词汇结构

Whitespace

whitespace 是仅包含具有 Pattern_White_Space Unicode 属性字符的任何非空字符串，即：

- U+0009 (horizontal tab, '\t')
- U+000A (line feed, '\n')
- U+000B (vertical tab)
- U+000C (form feed)
- U+000D (carriage return, '\r')
- U+0020 (space, ' ')
- U+0085 (next line)
- U+200E (left-to-right mark)
- U+200F (right-to-left mark)
- U+2028 (line separator)
- U+2029 (paragraph separator)

Rust 是一种“自由形式”语言，这意味着所有形式的 whitespace 仅用于分隔语法中的 token，并没有语义意义。因此 whitespace 可以是任意的形式，而不会使 Rust 程序产生不同的含义。

Comments

非文档注释

Rust 代码中的注释遵循行（//）和块（/* ... */）注释形式的一般 C++ 样式。支持嵌套块注释。

非文档注释被解释为 whitespace。

文档注释

以三个斜杠（///）开头的行 doc 注释和块 doc 注释（/** ... */）都是内部 doc 注释，被解释为 doc 属性的特殊语法。其等价于在注释正文周围书写#[doc="..."]，例如 /// Foo 变成 #[doc="Foo"]，/** Bar */ 变成 #[doc="Bar"]。

以 //! 开头的行注释和块注释 /*! ... */ 是适用于注释其所在内容的文档注释，而不是后面条目的文档注释。其等价于在注释正文周围书写#![doc="..."]。//! 注释通常用于对源文件模块进行注释。

注意：文档注释中不允许出现孤立的 CR（\r），即后面不跟 LF（\n）。

语法

```
LINE_COMMENT :
      // (~[/ !] | //) ~\n*
   | //

BLOCK_COMMENT :
      /* (~[* !] | ** | BlockCommentOrDoc) (BlockCommentOrDoc | ~*/)* */
   | /**/
   | /***/

INNER_LINE_DOC :
   //! ~[\n IsolatedCR]*

INNER_BLOCK_DOC :
   /*! ( BlockCommentOrDoc | ~[*/ IsolatedCR] )* */

OUTER_LINE_DOC :
   /// (~/ ~[\n IsolatedCR]*)?

OUTER_BLOCK_DOC :
   /** (~* | BlockCommentOrDoc ) (BlockCommentOrDoc | ~[*/ IsolatedCR])* */

BlockCommentOrDoc :
      BLOCK_COMMENT
   | OUTER_BLOCK_DOC
   | INNER_BLOCK_DOC
```

```
IsolatedCR :
   A \r not followed by a \n
```

Tokens

token 是由常规（非递归）语言定义的语法中的原始产物。 Rust 源输入可以分解为以下类型的 token：

- Keywords
- Identifiers
- Literals
- Lifetimes
- Punctuation
- Delimiters

Keywords

Rust 将 keyword 分为三类：

- strict keyword
- reserved keyword
- weak keyword

strict keyword

这些关键字只能在正确的上下文中使用。它们不能用做名称。strict keyword 如下：

```
KW_AS : as
KW_BREAK : break
KW_CONST : const
KW_CONTINUE : continue
KW_CRATE : crate
KW_ELSE : else
KW_ENUM : enum
KW_EXTERN : extern
KW_FALSE : false
KW_FN : fn
KW_FOR : for
KW_IF : if
KW_IMPL : impl
KW_IN : in
KW_LET : let
KW_LOOP : loop
KW_MATCH : match
KW_MOD : mod
KW_MOVE : move
KW_MUT : mut
KW_PUB : pub
KW_REF : ref
KW_RETURN : return
KW_SELFVALUE : self
KW_SELFTYPE : Self
KW_STATIC : static
KW_STRUCT : struct
KW_SUPER : super
KW_TRAIT : trait
KW_TRUE : true
KW_TYPE : type
KW_UNSAFE : unsafe
KW_USE : use
KW_WHERE : where
KW_WHILE : while
KW_ASYNC : async
KW_AWAIT : await
KW_DYN : dyn
```

reserved keyword

这些关键字尚未使用，但保留供将来使用。它们不能用做名称。reserved keyword 如下：

```
KW_ABSTRACT : abstract
```

```
KW_BECOME : become
KW_BOX : box
KW_DO : do
KW_FINAL : final
KW_MACRO : macro
KW_OVERRIDE : override
KW_PRIV : priv
KW_TYPEOF : typeof
KW_UNSIZED : unsized
KW_VIRTUAL : virtual
KW_YIELD : yield
KW_TRY : try
```

weak keyword

以下关键字仅在特定上下文中具有特殊含义，如果不在特定上下文中，可以作为名称使用，但是不建议这么做。例如，可以使用名称 union 声明变量或方法：

- macro_rules 用于创建自定义宏。
- union 用于声明一个联合体，并且在联合声明中使用时只是一个关键字。
- 'static 用于静态生命周期，不能用作通用生命周期参数或循环标签。

Identifiers

组成

标识符是以下形式的任何非空 Unicode 字符串：

- 第一个字符具有 XID_start 属性，其余字符具有 XID_continue 属性。
- 第一个字符是_，其余字符具有 XID_continue 属性，并且不是单字符字符串。

原始标识符类似于普通标识符，但以 r# 为前缀（注意，r# 前缀不作为实际标识符的一部分。）。与普通标识符不同，原始标识符可以是除了 crate, self, super, Self 外的任何 strict keyword 或 reserved keyword。

语法

```
IDENTIFIER_OR_KEYWORD :
      XID_start XID_continue*
   | _ XID_continue+

RAW_IDENTIFIER :
    r# IDENTIFIER_OR_KEYWORD Except crate, self, super, Self

NON_KEYWORD_IDENTIFIER :
    IDENTIFIER_OR_KEYWORD Except a strict or reserved keyword

IDENTIFIER :
    NON_KEYWORD_IDENTIFIER | RAW_IDENTIFIER
```

Literals

字面值由单个 token 构成而不是一系列 token。它立即且直接地表示它的评估值，而不是通过名称或其他评估规则来引用它。

字面值是常量表达式的一种形式，因此在编译时就会评估值。

转义字符

ASCII escapes

\x41	7-bit character code (exactly 2 digits, up to 0x7F)
\n	Newline
\r	Carriage return
\t	Tab
\\	Backslash
\0	Null

Byte escapes

\x7F	8-bit character code (exactly 2 digits)
\n	Newline
\r	Carriage return
\t	Tab
\\	Backslash
\0	Null

Unicode escapes

\u{7FFF}	24-bit Unicode character code (up to 6 digits)

Quote escapes

\’	Single quote
\″	Double quote

字符

组成

字符字面量是包含在两个单引号字符中的单个 Unicode 字符，但 U+0027(’) 本身除外，它必须由转义字符 U+005C(\) 转义。

例如：

```
’a’
’\’’
’\x6f’
’\uffffff’
```

语法

```
CHAR_LITERAL :
   ’ ( ~[’ \ \n \r \t] | QUOTE_ESCAPE | ASCII_ESCAPE | UNICODE_ESCAPE ) ’

QUOTE_ESCAPE :
   \’ | \″

ASCII_ESCAPE :
       \x OCT_DIGIT HEX_DIGIT
   | \n | \r | \t | \\ | \0

UNICODE_ESCAPE :
   \u{ ( HEX_DIGIT _* )1..6 }
```

注意：OCT_DIGIT 指单个八进制字符（0-7），HEX_DIGIT 指单个十六进制字符(0-9a-fa-F)。

字符串

组成

字符串字面量是包含在两个双引号字符中的多个 Unicode 字符，但 U+0022(″) 本身除外，它必须由转义字符 U+005C(\) 转义。

例如：

```
″a″
″abc″
″a\x35\udfef″
″a\0\r\n″
```

字符串字面量中允许换行，换行符是换行 U+000A(\n)或回车换行 U+000DU+000A(\r\n)，这两种字符序列通常都将转换为 U+000A(\n)。有一种特殊的情况，当转义字符(\)出现在换行符前面，那么在下一行开头前的所有换行符和 whitespace 都将被忽略。例如：

```
let a = ″foobar″;
let b = ″foo\
         bar″;
```

```
assert_eq!(a,b);
```

注意：字符串中不允许出现单独的回车符 U+000D(\r)。

语法

```
STRING_LITERAL :
   " (
      ~[" \ IsolatedCR]
      | QUOTE_ESCAPE
      | ASCII_ESCAPE
      | UNICODE_ESCAPE
      | STRING_CONTINUE
   )* "

STRING_CONTINUE :
   \ followed by \n
```

原始字符串

组成

原始字符串字面量不处理任何转义。它们以字符 U+0072 (r) 开头，后跟零个或多个 U+0023 (#)字符 和 一个 U+0022（双引号）字符。原始字符串主体可以包含任何 Unicode 字符序列，并且以另一个 U+0022（双引号）字符结尾，后跟与在开头 U+0022（双引号）前的相同数量的 U+0023 (#) 字符。#字符用于保证在字符串主体中存在双引号时，能够准确的判断字符串的结尾位置。

包含在原始字符串正文中的所有 Unicode 字符都表示它们自己，字符 U+0022（双引号）或 U+005C (\) 没有任何特殊含义。例如：

```
"foo"; r"foo";                     // foo
"\"foo\""; r#""foo""#;             // "foo"

"foo #\"# bar";
r##"foo #"# bar"##;                // foo #"# bar

"\x52"; "R"; r"R";                 // R
"\\x52"; r"\x52";                  // \x52
```

语法

```
RAW_STRING_LITERAL :
   r RAW_STRING_CONTENT

RAW_STRING_CONTENT :
      " ( ~ IsolatedCR )* (non-greedy) "
   | # RAW_STRING_CONTENT #
```

字节

组成

字节字面量由字符 U+0062 (b) 和 U+0027（单引号）开始，后面是单个 ASCII 字符（在 U+0000 到 U+007F 范围内）或单个转义字符，以单引号字符 U +0027 结尾。如果字符 U+0027 出现在文字中，则必须用前面的 U+005C (\) 字符对其进行转义。字节字面量等效于无符号 8 位整数 u8。

例如：

```
b'a'
b'\''
b'\xff'
```

语法

```
BYTE_LITERAL :
   b' ( ASCII_FOR_CHAR | BYTE_ESCAPE ) '

ASCII_FOR_CHAR :
   any ASCII (i.e. 0x00 to 0x7F), except ', \, \n, \r or \t

BYTE_ESCAPE :
      \x HEX_DIGIT HEX_DIGIT
```

```
    | \n | \r | \t | \\ | \0
```

注意：HEX_DIGIT 指单个十六进制字符(0-9a-fa-F)。

字节字符串

组成

字节字符串字面量由字符 U+0062 (b) 和 U+0022（双引号）开始，后面跟 ASCII 字符和转义符序列，最后是双引号字符 U+0022。 如果双引号字符 U+0022 出现在文字中，则必须用 U+005C(\)转义字符对其进行转义。 长度为 n 的字节字符串字面量的类型是 &'static[u8; n]。

例如：

```
b"a"
b"abc"
b"a\x35"
b"a\0\r\n"
```

和字符串字面量一样，换行符是换行 U+000A(\n)或回车换行 U+000DU+000A(\r\n)，这两种字符序列通常都将转换为 U+000A(\n)。当转义字符(\)出现在换行符前面，那么在下一行开头前的所有换行符和 whitespace 都将被忽略。

语法

```
BYTE_STRING_LITERAL :
   b" ( ASCII_FOR_STRING | BYTE_ESCAPE | STRING_CONTINUE )* "

ASCII_FOR_STRING :
   any ASCII (i.e 0x00 to 0x7F), except ", \ and IsolatedCR
```

原始字节字符串

组成

原始字节字符串字面量不处理任何转义。 它们以字符 U+0062 (b) 开头，然后是 U+0072 (r)，然后是零个或多个字符 U+0023 (#) 和一个 U+0022(双引号)字符。 原始字节字符串主体可以包含任何 ASCII 字符序列，并且仅以另一个 U+0022(双引号)字符结尾，后跟开头 U+0022（双引号）之前的相同数量的 U+0023 (#) 字符。 原始字节字符串文字不能包含任何非 ASCII 字节。#字符用于保证在字节字符串主体中存在双引号时，能够准确的判断字节字符串的结尾位置。

包含在原始字节字符串正文中的所有 Unicode 字符都表示它们自己，字符 U+0022（双引号）或 U+005C (\) 没有任何特殊含义。例如：

```
b"foo"; br"foo";                     // foo
b"\"foo\""; br#""foo""#;             // "foo"

b"foo #\"# bar";
br##"foo #"# bar"##;                 // foo #"# bar

b"\x52"; b"R"; br"R";                // R
b"\\x52"; br"\x52";                  // \x52
```

语法

```
RAW_BYTE_STRING_LITERAL :
   br RAW_BYTE_STRING_CONTENT

RAW_BYTE_STRING_CONTENT :
      " ASCII* (non-greedy) "
   | # RAW_BYTE_STRING_CONTENT #

ASCII :
   any ASCII (i.e. 0x00 to 0x7F)
```

布尔

组成

布尔字面量仅有两个确定值 true 和 false。

语法

```
BOOLEAN_LITERAL :
      true
```

```
    | false
```

数值

整数

组成

整数字面量具有以下四种形式：

- 十进制：以十进制数字开头，以十进制数字和下划线的混合形式呈现。
- 十六进制：以字符序列 U+0030 U+0078 (0x)开头，以十六进制数字和下划线的任何混合形式呈现。
- 八进制：以字符序列 U+0030 U+006F (0o)开头，以八进制数字和下划线的任何混合形式呈现。
- 二进制：以字符序列 U+0030 U+0062 (0b)开头，以二进制数字和下划线的任何混合形式呈现。

一个整数字面量可以跟随（中间没有任何空格）一个后缀，它用于强制设置数值字面量的类型。整数后缀必须是以下整数类型之一的名称：u8、i8、u16、i16、u32、i32、u64、i64、u128、i128、usize 或 isize。

无后缀的整数字面量的类型由以下类型推断规则确定：

- 如果可以从程序上下文中唯一确定整数类型，则无后缀的整数字面量具有该类型。
- 如果程序上下文对类型的约束不足，则默认为有符号的 32 位整数 i32。
- 如果程序上下文过度约束类型，则认为是静态类型错误。

例子：

```
123;                            // type i32
123i32;                         // type i32
123u32;                         // type u32
123_u32;                        // type u32
let a: u64 = 123;               // type u64

0xff;                           // type i32
0xff_u8;                        // type u8

0o70;                           // type i32
0o70_i16;                       // type i16

0b1111_1111_1001_0000;          // type i32
0b1111_1111_1001_0000i64;       // type i64
0b________1;                    // type i32

0usize;                         // type usize
```

语法

```
INTEGER_LITERAL :
   ( DEC_LITERAL | BIN_LITERAL | OCT_LITERAL | HEX_LITERAL ) INTEGER_SUFFIX?

DEC_LITERAL :
   DEC_DIGIT (DEC_DIGIT|_)*

BIN_LITERAL :
   0b (BIN_DIGIT|_)* BIN_DIGIT (BIN_DIGIT|_)*

OCT_LITERAL :
   0o (OCT_DIGIT|_)* OCT_DIGIT (OCT_DIGIT|_)*

HEX_LITERAL :
   0x (HEX_DIGIT|_)* HEX_DIGIT (HEX_DIGIT|_)*

BIN_DIGIT : [0-1]

OCT_DIGIT : [0-7]

DEC_DIGIT : [0-9]

HEX_DIGIT : [0-9 a-f A-F]

INTEGER_SUFFIX :
      u8 | u16 | u32 | u64 | u128 | usize
   | i8 | i16 | i32 | i64 | i128 | isize
```

浮点数

组成

浮点数字面量具有以下两种形式：

- 十进制数值字面量后跟句点字符 U+002E (.)，之后可选地跟随另一个十进制数值字面量，并带有可选的指数。
- 单个十进制数值字面量后跟一个指数。

像整数字面量一样，浮点数字面量后面可以跟一个后缀，只要后缀部分前面不以 U+002E (.) 结尾。后缀强制设置浮点数字面量的类型。有两个有效的浮点后缀 f32 和 f64（32 位和 64 位浮点类型）。

无后缀的浮点数字面量的类型由以下类型推断规则确定：

- 如果可以从程序上下文中唯一确定浮点数类型，则无后缀的浮点数字面量具有该类型。
- 如果程序上下文对类型的约束不足，则默认为有符号的 64 位浮点数 f64。
- 如果程序上下文过度约束类型，则认为是静态类型错误。

例子：

```
123.0f64;         // type f64
0.1f64;           // type f64
0.1f32;           // type f32
12E+99_f64;       // type f64
5f32;             // type f32
let x: f64 = 2.; // type f64
```

如果最后一个示例为 2.f64，那么表示 2 上名为 f64 的方法。

语法

```
FLOAT_LITERAL :
      DEC_LITERAL . (not immediately followed by ., _ or an identifier)
   | DEC_LITERAL FLOAT_EXPONENT
   | DEC_LITERAL . DEC_LITERAL FLOAT_EXPONENT?
   | DEC_LITERAL (. DEC_LITERAL)? FLOAT_EXPONENT? FLOAT_SUFFIX

FLOAT_EXPONENT :
   (e|E) (+|-)? (DEC_DIGIT|_)* DEC_DIGIT (DEC_DIGIT|_)*

FLOAT_SUFFIX :
   f32 | f64
```

元组索引

组成

元组索引字面量用于引用元组、元组结构和元组 variant 的字段。

元组索引直接与字面量标记进行比较。元组索引从 0 开始，每个连续的索引将值增加十进制值 1。 因此，只有十进制值会匹配，并且该值不能有任何额外的 0 前缀字符。

例如：

```
let example = ("dog", "cat", "horse");
let dog = example.0;
let cat = example.1;
// The following examples are invalid.
~~let cat = example.01;~~ // ERROR no field named `01`
~~let horse = example.0b10;~~  // ERROR no field named `0b10`
```

语法

```
TUPLE_INDEX:
   INTEGER_LITERAL
```

LifetimesOrLabel

说明

"生命周期参数"和"循环标签"使用 LIFETIME_OR_LABEL token。 语法分析器将接受任何 LIFETIME_TOKEN，例如，可以在宏中使用。

语法

```
LIFETIME_TOKEN :
      ' IDENTIFIER_OR_KEYWORD
   | '_

LIFETIME_OR_LABEL :
      ' NON_KEYWORD_IDENTIFIER
```

Punctuation

表达符号作用不一，具体看下面说明：

符号	名称	用途
+	Plus	Addition, Trait Bounds, Macro Kleene Matcher
-	Minus	Subtraction, Negation
*	Star	Multiplication, Dereference, Raw Pointers, Macro Kleene Matcher, Use wildcards
/	Slash	Division
%	Percent	Remainder
^	Caret	Bitwise and Logical XOR
!	Not	Bitwise and Logical NOT, Macro Calls, Inner Attributes, Never Type, Negative impls
&	And	Bitwise and Logical AND, Borrow, References, Reference patterns
|	Or	Bitwise and Logical OR, Closures, Patterns in match, if let, and while let
&&	AndAnd	Lazy AND, Borrow, References, Reference patterns
||	OrOr	Lazy OR, Closures
<<	Shl	Shift Left, Nested Generics
>>	Shr	Shift Right, Nested Generics
+=	PlusEq	Addition assignment
-=	MinusEq	Subtraction assignment
*=	StarEq	Multiplication assignment
/=	SlashEq	Division assignment
%=	PercentEq	Remainder assignment
^=	CaretEq	Bitwise XOR assignment
&=	AndEq	Bitwise And assignment
|=	OrEq	Bitwise Or assignment
<<=	ShlEq	Shift Left assignment
>>=	ShrEq	Shift Right assignment, Nested Generics
=	Eq	Assignment, Attributes, Various type definitions
==	EqEq	Equal
!=	Ne	Not Equal
>	Gt	Greater than, Generics, Paths
<	Lt	Less than, Generics, Paths
>=	Ge	Greater than or equal to, Generics
<=	Le	Less than or equal to
@	At	Subpattern binding
_	Underscore	Wildcard patterns, Inferred types, Unnamed items in constants, extern crates, and use declarations
.	Dot	Field access, Tuple index
..	DotDot	Range, Struct expressions, Patterns
...	DotDotDot	Variadic functions, Range patterns
..=	DotDotEq	Inclusive Range, Range patterns
,	Comma	Various separators
;	Semi	Terminator for various items and statements, Array types
:	Colon	Various separators

符号	名称	用途
::	PathSep	Path separator
->	RArrow	Function return type, Closure return type, Function pointer type
=>	FatArrow	Match arms, Macros
#	Pound	Attributes
$	Dollar	Macros
?	Question	Question mark operator, Questionably sized, Macro Kleene Matcher

Delimiters

括号标点符号用于语法的各个部分。 开括号必须始终与闭括号配对。 括号和其中的 token 在宏中称为“token 树”。 三种类型的括号是：

括号	类型
{ }	大括号
[ ]	方括号
( )	圆括号

重要概念

所有权

Rust 的值可以在堆上，也可以在栈上。栈上的值可以通过指向该值的"引用"或者该值"本身"进行访问，堆上的值只能通过指向该值的"智能指针"进行访问。注意：

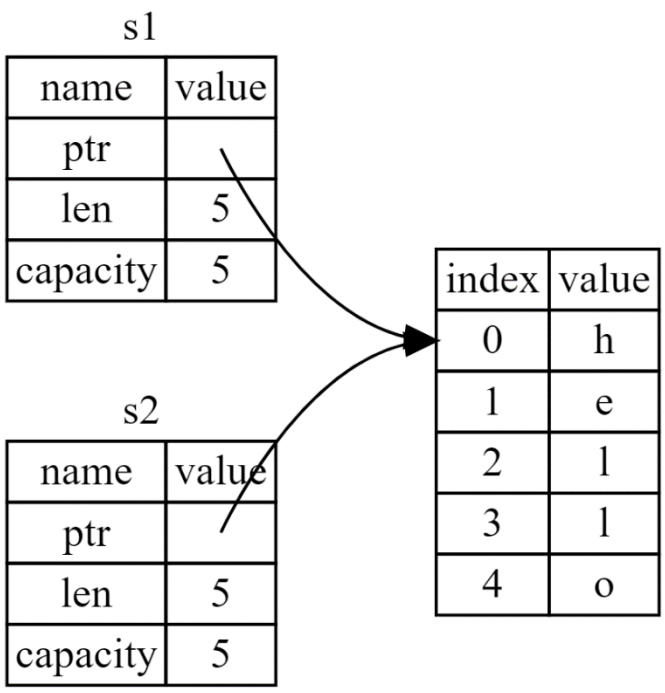
- 值的引用类似于 C++指针值，但是引用只可以引用栈上的值。
- 智能指针表现的像引用一样，智能指针的内部可以存在指向堆上的值，例如 Box、Rc、Week 等存在指向堆上值。
- 引用和智能指针本身都是栈上的值。

Rust 的所有权针对是栈上的值，具有以下规则：

1. 这些值都有一个被称为其所有者（owner）的变量。
2. 一个值在任意时刻有且只有一个所有者。
3. 当所有者（变量）离开作用域，这个值将被丢弃，如果该值的变量类型实现了 Drop trait，那么会调用其 drop 函数进行堆内存释放操作。

所有权转移

Rust 的一个变量 A 的值复制给另外一个变量 B 时，会进行一次浅拷贝，将栈上的值完全复制一份新值，新值的所有者为变量 B。如果此时变量 A 和变量 B 的类型是一个实现了 Drop trait 的类型，如果安装其他语言的逻辑，在变量 A 和变量 B 分别离开作用域时，将会导致两次的 drop，而导致堆内存的两次释放。例如：



因此 Rust 在进行变量复制时采用以下规则：

1. 默认情况下 B=A 将使用变量移动的方式，即 B=A 将导致变量 A 失效，在该语句之后无法再使用变量 A。
2. 如果变量类型添加 Copy trait 注解，那么 B=A 将不会导致变量 A 失效，Copy 注解指示值完全在栈上。以下基础类型是默认 Copy 类型：
    - 整数类型，比如 u32。
    - 布尔类型，true 和 false。
    - 浮点数类型，比如 f64。
    - 字符类型，char。
    - 元组，当且仅当其包含的类型也都是 Copy 的时候。比如，(i32, i32) 是 Copy 的，但 (i32, String) 就不是。

注意：

1. 只有在变量离开作用域的时候才会进行 drop，所有权转移的时候不会进行任何操作。
2. 引用离开作用域时不会导致其引用的值被 drop。因为引用其实就是 Copy 类型值。
3. 对于函数的入参以及返回值，遵循变量复制的规则。
4. 如果变量的类型实现了 Drop trait，那么其就无法添加 Copy trait 注解，他们是互斥的。

引用

默认情况下，如果一个变量既非引用，也非 Copy 类型，那么其作为参数传递给函数后将无法使用。通常我们使用引用来解决这类问题，并且引用可以避免栈上的浅拷贝。

"&"符号就是引用，它们允许你使用值但不获取其所有权。函数签名使用&来表明参数的类型是一个引用。例如：

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Rust 将获取引用作为函数参数称为借用（borrowing）。

注意：引用离开作用域时不会导致其引用的值被 drop。

生命周期参数

生命周期参数的主要目标是避免悬垂引用，避免引用的数据已经提前离开作用域，这会导致程序引用了非预期引用的数据。例如：

```
{
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
// 尝试使用离开作用域的值的引用
```

Rust 编译器有一个"借用检查器"，它比较作用域来确保所有的借用都是有效的。借用检查器会检查引用的生命周期是否比被引用的数据短。借用检查器需要配合"生命周期注解语法"来检查借用的有效性。

生命周期注解语法：生命周期参数名称必须以撇号（'）开头，其名称通常全是小写，类似于泛型其名称非常短。'a 是大多数人默认使用的名称。生命周期参数注解位于引用的 & 之后，并有一个空格来将引用类型与生命周期注解分隔开。例如：

```
&i32        // 引用
&'a i32     // 带有显式生命周期的引用
&'a mut i32 // 带有显式生命周期的可变引用
```

类型

Rust 程序中的每个变量、项和值都有一个类型。值的类型定义了如何解释保存了它的内存，以及可以对值执行的操作。内置类型以用户定义类型无法模拟的重要方式紧密集成到 rust 语言中，因为用户定义类型的功能有限。

类型一览

原始类型

Boolean type

boolean 类型名称为 bool，可以为以下两个值之一：true 和 false。这种类型的值可以使用字面量表达式创建，使用关键字 true 和 false 对应于相同名称的值。

boolean 类型的对象的大小和对齐方式均为 1。值 false 的位模式为 0x00，值 true 的位模式为 0x01。存在 boolean 类型对象的位操作具有未定义的行为。

与所有原始类型一样，boolean 类型实现了 Clone、Copy、Sized、Send 和 Sync 等 trait。

对于 boolean 类型操作数的运算结果如下：

- 逻辑非运算

b	!b
true	false
false	true

- 逻辑或运算

a	b	a | b
true	true	true
true	false	true
false	true	true
false	false	false

- 逻辑与运算

a	b	a & b
true	true	true
true	false	false
false	true	false
false	false	false

- 逻辑异或运算

a	b	a ^ b
true	true	false
true	false	true
false	true	true
false	false	false

- 比较运算

a	b	a == b
true	true	true
true	false	false
false	true	false
false	false	true

a	b	a > b
true	true	false
true	false	true
false	true	false
false	false	false

Numeric types

无符号整数类型包括：

Type	Minimum	Maximum
u8	0	2^8-1
u16	0	$2^{16}-1$
u32	0	$2^{32}-1$
u64	0	$2^{64}-1$
u128	0	$2^{128}-1$

有符号二进制补码整数类型包括：

Type	Minimum	Maximum
i8	$-(2^7)$	2^7-1
i16	$-(2^{15})$	$2^{15}-1$
i32	$-(2^{31})$	$2^{31}-1$
i64	$-(2^{63})$	$2^{63}-1$
i128	$-(2^{127})$	$2^{127}-1$

浮点数类型包括：f32 和 f64。

usize 类型是一个无符号整数类型，其位数与平台的指针类型相同。它可以代表进程中的每个内存地址。
isize 类型是一个有符号整数类型，其位数与平台的指针类型相同。它可以代表内存偏移，因此对象和数组大小上限是最大 isize 值。
usize 和 isize 至少为 16 位宽。

与所有原始类型一样，所有数值类型实现了 Clone、Copy、Sized、Send 和 Sync 等 trait。

Textual types

字符类型 char 和字符串类型 str 用于保存文本数据。

char 类型的值是 Unicode 标量值，表示为 0x0000 到 0xD7FF 或 0xE000 到 0x10FFFF 范围内的 32 位无符号字符。创建超出此范围的字符是未定义行为。

str 类型的值的表示方式与 [u8] 相同，它是 8 位无符号字节的切片。然而，Rust 标准库对 str 做了额外的假设：处理 str 的方法假设并确保其中的数据是有效的 UTF-8。使用非 UTF-8 缓冲区调用 str 方法可能会导致未定义行为。由于 str 是动态大小的类型，因此它只能通过指针类型进行实例化(编译器在编译的时候需要确切知道类型的大小)，例如 &str。

字符类型实现了 Clone、Copy、Sized、Send 和 Sync 等 trait。
字符串类型实现了 Send 和 Sync 等 trait。

Never type

never 类型!是一种没有值的类型，表示永远不会完成的计算结果。

类型!可以强制转换为任何其他类型。例如：

```
let x: ! = panic!();
// Can be coerced into any type.
let y: u32 = x;
```

注意：目前 never 类型只能出现在函数的返回值中。

序列类型

Tuple types

元组类型是一个结构化类型，用于异构其他类型的列表。元组类型具有如下格式：

```
TupleType :
      ( )
   | ( ( Type , )+ Type? )
```

元组类型的语法是带括号的、以逗号分隔的类型列表。

元组类型的字段数等于类型列表的长度。这个字段数决定了元组的数量。具有 n 个字段的元组称为 n 元元组。 元组的字段使用与它们在类型列表中的位置相匹配的递增数字命名。 第一个字段是 0，第二个字段是 1，依此类推。每个字段的类型是元组类型列表中相同位置的类型。没有字段的元组()被称为单元类型。

元组字段可以通过元组索引表达式或模式匹配来访问。

如果结构化类型的内部类型是等价的，那么结构化类型也是等价的。

需要注意：一元元组需要在其元素类型后紧跟逗号，用来区别带括号的类型。例如：

```
(i32) 是 i32 类型
(i32,) 是一元元组类型
```

例子：

```
()
(f64, f64)
(String, i32)
(i32, String)
(i32, f64, Vec<String>, Option<bool>)
```

Array types

数组是由 N 个 T 类型元素组成的固定大小序列。数组类型具有如下格式：

```
ArrayType :
   [ Type ; Expression ]
```

Expression 是一个计算结果为 usize 的常量表达式。

数组的所有元素总是被初始化，对数组的访问总是在安全的方法和运算符中进行边界检查。

例子：

```
// A stack-allocated array
let array: [i32; 3] = [1, 2, 3];

// A heap-allocated array, coerced to a slice
let boxed_array: Box<[i32]> = Box::new([1, 2, 3]);
```

Slice types

slice 是一种动态大小类型，其表示为 T 类型元素序列的视图。slice 类型具有如下格式：

```
SliceType :
   [ Type ]
```

由于 slice 是动态大小类型，因此它只能通过指针类型进行实例化，例如：

- &[T]：“共享切片”，它不拥有它指向的数据，而是借用它。
- &mut [T]：“可变切片”，它可以修改指向的数据。
- Box<[T]>：box slice。

切片的所有元素总是被初始化，对切片的访问总是在安全的方法和运算符中进行边界检查。

例子：

```
// A heap-allocated array, coerced to a slice
let boxed_array: Box<[i32]> = Box::new([1, 2, 3]);

// A (shared) slice into an array
let slice: &[i32] = &boxed_array[..];
```

定义类型

Struct types

struct 类型是其他类型的异构，这些类型称为 struct 类型的字段。其格式见 Struct 声明。

结构体的字段可以由可见性修饰符限定，以允许外部模块访问结构体中的数据。

元组结构类型就像 struct 类型一样，只是字段是匿名的。

默认情况下，struct 的内存布局是未定义的，以允许编译器优化，例如字段重新排序，但可以使用 repr 属性宏进行修复(见类型布局)。 在任何一种情况下，字段都可以在相应的结构表达式中以任何顺序给出，生成的结构值将始终具有相同的内存布局。

Enumerated types

枚举类型是异构不相交联合类型。其格式见 Enum 声明。

Enum 声明了类型和变体的数量，每个变体都被独立命名并具有结构体、元组结构体或类单元结构体的语法。

任何枚举值消耗与其对应枚举类型的最大变体一样多的内存，以及存储判别式所需的大小。

枚举类型不能在结构上表示为类型，必须通过对枚举项的命名引用来表示。

Union types

union 类型是异构的类 C 联合。其格式见 Union 声明。

union 没有“激活字段”的概念。 相反，每个 union 的访问都会将 union 的部分内容转换为访问字段的类型，由于转换会导致意外或未定义的行为，因此从 union 读取或写入未实现 Copy 或具有 ManuallyDrop 类型的字段需要使用 unsafe 关键字。

默认情况 union 类型的内存布局是未定义的，但可以使用 repr 属性宏进行修复(见类型布局)。

函数类型

Function item types

function item type 是函数项类型，其是函数的类型标识，明确的标识函数的以下特征：名称、参数、生命周期参数、泛型参数，这区别于仅标识函数参数的函数指针类型。由于函数项类型是函数类型，所以调用该类型的值对应的函数时不需要再转换为函数指针类型，而可以直接调用。没有直接引用 function item type 的语法，但编译器会在错误消息中将 function item type 显示为类似 fn(u32) -> i32 {foo::<i32>} 。

所有的 function item type 都实现了 Fn, FnMut, FnOnce, Copy, Clone, Send, Sync 等 trait。

function item type 是可以是以下三种类型值的引用：function item、元组结构体的构造函数、元组变体的构造函数：

- function item 引用类型：

```
fn foo<T>() { }
fn main() {
    let x = &mut foo::<i32>;
}
```

- 元组结构体的构造函数引用类型：

```
struct Foo<T>(T);
fn main() {
    let x = &mut Foo::<i32>; // x 是元组结构体的构造函数引用
    println!("{}", x(1).0); // x(1)将会产生定义的元组结构体的实例
}
// 打印出 1。
```

- 元组变体的构造函数引用类型：

```
enum  Foo<T>{
    A(T),
    B
}

fn main() {
    let x = &mut Foo::A::<i32>; // x 是元组变体的构造函数引用
    match x(1) { // x(1)将会产生定义的 enum 变体的实例
        Foo::A(i) => { println!("{}", i); }
        Foo::B => {}
    }
}
// 打印出 1。
```

因为 function item type 明确标识了函数，所以不同名称、参数、生命周期参数、泛型参数的函数都是不同的：

```
fn foo<T>() { }
fn main() {
    let x = &mut foo::<i32>;
    *x = foo::<u32>; //~ ERROR mismatched types
}
```

由于存在 function item type 到具有相同签名的函数指针类型的强制类型转换，因此允许将不同的 function item type 的值转换为具有相同签名的函数指针类型的值，例如：

```
// `foo_ptr_1` has function pointer type `fn()` here
let foo_ptr_1: fn() = foo::<i32>;

// ... and so does `foo_ptr_2` - this type-checks.
let foo_ptr_2 = if want_i32 {
    foo::<i32>
} else {
    foo::<u32>
};
```

Closure types

闭包表达式会产生一个闭包值，该值具有无法写出的唯一匿名类型。

闭包类型等效于一个包含捕获变量的结构体。例如：

```
fn f<F : FnOnce() -> String> (g: F) {
    println!("{}", g());
}
```

```
let mut s = String::from("foo");
let t = String::from("bar");

f(|| {
    s += &t;
    s
});
// Prints "foobar".
```

生成一个闭包类型大致如下：

```
struct Closure<'a> {
    s : String,
    t : &'a String,
}

impl<'a> FnOnce<()> for Closure<'a> {
    type Output = String;
    fn call_once(self) -> String {
        self.s += &*self.t;
        self.s
    }
}
```

因此 f 的调用就类似：

```
f(Closure{s: s, t: &t});
```

闭包类型根据其捕获环境的不同，对应实现了不同的 trait：

- Fn：不可变借用。Fn 用于从其环境获取不可变的借用值。
- FnMut：可变借用。FnMut 用于获取可变的借用值，因此其可以改变其环境。
- FnOnce：获取所有权。FnOnce 将消费从周围作用域捕获的变量，获取所有权并在定义闭包时将其移动进闭包。

所有闭包都实现了 FnOnce。没有移动被捕获变量的所有权到闭包内的闭包也实现了 FnMut。不需要对被捕获的变量进行可变访问的闭包则也实现了 Fn。对应的 trait object 类型，可以存储对应的闭包：

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    let y = 2;
    Box::new(move |x| x + y)
}
```

对于捕获闭包，则只能通过 Fn/FnMut/FnOnce 这三个 trait 来引用。而非捕获闭包是不从其环境中捕获任何内容的闭包，它们可以被强制为具有匹配签名的函数指针（例如 fn()）：

```
let add = |x, y| x + y;

let mut x = add(5,7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5,7);
```

所有闭包类型都实现了 Sized。 此外，如果它存储的捕获类型允许，闭包类型实现以下 trait：Clone、Copy、Sync、Send。Send 和 Sync 的规则与普通 struct 类型的规则相匹配，而 Clone 和 Copy 的行为就像派生的一样。 对于 Clone，未指定捕获变量的克隆顺序。由于捕获通常是通过引用进行的，因此出现以下一般规则：

- 如果所有捕获的变量都是 Sync 的，则闭包是 Sync 的。
- 如果非唯一不可变引用捕获的所有变量都是 Sync 的，并且唯一不可变或可变引用 copy 或 move 捕获的所有值都是 send，则闭包是 Send。
- 如果闭包不通过唯一不可变或可变引用捕获任何值，并且通过 copy 或 move 捕获的所有值分别是 Clone 或 Copy，则闭包是 Clone 或 Copy。

指针类型

Pointer types

Rust 中的所有指针都可以被移动或复制，存储到数据结构中，并从函数中返回。

引用指针

引用指针的语法格式如下：

```
& Lifetime? mut? TypeNoBounds
```

引用指针有两种形式：&不可变借用，&mut 可变借用。

- 不可变借用：不可变借用指向了其他人拥有的内存。共享引用类型写为 &type，或者为具有生命周期参数的&'a type。不可变借用是共享

引用，并且可以防止直接更改值。复制一个不可变引用是一个“浅拷贝”的操作：它只涉及复制指针本身，即指针是 Copy。 释放引用对其指向的值没有影响，但引用一个临时值将使其仅在引用本身的范围内保持活动状态。

- 可变借用：可变借用指向了其他人拥有的内存。可变引用类型写成 &mut 类型或具有生命周期参数的 &'a mut type。可变引用是访问它指向的值的唯一方式，所以是不可以 Copy 的。

默认情况下，一个值的可变借用只能存在一个，但是可以通过 rust 的内部可变性的模式来实现类似一个值存在多个可变借用的效果。std::cell::UnsafeCell<T> 类型可以实现内部可变性，当 UnsafeCell<T> 类型值是不可变时，对其包含的 T 进行改变或获取其可变引用仍然是安全的。与所有其他类型一样，不能具有多个 &mut UnsafeCell<T>。例子：

```
use std::cell::UnsafeCell;

let x: UnsafeCell<i32> = 42.into();
// Get multiple / concurrent / shared references to the same `x`.
let (p1, p2): (&UnsafeCell<i32>, &UnsafeCell<i32>) = (&x, &x);

unsafe {
    // SAFETY: within this scope there are no other references to `x`'s contents,
    // so ours is effectively unique.
    let p1_exclusive: &mut i32 = &mut *p1.get(); // -- borrow --+
    *p1_exclusive += 27; //                                    |
} // <---------- cannot go beyond this point -------------------+

unsafe {
    // SAFETY: within this scope nobody expects to have exclusive access to `x`'s contents,
    // so we can have multiple shared accesses concurrently.
    let p2_shared: &i32 = &*p2.get();
    assert_eq!(*p2_shared, 42 + 27);
    let p1_shared: &i32 = &*p1.get();
    assert_eq!(*p1_shared, *p2_shared);
}
```

可以通过使用 UnsafeCell<T> 作为字段来创建其他具有内部可变性的类型。标准库提供了多种类型来提供安全的内部可变性 API。例如，std::cell::RefCell<T> 使用运行时借用检查来确保围绕多个引用的通常规则。std::sync::atomic 模块包含包装仅通过原子操作访问的值的类型，允许跨线程共享和改变该值。

裸指针

裸指针的语法格式如下：

```
* ( mut | const ) TypeNoBounds
```

裸指针是没有安全性或有效性保证的指针。裸指针写为 *const T 或 *mut T。例如 *const i32 表示指向 32 位整数的裸指针。可以使用 core::ptr::addr_of 直接创建*const 裸指针，使用 core::ptr::addr_of_mut 创建 *mut 裸指针。

复制或删除裸指针不会影响任何其他值的生命周期。Dereferencing 裸指针是一种不安全的操作，但是可用于通过重新借用裸指针（&* 或 &mut *）将裸指针转换为引用。

在比较裸指针时，它们是按地址进行比较，而不是按指向的内容进行比较。 当将裸指针与动态大小的类型进行比较时，它们也会比较它们的附加数据。

Function pointer types

函数指针类型语法如下：

```
BareFunctionType :
   ForLifetimes? FunctionTypeQualifiers fn
      ( FunctionParametersMaybeNamedVariadic? ) BareFunctionReturnType?

FunctionTypeQualifiers:
   unsafe? (extern Abi?)?

BareFunctionReturnType:
   -> TypeNoBounds

FunctionParametersMaybeNamedVariadic :
   MaybeNamedFunctionParameters | MaybeNamedFunctionParametersVariadic

MaybeNamedFunctionParameters :
   MaybeNamedParam ( , MaybeNamedParam )* ,?
```

```
MaybeNamedParam :
   OuterAttribute* ( ( IDENTIFIER | _ ) : )? Type

MaybeNamedFunctionParametersVariadic :
   ( MaybeNamedParam , )* MaybeNamedParam , OuterAttribute* ...
```

unsafe 限定符表示该类型的值是一个不安全的函数，而 extern 限定符表示它是一个 extern 函数。只能使用具有“C”或“cdecl”调用约定的外部函数类型才能指定可变参数。

fn 关键字的函数指针类型指示了在编译时不一定知道其身份的函数。函数指针类型的值可以通过 function item 和非捕获闭包的强制类型转换进行创建。例子：

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}

let mut x = add(5,7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5,7);
```

对于存在生命周期参数的函数指针，其写法类似如下：

```
let subtype: &(for<'a> fn(&'a i32) -> &'a i32) = &((|x| x) as fn(&_) -> &_);
```

其中类型以 for 开头，<>中为一个或多个生命周期参数，后面为函数签名。

Trait 类型

Trait object types

trait object 类型语法格式如下：

```
TraitObjectType :
   dyn? TypeParamBounds

TraitObjectTypeOneBound :
   dyn? TraitBound
```

trait object 是一种类型不透明的值，该值实现了一组 trait,这组 trait 由一个对象安全的 base trait 和任意数量的 auto traits 组成。

Trait object 以可选关键字 dyn 开始后跟一组 trait bound，但对 trait bound 有以下限制：

- 除了第一个 trait 之外的所有 trait 都必须是 auto traits。
- 生命周期参数不能超过一个。
- 不允许使用 opt-out bound（例如?Sized)。
- trait 的路径可以用括号括起来。

下面是一些 trait object 类型的例子：

```
Trait
dyn Trait
dyn Trait + Send
dyn Trait + Send + Sync
dyn Trait + 'static
dyn Trait + Send + 'static
dyn Trait +
dyn 'static + Trait.
dyn (Trait)
```

注意：dyn 不建议忽略，除非代码库支持使用 Rust 1.26 或更低版本进行编译。

如果两个 trait object 类型的 base trait 彼此别名(等效)并且 auto traits 相同且生命周期参数相同，则两个 trait object 类型彼此别名。例如，dyn Trait + Send + UnwindSafe 与 dyn Trait + UnwindSafe + Send 是一样的。

由于值属于哪种具体类型的不透明性，trait object 是动态大小类型。 和所有的 DST(dynamically sized types) 一样，trait 对象被用在某种类型的指针后面； 例如 &dyn SomeTrait 或 Box<dyn SomeTrait>。 指向 trait object 的指针的每个实例包含了以下数据：

- 指向实现 SomeTrait 的 T 类型实例的指针。
- 一个虚方法表，通常简称为 vtable，其中包含了实现了 SomeTrait 的每个方法及其 T 父 trait 的每个方法的函数指针。

trait object 的作用是允许方法的“后期绑定”。在运行时调用 trait object 上的方法会导致虚拟分派：即，从 trait object 的 vtable 加载一个函数指针并间接调用。每个 vtable 条目的实际实现可能因对象而异。

例子：

```
trait Printable {
    fn stringify(&self) -> String;
}

impl Printable for i32 {
    fn stringify(&self) -> String { self.to_string() }
}

fn print(a: Box<dyn Printable>) {
    println!("{}", a.stringify());
}

fn main() {
    print(Box::new(10) as Box<dyn Printable>);
}
```

Impl trait type

impl trait 类型的语法如下：

```
ImplTraitType : impl TypeParamBounds

ImplTraitTypeOneBound : impl TraitBound
```

函数可以将参数声明为匿名类型，这个匿名类型有一组 trait bounds，并且函数只能使用匿名类型参数的 trait bound 的可用方法。它们被写成 impl 后跟一组 trait bound。例如：

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

如果需要同时实现两个不同的 trait，可以通过"+"语法实现。例如：

```
pub fn notify(item: impl Summary + Display) {
 ...
}
```

函数也可以返回一个抽象返回类型，这个抽象返回类型有一组 trait bounds，调用者只能使用抽象类型的 trait bound 的可用方法。例如：

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub fn notify(item: impl Summary) -> impl Summary {
    println!("Breaking news! {}", item.summarize());
    return item;
}
```

类型递归

定义类型(struct, enum, union)可能是递归的。每个 enum 或 struct 或 union 的字段都可以直接或间接引用 enum 或 struct 类型本身。这种递归存在如下限制：

- 递归类型只能出现在定义类型中，不能出现在类型别名或者数组或元组等结构化类型中。例如 type Rec = &'static [Rec] 是不允许的。
- 递归类型的大小必须是有限的，因此该类型的递归字段必须是指针类型。
- 递归类型定义可以跨越模块边界，但不能跨越模块可见性边界或 crate 边界（为了简化模块系统和类型检查器）。

例子：

```
enum List<T> {
    Nil,
    Cons(T, Box<List<T>>)
}

let a: List<i32> = List::Cons(7, Box::new(List::Cons(13, Box::new(List::Nil))));
```

子类型

子类型是隐式的，可以发生在类型检查或类型推断的任何阶段。在 rust 中，子类型的判断是非常严格的，子类型的产生可能是由于两个类型

的生命周期参数存在差异，或者是两个类型的构成类型具有子类型关系的差异。

具有生命周期参数的类型有以下几个：引用指针、函数指针、trait object、impl trait。对于引用指针，具有更长寿命的生命周期参数的类型是子类型，对于函数指针、trait object、impl trait 具有更长寿命的生命周期参数的类型是父类型。具体情况查看子类型推断。

例子：

```
// 由于 'static 比生命周期参数 'a 的寿命长，&'static str 是 &'a str 的子类型。
let subtype: &'static str = "hi";
let supertype: &'a str = s;

// Here 'a is substituted for 'static
let subtype: &(for<'a> fn(&'a i32) -> &'a i32) = &((|x| x) as fn(&_) -> &_);
let supertype: &(fn(&'static i32) -> &'static i32) = subtype;

// This works similarly for trait objects
let subtype: &(for<'a> Fn(&'a i32) -> &'a i32) = &|x| x;
let supertype: &(Fn(&'static i32) -> &'static i32) = subtype;

// We can also substitute one higher-ranked lifetime for another
let subtype: &(for<'a, 'b> fn(&'a i32, &'b i32))= &((|x, y| {}) as fn(&_, &_));
let supertype: &for<'c> fn(&'c i32, &'c i32) = subtype;
```

子类型推断

子类型推断根据两个类型的生命周期参数差异，或者构成类型的差异进行判断的：

- 如果两个类型相同，只有生命周期参数不同，具有更长寿命的生命周期参数的类型是子类型。
- 如果两个类型不同，但是结构一样(即都是 F<T>,F 代表类型构造器)，那么根据两个类型与构成类型的关系，F 可能有如下三种：
  - 如果 T1 是 T2 的子类型，并且 F<T1>也是 F<T2>的子类型，这种 F 被称为 covariant。比如 lifetime 就是这种情况。
  - 如果 T1 是 T2 的子类型，并且 F<T2>是 F<T1>的子类型，这种 F 被称为 contravariant。比如 fn(T) -> () 属于这种情况。
  - 否则 F 称为 invariant。

下面是不同的 F 对应的情况：

Type(F<T>)	'a 存在差异	T 存在差异
&'a T	covariant	covariant
&'a mut T	covariant	invariant
*const T		covariant
*mut T		invariant
[T] and [T; n]		covariant
fn() -> T		covariant
fn(T) -> ()		contravariant
fn(T) -> T		invariant
std::cell::UnsafeCell<T>		invariant
std::marker::PhantomData<T>		covariant
dyn Trait<T> + 'a	covariant	invariant

其他 struct、enum、union 和 tuple 等构造类型的差异是通过查看它们的字段类型的差异来决定的。

例子：

```
use std::cell::UnsafeCell;
struct Variance<'a, 'b, T, U: 'a> {
    x: &'a U,                // This makes `Variance` covariant in 'a, and would
                             // make it covariant in U, but U is used later
    y: *const T,             // Covariant in T
    z: UnsafeCell<&'b f64>, // Invariant in 'b
    w: *mut U,               // Invariant in U, makes the whole struct invariant
}
```

类型强制转换

类型强制转换是改变值类型的隐式操作。它们在特定位置自动发生，并且在实际可强制的类型方面受到高度限制。任何由强制转换允许的转换也可以由类型转换运算符 as 显式执行。

强制转换点

强制转换只能发生在程序中的某些强制位置；这些地方通常是在所需类型是显式声明的，或者是可以通过显式类型传播来派生的地方（无类型推断符_）。可能的强制转换点是：

- 给出显式类型的 let 语句。例如， &mut 42 在下面被强制为 &i8 类型：

```
let _: &i8 = &mut 42;
```

- static 和 const item 声明（类似于 let 语句）。
- 函数调用的参数。被强制转换是实参值，被强制转换为形参的类型。例如， &mut 42 在下面被强制为 &i8 类型：

```
fn bar(_: &i8) { }

fn main() {
    bar(&mut 42);
}
```

对于方法调用，接收者（self 参数）只能利用 unsized 强制转换。

- struct、union 或 enum 变量字段的实例化。例如， &mut 42 在下面被强制为 &i8 类型：

```
struct Foo<'a> { x: &'a i8 }

fn main() {
    Foo { x: &mut 42 };
}
```

- 函数返回值，即块的最后一行（如果它不是以分号结尾的）或 return 语句中的任何表达式。例如，x 被强制为类型 &dyn Display：

```
use std::fmt::Display;
fn foo(x: &u32) -> &dyn Display {
    x
}
```

如果这些强制转换点之一中的表达式是强制转换传播表达式，则该表达式中的相关子表达式也是强制转换点。 传播从这些新的强制转换点递归。以下表达式是传播表达式：

- 数组字面量，其中数组的类型为 [U; n]。 数组字面量中的每个子表达式都是强制转换为 U 类型的强制转换点。
- 具有重复语法的数组字面量，其中数组的类型为 [U; n]。 重复的子表达式是强制转换为 U 类型的强制转换点。
- 元组表达式。 每个子表达式都是相应类型的强制转换站点。
- 圆括号的子表达式（(e)）。如果表达式的类型为 U，则子表达式是 U 类型的强制转换点。
- 块表达式。如果块的类型为 U，则块中的最后一个表达式（如果它不是以分号结尾的）是 U 类型的强制转换点。

强制转换规则

以下类型之间允许强制转换：

- 如果 T 是 U 的子类型，则 T 可以强制转换为 U。
- 如果 T1 可以强制转换为 T2，T2 可以强制转换为 T3，那么 T1 可以强制转换为 T3。
- &mut T 可以强制转换为 &T。
- *mut T 可以强制转换为 *const T。
- &T 可以强制转换为 *const T。
- &mut T 可以强制转换为 *mut T。
- 当 T: Deref<Target=U> 时，&T 可以强制转换为 &U，&mut T 可以强制转换为 &U。
- 当 T: DerefMut<Target=U> 时，&mut T 可以强制转换为 &mut U。
- 没有捕获的闭包可以转换为 fn 指针类型。
- Never 类型"!"可以强制任意类型 T。
- unsized 强制转换。

例子：

```
use std::ops::Deref;

struct CharContainer {
    value: char,
}

impl Deref for CharContainer {
    type Target = char;

    fn deref<'a>(&'a self) -> &'a char {
        &self.value
    }
}

fn foo(arg: &char) {}
```

```
fn main() {
    let x = &mut CharContainer { value: 'y' };
    foo(x); //&mut CharContainer is coerced to &char.
}
```

unsized 强制转换

unsized 强制转换是将固定尺寸类型(sized types)转换为非固定尺寸类型(unsized types)有关。对于实现了 CoerceUnsized Trait 的类型，那么其可以强制转换到对应的类型，例如 T 实现了 CoerceUnsized<U> trait，那么 T 可以强制转换为 U。如果 T 实现 Unsize<U> trait，那么 TyCtor(T)或自动实现 CoerceUnsized< TyCtor(U)> trait，TyCtor 可以是如下一些：&T、&mut T、*const T、*mut T、Box<T>等。

满足以下情况之一 rust 会自动实现 Unsize trait:

- 对于[T; n]类型，会自动实现 Unsize<[T]> trait。因此类似&[T; n]的指针类型能够转换为&[T]的指针类型。
- 如果 T 实现了 Trait trait，那么会自动实现 Unsize<dyn Trait> trait。因此类似&T(T:Trait)能够转换为&dyn Trait。
- 如果 T 满足以下条件，那么 Foo<..., T, ...>会自动实现 Unsize<Foo<..., U, ...>> trait：
  - T 实现了 Unsize<U> trait。
  - Foo 是一个 struct 类型。
  - 只有 Foo 的最后一个字段具有涉及 T 的类型。
  - T 不属于任何其他字段的类型。
  - 如果 Foo 的最后一个字段的类型是 Bar<T>，Bar<T>也实现了 Unsize<Bar<U>> trait。

最小上限强制转换

在某些情况下，编译器必须将多种类型强制转换在一起以尝试找到最通用的类型。这称为“最小上限”(Least Upper Bound、LUB)强制转换。

LUB 强制转换仅在以下情况下使用：

- 查找一系列 if 分支的公共类型。
- 查找一系列 match case 的通用类型。
- 查找一系列数组元素的通用类型。
- 查找具有多个 return 语句的闭包的返回类型。
- 检查具有多个 return 语句的函数的返回类型。

在每种情况下，都有一组类型 T0..Tn 相互强制转换为某个未知的目标类型 T_t。计算 LUB 强制是迭代完成的，目标类型 T_t 从类型 T0 开始，对于每个新类型 Ti，考虑如下情况：

- 如果可以将 Ti 强制为当前目标类型 T_t，则不进行任何更改。
- 否则，检查是否可以将 T_t 强制为 Ti； 如果是，则将 T_t 更改为 Ti。
- 如果不是，尝试计算 T_t 和 Ti 的共有父类型，它将成为新的目标类型。

例子：

```
// For if branches
let bar = if true {
    a
} else if false {
    b
} else {
    c
};

// For match arms
let baw = match 42 {
    0 => a,
    1 => b,
    _ => c,
};

// For array elements
let bax = [a, b, c];

// For closure with multiple return statements
let clo = || {
    if true {
        a
    } else if false {
        b
    } else {
        c
```

```
    }
};
let baz = clo();

// For type checking of function with multiple return statements
fn foo() -> i32 {
    let (a, b, c) = (0, 1, 2);
    match 42 {
        0 => a,
        1 => b,
        _ => c,
    }
}
```

类型析构

当一个初始化的变量或临时变量离开作用域时，它的析构函数就会运行，或者说变量被 drop。 如果一个被赋值变量已初始化，赋值还会运行该变量的析构函数。如果变量已部分初始化，则仅删除其初始化字段。

对类型 T 的析构包括如下步骤：

- 如果 T 实现了 Drop trait，那么调用<T as std::ops::Drop>::drop 方法。
- 递归运行 T 的所有字段的析构函数。
  - struct 的字段按声明顺序进行。
  - enum variant 按声明顺序进行。
  - 元组按顺序进行。
  - 数组或 slice 按照元素顺序进行。
  - 闭包中由 move 关键字捕获的变量的析构顺序是未知的。
  - trait object 运行底层的析构函数。
  - 其他类型不会导致任何进一步析构。

如果必须手动运行析构函数，例如在实现自己的智能指针时，可以使用 std::ptr::drop_in_place。

drop 作用域

每个变量或临时变量都与一个 drop 作用域相关联。 当控制流离开 drop 作用域时，所有与该作用域关联的变量都以声明（对于变量）或创建（对于临时变量）的相反顺序删除。

drop 作用域是在将 for、if let 和 while let 表达式替换为使用 match 的等效表达式之后确定的。不区分重载运算符和内置运算符，也不考虑绑定模式。

给定一个函数或闭包，有以下 drop 作用域：

- 整个函数。
- 每个语句。
- 每个表达式。
- 每个块，包括函数体。在块表达式的情况下，块和表达式的作用域是相同的作用域。
- match 表达式的每个分支。

drop 作用域相互嵌套时，当同时离开多个作用域时，例如从函数返回时，作用域关联的变量会从内向外 drop：

- 整个函数作用域是最外面的作用域。
- 函数体块包含在整个函数的作用域内。
- 表达式语句中表达式的父级是表达式语句的作用域。
- let 语句的初始化项的父级是 let 语句的作用域。
- 语句作用域的父级是包含该语句的块的作用域。
- 匹配守卫表达式的父级是匹配守卫所针对的 match 分支的作用域。
- 匹配表达式中 => 之后的表达式的父级是它所在的 match 分支的作用域。
- 匹配分支作用域的父级是它所属的匹配表达式的作用域。
- 所有其他作用域的父级是直接封闭表达式的作用域。

函数参数作用域

所有函数参数都在整个函数的 drop 作用域内，因此在执行函数时最后进行 drop。如果在参数 pattern 中引入任何绑定之后，每个实际的函数参数都会被立即 drop。

例子：

```
// Drops the second parameter, then `y`, then the first parameter, then `x`
fn patterns_in_parameters(
```

```
    (x, _): (PrintOnDrop, PrintOnDrop),
    (_, y): (PrintOnDrop, PrintOnDrop),
) {}

// drop order is 3 2 0 1
patterns_in_parameters(
    (PrintOnDrop("0"), PrintOnDrop("1")),
    (PrintOnDrop("2"), PrintOnDrop("3")),
);
```

局部变量作用域

在 let 语句中声明的局部变量与包含 let 语句的块的 drop 作用域相关联。

在匹配表达式中声明的局部变量与声明它们的匹配分支的 drop 作用域相关联。如果在一个匹配表达式的同一个分支中使用多个匹配模式，那么将使用任意一个未知的匹配模式来确定删除顺序。

例子：

```
let declared_first = PrintOnDrop("Dropped last in outer scope");
{
    let declared_in_block = PrintOnDrop("Dropped in inner scope");
}
let declared_last = PrintOnDrop("Dropped first in outer scope");
```

临时作用域

表达式的临时作用域是用于在上下文中保存该表达式结果的临时变量的作用域，除非它被提升。

除非作用域临时扩展，表达式的临时作用域是包含该表达式的最小作用域，作用域有以下情况之一：

- 整个函数体。
- 一个语句。
- if、while 或 loop 表达式的主体。
- if 表达式的 else 块。
- if 或 while 表达式或匹配守卫的条件表达式。
- 匹配分支的表达式。
- 惰性布尔表达式的第二个操作数。

例子：

```

let local_var = PrintOnDrop("local var");

// Dropped once the condition has been evaluated
if PrintOnDrop("If condition").0 == "If condition" {
    // Dropped at the end of the block
    PrintOnDrop("If body").0
} else {
    unreachable!()
};

// Dropped at the end of the statement
(PrintOnDrop("first operand").0 == ""
// Dropped at the )
|| PrintOnDrop("second operand").0 == "")
// Dropped at the end of the expression
|| PrintOnDrop("third operand").0 == "";

// Dropped at the end of the function, after local variables.
// Changing this to a statement containing a return expression would make the
// temporary be dropped before the local variables. Binding to a variable
// which is then returned would also make the temporary be dropped first.
match PrintOnDrop("Matched value in final expression") {
    // Dropped once the condition has been evaluated
    _ if PrintOnDrop("guard condition").0 == "" => (),
    _ => (),
}
```

临时扩展作用域

let 语句中表达式的临时作用域有时会扩展到包含 let 语句的块的作用域。当通常的临时作用域太小时会这样做。例如：

```
let x = &mut 0;
// Usually a temporary would be dropped by now, but the temporary for `0` lives
// to the end of the block.
println!("{}", x);
```

如果借用、解引用、字段或元组索引表达式具有扩展的临时作用域，那么它的操作数也是如此。如果索引表达式具有扩展的临时作用域，则索引表达式也具有扩展的临时作用域。

基于 pattern 的扩展

可扩展的 pattern 是以下两种之一：

- 通过 ref 或 mut ref 的标识符绑定模式。
- 结构体模式、元组模式、元组结构体模式或切片模式，其中至少一个直接子模式是扩展模式。

因此 ref x, V(ref x) 和 [ref x, y] 都是扩展 pattern，但 x, &ref x 和 &(ref x,) 不是。

如果 let 语句中的 pattern 是扩展模式，则扩展初始化表达式的临时作用域。

例子：

```
// The temporary that stores the result of `temp()` lives in the same scope
// as x in these cases.
let ref x = temp();
let ref x = *&temp();
```

基于表达式的扩展

对于带有初始值设定项的 let 语句，扩展表达式是以下表达式之一：

- 初始化表达式。
- 扩展的借用表达式的操作数。
- 扩展的数组、强制转换、花括号结构或元组表达式的操作数。
- 扩展块表达式的最后一个表达式。

因此 &mut 0, (&1, &mut 2) 和 Some { 0: &mut 3 } 中的借用表达式都是扩展表达式。 &0 + &1 和 Some(&mut 0) 中的借用不是：后者在语法上是一个函数调用表达式。

任何扩展的借用表达式的操作数都扩展了其临时作用域。

例子：

```
// The temporary that stores the result of `temp()` lives in the same scope
// as x in these cases.
let x = &temp();
let x = &temp() as &dyn Send;
let x = (&*&temp(),);
let x = { [Some { 0: &temp(), }] };
```

避免析构

在 Rust 中不运行析构函数是安全的，即使它的类型不是“’static”。

std::mem::ManuallyDrop 提供了一个包装器来防止自动 drop 变量或字段。

省略生命周期

Rust 有一些规则，允许在编译器可以推断出合理的默认选择的各个地方省略生命周期。

函数中的生命周期省略

可以在函数项、函数指针和闭包 trait 签名中省略生命周期参数。编译器采用三条规则来判断引用何时不需要明确的注解，其中函数或方法的参数的生命周期被称为"输入生命周期"（input lifetimes），而返回值的生命周期被称为"输出生命周期"（output lifetimes）。如果编译器检查完这三条规则后仍然存在没有计算出生命周期的引用，编译器将会停止并生成错误。这些规则适用于 fn 定义，以及 impl 块：

1. 每一个是引用的参数都有它自己的生命周期参数。例如：fn foo<’a, ’b>(x: &’a i32, y: &’b i32)
2. 如果只有一个输入生命周期参数，那么它被赋予所有输出生命周期参数。例如：fn foo<’a>(x: &’a i32) -> &’a i32。
3. 如果方法有多个输入生命周期参数并且其中一个参数是 &self 或 &mut self，那么所有输出生命周期参数被赋予 self 的生命周期。

占位符生命周期 ’_ 也可用于以相同的方式推断生命周期。

例子：

```
fn print1(s: &str);                                 // elided
fn print2(s: &'_ str);                              // also elided
fn print3<'a>(s: &'a str);                          // expanded

fn debug1(lvl: usize, s: &str);                     // elided
fn debug2<'a>(lvl: usize, s: &'a str);              // expanded

fn substr1(s: &str, until: usize) -> &str;          // elided
fn substr2<'a>(s: &'a str, until: usize) -> &'a str;  // expanded

fn get_mut1(&mut self) -> &mut dyn T;               // elided
fn get_mut2<'a>(&'a mut self) -> &'a mut dyn T;     // expanded

fn args1<T: ToCStr>(&mut self, args: &[T]) -> &mut Command;                  // elided
fn args2<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command; // expanded

fn new1(buf: &mut [u8]) -> Thing<'_>;               // elided - preferred
fn new2(buf: &mut [u8]) -> Thing;                   // elided
fn new3<'a>(buf: &'a mut [u8]) -> Thing<'a>;        // expanded

type FunPtr1 = fn(&str) -> &str;                    // elided
type FunPtr2 = for<'a> fn(&'a str) -> &'a str;      // expanded

type FunTrait1 = dyn Fn(&str) -> &str;              // elided
type FunTrait2 = dyn for<'a> Fn(&'a str) -> &'a str;  // expanded
```

trait object 的默认生命周期

当 trait object 完全省略生命周期 bound 时，会使用默认的 trait object 生命周期 bound 代替上面定义的生命周期参数省略规则。但是如果 '_ 用作生命周期 bound，则该界限遵循通常的省略规则。

默认的 trait object 生命周期 bound 规则如下：

- 如果 trait object 用作泛型的类型参数，则首先根据泛型包含的类型参数来尝试推断 bound：
  - 如果泛型包含有唯一的生命周期 bound，那么这是就是 trait object 的默认生命周期 bound。
  - 如果泛型包含有多个生命周期 bound，则 trait object 必须显示的指定生命周期 bound。
- 如果 trait object 不满足上述规则：
  - 如果 trait 是用单个生命周期 bound 定义的，则使用该生命周期 bound。
  - 如果 'static 用于任意生命周期 bound，则使用 'static。
  - 如果 trait 没有生命周期 bound，那么生命周期由表达式推断出来，如果 trait bound 不在表达式中，则是'static。

例子：

```
// For the following trait...
trait Foo { }

// These two are the same as Box<T> has no lifetime bound on T
type T1 = Box<dyn Foo>;
type T2 = Box<dyn Foo + 'static>;

// ...and so are these:
impl dyn Foo {}
impl dyn Foo + 'static {}

// ...so are these, because &'a T requires T: 'a
type T3<'a> = &'a dyn Foo;
type T4<'a> = &'a (dyn Foo + 'a);

// std::cell::Ref<'a, T> also requires T: 'a, so these are the same
type T5<'a> = std::cell::Ref<'a, dyn Foo>;
type T6<'a> = std::cell::Ref<'a, dyn Foo + 'a>;

// This is an example of an error.
struct TwoBounds<'a, 'b, T: ?Sized + 'a + 'b> {
    f1: &'a i32,
    f2: &'b i32,
```

```
    f3: T,
}
type T7<'a, 'b> = TwoBounds<'a, 'b, dyn Foo>;
//                                  ^^^^^^^
// Error: the lifetime bound for this object type cannot be deduced from context
```

'static 生命周期省略

除非指定了显式生命周期 bound，否则引用类型的常量和静态声明都具有隐式的'static。

例子：

```
// STRING: &'static str
const STRING: &str = "bitstring";

struct BitsNStrings<'a> {
    mybits: [u32; 2],
    mystring: &'a str,
}

// BITS_N_STRINGS: BitsNStrings<'static>
const BITS_N_STRINGS: BitsNStrings<'_> = BitsNStrings {
    mybits: [1, 2],
    mystring: STRING,
};
```

请注意，如果 static 或 const 项包含本身包含引用的函数或闭包，编译器将首先尝试标准省略规则。 如果它无法通过其通常的规则解析生命周期，那么它就会出错。例如：

```
// Resolved as `fn<'a>(&'a str) -> &'a str`.
const RESOLVED_SINGLE: fn(&str) -> &str = |x| x;

// Resolved as `Fn<'a, 'b, 'c>(&'a Foo, &'b Bar, &'c Baz) -> usize`.
const RESOLVED_MULTIPLE: &dyn Fn(&Foo, &Bar, &Baz) -> usize = &somefunc;

// There is insufficient information to bound the return reference lifetime
// relative to the argument lifetimes, so this is an error.
const RESOLVED_STATIC: &dyn Fn(&Foo, &Bar) -> &Baz = &somefunc;
//                                             ^
// this function's return type contains a borrowed value, but the signature
// does not say whether it is borrowed from argument 1 or argument 2
```

类型布局

类型的布局包括其大小、对齐方式以及字段的相对偏移量。对于枚举，判别式的布局和解释方式也是类型布局的一部分。

大小和对齐

所有值都有一个对齐方式和大小。

- 值的对齐指定了哪些地址可用于存储值。对齐 n 的值只能存储在 n 的倍数的地址中。 例如，对齐为 2 的值必须存储在偶数地址，而对齐为 1 的值可以存储在任何地址。对齐以字节为单位，并且必须至少为 1，并且始终为 2 的幂。可以使用 align_of_val 函数检查值的对齐情况。
- 值的大小是具有该项类型（包括对齐填充）的数组中连续元素之间的偏移量（以字节为单位）。 值的大小始终是其对齐方式的倍数。 可以使用 size_of_val 函数检查值的大小。

编译时已知的类型的所有值都具有相同的大小和对齐方式，实现了 Sized trait 并且可以使用 size_of 和 align_of 函数进行检查。对于编译时大小未定的类型称为动态大小类型。

原始类型布局

下表中给出了大多数原始类型的大小：

Type	size_of::<Type>()
bool	1
u8 / i8	1
u16 / i16	2

Type	size_of::<Type>()
u32 / i32	4
u64 / i64	8
u128 / i128	16
f32	4
f64	8
char	4

usize 类型和 isize 类型的大小足以包含目标平台上的每个地址。 例如，在 32 位目标上，是 4 个字节，在 64 位目标上，是 8 个字节。

大多数原始类型通常与其大小对齐，尽管这是特定于平台的行为。 特别是，在 x86 上 u64 和 f64 只对齐到 32 位。

指针和引用布局

指针和引用具有相同的布局。 指针或引用的可变性不会改变布局。

指向固定大小类型的指针与 usize 类型 具有相同的大小和对齐方式。

指向动态大小的类型的指针是动态调整的。 大小和对齐方式保证至少等于指针的大小和对齐方式。目前所有指向动态大小类型的指针都是 usize 大小的两倍，并且具有相同的对齐方式。

数组布局

数组的布局使得数组的第 n 个元素从数组的开头偏移 n * 类型字节的大小。

[T; n] 数组的大小为 size_of::<T>() * n 并且与 T 的对齐方式相同。

slice 布局

切片与它们切的数组部分具有相同的布局。

注意，这里指的原始 [T] 类型，而不是指向切片的指针（&[T]、Box<[T]> 等）。

str 布局

字符串切片是字符的 UTF-8 表示，与 [u8] 类型的切片具有相同的布局。

元组布局

元组对其布局没有任何保证。唯一的例外是单元元组 (())，它保证为零大小类型，大小为 0，对齐方式为 1。

trait object 布局

trait object 与其实际值具有相同的布局。注意：这里指 trait object 类型，而不是 trait object 指针（&dyn Trait、Box<dyn Trait>）。

闭包布局

闭包没有布局保证。

布局陈述

所有用户定义的复合类型（struct、enum、union）都有一个陈述，用于指定类型的布局。类型的可能陈述有以下几种：

- Default representation
- C representations
- Primitive representations
- Transparent representations

由于陈述是项目的属性，因此陈述不依赖于泛型参数。 任何具有相同名称的两种类型具有相同的陈述。 例如， Foo<Bar> 和 Foo<Baz> 都有相同的陈述。

类型的陈述可以更改字段之间的填充，但不会更改字段本身的布局。 例如，具有 C 表示的结构包含具有默认陈述的 struct Inner 不会更改 Inner 的布局。

类型的布局陈述可以通过对其应用 repr 属性来更改。例如 C 表示形式的结构。:

```
#[repr(C)]
struct ThreeInts {
    first: i16,
    second: i8,
    third: i32
}
```

可以分别使用 align 和 packed 修饰符来改变对齐。它们改变属性中指定的陈述。 如果未指定陈述，则更改默认陈述：

```
// Default representation, alignment lowered to 2.
#[repr(packed(2))]
struct PackedStruct {
    first: i16,
    second: i8,
    third: i32
}

// C representation, alignment raised to 8
#[repr(C, align(8))]
struct AlignedStruct {
    first: i16,
    second: i8,
    third: i32
}
```

默认布局陈述

没有 repr 属性陈述的类型具有默认的布局陈述。这种陈述也称为 rust 陈述。这种陈述不会保证布局。

C 布局陈述

C 布局陈述是为双重目的而设计的。 一个目的是创建可与 C 语言互操作的类型。 第二个目的是创建可以根据数据布局对其进行合理的操作的类型，例如将值重新解释为不同的类型。

这种布局陈述可以应用于 struct、enum、union。 例外是 zero variant enum（如 enum xxx {}）， 对其添加 C 布局陈述是错误的。

Struct

结构体的对齐方式是其中最对齐的字段的对齐方式。字段的大小和偏移量由以下算法确定：

- 从 0 字节的当前偏移量开始。
- 对结构体中声明顺序的每个字段，首先确定字段的大小和对齐方式。 如果当前偏移量不是字段对齐的倍数，则将填充字节添加到当前偏移量，直到它是字段对齐的倍数。 该字段的偏移量就是当前的偏移量。 然后将当前偏移量增加字段的大小。
- 最后，结构的大小是当前偏移量四舍五入到结构对齐的最接近倍数。

下面是算法的伪代码：

```
/// Returns the amount of padding needed after `offset` to ensure that the
/// following address will be aligned to `alignment`.
fn padding_needed_for(offset: usize, alignment: usize) -> usize {
    let misalignment = offset % alignment;
    if misalignment > 0 {
        // round up to next multiple of `alignment`
        alignment - misalignment
    } else {
        // already a multiple of `alignment`
        0
    }
}

struct.alignment = struct.fields().map(|field| field.alignment).max();

let current_offset = 0;

for field in struct.fields_in_declaration_order() {
    // Increase the current offset so that it's a multiple of the alignment
    // of this field. For the first field, this will always be zero.
    // The skipped bytes are called padding bytes.
    current_offset += padding_needed_for(current_offset, field.alignment);
```

```
    struct[field].offset = current_offset;

    current_offset += field.size;
}

struct.size = current_offset + padding_needed_for(current_offset, struct.alignment);
```

union

使用#[repr(C)] 声明的 union 将与目标平台的 C 语言中的 C union 声明具有相同的大小和对齐方式。 union 的大小是其所有字段中的最大大小四舍五入到其对齐方式，对齐方式是所有字段中的最大对齐方式。这些最大值可能来自不同的字段。

例子：

```
#[repr(C)]
union Union {
    f1: u16,
    f2: [u8; 4],
}

assert_eq!(std::mem::size_of::<Union>(), 4);  // From f2
assert_eq!(std::mem::align_of::<Union>(), 2); // From f1

#[repr(C)]
union SizeRoundedUp {
   a: u32,
   b: [u16; 3],
}

assert_eq!(std::mem::size_of::<SizeRoundedUp>(), 8);  // Size of 6 from b,
                                                      // rounded up to 8 from
                                                      // alignment of a.
assert_eq!(std::mem::align_of::<SizeRoundedUp>(), 4); // From a
```

无字段 enum

对于无字段 enum(即没有值绑定)，使用#[repr(C)] 声明的 enum 的大小和对齐方式与目标平台的 C ABI 的默认枚举大小和对齐方式一致。

注意：C 语言中的枚举和 Rust 的具有这种表示的无字段枚举之间存在重大差异。 C 中的枚举主要是 typedef 加上一些命名常量； 换句话说，枚举类型的对象可以保存任何整数值。 例如，这通常用于 C 中的位标志。相比之下，Rust 的无字段枚举只能合法地保存判别值，其他一切都是未定义的行为。 因此，在 FFI 中使用无字段枚举来模拟 C 枚举通常是错误的。

有字段 enum

对于有字段 enum(即存在值绑定)，使用#[repr(C)] 声明的 enum 等效于一个带有两个字段的 repr(C) struct：

- 一个字段为删除 enum 中所有字段的 repr(C) 无字段 enum。
- 一个字段为具有 snum 中所有字段的 repr(C) union。

例子：

```
// This Enum has the same representation as ...
#[repr(C)]
enum MyEnum {
    A(u32),
    B(f32, u64),
    C { x: u32, y: u8 },
    D,
 }

// ... this struct.
#[repr(C)]
struct MyEnumRepr {
    tag: MyEnumDiscriminant,
    payload: MyEnumFields,
}

// This is the discriminant enum.
```

```
#[repr(C)]
enum MyEnumDiscriminant { A, B, C, D }

// This is the variant union.
#[repr(C)]
union MyEnumFields {
    A: MyAFields,
    B: MyBFields,
    C: MyCFields,
    D: MyDFields,
}

#[repr(C)]
#[derive(Copy, Clone)]
struct MyAFields(u32);

#[repr(C)]
#[derive(Copy, Clone)]
struct MyBFields(f32, u64);

#[repr(C)]
#[derive(Copy, Clone)]
struct MyCFields { x: u32, y: u8 }

// This struct could be omitted (it is a zero-sized type), and it must be in
// C/C++ headers.
#[repr(C)]
#[derive(Copy, Clone)]
struct MyDFields;
```

原始类型陈述

原始类型陈述是与使用原始整数类型同名的陈述。 即：u8、u16、u32、u64、u128、usize、i8、i16、i32、i64、i128 和 isize。

原始类型陈述只能应用于枚举并且无论枚举有字段还是无字段都具有不同的行为。zero variant enum（如 enum xxx {}）如果具有原始类型陈述是错误的。将两个原始类型陈述组合在一起是错误的。

无字段 enum

对于无字段 enum，原始类型陈述将大小和对齐方式设置为与同名原始类型相同的大小和对齐方式。

例如，具有 u8 表示的无字段枚举只能具有介于 0 和 255 之间的判别式。

有字段 enum

对于有字段 enum(即存在值绑定)，enum 等效于一个带有两个字段的 repr(C) struct：

- 一个字段为删除 enum 中所有字段的 repr(C) 无字段 enum。
- 一个字段为具有 snum 中所有字段的 repr(C) union。

原始类型陈述仅作用于拆分后的无字段 enum 上，例如：

```
// This enum has the same representation as ...
#[repr(u8)]
enum MyEnum {
    A(u32),
    B(f32, u64),
    C { x: u32, y: u8 },
    D,
 }

// ... this union.
#[repr(C)]
union MyEnumRepr {
    A: MyVariantA,
    B: MyVariantB,
    C: MyVariantC,
    D: MyVariantD,
```

```
}

// This is the discriminant enum.
#[repr(u8)]
#[derive(Copy, Clone)]
enum MyEnumDiscriminant { A, B, C, D }

#[repr(C)]
#[derive(Clone, Copy)]
struct MyVariantA(MyEnumDiscriminant, u32);

#[repr(C)]
#[derive(Clone, Copy)]
struct MyVariantB(MyEnumDiscriminant, f32, u64);

#[repr(C)]
#[derive(Clone, Copy)]
struct MyVariantC { tag: MyEnumDiscriminant, x: u32, y: u8 }

#[repr(C)]
#[derive(Clone, Copy)]
struct MyVariantD(MyEnumDiscriminant);
```

混合原始类型陈述和 C 布局陈述

对于带有字段的枚举，可以组合 repr(C) 和原始类型陈述（例如，repr(C, u8)）。通过这种方式，可以专门设置拆分后的无字段 enum 的大小和对齐方式。

例如如果选择 u8 表示，则判别式枚举的大小和对齐为 1 个字节：

```
#[repr(C, u8)] // `u8` was added
enum MyEnum {
    A(u32),
    B(f32, u64),
    C { x: u32, y: u8 },
    D,
 }

// ...

#[repr(u8)] // So `u8` is used here instead of `C`
enum MyEnumDiscriminant { A, B, C, D }

// ...
```

透明陈述

透明陈述只能用于具有单个字段的 enum 或 struct，该 enum 或 struct 具有如下性质：

- 具有非零大小的单个字段。
- 任意数量的大小为 0 且对齐为 1 的字段（例如 PhantomData<T>）。

具有这种陈述的结构和枚举与单个非零大小的字段具有相同的布局和 ABI。因此为透明陈述，相当于将类型布局委托给另一种类型，因此不能与任何其他陈述一起使用。

alian 对齐修饰符

align 和 packed 修饰符可用于分别提高或降低 struct 和 union 的对齐。packed 也可能改变字段之间的填充。

对齐方式以#[repr(align(x))] 或#[repr(packed(x))] 的形式指定为整数参数。 对齐值必须是从 1 到 229 的 2 的幂。对于 packed，如果没有给出值，如#[repr(packed)]，则默认值为 1。

对于 align，如果指定的对齐方式小于没有 align 修饰符的类型的对齐方式，则该对齐方式不受影响。
对于 packed，如果指定的对齐方式大于没有 packed 修饰符的类型的对齐方式，那么对齐方式和布局不受影响。
为了定位字段，每个字段的对齐方式是指定的对齐方式和字段类型的对齐方式中较小的一个。

align 和 packed 修饰符不能应用于同一类型，并且一个 packed 类型不能传递性地包含另一个 align 类型。 align 和 packed 只能应用于默认布局陈述和 C 布局陈述，例如#[repr(C, align(8))]。

align 修饰符也可以应用于 enum。如果是，则对枚举的对齐方式的影响与将枚举包装在具有相同对齐修饰符的 newtype 结构中的效果相同。

注意：解引用未对齐的指针是未定义的行为，但是可以安全地创建指向未对齐的 packed 字段指针。

动态大小类型

大多数类型具有在编译时已知的固定大小，并实现了 trait Sized。大小仅在运行时已知的类型称为动态大小类型(dynamically sized types 简称 DST)。str、slice 和 trait object 是 DST 的两个 DST 示例。

这些类型只能在特定情况下使用：

- 指向 DST 的指针类型是有大小的，其大小是指向大小类型的指针的两倍：
  - slice 和 str 的指针还存储其元素数量。
  - trait object 的指针还存储了一个指向 vtable 的指针。
- 当类型参数中存在?Sized 的 bound 时，动态大小类型可以作为类型参数提供。默认情况下，任何类型参数都有一个 Sized bound。
- struct 可以包含一个 DST 作为最后一个字段，这使得 struct 本身就是一个 DST。
- 为 DST 实现的 trait 中，在 trait 定义的类型参数默认存在 Self: ?Sized。

特殊 type

Rust 编译器知道标准库中存在的某些类型。

Box<T>

Box<T> 有一些 Rust 目前不允许用户定义类型的特殊功能：

- Box<T> 的解引用操作和析构是内置。
- 方法可以将 Box<Self> 作为 receiver。
- 可以在与 T 相同的 crate 中为 Box<T> 实现 trait，孤儿规则阻止了其他泛型类型可以这样。

Rc<T>

方法可以将 Rc<Self> 作为 receiver。

Arc<T>

方法可以将 Arc<Self> 作为 receiver。

Pin<P>

方法可以将 Pin<P> 作为 receiver。

UnsafeCell<T>

std::cell::UnsafeCell<T> 用于内部可变性。它确保编译器不会执行对此类类型不正确的优化。它还确保具有内部可变性类型的 static item 不会被放置到只读的内存中。

PhantomData<T>

std::marker::PhantomData<T> 是一个大小为 0 对齐方式为 1 类型，出于 type variance、drop check 和 auto trait 的目的，它被认为拥有一个 T。

特殊 trait

Rust 编译器知道标准库中存在的某些 trait。

Operator Traits

std::ops 和 std::cmp 模块中的 trait 用于重载运算符、index 表达式和 call 表达式。

Deref/DerefMut trait

Deref/DerefMut trait 除了用于对一元运算符*进行重载之外，还用于方法调用解析和解引用强制转换。

Drop trait

Drop trait 提供了一个析构函数，每当实现了 drop trait 类型的值被销毁时就会运行实现的析构函数。

Copy trait

Copy trait 改变了实现它的类型的语义。实现 Copy trait 的类型的值在赋值时被复制而不是进行所有权转移。

编译器会对以下类型自动实现 Copy trait：
- u8、f32 等数值类型。
- char、bool、!这三个类型。
- 元素都实现 Copy trait 的元组。
- 元素实现 Copy trait 的数组。
- 共享引用类型(&T)，无论 T 是否实现 Copy trait。注意：可变引用指针是无法 copy 的，因为其必须唯一。
- 裸指针(*const T、*mut T)，无论 T 是否实现 Copy trait。
- 函数指针类型和函数项类型。

注意：只有未实现 Drop trait 的类型才能实现 Copy trait，并且只有字段都是实现 Copy trait 的类型才能实现 Copy trait。因此对于枚举，这意味着所有变体的所有字段都必须为 Copy trait。对于 union，这意味着所有变体都必须实现 Copy trait。

Clone

Clone trait 是 Copy trait 的父 trait。

编译器同样会对以下类型自动实现 Clone trait：
- 自动实现 Copy trait 的那些类型。
- 元素都实现 Clone trait 的元组。
- 元素实现 Clone trait 的数组。

Send trait

实现了 Send trait 的类型的值可以安全地从一个线程发送到另一个线程。

Sync trait

实现了 Sync trait 的类型的值可以安全地在多个线程之间共享。

必须为不可变的 static item 中使用的所有类型实现此 trait。

Sized trait

实现了 Sized trait 的类型的大小在编译时是已知的，也就是说，它不是动态大小的类型。

泛型的类型参数默认为 Sized。 Sized 总是由编译器自动实现，而不是由 impl 项实现。

Auto traits

Send、Sync、Unpin、UnwindSafe 和 RefUnwindSafe trait 是 auto trait。auto trait 具有特殊属性。

如果没有为给定类型的 auto trait 写出明确的实现或否定的实现，那么编译器会根据以下规则自动实现它：
- 对于&T, &mut T, *const T, *mut T, [T; n]，[T]，如果 T 实现了 auto trait，则对应实现这些类型的 auto trait。
- 函数项类型和函数指针类型自动实现 auto trait。
- 如果结构体、枚举、联合和元组的所有字段都实现了 auto trait，则它们也实现对应的 auto trait。
- 闭包中如果所有捕获的类型都实现了 auto trait，则闭包自动实现对应 auto trait

对于泛型类型，如果存在泛型 impl 可用，那么编译器不会为可以使用该实现的类型自动实现它，除非它们不满足必要的 trait bound(不包括 auto trait 部分是否匹配)。例如，标准库为所有 &T 实现 Send，其中 T 是 Sync；这意味着如果 T 是 Send 但不是 Sync，编译器将不会为 &T 实现 Send。

Auto trait 也可以有否定的实现，在标准库文档中显示为 impl !AutoTrait for T，用于覆盖自动实现。例如 *mut T 有一个否定的 Send 实现，所以 *mut T 不是 Send，即使 T 是。 目前没有稳定的方法来指定额外的否定实现； 它们只存在于标准库中。

Auto trait 可以作为附加绑定添加到任何 trait object，即使通常只允许一个 trait。例如，Box<dyn Debug + Send + UnwindSafe>。

语法

```
Type :
      TypeNoBounds
   | ImplTraitType
   | TraitObjectType

TypeNoBounds :
      ParenthesizedType
   | ImplTraitTypeOneBound
   | TraitObjectTypeOneBound
   | TypePath
   | TupleType
   | NeverType
   | RawPointerType
   | ReferenceType
   | ArrayType
   | SliceType
   | InferredType
   | QualifiedPathInType
   | BareFunctionType
   | MacroInvocation

ParenthesizedType :
   ( Type )
```

在某些情况下，类型的组合可能不明确。 在类型周围使用括号以避免歧义。 例如，引用类型中类型 bound 的 + 运算符不清楚 bound 适用于何处，因此需要使用括号。需要这种消歧的语法规则使用 TypeNoBounds 规则而不是 Type：

```
type T<'a> = &'a (dyn Any + Send);
```

语句

语句是块的组成部分，而块又是外部表达式或函数的组成部分。

Rust 有两种语句：声明语句和表达式语句。

- 声明语句是将一个或多个名称引入封闭语句块的语句。 声明的名称可能表示新变量或新 item。声明语句有：项声明语句和 let 语句。
- 表达式语句是对表达式求值并忽略其结果的语句。 通常，表达式语句的目的是触发对其表达式求值的效果。

语句接受外部属性。对语句有意义的属性是 cfg 和 lint 检查属性。

项声明语句

项声明语句的语法形式与模块内的项声明相同。在语句块中声明的项会将其范围限制为包含该项声明语句的块。项声明语句不会像模块声明中的项一样具有规范路径。例外的情况是实现的关联项仍然可以在外部访问。在其他方面项声明语句与在模块内声明项的含义相同。

注意：项声明语句没有对所在语句块包含的函数的泛型参数、参数和局部变量的进行隐式捕获，因此不能使用外部的内容。例子：

```
fn outer() {
  let outer_var = true;

  fn inner() { /* outer_var is not in scope here */ }

  inner();
}
```

fn inner() {}是一个项声明语句，其内部不能使用任何所在语句块的内容。

模块声明

模块是零个或多个 item 的容器。模块项是以大括号括起来的、命名的、并以关键字 mod 为前缀的模块。 一个模块项将一个新的命名模块引入构成 crate 的模块树中。 模块可以任意嵌套。

模块和类型共享相同的命名空间。禁止在作用域中声明与模块同名的命名类型：类型定义、trait、struct、enum、union、类型参数。否则 crate 不能隐藏作用域中模块的名称，反之亦然。通过 use 带入到模块作用域内的 item 也受此限制。

unsafe 关键字在语法上允许出现在 mod 关键字之前，但在语义级别被拒绝。这允许宏在将 unsafe 从令牌流中删除之前使用语法并使用 unsafe 关键字。

例子：

```
mod math {
    type Complex = (f64, f64);
    fn sin(f: f64) -> f64 {
        /* ... */
    }
    fn cos(f: f64) -> f64 {
        /* ... */
    }
    fn tan(f: f64) -> f64 {
        /* ... */
    }
}
```

模块文件路径

一个没有主体定义的模块(如 mod math;)是从外部加载的。当模块没有 path 属性时，模块路径即文件路径，例如模块路径为 crate::util::config，那么文件路径为 util/config.rs。父模块路径组件是目录，模块的内容在一个文件中，该文件具有模块名称加上.rs 扩展名。例如，以下模块结构可以具有相应的文件系统结构：

Module Path	Filesystem Path	File Contents
crate	lib.rs	mod util;
crate::util	util.rs	mod config;
crate::util::config	util/config.rs	

模块文件也可以是模块名称的目录中名为 mod.rs 的文件中的内容。上面的例子可以交替地用名为 util/mod.rs 的文件中的 crate::util 的内容来表达。 不允许同时拥有 util.rs 和 util/mod.rs。例如：

Module Path	Filesystem Path	File Contents
crate	lib.rs	mod util;
crate::util	util/mod.rs 或 util.rs	mod config;
crate::util::config	util/config/mod.rs 或 util/config.rs	

path 属性

path 属性可以影响用于加载外部文件模块的目录和文件。

对于不在内联模块 block 内的路径属性，文件路径相对于当前源文件所在的目录。例如对于如下内容：

```
#[path = "foo.rs"]
mod c;
```

其模块文件和模块路径对应如下：

Source File	c's File Location	c's Module Path
src/a/b.rs	src/a/foo.rs	crate::a::b::c
src/a/mod.rs	src/a/foo.rs	crate::a::c

对于内联模块 block 内的路径属性，文件路径的相对位置取决于路径属性所在的源文件类型。

- mod-rs 源文件类型是根模块（例如 lib.rs 或 main.rs） 和带有名为 mod.rs 的文件的模块。mod-rs 文件中内联模块 block 内的路径属性的路径相对于 mod-rs 文件所在的目录，包括以内联模块组件名称的目录。
- non-mod-rs 源文件类型是所有其他模块文件。non-mod-rs 文件中内联模块 block 内的路径属性的路径相对于 no-mod-rs 文件所在的目录，包括 no-mod-rs 文件名称的目录，以及内联模块组件名称的目录。

例如对于如下内容：

```
mod inline {
    #[path = "other.rs"]
    mod inner;
}
```

其模块文件和模块路径对应如下：

Source File	inner's File Location	inner's Module Path
src/a/b.rs(non-mod-rs)	src/a/b/inline/other.rs	crate::a::b::inline::inner
src/a/mod.rs(mod-rs)	src/a/inline/other.rs	crate::a::inline::inner

以下是混合内联模块和嵌套模块的模块引入例子：

```
#[path = "thread_files"]
mod thread {
```

```
    // Load the `local_data` module from `thread_files/tls.rs` relative to
    // this source file's directory.
    #[path = "tls.rs"]
    mod local_data;
}
```

其模块文件和模块路径对应如下：

Source File	local_data's File Location	local_data's Module Path
src/a/b.rs	src/a/b/thread_files/tls.rs	crate::a::b::thread::local_data
src/a/mod.rs	src/a/thread_files/tls.rs	crate::a::thread::local_data

模块的属性

与所有 item 一样，模块接受外部属性。另外模块也接受内部属性，对于带有主体的模块，内部属性可以在{之后，或者在源文件的开头。

对模块有意义的内置属性有以下几个： cfg、deprecated、doc、lint check。

语法

```
Module :
      unsafe? mod IDENTIFIER ;
   | unsafe? mod IDENTIFIER {
        InnerAttribute*
        Item*
      }
```

外部 crate 声明

extern crate 声明指定了对外部 crate 的依赖。它将外部 crate 作为 extern crate 声明中提供的标识符绑定到声明范围中。as 子句可用于将导入的 crate 绑定到不同的名称。可以使用 extern crate self 导入 self crate，这会创建与当前 crate 的绑定，在这种情况下，必须使用 as 子句来指定要绑定到的名称。

如果 extern crate 出现在 crate 根中，那么该 crate 名称也会添加到 extern prelude 中，使其自动在所有模块的范围内可用。

在编译时，会将 extern crate 解析为特定的 soname，并且运行时的链接要求 soname 被传递给链接器用于运行时加载。soname 在编译时通过扫描编译器的库路径并与在 extern crate 上声明的 crateid 属性进行匹配来解析的。如果未提供 crateid 属性，则假定默认名称属性为 extern crate 声明中给出的标识符。

在命名 Rust crate 时，不允许使用连字符"-"。但是，cargo 包可能会使用它们。在这种情况下，当 Cargo.toml 没有指定 crate 名称时，Cargo 将透明地将 - 替换为 _。

例子：

```
extern crate pcre;

extern crate std; // equivalent to: extern crate std as std;

extern crate std as ruststd; // linking to 'std' under another name

// Importing the Cargo package hello-world
extern crate hello_world; // hyphen replaced with an underscore
```

_导入

可以通过使用形式为 extern crate foo as _ 的方式来声明外部 crate 依赖项，这样将不会将 crate 绑定到名称。这对于只需要链接但不会被引用的 crate 可能很有用，并且将避免被报告为未使用。

macro_use 属性仍然照常工作，并将 extern crate 中导出的宏导入到 macro_use prelude 中。

外部 crate 声明的属性

可以在 extern crate 项上指定 macro_use 属性，用于导入 extern crate 中导出的宏。

no_link 属性

可以在 extern crate 项上指定 no_link 属性，以防止将 crate 链接到 output 中。这通常用于加载 crate 以仅访问其导出的宏。

语法

```
ExternCrate :
   extern crate CrateRef AsClause? ;

CrateRef :
   IDENTIFIER | self

AsClause :
   as ( IDENTIFIER | _ )
```

Use 声明

use 声明创建一个或多个与指定路径具有相同含义的本地名称绑定。通常 use 声明是来缩短引用模块项所需的路径。这些声明可能出现在模块和块中，通常位于顶部。

use 声明支持许多方便的快捷方式：

- 如果同时绑定具有公共前缀的路径列表，可以使用类似 glob 的大括号语法 use a::b::{c, d, e::f, g::h::i};
- 如果同时绑定具有公共前缀的路径列表及其公共父模块，使用 self 关键字，例如 use a::b::{self, c, d::e};
- 将目标名称重新绑定为新的本地名称，使用 use p::q::r as x;。也可以与上面两个功能一起使用：use a::b::{self as ab, c as abc}。
- 绑定所有匹配给定前缀的路径，使用星号通配符语法使用 a::b::*;。
- 混合使用上面的功能，如使用 a::b::{self as ab, c, d::{*, e::f}};。

例子：

```
use std::option::Option::{Some, None};
use std::collections::hash_map::{self, HashMap};

fn foo<T>(_: T){}
fn bar(map1: HashMap<String, usize>, map2: hash_map::HashMap<String, usize>){}

fn main() {
    // Equivalent to 'foo(vec![std::option::Option::Some(1.0f64),
    // std::option::Option::None]);'
    foo(vec![Some(1.0f64), None]);

    // Both `hash_map` and `HashMap` are in scope.
    let map1 = HashMap::new();
    let map2 = hash_map::HashMap::new();
    bar(map1, map2);
}
```

use 的可见性

与 item 一样，默认情况下，use 声明对包含的模块是私有的。与 item 一样，如果使用 pub 关键字限定，则 use 声明可以是公共的，这样的 use 声明用于重新导出名称。

pub use 声明可以将某些公共名称重定向到不同的目标定义：甚至是在不同模块内具有私有规范路径的定义。

如果一系列此类重定向形成一个循环或者是无法明确解析，则它们将导致编译时错误。

例子：

```
mod quux {
    pub use self::foo::{bar, baz};
    pub mod foo {
        pub fn bar() {}
        pub fn baz() {}
    }
}

fn main() {
    quux::bar();
    quux::baz();
}
```

use 路径规范

当使用 use 声明时，尽量使用以下四种方式的路径：

- std::path：这种格式以 extern crate 开头，是绝对路径，其中 std 为 extern crate。
- crate::foo::baz：这种格式以 root crate 开头，是绝对路径，其中 crate 为该 crate 的根模块。
- self::baz::foobaz：这种格式以 self 开头，是相对路径，其中 self 为当前模块路径。
- super::baz::foobaz：这种格式以 super 开头，是相对路径，其中 super 为当前模块的父模块路径。

另外 2018 版本另外支持以下两种路径：

- foo::example::iter：这种格式是相对路径，相对于当前模块路径。
- ::foo::baz::foobaz：这种格式是绝对路径，其中 foo 为 extern crate。

出现的原因在于如果不是以 crate、self、super 起始时，如果 extern crate 与当前模块中的 item 存在同名冲突，那么就不知道类似 foo::example 的路径指的是什么，因此通过前缀::来显示指定是 extern crate。

例子：

```
use std::path::{self, Path, PathBuf};  // good: std is a crate name
use crate::foo::baz::foobaz;    // good: foo is at the root of the crate

mod foo {

    pub mod example {
        pub mod iter {}
    }

    use crate::foo::example::iter; // good: foo is at crate root
//  use example::iter;      // bad in 2015 edition: relative paths are not allowed without `self`; good in 2018 edition
    use self::baz::foobaz;  // good: self refers to module 'foo'
    use crate::foo::bar::foobar;   // good: foo is at crate root

    pub mod bar {
        pub fn foobar() { }
    }

    pub mod baz {
        use super::bar::foobar; // good: super refers to module 'foo'
        pub fn foobaz() { }
    }
}

fn main() {}
```

_导入

通过 use path as _的下划线导入方式，可以仅导入 item 而不绑定到名称。这对于导入 trait 特别有用，以便可以在不导入 trait 符号的情况下使用其特定的实现。

例子：

```
mod foo {
    pub trait Zoo {
        fn zoo(&self) {}
    }

    impl<T> Zoo for T {}
}

use self::foo::Zoo as _;
struct Zoo;  // Underscore import avoids name conflict with this item.

fn main() {
    let z = Zoo;
    z.zoo();
}
```

语法

```
Syntax:
```

```
UseDeclaration :
   use UseTree ;

UseTree :
      (SimplePath? ::)? *
   | (SimplePath? ::)? { (UseTree ( , UseTree )* ,?)? }
   | SimplePath ( as ( IDENTIFIER | _ ) )?
```

extern 声明

extern block 提供了未在当前 crate 中定义的 item 的声明，并且是 Rust 外部函数接口的基础。这些类似于未经检查的 use 导入。

extern block 中允许有两种 item 声明：函数项声明和静态变量声明。只有在不安全的上下文中才允许调用或访问在外部块中声明的函数或静态变量。

unsafe 关键字在语法上允许出现在 extern 关键字之前，但在语义级别被拒绝。这允许宏在将其从令牌流中删除之前，消费语法并使用 unsafe 关键字。

函数项声明

extern block 中的函数项与其他 Rust 函数相同的方式声明，不同之处在于它们不能有主体，而是以分号终止。不允许使用函数限定符(const、async、unsafe 和 extern)。函数参数中不允许使用 pattern，只能使用 IDENTIFIER 或 _。

extern block 中的函数可以被 Rust 代码调用，就像在 Rust 中定义的函数一样。 Rust 编译器会自动在 Rust ABI 和外部 ABI 之间进行转换。

extern block 中声明的函数是隐式不安全的。当强制转换为函数指针时，在 extern block 中声明的函数具有不安全的 extern "abi" for<'l1, ..., 'lm> fn(A1, ..., An) -> R 类型，其中 'l1 , ... 'lm 是它的生命周期参数，A1, ..., An 是其参数的声明类型，R 是声明的返回类型。

静态变量声明

extern block 中的静态变量的声明方式与 extern block 之外静态变量的声明方式相同，只是它们没有初始化其值的表达式。访问在 extern block 中声明的静态变量项是不安全的，无论它是否可变，因为无法保证静态内存中的位模式对其声明的类型有效，因为某些独断（例如 C）代码负责初始化静态。

extern block 中的静态变量可以是不可变的或可变的，就像 extern block 之外的静态变量一样。仅在 Rust 代码读取静态变量之前初始化静态变量是不够的，应该在执行任何 Rust 代码之前，必须初始化一个不可变的静态变量。

ABI

ABI 意为应用二进制接口，即 application binary interface。默认情况下，extern block 假定它们调用的库使用特定平台上的标准 C ABI。可以使用 abi 字符串指定其他 ABI，如下所示：

```
// Interface to the Windows API
extern "stdcall" { }
```

有三个跨平台的 ABI 字符串，所有编译器都保证支持：

- extern "Rust"：这是在 Rust 代码中编写普通 fn foo() 时的默认 ABI。
- extern "C"：这与 extern fn foo(); 相同(见 extern 函数限定符)。 无论您的 C 编译器支持什么默认设置。
- extern "system"：通常与 extern "C" 相同，但在 Win32 上它是 "stdcall"，或者您应该使用它来链接到 Windows API 本身。

还有一些特定于平台的 ABI 字符串：

- extern "cdecl"：x86_32 C 代码的默认值。
- extern "stdcall"：x86_32 上 Win32 API 的默认值。
- extern "win64"：x86_64 Windows 上 C 代码的默认值。
- extern "sysv64"：非 Windows x86_64 上 C 代码的默认值。
- extern "aapcs"：ARM 的默认设置。
- extern "fastcall"：fastcall ABI -- 对应于 MSVC 的 __fastcall 和 GCC 以及 clang 的 __attribute__((fastcall))。
- extern "vectorcall"：vectorcall ABI -- 对应于 MSVC 的 __vectorcall 和 clang 的 __attribute__((vectorcall))。

变参函数

通过将 ... 指定为最后一个参数，extern block 中的函数可以是可变参数的。在可变参数之前必须至少有一个参数。可变参数可以选择用标识符指定。

例子：

```
extern "C" {
    fn foo(x: i32, ...);
    fn with_name(format: *const u8, args: ...);
}
```

extern 声明的属性

以下属性控制 extern block 的行为。

link 属性

link 属性指定编译器应该为 extern block 中的项链接的本机库的名称。它使用 MetaListNameValueStr 语法来指定其输入。name 键是要链接的本机库的名称。 kind 键是一个可选值，它指定库的类型，具有以下可能值：

- dylib：表示动态库。如果未指定种类，则这是默认设置。
- static：表示静态库。
- framework：表示 macOS 框架。 这仅对 macOS 目标有效。

如果指定了 kind，则必须包含 name 键。

当从宿主环境导入符号时，wasm_import_module 键可用于为 extern blokc 中的项指定 WebAssembly 模块名称。 如果未指定 wasm_import_module，则默认模块名称为 env。

在空的 extern block 上添加 link 属性是有效的。 您可以使用它来满足代码中其他地方（包括上游 crate）的 extern block 的链接要求，而不是将属性添加到每个 extern 块。

例子：

```
#[link(name = "crypto")]
extern {
    // …
}

#[link(name = "CoreFoundation", kind = "framework")]
extern {
    // …
}

#[link(wasm_import_module = "foo")]
extern {
    // …
}
```

link_name 属性

可以在 extern block 内的声明项上指定 link_name 属性，以指示要为给定函数或静态变量导入的符号。它使用 MetaNameValueStr 语法来指定符号的名称。

例子：

```
extern {
    #[link_name = "actual_symbol_name"]
    fn name_in_rust();
}
```

语法

```
ExternBlock :
   unsafe? extern Abi? {
      InnerAttribute*
      ExternalItem*
   }

ExternalItem :
   OuterAttribute* (
         MacroInvocationSemi
      | ( Visibility? ( StaticItem | Function ) )
   )
```

Struct 声明

一个结构体是使用 struct 关键字定义的 struct 类型。

例子：

```
struct Point {x: i32, y: i32}
let p = Point {x: 10, y: 11};
let px: i32 = p.x;
```

元组结构

元组结构是名义元组类型，也用关键字 struct 定义。

例子：

```
struct Point(i32, i32); // 注意尾部的分号
let p = Point(10, 11);
let px: i32 = match p { Point(x, _) => x };
```

类单元结构

类单元结构是一个没有任何字段的结构体，它是通过完全不使用字段列表来定义的。这样的结构隐式地定义了一个同名类型的常量。

例子：

```
struct Cookie;
let c = [Cookie, Cookie {}, Cookie, Cookie {}];
```

等价于

```
struct Cookie {}
const Cookie: Cookie = Cookie {};
let c = [Cookie, Cookie {}, Cookie, Cookie {}];
```

语法

```
Struct :
      StructStruct
   | TupleStruct

StructStruct :
   struct IDENTIFIER  GenericParams? WhereClause? ( { StructFields? } | ; )

TupleStruct :
   struct IDENTIFIER  GenericParams? ( TupleFields? ) WhereClause? ;

StructFields :
   StructField (, StructField)* ,?

StructField :
   OuterAttribute*
   Visibility?
   IDENTIFIER : Type

TupleFields :
   TupleField (, TupleField)* ,?

TupleField :
   OuterAttribute*
   Visibility?
   Type
```

Enum 声明

enum 定义了一组虚拟的枚举类型以及他们对应的构造函数。enum 可用于创建枚举类型值，也可以用于匹配枚举类型值。

例子下面是一组 enum item：

```
enum Animal {
    Dog,
```

```
    Cat,
}

let mut a: Animal = Animal::Dog;
a = Animal::Cat;
```

Dog 和 Cat 是无字段枚举变体。

enum 构造函数可以拥有命名或未命名的字段，例如：

```
enum Animal {
    Dog(String, f64),
    Cat { name: String, weight: f64 },
}

let mut a: Animal = Animal::Dog("Cocoa".to_string(), 37.2);
a = Animal::Cat { name: "Spotty".to_string(), weight: 2.7 };
```

Cat 是一个类似结构体的枚举变体，而 Dog 是一个类似元组的枚举变体。

判别式

每个枚举实例都有一个判别式，它是一个与之关联的整数，用于确定它拥有哪个变体。可以使用 mem::discriminant 函数获得对该判别式的不透明引用。

如果枚举的任何变体都没有附加数据(无字段枚举变体)，则可以直接指定判别式和访问判别式：

- 枚举定义时可以选择指定每个判别式获得哪个整数，方法是在变体名称后面加上 = 和一个常量表达式。
- 可以通过使用 as 运算符将枚举变体转换为整数类型。

如果声明中的第一个变体未指定，则将其判别式设置为零。对于每个其他未指定的判别式，它被设置为比声明中的前一个变体值大 1。例子：

```
enum Foo {
    Bar,            // 0
    Baz = 123,      // 123
    Quux,           // 124
}

let baz_discriminant = Foo::Baz as u32;
assert_eq!(baz_discriminant, 123);
```

如果两个变体具有相同的判别式值时，会产生一个错误。例如：

```
enum SharedDiscriminantError {
    SharedA = 1, // 1
    SharedB = 1 // 1
}

enum SharedDiscriminantError2 {
    Zero,       // 0
    One,        // 1
    OneToo = 1  // 1 (collision with previous!)
}
```

当在 enum 上使用原始布局陈述时，枚举变体的判别式具有最大值，当一个未指定判断式的前一个判别式的值已经最大时，将出错。例如：

```
#[repr(u8)]
enum OverflowingDiscriminantError {
    Max = 255,
    MaxPlusOne // Would be 256, but that overflows the enum.
}

#[repr(u8)]
enum OverflowingDiscriminantError2 {
    MaxMinusOne = 254, // 254
    Max,               // 255
    MaxPlusOne         // Would be 256, but that overflows the enum.
}
```

在默认布局陈述下，指定的判别式被解释为 isize 值，尽管允许编译器在实际内存布局中使用较小的类型。大小和可接受的值可以通过使用原始布局陈述或 C 布局陈述进行更改。

无变体枚举

具有零变体的枚举称为无变体枚举。 由于它们没有有效值，因此无法实例化。例如：

```
enum ZeroVariants {}
```

无变体枚举等价于 never 类型，但它们不能被强制转换为其他类型。例如：

```
let x: ZeroVariants = panic!();
let y: u32 = x; // mismatched type error
```

变体可见性

枚举变体在语法上允许 Visibility 注释，但在验证枚举时会被拒绝掉。 这允许使用统一的语法跨不同上下文来解析 item。

例子：

```
macro_rules! mac_variant {
    ($vis:vis $name:ident) => {
        enum $name {
            $vis Unit,

            $vis Tuple(u8, u16),

            $vis Struct { f: u8 },
        }
    }
}

// Empty `vis` is allowed.
mac_variant! { E }

// This is allowed, since it is removed before being validated.
#[cfg(FALSE)]
enum E {
    pub U,
    pub(crate) T(u8),
    pub(super) T { f: String }
}
```

语法

```
Enumeration :
   enum IDENTIFIER  GenericParams? WhereClause? { EnumItems? }

EnumItems :
   EnumItem ( , EnumItem )* ,?

EnumItem :
   OuterAttribute* Visibility?
   IDENTIFIER ( EnumItemTuple | EnumItemStruct | EnumItemDiscriminant )?

EnumItemTuple :
   ( TupleFields? )

EnumItemStruct :
   { StructFields? }

EnumItemDiscriminant :
   = Expression
```

Unoin 声明

union 声明使用与 struct 声明相同的语法，除了用 union 关键字代替 struct 关键字。

例子：

```
#[repr(C)]
union MyUnion {
    f1: u32,
```

```
    f2: f32,
}
```

union 的特点在于 union 的所有字段共享公共存储。因此，写入 union 的一个字段可以覆盖其其他字段，并且 union 的大小由其最大字段的大小决定。

初始化

union 类型的值可以使用与 struct 类型相同的语法创建，除了它必须指定一个字段：

```
let u = MyUnion { f1: 1 };
```

字段访问

每个 union 字段访问只是访问用于解释字段类型的存储。 读取 union 字段会读取该字段类型的对应的位联合。 字段可能具有非零偏移量（使用 C 表示时除外）； 在这种情况下，读取从字段偏移量开始的位。

可以使用与 struct 字段相同的语法访问 union 中的字段，并且对字段的访问需要放在 unsafe 块中。例如：

```
let f = unsafe { u.f1 };
```

写入实现了 Copy trait 或 ManuallyDrop 的 union 字段不需要为了读而运行析构函数，因此在写入这些字段时，不需要放置在 unsafe 块中。例如：

```
union MyUnion { f1: u32, f2: ManuallyDrop<String> }
let mut u = MyUnion { f1: 1 };

// These do not require `unsafe`.
u.f1 = 2;
u.f2 = ManuallyDrop::new(String::from("example"));
```

Union Drop

当 union 需要被 drop 时，它并不知道哪些字段需要被 drop。因此，所有联合字段必须实现 Copy trait 或 是 ManuallyDrop<_>类型，这确保了 union 在离开作用域时不需要删除任何内容。

就像结构体和枚举一样，可以为 union 实现 Drop trait 来手动定义当它被 drop 时需要做什么。

pattern match

另一种访问 union 字段的方法是使用模式匹配。union 字段的模式匹配使用了与 struct 模式相同的语法，除了模式必须指定一个字段。 由于模式匹配就像读取 union 的特定字段，因此它也必须放在 unsafe 块中。

例子：

```
fn f(u: MyUnion) {
    unsafe {
        match u {
            MyUnion { f1: 10 } => { println!("ten"); }
            MyUnion { f2 } => { println!("{}", f2); }
        }
    }
}
```

模式匹配可以将 union 作为 struct 的字段进行匹配。例如：

```
#[repr(u32)]
enum Tag { I, F }

#[repr(C)]
union U {
    i: i32,
    f: f32,
}

#[repr(C)]
struct Value {
    tag: Tag,
    u: U,
}
```

```
fn is_zero(v: Value) -> bool {
    unsafe {
        match v {
            Value { tag: Tag::I, u: U { i: 0 } } => true,
            Value { tag: Tag::F, u: U { f: num } } if num == 0.0 => true,
            _ => false,
        }
    }
}
```

引用 union 字段

由于 union 字段共享公共存储，因此获得对 union 的一个字段的写访问权限同样也授予了对其所有剩余字段的写访问权限。因此在引用时，需要注意可变冲突。例子：

```
// ERROR: cannot borrow `u` (via `u.f2`) as mutable more than once at a time
fn test() {
    let mut u = MyUnion { f1: 1 };
    unsafe {
        let b1 = &mut u.f1;
//                   ---- first mutable borrow occurs here (via `u.f1`)
        let b2 = &mut u.f2;
//                   ^^^^ second mutable borrow occurs here (via `u.f2`)
        *b1 = 5;
    }
//  - first borrow ends here
    assert_eq!(unsafe { u.f1 }, 5);
}
```

语法

```
Union :
   union IDENTIFIER GenericParams? WhereClause? {StructFields }
```

Function 声明

一个函数由一个块、一个名称和一组参数组成。除了名称之外，所有这些都是可选的。 函数是用关键字 fn 声明的。 函数可以将一组输入变量声明为参数，调用者通过这些变量将参数传递给函数，以及函数在完成时将输出类型的值返回给调用者。

当被引用时，一个函数会产生一个对应大小为零的函数项类型的一类值，当被调用时，它被评估为对该函数的直接调用。

没有函数主体的函数以分号结束，这种形式只能出现在 trait 或 extern 块中。

例子：

```
fn answer_to_life_the_universe_and_everything() -> i32 {
    return 42;
}
```

函数参数

与 let 绑定一样，函数参数也是 irrefutable pattern，因此任何在 let 绑定中有效的 pattern 也可以作为有效参数，例如：

```
fn first((value, _): (i32, i32)) -> i32 { value }
```

如果第一个参数是 Self 参数，则表明该函数是一个方法。带有 self 参数的函数只能在 trait 或 impl 中作为关联函数出现。

带有 ... 的参数表示可变参数函数，并且只能用作 extern 块中的函数的最后一个参数。可变参数可以有一个可选的标识符，例如 args: ...。因此 rust 本身定义的函数是不接受可变参数的。

函数体

函数块在概念上是被包装在一个块中的，该块绑定参数的 patterns，然后返回函数块的值。因此如果块的尾部表达式被求值，那么其结果最终会返回给调用者。也可以在函数体内使用显式的 return 表达式进行提前返回。

例如以下函数的函数体：

```
fn first((value, _): (i32, i32)) -> i32 { value }
```

就像如下行为一样：

```
// argument_0 is the actual first argument passed from the caller
let (value, _) = argument_0;
return {
    value
};
```

泛型函数

泛型函数允许一个或多个参数化类型出现在签名中。每个类型参数都必须在函数名称后面的尖括号中显式声明，并且用逗号进行分隔。例如：

```
// foo is generic over A and B

fn foo<A, B>(x: A, y: B) {
```

在函数签名和主体内部，类型参数的名称可以用作类型名称。可以为类型参数指定 trait bound，以允许在函数体内对该类型的值调用具有该 trait 的方法。下面使用 where 语法指定的 trait bound：

```
fn foo<T>(x: T) where T: Debug {
```

当一个泛型函数被引用时，它的类型参数根据引用的上下文被实例化。例如，在这里调用 foo 函数：

```
use std::fmt::Debug;

fn foo<T>(x: &[T]) where T: Debug {
    // details elided
}

foo(&[1, 2]);
```

这里将使用 i32 实例化类型参数 T。

类型参数也可以在函数名称后的尾随路径中显式的来进行实例化。在没有足够的上下文来确定类型参数的情况下，这是必要的。例如：

```
// fn mem::size_of<T>() -> usize

mem::size_of::<u32>() == 4。
```

这里将显式的使用 u32 实例化类型参数 T。

extern 函数限定符

extern 函数限定符定义了可以使用特定 ABI 进行调用的函数，例如：

```
extern "ABI" fn foo() { /* ... */ }
```

这些通常与提供函数声明的 extern block item 结合使用，这些声明将允许调用外部函数而无需定义，例如：

```
extern "ABI" {
  fn foo(); /* no body */
}
unsafe { foo() }
```

当函数项中的限定符省略“extern”Abi?* 时，将会分配 ABI 为“Rust”。 例如：

```
fn foo() {}
//等价于如下代码
extern "Rust" fn foo() {}
```

当使用 extern 关键字并且省略“ABI”时，使用的 ABI 默认为“C”。例如：

```
extern fn new_i32() -> i32 { 0 }
let fptr: extern fn() -> i32 = new_i32;
//等价于如下代码
extern "C" fn new_i32() -> i32 { 0 }
let fptr: extern "C" fn() -> i32 = new_i32;
```

Rust 中的函数可以被外部代码调用，例如，使用不同于 Rust 的 ABI 允许提供可以从其他编程语言（如 C）调用的函数：

```
// Declares a function with the "C" ABI
extern "C" fn new_i32() -> i32 { 0 }

// Declares a function with the "stdcall" ABI
extern "stdcall" fn new_i32_stdcall() -> i32 { 0 }
```

具有不同于“Rust”的 ABI 的函数不支持以与 Rust 完全相同的方式展开栈。因此，在具有此类 ABI 的函数结束后展开会导致进程中止。

const 函数

用 const 关键字限定的函数是 const 函数，元组结构的构造函数和元组变体的构造函数也是 const 函数。

可以在 const 上下文中调用 const 函数。

const 函数不允许是 async 的，并且不能使用 extern 函数限定符。

async 函数

函数可以被限定为 async，这也可以与 unsafe 限定符结合使用，例如：

```
async fn regular_example() { }
async unsafe fn unsafe_example() { }
```

async 函数在被调用时并不会执行函数体，它们只是将参数捕获到 future 中。当被轮询时，该 future 将执行该函数的主体。

async 函数大致相当于一个返回 impl Future 的函数，该函数以 async move block 作为其主体。例如：

```
// Source
async fn example(x: &str) -> usize {
    x.len()
}
```

等价于：

```
// Desugared
fn example<'a>(x: &'a str) -> impl Future<Output = usize> + 'a {
    async move { x.len() }
}
```

async unsafe 函数

声明一个既 async 又 unsafe 的函数是合法的。对 async unsafe 函数的调用是不安全的，同样也会返回一个 future，这个 future 只是一个普通的 future，不需要不安全的上下文来“await”它。例如：

```
// Returns a future that, when awaited, dereferences `x`.
//
// Soundness condition: `x` must be safe to dereference until
// the resulting future is complete.
async unsafe fn unsafe_example(x: *const i32) -> i32 {
  *x
}

async fn safe_example() {
    // An `unsafe` block is required to invoke the function initially:
    let p = 22;
    let future = unsafe { unsafe_example(&p) };

    // But no `unsafe` block required here. This will
    // read the value of `p`:
    let q = future.await;
}
```

Unsafe 用于 async 函数的方式与用于其他函数的方式完全相同：它表明该函数对其调用者施加了一些额外的义务以确保健全性。 与任何其他不安全函数一样，这些条件可能超出初始调用本身——例如，在上面的代码片段中， unsafe_example 函数将指针 x 作为参数，然后（在等待时）解引用该指针，这意味着 x 在 future 完成执行之前必须是有效的，并且调用者有责任确保这一点。

function 声明的属性

在函数上允许使用外部属性。 在其块内的 { 之后允许立即使用内部属性。例如：

```
fn documented() {
    #![doc = "Example"]
}
```

对函数有意义的属性是 cfg、cfg_attr、deprecated、doc、export_name、link_section、no_mangle、lint 检查属性、must_use、过程宏属性、测试属性和优化提示属性。 函数也接受属性宏。

function 声明参数的属性

函数参数仅允许使用外部属性，下面是一些允许使用的内置属性：cfg、cfg_attr、allow、warn、deny 和 forbid。例如：

```
fn len(
    #[cfg(windows)] slice: &[u16],
    #[cfg(not(windows))] slice: &[u8],
) -> usize {
    slice.len()
}
```

也允许应用于项目的程序宏属性使用的惰性辅助属性，但要注意不要在最终的 TokenStream 中包含这些惰性属性。例如，以下代码定义了一个没有在任何地方正式定义的惰性 some_inert_attribute 属性，并且 some_proc_macro_attribute 过程宏负责检测它的存在并将其从输出令牌流中删除：

```
#[some_proc_macro_attribute]
fn foo_oof(#[some_inert_attribute] arg: u8) {
}
```

语法

```
Function :
   FunctionQualifiers fn IDENTIFIER GenericParams?
      ( FunctionParameters? )
      FunctionReturnType? WhereClause?
      ( BlockExpression | ; )

FunctionQualifiers :
   const? async1? unsafe? (extern Abi?)?

Abi :
   STRING_LITERAL | RAW_STRING_LITERAL

FunctionParameters :
      SelfParam ,?
   | (SelfParam ,)? FunctionParam (, FunctionParam)* ,?

SelfParam :
   OuterAttribute* ( ShorthandSelf | TypedSelf )

ShorthandSelf :
   (& | & Lifetime)? mut? self

TypedSelf :
   mut? self : Type

FunctionParam :
   OuterAttribute* ( FunctionParamPattern | ... | Type 2 )

FunctionParamPattern :
   PatternNoTopAlt : ( Type | ... )

FunctionReturnType :
   -> Type
```

Trait 声明

trait 描述了类型实现的抽象接口。所有 trait 都定义了一个隐式类型参数 Self，它指的是“实现这个接口的类型”。trait 也可能包含额外的类型参数，这些类型参数，包括 Self，可能会像往常一样受到其他 trait 等的约束。

trait 由以下三种关联 item 组成，可以通过单独的 trait impl 声明为特定类型实现 trait：

- types：关联类型不能定义类型，类型只能在类型的 trait impl 声明中指定。
- functions：关联函数可以通过用分号代替函数体来省略函数体，这表明必须在类型的 trait impl 声明中实现该函数。如果函数已经定义了一个函数体，这个定义将作为任何不覆盖它的实现的默认值。
- constants：关联常量可以省略等号和表达式，以表明必须在类型的 trait impl 声明中定义常量值。如果常量定义了常量值，这个定义将作为任何不覆盖该的实现的默认值。

注意：trait 中关联函数不允许是 async 或 const。

例子：

```
// Examples of associated trait items with and without definitions.
```

```
trait Example {
    const CONST_NO_DEFAULT: i32;
    const CONST_WITH_DEFAULT: i32 = 99;
    type TypeNoDefault;
    fn method_without_default(&self);
    fn method_with_default(&self) {}
}
```

关联项

关联项是在 trait 声明或在 implementation 声明中定义的项。关联项有如下几个，有关联函数（包括方法）、关联类型和关联常量。

关联项的种类都有以下两种：包含实际实现的定义以及定义的签名声明。

关联函数或方法

关联函数是与类型关联的函数。关联函数的两种种类：

- "关联函数定义"定义了与另一种类型关联的函数。它的写法与函数项相同。
- "关联函数声明"声明了"关联函数定义"的签名。它被写成一个函数项，只是函数体被一个;代替。

在 trait 声明中"关联函数定义"提供了关联函数的默认实现。

标识符是函数的名称。关联函数的泛型、参数列表、返回类型和 where 子句必须与"关联函数声明"相同。

例子：

```
struct Struct {
    field: i32
}

impl Struct {
    fn new() -> Struct {
        Struct {
            field: 0i32
        }
    }
}

fn main () {
    let _struct = Struct::new();
}
```

trait 声明的关联函数可以使用路径调用该函数，该路径是 trait 的名称添加 trait 的路径。实际等效于<_ as Trait>::function_name：

```
trait Num {
    fn from_i32(n: i32) -> Self;
}

impl Num for f64 {
    fn from_i32(n: i32) -> f64 { n as f64 }
}

// These 4 are all equivalent in this case.
let _: f64 = Num::from_i32(42);
let _: f64 = <_ as Num>::from_i32(42);
let _: f64 = <f64 as Num>::from_i32(42);
let _: f64 = f64::from_i32(42);
```

关联函数参数

对于存在函数主体的关联函数参数类似普通函数参数，没有主体的关联函数声明只允许 IDENTIFIER 或 _ 通配符模式。mut IDENTIFIER 目前是允许的，但它已被弃用，将来会成为一个硬错误。

例子，不允许使用元组模式当不存在函数体的时候：

```
trait T {
    fn f1((a, b): (i32, i32)) {}
    fn f2(_: (i32, i32));  // Cannot use tuple pattern without a body.
}
```

关联方法

第一个参数名为 self 的关联函数称为方法，可以使用方法调用运算符（例如 x.foo()）以及通常的函数调用符号来调用。

如果指定了 self 参数的类型，则它仅限于解析为由以下语法生成的类型（其中 'lt 表示某个任意生命周期）：

```
P = &'lt S | &'lt mut S | Box<S> | Rc<S> | Arc<S> | Pin<P>
S = Self | P
```

语法中 Self 表示为实现的类型，这还可以包括上下文的类型别名 Self、其他类型别名或解析为实现类型的关联类型投影。例如：

```
// Examples of methods implemented on struct `Example`.
struct Example;
type Alias = Example;
trait Trait { type Output; }
impl Trait for Example { type Output = Example; }
impl Example {
    fn by_value(self: Self) {}
    fn by_ref(self: &Self) {}
    fn by_ref_mut(self: &mut Self) {}
    fn by_box(self: Box<Self>) {}
    fn by_rc(self: Rc<Self>) {}
    fn by_arc(self: Arc<Self>) {}
    fn by_pin(self: Pin<&Self>) {}
    fn explicit_type(self: Arc<Example>) {}
    fn with_lifetime<'a>(self: &'a Self) {}
    fn nested<'a>(self: &mut &'a Arc<Rc<Box<Alias>>>) {}
    fn via_projection(self: <Example as Trait>::Output) {}
}
```

第一个参数可以在不指定类型的情况下使用速记语法，其具有以下等效项：

Shorthand	Equivalent
self	self: Self
&'lifetime self	self: &'lifetime Self
&'lifetime mut self	self: &'lifetime mut Self

如果 self 参数以 mut 为前缀，它将成为可变变量，类似于使用 mut 标识符模式的常规参数。 例如：

```
trait Changer: Sized {
    fn change(mut self) {}
    fn modify(mut self: Box<Self>) {}
}
```

trait 上的方法例子：

```
trait Shape {
    fn draw(&self, surface: Surface);
    fn bounding_box(&self) -> BoundingBox;
}

struct Circle {
    // ...
}

impl Shape for Circle {
    // ...
}

let circle_shape = Circle::new();
let bounding_box = circle_shape.bounding_box();
```

关联类型

关联类型是与类型关联的类型别名。关联类型不能在 inherent impl 中定义，也不能在 trait 中给出默认实现。

- "关联类型定义"定义了另一个类型的类型别名。它被写成 type 关键字，然后是标识符，然后是 =，最后是类型。
- "关联类型声明"声明了"关联类型定义"的签名。 它被写成 type 关键字，然后是一个标识符，最后是一个可选的 trait bound 列表。

标识符是声明的类型别名的名称。可选的 trait bound 必须由类型别名的实现来满足。

关联类型不得包含泛型参数或 where 子句。

如果类型项具有来自 trait Trait 的关联类型 Assoc，则 <Item as Trait>::Assoc 是一种类型表示，它是关联类型定义中指定的类型的别名。 此外，如果 Item 是类型参数，则 Item::Assoc 可用于类型参数。

例子：

```
trait AssociatedType {
    // Associated type declaration
    type Assoc;
}

struct Struct;

struct OtherStruct;

impl AssociatedType for Struct {
    // Associated type definition
    type Assoc = OtherStruct;
}

impl OtherStruct {
    fn new() -> OtherStruct {
        OtherStruct
    }
}

fn main() {
    // Usage of the associated type to refer to OtherStruct as <Struct as AssociatedType>::Assoc
    let _other_struct: OtherStruct = <Struct as AssociatedType>::Assoc::new();
}
```

关联常量

关联常量是与类型关联的常量。

- "关联常量定义"定义了与类型关联的常量。它的写法与常量声明相同。
- "关联常量声明"声明了"关联常量定义"的签名。它被写成 const 关键字，然后是一个标识符，然后是:，然后是一个类型，最后是一个;。

在 trait 声明中"关联常量定义"提供了关联常量的默认值。

标识符是路径中使用的常量的名称。类型是定义必须实现的类型。

例子：

```
trait ConstantIdDefault {
    const ID: i32 = 1;
}

struct Struct;
struct OtherStruct;

impl ConstantIdDefault for Struct {}

impl ConstantIdDefault for OtherStruct {
    const ID: i32 = 5;
}

fn main() {
    assert_eq!(1, Struct::ID);
    assert_eq!(5, OtherStruct::ID);
}
```

语法

```
AssociatedItem :
   OuterAttribute* (
         MacroInvocationSemi
       | ( Visibility? ( TypeAlias | ConstantItem | Function ) )
   )
```

关联项可见性

trait 的关联项在语法上允许 Visibility 注释，但在验证时会被拒绝掉。 这允许使用统一的语法跨不同上下文来解析 item。

例子：

```
macro_rules! create_method {
    ($vis:vis $name:ident) => {
        $vis fn $name(&self) {}
    };
}

trait T1 {
    // Empty `vis` is allowed.
    create_method! { method_of_t1 }
}

struct S;

impl S {
    // Visibility is allowed here.
    create_method! { pub method_of_s }
}

impl T1 for S {}

fn main() {
    let s = S;
    s.method_of_t1();
    s.method_of_s();
}
```

Trait bounds

泛型项可以使用 trait 作为其类型参数的界限。见 Trait bounds。

泛型 Trait

可以为 trait 指定类型参数以使其更加通用。类型参数在 trait 名称之后，使用与泛型函数相同的语法。

例子：

```
trait Seq<T> {
    fn len(&self) -> u32;
    fn elt_at(&self, n: u32) -> T;
    fn iter<F>(&self, f: F) where F: Fn(T);
}
```

父 trait

父 trait 是需要类型为了实现特定 trait 的而必须实现的 trait。在任何受 trait 限制的泛型或 trait object，它们不仅可以访问该 trait 的关联项，同时也可以访问该 trait 的父 trait 的关联项。存在父 trait 的 trait 被该父 trait 称为子 trait。

一个 trait 不能称为自己的父 trait。

例子，其中 Shape 是 Circle 的父 trait，Circle 是 Shape 的子 trait：

```
trait Shape { fn area(&self) -> f64; }
trait Circle : Shape { fn radius(&self) -> f64; }
```

等价于：

```
trait Shape { fn area(&self) -> f64; }
trait Circle where Self: Shape { fn radius(&self) -> f64; }
```

在子 trait 的默认实现中，可以通过 self 调用父 trait 的关联函数。例如：

```
trait Circle where Self: Shape {
    fn radius(&self) -> f64 {
        // A = pi * r^2
        // so algebraically,
        // r = sqrt(A / pi)
```

```
        (self.area() /std::f64::consts::PI).sqrt()
    }
}
```

受子 trait 限制的泛型或 trait object，也可以访问该 trait 的父 trait 的关联项。例如：

```
fn print_area_and_radius<C: Circle>(c: C) {
    // Here we call the area method from the supertrait `Shape` of `Circle`.
    println!("Area: {}", c.area());
    println!("Radius: {}", c.radius());
}

let circle = Box::new(circle) as Box<dyn Circle>;
let nonsense = circle.radius() * circle.area();
```

trait object safe

object safe 特征可以是 trait object 的基本特征。如果 trait 具有以下特性，则它是对象安全的：

- 所有的父 trait 同样是 object safe。
- Sized 不能是父 trait。即 trait 不能存在 where Self: Sized bound。
- 不能包含任何关联常量。
- 所有的关联函数必须可以从一个 trait object 进行调用，或显式的不可以从一个 trait object 进行调用：
  - 显式的不可以从一个 trait object 进行调用有如下要求：
    - 关联函数存在 where Self: Sized bound。
  - 可以从一个 trait object 进行调用有如下要求：
    - 关联函数不存在 where Self: Sized bound
    - 关联函数不能存在类型参数（只允许生命周期参数），
    - 关联函数是一个方法，并且除了在 receiver 中使用 Self，其他地方都不使用，
    - 关联函数具有如下类型的 receiver：
      - &Self (i.e. &self)
      - &mut Self (i.e &mut self)
      - Box<Self>
      - Rc<Self>
      - Arc<Self>
      - Pin<P> where P is one of the types above

例子，对象安全的 trait：

```
// Examples of object safe methods.
trait TraitMethods {
    fn by_ref(self: &Self) {}
    fn by_ref_mut(self: &mut Self) {}
    fn by_box(self: Box<Self>) {}
    fn by_rc(self: Rc<Self>) {}
    fn by_arc(self: Arc<Self>) {}
    fn by_pin(self: Pin<&Self>) {}
    fn with_lifetime<'a>(self: &'a Self) {}
    fn nested_pin(self: Pin<Arc<Self>>) {}
}
```

例子，对象安全的 trait，但是无法从 trait object 调用：

```
trait NonDispatchable {
    // Non-methods cannot be dispatched.
    fn foo() where Self: Sized {}
    // Self type isn't known until runtime.
    fn returns(&self) -> Self where Self: Sized;
    // `other` may be a different concrete type of the receiver.
    fn param(&self, other: Self) where Self: Sized {}
    // Generics are not compatible with vtables.
    fn typed<T>(&self, x: T) where Self: Sized {}
}

struct S;
impl NonDispatchable for S {
    fn returns(&self) -> Self where Self: Sized { S }
}
let obj: Box<dyn NonDispatchable> = Box::new(S);
~~obj.returns()~~; // ERROR: cannot call with Self return
```

```
~~obj.param(S)~~;  // ERROR: cannot call with Self parameter
~~obj.typed(1)~~;  // ERROR: cannot call with generic type
```

例子，非对象安全的 trait，因为存在关联常量：

```
trait NotObjectSafe {
    const CONST: i32 = 1;  // ERROR: cannot have associated const

    fn foo() {}  // ERROR: associated function without Sized
    fn returns(&self) -> Self; // ERROR: Self in return type
    fn typed<T>(&self, x: T) {} // ERROR: has generic type parameters
    fn nested(self: Rc<Box<Self>>) {} // ERROR: nested receiver not yet supported
}

struct S;
impl NotObjectSafe for S {
    fn returns(&self) -> Self { S }
}
~~let obj: Box<dyn NotObjectSafe> = Box::new(S);~~ // ERROR
```

例子：非对象安全的 trait，因为 trait 存在 where Self: Sized bound：

```
trait TraitWithSize where Self: Sized {}

struct S;
impl TraitWithSize for S {}
let obj: Box<dyn TraitWithSize> = Box::new(S); // ERROR
```

不安全 trait

以 unsafe 关键字开头的 trait 项表示实现该 trait 可能是不安全的。使用正确实现的不安全特性是安全的。实现 unsafe trait 的 trait impl 项也必须以 unsafe 关键字开头。

sync 和 send 是不安全 trait 的例子。

语法

```
Trait :
   unsafe? trait IDENTIFIER  GenericParams? ( : TypeParamBounds? )? WhereClause? {
     InnerAttribute*
     AssociatedItem*
   }
```

Implementation 声明

Implementation 声明是将 items 与实现类型进行关联。Implementation 是用关键字 impl 定义的，并且包含属于正在实现类型实例的方法或类型的关联函数。

有以下两种类型的 Implementation：

- inherent impl
- trait impl

inherent impl

inherent impl 被定义为以 impl 关键字开始，然后紧跟泛型声明，然后是名义类型的路径以及可选的 where 子句，接下来为大括号，括号中为一组关联 items。名义类型称为实现的类型，关联项是该实现的类型的关联项。

inherent impl 将声明中包含的关联项与实现的类型相关联。inherent impl 可以包含关联的函数（包括方法）和关联的常量，但是不能包含关联的类型别名。

inherent impl 中的关联项的路径是实现的类型的路径，后跟关联项的标识符作为最终路径。路径类似如下：type_path as::item_id。

一个类型也可以有多个 inherent impl，但是必须在与原始类型定义相同的 crate 中进行定义。

例子：

```
pub mod color {
    pub struct Color(pub u8, pub u8, pub u8);
```

```
    impl Color {
        pub const WHITE: Color = Color(255, 255, 255);
    }
}

mod values {
    use super::color::Color;
    impl Color {
        pub fn red() -> Color {
            Color(255, 0, 0)
        }
    }
}

pub use self::color::Color;
fn main() {
    // Actual path to the implementing type and impl in the same module.
    color::Color::WHITE;

    // Impl blocks in different modules are still accessed through a path to the type.
    color::Color::red();

    // Re-exported paths to the implementing type also work.
    Color::red();

    // Does not work, because use in `values` is not pub.
    // values::Color::red();
}
```

trait impl

trait impl 的定义与 inherent impl 类似，不同之处在于可选的泛型类型声明后跟一个 trait，然后后跟关键字 for，再跟名义类型路径。trait 被称为实现的 trait。

trait impl 必须定义由实现的 trait 声明的所有非默认关联项，也可以重新定义由实现的 trait 定义的默认关联项，但是不能定义任何非实现的 trait 声明中的项。关联项见这里。

trait impl 中的关联项的路径类似如下：<type_path as trait_path>::item_id。

unsafe trait 要求 trait impl 以 unsafe 关键字开头。

例子：

```
struct Circle {
    radius: f64,
    center: Point,
}

impl Copy for Circle {}

impl Clone for Circle {
    fn clone(&self) -> Circle { *self }
}

impl Shape for Circle {
    fn draw(&self, s: Surface) { do_draw_circle(s, *self); }
    fn bounding_box(&self) -> BoundingBox {
        let r = self.radius;
        BoundingBox {
            x: self.center.x - r,
            y: self.center.y - r,
            width: 2.0 * r,
            height: 2.0 * r,
        }
    }
}
```

trait impl 一致性

如果 trait impl 不满足孤儿规则，或者出现 trait impl 实现重叠，那么编译将会失败。

孤儿规则：给定 impl<P1..=Pn> Trait<T1..=Tn> for T0，仅当以下至少一项为真时，impl 才有效：

- trait 与 trait impl 定义在同一个 crate 中。
- 满足以下两个条件：
  - T0..=Tn 类型必须有一个是本地类型。 假设 Ti 成为第一个这样的类型。
  - 对于 T0..Ti(不包括 Ti)来说，P1..=Pn 是非裸露类型参数。

名词含义：

- 裸露类型：不作为其他类型的类型参数的类型。例如 T 是裸露类型，在 Vec<T>中的 T 不是裸露类型。
- 本地类型：类型定义在当前 crate 中，表示该类型为本地类型。

请注意，出于一致性的目的，基本类型是特殊的，并且 Box<T> 中的 T 为裸露类型， Box<LocalType> 被视为本地类型。

泛型 trait impl

trait impl 可以包含泛型参数，这些参数可以在实现的其余部分中使用。trait impl 的泛型参数直接写在 impl 关键字之后。例如：

```
impl<T> Seq<T> for Vec<T> {
    /* ... */
}
impl Seq<bool> for u32 {
    /* Treat the integer as a sequence of bits */
}
```

如果泛型参数具有如下描述情况之一，则泛型参数对实现具有约束：

- 实现的 trait 中具有类型参数之一。
- 实现的类型中具有类型参数之一。
- 作为包含另一个限制实现的参数的类型边界中的关联类型。

如果生命周期参数用于关联类型，则生命周期必须约束实现。

例子，具有约束的情况：

```
// T constrains by being an argument to GenericTrait.
impl<T> GenericTrait<T> for i32 { /* ... */ }

// T constrains by being an arguement to GenericStruct
impl<T> Trait for GenericStruct<T> { /* ... */ }

// Likewise, N constrains by being an argument to ConstGenericStruct
impl<const N: usize> Trait for ConstGenericStruct<N> { /* ... */ }

// T constrains by being in an associated type in a bound for type `U` which is
// itself a generic parameter constraining the trait.
impl<T, U> GenericTrait<U> for u32 where U: HasAssocType<Ty = T> { /* ... */ }

// Like previous, except the type is `(U, isize)`. `U` appears inside the type
// that includes `T`, and is not the type itself.
impl<T, U> GenericStruct<U> where (U, isize): HasAssocType<Ty = T> { /* ... */ }
```

例子，不具有约束的情况：

```
// The rest of these are errors, since they have type or const parameters that
// do not constrain.

// T does not constrain since it does not appear at all.
impl<T> Struct { /* ... */ }

// N does not constrain for the same reason.
impl<const N: usize> Struct { /* ... */ }

// Usage of T inside the implementation does not constrain the impl.
impl<T> Struct {
    fn uses_t(t: &T) { /* ... */ }
}

// T is used as an associated type in the bounds for U, but U does not constrain.
impl<T, U> Struct where U: HasAssocType<Ty = T> { /* ... */ }
```

```
// T is used in the bounds, but not as an associated type, so it does not constrain.
impl<T, U> GenericTrait<U> for u32 where U: GenericTrait<T> {}
```

例子，允许的无约束的生命周期参数：

```
impl<'a> Struct {}
```

例子，不允许的无约束的生命周期参数：

```
impl<'a> HasAssocType for Struct {
    type Ty = &'a Struct;
}
```

implementation 声明的属性

可以在 impl 关键字之前包含外部属性，在具有关联项的大括号内可以包含内部属性。 内部属性必须在任何关联项目之前。 此处有意义的属性是 cfg, deprecated, doc 和 lint 检查属性。

语法

```
Implementation :
   InherentImpl | TraitImpl

InherentImpl :
   impl GenericParams? Type WhereClause? {
      InnerAttribute*
      AssociatedItem*
   }

TraitImpl :
   unsafe? impl GenericParams? !? TypePath for Type
   WhereClause?
   {
      InnerAttribute*
      AssociatedItem*
   }
```

type 声明

type 别名为现有类型定义新名称。每个值都有一个特定的类型，但可能实现几个不同的 trait，或者与几个不同的类型约束兼容。

例如，下面将类型 Point 定义为类型 (u8, u8) 的同义词：

```
type Point = (u8, u8);
let p: Point = (41, 68);
```

注意：元组结构或元组变体的类型别名不能用于限定该类型的构造函数，例如：

```
struct MyStruct(u32);

use MyStruct as UseAlias;
type TypeAlias = MyStruct;

let _ = UseAlias(5); // OK
~~let _ = TypeAlias(5);~~ // Doesn't work
let _ : TypeAlias = MyStruct(5); // work
```

这是因为元组结构或单元结构的类型不仅仅是类型，也是函数，但是 type 定义的仅仅是类型而不能作为函数。

对于没有类型规范(=type 被省略)的类型别名只能在 trait 中作为关联类型出现。

语法

```
TypeAlias :
   type IDENTIFIER GenericParams? WhereClause? ( = Type )? ;
```

常量声明

常量项是一个可选命名的常量值，它与程序中的特定内存位置无关。常量本质上是内联的，无论在哪里使用它们，这意味着它们在使用时被

直接复制到相关的上下文中。这包括了使用来自外部 crate 和非复制类型的常量。

常量必须显式的注解类型。该类型必须具有“static 生命周期”：初始化程序中的任何引用都必须具有“static 生命周期”。

常量可能引用其他常量的地址，在这种情况下，地址将在适用的情况下省略生命周期，否则在大多数情况下默认为静态生命周期。然而，编译器仍然可以自由地多次翻译常量，因此所引用的地址可能不稳定，因此对相同常量的引用不一定保证引用相同的内存地址。

例子：

```
const BIT1: u32 = 1 << 0;
const BIT2: u32 = 1 << 1;

const BITS: [u32; 2] = [BIT1, BIT2];
const STRING: &'static str = "bitstring";

struct BitsNStrings<'a> {
    mybits: [u32; 2],
    mystring: &'a str,
}

const BITS_N_STRINGS: BitsNStrings<'static> = BitsNStrings {
    mybits: BITS,
    mystring: STRING,
};
```

注意：常量的表达式(=Expression 部分)只能在 trait 定义的关联常量中省略。

未命名常量

与关联常量不同，自由常量可以通过使用下划线而不是名称来取消命名。 例如：

```
const _: () =  { struct _SameNameTwice; };

// OK although it is the same name as above:
const _: () =  { struct _SameNameTwice; };
```

宏可以安全地在同一范围内多次生成相同的未命名常量，而不产生错误。 例如：

```
macro_rules! m {
    ($item: item) => { $item $item }
}

m!(const _: () = (););
// This expands to:
// const _: () = ();
// const _: () = ();
```

语法

```
ConstantItem :
   const ( IDENTIFIER | _ ) : Type ( = Expression )? ;
```

静态变量声明

静态变量类似于常量，不同之处在于它表示程序中的精确内存位置，所有对静态变量的引用都指向相同的内存位置。静态变量具有 static 生命周期，它比 Rust 程序中的所有其他生命周期都长。 静态变量不会在程序结束时调用 drop。

例子：

```
static LEVELS: u32 = 0;
```

可变静态变量

可变静态变量非常有用，它们可以与 C 库一起使用，也可以在 extern 块中从 C 库绑定。可变静态变量与普通静态变量具有相同的限制，不同之处在于可变静态变量的类型不必实现 Sync trait。

例子：

```
static mut LEVELS: u32 = 0;

// This violates the idea of no shared state, and this doesn't internally
```

```
// protect against races, so this function is `unsafe`
unsafe fn bump_levels_unsafe1() -> u32 {
    let ret = LEVELS;
    LEVELS += 1;
    return ret;
}

// Assuming that we have an atomic_add function which returns the old value,
// this function is "safe" but the meaning of the return value may not be what
// callers expect, so it's still marked as `unsafe`
unsafe fn bump_levels_unsafe2() -> u32 {
    return atomic_add(&mut LEVELS, 1);
}
```

Statics or Consts

一般而言，常量应优于静态变量，除非以下情况之一为真：

- 正在存储大量数据。
- 静态变量的单地址属性是必需的。
- 需要内部可变性。

语法

```
StaticItem :
   static mut? IDENTIFIER : Type ( = Expression )? ;
```

泛型参数

function 声明、type 声明、struct 声明、enum 声明、union 声明、trait 声明和 implementation 声明可以通过类型参数、常量参数和生命周期参数进行参数化。这些参数列在尖括号（<...>）中，通常紧跟在项名称之后，定义之前，对于没有名称的 implementation 声明，它们直接出现在 impl 之后。 泛型参数的顺序被限制为生命周期参数，然后是类型参数，然后是常量参数。

具有类型、常量和生命周期参数的项的示例：

```
fn foo<'a, T>() {}
trait A<U> {}
struct Ref<'a, T> where T: 'a { r: &'a T }
struct InnerArray<T, const N: usize>([T; N]);
```

泛型参数在声明它们的项的定义范围内，但是泛型参数的范围不包括在函数体内定义的项。

引用、裸指针、数组、切片、元组和函数指针也有生命周期参数或类型参数，但不使用路径语法引用。

生命周期参数

lifetime bound 的语法格式如下：

```
LifetimeBounds :
   ( Lifetime + )* Lifetime?

Lifetime :
      LIFETIME_OR_LABEL
   | 'static
   | '_
```

lifetime bound 可以应用于类型或其他生命周期。bound 'a:'b 通常读作'a 比'b 长。'a: 'b 意味着 'a 的生命周期持续时间比 'b 长，所以只要 &'b() 有效，引用 &'a() 就是有效的。

T: 'a 意味着 T 的所有生命周期参数都比 'a 长。 例如，如果 'a 是一个不受约束的生命周期参数，那么 i32: 'static 和 &'static str: 'a 是满足的，但 Vec<&'a ()>: 'static 不是。

例子：

```
fn f<'a, 'b>(x: &'a i32, mut y: &'b i32) where 'a: 'b {
    y = x;                          // &'a i32 is a subtype of &'b i32 because 'a: 'b
    let r: &'b &'a i32 = &&0;       // &'b &'a i32 is well formed because 'a: 'b
}
```

类型泛型参数

在具有类型参数声明的 item 的主体内，其类型参数的名称是就是一个类型，例如：

```
fn to_vec<A: Clone>(xs: &[A]) -> Vec<A> {
    if xs.is_empty() {
        return vec![];
    }
    let first: A = xs[0].clone();
    let mut rest: Vec<A> = to_vec(&xs[1..]);
    rest.insert(0, first);
    rest
}
```

其中 A 是类型参数名称，但是在函数项声明体内，A 就是一个类型。

Trait bounds

Trait bound 的语法格式如下：

```
TypeParamBounds :
   TypeParamBound ( + TypeParamBound )* +?

TypeParamBound :
      Lifetime | TraitBound

TraitBound :
      ?? ForLifetimes? TypePath
   | ( ?? ForLifetimes? TypePath )

Lifetime :
      LIFETIME_OR_LABEL
   | 'static
   | '_
```

? 仅用于声明类型参数或关联类型可能没有实现 Sized trait。 ?Sized 不能用作其他类型的 bound。

当类型检查和借用检查泛型 item 时，bound 可用于确定是否为类型实现了 trait。例如，给定 Ty: Trait：

- 在泛型函数的主体中，可以在 Ty 上调用来自 Trait 的方法。
- 可以使用 Trait 上的关联常量。
- 可以使用 Trait 上的关联类型。
- 具有 T: Trait bound 的泛型函数和类型可以与用于 T 的 Ty 一起使用。

例子:

```
trait Shape {
    fn draw(&self, Surface);
    fn name() -> &'static str;
}

fn draw_twice<T: Shape>(surface: Surface, sh: T) {
    sh.draw(surface);           // Can call method because T: Shape
    sh.draw(surface);
}

fn copy_and_draw_twice<T: Copy>(surface: Surface, sh: T) where T: Shape {
    let shape_copy = sh;        // doesn't move sh because T: Copy
    draw_twice(surface, sh);    // Can use generic function because T: Shape
}

struct Figure<S: Shape>(S, S);

fn name_figure<U: Shape>(
    figure: Figure<U>,          // Type Figure<U> is well-formed because U: Shape
) {
    println!(
        "Figure of two {}",
        U::name(),              // Can use associated function
    );
}
```

trait bound 和 lifetime bound 为泛型 item 提供了一种方法来限制使用哪些类型和生命周期作为它们的参数。可以在 where 子句中为任何类型提供 bound。对于某些常见情况，还有更短的形式：

- 在泛型参数声明后面添加 bound：fn f<A: Copy>() {} 与 fn f<A> where A: Copy () {}是一样的。
- 在存在父 trait 的 trait item 声明中：trait Circle : Shape {} 与 trait Circle where Self : Shape {}是一样的。
- 在存在关联类型的 trait item 声明中：trait A { type B: Copy; }与 trait A where Self::B: Copy { type B; }是一样的。

trait bound 可能比生命周期中更优先，即 trait bound 存在生命周期参数。例如 for<'a> &'a T: PartialEq<i32>需要如下的实现：

```
fn isi32<F>(f: F) where for<'a> &'a T: PartialEq<i32> {
    // ...
}

impl<'a> PartialEq<i32> for &'a T {
    // ...
}
```

可以看出对于任意的生命周期&'a 都需要实现 PartialEq<i32>。

当 trait bound 存在生命周期参数时，可以使用如下两种简短方式：

```
fn call_on_ref_zero<F>(f: F) where for<'a> F: Fn(&'a i32) {
    let zero = 0;
    f(&zero);
}
```

或者

```
fn call_on_ref_zero<F>(f: F) where F: for<'a> Fn(&'a i32) {
    let zero = 0;
    f(&zero);
}
```

两者区别在于：第一种形式的生命周期参数范围在整个 bound,第二种生命周期参数返回仅限第一个 trait 尾部。

默认类型参数

当在泛型类型参数时，可以为泛型指定一个默认的具体类型。当实现未指定泛型类型时，则泛型使用默认类型。

默认参数类型主要用于如下两个方面：

- 扩展类型而不破坏现有代码。
- 在大部分用户都不需要的特定情况进行自定义。

例子：

```
trait Add<T=Self> {
    type Output;

    fn add(self, rhs: T) -> Self::Output;
}

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
               Point { x: 3, y: 3 });
}
```

Inferred type

符号"_"是推断的类型，用于要求编译器根据可用的周围信息在可能的情况下推断类型。它不能用于 item 的签名中，通常用于需要指定泛型参数的位置：

```
let x: Vec<_> = (0..10).collect();
```

常量泛型参数

常量泛型参数允许项的常量值是泛型的。const 标识符为常量参数引入了一个名称，并且该项的实例必须使用给定类型的值进行实例化。唯一允许的 const 参数的类型是 u8、u16、u32、u64、u128、usize i8、i16、i32、i64、i128、isize、char 和 bool。

const 泛型参数可以在任何可以使用 const 项的地方使用，但在类型或数组的重复表达式中使用时，它必须是独立的。 也就是说，它们被允许在以下地方使用：

- 作为一个类型的应用常量，称为相关 item 的签名一部分。
- 作为用于定义关联常量的 const 表达式的一部分，或作为关联类型的参数。
- 作为项中的任何函数体中的表达式中的值。
- 作为项中任何函数体中的任何类型的参数。
- 作为项中任何字段类型的一部分。

例子，常量泛型参数允许使用的地方：

```
// Examples where const generic parameters can be used.

// Used in the signature of the item itself.
fn foo<const N: usize>(arr: [i32; N]) {
    // Used as a type within a function body.
    let x: [i32; N];
    // Used as an expression.
    println!("{}", N * 2);
}

// Used as a field of a struct.
struct Foo<const N: usize>([i32; N]);

impl<const N: usize> Foo<N> {
    // Used as an associated constant.
    const CONST: usize = N * 4;
}

trait Trait {
    type Output;
}

impl<const N: usize> Trait for Foo<N> {
    // Used as an associated type.
    type Output = [i32; N];
}
```

例子，常量泛型参数不允许使用的地方：

```
// Examples where const generic parameters cannot be used.
fn foo<const N: usize>() {
    // Cannot use in item definitions within a function body.
    const BAD_CONST: [usize; N] = [1; N];
    static BAD_STATIC: [usize; N] = [1; N];
    fn inner(bad_arg: [usize; N]) {
        let bad_value = N * 2;
    }
    type BadAlias = [usize; N];
    struct BadStruct([usize; N]);
}
```

例子，const 参数只能作为类型或数组的重复表达式内的独立参数出现。 在这些上下文中，它们可能仅用作单段路径表达式，可能在块内（例如 N 或 {N}）。 也就是说，它们不能与其他表达式组合。例如下面使用 N+1 是不合法的：

```
fn bad_function<const N: usize>() -> [u8; {N + 1}] {
    // Similarly not allowed for array repeat expressions.
    [1; {N + 1}]
}
```

路径中的常量参数

路径中的 const 参数指定用于该项的 const 值。参数必须是属于 const 参数类型的 const 表达式。const 表达式必须是块表达式（用大括号括起来），除非它是单个路径段（IDENTIFIER）或字面量（可能带有前导 - 标记）。例如：

```
fn double<const N: i32>() {
    println!("doubled: {}", N * 2);
}

const SOME_CONST: i32 = 12;

fn example() {
    // Example usage of a const argument.
    double::<9>();
    double::<-123>();
    double::<{7 + 8}>();
    double::<SOME_CONST>();
    double::<{ SOME_CONST + 5 }>();
}
```

类型参数与常量参数冲突

当泛型参数既可以解释为类型参数又可以解释为常量参数时，它总是被解释为类型参数。 将参数放在块表达式中可以强制将其解释为 const 参数。例如：

```
type N = u32;
struct Foo<const N: usize>;
// The following is an error, because `N` is interpreted as the type alias `N`.
fn foo<const N: usize>() -> Foo<N> { todo!() } // 此处的 N 由于默认解释为类型参数，出现错误
// Can be fixed by wrapping in braces to force it to be interpreted as the `N`
// const parameter:
fn bar<const N: usize>() -> Foo<{ N }> { todo!() } // 通过将其放置在大括号中，强制解释为 const 参数。
```

未使用的常量参数

与类型参数和生命周期参数不同，const 参数可以在项内部使用的情况下声明，除了在泛型 implementations 中，例子：

```
// ok
struct Foo<const N: usize>;
enum Bar<const M: usize> { A, B }

// ERROR: unused parameter
struct Baz<T>;
struct Biz<'a>;
struct Unconstrained;
impl<const N: usize> Unconstrained {}
```

带有常量参数的类型的 trait bound 解析

当解析一个 trait bound 时，在确定一个带有 const 参数的类型是否实现了 trait 时，不考虑 const 参数的所有实现的穷举性，而是考虑指定的 const 参数的类型是否实现了 trait，因为编译时是是否知道 const 参数的确定值的，因此仍然需要用 const 参数匹配。

例如，如下指定了 Foo 中 const 参数为 const 参数 B，Foo::<B>没有对应的类型实现了 Bar trait，因此编译错误：

```
struct Foo<const B: bool>;
trait Bar {}
impl Bar for Foo<true> {}
impl Bar for Foo<false> {}

fn needs_bar(_: impl Bar) {}
fn generic<const B: bool>() {
    let v = Foo::<B>;
    needs_bar(v); // ERROR: trait bound `Foo<B>: Bar` is not satisfied
}
```

正确的实现应该为：

```
struct Foo<const B: bool>;
trait Bar {}
impl<const B: bool> Bar for Foo<B> {}

fn needs_bar(_: impl Bar) {}
```

```
fn generic<const B: bool>() {
    let v = Foo::<B>;
    needs_bar(v); // ERROR: trait bound `Foo<B>: Bar` is not satisfied
}
```

where 子句

Where 子句提供了另一种指定类型参数界限和生命周期参数界限的方法，以及一种指定非类型参数类型界限的方法。

for 关键字可用于引入 trait bound 优先于生命周期。它只允许 LifetimeParam 参数。

定义项时，还会针对某些通用类型检查 Copy，Clone 和 Sized bound。将 Copy 或 Clone 作为可变引用、特征对象或切片的 bound 或将 Sized 作为特征对象或切片的 bound 都是错误的，因为类型明显是不会实现这些 trait 的。

例子：

```
struct A<T>
where
    T: Iterator,            // Could use A<T: Iterator> instead
    T::Item: Copy,
    String: PartialEq<T>,
    i32: Default,           // Allowed, but not useful
    i32: Iterator,          // Error: the trait bound is not satisfied
    [T]: Copy,              // Error: the trait bound is not satisfied
{
    f: T,
}
```

语法

```
WhereClause :
   where ( WhereClauseItem , )* WhereClauseItem ?

WhereClauseItem :
      LifetimeWhereClauseItem
   | TypeBoundWhereClauseItem

LifetimeWhereClauseItem :
   Lifetime : LifetimeBounds

TypeBoundWhereClauseItem :
   ForLifetimes? Type : TypeParamBounds?
```

泛型参数属性

生命周期参数和类型参数允许对它们进行属性注解。没有内置属性可以在此位置执行任何操作，但是可以自定义派生属性赋予其意义。

此示例显示使用自定义派生属性来修改泛型参数的含义。

```
// Assume that the derive for MyFlexibleClone declared `my_flexible_clone` as
// an attribute it understands.
#[derive(MyFlexibleClone)]
struct Foo<#[my_flexible_clone(unbounded)] H> {
    a: *const H
}
```

语法

```
GenericParams :
      < >
   | < (GenericParam ,)* GenericParam ,? >

GenericParam :
   OuterAttribute* ( LifetimeParam | TypeParam | ConstParam )

LifetimeParam :
   LIFETIME_OR_LABEL ( : LifetimeBounds )?
```

```
TypeParam :
   IDENTIFIER( : TypeParamBounds? )? ( = Type )?

ConstParam:
   const IDENTIFIER : Type
```

语法

```
Item:
   OuterAttribute*
      VisItem
   | MacroItem

VisItem:
   Visibility?
   (
         Module
      | ExternCrate
      | UseDeclaration
      | Function
      | TypeAlias
      | Struct
      | Enumeration
      | Union
      | ConstantItem
      | StaticItem
      | Trait
      | Implementation
      | ExternBlock
   )

MacroItem:
      MacroInvocationSemi
   | MacroRulesDefinition
```

Let 语句

let 语句引入了一组由一个 irrefutable pattern(始终能匹配的 pattern)给出的新变量。pattern 后跟可选的类型注解，然后是可选的初始化表达式。由变量声明引入的变量的可见范围是从声明点到封闭的块作用域结束位置。

当没有给出类型注解时，编译器将推断变量类型，或者如果没有足够的类型信息可用于推断，则编译会抛出错误。

例子：

```
let foo = 1;
let guess: u32 = 1;
```

语法

```
LetStatement :
   OuterAttribute* let PatternNoTopAlt ( : Type )? (= Expression )? ;
```

表达式语句

表达式语句是对表达式进行求值并忽略其结果的语句。通常，表达式语句的目的是触发对表达式求值的效果。

仅由块表达式或控制流表达式组成的表达式，如果在允许使用语句的上下文中使用，则可以省略尾随分号。省略尾随分号这可能会导致将其解析为独立语句和作为另一个表达式的一部分之间的歧义，但是在这种情况下，它被解析为一个语句。当表达式(没有尾部分号)直接用作语句时 ExpressionWithBlock 表达式的类型必须是单元类型"()"。

例子：

```
v.pop();          // Ignore the element returned from pop
if v.is_empty() {
    v.push(5);
} else {
    v.remove(0);
}                 // Semicolon can be omitted.
```

```
[1];              // Separate expression statement, not an indexing expression.
```

如果表达式块表达式或控制流表达式的值的类型不为单元类型"()"，那么必须添加分号指定为表达式语句。例如：

```
// bad: the block's type is i32, not ()
// Error: expected `()` because of default return type
// if true {
//   1
// }

// good: the block's type is i32
if true {
  1
} else {
  2
};
```

语法

```
ExpressionStatement :
      ExpressionWithoutBlock ;
   | ExpressionWithBlock ;?
```

宏调用

宏调用必须以分号结尾，其格式为 MacroInvocationSemi。详见宏。

语法

```
Statement :
      ;
   | Item
   | LetStatement
   | ExpressionStatement
   | MacroInvocationSemi
```

表达式

一个表达式可能有两个作用：它总是产生一个值，并且可能会产生影响（也称为“副作用”）。表达式计算为一个值，并在计算过程中产生影响。许多表达式包含子表达式，称为表达式的操作数。每种表达的含义决定了几件事：

- 计算表达式时是否计算操作数。
- 计算操作数的顺序。
- 如何组合操作数的值以获得表达式的值。

这样，表达式的结构决定了执行的结构。块只是另一种表达式，所以块、语句、表达式然后又块可以递归嵌套到任意深度。

字面量表达式

字面量表达式由字面量组成。它直接描述数字、字符、字符串或布尔值。

例子：

```
"hello";   // string type
'5';       // character type
5;         // integer type
```

语法

```
LiteralExpression :
      CHAR_LITERAL
   | STRING_LITERAL
   | RAW_STRING_LITERAL
   | BYTE_LITERAL
   | BYTE_STRING_LITERAL
   | RAW_BYTE_STRING_LITERAL
   | INTEGER_LITERAL
   | FLOAT_LITERAL
   | BOOLEAN_LITERAL
```

路径表达式

用作表达式上下文的路径用来表示局部变量或一个 item。

解析为局部变量或静态变量的路径表达式是位置表达式，其他路径是值表达式。

使用静态 mut 变量需要一个在 unsafe 块内。

具体路径语法见这里。

例子：

```
local_var; // 局部变量路径
globals::STATIC_VAR; //静态变量路径
unsafe { globals::STATIC_MUT_VAR }; //unsafe 块内的静态可变变量路径
let some_constructor = Some::<i32>;
let push_integer = Vec::<i32>::push;
let slice_reverse = <[i32]>::reverse;
```

语法

```
PathExpression :
      PathInExpression
   | QualifiedPathInExpression
```

块表达式

块表达式或块是一个控制流表达式，同时也是程序项声明和变量声明的匿名空间作用域。 作为控制流表达式，块按顺序执行其非项声明的语句组件，最后执行可选的最终表达式。 作为一个匿名空间作用域，在本块内声明的项只在块作用域内有效，而块内由 let 语句声明的变量的作用域为下一条语句到块结尾。

块的语法规则为：先是{，后跟内部属性，再后是任意条语句，再后是一个被称为最终操作数的可选表达式，以 }结尾。

语句之间通常需要后跟分号，但有两个例外：

- 项声明语句不需要后跟分号。
- 表达式语句通常需要后面的分号，但它的外层表达式是控制流表达式时不需要。

此外，允许在语句之间使用额外的分号，但是这些分号并不影响语义。

在对块表达式进行求值时，除了项声明语句外，每个语句都是按顺序执行的。如果给出了块尾的可选的最终操作数，则最后会执行它。块总是值表达式，并会在值表达式上下文中对最后的那个操作数进行求值。块的类型是最此块的最终操作数的类型，但如果省略了最终操作数，则块的类型为 ()。

例子：

```
let _: () = {
    fn_call();
};

let five: i32 = {
    fn_call();
    5
};

assert_eq!(5, five);
```

async 块表达式

async 块是求值为 future 的块表达式的一个变种。 块的最终表达式（如果存在）决定了 future 的结果值。

执行一个异步块类似于执行一个闭包表达式：它的即时效果是生成并返回一个匿名类型，异步块返回的类型实现了 std::future::Future trait。 此类型的实际数据格式规范还未确定下来。

注意： rustc 生成的 future 类型大致相当于一个枚举，rustc 为这个 future 的每个 await 点生成一个此枚举的变体，其中每个变体都存储了对应点再次恢复执行时需要的数据。

捕获模式

异步块使用与闭包相同的捕获模式从其环境中捕获变量。 跟闭包一样，当编写 async { .. } 时，每个变量的捕获方式将从该块里的内容中推断出来。 而 async move { .. } 类型的异步块将把所有需要捕获的变量使用移动语义移入(move)到相应的结果 future 中。

异步上下文

因为异步块构造了一个 future，所以它们定义了一个 async 上下文，这个上下文可以相应地包含 await 表达式。 异步上下文是由异步块和异步函数的函数体建立的，它们的语义是依照异步块定义的。

控制流操作符

异步块的作用类似于函数边界，或者更类似于闭包。 因此 ?操作符和 返回(return)表达式也都能影响 future 的输出，且不会影响封闭它的函数或其他上下文。 也就是说，future 的输出跟闭包将其中的 return <expr> 的表达式 <expr> 的计算结果作为未来的输出的做法是一样的。 类似地,如果 <expr>? 传播(propagate)一个错误，这个错误也会被 future 在未来的某个时候作为返回结果被传播出去。

关键字 break 和 continue 不能用于跳出异步块。 因此，以下内容是非法的：

```
loop {
    async move {
        break; // Error。
    }
}
```

语法

```
AsyncBlockExpression :
   async move? BlockExpression
```

unsafe 块表达式

一个代码块可以以关键字 unsafe 作为前缀，以允许在安全(safe)函数中调用非安全(unsafe)函数或对裸指针做解引用操作。

当程序员确信某些潜在的非安全操作实际上是安全的，他们可以将这段代码（作为一个整体）封装进一个非安全(unsafe)块中。编译器将认为在当前的上下文中使用这样的代码是安全的。

非安全块用于封装外部库、直接操作硬件或实现语言中没有直接提供的特性。例如，Rust 提供了实现内存安全并发所需的语言特性，但是线程和消息传递的实现（没在语言中实现，而）是在标准库中实现的。

Rust 的类型系统是动态安全条款(dynamic safety requirements)的保守近似值，因此在某些情况下使用安全代码会带来性能损失。例如，双向链表不是树型结构，那在安全代码中，只能妥协使用引用计数指针表示。通过使用非安全(unsafe)块，可以将反向链接表示为原始指针，这样只用一层 box 封装就能实现了。

例子：

```
unsafe {
    let b = [13u8, 17u8];
    let a = &b[0] as *const u8;
    assert_eq!(*a, 13);
    assert_eq!(*a.offset(1), 17);
}

let a = unsafe { an_unsafe_fn() };
```

语法

```
UnsafeBlockExpression :
   unsafe BlockExpression
```

块表达式上的属性

在以下情况下，允许在块表达式的左大括号后直接使用内部属性：

- 函数体或方法体。
- 循环体（loop、while、while let 和 for）。
- 用作语句的块表达式。
- 作为数组表达式、元组表达式、调用表达式和元组结构表达式初始化元素的块表达式。
- 作为另一个块表达式的尾块表达式。

在块表达式上有意义的外部属性有 cfg 和 lint 检查属性。

例子：

```
fn is_unix_platform() -> bool {
    #[cfg(unix)] { true }
    #[cfg(not(unix))] { false }
}
```

语法

```
BlockExpression :
   {
      InnerAttribute*
      Statements?
   }

Statements :
      Statement+
   | Statement+ ExpressionWithoutBlock
   | ExpressionWithoutBlock
```

运算符表达式

运算符是 Rust 语言为其内建类型定义的。许多操作符都可以使用 std::ops 或 std::cmp 中的 trait 进行重载。

借用运算符

&（共享借用）和 &mut（可变借用）运算符是一元前缀运算符。

当借用运算符应用于位置表达式上时，此表达式生成指向值所在的内存位置的引用（指针）。在引用存续期间，该内存位置也被置于借出状态。对于共享借用（&），这意味着该位置可能不会发生变化，但可能会被再次读取或共享。 对于可变借用（&mut），在借用到期之前，不能以任何方式访问该位置。&mut 在可变位置表达式上下文中会对其操作数求值。如果 & 或 &mut 运算符应用于值表达式上，则会创建一个临时值。

借用运算符是不能重载的。

例子：

```
{
    // a temporary with value 7 is created that lasts for this scope.
    let shared_reference = &7;
}
let mut array = [-2, 3, 9];
{
    // Mutably borrows `array` for this scope.
    // `array` may only be used through `mutable_reference`.
    let mutable_reference = &mut array;
}
```

&&借用运算符

尽管 && 是一个单一 token（and 运算符），但在借用表达式上下文中使用时，它是作为两个借用操作符用的。

例子：

```
// 意义相同：
let a = &&  10;
let a = & & 10;

// 意义相同：
let a = &&&&  mut 10;
let a = && && mut 10;
let a = & & & & mut 10;
```

语法

```
BorrowExpression :
      (&|&&) Expression
```

```
    | (&|&&) mut Expression
```

解引用运算符

*（解引用）操作符也是一元前缀操作符。

当应用于指针上时，表达式表示该指针指向的内存位置。 如果表达式的类型为 &mut T 或 *mut T，并且该表达式是局部变量、局部变量的（内嵌）字段、或是可变的位置表达式，则它代表的内存位置可以被赋值。解引用裸指针需要在非安全(unsafe)块才能进行。

在不可变位置表达式上下文中对非指针类型作 *x 相当于执行 *std::ops::Deref::deref(&x);
在可变位置表达式上下文中对非指针类型作 *x 相当于执行 *std::ops::DerefMut::deref_mut(&mut x)。

例子：

```
let x = &7;
assert_eq!(*x, 7);
let y = &mut 9;
*y = 11;
assert_eq!(*y, 11);
```

语法

```
DereferenceExpression :
   * Expression
```

问号运算符

问号操作符（?）是一个一元后缀操作符，只能应用于类型 Result<T, E> 和 Option<T>。

问号操作符（?）将对有效值进行解包，如果是错误值则返回错误值将它们传播(propagate)给调用函数。

问号操作符 ？ 不能被重载。

应用于 Result<T, E>

当应用在 Result<T, E> 类型的值上时，它可以传播错误。如果值是 Err(e)，那么它实际上将从此操作符所在的函数体或闭包中返回 Err(From::from(e))。 如果应用到 Ok(x)，那么它将解包此值以求得 x。

例子：

```
fn try_to_parse() -> Result<i32, ParseIntError> {
    let x: i32 = "123".parse()?; // x = 123
    let y: i32 = "24a".parse()?; // 立即返回一个 Err()
    Ok(x + y)                    // 不会执行到这里
}

let res = try_to_parse();
println!("{:?}", res);
```

应用于 Option<T>

当应用到 Option<T> 类型的值时，它向调用者传播错误 None。 如果它应用的值是 None，那么它将返回 None。 如果应用的值是 Some(x)，那么它将解包此值以求得 x。

例子：

```
fn try_option_some() -> Option<u8> {
    let val = Some(1)?;
    Some(val)
}
assert_eq!(try_option_some(), Some(1));

fn try_option_none() -> Option<u8> {
    let val = None?;
    Some(val)
}
assert_eq!(try_option_none(), None);
```

语法

```
ErrorPropagationExpression :
   Expression ?
```

取反运算符

运算符-和!也是一元运算符。

下表总结了它们用在基本类型上的表现，同时指出其他类型要重载这些操作符需要实现的 trait。

符号	整数	bool	浮点数	用于重载的 trait
-	符号取反*		符号取反	std::ops::Neg
!	按位取反	逻辑非		std::ops::Not

* 仅适用于有符号整数类型。

所有这些运算符的操作数都在值表达式上下文中被求值，所以这些操作数的值会被移走或复制。

例子：

```
let x = 6;
assert_eq!(-x, -6);
assert_eq!(!x, -7);
assert_eq!(true, !false);
```

语法

```
NegationExpression :
      - Expression
   | ! Expression
```

算术和逻辑二元运算符

二元运算符表达式都用中缀表示法书写。

下表总结了算术和逻辑二元运算符在原生类型(primitive type)上的行为，同时指出其他类型要重载这些操作符需要实现的 trait。

符号	整数	bool	浮点数	用于重载此运算符的 trait	用于重载此运算符的复合赋值(Compound Assignment) Trait
+	加法		加法	std::ops::Add	std::ops::AddAssign
-	减法		减法	std::ops::Sub	std::ops::SubAssign
*	乘法		乘法	std::ops::Mul	std::ops::MulAssign
/	除法*		取余	std::ops::Div	std::ops::DivAssign
%	取余		Remainder	std::ops::Rem	std::ops::RemAssign
&	按位与	逻辑与		std::ops::BitAnd	std::ops::BitAndAssign
|	按位或	逻辑或		std::ops::BitOr	std::ops::BitOrAssign
^	按位异或	逻辑异或		std::ops::BitXor	std::ops::BitXorAssign
<<	左移位			std::ops::Shl	std::ops::ShlAssign
>>	右移位**			std::ops::Shr	std::ops::ShrAssign

* 整数除法趋零取整。
** 有符号整数类型算术右移位，无符号整数类型逻辑右移位。

例子：

```
assert_eq!(3 + 6, 9);
assert_eq!(5.5 - 1.25, 4.25);
assert_eq!(-5 * 14, -70);
assert_eq!(14 / 3, 4);
assert_eq!(100 % 7, 2);
assert_eq!(0b1010 & 0b1100, 0b1000);
assert_eq!(0b1010 | 0b1100, 0b1110);
assert_eq!(0b1010 ^ 0b1100, 0b110);
assert_eq!(13 << 3, 104);
assert_eq!(-10 >> 2, -3);
```

语法

```
ArithmeticOrLogicalExpression :
      Expression + Expression
    | Expression - Expression
    | Expression * Expression
    | Expression / Expression
    | Expression % Expression
    | Expression & Expression
    | Expression | Expression
    | Expression ^ Expression
    | Expression << Expression
    | Expression >> Expression
```

比较运算符

Rust 还为原生类型以及标准库中的多种类型都定义了比较运算符。

与算术运算符和逻辑运算符不同，重载这些运算符的 trait 通常用于显示/约定如何比较一个类型，并且还很可能会假定使用这些 trait 作为约束条件的函数定义了实际的比较逻辑。 其实标准库中的许多函数和宏都使用了这个假定（尽管不能确保这些假定的安全性）。 与上面的算术和逻辑运算符不同，比较运算符会隐式地对它们的操作数执行共享借用，并在位置表达式上下文中对它们进行求值，例如：

```
a == b;
// 等价于：
::std::cmp::PartialEq::eq(&a, &b);
```

这意味着不需要将值从操作数移出。

下表总结了比较运算符的行为，同时指出其他类型要重载这些操作符需要实现的 trait。

符号	含义	须重载方法
==	等于	std::cmp::PartialEq::eq
!=	不等于	std::cmp::PartialEq::ne
>	大于	std::cmp::PartialOrd::gt
<	小于	std::cmp::PartialOrd::lt
>=	大于或等于	std::cmp::PartialOrd::ge
<=	小于或等于	std::cmp::PartialOrd::le

链式比较运算时需要借助圆括号，例如，表达式 a == b == c 是无效的，（但如果逻辑允许）可以写成 (a == b) == c。

例子：

```
assert!(123 == 123);
assert!(23 != -12);
assert!(12.5 > 12.2);
assert!([1, 2, 3] < [1, 3, 4]);
assert!('A' <= 'B');
assert!("World" >= "Hello");
```

语法

```
ComparisonExpression :
      Expression == Expression
    | Expression != Expression
    | Expression > Expression
    | Expression < Expression
    | Expression >= Expression
    | Expression <= Expression
```

短路布尔运算符

运算符 || 和 && 可以应用在布尔类型的操作数上。 运算符 || 表示逻辑“或”，运算符 && 表示逻辑“与”。 它们与 | 和 & 的不同之处在于，只有在左操作数尚未确定表达式的结果时，才计算右操作数。 也就是说，|| 只在左操作数的计算结果为 false 时才计算其右操作数，而只有在计算结果为 true 时才计算 && 的操作数。

例子：

```
let x = false || true; // true
let y = false && panic!(); // false, 不会计算 `panic!()`
```

语法

```
LazyBooleanExpression :
      Expression || Expression
   | Expression && Expression
```

赋值运算符

赋值运算符表达式会把某个值移入到一个特定的位置。

赋值表达式由一个可变位置表达式（就是被赋值的位置操作数）后跟等号（=）和值表达式（就是被赋值的值操作数）组成。与其他位置操作数不同，赋值位置操作数必须是一个位置表达式。 试图使用值表达式将导致编译器报错，而不是将其提升转换为临时位置。

赋值表达式要先计算它的操作数。赋值的值操作数先被求值，然后是赋值的位置操作数。

对赋值表达的位置表达式求值时会先销毁(drop)此位置（如果是未初始化的局部变量或未初始化的局部变量的字段则不会启动这步析构操作），然后将赋值值复制(copy)或移动(move)到此位置中。

赋值表达式总是会生成单元类型值。

例子：

```
let mut x = 0;
let y = 0;
x = y;
```

语法

```
AssignmentExpression :
   Expression = Expression
```

复合赋值运算符

复合赋值表达式将算术运算符（以及二进制逻辑操作符）与赋值运算符相结合在一起使用。

复合赋值的语法是可变位置表达式（被赋值操作数），然后是一个操作符再后跟一个 =（这两个符号共同作为一个单独的 token），最后是一个值表达式（也叫被复合修改操作数(modifying operand)）。与其他位置操作数不同，被赋值的位置操作数必须是一个位置表达式。试图使用值表达式将导致编译器报错，而不是将其提升转换为临时位置。

复合赋值表达式的求值取决于操作符的类型。如果复合赋值表达式了两个操作数的类型都是原生类型，则首先对被复合修改操作数进行求值，然后再对被赋值操作数求值。 最后将被赋值操作数的位置值设置为原被赋值操作数的值和复合修改操作数执行运算后的值。而对于非原生类型，左边操作数将首先被求值。

此外，复合赋值表达式是调用操作符重载复合赋值 trait 的函数的语法糖。 被赋值操作数必须是可变的。

复合赋值表达式也总是会生成单元类型值。

例子：

```
impl AddAssign<Addable> for Addable {
    /* */
}

fn example() {
  a1 += a2;
  // 等价于
  AddAssign::add_assign(&mut a1, a2);
}
```

语法

```
CompoundAssignmentExpression :
      Expression += Expression
   | Expression -= Expression
   | Expression *= Expression
   | Expression /= Expression
   | Expression %= Expression
```

```
    | Expression &= Expression
    | Expression |= Expression
    | Expression ^= Expression
    | Expression <<= Expression
    | Expression >>= Expression
```

运算溢出

在 debug 模式下编译整数运算时，如果发生溢出，会触发 panic。 可以使用命令行参数 -C debug-assertions 和 -C overflow-checks 设置编译器标志位来更直接地控制这个溢出过程。 以下情况被认为是溢出：

- 当 +、* 或 - 创建的值大于当前类型可存储的最大值或小于最小值。这包括任何有符号整型的最小值上的一元运算符 -。
- 使用 / 或 %，其中左操作数是某类有符号整型的最小整数，而右操作数是 -1。
- 使用 << 或 >>，其中右操作数大于或等于左操作数类型的 bit 数，或右操作数为负数。

语法

```
OperatorExpression :
      BorrowExpression
    | DereferenceExpression
    | ErrorPropagationExpression
    | NegationExpression
    | ArithmeticOrLogicalExpression
    | ComparisonExpression
    | LazyBooleanExpression
    | TypeCastExpression
    | AssignmentExpression
    | CompoundAssignmentExpression
```

类型转换表达式

类型转换表达式用二元运算符 as 表示。

执行类型转换(as)表达式将左侧的值显式转换为右侧的类型。

as 不仅可用于强制转换点的显式强制转换，也可以用于下列形式的强制转换。 任何不符合强制转换规则或不在下表中的转换都会导致编译器报错。下表中 *T 代表 *const T 或 *mut T。

e 的类型	U	通过 e as U 执行转换
整型或浮点型	整型或浮点型	数字转换
类 C(C-like)枚举	整型	枚举转换
bool 或 char	整型	原生类型到整型的转换
u8	char	u8 到 char 的转换
*T	*V where V: Sized *	指针到指针的转换
*T where T: Sized	数字型(Numeric type)	指针到地址的转换
整型	*V where V: Sized	地址到指针的转换
&m₁ T	*m₂ T **	引用到指针的转换
&m₁ [T; n]	*m₂ T **	数组到指针的转换
函数项	函数指针	函数到函数指针的转换
函数项	*V where V: Sized	函数到指针的转换
函数项	整型	函数到地址的转换
函数指针	*V where V: Sized	函数指针到指针的转换
函数指针	整型	函数指针到地址的转换
闭包 ***	函数指针	闭包到函数指针的转换

* 或者 T 和 V 也可以都是兼容的 unsized 类型，例如，两个都是切片，或者都是同一种 trait 对象。
** 仅当 m₁ 是 mut 或 m₂ 是 const 时， 可变引用到 const 指针才会被允许。
*** 仅适用于不捕获（遮蔽(close over)）任何环境变量的闭包。

不同的转换的含义如下：

- 数字转换(Numeric cast)
  - 在两个尺寸(size)相同的整型数值（例如 i32 -> u32）之间进行转换是一个空操作(no-op)
  - 从一个较大尺寸的整型转换为较小尺寸的整型（例如 u32 -> u8）将会采用截断(truncate)算法 [1]
  - 从较小尺寸的整型转换为较大尺寸的整型（例如 u8 -> u32）将

        - 如果源数据是无符号的，则进行零扩展(zero-extend)
        - 如果源数据是有符号的，则进行符号扩展(sign-extend)
    - 从浮点数转换为整型将使浮点数趋零取整(round the float towards zero)
        - NaN 将返回 0
        - 大于转换到的整型类型的最大值时，取该整型类型的最大值。
        - 小于转换到的整型类型的最小值时，取该整型类型的最小值。
    - 从整数强制转换为浮点数将产生最接近的浮点数 *
        - 如有必要，舍入采用 roundTiesToEven 模式 ***
        - 在溢出时，将会产生该浮点型的常量 Infinity(∞)（与输入符号相同）
        - 注意：对于当前的数值类型集，溢出只会发生在 u128 as f32 这种转换形式，且数字大于或等于 f32::MAX + (0.5 ULP) 时。
    - 从 f32 到 f64 的转换是无损转换
    - 从 f64 到 f32 的转换将产生最接近的 f32 **
        - 如有必要，舍入采用 roundTiesToEven 模式 ***
        - 在溢出时，将会产生 f32 的常量 Infinity(∞)（与输入符号相同）
- 枚举转换(Enum cast)
    - 先将枚举转换为它的判别值(discriminant)，然后在需要时使用数值转换。
- 原生类型到整型的转换
    - false 转换为 0, true 转换为 1
    - char 会先强制转换为代码点的值，然后在需要时使用数值转换。
- u8 到 char 的转换
    - 转换为具有相应代码点的 char 值。

* 如果硬件本身不支持这种舍入模式和溢出行为，那么这些整数到浮点型的转换可能会比预期的要慢。
** 如果硬件本身不支持这种舍入模式和溢出行为，那么这些 f64 到 f32 的转换可能会比预期的要慢。
*** 按照 IEEE 754-2008§4.3.1 的定义：选择最接近的浮点数，如果恰好在两个浮点数中间，则优先选择最低有效位为偶数的那个。

例子：

```
fn average(values: &[f64]) -> f64 {
    let sum: f64 = sum(values);
    let size: f64 = len(values) as f64;
    sum / size
}
```

语法

```
TypeCastExpression :
   Expression as TypeNoBounds
```

分组表达式

分组表达式使用圆括号包装单个表达式，并对该表达式求值分组表达式的语法规则就是一对圆括号封闭一个被称为封闭操作数的表达式。

分组表达式的求值结果就是在其内的表达式的求值结果。与其他表达式不同，分组表达式可以是位置表达式或值表达式。当封闭操作数是位置表达式时，它是一个位置表达式；当封闭操作数是一个值表达式是，它是一个值表达式。

圆括号可用于显式修改表达式中的子表达式的优先顺序。

例子：

```
let x: i32 = 2 + 3 * 4;
let y: i32 = (2 + 3) * 4;
assert_eq!(x, 14);
assert_eq!(y, 20);

//当调用结构体的函数指针类型的成员时，必须使用括号
assert_eq!( a.f (), "The method f");
assert_eq!((a.f)(), "The field f");
```

分组表达式上的属性

在允许块表达式上的属性存在的那几种表达式上下文中，可以在分组表达式的左括号后直接使用内部属性。

语法

```
GroupedExpression :
   ( InnerAttribute* Expression )
```

数组表达式

数组表达式用来构建数组。 数组表达式有两种形式。

- 第一种形式是在数组中列举出所有的元素值，即[a,b,c...]。 这种形式的语法规则通过在方括号中放置统一类型的、逗号分隔的表达式来表现。这样编写将生成一个包含这些表达式的值的数组，其中数组元素的顺序就是这些表达式写入的时顺序。
- 第二种形式的语法规则通过在方括号内放置两个用分号(;)分隔的表达式来表现，即[a;b]。分号(;)前的表达式被称为重复操作数，分号(;)后的表达式被称为数组长度操作数。其中，数组长度操作数必须是 usize 类型的，并且必须是常量表达式，比如是字面量或常量项。

[a; b] 这种形式会创建包含 b 个 a 值的数组。当 [a; b] 形式的重复体操作数 a 是一个常量项时，其将被计算求值数组长度操作数 b 次。 如果数组长度操作数 b 为 0，则常量项根本不会被求值。 对于非常量项的表达式，只计算求值一次，然后将结果复制数组长度操作数 b 次。因此，对于[a; b]这种形式时，如果数组长度操作数的值大于 1，则要求 a 的类型实现了 Copy，或 a 自己是一个常量项的路径。

注意：[a; b] 这种形式，当数组长度操作数为 0 时，而重复操作数是一个非常量项的情况下，目前在 rustc 中存在一个 bug，即值 a 会被求值，但不会被销毁而导致的内存泄漏。

例子：

```
[1, 2, 3, 4];
["a", "b", "c", "d"];
[0; 128];                 // 内含 128 个 0 的数组
[0u8, 0u8, 0u8, 0u8,];
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]; // 二维数组
const EMPTY: Vec<i32> = Vec::new();
[EMPTY; 2];
```

数组表达式上的属性

在允许块表达式上的属性存在的那几种表达式上下文中，可以在数组表达式的左括号后直接使用内部属性。

语法

```
ArrayExpression :
   [ InnerAttribute* ArrayElements? ]

ArrayElements :
      Expression ( , Expression )* ,?
   | Expression ; Expression
```

索引表达式

数组和切片类型的值可以通过后跟一个由方括号封闭一个类型为 usize 的表达式（索引）的方式来对此数组或切片进行索引检索。 如果数组是可变的，则其检索出的内存位置还可以被赋值。

数组和切片以外的类型可以通过实现 Index trait 和 IndexMut trait 来达成数组索引表达式的效果。对于数组和切片类型之外的索引表达式 a[b] 其实相当于执行 *std::ops::Index::index(&a, b)， 或者在可变位置表达式上下文中相当于执行 *std::ops::IndexMut::index_mut(&mut a, b)。 与普通方法一样，Rust 也将在 a 上反复插入解引用操作，直到查找到对上述方法的实现。

数组和切片的索引是从零开始的。数组访问是一个常量表达式，因此数组索引的越界检查可以在编译时通过检查常量索引值本身进行。否则，越界检查将在运行时执行，如果此时越界检查未通过，那将把当前线程置于 panicked 状态。

例子：

```
// 默认情况下,`unconditional_panic` lint 检查会执行 deny 级别的设置，
// 即 crate 在默认情况下会有外部属性设置 `#[deny(unconditional_panic)]`
// 而像 `(["a", "b"])[n]` 这样的简单动态索引检索会被该 lint 检查出来，而提前报错，导致程序被拒绝编译。
// 因此这里调低 `unconditional_panic` 的 lint 级别以通过编译。
#![warn(unconditional_panic)]

([1, 2, 3, 4])[2];        // 3

let b = [[1, 0, 0], [0, 1, 0], [0, 0, 1]];
b[1][2];                  // 多维数组索引

let x = (["a", "b"])[10]; // 告警：索引越界

let n = 10;
// 译者注：上行可以在 `#![warn(unconditional_panic)]` 被注释的情况下换成
// let n = if true {10} else {0};
```

```
// 试试，那下行就不会被 unconditional_panic lint 检查到了

let y = (["a", "b"])[n];  // panic

let arr = ["a", "b"];
arr[10];                  // 告警：索引越界
```

语法

```
IndexExpression :
   Expression [ Expression ]
```

元组表达式

元组表达式用来构建元组类型的值。

元组表达式的语法规则为：一对圆括号封闭的以逗号分隔的表达式列表，这些表达式被称为元组初始化操作数。为了避免和分组表达式混淆，一元元组表达式的元组初始化操作数后的逗号不能省略。

元组表达式是一个值表达式，它会被求值计算成一个元组类型的新值。 元组初始化操作数的数量构成元组的元数(arity)。 没有元组初始化操作数的元组表达式生成单元元组(unit tuple)。 对于其他元组表达式，第一个被写入的元组初始化操作数初始化第 0 个元素，随后的操作数依次初始化下一个开始的元素。 例如，在元组表达式（'a', 'b', 'c'）中，'a' 初始化第 0 个元素的值，'b' 初始化第 1 个元素，'c' 初始化第 2 个元素。

元组表达式和相应类型的示例：

表达式	类型
()	() (unit)
(0.0, 4.5)	(f64, f64)
("x".to_string(),)	(String,)
("a", 4usize, true)	(&'static str, usize, bool)

元组表达式上的属性

在允许块表达式上的属性存在的那几种表达式上下文中，可以在元组表达式的左括号后直接使用内部属性。

语法

```
TupleExpression :
   ( InnerAttribute* TupleElements? )

TupleElements :
   ( Expression , )+ Expression?
```

元组索引表达式

元组索引表达式被用来存取元组或元组结构体的字段。

元组索引表达式的语法规则为：一个被称为元组操作数的表达式后跟一个句点，最后再后跟一个元组索引。元组操作数的类型必须是元组类型或元组结构体。元组索引的语法规则要求该索引必须写成一个不能有前导零、下划线和后缀的十进制字面量的形式。 例如 0 和 2 是合法的元祖索引，但 01、0_、0i32 这些不行。

对元组索引表达式的求值计算除了能求取其元组操作数的对应位置的值之外没有其他作用。作为位置表达式，元组索引表达式的求值结果是元组操作数字段的位置，该字段与元组索引同名。

与字段访问表达式不同，元组索引表达式可以是调用表达式的函数操作数。因为元组索引表达式不会与方法调用相混淆，因为方法名不可能是数字。

例子：

```
// 索引检索一个元组
let pair = ("a string", 2);
assert_eq!(pair.1, 2);

// 索引检索一个元组结构体
let point = Point(1.0, 0.0);
```

```
assert_eq!(point.0, 1.0);
assert_eq!(point.1, 0.0);
```

语法

```
TupleIndexingExpression :
   Expression . TUPLE_INDEX
```

结构体表达式

结构体表达式可以创建 struct、enum 或 union 值。 它由 struct、enum variant 或 union 项的路径以及与此项的字段对应的值组成。结构体表达式共有三种形式：字段结构体、元组结构体和单元结构体。

例子：

```
Point {x: 10.0, y: 20.0};
NothingInMe {};
TuplePoint(10.0, 20.0);
TuplePoint { 0: 10.0, 1: 20.0 }; // 效果和上一行一样
let u = game::User {name: "Joe", age: 35, score: 100_000};
some_fn::<Cookie>(Cookie);
```

字段结构体表达式

用花括号把字段括起来的字段结构体表达式允许以任意顺序指定每个字段的值。字段名与值之间用冒号分隔。

union 类型的值只能使用此语法创建，并且只能指定一个字段。

带花括号的字段结构体表达式不能直接用在循环表达式或 if 表达式的头部，也不能直接用在 if let 或匹配表达式的检验对象(scrutinee)上。 但是，如果结构体表达式在另一个表达式内（例如在圆括号内），则可以用在这些情况下。

构造元组结构体时其字段名可以是代表索引的十进制整数数值：

```
struct Color(u8, u8, u8);
let c1 = Color(0, 0, 0);  // 创建元组结构体的典型方法。
let c2 = Color{0: 255, 1: 127, 2: 0};  // 按索引来指定字段。
```

初始化字段的快捷方法

当使用字段的名字（注意不是位置索引数字）初始化某数据结构（结构体、枚举、联合体）时，允许将 fieldname: fieldname 写成 fieldname 这样的简化形式。 这种句法让代码更少重复，更加紧凑。

例子：

```
Point3d { x: x, y: y_value, z: z };
Point3d { x, y: y_value, z };
```

函数式更新

当使用字段结构体表达式构建一个结构体类型的值时可以以 .. 后跟一个表达式的语法结尾，这种句法表示这是一种函数式更新。 .. 后跟的表达式（此表达式被称为此函数式更新的基(base)）必须与正在构造的新结构体值是同一种结构体类型的。

字段结构体表达式执行时先为已指定的字段使用已给定的值，然后再从基表达式(base expression)里为剩余未指定的字段移动或复制值。与所有结构体表达式一样，字段结构体类型的所有字段必须是可见的，甚至那些没有显式命名的字段也是如此。

例子：

```
let mut base = Point3d {x: 1, y: 2, z: 3};
let y_ref = &mut base.y;
Point3d {y: 0, z: 10, .. base}; // OK, 只有 base.x 获取进来了
drop(y_ref);
```

构造元组结构体时其字段名可以是代表索引的十进制整数数值。这中表达方法还可以与基结构体一起使用来填充其余未指定的索引：

```
struct Color(u8, u8, u8);
let c1 = Color(0, 0, 0);  // 创建元组结构体的典型方法。
let c2 = Color{0: 255, 1: 127, 2: 0};  // 按索引来指定字段。
let c3 = Color{1: 0, ..c2};  // 使用基的字段值来填写结构体的所有其他字段。
```

元组结构体表达式

用圆括号括起字段的结构体表达式构造出来的结构体为元组结构体。 虽然为了完整起见，也把它作为一个特定的（结构体）表达式列在这里，但实际上它等价于执行元组结构体构造器的调用表达式。

例子：

```
struct Position(i32, i32, i32);
Position(0, 0, 0);  // 创建元组结构体的典型方法。
let c = Position;  // `c` 是一个接收 3 个参数的函数。
let pos = c(8, 6, 7);  // 创建一个 `Position` 值。
```

单元结构体表达式

单元结构体表达式只是单元结构体项的路径。也是指向此单元结构体的值的隐式常量。 单元结构体的值也可以用无字段结构体表达式来构造。

例子：

```
struct Gamma;
let a = Gamma;  // Gamma 的值。
let b = Gamma{};  // 和`a`的值完全一样。
```

结构体表达式上的属性

在允许块表达式上的属性存在的那几种表达式上下文中，可以在结构体表达式的左括号后直接使用内部属性。

语法

```
StructExpression :
      StructExprStruct
   | StructExprTuple
   | StructExprUnit

StructExprStruct :
   PathInExpression { InnerAttribute* (StructExprFields | StructBase)? }

StructExprFields :
   StructExprField (, StructExprField)* (, StructBase | ,?)

StructExprField :
      IDENTIFIER
   | (IDENTIFIER | TUPLE_INDEX) : Expression

StructBase :
   .. Expression

StructExprTuple :
   PathInExpression (
      InnerAttribute*
      ( Expression (, Expression)* ,? )?
   )

StructExprUnit : PathInExpression
```

字段访问表达式

字段访问表达式是求取结构体或联合体的字段的内存位置的位置表达式。当操作数可变时，字段访问表达式也是可变的。

字段表达式的语法规则为：一个被称为容器操作数的表达式后跟一个单点号(.)，最后是一个标识符。

字段访问表达式后面不能再紧跟着一个被圆括号封闭起来的逗号分割的表达式列表（这种表示这是一个方法调用表达式）。 因此字段访问表达式不能是函数调用表达式的函数调用者。字段访问表达式代表结构体(struct)或联合体(union)的字段。要调用存储在结构体的字段中的函数，需要在此字段表达式外加上圆括号。

例子：

```
let holds_callable = HoldsCallable { callable: || () };

// 非法: 会被解析为调用 "callable"方法
```

```
// holds_callable.callable();

// 合法
(holds_callable.callable)();
```

自动解引用

如果容器操作数的类型实现了 Deref 或 DerefMut（这取决于该操作数是否为可变），则会尽可能多次地自动解引用(automatically dereferenced)，以使字段访问成为可能。 这个过程也被简称为自动解引用(autoderef)。

字段借用

当发生字段借用时，结构体的各个字段以及对结构体的整体引用都被视为彼此分离的实体。如果结构体没有实现 Drop，同时该结构体又存储在局部变量中，则将各个字段被视为彼此分离的单独实体的逻辑还适用于每个字段的移出(move out)。 但如果对 Box 之外的用户自定义类型执行自动解引用，将各个字段被视为彼此分离的单独实体的逻辑就不适用了。

例子：

```
struct A { f1: String, f2: String, f3: String }
let mut x: A;
let a: &mut String = &mut x.f1; // x.f1 被可变借用
let b: &String = &x.f2;         // x.f2 被不可变借用
let c: &String = &x.f2;         // 可以被再次借用
let d: String = x.f3;           // 从 x.f3 中移出
```

语法

```
FieldExpression :
   Expression . IDENTIFIER
```

方法调用表达式

方法调用由一个表达式（receiver）后跟一个单点(.)、一个表达式路径段和一个圆括号封闭的表达式列表组成。

当方法调用被解析为特定 trait 的关联方法时，如果左侧为确切的已知 self 类型，则静态调度到方法，或者如果左侧表达式是 trait object，则动态调度到方法(由 trait object 上的虚拟表进行调度)。

例子：

```
let pi: Result<f32, _> = "3.14".parse();
let log_pi = pi.unwrap_or(1.0).log(2.72);
```

方法查找

在查找方法调用时，为了调用某个方法，可能会自动对 receiver 做解引用或借用。 这需要比其他函数更复杂的查找流程，因为这可能需要调用许多可能的方法。具体会用到下述步骤：

第一步是构建候选接受者类型的列表。通过重复对 receiver 表达式的类型作解引用或借用，将遇到的每个类型添加到列表中，然后在最后再尝试进行一次 unsized 强制转换(类似数组引用转切片引用或实现了 Trait 的引用转换为 Trait 的引用)，如果成功，则将结果类型也添加到此类型列表里。 然后，再在这个列表中的每个候选类型 T 后紧跟着添加 &T 和 &mut T 候选项。例如，receiver 的类型为 Box<[i32;2]>，则候选类型为 Box<[i32;2]>，&Box<[i32;2]>，&mut Box<[i32;2]>，[i32; 2]（[i32; 2]通过解引用得到），&[i32; 2]，&mut [i32; 2]，[i32]（[i32]通过 unsized 强制转换得到），&[i32]，最后是 &mut [i32]。

第二部，对每个候选类型 T，编译器会它的以下位置上搜索一个可见的同名方法，找到后还会把此方法所属的类型当做 receiver：

- T 的固有方法（直接在 T 上实现的方法）。
- 由 T 已实现的可见的 trait 所提供的任何方法。如果 T 是一个类型参数，则首先查找由 T 上的 trait 约束所提供的方法。然后查找作用域内所有其他的方法。

如果上面这第二步查找时，如果碰到了存在多个可能性方法的情况，比如泛型方法之间或 trait 方法之间被认为是相同的，那么它就会导致编译错误。这些情况就需要使用函数调用消歧语法来为方法调用消除歧义。

注意：查找是按顺序进行的，这有时会导致出现不太符合直觉的结果。 下面的代码将打印 “In trait impl!”，因为首先会查找 &self 上的方法，在找到结构体 Foo 的（接受者类型为）&mut self 的（固有）方法（Foo::bar）之前先找到（接受者类型为 &self 的）trait 方法（Bar::bar）：

```
struct Foo {}

trait Bar {
  fn bar(&self);
```

```
}

impl Foo {
  fn bar(&mut self) {
    println!("In struct impl!")
  }
}

impl Bar for Foo {
  fn bar(&self) {
    println!("In trait impl!")
  }
}

fn main() {
  let mut f = Foo{};
  f.bar();
}
```

语法

```
MethodCallExpression :
   Expression . PathExprSegment (CallParams? )
```

函数调用表达式

函数调用表达式用来调用函数。

函数调用表达式的语法规则为：一个被称作函数操作数的表达式，后跟一个圆括号封闭的逗号分割的被称为参数操作数的表达式列表。 如果函数最终返回，则此调用表达式执行完成。

对于非函数类型，表达式 f(...) 会使用 std::ops::Fn、std::ops::FnMut 或 std::ops::FnOnce 这些 trait 上的某一方法，选择使用哪个要看 f 如何获取其输入的参数，具体就是看是通过引用、可变引用、还是通过获取所有权来获取的， Rust 也会根据需要自动对 f 作解引用或借用处理。

例子：

```
let three: i32 = add(1i32, 2i32);
let name: &'static str = (|| "Rust")();
```

函数调用消歧

为获得更直观的完全限定的句法规则，Rust 对所有函数调都作了糖化(sugar)处理。 根据当前作用域内的程序项调用的二义性，函数调用有可能需要完全限定。

少数几种情况下经常会出现一些导致方法调用或关联函数调用的 receiver 或引用对象不明确的情况。这些情况可包括：

- 作用域内的多个 trait 为同一类型定义了相同名称的方法。
- 自动解引(Auto-deref)用搞不定的情况；例如，区分智能指针本身的方法和指针所指对象上的方法。
- 不带参数的方法，就像 default() 这样的和返回类型的属性(properties)的，如 size_of()。

为了解决这种二义性，可以使用更具体的路径、类型或 trait 来明确指代他们想要的方法或函数。例如：

```
trait Pretty {
    fn print(&self);
}

trait Ugly {
  fn print(&self);
}

struct Foo;
impl Pretty for Foo {
    fn print(&self) {}
}

struct Bar;
impl Pretty for Bar {
    fn print(&self) {}
```

```
}
impl Ugly for Bar {
    fn print(&self) {}
}

fn main() {
    let f = Foo;
    let b = Bar;

    // 我们可以这样做，因为对于`Foo`，我们只有一个名为 `print` 的项
    f.print();
    // 对于 `Foo`来说，这样是更明确了，但没必要
    Foo::print(&f);
    // 如果你不喜欢简洁的话，那，也可以这样
    <Foo as Pretty>::print(&f);

    // b.print(); // 错误： 发现多个 `print`
    // Bar::print(&b); // 仍错： 发现多个 `print`

    // 必要，因为作用域内的多个程序项定义了 `print`
    <Bar as Pretty>::print(&b);
}
```

语法

```
CallExpression :
   Expression ( CallParams? )

CallParams :
   Expression ( , Expression )* ,?
```

闭包表达式

闭包表达式，也被称为 lambda 表达式或 lambda，它定义了一个闭包类型，并把此表达式求值计算为该类型的值。

闭包表达式的语法规则为：先是一个可选的 move 关键字，后跟一对管道定界符(|)封闭的逗号分割的被称为闭包参数的 pattern 列表（每个闭包参数都可选地通过:后跟其类型），再可选地通过 -> 后跟一个返回类型，最后是被称为闭包体操作数的表达式。 代表闭包参数的每个 pattern 后面的可选类型是该 pattern 的类型注解。 如果存在返回类型，则闭包体表达式必须是一个普通的块(表达式)。

闭包表达式本质是将一组参数映射到参数后面的表达式的函数。 与 let 绑定一样，闭包参数也是不可反驳型 pattern 的，其类型注解是可选的，如果没有给出，则从上下文推断。每个闭包表达式都有一个唯一的匿名类型。

捕获模式

编译器默认通过以下顺序进行变量捕获：不可变借用、唯一不可变借用、可变借用、所有权转移。 它将选择允许闭包编译的第一个选项。 仅根据闭包表达式的内容进行选择； 编译器不会考虑周围的代码，例如相关变量的生命周期。

唯一不可变借用是特殊的借用类型，它不能在语言中的任何其他地方使用，也不能显式指定。它发生在闭包中修改可变引用的所指对象时：

```
let mut b = false;
let x = &mut b;
{
    let mut c = || { *x = true; };
    // The following line is an error:
    // let y = &x;
    c();
}
let z = &x;
```

其中 x 是不可变变量，因此只能不可变借用 x，但是不变地借用 x 会使*x = true 赋值非法，因为 & &mut 引用可能不是唯一的(例如将 let y = &x 打开将产生错误)，因此不能安全地用于修改值(在闭包中实际需要使用类似"**y=false"来修改值，而这必须要求 y 拥有&mut &mut 类型，因此只能使用唯一不可变借用的概念来体现可变借用的思想，即生命周期内只存在唯一借用)。使用了唯一的不可变借用：它不可变地借用 x，但就像可变借用一样，它必须是唯一的。而 z 的声明是有效的，因为闭包的生命周期在块结束时已经到期，释放了借用。

move 捕获

如果使用 move 关键字，则所有捕获都是通过 move(所有权转移) 进行的，对于 Copy 类型，则是通过复制，无论借用是否有效。 move 关键字通常用于允许闭包比捕获的值更有效，例如如果闭包正在返回或用于生成新线程。

结构、元组和枚举等复合类型总是被完全捕获，而捕获其中的单个字段，如果仅需要捕获单个字段，则需要借用局部变量：

```
struct SetVec {
    set: HashSet<u32>,
    vec: Vec<u32>
}

impl SetVec {
    fn populate(&mut self) {
        let vec = &mut self.vec;
        self.set.iter().for_each(|&n| {
            vec.push(n);
        })
    }
}
```

闭包的 trait 实现

闭包类型根据其捕获环境的不同，对应实现了不同的 trait：

- Fn：不可变借用。Fn 用于从其环境获取不可变的借用值。
- FnMut：可变借用。FnMut 用于获取可变的借用值，因此其可以改变其环境。
- FnOnce：获取所有权。FnOnce 将消费从周围作用域捕获的变量，获取所有权并在定义闭包时将其移动进闭包。

所有闭包都实现了 FnOnce。没有移动被捕获变量的所有权到闭包内的闭包也实现了 FnMut。不需要对被捕获的变量进行可变访问的闭包则也实现了 Fn。对应的 trait object 类型，可以存储对应的闭包：

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    let y = 2;
    Box::new(move |x| x + y)
}
```

对于捕获闭包，则只能通过 Fn/FnMut/FnOnce 这三个 trait 来引用。而非捕获闭包是不从其环境中捕获任何内容的闭包，它们可以被强制为具有匹配签名的函数指针（例如 fn()）：

```
let add = |x, y| x + y;

let mut x = add(5,7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5,7);
```

所有闭包类型都实现了 Sized。 此外，如果它存储的捕获类型允许，闭包类型实现以下 trait：Clone、Copy、Sync、Send。Send 和 Sync 的规则与普通 struct 类型的规则相匹配，而 Clone 和 Copy 的行为就像派生的一样。 对于 Clone，未指定捕获变量的克隆顺序。由于捕获通常是通过引用进行的，因此出现以下一般规则：

- 如果所有捕获的变量都是 Sync 的，则闭包是 Sync 的。
- 如果非唯一不可变引用捕获的所有变量都是 Sync 的，并且唯一不可变或可变引用 copy 或 move 捕获的所有值都是 send，则闭包是 Send。
- 如果闭包不通过唯一不可变或可变引用捕获任何值，并且通过 copy 或 move 捕获的所有值分别是 Clone 或 Copy，则闭包是 Clone 或 Copy。

闭包参数上的属性

闭包参数上的属性遵循与常规函数参数上相同的规则和限制。

语法

```
ClosureExpression :
   move?
   ( || | | ClosureParameters? | )
   (Expression | -> TypeNoBounds BlockExpression)

ClosureParameters :
   ClosureParam (, ClosureParam)* ,?

ClosureParam :
   OuterAttribute* PatternNoTopAlt ( : Type )?
```

循环表达式

Rust 支持四种循环表达式：

- loop 表达式表示一个无限循环。
- while 表达式不断循环，直到谓词为假。
- while let 表达式循环测试给定 pattern。
- for 表达式从迭代器中循环取值，直到迭代器为空。

所有四种类型的循环都支持 break 表达式、continue 表达式和循环标签(label)。 只有 loop 循环支持对循环体求值。

loop 表达式

loop 表达式会不断地重复地执行它代码体内的代码：loop { println!("I live."); }。

没有包含关联的 break 表达式的 loop 表达式是发散的，并且具有类型 !。 包含相应 break 表达式的 loop 表达式可以结束循环，并且此表达式的类型必须与 break 表达式的类型兼容。

语法

```
InfiniteLoopExpression :
   loop BlockExpression
```

while 表达式

while 循环从对布尔型的循环条件操作数求值开始。如果循环条件操作数的求值结果为 true，则执行循环体块，然后控制流返回到循环条件操作数。如果循环条件操作数的求值结果为 false，则 while 表达式完成。

例子：

```
let mut i = 0;

while i < 10 {
    println!("hello");
    i = i + 1;
}
```

语法

```
PredicateLoopExpression :
   while Expression BlockExpression
```

while let 表达式

while let 循环在语义上类似于 while 循环，但它用 let 关键字后紧跟着一个 pattern、一个 =、一个检验对象表达式和一个块表达式，来替代原来的条件表达式。 如果检验对象表达式的值与 pattern 匹配，则执行循环体块，然后控制流再返回到 pattern 匹配语句。如果不匹配，则 while 表达式执行完成。

与 if let 表达式的情况一样，检验表达式不能是一个懒惰布尔运算符表达式。

while let 循环等价于包含匹配(match)表达式的 loop 表达式。例如：

```
'label: while let PATS = EXPR {
    /* loop body */
}
```

等价于

```
'label: loop {
    match EXPR {
        PATS => { /* loop body */ },
        _ => break,
    }
}
```

例子：

```
let mut x = vec![1, 2, 3];

while let Some(y) = x.pop() {
    println!("y = {}", y);
}

while let _ = 5 {
    println!("不可反驳模式总是会匹配成功");
```

```
    break;
}
```

多 pattern 匹配

可以使用操作符 | 指定多个 pattern。 这与匹配(match)表达式中的 | 具有相同的语义。

例子：

```
let mut vals = vec![2, 3, 1, 2, 2];
while let Some(v @ 1) | Some(v @ 2) = vals.pop() {
    // 打印 2, 2, 然后 1
    println!("{}", v);
}
```

语法

```
PredicatePatternLoopExpression :
   while let Pattern = Expression BlockExpression
```

for 表达式

for 表达式是一个用于在实现了 std::iter::IntoIterator 的某个迭代器提供的元素上进行循环的语法结构。如果迭代器生成一个值，该值将与此 for 表达式提供的不可反驳型 pattern 进行匹配，执行循环体，然后控制流返回到 for 循环的头部。 如果迭代器为空了，则 for 表达式执行完成。

for 循环等价于下面的块表达式。如：

```
'label: for PATTERN in iter_expr {
    /* loop body */
}
```

等价于

```
{
    let result = match IntoIterator::into_iter(iter_expr) {
        mut iter => 'label: loop {
            let mut next;
            match Iterator::next(&mut iter) {
                Option::Some(val) => next = val,
                Option::None => break,
            };
            let PATTERN = next;
            let () = { /* loop body */ };
        },
    };
    result
}
```

上面代码里使用外层 match 来确保 iter_expr 中的任何临时值在循环结束前不会被销毁。 next 先声明后赋值是因为这样能让编译器更准确地推断出类型。

例子：

```
let v = &["apples", "cake", "coffee"];

for text in v {
    println!("I like {}.", text);
}
```

语法

```
IteratorLoopExpression :
   for Pattern in Expression BlockExpression
```

循环 label

循环表达式可以选择设置一个标签。

标签被标记为循环表达式之前的标签，如 'foo: loop { break 'foo; }、'bar: while false {}、'humbug: for _ in 0..0 {}。 如果循环存在标签，则嵌套在该循环中的带此标签的 break 表达式和 continue 表达式可以退出此标签标记的循环层或将控制流返回至此标签标记的

循环层的头部。 具体请参见后面的 break 表达式和 continue 表达式。

语法

```
LoopLabel :
   LIFETIME_OR_LABEL :
```

break 表达式

当遇到 break 时，相关的循环体的执行将立即结束。

例子：

```
let mut last = 0;
for x in 1..100 {
    if x > 12 {
        break;
    }
    last = x;
}
assert_eq!(last, 12);
```

break 多层循环

break 表达式通常与包含 break 表达式的最内层 loop、for 或 while 循环相关联，但是可以使用循环标签来指定受影响的循环层（此循环层必须是封闭该 break 表达式的循环之一）。

例子：

```
'outer: loop {
    while true {
        break 'outer;
    }
}
```

break 和 loop 返回值

当使用 loop 循环时，可以使用 break 表达式从循环中返回一个值，通过形如 break EXPR 或 break 'label EXPR 来返回，其中 EXPR 是一个表达式，它的结果被从 loop 循环中返回。

如果 loop 有关联的 break，则不认为该循环是发散的，并且 loop 表达式的类型必须与每个 break 表达式的类型兼容。 其后不跟表达式的 break 被认为与后跟 () 的 break 表达式的效果相同。

例子：

```
let (mut a, mut b) = (1, 1);
let result = loop {
    if b > 10 {
        break b;
    }
    let c = a + b;
    a = b;
    b = c;
};
// 斐波那契数列中第一个大于 10 的值：
assert_eq!(result, 13);
```

语法

```
BreakExpression :
   break LIFETIME_OR_LABEL? Expression?
```

continue 表达式

当遇到 continue 时，相关的循环体的当前迭代将立即结束，并将控制流返回到循环头。 在 while 循环的情况下，循环头是控制循环的条件表达式。 在 for 循环的情况下，循环头是控制循环的调用表达式。

continue 多层循环

continue 表达式通常与包含 continue 表达式的最内层 loop、for 或 while 循环相关联，但可以使用 continue 'label 来指定受影响的循环层。 continue 表达式只允许在循环体内部使用。

例子：

```
'outer: loop {
    while true {
        continue 'outer;
    }
}
```

语法

```
ContinueExpression :
   continue LIFETIME_OR_LABEL?
```

语法

```
LoopExpression :
   LoopLabel? (
         InfiniteLoopExpression
       | PredicateLoopExpression
       | PredicatePatternLoopExpression
       | IteratorLoopExpression
   )
```

区间表达式

.. 和 ..= 操作符会根据下表中的规则构造 std::ops::Range（或 core::ops::Range）中的某一个 enum 变体类型的对象：

产生式/句法规则	句法	类型	区间语义
RangeExpr	start..end	std::ops::Range	start ⩽ x < end
RangeFromExpr	start..	std::ops::RangeFrom	start ⩽ x
RangeToExpr	..end	std::ops::RangeTo	x < end
RangeFullExpr	..	std::ops::RangeFull	-
RangeInclusiveExpr	start..=end	std::ops::RangeInclusive	start ⩽ x ⩽ end
RangeToInclusiveExpr	..=end	std::ops::RangeToInclusive	x ⩽ end

下面的表达式是等价的：

```
let x = std::ops::Range {start: 0, end: 10};
let y = 0..10;

assert_eq!(x, y);
```

例子：

```
1..2;   // std::ops::Range
3..;    // std::ops::RangeFrom
..4;    // std::ops::RangeTo
..;     // std::ops::RangeFull
5..=6;  // std::ops::RangeInclusive
..=7;   // std::ops::RangeToInclusive
```

用于 for 表达式

区间表达式可用于 for 表达式。

例子：

```
for i in 1..11 {
    println!("{}", i);
}
```

语法

```
RangeExpression :
      RangeExpr
   | RangeFromExpr
   | RangeToExpr
   | RangeFullExpr
   | RangeInclusiveExpr
   | RangeToInclusiveExpr

RangeExpr :
   Expression .. Expression

RangeFromExpr :
   Expression ..

RangeToExpr :
   .. Expression

RangeFullExpr :
   ..

RangeInclusiveExpr :
   Expression ..= Expression

RangeToInclusiveExpr :
   ..= Expression
```

if 表达式

if 表达式是程序控制中的一个条件分支。

if 表达式的语法是一个条件操作数后紧跟一个块，再后面是任意数量的 else if 条件表达式和块，最后是一个可选的尾部 else 块。

条件操作数的类型必须是布尔型。如果条件操作数的求值结果为 true，则执行紧跟的块，并跳过后续的 else if 块或 else 块。 如果条件操作数的求值结果为 false，则跳过紧跟的块，并按顺序求值后续的 else if 条件表达式。 如果所有 if 条件表达式和 else if 条件表达式的求值结果均为 false，则执行 else 块。

if 表达式的求值结果就是所执行的块的返回值，或者如果没有块被求值那 if 表达式的求值结果就是 ()。 if 表达式在所有情况下的类型必须一致。

例子：

```
if x == 4 {
    println!("x is four");
} else if x == 3 {
    println!("x is three");
} else {
    println!("x is something else");
}

let y = if 12 * 15 > 150 {
    "Bigger"
} else {
    "Smaller"
};
assert_eq!(y, "Bigger");
```

语法

```
IfExpression :
   if Expression BlockExpression
   (else ( BlockExpression | IfExpression | IfLetExpression ) )?
```

if let 表达式

if let 表达式在语义上类似于 if 表达式，但是代替条件操作数的是一个关键字 let，再后面是一个 pattern、一个 = 和一个检验对象操作

数。 如果检验对象操作数的值与 pattern 匹配，则执行相应的块。 否则，如果存在 else 块，则继续处理后面的 else 块。和 if 表达式一样，if let 表达式也可以有返回值，这个返回值是由被求值的块确定。

if let 表达式等价于 match 表达式：

```
if let PATS = EXPR {
    /* body */
} else {
    /*else */
}
```

等价于：

```
match EXPR {
    PATS => { /* body */ },
    _ => { /* else */ },    // 如果没有 else 块，这相当于 `()`
}
```

例子：

```
let dish = ("Ham", "Eggs");

// 此主体代码将被跳过，因为该模式被反驳
if let ("Bacon", b) = dish {
    println!("Bacon is served with {}", b);
} else {
    // 这个块将被执行。
    println!("No bacon will be served");
}

// 此主体代码将被执行
if let ("Ham", b) = dish {
    println!("Ham is served with {}", b);
}

if let _ = 5 {
    println!("不可反驳型的模式总是会匹配成功的");
}
```

多 pattern 匹配

可以使用操作符 | 指定多个 pattern。 这与 match 表达式中的 | 具有相同的语义。

例如：

```
enum E {
    X(u8),
    Y(u8),
    Z(u8),
}
let v = E::Y(12);
if let E::X(n) | E::Y(n) = v {
    assert_eq!(n, 12);
}
```

短路布尔表达式检验对象

如果 if let 表达式中的检验对象是短路布尔表达式。需要使用圆括号来实现：

```
if let PAT = ( EXPR && EXPR ) { .. }
if let PAT = ( EXPR || EXPR ) { .. }
```

语法

```
IfLetExpression :
   if let Pattern = Expression BlockExpression
   (else ( BlockExpression | IfExpression | IfLetExpression ) )?
```

match 表达式

match 表达式在 pattern 上建立代码逻辑分支。

match 的确切形式取决于其应用的 pattern。一个 match 表达式带有一个要与 pattern 进行比较的检验对象表达式。检验对象表达式和 pattern 必须具有相同的类型。

根据检验对象表达式是位置表达式或值表达式，匹配(match)的行为表现会有所不同。

- 如果检验对象表达式是一个值表达式，则这个表达式首先会在被求值到一个临时内存位置，然后将这个结果值按顺序与匹配臂(arms)中的 pattern 进行比较，直到找到一个成功的匹配。 第一个匹配成功的 pattern 所在的匹配臂会被选中为当前匹配的分支目标，然后以该 pattern 绑定的变量为中介，（把它从检验对象那里匹配到的变量值）转赋值给该匹配臂的块中的局部变量，然后控制流进入该块。
- 当检验对象表达式是一个位置表达式时，此 match 表达式不用先去内存上分配一个临时位置；但是，按值匹配的绑定方式(by-value binding)会复制或移动这个（位置表达式代表的）内存位置里面的值。

如果可能，最好还是在位置表达式上进行匹配，因为这种匹配的生存期继承了该位置表达式的生存期，而不会（让其生存期仅）局限于此匹配的内部。模式中绑定到的变量的作用域可以覆盖到匹配守卫(match guard)和匹配臂的表达式里。 变量绑定方式（移动、复制或引用）取决于使用的具体模式。

例子：

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

多 pattern 匹配

可以使用操作符 | 连接多个匹配 pattern。 每个 pattern 将按照从左到右的顺序进行测试，直到找到一个成功的匹配。

每个 | 分隔的 pattern 里出现的变量绑定必须出现在匹配臂的所有 pattern 里。 相同名称的绑定变量必须具有相同的类型和相同的变量绑定 pattern。

例子：

```
let x = 9;
let message = match x {
    0 | 1  => "not many",
    2 ..= 9 => "a few",
    _       => "lots"
};

assert_eq!(message, "a few");

// 演示模式匹配顺序。
struct S(i32, i32);

match S(1, 2) {
    S(z @ 1, _) | S(_, z @ 2) => assert_eq!(z, 1),
    _ => panic!(),
}
```

匹配守卫

匹配臂可以接受匹配守卫来进一步改进匹配标准。模式守卫出现在 pattern 的后面，由关键字 if 后面的布尔类型表达式组成。

当 pattern 匹配成功时，将执行匹配守卫表达式。 如果此表达式的计算结果为 true，则此 pattern 将进一步被确认为匹配成功，如果结果为 false，匹配将测试下一个 pattern，包括测试同一匹配臂中运算符 | 分割的后续匹配 pattern。

匹配守卫可以引用绑定在它们前面的 pattern 里的变量。 在对匹配守卫进行计算之前，将对检验对象内部被 pattern 的变量匹配上的那部分进行共享引用。在对匹配守卫进行计算时，访问守卫里的这些变量就会使用这个共享引用。只有当匹配守卫最终计算为 true 时，此共享引用的值才会从检验对象内部移动或复制到相应的匹配臂的变量中。 这使得共享借用可以在守卫内部使用，还不会在守卫不匹配的情况下将值移出检验对象。此外，通过在计算匹配守卫的同时持有共享引用，也可以防止匹配守卫内部意外修改检验对象。

例子：

```
let message = match maybe_digit {
```

```
    Some(x) if x < 10 => process_digit(x),
    Some(x) => process_other(x),
    None => panic!(),
};
```

注意：对于多 pattern 匹配，没执行到一个成功匹配的 pattern，都会执行依次守卫。因此使用操作符 | 的多次匹配可能会导致后跟的匹配守卫必须多次执行的副作用。 例如：

```
let i : Cell<i32> = Cell::new(0);
match 1 {
    1 | _ if { i.set(i.get() + 1); false } => {} // 多次执行了匹配守卫
    _ => {}
}
assert_eq!(i.get(), 2);
```

匹配臂上的属性

在匹配臂上允许使用外部属性，但在匹配臂上只有 cfg、cold 和 lint 检查类属性这些属性才有意义。

在允许块表达式上的属性存在的那几种表达式上下文中，可以在匹配表达式的左括号后直接使用内部属性。

语法

```
MatchExpression :
   match Expressionexcept struct expression {
      InnerAttribute*
      MatchArms?
   }

MatchArms :
   ( MatchArm => ( ExpressionWithoutBlock , | ExpressionWithBlock ,? ) )*
   MatchArm => Expression ,?

MatchArm :
   OuterAttribute* Pattern MatchArmGuard?

MatchArmGuard :
   if Expression
```

await 表达式

await 表达式挂起当前计算，直到给定的 future 准备好生成值。

await 表达式的语法格式为：一个其类型实现了 Future trait 的表达式（此表达式本身被称为 future 操作数）后跟一个 .标记，再后跟一个 await 关键字。

await 表达式仅在异步上下文中才能使用，例如 异步函数(async fn) 或 异步(async)块。

await 表达式具有以下效果：

- 把 future 操作数求值计算到一个 future 类型的 tmp 中；
- 使用 Pin::new_unchecked 固定住(Pin)这个 tmp；
- 然后通过调用 Future::poll 方法对这个固定住的 future 进行轮询，同时将当前任务上下文传递给它；
- 如果轮询(poll)调用返回 Poll::Pending，那么这个 future 就也返回 `Expression

任务上下文是指在对异步上下文本身进行轮询时提供给当前异步上下文的上下文。 因为 await 表达式只能在异步上下文中才能使用，所以此时必须有一些任务上下文可用。

实际上,一个 await 表达式大致相当于如下这个非正规的脱糖过程：

```
match future_operand {
    mut pinned => loop {
        let mut pin = unsafe { Pin::new_unchecked(&mut pinned) };
        match Pin::future::poll(Pin::borrow(&mut pin), &mut current_context) {
            Poll::Ready(r) => break r,
            Poll::Pending => yield Poll::Pending,
        }
    }
}
```

其中，yield 伪代码返回 Poll::Pending，当再次调用时，从该点继续执行。 变量 current_context 是指从异步环境中获取的上下文。

语法

```
AwaitExpression :
   Expression . await
```

return 表达式

return 表达式使用关键字 return 来标识。

对 return 表达式求值会将其参数移动到当前函数调用的指定输出位置，然后销毁当前函数的激活帧，并将控制权转移到此函数的调用帧。

例子：

```
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        return a;
    }
    return b;
}
```

语法

```
ReturnExpression :
   return Expression?
```

位置表达式和值表达式

表达式分为两大类：位置表达式和值表达式。同样，在每个表达式中，操作数可以出现在位置上下文或值上下文中。表达式的计算取决于它自己的类别和它出现的上下文。

值表达式

值表达式是表示实际值的表达式。

值表达式上下文表示此处需要一个值。

位置表达式

位置表达式是返回内存位置的表达式。这些表达式可以是局部变量、静态变量、解引用（*expr）、数组索引表达式（expr[expr]）、字段访问表达式（expr.f）或圆括号位置表达式（(palce expression)）。所有其他表达式都是值表达式。

位置表达式上下文表示此处需要一个内存位置，以下上下文是位置表达式上下文：

- 赋值或复合赋值表达式的左操作数。
- 借用运算符&的操作数，(类似常说的取地址操作)。
- 解引用运算符*的操作数。
- 字段访问表达式的操作数。
- 数组索引表达式的索引操作数。
- 任何隐式借用的操作数。
- let 语句的初始值设定项。
- if let、match 或 while let 表达式的检查者。
- 函数式更新结构表达式的 base。

可变位置表达式

可变位置表达式不仅是一个位置表达式，并且内存位置的内容是可变的。可变位置表达式满足以下条件：

- 可以被赋值。
- 可以被可变借用。
- 可以被隐式的可变借用。
- 可以被绑定到一个具有 ref mut 的 pattern。

以下表达式可以作为可变位置表达式：

- 可变变量(非可变引用变量)
- 可变静态变量
- 临时变量

- 字段访问
- 对*mut T 类型的值的解引用。
- 对&mut T 类型的值的解引用。
- 对实现 DerefMut trait 类型的值的解引用。
- 对实现 IndexMut trait 类型的值的索引访问。

因此只有以上表达式可以被赋值，被可变借用，被可变的隐式借用，被绑定到一个具有 ref mut 的 pattern。

临时变量

在位置表达式上下文中使用值表达式时，会创建一个临时未命名内存位置并初始化为该值，并且表达式计算为该位置而非原先的值。

除非提升为静态变量，否则临时变量的 drop 作用域通常是封闭语句的结尾。

隐式借用

某些表达式会通过隐式借用一个表达式将这个表达式视为位置表达式。以下表达式会对操作数进行隐式借用：

- 方法调用表达式中的左操作数。
- 字段访问表达式中的左操作数。
- 调用表达式中的左操作数。
- 索引表达式的左操作数。
- 复合赋值的左操作数。
- 解引用操作符的操作数。
- 比较操作符的操作数。

例如，可以直接比较两个未定义大小的 slice 是否相等，因为 == 运算符隐式借用了它的操作数：

```
let a: &[i32];
let b: &[i32];
// ...
*a == *b;
// 等价于:
::std::cmp::PartialEq::eq(&*a, &*b);
```

move 与 copy

当位置表达式在值表达式上下文中求值时，或由于 pattern 被约束为一个值，则位置表达式表示该内存位置中保存的值。如果该值的类型实现了 Copy，则该值将被复制。在其余情况下，如果该类型为 Sized，则该值将被 move(所有权转移)。

将值移出位置表达式，将导致该位置被取消初始化，并且在重新初始化之前无法再次读取。只有以下位置表达式可能会发生 move：

- 变量(非引用变量)。
- 临时变量。
- 可以移出且未实现 Drop 的字段访问表达式的值。
- 类型为 Box<T> 的表达式的解引用值，并且该值也可以移出。

在其他情况下，尝试在值表达式上下文中使用位置表达式都是错误的。

表达式优先级

Rust 运算符和表达式的优先顺序如下，从强到弱。 相同优先级的二元运算符按其结合性给出的顺序分组：

运算符/表达式	结合性
Paths	
Method calls	
Field expressions	left to right
Function calls, array indexing	
?	
Unary - * ! & &mut	
as	left to right
* / %	left to right
+ -	left to right
<< >>	left to right
&	left to right
^	left to right

运算符/表达式	结合性
|	left to right
== != < > <= >=	Require parentheses
&&	left to right
||	left to right
.. ..=	Require parentheses
= += -= *= /= %= &= |= ^= <<= >>=	right to left
return break closures	

操作数计算顺序

下面的表达式列表都以相同的方式计算它们的操作数，如列表后面所述。 其他表达式要么不接受操作数，要么按照各自描述有条件地评估：

- 解引用表达式
- 错误传播表达式
- 否定表达式
- 算术和逻辑二元运算符
- 比较运算符
- 类型转换表达式
- 分组表达式
- 数组表达式
- 索引表达式
- 元组表达式
- 元组索引表达式
- 结构体表达式
- 字段访问表达式
- 调用表达式
- 方法调用表达式
- Break 表达式
- Range 表达式
- Return 表达式
- Await 表达式

这些表达式的操作数在应用表达式的效果之前进行评估。 使用多个操作数的表达式按照源代码中的描述从左到右求值。

例子：

```
let mut one_two = vec![1, 2].into_iter();
assert_eq!(
    (1, 2),
    (one_two.next().unwrap(), one_two.next().unwrap())
);
```

表达式的属性

仅在几种特定情况下才允许表达式前的外部属性：

- 在用作语句的表达式之前。
- 数组表达式、元组表达式、调用表达式和元组结构表达式的元素。
- 块表达式的最后一个表达式。

以下表达式前不被允许添加属性：

- range 表达式
- 二元运算符表达式（ArithmeticOrLogicalExpression、ComparisonExpression、LazyBooleanExpression、TypeCastExpression、AssignmentExpression、CompoundAssignmentExpression）。

语法

```
Expression :
      ExpressionWithoutBlock
   | ExpressionWithBlock

ExpressionWithoutBlock :
   OuterAttribute*†
   (
         LiteralExpression
```

```
      | PathExpression
      | OperatorExpression
      | GroupedExpression
      | ArrayExpression
      | AwaitExpression
      | IndexExpression
      | TupleExpression
      | TupleIndexingExpression
      | StructExpression
      | CallExpression
      | MethodCallExpression
      | FieldExpression
      | ClosureExpression
      | ContinueExpression
      | BreakExpression
      | RangeExpression
      | ReturnExpression
      | MacroInvocation
   )

ExpressionWithBlock :
   OuterAttribute*†
   (
         BlockExpression
      | AsyncBlockExpression
      | UnsafeBlockExpression
      | LoopExpression
      | IfExpression
      | IfLetExpression
      | MatchExpression
   )
```

常量求值

常量求值是在编译过程中计算表达式结果的过程。不是所有表达式都可以在编译时求值，只有全部表达式的某个子集可以在编译时求值。

常量表达式

某些形式的表达式（被称为常量表达式）可以在编译时求值。在常量上下文中，常量表达式是唯一允许的表达式，并且总是在编译时求值。在其他地方，比如 let 语句，常量表达式可以在编译时求值，但不能保证总能在此时求值。

如果值必须在编译时求得（例如在常量上下文中），则像数组索引越界或溢出这样的行为都是编译错误。如果不是必须在编译时求值，则这些行为在编译时只是警告，但它们在运行时可能会触发 panic。

下列表达式中，只要它们的所有操作数都是常量表达式，并且求值不会引起任何 Drop::drop 函数的运行，那这些表达式就是常量表达式：

- 字面量。
- 常量泛型参数。
- 指向函数项和常量项的路径。不允许递归地定义常量项。
- 指向静态项的路径。这种路径只允许出现在静态项的初始化器中。
- 元组表达式。
- 数组表达式。
- 结构体表达式。
- 块表达式，包括非安全(unsafe)块。其中包含以下语句：
  - let 语句以及类似这样的不可反驳型模式绑定，包括可变绑定。
  - 赋值表达式
  - 复合赋值表达式
  - 表达式语句
- 字段访问表达式。
- 索引表达式，长度为 usize 的数组索引或切片。
- 区间表达式。
- 未从环境捕获变量的闭包。
- 在整型、浮点型、布尔型(bool)和字符型(char)上做的各种内置运算，包括：取反、算术、逻辑、比较 或 惰性布尔运算。
- 排除借用类型为内部可变借用的共享借用表达式。
- 排除解引用裸指针的解引用操作。
  - 指针到地址的强制转换，
  - 函数指针到地址的强制转换，和

    - 到 trait 对象的非固定尺寸类型强换(unsizing casts)。
- 调用常量函数和常量方法。
- loop 表达式, while 表达式和 while let 表达式。
- if 表达式, if let 表达式 和 match 表达式。

常量上下文

下述位置是常量上下文：
- 数组类型的长度表达式。
- 分号分隔的数组创建形式中的长度表达式。
- 下述表达式的初始化器：
    - 常量项
    - 静态项
    - 枚举判别值
- 常量泛型实参

常量函数

常量函数(const fn)可以在常量上下文中调用。给一个函数加一个常量(const)标志对该函数的任何现有的使用都没有影响，它只限制参数和返回可以使用的类型，并防止在这两个位置上使用不被允许的表达式类型。程序员可以自由地用常量函数去做任何用常规函数能做的事情。

当从常量上下文中调用这类函数时，编译器会在编译时解释该函数。这种解释发生在编译目标环境中，而不是在当前主机环境中。因此，如果是针对一个 32 位目标系统进行编译，那么 usize 就是 32 位，这与在一个 64 位还是在一个 32 位主机环境中进行编译动作无关。

常量函数有各种限制以确保其可以在编译时可被求值。因此，例如，不可以将随机数生成器编写为常量函数。在编译时调用常量函数将始终产生与运行时调用它相同的结果，即使多次调用也是如此。这个规则有一个例外：如果在极端情况下执行复杂的浮点运算，那么可能得到（非常轻微）不同的结果。建议不要让数组长度和枚举判别值/式依赖于浮点计算。

常量上下文有，但常量函数不具备的显著特性有：
- 浮点运算。在常量函数中，处理浮点值就像处理只有 Copy trait 约束的泛型参数一样，不能用它们做任何事，只能复制/移动它们。
- trait 对象(dyn Trait)/动态分发类型。
- 泛型参数上除 Sized 之外的泛型约束。
- 比较裸指针。
- 访问联合体字段。
- 调用 transmute。

以下情况在常量函数中是有可能的，但在常量上下文中则不可能：
- 使用泛型类型和生存期参数。常量上下文只允许有限地使用常量型泛型形参。

patterns

pattern 基于给定数据结构去匹配值，并可选地将变量和这些结构中匹配到的值绑定起来。pattern 也用在变量声明上和函数（包括闭包）的参数上。

例子：

```
if let
    Person {
        car: Some(_),
        age: person_age @ 13..=19,
        name: ref person_name,
        ..
    } = person
{
    println!("{} has a car and is {} years old.", person_name, person_age);
}
```

上例中的 pattern 完成四件事：
- 测试 person 是否在其 car 字段中填充了内容。
- 测试 person 的 age 字段（的值）是否在 13 到 19 之间，并将其值绑定到给定的变量 person_age 上。
- 将对 name 字段的引用绑定到给定变量 person_name 上。
- 忽略 person 的其余字段。其余字段可以有任何值，并且不会绑定到任何变量上。

可用位置

pattern 可用于以下位置：
- let 声明
- 函数和闭包的参数。

- match 表达式
- if let 表达式
- while let 表达式
- for 表达式

解构

pattern 可用于解构结构体(struct)、枚举(enum)和元组。解构将一个值分解成组成它的组件，使用的语法与创建此类值时的几乎相同。

在检验对象表达式的类型为结构体(struct)、枚举(enum)或元组(tuple)的模式中，占位符(_) 代表一个数据字段，而通配符 .. 代表特定变量/变体(variant)的所有剩余字段。当使用字段的名称(而不是字段序号)来解构数据结构时，允许将 fieldname 当作 fieldname: fieldname 的简写形式书写。

例子：

```
match message {
    Message::Quit => println!("Quit"),
    Message::WriteString(write) => println!("{}", &write),
    Message::Move{ x, y: 0 } => println!("move {} horizontally", x),
    Message::Move{ .. } => println!("other move"),
    Message::ChangeColor { 0: red, 1: green, 2: _ } => {
        println!("color change, red: {}, green: {}", red, green);
    }
};
```

可反驳性

当一个 pattern 有可能与它所匹配的值不匹配时，我们就说它是可反驳型的(refutable)。也就是说，不可反驳型(irrefutable)pattern 总是能与它们所匹配的值匹配成功。

例子：

```
let (x, y) = (1, 2);                // "(x, y)" 是一个不可反驳型模式

if let (a, 3) = (1, 2) {            // "(a, 3)" 是可反驳型的, 将不会匹配
    panic!("Shouldn't reach here");
} else if let (a, 4) = (3, 4) {     // "(a, 4)" 是可反驳型的, 将会匹配
    println!("Matched ({}, 4)", a);
}
```

无分界符模式优先级

有几种类型的 pattern 在语法上没有定义分界符，它们包括标识符模式、引用模式和或模式。它们组合在一起时，或模式的优先级总是最低的。这允许我们为将来可能的类型特性保留语法空间，同时也可以减少歧义。

例如，x @ A(..) | B(..) 将导致一个错误，即 x 不是在所有 pattern 中都存在绑定关系。 &A(x) | B(x)将导致不同子模式中的 x 之的类型不匹配。

pattern

字面量模式

字面量模式匹配的值与字面量所创建的值完全相同。由于负数不是字面量，特设定字面量模式也接受字面量前的可选负号，它的作用类似于负号运算符。

字面量模式总是可以反驳型的。

例子：

```
for i in -2..5 {
    match i {
        -1 => println!("It's minus one"),
        1 => println!("It's a one"),
        2|4 => println!("It's either a two or a four"),
        _ => println!("Matched none of the arms"),
    }
}
```

语法

```
LiteralPattern :
      BOOLEAN_LITERAL
    | CHAR_LITERAL
    | BYTE_LITERAL
    | STRING_LITERAL
    | RAW_STRING_LITERAL
    | BYTE_STRING_LITERAL
    | RAW_BYTE_STRING_LITERAL
    | -? INTEGER_LITERAL
    | -? FLOAT_LITERAL
```

标识符模式

标识符模式将它们匹配的值绑定到一个变量上。此标识符在该模式中必须是唯一的。该变量会在作用域中遮蔽任何同名的变量。这种绑定的作用域取决于使用 pattern 的上下文（例如 let 绑定或匹配臂(match arm)1)。

标识符模式只能包含一个标识符（也可能前带一个 mut），它能匹配任何值，并将其绑定到该标识符上。最常见的标识符模式应用场景就是用在变量声明上和用在函数（包括闭包）的参数上。

例子：

```
let mut variable = 10;
fn sum(x: i32, y: i32) -> i32 {
```

子匹配绑定

在不存在子模式的情况下，一个标识符模式会匹配任何值并进行绑定。可以使用 variable @ subpattern 的语法方式，仅在值匹配子模式的时候，才将其绑定到变量。

例子：

```
let x = 2;

match x {
    e @ 1 ..= 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

上例中将值 2 绑定到 e 上（不是整个区间(range)：这里的区间是一个区间子模式)。

如果 @subpattern 是不可反驳型的或未指定 subpattern，则标识符模式是不可反驳型的。

引用绑定

默认情况下，标识符模式里的变量会和匹配到的值的一个拷贝副本绑定，或匹配值自身移动过来和变量完成绑定，具体是使用拷贝语义还是移动语义取决于匹配到的值是否实现了 Copy。也可以通过使用关键字 ref 将变量和值的引用绑定，或者使用 ref mut 将变量和值的可变引用绑定。

之所以需要这种句法，是因为在解构子模式里，操作符 & 不能应用在值的字段上。例如以下内容无效：

```
if let Person { name: &person_name, age: 18..=150 } = value { }
```

要使其有效，请按如下方式编写代码：

```
if let Person { name: ref person_name, age: 18..=150 } = value { }
```

ref 不是被匹配的一部分。这里它唯一的目的就是使变量和匹配值的引用绑定起来，而不是潜在地拷贝或移动匹配到的内容。

路径模式优先于标识符模式。如果给某个标识符指定了 ref 或 ref mut，同时该标识符又遮蔽了某个常量，这会导致错误。

例子：

```
match a {
    None => (),
    Some(value) => (),
}

match a {
    None => (),
    Some(ref value) => (),
}
```

在第一个匹配表达式中，值被拷贝（或移动）(到变量 value 上)。在第二个匹配中，对相同内存位置的引用被绑定到变量上。

绑定方式

基于人类工程学的考虑，为了让引用和匹配值的绑定更容易一些，pattern 会自动选择不同的绑定方式。当引用值与非引用 pattern 匹配时，这将自动地被视为 ref 或 ref mut 绑定方式。非引用 pattern 除上面这种绑定模式外还包括除通配符模式、匹配引用类型的常量路径模式和引用模式以外的所有 pattern。

例子：

```
let x: &Option<i32> = &Some(3);
if let Some(y) = x {
    // y 被转换为`ref y` ，其类型为 &i32
}
```

如果绑定模式中没有显式地包含 ref、ref mut、mut，那么它将使用默认绑定方式来确定如何绑定变量。默认绑定方式以使用移动语义的“移动(move)”方式开始。当匹配一个 pattern 时，编译器对 pattern 从外到内逐层匹配。每次非引用 pattern 和引用匹配上了时，引用都会自动解引用出最后的值，并更新默认绑定方式，再进行最终的匹配。此时引用会将默认绑定方式设置为 ref 方式。可变引用会将 pattern 设置为 ref mut 方式，除非绑定方式已经是 ref 了（在这种情况下它仍然是 ref 方式）。如果自动解引用解出的值仍然是引用，则会重复解引用。

移动语义的绑定方式和引用语义的绑定方式可以在同一个 pattern 中混合使用，这样做会导致绑定对象的部分被移走，并且之后无法再使用该对象。这只适用于类型无法拷贝的情况下。下面的示例中，name 被移出了 person，因此如果再试图把 person 作为一个整体使用，或再次使用 person.name，将会因为部分移出的问题而报错：

```
// 在 `age` 被引用绑定的情况下，`name` 被从 person 中移出
let Person { name, ref age } = person;
```

语法

```
IdentifierPattern :
      ref? mut? IDENTIFIER (@ Pattern ) ?
```

通配符模式

通配符模式能与任何值匹配。常用它来忽略那些无关紧要的值。在其他 pattern 中使用该 pattern 时，它与标识符模式不同，它不会复制、移动或借用它匹配的值。

通配符模式总是不可反驳型的。

例子：

```
let (a, _) = (10, x);   // x 一定会被 _ 匹配上

// 忽略一个函数/闭包参数
let real_part = |a: f64, _: f64| { a };

// 忽略结构体的一个字段
let RGBA{r: red, g: green, b: blue, a: _} = color;

// 能接收带任何值的任何 Some
if let Some(_) = x {}
```

语法

```
WildcardPattern :
   _
```

路径模式

路径模式是指向常量值或指向没有字段的结构体或没有字段的枚举变体的 pattern。

非限定路径模式可以指向：

- 枚举变体
- 结构体
- 常量
- 关联常量

限定路径模式只能指向关联常量。

常量不能是 union 类型。结构体常量和枚举常量必须带有 #[derive(PartialEq, Eq)] 属性（不只是实现）。

当路径模式指向结构体或枚举变体(枚举只有一个变体)或类型为不可反驳型的常量时，该路径模式是不可反驳型的。当路径模式指向的是可反驳型常量或带有多个变体的枚举时，该路径模式是可反驳型的。

语法

```
PathPattern :
      PathInExpression
    | QualifiedPathInExpression
```

剩余模式

剩余模式是匹配长度可变的 pattern，它匹配之前之后没有匹配的零个或多个元素。它只能在元组模式、元组结构体模式和切片模式中使用，并且只能作为这些 pattern 中的一个元素出现一次。当作为标识符模式的子模式时，它也可出现在切片模式里。

剩余模式总是不可反驳型的。

例子：

```
match slice {
    [] => println!("slice is empty"),
    [one] => println!("single element {}", one),
    [head, tail @ ..] => println!("head={} tail={:?}", head, tail),
}

match slice {
    // 忽略除最后一个元素以外的所有元素，并且最后一个元素必须是 "!".
    [.., "!"] => println!("!!!"),

    // `start` 是除最后一个元素之外的所有元素的一个切片，最后一个元素必须是 “z”。
    [start @ .., "z"] => println!("starts with: {:?}", start),

    // `end` 是除第一个元素之外的所有元素的一个切片，第一个元素必须是 “a”
    ["a", end @ ..] => println!("ends with: {:?}", end),

    rest => println!("{:?}", rest),
}

if let [.., penultimate, _] = slice {
    println!("next to last is {}", penultimate);
}

// 剩余模式也可是在元组和元组结构体模式中使用。
match tuple {
    (1, .., y, z) => println!("y={} z={}", y, z),
    (.., 5) => println!("tail must be 5"),
    (..) => println!("matches everything else"),
}
```

语法

```
RestPattern :
   ..
```

区间模式

区间模式匹配在其上下边界定义的封闭区间内的值。例如，一个 pattern 'm'..='p' 将只能匹配值'm'、'n'、'o' 和 'p'。它的边界值可以是字面量，也可以是指向常量值的路径。

区间模式只适用于标量类型。可接受的类型有：

- 整型（u8、i8、u16、i16、usize、isize ...）。
- 字符型（char）。
- 浮点类型（f32 和 f64）。这已被弃用，在未来版本的 Rust 中将不可用。

一个 pattern a ..= b 必须总是有 a ≤ b。例如，10..=0 这样的区间模式是错误的。

保留 ...语法只是为了向后兼容。

当区间模式匹配某（非 usize 和 非 isize）整型类型和字符型的整个值域时，此 pattern 是不可反驳型的。

例子：

```
let valid_variable = match c {
    'a'..='z' => true,
    'A'..='Z' => true,
    'α'..='ω' => true,
    _ => false,
};

println!("{}", match ph {
    0..=6 => "acid",
    7 => "neutral",
    8..=14 => "base",
    _ => unreachable!(),
});

// 使用指向常量值的路径：
println!("{}", match altitude {
    TROPOSPHERE_MIN..=TROPOSPHERE_MAX => "troposphere",
    STRATOSPHERE_MIN..=STRATOSPHERE_MAX => "stratosphere",
    MESOSPHERE_MIN..=MESOSPHERE_MAX => "mesosphere",
    _ => "outer space, maybe",
});

if let size @ binary::MEGA..=binary::GIGA = n_items * bytes_per_item {
    println!("这适用并占用{}个字节", size);
}

// 使用限定路径：
println!("{}", match 0xfacade {
    0 ..= <u8 as MaxValue>::MAX => "fits in a u8",
    0 ..= <u16 as MaxValue>::MAX => "fits in a u16",
    0 ..= <u32 as MaxValue>::MAX => "fits in a u32",
    _ => "too big",
});
```

语法

```
RangePattern :
   RangePatternBound ..= RangePatternBound

ObsoleteRangePattern :
   RangePatternBound ... RangePatternBound

RangePatternBound :
      CHAR_LITERAL
    | BYTE_LITERAL
    | -? INTEGER_LITERAL
    | -? FLOAT_LITERAL
    | PathInExpression
    | QualifiedPathInExpression
```

引用模式

引用模式对当前匹配的指针做解引用，从而能借用它们。引用模式用于在匹配值是指针类型时，匹配其指向的值。

例如，下面 x: &i32 上的两个匹配是等效的：

```
let int_reference = &3;

let a = match *int_reference { 0 => "zero", _ => "some" };
let b = match int_reference { &0 => "zero", _ => "some" };

assert_eq!(a, b);
```

对于引用的引用，必须使用&&作为引用模式。&& 本身是一个单独的 token。例子：

```
let a = Some(&&10);
match a {
   Some( &&value ) => println!("{}", value),
   None => {}
}
```

引用模式中添加关键字 mut 可对可变引用做解引用。引用模式中的可变性标记必须与作为匹配对象的那个引用的可变性匹配。

引用模式总是不可反驳型的。

语法

```
ReferencePattern :
   (&|&&) mut? PatternWithoutRange
```

结构体模式

结构体模式匹配与子模式定义的所有条件匹配的结构体值。它也被用来解构结构体。

在结构体模式中，结构体字段需通过名称、索引（对于元组结构体来说）来指代，或者通过使用 .. 来忽略：

```
match s {
    Point {x: 10, y: 20} => (),
    Point {y: 10, x: 20} => (),     // 顺序没关系
    Point {x: 10, ..} => (),
    Point {..} => (),
}

match t {
    PointTuple {0: 10, 1: 20} => (),
    PointTuple {1: 10, 0: 20} => (),    // 顺序没关系
    PointTuple {0: 10, ..} => (),
    PointTuple {..} => (),
}
```

如果没使用 .. ，需要提供所有字段的详尽匹配：

```
match struct_value {
    Struct{a: 10, b: 'X', c: false} => (),
    Struct{a: 10, b: 'X', ref c} => (),
    Struct{a: 10, b: 'X', ref mut c} => (),
    Struct{a: 10, b: 'X', c: _} => (),
    Struct{a: _, b: _, c: _} => (),
}
```

ref 和/或 mut IDENTIFIER 这样的语法格式能匹配任意值，并将其绑定到与给定字段同名的变量上。例如：

```
let Struct{a: x, b: y, c: z} = struct_value;          // 解构所有的字段
```

当一个结构体模式的子模式是可反驳型的，那这个结构体模式就是可反驳型的。

语法

```
StructPattern :
   PathInExpression {
      StructPatternElements ?
   }

StructPatternElements :
      StructPatternFields (, | , StructPatternEtCetera)?
   | StructPatternEtCetera

StructPatternFields :
   StructPatternField (, StructPatternField) *

StructPatternField :
   OuterAttribute *
```

```
   (
         TUPLE_INDEX : Pattern
       | IDENTIFIER : Pattern
       | ref? mut? IDENTIFIER
   )

StructPatternEtCetera :
   OuterAttribute *
   ..
```

元组模式

元组模式匹配与子模式定义的所有条件匹配的元组值。它们还被用来解构元组值。

内部只带有一个剩余模式的元组语法形式（..）是一种内部不需要逗号分割的特殊匹配形式，它可以匹配任意长度的元组。

当元组模式的一个子模式是可反驳型的，那该元组模式就是可反驳型的。

例子：

```
let pair = (10, "ten");
let (a, b) = pair;

assert_eq!(a, 10);
assert_eq!(b, "ten");
```

语法

```
TupleStructPattern :
   PathInExpression ( TupleStructItems? )

TupleStructItems :
   Pattern ( , Pattern )* ,?
```

切片模式

切片模式可以匹配固定长度的数组和动态长度的切片。

在匹配数组时，只要每个元素是不可反驳型的，切片模式就是不可反驳型的。例子：

```
// 固定长度
let arr = [1, 2, 3];
match arr {
    [1, _, _] => "从 1 开始",
    [a, b, c] => "从其他值开始",
};
```

当匹配切片时，只有单个 .. 剩余模式或带有 ..（剩余模式）作为子模式的标识符模式的情况才是不可反驳型的。例子：

```
// 动态长度
let v = vec![1, 2, 3];
match v[..] {
    [a, b] => { /* 这个匹配臂不适用，因为长度不匹配 */ }
    [a, b, c] => { /* 这个匹配臂适用 */ }
    _ => { /* 这个通配符是必需的，因为长度不是编译时可知的 */ }
};
```

语法

```
SlicePattern :
   [ SlicePatternItems? ]

SlicePatternItems :
   Pattern (, Pattern)* ,?
```

分组模式

将 pattern 括在圆括号内可用来显式控制复合模式的优先级。例如，像 &0..=5 这样的引用模式和区间模式相邻就会引起歧义，这时可以用圆括号来消除歧义：

```
let int_reference = &3;
match int_reference {
    &(0..=5) => (),
    _ => (),
}
```

语法

```
GroupedPattern :
   ( Pattern )
```

或模式

或模式是能匹配两个或多个并列子模式（例如：A | B | C）中的一个的 pattern。此 pattern 可以任意嵌套。

除了 let 绑定和函数参数（包括闭包参数）中的 pattern（此时句法上使用 _PatternNoTopAlt_产生式），或 pattern 在语法上允许在任何其他 pattern 出现的地方出现（这些 pattern 语法上使用 _Pattern_产生式）。

静态语义

假定在某个代码深度上给定任意模式 p 和 q，现假定它们组成模式 p | q，则以下情况会导致这种组成的非法：

- 从 p 推断出的类型和从 q 推断出的类型不一致，或
- p 和 q 引入的绑定标识符不一样，或
- p 和 q 中引入的同名绑定标识符的类型和绑定模式中的类型不一致。

前面提到的所有实例中的类型都必须是精确的，隐式的类型强转在这里不适用。

当对表达式 match e_s { a_1 => e_1, ... a_n => e_n } 做类型检查时，假定在 e_s 内部深度为 d 的地方存一个表达式片段，那对于此片段，每一个匹配臂 a_i 都包含了一个 p_i | q_i 来与此段内容进行匹配，但如果表达式片段的类型与 p_i | q_i 的类型不一致，则该模式 p_i | q_i 被认为是格式错误的。

为了遵从匹配模式的穷尽性检查，模式 p | q 被认为同时覆盖了 p 和 q。对于某些构造器 c(x, ..) 来说，此时应用分配律能使 c(p | q, ..rest) 与 c(p, ..rest) | c(q, ..rest) 覆盖相同的一组匹配值。这个规律可以递归地应用，直到不再有形式为 p | q 的嵌套模式。

动态语义

检查对象表达式 e_s 与深度为 d 的模式 c(p | q, ..rest)（这里 c 是某种构造器，p 和 q 是任意的模式，rest 是 c 构造器的任意的可选因子）进行匹配的动态语义与此表达式与 c(p, ..rest) | c(q, ..rest) 进行匹配的语法定义相同。

语法

```
Pattern :
      |? PatternNoTopAlt ( | PatternNoTopAlt )*

PatternNoTopAlt :
      PatternWithoutRange
    | RangePattern

PatternWithoutRange :
      LiteralPattern
    | IdentifierPattern
    | WildcardPattern
    | RestPattern
    | ObsoleteRangePattern
    | ReferencePattern
    | StructPattern
    | TupleStructPattern
    | TuplePattern
    | GroupedPattern
    | SlicePattern
    | PathPattern
    | MacroInvocation
```

属性

属性是一种通用的、格式自由的元数据(free-form metadatum)，这种元数据会被编译器依据名称、约定、语言和编译器版本进行解释。

内部属性以 #! 开头的方式编写，应用于它在其中声明的项。
外部属性以不后跟感叹号的(!)的 # 开头的方式编写，应用于属性后面的内容。

属性由指向属性的路径和路径后跟的可选的带定界符的 token 树（其解释由属性定义）组成。除了宏属性之外，其他属性的输入也允许使用等号(=)后跟字面量表达式的格式。更多细节见下面的元项属性语法。

属性可以分为以下几类：
- 内置属性
- 宏属性
- 派生宏辅助属性
- 外部工具属性

例子：

```
// 应用于当前模块或 crate 的一般性元数据。
#![crate_type = "lib"]

// 标记为单元测试的函数
#[test]
fn test_foo() {
    /* ... */
}

// 一个条件编译模块
#[cfg(target_os = "linux")]
mod bar {
    /* ... */
}

// 用于静音 lint 检查后报告的告警和错误提醒
#[allow(non_camel_case_types)]
type int8_t = i8;

// 适用于整个函数的内部属性
fn some_unused_variables() {
  #![allow(unused_variables)]

  let x = ();
  let y = ();
  let z = ();
}
```

属性应用场景

属性可以应用于语言中的许多场景：
- 所有的项声明都可接受外部属性，同时外部块、函数、实现和模块都可接受内部属性。
- 大多数语句都可接受外部属性（参见表达式属性，了解表达式语句的限制）。
- 块表达式也可接受外部属性和内部属性，但只有当它是另一个表达式语句的外层表达式时，或是另一个块表达式的最终表达式时才有效。
- 枚举变体和结构体、联合体的字段可接受外部属性。
- 匹配表达式的匹配臂可接受外部属性。
- 泛型生命周期参数或类型参数可接受外部属性。
- 表达式在有限的情况下可接受外部属性，详见表达式属性。
- 函数、闭包和函数指针的参数可接受外部属性。这包括函数指针和外部块中用 ... 表示的可变参数上的属性。

元项属性语法

元项是遵循 Attr 产生式的语法，Rust 的大多数内置属性都使用了此语法。它有以下语法格式：

```
MetaItem :
      SimplePath
   | SimplePath = Expression
   | SimplePath ( MetaSeq? )

MetaSeq :
   MetaItemInner ( , MetaItemInner )* ,?

MetaItemInner :
      MetaItem
```

```
    | Expression
```

元项中的字面量表达式不能包含整型或浮点类型的后缀。

各种内置属性使用元项语法的不同子集来指定它们的输入。下面的语法规则展示了一些常用的使用形式：

```
MetaWord:
   IDENTIFIER

MetaNameValueStr:
   IDENTIFIER = (STRING_LITERAL | RAW_STRING_LITERAL)

MetaListPaths:
   IDENTIFIER ( ( SimplePath (, SimplePath)* ,? )? )

MetaListIdents:
   IDENTIFIER ( ( IDENTIFIER (, IDENTIFIER)* ,? )? )

MetaListNameValueStr:
   IDENTIFIER ( ( MetaNameValueStr (, MetaNameValueStr)* ,? )? )
```

元项语法的一些例子是：

形式	示例
MetaWord	no_std
MetaNameValueStr	doc = "example"
MetaListPaths	allow(unused, clippy::inline_always)
MetaListIdents	macro_use(foo, bar)
MetaListNameValueStr	link(name = "CoreFoundation", kind = "framework")

活跃属性和惰性属性

属性要么是活跃的，要么是惰性的。在属性处理过程中，活跃属性将自己从它们所在的对象上移除，而惰性属性依然保持原位置不变。

cfg 和 cfg_attr 属性是活跃的。test 属性在为测试所做的编译形式中是惰性的，在其他编译形式中是活跃的。宏属性是活跃的。所有其他属性都是惰性的。

外部工具属性

编译器可能允许和具体外部工具相关联的属性，但这些工具在编译和检查过程中必须存在并驻留在编译器提供的工具类预导入包下对应的命名空间中（才能让这些属性生效）。这种属性的（命名空间）路径的第一段是工具的名称，后跟一个或多个工具自己解释的附加段。

当工具在编译期不可用时，该工具的属性将被静默接受而不提示警告。当工具可用时，该工具负责处理和解释这些属性。

如果使用了 no_implicit_prelude 属性，则外部工具属性不可用。

注意：rustc 目前能识别的工具是 “clippy” 和 “rustfmt”。

例子：

```
// 告诉 rustfmt 工具不要格式化以下元素。
#[rustfmt::skip]
struct S {
}

// 控制 clippy 工具的“圈复杂度(cyclomatic complexity)”极限值。
#[clippy::cyclomatic_complexity = "100"]
pub fn f() {}
```

测试类属性

以下属性用于指定函数来执行测试。在“测试”模式下编译 crate 可以构建测试函数以及构建用于执行测试（函数）的测试套件。启用测试模式还会启用 test 条件编译选项。

test 属性

test 属性标记一个用来执行测试的函数，这些函数只在测试模式下编译。测试模式是通过将 --test 参数选项传递给 rustc 或使用 cargo test 来启用的。

测试函数必须是自由函数和单态函数，不能有参数，返回类型必须是以下类型之一：

- ()。返回 () 的测试只要结束且没有触发 panic 就会通过。
- Result<(), E> where E: Error。返回 Result<(), E> 的测试只要它们返回 Ok(()) 就算通过。不结束的测试既不通过也不失败。

例子：

```
#[test]
fn test_the_thing() -> io::Result<()> {
    let state = setup_the_thing()?; // 预期成功
    do_the_thing(&state)?;          // 预期成功
    Ok(())
}
```

ignore 属性

被 test 属性标注的函数也可以被 ignore 属性标注。ignore 属性告诉测试套件不要将该函数作为测试执行。但在测试模式下，这类函数仍然会被编译。

rustc 的测试套件支持使用 --include-ignored 参数选项来强制运行那些被忽略测试的函数。

ignore 属性可以选择使用 MetaNameValueStr 元项属性句法来说明测试被忽略的原因。例如：

```
#[test]
#[ignore = "not yet implemented"]
fn mytest() {
    // ...
}
```

should_panic 属性

被 test 属性标注并返回 () 的函数也可以被 should_panic 属性标注。should_panic 属性使测试函数只有在实际发生 panic 时才算通过。

should_panic 属性可选输入一条出现在 panic 消息中的字符串。如果在 panic 消息中找不到该字符串，则测试将失败。可以使用 MetaNameValueStr 元项属性句法或带有 expected 字段的 MetaListNameValueStr 元项属性句法来传递字符串。

例子：

```
#[test]
#[should_panic(expected = "值未匹配上")]
fn mytest() {
    assert_eq!(1, 2, "值未匹配上");
}
```

派生属性

derive 属性

derive 属性允许为数据结构自动生成新的项。它使用 MetaListPaths 元项属性语法为项指定一系列要实现的 trait 或指定要执行的派生宏的路径。

例如，下面的派生属性将为结构体 Foo 创建一个实现 PartialEq trait 和 Clone trait 的实现(impl item)，类型参数 T 将被派生出的实现(impl)加上 PartialEq 或 Clone 约束：

```
#[derive(PartialEq, Clone)]
struct Foo<T> {
    a: i32,
    b: T,
}
```

上面代码为 PartialEq 生成的实现(impl)等价于

```
impl<T: PartialEq> PartialEq for Foo<T> {
    fn eq(&self, other: &Foo<T>) -> bool {
        self.a == other.a && self.b == other.b
    }

    fn ne(&self, other: &Foo<T>) -> bool {
        self.a != other.a || self.b != other.b
    }
}
```

可以通过过程宏为自定义的 trait 实现自动派生(derive)功能。

automatically_derived 属性

automatically_derived 属性会被自动添加到由 derive 属性为一些内置 trait 自动派生的实现中。它对派生出的实现没有直接影响，但是工具和诊断 lint 可以使用它来检测这些自动派生的实现。

诊断属性

诊断属性用于在编译期间控制或生成诊断消息。

lint 检查类属性

lint 检查系统命名了一些潜在的不良编码模式，这些被命名的 lint 检查就是一个一个的 lint，例如编写了不可能执行到的代码，就被命名为 unreachable-code lint，编写未提供文档的代码就被命名为 missing_docs lint。可以通过 rustc -W help 找到所有 rustc 支持的 lint，以及它们的默认设置。

allow、warn、deny 和 forbid 这些能调整代码检查级别的属性被称为 lint 级别属性，它们使用 MetaListPaths 元项属性语法来指定接受此级别的各种 lint。代码实体应用了这些带了具体 lint 列表的 lint 级别属性，编译器或相关代码检查工具就可以结合这两层属性对这段代码执行相应的代码检查和检查报告。

对任何名为 C 的 lint 来说：

- allow(C) 会压制对 C 的检查，那这样的违规行为就不会被报告，
- warn(C) 警告违反 C 的，但继续编译。
- deny(C) 遇到违反 C 的情况会触发编译器报错(signals an error)，
- forbid(C) 与 deny(C) 相同，但同时会禁止以后再更改 lint 级别，

Lint 属性可以覆盖上一个属性指定的级别，但该级别不能更改 forbid 的 Lint。这里的上一个属性是指语法树中更高级别的属性，或者是同一实体上的前一个属性，按从左到右的源代码顺序列出。

注意：rustc 允许在命令行上设置 lint 级别，也支持在 setting caps 中设置 lint 报告中的 caps。

例子：

```
#[warn(missing_docs)]
pub mod m2{
    #[allow(missing_docs)]
    pub mod nested {
        // 这里忽略未提供文档的编码行为
        pub fn undocumented_one() -> i32 { 1 }

        // 尽管上面允许，但未提供文档的编码行为在这里将触发编译器告警
        #[warn(missing_docs)]
        pub fn undocumented_two() -> i32 { 2 }
    }

    // 未提供文档的编码行为在这里将触发编译器告警
    pub fn undocumented_too() -> i32 { 3 }
}
```

lint 分组

lint 被组织成不同命名的组，以便相关 lint 的等级可以一起调整。使用命名组相当于给出该组中的 lint。例如：

```
// 这允许 "unused"组下的所有 lint。
#[allow(unused)]
// 这个禁止动作将覆盖前面 "unused"组中的 "unused_must_use" lint。
#[deny(unused_must_use)]
fn example() {
    // 这里不会生成告警，因为 "unused_variables" lint 在 "unused"组下
    let x = 1;
    // 这里会产生报错信息，因为 "unused_must_use" lint 被标记为"deny"，而这行的返回结果未被使用
    std::fs::remove_file("some_file"); // ERROR: unused `Result` that must be used
}
```

工具类 lint 属性

可以为 allow、warn、deny 或 forbid 这些调整代码检查级别的 lint 属性添加/输入基于特定工具的 lint。注意该工具在当前作用域内可

用才会起效。

工具类 lint 只有在相关工具处于活动状态时才会做相应的代码模式检查。如果一个 lint 级别属性，如 allow，引用了一个不存在的工具类 lint，编译器将不会去警告不存在该 lint 类，只有使用了该工具（，才会报告该 lint 类不存在）。

注意：rustc 当前识别的工具类 lint 有 "clippy" 和 "rustdoc"。

例子：

```
// 将 clippy 的整个 `pedantic` lint 组设置为告警级别
#![warn(clippy::pedantic)]
// 使来自 clippy 的 `filter_map` lint 的警告静音
#![allow(clippy::filter_map)]

fn main() {
    // ...
}

// 使 clippy 的 `cmp_nan` lint 静音，仅用于此函数
#[allow(clippy::cmp_nan)]
fn foo() {
    // ...
}
```

deprecated 属性

deprecated 属性将项标记为已弃用。rustc 将对被 #[deprecated] 限定的程序项的行为发出警告。rustdoc 将特别展示已弃用的程序项，包括展示 since 版本和 note 提示（如果可用）。

deprecated 属性有几种形式：

- deprecated — 发布一个通用的弃用消息。
- deprecated = "message" — 在弃用消息中包含给定的字符串。
- MetaListNameValueStr 元项属性语法形式里带有两个可选字段：
  - since — 指定项被弃用时的版本号。rustc 目前不解释此字符串，但是像 Clippy 这样的外部工具可以检查此值的有效性。
  - note — 指定一个应该包含在弃用消息中的字符串。这通常用于提供关于不推荐的解释和推荐首选替代。

deprecated 属性可以应用于任何项，trait 项，枚举变体，结构体字段，外部块项，或宏定义。它不能应用于 trait 实现项。当应用于包含其他项的项时，如模块或实现，所有子项都继承此 deprecation 属性。

例子：

```
#[deprecated(since = "5.2", note = "foo was rarely used. Users should instead use bar")]
pub fn foo() {}

pub fn bar() {}
```

must_use 属性

must_use 属性用于在值未被“使用”时发出诊断告警。它可以应用于用户定义的复合类型(结构体(struct)、枚举(enum) 和 联合体(union))、函数和 trait。

must_use 属性可以使用 MetaNameValueStr 元项属性句法添加一些附加消息，如 #[must_use = "example message"]。该字符串将出现在告警消息里。

当用户定义的复合类型上使用了此属性，如果有该类型的表达式正好是表达式语句的表达式，那就违反了 unused_must_use 这个 lint。如：

```
#[must_use]
struct MustUse {
    // 一些字段
}

// 违反 `unused_must_use` lint。
MustUse::new();
```

当函数上使用了此属性，如果此函数被当作表达式语句的表达式的调用表达式，那就违反了 unused_must_use lint。例如：

```
#[must_use]
fn five() -> i32 { 5i32 }

// 违反 unused_must_use lint。
```

```
five(); // 返回值未使用
```

当 trait 声明中使用了此属性，如果表达式语句的调用表达式返回了此 trait 的 trait impl，则违反了 unused_must_use lint。例如：

```
#[must_use]
trait Critical {}
impl Critical for i32 {}

fn get_critical() -> impl Critical {
    4i32
}

// 违反 `unused_must_use` lint。
get_critical();
```

当 trait 声明中的一函数上使用了此属性时，如果调用表达式是此 trait 的某个实现中的该函数时，该行为同样违反 unused_must_use lint。例如：

```
trait Trait {
    #[must_use]
    fn use_me(&self) -> i32;
}

impl Trait for i32 {
    fn use_me(&self) -> i32 { 0i32 }
}

// 违反 `unused_must_use` lint。
5i32.use_me();
```

当在 trait 实现里的函数上使用 must_use 属性时，此属性将被忽略。

代码生成属性

下述属性用于控制代码生成。

inline 属性

inline 属性给出了某种代码生成方式的提示建议，这种方式可能比没有此提示时更快。这些属性只是提示，可能会被忽略。

inline 属性可以在函数上使用。当这类属性应用于 trait 中的函数上时，它们只在那些没有被 trait 实现所覆盖的默认函数上生效，而不是所有 trait 实现中用到的函数上都生效。inline 属性对 trait 中那些没有函数体的函数没有影响。

inline 属性的意义是暗示在调用者(caller)中放置此（属性限定的）函数的副本，而不是在定义此（属性限定的）函数的地方生此函数的代码，然后去让别处代码来调用此函数。

使用内联(inline)属性有三种方法：

- #[inline] 暗示执行内联扩展。
- #[inline(always)] 暗示应该一直执行内联扩展。
- #[inline(never)] 暗示应该从不执行内联扩展。

注意：rustc 编译器会根据启发式算法(internal heuristics)1 自动内联函数。不正确的内联函数会使程序变慢，所以应该小心使用此属性。

cold 属性

cold 属性给出了某种代码生成方式的提示建议，这种方式可能比没有此提示时更快。这些属性只是提示，可能会被忽略。

cold 属性可以在函数上使用。当这类属性应用于 trait 中的函数上时，它们只在那些没有被 trait 实现所覆盖的默认函数上生效，而不是所有 trait 实现中用到的函数上都生效。cold 属性对 trait 中那些没有函数体的函数没有影响。

cold 属性 暗示此（属性限定的）函数不太可能被调用。

no_builtins 属性

no_builtins 属性可以应用在 crate 级别，用以禁用对假定存在的库函数调用的某些代码模式优化。

target_feature 属性

target_feature 属性可应用于非安全(unsafe)函数上，用来为特定的平台架构特性启用该函数的代码生成功能。

它使用 MetaListNameValueStr 元项属性语法来启用（该平台支持的）特性，但这次要求这个语法里只能有一个 enable 键，其对应值是一个逗号分隔的由平台特性名字组成的符串。例如：

```
#[target_feature(enable = "avx2")]
unsafe fn foo_avx2() {}
```

每个目标架构都有一组可以被启用的特性。为不是当前 crate 的编译目标下的 CPU 架构指定需启用的特性是错误的。

调用一个编译时启用了某特性的函数，但当前程序运行的平台并不支持该特性，那这将导致出现未定义行为。

应用了 target_feature 的函数不会内联到不支持给定特性的上下文中。#[inline(always)] 属性不能与 target_feature 属性一起使用。

下面是 x86 or x86_64 可用特性列表：

特性	隐式启用	描述	中文描述
aes	sse2	AES — Advanced Encryption Standard	高级加密标准
avx	sse4.2	AVX — Advanced Vector Extensions	高级矢量扩展指令集
avx2	avx	AVX2 — Advanced Vector Extensions 2	高级矢量扩展指令集 2
bmi1		BMI1 — Bit Manipulation Instruction Sets	位操作指令集
bmi2		BMI2 — Bit Manipulation Instruction Sets 2	位操作指令集 2
fma	avx	FMA3 — Three-operand fused multiply-add	三操作乘加指令
fxsr		fxsave and fxrstor — Save and restore x87 FPU, MMX Technology, and SSE State	保存/恢复 x87 FPU、MMX 技术，SSE 状态
lzcnt		lzcnt — Leading zeros count	前导零计数
pclmulqdq	sse2	pclmulqdq — Packed carry-less multiplication quadword	压缩的四字（16 字节）无进位乘法，主用于加解密处理
popcnt		popcnt — Count of bits set to 1	位 1 计数，即统计有多少个“为 1 的位”
rdrand		rdrand — Read random number	从芯片上的硬件随机数生成器中获取随机数
rdseed		rdseed — Read random seed	从芯片上的硬件随机数生成器中获取为伪随机数生成器设定的种子
sha	sse2	SHA — Secure Hash Algorithm	安全哈希算法
sse		SSE — Streaming SIMD Extensions	单指令多数据流扩展指令集
sse2	sse	SSE2 — Streaming SIMD Extensions 2	单指令多数据流扩展指令集 2
sse3	sse2	SSE3 — Streaming SIMD Extensions 3	单指令多数据流扩展指令集 3
sse4.1	ssse3	SSE4.1 — Streaming SIMD Extensions 4.1	单指令多数据流扩展指令集 4.1
sse4.2	sse4.1	SSE4.2 — Streaming SIMD Extensions 4.2	单指令多数据流扩展指令集 4.2
ssse3	sse3	SSSE3 — Supplemental Streaming SIMD Extensions 3	增补单指令多数据流扩展指令集 3
xsave		xsave — Save processor extended states	保存处理器扩展状态
xsavec		xsavec — Save processor extended states with compaction	压缩保存处理器扩展状态
xsaveopt		xsaveopt — Save processor extended states optimized	xsave 指令集的优化版
xsaves		xsaves — Save processor extended states supervisor	保存处理器扩展状态监视程序

请参阅 target_feature-条件编译选项，了解如何基于编译时的设置来有选择地启用或禁用对某些代码的编译。注意，条件编译选项不受 target_feature 属性的影响，只是被整个 crate 启用的特性所驱动。

有关 x86 平台上的运行时特性检测，请参阅标准库中的 is_x86_feature_detected 宏。

注意：rustc 为每个编译目标和 CPU 启用了一组默认特性。编译时，可以使用命令行参数 -C target-cpu 选择目标 CPU。可以通过命令行参数 -C target-feature 来为整个 crate 启用或禁用某些单独的特性。

track_caller 属性

track_caller 属性可以应用于除程序入口函数 fn main 之外的任何带有 "Rust" ABI 的函数。当此属性应用于 trait 声明中的函数或方法时，该属性将应用在其所有的实现上。如果 trait 本身提供了带有该属性的默认函数实现，那么该属性也应用于其覆盖实现。

当应用于外部块中的函数上时，该属性也必须应用于此函数的任何链接实现上，否则将导致未定义行为。当此属性应用在一个外部块内可用的函数上时，该外部块中的对该函数的声明也必须带上此属性，否则将导致未定义行为。

将此属性应用到函数 f 上将允许 f 内的代码获得 f 被调用时建立的调用栈的“最顶层”的调用的位置(Location)信息的提示。从观察的角度来看，此属性的实现表现地就像从 f 所在的帧向上遍历调用栈，定位找到最近的有非此属性限定的调用函数 outer，并返回 outer 调用时的位置(Location)信息。

core 提供了 core::panic::Location::caller 来观察调用者。它封装了由 rustc 实现的内部函数 core::intrinsics::caller_location。

例子：

```
#[track_caller]
fn f() {
    println!("{}", std::panic::Location::caller());
}
```

track_caller 属性获取的信息是只是一个提示信息，实现不需要维护它。

特别是，将带有 #[track_caller] 的函数自动强转为函数指针会创建一个填充对象，该填充对象在观察者看来似乎是在此(属性限定的)函数的定义处调用的，从而在这层虚拟调用中丢失了实际的调用者信息。这种自动强转情况的一个常见示例是创建方法被此属性限定的 trait 对象。

注意：前面提到的函数指针填充对象是必需的，因为 rustc 会通过向函数的 ABI 附加一个隐式参数来实现代码生成(codegen)上下文中的 track_caller，但这种添加是不健壮的(unsound)，因为该隐式参数不是函数类型的一部分，那给定的函数指针类型可能引用也可能不引用具有此属性的函数。这里创建一个填充对象会对函数指针的调用方隐藏隐式参数，从而保持可靠性。

极值设置属性

recursion_limit 属性

recursion_limit 属性可以应用于 crate 级别，为可能无限递归的编译期操作（如宏扩展或自动解引用）设置最大递归深度。它使用 MetaNameValueStr 元项属性语法来指定递归深度。

注意：rustc 中这个参数的默认值是 128。

例子：

```
#![recursion_limit = "4"]

macro_rules! a {
    () => { a!(1) };
    (1) => { a!(2) };
    (2) => { a!(3) };
    (3) => { a!(4) };
    (4) => { };
}

// 这无法扩展，因为它需要大于 4 的递归深度。
a!{}

#![recursion_limit = "1"]

// 这里的失败是因为需要两个递归步骤来自动解引用
(|_: &u8| {})(&&&1);
```

type_length_limit 属性

type_length_limit 属性限制在单态化过程中构造具体类型时所做的最大类型替换次数。它应用于 crate 级别，并使用 MetaNameValueStr 元项属性语法来设置类型替换数量的上限。

注意：rustc 中这个参数的默认值是 1048576。

例子：

```
#![type_length_limit = "8"]

fn f<T>(x: T) {}

// 这里的编译失败是因为单态化 `f::<(i32, i32, i32, i32, i32, i32, i32, i32, i32)>>` 需要大于 8 个类型元素。
f((1, 2, 3, 4, 5, 6, 7, 8, 9));
```

类型系统属性

以下属性用于改变类型的使用方式。

non_exhaustive 属性

non_exhaustive 属性表示类型或变体将来可能会添加更多字段或变体。它可以应用在结构体(struct)上、枚举(enum)上 和 枚举变体上。

non_exhaustive 属性使用 MetaWord 元项属性语法，因此不接受任何输入。

在具有 non_exhaustive 限制的类型的定义所在的 crate 内，non_exhaustive 没有效果，在定义所在的 crate 之外，标注为 non_exhaustive 的类型须在添加新字段或变体时保持向后兼容性。例如：

```
#[non_exhaustive]
pub struct Config {
    pub window_width: u16,
    pub window_height: u16,
}

#[non_exhaustive]
pub enum Error {
    Message(String), // 译者注：此变体为元组变体
    Other,
}

pub enum Message {
    #[non_exhaustive] Send { from: u32, to: u32, contents: String },
    #[non_exhaustive] Reaction(u32),
    #[non_exhaustive] Quit,
}

// 非穷尽结构体可以在定义它的 crate 中正常构建。
let config = Config { window_width: 640, window_height: 480 };

// 非穷尽结构体可以在定义它的 crate 中进行详尽匹配
if let Config { window_width, window_height } = config {
    // ...
}

let error = Error::Other;
let message = Message::Reaction(3);

// 非穷尽枚举可以在定义它的 crate 中进行详尽匹配
match error {
    Error::Message(ref s) => { },
    Error::Other => { },
}

match message {
    // 非穷尽变体可以在定义它的 crate 中进行详尽匹配
    Message::Send { from, to, contents } => { },
    Message::Reaction(id) => { },
    Message::Quit => { },
}
```

在定义所在的 crate 之外，非穷尽类型不能被构建：

- 非穷尽结构体或枚举变体不能用 StructExpression 句法（包括函数式更新(functional update)句法）构建。
- 非穷尽枚举实例能被构建。

例如：

```
// `Config`、`Error` `Message`是在上游 crate 中定义的类型，这些类型已被标注为 `#[non_exhaustive]`。
```

```
use upstream::{Config, Error, Message};

// 不能构造 `Config` 的实例，如果在 `upstream` 的新版本中添加了新字段，则本地编译会失败，因此不允许这样做。
let config = Config { window_width: 640, window_height: 480 };

// 可以构造 `Error` 的实例，引入的新变体不会导致编译失败。
let error = Error::Message("foo".to_string());

// 无法构造 `Message::Send` 或 `Message::Reaction` 的实例，
// 如果在 `upstream` 的新版本中添加了新字段，则本地编译失败，因此不允许。
let message = Message::Send { from: 0, to: 1, contents: "foo".to_string(), };
let message = Message::Reaction(0);

// 无法构造 `Message::Quit` 的实例，
// 如果 `upstream` 内的 `Message::Quit` 的因为添加字段变成元组变体(tuple-variant/tuple variant)后，则本地编译失败。
let message = Message::Quit;
```

在定义所在的 crate 之外对非穷尽类型进行匹配，有如下限制：

- 当模式匹配一个非穷尽结构体或枚举变体时，必须使用 StructPattern 句法进行匹配，其匹配臂必须有一个为 ..。元组变体的构造函数的可见性降低为 min($vis, pub(crate))。
- 当模式匹配在一个非穷尽的枚举上时，增加对单个变体的匹配无助于匹配臂需满足枚举变体的穷尽性的这一要求。

例如：

```
// `Config`、`Error` `Message` 是在上游 crate 中定义的类型，这些类型已被标注为 `#[non_exhaustive]`。
use upstream::{Config, Error, Message};

// 不包含通配符匹配臂，无法匹配非穷尽枚举。
match error {
  Error::Message(ref s) => { },
  Error::Other => { },
  // 加上 `_ => {},` 就能编译通过
}

// 不包含通配符匹配臂，无法匹配非穷尽结构体
if let Ok(Config { window_width, window_height }) = config {
    // 加上 `..` 就能编译通过
}

match message {
  // 没有通配符，无法匹配非穷尽（结构体/枚举内的）变体
  Message::Send { from, to, contents } => { },
  // 无法匹配非穷尽元组或单元枚举变体(unit enum variant)。
  Message::Reaction(type) => { },
  Message::Quit => { },
}
```

内置属性索引表

下面是所有内置属性的索引表：

- 模块
  - path — 指定模块的源文件名。
- 条件编译
  - cfg — 控制条件编译。
  - cfg_attr — 选择性包含属性。
- 测试
  - test — 将函数标记为测试函数。
  - ignore — 禁止测试此函数。
  - should_panic — 表示测试应该产生 panic。
- 派生
  - derive — 自动部署 trait 实现
  - automatically_derived — 用在由 derive 创建的实现上的标记。
- 诊断
  - allow、warn、deny、forbid — 更改默认的 lint 检查级别。
  - deprecated — 生成弃用通知。
  - must_use — 为未使用的值生成 lint 提醒。
- 代码生成
  - inline — 内联代码提示。

    - cold — 提示函数不太可能被调用。
    - no_builtins — 禁用某些内置函数。
    - target_feature — 配置特定于平台的代码生成。
    - track_caller - 将父调用位置传递给 std::panic::Location::caller()。
- 极限值设置
    - recursion_limit — 设置某些编译时操作的最大递归限制。
    - type_length_limit — 设置多态类型(polymorphic type)单态化过程中构造具体类型时所做的最大类型替换次数。
- 类型系统(Type System)
    - non_exhaustive — 表明一个类型将来会添加更多的字段/变体。
- ABI、链接(linking)、符号(symbol)、和 FFI
    - link — 指定要与外部(extern)块链接的本地库。
    - link_name — 指定外部(extern)块中的函数或静态项的符号(symbol)名。
    - no_link — 防止链接外部 crate。
    - repr — 控制类型的布局。
    - crate_type — 指定 crate 的类别(库、可执行文件等)。
    - crate_name — 指定 crate 名。
    - no_main — 禁止发布 main 符号(symbol)。
    - export_name — 指定函数或静态项导出的符号(symbol)名。
    - link_section — 指定用于函数或静态项的对象文件的部分。
    - no_mangle — 禁用对符号(symbol)名编码。
    - used — 强制编译器在输出对象文件中保留静态项。
- 宏
    - macro_export — 导出声明宏（macro_rules 宏），用于跨 crate 的使用。
    - macro_use — 扩展宏可见性，或从其他 crate 导入宏。
    - proc_macro — 定义类函数宏。
    - proc_macro_derive — 定义派生宏。
    - proc_macro_attribute — 定义属性宏。
- 预导入包(Preludes)
    - no_std — 从预导入包中移除 std。
    - no_implicit_prelude — 禁用模块内的预导入包查找。
- Runtime
    - panic_handler — 设置处理 panic 的函数。
    - global_allocator — 设置全局内存分配器。
    - windows_subsystem — 指定要链接的 windows 子系统。
- 特性(Features)
    - feature — 用于启用非稳定的或实验性的编译器特性。
- 文档(Documentation)
    - doc — 指定文档。

语法

```
InnerAttribute :
   # ! [ Attr ]

OuterAttribute :
   # [ Attr ]

Attr :
   SimplePath AttrInput?

AttrInput :
      DelimTokenTree
   | = Expression
```

宏

可以使用称被为宏的自定义语法形式来扩展 Rust 的功能和句法。宏需要被命名，并通过一致的语法去调用：some_extension!(...)。

定义新宏有两种方式：
- 声明宏(Macros by Example)以更高级别的声明性的方式定义了一套新句法规则。
- 过程宏(Procedural Macros)可用于实现自定义派生。

声明宏

macro_rules 允许用户以声明性的方式定义语法扩展。我们称这种扩展形式为“声明宏”或简称“宏”。

每个声明宏都有一个名称和一条或多条规则。每条规则都有两部分：
- 一个匹配器(matcher)，描述它匹配的语法；
- 一个转码器(transcriber)，描述成功匹配后将执行的替代调用语法。

匹配器和转码器都必须由定界符包围。宏可以扩展为表达式、语句、项（包括 trait、impl 和外部项）、类型或 pattern。

定界符

声明宏中可以使用以下三种定界符之一：()、{}、[]。定界符必须配对出现。

元变量

在匹配器中，$名称:匹配段选择器 这种语法格式匹配符合指定语法类型的 Rust 语法段，并将其绑定到元变量 $名称 上。有效的匹配段选择器包括：
- item: 项
- block: 块表达式
- stmt: 语句，注意此选择器不匹配句尾的分号，但碰到分号是自身的一部分的程序项语句的情况又会匹配。
- pat: 模式
- expr: 表达式
- ty: 类型
- ident: 标识符或关键字
- path: 类型表达式 形式的路径
- tt: token 树（单个 token 或宏匹配定界符 ()、[] 或{} 中的标记）
- meta: 属性，属性中的内容
- lifetime: 生存期 token
- vis: 可能为空的可见性限定符
- literal: 匹配 -?字面量表达式

由于匹配段类型已在匹配器中指定了，则在转码器中，元变量只简单地用 $名称 这种形式来指代就行了。

元变量最终将被替换为跟它们匹配上的句法元素。元变量关键字 $crate 可以用来指代当前的 crate。元变量可以被多次转码，也可以完全不转码。

重复元

在匹配器和转码器中，重复元被表示为：将需要重复的 token 放在 $(…) 内，然后后跟一个重复运算符，这两者之间可以放置一个可选的分隔符。分隔符可以是除定界符或重复运算符之外的任何 token，其中分号(;)和逗号(,)最常见。例如： $($i:ident),* 表示用逗号分隔的任何数量的标识符。嵌套的重复元是合法的。

重复运算符有以下三个：
- * — 表示任意数量的重复元。
- + — 表示至少有一个重复元。
- ? — 表示一个可选的匹配段，可以出现零次或一次。

由于 ? 表示最多出现一次，所以它不能与分隔符一起使用。

通过分隔符的分隔，重复的匹配段都会被匹配和转码为指定的数量的匹配段。元变量就和这些每个段中的重复元相匹配。例如，之前示例中的 $($i:ident),* 将 $i 去匹配列表中的所有标识符。

在转码过程中，重复元会受到额外的限制，以便于编译器知道该如何正确地扩展它们：
- 在转码器中，元变量必须与它在匹配器中出现的次数、指示符类型以及其在重复元内的嵌套顺序都完全相同。因此，对于匹配器 $($i:ident),*，转码器 => { $i }, => { $($($i)*)* } 和 => { $($i)+ } 都是非法的，但是 => { $($i);* } 是正确的，它用分号分隔的标识符列表替换了逗号分隔的标识符列表。
- 转码器中的每个重复元必须至少包含一个元变量，以便确定扩展多少次。如果在同一个重复元中出现多个元变量，则它们必须绑定到相同数量的匹配段上，不能有的多，有的少。例如，($($i:ident),* ; $($j:ident),*) => (($(($i,$j)),*)) 里，绑定到 $j 的匹配段的数量必须与绑定到 $i 上的相同。这意味着用 (a, b, c; d, e, f) 调用这个宏是合法的，并且可扩展到 ((a,d), (b,e), (c,f))，但是 (a, b, c; d, e) 是非法的，因为前后绑定的数量不同。此要求适用于嵌套的重复元的每一层。

转码

当宏被调用时，宏扩展器按名称查找宏调用，并依次尝试此宏中的每条宏规则。宏会根据第一个成功的匹配进行转码；如果当前转码结果导致错误，不会再尝试进行后续匹配。在匹配时，不会执行预判；如果编译器不能明确地确定如何一个 token 一个 token 地解析宏调用，则会报错。在下面的示例中，编译器不会越过标识符，去提前查看后跟的 token 是)，尽管这能帮助它明确地解析调用：

```
macro_rules! ambiguity {
    ($($i:ident)* $j:ident) => { };
}

ambiguity!(error); // 错误: 局部歧义(local ambiguity)
```

在匹配器和转码器中，token $ 用于从宏引擎中调用特殊行为。不属于此类调用的 token 将按字面意义进行匹配和转码，除了一个例外。这个例外是匹配器的外层定界符将匹配任何一对定界符。因此，比如匹配器 (()) 将匹配 {()}，而 {{}} 不行。字符 $ 不能按字面意义匹配或转码。

当将当前匹配的匹配段转发给另一个声明宏时，第二个宏中的匹配器看到的将是此匹配段类型的不透明抽象语法树。第二个宏不能使用字面量 token 来匹配匹配器中的这个匹配段，唯一可使用的就是此匹配段类型一样的匹配段选择器。但匹配段类型 ident、lifetime、和 tt 是几个例外，它们可以通过字面量 token 进行匹配。下面示例展示了这一限制：

```
macro_rules! foo {
    ($l:expr) => { bar!($l); }
// ERROR:                ^^ no rules expected this token in macro call
}

macro_rules! bar {
    (3) => {}
}

foo!(3);
```

以下示例展示了 tt 类型的匹配段在成功匹配（转码）一次之后生成的 tokens 如何能够再次直接匹配：

```
// 成功编译
macro_rules! foo {
    ($l:tt) => { bar!($l); }
}

macro_rules! bar {
    (3) => {}
}

foo!(3);
```

作用域

声明宏的作用域并不完全像各种项那样工作。宏有两种形式的作用域：

- 文本作用域：文本作用域基于宏在源文件中定义和使用所出现的顺序，或是跨多个源文件出现的顺序，文本作用域是默认的作用域。
- 基于路径的作用域：基于路径的作用域与其他项作用域的运行方式相同。

宏的作用域、导出和导入主要由其属性控制。

当声明宏被非限定标识符（非多段路径段组成的限定性路径）调用时，会首先在文本作用域中查找。如果文本作用域中没有任何结果，则继续在基于路径的作用域中查找。如果宏的名称由路径限定，则只在基于路径的作用域中查找。例子：

```
use lazy_static::lazy_static; // 基于路径的导入.

macro_rules! lazy_static { // 文本定义.
    (lazy) => {};
}

lazy_static!{lazy} // 首先通过文本作用域来查找我们的宏.
self::lazy_static!{} // 忽略文本作用域查找，直接使用基于路径的查找方式找到一个导入的宏.
```

文本作用域

文本作用域很大程度上取决于宏本身在源文件中的出现顺序，其工作方式与用 let 语句声明的局部变量的作用域类似，只不过它可以直接位于模块下。

当使用 macro_rules! 定义宏时，宏在定义之后进入其作用域（请注意，这不影响宏在定义中递归调用自己，因为宏调用的入口还是在定义之后的某次调用点上，此点开始的宏名称递归查找一定有效），在封闭它的作用域结束时离开。

文本作用域可以进入子模块，甚至跨越多个文件：

```
//// src/lib.rs
mod has_macro {
    // m!{} // 报错: m 未在作用域内.

    macro_rules! m {
        () => {};
    }
    m!{} // OK: 在声明 m 后使用.
```

```
    mod uses_macro;
}

// m!{} // Error: m 未在作用域内.

//// src/has_macro/uses_macro.rs

m!{} // OK: m 在上层模块文件 src/lib.rs 中声明后使用
```

多次定义宏并不报错；除非超出作用域，否则最近的宏声明将屏蔽前一个：

```
macro_rules! m {
    (1) => {};
}

m!(1);

mod inner {
    m!(1);

    macro_rules! m {
        (2) => {};
    }
    // m!(1); // 报错: 没有设定规则来匹配 '1'
    m!(2);

    macro_rules! m {
        (3) => {};
    }
    m!(3);
}

m!(1);
```

宏也可以在函数内部声明和使用，其工作方式类似：

```
fn foo() {
    // m!(); // 报错: m 未在作用域内.
    macro_rules! m {
        () => {};
    }
    m!();
}

// m!(); // Error: m 未在作用域内.
```

macro_use 属性

macro_use 属性有两种用途：

- 它可以通过作用于模块的方式让模块内的宏的作用域在模块关闭时不结束：

```
#[macro_use]
mod inner {
    macro_rules! m {
        () => {};
    }
}

m!();
```

- 它可以用于从另一个 crate 里来导入宏，方法是将它附加到当前 crate 根模块中的 extern crate 声明前。以这种方式导入的宏会被导入到 macro_use 预导入包里，而不是直接文本导入，这意味着它们可以被任何其他同名宏屏蔽。要用 #[macro_use] 导入宏必须先使用 #[macro_export] 导出。可以使用可选的 MetaListIdents 元项属性句法指定要导入的宏列表：

```
#[macro_use(lazy_static)] // 或者使用 #[macro_use] 来导入所有宏.
extern crate lazy_static;

lazy_static!{}
// self::lazy_static!{} // 报错: lazy_static 没在 `self` 中定义
```

基于路径的作用域

默认情况下，宏没有基于路径的作用域。但是如果该宏带有 #[macro_export] 属性，则相当于它在当前 crate 的根作用域的顶部被声明，它通常可以这样引用：

```
self::m!();
m!(); // OK: 基于路径的查找发现 m 在当前模块中有声明.

mod inner {
    super::m!();
    crate::m!();
}

mod mac {
    #[macro_export]
    macro_rules! m {
        () => {};
    }
}
```

标有 #[macro_export] 的宏始终是 pub 的，以便可以通过路径或前面所述的 #[macro_use] 方式让其他 crate 来引用。

卫生性

默认情况下，宏中引用的所有标识符都按原样展开，并在宏的调用位置上去查找。如果宏引用的项或宏不在调用位置的作用域内，则这可能会导致问题。为了解决这个问题，可以在路径的开头使用元变量 $crate，强制在定义宏的 crate 中进行查找。

例子：

```
//// 在 `helper_macro` crate 中.
#[macro_export]
macro_rules! helped {
    // () => { helper!() } // 这可能会导致错误，因为 'helper' 在当前作用域之后才定义.
    () => { $crate::helper!() }
}

#[macro_export]
macro_rules! helper {
    () => { () }
}

//// 在另一个 crate 中使用.
// 注意没有导入 `helper_macro::helper`!
use helper_macro::helped;

fn unit() {
    helped!();
}
```

请注意，由于 $crate 指的是当前的（$crate 源码出现的）crate，因此在引用非宏项时，它必须与全限定模块路径一起使用：

```
pub mod inner {
    #[macro_export]
    macro_rules! call_foo {
        () => { $crate::inner::foo() };
    }

    pub fn foo() {}
}
```

尽管 $crate 允许宏在扩展时引用其自身 crate 中的项，但它的使用对可见性没有影响，引用的项或宏必须仍然在调用位置处可见。

当一个宏被导出时，可以在 #[macro_export] 属性里添加 local_inner_macros 属性值，用以自动为该属性修饰的宏内包含的所有宏调用自动添加 $crate:: 前缀。这主要是作为一个工具来迁移那些在引入 $crate 之前的版本编写的 Rust 代码，以便它们能与 Rust 2018 版中基于路径的宏导入一起工作。在使用新版本编写的代码中不鼓励使用它。例子：

```
#[macro_export(local_inner_macros)]
macro_rules! helped {
    () => { helper!() } // 自动转码为 $crate::helper!().
}
```

```
#[macro_export]
macro_rules! helper {
    () => { () }
}
```

歧义限制

宏系统使用的解析器相当强大，但是为了防止其在 Rust 的当前或未来版本中出现二义性解析，因此对它做出了限制。特别地，在消除二义性展开的基本规则之外又增加了一条：元变量匹配的非终结符必须后跟一个已经确定为可以用来安全分隔匹配段的分隔符。

像$i:expr, 或 $i:expr; 这样的匹配符始终是合法的，因为 , 和; 是合法的表达式分隔符。目前规范中的规则是：

- expr 和 stmt 只能后跟一个： =>、,、;。
- pat 只能后跟一个： =>、,、=、|、if、in。
- path 和 ty 只能后跟一个： =>、,、=、|、;、:、>、>>、[、{、as、where、块(block)型非终结符(block nonterminals)。
- vis 只能后跟一个：,、非原生字符串 priv 以外的任何标识符和关键字、可以表示类型开始的任何 token、ident 或 ty 或 path 型非终结符。可以表示类型开始的 token 有：{(, [, !, *,&, &&, ?, 生存期, >, >>, ::, 非关键字标识符, super,self, Self, extern, crate, $crate, _, for, impl, fn, unsafe,typeof, dyn}。
- 其它所有的匹配段选择器没有限制。

当涉及到重复元时，歧义限制适用于所有可能的展开次数，注意需将重复元中的分隔符考虑在内。这意味着：

- 如果重复元包含分隔符，则分隔符必须能够跟随重复元的内容重复。
- 如果重复元可以重复多次（* 或 +），那么重复元的内容必须能自我重复。
- 重复元前后内容必须严格匹配匹配器中指定的前后内容。
- 如果重复元可以匹配零次（* 或 ?），那么它后面的内容必须能够直接跟在它前面的内容后面。

宏调用

宏调用是在编译时执行宏，并用执行结果替换该调用。

可以在下述情况里调用宏：

- 表达式和语句
- patterns
- 类型
- 项，包括关联项
- macro_rules 转码器
- 外部块

当宏调用被用作项或语句时，此时它应用的 MacroInvocationSemi 语法规则要求它如果不使用花括号，则在结尾处须添加分号。在宏调用或宏定义之前不允许使用可见性限定符。

例子：

```
// 作为表达式使用.
let x = vec![1,2,3];

// 作为语句使用.
println!("Hello!");

// 在模式中使用.
macro_rules! pat {
    ($i:ident) => (Some($i))
}

if let pat!(x) = Some(1) {
    assert_eq!(x, 1);
}

// 在类型中使用.
macro_rules! Tuple {
    { $A:ty, $B:ty } => { ($A, $B) };
}

type N2 = Tuple!(i32, i32);

// 作为程序项使用.
thread_local!(static FOO: RefCell<u32> = RefCell::new(1));
```

```
// 作为关联程序项使用.
macro_rules! const_maker {
    ($t:ty, $v:tt) => { const CONST: $t = $v; };
}
trait T {
    const_maker!{i32, 7}
}

// 宏内调用宏
macro_rules! example {
    () => { println!("Macro call in a macro!") };
}
// 外部宏 `example` 展开后, 内部宏 `println` 才会展开.
example!();
```

语法

```
MacroInvocation :
   SimplePath ! DelimTokenTree

DelimTokenTree :
      ( TokenTree* )
   | [ TokenTree* ]
   | { TokenTree* }

TokenTree :
   Token | DelimTokenTree

MacroInvocationSemi :
      SimplePath ! ( TokenTree* ) ;
   | SimplePath ! [ TokenTree* ] ;
   | SimplePath ! { TokenTree* }
```

语法

```
MacroRulesDefinition :
   macro_rules ! IDENTIFIER MacroRulesDef

MacroRulesDef :
      ( MacroRules ) ;
   | [ MacroRules ] ;
   | { MacroRules }

MacroRules :
   MacroRule ( ; MacroRule )* ;?

MacroRule :
   MacroMatcher => MacroTranscriber

MacroMatcher :
      ( MacroMatch* )
   | [ MacroMatch* ]
   | { MacroMatch* }

MacroMatch :
      Token (except $ and delimiters)
   | MacroMatcher
   | $ IDENTIFIER : MacroFragSpec
   | $ ( MacroMatch+ ) MacroRepSep? MacroRepOp

MacroFragSpec :
      block | expr | ident | item | lifetime | literal
   | meta | pat | pat_param | path | stmt | tt | ty | vis

MacroRepSep :
```

```
   Token (except delimiters and repetition operators)

MacroRepOp :
   * | + | ?

MacroTranscriber :
   DelimTokenTree
```

过程宏

过程宏允许在执行函数时创建句法扩展。过程宏有三种形式:

- 类函数宏(function-like macros) - custom!(...)
- 派生宏(derive macros)- #[derive(CustomDerive)]
- 属性宏(attribute macros) - #[CustomAttribute]

过程宏允许在编译时运行对 Rust 语法进行操作的代码，它可以在消费掉一些 Rust 语法输入的同时产生新的 Rust 语法输出。可以将过程宏想象成是从一个 AST 到另一个 AST 的函数映射。

过程宏必须在 crate 类型为 proc-macro 的 crate 中定义。使用 Cargo 时，定义过程宏的 crate 的配置文件里要使用 proc-macro 键做如下设置：

```
[lib]
proc-macro = true
```

作为函数，它们要么有返回，要么触发 panic，要么永无休止地循环。

- 返回根据过程宏的类型替换或添加语法；
- panic 会被编译器捕获，并将其转化为编译器错误；
- 无限循环不会被编译器不会捕获，但会导致编译器被挂起。

过程宏在编译时运行，因此具有与编译器相同的环境资源。例如，它可以访问的标准输入、标准输出和标准错误输出等，这些编译器可以访问的资源。类似地，文件访问也是一样的。因此，过程宏与 Cargo 构建脚本具有相同的安全考量。

过程宏有两种报告错误的方法。首先是 panic；第二个是发布 compile_error 性质的宏调用。

proc_macro crate

过程宏类型的 crate 几乎总是会去链接 proc_macro crate。proc_macro crate 提供了编写过程宏所需的各种类型和工具来让编写更容易。

此 crate 主要包含了一个 TokenStream 类型。过程宏其实是在 *token 流(token streams)*上操作，而不是在某个或某些 AST 节点上操作，因为这对于编译器和过程宏的编译目标来说，这是一个随着时间推移要稳定得多的接口。token stream 大致相当于 Vec<TokenTree>，其中 TokenTree 可以大致视为词法 token。例如，foo 是标识符(Ident)类型的 token，. 是一个标点符号(Punct)类型的 token，1.2 是一个字面量(Literal)类型的 token。不同于 Vec<TokenTree> 的是 TokenStream 的克隆成本很低。

所有类型的 token 都有一个与之关联的 Span。Span 是一个不透明的值，不能被修改，但可以被制造。Span 表示程序内的源代码范围，主要用于错误报告。可以事先（通过函数 set_span）配置任何 token 的 Span。

卫生性

过程宏是非卫生的(unhygienic)。这意味着它的行为就好像它输出的 token 流是被简单地内联写入它周围的代码中一样。这意味着它会受到外部程序项的影响，也会影响外部导入。

鉴于此限制，宏作者需要小心地确保他们的宏能在尽可能多的上下文中正常工作。这通常包括对库中程序项使用绝对路径(例如，使用 ::std::option::Option 而不是 Option)，或者确保生成的函数具有不太可能与其他函数冲突的名称(如 __internal_foo，而不是 foo)。

类函数宏

类函数宏是由一个带有 proc_macro 属性和 (TokenStream) -> TokenStream 签名的公有可见性函数定义。输入 TokenStream 是由宏调用的定界符界定的内容，输出 TokenStream 将替换整个宏调用。

类函数宏是使用宏调用运算符（!）调用的过程宏。类函数过程宏可以在任何宏调用位置调用，这些位置包括语句、表达式、模式、类型表达式、项声明可以出现的位置（包括 extern 块里、固有实现里和 trait 实现里、以及 trait 声明里）。

例如，下面的宏定义忽略它的输入，并将函数 answer 输出到它的作用域。

```
extern crate proc_macro;
use proc_macro::TokenStream;
```

```
#[proc_macro]
pub fn make_answer(_item: TokenStream) -> TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}
```

然后我们用它在一个二进制 crate 里打印 “42” 到标准输出。

```
extern crate proc_macro_examples;
use proc_macro_examples::make_answer;

make_answer!();

fn main() {
    println!("{}", answer());
}
```

派生宏

派生宏为派生(derive)属性定义新输入。这类宏在给定输入结构体(struct)、枚举(enum)或联合体(union) token 流的情况下创建新项。它们也可以定义派生宏辅助属性。

自定义派生宏由带有 proc_macro_derive 属性和 (TokenStream) -> TokenStream 签名的公有可见性函数定义。输入 TokenStream 是带有 derive 属性的项的 token 流。输出 TokenStream 必须是一组项，然后将这组项追加到输入 TokenStream 中的那条项所在的模块或块中。

下面是派生宏的一个示例。它没有对输入执行任何有用的操作，只是追加了一个函数 answer。

```
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(AnswerFn)]
pub fn derive_answer_fn(_item: TokenStream) -> TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}
```

然后使用这个派生宏：

```
extern crate proc_macro_examples;
use proc_macro_examples::AnswerFn;

#[derive(AnswerFn)]
struct Struct;

fn main() {
    assert_eq!(42, answer());
}
```

派生宏辅助属性

派生宏可以将额外的属性添加到它们所在的项的作用域中。这些属性被称为派生宏辅助属性。这些属性是惰性的，它们存在的唯一目的是将这些属性在使用现场获得的属性值反向输入到定义它们的派生宏中。可以认为是设置了一些标志，给宏解析携带信息。

定义辅助属性的方法是在 proc_macro_derive 宏中放置一个 attributes 键，此键带有一个使用逗号分隔的标识符列表，这些标识符是辅助属性的名称。

辅助属性的使用类似于一个属性。

例如，下面的派生宏定义了一个辅助属性 helper，但最终没有用它做任何事情。

```
#[proc_macro_derive(HelperAttr, attributes(helper))]
pub fn derive_helper_attr(_item: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

然后在一个结构体上使用这个派生宏：

```
#[derive(HelperAttr)]
struct Struct {
    #[helper] field: ()
}
```

属性宏

属性宏定义可以附加到项上的新的外部属性，这些项包括外部(extern)块、固有实现、trait 实现，以及 trait 声明中的各类项。

属性宏由带有 proc_macro_attribute 属性和 (TokenStream, TokenStream) -> TokenStream 签名的公有可见性函数定义。签名中的第一个 TokenStream 是属性名称后面的定界 token 树（不包括外层定界符）。如果该属性作为裸属性(bare attribute)给出，则第一个 TokenStream 值为空。第二个 TokenStream 是项的其余部分，包括该程序项的其他属性。输出的 TokenStream 将此属性宏应用的程序项替换为任意数量的项。

例如，下面这个属性宏接受输入流并按原样返回，实际上对属性并无操作。

```
#[proc_macro_attribute]
pub fn return_as_is(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

下面示例显示了属性宏看到的字符串化的 TokenStream。输出将显示在编译时的编译器输出窗口中。（具体格式是以 "out:"为前缀的）输出内容也都在后面每个示例函数后面的注释中给出了。

```
// my-macro/src/lib.rs

#[proc_macro_attribute]
pub fn show_streams(attr: TokenStream, item: TokenStream) -> TokenStream {
    println!("attr: \"{}\"", attr.to_string());
    println!("item: \"{}\"", item.to_string());
    item
}

// src/lib.rs
extern crate my_macro;

use my_macro::show_streams;

// 示例: 基础函数
#[show_streams]
fn invoke1() {}
// out: attr: ""
// out: item: "fn invoke1() { }"

// 示例: 带输入参数的属性
#[show_streams(bar)]
fn invoke2() {}
// out: attr: "bar"
// out: item: "fn invoke2() {}"

// 示例: 输入参数中有多个 token 的
#[show_streams(multiple => tokens)]
fn invoke3() {}
// out: attr: "multiple => tokens"
// out: item: "fn invoke3() {}"

// 示例:
#[show_streams { delimiters }]
fn invoke4() {}
// out: attr: "delimiters"
// out: item: "fn invoke4() {}"
```

非安全性

非安全操作(Unsafe operations)是那些可能潜在地违反 Rust 静态语义里的和内存安全保障相关的操作。

以下语言级别的特性不能在 Rust 的安全(safe)子集中使用:

- 读取或写入可变静态变量；读取或写入或外部静态变量。
- 访问联合体的字段，注意不是给它的字段赋值。
- 调用一个非安全(unsafe)函数（包括外部函数和和内部函数(intrinsic)）。
- 实现非安全(unsafe) trait.

非安全函数

非安全函数是指在任何可能的上下文和/或任何可能的输入中可能出现不安全情况的函数。这样的函数必须以关键字 unsafe 前缀修饰，并且只能在非安全(unsafe)块或其他非安全(unsafe)函数中调用此类函数。

非安全块

一个代码块可以以关键字 unsafe 作为前缀，以允许在安全(safe)函数中调用非安全(unsafe)函数或对裸指针做解引用操作。

当程序员确信某些潜在的非安全操作实际上是安全的，他们可以将这段代码（作为一个整体）封装进一个非安全(unsafe)块中。编译器将认为在当前的上下文中使用这样的代码是安全的。

非安全块用于封装外部库、直接操作硬件或实现语言中没有直接提供的特性。例如，Rust 提供了实现内存安全并发所需的语言特性，但是线程和消息传递的实现（没在语言中实现，而）是在标准库中实现的。

Rust 的类型系统是动态安全条款的保守近似值，因此在某些情况下使用安全代码会带来性能损失。例如，双向链表不是树型结构，那在安全代码中，只能妥协使用引用计数指针表示。通过使用非安全(unsafe)块，可以将反向链接表示为原始指针，这样只用一层 box 封装就能实现了。

未定义的行为

如果 Rust 代码出现了下面列表中的任何行为，则此代码被认为不正确。这包括非安全(unsafe)块和非安全函数里的代码。非安全只意味着避免出现未定义行为的责任在程序员；它没有改变任何关于 Rust 程序必须确保不能写出导致未定义行为的代码的事实。

在编写非安全代码时，确保任何与非安全代码交互的安全代码不会触发下述未定义行为是程序员的责任。对于任何使用非安全代码的安全客户端(safe client)，如果当前条件满足了此非安全代码对于安全条件的要求，那此此非安全代码对于此安全客户端就是健壮的(sound)；如果非安全(unsafe)代码可以被安全代码滥用以致出现未定义行为，那么此非安全(unsafe)代码对这些安全代码来说就是不健壮的(unsound)。

以下是一些未定义的行为：

- 数据竞争。
- 在悬垂或未对齐的裸指针上执行解引用操作 (*expr)，甚至位置表达式(e.g. addr_of!(&*expr))上也不安全。
- 破坏指针别名规则。&mut T 和 &T 遵循 LLVM 的作用域无别名模型，除非 &T 内部包含 UnsafeCell<U> 类型。
- 修改不可变的数据。常量项内的所有数据都是不可变的。此外，所有通过共享引用接触到的数据或不可变绑定所拥有的数据都是不可变的，除非该数据包含在 UnsafeCell<U> 中。
- 通过编译器内部函数调用未定义行为。
- 执行基于当前平台不支持的平台特性编译的代码。
- 用错误的 ABI 约定来调用函数，或使用错误的 ABI 展开约定来从某函数里发起展开。
- 产生非法值，即使在私有字段和本地变量中也是如此。“产生”值发生在这些时候：把值赋给位置表达式、从位置表达式里读取值、传递值给函数/基本运算(primitive operation)或从函数/基本运算中返回值。 以下值非法值（相对于它们各自的类型来说）：
  - 布尔型 bool 中除 false (0) 或 true (1) 之外的值。
  - 不包括在该枚举(enum)类型定义中的判别值。
  - 指向为空(null)的函数指针(fn pointer)。
  - 代理码点(Surrogate)或码点大于 char::MAX 的字符(char)值。
  - !类型的值（任何此类型的值都是非法的）。
  - 从未初始化的内存中，或从字符串切片(str)的未初始化部分获取的整数（i*/u*）、浮点值（f*）或裸指针。
  - 引用或 Box<T> （代表的指针）指向了悬垂(dangling)、未对齐或指向非法值。
  - 宽(wide)引用、Box<T> 或原始指针中带有非法元数据(metadata)：
    - 如果 trait 对象(dyn Trait)的元数据不是指向 Trait 的虚函数表(vtable)的指针，则 dyn Trait 元数据非法。
    - 如果切片的长度不是有效的 usize，则该切片的元数据是非法的（也就是说，不能从它未初始化的内存中读取它）。
  - 带有非法值的自定义类型的值非法。在标准库中，这条促成了 NonNull<T> 和 NonZero* 的出现。

注意： 未初始化的内存对于任何具有有限有效值集的类型来说也隐式非法。也就是说，允许读取未初始化内存的情况只发生在联合体(union)内部和“对齐填充区(padding)”里（类型的字段/元素之间的间隙）。

悬垂指针

如果引用/指针为空或者它指向的所有字节不是同一次分配的一部分，那么它就是“悬垂”的。

它指向的字节跨度(span)由指针本身和指针所指对象的类型的尺寸决定（此尺寸可使用 size_of_val 检测）。因此，如果这个字节跨度为空，则此时“悬垂”与“非空(non-null)”相同。请注意，切片和字符串指向它们的整个区间(range)，因此切片的长度元数据永远不要太大这点很重要。因此切片和字符串的分配不能大于 isize::MAX 个字节。

不被认为是非安全的行为

Rust 编译器并不认为以下行为是非安全的(unsafe)。

- 死锁
- 内存和其他资源的泄漏
- 退出而不调用析构函数
- 通过指针泄漏暴露随机基地址
- 整数溢出

整形溢出

当程序员启用了 debug_assert! 断言（例如，通过启用非优化的构建方式），相应的实现就必须插进来以便在溢出时触发 panic。而其他类型的构建形式有可能也在溢出时触发 panics，也有可能仅仅隐式包装一下溢出值，对溢出过程做静音处理。也就是说具体怎么对待溢出由插进来的编译实现决定。

在隐式包装溢出的情况下，（编译器实现的包装算法）实现必须通过使用二进制补码溢出约定来提供定义良好的溢出包装结果。

整型提供了一些方法，允许显式地执行包装算法。例如，i32::wrapping_add 提供了使用二进制补码溢出约定算法的加法，即包装类加法。

标准库还提供了一个 Wrapping<T> 的新类型，该类型确保 T 的所有标准算术操作都具有包装语义。

逻辑错误

安全代码可能会被添加一些既不能在编译时也不能在运行时检查到的逻辑限制。如果程序打破了这样的限制，其表现可能是未指定的，但不会导致未定义行为。这些表现可能包括 panics、错误的结果、程序中止(aborts)和程序无法终止(non-termination)。并且这些表现在运行期、构建期或各种构建期之间的具体表现也可能有所不同。

例如，同时实现 Hash 和 Eq 就要求被认为相等的值具有相等的散列。另一个例子是像 BinaryHeap、BTreeMap、BTreeSet、HashMap 和 HashSet 这样的数据结构，它们描述了在数据结构中修改键的约束。违反这些约束并不被认为是非安全的，但程序（在逻辑上却）被认为是错误的，其行为是不可预测的。

实体和可见性

实体是一种语言结构，在源程序中可以以某种方式被引用，通常是通过路径。实体包括类型、项、泛型参数、变量绑定、循环标签、生存期、字段、属性和各种 lints。

声明是一种语法结构，它可以引入名称来引用实体。实体的名称在相关作用域内有效。作用域是指可以引用该名称的源码区域。

有些实体是在源码中显式声明的，有些则隐式声明为语言或编译器扩展的一部分。

路径用于引用实体，该引用的实体可以在其他的作用域内。生存期和循环标签使用一个带有前导单引号的专用语法来表达。

名称被分隔成不同的命名空间，这样允许不同名称空间中的实体拥有相同的名称，且不会发生冲突。

名称解析是将路径、标识符和标签绑定到实体声明的编译时过程。

对某些名称的访问可能会受到此名称的可见性的限制。

显式声明的实体

在源码中显式引入名称的实体有：

- 项:
  - 模块声明
  - 外部 crate 声明
  - Use 声明
  - 函数声明 和 函数的参数
  - 类型别名
  - 结构体、联合体、枚举、枚举变体声明和它们的字段
  - 常量项声明
  - 静态项声明
  - trait 项声明和它们的关联项
  - 外部块
  - 声明宏 和 匹配器元变量
  - 实现中的关联项
- 表达式:
  - 闭包的参数
  - while let 模式绑定
  - for 模式绑定
  - if let 模式绑定
  - match 模式绑定
  - 循环标签
- 泛型参数
- trait 约束
- let 语句中的模式绑定
- macro_use 属性可以从其他 crate 里引入宏名称。

- macro_export 属性可以为当前宏引入一个在当前 crate 的根模块下生效的别名

此外，宏调用和属性可以通过扩展源代码到上述项之一来引入名称。

隐式声明的实体

以下实体由语言隐式定义，或由编译器选项和编译器扩展引入：
- 语言预导入包:
  - 布尔型 — bool
  - 文本型 — char and str
  - 整型 — i8, i16, i32, i64, i128, u8, u16, u32, u64, u128
  - 和机器平台相关的整型 — usize and isize
  - 浮点型 — f32 and f64
- 内置属性
- 标准库预导入包里的程序项、属性和宏
- 在根模块下的标准库里的 crate
- 通过编译器链接进的外部 crate
- 工具类属性
- Lints 和 工具类 lint 属性
- 派生辅助属性无需显示引入，就在其程序项内有效
- 'static 生存期标签

此外，crate 的根模块没有名称，但可以使用某些路径限定符或别名来引用。

命名空间

命名空间是已声明的名称的逻辑分组。根据名称所指的实体类型，名称被分隔到不同的命名空间中。 名称空间允许一个名称空间中出现的名称与另一个名称空间中的相同，且不会导致冲突。

在命名空间中，名称被组织在不同的层次结构中，层次结构的每一层都有自己的命名实体集合。

程序有几个不同的命名空间，每个名称空间包含不同种类的实体。使用名称时将根据上下文来在不同的命名空间中去查找该名称的声明。

命名空间实体

下面是一系列命名空间及其对应实体的列表：
- 类型命名空间
  - 模块声明
  - 外部 crate 声明
  - 外部预导入包中的程序项
  - 结构体声明、联合体声明、枚举声明和枚举变体声明
  - trait 项声明
  - 类型别名
  - 关联类型声明
  - 内置类型：布尔型、数字型和文本型
  - 泛型类型参数
  - Self 类型
  - 工具类属性模块
- 值命名空间
  - 函数声明
  - 常量项声明
  - 静态项声明
  - 结构体构造器
  - 枚举变体构造器
  - Self 构造器
  - 泛型常量参数
  - 关联常量声明
  - 关联函数声明
  - 本地绑定 — let, if let, while let, for, match 臂, 函数参数, 闭包参数
  - 闭包捕获的变量
- 宏命名空间
  - 宏声明
  - 内置属性
  - 工具类属性
  - 类函数过程宏
  - 派生宏

    - 辅助派生宏
    - 属性宏
- 生存期命名空间
    - 泛型生存期参数
- 标签命名空间
    - 循环标签

使用不同命名空间中的同名名称的示例：

```
// Foo 在类型命名空间中引入了一个类型，在值命名空间中引入了一个构造函数
struct Foo(u32);

// 宏`Foo`在宏命名空间中声明
macro_rules! Foo {
    () => {};
}

// 参数`f` 的类型中的 `Foo` 指向类型命名空间中的 `Foo`
// `'Foo` 引入一个生存期命名空间里的新的生存期
fn example<'Foo>(f: Foo) {
    // `Foo` 引用值命名空间里的 `Foo`构造器。
    let ctor = Foo;
    // `Foo` 引用宏命名空间里的 `Foo`宏。
    Foo!{}
    // `'Foo` 引入一个标签命名空间里的标签。
    'Foo: loop {
        // `'Foo` 引用 `'Foo`生存期参数, `Foo` 引用类型命名空间中的类型。
        let x: &'Foo Foo;
        // `'Foo` 引用了 `'Foo`标签.
        break 'Foo;
    }
}
```

无命名空间实体

下面的实体有显式的名称，但是这些名称不属于任何特定的命名空间。

- 字段：即使结构体、枚举和联合体的字段被命名，但这些命名字段并不存在于任何显式的命名空间中。它们只能通过字段表达式访问，该表达式只检测被访问的特定类型的字段名。
- use 声明：use 声明命名了导入到当前作用域中的实体，但 use 项本身不属于任何特定的命名空间。相反，它可以在多个名称空间中引入别名，这取决于所导入的项类型。

预导入包

预导入包是一组名称的集合，它会自动把这些名称导入到 crate 中的每个模块的作用域中。

预导入包中的那些名称不是当前模块本身的一部分，它们在名称解析期间被隐式导入。例如，即使像 Box 这样在每个模块的作用域中到处使用的名称，你也不能通过 self::Box 来引用它，因为它不是当前模块的成员。

有几个不同的预导入包：

- 标准库预导入包
- 外部预导入包
- 语言预导入包
- 宏预导入包
- 工具类预导入包

标准库预导入包

标准库预导入包包含了 std::prelude::v1 模块中的名称。如果使用 no_std 属性，那么它将使用 core::prelude::v1 模块中的名称。

外部预导入包

在根模块中使用 extern crate 导入的外部 crate 或直接给编译器提供的外部 crate（也就是在 rustc 命令下使用 --extern 命令行参数选项）会被添加到外部预导入包中。如果使用 extern crate orig_name as new_name 这类别名导入，则符号 new_name 将被添加到此预导入包。

core crate 总是会被添加到外部预导入包中。只要 no_std 属性没有在 crate 根模块中指定，那么 std crate 就会被添加进来

注意：版本差异：在 2015 版中，在外部预导入包中的 crate 不能通过 use 声明来直接引用，因此通常标准做法是用 extern crate 将那它们纳入到当前作用域。从 2018 版开始， use 声明可以直接引用外部预导入包里的 crate，所以再在代码里使用 extern crate 就会被认为是不规范的。

随 rustc 一起引入的 crate,如 alloc 和 test,在使用 Cargo 时不会自动被包含在 --extern 命令行参数选项中,但是却会将 proc_macro 带入到编译类型为 proc-macro 的 crate 的外部预导入包中。即使在 2018 版中，也必须通过外部 crate(extern crate)声明来把它们引入到当前作用域内。例如：

```
extern crate alloc;
use alloc::rc::Rc;
```

语言预导入包

语言预导入包包括语言内置的类型名称和属性名称。语言预导入包总是在当前作用域内有效的。它包括以下内容：

- 类型命名空间
  - 布尔型 — bool
  - 文本型 — char 和 str
  - 整型 — i8, i16, i32, i64, i128, u8, u16, u32, u64, u128
  - 和机器平台相关的整型 — usize 和 isize
  - 浮点型 — f32 和 f64
- 宏命名空间
  - 内置属性

宏预导入包

宏预导入包包含了外部 crate 中的宏，这些宏是通过在当前文档源码内部的 extern crate 声明语句上应用 macro_use 属性来导入此声明中的 crate 内部的宏。

工具类预导入包

工具类预导入包包含了在类型命名空间中声明的外部工具的工具名称。

no_std 属性

默认情况下，标准库自动包含在 crate 根模块中。在 std crate 被添加到根模块中的同时，还会隐式生效一个 macro_use 属性，它将所有从 std 中导出的宏放入到宏预导入包中。

默认情况下，core 和 std 都被添加到外部预导入包中。标准库预导入包包含了 std::prelude::v1 模块中的所有内容。

no_std 属性可以应用在 crate 级别上，用来防止 std crate 被自动添加到相关作用域内。此属性作了如下三件事：

- 阻止 std crate 被添加进外部预导入包。
- 使用标准库预导入包中的 core::prelude::v1 来替代默认情况下导入的 std::prelude::v1。
- 使用 core crate 替代 std crate 来注入到当前 crate 的根模块中，同时把 core crate 下的所有宏导入到宏预导入包中。

注意：当 crate 的目标平台不支持标准库或者故意不使用标准库的功能时，使用 core 预导入包而不是 std 预导入包是很有用的。此时没有导入的标准库的那些功能主要是动态内存分配（例如：Box 和’ Vec）和文件，以及网络功能（例如：std::fs 和 std::io)。

警告：使用 no_std 并不会阻止标准库被链接进来。使用 extern crate std; 将 std crate 导入仍然有效，相关的依赖项也可以被正常链接进来。

no_implicit_prelude 属性

no_implicit_prelude 属性可以应用在 crate 级别或模块上，用以指示它不应该自动将标准库预导入包、外部预导入包或工具类预导入包引入到当前模块或其任何子模块的作用域中。

此属性不影响语言预导入包。

注意：在 2015 版中，no_implicit_prelude 属性不会影响宏预导入包，从标准库导出的所有宏仍然包含在宏预导入包中。从 2018 版开始，它也会禁止 宏预导入包生效。

路径

路径是由命名空间限定符 （::） 逻辑分隔的一个或多个路径段的序列。如果路径仅包含一个段，则它指的是局部作用域内的一个 item 或变量。如果一个路径有多个段，它总是指一个 item。

仅由标识符段组成的简单路径的两个示例：

```
x;
```

```
x::y::z;
```

路径种类

简单路径

简单路径用于可见性标记、属性、宏和 use item 中。

例子：

```
use std::io::{self, Write}; // 用于 use item 中
mod m {
    #[clippy::cyclomatic_complexity = "0"] // 用于属性
    pub (in super) fn f1() {} // 用于可见性标记
}
```

语法

```
SimplePath :
   ::? SimplePathSegment (:: SimplePathSegment)*

SimplePathSegment :
   IDENTIFIER | super | self | crate | $crate
```

表达式中的路径

表达式中的路径允许指定具有泛型的类型参数的路径。它们用于表达式和 pattern 的各个地方。

:: 标记在泛型的类型参数的开头 < 之前是必需的，以避免与小于运算符产生歧义。 这通俗地称为“turbofish”语法。例如：

```
(0..10).collect::<Vec<_>>();
Vec::<u8>::with_capacity(1024);
```

泛型的类型参数的顺序被限制为生命周期参数，然后是类型参数，然后是常量参数，然后是等式约束。其中：

- 常量参数必须用大括号括起来，除非它们是字面量或单段路径。
- 等式约束用于约束 trait 的关联类型。即指定 trait 的关联类型。

语法

```
PathInExpression :
   ::? PathExprSegment (:: PathExprSegment)*

PathExprSegment :
   PathIdentSegment (:: GenericArgs)?

PathIdentSegment :
   IDENTIFIER | super | self | Self | crate | $crate

GenericArgs :
      < >
   | < ( GenericArg , )* GenericArg ,? >

GenericArg :
   Lifetime | Type | GenericArgsConst | GenericArgsBinding

GenericArgsConst :
      BlockExpression
   | LiteralExpression
   | - LiteralExpression
   | SimplePathSegment

GenericArgsBinding :
   IDENTIFIER = Type
```

类型路径

类型路径用于类型定义、implementation 声明、trait bound(包括泛型参数、trait object 类型、impl trait 类型)以及限定路径。它用来指示一个 item 声明的类型。

类型路径同样允许指定具有泛型的类型参数的路径。对于类型路径，尽管在泛型的类型参数之前允许使用 :: 标记，但它不是必需的，因为没有像表达式中的路径那样的歧义。

泛型的类型参数的顺序被限制为生命周期参数，然后是类型参数，然后是常量参数，然后是等式约束。其中：

- 常量参数必须用大括号括起来，除非它们是字面量或单段路径。
- 等式约束用于约束 trait 的关联类型。即指定 trait 的关联类型。例如 Iterator<Item = ops::Example<'a>>。

例子：

```
impl ops::Index<ops::Range<usize>> for S { /*...*/ }
fn i<'a>() -> impl Iterator<Item = ops::Example<'a>> {
    // ...
}
type G = std::boxed::Box<dyn std::ops::FnOnce(isize) -> isize>;
```

Fn/FnMut/FnOnce trait

对于 Fn/FnMut/FnOnce trait 都有类似如下的定义：

```
pub trait Fn<Args> {
    type Output;
    ...
}
```

假如和上面一个书写其类型路径，那么路径类似为 SomePath::Fn<TypePara,Output=TypeBind>，Rust 提供了一种语法糖：

- 如果标识符是一个 Trait，那么类似 Foo(i32,i64)会被扩展为 Foo<(i32,i64),Output=()>
- 如果标识符是一个 Trait，那么类似 Foo(i32,i64)->i32 会被扩展为 Foo<(i32,i64),Output=i32>

因此，对于 Fn/FnMut/FnOnce 这三个 trait，他们的类型路径可以简化为类似一个函数签名一样。例如：

```
type G = std::boxed::Box<dyn std::ops::FnOnce(isize) -> isize>;
// 等价于
type G = std::boxed::Box<dyn std::ops::FnOnce<(isize,),Output=isize>>;
```

语法

```
TypePath :
   ::? TypePathSegment (:: TypePathSegment)*

TypePathSegment :
   PathIdentSegment ::? (GenericArgs | TypePathFn)?

TypePathFn :
( TypePathFnInputs? ) (-> Type)?

TypePathFnInputs :
Type (, Type)* ,?

PathIdentSegment :
   IDENTIFIER | super | self | Self | crate | $crate

GenericArgs :
      < >
   | < ( GenericArg , )* GenericArg ,? >

GenericArg :
   Lifetime | Type | GenericArgsConst | GenericArgsBinding

GenericArgsConst :
      BlockExpression
   | LiteralExpression
   | - LiteralExpression
   | SimplePathSegment

GenericArgsBinding :
   IDENTIFIER = Type
```

限定路径

完全限定的路径用于消除 trait impl 的路径与规范路径之间的歧义。通过这种方式，可以显式的指定实现了指定 trait impl 的类型的路径。

在类型规范中使用时，它支持使用下面指定的类型语法。

```
struct S;
impl S {
    fn f() { println!("S"); }
}
trait T1 {
    fn f() { println!("T1 f"); }
}
impl T1 for S {}
trait T2 {
    fn f() { println!("T2 f"); }
}
impl T2 for S {}
S::f();  // Calls the inherent impl.
<S as T1>::f();  // Calls the T1 trait function.
<S as T2>::f();  // Calls the T2 trait function.
```

语法

```
QualifiedPathInExpression :
   QualifiedPathType (:: PathExprSegment)+

QualifiedPathType :
   < Type (as TypePath)? >

QualifiedPathInType :
   QualifiedPathType (:: TypePathSegment)+
```

路径限定符

路径可以用各种前导限定符表示，以改变其解析方式的含义。

::

以 :: 开头的路径被认为是全局路径，根据版本的不同解析的方式不同：

- 在 2015 版本中，标识符从“crate 根”开始解析，类似于 2018 版本的 crate::。
- 在 2018 版本中，以 :: 开头的路径从 extern crate 声明中的 crates 开始解析。也就是说，它们后面必须跟一个 crate 的名称。

路径中的每个标识符都必须解析为一个 item。

例子：

```
mod a {
    pub fn foo() {}
}
mod b {
    pub fn foo() {
        ::a::foo(); // In Rust 2015, call `a`'s foo function
        // In Rust 2018, `::a` would be interpreted as the crate `a`.
    }
}
```

self

self 解析相对于当前模块的路径。self 只能用作第一段，前面没有 ::。

例子：

```
fn foo() {}
fn bar() {
    self::foo();
}
```

Self

Self，带有大写的“S”，用于指代 trait 和 implementation 中的实现类型。

Self 只能用作第一段，前面没有 ::。

例子：

```
trait T {
    type Item;
    const C: i32;
    // `Self` will be whatever type that implements `T`.
    fn new() -> Self;
    // `Self::Item` will be the type alias in the implementation.
    fn f(&self) -> Self::Item;
}
struct S;
impl T for S {
    type Item = i32;
    const C: i32 = 9;
    fn new() -> Self {           // `Self` is the type `S`.
        S
    }
    fn f(&self) -> Self::Item {  // `Self::Item` is the type `i32`.
        Self::C                  // `Self::C` is the constant value `9`.
    }
}
```

super

路径中的 super 解析为父模块。

它只能在路径的前导段中使用，或在初始 self 段之后使用。

例子：

```
mod a {
    fn foo() {}

    mod b {
        mod c {
            fn foo() {
                super::super::foo(); // call a's foo function
                self::super::super::foo(); // call a's foo function
            }
        }
    }
}
```

crate

crate 解析为相对于当前 crate 的路径。

crate 只能用作第一段，前面没有 ::。

例子：

```
fn foo() {}
mod a {
    fn bar() {
        crate::foo();
    }
}
```

$crate

$crate 只在宏声明器中使用。$crate 将扩展为从定义宏的 crate 的顶层访问项的路径，无论宏被调用哪个 crate。

只能用作第一段，前面没有 ::

例子：

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
```

规范路径

模块或 implementation 中定义的 item 具有一个规范路径，路径对应于它在 crate 中定义的位置。规范路径被定义路径前缀附加项本身定义的路径段。

以下情况是没有规范路径的：

- implementation 和 use 声明本身是没有规范路径，尽管 implementation 中定义的项确实有它们。
- 块表达式中定义的项是没有规范路径的。
- 在没有规范路径的项中定义的项也是没有规范路径。
- implementation 声明中存在某一处没有规范路径，那么 implementation 声明中的关联项也是没有规范路径的。例如：
  - 实现的类型没有规范路径，那么 implementation 声明中的关联项也是没有规范路径。
  - 正在实现的 trait 没有规范路径，那么 implementation 声明中的关联项也是没有规范路径。
  - 类型参数没有规范路径，那么 implementation 声明中的关联项也是没有规范路径。
  - 类型参数上的绑定没有规范路径，那么 implementation 声明中的关联项也是没有规范路径。

模块和 implementation 的规范路径如下：

- 模块的路径前缀是该模块的规范路径。例如：::a。
- 对于 inherent impl，它是被实现的项的规范路径，被尖括号（<>）括起来，例如：<::a::Struct>。
- 对于 trait impl，它是被实现的项的规范路径，然后是 as 关键字，后面跟 trait 的规范路径，所有这些都用尖角（<>）括号括起来。例如：<::a::Struct as ::a::Trait>。

规范路径仅在给定的 crate 中有意义。跨 crate 没有全局命名空间；项的规范路径仅在 crate 内标识它。

例子：

```
// Comments show the canonical path of the item.

mod a { // ::a
    pub struct Struct; // ::a::Struct

    pub trait Trait { // ::a::Trait
        fn f(&self); // ::a::Trait::f
    }

    impl Trait for Struct {
        fn f(&self) {} // <::a::Struct as ::a::Trait>::f
    }

    impl Struct {
        fn g(&self) {} // <::a::Struct>::g
    }
}

mod without { // ::without
    fn canonicals() { // ::without::canonicals
        struct OtherStruct; // None

        trait OtherTrait { // None
            fn g(&self); // None
        }

        impl OtherTrait for OtherStruct {
            fn g(&self) {} // None
        }

        impl OtherTrait for ::a::Struct {
```

```
            fn g(&self) {} // None
        }

        impl ::a::Trait for OtherStruct {
            fn f(&self) {} // None
        }
    }
}
```

可见性

Rust 的名称解析是在命名空间的层级结构的全局层（即最顶层）上运行的。此层级结构中的每个级别都可以看作是某个项。这些项就是我们前面提到的那些项之一（注意这也包括外部 crate）。声明或定义一个新模块可以被认为是在定义的位置向此层次结构中插入一个新的（层级）树。

为了控制接口是否可以被跨模块使用，Rust 会检查每个项的使用，来看看它是否被允许使用。此处就是生成隐私告警的地方，或者说是提示：“you used a private item of another module and weren't allowed to.”的地方。

默认情况下，Rust 中的所有内容都是私有的，但有两个例外：pub trait 中的关联项默认为公有的；pub 枚举中的枚举变体也默认为公有的。当一个项被声明为 pub 时，它可以被认为是外部世界能以访问的。

依据项是公有的还是私有的，Rust 分两种情况来访问数据：

- 如果某个项是公有的，并且如果可以从外部的某一模块 m 访问到该项的所有祖先模块，则一定可以从这个模块 m 中访问到该项。
- 如果某个项是私有的，则当前模块及当前模块的后代模块都可以访问它。

对于可访问的项，还可以通过重导出来命名该项。对于一个 Rust 程序要通过隐私检查，所有的路径都必须满足上述两个访问规则。这里说的路径包括所有的 use 语句、表达式、类型等。

这两种情况在创建能对外暴露公共 API 同时又隐藏内部实现细节的模块层次结构时非常好用。以下是一些常见案例和它们需要做的事情：

- 库开发人员需要将一些功能暴露给链接了其库的 crate。作为第一种情况的结果，这意味着任何那些在外部可用的程序项自身以及其路径层级结构中的每一层都必须是公有的(pub)的。并且此层级链中的任何私有程序项都不允许被外部访问。
- crate 需要一个全局可用的“辅助模块(helper module)”，但又不想将辅助模块公开为公共 API。为了实现这一点，可以在整个 crate 的根模块（路径层级结构中的最顶层）下建一个私有模块，该模块在内部是“公共 API”。因为整个 crate 都是根模块的后代，所以整个本地 crate 里都可以通过第二种情况访问这个私有模块。
- 在为模块编写单元测试时，通常的习惯做法是给要测试的模块加一个命名为 mod test 的直接子模块。这个模块可以通过第二种情况访问父模块的任何程序项，这意味着内部实现细节也可以从这个子模块里进行无缝地测试。

例子：

```
// 这个模块是私有的，这意味着没有外部 crate 可以访问这个模块。
// 但是，由于它在当前 crate 的根模块下，
// 因此当前 crate 中的任何模块都可以访问该模块中任何公有可见性程序项。
mod crate_helper_module {

    // 这个函数可以被当前 crate 中的任何东西使用
    pub fn crate_helper() {}

    // 此函数*不能*被用于 crate 中的任何其他模块中。它在 `crate_helper_module` 之外不可见，
    // 因此只有当前模块及其后代可以访问它。
    fn implementation_detail() {}
}

// 此函数“对根模块是公有”的，这意味着它可被链接了此 crate 的其他 crate 使用。
pub fn public_api() {}

// 与 'public_api' 类似，此模块是公有的，因此其他的 crate 是能够看到此模块内部的。
pub mod submodule {
    use crate_helper_module;

    pub fn my_method() {
        // 本地 crate 中的任何程序项都可以通过上述两个规则的组合来调用辅助模块里的公共接口。
        crate_helper_module::crate_helper();
    }

    // 此函数对任何不是 `submodule` 的后代的模块都是隐藏的
    fn my_implementation() {}
```

```
    #[cfg(test)]
    mod test {

        #[test]
        fn test_my_implementation() {
            // 因为此模块是 `submodule` 的后代，因此允许它访问 `submodule` 内部的私有项，而不会侵犯隐私权。
            super::my_implementation();
        }
    }
}
```

限定可见性

除了公有和私有之外，Rust 还允许用户（用关键字 pub ）声明仅在给定作用域内可见的项。声明形式的限制规则如下：

- pub(in path) 使一个项在提供的 path 中可见。path 必须是声明其可见性的项的祖先模块。
- pub(crate) 使一个项在当前 crate 中可见。
- pub(super) 使一个项对父模块可见。这相当于 pub(in super)。
- pub(self) 使一个项对当前模块可见。这相当于 pub(in self) 或者根本不使用 pub。

注意：从 2018 版开始，pub(in path) 的路径必须以 crate、self 或 super 开头。2015 版还可以使用以 :: 开头的路径，或以根模块下的模块名的开头的路径。

此语法仅对项的可见性添加了另一个限制。它不能保证该项在指定作用域的所有部分都可见。要访问一个项，当前作用域内它的所有父项还是必须仍然可见。

例子：

```
pub mod outer_mod {
    pub mod inner_mod {
        // 此函数在 `outer_mod` 内部可见
        pub(in crate::outer_mod) fn outer_mod_visible_fn() {}
        // 同上，但这只能在 2015 版中有效
        pub(in outer_mod) fn outer_mod_visible_fn_2015() {}

        // 此函数对整个 crate 都可见
        pub(crate) fn crate_visible_fn() {}

        // 此函数在 `outer_mod` 下可见
        pub(super) fn super_mod_visible_fn() {
            // 此函数之所以可用，是因为我们在同一个模块下
            inner_mod_visible_fn();
        }

        // 这个函数只在 `inner_mod` 中可见，这与它保持私有的效果是一样的。
        pub(self) fn inner_mod_visible_fn() {}
    }
    pub fn foo() {
        inner_mod::outer_mod_visible_fn();
        inner_mod::crate_visible_fn();
        inner_mod::super_mod_visible_fn();

        // 此函数不再可见，因为我们在 `inner_mod` 之外
        // 错误! `inner_mod_visible_fn` 是私有的
        //inner_mod::inner_mod_visible_fn();
    }
}

fn bar() {
    // 此函数仍可见，因为我们在同一个 crate 里
    outer_mod::inner_mod::crate_visible_fn();

    // 此函数不再可见，因为我们在`outer_mod`之外
    // 错误! `super_mod_visible_fn` 是私有的
    //outer_mod::inner_mod::super_mod_visible_fn();

    // 此函数不再可见，因为我们在`outer_mod`之外
    // 错误! `outer_mod_visible_fn` 是私有的
    //outer_mod::inner_mod::outer_mod_visible_fn();
```

```
    outer_mod::foo();
}

fn main() { bar() }
```

重导出和可见性

Rust 允许使用指令 pub use 公开重导出项。因为这是一个公有指令，所以允许通过上面的规则验证后在当前模块中使用该项。重导出本质上允许使用公有方式访问重导出的项的内部。

例如，下面程序是有效的：

```
pub use self::implementation::api;

mod implementation {
    pub mod api {
        pub fn f() {}
    }
}
```

这意味着任何外部 crate，只要引用 implementation::api::f 都将收到违反隐私的错误报告，而使用路径 api::f 则被允许。

当重导出私有程序项时，可以认为它允许通过重导出短路了“隐私链”，而不是像通常那样通过命名空间层次结构来传递“隐私链”。

语法

```
Visibility :
      pub
   | pub ( crate )
   | pub ( self )
   | pub ( super )
   | pub ( in SimplePath )
```

crate 和源文件

编译模型以 crate 为中心。每次编译都以源码的形式处理单个的 crate，如果成功，将生成二进制形式的单个 crate：可执行文件或某种类型的库文件。

crate 是编译和链接的单元，也是版本控制、版本分发和运行时加载的基本单元。一个 crate 包含一个嵌套的带作用域的模块树。这个树的顶层是一个匿名的模块（从模块内部路径的角度来看），并且一个 crate 中的任何项都有一个规范的模块路径来表示它在 crate 的模块树中的位置。

Rust 编译器总是使用单个源文件作为输入来开启编译过程的，并且总是生成单个输出 crate。对输入源文件的处理可能导致其他源文件作为模块被加载进来。源文件的扩展名为 .rs。

Rust 源文件描述了一个模块，其名称和（在当前 crate 的模块树中的）位置是从源文件外部定义的：要么通过引用源文件中的显式模块(Module)项，要么由 crate 本身的名称定义。每个源文件都是一个模块，但并非每个模块都需要自己的源文件：多个模块定义可以嵌套在同一个文件中。

每个源文件包含一个由零个或多个项定义组成的代码序列，并且这些源文件都可选地从应用于其内部模块的任意数量的属性开始，大部分这些属性都会影响编译器行为。匿名的 crate 根模块可附带一些应用于整个 crate 的属性。

例子：

```
// 指定 crate 名称.
#![crate_name = "projx"]

// 指定编译输出文件的类型
#![crate_type = "lib"]

// 打开一种警告
// 这句可以放在任何模块中，而不是只能放在匿名 crate 模块里。
#![warn(non_camel_case_types)]
```

字节顺序标记(BOM)

可选的 UTF8 字节序标记（UTF8BOM 产生式）表示该文件是用 UTF8 编码的。它只能出现在文件的开头，并且编译器会忽略它。

Shebang

源文件可以有一个 shebang（SHEBANG 产生式），它指示操作系统使用什么程序来执行此文件。它本质上是将源文件作为可执行脚本处理。shebang 只能出现在文件的开头（但是要在可选的 UTF8BOM 生产式之后）。它会被编译器忽略。

例子：

```
#!/usr/bin/env rustx

fn main() {
    println!("Hello!");
}
```

main 函数

包含 main 函数的 crate 可以被编译成可执行文件。

如果一个 main 函数存在，它必须不能有参数，不能对其声明任何 trait 约束或生存期约束，不能有任何 where 子句，并且它的返回类型必须是以下类型之一：

- ()
- Result<(), E> where E: Error

no_main 属性

可在 crate 层级使用 no_main 属性来禁止对可执行二进制文件发布 main symbol，即禁止当前 crate 的 main 函数的执行。当链接的其他对象定义了 main 函数时，这很有用。

crate_name 属性

可在 crate 层级应用 crate_name 属性，用来指定 crate 的名称。crate 名称不能为空，且只能包含 [Unicode 字母数字]或字符 -(U+002D)。

crate_name 属性使用 MetaNameValueStr 元项属性语法。

例子：

```
#![crate_name = "mycrate"]
```

语法

```
Crate :
   UTF8BOM?
   SHEBANG?
   InnerAttribute*
   Item*

UTF8BOM : \uFEFF
SHEBANG : #! ~\n+†
```

编译

条件编译

根据某些条件，条件性编译的源代码可以被认为是 crate 源代码的一部分，也可以不被认为是 crate 源代码的一部分。可以使用属性 cfg 和 cfg_attr 以及内置的 cfg macro 来有条件地对源代码进行编译，这些条件可以基于被编译的 crate 的目标架构、传递给编译器的值。

每种形式的编译条件都有一个计算结果为真或假的配置谓词。谓词是以下内容之一：

- 一个配置选项。如果设置了该选项，则为真，如果未设置则为假。
- all() 这样的配置谓词列表，列表内的配置谓词以逗号分隔。如果至少有一个谓词为假，则为假。如果没有谓词，则为真。
- any() 这样的配置谓词列表，列表内的配置谓词以逗号分隔。如果至少有一个谓词为真，则为真。如果没有谓词，则为假。
- 带一个配置谓词的 not() 模式 。如果此谓词为假，整个配置它为真；如果此谓词为真，整个配置为假。

配置选项可以是名称，也可以是键值对，它们可以设置，也可以不设置。名称以单个标识符形式写入，例如 unix。键值对被写为标识符后跟=，然后再跟一个字符串。例如，target_arch=“x86_64” 就是一个配置选项。= 周围的空白符将被忽略，例如 foo="bar" 和 foo = "bar" 是等价的配置选项。

键在键值对配置选项列表中不是唯一的。例如，feature = "std" and feature = "serde" 可以同时设置。

设置配置选项

设置哪些配置选项是在 crate 编译期时就静态确定的。由以下部分组成：

- 一些选项属于编译器设置集，这部分选项是编译器根据相关编译数据设置的。
- 其他选项属于任意设置集，这部分设置必须从代码之外传参给编译器来自主设置。正在编译的 crate 的源代码中无法设置编译配置选项。
  - 对于 rustc，任意配置集的配置选项要使用命令行参数 --cfg 来设置。
  - 键名为 feature 的配置选项一般被 Cargo 约定用于指定编译期（用到的各种编译）选项和可选依赖项。

注意：任意配置集的配置选项可能与编译器设置集的配置选项设置相同的值。例如，在编译一个 Windows 目标时，可以执行命令行 rustc --cfg "unix" program.rs，这样就同时设置了 unix 和 windows 配置选项。

以下是编译器设置集选项：

target_arch

键值对选项，用于一次性设置编译目标的 CPU 架构。该值类似于平台的目标三元组中的第一个元素，但也不完全相同。

示例值：
- "x86"
- "x86_64"
- "mips"
- "powerpc"
- "powerpc64"
- "arm"
- "aarch64"

target_feature

键值对选项，用于设置当前编译目标的可用平台特性。

示例值：
- "avx"
- "avx2"
- "crt-static"
- "rdrand"
- "sse"
- "sse2"
- "sse4.1"

target_os

键值对选项，用于一次性设置编译目标的操作系统类型。该值类似于平台目标三元组中的第二和第三个元素。

示例值：
- "windows"
- "macos"
- "ios"
- "linux"
- "android"
- "freebsd"
- "dragonfly"
- "openbsd"
- "netbsd"

target_family

键值对选项提供了对具体目标平台更通用化的描述，比如编译目标操作系统或架构。可以设置任意数量的键值对。 最多设置一次，用于设置编译目标的操作系统类别。

示例值：
- "unix"
- "windows"
- "wasm"

unix 和 windows

如果设置了 target_family = "unix" 则谓词 unix 为真；如果设置了 target_family = "windows" 则谓词 windows 为真。

target_env

键值对选项，用来进一步消除编译目标平台信息与所用 ABI 或 libc 相关的歧义。

由于历史原因，仅当实际需要消除歧义时，才将此值定义为非空字符串。因此，例如在许多 GNU 平台上，此值将为空。该值类似于平台目标三元组的第四个元素，但也有区别。一个区别是在嵌入式 ABI 上，比如在目标为嵌入式系统时，gnueabihf 会简单地将 target_env 定义为 "gnu"。

示例值：
- ""
- "gnu"
- "msvc"
- "musl"
- "sgx"

target_endian

键值对选项，根据编译目标的 CPU 的字节序(endianness)属性一次性设置值为 “little” 或 “big”。

target_pointer_width

键值对选项，用于一次性设置编译目标的指针位宽。

示例值：
- "16"
- "32"
- "64"

target_vendor

键值对选项，用于一次性设置编译目标的供应商。

示例值：
- "apple"
- "fortanix"
- "pc"
- "unknown"

test

在编译测试套件时启用。通过在 rustc 里使用 --test 命令行参数来完成此启用。

debug_assertions

在进行非优化编译时默认启用。这可以用于在开发中启用额外的代码调试功能，但不能在生产中启用。例如，它控制着标准库的 debug_assert! 宏（是否可用）。

proc_macro

当须要指定当前 crate 的编译输出文件类型(crate-type)为 proc_macro 时设置。

条件编译的形式

cfg 属性

cfg 属性根据配置的谓词有条件的包括它所附加的东西。如果谓词为真，则重写该部分代码，使其上没有 cfg 属性。如果谓词为假，则从源代码中删除该内容。

cfg 属性允许在任何允许属性的地方上使用。

例子：

```
// 该函数只会在编译目标为 macOS 时才会包含在构建中
#[cfg(target_os = "macos")]
fn macos_only() {
```

```
  // ...
}

// 此函数仅在定义了 foo 或 bar 时才会被包含在构建中
#[cfg(any(foo, bar))]
fn needs_foo_or_bar() {
  // ...
}

// 此函数仅在编译目标是 32 位体系架构的类 unix 系统时才会被包含在构建中
#[cfg(all(unix, target_pointer_width = "32"))]
fn on_32bit_unix() {
  // ...
}

// 此函数仅在没有定义 foo 时才会被包含在构建中
#[cfg(not(foo))]
fn needs_not_foo() {
  // ...
}
```

语法

```
CfgAttrAttribute :
   cfg ( ConfigurationPredicate )
```

cfg_attr 属性

cfg_attr 属性根据配置谓词有条件地包含属性。当配置谓词为真时，此属性展开为谓词后列出的属性。

cfg_attr 属性允许在任何允许属性的地方上使用。

例子：

```
#[cfg_attr(target_os = "linux", path = "linux.rs")]
#[cfg_attr(windows, path = "windows.rs")]
mod os;
```

可以列出零个、一个或多个属性。多个属性将各自展开为单独的属性。例如：

```
#[cfg_attr(feature = "magic", sparkles, crackles)]
fn bewitched() {}

// 当启用了 `magic` 特性时，上面的代码将会被展开为:
#[sparkles]
#[crackles]
fn bewitched() {}
```

cfg_attr 能展开为另一个 cfg_attr。例如：

```
#[cfg_attr(target_os = "linux", cfg_attr(feature = "multithreaded", some_other_attribute))]
// 等效于：
#[cfg_attr(all(target_os = "linux", feature ="multithreaded"), some_other_attribute)].
```

语法

```
CfgAttrAttribute :
   cfg_attr ( ConfigurationPredicate , CfgAttrs? )

CfgAttrs :
   Attr (, Attr)* ,?
```

cfg 宏

内置的 cfg 宏接受单个配置谓词，当谓词为真时计算为 true 字面量，当谓词为假时计算为 false 字面量。

例子：

```
let machine_kind = if cfg!(unix) {
  "unix"
} else if cfg!(windows) {
```

```
  "windows"
} else {
  "unknown"
};

println!("I'm running on a {} machine!", machine_kind);
```

语法

```
ConfigurationPredicate :
      ConfigurationOption
   | ConfigurationAll
   | ConfigurationAny
   | ConfigurationNot

ConfigurationOption :
   IDENTIFIER (= (STRING_LITERAL | RAW_STRING_LITERAL))?

ConfigurationAll
   all ( ConfigurationPredicateList? )

ConfigurationAny
   any ( ConfigurationPredicateList? )

ConfigurationNot
   not ( ConfigurationPredicate )

ConfigurationPredicateList
   ConfigurationPredicate (, ConfigurationPredicate)* ,?
```

编译属性

used 属性

used 属性只能用在静态(static)项上。

此属性强制编译器将该变量保留在输出对象文件中(.o、.rlib 等，不包括最终的二进制文件)，即使该变量没有被 crate 中的任何其他项使用或引用。注意，链接器(linker)仍有权移除此类变量。

例子：

```
// foo.rs

// 将保留，因为 `#[used]`:
#[used]
static FOO: u32 = 0;

// 可移除，因为没实际使用:
#[allow(dead_code)]
static BAR: u32 = 0;

// 将保留，因为这个是公有的:
pub static BAZ: u32 = 0;

// 将保留，因为这个被可达公有函数引用:
static QUUX: u32 = 0;

pub fn quux() -> &'static u32 {
    &QUUX
}

// 可移除，因为被私有且未被使用的函数引用:
static CORGE: u32 = 0;

#[allow(dead_code)]
fn corge() -> &'static u32 {
    &CORGE
```

```
}
```

运行 rustc 命令查看

```
$ rustc -O --emit=obj --crate-type=rlib foo.rs

$ nm -C foo.o
0000000000000000 R foo::BAZ
0000000000000000 r foo::FOO
0000000000000000 R foo::QUUX
0000000000000000 T foo::quux
```

no_mangle 属性

可以在任何项上使用 no_mangle 属性来禁用标准名称符号名混淆。禁用此功能后，此项的导出符号(symbol)名将直接是此项的原来的名称标识符。

此外，就跟 used 属性一样，此属性修饰的项也将从生成的库或对象文件中公开导出。

link_section 属性

link_section 属性指定了输出对象文件中函数或静态项的内容将被放置到的节点位置。

它使用 MetaNameValueStr 元项属性句法指定节点名称。

例子：

```
#[no_mangle]
#[link_section = ".example_section"]
pub static VAR1: u32 = 1;
```

export_name 属性

export_name 属性指定函数或静态项的导出符号的名称。

它使用 MetaNameValueStr 元项属性句法指定符号名。

例子：

```
#[export_name = "exported_symbol_name"]
pub fn name_in_rust() { }
```

链接

Rust 编译器支持多种将 crate 链接起来使用的方法，这些链接方法可以是静态的，也可以是动态的。

编译器可以通过使用命令行参数或内部 crate_type 属性来生成多个构件(artifacts)。如果指定了一个或多个命令行参数，则将忽略所有 crate_type 属性，以便只构建由命令行指定的构件。

- --crate-type=bin 或 #[crate_type = "bin"] - 将生成一个可执行文件。这就要求在 crate 中有一个 main 函数，它将在程序开始执行时运行。这将链接所有 Rust 和本地依赖，生成一个单独的可分发的二进制文件。此类型为默认的 crate 类型。
- --crate-type=lib 或 #[crate_type = "lib"] - 将生成一个 Rust 库(library)。但最终会确切输出/生成什么类型的库在未生成之前还不好清晰确定，因为库有多种表现形式。使用 lib 这个通用选项的目的是生成“编译器推荐”的类型的库。像种指定输出库类型的选项在 rustc 里始终可用，但是每次实际输出的库的类型可能会随着实际情况的不同而不同。其它的输出（库的）类型选项都指定了不同风格的库类型，而 lib 可以看作是那些类型中的某个类型的别名（具体实际的输出的类型是编译器决定的）。
- --crate-type=rlib 或 #[crate_type = "rlib"] - 将生成一个“Rust 库”。它被用作一个中间构件，可以被认为是一个“静态 Rust 库”。与 staticlib 类型的库文件不同，这些 rlib 类型的库文件以后会作为其他 Rust 代码文件的上游依赖，未来对那些 Rust 代码文件进行编译时,那时的编译器会链并解释此 rlib 文件。这本质上意味着(那时的) rustc 将在(此) rlib 文件中查找元数据(metadata)，就像在动态库中查找元数据一样。跟 staticlib 输出类型类似，这种类型的输出常配合用于生成静态链接的可执行文件(statically linked executable)。
- --crate-type=dylib 或 #[crate_type = "dylib"] - 将生成一个动态 Rust 库。这与 lib 选项的输出类型不同，因为这个选项会强制生成动态库。生成的动态库可以用作其他库和/或可执行文件的依赖。这种输出类型将创建依赖于具体平台的库(Linux 上为 *.so，macOS 上为 *.dylib、Windows 上为 *.dll)。
- --crate-type=staticlib 或 #[crate_type = "staticlib"] - 将生成一个静态系统库。这个选项与其他选项的库输出的不同之处在于——当前编译器永远不会尝试去链接此 staticlib 输出 1。此选项的目的是创建一个包含所有本地 crate 的代码以及所有上游依赖的静态库。此输出类型将在 Linux、macOS 和 Windows(MinGW) 平台上创建 *.a 归档文件(archive)，或者在 Windows(MSVC) 平台上创建 *.lib 库文件。在这些情况下，例如将 Rust 代码链接到现有的非 Rust 应用程序中，推荐使用这种类型，因为它不会动态依赖于其他 Rust 代码。
- --crate-type=cdylib 或 #[crate_type = "cdylib"] - 将生成一个动态系统库。如果编译输出的动态库要被另一种语言加载使用，请使用这种编译选项。这种选项的输出将在 Linux 上创建 *.so 文件，在 macOS 上创建 *.dylib 文件，在 Windows 上创建 *.dll 文件。

- --crate-type=proc-macro 或 #[crate_type = "proc-macro"] - 生成的输出类型没有被指定，但是如果通过 -L 提供了路径参数，编译器将把输出构件识别为宏，输出的宏可以被其他 Rust 程序加载使用。使用此 crate 类型编译的 crate 只能导出过程宏。编译器将自动设置 proc_macro 属性配置选项。编译 crate 的目标平台(target)总是和当前编译器所在平台一致。例如，如果在 x86_64 CPU 的 Linux 平台上执行编译，那么目标将是 x86_64-unknown-linux-gnu，即使该 crate 是另一个不同编译目标的 crate 的依赖。

这些选项是可堆叠使用的，如果同时使用了多个选项，那么编译器将一次生成所有这些选项关联的的输出类型，而不必反复多次编译。但是，命令行和内部 crate_type 属性配置不能同时起效。如果只使用了不带属性值的 crate_type 属性配置，则将生成所有类型的输出，但如果同时指定了一个或多个 --crate-type 命令行参数，则只生成这些指定的输出。

依赖链接

对于所有这些不同类型的输出，如果 crate A 依赖于 crate B，那么整个系统中很可能有多种不同形式的 B，但是，编译器只会查找 rlib 类型的和动态库类型的。

有了依赖库的这两个选项，编译器在某些时候还是必须在这两种类型之间做出选择。考虑到这一点，编译器在决定使用哪种依赖关系类型时将遵循以下规则：

- 如果当前生成静态库，则需要所有上游依赖都以 rlib 类型可用。这个需求源于不能将动态库转换为静态类型的原因。注意，不可能将本地动态依赖链接到静态库，在这种情况下，将打印有关所有未链接的本地动态依赖的警告。
- 如果当前生成 rlib 文件，则对上游依赖的可用类型没有任何限制，仅需要求所有这些文件都可以从其中读出元数据。原因是 rlib 文件不包含它们的任何上游依赖。但如果所有的 rlib 文件都包含一份 libstd.rlib 的副本，那编译效率和执行效率将大幅降低。
- 如果当前生成可执行文件，并且没有指定 -C prefer-dynamic 参数，则首先尝试以 rlib 类型查找依赖。如果某些依赖在 rlib 类型文件中不可用，则尝试动态链接（见下文）。
- 如果当前生成动态链接的动态库或可执行文件，则编译器将尝试协调从 rlib 或 dylib 类型的文件里获取可用依赖关系，以创建最终产品。

编译器的主要目标是确保任何一个库不会在任何构件中出现多次。例如，如果动态库 B 和 C 都静态地去链接了库 A，那么当前 crate 就不能同时链接到 B 和 C，因为 A 有两个副本。编译器允许混合使用 rlib 和 dylib 类型，但这一限制必须被满足。

编译器目前没有实现任何方法来提示库应该链接到哪种类型的库。当选择动态链接时，编译器将尝试最大化动态依赖，同时仍然允许通过 rlib 类型链接某些依赖。

对于大多数情况，如果所有的可用库都是 dylib 类型的动态库，则推荐选择动态链接。对于其他情况，如果编译器无法确定一个库到底应该去链接它的哪种类型的版本，则会发布警告。

通常，--crate-type=bin 或 --crate-type=lib 应该足以满足所有的编译需求，只有在需要对 crate 的输出类型进行更细粒度的控制时，才需要使用其他选项。

静态 C runtime 和动态 C runtime

一般来说，标准库会同时尽力支持编译目标的静态链接型 C runtime 和动态链接型 C runtime。

例如，目标 x86_64-pc-windows-msvc 和 x86_64-unknown-linux-musl 通常都带有 C runtime，用户可以按自己的偏好去选择静态链接或动态链接到此 runtime。编译器中所有的编译目标都有一个链接到 C runtime 的默认模式。默认情况下，常见的编译目标都默认是选择动态链接的，但也存在默认情况下是静态链接的情况，例如：

- arm-unknown-linux-musleabi
- arm-unknown-linux-musleabihf
- armv7-unknown-linux-musleabihf
- i686-unknown-linux-musl
- x86_64-unknown-linux-musl

C runtime 的链接类型被配置为通过 crt-static 目标特性值来开启。这些目标特性通常是从命令行通过命令行参数传递给编译器来设置的。例如，要启用静态 C runtime，应该执行：

```
rustc -C target-feature=+crt-static foo.rs
```

如果想动态链接到 C runtime，应该执行：

```
rustc -C target-feature=-crt-static foo.rs
```

crate 本身也可以检测如何链接 C runtime。例如，MSVC 平台上的代码需要根据链接 C runtime 的方式进行差异性的编译（例如选择使用 /MT 或 /MD）。目前可通过 cfg 属性的 target_feature 选项导出检测结果：

```
#[cfg(target_feature = "crt-static")]
fn foo() {
    println!("C 运行时应该被静态链接");
}

#[cfg(not(target_feature = "crt-static"))]
fn foo() {
    println!("C 运行时应该被动态链接");
```

```
}
```

还请注意，Cargo 构建脚本可以通过环境变量来检测此特性。在构建脚本中，您可以通过如下代码检测链接类型：

```
use std::env;

fn main() {
    let linkage = env::var("CARGO_CFG_TARGET_FEATURE").unwrap_or(String::new());

    if linkage.contains("crt-static") {
        println!("C 运行时应该被静态链接");
    } else {
        println!("C 运行时应该被动态链接");
    }
}
```

要在本地使用此特性，通常需要使用 RUSTFLAGS 环境变量通过 Cargo 来为编译器指定参数。例如，要在 MSVC 平台上编译静态链接的二进制文件，需要执行：

```
RUSTFLAGS='-C target-feature=+crt-static' cargo build --target x86_64-pc-windows-msvc
```

Rust Runtime

panic_handler 属性

panic_handler 属性只能应用于签名为 fn(&PanicInfo) -> ! 的函数。有此属性标记的函数定义了发生 panic 时的行为。核心库内的结构体 PanicInfo 可以收集 panic 发生点的一些信息。在二进制、dylib 或 cdylib 类型的 crate 的依赖关系图中必须有一个 panic_handler 函数。

下面展示了一个 panic_handler 函数，它记录(log) panic 消息，然后终止(halts)线程：

```
#![no_std]

use core::fmt::{self, Write};
use core::panic::PanicInfo;

struct Sink {
    // ..
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    let mut sink = Sink::new();

    // logs "panicked at '$reason', src/main.rs:27:4" to some `sink`
    let _ = writeln!(sink, "{}", info);

    loop {}
}
```

标准库提供了 panic_handler 的一个实现，它的默认设置是展开堆栈，但也可以更改为中止(abort)进程。标准库的 panic 行为可以使用 set_hook 函数在运行时里去修改。

global_allocator 属性

在实现 GlobalAlloc trait 的静态项上使用 global_allocator 属性来设置全局分配器。

windows_subsystem 属性

当为一个 Windows 编译目标配置链接属性时，windows_subsystem 属性可以用来在 crate 级别上配置子系统类别。

它使用 MetaNameValueStr 元项属性语法用 console 或 windows 两个可行值指定子系统。对于非 windows 编译目标和非二进制的 crate 类型，该属性将被忽略。

例子：

```
#![windows_subsystem = "windows"]
```