能定义同一个类别的相同运算符多于一次。如果自定义运算符产生了语法歧义，则这是一个编译错误。

[例：

```
operator ** suffix = A::a;
operator ** infix = B::b;
operator && prefix = C::c;
operator && infix = D::d;

a ** && b; // 错误，无法确定谁为中缀运算符
(a**) && b; // 可以
a ** (&&b); // 可以
```

]

7 解析表达式的流程如下：

(7.1) — 构造基本表达式和运算符。运算符将尽可能长地构造，请在必要的时候使用空格分隔。

(7.2) — 如果有运算符`.`，将它及后面的必需成分一起替换成一个基本表达式。

(7.3) — 如果有运算符 `as` 或 `is`，将它及后面的必需成分一起替换成一个后缀运算符。[注：`is` 运算符之后的模式将会尽可能长地构造。请在需要的情况下加上括号。]

(7.4) — 对每一组连续且多于一个的基本表达式，如果除了第一个以外都是`.`后缀运算、结构字面量、数组字面量或块，将它们替换为后缀运算符。否则，这是一个编译错误。

(7.5) — 确定每个运算符是前缀、后缀还是中缀的。

(7.6) — 对每组连续的表达式-运算符-表达式组，如果存在唯一一个运算符满足：

(7.6.1) — 它可以作为中缀运算符；

(7.6.2) — 它前面的所有运算符都可以作为后缀运算符；

(7.6.3) — 它后面的所有运算符都可以作为前缀运算符。

那么这个运算符是中缀运算符，之前的所有运算符是后缀运算符，之后的所有运算符是前缀运算符。如果存在多于一个或不存在这样的运算符，这是一个编译错误。

(7.7) — 第一个子表达式之前的所有运算符都是前缀运算符。最后一个子表达式之后的所有运算符都是后缀运算符。如果有运算符不是这个类别，这是一个编译错误。

(7.8) — 将基本表达式与其后的后缀运算符替换成一个后缀表达式。

(7.9) — 将后缀表达式与其前的前缀运算符替换成一个前缀表达式。

(7.10) — 此时，表达式将成为子表达式与中缀运算符交替的链，且总以子表达式开头和结尾。

(7.11) — 如果表达式中同时包含未指定优先级的运算符和另一个优先级高于 `==` 的运算符，这是一个编译错误。

(7.12) — 运算符按照优先级组依次选择其操作数，并结合成为子表达式。如果此时存在非法的运算符混用，这是一个编译错误。

(7.13) — 如果此时仍然剩余多于一个子表达式，这是一个编译错误。否则，表达式解析完成。

11.3 运算符重载 [op.over]

1 运算符可以通过实现其对应特征的方式重载。如果一个类型为一个运算符实现了多个特征，则将会使用第一个可使用的特征。

[例：

```
trait A { func a(&mut this) → int; }
trait B { func b(this) → int; }

operator ** prefix = A::a, B::b;
```

```
class X = ();

impl X : A { func a(&mut this) ⇒ 1; }
impl X : B { func b(this) ⇒ 2; }

let x = () as X;

**x; // A::a 不适用，调用 B::b，返回 2

let mut x = () as X;

**x; // 调用 A::a，返回 1
```

]

2 可重载的内建运算符也都有其对应的特征方法。所有特征方法都位于模块 `core.ops` 中。

11.3.1 后缀运算符 [op.over.suffix]

1 表 10 列出了可重载的内建后缀运算符对应的特征。

表 10 — 后缀运算符特征

运算符	特征
`[]`	`IndexMut::indexMut`, `Index::index`
`()`	`FunctorMut::callMut`, `Functor::call`, `FunctorOnce::callOnce`
`?`	`Failable::chain`
`!`	`Failable::unwrap`
`+?`	`Successor::next`
`-?`	`Predecessor::prev`
`++`	`Increment::inc`
`--`	`Decrement::dec`
`as`	`Into::into`
`as?`	`TryInto::tryInto`

11.3.2 下标运算符 [op.over.index]

1 下标运算符对应四个特征方法：`Index::index`、`IndexAssign::indexAssign`、`IndexRef::indexRef` 和 `IndexRefMut::indexRefMut`。这四个特征都需要一个函数类型的泛型参数。

2 对表达式 `array[key]`（不作为赋值的左操作数），它等价于 `*array.indexRef(key)` 和 `array.index(key)` 中先成立的一个。

3 对表达式 `array[key] = value`，它等价于 `*array.indexRefMut(key) = value` 和 `array.indexAssign(key, value)` 中先成立的一个。

11.3.3 函数调用运算符 [op.over.call]

1 函数调用运算符有三个对应的特征方法:`FunctorMut::callMut`、`Functor::call` 和 `FunctorOnce::callOnce`，分别对应可变、不可变和只能调用一次的情况。这三个特征都需要一个函数类型的泛型参数。

11.3.4 空值检测与空值合并运算符 [op.over.null]

1 后缀`?` 和后缀`!` 运算符是使用到了 `FromResidual` 及 `Try` 两个特征的语法糖。

2 e`!` 将被翻译为如下代码：

```
match Try::branch(e) {
    case .Continue(c) ⇒ c,
    case .Break(b) ⇒ return FromResidual::fromResidual(b),
}
```

其中 c 和 b 是内部使用的匿名变量名称。

3 后缀`?` 与后缀`!` 类似，但是残差值不会直接返回而是作为完整表达式的值。

4 a `??` b 将被翻译为如下代码：

```
match Try::branch(a) {
    case .Continue(c) ⇒ c,
    case .Break(_) ⇒ b,
}
```

其中 c 是内部使用的匿名变量名称。

11.3.5 类型转换运算符 [op.over.as]

1 转换运算符 `as` 对应 `Into::into`。转换运算符 `as?` 对应 `TryInto::tryInto`。它们也可能由对应的 `From` 或 `TryFrom` 生成。

2 转换运算符 `as!` 采用内建语义。它不能被显式重载以与 `as?` 采取不一致的语义。

11.3.6 前缀运算符 [op.over.prefix]

1 表 11 列出了可重载的内建前缀运算符对应的特征。

表 11 — 前缀运算符特征

运算符	特征
`+`	`Positive::positive`
`-`	`Negative::negative`
`!`	`Not::not`
`'~`	`Complement::compl`
`*`	`DerefMut::derefMut`, `Deref::deref`

11.3.7 解引用运算符 [op.over.deref]

1 前缀 `*` 运算符对应 `Deref::deref` 和 `DerefMut::derefMut`，分别为不可变和可变版本。其返回类型必须是引用类型。

11.3.8 中缀运算符 [op.over.infix]

1 对中缀运算符 `@` 而言，e_1 `@` e_2 `@` ... `@` e_n 将会调用 `Trait::method(`e_1`,` e_2`,` ...`,` e_n`)` 或 `Trait::method([`e_1`,` e_2`,` ...`,` e_n`])`，取先匹配的那一个。如果没有找到匹配的重载函数，且该运算符所处的优先级组未定义其结合性，则这是一个编译错误。否则，将会按照其结合性拆分为多次以两个参数调用的形式。若仍然无法找到匹配的重载函数，则这是一个编译错误。

2 表 12 列出了可重载的内建中缀运算符对应的特征。

表 12 — 中缀运算符特征

运算符	特征
`*`	`Multiply::multiply`
`/`	`Divide::divide`
`%`	`Modulo::modulo`
`+`	`Add::add`
`-`	`Subtract::subtract`
`shl`	`ShiftLeft::shiftLeft`
`shr`	`ShiftRight::shiftRight`
`'&`	`BitAnd::bitAnd`
`'^`	`BitXor::bitXor`
`'|`	`BitOr::bitOr`
`..`	`RangeTo::rangeTo`
`..=`	`ClosedRangeTo::closedrangeTo`
`~`	`Concat::concat`
`in`	`Include::include`
`&`	`LogicAnd::logicAnd`
`|`	`LogicOr::logicOr`
`+=`	`AddAssign::addAssign`
`-=`	`SubtractAssign::subtractAssign`
`*=`	`MultiplyAssign::multiplyAssign`
`/=`	`DivideAssign::divideAssign`
`%=`	`ModuloAssign::moduloAssign`
`shl_eq`	`ShiftLeftAssign::shiftLeftAssign`
`shr_eq`	`ShiftRightAssign::shiftRightAssign`
`'&=`	`BitAndAssign::bitAndAssign`
`'^=`	`BitXorAssign::bitXorAssign`
`'|=`	`BitOrAssign::bitOrAssign`
`<~`	`AppendRight::appendRight`
`~>`	`AppendLeft::appendLeft`

11.3.9 比较运算符 [op.over.cmp]

1 相等运算符 `==` 与 `Equal::equal` 或 `PartialEqual::partialEqual` 关联。

2 不等运算符≠ 与 `PartialEqual::partialNotEqual` 关联。它提供了默认实现，因此重载了 `==` 的类型不需要显式重载≠。

3 类型总是默认实现 `Equal`。也可以通过显式实现来替换默认的实现。

4 比较运算符 `cmp` 与 `Ordered::compare` 或 `PartialOrdered::partialCompare` 关联。

5 `<`、⩽、`>`、⩾ 分别与 `PartialOrdered::lessThan`、`PartialOrdered::lessEqual`、`PartialOrdered::greaterTh` `PartialOrdered::greaterEqual` 关联。它们提供了默认实现，因此重载了 `cmp` 的类型不需要显式重载这些运算符。

11.3.10 包含运算符 [op.over.in]

1 `in` 运算符与 `Include::include` 关联。与其他二元运算符相反的是，a `in` b 将被翻译为 b`.include(`a`)`。

2 `!in` 的语义总是保持与 `in` 相反。它不能被重载。

11.3.11 逻辑运算符 [op.over.logic]

1 逻辑运算符的重载解析与一般的中缀运算符相同，但参数必须均以 `lazy` 修饰。重载调用时会隐式加入 `lazy` 修饰。

2 自定义的逻辑运算符也必须遵循上述规则。

11.3.12 赋值运算符 [op.over.assign]

1 赋值运算符，及与其相同优先级的运算符的重载函数必须返回 `void` 或 `never`。

11.3.13 `$` 运算符 [op.over.dollar]

1 `$` 表达式将被转换为对当前序列 s 调用 s`.dollar()`。它只能接受一个 `this` 参数。当前序列必须实现 `Dollar`。

12 特征 [trait]

TraitDecl:
*TraitQual** `trait` *Identifier SuperTrait*? *TraitBody*

TraitQual:
`const`

SuperTrait:
`:` *EntityID*
SuperTrait `,` *EntityID*

TraitBody:
`{` *TraitMember** `}`

TraitMember:
TypeConstraint
ConstConstraint
FuncConstraint
PropConstraint
StaticAssertion

TypeConstraint:
`type` *Identifier* `;`
`type` *Identifier* `:` *Type* `;`
`type` *Identifier* `=` *Type* `;`

ConstConstraint:
`const` *Identifier* `;`
`const` *Identifier* `=` *Expression* `;`

FuncConstraint:
*FuncQual** `func` *FuncName Parameter ThrowQual*? *ReturnType* `;`
FuncDecl

PropConstraint:
*PropertyQual** *BindKeyword Identifier TypeNotation*? *PropertyBody* `;`
PropertyDecl

1 特征描述了类型可以满足的抽象接口，包括类型约束、常量约束、函数约束和属性约束。如果这些约束提供了一个具体的类型、值或函数实现，则它会被用作该约束的默认实现。如果一个约束没有提供默认实现，则实现该特征的类型必须提供一个实现。

2 特征可以具有一个或数个父特征。实现特征的类型必须实现其所有父特征。

13 实现 [impl]

ImplDecl:
`impl` *Type* *TraitSpec*`?` *ImplBody*

TraitSpec:
`:` *Type*

ImplBody:
`{` *ImplMember*`*` `}`
`;`
`= never ;`

ImplMember:
TypeDecl
FuncDecl
PropertyDecl
StaticAssertion

1 实现用于为类型添加额外的特性。实现可以显式指定以令类型满足某个特征，也可以不指定特征而为类型添加任意特性。

2 不指定特征的实现称为固有实现。一个类型可以拥有多个固有实现，但必须与该类型定义在相同的模块中。

3 指定了特征的实现称为特征实现。一个类型只能拥有一个特征的一个实现，但不要求该实现必须在类型或特征定义所在的模块中。该实现必须为特征中所有未提供默认实现的项提供实现。实现可以替换特征中已存在默认实现的项的实现。特征实现可以包含未在特征中定义的项，但是这些项不允许包含可见性修饰符且可见性总为 `private`。

13.1 属性 [impl.prop]

PropertyDecl:
PropertyQual`*` *BindKeyword* *Identifier* *TypeNotation*`?` *PropertyBody* `;`
PropertyQual`*` *BindKeyword* *Identifier* *TypeNotation*`?` `⇒` *Expression* `;`

PropertyQual:
AccessQual

PropertyBody:
`{` *PropertyMember*`+` `}`

PropertyMember:
PropertyQual`*` *PropertyKeyword* *PropertyBlockParam*`?` *Block*
PropertyQual`*` *PropertyKeyword* *PropertyExprParam*`?` `⇒` *Expression* `;`
PropertyQual`*` *PropertyKeyword* `;`

PropertyKeyword: 以下之一
`get set willSet didSet`

PropertyBlockParam:
Identifier
Identifier `,` *Identifier*

PropertyExprParam:
Identifier
`(` *Identifier* `,` *Identifier* `)`

1 实现可以给类型添加属性。属性的定义至少需要包含一个访问器。访问器的块或表达式具有 lambda 作用域，且隐含 `this` 参数。

2 属性的访问器可以以上下文关键字 `get`、`set`、`willSet` 或 `didSet` 开始。`get` 访问器不接受任何参数。`set` 访问器接受一个参数，其类型为该属性的类型。`willSet` 和 `didSet` 访问器可以接受一个或两个参数，其类型为属性的类型。

3 如果 `set` 访问器不显式写出参数，则视为其具有 lambda 参数 `$value`。如果 `willSet` 或 `didSet` 不显式写出参数，则视为其具有 lambda 参数 `$oldValue` 和 `$newValue`。

4 如果属性以直接后跟 $\Rightarrow$ e 的方式声明，则视作该属性具有访问器 `get` $\Rightarrow$ e。

5 未使用 `mut` 声明的属性不能包含 `set`、`willSet` 和 `didSet` 访问器。在考虑自动生成的访问器之后，如果一个属性缺少 `get` 访问器，或一个 `mut` 属性缺少 `set` 访问器，则这是一个编译错误。

6 属性可以具有类型提示或初始化器。如果属性未按前文所述具有对应的字段，则不能具有初始化器。如果属性省略类型提示，则其类型将从 `get` 访问器中推导。如果它的 `get` 访问器是自动生成的，这是一个编译错误。

7 当读取属性 p 时，会调用 `get` 访问器并将其返回值作为 p 的新值。

8 当给属性 p 赋值时，首先将该值隐式转换到属性的类型，令结果为 v：

(8.1) — 如果属性具有 `willSet` 访问器：

(8.1.1) — 如果它接受一个参数，将以 v 调用；

(8.1.2) — 如果它接受两个参数，将以 p（旧值）和 v 调用。

(8.2) — 以 v 调用属性的 `set` 访问器。

(8.3) — 如果属性具有 `didSet` 访问器：

(8.3.1) — 如果它接受一个参数，将以 p（旧值）调用；

(8.3.2) — 如果它接受两个参数，将以 p 和 v 调用。

(8.3.3) — 如果函数体内没有使用新值，则不会调用 `get` 访问器并使用第一种形式调用。

13.2 方法 [method]

1 实现中可以包含函数声明，其中含有 `this` 参数的称为方法。`this` 参数指代调用该方法的对象。`this` 参数必须出现在参数列表的第一个位置。

2 `this` 参数可以省略类型指示并具有特殊的形式，参见表 13。

表 13 — `this` 参数的形式

形式	类型
`this`	`self`
`mut this`	`self mut`
`&self`	`self&`
`&mut self`	`self mut&`

[例：

```
impl A {
    f(this) { } // #1
    f(&mut this) { } // #2, 可以修改 A 的成员
}
```

]

13.3 构造器 [impl.init]

1 构造器是名称为关键字 `init` 的函数。构造器可以接受任意类型的参数并且返回一个 `self` 类型的值，或者返回一个 `self` 类型的值的同时抛出一个异常。构造器不能是方法。

2 具有泛型参数的构造器可以指定与 `self` 不同的返回类型，这称为参与推导的构造器。

14 泛型 [generic]

GenericSpecification:
`<` *GenericParameters* `>`

GenericParameters:
GenericParameter
GenericParameters `,` *GenericParameter*

GenericParameter:
Identifier `...?` *GenericConstraint?* *GenericIfConstraint?*

GenericConstraint:
`: type`
`:` *Type*
GenericTraitConstraint

GenericTraitConstraint:
`impl` *Type*

GenericIfConstraint:
`if` *Expression*

GenericArguments:
GenericArgument
GenericArguments `,` *GenericArgument*

GenericArgument:
Type
Expression

1 *X* 中的实体可以带有编译器的类型或非类型参数进行泛化。

14.1 高阶类型 [kind]

14.2 `impl` 泛型 [generic.impl]

1 如果 `impl` 被用于一个函数的参数类型中，则其相当于一个匿名的泛型参数。

[例:

```
func f(a: impl A) { }

// 等价于
func<T impl A> f(a: T) { }
```

]

15 类型推导 [deduct]

1 X 可以在适当的地方不提供显式类型，而是让类型自动推导。

15.1 自动推导的静态成员 [deduct.static]

1 使用类型的静态成员和枚举符时，可以省略类型名称。

2 匿名静态成员表达式 e 按照如下方法匹配类型 T：

(2.1) — 如果 T 是枚举类型，且具有枚举符 e，且枚举符参数与 e 匹配，则匹配成功。

(2.2) — 否则，如果 T 具有类型为 T 的静态成员 e，且 e 不含有枚举符参数，则匹配成功。

(2.3) — 否则，如果 e 是函数类型的表达式，且 T 具有名称为 e、返回类型为 T 的静态方法，且 $T.e$ 的参数类型与 e 的兼容，则匹配成功。

(2.4) — 其他情况匹配失败。

[例：

```
enum E { A, B(int), C };

class X {
    static let x = X();
    static func v(i: int) ⇒ X();
    init() { }
}

func f(e: E) { }
func g(x: X) { }

f(.A); // 可以
f(.B); // 错误，枚举符参数不一致
f(.B(0)); // 可以
g(.x); // 可以，等价于 g(X::x)
g(.v(0)); // 可以，等价于 g(X::v(0))
```

]

3 匿名静态成员表达式匹配可以跨越方法调用。对方法调用 $o.f$ 而言，如果 T 有方法 f 其类型与 $o.f$ 兼容，且返回类型为 T，则 $o.f$ 匹配 T 当且仅当 o 匹配 T。

[例：

```
enum E { A, B(int), C };
```

```
  enum F { X, Y, Z };

  impl E {
      func f() ⇒ self::C;
      func g() ⇒ F::X;
  }

  impl F {
      func f() ⇒ E::A;
      func g() ⇒ E::C;
  }

  func f(e: E) { }

  f(.C.f()); // 可以，推导可以穿过方法调用，且只会检查 E.f
  f(.Y.g()); // 错误，E 上的方法 g 不返回 E
]
```

16 模块 [module]

1 *X* 将程序视为若干个*模块*的组织。模块可以是一个文件、一个目录或者一整个库；模块也可以通过模块声明中定义。

2 模块可以拥有*子模块*。子模块可以通过目录结构自然生成，也可以在代码中指定。

3 模块可以以*代码库*的形式组织。代码库内部由目录形成的子模块结构是对外不透明的，外部只能访问库显式导出的实体。

4 每个文件构成一个*文件模块*，其名称为文件的无后缀名称。

5 每个目录构成一个*目录模块*，其名称为目录的名称。目录包含的所有文件或子目录构成的模块都是该目录模块的子模块。如果目录下包含一个名称为 `index.x` 的文件，则该目录模块导出的内容与该文件模块相同。否则，目录模块本身不导出任何内容。

6 代码库导出的内容即其根目录模块导出的内容。代码库的名称是可配置的。

16.1 模块声明 [module.decl]

ModuleDecl:
AccessQualifier`?` `module` *ModuleID*`?` `{` *ModuleContent* `}`
AccessQualifier`?` `module` *ModuleID*`?` `=` *ImportPath* `;`

ModuleID:
Identifier
ModuleID `.` *Identifier*

1 *模块声明*在小于文件的范围内定义模块。模块声明中的模块 ID 指定模块的名称。如果模块名称是以 `.` 连接的若干标识符，则等价于将该名称拆分为嵌套的模块声明。

[例：

```
module A.B { /* ... */ }

// 等价于
module A { module B { /* ... */} }
```

]

2 模块声明可以使用大括号包裹。大括号内部的全部项将作为该模块的内容。模块声明可以使用等号来定义*模块别名*。在这两种情况下，如果省略模块名称，则整个文件不能包含任何其他声明项。该显式定义或别名将替换掉整个文件的模块定义。

16.2 导入指令 [import]

ImportDirective:
    `public?` `import` *ImportPath* `;`
    `public?` `import` *ImportPath* `:` *ImportItems* `;`

ImportPath:
    ExternalImportPath
    InternalImportPath
    RelativeImportPath

ExternalImportPath:
    ImportPathPart
    ExternalImportPath `.` *ImportPathPart*

InternalImportPath:
    `root` `.` *ImportPathPart*
    InternalImportPath `.` *ImportPathPart*

RelativeImportPath:
    `this` `.` *ImportPathPart*
    RelativeImportPath `.` *ImportPathPart*

ImportPathPart:
    Identifier
    StringLiteral
    `super`

ImportItem:
    `*`
    `operator`
    `operator` *OperatorType?* *OperatorName*
    Identifier
    Identifier `as` *Identifier*

1 导入指令用于将其他模块的内容引入到当前模块中。

2 被导入的模块可以通过三种方式指定：外部路径、内部路径或相对路径。外部路径以模块名开始，用于导入外部代码库。内部路径以 `root` 开始，从当前代码库根目录开始寻找模块。相对路径以 `this` 开始，从当前目录开始寻找模块。

3 模块的路径各部分可以使用一个标识符或者字符串标识。上下文关键字 `root` 指代当前模块的根目录，但只能用于开头；上下文关键字 `super` 指代当前目录的父目录。其他关键字或上下文关键字都将作为普通标识符处理。模块目录名称可以与 `this`、`root` 或 `super` 相同，但此时必须使用字符串表示。

4 模块的目录名称可以包含连字符，并可以通过将连字符转换为下划线的对应标识符引用。如果对应位置分别为下划线和连字符的目录都存在，则该标识符将会引用下划线的目录。以字符串方式引用目录名不会进行此类转换。

5 如果一个路径能以不同方式引用到同一个模块，则这是一个编译错误。

[例：

```
import root.foo.bar; // 导入/foo/bar 模块
import this.baz; // 导入当前目录下的 baz 模块
import this.super.qux; // 导入父目录下的 qux 模块

import some_hypen; // 导入外部模块 some-hypen
import "super"; // 导入外部模块 super
```

]

6 `*` 导入指定模块的所有内容，但是不包括运算符定义。

7 导入指令可以带有一个 `public` 修饰符。此时该指令除了将被导入的实体引入当前文件外，还将这些实体重新作为公开实体导出。

16.3 访问控制 [access]

AccessQualifier: 以下之一

```
public internal private
```

1 声明具有*访问可见性*，用于指定该程序实体能被什么范围的其他实体所访问。访问可见性有以下四种类型：公开（`public`）、内部（`internal`）、私有（`private`）、局部，其级别依次降低。

2 公开级别的实体能被任何位置的程序实体访问。在导入外部模块时，只能导入公开级别的实体。

3 内部级别的实体能被同一代码库中的程序实体访问。

4 私有级别的实体只能被同一文件中的实体访问。

5 如果一个声明位于函数内部，则它具有局部可见性。不能为它指定访问修饰符。函数参数也具有局部可见性。只能在函数内部访问该声明。

6 声明的默认可见性为 `public`。如果一个成员没有显式指定可见性，且其所在实体的可见性比其默认可见性要更低，则使用其所在实体的可见性。可见性对实现没有意义，但固有实现可以包含一个可见性修饰符，用于指定其所有项的可见性。

7 特征的项具有的可见性与特征相同。不能为这些项单独指定可见性。在特征实现中，包含在特征中的项的可见性与该特征相同；未包含在特征中的项的可见性总是为 `private`。不能为这些项重新指定可见性。

[例：

```
class X {
    x: int; // 默认为 private
    public y: int; // 显式指定为 public
}

impl X {
    func f() {} // 默认为 public
```

```
    private func g() {} // 显式指定为 private
}

internal class Y {
    x: int; // 默认为 private
    public y: int; // 显式指定为 public
}

impl Y {
    func f() {} // 默认为 internal
    public func g() {} // 显式指定为 public
}
```

17 特性与修饰符 [attr]

Attribute:

`@` *Identifier*
`@` *Identifier* `(` *Arguments?* `)`
`@` *Identifier* `[` *ExprList?* `]`
`@` *Identifier* `{` *StructItems?* `}`
`@` *Identifier DictLiteral*

1 特性可以修饰特定的程序实体，以对其添加特定的描述或限制。

17.1 `noreturn` [attr.noreturn]

1 `noreturn` 修饰的函数将不会返回。函数的返回类型必须是 `never`。编译器可以对实际控制流能到达函数结尾的 `noreturn` 函数提出一个警告或编译错误。

17.2 `deprecated` [attr.deprecated]

1 `deprecated` 特性用于标记已经过时的程序实体。编译器可以对使用已经过时的程序实体提出一个警告或编译错误。

18 宏 [macro]

MacroInvocation:
`#` *MacroName* *TokenDelimited*`?`

MacroName:
Identifier

1 宏提供了一种在编译期生成代码的能力。宏可以接受参数，并展开成为特定的标记序列。宏的参数由一对对应的分隔符包围，且内部的分隔符也必须成对。

2 宏名称是一个标识符，并且可以与关键字相同。

18.1 宏定义 [macro.def]

MacroDefinition:
`macro` *MacroName* `{` *MacroRule*`*` `}`

MacroRule:
MacroPattern → *TokenDelimited*

MacroPattern:
`(` *MacroItem*`*` `)`
`[` *MacroItem*`*` `]`
`{` *MacroItem*`*` `}`

MacroItem:
MacroToken
MacroPattern
`#` *Identifier* `:` *MacroItemType*
`#` `(` *MacroItem*`+` `)` *MacroSep*`*` *MacroRepeat*

MacroItemType:
以下之一 `id expr block stmt decl pat type path qual attr tokenTree`

MacroToken:
Token 但不是 `(` `)` `[` `]` `{` `}`

MacroSep:
Token 但不是 `(` `)` `[` `]` `{` `}` `+` `*` `?`

MacroToken:
以下之一 `+` `*` `?`

MacroReplacer:
`{` *MacroReplacerItem*`*` `}`

MacroReplacerItem:
MacroToken
TokenDelimited
MacroInvocation
`#` `(` *MacroReplacerItem*`+` `)` *MacroSep*`*` *MacroRepeat*

1 宏定义通过称为*声明宏*的方式创建宏。宏定义在关键字后跟 `macro` 和宏名称，然后是一对大括号，其中包含宏规则。宏规则由宏模式和宏转换两部分组成。宏模式和宏转换是由一对定界符包括的标记序列。

2 宏模式定义了宏替换的模式。宏模式由一系列项组成。每个项可以是标记、定界符包含的宏模式、宏匹配项或者宏重复项。标记匹配完全相同的单个标记，但不能指定匹配定界符。定界符包含的宏模式匹配成对的定界符以及内部的模式。最外层的宏模式也按照相同方式匹配。宏匹配项匹配一个特定的程序项。宏重复项匹配括号内部的程序项，但是可以以指定的分隔符分隔，并可以指定为重复任意次、重复至少一次或者重复零次或一次。

3 宏匹配项指定的特定程序项的含义如表 14 所示。

表 14 — 整数字面量后缀

类型	匹配的程序项
`id`	单个标识符
`expr`	表达式
`block`	块
`stmt`	语句
`decl`	声明
`pat`	模式
`type`	类型
`path`	路径
`qual`	单个修饰符
`attr`	属性
`tokenTree`	成对定界符及其内部的标记，或单个非定界符标记

4 当宏展开时，首先将参数中的所有宏递归展开（在展开点）。然后对宏定义中每条规则的模式依次进行匹配。如果某一项的模式匹配成功，则按照对应的宏转换将输入的标记序列转换为结果标记序列。如果所有规则都无法匹配，则这是一个编译错误。

5 宏转换指定了宏展开的结果。其中的普通标记项及定界符将按原样输出。宏展开项如果其名称与模式中的匹配项相同，则展开为特定的展开项；否则按照一般的宏展开规则递归展开（在定义点）。宏转换中的重复项表示重复展开。其中被匹配了多次的匹配项也必须在转换中被使用了等量次数，否则这是一个编译错误。产生的标记序列（不包括宏展开最外层的定界符）被插入到宏展开点。

18.2 内建宏 [macro.builtin]

1 一些宏内建于语言之中，它们提供了无法通过用户实现的功能。

18.2.1 源代码位置 [macro.src-pos]

1 `#file` 扩展为标识当前文件的路径的常量字符串。`#line` 扩展为当前行号。`#column` 扩展为当前列号。如果这些宏由一个宏展开调用，则使用的是最外层宏展开的对应位置。

18.2.2 `nameof` [macro.nameof]

1 `#nameof(`id`)` 扩展为值为 id 的常量字符串。

19 `core` 库介绍 [core]

1 `core` 库是唯一与语言相互作用的库。编译器了解 `core` 库的所有组件的接口以及（需要的话）内部细节。编译器总会提供 `core` 库。

20 杂项 [core.misc]

20.1 类型 [core.type]

1 本节包含了若干类型，它们均为语言内建类型。

```
type Symbol = __intrinsic;
```

2 `Symbol` 是所有符号字面量的公共类型。它只能通过内建的符号字面量来获得值。

```
enum Optional<T> {
    Some(T),
    Nil,
}
```

3 `Optional` 是可空类型的核心库定义；`Optional<`T`>` 等价于 T`?`。其中 `Optional<`T`>.Some(`v`)` 对应 `some` v，`Optional<`T`>.Nil` 对应 `nil`。

20.2 序 [core.order]

```
enum Order {
    less,
    equal,
    greater
}
```

1 `Order` 定义了序关系。它是内建 `cmp` 运算符的返回值。自定义类型可以通过实现 `cmp` 运算符来支持序关系，参见 11.3.9。

2 `.less` 代表左操作数小于右操作数。`.equal` 代表左操作数等于右操作数。`.greater` 代表左操作数大于右操作数。

3 `cmp` 也可以返回 `Order?`。如果返回值为 `nil`，则代表两个操作数之间没有序关系。

20.3 析构器 [core.deinit]

```
auto trait Deinit {
    func deinit(&mut this) -> void;
}
```

1 `Deinit` 为类型提供析构器。具有析构器的值在被销毁时会调用其析构器。

2 对于没有显式实现 `Deinit` 的类型，编译器会自动生成依次析构其所有成员的析构器。

20.4 范围 [core.range]

```
class Range<T>;

class ClosedRange<T>;
```

1 Range 表示左闭右开区间。它是内建 .. 的结果类型。

2 ClosedRange 表示闭区间。它是内建 ..= 的结果类型。

20.4.1 构造器 [core.range.init]

```
func init<T>(start: T, end: T) -> Range<T>;

func init<T>(start: T, end: T) -> ClosedRange<T>;
```

1 Range 和 ClosedRange 可以使用两个参数 start 与 end 构造，分别表示开头与结尾。如果 start > end 则会抛出.InvalidBounds。

20.4.2 实现 [core.range.impl]

```
impl<T> Range<T> : Sequence<T> {
    type Iterator = RangeIterator<T>;
}

impl<T> ClosedRange<T> : Sequence<T> {
    type Iterator = ClosedRangeIterator<T>;
}
```

1 Range 和 ClosedRange 是序列，其迭代器类型是 RangeIterator。

20.4.3 迭代器 [core.range.iter]

```
class RangeIterator<T>;

class ClosedRangeIterator<T>;
```

1 RangeIterator 是 Range 的迭代器类型。

2 ClosedRangeIterator 是 ClosedRange 的迭代器类型。

20.5 错误处理 [core.error]

20.5.1 特征 [core.error.trait]

```
trait ErrorCode { }
```

1 ErrorCode 是一个特征，用于表示错误。它是一个空的 trait，用于标记实现了错误的类型。

```
impl never : ErrorCode { }
```

2 never 实现了 ErrorCode，表示不会产生错误。

```
impl void : ErrorCode = never;
```

3 void 被禁止实现 ErrorCode，这是为了避免不带参数的 throw 表达式产生理解歧义。

20.5.2 实现 [core.error.impl]

```
impl<T, E> T !! E { /* ... */ }

let isOK: bool {
    get {
        try this catch { false } else { true }
    }
}

let isError: bool {
    get { !this.isOK }
}
```

1 isOK 当结果是正确值的时候返回 true，否则返回 false。isError 与之相反。

```
func takeOK() -> T? { this? }
func takeError() -> E? { try this catch let e { e } else { nil } }
```

2 takeOK 取出正确值。如果结果是错误值，则返回 nil。takeError 取出错误值。如果结果是正确值，则返回 nil。

```
func map<U>(f: T -> U) throw(E) -> U {
    f(try this)
}

func map<U>(f: T -> U, or or: U) -> U {
    f(this? ?? or)
}
```

3 map 将正确值映射为其他值。错误值会被抛出。如果提供了 or 参数，则是错误值时会返回 or。

```
func inspect(f: T -> void) -> void {
    f(this?)
}
```

4 inspect 将使用正确值调用函数 f。如果结果是错误值，则不会调用 f。

```
func expect(failMessage: string) -> T {
    try this catch { panic(failMessage) }
}
```

5 expect 取出结果的值。如果结果是错误值，则调用 panic 并输出 failMessage。

```
func expectError(failMessage: string) -> E {
    try this catch let e { e } else { panic(failMessage) }
```

```
}
```

6 `expectError` 取出结果的错误值。如果结果是正确值，则调用 `panic` 并输出 `failMessage`。

```
func and<U>(other: U !! E) throw -> U {
    try this catch let e { throw e } else { try other }
}
```

7 `and` 返回第一个错误值。如果第一个结果是正确值，则返回第二个结果。

```
func andThen<U>(f: T -> U !! E) throw(E) -> U {
    try f(try this)
}
```

8 如果结果是错误值，则直接抛出该值。否则以正确值调用函数 `f`。[注：与 `map` 不同的是，`andThen` 的 `f` 本身也可能产生错误。]

```
func or(other: T !! E) -> T {
    this? ?? try other
}
```

9 `or` 返回第一个正确值。如果第一个结果是错误值，则返回第二个结果。

```
func orElse<F>(f: E -> T !! F) throw(F) -> T {
    try this catch let e { try f(e) }
}
```

10 如果结果是正确值，则直接返回该值。否则以错误值调用函数 `f`。

```
func unwrapOr(default: T) -> T {
    this? ?? default
}
```

11 `unwrapOr` 返回结果的正确值。如果结果是错误值，则返回 `default`。

```
impl<T, E> T !! E if T is default {
    func unwrapOrDefault() -> T {
        this? ?? self::default
    }
}
```

12 如果 T 是 `core::Default`，则 `unwrapOrDefault` 等同于以 `T::default` 调用的 `unwrapOr`。

```
impl<T> T !! never {
    func toOK() -> T {
        try this
    }
}
```

13 如果错误类型为 `never`，则意味着永远不会发生错误。此时 `toOK` 将结果类型直接转换为正确值。

```
impl<E> never !! E {
    func toError() → E {
        try this
    }
}
```

14 如果正确类型为 never，则意味着永远不会得到正确值。此时 toError 将结果类型直接转换为错误值。

```
impl<T, E> (T !! E) !! E {
    func flatten() throw(E) → T {
        this!!
    }
}
```

15 flatten 将嵌套的结果类型展平。

20.5.3 panic [core.panic]

```
func panic(message: string = "") → never;
```

1 panic 在程序遇到无法处理的错误时终止程序的执行。它会向标准错误流打印错误信息 message，并析构当前线程的所有对象，然后终止程序的执行。

2 如果 panic 在执行过程中再次调用了 panic，则会直接终止程序的执行而略过任何对象的析构。

21 序列库 [core.seq]

1 core.seq 库定义了序列特征，并提供了一些操作序列的函数。

21.1 特征 [core.seq.traits]

```
trait Sequence<T> {
    type Item = T;
    type Iterator : core::Iterator<T>;

    let size: uint;
    let iter: Iterator;

    let isEmpty ⇒ this.size == 0;
}
```

1 Sequence 表示序列。

2 Item 是序列的元素类型，其始终为 T。

3 Iterator 是序列的迭代器类型，其必须实现了 Iterator<T>。

4 size 是序列的大小。序列实现了 Dollar，返回 size。

5 iter 是序列的迭代器。

6 isEmpty 返回序列是否为空。它的默认实现将序列大小与 0 比较。

```
impl<T> Sequence<T> : Dollar {
    func dollar() { this.size }
}
```

7 序列实现了 Dollar，返回序列的大小。

```
trait Iterator<T> {
    type Item = T;

    func next(this: mut) → T?;
}
```

8 Iterator 表示迭代器。

9 next 返回迭代器的下一个元素，如果迭代器已经到达末尾，则返回 nil。

21.2 辅助函数 [core.seq.helper]

索引

语法产生式索引

库名称索引