简介

中文翻译注（Chinese translation of the The Rust Reference）：

1. 《Rust 参考手册》(The Rust Reference 中文版) 翻译自 The Rust Reference，查看此书的 Github 翻译项目。
2. 《Rust 参考手册》中文版已经有由 minstrel 翻译完整的版本，故 *rust-lang-cn* 不再单独翻译，而直接采用 minstrel 翻译的内容，若对翻译内容改正和完善，可到 minstrel 维护的仓库上提出和 PR。在此，*rust-lang-cn* 对译者表示衷心的感谢，您的翻译对我们使用中文资料学习和了解 Rust 起到极大作用。
3. 许可协议：中文翻译版（位于 *src* 目录）的版权我们也遵循原仓库的木兰宽松许可证，本仓库下的英文版内容以及其他文件均保持 Rust 官方的原有授权协议（即 MIT 协议和 Apache 2.0 协议）。
4. 本站支持文档中英文切换，点击页面右上角语言图标可切换到相同章节的英文页面。
5. *rust-lang-cn* 只会对格式和部分内容进行微调，其余内容我们建议大家都反馈到原仓库。若发现本页表达错误或帮助我们改进翻译，可点击右上角的编辑按钮打开本页对应源码文件进行编辑和修改，Rust 中文资源的开源组织发展离不开大家，感谢您的支持和帮助！

本书是 Rust 编程语言的主要参考手册，本书提供了3类资料：

- 一些章节非正式地介绍了该语言的各种语言结构及其用法。
- 一些章节非正式地介绍了该语言的内存模型、并发模型、运行时服务、链接模型，以及调试工具。-
- 附录章节提供了一些对 Rust 语言有影响的编程原理和参考资料。

⚠️ 警告：此书尚未完成，记录 Rust 的所有内容需要花些时间。有关本书未记录的内容，请查阅 GitHub issues。

Rust 发行版

Rust 每六周发布一种新的版本。 该语言的第一个稳定版本是 Rust 1.0.0，然后是 Rust 1.1.0，以此类推。 相应的工具（`rustc`、'`cargo`，等）和文档(标准库、本书，等等)与语言版本一起

发布。

本书的最新版本，与最新的 Rust 版本相匹配，它总是在 https://doc.rust-lang.org/reference/ 等你。

通过在此链接的“reference”路径段之前添加相应的 Rust版本号，可以找到以往的版本。 比如，Rust 1.49.0版在 https://doc.rust-lang.org/1.49.0/reference/ 下。

参考手册 并非——

这本书不是对这门语言的入门介绍。本书假设您熟悉该语言。若您需要学习该语言的基础知识，请阅读 Rust程序设计语言。

这本书也不作为 Rust 语言发行版中包含的标准库的参考资料。Rust 的库文档是从其源代码文件中提取的文档属性。此外，有许多可能被可能认为是语言自带特性(features)的特性其实都是 Rust 的标准库的特性，所以您要寻找的特性可能在那里，而不是在这里。

类似地，本书通常不能作为记录 rustc 或者 Cargo 细节的工具书。rustc 有自己专门的书 rustc book，Cargo 也有一本书 cargo book，该书中包含了 Cargo 的[参考手册] cargo reference。本书也涉及了少量和它们的相关知识，比如链接的章节，介绍了 rustc 是如何工作的。

本书仅作为稳定版 Rust 的参考资料存在，关于尚在开发中的非稳定特性，请参阅 Unstable Book。

Rust编译器（包括 `rustc`）将执行编译优化，但本参考手册不指导这些编译器的优化工作，所以只能把编译后的程序看作一个黑盒子。如果想侦测它的优化方式，只能通过运行它、给它提供输入并观察它的输出来推断。所有的编译器优化结果和程序的运行效果都必须和本参考手册所说的保持一致。

最后，本书并非 Rust 语言规范。它可能包含特定于 `rustc` 的细节，这不应该被当作 Rust 语言的规范。我们打算以后出版这样一本书，但在那之前，本手册是最接近的东西。

如何使用此书

本书不会假定您是按顺序阅读本书。本书的每一章一般都可以独立阅读，但会交叉链接到其他章节，以了解它们所相关的内容，但不会进行讨论。

阅读本书有两种主要方式。

第一是寻找特定问题的答案。如果您知道回答问题的章节，您可以直接从目录跳入该章节进行阅读。否则，您可以按 `s` 键或单击顶部栏上的放大镜来搜索与问题相关的关键字（译者注：目前本翻译版还不支持这种搜索功能）。例如，假设您想知道在 `let` 语句中创建的临时值何时被销毁；同时，假设您还不知道表达式那一章中定义了临时对象的生存期，那么可以搜索 “temporary let”，第一个搜索结果将带您去阅读该部分。

第二是提高您对此语言某一方面的认知。在这种情况下，只需浏览目录，直到看到您想了解的内容，然后点开阅读。阅读中如果某个链接看起来很有帮助，那就点击并阅读该部分内容。

也就是说，本书没有错误的阅读方式。您觉得怎样读对您最有帮助就怎样读。

约定

像所有技术书籍一样，在如何展示信息方面，本书有一些约定。这些约定记录如下。

- 定义术语的语句中，术语写为*斜体*。当术语在其定义章节之外使用时，通常会有一个链接来指向该术语定义的章节。

  示例术语 这是一个定义术语的示例。

- 编译 crate 所使用的版本之间的语言差异用一个块引用表示，以**粗体**的“版本差异：”开头。

  > **版本差异**：此句法(syntax)在 2015 版本有效，2018 版本起不允许使用。

- 一些有用信息，比如有关本书状态，或指出有用但大多超出本书范围的信息，大都位于以**粗体**的“注：”开头的块注释中。

  > **注**：这是一个注释示例。

- 有关对语言的不健全(sound)行为，或者针对易于混淆的语言特性的警告，记录在特殊的警告框里。

  > ⚠ 警告：这是一个示例警告。

- 文本中内联的代码片段在 `<code>` 标签里。

较长的代码示例放在突出显示句法(syntax)的框中，该框的右上角有用于复制、执行和显示隐藏行的控件

```
fn main() {
    println!("这是一段示例代码。");
}
```

- 文法和词法结构放在块引用中，第一行为粗体上标的 **词法** 或 **句法**。

> **句法**
>
> *ExampleGrammar*:
>
> `~` *Expression*
>
> | `box` *Expression*

查阅表义符(notation)以获取更多细节。

参与贡献

我们欢迎各种形式的贡献。

您可以通过开启议题或向 Rust 参考手册仓库发送 PR 来为本书做出贡献。如果这本书没有回答您的问题，并且您认为它的答案应该在本书的范围内，请不要犹豫，提交议题或在 Zulip 的 `t-lang/doc` 流频道上询问。知道人们最喜欢用这本书来做什么将有助于引导我们的注意力来让这些部分变得更好。我们也希望此手册尽可能地规范，所以如果你看到任何错误或非规范的地方，但没有明确指出，也请[提交议题]。

表义符/符号

notation.md
commit: dd1b9c331eb14ea7047ed6f2b12aaadab51b41d6
本章译文最后维护日期：2020-11-7

文法/语法

下表中的各种表义符会在本书中标有 *词法* 和 *句法* 的文法片段中用到：

表义符	示例	释义
CAPITAL	KW_IF, INTEGER_LITERAL	由词法分析生成的单一 token
ItalicCamelCase	*LetStatement*, *Item*	句法产生式(syntactical production)
`string`	`x`, `while`, `*`	确切的字面字符(串)
\x	\n, \r, \t, \0	转义字符
x?	`pub`?	可选项
x*	*OuterAttribute**	x 重复零次或多次
x+	*MacroMatch*+	x 重复一次或多次
xa..b	HEX_DIGIT1..6	x 重复 a 到 b 次
|	`u8` | `u16`, Block | Item	或
[ ]	[`b` `B`]	列举的任意字符
[-]	[`a` - `z`]	a 到 z 范围内的任意字符(包括 a 和 z)
~[ ]	~[`b` `B`]	列举范围外的任意字符(序列)
~ `string`	~ `\n`, ~ `*/`	此字符序列外的任意字符(序列)
( )	(, *Parameter*)?	程序项分组

字符串表产生式

文法中的一些规则 — 比如一元运算符，二元运算符和关键字 — 会以简化形式：作为可打印字符串的列表形式（在本书的相关章节的头部的各种产生式定义/句法规则里）给出。这些规则构成了关于 token规则的规则子集，并且它们被假定为源码编译时的词法分析阶段的结果被再次输入给解析器，然后由一个DFA驱动，对此字符串表里的所有条目(string table entries)进行析取(disjunction)操作（来进行句法分析）。

本书还约定，当文法表中出现如 `monospace` 这样的字符串时，它代表对这些产生式中的单一 token 成员的隐式引用。查阅 tokens 以获取更多信息。

词法结构

输入格式

notation.md
commit: a2405b970b7c8222a483b82213adcb17d646c75d
本章译文最后维护日期：2020-11-5

Rust 输入被解释为用 UTF-8 编码的 Unicode 字符序列。

关键字

keywords.md
commit: 6eb3e87af2c7743d6c7c783154cc380c4b0ea270 本章译文最后维护日期: 2021-04-23

Rust 将关键字分为三类:

- 严格关键字
- 保留关键字
- 弱关键字

严格关键字

这类关键字只能在正确的上下文中使用。它们不能用作以下名称:

- 程序项(item)
- 变量和函数参数
- 字段(field)和变体
- 类型参数
- 生存期参数或者循环标签
- 宏或属性
- 宏占位符
- crate

词法分析:

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`

以下关键字从 2018 版开始启用。

词法分析 2018+

KW_ASYNC : `async`
KW_AWAIT : `await`
KW_DYN : `dyn`

保留关键字

这类关键字目前还没有被使用，但是它们被保留以备将来使用。它们具有与严格关键字相同的限制。这样做的原因是通过禁止当前程序使用这些关键字，从而使当前程序能兼容 Rust 的未来版本。

词法分析

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`

以下关键字从 2018 版开始成为保留关键字。

词法分析 2018+

KW_TRY : `try`

弱关键字

这类关键字只有在特定的上下文中才有特殊的意义。例如，可以声明名为 `union` 的变量或方法。

- `macro_rules` 用于创建自定义宏。
- `union` 用于声明联合体(`union`)，它只有在联合体声明中使用时才是关键字。

- `'static` 用于静态生存期，不能用作通用泛型生存期参数和循环标签

```
// error[E0262]: invalid lifetime parameter name: `'static`
fn invalid_lifetime_parameter<'static>(s: &'static str) -> &'static str { s }
```

- 在 2015 版本中，当 `dyn` 用在非 `::` 开头的路径限定的类型前时，它是关键字。

从 2018 版开始，`dyn` 被提升为一个严格关键字。

词法分析

KW_UNION : `union`
KW_STATICLIFETIME : `'static`

词法分析 2015

KW_DYN : `dyn`

标识符

identifiers.md
commit: 28d628166940ffa982a1da552e0acfcc88ff8889
本章译文最后维护日期：2021-04-23

词法分析:

IDENTIFIER_OR_KEYWORD :
XID_start XID_continue*
| `_` XID_continue+

RAW_IDENTIFIER : `r#` IDENTIFIER_OR_KEYWORD *排除* `crate`, `self`, `super`, `Self`

NON_KEYWORD_IDENTIFIER : IDENTIFIER_OR_KEYWORD *排除严格关键字和保留关键字 *

IDENTIFIER :
NON_KEYWORD_IDENTIFIER | RAW_IDENTIFIER

标识符是如下形式的任何非空 Unicode 字符串：

要么是:

- 首字符拥有 `XID_start` 字符属性。
- 其余字符拥有 `XID_continue` 字符属性。

要么是：

- 首字符是 `_`。
- 整个字符串由多个字符组成。单个 `_` 不是有效标识符。
- 其余字符拥有 `XID_continue` 字符属性。

注意： `XID_start` 和 `XID_continue` 作为字符属性涵盖了用于构成常见的 C 和 Java语言族标识符的字符范围。

除了有形式前缀 `r#` 修饰外，原生标识符(raw identifier)与普通标识符类似。（注意形式前缀

`r#` 不包括在实际标识符中。）与普通标识符不同，原生标识符可以是除上面列出的 `RAW_IDENTIFIER` 之外的任何严格关键字或保留关键字。

注释

comments.md
commit: 993393d362cae51584d580f86c4f38d43ae76efc
本章译文最后维护日期：2020-10-17

词法分析

LINE_COMMENT :(译者注：行注释)
`/*` (~[`*` `!`] | `**` | *BlockCommentOrDoc*) | `//`

BLOCK_COMMENT :(译者注：块注释)
`/*` (~[`*` `!`] | `**` | *BlockCommentOrDoc*) (*BlockCommentOrDoc* | ~`*/`)* `*/`
| `/**/`
| `/***/`

INNER_LINE_DOC :(译者注：内部行文档型注释)
`//!` ~[`\n` *IsolatedCR*]*

INNER_BLOCK_DOC :(译者注：内部块文档型注释)
`/*!` (*BlockCommentOrDoc* | ~[`*/` *IsolatedCR*])* `*/`

OUTER_LINE_DOC :(译者注：外部行文档型注释)
`///` (~ `/` ~[`\n` *IsolatedCR*]*)?

OUTER_BLOCK_DOC :(译者注：外部块文档型注释)
`/**` (~ `*` | *BlockCommentOrDoc*) (*BlockCommentOrDoc* | ~[`*/` *IsolatedCR*])* `*/`

BlockCommentOrDoc :(译者注：块注释或文档型注释)
BLOCK_COMMENT
| OUTER_BLOCK_DOC
| INNER_BLOCK_DOC

IsolatedCR :
后面没有跟 `\n` *的* `\r`

非文档型注释

Rust 代码中的注释一般遵循 C++ 风格的行（`//`）和块（`/* ... */`）注释形式，也支持嵌套的块注释。

非文档型注释(Non-doc comments)被解释为某种形式的空白符。

文档型注释

以三个斜线（`///`）开始的行文档型注释，以及块文档型注释（`/** ... */`），均为内部文档型注释。它们被当做 [`doc` 属性][`doc` attributes]的特殊句法解析。也就是说，它们等同于把注释内容写入 `#[doc="..."]` 里。例如：`/// Foo` 等同于 `#[doc="Foo"]`，`/** Bar */` 等同于 `#[doc="Bar"]`。

以 `//!` 开始的行文档型注释，以及 `/*! ... */` 形式的块文档型注释属于注释体所在对象的文档型注释，而非注释体之后的程序项的。也就是说，它们等同于把注释内容写入 `#![doc="..."]` 里。`//!` 注释通常用于标注模块位于的文件。

孤立的 CRs（`\r`），如果其后没有紧跟有 LF（`\n`），则不能出现在文档型注释中。

示例

```
//! 应用于此 crate 的隐式匿名模块的文档型注释

pub mod outer_module {

    //!  - 内部行文档型注释
    //!! - 仍是内部行文档型注释（但是这样开头会更具强调性）

    /*!  - 内部块文档型注释 */
    /*!! - 仍是内部块文档型注释（但是这样开头会更具强调性）*/

    //   - 普通注释
    ///  - 外部行文档型注释（以 3 个 `///` 开始）
    //// - 普通注释

    /*   - 普通注释 */
```

```
    /**  - 外部块文档型注释 (exactly) 2 asterisks */
    /*** - 普通注释 */

    pub mod inner_module {}

    pub mod nested_comments {
        /* 在 Rust 里 /* 我们可以 /* 嵌套注释 */ */ */

        // 所有这三种类型的块注释都可以包含或嵌套在任何其他类型的
        // 注释中:

        /*   /* */  /** */  /*! */  */
        /*!  /* */  /** */  /*! */  */
        /**  /* */  /** */  /*! */  */
        pub mod dummy_item {}
    }

    pub mod degenerate_cases {
        // 空内部行文档型注释
        //!

        // 空内部块文档型注释
        /*!*/

        // 空行注释
        //

        // 空外部行文档型注释
        ///

        // 空块注释
        /**/

        pub mod dummy_item {}

        // 空的两个星号的块注释不是一个文档块，它是一个块注释。
        /***/

    }

    /* 下面这个是不允许的，因为外部文档型注释需要一个
       接收该文档的程序项 */

    /// 我的程序项呢?
}
```

空白符

whitespace.md
commit: dd1b9c331eb14ea7047ed6f2b12aaadab51b41d6
本章译文最后维护日期：2020-11-5

空白符是非空字符串，它里面只包含具有 `Pattern_White_Space` 属性的 Unicode 字符，即:

- `U+0009` (水平制表符, `'\t'`)
- `U+000A` (换行符, `'\n'`)
- `U+000B` (垂直制表符)
- `U+000C` (分页符)
- `U+000D` (回车符, `'\r'`)
- `U+0020` (空格符, `' '`)
- `U+0085` (下一行标记符)
- `U+200E` (从左到右标记符)
- `U+200F` (从右到标左记符)
- `U+2028` (行分隔符)
- `U+2029` (段分隔符)

Rust是一种“格式自由(free-form)”的语言，这意味着所有形式的空白符在文法中仅用于分隔 *tokens* 的作用，没有语义意义。

Rust 程序中，如果将一个空白符元素替换为任何其他合法的空白符元素（例如单个空格字符），它们仍有相同的意义。

token

token 是采用非递归方式的正则文法(regular languages)定义的基本语法产生式(primitive productions)。Rust 源码输入可以被分解成以下几类 token：

- 关键字
- 标识符
- 字面量
- 生存期
- 标点符号
- 分隔符

在本文档中，“简单”token 会直接在（相关章节头部的）[字符串表产生式(production)][string table production]表单中给出，并以 `monospace` 字体显示。（译者注：本译作的原文中，在文法表之外的行文中也会大量出现这种直接使用简单token 来替代相关名词的做法，一般此时如果译者觉得这种 token 需要翻译时，会使用诸如：结构体(`struct`) 这种形式来翻译。读者需要意识到“struct”是文法里的一个 token，能以其字面形式直接出现在源码里。）

字面量

字面量是一个由单一 token（而不是由一连串 tokens）组成的表达式，它立即、直接表示它所代表的值，而不是通过名称或其他一些求值/计算规则来引用它。字面量是常量表达式的一种形式，所以它（主要）用在编译时求值。

示例

字符和字符串

	举例	`#` 号的数量	字符集	转义
字符	`'H'`	0	全部 Unicode	引号 & ASCII & Unicode
字符串	`"hello"`	0	全部 Unicode	引号 & ASCII & Unicode
原生字符	`r#"hello"#`	0 或更多	全部	`N/A`

串		*	Unicode	
字节	`b'H'`	0	全部 ASCII	引号 & 字节
字节串	`b"hello"`	0	全部 ASCII	引号 & 字节
原生字节串	`br#"hello"#`	0 或更多 *	全部 ASCII	`N/A`

* 字面量两侧的 `#` 数量必须相同。

ASCII 转义

名称	
`\x41`	7-bit 字符编码（2位数字，最大值为 `0x7F`）
`\n`	换行符
`\r`	回车符
`\t`	制表符
`\\`	反斜线
`\0`	Null

字节转义

名称	
`\x7F`	8-bit 字符编码（2位数字）
`\n`	换行符
`\r`	回车符
`\t`	制表符
`\\`	反斜线
`\0`	Null

unicode 转义

名称	
`\u{7FFF}`	24-bit Unicode 字符编码（最多6个数字）

引号转义

	Name
`\'`	单引号
`\"`	双引号

数字

数字字面量 *	示例	指数	后缀
十进制整数	`98_222`	`N/A`	整数后缀
十六进制整数	`0xff`	`N/A`	整数后缀
八进制整数	`0o77`	`N/A`	整数后缀
二进制整数	`0b1111_0000`	`N/A`	整数后缀
浮点数	`123.0E+77`	`Optional`	浮点数后缀

`*` 所有数字字面量允许使用 `_` 作为可视分隔符，比如：`1_234.0E+18f64`

后缀

后缀是紧跟（无空白符）在字面量主体部分之后的非原生标识符(non-raw identifier)。

任何带有后缀的字面量（如字符串、整数等）都可以作为有效的 token，并且可以传递给宏而不会产生错误。宏自己决定如何解释这种 token，以及是否该报错。

```
macro_rules! blackhole { ($tt:tt) => () }

blackhole!("string"suffix); // OK
```

但是，最终被解析为 Rust 代码的字面量token 上的后缀是受限制的。对于非数字字面量 token，任何后缀都最终将被弃用，而数字字面量token 只接受下表中的后缀。

整数	浮点数
`u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `u128`, `i128`, `usize`, `isize`	`f32`, `f64`

字符和字符串字面量

字符字面量

词法

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 `_`*)$^{1..6}$ `}`

*字符字面量*是位于两个 `U+0027`（单引号 `'`）字符内的单个 Unicode 字符。当它是 `U+0027` 自身时，必须前置*转义*字符 `U+005C`（`\`）。

字符串字面量

词法

STRING_LITERAL :
`"` (
~[`"` `\` *IsolatedCR*] (译者注：IsolatedCR：后面没有跟 `\n` 的 `\r`，首次定义见注释)
| QUOTE_ESCAPE
| ASCII_ESCAPE
| UNICODE_ESCAPE
| STRING_CONTINUE
)* `"`

STRING_CONTINUE :
`\` *后跟* \n

*字符串字面量*是位于两个 `U+0022`（双引号 `"`）字符内的任意 Unicode 字符序列。当它是 `U+0022` 自身时，必须前置*转义*字符 `U+005C`（`\`）。

字符串字面量允许换行书写。换行可以用换行符（`U+000A`）表示，也可以用一对回车符换行符（`U+000D`, `U+000A`）的字节序列表示。这两种字节序列通常都会被转换为 `U+000A`，但有例外：当换行符前置一个未转义的字符 `U+005C`（`\`）时，会导致字符 `U+005C`、换行符和下一行开头的所有空白符都被忽略。因此下述示例中，`a` 和 `b` 是一样的：

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

assert_eq!(a,b);
```

字符转义

不管是字符字面量，还是非原生字符串字面量，Rust 都为其提供了额外的*转义*功能。转义以一个 `U+005C`（`\`）开始，并后跟如下形式之一：

- *7-bit 码点转义*以 `U+0078`（`x`）开头，后面紧跟两个*十六进制数字*，其最大值为 `0x7F`。它表示 ASCII 字符，其码值就等于字面提供的十六进制值。不允许使用更大的值，因为不能确定其是 Unicode 码点还是字节值(byte values)。
- *24-bit 码点转义*以 `U+0075`（`u`）开头，后跟多达六位*十六进制数字*，位于花括号 `U+007B`（`{`）和 `U+007D`（`}`）之间。这表示（需转义到的）Unicode 字符的码点等于花括号里的十六进制值。
- *空白符转义*是 `U+006E`（`n`）、`U+0072`（`r`）或者 `U+0074`（`t`）之一，依次分别表示 Unicode 码点 `U+000A`（LF），`U+000D`（CR），或者 `U+0009`（HT）。
- *null转义* 是字符 `U+0030`（`0`），表示 Unicode 码点 `U+0000`（NUL）。
- *反斜线转义* 是字符 `U+005C`（`\`），反斜线必须通过转义才能表示其自身。

原生字符串字面量

词法

RAW_STRING_LITERAL :
  `r` RAW_STRING_CONTENT

RAW_STRING_CONTENT :
    `"` (~ *IsolatedCR*)* (非贪婪模式) `"`
  | `#` RAW_STRING_CONTENT `#`

原生字符串字面量不做任何转义。它以字符 `U+0072`（`r`）后跟零个或多个字符 `U+0023`（`#`），以及一个字符 `U+0022`（双引号 `"`），这样的字符组合开始；中间*原生字符串文本主体*部分可包含任意的 Unicode 字符序列；再后跟另一个 `U+0022`（双引号 `"`）字符表示文本主体结束；最后再后跟与文本主体前的那段字符组合中的同等数量的 `U+0023`（`#`）字符。

所有包含在原生字符串文本主体中的 Unicode 字符都代表他们自身：字符 `U+0022`（双引号 `"`）（除非后跟的纯 `U+0023`（`#`)字符串与文本主体开始前的对称相等）或字符 `U+005C`（`\`）此时都没有特殊含义。

字符串字面量示例:

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

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

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

字节和字节串字面量

字节字面量

词法

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

ASCII_FOR_CHAR :
  任何 ASCII 字符 （0x00 到 0x7F），排除 `'`, `\`, \n, \r 或者 \t

BYTE_ESCAPE :
    `\x` HEX_DIGIT HEX_DIGIT
  | `\n` | `\r` | `\t` | `\\` | `\0`

*字节字面量*是单个 ASCII 字符（码值在 `U+0000` 到 `U+007F` 区间内）或一个*转义字节*作为字节字面量的真实主体跟在表示形式意义的字符 `U+0062`（`b`）和字符 `U+0027`（单引号 `'`）组合

之后，然后再后接字符 `U+0027`。如果字符 `U+0027` 本身要出现在字面量中，它必须经由前置字符 `U+005C`（`\`）*转义*。字节字面量等价于一个 `u8` 8-bit 无符号整型*数字字面量*。

字节串字面量

词法

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

ASCII_FOR_STRING :
任何 ASCII 字符(码值位于 0x00 到 0x7F 之间), 排除 `"`, `\` *和 IsolatedCR*

非原生*字节串字面量*是 ASCII 字符和转义字符组成的字符序列，形式是以字符 `U+0062`（`b`）和字符 `U+0022`（双引号 `"`）组合开头，以字符 `U+0022` 结尾。如果字面量中包含字符 `U+0022`，则必须由前置的 `U+005C`（`\`）*转义*。此外，字节串字面量也可以是*原生字节串字面量*（下面有其定义）。长度为 `n` 的字节串字面量类型为 `&'static [u8; n]`。

一些额外的*转义*可以在字节或非原生字节串字面量中使用，转义以 `U+005C`（`\`）开始，并后跟如下形式之一：

- *字节转义*以 `U+0078`（`x`）开始，后跟恰好两位*十六进制数字*来表示十六进制值代表的字节。
- *空白符转义*是字符 `U+006E`（`n`）、`U+0072`（`r`），或 `U+0074`（`t`）之一，分别表示字节值 `0x0A`（ASCII LF）、`0x0D`（ASCII CR），或 `0x09`（ASCII HT）。
- *null转义*是字符 `U+0030`（`0`），表示字节值 `0x00`（ASCII NUL）。
- *反斜线转义*是字符 `U+005C`（`\`），必须被转义以表示其 ASCII 编码 `0x5C`。

原生字节串字面量

词法

RAW_BYTE_STRING_LITERAL :
`br` RAW_BYTE_STRING_CONTENT

RAW_BYTE_STRING_CONTENT :
`"` ASCII* (非贪婪模式) `"`
| `#` RAW_BYTE_STRING_CONTENT `#`

ASCII :
任何 ASCII 字符（0x00 到 0x7F）

原生字节串字面量不做任何转义。它们以字符 `U+0062`（`b`）后跟 `U+0072`（`r`），再后跟零个或多个字符 `U+0023`（`#`）及字符 `U+0022`（双引号 `"`），这样的字符组合开始；之后是*原生字节串文本主体*，这部分可包含任意的 ASCII 字符序列；后跟另一个 `U+0022`（双引号 `"`）字符表示文本主体结束；最后再后跟与文本主体前的那段字符组合中的同等数量的 `U+0023`（`#`）字符。原生字节串字面量不能包含任何非 ASCII 字节。

原生字节串文本主体中的所有字符都代表它们自身的 ASCII 编码，字符 `U+0022`（双引号 `"`）（除非后跟的纯 `U+0023`（`#`）字符串与文本主体开始前的对称相等）或字符 `U+005C`（`\`）此时都没有特殊含义。

字节串字面量示例：

```
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
```

数字字面量

*数字字面量*可以是*整型字面量*，也可以是*浮点型字面量*，识别这两种字面量的文法是混合在一起的。

整型字面量

词法

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`

---

*整型字面量*具备下述 4 种形式之一：

- *十进制字面量*以*十进制数字*开头，后跟*十进制数字*和*下划线(`_`)*的任意组合。
- *十六进制字面量*以字符序列 `U+0030` `U+0078`（`0x`）开头，后跟十六进制数字和下划线的任意组合（至少一个数字）。
- *八进制字面量*以字符序列 `U+0030` `U+006F`（`0o`）开头，后跟八进制数字和下划线的任意组合（至少一个数字）。
- *二进制字面量*以字符序列 `U+0030` `U+0062`（`0b`）开头，后跟二进制数字和下划线的任意组合（至少一个数字）。

与其它字面量一样，整型字面量后面可紧跟一个*整型后缀*，该后缀强制设定了字面量的数据类型。整型后缀须为如下整型类型之一：`u8`、`i8`、`u16`、`i16`、`u32`、`i32`、`u64`、`i64`、`u128`、`i128`、`usize` 或 `isize`。

*无后缀*整型字面量的类型通过类型推断确定：

- 如果整型类型可以通过程序上下文*唯*一确定，则无后缀整型字面量的类型即为该类型。
- 如果程序上下文对类型约束不足，则默认为 32-bit 有符号整型，即 `i32`。
- 如果程序上下文对类型约束过度，则报静态类型错误。

各种形式的整型字面量示例：

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

0xff;                               // 类型 i32
0xff_u8;                            // 类型 u8

0o70;                               // 类型 i32
0o70_i16;                           // 类型 i16

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

0usize;                             // 类型 usize
```

无效整型字面量示例:

```
// 无效后缀

0invalidSuffix;

// 数字进制错误

123AFB43;
0b0102;
0o0581;

// 类型溢出

128_i8;
256_u8;

// 二进制、十六进制、八进制的进制前缀后至少需要一个数字

0b_;
0b____;
```

请注意，Rust 句法将 `-1i8` 视为一元取反运算符对整型字面量 `1i8` 的应用，而不是将它视为单个整型字面量。

元组索引

词法

TUPLE_INDEX:
  INTEGER_LITERAL

元组索引用于引用元组、元组结构体和元组变体的字段。

元组索引直接与字面量token 进行比较。元组索引以 `0` 开始，每个后续索引的值以十进制的 `1` 递增。因此，元组索引只能匹配十进制值，并且该值不能用 `0` 做前缀字符。

```
let example = ("dog", "cat", "horse");
let dog = example.0;
let cat = example.1;
// 下面的示例非法.
let cat = example.01;  // 错误: 没有 `01` 字段
let horse = example.0b10;  // 错误: 没有 `0b10` 字段
```

注意: 元组索引可能包含一个 `INTEGER_SUFFIX` ，但是这不是有效的，可能会在将来的版本中被删除。更多信息请参见https://github.com/rust-lang/rust/issues/60210。

浮点型字面量

词法

FLOAT_LITERAL :
    DEC_LITERAL `.` *（紧跟着的不能是 `.`, `_` 或者标识符）*
  | 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`

*浮点型字面量*有如下两种形式：

- *十进制字面量*后跟句点字符 `U+002E`（`.`）。后面可选地跟着另一个十进制数字，还可以再接一个可选的*指数*。
- *十进制字面量*后跟一个*指数*。

如同整型字面量，浮点型字面量也可后跟一个后缀，但在后缀之前，浮点型字面量部分不以 `U+002E`（`.`）结尾。后缀强制设定了字面量类型。有两种有效的*浮点型后缀*：`f32` 和 `f64`（32-bit 和 64-bit 浮点类型），它们显式地指定了字面量的类型。

- If the program context under-constrains the type, it defaults to `f64`.
- If the program context over-constrains the type, it is considered a static type error.
*无后缀*浮点型字面量的类型通过类型推断确定：
- 如果浮点型类型可以通过程序上下文*唯一*确定，则无后缀浮点型字面量的类型即为该类型。
- 如果程序上下文对类型约束不足，则默认为 `f64`。
- 如果程序上下文对类型过度约束，则报静态类型错误。

各种形式的浮点型字面量示例：

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

最后一个例子稍显不同，因为不能对一个以句点结尾的浮点型字面量使用后缀句法，`2.f64` 会尝试在 `2` 上调用名为 `f64` 的方法。

浮点数的表形(representation)语义在“和平台相关的类型”中有描述。

布尔型字面量

词法

BOOLEAN_LITERAL :
`true`
| `false`

布尔类型有两个值，写为：`true` 和 `false`。

生存期和循环标签

词法

LIFETIME_TOKEN :
`'` IDENTIFIER_OR_KEYWORD
| `'_`

LIFETIME_OR_LABEL :
`'` NON_KEYWORD_IDENTIFIER

生存期参数和循环标签使用 LIFETIME_OR_LABEL 类型的 token。（尽管 LIFETIME_OR_LABEL 是 LIFETIME_TOKEN 的子集，但）任何符合 LIFETIME_TOKEN 约定的 token 也都能被上述词法分析规则所接受，比如 LIFETIME_TOKEN 类型的 token 在宏中就可以畅通无阻的使用。

标点符号

为了完整起见，这里列出了（Rust 里）所有的标点符号的 symbol token。它们各自的用法和含义在链接页面中都有定义。

符号	名称	使用方法
`+`	Plus	算术加法, trait约束, 可匹配空的宏匹配器(Macro Kleene Matcher)
`-`	Minus	算术减法, 取反
`*`	Star	算术乘法, 解引用, 裸指针, 可匹配空的宏匹配器, use 通配符
`/`	Slash	算术除法
`%`	Percent	算术取模

`^`	Caret	位和逻辑异或
`!`	Not	位和逻辑非, 宏调用, 内部属性, never型, 否定实现
`&`	And	位和逻辑与, 借用, 引用, 引用模式
`|`	Or	位和逻辑或, 闭包, match 中的模式, `if let`, 和 `while let`
`&&`	AndAnd	短路与, 借用, 引用, 引用模式
`||`	OrOr	短路或, 闭包
`<<`	Shl	左移位, 嵌套泛型
`>>`	Shr	右移位, 嵌套泛型
`+=`	PlusEq	加法及赋值
`-=`	MinusEq	减法及赋值
`*=`	StarEq	乘法及赋值
`/=`	SlashEq	除法及赋值
`%=`	PercentEq	取模及赋值
`^=`	CaretEq	按位异或及赋值
`&=`	AndEq	按位与及赋值
`|=`	OrEq	按位或及赋值
`<<=`	ShlEq	左移位及赋值
`>>=`	ShrEq	右移位及赋值, 嵌套泛型
`=`	Eq	赋值, 属性, 各种类型定义
`==`	EqEq	等于
`!=`	Ne	不等于
`>`	Gt	大于, 泛型, 路径
`<`	Lt	小于, 泛型, 路径
`>=`	Ge	大于或等于, 泛型
`<=`	Le	小于或等于
`@`	At	子模式绑定
`_`	Underscore	通配符模式, 自动推断型类型, 常量项中的非命名程序项, 外部 crate, 和 use声明
`.`	Dot	字段访问, 元组索引
`..`	DotDot	区间, 结构体表达式, 模式
`...`	DotDotDot	可变参数函数, 区间模式
`..=`	DotDotEq	闭区间, 区间模式

`,`	Comma	各种分隔符
`;`	Semi	各种程序项和语句的结束符, 数组类型
`:`	Colon	各种分隔符
`::`	PathSep	[路径分隔符]路径
`->`	RArrow	函数返回类型, 闭包返回类型, 数组指针类型
`=>`	FatArrow	匹配臂, 宏
`#`	Pound	属性
`$`	Dollar	宏
`?`	Question	问号运算符, 非确定性尺寸, 可匹配空的宏匹配器

定界符

括号用于文法的各个部分，左括号必须始终与右括号配对。括号以及其内的 token 在宏中被称作“token树(token trees)”。括号有三种类型：

括号	类型
`{ }`	花/大括号
`[ ]`	方/中括号
`( )`	圆/小括号

宏

macros.md
commit: 012bfafbd995c54a86ebb542bbde5874710cba19
本章译文最后维护日期：2020-1-24

可以使用称被为宏的自定义句法形式来扩展 Rust 的功能和句法。宏需要被命名，并通过一致的句法去调用：`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。

Macro Invocation

宏调用

句法

MacroInvocation :
  SimplePath `!` *DelimTokenTree*

DelimTokenTree :
    `(` *TokenTree** `)`
  | `[` *TokenTree** `]`
  | `{` *TokenTree** `}`

TokenTree :
  Token *排除 定界符(delimiters)* | *DelimTokenTree*

MacroInvocationSemi :

SimplePath `!` `(` *TokenTree*`*` `)` `;`
| *SimplePath* `!` `[` *TokenTree*`*` `]` `;`
| *SimplePath* `!` `{` *TokenTree*`*` `}`

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
- 模式
- 类型
- 程序项，包括关联程序项
- `macro_rules` 转码器
- 外部块

当宏调用被用作程序项或语句时，此时它应用的 *MacroInvocationSemi* 句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(`macro_rules`)定义之前不允许使用可见性限定符。

```
// 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!();
```

声明宏

macros-by-example.md
commit: d23f9da8469617e6c81121d9fd123443df70595d
本章译文最后维护日期：2021-5-6

句法

MacroRulesDefinition :
`macro_rules` `!` IDENTIFIER *MacroRulesDef*

MacroRulesDef :
`(` *MacroRules* `)` `;`
| `[` *MacroRules* `]` `;`
| `{` *MacroRules* `}`

MacroRules :
MacroRule (`;` *MacroRule*)* `;`?

MacroRule :
MacroMatcher `=>` *MacroTranscriber*

MacroMatcher :
`(` *MacroMatch** `)`
| `[` *MacroMatch** `]`
| `{` *MacroMatch** `}`

MacroMatch :
Token *排除 $ 和 定界符*
| *MacroMatcher*
| `$` IDENTIFIER `:` *MacroFragSpec*
| `$` `(` *MacroMatch*+ `)` *MacroRepSep*? *MacroRepOp*

MacroFragSpec :
`block` | `expr` | `ident` | `item` | `lifetime` | `literal`
| `meta` | `pat` | `path` | `stmt` | `tt` | `ty` | `vis`

MacroRepSep :
  *Token*排除 *定界符* 和 *重复操作符*

MacroRepOp :
  `*` | `+` | `?`

MacroTranscriber :
  DelimTokenTree

`macro_rules` 允许用户以声明性的(declarative)方式定义句法扩展。我们称这种扩展形式为“声明宏（macros by example）”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)*，描述它匹配的句法；一个*转码器(transcriber)*，描述成功匹配后将执行的替代调用句法。匹配器和转码器都必须由定界符(delimiter)包围。宏可以扩展为表达式、语句、程序项（包括 trait、impl 和外来程序项）、类型或模式。

转码

当宏被调用时，宏扩展器(macro expander)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 `$` 用于从宏引擎中调用特殊行为（下文元变量和重复元中有详述）。不属于此类调用的 token 将按字面意义进行匹配和转码，除了一个例外。这个例外是匹配器的外层定界符将匹配任何一对定界符。因此，比如匹配器 `(())` 将匹配 `{()}`，而 `{{}}` 不行。字符 `$` 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句法树(opaque AST)。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，唯一可看到/使用的就是此匹配段类型一样的匹配段选择器(fragment specifier)。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：（译者注：匹配段选择器和匹配段，以及宏中各部件的定义可以凑合着看看译者未能翻译完成的宏定义规范）

```
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);
```

元变量

在匹配器中，`$` *名称* `:` *匹配段选择器* 这种句法格式匹配符合指定句法类型的 Rust 句法段，并将其绑定到元变量 `$` *名称* 上。有效的匹配段选择器包括：

- `item`：*程序项*
- `block`：*块表达式*
- `stmt`：*语句*，注意此选择器不匹配句尾的分号（如果匹配器中提供了分号，会被当做分隔符），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*

因为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（请参阅后面的卫生性(hygiene)章节）。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 `$(` … `)` 内，然后后跟一个重复运算符(repetition operator)，这两者之间可以放置一个可选的分隔符(separator token)。分隔符可以是除定界符或重复运算符之外的任何 token，其中分号(`;`)和逗号(`,`)最常见。例如：`$( $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。

重复运算符为：

- `*` — 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。

因为 `?` 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：

1. 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。

2. 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。

作用域、导出以及导入

由于历史原因，声明宏的作用域并不完全像各种程序项那样工作。宏有两种形式的作用域：文本作用域(textual scope)和基于路径的作用域(path-based scope)。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。（后本节面将进一步解释这个。）基于路径的作用域与其他程序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(unqualified identifier)（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。

```
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（通常是模块）结束时离开。文本作用域可以覆盖/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。首先，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!();
```

其次，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。虽然可以在导入语句之前使用 `#[macro_use]` 导入宏，但如果发生冲突，则最后导入的宏将胜出。可以使用可选的 *MetaListIdents*元项属性句法指定要导入的宏列表；当将 `#[macro_use]` 应用于模块上时，则不支持此指定操作。

```
#[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
```

要用 `#[macro_use]` 导入宏必须先使用 `#[macro_export]` 导出，下文会有讲解。

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。如果宏引用的程序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，可以替代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。

```
//// 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，因此在引用非宏程序项时，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() {}
}
```

此外，尽管 `$crate` 允许宏在扩展时引用其自身 crate 中的程序项，但它的使用对可见性没有影响（，或者说它的使用仍受可见性条件的约束）。引用的程序项或宏必须仍然在调用位置处可见。在下面的示例中，任何试图从此 crate 外部调用 `call_foo!()` 的行为都将失败，因为 `foo()` 不是公有的。

```
#[macro_export]
macro_rules! call_foo {
    () => { $crate::foo() };
}

fn foo() {}
```

（译者注：原文给出的这个例子是能正常调用的，原文并没有给出在 crate 外部调用的例子）

> **版本&版次差异**：在 Rust 1.30 之前，`$crate` 和 `local_inner_macros`（后面会讲）不受支持。从该版本开始，它们与基于路径的宏导入（前面讲过）一起被添加进来，用以确保不需要在当前 crate 已经使用导入了某导出宏的情况下再额外手动导入此导出宏下面用到的辅助宏。如果要让 Rust 的早期版本编写的 crate 要使用辅助宏，需要修改为使用 `$crate` 或 `local_inner_macros`，以便与基于路径的导入一起工作。

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。

```
#[macro_export(local_inner_macros)]
macro_rules! helped {
    () => { helper!() } // 自动转码为 $crate::helper!().
}

#[macro_export]
macro_rules! helper {
    () => { () }
}
```

随集歧义限制（译者注：该节还需要继续校对打磨，主要难点还是因为附录的宏定义规范译者还没有能全部搞懂）

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(nonterminal)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。

例如，像 `$i:expr [ , ]` 这样的宏匹配器在现今的 Rust 中理论上是可以接受的，因为现在 `[,]` 不可能是合法表达式的一部分，因此解析始终是明确的。但是，由于 `[` 可以开始一个尾随表达式(trailing expressions)，因此 `[` 不是一个可以安全排除在表达式后面出现的字符。如果在接下来的 Rust 版本中接受了 `[,]`，那么这个匹配器就会产生歧义或是错误解析，破坏正常代码。但是，像 `$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：（译者注：下面的规则不是绝对的，因为宏的基础理

论还在发展中。）

- `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。

  （译者注：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`}。注意这个列表也不一定全。）
- 其它所有的匹配段选择器没有限制。

当涉及到重复元时，随集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。

有关更多详细信息，请参阅[正式规范]。

过程宏

> procedural-macros.md
> commit: a1ef5a09c0281b0f2a65c18670e927ead61eb1b2
> 本章译文最后维护日期：2020-11-6

*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 句法进行操作的代码，它可以在消费掉一些 Rust 句法输入的同时产生新的 Rust 句法输出。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。

> **注意**: 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，它们要么有返回句法，要么触发 panic，要么永无休止地循环。返回句法根据过程宏的类型替换或添加句法；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。

The `proc_macro` crate

过程宏类型的 crate 几乎总是会去链接编译器提供的 `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流*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` 。

过程宏的卫生性

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` ）。

类函数过程宏

*类函数过程宏*是使用宏调用运算符（ `!` ）调用的过程宏。

这种宏是由一个带有 `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。

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());
}
```

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、程序项可以出现的位置（包括 `extern` 块里、固有(inherent)实现里和 trait实现里、以及 trait声明里）。

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流的情况下创建新程序项。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` 属性的程序项的 token流。输出 `TokenStream` 必须是一组程序项，然后将这组程序项追加到输入 `TokenStream` 中的那条程序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程序项的作用域中。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。也就是说所有该宏的宏应用都可以看到它们。

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。

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*定义可以附加到程序项上的新的外部属性，这些程序项包括外部(`extern`)块、固有实现、trate实现，以及 trait声明中的各类程序项。

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(delimited token tree)（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` 是程序项的其余部分，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程序项。

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
}
```

```rust
// 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() {}"
```

crate 和源文件

crates-and-source-files.md
commit: eabdf09207bf3563ae96db9d576de0758c413d5d
本章译文最后维护日期：2021-1-24

句法

Crate :

UTF8BOM?

SHEBANG?

*InnerAttribute**

*Item**

词法

UTF8BOM : `\uFEFF`

SHEBANG : `#!` ~ `\n`+†

注意：尽管像任何其他语言一样，Rust 也都可以通过解释器和编译器实现，但现在唯一存在的实现是编译器，并且该语言也是一直被设计为可编译的。因为这些原因，所以本章节所有的讨论都是基于编译器这条路径的。

Rust 的语义有编译时和运行时之间的*阶段差异(phase distinction)*。[1] 其中*静态解释*的语义规则控制编译的成败，而*动态解释*的语义规则控制程序在运行时的行为。

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。[2]

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 中的任何程序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程序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。

```
// 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。例如：

```
#!/usr/bin/env rustx

fn main() {
    println!("Hello!");
}
```

为了避免与属性混淆， Rust 对 shebang 句法做了一个限制：是 `#!` 字符不能后跟 token `[`，忽略中间的注释或空白符。如果违反此限制，则不会将其视为 shebang，而会将其视为属性的开始。

预导入包和 `no_std`

本节内容已经移入预导入包那章里了。

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`

注意: 允许哪些返回类型的实现是由暂未稳定的 `Termination` trait 决定的。

`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*，并通过使用 *MetaNameValueStr*元项属性句法来指定 crate 的名称。

```
#![crate_name = "mycrate"]
```

crate 名称不能为空，且只能包含 [Unicode字母数字]或字符 `-`(U+002D)。

[1] 这种区别也存在于解释器中。静态检查，如语法分析、类型检查和 lint检查，都应该在程序执行之前进行，而不要去管程序何时执行。

[2] crate 有点类似于 ECMA-335 CLI 模型中的 *assembly*、SML/NJ 编译管理器中的 *library*、Owens 和 Flatt 模块系统中的 *unit*， 或 Mesa 中的 *configuration*。

条件编译

conditional-compilation.md
commit: 949726950a4ac5c8674e902dc33c09b48fc7434c
本章译文最后维护日期：2021-5-6

句法

ConfigurationPredicate :
    ConfigurationOption
  | *ConfigurationAll*
  | *ConfigurationAny*
  | *ConfigurationNot*

ConfigurationOption :
  IDENTIFIER (`=` (STRING_LITERAL | RAW_STRING_LITERAL))?

ConfigurationAll
  `all` `(` *ConfigurationPredicateList*? `)`

ConfigurationAny
  `any` `(` *ConfigurationPredicateList*? `)`

ConfigurationNot
  `not` `(` *ConfigurationPredicate* `)`

ConfigurationPredicateList
  ConfigurationPredicate (`,` *ConfigurationPredicate*)* `,`?

根据某些条件， *条件性编译的源代码(Conditionally compiled source code)*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，以及下面将详细描述的一些其他事项。

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(configuration predicate)*。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。

注意: `=` 周围的空白符将被忽略。 `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。一些选项属于*编译器设置集(compiler-set)*，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(arbitrarily-set)*，这部分设置必须从代码之外传参给编译器来自主设置。无法在正在编译的 crate 的源代码中设置编译配置选项。

注意: 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。

注意: 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。

⚠️ 警告：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。但实际上这样做是不明智的。

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(target triple)[1]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"`

有关可用特性的更多细节，请参见 `target_feature` 属性。此外，编译器还为 `target_feature` 选项提供了一个额外的 `crt-static` 特性，它表示需要链接一个可用的静态C运行时。

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(pointer width in bits)。

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。请参阅测试章节来获取更多和测试支持相关的信息。

`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

句法

CfgAttrAttribute :
`cfg` `(` *ConfigurationPredicate* `)`

`cfg` 属性根据配置谓词有条件地包括它所附加的东西。

它被写成 `cfg` 后跟一个 `(`，再跟一个配置谓词，最后是一个 `)`。

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。

在函数上的一些例子:

```
// 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() {
  // ...
}
```

`cfg` 属性允许在任何允许属性的地方上使用。

`cfg_attr`属性

句法

CfgAttrAttribute :

`cfg_attr` `(` *ConfigurationPredicate* `,` *CfgAttrs*? `)`

CfgAttrs :

Attr (`,` *Attr*)* `,`?

`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。例如，下面的模块可以在 `linux.rs` 或 `windows.rs` 中都能找到。

```
#[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() {}
```

> **注意**：`cfg_attr` 能展开为另一个 `cfg_attr`。比如：`#[cfg_attr(target_os = "linux", cfg_attr(feature = "multithreaded", some_other_attribute))]`
> 这个是合法的。这个示例等效于：`#[cfg_attr(all(target_os = "linux", feature ="multithreaded"), some_other_attribute)]`.

`cfg_attr` 属性允许在任何允许属性的地方上使用。

The `cfg` macro

`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。

例如：

```
let machine_kind = if cfg!(unix) {
  "unix"
} else if cfg!(windows) {
  "windows"
} else {
  "unknown"
};

println!("I'm running on a {} machine!", machine_kind);
```

[1] 首先给出 *目标三元组* 的参考资料地址：
https://www.bookstack.cn/read/rCore_tutorial_doc/d997e9cbdfeef7d4.md。
接下来为防止该地址失效，我用自己的理解简单重复一下我对这个名词的理解:
目标三元组可以理解为我们常说的平台信息，包含这些信息：第一项元素：CPU 架构；第二项元素：供应商；第三项元素：操作系统；第四项元素：ABI。
Rust 下查看当前平台的三元组属性可以用 `rustc --version --verbose` 命令行。比如我在我工作机下下执行这行命令行的输出的 `host` 信息为：`host: x86_64-unknown-linux-gnu`，那我都工作机的目标三元组的信息就是：CPU 架构为 x86_64 ，供应商为 unknown ，操作系统为 linux ，ABI 为 gnu 。
我另一台 windows 机器为：`host: x86_64-pc-windows-msvc`，那这台的目标三元组的信息为：CPU 架构为 x86_64 ，供应商为 pc ，操作系统为 windows ，ABI 为 msvc 。
Rust 官方对一些平台提供了默认的目标三元组，我们可以通过 `rustc --print target-list` 命令来查看完整列表。

程序项

items.md
commit: b0e0ad6490d6517c19546b1023948986578fc378
本章译文最后维护日期：2020-11-8

句法:

Item:
  *OuterAttribute**
    VisItem
  | *MacroItem*

VisItem:
  Visibility?
  (
      Module
    | *ExternCrate*
    | *UseDeclaration*
    | *Function*
    | *TypeAlias*
    | *Struct*
    | *Enumeration*
    | *Union*
    | *ConstantItem*
    | *StaticItem*
    | *Trait*
    | *Implementation*
    | *ExternBlock*
)

MacroItem:
    MacroInvocationSemi
  | *MacroRulesDefinition*

*程序项*是 crate 的组成单元。程序项由一套嵌套的模块被组织在一个 crate 内。每个 crate 都

有一个“最外层”的匿名模块；crate 中所有的程序项都在其 crate 的模块树中自己的路径。

程序项在编译时就完全确定下来了，通常在执行期间保持结构稳定，并可以驻留在只读内存中。

有以下几类程序项:

- 模块
- 外部crate(`extern crate`)声明
- `use`声明
- 函数定义
- 类型定义
- 结构体定义
- 枚举定义
- 联合体定义
- 常量项
- 静态项
- trait定义
- 实现
- 外部块(`extern` blocks)

有些程序项会形成子（数据）项声明的隐式作用域。换句话说，在一个函数或模块中，程序项的声明可以（在许多情况下）与语句、控制块、以及类似的能构成程序项主体的部件混合在一起。这些在作用域内的程序项的意义与在作用域外声明的程序项的意义相同（它仍然是静态项），只是该程序项在模块的命名空间中的*路径名*由封闭它的程序项的名称限定，或该程序项也可能是封闭它的程序项的私有程序项（比如函数的情况）。语法规范指定了子项声明可能出现的合法位置。

模块

modules.md
commit: eabdf09207bf3563ae96db9d576de0758c413d5d
本章译文最后维护日期：2021-1-24

句法:

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

模块是零个或多个程序项的容器。

*模块项*是一个用花括号括起来的，有名称的，并以关键字 `mod` 作为前缀的模块。模块项将一个新的具名模块引入到组成 crate 的模块树中。模块可以任意嵌套。

模块的一个例子：

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

模块和类型共享相同的命名空间。禁止在同一个作用域中声明与此作用域下模块同名的具名类型(named type)：也就是说，类型定义、trait、结构体、枚举、联合体、类型参数或 crate 不能在其作用域中屏蔽此作用域中也生效的模块名称，反之亦然。使用 `use` 引入到当前作用域的

程序项也受这个限制。

在句法上，关键字 `unsafe` 允许出现在关键字 `mod` 之前，但是在语义层面却会被弃用。这种设计允许宏在将关键字 `unsafe` 从 token流中移除之前利用此句法来使用此关键字。

Module Source Filenames

模块的源文件名

没有代码体的模块是从外部文件加载的。当模块没有 `path` 属性限制时，文件的路径和逻辑上的模块路径互为镜像。祖先模块的路径组件(path component)是此模块文件的目录，而模块的内容存在一个以该模块名为文件名，以 `.rs` 为扩展文件名的文件中。例如，下面的示例可以反映这种模块结构和文件系统结构相互映射的关系：

模块路径	文件系统路径	文件内容
`crate`	`lib.rs`	`mod util;`
`crate::util`	`util.rs`	`mod config;`
`crate::util::config`	`util/config.rs`	

当一个目录下有一个名为 `mod.rs` 的源文件时，模块的文件名也可以和这个目录互相映射。上面的例子也可以用一个承载同一源码内容的名为 `util/mod.rs` 的实体文件来表达模块路径 `crate::util`。注意不允许 `util.rs` 和 `util/mod.rs` 同时存在。

> **注意**：在 `rustc` 1.30 版本之前，使用文件 `mod.rs` 是加载嵌套子模块的方法。现在鼓励使用新的命名约定，因为它更一致，并且可以避免在项目中搞出许多名为 `mod.rs` 的文件。

The `path` attribute

`path`属性

用于加载外部文件模块的目录和文件可以受 `path` 属性的影响。（或者说可以联合使用 `path` 属性来重新指定那种没有代码体的模块声明的加载对象的文件路径。）

对于不在内联模块(inline module)块内的模块上的 `path` 属性，此属性引入的文件的路径为相对于当前源文件所在的目录。例如，下面的代码片段将使用基于其所在位置的路径:

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

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`

对于处在内联模块块内的 `path` 属性，此属性引入的文件的路径取决于 `path` 属性所在的源文件的类型。（先对源文件进行分类，）“mod-rs”源文件是根模块（比如是 `lib.rs` 或 `main.rs`）和文件名为 `mod.rs` 的模块，“非mod-rs”源文件是所有其他模块文件。（那）在 mod-rs 文件中，内联模块块内的 `path` 属性（引入的文件的）路径是相对于 mod-rs 文件的目录（该目录包括作为目录的内联模块组件名）。对于非mod-rs 文件，除了路径以此模块名为目录前段外，其他是一样的。例如，下面的代码片段将使用基于其所在位置的路径:

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

Source File	`inner` 's File Location	`inner` 's Module Path
`src/a/b.rs`	`src/a/b/inline/other.rs`	`crate::a::b::inline::inner`
`src/a/mod.rs`	`src/a/inline/other.rs`	`crate::a::inline::inner`

在内联模块和其内嵌模块上混合应用上述 `path` 属性规则的一个例子（mod-rs 和非mod-rs 文件都适用）：

```
#[path = "thread_files"]
mod thread { // 译者注：有模块要内联进来的内联模块
    // 从相对于当前源文件的目录下的 `thread_files/tls.rs` 文件里载 `local_data` 模块。
    #[path = "tls.rs"]
    mod local_data; // 译者注：内嵌模块
}
```

Attributes on Modules

模块上的属性

模块和所有程序项一样能接受外部属性。它们也能接受内部属性：可以在带有代码体的模块的 `{` 之后，也可以在模块源文件的开头（但须在可选的 BOM 和 shebang 之后）。

在模块中有意义的内置属性是 `cfg`、`deprecated`、`doc`、lint检查类属性、`path` 和 `no_implicit_prelude`。模块也能接受宏属性。

外部crate声明

extern-crates.md
commit: 0ce54e64e3c98d99862485c57087d0ab36f40ef0
本章译文最后维护日期：2021-1-24

句法:

ExternCrate :
`extern` `crate` *CrateRef AsClause*? `;`

CrateRef :
IDENTIFIER | `self`

AsClause :
`as` (IDENTIFIER | `_`)

*外部crate(`extern crate`)声明*指定了对外部 crate 的依赖关系。（这种声明让）外部的 crate 作为外部crate(`extern crate`)声明中提供的标识符被绑定到当前声明的作用域中。此外，如果 `extern crate` 出现在 crate的根模块中，那么此 crate名称也会被添加到外部预导入包中，以便使其自动出现在所有模块的作用域中。`as` 子句可用于将导入的 crate 绑定到不同的名称上。

外部crate 在编译时被解析为一个特定的 `soname`[1]， 并且一个到此 `soname` 的运行时链接会传递给链接器，以便在运行时加载此 `soname`。`soname` 在编译时解析，方法是扫描编译器的库文件路径，匹配外部crate 的 `crateid`。因为 `crateid` 是在编译时通过可选的 `crateid` 属性声明的，所以如果外部 crate 没有提供 `crateid`， 则默认拿该外部crate 的 `name` 属性值来和外部crate(`extern crate`)声明中的[标识符]绑定。

导入 `self` crate 会创建到当前 crate 的绑定。在这种情况下，必须使用 `as` 子句指定要绑定到的名称。

三种外部crate(`extern crate`)声明的示例:

```
extern crate pcre;

extern crate std; // 等同于: extern crate std as std;

extern crate std as ruststd; // 使用其他名字去链接 'std'
```

当给 Rust crate 命名时，不允许使用连字符(`-`)。然而 Cargo 包却可以使用它们。在这种情况下，当 `Cargo.toml` 文件中没有指定 crate 名称时， Cargo 将透明地将 `-` 替换为 `_` 以供 Rust 源文件内的外部crate(`extern crate`)声明引用 (详见 RFC 940)。

这有一个示例：

```
// 导入 Cargo 包 hello-world
extern crate hello_world; // 连字符被替换为下划线
```

Extern Prelude

外部预导入包

本节内容已经移入预导入包 — 外部预导入包中了。

Underscore Imports

下划线导入

外部的 crate依赖可以通过使用带有下划线形如 `extern crate foo as _` 的形式来声明，而无需将其名称绑定到当前作用域内。这种声明方式对于只需要 crate 被链接进来，但 crate 从不会被当前代码引用的情况可能很有用，并且还可以避免未使用的 lint 提醒。

下划线导入不会影响 `macro_use` 属性的正常使用，这情况下使用 `macro_use` 属性，宏名称仍会正常导入到 `macro_use` 预导入包中。

The `no_link` attribute

`no_link`属性

可以在外部项(`extern crate` item)上指定使用 *`no_link`属性*，以防止此 crate 被链接到编译输出中。这通常用于加载一个 crate 而只访问它的宏。

[1] 译者注：这里的 `soname` 是 linux系统里的动态库文件的 soname。

Use声明

use-declarations.md
commit: eabdf09207bf3563ae96db9d576de0758c413d5d
本章译文最后维护日期：2021-1-24

句法:

UseDeclaration :
`use` *UseTree* `;`

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

*use声明*用来创建一个或多个与程序项路径同义的本地名称绑定。通常使用 `use` 声明来缩短引用模块所需的路径。这些声明可以出现在模块和块中，但通常在作用域顶部。

use声明支持多种便捷方法:

- 使用带有花括号的 glob-like(`::`) 句法 `use a::b::{c, d, e::f, g::h::i};` 来同时绑定一个系列有共同前缀的路径。
- 使用关键字 `self`，例如 `use a::b::{self, c, d::e};`，来同时绑定一系列有共同前缀和共同父模块的路径。
- 使用句法 `use p::q::r as x;` 将编译目标名称重新绑定为新的本地名称。这种也可以和上面两种方法一起使用：`use a::b::{self as ab, c as abc}`。
- 使用星号通配符句法 `use a::b::*;` 来绑定与给定前缀匹配的所有路径。
- 将前面的方法嵌套重复使用，例如 `use a::b::{self as ab, c, d::{*, e::f}};`。

`use` 声明的一个示例：

```
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() {
    // 等价于 'foo(vec![std::option::Option::Some(1.0f64),
std::option::Option::None]);'
    foo(vec![Some(1.0f64), None]);

    // `hash_map` 和 `HashMap` 在当前作用域内都有效.
    let map1 = HashMap::new();
    let map2 = hash_map::HashMap::new();
    bar(map1, map2);
}
```

use可见性

与其他程序项一样，默认情况下，`use`声明对包含它的模块来说是私有的。同样的，如果使用关键字`pub`进行限定，`use`声明也可以是公有的。`use`声明可用于*重导出(re-expor)*名称。因此，公有的`use`声明可以将某些公有名称重定向到不同的目标定义中：甚至是位于不同模块内具有私有可见性的规范路径定义中。如果这样的重定向序列形成一个循环或不能明确地解析，则会导致编译期错误。

重导出的一个示例：

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

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

在本例中，模块`quux`重导出了在模块`foo`中定义的两个公共名称。

use 路径

> **注意**：本章节内容还不完整。

一些正常和不正常的使用 `use` 程序项的例子：

```
use std::path::{self, Path, PathBuf};  // good: std 是一个 crate 名称
use crate::foo::baz::foobaz;    // good: foo 在当前 crate 的第一层

mod foo {

    pub mod example {
        pub mod iter {}
    }

    use crate::foo::example::iter; // good: foo 在当前 crate 的第一层
//  use example::iter;      // 在 2015 版里不行，2015 版里相对路径必须以 `self` 开头；2018 版这样写没问题
    use self::baz::foobaz;  // good: `self` 指的是 'foo' 模块
    use crate::foo::bar::foobar;   // good: foo 在当前 crate 的第一层

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

    pub mod baz {
        use super::bar::foobar; // good: `super` 指的是 'foo' 模块
        pub fn foobaz() { }
    }
}

fn main() {}
```

：

> **版本差异**: 在 2015 版中，`use` 路径也允许访问 crate 根模块中的程序项。继续使用上面的例子，那以下 `use` 路径的用法在 2015 版中有效，在 2018 版中就无效了：
>
> ```
> use foo::example::iter;
> use ::foo::baz::foobaz;
> ```
>
> 2015 版不允许用 use声明来引用外部预导入包里的 crate。因此，在2015 版中仍然需要

使用 `extern crate` 声明，以便在 use声明中去引用外部 crate。从 2018 版开始，use声明可以像 `extern crate` 一样指定外部 crate 依赖关系。

在 2018 版中，如果本地程序项与外部的 crate 名称相同，那么使用该 crate 名称需要一个前导的 `::` 来明确地选择该 crate 名称。这种做法是为了与未来可能发生的更改保持兼容。

```
// use std::fs; // 错误，这样有歧义.
use ::std::fs;  // 从`std` crate 里导入，不是下面这个 mod.
use self::std::fs as self_fs;  // 从下面这个 mod 导入.

mod std {
    pub mod fs {}
}
```

下划线导入

通过使用形如为 `use path as _` 的带下划线的 use声明，可以在不绑定名称的情况下导入程序项。这对于导入一个 trait 特别有用，这样就可以在不导入 trait 的 symbol 的情况下使用这个 trait 的方法，例如，如果 trait 的 symbol 可能与另一个 symbol 冲突。再一个例子是链接外部的 crate 而不导入其名称。

使用星号全局导入(Asterisk glob imports,即 `::*`)句法将以 `_` 的形式导入能匹配到的所有程序项，但这些程序项在当前作用域中将处于不可命名的状态。

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

    impl<T> Zoo for T {}
}

use self::foo::Zoo as _;
struct Zoo;  // 下划线形式的导入就是为了避免和这个程序项在名字上起冲突

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

在宏扩展之后会创建一些惟一的、不可命名的 symbols，这样宏就可以安全地扩展出(emit)对 `_` 导入的多个引用。例如，下面代码不应该产生错误:

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

m!(use std as _;);
// 这会扩展出:
// use std as _;
// use std as _;
```

函数

functions.md
commit: 245b8336818913beafa7a35a9ad59c85f28338fb
本章译文最后维护日期：2021-5-6

句法

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

FunctionQualifiers :
  `const`? `async`[1]? `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*

[1] 限定符 `async` 不能在 2015版中使用。

[2] 在2015版中，只有类型的函数参数只允许出现在trait项的关联函数中。

---

*函数*由一个块以及一个名称和一组参数组成。除了名称，其他的都是可选的。函数使用关键字 `fn` 声明。函数可以声明一组*输入变量*作为参数，调用者通过它向函数传递参数，函数完成后，它再将带有*输出类型*的结果值返回给调用者。

当一个*函数*被引用时，该函数会产生一个相应的零尺寸*函数项类型(function item type)*的一等(first-class)*值*，调用该值就相当于直接调用该函数。

举个简单的定义函数的例子:

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

函数参数

和 `let` 绑定一样，函数参数是不可反驳型模式，所以任何在 let绑定中有效的模式都可以有效应用在函数参数上:

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

如果第一个参数是一个*SelfParam*类型的参数，这表明该函数是一个方法。带 self参数的函数只能在 trait 或实现中作为关联函数出现。

带有 `...` token 的参数表示此函数是一个可变参数函数， `...` 只能作为外部块里的函数的最后一个参数使用。可变参数可以有一个可选标识符，例如 `args: ...`。

函数体

函数的块在概念上被包装进在一个块中，该块绑定该函数的参数模式，然后返回(`return`)该函数的块的值。这意味着如果轮到块的*尾部表达式(tail expression)*被求值计算了，该块将结束，求得的值将被返回给调用者。通常，程序执行时如果流碰到函数体中的显式返回表达式(return expression)，就会截断那个隐式的最终表达式的执行。

例如，上面例子里函数的行为就像下面被改写的这样:

```
// argument_0 是调用者真正传过来的第一个参数
let (value, _) = argument_0;
return {
    value
};
```

没有主体块的函数以分号结束。这种形式只能出现在 trait 或外部块中。

泛型函数

*泛型函数*允许在其签名中出现一个或多个*参数化类型(parameterized types)*。每个类型参数必须在函数名后面的尖括号封闭逗号分隔的列表中显式声明。

```
// foo 是建立在 A 和 B 基础上的泛型函数

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

在函数签名上和函数体内部，类型参数的名称可以被用作类型名。可以为类型参数指定 trait约束，以允许对该类型的值调用这些 trait 的方法。这种约束是使用 `where` 句法指定的:

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

当使用泛型函数时，它的（类型参数的）类型会基于调用的上下文被实例化。例如，这里调用 `foo` 函数:

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

fn foo<T>(x: &[T]) where T: Debug {
    // 省略细节
}

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

将用 `i32` 实例化类型参数 `T`。

类型参数也可以在函数名之后的末段路径组件(trailing path component，即 `::<type>`)中显式地提供。如果没有足够的上下文来确定类型参数，那么这可能是必要的。例如：`mem::size_of::<u32>() == 4`。

外部函数限定符

外部函数限定符(`extern`)可以提供那些能通过特定 ABI 才能调用的函数的*定义*：

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

这些通常与提供了函数*声明*的外部块程序项一起使用，这样就可以调用此函数而不必同时提供此函数的*定义*：

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

当函数的 `FunctionQualifiers` 句法规则中的 `"extern" Abi?*` 选项被省略时，会默认使用 `"Rust"` 类型的 ABI。例如：

```
fn foo() {}
```

等价于：

```
extern "Rust" fn foo() {}
```

使用 `"Rust"` 之外的 ABI 可以让 Rust 中声明的函数被其他编程语言调用。比如下面声明了一

个可以从 C 中调用的函数：

```
// 使用 "C" ABI 声明一个函数
extern "C" fn new_i32() -> i32 { 0 }

// 使用 "stdcall" ABI声明一个函数
extern "stdcall" fn new_i32_stdcall() -> i32 { 0 }
```

与外部块一样，当使用关键字 `extern` 而省略 `"ABI` 时，ABI 默认使用的是 `"C"` 。也就是说这个：

```
extern fn new_i32() -> i32 { 0 }
let fptr: extern fn() -> i32 = new_i32;
```

等价于：

```
extern "C" fn new_i32() -> i32 { 0 }
let fptr: extern "C" fn() -> i32 = new_i32;
```

非 `"Rust"` 的 ABI 函数不支持与 Rust 函数完全相同的展开(unwind)方式。因此展开进程过这类 ABI 函数的尾部时就会导致该进程被终止。（译者注：展开是逆向的。）

> **注意**: `rustc` 背后的 LLVM 是通过执行一个非法指令来实现中止进程的功能的。

常量函数

使用关键字 `const` 限定的函数是常量(const)函数，元组结构体构造函数和元组变体构造函数也是如此。可以在常量上下文中调用*常量函数*。

常量函数不允许是 async类型的，并且不能使用 `extern` 函数限定符。

异步函数

函数可以被限定为异步的，这还可以与 `unsafe` 限定符结合在一起使用：

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

异步函数在调用时不起作用：相反，它们将参数捕获进一个 future。当该函数被轮询(polled)时，该 future 将执行该函数的函数体。

一个异步函数大致相当于返回一个以 `async move` 块为代码体的 `impl Future` 的函数：

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

大致等价于：

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

实际的脱糖(desugaring)过程相当复杂：

- 脱糖过程中的返回类型被假定捕获了 *`async fn` 声明*中的所有的生存期参数（包括省略的）。这可以在上面的脱糖示例中看到：（我们）显式地给它补上了一个生存期参数 `'a`，因此捕捉到 `'a`。
- 代码体中的 [`async move` 块][async blocks]捕获所有函数参数，包括未使用或绑定到 `_` 模式的参数。参数销毁方面，除了销毁动作需要在返回的 future 完全执行(fully awaited)完成后才会发生外，可以确保异步函数参数的销毁顺序与函数非异步时的顺序相同。

有关异步效果的详细信息，请参见 `async` 块。

> **版本差异**: 异步函数只能从 Rust 2018 版才开始可用。

`async` 和 `unsafe` 的联合使用

声明一个既异步又非安全(`unsafe`)的函数是合法的。调用这样的函数是非安全的，并且（像任何异步函数一样）会返回一个 future。这个 future 只是一个普通的 future，因此“await”它不

需要一个 `unsafe` 的上下文：

```
// 等待这个返回的 future 相当于解引用 `x`。
//
// 安全条件：在返回的 future 执行完成前，`x` 必须能被安全解引用
async unsafe fn unsafe_example(x: *const i32) -> i32 {
  *x
}

async fn safe_example() {
    // 起初调用上面这个函数时需要一个非安全(`unsafe`)块：
    let p = 22;
    let future = unsafe { unsafe_example(&p) };

    // 但是这里非安全(`unsafe`)块就没必要了，这里能正常读到 `p`：
    let q = future.await;
}
```

请注意，此行为是对返回 `impl Future` 的函数进行脱糖处理的结果——在本例中，我们脱糖处理生成的函数是一个非安全(`unsafe`)函数，这个非安全(`unsafe`)函数返回值与原始定义的函数的返回值仍保持一致。

非安全限定在异步函数上的使用方式与它在其他函数上的使用方式完全相同：只是它表示该函数会要求它的调用者提供一些额外的义务/条件来确保该函数的健全性(soundness)。与任何其他非安全函数一样，这些条件可能会超出初始调用本身——例如，在上面的代码片段中， 函数 `unsafe_example` 将指针 `x` 作为参数，然后（在执行 await 时）解引用了对该指针的引用。这意味着在 future 完成执行之前， `x` 必须是有效的，调用者有责任确保这一点。

函数上的属性

在函数上允许使用外部属性，也允许在函数体中的 `{` 后面直接放置内部属性。

下面这个例子显示了一个函数的内部属性。该函数的文档中只会出现单词“Example”。

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

注意：除了 lint检查类属性，函数上一般惯用的还是外部属性。

在函数上有意义的属性是 `cfg`、`cfg_attr`、`deprecated`、`doc`、`export_name`、`link_section`、`no_mangle`、lint检查类属性、`must_use`、过程宏属性、测试类属性和优化提示类属性。函数也可以接受属性宏。

函数参数上的属性

函数参数允许使用外部属性，允许的内置属性仅限于 `cfg`、`cfg_attr`、`allow`、`warn`、`deny` 和 `forbid`。

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

应用于程序项的过程宏属性所使用的惰性辅助属性也是允许的，但是要注意不要在最终（输出）的 `TokenStream` 中包含这些惰性属性。

例如，下面的代码定义了一个未在任何地方正式定义的惰性属性 `some_inert_attribute`，而 `some_proc_macro_attribute` 过程宏负责检测它的存在，并从输出 token流 `TokenStream` 中删除它。

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

类型别名

type-aliases.md
commit: 4e7812ddd9e75ce5623bddb24fa04efcebe2e98f
本章译文最后维护日期：2021-3-26

句法

TypeAlias :

`type` IDENTIFIER *GenericParams*? *WhereClause*? (`=` *Type*)? `;`

761ad774fcb300f2b506fed7b4dbe753cda88d80 *类型别名*为现有的类型定义一个新名称。类型别名用关键字 `type` 声明。每个值都是一个唯一的特定的类型，但是可以实现几个不同的 trait，或者兼容几个不同的类型约束。

例如，下面将类型 `Point` 定义为类型 `(u8, u8)` 的同义词/别名：

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

761ad774fcb300f2b506fed7b4dbe753cda88d80 元组结构体或单元结构体的类型别名不能用于充当该类型的构造函数: 761ad774fcb300f2b506fed7b4dbe753cda88d80

没有使用 *Type* 来定义的类型别名只能作为 trait 中的关联类型出现。

结构体

structs.md
commit: 9d0aa172932ed15ec1b13556e6809b74bc58a02b
本章译文最后维护日期：2021-1-17

句法

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

*结构体*是一个使用关键字 `struct` 定义的标称型(nominal)结构体类型。

结构体(`struct`)程序项的一个示例和它的使用方法：

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

*元组结构体*是一个标称型(nominal)元组类型，也是用关键字 `struct` 定义的。例如：

```
struct Point(i32, i32);
let p = Point(10, 11);
let px: i32 = match p { Point(x, _) => x };
```

*单元结构体(unit-like struct)*是没有任何字段的结构体，它的定义完全不包含字段(fields)列表。这样的结构体隐式定义了其类型的同名常量（值）。例如:

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

等价于

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

结构体的精确内存布局还没有规范下来。目前可以使用 `repr` 属性来指定特定的布局。

枚举

enumerations.md
commit: d8cbe4eedb77bae3db9eff87b1238e7e23f6ae92
本章译文最后维护日期：2021-2-21

句法

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

EnumItems :
EnumItem (`,` *EnumItem*)* `,`?

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

EnumItemTuple :
`(` *TupleFields*? `)`

EnumItemStruct :
`{` *StructFields*? `}`

EnumItemDiscriminant :
`=` *Expression*

枚举，英文为 *enumeration*，常见其简写形式 *enum*，它同时定义了一个标称型(nominal)枚举类型和一组*构造器*，这可用于创建或使用模式来匹配相应枚举类型的值。

枚举使用关键字 `enum` 来声明。

`enum` 程序项的一个示例和它的使用方法：

```
enum Animal {
    Dog,
    Cat,
}

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

枚举构造器可以带有具名字段或未具名字段：

```
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 };
```

在这个例子中，`Cat` 是一个*类结构体枚举变体(struct-like enum variant)*，而 `Dog` 则被简单地称为枚举变体。每个枚举实例都有一个*判别值/判别式(discriminant)*，它是一个与此枚举实例关联的整数，用来确定它持有哪个变体。可以通过 `mem::discriminant` 函数来获得对这个判别值的不透明引用。

为无字段枚举自定义判别值

如果枚举的*任何*变体都没有附加字段，则可以直接设置和访问判别值。

可以使用操作符 `as` 通过数值转换将这些枚举类型转换为整型。枚举可以可选地指定每个判别值的具体值，方法是在变体名后面追加 `=` 和常量表达式。如果声明中的第一个变体未指定，则将其判别值设置为零。对于其他未指定的判别值，它比照前一个变体的判别值按 1 递增。

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

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

尽管编译器被允许在实际的内存布局中使用较小的类型，但在默认表形(default representation)下，指定的判别值会被解释为一个 `isize` 值。也可以使用[原语表形(primitive representation)]或 `C` 表形来更改成大小可接受的值。

同一枚举中，两个变体使用相同的判别值是错误的。

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

enum SharedDiscriminantError2 {
    Zero,       // 0
    One,        // 1
    OneToo = 1  // 1 (和前值冲突！)
}
```

当前一个变体的判别值是当前表形允许的的最大值时，再使用默认判别值就也是错误的。

```
#[repr(u8)]
enum OverflowingDiscriminantError {
    Max = 255,
    MaxPlusOne // 应该是256，但枚举溢出了
}

#[repr(u8)]
enum OverflowingDiscriminantError2 {
    MaxMinusOne = 254, // 254
    Max,               // 255
    MaxPlusOne         // 应该是256，但枚举溢出了。
}
```

无变体枚举

没有变体的枚举称为*零变体枚举/无变体枚举*。因为它们没有有效的值，所以不能被实例化。

```
enum ZeroVariants {}
```

零变体枚举与 *never类型*等效，但它不能被强转为其他类型。

```
let x: ZeroVariants = panic!();
let y: u32 = x; // 类型不匹配错误
```

变体的可见性

依照句法规则，枚举变体是允许有自己的[*可见性(visibility)*][Visibility]限定/注解(annotation)的，但当枚举被（句法分析程序）验证(validate)通过后，可见性注解又被弃用。因此，在源码解析层面，允许跨不同的上下文对其中不同类型的程序项使用统一的句法规则进行解析。

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

            $vis Tuple(u8, u16),

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

// 允许空 `vis`.
mac_variant! { E }

// 这种也行，因为这段代码在被验证通过前会被移除。
#[cfg(FALSE)]
enum E {
    pub U,
    pub(crate) T(u8),
    pub(super) T { f: String }
}
```

联合体

unions.md
commit: 7c6e0c00aaa043c89e0d9f07e78999268e8ac054
本章译文最后维护日期：2021-2-10

句法

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

除了用 `union` 代替 `struct` 外，联合体声明使用和结构体声明相同的句法。

```
#[repr(C)]
union MyUnion {
    f1: u32,
    f2: f32,
}
```

联合体的关键特性是联合体的所有字段共享同一段存储。因此，对联合体的一个字段的写操作会覆盖其他字段，而联合体的尺寸由其尺寸最大的字段的尺寸所决定。

联合体的初始化

可以使用与结构体类型相同的句法创建联合体类型的值，但必须只能指定一个字段：

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

上面的表达式创建了一个类型为 `MyUnion` 的值，并使用字段 `f1` 初始化了其存储。可以使用与结构体字段相同的句法访问联合体：

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

读写联合体字段

联合体没有“活跃字段(active field)”的概念。相反，每次访问联合体只是用访问所指定的字段的类型解释此联合体的存储。读取联合体的字段就是以当前读取字段的类型来解读此联合体的存储位。字段（间）可以有非零的偏移量存在（使用 C表型的除外）；在这种情况下，读取将从字段的相对偏移量的 bit 开始。程序员有责任确保此数据在当前字段类型下有效。否则会导致未定义行为(undefined behavior)。例如，在 `bool` 类型的字段下读取到数值 `3` 是未定义行为。实际上，对一个 C表型的联合体进行写操作，然后再从中读取，就好比从用于写入的类型到用于读取的类型的 `transmute` 操作。

因此，所有的联合体字段的读取必须放在非安全(`unsafe`)块里：

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

对实现了 `Copy` trait 或 [`ManuallyDrop`][ManuallyDrop] trait 的联合体字段的写操作不需要事先的析构读操作，也因此这些写操作不必放在非安全(`unsafe`)块中。[1]

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

// 这些都不是必须要放在 `unsafe` 里的
u.f1 = 2;
u.f2 = ManuallyDrop::new(String::from("example"));
```

通常，那些用到联合体的程序代码会先在非安全的联合体字段访问操作上提供一层安全包装，然后再使用。

联合体和 `Drop`

当一个联合体被销毁时，它无法知道需要销毁它的哪些字段。因此，所有联合体的字段都必须实现 `Copy` trait 或被包装进 `ManuallyDrop<_>` 。这确保了联合体在超出作用域时不需要销毁任何内容。

与结构体和枚举一样，联合体也可以通过 `impl Drop` 手动定义被销毁时的具体动作。

联合体上的模式匹配

访问联合体字段的另一种方法是使用模式匹配。联合体字段上的模式匹配与结构体上的模式匹配使用相同的句法，只是这种模式只能一次指定一个字段。因为模式匹配就像使用特定字段来读取联合体，所以它也必须被放在非安全(`unsafe`)块中。

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

模式匹配可以将联合体作为更大的数据结构的一个字段进行匹配。特别是，当使用 Rust 联合体通过 FFI 实现 C标签联合体(C tagged union)时，这允许同时在标签和相应字段上进行匹配：

```
#[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,
        }
    }
}
```

引用联合体字段

由于联合体字段共享存储，因此拥有对联合体一个字段的写访问权就同时拥有了对其所有其他字段的写访问权。因为这一事实，引用的借用检查规则必须调整。因此，如果联合体的一个字段是被出借，那么在相同的生存期内它的所有其他字段也都处于出借状态。

```
// 错误: 不能同时对 `u` (通过 `u.f2`)拥有多余一次的可变借用
fn test() {
    let mut u = MyUnion { f1: 1 };
    unsafe {
        let b1 = &mut u.f1;
//                   ---- 首次可变借用发生在这里 (通过 `u.f1`)
        let b2 = &mut u.f2;
//                   ^^^^ 二次可变借用发生在这里 (通过 `u.f2`)
        *b1 = 5;
    }
//  - 首次借用在这里结束
    assert_eq!(unsafe { u.f1 }, 5);
}
```

如您所见，在许多方面（除了布局、安全性和所有权），联合体的行为与结构体完全相同，这很大程度上是因为联合体继承使用了结构体的句法的结果。对于 Rust 语言未明确提及的许多方面（比如隐私性(privacy)、名称解析、类型推断、泛型、trait实现、固有实现、一致性、模式检查等等）也是如此。

[1] 这句译者简单理解就是对已经初始化的变量再去覆写的时候要先去读一下这个变量代表的地址上的值的状态，如果有值，并且允许覆写，那 Rust 为防止内存泄漏就先执行那变量的析构行为（drop()），清空那个地址上的关联堆数据，再写入。我们这里对联合体的预设条件是此联合体值有 Copy特性，有 Copy特性了，对值的直接覆写不会造成内存泄漏，就不必调用析构行为，也不需要事先的非安全读操作了。对于这个问题nomicon的“未初始化内存”章有讲述，博主CrLF0710的两篇“学一点 Rust 内存模型会发生什么呢？”里也都有精彩讲解。

常量项

constant-items.md
commit: 761ad774fcb300f2b506fed7b4dbe753cda88d80
本章译文最后维护日期：2020-1-17

句法

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

*常量项*是一个可选的具名 *常量值*，它与程序中的具体内存位置没有关联。无论常量在哪里使用，它们本质上都是内联的，这意味着当它们被使用时，都是直接被拷贝到相关的上下文中来使用的。这包括使用非拷贝(non-`Copy`)类型的值和来自外部的 crate 的常量。对相同常量的引用不保证它们引用的是相同的内存地址。

常量必须显式指定数据类型。类型必须具有 `'static` 生存期：程序初始化器(initializer)中的任何引用都必须具有 `'static` 生存期。

常量可以引用其他常量的地址，在这种情况下，如果适用，该地址将具有省略的生存期，否则（在大多数情况下）默认为 `'static` 生存期。（请参阅静态生存期省略。）但是，编译器仍有权多次调整转移该常量，因此引用的地址可能并不固定。

```
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,
};
```

常量表达式只能在trait定义中省略。

Constants with Destructors

常量与析构函数

常量可以包含析构函数。析构函数在值超出作用域时运行。[1]

```
struct TypeWithDestructor(i32);

impl Drop for TypeWithDestructor {
    fn drop(&mut self) {
        println!("Dropped. Held {}.", self.0);
    }
}

const ZERO_WITH_DESTRUCTOR: TypeWithDestructor = TypeWithDestructor(0);

fn create_and_drop_zero_with_destructor() {
    let x = ZERO_WITH_DESTRUCTOR;
    // x 在函数的结尾处通过调用 drop 方法被销毁。
    // 打印出 "Dropped. Held 0.".
}
```

Unnamed constant

未命名常量

不同于关联常量(associated constant)，自由常量(free constant)可以使用下划线来命名。例如:

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

// OK 尽管名称和上面的一样：
const _: () =  { struct _SameNameTwice; };
```

与下划线导入一样，宏可以多次安全地在同一作用域中扩展出相同的未具名常量。例如，以下内容不应该产生错误：

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

m!(const _: () = (););
// 这会展开出：
// const _: () = ();
// const _: () = ();
```

[1] 在程序退出前，析构销毁的只是其中的一份拷贝；这句还有另一层含义是常量在整个程序结束时会调用析构函数。

静态项

static-items.md
commit: 761ad774fcb300f2b506fed7b4dbe753cda88d80
本章译文最后维护日期：2021-1-17

句法

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

*静态项*类似于常量项，除了它在程序中表示一个精确的内存位置。所有对静态项的引用都指向相同的内存位置。静态项拥有 `'static` 生存期，它比 Rust 程序中的所有其他生存期都要长。静态项不会在程序结束时调用析构动作 `drop`。

静态初始化器是在编译时求值的常量表达式。静态初始化器可以引用其他静态项。

包含非内部可变类型的非 `mut` 静态项可以放在只读内存中。

所有访问静态项的操作都是安全的，但对静态项有一些限制：

- 静态项的数据类型必须有 `Sync` trait约束，这样才可以让线程安全地访问。
- 常量项不能引用静态项。

必须为自由静态项提供初始化表达式，但在外部块中静态项必须省略初始化表达式。

可变静态项

如果静态项是用关键字 `mut` 声明的，则它允许被程序修改。Rust 的目标之一是使尽可能的避免出现并发 bug，那允许修改可变静态项显然是竞态(race conditions)或其他 bug 的一个重要来源。因此，读取或写入可变静态项变量时需要引入非安全(`unsafe`)块。应注意确保对可变静态项的修改相对于运行在同一进程中的其他线程来说是安全的。

尽管有这些缺点，可变静态项仍然非常有用。它们可以与 C库一起使用，也可以在外部(`extern`)块中从 C库中来绑定它。

```
static mut LEVELS: u32 = 0;

// 这违反了不共享状态的思想，而且它在内部不能防止竞争，所以这个函数是非安全的(`unsafe`)
unsafe fn bump_levels_unsafe1() -> u32 {
    let ret = LEVELS;
    LEVELS += 1;
    return ret;
}

// 这里我们假设有一个返回旧值的 atomic_add 函数，这个函数是“安全的(safe)”，
// 但是返回值可能不是调用者所期望的，所以它仍然被标记为 `unsafe`
unsafe fn bump_levels_unsafe2() -> u32 {
    return atomic_add(&mut LEVELS, 1);
}
```

除了可变静态项的类型不需要实现 `Sync` trait 之外，可变静态项与普通静态项具有相同的限制。

使用常量项或静态项

应该使用常量项还是应该使用静态项可能会令人困惑。一般来说，常量项应优先于静态项，除非以下情况之一成立：

- 存储大量数据
- 需要静态项的存储地址不变的特性。
- 需要内部可变性。

Trait

traits.md
commit: 20340ce30db8ec17b0a09dc6e07c0fa2f5c3c0ab
本章译文最后维护日期：2021-5-6

句法

Trait :

`unsafe`? `trait` IDENTIFIER *GenericParams*? (`:` *TypeParamBounds*?)? *WhereClause*? `{`

*InnerAttribute**

*AssociatedItem**

`}`

trait 描述类型可以实现的抽象接口。这类接口由三种关联程序项(associated items)组成，它们分别是：

- 函数
- 类型
- 常量

所有 trait 都定义了一个隐式类型参数 `Self` ，它指向“实现此接口的类型”。trait 还可能包含额外的类型参数。这些类型参数，包括 `Self` 在内，都可能会跟正常类型参数一样受到其他 trait 的约束。

trait 需要具体的类型去实现，具体的实现方法是通过该类型的各种独立实现(implementations)来完成的。

trait函数可以通过使用分号代替函数体来省略函数体。这表明此 trait的实现必须去定义实现该函数。如果 trait函数定义了一个函数体，那么这个定义就会作为任何不覆盖它的实现的默认函数实现。类似地，关联常量可以省略等号和表达式，以指示相应的实现必须定义该常量值。关联类型不能定义类型，只能在实现中指定类型。

```
// 有定义和没有定义的相关联trait项的例子
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) {}
}
```

Trait函数不能是 `async` 或 `const` 类型的。

Trait bounds

trait约束

泛型程序项可以使用 trait 作为其类型参数的约束。

Generic Traits

泛型trait

可以为 trait 指定类型参数来使该 trait 成为泛型trait/泛型类型。这些类型参数出现在 trait 名称之后，使用与泛型函数相同的句法。

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

Object Safety

对象安全条款

对象安全的 trait 可以是 trait对象的底层 trait。如果 trait 符合以下限定条件（在 RFC 255 中定义），则认为它是*对象安全的(object safe)*：

- 所有的超类traitsupertraits 也必须也是对象安全的。
- 超类trait 中不能有 `Sized`。也就是说不能有 `Self: Sized` 约束。
- 它必须没有任何关联常量。
- 所有关联函数必须可以从 trait对象调度分派，或者是显式不可调度分派：
  - 可调度分派函数要求：
    - 不能有类型参数（尽管生存期参数可以有）
    - 作为方法时，`Self` 只能出现在方法的接受者(receiver)的类型里，其它地方不能使用 `Self`。
    - 方法的接受者的类型必须是以下类型之一：
      - `&Self` (例如：`&self`)
      - `&mut Self` (例如：`&mut self`)
      - `Box<Self>`
      - `Rc<Self>`
      - `Arc<Self>`
      - `Pin<P>` 当 `P` 是上面类型中的一种
    - 没有 `where Self: Sized` 约束（即接受者的类型 `Self`(例如：`self`) 不能有 `Sized` 约束）。
  - 显式不可调度分派函数要求：
    - 有 `where Self: Sized` 约束（即接受者的类型 `Self`(例如：`self`) 有 `Sized` 约束）。

```
// 对象安全的 trait 方法。
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>>) {}
}
```

```
// 此 trait 是对象安全的，但不能在 trait对象上分发(dispatch)使用这些方法。
trait NonDispatchable {
    // 非方法不能被分发。
    fn foo() where Self: Sized {}
    // 在运行之前 Self 类型未知。
    fn returns(&self) -> Self where Self: Sized;
    // `other` 可能是另一具体类型的接受者。
    fn param(&self, other: Self) where Self: Sized {}
    // 泛型与虚函数指针表(Virtual Function Pointer Table, vtable)不兼容。
    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(); // 错误：不能调用带有 Self 返回类型的方法
obj.param(S);  // 错误：不能调用带有 Self 类型的参数的方法
obj.typed(1);  // 错误：不能调用带有泛型类型参数的方法
```

```
// 非对象安全的 trait
trait NotObjectSafe {
    const CONST: i32 = 1;  // 错误：不能有关联常量

    fn foo() {}  // 错误：关联函数没有 `Sized` 约束
    fn returns(&self) -> Self; // 错误：`Self` 在返回类型中
    fn typed<T>(&self, x: T) {} // 错误：泛型类型参数
    fn nested(self: Rc<Box<Self>>) {} // 错误：嵌套接受者还未被完全支持。（译者注：有限支持见上面的补充规则。）
}

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

```
// Self: Sized trait 非对象安全的。
trait TraitWithSize where Self: Sized {}

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

```
// 如果有 `Self` 这样的泛型参数，那 trait 就是非对象安全的
trait Super<A> {}
trait WithSelf: Super<Self> where Self: Sized {}

struct S;
impl<A> Super<A> for S {}
impl WithSelf for S {}
let obj: Box<dyn WithSelf> = Box::new(S); // 错误：不能使用 `Self` 作为类型参数
```

Supertraits

超类trait

超类trait 是类型为了实现某特定 trait 而需要一并实现的 trait。此外，在任何地方，如果泛型

或 trait对象被某个 trait约束，那这个泛型或 trait对象就可以访问这个*超类trait* 的关联程序项。

超类trait 是通过 trait 的 `Self` 类型上的 trait约束来声明的，并且通过这种声明 trait约束的方式来传递这种超类trait 关系。一个 trait 不能是它自己的超类trait。

有超类trait 的 trait 称其为其超类trait 的**子trait(subtrait)**。

下面是一个声明 `Shape` 是 `Circle` 的超类trait 的例子。

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

下面是同一个示例，除了改成使用 where子句来等效实现。

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

下面例子通过 `Shape` 的 `area` 函数为 `radius` 提供了一个默认实现：

```
trait Circle where Self: Shape {
    fn radius(&self) -> f64 {
        // 因为 A = pi * r^2, 所以通过代数推导得: r = sqrt(A / pi)
        (self.area() /std::f64::consts::PI).sqrt()
    }
}
```

下一个示例调用了一个泛型参数的超类trait 上的方法。

```
fn print_area_and_radius<C: Circle>(c: C) {
    // 这里我们调用 `Circle` 的超类trait 的 area 方法。
    println!("Area: {}", c.area());
    println!("Radius: {}", c.radius());
}
```

类似地，这里是一个在 trait对象上调用超类trait 的方法的例子。

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

Unsafe traits

非安全trait

以关键字 `unsafe` 开头的 trait程序项表示*实现*该 trait 可能是非安全的。使用正确实现的非安全trait 是安全的。trait实现也必须以关键字 `unsafe` 开头。

`Sync` 和 `Send` 是典型的非安全trait。

Parameter patterns

参数模式

（trait 中）没有代码体的函数声明或方法声明（的参数模式）只允许使用标识符/IDENTIFIER模式 或 `_` 通配符模式。当前 `mut` IDENTIFIER 还是允许的，但已被弃用，未来将成为一个硬编码错误(hard error)。

在 2015 版中，trait 的函数或方法的参数模式是可选的：

```
trait T {
    fn f(i32);  // 不需要参数的标识符.
}
```

所有的参数模式被限制为下述之一：

- IDENTIFIER
- `mut` IDENTIFIER
- `_`
- `&` IDENTIFIER
- `&&` IDENTIFIER

（跟普通函数一样，）从 2018 版开始，（trait 中）函数或方法的参数模式不再是可选的。同时，也跟普通函数一样，（trait 中）函数或方法只要有代码体，其参数模式可以是任何不可反驳型模式。但如果没有代码体，上面列出的限制仍然有效。

```
trait T {
    fn f1((a, b): (i32, i32)) {}
    fn f2(_: (i32, i32));  // 没有代码体不能使用元组模式。
}
```

Item visibility

程序项的可见性

依照句法规定，trait程序项在语法上允许使用 *Visibility*句法的注释，但是当 trait 被（句法法分析程序）验证(validate)后，该可见性注释又被弃用。因此，在源码解析层面，可以在使用程序项的不同上下文中使用统一的语法对这些程序项进行解析。例如，空的 `vis` 宏匹配段选择器可以用于 trait程序项，而在其他允许使用非空可见性的情况下，也可使用这同一套宏规则。

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

trait T1 {
    // 只允许空 `vis`。
    create_method! { method_of_t1 }
}

struct S;

impl S {
    // 这里允许使用非空可见性。
    create_method! { pub method_of_s }
}

impl T1 for S {}

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

[1] 两点提醒：所有 trait 都定义了一个隐式类型参数 `Self`，它指向“实现此接口的类型”；trait 的 Self 默认满足：`Self: ?Sized`

实现

implementations.md
commit: 761ad774fcb300f2b506fed7b4dbe753cda88d80
本章译文最后维护日期：2021-1-17

句法

Implementation :
  InherentImpl | *TraitImpl*

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

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

*实现*是将程序项与*实现类型(implementing type)*关联起来的程序项。实现使用关键字 `impl` 定义，它包含了属于当前实现的类型的实例的函数，或者包含了当前实现的类型本身的静态函数。

有两种类型的实现:

- 固有实现(inherent implementations)
- trait实现(trait implementations)

固有实现

固有实现被定义为一段由关键字 `impl`，泛型类型声明，指向标称类型(nominal type)的路径，一个 where子句和一对花括号括起来的一组*类型关联项(associable items)*组成的序列。

（这里）*标称类型*也被称作*实现类型(implementing type)*；*类型关联项(associable items)可理解为实现类型的各种关联程序项(associated items)*。

固有实现将其包含的程序项与其的实现类型关联起来。固有实现可以包含关联函数（包括方法）和关联常量。固有实现不能包含关联类型别名。

关联程序项的路径是其实现类型的所有（形式的）路径中的任一种，然后再拼接上这个关联程序项的标识符来作为整个路径的末段路径组件(final path component)。

类型可以有多个固有实现。但作为原始类型定义的实现类型必须与这些固有实现处在同一个 crate 里。

```
pub mod color {
    // 译者添加的注释：这个结构体是 Color 的原始类型，是一个标称类型，也是后面两个实现的实现类型
    pub struct Color(pub u8, pub u8, pub u8);
    // 译者添加的注释：这个实现是 Color 的固有实现
    impl Color {
        // 译者添加的注释：类型关联项(associable items)
        pub const WHITE: Color = Color(255, 255, 255);
    }
}

mod values {
    use super::color::Color;
    // 译者添加的注释：这个实现也是 Color 的固有实现
    impl Color {
        // 译者添加的注释：类型关联项(associable items)
        pub fn red() -> Color {
            Color(255, 0, 0)
        }
    }
}

pub use self::color::Color;
fn main() {
    // 实现类型 和 固有实现 在同一个模块下。
    color::Color::WHITE;

    // 固有实现和类型声明不在同一个模块下，此时对通过固有实现关联进的程序项的存取仍通过指向实现类型的路径
    color::Color::red();

    // 实现类型重导出后，使用这类快捷路径效果也一样。
    Color::red();

    // 这个不行，因为 `values` 非公有。
    // values::Color::red();
}
```

Trait Implementations

trait实现

*trait实现*的定义与固有实现类似，只是可选的泛型类型声明后须跟一个 trait，再后跟关键字 `for`，之后再跟一个指向标称类型的路径。

这里讨论的 trait 也被称为*被实现trait(implemented trait)*。实现类型去实现该被实现trait。

trait实现必须去定义被实现trait 声明里的所有非默认关联程序项，可以重新定义被实现trait 定义的默认关联程序项，但不能定义任何其他程序项。

关联程序项的完整路径为 `<` 后跟实现类型的路径，再后跟 `as`，然后是指向 trait 的路径，再后跟 `>`，这整体作为一个路径组件，然后再后接关联程序项自己的路径组件。

非安全(unsafe) trait 需要 trait实现以关键字 `unsafe` 开头。

```
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实现的一致性

一个 trait实现如果未通过孤儿规则(orphan rules)检查或有 trait实现重叠(implementations overlap)发生，则认为 trait实现不一致(incoherent)。

当两个实现各自的 *trait接口*集之间存在非空交集时即为这两个 trait实现 重叠了，（这种常情况下）这两个 trait实现可以用相同的类型来实例化。

孤儿规则

给定 `impl<P1..=Pn> Trait<T1..=Tn> for T0`，只有以下至少一种情况为真时，此 `impl` 才能成立：

- `Trait` 是一个本地 trait
- 以下所有
  - `T0..=Tn` 中的类型至少有一种是本地类型。假设 `Ti` 是第一个这样的类型。
  - 没有无覆盖类型参数 `P1..=Pn` 会出现在 `T0..Ti` 里（注意 `Ti` 被排除在外）。

（译者注：为理解上面两条规则，举几个例子、`impl<T> ForeignTrait<LocalType> for ForeignType<T>` 这样的实现也是被允许的，而 `impl<Vec<T>> ForeignTrait<LocalType> for ForeignType<T>` 和 `impl<T> ForeignTrait<Vec<T>> for ForeignType<T>` 不被允许。）

这里只有*无覆盖*类型参数的外观被限制（译者注：为方便理解“无覆盖类型参数”，译者提醒读者把它想象成上面译者举例中的 `T`）。注意，理解“无覆盖类型参数”时需要注意：为了保持一致性，基本类型虽然外观形式特殊，但仍不认为是有覆盖的，比如 `Box<T>` 中的 `T` 就不认为是有覆盖的，`Box<LocalType>` 这样的就被认为是本地的。

泛型实现

实现可以带有泛型参数，这些参数可用在此实现中的其他地方。实现里的泛型参数直接写在关键字 `impl` 之后。

```
impl<T> Seq<T> for Vec<T> {
    /* ... */
}
impl Seq<bool> for u32 {
    /* 将整数处理为 bits 序列 */
}
```

如果泛型参数在以下情况下出现过，则其就能约束某个实现：

- 需要被实现的 trait中存在泛型参数
- 实现类型中存在泛型参数
- 作为类型的约束中的[关联类型]，该类型包含另一个约束实现的形参。

类型和常量参数必须能在相应实现里体现出约束逻辑。如果关联类型中有生存期参数在使用，则该生存期的约束意义也必须在实现中体现。

约束必须被传递实现的例子：

```
// T 通过作为 GenericTrait 的参数来体现约束
impl<T> GenericTrait<T> for i32 { /* ... */ }

// T 是作为 GenericStruct 的参数来体现约束
impl<T> Trait for GenericStruct<T> { /* ... */ }

// 同样，N 作为 ConstGenericStruct 的参数来体现约束
impl<const N: usize> Trait for ConstGenericStruct<N> { /* ... */ }

// T 通过类型 `U` 内的关联类型来体现约束，而 `U` 本身就是一个trait的泛型参数
impl<T, U> GenericTrait<U> for u32 where U: HasAssocType<Ty = T> { /* ... */
}

// 这个和前面一样，除了类型变为 `(U, isize)`。`U` 出现在包含 `T` 的类型中，而不是类型本
身。
impl<T, U> GenericStruct<U> where (U, isize): HasAssocType<Ty = T> { /* ...
*/ }
```

约束未能被传递实现的例子：

```
// 下面的都是错误的，因为它们的类型或常量参数没能被传递和实现约束

// T没有实现约束，因为实现内部根本就没有出现 T
impl<T> Struct { /* ... */ }

// 同样 N 也没有可能被实现
impl<const N: usize> Struct { /* ... */ }

// 在实现中使用了 T，但却并不约束此实现
impl<T> Struct {
    fn uses_t(t: &T) { /* ... */ }
}

// 在 U 的约束中，T 被用作关联类型，但 U 本身在 Struct 里无法被约束
impl<T, U> Struct where U: HasAssocType<Ty = T> { /* ... */ }

// T 是在约束中使用了，但不是作为关联类型使用的，所以它没有实现约束
impl<T, U> GenericTrait<U> for u32 where U: GenericTrait<T> {}
```

允许的非约束生存期参数的示例：

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

不允许的非约束生存期参数的示例：

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

实现上的属性

实现可以在关键字 `impl` 之前引入外部属性，在代码体内引入内部属性。内部属性必须位于任何关联程序项之前。这里有意义的属性有 `cfg`、`deprecated`、`doc` 和 lint检查类属性。

外部块

external-blocks.md
commit: 761ad774fcb300f2b506fed7b4dbe753cda88d80
本章译文最后维护日期：2021-1-17

句法

ExternBlock :

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

ExternalItem :

*OuterAttribute** (
MacroInvocationSemi
| (*Visibility*? (*StaticItem* | *Function*))
)

外部块提供未在当前 crate 中*定义*的程序项的*声明*，外部块是 Rust 外部函数接口的基础。这其实是某种意义上的不受安全检查的导入入口。

外部块里允许存在两种形式的程序项*声明*：函数和静态项。只有在非安全(`unsafe`)上下文中才能调用在外部块中声明的函数或访问在外部块中声明的静态项。

在句法上，关键字 `unsafe` 允许出现在关键字 `extern` 之前，但是在语义层面却会被弃用。这种设计允许宏在将关键字 `unsafe` 从 token流中移除之前利用此句法来使用此关键字。

函数

外部块中的函数与其他 Rust函数的声明方式相同，但这里的函数不能有函数体，取而代之的是直接以分号结尾。外部块中的函数的参数不允许使用模式，只能使用标识符(IDENTIFIER) 或 `_`。函数限定符（`const`、`async`、`unsafe` 和 `extern`）也不允许在这里使用。

外部块中的函数可以被 Rust 代码调用，就跟调用在 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。

在外部块中声明的函数隐式为非安全(`unsafe`)的。当强转为函数指针时，外部块中声明的函数的类型就为 `unsafe extern "abi" for<'l1, ..., 'lm> fn(A1, ..., An) -> R`，其中 `'l1`，…`'lm` 是其生存期参数， `A1`，…， `An` 是该声明的参数的类型， `R` 是该声明的返回类型。

静态项

静态项在外部块内部与在外部块之外的声明方式相同，只是在外部块内部声明的静态项没有对应的初始化表达式。访问外部块中声明的静态项是 `unsafe` 的，不管它是否可变，因为没有任何保证来确保静态项的内存位模式(bit pattern)对声明它的类型是有效的，因为初始化这些静态项的可能是其他的任意外部代码（例如 C）。

就像外部块之外的静态项，外部静态项可以是不可变的，也可以是可变的。在执行任何 Rust 代码之前，不可变外部静态项*必须*被初始化。也就是说，对于外部静态项，仅在 Rust 代码读取它之前对它进行初始化是不够的。

ABI

不指定 ABI 字符串的默认情况下，外部块会假定使用指定平台上的标准 C ABI 约定来调用当前的库。其他的 ABI 约定可以使用字符串 `abi` 来指定，具体如下所示：

```
// 到 Windows API 的接口。（译者注：指定使用 stdcall调用约定去调用 Windows API）
extern "stdcall" { }
```

有三个 ABI 字符串是跨平台的，并且保证所有编译器都支持它们：

- `extern "Rust"` -- 在任何 Rust 语言中编写的普通函数 `fn foo()` 默认使用的 ABI。
- `extern "C"` -- 这等价于 `extern fn foo()`；无论您的 C编译器支持什么默认 ABI。
- `extern "system"` -- 在 Win32 平台之外，中通常等价于 `extern "C"`。在 Win32 平台上，应该使用 `"stdcall"`，或者其他应该使用的 ABI 字符串来链接它们自身的 Windows API。

还有一些特定于平台的 ABI 字符串：

- `extern "cdecl"` -- 通过 FFI 调用 x86_32 C 资源所使用的默认调用约定。
- `extern "stdcall"` -- 通过 FFI 调用 x86_32架构下的 Win32 API 所使用的默认调用约定
- `extern "win64"` -- 通过 FFI 调用 x86_64 Windows 平台下的 C 资源所使用的默认调用约定。
- `extern "sysv64"` -- 通过 FFI 调用 非Windows x86_64 平台下的 C 资源所使用的默认调用约定。
- `extern "aapcs"` --通过 FFI 调用 ARM 接口所使用的默认调用约定
- `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))`。

可变参数函数

可以在外部块内的函数的参数列表中的一个或多个具名参数后通过引入 `...` 来让该函数成为可变参数函数。注意可变参数 `...` 前至少有一个具名参数，并且只能位于参数列表的最后。可变参数可以通过标识符来指定：

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

外部块上的属性

下面列出的属性可以控制外部块的行为。

`link`属性

*`link` 属性*为外部(`extern`)块中的程序项指定编译器应该链接的本地库的名称。它使用*MetaListNameValueStr*元项属性句法指定其输入参数。 `name` 键指定要链接的本地库的名称。 `kind` 键是一个可选值，它指定具有以下可选值的库类型：

- `dylib` — 表示库类型是动态库。如果没有指定 `kind` ，这是默认值。

- `static` — 表示库类型是静态库。
- `framework` — 表示库类型是 macOS 框架。这只对 macOS 目标平台有效。

如果指定了 `kind` 键，则必须指定 `name` 键。

当从主机环境导入 symbols 时，`wasm_import_module` 键可用于为外部(`extern`)块中的程序项指定 WebAssembly模块名称。如果未指定 `wasm_import_module`，则默认模块名为 `env`。

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

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

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

在空外部块上添加 `link` 属性是有效的。可以用这种方式来满足代码中其他地方的外部块的链接需求（包括上游 crate），而不必向每个外部块都添加此属性。

`link_name`属性

可以在外部(`extern`)块内的程序项声明上指定 `link_name` 属性，可以用它来指示要为给定函数或静态项导入的具体 symbol。它使用 *MetaNameValueStr*元项属性句法指定 symbol 的名称。

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

函数参数上的属性

外部函数参数上的属性遵循与常规函数参数相同的规则和限制。

类型参数和生存期参数

generics.md
commit: 7fd47ef4786f86ccdb5d6f0f198a6a9fdec5497c
本章译文最后维护日期：2021-1-17

句法

GenericParams :
`<` `>`
| `<` (*GenericParam* `,`)* *GenericParam* `,`? `>`

GenericParam :
[*OuterAttribute*]* (*LifetimeParam* | *TypeParam* | *ConstParam*)

LifetimeParam :
[LIFETIME_OR_LABEL] (`:` [*LifetimeBounds*])?

TypeParam :
[IDENTIFIER](`:` [*TypeParamBounds*]?)? (`=` [*Type*])?

ConstParam:
`const` [IDENTIFIER] `:` [*Type*]

函数、类型别名、结构体、枚举、联合体、trait 和实现可以通过类型参数、常量参数和生存期参数达到*参数化*配置的的效果。这些参数在尖括号（`<...>`）中列出，通常都是紧跟在程序项名称之后和程序项的定义之前。对于实现，因为它没有名称，那它们就直接位于关键字 `impl` 之后。泛型参数的申明顺序是生存期参数在最前面，然后是类型参数，最后是常量参数。

下面给出一些带类型参数、常量参数和生存期参数的程序项的示例：

```
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程序项定义的范围内有效。它们不是函数体中声明的程序项，这个在程

序项声明中有讲述。

引用、裸指针、数组、切片、元组和函数指针也有生存期参数或类型参数，但这些程序项不能使用路径句法去引用。

常量泛型

*常量泛型参数*允许程序项在常量值上泛型化。const标识符为常量参数引入了一个名称，并且该程序项的所有实例必须用给定类型的值去实例化该参数。

常量参数类型值允许为：`u8`，`u16`，`u32`，`u64`，`u128`，`usize`，`i8`，`i16`，`i32`，`i64`，`i128`，`isize`，`char` 和 `bool` 这些类型。

常量参数可以在任何可以使用常量项的地方使用，但在类型或数组定义中的重复表达式中使用时，必须如下所述是独立的。也就是说，它们可以在以下地方上允许：

1. 可以用于类型内部，用它来构成所涉及的程序项签名的一部分。
2. 作为常量表达式的一部分，用于定义关联常量项，或作为关联类型的形参。
3. 作为程序项里的任何函数体中的任何运行时表达式中的值。
4. 作为程序项中任何函数体中使用到的任何类型的参数。
5. 作为程序项中任何字段类型的一部分使用。

```
// 可以使用常量泛型参数的示例。

// 用于程序项本身的签名
fn foo<const N: usize>(arr: [i32; N]) {
    // 在函数体中用作类型。
    let x: [i32; N];
    // 用作表达。
    println!("{}", N * 2);
}

// 用作结构体的字段
struct Foo<const N: usize>([i32; N]);

impl<const N: usize> Foo<N> {
    // 用作关联常数
    const CONST: usize = N * 4;
}

trait Trait {
    type Output;
}

impl<const N: usize> Trait for Foo<N> {
    // 用作关联类型
    type Output = [i32; N];
}
```

```
// 不能使用常量泛型参数的示例
fn foo<const N: usize>() {
    // 能在函数体中的程序项定义中使用
    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]);
}
```

作为进一步的限制，常量只能作为类型或数组定义中的重复表达式中的独立实参出现。在这种上下文限制下，它们只能以单段路径表达式的形式使用（例如 `N` 或以块 `{N}` 的形式出现）。也就是说，它们不能与其他表达式结合使用。

```
// 不能使用常量参数的示例。

// 不允许在类型中的表达式中组合使用，例如这里的返回类型中的算术表达式
fn bad_function<const N: usize>() -> [u8; {N + 1}] {
    // 同样的，这种情况也不允许在数组定义里的重复表达式中使用
    [1; {N + 1}]
}
```

路径中的常量实参指定了该程序项使用的常量值。实参必须是常量形参所属类型的常量表达式。常量表达式必须是块表达式（用花括号括起来），除非它是单独路径段（一个[标识符] [IDENTIFIER]）或一个字面量（此字面量可以是以 `-` 打头的 token）。

> **注意**：这种句法限制是必要的，用以避免在解析类型内部的表达式时可能会导致无限递归 (infinite lookahead)。

```
fn double<const N: i32>() {
    println!("doubled: {}", N * 2);
}

const SOME_CONST: i32 = 12;

fn example() {
    // 常量参数的使用示例。
    double::<9>();
    double::<-123>();
    double::<{7 + 8}>();
    double::<SOME_CONST>();
    double::<{ SOME_CONST + 5 }>();
}
```

当存在歧义时，如果泛型参数可以同时被解析为类型或常量参数，那么它总是被解析为类型。在块表达式中放置实参可以强制将其解释为常量实参。

```
type N = u32;
struct Foo<const N: usize>;
// 下面用法是错误的，因为 `N` 被解释为类型别名。
fn foo<const N: usize>() -> Foo<N> { todo!() } // ERROR
// 可以使用花括号来强制将 `N` 解释为常量形参。
fn bar<const N: usize>() -> Foo<{ N }> { todo!() } // ok
```

与类型参数和生存期参数不同，常量参数可以声明而不必在被它参数化的程序项中使用，但和泛型实现关联的实现例外：

```
// ok
struct Foo<const N: usize>;
enum Bar<const M: usize> { A, B }

// ERROR: 参数未使用
struct Baz<T>;
struct Biz<'a>;
struct Unconstrained;
impl<const N: usize> Unconstrained {}
```

当处理 trait约束时，在确定是否满足相关约束时，不会考虑常量参数的所有实现的穷尽性。例如，在下面的例子中，即使实现了 `bool` 类型的所有可能的常量值，仍会报错提示 trait约束不满足。

```
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约束 `Foo<B>: Bar` 未被满足
}
```

where子句

句法

WhereClause :
`where` (*WhereClauseItem* `,`)* *WhereClauseItem* ?

WhereClauseItem :
LifetimeWhereClauseItem
| *TypeBoundWhereClauseItem*

LifetimeWhereClauseItem :

[*Lifetime*] `:` [*LifetimeBounds*]

TypeBoundWhereClauseItem :
ForLifetimes? [*Type*] `:` [*TypeParamBounds*]?

ForLifetimes :
`for` `<` *GenericParams* `>`

---

*where子句*提供了另一种方法来为类型参数和生存期参数指定约束(bound)，甚至可以为非类型参数的类型指定约束。

关键字 `for` 可以用来引入高阶生存期参数。它只允许在 [*LifetimeParam*] 参数上使用。

定义程序项时，其约束没有使用程序项的泛型参数或高阶生存期参数，这样是可以通过编译器的安全检查，但这样的做法将必然导致错误。

在定义程序项时，编译器还会检查某些泛型参数的类型是否存在 `Copy`、`Clone` 和 `Sized` 这些约束。将 `Copy` 或 `Clone` 作为可变引用、trait object或slice这些程序项的约束是错误的，或将 `Sized` 作为 trait对象或切片的约束也是错误的。

```
struct A<T>
where
    T: Iterator,            // 可以用 A<T: Iterator> 来替代
    T::Item: Copy,
    String: PartialEq<T>,
    i32: Default,           // 允许，但没什么用
    i32: Iterator,          // 错误：此 trait约束不适合，i32 没有实现 Iterator
    [T]: Copy,              // 错误：此 trait约束不适合，切片上不能有此 trait约束
{
    f: T,
}
```

属性

泛型生存期参数和泛型类型参数允许属性，但在目前这个位置还没有任何任何有意义的内置属性，但用户可能可以通过自定义的派生属性来设置一些有意义的属性。

下面示例演示如何使用自定义派生属性修改泛型参数的含义。

```
// 假设 MyFlexibleClone 的派生项将 `my_flexible_clone` 声明为它可以理解的属性。
#[derive(MyFlexibleClone)]
struct Foo<#[my_flexible_clone(unbounded)] H> {
    a: *const H
}
```

关联项

associated-items.md
commit: 761ad774fcb300f2b506fed7b4dbe753cda88d80
本章译文最后维护日期：2021-1-17

句法

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

*关联程序项*是在 traits 中声明或在实现中定义的程序项。之所以这样称呼它们，是因为它们是被定义在一个相关联的类型（即实现里指定的类型）上的。关联程序项是那些可在模块中声明的程序项的子集。具体来说，有[关联函数][associated functions]（包括方法）、[关联类型][associated types]和[关联常量][associated constants]。

当关联程序项与被关联程序项在逻辑上相关时，关联程序项就非常有用。例如，`Option` 上的 `is_some` 方法内在逻辑定义上就与 Option枚举类型相关，所以它应该和 Option 关联在一起。

每个关联程序项都有两种形式：（包含实际实现的）定义和（为定义声明签名的）声明。[1]

正是这些声明构成了 trait 的契约(contract)以及其泛型参数中的可用内容。

关联函数和方法

*关联函数*是与一个类型相关联的函数。

*关联函数声明*为*关联函数定义*声明签名。它的书写格式和函数项一样，除了函数体被替换为 `;`。

标识符是关联函数的名称。关联函数的泛型参数、参数列表、返回类型 和 where子句必须与它们在*关联函数声明*中声明的格式一致。

*关联函数定义*定义与另一个类型相关联的函数。它的编写方式与函数项相同。

常见的关联函数的一个例子是 `new` 函数，它返回此关联函数所关联的类型的值。

```
struct Struct {
    field: i32
}

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

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

当关联函数在 trait 上声明时，此函数也可以通过一个指向 trait，再后跟函数名的路径来调用。当发生这种情况时，可以用 trait 的实际路径和关联函数的标识符按 `<_ as Trait>::function_name` 这样的形式来组织实际的调用路径。

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

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

// 在这个案例中，这4种形式都是等价的。
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);
```

方法

如果关联函数的参数列表中的第一个参数名为 `self`[2]，则此关联函数被称为*方法*，方法可以使用方法调用操作符(`.`)来调用，例如 `x.foo()`，也可以使用常用的函数调用形式进行调用。

如果名为 `self` 的参数的类型被指定了，它就通过以下文法（其中 `'lt` 表示生存期参数）来把此指定的参数限制解析成此文法中的一个类型：

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

此文法中的 `Self` 终结符(terminal)表示解析为实现类型(implementing type)的类型。这种解析包括解析上下文中的类型别名 `Self`、其他类型别名、或使用投射解析把（self 的类型中的）关联类型解析为实现类型。[3]

译者注：原谅译者对上面这句背后知识的模糊理解，那首先给出原文：
The `Self` terminal in this grammar denotes a type resolving to the implementing type. This can also include the contextual type alias `Self`, other type aliases, or associated type projections resolving to the implementing type.
译者在此先邀请读者中的高手帮忙翻译清楚。感谢感谢。另外译者还是要啰嗦以下译者对这句话背后知识的理解，希望有人能指出其中的错误，以让译者有机会进步：
首先终结符(terminal)就是不能再更细分的词法单元，可以理解它是一个 token，这里它代表 self（即方法接受者）的类型的基础类型。上面句法中的 P 代表一个产生式，它内部定义的规则是并联的，就是自动机在应用这个产生式时碰到任意符合条件的输入就直接进入终态。S 代表有限自动机从 S 这里开始读取 self 的类型。这里 S 是 Self 和 P 的并联，应该表示是：如果 self 的类型直接是 Self，那就直接进入终态，即返回 Self，即方法接收者的直接类型就是结果类型；如果 self 的类型是 P 中的任一种，就返回那一种，比如 self 的类型是一个 box指针，那么就返回 `Box<S>`。

```
// 结构体 `Example` 上的方法示例
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) {}
}
```

（方法的首参）可以在不指定类型的情况下使用简写句法，具体对比如下：

简写模式	等效项
`self`	`self: Self`
`&'lifetime self`	`self: &'lifetime Self`
`&'lifetime mut self`	`self: &'lifetime mut Self`

注意：（方法的）生存期也能，其实也经常是使用这种方式来省略。

如果 `self` 参数以 `mut` 为前缀，它就变成了一个可变的变量，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>) {}
}
```

以下是一个关于 trait 的方法的例子，现给定如下内容：

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

这里定义了一个带有两个方法的 trait。当此 trait 被引入当前作用域内后，所有此 trait 的实现的值都可以调用此 trait 的 `draw` 和 `bounding_box` 方法。

```
struct Circle {
    // ...
}

impl Shape for Circle {
    // ...
}

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

> **版本差异**: 在 2015 版中, 使用匿名参数来声明 trait方法是可能的 (例如： `fn foo(u8)`)。在 2018 版本中，这已被弃用，再用会导致编译错误。新版本种所有的参数都必须有参数名。

方法参数上的属性

方法参数上的属性遵循与常规函数参数上相同的规则和限制。

关联类型

*关联类型*是与另一个类型关联的类型别名(type aliases) 。关联类型不能在固有实现中定义，也不能在 trait 中给它们一个默认实现。

*关联类型声明*为*关联类型定义*声明签名。书写形式为：先是 `type`，然后是一个标识符，最后是一个可选的 trait约束列表。

这里的标识符是声明的类型的别名名称；可选的 trait约束必须由此类型别名的实现来履行实现。

*关联类型定义*在另一个类型上定义了一个类型别名。书写形式为：先是 `type`，然后是一个[标识符]，然后再是一个 `=`，最后是一个类型。

如果类型 `Item` 上有一个来自 trait `Trait` 的关联类型 `Assoc`，则表达式 `<Item as Trait>::Assoc` 也是一个类型，具体就是*关联类型定义*中指定的类型的一个别名。此外，如果 `Item` 是类型参数，则 `Item::Assoc` 也可以在类型参数中使用。

关联类型不能包括泛型参数或 where子句。

```
trait AssociatedType {
    // 关联类型声明
    type Assoc;
}

struct Struct;

struct OtherStruct;

impl AssociatedType for Struct {
    // 关联类型定义
    type Assoc = OtherStruct;
}

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

fn main() {
    // 使用 <Struct as AssociatedType>::Assoc 来引用关联类型 OtherStruct
    let _other_struct: OtherStruct = <Struct as
AssociatedType>::Assoc::new();
}
```

示例展示容器内的关联类型

下面给出一个 `Container` trait 示例。请注意，该类型可用在方法签名内：

```
trait Container {
    type E;
    fn empty() -> Self;
    fn insert(&mut self, elem: Self::E);
}
```

为了能让实现类型来实现此 trait，实现类型不仅必须为每个方法提供实现，而且必须指定类型 `E`。下面是一个为标准库类型 `Vec` 实现了此 `Container` 的实现：

```
impl<T> Container for Vec<T> {
    type E = T;
    fn empty() -> Vec<T> { Vec::new() }
    fn insert(&mut self, x: T) { self.push(x); }
}
```

关联常量

*关联常量*是与具体类型关联的常量。

*关联常量声明*为*关联常量定义*声明签名。书写形式为：先是 `const` 开头，然后是标识符，然后是 `:`，然后是一个类型，最后是一个 `;`。

这里标识符是（外部引用）路径中使用的常量的名称；类型是（此关联常量的）定义必须实现的类型。

*关联常量定义*定义了与类型关联的常量。它的书写方式与常量项相同。

示例展示关联常量

基本示例：

```
trait ConstantId {
    const ID: i32;
}

struct Struct;

impl ConstantId for Struct {
    const ID: i32 = 1;
}

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

使用默认值：

```
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);
}
```

[1] 固有实现中声明和定义是在一起的。

[2] 把简写形式转换成等价的标准形式。

[3] 结合下面的示例理解。

属性

attributes.md
commit: eabdf09207bf3563ae96db9d576de0758c413d5d
本章译文最后维护日期：2021-1-24

句法

InnerAttribute :
`#` `!` `[` *Attr* `]`

OuterAttribute :
`#` `[` *Attr* `]`

Attr :
SimplePath *AttrInput*?

AttrInput :
DelimTokenTree
| `=` *LiteralExpression* *不带后缀*

*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/解释器）依据名称、约定、语言和编译器版本进行解释。（Rust 语言中的）属性是根据 ECMA-335 标准中的属性规范进行建模的，其语法来自 ECMA-334 (C#) 。

*内部属性(Inner attributes)*以 `#!` 开头的方式编写，应用于它在其中声明的程序项。*外部属性(Outer attributes)*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(delimited token tree)（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(`=`)后跟文字表达式的格式。更多细节请参见下面的元项属性句法(meta item syntax)。

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

属性可以应用于语言中的许多场景：

- 所有的程序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，但只有当它们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(final expression)时才有效。
- 枚举(`enum`)变体和结构体(`struct`)、联合体(`union`)的字段可接受外部属性。
- 匹配表达式的匹配臂(arms)可接受外部属性。
- 泛型生存期(Generic lifetime)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。

属性的一些例子：

```
// 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 = ();
}
```

元项/元程序项属性句法

“元项(meta item)”是遵循 *Attr* 产生式（见本章头部）的句法，Rust 的大多数内置属性(built-in attributes)都使用了此句法。它有以下文法格式：

句法

MetaItem :
  SimplePath
| *SimplePath* `=` *LiteralExpression* *不带后缀*
| *SimplePath* `(` *MetaSeq*? `)`

MetaSeq :
  MetaItemInner (`,` MetaItemInner)* `,`?

MetaItemInner :
  MetaItem
| *LiteralExpression* *不带后缀*

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句法的不同子集来指定它们的输入。下面的文法规则展示了一些常用的使用形式：

句法

MetaWord:
  IDENTIFIER

MetaNameValueStr:
  IDENTIFIER `=` (STRING_LITERAL | RAW_STRING_LITERAL)

MetaListPaths:
  IDENTIFIER `(` (*SimplePath* (`,` *SimplePath*)* `,`?)? `)`

MetaListIdents:
  IDENTIFIER `(` (IDENTIFIER (`,` IDENTIFIER)* `,`?)? `)`

MetaListNameValueStr:

IDENTIFIER `(` (*MetaNameValueStr* (`,` *MetaNameValueStr*)* `,`?)? `)`

元项句法的一些例子是：

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。

```
// 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() {}
```

> 注意: `rustc` 目前能识别的工具是 “clippy” 和 “rustfmt”。

内置属性的索引表

下面是所有内置属性的索引表：

- 条件编译(Conditional compilation)
  - `cfg` — 控制条件编译。
  - `cfg_attr` — 选择性包含属性。
- 测试(Testing)
  - `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)
  - `derive` — 自动部署 trait实现
  - `automatically_derived` — 用在由 `derive` 创建的实现上的标记。
- 宏(Macros)
  - `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。
- 诊断(Diagnostics)
  - `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。
- 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)。
    - `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。
    - `crate_name` — 指定 crate名。
- 代码生成(Code generation)
    - `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()`。
- 文档(Documentation)
    - `doc` — 指定文档。更多信息见 The Rustdoc Book。Doc注释会被转换为 `doc` 属性。
- 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。
- 模块(Modules)
    - `path` — 指定模块的源文件名。
- 极限值设置(Limits)
    - `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。
- 运行时(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。参见 The Unstable Book 了解在 `rustc` 中实现的特性。
- 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。

测试类属性

testing.md
commit: b0e0ad6490d6517c19546b1023948986578fc378
本章译文最后维护日期：2020-11-10

以下属性用于指定函数来执行测试。在“测试(test)”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(test harness)。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：

- `()`
- `Result<(), E> where E: Error`

注意：允许哪些返回类型是由暂未稳定的 `Termination` trait 决定的。

注意：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。

返回 `()` 的测试只要结束(terminate)且没有触发 panic 就会通过。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（计为）通过也不（计为）失败。

```
#[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(annotated with)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。

`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。

```
#[test]
#[ignore = "not yet implemented"]
fn mytest() {
    // …
}
```

> **注意**： `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。

`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。

```
#[test]
#[should_panic(expected = "值未匹配上")]
fn mytest() {
    assert_eq!(1, 2, "值未匹配上");
}
```

派生

derive.md
commit: a52543267554541a95088b79f46a8bd36f487603
本章译文最后维护日期：2020-11-10

*derive属性*允许为数据结构自动生成新的程序项。它使用 *MetaListPaths*元项属性句法（为程序项）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[1] `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。

[1] 原文后半句是："and the type parameter `T` will be given the `PartialEq` or `Clone` constraints for the appropriate `impl`:"，这里译者也搞不清楚为什么 `PartialEq` 和 `Clone` 之间用了"or"，而不是"and"？这里译者就先采用直译。

诊断属性

diagnostics.md
commit: 12a4832c8eec1ad0df3edfb681571821708e0410
本章译文最后维护日期：2021-4-6

以下属性用于在编译期间控制或生成诊断消息。

lint检查类属性

（译者注：lint在原文里有时当名词用，有时当动词用，本文统一翻译成名词，意思就是一种被命名的 lint检查模式）

lint检查(lint check)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。`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*元项属性句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，
-

注意：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，也可以在 rustc book 中找到相关文档。

```
pub mod m1 {
    // 这里忽略未提供文档的编码行为
    #[allow(missing_docs)]
    pub fn undocumented_one() -> i32 { 1 }

    // 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_too() -> i32 { 2 }

    // 未提供文档的编码行为在这里将触发编译器报错
    #[deny(missing_docs)]
    pub fn undocumented_end() -> i32 { 3 }
}
```

Lint属性可以覆盖上一个属性指定的级别，但该级别不能更改禁止性的 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。

此示例展示了如何使用 `allow` 和 `warn` 来打开和关闭一个特定的 lint检查：

```
#[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 }
}
```

此示例展示了如何使用 `forbid` 来禁止对该 lint 使用 `allow`：

```
#[forbid(missing_docs)]
pub mod m3 {
    // 试图切换到 allow级别将触发编译器报错
    #[allow(missing_docs)]
    /// 返回 2。
    pub fn undocumented_too() -> i32 { 2 }
}
```

> 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。

lint分组

lint可以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。

```
// 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
}
```

有一个名为 “warnings” 的特殊组，它包含 “warn”级别的所有 lint。“warnings”组会忽略属性的顺序，并对实体执行所有告警lint 检查。

```
// 这里的两个属性的前后顺序无所谓
#[deny(warnings)]
// unsafe_code lint 默认是 "allow" 的。
#[warn(unsafe_code)]
fn example_err() {
    // 这里会编译报错，因为 `unsafe_code`告警的 lint 被提升为 "deny"级别了。
    unsafe { an_unsafe_fn() } // ERROR: usage of `unsafe` block
}
```

Tool lint attributes

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）。

在其他方面，这些工具类lint 就跟像常规的 lint 一样：

```
// 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() {
    // ...
}
```

注意：`rustc` 当前识别的工具类lint 有 "clippy" 和 "rustdoc"。

The `deprecated` attribute

`deprecated`属性

*`deprecated`属性*将程序项标记为已弃用。`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*元项属性句法形式里带有两个可选字段：
  - `since` — 指定程序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程序项，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。当应用于包含其他程序项的程序项时，如模块或实现，所有子程序项都继承此 deprecation属性。

这有个例子：

```
#[deprecated(since = "5.2", note = "foo was rarely used. Users should instead
use bar")]
pub fn foo() {}

pub fn bar() {}
```

RFC 内有动机说明和更多细节。

The `must_use` attribute

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。

```
#[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。

```
#[must_use]
fn five() -> i32 { 5i32 }

// 违反 unused_must_use lint。
five();
```

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实现(impl trait) ，则违反了 unused_must_use lint。

```
#[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。

```
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。

注意：包含了此（属性应用的程序项产生的值）的普通空操作(no-op)表达式不会违反该 lint。例如，将此类值包装在没有实现 `Drop` 的类型中，然后不使用该类型，并成为未使用的块表达式的最终表达式(final expression)。

```
#[must_use]
fn five() -> i32 { 5i32 }

// 这些都不违反 unused_must_use lint。
(five(),);
Some(five());
{ five() };
if true { five() } else { 0i32 };
match true {
    _ => five()
};
```

注意：当一个必须使用的值被故意丢弃时，使用带有模式 `_` 的let语句是惯用的方法。

```
#[must_use]
fn five() -> i32 { 5i32 }

// 不违反 unused_must_use lint。
let _ = five();
```

代码生成属性

codegen.md
commit: 646ef8d240a798da5891deb5dbdbebe557f878b8
本章译文最后维护日期：2020-11-10

下述属性用于控制代码生成。

优化提示

`cold` 属性和 `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。

这两个属性都可以在函数上使用。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。这两属性对 trait 中那些没有函数体的函数没有影响。

内联(`inline`)属性

`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。

> ***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。

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***: `#[inline]` 在每种形式中都是一个提示，不是*必须*要在调用者放置此属性限定的函

数的副本。

`cold`属性

`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。[2]

`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(platform architecture features)启用该函数的代码生成功能。它使用 *MetaListNameValueStr* 元项属性句法来启用（该平台支持的）特性，但这次要求这个句法里只能有一个 `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。

```
#[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。

可用特性

下面是可用特性列表。

`x86` or `x86_64`

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 — Streaming SIMD	单指令多数据流扩展

`sse`		Extensions	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

附加信息

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(override implementations)。

当应用于外部(`extern`)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(linked implementations)上，否则将导致未定义行为。当此属性应用在一个外部(`extern`)块内可用的函数上时，该外部(`extern`)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。

表现

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。

```
#[track_caller]
fn f() {
    println!("{}", std::panic::Location::caller());
}
```

> 注意：`core` 提供了 `core::panic::Location::caller` 来观察调用者的位置。它封装(wrap)了由 `rustc` 实现的内部函数(intrinsic) `core::intrinsics::caller_location`。

> 注意：由于结果 `Location` 是一个提示，所以具体实现可能会提前终止对堆栈的遍历。请参阅限制以了解重要的注意事项。

示例

当 `f` 直接被 `calls_f` 调用时， `f` 中的代码观察其在 `calls_f` 内的调用位置：

```
fn calls_f() {
    f(); // <-- f() 将打印此处的位置信息
}
```

`f` 被另一个有此属性限定的函数 `g` 调用， `g` 又被 `calls_g` 调用， `f` 和 `g` 内的代码又同时观察 `g` 在 `calls_g` 内的调用位置：

```
#[track_caller]
fn g() {
    println!("{}", std::panic::Location::caller());
    f();
}

fn calls_g() {
    g(); // <-- g() 将两次打印此处的位置信息，一次是它自己，一次是此 f() 里来的
}
```

当 `g` 又被另一个有此属性限定的函数 `h` 调用，而 `g` 又被 `calls_h` 调用， `f` 、 `g` 和 `h` 内的代码又同时观察 `h` 在 `calls_h` 内的调用位置：

```
#[track_caller]
fn h() {
    println!("{}", std::panic::Location::caller());
    g();
}

fn calls_h() {
    h(); // <-- 将三次打印此处的位置信息，一次是它自己，一次是此 g() 里来，一次是从 f() 里来的
}
```

以此类推。

限制

`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[3]

。

> 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。

[1] 可字面理解为内部反复试探。

[2] 默认情况下，Rust 编译器会默认某些标准库函数在编译时可用，编译器也会把当前编译的代码往这些库函数可用的方向去优化。

[3] 因为 trait对象的值不能直接使用，只能自动强转为指针引用，那这里的调用就无法观察到真实的调用位置。

极值设置

limits.md
commit: e7208a29f943e986c815734282c5cc5fd30f4708
本章译文最后维护日期：2021-3-26

以下属性影响部分编译期参数的极限值设置。

`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*元项属性句法来指定递归深度。

注意：`rustc` 中这个参数的默认值是128。

```
#![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*元项属性句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。

```
#![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

type_system.md
commit: d8cbe4eedb77bae3db9eff87b1238e7e23f6ae92
本章译文最后维护日期：2021-02-21

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*元项属性句法，因此不接受任何输入。

在当前（`non_exhaustive`限制的类型的）定义所在的 crate 内，`non_exhaustive` 没有效果。

```rust
#[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，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。

非穷尽类型(non-exhaustive types)不能在定义它的 crate 之外构建：

- 非穷尽变体（结构体(`struct`)或枚举变体(`enum` variant)）不能用 *StructExpression*句法（包括函数式更新(functional update)句法）构建。
- 枚举(`enum`)实例能被构建。

示例：（译者注：本例把上例看成本例的 `upstream` ）

```
// `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变体（结构体(`struct`)或枚举变体(`enum` variant)）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(`enum`)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(exhaustiveness)的这一要求。

示例：（译者注：可以把上上例看成本例的 `upstream` ）

```
// `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 => { },
}
```

非穷尽类型最好放在下游 crate 里。

语句和表达式

statements-and-expressions.md
commit: 4a2bdf896cd2df370a91d14cb8ba04e326cd21db
本章译文最后维护日期：2020-11-11

Rust *基本上*是一种表达式语言。这意味着大多数形式的求值或生成表达效果的计算的都是由*表达式*的统一句法类别来指导的。每一种表达式通常都可以*内嵌*到另一种表达式中，表达式的求值规则包括指定表达式产生的值和指定其各个子表达式的求值顺序。

对比之下，Rust 中的语句则*主要*用于包含表达式求值，以及显式地安排表达式的求值顺序。

语句

statements.md
commit: 245b8336818913beafa7a35a9ad59c85f28338fb
本章译文最后维护日期：2021-5-6

句法

Statement :
`;`
| *Item*
| *LetStatement*
| *ExpressionStatement*
| *MacroInvocationSemi*

*语句*是块(block)[1]的一个组件，反过来，块又是其外层表达式或函数的组件。

Rust 有两种语句：声明语句(declaration statements)和表达式语句(expression statements)。

声明语句

*声明语句*是在它自己封闭的语句块的内部引入一个或多个*名称*的语句。声明的名称可以表示新变量或新的程序项。

这两种声明语句就是程序项声明语句和 let声明语句。

程序项声明语句

*程序项声明语句*的句法形式与模块中的程序项声明的句法形式相同。在语句块中声明的程序项会将其作用域限制为包含该语句的块。这类程序项以及在其内声明子项(sub-items)都没有给定的规范路径。例外的是，只要程序项和 trait（如果有的话）的可见性允许，在（程序项声明语句内定义的和此程序项或 trait 关联的）实现中定义的关联项在外层作用域内仍然是可访问的。除了这些区别外，它与在模块中声明的程序项的意义也是相同的。

程序项声明语句不会隐式捕获包含它的函数的泛型参数、参数和局部变量。如下，`inner` 不能访问 `outer_var`。

`outer_var`.

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

  fn inner() { /* outer_var 的作用域不包括这里 */ }

  inner();
}
```

`let`语句

句法

LetStatement :
  *OuterAttribute** `let` *PatternNoTopAlt* (`:` *Type*)? (`=` *Expression*)? `;`

*`let`语句*通过一个不可反驳型模式引入了一组新的变量，变量由该模式给定。模式后面有一个可选的类型标注(annotation)，再后面是一个可选的初始化表达式。当没有给出类型标注时，编译器将自行推断类型，如果没有足够的信息来执行有限次的类型推断，则将触发编译器报错。由变量声明引入的任何变量从声明开始直到封闭块作用域结束都是可见的。

表达式语句

句法

ExpressionStatement :
    ExpressionWithoutBlock `;`
  | *ExpressionWithBlock* `;`?

*表达式语句*是对表达式求值并忽略其结果的语句。通常，表达式语句存在的目的是触发对其内部的表达式的求值时的效果。

仅由块表达式或控制流表达式组成的表达式，如果它们在允许使用语句的上下文中使用时，是可以省略其后面的分号的。这有可能会导致解析歧义，因为它可以被解析为独立语句，也可以被解析为另一个表达式的一部分；下例中的控制流表达式被解析为一个语句。注意 *ExpressionWithBlock* 形式的表达式用作语句时，其类型必须是单元类型(`()`)。

```
v.pop();            // 忽略从 pop 返回的元素
if v.is_empty() {
    v.push(5);
} else {
    v.remove(0);
}                   // 分号可以省略。
[1];                // 单独的表达式语句，而不是索引表达式。
```

当省略后面的分号时，结果必须是 `()` 类型。

```
// bad: 下面块的类型是i32，而不是 `()`
// Error: 预期表达式语句的返回值是 `()`
// if true {
//   1
// }

// good: 下面块的类型是i32，（加`;`后的语句的返回值就是 `()`了）
if true {
  1
} else {
  2
};
```

语句上的属性

语句可以有外部属性。在语句中有意义的属性是 `cfg` 和 lint检查类属性。

[1] 本书原文还有 `block of code` 的写法，这种有些类似于我们口语中说的那种任意的代码段的“代码块”。原文中 `block of code` 的典型情况是非安全(`unsafe`)块。

表达式

expressions.md
commit: 31dc83fe187a87af2b162801d50f4bed171fecdb
本章译文最后维护日期：2021-4-5

句法

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*
)

一个表达式可能有两个角色：它总是能产生[1]一个*值*；它还有可能表达出*效果(effects)*（也被称为“副作用(side effects)”）。表达式 *求值/计算为(evaluates to)* 值，并在 *求值(evaluation)* 期间表达出效果。
许多表达式包含子表达式，此子表达式也被称为此表达式的*操作数*。每种表达式都表达了以下几点含义：

- 在对表达式求值时是否对操作数求值
- 对操作数求值的顺序
- 如何组合操作数的值来获取表达式的值

基于对这几种含义的实现要求，表达式通过其内在结构规定了其执行结构。
块只是另一种表达式，所以块、语句和表达式可以递归地彼此嵌套到任意深度。

注意：我们给表达式的操作数做了（如上）命名，以便于我们去讨论它们，但这些名称并没有最终稳定下来，以后可能还会更改。

表达式的优先级

Rust 运算符和表达式的优先级顺序如下，从强到弱。具有相同优先级的二元运算符按其结合(associativity)顺序做了分组。

运算符/表达式	结合性
Paths（路径）	
Method calls（方法调用）	

Field expressions （字段表达式）	从左向右
Function calls, array indexing（函数调用，数组索引）	
`?`	
Unary（一元运算符） `-` `*` `!` `&` `&mut`	
`as`	从左向右
`*` `/` `%`	从左向右
`+` `-`	从左向右
`<<` `>>`	从左向右
`&`	从左向右
`^`	从左向右
`|`	从左向右
`==` `!=` `<` `>` `<=` `>=`	需要圆括号
`&&`	从左向右
`||`	从左向右
`..` `..=`	需要圆括号
`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`	从右向左
`return` `break` closures（返回、中断、闭包）	

操作数的求值顺序

下面的表达式列表都以相同的方式计算它们的操作数，具体列表后面也有详述。其他表达式要么不接受操作数，要么按照各自约定（后续章节会有讲述）的条件进行求值。

- 解引用表达式(Dereference expression)
- 错误传播表达式(Error propagation expression)
- 取反表达式(Negation expression)
- 算术和二进制逻辑运算(Arithmetic and logical binary operators)
- 比较运算(Comparison operators)
- 类型转换表达式(Type cast expression)
- 分组表达式(Grouped expression)
- 数组表达式(Array expression)
- 等待表达式(Await expression)

- 索引表达式(Index expression)
- 元组表达式(Tuple expression)
- 元组索引表达式(Tuple index expression)
- 结构体表达式(Struct expression)
- 调用表达式(Call expression)
- 方法调用表达式(Method call expression)
- 字段表达式(Field expression)
- 中断表达式(Break expression)
- 区间表达式(Range expression)
- 返回表达式(Return expression)

在实现执行这些表达式的效果之前，会先对这些表达式的操作数进行求值。拥有多个操作数的表达式会按照源代码书写的顺序从左到右计算。

> **注意**：子表达式是一个表达式的操作数时，此子表达式内部的求值顺序是由根据前面的章节规定的优先级来确定的。

例如，下面两个 `next` 方法的调用总是以相同的顺序调用：

```
let mut one_two = vec![1, 2].into_iter();
assert_eq!(
    (1, 2),
    (one_two.next().unwrap(), one_two.next().unwrap())
);
```

> **注意**：由于表达式是递归执行的，那这些表达式也会从最内层到最外层逐层求值，忽略兄弟表达式，直到没有（未求值的）内部子表达式为止。

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（，它们各自形成了自己的上下文）。因此，在每个表达式中，操作数可以出现在位置上下文，也可出现在值上下文中。表达式的求值既依赖于它自己的类别，也依赖于它所在的上下文。

*位置表达式*是表示内存位置的表达式。语言中表现为指向**局部变量**、**静态变量**、**解引**

用(`*expr`)、数组索引表达式(`expr[expr]`)、字段引用(`expr.f`)、圆括号括起来的位置表达式的路径。所有其他形式的表达式都是值表达式。

*值表达式*是代表实际值的表达式。

下面的上下文是*位置表达式*上下文：

- 赋值或复合赋值表达式的左操作数。
- 一元运算符借用或解引用的操作数。
- 字段表达式的操作数。
- 数组索引表达式的被索引操作数。
- 任何隐式借用的操作数。
- let语句的初始化器(initializer)。
- `if let` 表达式、 `while let` 表达式或匹配(`match`)表达式的检验对象(scrutinee)。
- 结构体表达式里的函数式更新(functional update)的基(base)。

注意：历史上，位置表达式被称为 *左值/lvalues*，值表达式被称为 *右值/rvalues*。

移动语义类型和复制语义类型

当位置表达式在值表达式上下文中被求值时，或在模式中被值绑定时，这表示此值会*保存进(held in)*当前表达式代表的内存地址。如果该值的类型实现了 `Copy` ，那么该值将被从原来的位置复制一份过来。如果该值的类型没有实现 `Copy` ，但实现了 `Sized` ，那么就有可能把该值从原来的位置移动(move)过来。从位置表达式里移出值对位置表达式也有要求，只有如下的位置表达式才可能把值从其中移出(move out)：

- 变量当前未被借用。
- 临时值(Temporary values)。
- 可以移出且没实现 `Drop` 的位置表达式的字段。
- 对可移出且类型为 `Box<T>` 的表达式作解引用的结果。

把值从一个位置表达式里移出到一个局部变量，那此（表达式代表的）地址将被去初始化(deinitialized)，并且该地址在重新初始化之前无法被再次读取。
除以上列出的情况外，任何在值表达式上下文中使用位置表达式都是错误的。

可变性

如果一个位置表达式将会被赋值、可变借出、隐式可变借出或被绑定到包含 `ref mut` 模式上，则该位置表达式必须是*可变的(mutable)*。我们称这类位置表达式为*可变位置表达式*。与之相对，其他位置表达式称为*不可变位置表达式*。

下面的表达式可以是可变位置表达式上下文：

- 当前未被借出的可变变量。
- 可变静态(`static`)项。
- 临时值。
- 字段，在可变位置表达式上下文中，可以对此子表达式求值。
- 对 `*mut T` 指针的解引用。
- 对类型为 `&mut T` 的变量或变量的字段的解引用。注意：这条是下一条规则的必要条件的例外情况。[2]
- 对实现 `DerefMut` 的类型的解引用，那对这里解出的表达式求值就需要在一个可变位置表达式上下文中进行。
- 对实现 `IndexMut` 的类型做索引，那对此检索出的表达式求值就需要在一个可变位置表达式上下文进行。注意对索引(index)本身的求值不用。

临时位置/临时变量

在大多数位置表达式上下文中使用值表达式时，会创建一个临时的未命名内存位置，并将该位置初始为该值，然后这个表达式的求值结果就为该内存位置。此过程也有例外，就是把此临时表达式提升为一个静态项(`static`)。（译者注：这种情况下表达式将直接在编译时就求值了，求值的结果会根据编译器要求重新选择地址存储）。临时位置/临时变量的销毁作用域(drop scope)通常在包含它的最内层语句的结尾处。

隐式借用

某些特定的表达式可以通过隐式借用一个表达式来将其视为位置表达式。例如，可以直接比较两个非固定尺寸的切片是否相等，因为 `==` 操作符隐式借用了它自身的操作数：

```
let a: &[i32];
let b: &[i32];
// ...
*a == *b; //译者注：&[i32] 解引用后是一个动态尺寸类型，理论上两个动态尺寸类型上无法比较大小的，但这里因为隐式借用此成为可能
// 等价于下面的形式：
::std::cmp::PartialEq::eq(&*a, &*b);
```

隐式借用可能会被以下表达式采用：

- 方法调用表达式中的左操作数。
- 字段表达式中的左操作数。
- 调用表达式中的左操作数。
- 数组索引表达式中的左操作数。
- 解引用操作符（`*`）的操作数。
- 比较运算的操作数。
- 复合赋值(compound assignment)的左操作数。

重载 trait

本书后续章节的许多操作符和表达式都可以通过使用 `std::ops` 或 `std::cmp` 中的 trait 被其他类型重载。这些 trait 也存在于同名的 `core::ops` 和 `core::cmp` 中。

表达式属性

只有在少数特定情况下，才允许在表达式之前使用外部属性：

- 在被当作语句用的表达式之前。
- 数组表达式、元组表达式、调用表达式、元组结构体(tuple-style struct)表达式这些中的元素。
- 块表达式的尾部表达式(tail expression)。

在下面情形之前是不允许的：

- 区间(Range)表达式。
- 二元运算符表达式
  （*ArithmeticOrLogicalExpression*、*ComparisonExpression*、*LazyBooleanExpression*、*Typ*

[1] 通俗理解“求值”，可以理解为：一方面可以从表达式求出值，另一方面可以为表达式赋值。

[2] 译者暂时还未用代码测试这种例外情况。

字面量表达式

literal-expr.md
commit: eb5290329316e96c48c032075f7dbfa56990702b
本章译文最后维护日期：2021-02-21

句法

LiteralExpression :
CHAR_LITERAL
| STRING_LITERAL
| RAW_STRING_LITERAL
| BYTE_LITERAL
| BYTE_STRING_LITERAL
| RAW_BYTE_STRING_LITERAL
| INTEGER_LITERAL
| FLOAT_LITERAL
| BOOLEAN_LITERAL

*字面量表达式*由字面量里讲过的任一形式组成。 它直接描述一个数字、字符、字符串或布尔值。

```
"hello";   // 字符串类型
'5';       // 字符类型
5;         // 整型
```

路径表达式

path-expr.md
commit: eb5290329316e96c48c032075f7dbfa56990702b
本章译文最后维护日期：2021-02-21

句法

PathExpression :
PathInExpression
| *QualifiedPathInExpression*

路径被用做表达式上下文时表示局部变量或程序项。 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 `static mut` 变量需在 `unsafe` 块中。

```
local_var;
globals::STATIC_VAR;
unsafe { globals::STATIC_MUT_VAR };
let some_constructor = Some::<i32>;
let push_integer = Vec::<i32>::push;
let slice_reverse = <[i32]>::reverse;
```

块表达式

block-expr.md
commit: 23672971a16c69ea894bef24992b74912cfe5d25
本章译文最后维护日期：2021-4-5

句法

BlockExpression :
`{`
*InnerAttribute**
Statements?
`}`

Statements :
Statement+
| *Statement*+ *ExpressionWithoutBlock*
| *ExpressionWithoutBlock*

*块表达式*或*块*是一个控制流表达式(control flow expression)，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，块按顺序执行其非程序项声明的语句组件，最后执行可选的最终表达式(final expression)。 作为一个匿名空间作用域，在本块内声明的程序项只在块本身围成的作用域内有效，而块内由 `let` 语句声明的变量的作用域为下一条语句到块尾。

块的句法规则为：先是一个 `{`，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（final operand）的可选表达式，最后是一个 `}`。

语句之间通常需要后跟分号，但有两个例外： 1、程序项声明语句不需要后跟分号。 2、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，除了程序项声明语句外，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(final operand)，则最后会执行它。

块的类型是最此块的最终操作数(final operand)的类型，但如果省略了最终操作数，则块的类型为 `()`。

```
let _: () = {
    fn_call();
};

let five: i32 = {
    fn_call();
    5
};

assert_eq!(5, five);
```

> 注意：作为控制流表达式，如果块表达式是一个表达式语句的外层表达式，则该块表达式的预期类型为 `()`，除非该块后面紧跟着一个分号。

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。

> **注意**：如果确实有需要，块可以用于强制移动值。 例如，下面的示例在调用 `consume_self` 时失败，因为结构体已经在之前的块表达式里被从 `s` 里移出了。

```
struct Struct;

impl Struct {
    fn consume_self(self) {}
    fn borrow_self(&self) {}
}

fn move_by_block_expression() {
    let s = Struct;

    // 将值从块表达式里的 `s` 里移出。
    (&{ s }).borrow_self();

    // 执行失败，因为 `s` 里的值已经被移出。
    s.consume_self();
}
```

async块

句法

AsyncBlockExpression :
`async` `move`? *BlockExpression*

---

*异步块(async block)*是求值为 *future* 的块表达式的一个变体。 块的最终表达式（如果存在）决定了 future 的结果值。（译者注：单词 future 对应中文为“未来”。原文可能为了双关的行文效果，经常把作为类型的 future 和字面意义上的 future 经常混用，所以译者基本保留此单词不翻译，特别强调“未来”的意义时也会加上其英文单词。）

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。 类似闭包返回的类型实现了一个或多个 `std::ops::Fn` trait，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。

版本差异: 异步块从 Rust 2018 版才开始可用。

捕获方式

异步块使用与闭包相同的捕获方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。

最后，关键字 `break` 和 `continue` 不能用于从异步块中跳出分支。 因此，以下内容是非法的：

```
loop {
    async move {
        break; // 这将打破循环。
    }
}
```

非安全(`unsafe`)块

句法\23672971a16c69ea894bef24992b74912cfe5d25e` 的信息_

可以在代码块前面加上关键字 `unsafe` 以允许非安全操作。 例如：

```
#![allow(unused)]
fn main() {
```

在以下上下文中，允许在块表达式的左括号之后直接使用[内部属性][inner attributes]：

* [函数][function]和[方法][method]的代码体。
* 循环体 ([`loop`], [`while`], [`while let`], 和 [`for`]) 。
* 被用作[语句][statement]的块表达式。
* 块表达式作为[数组表达式][array expressions]、[元组表达式][tuple expressions]、[调用表达式][call expressions]、[元组结构体][struct]表达式和[枚举变体][enum variant]表达式的元素。
* 作为另一个块表达式的尾部表达式(tail expression)的块表达式。

<!-- 本列表需要和 expressions.md 保持同步 -->

在块表达式上有意义的属性有 [`cfg`] 和 [lint检查类属性][the lint check attributes]。

例如，下面这个函数在 unix 平台上返回 `true`，在其他平台上返回 `false`。

```rust
fn is_unix_platform() -> bool {
    #[cfg(unix)] { true }
    #[cfg(not(unix))] { false }
}
}
```

操作符/运算符表达式

operator-expr.md
commit: 0a626ce599bcae4fa1a48535c0883beaca38f4db
本章译文最后维护日期：2021-5-6

句法

OperatorExpression :
BorrowExpression
| *DereferenceExpression*
| *ErrorPropagationExpression*
| *NegationExpression*
| *ArithmeticOrLogicalExpression*
| *ComparisonExpression*
| *LazyBooleanExpression*
| *TypeCastExpression*
| *AssignmentExpression*
| *CompoundAssignmentExpression*

操作符是 Rust 语言为其内建类型定义的。 本文后面的许多操作符都可以使用 `std::ops` 或 `std::cmp` 中的 trait 进行重载。

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，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。

借用/引用操作符/运算符

句法

BorrowExpression :
`&` | `&&` *Expression*
| `&` | `&&` `mut` *Expression*

`&`（共享借用）和 `&mut`（可变借用）运算符是一元前缀运算符。 当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。

这类操作符不能重载。

```
{
    // 将创建一个存值为7的临时位置，该该临时位置在此作用域内持续存在
    let shared_reference = &7;
}
let mut array = [-2, 3, 9];
{
    // 在当前作用域内可变借用了 `array`。那 `array` 就只能通过 `mutable_reference`
来使用。
    let mutable_reference = &mut array;
}
```

尽管 `&&` 是一个单一 token（惰性与(`and`)操作符），但在借用表达式(borrow expressions)上下文中使用时，它是作为两个借用操作符用的：

```
// 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;
```

解引用操作符

句法

DereferenceExpression :
`*` *Expression*

`*`（解引用）操作符也是一元前缀操作符。当应用于指针上时，它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。解引用原始指针需要在非安全(`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)`；同样的，在可变位置表达式上下文中这个动作就相当于执行 `*std::ops::DerefMut::deref_mut(&mut x)`。

```
let x = &7;
assert_eq!(*x, 7);
let y = &mut 9;
*y = 11;
assert_eq!(*y, 11);
```

问号操作符

句法

ErrorPropagationExpression :
Expression `?`

问号操作符（`?`）解包(unwrap)有效值或返回错误值，并将它们传播(propagate)给调用函数。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>`。

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`。

```
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);
```

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`。

```
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);
```

操作符 `?` 不能被重载。

取反运算符

语法

NegationExpression :
    `-` *Expression*
  | `!` *Expression*

这是最后两个一元运算符。 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。 记住，有符号整数总是用二进制补码形式表示。 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。

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。

下面是这些运算符的一些示例：

```
let x = 6;
assert_eq!(-x, -6);
assert_eq!(!x, -7);
assert_eq!(true, !false);
```

算术和逻辑二元运算符

句法

ArithmeticOrLogicalExpression :
Expression `+` *Expression*
| *Expression* `-` *Expression*
| *Expression* `*` *Expression*
| *Expression* `/` *Expression*
| *Expression* `%` *Expression*
| *Expression* `&` *Expression*
| *Expression* `|` *Expression*
| *Expression* `^` *Expression*
| *Expression* `<<` *Expression*
| *Expression* `>>` *Expression*

二元运算符表达式都用中缀表示法(infix notation)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。 记住，有符号整数总是用二进制补码形式表示。 所有这些运算符的操作数都在值表达式上下文中求值，因此这些操作数的值会被移走或复制。

符号	整数	`bool`	浮点数	用于重载此运算符的 trait	用于重载此运算符值(Compou Assignment)

`+`	加法		加法	`std::ops::Add`	`std::ops::AddAs`
`-`	减法		减法	`std::ops::Sub`	`std::ops::SubAs`
`*`	乘法		乘法	`std::ops::Mul`	`std::ops::MulAs`
`/`	除法*		取余	`std::ops::Div`	`std::ops::DivAs`
`%`	取余		Remainder	`std::ops::Rem`	`std::ops::RemAs`
`&`	按位与	逻辑与		`std::ops::BitAnd`	`std::ops::BitAn`
`|`	按位或	逻辑或		`std::ops::BitOr`	`std::ops::BitOr`
`^`	按位异或	逻辑异或		`std::ops::BitXor`	`std::ops::BitXo`
`<<`	左移位			`std::ops::Shl`	`std::ops::ShlAs`
`>>`	右移位**			`std::ops::Shr`	`std::ops::ShrAs`

* 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。

下面是使用这些操作符的示例:

```
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);
```

比较运算符

句法

ComparisonExpression :
Expression `==` *Expression*
| *Expression* `!=` *Expression*
| *Expression* `>` *Expression*
| *Expression* `<` *Expression*
| *Expression* `>=` *Expression*
| *Expression* `<=` *Expression*

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。 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`。

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，这些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：

```
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(moved out of)。

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`

下面是使用比较运算符的示例：

```
assert!(123 == 123);
assert!(23 != -12);
assert!(12.5 > 12.2);
assert!([1, 2, 3] < [1, 3, 4]);
assert!('A' <= 'B');
assert!("World" >= "Hello");
```

短路布尔运算符

句法

LazyBooleanExpression :
    Expression `||` *Expression*
  | *Expression* `&&` *Expression*

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。

```
let x = false || true; // true
let y = false && panic!(); // false, 不会计算 `panic!()`
```

类型转换表达式

句法

TypeCastExpression :
Expression `as` *TypeNoBounds*

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。

类型转换(`as`)表达式的一个例子：

```
fn average(values: &[f64]) -> f64 {
    let sum: f64 = sum(values);
    let size: f64 = len(values) as f64;
    sum / size
}
```

`as` 可用于显式执行自动强转(coercions)，以及下列形式的强制转换。 任何不符合强转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` 。 `m` 引用类型中代表可选的 `mut` 或指针类型中的 `mut` 或 `const` 。

`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
$\&m_1$ `T`	$*m_2$ `T` **	引用到指针的转换
$\&m_1$ `[T; n]`	$*m_2$ `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 m_1 是 `mut` 或 m_2 是 `const` 时， 可变(`mut`)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。

语义

- 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。

赋值表达式

句法

AssignmentExpression :
  Expression `=` *Expression*

*赋值表达式*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。

> **注意**：此表达式与其他表达式的求值顺序不同，此表达式的右操作数在左操作数之前被求值。

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。

示例：

```
let mut x = 0;
let y = 0;
x = y;
```

复合赋值表达式

句法

CompoundAssignmentExpression :
    Expression `+=` *Expression*
  | *Expression* `-=` *Expression*
  | *Expression* `*=` *Expression*
  | *Expression* `/=` *Expression*
  | *Expression* `%=` *Expression*
  | *Expression* `&=` *Expression*
  | *Expression* `|=` *Expression*
  | *Expression* `^=` *Expression*
  | *Expression* `<<=` *Expression*
  | *Expression* `>>=` *Expression*

*复合赋值表达式*将算术符（以及二进制逻辑操作符）与赋值表达式相结合在一起使用。

比如：

```
let mut x = 5;
x += 1;
assert!(x == 6);
```

复合赋值的句法是可变 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。

> **注意**：此表达式与其他表达式的求值顺序不同，此表达式的右操作数在左操作数之前被求值。

此外，这个表达式是调用操作符重载复合赋值trait 的函数的语法糖（见本章前面的表格）。 被赋值操作数必须是可变的。

例如，下面 `example` 函数中的两个表达式语句是等价的：

```
impl AddAssign<Addable> for Addable {
    /* */
}

fn example() {
  a1 += a2;

  AddAssign::add_assign(&mut a1, a2);
}
```

与赋值表达式一样，复合赋值表达式也总是会生成单元类型值。

> ⚠️ 警告：复合赋值表达式的操作数的求值顺序取决于操作数的类型：对于原生类型，右边操作数将首先被求值，而对于非原生类型，左边操作数将首先被求值。 建议尽量不要编写依赖于复合赋值表达式中操作数的求值顺序的代码。请参阅这里的测试以获得使用此依赖项的示例。

[1] 截断，即一个值范围较大的变量A转换为值范围较小的变量B，如果超出范围，则将A减去B的区间长度。例如，128超出了i8类型的范围（-128,127），截断之后的值等于128-256=-128。

圆括号表达式(分组表达式)

grouped-expr.md
commit: 31dc83fe187a87af2b162801d50f4bed171fecdb
本章译文最后维护日期：2021-4-5

句法

GroupedExpression :
`(` *InnerAttribute** *Expression* `)`

由圆括号封闭的表达式的求值结果就是在其内的表达式的求值结果。 在表达式内部，圆括号可用于显式地指定表达式内部的求值顺序。

*圆括号表达式(parenthesized expression)*包装单个表达式，并对该表达式求值。 圆括号表达式的句法规则就是一对圆括号封闭一个被称为*封闭操作数(enclosed operand)*的表达式。

圆括号表达式被求值为其封闭操作数的值。 与其他表达式不同，圆括号表达式可以是[位置表达式或值表达式][place]。 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。

圆括号表达式的一个例子：

```
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，示例如下：

```
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。

md#attributes-on-block-expressions [place]: ../expressions.md#place-expressions-and-value-expressions

数组和数组索引表达式

array-expr.md
commit: 31dc83fe187a87af2b162801d50f4bed171fecdb
本章译文最后维护日期：2021-4-5

数组表达式

句法

ArrayExpression :
`[` *InnerAttribute** *ArrayElements*? `]`

ArrayElements :
Expression (`,` *Expression*)* `,`?
| *Expression* `;` *Expression*

*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。 这种形式的句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。

第二种形式的句法规则通过在方括号内放置两用个用分号(`;`)分隔的表达式来表现。 分号(`;`)前的表达式被称为*重复体操作数(repeat operand)* 分号(`;`)后的表达式被称为*数组长度操作数(length operand)* 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。 也就是说， `[a; b]` 这种形式会创建包含 `b` 个 `a` 值的数组。 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。

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。

⚠️ 警告：当数组长度操作数为 0 时，而重复体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。参见 issue#74836。

```
[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。

数组和切片索引表达式

句法

IndexExpression :
  Expression `[` *Expression* `]`

数组和切片类型的值(slice-typed values)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。

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*。

```
// 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
```

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。

元组和元组索引表达式

tuple-expr.md
commit: 09142b820fe8713c4cba451713c7d11e67d7fbd8
本章译文最后维护日期：2021-4-6

元组表达式

句法

TupleExpression :
`(` *InnerAttribute** *TupleElements*? `)`

TupleElements :
(*Expression* `,`)+ *Expression*?

*元组表达式*用来构建元组值。

元组表达式的句法规则为：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(tuple initializer operands)*。 为了避免和圆括号表达式混淆，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。

元组索引表达式

句法

TupleIndexingExpression :
Expression `.` TUPLE_INDEX

*元组索引表达式*被用来存取元组或[元组结构体][tuple structs]的字段。

元组索引表达式的句法规则为：一个被称为*元组操作数(tuple operand)*的表达式后跟一个`.`，最后再后跟一个元组索引。 *元组索引*的句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。

元组操作数的类型必须是元组类型或[元组结构体][tuple structs]。 元组索引必须是元组操作数类型的字段的名称。（译者注：这句感觉原文表达有问题，这里也给出原文 The tuple index must be a name of a field of the type of the tuple operand.）

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。

元组索引表达式示例：

```
// 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);
```

注意：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。（这之所以可行，）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。

注意：虽然数组和切片也有元素，但它们必须使用数组或切片索引表达式或切片模式去访问它们的元素。

结构体表达式

struct-expr.md
commit: eb5290329316e96c48c032075f7dbfa56990702b
本章译文最后维护日期：2021-02-21

句法

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*

*结构体表达式*创建结构体或联合体的值。它由指向结构体程序项、枚举变体、联合体程序项的路径，以及与此程序项的字段对应的值组成。 结构体表达式有三种形式：结构体(struct)、元组结构体(tuple)和单元结构体(unit)。

下面是结构体表达式的示例：

```
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);
```

结构体表达式的字段设置

用花括号把字段括起来的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。

联合体类型的值只能使用此句法创建，并且只能指定一个字段。

函数式更新句法

结构体表达式构建一个结构体类型的值时可以以 `..` 后跟一个表达式的句法结尾，这种句法表示这是一种函数式更新(functional update)。 `..` 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。

整个结构体表达式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，此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。

```
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);
```

带花括号的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。 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。
```

初始化结构体字段的快捷方法

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。 例如：

For example: For example:

```
Point3d { x: x, y: y_value, z: z };
Point3d { x, y: y_value, z };
```

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。例如：

```
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程序项(unit struct item)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。例如：

```
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。

调用表达式

call-expr.md
commit: 31dc83fe187a87af2b162801d50f4bed171fecdb
本章译文最后维护日期：2021-4-5

句法

CallExpression :

Expression `(` *CallParams*? `)`

CallParams :

Expression (`,` *Expression*)* `,`?

*调用表达式*用来调用函数。 调用表达式的句法规则为：一个被称作*函数操作数(function operand)*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(argument operands)*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。 如有需要，也可通过自动借用。 Rust 也会根据需要自动对 `f` 作解引用处理。

下面是一些调用表达式的示例：

```
let three: i32 = add(1i32, 2i32);
let name: &'static str = (|| "Rust")();
```

函数调用的消歧

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。

注意：过去，Rust 社区在文档、议题、RFC 和其他社区文章中使用了术语“确定性函数调

> 用句法(Unambiguous Function Call Syntax)”、“通用函数调用句法(Universal Function Call Syntax)” 或 “UFCS”。 但是，这个术语缺乏描述力，可能还会混淆当前的议题。 我们在这里提起这个词是为了便于搜索。

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接受者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，程序员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` 的程序项
    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);
}
```

更多细节和动机说明请参考RFC 132。

方法调用表达式

method-call-expr.md
commit: eb5290329316e96c48c032075f7dbfa56990702b
本章译文最后维护日期：2021-02-21

句法

MethodCallExpression :

Expression `.` *PathExprSegment* `(` *CallParams*? `)`

*方法调用*由一个表达式（*接受者(receiver)*）后跟一个单点号(`.`)、一个表达式路径段(path segment)和一个圆括号封闭的的表达式列表组成。 方法调用被解析为特定 trait 上的关联方法时，如果点号左边的表达式有确切的已知的 `self` 类型，则会静态地分发(statically dispatch)给在此类型下查找到的某个同名方法来执行；如果点号左边的表达式是间接的 trait对象，则会采用动态分发(dynamically dispatch)的方式。

trait object.

```
let pi: Result<f32, _> = "3.14".parse();
let log_pi = pi.unwrap_or(1.0).log(2.72);
```

在查找方法调用时，为了调用某个方法，可能会自动对接受者做解引用或借用。 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。通过重复对接受者表达式的类型作解引用，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非固定尺寸类型自动强转(unsized coercion)，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，接受者的类型为 `Box<[i32;2]>`，则候选类型为 `Box<[i32;2]>`， `&Box<[i32;2]>`， `&mut Box<[i32;2]>`， `[i32; 2]`（通过解引用得到）， `&[i32; 2]`， `&mut [i32; 2]`， `[i32]`（通过非固定尺寸类型自动强转得到）， `&[i32]`，最后是 `&mut [i32]`。

然后，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接受者：

1. `T` 的固有方法（直接在 `T` 上实现的方法）。
2. 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。

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();
}
```

如果上面这第二步查找导致了多个可能的候选类型[1]，就会导致报错，此时必须将接受者转换为适当的接受者类型再来进行方法调用。

此方法过程不考虑接受者的可变性或生存期，也不考虑方法是否为非安全(`unsafe`)方法。 一旦查找到了一个方法，如果由于这些（可变性、生存期或健全性）原因中的一个（或多个）而不能调用，则会报编译错误。

如果某步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的消歧句法来为方法调用或函数调用消除歧义。

⚠ ***警告：*** 对于 trait对象，如果有一个与 trait方法同名的固有方法，那么当尝试在方法调用表达式(method call expression)中调用该方法时，将编译报错。 此时，可以使用消除函数调用歧义的句法来明确调用语义。 在 trait对象上使用消除函数调用歧义的句法，将只能调用 trait方法，无法调用固有方法。 所以只要不在 trait对象上定义和 trait方法同名的固有方法就不会碰到这种麻烦。

[1]：这个应该跟后面说的方法名歧义了一样，如果类型 `T` 的两个 trait 都有调用的那个方法，那此方法的调用者类型就不能确定了，就需要消歧。

字段访问表达式

field-expr.md
commit: 23672971a16c69ea894bef24992b74912cfe5d25
本章译文最后维护日期：2021-4-5

句法

FieldExpression :
Expression `.` IDENTIFIER

*字段表达式(field expression)*是计算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，其字段表达式也是可变的。

*字段表达式(field expression)*的句法规则为：一个被称为*容器操作数(container operand)*的表达式后跟一个单点号(`.`)，最后是一个标识符。 字段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）。 因此字段表达式不能是调用表达式的函数调用者。

字段表达式代表结构体(`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。

注意：如果要在调用表达式中使用它(来调用函数)，要把此字段表达式先用圆括号包装成一个圆括号表达式。

```
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)();
```

示例：

```
mystruct.myfield;
foo().x;
(Struct {a: 10, b: 20}).a;
(mystruct.function_field)() // 调用表达式里包含一个字段表达式
```

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)*。

借用

当借用时，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，（这种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` 化之外的用户自定义类型执行自动解引用，这（种各个字段被视为彼此分离的单独实体的逻辑）就不适用了。

```
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
```

闭包表达式

closure-expr.md
commit: d23f9da8469617e6c81121d9fd123443df70595d
本章译文最后维护日期：2021-5-6

句法

ClosureExpression :
`move`?
(`||` | `|` *ClosureParameters*? `|`)
(*Expression* | `->` *TypeNoBounds BlockExpression*)

ClosureParameters :
ClosureParam (`,` *ClosureParam*)* `,`?

ClosureParam :
*OuterAttribute** *PatternNoTopAlt* (`:` *Type*)?

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。 闭包表达式的句法规则为：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(closure parameters)的模式列表（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(closure body operand)* 的表达式。 代表闭包参数的每个模式后面的可选类型是该模式的类型标注(type annotations)。 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模式的，其类型标注是可选的，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。

特别值得注意的是闭包表达式能*捕获它们被定义时的环境中的变量(capture their environment)*，而普通的函数定义则不能。 如果没有关键字 `move`，闭包表达式将[推断它该如何从其环境中捕获每个变量][infers how it captures each variable from its environment]，它倾向于通过共享引用来捕获，从而有效地借用闭包体中用到的所有外部变量。 如果有必要，编译器会推断出应该采用可变引用，还是应该从环境中移动或复制值（取决于这些变量的类型）。 闭包可以通过前缀关键字 `move` 来强制通过复制值或移动值的方式捕获其环境变量。 这

通常是为了确保当前闭包的生存期类型为 `'static`。

编译器将通过闭包对其捕获的变量的处置方式来确定此闭包类型将实现的[闭包trait][closure traits]。 如果所有捕获的类型都实现了 `Send` 和/或 `Sync`，那么此闭包类型也实现了 `Send` 和/或 `Sync`。 这些存在这些 trait，函数可以通过泛型的方式接受各种闭包，即便闭包的类型名无法被确切指定。

闭包 trait 的实现

当前闭包类型实现哪一个闭包trait 依赖于该闭包如何捕获变量和这些变量的类型。 了解闭包如何和何时实现 `Fn`、`FnMut` 和 `FnOnce` 这三类 trait，请参看调用trait 和自动强转那一章。 如果所有捕获的类型都实现了 `Send` 和/或 `Sync`，那么此闭包类型也实现了 `Send` 和/或 `Sync`。

Example

示例

在下面例子中，我们定义了一个名为 `ten_times` 的函数，它接受高阶函数参数，然后我们传给它一个闭包表达式作为实参并调用它。 之后又定义了一个使用移动语义从环境中捕获变量的闭包表达式来供该函数调用。

```
fn ten_times<F>(f: F) where F: Fn(i32) {
    for index in 0..10 {
        f(index);
    }
}

ten_times(|j| println!("hello, {}", j));
// 带类型标注 i32
ten_times(|j: i32| -> () { println!("hello, {}", j) });

let word = "konnichiwa".to_owned();
ten_times(move |j| println!("{}, {}", word, j));
```

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。

循环

loop-expr.md
commit: d23f9da8469617e6c81121d9fd123443df70595d
本章译文最后维护日期：2021-5-6

句法

LoopExpression :
LoopLabel? (
InfiniteLoopExpression
| *PredicateLoopExpression*
| *PredicatePatternLoopExpression*
| *IteratorLoopExpression*
)

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模式。
- `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` 循环支持对循环体非平凡求值(evaluation to non-trivial values)[1]。

无限循环

句法

InfiniteLoopExpression :
`loop` *BlockExpression*

`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。

谓词循环

句法

PredicateLoopExpression :
`while` *Expression*_{*排除结构体表达式*} *BlockExpression*

`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。

举个例子：

```
let mut i = 0;

while i < 10 {
    println!("hello");
    i = i + 1;
}
```

谓词模式循环

句法

PredicatePatternLoopExpression :
`while` `let` *Pattern* `=` *Expression*_{*排除结构体表达式和惰性布尔运算符表达式*} *BlockExpression*

`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模式、一个 `=`、一个检验对象(scrutinee)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模式匹配，则执行循环体块，然后控制流再返回到模式匹配语句。如果不匹配，则 `while` 表达式执行完成。

```
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;
}
```

`while let`循环等价于包含匹配(`match`)表达式的`loop`表达式。 如下：

```
'label: while let PATS = EXPR {
    /* loop body */
}
```

等价于

```
'label: loop {
    match EXPR {
        PATS => { /* loop body */ },
        _ => break,
    }
}
```

可以使用操作符`|`指定多个模式。 这与匹配(`match`)表达式中的`|`具有相同的语义：

```
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);
}
```

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。

迭代器循环

句法

IteratorLoopExpression :
`for` *Pattern* `in` *Expression*排除结构体表达式 *BlockExpression*

`for`表达式是一个用于在`std::iter::IntoIterator`的某个迭代器实现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模式进行匹配，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`循环遍历数组内容的示例：

```
let v = &["apples", "cake", "coffee"];

for text in v {
    println!("I like {}.", text);
}
```

`for`循环遍历一个整数序列的例子：

```
let mut sum = 0;
for n in 1..11 {
    sum += n;
}
assert_eq!(sum, 55);
```

`for`循环等价于后面的块表达式。

```
'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
}
```

这里的 `IntoIterator`、`Iterator` 和 `Option` 是标准库的程序项(standard library item)，不是当前作用域中解析的的任何名称。 变量名 `next`、`iter` 和 `val` 也仅用于表述需要，实际上它们不是用户可以输入的名称。

注意：上面代码里使用外层 `matche` 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。

循环标签

句法

LoopLabel :
LIFETIME_OR_LABEL `:`

一个循环表达式可以选择设置一个*标签*。 这类标签被标记为循环表达式之前的生存期（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。

`break`表达式

句法

BreakExpression :

`break` LIFETIME_OR_LABEL? *Expression*?

当遇到 `break` 时，相关的循环体的执行将立即结束，例如：

```
let mut last = 0;
for x in 1..100 {
    if x > 12 {
        break;
    }
    last = x;
}
assert_eq!(last, 12);
```

`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）。例如：

```
'outer: loop {
    while true {
        break 'outer;
    }
}
```

`break`表达式只允许在循环体内使用，它有 `break`、`break 'label` 或（参见后面） `break EXPR` 或 `break 'label EXPR` 这四种形式。

`continue`表达式

句法

ContinueExpression :

`continue` LIFETIME_OR_LABEL?

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。

与`break`一样，`continue`通常与最内层的循环相关联，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。

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。例如：

```
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);
```

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。

[1] 求得`()`类型以外的值。

区间表达式

range-expr.md
commit: eb5290329316e96c48c032075f7dbfa56990702b
本章译文最后维护日期：2021-02-21

句法

RangeExpression :
    RangeExpr
  | *RangeFromExpr*
  | *RangeToExpr*
  | *RangeFullExpr*
  | *RangeInclusiveExpr*
  | *RangeToInclusiveExpr*

RangeExpr :
  Expression `..` *Expression*

RangeFromExpr :
  Expression `..`

RangeToExpr :
  `..` *Expression*

RangeFullExpr :
  `..`

RangeInclusiveExpr :
  Expression `..=` *Expression*

RangeToInclusiveExpr :
  `..=` *Expression*

`..` 和 `..=` 操作符会根据下表中的规则构造 `std::ops::Range`（或 `core::ops::Range`）的某一变体类型的对象：

区间语

产生式/句法规则	句法	类型	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

举例：

```
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
```

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);
```

区间能在 `for` 循环里使用：

```
for i in 1..11 {
    println!("{}", i);
}
```

`if`和 `if let`表达式

if-expr.md
commit: d23f9da8469617e6c81121d9fd123443df70595d
本章译文最后维护日期：2021-5-6

`if`表达式

句法

IfExpression :
`if` *Expression*^{*排除结构体表达式*} *BlockExpression*

(`else` (*BlockExpression* | *IfExpression* | *IfLetExpression*))?

`if`表达式是程序控制中的一个条件分支。`if`表达式的句法是一个条件操作数（operand）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。

```
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");
```

`if let`表达式

句法

IfLetExpression :
`if` `let` *Pattern* `=` *Expression*排除结构体表达式和惰性布尔运算符表达式 *BlockExpression*
(`else` (*BlockExpression* | *IfExpression* | *IfLetExpression*))?

`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`，再后面是一个模式、一个`=`和一个检验对象(scrutinee)操作数。 如果检验对象操作数的值与模式匹配，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。

```
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");
}
```

`if`表达式和 `if let`表达式能混合使用:

```
let x = Some(3);
let a = if let Some(1) = x {
    1
} else if x == Some(2) {
    2
} else if let Some(y) = x {
    y
} else {
    -1
};
assert_eq!(a, 3);
```

`if let`表达式等价于match表达式，例如:

```
if let PATS = EXPR {
    /* body */
} else {
    /*else */
}
```

is equivalent to

```
match EXPR {
    PATS => { /* body */ },
    _ => { /* else */ },    // 如果没有 else块，这相当于 `()`
}
```

可以使用操作符 `|` 指定多个模式。 这与匹配(`match`)表达式中的 `|` 具有相同的语义：

```
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);
}
```

`if let`表达式不能是惰性布尔运算符表达式。 使用惰性布尔运算符的效果是不明确的，因为Rust 里一个新特性（if-let执行链(if-let chains)的实现-请参阅eRFC 2947）正被提上日程。 当确实需要惰性布尔运算符表达式时，可以像下面一样使用圆括号来实现：

```
// Before...
if let PAT = EXPR && EXPR { .. }

// After...
if let PAT = ( EXPR && EXPR ) { .. }

// Before...
if let PAT = EXPR || EXPR { .. }

// After...
if let PAT = ( EXPR || EXPR ) { .. }
```

匹配(`match`)表达式

match-expr.md
commit: d23f9da8469617e6c81121d9fd123443df70595d
本章译文最后维护日期：2021-5-6

句法

MatchExpression :
`match` *Expression*排除结构体表达式 `{`
*InnerAttribute**
MatchArms?
`}`

MatchArms :
(*MatchArm* `=>` (*ExpressionWithoutBlock* `,` | *ExpressionWithBlock* `,`?))*
MatchArm `=>` *Expression* `,`?

MatchArm :
*OuterAttribute** *Pattern* *MatchArmGuard*?

MatchArmGuard :
`if` *Expression*

*匹配(`match`)表达式*在模式(pattern)上建立代码逻辑分支(branch)。 匹配的确切形式取决于其应用的模式。 一个匹配(`match`)表达式带有一个要与模式进行比较的 *检验对象(scrutinee)表达式*。 检验对象表达式和模式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)中的模式进行比较，直到找到一个成功的匹配。 第一个匹配成功的模式所在的匹配臂会被选中为当前匹配(`match`)的分支目标，然后以该模式绑定的变量为中介，（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，此匹配(`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。

匹配(`match`)表达式的一个示例：

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

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。

可以使用操作符 `|` 连接多个匹配模式。 每个模式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。

```
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!(),
}
```

注意: `2..=9` 是一个区间(Range)模式，不是一个区间表达式。 因此，只有区间模式支持的区间类型才能在匹配臂中使用。

每个 `|` 分隔的模式里出现的变量绑定必须出现在匹配臂的所有模式里。 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模式。

匹配守卫

匹配臂可以接受*匹配守卫(Pattern guard)*来进一步改进匹配标准。 模式守卫出现在模式的后面，由关键字 `if` 后面的布尔类型表达式组成。

当模式匹配成功时，将执行匹配守卫表达式。 如果此表达式的计算结果为真，则此模式将进一步被确认为匹配成功。 否则，匹配将测试下一个模式，包括测试同一匹配臂中运算符 `|` 分割的后续匹配模式。

```
let message = match maybe_digit {
    Some(x) if x < 10 => process_digit(x),
    Some(x) => process_other(x),
    None => panic!(),
};
```

注意：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 } => {}
    _ => {}
}
assert_eq!(i.get(), 2);
```

匹配守卫可以引用绑定在它们前面的模式里的变量。 在对匹配守卫进行计算之前，将对检验对象内部被模式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真时，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。

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。

返回(`return`)表达式

return-expr.md
commit: eb5290329316e96c48c032075f7dbfa56990702b
本章译文最后维护日期：2021-02-21

句法

ReturnExpression :
`return` *Expression*?

返回(return)表达式使用关键字 `return` 来标识。 对返回(`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(activation frame)，并将控制权转移到此函数的调用帧(caller frame)。

一个返回(`return`)表达式的例子：

```
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        return a;
    }
    return b;
}
```

等待(await)表达式

await-expr.md
commit: 23672971a16c69ea894bef24992b74912cfe5d25
本章译文最后维护日期：2021-4-5

句法

AwaitExpression :
Expression `.` `await`

*等待(await)表达式*挂起当前计算，直到给定的 future 准备好生成值。 等待(await)表达式的句法格式为：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 `.` 标记，再后跟一个 `await` 关键字。

等待(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。

更具体地说，等待(await)表达式具有以下效果：

1. 把 future操作数求值计算到一个 future类型的 `tmp` 中；
2. 使用 `Pin::new_unchecked` 固定住(Pin)这个 `tmp` ；
3. 然后通过调用 `Future::poll` 方法对这个固定住的 future 进行轮询，同事将当前任务上下文传递给它；
4. 如果轮询(`poll`)调用返回 `Poll::Pending` ，那么这个 future 就也返回 `*Expression*

版本差异： 等待(await)表达式只能从 Rust 2018 版开始才可用

任务上下文

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(`Context`)。因为等待(`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。

近似脱糖

实际上,一个等待(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。

模式

patterns.md
commit: 100fa060ce1d2db7e4d1533efd289f9c18d08d53
本章译文最后维护日期：2021-5-6

句法

Pattern :
`|`? *PatternNoTopAlt* (`|` *PatternNoTopAlt*)*

PatternNoTopAlt :
PatternWithoutRange
| *RangePattern*

PatternWithoutRange :
LiteralPattern
| *IdentifierPattern*
| *WildcardPattern*
| *RestPattern*
| *ObsoleteRangePattern*
| *ReferencePattern*
| *StructPattern*
| *TupleStructPattern*
| *TuplePattern*
| *GroupedPattern*
| *SlicePattern*
| *PathPattern*
| *MacroInvocation*

模式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。模式也用在变量声明上和函数（包括闭包）的参数上。

下面示例中的模式完成四件事：

- 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。

```
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);
}
```

模式用于：

- `let` 声明
- 函数和闭包的参数。
- 匹配(`match`)表达式
- `if let` 表达式
- `while let` 表达式
- `for` 表达式

解构

模式可用于*解构*结构体(`struct`)、枚举(`enum`)和元组。解构将一个值分解成它的组件组成，使用的句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。

```
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

当一个模式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)*模式总是能与它们所匹配的值匹配成功。例如：

```
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);
}
```

字面量模式

句法

LiteralPattern :
BOOLEAN_LITERAL
| CHAR_LITERAL
| BYTE_LITERAL
| STRING_LITERAL
| RAW_STRING_LITERAL
| BYTE_STRING_LITERAL
| RAW_BYTE_STRING_LITERAL
| `-`? INTEGER_LITERAL

| `-`? FLOAT_LITERAL

*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，它的作用类似于否定运算符。

> ⚠️ 浮点字面量目前还可以使用，但是由于它们在数值比较时带来的复杂性，在将来的 Rust 版本中，它们将被禁止用于字面量模式(参见 issue #41620)。

字面量模式总是可以反驳型的。

示例：

```
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"),
    }
}
```

标识符模式

句法

IdentifierPattern :

`ref`? `mut`? IDENTIFIER (`@` *Pattern*)?

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模式的上下文（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`），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。

```
let mut variable = 10;
fn sum(x: i32, y: i32) -> i32 {
```

要将模式匹配到的值绑定到变量上，也可使用句法 `variable @ subpattern`。例如，下面示例中将值 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(range subpattern)）。

```
let x = 2;

match x {
    e @ 1 ..= 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

默认情况下，标识符模式里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。示例：

```
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。之所以需要这种句法，是因为在解构子模式(destructuring subpatterns)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 { }
```

这里，`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(Path pattern)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。

如果 `@` 子模式是不可反驳型的或未指定子模式，则标识符模式是不可反驳型的。

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，模式会自动选择不同的*绑定方式*。当引用值与非引用模式匹配时，这将自动地被视为 `ref` 或 `ref mut` 绑定方式。示例：

```
let x: &Option<i32> = &Some(3);
if let Some(y) = x {
    // y 被转换为`ref y` ，其类型为 &i32
}
```

*非引用模式(Non-reference patterns)*包括**除**上面这种绑定模式和后面会讲到的通配符模式（`_`）、匹配引用类型的常量(`const`)模式和引用模式这些模式以外的所有模式。

如果绑定模式(binding pattern)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。当匹配一个模式时，编译器对模式从外到内逐层匹配。每次非引用模式和引用匹配上了时，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。可变引用会将模式设置为 `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。[2]

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模式中混合使用，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(partial move)*的问题而报错。

示例：

```
// 在 `age` 被引用绑定的情况下，`name` 被从 person 中移出
let Person { name, ref age } = person;
```

通配符模式

句法

WildcardPattern :
`_`

通配符模式（下划线符号）能与任何值匹配。常用它来忽略那些无关紧要的值。在其他模式中使用该模式时，它匹配单个数据字段（与和匹配所有其余字段的 `..` 相对）。与标识符模式不同，它不会复制、移动或借用它匹配的值。

示例：

```
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 {}
```

通配符模式总是不可反驳型的。

剩余模式

句法

RestPattern :
`..`

剩余模式（`..` token）充当匹配长度可变的模式(variable-length 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，并且只能作为这些模式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。

示例：

```
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"),
}
```

区间模式

句法

RangePattern :
  RangePatternBound `..=` *RangePatternBound*

ObsoleteRangePattern :(译者注：废弃的区间模式句法/产生式) \ *RangePatternBound* `...` *RangePatternBound*

RangePatternBound :
CHAR_LITERAL
| BYTE_LITERAL
| `-`? INTEGER_LITERAL
| `-`? FLOAT_LITERAL
| *PathInExpression*
| *QualifiedPathInExpression*

区间模式匹配在其上下边界定义的封闭区间内的值。例如，一个模式 `'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。

一个模式 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 `...` 句法只是为了向后兼容。

区间模式只适用于标量类型(scalar type)。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（参见 issue #41620）。

示例：

```
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",
});
```

当区间模式匹配某（非usize 和 非isize）整型类型和字符型(`char`)的整个值域时，此模式是不可反驳型的。例如， `0u8..=255u8` 是不可反驳型的。某类整型的值区间是从该类型的最小值到该类型最大值的闭区间。字符型(`char`)的值的区间就是那些包含所有 Unicode 标量值的区间，即 `'\u{0000}'..='\u{D7FF}'` 和 `'\u{E000}'..='\u{10FFFF}'` 。

引用模式

句法

ReferencePattern :
`(` `&` | `&&` `)` `mut`? *PatternWithoutRange*

引用模式对当前匹配的指针做解引用，从而能借用它们：

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);
```

引用模式的文法产生式(grammar production)要求必须使用 token `&&` 来匹配对引用的引用，因为 `&&` 本身是一个单独的 token，而不是两个 `&` token。

译者注：举例

```
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。

结构体模式

句法

StructPattern :
PathInExpression `{`
StructPatternElements?

`}`

StructPatternElements :

StructPatternFields (`,` | `,` *StructPatternEtCetera*)?
| *StructPatternEtCetera*

StructPatternFields :

StructPatternField (`,` *StructPatternField*) *

StructPatternField :

OuterAttribute *
(
TUPLE_INDEX `:` *Pattern*
| IDENTIFIER `:` *Pattern*
| `ref`? `mut`? IDENTIFIER
)

StructPatternEtCetera :

OuterAttribute *
`..`

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* 这样的句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。

```
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。

元组结构体模式

句法

TupleStructPattern :

PathInExpression `(` *TupleStructItems*? `)`

TupleStructItems :

Pattern (`,` *Pattern*)* `,`?

元组结构体模式匹配元组结构体值和枚举值，这些值将与该模式的子模式定义的所有条件进行匹配。它还被用于解构元组结构体值或枚举值。

当元组结构体模式的一个子模式是可反驳型的，则该元组结构体模式就是可反驳型的。

元组模式

句法

TuplePattern :

`(` *TuplePatternItems*? `)`

TuplePatternItems :
Pattern `,`
| *RestPattern*
| *Pattern* (`,` *Pattern*)+ `,`?

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(*RestPattern*)的元组句法形式 `(..)` 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。

使用元组模式的示例：

```
let pair = (10, "ten");
let (a, b) = pair;

assert_eq!(a, 10);
assert_eq!(b, "ten");
```

分组模式

句法

GroupedPattern :
`(` *Pattern* `)`

将模式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) => (),
    _ => (),
}
```

切片模式

句法

SlicePattern :

`[` *SlicePatternItems*? `]`

SlicePatternItems :

Pattern (`,` *Pattern*)* `,`?

切片模式可以匹配固定长度的数组和动态长度的切片。

```
// 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",
};
```

```
// 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 */ }
};
```

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。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。

路径模式

句法

PathPattern :

PathInExpression

| *QualifiedPathInExpression*

*路径模式*是指向(refer to)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模式。

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。

常量不能是联合体类型。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。

or 模式

_or模式_是能匹配两个或多个并列子模式（例如：`A | B | C`）中的一个的模式。此模式可以任意嵌套。除了 `let` 绑定和函数参数（包括闭包参数）中的模式（此时句法上使用 _PatternNoTopAlt_产生式），or模式在句法上允许在任何其他模式出现的地方出现（这些模式句法上使用 _Pattern_产生式）。

静态语义

1. 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。

2. 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。

3. 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。

注意这里的*“构造器”*这个用词，我们并没有特定提到它是元组结构模式，因为它本意是指任何能够生成类型的模式。这包括枚举变量、元组结构、具有命名字段的结构、数组、元组和切片。

动态语义

1. 检查对象表达式(scrutinee expression) `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。

无分解符模式的优先级

如本章其他部分所示，有几种类型的模式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 or模式。它们组合在一起时，or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` 不是在所有模式中都存在绑定关系； `&A(x) | B(x)` 将导致不同子模式中的 `x` 之的类型不匹配。

[1] 请仔细参研匹配表达式中的 MatchExpression产生式，搞清楚匹配臂(MatchArm)的位置。

[2] 文字叙述有些晦涩，译者举个例子：假如 `if let &Some(y) = &&&Some(3) {`，此时会首先剥掉等号两边的第一层 `&` 号，然后是 `Some(y)` 和 `&&Some(3)` 匹配，此时发现是非引用模式和引用匹配上了，就再对 `&&Some(3)` 做重复解引用，解出 `Some(3)`，然后从外部转向内部，见到最后的变量 `y` 和检验对象 `3`，就更新 `y` 的默认绑定方式为 `ref`，所以 `y` 就匹配为 `&3`；如果我们这个例子的变量 `y` 改为 `ref y`，不影响 `y` 的绑定效果；极端的情况 `if let &Some(y) = &&&Some(x) {`，如果 `x` 是可变的，那么此时 `y` 的绑定方式就是 `ref mut`，再进一步极端 `if let &Some(ref y) = &&&Some(x) {`，此时 `y` 的绑定方式仍是 `ref`。

类型系统

类型

types.md
commit: af1cf6d3ca3b7a8c434c142148742aa912e37c34
本章译文最后维护日期：2020-11-14

Rust 程序中的每个变量、程序项和值都有一个类型。*值*的*类型*定义了该如何解释用于保存它的内存数据，以及定义了可以对该值执行的操作。

内置的类型以非平凡的方式(in nontrivial ways)紧密地集成到语言中，这种方式是不可能在用户定义的类型中模拟的。用户定义的类型功能有限。

内置类型列表：

- 原生类型(primitive types):
  - 布尔型(Boolean) — `true` 或 `false`
  - 数字类(Numeric) — 整型(integer) 和 浮点型(float)
  - 文本类(Textual) — 字符型(`char`) 和 字符串切片(`str`)
  - never类型 — `!` — 没有值的类型
- 序列类型(sequence types)：
  - 元组(Tuple)
  - 数组(Array)
  - 切片(Slice)
- 用户自定义类型(user-defined types)：
  - 结构体(Struct)
  - 枚举(Enum)
  - 联合体(Union)
- 函数类型(function types)：
  - 函数(Functions)
  - 闭包(Closures)
- 指针类型(pointer types)：
  - 引用(References)
  - 裸指针(Raw pointers)
  - 函数指针(Function pointers)
- trait类型(Trait types):
  - trait对象(Trait objects)
  - 实现trait(Impl trait)

类型表达式

句法

Type :
    TypeNoBounds
  | *ImplTraitType*
  | *TraitObjectType*

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

上表中的 *Type* 文法规则中定义的各种*类型表达式*都是某个具体类型的句法产生式。它们可以覆盖指代：

- 序列类型（tuple, array, slice）。
- 类型路径(type paths)，这些包括：
  - 原生类型（布尔型, 数字类类型, 文本类类型）。
  - 指向程序项（结构体(`struct`), 枚举(`enum`), 联合体(`union`), 类型别名, trait）的路径。
  - `Self`路径，其中 `Self` 是实现类型。
  - 泛型类型参数。
- 指针类型（引用, 裸指针, 函数指针）。
- 自动推断型类型(inferred type)，就是请求编译器确定类型的类型。
- 用来消除歧义的圆括号。
- Trait类型：trait对象(trait object) 和 实现trait(impl trait).

- never型(`!`)。
- 展开成类型表达式的宏。

圆括号组合类型

ParenthesizedType :
`(` *Type* `)`

在某些情况下，类型组合在一起时可能会产生二义性。此时可以在类型周围使用元括号来避免歧义。例如，引用类型的类型约束列表中的 `+` 运算符搞不清楚其左值的边界位置在哪里，因此需要使用圆括号来明确其边界。这里需要的消歧文法就是使用 *TypeNoBounds* 句法规则替代 *Type* 句法规则。

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

递归类型

标称类型(nominal types) — 结构体(`struct`)、枚举(`enum`)和联合体(`union`) — 可以是递归的。也就是说，每个枚举(`enum`)变体或结构体(`struct`)或联合体(`union`)的字段可以直接或间接地指向此枚举(`enum`)或结构体(`struct`)类型本身。这种递归有一些限制：

- 递归类型必须在递归中包含一个标称类型（不能仅是类型别名或其他结构化的类型，如数组或元组）。因此不允许使用 `type Rec = &'static [Rec]`。
- 递归类型的尺寸必须是有限的；也就是说，类型的递归字段必须是指针类型。
- 递归类型的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）。

*递归*类型及使用示例：

```
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))));
```

布尔型

boolean.md
commit: 1804726424c5cbc97f3d9d4adf6236980e2ff7a1
本章译文最后维护日期：2021-2-10

```
let b: bool = true;
```

*布尔型*或*布尔数*是一种可以为*真(`true`)*或假(`false`)*的原语数据类型。

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` 来表达对应名称的值。

该类型是此语言的预导入包的一部分，使用名称 `bool` 来表示。

布尔型的对象尺寸和对齐量均为1。false 的位模式为 `0x00`， true 的位模式为 `0x01`。其他的任何其他位模式的布尔型的象都是未定义的行为。

布尔型是多种表达式的操作数的类型:

- if表达式和 while表达式中的条件操作数
- 惰性布尔运算表达式的操作数

注意：布尔型的行为类似于枚举类型，但它确实不是枚举类型。在实践中，这主要意味着构造函数不与类型相关联（例如没有 `bool::true` 这种写法）。

和其他所有的原语类型一样，布尔型实现了 `Clone`、`Copy`、`Sized`、`Send` 和 `Sync` 这些 traits。

注意: 参见标准库文档中的相关操作运算。

布尔运算

当使用带有布尔型的操作数的特定操作符表达式时，它们使用[布尔逻辑规则][boolean logic]进行计算。

逻辑非

b	!b
true	false
false	true

逻辑或

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

逻辑与

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

逻辑异或

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

false	false	false

比较

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

- `a != b` 等同于 `!(a == b)`
- `a >= b` 等同于 `a == b | a > b`
- `a < b` 等同于 `!(a >= b)`
- `a <= b` 等同于 `a == b | a < b`

数字型

mumeric.md
commit: 73ca198fb3ab52283d67d5fe28c541ee1d169f48
本章译文最后维护日期：2020-11-14

整型/整数类型

无符号整数类型：

类型	最小值	最大值
`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型包括：

类型	最小值	最大值
`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

浮点型

Rust 对应 IEEE 754-2008 的“binary32”和“binary64”浮点类型分别是 `f32` 和 `f64`。

和计算平台相关的整型

`usize`类型是一种无符号整型，其宽度与平台的指针类型的宽度相同。它可以表示进程中的每个内存地址。

`isize`类型是一种有符号整型，其宽度与平台的指针类型的宽度相同。（Rust 规定）对象的尺寸和数组的长度的理论上限是 `isize` 的最大值。这确保了 `isize` 可以用来计算指向对象或数组内部的指针之间的差异，并且可以寻址对象中的每个字节以及末尾之后的那个字节。

`usize` 和 `isize` 的宽度至少是 16-bits。

> **注意**：许多 Rust 代码可能会假设指针、`usize` 和 `isize` 是 32-bit 或 64-bit 的。因此，16-bit 指针的支持是有限的，这部分支持可能需要来自库的明确关注和确认。

文本类类型

textual.md
commit: 9af5071f876111a09ba54a86655679de83eb464c
本章译文最后维护日期：2020-11-14

类型 `char` 和 `str` 用于保存文本数据。

字符型(`char`)的值是 Unicode 标量(scalar)值（即不是代理项(surrogate)的代码点），可以表示为 0x0000~0xD7FF 或 0xE000~0x10FFFF 范围内的 32-bit 无符号字符。创建超出此范围的字符直接触发未定义行为(Undefined Behavior)。一个 `[char]` 实际上是长度为1的 UCS-4 / UTF-32 字符串。

`str`类型的值的表示方法与 `[u8]` 相同，它是一个 8-bit 无符号字节类型的切片。但是，Rust 标准库对 `str` 做了额外的假定： `str` 上的方法会假定并确保其中的数据是有效的 UTF-8。调用 `str` 的方法来处理非UTF-8 缓冲区上的数据可能或早或晚地出现未定义行为。

由于 `str` 是一个动态尺寸类型，所以它只能通过指针类型实例化，比如 `&str`。

never 类型

never.md
commit: 91486df597a9e8060f9c67587359e9f168dea7ef
本章译文最后维护日期：2020-11-14

句法

NeverType : `!`

never类型(`!`)是一个没有值的类型，表示永远不会完成计算的结果。`!` 的类型表达式可以强转为任何其他类型。

```
let x: ! = panic!();
// 可以强转为任何类型
let y: u32 = x;
```

注意： never类型原本预计在1.41中稳定下来，但由于最后一分钟检测到一些意想不到的回归，该类型的稳定进程临时暂停。目前 `!` 类型只能出现在函数返回类型中。有关详细信息，请参阅议题跟踪。

元组类型

tuple.md
commit: ff6c3dc120ce6d06548b9045c43103f58720ee62
本章译文最后维护日期：2021-4-6

句法

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

*元组类型*是由其他类型的异构列表组合成的一类结构化类型[1]。

元组类型的语法规则为一对圆括号封闭的逗号分割的类型列表。 为和圆括号类型区分开来，一元元组的元素类型后面需要有一个逗号。

元组类型的字段数量等同于其封闭的异构类型列表的长度。 字段的数量决定元组的*元数(arity)*。 有 `n` 个字段的元组叫做 *n元元组(n-ary tuple)*。 例如，有两个字段的元组就是二元元组。

元组的字段用它在列表中的位置数字来索引。 第一个字段索引为 `0`。 第二个字段索引为 `1`。 然后以此类推。 每个字段的类型都是元组类型列表中相同位置的类型。

出于方便和历史原因，不带元素(`()`)的元组类型通常被称为*单元(unit)*或*单元类型(unit type)*。 它的值也被称为*单元*或*单元值*。

元组类型的示例：

- `()` (单元)
- `(f64, f64)`
- `(String, i32)`
- `(i32, String)` (跟前一个示例类型不一样)
- `(i32, f64, Vec<String>, Option<bool>)`

这种类型的值是使用元组表达式来构造的。 此外，如果没有其他有意义的值可供求得/返回，很多种表达式都将生成单元值。 元组字段可以通过元组索引表达式或模式匹配来访问。

[1] 结构化类型的特点就是其内部对等位置的类型如果是相等的，那么这些结构化类型就是相等的。有关元组的标称类型版本，请参见元组结构体。

数组类型

array.md
commit: 2f459e22ec30a94bafafe417da4e95044578df73
本章译文最后维护日期：2020-11-14

句法

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

数组是 `N` 个类型为 `T` 的元素组成的固定长度(fixed-size)的序列，数组类型写为 `[T; N]`。长度(size)是一个计算结果为 `usize` 的常量表达式。

示例：

```
// 一个栈分配的数组
let array: [i32; 3] = [1, 2, 3];

// 一个堆分配的数组，被自动强转成切片
let boxed_array: Box<[i32]> = Box::new([1, 2, 3]);
```

数组的所有元素总是初始化过的，使用 Rust 中的安全(safe)方法或操作符来访问数组时总是会先做越界检查。

注意：标准库类型 `Vec<T>` 提供了堆分配方案的可调整大小的数组类型。

切片类型

slice.md
commit: b0e0ad6490d6517c19546b1023948986578fc378
本章译文最后维护日期：2020-11-14

句法

SliceType :
`[` *Type* `]`

切片是一种动态尺寸类型(dynamically sized type)，它代表类型为 `T` 的元素组成的数据序列的一个“视图(view)”。切片类型写为 `[T]`。

要使用切片类型，通常必须放在指针后面使用，例如：

- `&[T]`，共享切片('shared slice')，常被直接称为切片(`slice`)，它不拥有它指向的数据，只是借用。
- `&mut [T]`，可变切片('mutable slice')，可变借用它指向的数据。
- `Box<[T]>`, boxed切片('boxed slice')。

示例：

```
// 一个堆分配的数组，被自动强转成切片
let boxed_array: Box<[i32]> = Box::new([1, 2, 3]);

// 数组上的（共享）切片
let slice: &[i32] = &boxed_array[..];
```

切片的所有元素总是初始化过的，使用 Rust 中的安全(safe)方法或操作符来访问切片时总是会做越界检查。

结构体类型

struct.md
commit: 8a98437835db5f3cf57044aedcd1e00bd0d889f9
本章译文最后维护日期：2021-1-17

结构体(`struct`)*类型*是其由他类型异构产生的类型，这些其他类型被称为结构体类型的字段。[1]

结构体(`struct`)的实例可以用结构体表达式来构造。

默认情况下，结构体(`struct`)的内存布局是未定义的（默认允许进行一些编译器优化，比如字段重排），但也可以使用`repr`属性来使其布局在定义时就固定下来。在这两种情况下，字段在相应的结构体*表达式*中都可以以任何顺序给出；（但在相同的编译目标中，在）同一布局规则下生成的结构体(`struct`)值将始终具有相同的内存布局。

结构体(`struct`)的字段可以由可见性修饰符(visibility modifiers)限定，以允许从模块之外来访问结构体中的数据。

*元组结构体(uple struct)*类型与结构体类型类似，只是字段是匿名的。

*单元结构体(unit-like struct)*类型类似于结构体类型，只是它没有字段。由初始结构体表达式构造的值是驻留在此类类型中惟一的值。

[1] `struct`类型类似于 C 中的 `struct` 类型、ML家族的 *record* 类型或 Lisp 家族的 *struct* 类型。

枚举类型

enum.md
commit: d8cbe4eedb77bae3db9eff87b1238e7e23f6ae92
本章译文最后维护日期：2021-2-21

*枚举类型*是一种标称型(nominal)的、异构的、不相交的类型联合起来组成的类型，它直接用枚举(`enum`)程序项的名称来表示。[1]

枚举(`enum`)程序项同时声明了类型和它的各种*变体(variants)*，其中每个变体都独立命名，可使用定义结构体、元组结构体或单元结构体(unit-like struct)的句法来定义它们。

枚举(`enum`)的实例可以通过结构体体表达式来构造。

任何枚举值消耗的内存和其同类型的其他变体都是相同的，具体都为其枚举(`enum`)类型的最大变体所需的内存再加上存储其判别值(discriminant)所需的内存。

枚举类型不能在*结构上*表示为类型，必须通过对枚举程序项的具名引用(named reference)来表示。[2]

[1] `enum`类型类似于 ML 中的数据(`data`)构造函数声明，或 Limbo 中的 *pick ADT*。

[2] 译者理解这句话的意思是：枚举不同于普通结构化的类型，所有的枚举类型都是对枚举程序项的引用；这里引用分两种，一种是类C枚举，就是对程序项的直接具名引用；另一种是带字段的枚举变体，这种其实是类似于 `Box`、`Rc` 这样的具名引用，它通过封装其他类型来指导数据的存储和限定其上可用的操作。

联合体类型

union.md
commit: cd996a4e6e9b98929c3718fd76988bfee51d757c
本章译文最后维护日期：2020-12-24

*联合体类型*是一种标称型(nominal)的、异构的、类似C语言里的 union 的类型，具体的类型名称由联合体(`union`)程序项的名称表示。

Since transmutes can cause unexpected or undefined behaviour, `unsafe` is required to read from a union field, or to write to a field that doesn't implement `Copy` or has a [`ManuallyDrop`] type. See the item documentation for further details. 联合体没有“活跃字段(active field)”的概念。相反，每次对联合体的访问都将联合体的部分存储内容转换为被访问字段的类型。由于转换可能会导致意外或未定义行为，所以读取联合体字段，或写入未实现 `Copy` 或 [`ManuallyDrop`] 的联合体字段的操作都需要放在 `unsafe` 块内进行。有关详细信息，请参阅相应的程序项文档。

默认情况下，联合体(`union`)的内存布局是未定义的，但是可以使用 `#[repr(...)]` 属性来固定为某一类型布局。

函数项类型

function-item.md
commit: b0e0ad6490d6517c19546b1023948986578fc378
本章译文最后维护日期：2020-11-14

当被引用函数项、元组结构体的构造函数或枚举变体的构造函数时，会产生它们的*函数项类型(function item type)*的零尺寸的值。这种类型（也就是此值）显式地标识了该函数——标识出的内容包括程序项定义时的名字、类型参数，及其定义时的生存期参数（不是后期绑定的生存期参数，后期绑定的生存期参数只在函数被调用时才被赋与）——所以该值不需要包含一个实际的函数指针，当此函数被调用时也不需要一个间接的寻址操作去查找此函数。

没有直接引用函数项类型的句法，但是编译器会在错误消息中会显示类似于 `fn(u32) -> i32 {fn_name}` 这样的“类型”。

因为函数项类型显式地标识了其函数，所以如果它们所指的程序项不同（包括不同的程序项，或相同的程序项但泛型参数的实际类型不同），则此函数项类型也不同，混合使用它们将导致类型错误：

```
fn foo<T>() { }
let x = &mut foo::<i32>;
*x = foo::<u32>; //~ 错误：类型不匹配
```

但确实存在从函数项类型到具有相同签名的函数指针的自动强转(coercion)，这种自动强转一般发生在使用函数项类型却直接预期函数指针时；另一种发生情况是 `if` 表达式或匹配(`match`)表达式的不同分支返回使用了具有相同签名却使用了不同函数项类型的情况：

```
// 这里 `foo_ptr_1` 标注使用了 `fn()` 这个函数指针类型。
let foo_ptr_1: fn() = foo::<i32>;

// ... `foo_ptr_2` 也行 - 这次是通过类型检查(type-checks)做到的。
let foo_ptr_2 = if want_i32 {
    foo::<i32>
} else {
    foo::<u32>
};
```

所有的函数项类型都实现了 `Fn` 、 `FnMut` 、 `FnOnce` 、 `Copy` 、 `Clone` 、 `Send` 和 `Sync` 。

闭包类型

closure.md
commit: 5642af891714145cb2a765f244fff7d6b618a4c7
本章译文最后维护日期：2020-11-14

闭包表达式生成的闭包值具有唯一性和无法写出的匿名性。闭包类型近似相当于包含捕获变量的结构体。比如以下闭包示例：

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

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

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

生成大致如下所示的闭包类型：

```
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
    }
}
```

所以调用 `f` 相当于：

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

捕获方式

编译器倾向于优先通过不可变借用(immutable borrow)来捕获闭合变量(closed-over variable)，其次是通过唯一不可变借用(unique immutable borrow)（见下文），再其次可变借用(mutable borrow)，最后使用移动语义(move)来捕获。编译器将选择这些中的第一个能让此闭包编译通过的选项。这个选择只与闭包表达式的内容有关；编译器不考虑闭包表达式之外的代码，比如所涉及的变量的生存期。

如果使用了关键字 `move` ，那么所有捕获都是通过移动(move)语义进行的（当然对于 `Copy` 类型，则是通过拷贝语义进行的），而不管借用是否可用。关键字 `move` 通常用于允许闭包比其捕获的值活得更久，例如返回闭包或用于生成新线程。

复合类型（如结构体、元组和枚举）始终是全部捕获的，而不是各个字段分开捕获的。如果真要捕获单个字段，那可能需要先借用该字段到本地局部变量中：

```
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);
        })
    }
}
```

相反，如果闭包直接使用了 `self.vec`，那么它将尝试通过可变引用捕获 `self`。但是因为 `self.set` 已经被借出用来迭代了，所以代码将无法编译。

捕获中的唯一不可变借用

捕获方式中有一种被称为*唯一不可变借用*的特殊类型的借用捕获，这种借用不能在语言的其他任何地方使用，也不能显式地写出。唯一不可变借用发生在修改可变引用的引用对象(referent)时，如下面的示例所示：

```
let mut b = false;
let x = &mut b;
{
    let mut c = || { *x = true; };
    // 下行代码不正确
    // let y = &x;
    c();
}
let z = &x;
```

在这种情况下，不能去可变借用 `x`，因为 `x` 没有标注 `mut`。但与此同时，如果不可变借用 `x`，那对其赋值又会非法，因为 `& &mut` 引用可能不是唯一的，因此此引用不能安全地用于修改值。所以这里闭包使用了唯一不可变借用：它采用了不可变的方式借用了 `x`，但是又像可变借用一样，当然前提是此借用必须是唯一的。在上面的例子中，解开 `y` 那行上的注释将产生错误，因为这将违反闭包对 `x` 的借用的唯一性；`z` 的声明是有效的，因为闭包的生存期在块结束时已过期，从而已经释放了对 `x` 的借用。

调用trait 和自动强转

闭包类型都实现了 `FnOnce`，这表明它们可以通过消耗掉闭包的所有权来调用执行它一次。此外，一些闭包实现了更具体的调用trait：

- 对没采用移动语义来捕获任何变量（译者注：变量的原始所有者仍拥有该变量的所有权）的闭包都实现了 `FnMut`，这表明该闭包可以通过可变引用来调用。
- 对捕获的变量没移出其值，也没去修改其值的闭包都实现了 `Fn`，这表明该闭包可以通过共享引用来调用。

> 注意：`move` 闭包可能仍然实现 `Fn` 或 `FnMut`，即使它们通过移动(move)语义来捕获变量。这是因为闭包类型实现什么样的 trait 是由闭包对捕获的变量值做了什么来决定的，而不是由闭包如何捕获它们来决定的。[1]

---

*非捕获闭包(Non-capturing closures)*是指不捕获环境中的任何变量的闭包。它们可以通过匹配签名的方式被自动强转成函数指针（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);
```

其他 trait

所有闭包类型都实现了 `Sized`。此外，闭包捕获的变量的类型如果实现了如下 trait，此闭包类型也会自动实现这些 trait：

- `Clone`
- `Copy`
- `Sync`
- `Send`

闭包类型实现 `Send` 和 `Sync` 的规则与普通结构体类型实现这俩 trait 的规则一样，而 `Clone` 和 `Copy` 就像它们在 `derived` 属性中表现的一样。对于 `Clone`，捕获变量的克隆顺序目前还没正式的规范出来。

由于捕获通常是通过引用进行的，因此会出现以下一般规则：

- 如果所有的捕获变量都实现了 `Sync`，则此闭包就也实现了 `Sync`。
- 如果所有非唯一不可变引用捕获的变量都实现了 `Sync`，并且所有由唯一不可变、可变引用、复制或移动语义捕获的值都实现了 `Send`，则此闭包就也实现了 `Send`。
- 如果一个闭包没有通过唯一不可变引用或可变引用捕获任何值，并且它通过复制或移动语义捕获的所有值都分别实现了 `Clone` 或 `Copy`，则此闭包就也实现了 `Clone` 或 `Copy`。

[1] 译者的实验代码：

```
fn f1<F : Fn() -> i32> (g: F) { println!("{}", g());}
fn f2<F : FnMut() -> i32> (mut g: F) { println!("{}", g());}
let t = 8;
f1(move || { t });
f2(move || { t });
```

指针类型

pointer.md
commit: 4664361ba2f7bcc568f6bef4d119b53971fdf8ad
本章译文最后维护日期：2021-4-6

Rust 中所有的指针都是显式的头等(first-class)值。 它们可以被移动或复制，存储到其他数据结构中，或从函数中返回。

引用(`&` 和 `&mut`)

句法

ReferenceType :
`&` *Lifetime*? `mut`? *TypeNoBounds*

共享引用(`&`)

共享引用(`&`)指向*由其他值拥有的*内存。 创建了对值的共享引用可以防止对该值的直接更改。 但在某些特定情况下，内部可变性又提供了这种情况的一种例外。 顾名思义，对一个值的共享引用的次数没有限制。共享引用类型被写为 `&type`；当需要指定显式的生存期时可写为 `&'a type`。 拷贝一个引用是一个“浅拷贝(shallow)”操作：它只涉及复制指针本身，也就是指针实现了 `Copy` trait 的意义所在。 释放引用对共享引用所指向的值没有影响，但是对临时值的引用的存在将使此临时值在此引用的作用域内保持存活状态。

可变引用(`&mut`)

可变引用(`&mut`)也指向其他值所拥有的内存。 可变引用类型被写为 `&mut type` 或 `&'a mut type`。 可变引用（其还未被借出[1]）是访问它所指向的值的唯一方法，所以可变引用没有实现 `Copy` trait。

裸指针(*const 和 *mut)

句法

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

裸指针是没有安全性或可用性(liveness)保证的指针。 裸指针写为 `*const T` 或 `*mut T`，例如， `*const i32` 表示指向 32-bit 有符号整数的裸指针。 拷贝或销毁(dropping)裸指针对任何其他值的生命周期(lifecycle)都没有影响。对 裸指针的解引用是非安全(`unsafe`)操作，可以通过重新借用裸指针（ `&*` 或 `&mut *` ）将其转换为引用。 在 Rust 代码中通常不鼓励使用裸指针；它们的存在是为了提升与外部代码的互操作性，以及编写对性能要求很高的函数或很底层的函数。

在比较裸指针时，比较的是它们的地址，而不是它们指向的数据。 当比较裸指针和动态尺寸类型时，还会比较它们指针上的附加/元数据。

可以直接使用 `core::ptr::addr_of!` 创建 `*const` 类型的裸指针，通过 `core::ptr::addr_of_mut!` 创建 `*mut` 类型的裸指针。

智能指针

标准库包含了一些额外的“智能指针”类型，它们提供了在引用和裸指针这类低级指针之外的更多的功能。

[1] 译者理解这里是指 `&mut type` 如果被借出，就成了 `&&mut type`，这样就又成了不可变借用了。

函数指针类型

function-pointer.md
commit: 761ad774fcb300f2b506fed7b4dbe753cda88d80
本章译文最后维护日期：2021-1-17

句法

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** `...`

函数指针类型（使用关键字 `fn` 写出）指向那些在编译时不必知道函数标识符的函数。它们也可以由函数项类型或非捕获(non-capturing)闭包经过一次自动强转(coercion)来创建。

非安全(`unsafe`)限定符表示类型的值是一个非安全函数，而外部(`extern`)限定符表示它是一个外部函数。

可变参数只能通过使用 `"C"` 或 `"cdecl"` 的 ABI调用约定的 `extern` 函数类型来指定。

下面示例中 `Binop` 被定义为函数指针类型：

```
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);
```

函数指针参数上的属性

函数指针参数上的属性遵循与常规函数参数相同的规则和限制。

trait对象

trait-object.md
commit: fd10e7043934711ef96b4dd2009db3e4d0182a33
本章译文最后维护日期：2020-11-14

句法

TraitObjectType :
`dyn`? *TypeParamBounds*

TraitObjectTypeOneBound :
`dyn`? *TraitBound*

*trait对象[1]*是另一种类型的不透明值(opaque value)，它实现了一组 trait。[2] 这组 trait 是由一个对象安全的*基础trait(base trait)* 加上任意数量的自动trait(auto traits)组成。

trait对象实现了基础trait、它的自动trait 以及其基础trait 的任何超类trait(supertraits)。

trait对象被写为为可选的关键字 `dyn` 后跟一组 trait约束，这些 trait约束有如此限制：除了第一个 trait 外，其他所有 trait 都必须是自动trait；生存期不能超过一个；不允许选择退出约束(opt-out bounds)（例如 `?Sized`）。此外，trait 的路径可以用圆括号括起来。

例如，给定一个trait `Trait`，下面所有的形式都是 trait对象：

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

版本差异：在 2015 版里，如果 trait对象的第一个约束是以 `::` 开头的路径，那么 `dyn`

> 会被视为路径的一部分。可以把第一条路径放在圆括号中来绕过这个问题。因此，如果希望 trait对象具有 `::your_module::Trait` 路径，那么应该将其写为 `dyn (::your_module::Trait)`。
>
> 从2018版本开始，`dyn` 是一个真正的关键字了，不允许在路径中使用，（不存在二义性了，）因此括号就没必要了。
>
> 注意：为了清晰起见，建议总是在 trait对象上使用关键字 `dyn`，除非你的代码库支持用 Rust 1.26 或更低的版本来编译。

如果基础trait 互为别名，并且自动trait 相同，生存期约束也相同，则这两种 trait对象类型互为别名。例如，`dyn Trait + Send + UnwindSafe` 和 `dyn Trait + UnwindSafe + Send` 是等价的。

由于值的具体类型是不透明的，trait对象是动态尺寸类型。像所有的 DST 一样，trait对象常被用在某种类型的指针后面；例如 `&dyn SomeTrait` 或 `Box<dyn SomeTrait>`。每个指向 trait 对象的指针实例包括：

- 一个指向实现 `SomeTrait` 的（那个真实的不透明的）类型 `T` 的实例的指针
- 一个指向*虚拟方法表(virtual method table_)*的指针。虚拟方法表也通常被称为 *虚函数表(vtable)*，它包含了 `T` 实现的 `SomeTrait` 的所有方法，`T` 实现的 `SomeTrait` 的超类 trait 的每个方法，还有指向 `T` 的实现的指针（即函数指针）。

trait对象的目的是允许方法的“延迟绑定(late binding)”。在 trait对象上调用一个方法会导致运行时的虚拟分发(virtual dispatch)：也就是说，一个函数指针从 trait对象的虚函数表(vtable)中被加载进来，并被间接调用。每个虚函数表实体的实际实现可能因对象的不同而不同。

一个 trait对象的例子：

```
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>);
}
```

在本例中，trait `Printable` 作为 trait对象出现在 以 `print` 为类型签名的函数的参数中 和 `main` 中的类型转换表达式(cast expression)中。

trait对象的生存期约束

因为 trait对象可以包含引用，所以这些引用的生存期需要表示为 trait对象的一部分。这种生存期被写为 `Trait + 'a`。默认情况下，可以通过合理的选择来推断此生存期。

[1] 本书行文中，使用“trait对象”这个词时，没区分 trait对象的值和类型本身，但一般都指类型。

[2] 译者认为这样翻译可能更容易理解：*trait对象*是丢失了/隐藏了具体真实类型的 trait实现的类型，此类型本身一般会实现了一组 trait。

实现trait

impl-object.md
commit: b0e0ad6490d6517c19546b1023948986578fc378
本章译文最后维护日期：2020-11-14

句法

ImplTraitType : `impl` *TypeParamBounds*

ImplTraitTypeOneBound : `impl` *TraitBound*

匿名类型参数

注意：此部分只是一个占位符，预备将来这里会有更全面的参考资料。

注意：匿名类型参数通常被称为“参数位置上的实现trait(impl Trait in argument position)”。

函数可以将其参数声明为匿名类型的参数，这种情况下函数内部只能使用匿名类型参数的 trait 约束所提供的方法。之后的调用者(callee)在调用此函数时必须提供具体的类型，并且要求提供的具体类型必须拥有此匿名类型参数声明的约束。

它们被写成 `impl` 后跟一组 trait约束。

抽象返回类型

注意：此部分只是一个占位符，预备将来这里会有更全面的参考资料。

> 注意：抽象返回类型(abstract return types)通常被称为“函数返回位置上的实现trait(impl Trait in return position)”。

除了关联trait函数(associated trait function)外，其他函数都可以返回抽象返回类型。这些类型代表了某种具体类型，它要求在此返回类型的使用点(use-site)上只能使用由该类型的 trait约束声明的 trait方法。

它们被写成 `impl` 后跟一组 trait约束。

类型参数

parameters.md
commit: eb02dd5194a747277bfa46b0185d1f5c248f177b
本章译文最后维护日期：2020-11-14

在带有类型参数声明的程序项的代码体内，这些类型参数的名称可以直接当做类型使用：

```
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
}
```

这里，`first` 的类型为 `A`，援引的是 `to_vec` 的类型参数 `A`；`rest` 的类型为 `Vec<A>`，它是一个元素类型为 `A` 向量(vector)。

自动推断型类型

parameters.md
commit: 43dc1a42f19026f580e34a095e91804c3d6da186
本章译文最后维护日期：2020-11-14

句法

InferredType : `_`

自动推断型类型要求编译器尽可能根据周围可用的信息推断出实际使用类型。它不能用于程序项的签名中。它经常用于泛型参数中：

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

动态尺寸类型

dynamically-sized-types.md
commit: af1cf6d3ca3b7a8c434c142148742aa912e37c34
本章译文最后维护日期：2020-11-14

大多数的类型都有一个在编译时就已知的固定尺寸，并实现了 trait `Sized`。只有在运行时才知道尺寸的类型称为*动态尺寸类型(dynamically sized type)*（*DST*），或者非正式地称为非固定尺寸类型(unsized type)。切片和 trait对象是 DSTs 的两个例子。此类类型只能在某些情况下使用:

- 指向 DST 的指针类型的尺寸是固定的(sized)，但是是指向固定尺寸类型的指针的尺寸的两倍
  - 指向切片的指针也存储了切片的元素的数量。
  - 指向 trait对象的指针也存储了一个指向虚函数表(vtable)的指针地址
- 当接受了 `?Sized` 约束时，DST 可以作为类型实参(type arguments)使用。默认情况下，任何类型形参(type parameter)都拥有 `Sized` 约束。
- 可以为 DST 实现 trait。与类型参数中的默认设置不同，在 trait定义中默认存在 `Self: ?Sized` 约束。
- 结构体可以包含一个 DST 作为最后一个字段，这使得该结构体也成为是一个 DST。

注意：变量、函数参数、常量项和静态项必须是 `Sized`。

类型布局

type-layout.md
commit: be4332c6a8f93c425d6f9a8f59b2a6dec63f8ffe
本章译文最后维护日期：2020-12-17

类型的布局描述类型的尺寸(size)、对齐量(alignment)和字段(fields)的*相对偏移量(relative offsets)*。对于枚举，其判别值(discriminant)的布局和解释也是类型布局的一部分。

每次编译都有可能更改类型布局。这里我们只阐述当前编译器所保证的内容，而没试图去阐述编译器对此做了什么。

尺寸和对齐量

所有值都有一个对齐量和尺寸。

值的*对齐量*指定了哪些地址可以有效地存储该值。对齐量为 `n` 的值只能存储地址为 n 的倍数的内存地址上。例如，对齐量为 2 的值必须存储在偶数地址上，而对齐量为 1 的值可以存储在任何地址上。对齐量是用字节数来度量的，必须至少是 1，并且总是 2 的幂次。值的对齐量可以通过函数 `align_of_val` 来检测。

值的*尺寸*是同类型的值组成的数组中连续两个元素之间的字节偏移量，此偏移量包括了为保持程序项类型内部对齐而对此类型做的对齐填充。值的尺寸总是其对齐量的非负整数倍数。值的尺寸可以通过函数 `size_of_val` 来检测。

如果某类型的所有的值都具有相同的尺寸和对齐量，并且两者在编译时都是已知的，并且实现了 `Sized` trait，则可以使用函数 `size_of` 和 `align_of` 对此类型进行检测。没有实现 `Sized` trait 的类型被称为动态尺寸类型。由于实现了 `Sized` trait 的某一类型的所有值共享相同的尺寸和对齐量，所以我们分别将这俩共享值称为该类型的尺寸和该类型的对齐量。

原生类型的布局

下表给出了大多数原生类型(primitives)的尺寸。

类型	`size_of::<Type>()`
`bool`	1
`u8` / `i8`	1
`u16` / `i16`	2
`u32` / `i32`	4
`u64` / `i64`	8
`u128` / `i128`	16
`f32`	4
`f64`	8
`char`	4

`usize` 和 `isize` 的尺寸足以包含目标平台上的每个内存地址。例如，在 32-bit 目标上，它们是 4 个字节，而在 64-bit 目标上，它们是 8 个字节。

大多数原生类型的对齐量通常与它们的尺寸保持一致，尽管这是特定于平台的行为。比较典型的就是在 x86 平台上，u64 和 f64 都上 32-bit 的对齐量。

指针和引用的布局

指针和引用具有相同的布局。指针或引用的可变性不会影响其布局。

指向固定尺寸类型(sized type)的值的指针具有和 `usize` 相同的尺寸和对齐量。

指向非固定尺寸类型(unsized types)的值的指针是固定尺寸的。其尺寸和对齐量至少等于一个指针的尺寸和对齐量

> 注意：虽然不应该依赖于此，但是目前所有指向 DST 的指针都是 `usize` 的两倍尺寸，并且具有相同的对齐量。

数组的布局

数组的布局使得数组的第 n 个(`nth`)元素为从数组开始的位置向后偏移 *n * 元素类型的尺寸(`n * the size of the element's type`)* 个字节数。数组 `[T; n]` 的尺寸为 `size_of::<T>() *`

`n`，对齐量和 `T` 的对齐量相同。

切片的布局

切片的布局与它们所切的那部分数组片段相同。

> 注意：这是关于原生的 `[T]` 类型，而不是指向切片的指针（`&[T]`、`Box<[T]>` 等）。

字符串切片(`str`)的布局

字符串切片是一种 UTF-8 表示形式(representation)的字符序列，它们与 `[u8]` 类型的切片拥有相同的类型布局。

元组的布局

元组对于其布局没有任何保证。

一个例外情况是单元结构体(unit tuple)(`()`)类型，它被保证为尺寸为 0，对齐量为 1。

trait对象的布局

trait对象的布局与 trait对象的值相同。

> 注意：这是关于原生 trait对象类型(raw trait object type)的，而不是指向 trait对象的指针（`&dyn Trait`， `Box<dyn Trait>` 等）。

闭包的布局

闭包的布局没有保证。

表形/表示形式

所有用户定义的复合类型（结构体(`struct`)、枚举(`enum`)和联合体(`union`)）都有一个*表形(representation)*属性，该属性用于指定该类型的布局。类型的可能表形有：

- 默认(default)表形
- `C`表形
- 原语表形(primitive representation)
- 透明表形(`transparent`)

类型的表形可以通过对其应用`repr`属性来更改。下面的示例展示了一个`C`表形的结构体。

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

可以分别使用`align`和`packed`修饰符增大或缩小对齐量。它们可以更改属性中指定表形的对齐量。如果未指定表形，则更改默认表形的。

```
// 默认表形，把对齐量缩小到2。
#[repr(packed(2))]
struct PackedStruct {
    first: i16,
    second: i8,
    third: i32
}

// C表形，把对齐量增大到8
#[repr(C, align(8))]
struct AlignedStruct {
    first: i16,
    second: i8,
    third: i32
}
```

> 注意：由于表形是程序项的属性，因此表形不依赖于泛型参数。具有相同名称的任何两种类型都具有相同的表形。例如，`Foo<Bar>` 和 `Foo<Baz>` 都有相同的表形。

类型的表形可以更改字段之间的填充，但不会更改字段本身的布局。例如一个使用 `C` 表形的结构体，如果它包含一个默认表形的字段 `Inner`，那么它不会改变 `Inner` 的布局。

默认表形

没有 `repr` 属性的标称(nominal)类型具有默认表形。非正式地的情况下，也称这种表形为 `rust` 表形。

这种表形不保证每次编译都有统一的数据布局。

C表形

`C` 表形被设计用于双重目的：一个目的是创建可以与 C语言互操作的类型；第二个目的是创建可以正确执行依赖于数据布局的操作的类型，比如将值重新解释为其他类型。

因为这种双重目的存在，可以只利用其中的一个目的，如只创建有固定布局的类型，而放弃与C语言的互操作。

这种表型可以应用于结构体(structs)、联合体(unions)和枚举(enums)。一个例外是零变体枚举(zero-variant enums)，它的 `C` 表形是错误的。

`#[repr(C)]`结构体

结构体的对齐量是其*最大对齐量的字段(most-aligned field)*的对齐量。

字段的尺寸和偏移量则由以下算法确定：

1. 把当前偏移量设为从 0 字节开始。
2. 对于结构体中的每个字段，按其声明的先后顺序，首先确定其尺寸和对齐量；如果当前偏移量不是对其齐量的整倍数，则向当前偏移量添加填充字节，直至其对齐量的倍数[1]；至此，当前字段的偏移量就是当前偏移量；下一步再根据当前字段的尺寸增加当前偏移量。
3. 最后，整个结构体的尺寸就是当前偏移量向上取整到结构体对齐量的最小整数倍数。

下面用伪代码描述这个算法：

```
/// 返回偏移(`offset`)之后需要的填充量，以确保接下来的地址将被安排到可对齐的地址。
fn padding_needed_for(offset: usize, alignment: usize) -> usize {
    let misalignment = offset % alignment;
    if misalignment > 0 {
        // 向上取整到对齐量(`alignment`)的下一个倍数
        alignment - misalignment
    } else {
        // 已经是对齐量(`alignment`)的倍数了
        0
    }
}

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

let current_offset = 0;

for field in struct.fields_in_declaration_order() {
    // 增加当前字的偏移量段(`current_offset`)，使其成为该字段对齐量的倍数。
    // 对于第一个字段，此值始终为零。
    // 跳过的字节称为填充字节。
    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);
```

⚠️ 警告:这个伪代码使用了一个简单粗暴的算法，是为了清晰起见，它忽略了溢出问题。要在实际代码中执行内存布局计算，请使用 `Layout`。

注意：此算法可以生成零尺寸的结构体。在 C 语言中，像 `struct Foo { }` 这样的空结构体声明是非法的。然而，gcc 和 clang 都支持启用此类结构体的选项，并将其尺寸指定为零。跟 Rust 不同的是 C++ 给空结构体指定的尺寸为 1，并且除非它们是继承的，否则它们是具有 `[[no_unique_address]]` 属性的字段（在这种情况下，它们不会增大结构体的整体尺寸）。

`#[repr(C)]`联合体

使用 `#[repr(C)]` 声明的联合体将与相同目标平台上的 C语言中的 C联合体声明具有相同的尺寸和对齐量。联合体的对齐量等同于其所有字段的最大对齐量，尺寸将为其所有字段的最大尺寸，再对其向上取整到对齐量的最小整数倍。这些最大值可能来自不同的字段。

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

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

#[repr(C)]
union SizeRoundedUp {
   a: u32,
   b: [u16; 5],
}

assert_eq!(std::mem::align_of::<SizeRoundedUp>(), 4); // 来自于 a

assert_eq!(std::mem::size_of::<SizeRoundedUp>(), 12);  // 首先来自于b的尺寸10，然后向上取整到最近的4的整数倍12。
```

`#[repr(C)]`无字段枚举

对于无字段枚举(field-less enums)， `C` 表形的尺寸和对齐量与目标平台的 C ABI 的默认枚举尺寸和对齐量相同。

> 注意：C中的枚举的表形是由枚举的相应实现定义的，所以在 Rust 中，给无字段枚举应用C表形得到的表型很可能是一个“最佳猜测”。特别是，当使用某些特定命令行参数来编译特定的 C代码时，这可能是不正确的。

> ⚠ 警告：C语言中的枚举与 Rust 中的那些应用了 `#[repr(C)]` 表型的无字段枚举之间有着重要的区别。C语言中的枚举主要是 `typedef` 加上一些具名常量；换句话说，C枚举(`enum`)类型的对象可以包含任何整数值。例如，C枚举通常被用做标志位。相比之下，Rust的无字段枚举只能合法地[2]保存判别式的值，其他的都是未定义行为。因此，在 FFI 中使用无字段枚举来建模 C语言中的枚举(`enum`)通常是错误的。

`#[repr(C)]`带字段枚举

带字段的 `repr(C)` 枚举的表形其实等效于一个带两个字段的 `repr(C)` 结构体（这种在 C语言中也被称为“标签联合(tagged union)”），这两个字段：

- 一个为 `repr(C)` 表形的枚举（在这个等效结构体内，它也被叫做标签(the tag)字段），它就是原枚举所有的判别值组合成的新枚举，也就是它的变体是原枚举变体移除了它们自身所带的所有字段。
- 一个为 `repr(C)` 表形的联合体（在这个等效结构体内，它也被叫做载荷(the payload)字段），它的各个字段就是原枚举的各个变体把自己下面的字段重新组合成的 `repr(C)` 表形的结构体。

注意：由于等效出的结构体和联合体是 `repr(C)` 表形的，因此如果原来某一变体只有单个字段，则直接将该字段放入等效出的联合体中，或将其包装进一个次级结构体后再放入联合体中是没有区别的；因此，任何希望操作此类枚举表形的系统都可以选择使用这两种形式里对它们来说更方便或更一致的形式。

```
// 这个枚举的表形等效于 ...
#[repr(C)]
enum MyEnum {
    A(u32),
    B(f32, u64),
    C { x: u32, y: u8 },
    D,
 }

// ... 这个结构体
#[repr(C)]
struct MyEnumRepr {
    tag: MyEnumDiscriminant,
    payload: MyEnumFields,
}

// 这是原判别式组成的新枚举类型.
#[repr(C)]
enum MyEnumDiscriminant { A, B, C, D }

// 这是原变体的字段组成的联合体.
#[repr(C)]
union MyEnumFields {
    A: MyAFields,   // 译者注：因为原枚举变体A只有一个字段，所以此处的类型标注也可以直接替换为 u32,以省略 MyAFields这层封装
    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 }

// 这个结构体可以被省略(它是一个零尺寸类型)，但它必须出现在 C/C++ 头文件中
#[repr(C)]
#[derive(Copy, Clone)]
struct MyDFields;
```

> 注意： 联合体(`union`)可带有未实现 `Copy` 的字段的功能还没有纳入稳定版，具体参见 55149。

原语表形

*原语表形*是与原生整型具有相同名称的表形。也就是：`u8`，`u16`，`u32`，`u64`，`u128`，`usize`，`i8`，`i16`，`i32`，`i64`，`i128` 和 `isize`。

原语表形只能应用于枚举，此时枚举有没有字段会给原语表形带来不同的表现。给零变体枚举应用原始表形是错误的。将两个原语表形组合在一起也是错误的

无字段枚举的原语表形

对于无字段枚举，原语表形将其尺寸和对齐量设置成与给定表形同名的原生类型的表形的值。例如，一个 `u8` 表形的无字段枚举只能有0和255之间的判别值。

带字段枚举的原语表形

带字段枚举的原语表形是一个 `repr(C)` 表形的联合体，此联合体的每个字段对应一个和原枚举变体对应的 `repr(C)` 表形的结构体。这些结构体的第一个字段是原枚举的变体移除了它们所有的字段组成的原语表形版的无字段枚举（“the tag”），那这些结构体的其余字段是原变体移走的字段。

> 注意：如果在联合体中，直接把标签的成员赋予给标签(“the tag”)，那么这种表形结构仍不变的，并且这样操作对您来说可能会更清晰（尽管遵循 c++ 的标准，标签也应该被包装在结构体中）。

```
// 这个枚举的表形效同于 ...
#[repr(u8)]
enum MyEnum {
    A(u32),
    B(f32, u64),
    C { x: u32, y: u8 },
    D,
 }

// ... 这个联合体.
#[repr(C)]
union MyEnumRepr {
    A: MyVariantA,  //译者注：此字段类型也可直接用 u32 直接替代
    B: MyVariantB,  //译者注：此字段类型也可直接用 (f32, u64) 直接替代
    C: MyVariantC,
    D: MyVariantD,
}

// 这是原判别值组合成的新枚举。
#[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);
```

注意： 联合体(`union`)带有未实现 `Copy` trait 的字段的功能还没有纳入稳定版，具体参见 55149。

带字段枚举的原语表形与`#[repr(C)]`表形的组合使用

对于带字段枚举，还可以将 `repr(C)` 和原语表形（例如，`repr(C, u8)`）结合起来使用。这是通过将判别值组成的枚举的表形改为原语表形来实现的。因此，如果选择组合 `u8` 表形，那么组合出的判别值枚举的尺寸和对齐量将为 1 个字节。

那么这个判别值枚举就从前面示例中的样子变成：

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

// ...

#[repr(u8)] // 所以这里就用 `u8` 替代了 `C`
enum MyEnumDiscriminant { A, B, C, D }

// ...
```

例如，对于有 `repr(C, u8)` 属性的枚举，不可能有257个唯一的判别值（“tags”），而同一个枚举，如果只有单一 `repr(C)` 表形属性，那在编译时就不会出任何问题。

在 `repr(C)` 附加原语表形可以改变 `repr(C)` 表形的枚举的尺寸：

```
#[repr(C)]
enum EnumC {
    Variant0(u8),
    Variant1,
}

#[repr(C, u8)]
enum Enum8 {
    Variant0(u8),
    Variant1,
}

#[repr(C, u16)]
enum Enum16 {
    Variant0(u8),
    Variant1,
}

// C表形的尺寸依赖于平台
assert_eq!(std::mem::size_of::<EnumC>(), 8);
// 一个字节用于判别值，一个字节用于 Enum8::Variant0 中的值
assert_eq!(std::mem::size_of::<Enum8>(), 2);
// 两个字节用于判别值，一个字节用于Enum16::Variant0中的值，加上一个字节的填充
assert_eq!(std::mem::size_of::<Enum16>(), 4);
```

对齐量的修饰符

`align` 和 `packed` 修饰符可分别用于增大和减小结构体的和联合体的对齐量。`packed` 也可以改变字段之间的填充。

对齐量被指定为整型参数，形式为 `#[repr(align(x))]` 或 `#[repr(packed(x))]`。对齐量的值必须是从1到2^{29}之间的2的次幂数。对于 `packed`，如果没有给出任何值，如 `#[repr(packed)]`，则对齐量的值为1。

对于 `align`，如果类型指定的对齐量比其不带 `align` 修饰符时的对齐量小，则该指定的对齐量无效。

对于 `packed`，如果类型指定的对齐量比其不带 `packed` 修饰符时的对齐量大，则该指定的对齐量和布局无效。为了定位字段，每个字段的对齐量是指定的对齐量和字段的类型的对齐量中较小的那个对齐量。

`align` 和 `packed` 修饰符不能应用于同一类型，且 `packed` 修饰的类型不能直接或间接地包含另一个 `align` 修饰的类型。`align` 和 `packed` 修饰符只能应用于默认表形和 C表形中。

`align` 修饰符也可以应用在枚举上。如果这样做了，其对枚举对齐量的影响与将此枚举包装在一个新的使用了相同的 `align` 修饰符的结构体中的效果相同。

> ⚠ ***警告：***解引用一个未对齐的指针是未定义行为，但可以安全地创建指向 `packed` 修饰的字段的未对齐指针。就像在安全(safe) Rust 中所有创建未定义行为的方法一样，这是一个 bug。

透明(`transparent`)表形

透明(`transparent`)表型只能在只有一个字段的结构体(`struct`)上或只有一个变体的枚举(`enum`)上使用，这里只有一个字段/变体的意思是：

- 只能有一个非零尺寸的字段/变体，和
- 任意数量的尺寸为零对齐量为1的字段（例如： `PhantomData<T>` ）

使用这种表形的结构体和枚举与只有那个非零尺寸的字段具有相同的布局和 ABI。

这与 `C` 表形不同，因为带有 `C` 表形的结构体将始终拥有 C结构体(`C` `struct`)的ABI，例如，那些只有一个原生类型字段的结构体如果应用了透明表形(`transparent`)，将具有此原生类型字段的ABI。

因为此表形将类型布局委托给另一种类型，所以它不能与任何其他表形一起使用。

[1] 至此，上一个字段就填充完成，开始计算本字段了。也就是说每一个字段的偏移量是其字段的段首位置；那第一个字段的偏移量就始终为 0。

[2] 这里合法的意思是变体的判别值受 `repr(u8)` 这样的表形属性约束，像这个例子中，变体的判别值就只能位于 0~255 之间。

内部可变性

interior-mutability.md
commit: e7dd3618d78928322f53a20e2947e428b12eda2b
本章译文最后维护日期：2020-11-15

有时一个类型需要在存在多个别名时进行更改。在 Rust 中，这是通过一种叫做*内部可变性*的模式实现的。如果一个类型的内部状态可以通过对它的共享引用来进行更改，那么就说这个类型就具有内部可变性。这违背了共享引用所指向的值不能被更改的通常要求。

`std::cell::UnsafeCell<T>` 类型是 Rust 中唯一可以合法失效此要求的方法。当 `UnsafeCell<T>` 存在其他不变性的别名[1]时，仍然可以安全地对它包含的 `T` 进行更改或获得 `T` 的一个可变引用。与所有其他类型一样，拥有多个 `&mut UnsafeCell<T>` 别名是未定义行为。

通过使用 `UnsafeCell<T>` 作为字段，可以创建具有内部可变性的其他类型。标准库提供了几种这样的类型，这些类型都提供了安全的内部变更的 API。例如， `std::cell::RefCell<T>` 使用运行时借用检查来确保多个引用存在时的常规规则的执行。 `std::sync::atomic` 模块包含了一些类型，这些类型包装了一个只能通过原子操作访问的值，以允许在线程之间共享和修改该值。

[1] 当变量和指针表示的内存区域有重叠时，它们互为对方的别名。

子类型化和型变

subtyping.md
commit: 3b6fe80c205d2a2b5dc8a276192bbce9eeb9e9cf
本章译文最后维护日期：2021-03-02

子类型化是隐式的，可以出现在类型检查或类型推断的任何阶段。 Rust 中的子类型化的适用范围非常有限，仅出现在和生存期(lifetimes)的型变(variance)相关的地方，以及那些和高阶生存期相关的类型型变之间。 如果我们擦除了类型的生存期，那么唯一的子类型化就只是类型相等(type equality)了。

考虑下面的例子：字符串字面量总是拥有 `'static` 生存期。不过，我们还是可以把 `s` 赋值给 `t`：

```
fn bar<'a>() {
    let s: &'static str = "hi";
    let t: &'a str = s;
}
```

因为 `'static` 比生存期参数 `'a` 的寿命长，所以 `&'static str` 是 `&'a str` 的子类型。

高阶函数指针和trait对象可以形成另一种父子类型的关系。 它们是那些通过替换高阶生存期而得出的类型的子类型。举些例子：

```
// 这里 'a 被替换成了 'static
let subtype: &(for<'a> fn(&'a i32) -> &'a i32) = &((|x| x) as fn(&_) -> &_);
let supertype: &(fn(&'static i32) -> &'static i32) = subtype;

// 这对于 trait对象也是类似的
let subtype: &(for<'a> Fn(&'a i32) -> &'a i32) = &|x| x;
let supertype: &(Fn(&'static i32) -> &'static i32) = subtype;

// 我们也可以用一个高阶生存期来代替另一个
let subtype: &(for<'a, 'b> fn(&'a i32, &'b i32))= &((|x, y| {}) as fn(&_,
&_));
let supertype: &for<'c> fn(&'c i32, &'c i32) = subtype;
```

型变

型变是泛型类型相对其参数具有的属性。 泛型类型在它的某个参数上的*型变*是描述该参数的子类型化去如何影响此泛型类型的子类型化。

- 如果 `T` 是 `U` 的一个子类型意味着 `F<T>` 是 `F<U>` 的一个子类型（即子类型化"通过(passes through)"），则 `F<T>` 在 `T` 上是*协变的(covariant)*。
- 如果 `T` 是 `U` 的一个子类型意味着 `F<U>` 是 `F<T>` 的一个子类型，则 `F<T>` 在 `T` 上是*逆变的(contravariant)*。
- 其他情况下（即不能由参数类型的子类型化关系推导出此泛型的型变关系）， `F<T>` 在 `T` 上是的*不变的(invariant)*。

类型的型变关系由下表中的规则自动确定：

Type	在 `'a` 上的型变	在 `T` 上的型变
`&'a T`	协变的	协变的
`&'a mut T`	协变的	不变的
`*const T`		协变的
`*mut T`		不变的
`[T]` 和 `[T; n]`		协变的
`fn() -> T`		协变的
`fn(T) -> ()`		逆变的
`fn(T) -> T`		不变的
`std::cell::UnsafeCell<T>`		不变的
`std::marker::PhantomData<T>`		协变的
`dyn Trait<T> + 'a`	协变的	不变的

结构体(`struct`)、枚举(`enum`)、联合体(`union`)和元组(tuple)类型上的型变关系是通过查看其字段类型的型变关系来决定的。 如果参数用在了多处且具有不同型变关系的位置上，则该类型在该参数上是不变的。 例如，下面示例的结构体在 `'a` 和 `T` 上是协变的，在 `'b` 和 `U` 上是不变的。

```
use std::cell::UnsafeCell;
struct Variance<'a, 'b, T, U: 'a> {
    x: &'a U,               // 这让 `Variance` 在 'a 上是协变的，也让在 U 上是协变
的，但是后面也使用了 U
    y: *const T,            // 在 T 上是协变的
    z: UnsafeCell<&'b f64>, // 在 'b 上是不变的
    w: *mut U,              // 在 U 上是不变的，所以让整个结构体在 U 上是不变的
}
```

trait 约束和生命周期约束

trait-bounds.md
commit: f8e76ee9368f498f7f044c719de68c7d95da9972
本章译文最后维护日期：2020-11-15

句法

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

TypeParamBound :
Lifetime | *TraitBound*

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

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

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

trait约束和生命周期约束为泛型程序项提供了一种方法来限制将哪些类型和生命周期可被用作它们的参数。通过 where子句可以为任何泛型提供约束。对于某些常见的情况，也可以使用如下简写形式：

- 跟在泛型参数声明之后的约束：`fn f<A: Copy>() {}` 与 `fn f<A> where A: Copy () {}` 效果等价。
- 在 trait声明中作为指定超类trait(supertraits) 约束时：`trait Circle : Shape {}` 等同于 `trait Circle where Self : Shape {}`。
- 在 trait声明中作为指定关联类型上的约束时：`trait A { type B: Copy; }` 等同于 `trait A where Self::B: Copy { type B; }`。

在程序项上应用了约束就要求在使用该程序项时使用者必须满足这些约束。当对泛型程序项进行类型检查和借用检查时，约束可用来确认当前准备用来单态化此泛型的实例类型是否实现了约束给出的 trait。例如，给定 `Ty: Trait`：

- 在泛型函数体中，`Trait` 中的方法可以被 `Ty` 类型的值调用。同样，`Trait` 上的相关常数也可以被使用。
- `Trait` 上的关联类型可以被使用。
- 带有 `T: Trait` 约束的泛型函数或类型可以在使用 `T` 的地方替换使用 `Ty`。

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

fn draw_twice<T: Shape>(surface: Surface, sh: T) {
    sh.draw(surface);           // 能调用此方法上因为 T: Shape
    sh.draw(surface);
}

fn copy_and_draw_twice<T: Copy>(surface: Surface, sh: T) where T: Shape {
    let shape_copy = sh;        // sh 没有被使用移动语义移走，是因为 T: Copy
    draw_twice(surface, sh);    // 能使用泛型函数 draw_twice 是因为 T: Shape
}

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

fn name_figure<U: Shape>(
    figure: Figure<U>,          // 这里类型 Figure<U> 的格式正确是因为 U: Shape
) {
    println!(
        "Figure of two {}",
        U::name(),              // 可以使用关联函数
    );
}
```

trait 和生命周期约束也被用来命名 trait对象。

?Sized

`?` 仅用于声明 `Sized` trait 可能不会被某类型参数或关联类型实现。`?Sized` 还不能用作其他类型的约束。

生命周期约束

生命周期约束可以应用于类型或其他生命周期。约束 `'a: 'b` 通常被解读为 *`'a` 比 `'b` 存活的时间久*。`'a: 'b` 意味着 `'a` 持续的时间比 `'b` 长，所以只要 `&'b ()` 有效，引用 `&'a ()` 就有效。[1]

```
fn f<'a, 'b>(x: &'a i32, mut y: &'b i32) where 'a: 'b {
    y = x;                          // 因为 'a: 'b，所以&'a i32 是 &'b i32 的子类型
    let r: &'b &'a i32 = &&0;       // &'b &'a i32 格式合法是因为 'a: 'b
}
```

`T: 'a` 意味着 `T` 的所有生命周期参数都比 `'a` 存活得时间长。例如，如果 `'a` 是一个任意的(unconstrained)生命周期参数，那么 `i32: 'static` 和 `&'static str: 'a` 都合法，但 `Vec<&'a ()>: 'static` 不合法。

高阶 trait 约束

可以在生命周期上再进行*更高阶的*类型约束。这些高阶约束指定了一个对*所有*生命周期都为真的约束。例如，像 `for<'a> &'a T: PartialEq<i32>` 这样的约束需要一个如下的实现

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

这样就可以拿任意生命周期的 `&'a T` 和 `i32` 做比较啦。

下面这类场景只能使用高阶trait约束，因为引用的生命周期比函数的生命周期参数短：[2]

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

译者注：译者下面举例代码可以和上面原文的代码对比着看。下面代码中，因为 `F` 没约束 `'a`，导致参数 `f` 引用了未经扩展生命周期的 `zero`

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

高阶生命周期也可以贴近 trait 来指定，唯一的区别是生命周期参数的作用域，像下面这样 `'a` 的作用域只扩展到后面跟的 trait 的末尾，而不是整个约束[3]。下面这个函数和上一个等价。

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

[1] 译者理解：理解这种关系时，可以把生命周期 `'a` 和 `'b` 理解成去引用对象时需传入的参数，给定 `'a: 'b` 和类型 `T`，如果 `'b T` 有效，那此时再传入 `'a` 就去引用 `T` 必定有效。

[4] 译者理解：高阶 trait约束就是对带生命周期的类型重新进行约束。像这句中的例子就是对 `&'a T` 加上了 `PartialEq<i32>` 的约束，其中 `for<'a>` 可以理解为：对于 `'a` 的所有可能选择。更多信息请参见：https://doc.rust-lang.org/std/cmp/trait.PartialEq.html 和 https://doc.rust-lang.org/nightly/nomicon/hrtb.html

[2] 译者理解此例中的代码 `for<'a> F: Fn(&'a i32)` 为：`F` 对于 `'a` 的所有可能选择都受 `Fn(&'a i32)` 的约束。

[3] 译者理解这句的意思是：如果 `F` 的约束有多个 trait，那这种方式里， `'a` 的作用域只是扩展它后面紧跟的那个 trait 的方法，即 `Fn(&'a i32)` 里。

类型自动强转

type-coercions.md
commit: d5a5e32d3cda8a297d2a91a85b91ff2629b0e896
本章译文最后维护日期：2020-11-15

类型自动强转是改变值的类型的隐式操作。它们在特定的位置自动发生，但实际自动强转的类型也受到很多限制。

任何允许自动强转的转换都可以由类型强制转换操作符 `as` 来显式执行。

自动强转最初是在 RFC 401 中定义的，并在[RFC 1558] 中进行了扩展。

自动强转点

自动强转只能发生在程序中的某些自动强转点(coercion sites)上；典型的位置是那些所需的类型是显式给出了的地方，或者是那些可以从给出的显式类型传播推导(be derived by propagation)出所需的类型（注意这里不是类型推断）的地方。可能的强转点有：

- `let` 语句中显式给出了类型。

  例如，下面例子中 `&mut 42` 自动强转成 `&i8` 类型：

  ```
  let _: &i8 = &mut 42; // 译者注释：`&i8` 是显示给出的所需类型
  ```

- 静态(`static`)项和常量(`const`)项声明（类似于 `let` 语句）。
- 函数调用的参数

  被强制的值是实参(actual parameter)，它的类型被自动强转为形参(formal parameter)的类型。

  例如，下面例子中 `&mut 42` 自动强转成 `&i8` 类型：

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

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

对于方法调用，接受者（`self`参数）只能使用非固定尺寸类型自动强转(unsized coercion)。

- 实例化结构体、联合体或枚举变体的字段。

  例如，下面例子中 `&mut 42` 自动强转成 `&i8` 类型：

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

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

- 函数结果—块中的最终表达式或者 `return` 语句中的任何表达式。

  例如，下面例子中 `x` 将自动强转成 `&dyn Display` 类型：

```

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

如果一在自动强转点中的表达式是自动强转传播型表达式(coercion-propagating expression)，那么该表达式中的对应子表达式也是自动强转点。传播从这些新的自动强转点开始递归。传播表达式(propagating expressions)及其相关子表达式有：

- 数组字面量，其数组的类型为 `[U; n]`。数组字面量中的每个子表达式都是自动强转到类型 `U` 的自动强转点。
- 重复句法声明的数组字面量，其数组的类型为 `[U; n]`。重复子表达式是用于自动强转到类型 `U` 的自动强转点。
- 元组，其中如果元组是自动强转到类型 `(U_0, U_1, ..., U_n)` 的强转点，则每个子表

达式都是相应类型的自动强转点，比如第0个子表达式是到类型 `U_0` 的 自动强转点。

- 圆括号括起来的子表达式（`(e)`）：如果整个括号表达式的类型为 `U`，则子表达式 `e` 是自动强转到类型 `U` 的自动强转点。
- 块：如果块的类型是 `U`，那么块中的最后一个表达式（如果它不是以分号结尾的）就是一个自动强转到类型 `U` 的自动强转点。这里的块包括作为控制流语句的一部分的条件分支代码块，比如 `if`/`else`，当然前提是这些块的返回需要有一个已知的类型。

自动强转类型

自动强转允许发生在下列类型之间：

- `T` 到 `U` 如果 `T` 是 `U` 的一个子类型 (*反射性场景(reflexive case)*)
- `T_1` 到 `T_3` 当 `T_1` 可自动强转到 `T_2` 同时 `T_2` 又能自动强转到 `T_3` (*传递性场景(transitive case)*)

  注意这个还没有得到完全支持。
- `&mut T` 到 `&T`
- `*mut T` 到 `*const T`
- `&T` 到 `*const T`
- `&mut T` 到 `*mut T`
- `&T` 或 `&mut T` 到 `&U` 如果 `T` 实现了 `Deref<Target = U>`。例如：

```
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 自动强转成 &char.
}
```

- `&mut T` 到 `&mut U` 如果 `T` 实现了 `DerefMut<Target = U>`.
- TyCtor(`T`) 到 TyCtor(`U`)，其中 TyCtor(`T`) 是下列之一[1]
  - `&T`
  - `&mut T`
  - `*const T`
  - `*mut T`
  - `Box<T>`

  并且 `U` 能够通过非固定尺寸类型自动强转得到。
- 非捕获闭包(Non capturing closures)到函数指针(`fn` pointers)
- `!` 到任意 `T`

非固定尺寸类型自动强转

下列自动强转被称为非固定尺寸类型自动强转(`unsized coercions`)，因为它们与将固定尺寸类型(sized types)转换为非固定尺寸类型(unsized types)有关，并且在一些其他自动强转不允许的情况（也就是上面罗列的情况之外的情况）下允许使用。也就是说它们可以发生在任何自动强转发生的地方。

`Unsize` 和 `CoerceUnsized` 这两个 trait 被用来协助这种转换的发生，并公开给标准库来使用。以下自动强转方式是内置的，并且，如果 `T` 可以用其中任一方式自动强转成 `U`，那么就会为 `T` 提供一个 `Unsize<U>` 的内置实现：

- `[T; n]` 到 `[T]`.
- `T` 到 `dyn U`, 当 `T` 实现 `U + Sized`, 并且 `U` 是对象安全的时。
- `Foo<..., T, ...>` 到 `Foo<..., U, ...>`, 当：
  - `Foo` 是一个结构体。
  - `T` 实现了 `Unsize<U>`。
  - `Foo` 的最后一个字段是和 `T` 相关的类型。
  - 如果这最后一个字段是类型 `Bar<T>`，那么 `Bar<T>` 实现了 `Unsized<Bar<U>>`。
  - `T` 不是任何其他字段的类型的一部分。

此外，当 `T` 实现了 `Unsize<U>` 或 `CoerceUnsized<Foo<U>>` 时，类型 `Foo<T>` 可以实现 `CoerceUnsized<Foo<U>>`。这就允许 `Foo<T>` 提供一个到 `Foo<U>` 的非固定尺寸类型自动强转。

> 注：虽然非固定尺寸类型自动强转的定义及其实现已经稳定下来，但 `Unsize` 和 `CoerceUnsized` 这两个 trait 本身还没稳定下来，因此还不能直接用于稳定版的 Rust。

最小上界自动强转

在某些上下文中，编译器必须将多个类型强制在一起，以尝试找到最通用的类型。这被称为“最小上界(Least Upper Bound,简称 LUB)”自动强转。LUB自动强转只在以下情况中使用：

- 为一系列的 if分支查找共同的类型。
- 为一系列的匹配臂查找共同的类型。
- 为数组元素查找共同的类型。
- 为带有多个返回项语句的闭包的返回类型查找共同的类型。

- 检查带有多个返回语句的函数的返回类型。

在这每种情况下，都有一组类型 `T0..Tn` 被共同自动强转到某个未知的目标类型 `T_t`，注意开始时 `T_t` 是未知的。LUB 自动强转的计算过程是不断迭代的。首先把目标类型 `T_t` 定为从类型 `T0` 开始。对于每一种新类型 `Ti`，考虑如下步骤是否成立：

- 如果 `Ti` 可以自动强转为当前目标类型 `T_t`，则不做任何更改。
- 否则，检查 `T_t` 是否可以被自动强转为 `Ti`；如果是这样，`T_t` 就改为 `Ti`。（此检查还取决于到目前为止所考虑的所有源表达式是否带有隐式自动强转。）
- 如果不是，尝试计算一个 `T_t` 和 `Ti` 的共同的超类型(supertype)，此超类型将成为新的目标类型。

示例：

```
// if分支的情况
let bar = if true {
    a
} else if false {
    b
} else {
    c
};

// 匹配臂的情况
let baw = match 42 {
    0 => a,
    1 => b,
    _ => c,
};

// 数组元素的情况
let bax = [a, b, c];

// 多个返回项语句的闭包的情况
let clo = || {
    if true {
        a
    } else if false {
        b
    } else {
        c
    }
};
let baz = clo();

// 检查带有多个返回语句的函数的情况
fn foo() -> i32 {
    let (a, b, c) = (0, 1, 2);
    match 42 {
        0 => a,
        1 => b,
        _ => c,
    }
}
```

在这些例子中， `ba*` 的类型可以通过 LUB自动强转找到。编译器检查 LUB自动强转在处理函数 `foo` 时，是否把 `a`， `b`， `c` 的结果转为了 `i32` 。

附加说明

我们这种描述显然是非正式的，但目前使文字描述更精确的工作正作为精细化 Rust 类型检查器的一般性工作的一部分正紧锣密鼓的进行中。

[1] TyCtor为类型构造器 type constructor 的简写。

析构函数

destructors.md
commit: b2d11240bd9a3a6dd34419d0b0ba74617b23d77e
本章译文最后维护日期：2020-11-16

当一个初始化了的变量或临时变量超出作用域时，*其析构函数(destructor)将运行，或者说它将被销毁(dropped)*。此外赋值操作也会运行其左操作数的析构函数（如果它已经初始化了）。如果变量已部分初始化了，则只销毁其已初始化的字段。

类型 `T` 的析构函数由以下内容组成：

1. 如果有约束 `T: Drop`，则调用 `<T as std::ops::Drop>::drop`
2. 递归运行其所有字段的析构函数。
    - 结构体(`struct`)的字段按照声明顺序被销毁。
    - 活动枚举变体的字段按声明顺序销毁。
    - 元组中的字段按顺序销毁。
    - 数组或拥有所有权的切片的元素的销毁顺序是从第一个元素到最后一个元素。
    - 闭包通过移动(move)语义捕获的变量的销毁顺序未明确指定。
    - trait对象的销毁会运行其非具名基类(underlying type)的析构函数。
    - 其他类型不会导致任何进一步的销毁动作发生。

如果析构函数必须手动运行，比如在实现自定义的智能指针时，可以使用标准库函数 `std::ptr::drop_in_place`。

举些（析构函数的）例子：

```
struct PrintOnDrop(&'static str);

impl Drop for PrintOnDrop {
    fn drop(&mut self) {
        println!("{}", self.0);
    }
}

let mut overwritten = PrintOnDrop("当覆写时执行销毁");
overwritten = PrintOnDrop("当作用域结束时执行销毁");

let tuple = (PrintOnDrop("Tuple first"), PrintOnDrop("Tuple second"));

let moved;
// 没有析构函数在赋值时运行
moved = PrintOnDrop("Drops when moved");
// 这里执行销毁，但随后变量进入未初始化状态
moved;

// 未初始化不会被销毁
let uninitialized: PrintOnDrop;

// 在部分移动之后，后续销毁动作只销毁剩余字段。
let mut partial_move = (PrintOnDrop("first"), PrintOnDrop("forgotten"));
// 执行部分移出，只留下 `partial_move.0` 处于初始化状态
core::mem::forget(partial_move.1);
// 当 partial_move 的作用域结束时，这里就只有第一个字段被销毁。
```

销毁作用域

每个变量或临时变量都与一个*销毁作用域(drop scope)*[1]相关联。当控制流离开一个销毁作用域时，与该作用域关联的所有变量都将按照其声明（对变量而言）或创建（对临时变量而言）的相反顺序销毁。

销毁作用域是在将 `for`、`if let` 和 `while let` 这些表达式替换为等效的 `match` 表达式之后确定的。在确定销毁作用域这事儿上，对重载操作符与内置操作符不做区分[2]，匹配模式的变量绑定方式(binding modes)也不去考虑。

给定一个函数或闭包，存在以下的销毁作用域：

- 整个函数
- 每个语句

- 每个表达式
- 每个块，包括函数体
  - 当在块表达式上时，整个块和整个块表达式的销毁作用域是相同的
- 匹配(`match`)表达式的每条匹配臂(arm)

销毁作用域相互嵌套如规则下。当同时离开多个作用域时，比如从函数返回时，变量会从内层的向层依次销毁。

- 整个函数作用域是最外层的作用域。
- 函数体块包含在整个函数作用域内。
- 表达式语句中的表达式的父作用域是该语句的作用域。
- `let`语句的初始化器(initializer)的父作用域是`let`语句的作用域。
- 语句作用域的父作用域是包含该语句的块作用域。
- 匹配守卫(`match` guard)表达式的父作用域是该守卫所在的匹配臂的作用域。
- 在匹配表达式(`match` expression)里的`=>`之后的表达式的父作用域是此表达式对应的匹配臂所在的那个作用域。
- 匹配臂的作用域的父作用域是它所在的匹配表达式(`match` expression)的作用域。
- 所有其他作用域的父作用域都是直接封闭该表达式的作用域。

函数参数的作用域

所有函数参数都在整个函数体的作用域内有效，因此在对函数求值时，它们是最后被销毁的。实参会在其内部值被形参的模式绑定之后销毁。

```
// 先销毁第二个参数，接下来是 `y`，然后是第一个参数r，最后是 `x`
fn patterns_in_parameters(
    (x, _): (PrintOnDrop, PrintOnDrop),
    (_, y): (PrintOnDrop, PrintOnDrop),
) {}

// 销毁顺序是 3 2 0 1
patterns_in_parameters(
    (PrintOnDrop("0"), PrintOnDrop("1")),
    (PrintOnDrop("2"), PrintOnDrop("3")),
);
```

本地变量的作用域

在`let`语句中声明的局部变量与包含此`let`语句的块作用域相关联。在匹配(`match`)表达式中

声明的局部变量与声明它们的匹配(`match`)臂的匹配臂作用域相关联。

```
let declared_first = PrintOnDrop("在外层作用域内最后销毁");
{
    let declared_in_block = PrintOnDrop("在内层作用域内销毁");
}
let declared_last = PrintOnDrop("在外层作用域内最先销毁");
```

如果在一个匹配(`match`)表达式的同一个匹配臂中使用了多个模式，则毁顺序不确定。（译者注：这里译者不确定后半句翻译是否准确，这里给出原文：then an unspecified pattern will be used to determine the drop order.）

临时作用域

表达式的*临时作用域*用于该表达在位置上下文中求出的结果被保存进的那个临时变量的作用域。有些情况下，此表达式求出的结果会被提升，则此表达式不存在临时作用域。[3]

除了生存期扩展之外，表达式的临时作用域是包含该表达式的最小作用域，它适用于以下情况之一:

- 整个函数体。
- 一条语句。
- `if`表达式、`while`表达式 或 `loop`表达式这三种表达式的代码体。
- `if`表达式的 `else`块。
- `if`表达式的条件表达式， `while`表达式的条件表达式，或匹配表达式中的匹配(`match`)守卫。
- 匹配臂上的表达式。
- 惰性布尔表达式的第二操作数。

注意:

在函数体的最终表达式(final expression)中创建的临时变量会在任何具名变量销毁*之后*销毁，因为这里没有更小的封闭它的临时作用域。

匹配表达式的检验对象表达式本身不是一个临时作用域（，但它内部可以包含临时作用域），因此可以在匹配(`match`)表达式之后销毁检验对象表达式中的临时作用域。例如，`match 1 { ref mut z => z };` 中的 `1` 所在的临时变量一直存活到此语句结束。

一些示例：

```
let local_var = PrintOnDrop("local var");

// 在条件表达式执行后立即销毁
if PrintOnDrop("If condition").0 == "If condition" {
    // 在此块的末尾处销毁
    PrintOnDrop("If body").0
} else {
    unreachable!()
};

// 在此条语句的末尾处销毁
(PrintOnDrop("first operand").0 == ""
// 在 ) 处销毁
|| PrintOnDrop("second operand").0 == "")
// 在此表达式的末尾处销毁
|| PrintOnDrop("third operand").0 == "";

// 在函数末尾处，局部变量之后销毁之后销毁
// 将下面这段更改为一个包含返回(return)表达式的语句将使临时变量在本地变量之前被删除。
// 如果把此临时变量绑定到一个变量，然后返回这个变量，也会先删除这个临时变量
match PrintOnDrop("Matched value in final expression") {
    // 在条件表达式执行后立即销毁
    _ if PrintOnDrop("guard condition").0 == "" => (),
    _ => (),
}
```

操作数

在同一表达式中，在对其他操作数求值时，也会创建临时变量来将已求值的操作数的结果保存起来。临时变量与该操作数所属的表达式的作用域相关联。因为一旦表达式求值，临时变量就被移走了，所以销毁它们没有任何效果和意义，除非整个表达式的某一操作数出现异常，导致表达式求值失败，或提前返回，或触发了 panic。

```
loop {
    // 元组表达式未结束求值就提前返回了，所以其操作数按声明的反序销毁
    (
        PrintOnDrop("Outer tuple first"),
        PrintOnDrop("Outer tuple second"),
        (
            PrintOnDrop("Inner tuple first"),
            PrintOnDrop("Inner tuple second"),
            break,
        ),
        PrintOnDrop("Never created"),
    );
}
```

常量提升

如果可以通过借用的方式将某一值表达式写入常量，并且还能通过解引用此借用的方式来解出原始写入的表达式，并且如果这种做法也不更改运行时行为，那 Rust 会将值表达式提升到静态（`'static`）slot 作用域内。也就是说，提升的表达式可以在编译时求值，这求得的值不具备内部可变性或不包含析构函数（。这些属性是根据可能的值来确定的，例如 `&None` 的类型总是 `&'static Option<_>`，因为 `&None` 的值是唯一确定的）。

临时生存期扩展

注意：临时生存期扩展的确切规则可能还会改变。这里只描述了当前的行为表现。

`let` 语句中表达式的临时作用域有时会*扩展*到包含此 `let` 语句的块作用域内。根据某些句法规则，当通常的临时作用域太小时，就会这样做。例如：

```
let x = &mut 0;
// 通常上面存储0的临时变量（或者临时位置）到这里就会被丢弃，但这里是一直存在到块的末尾。
println!("{}", x);
```

如果一个借用、解引用、字段或元组索引表达式有一个扩展的临时作用域，那么它们的操作数也会同样扩展。如果索引表达式有扩展的临时作用域，那么被索引的表达式也会一并扩展。

基于模式的扩展

*扩展临时作用域的模式(extending pattern)*是下面任一：

- 绑定方式为引用或可变引用的标识符模式。
- 结构体(`struct`)模式、元组模式、元组结构体模式或切片模式，其中它们至少有一个直接子模式是扩展临时作用域的模式。

所以 `ref x` 、 `V(ref x)` 和 `[ref x, y]` 都是扩展临时作用域的模式，但是 `x` 、 `&ref x` 和 `&(ref x,)` 不是。

如果 `let` 语句中的模式是扩展临时作用域的模式，那么初始化器表达式中的临时作用域将被扩展。

基于表达式的扩展

对于带有初始化器的 let语句来说，*扩展临时作用域的表达式(extending expression)*是以下表达式之一：

- 初始化表达式(initializer expression)。
- 扩展临时作用域的借用表达式的操作数。
- 扩展临时作用域的数组、强制转换(cast)、花括号括起来的结构体或元组表达式的操作数。
- 任何扩展临时作用域的块表达式的最终表达式(final expression);

因此，在 `&mut 0` 、 `(&1, &mut 2)` 和 `Some { 0: &mut 3 }` 中的借用表达式都是扩展临时作用域的表达式。在 `&0 + &1` 和一些 `Some(&mut 0)` 中的借用不是：它们在句法上是函数调用表达式。

任何扩展了临时作用域的借用表达式的操作数的临时作用域都随此表达式的临时作用域的扩展而扩展。

示例

这是一些带有扩展的临时作用域的表达式：

```
// 在这些情况下，存储 `temp()` 结果的临时变量与x在同一个作用域中。
let x = &temp();
let x = &temp() as &dyn Send;
let x = (&*&temp(),);
let x = { [Some { 0: &temp(), }] };
let ref x = temp();
let ref x = *&temp();
```

下面是一些表达式没有扩展临时作用域的例子：

```
// 在这些情况下，存储 `temp()` 结果的临时变量只存活到 let语句结束。

let x = Some(&temp());          // ERROR
let x = (&temp()).use_temp();  // ERROR
```

阻断运行析构函数

在 Rust 中，即便类型不是 `'static`，禁止运行析构函数也是允许的，也是安全的。`std::mem::ManuallyDrop` 提供了一个包装器(wrapper)来防止变量或字段被自动销毁。

[1] 后文有时也直接简称为作用域。

[2] 这里说这句是因为操作符的操作数也涉及到销毁作用域范围的确定。

[3] 对这句话，这里译者按自己的理解再翻译一遍：一个表达式在位置表达式上使用时会被求值，如果此时没有具名变量和此值绑定，那就会先被保存进一个临时变量里，临时作用域就是伴随这此临时变量而生成。此作用域通常在此表达式所在的语句结束时结束，但如果求出的值被通过借用绑定给具名变量，此作用域会扩展到此具名变量的作用域（后面生存期扩展会讲到）。如果求出的值比较特殊，这个作用域还会提升到全局作用域，这就是所谓的常量提升。

生命周期(类型参数)省略

lifetime-elision.md
commit: f8e76ee9368f498f7f044c719de68c7d95da9972
本章译文最后维护日期：2020-11-16

Rust 拥有一套允许在多种位置省略生命周期的规则，但前提是编译器在这些位置上能推断出合理的默认生命周期。

函数上的生命周期(类型参数)省略

为了使常用模式使用起来更方便，可以在函数项、函数指针和闭包trait[1]的签名中*省略*生命周期类型参数。以下规则用于推断出被省略的生命周期类型参数。省略不能被推断出的生命周期类型参数是错误的。占位符形式的生命周期，`'_`，也可以用这一套规则来推断出。对于路径中的生命周期，首选使用`'_`。trait对象的生命周期类型参数遵循不同的规则，具体这里讨论。

- 参数中省略的每个生命周期类型参数都会（被推断）成为一个独立的生命周期类型参数。
- 如果参数中只使用了一个生命周期（省略或不省略都行），则将该生命周期作为*所有*省略的输出生命周期类型参数。

在方法签名中有另一条规则

- 如果接受者(receiver)类型为`&Self`或`&mut Self`，那么对`Self`的引用的生命周期会被作为所有省略的输出生命周期类型参数。

示例：

```
fn print1(s: &str);                                 // 省略
fn print2(s: &'_ str);                              // 也省略
fn print3<'a>(s: &'a str);                          // 未省略

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

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

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

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

fn new1(buf: &mut [u8]) -> Thing<'_>;               // 省略 - 首选的
fn new2(buf: &mut [u8]) -> Thing;                   // 省略
fn new3<'a>(buf: &'a mut [u8]) -> Thing<'a>;        // 未省略

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

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

```
// 下面的示例展示了不允许省略生命周期类型参数的情况。

// 无法推断，因为没有可以推断的起始参数。
fn get_str() -> &str;                               // 非法

// 无法推断，这里无法确认输出的生命周期类型参数该遵从从第一个还是第二个参数的。
fn frob(s: &str, t: &str) -> &str;                  // 非法
```

默认的 trait对象的生命周期

假定存在于（代表） trait对象（的那个胖指针）上的生命周期(assumed lifetime)类型参数称为此 trait对象的*默认对象生命周期约束(default object lifetime bound)*。这些在 RFC 599 中定

义，在 RFC 1156 中修定增补。

当 trait对象的生命周期约束被完全省略时，会使用默认对象生命周期约束来替代上面定义的生命周期类型参数省略规则。但如果使用 `'_` 作为生命周期约束，则该约束仍遵循上面通常的省略规则。

如果将 trait对象用作泛型类型的类型参数，则首先使用此容器泛型来尝试为此 trait对象推断一个约束（来替代那个假定的生命周期）。

- 如果存在来自此容器泛型的唯一约束，则该约束就为此 trait对象的默认约束
- 如果此容器泛型有多个约束，则必须指定一个约显式束为此 trait对象的默认约束

如果这两个规则都不适用，则使用该 trait对象的声明时的 trait约束：

- 如果原 trait 声明为单生命周期*约束*，则此 trait对象使用该约束作为默认约束。
- 如果 `'static` 被用做原 trait声明的任一一个生命周期约束，则此 trait对象使用 `'static` 作为默认约束。
- 如果原 trait声明没有生命周期约束，那么此 trait对象的生命周期会在表达式中根据上下文被推断出来，在表达式之外直接用 `'static`。

```
// 对下面的 trait 来说，...
trait Foo { }

// 这两个是等价的，就如 `Box<T>` 对 `T` 没有生命周期约束一样
// These two are the same as Box<T> has no lifetime bound on T
type T1 = Box<dyn Foo>;  //译者注：此处的 `T1` 和 上行备注中提到的 `Box<T>` 都是本节规则中所说的泛型类型，即容器泛型
type T2 = Box<dyn Foo + 'static>;

// ...这也是等价的
impl dyn Foo {}
impl dyn Foo + 'static {}

// ...这也是等价的，因为 `&'a T` 需要 `T: 'a`
type T3<'a> = &'a dyn Foo;
type T4<'a> = &'a (dyn Foo + 'a);

// `std::cell::Ref<'a, T>` 也需要 `T: 'a`，所以这俩也是等价的
type T5<'a> = std::cell::Ref<'a, dyn Foo>;
type T6<'a> = std::cell::Ref<'a, dyn Foo + 'a>;
```

```
// 这是一个反面示例:
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>;
//                                  ^^^^^^^
// 错误: 不能从上下文推导出此对象类型的生命周期约束
```

注意，像 `&'a Box<dyn Foo>` 这样多层包装的，只需要看最内层包装 `dyn Foo` 的那层，所以扩展后仍然为 `&'a Box<dyn Foo + 'static>`

```
// 对下面的 trait 来说, ...
trait Bar<'a>: 'a { }

// ...这两个是等价的:
type T1<'a> = Box<dyn Bar<'a>>;
type T2<'a> = Box<dyn Bar<'a> + 'a>;

// ...这俩也是等价的:
impl<'a> dyn Bar<'a> {}
impl<'a> dyn Bar<'a> + 'a {}
```

静态(`'static`)生命周期省略

除非指定了显式的生命周期，引用类型的常量项声明和静态项声明都具有*隐式的*静态(`'static`)生命周期。因此，有 `'static` 生命周期的常量项声明在编写时可以略去其生命周期。

```
// 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,
};
```

注意，如果静态项(`static`)或常量项(`const`)包含对函数或闭包的引用，而这些函数或闭包本身也包含引用，此时编译器将首先尝试使用标准的省略规则来推断生命周期类型参数。如果它不能通过通常的生命周期省略规则来推断出生命周期类型参数，那么它将报错。举个例子：

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

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

```
// 没有足够的信息将返回值的生命周期与参数的生命周期绑定起来，因此这是一个错误
const RESOLVED_STATIC: &dyn Fn(&Foo, &Bar) -> &Baz = &somefunc;
//                                               ^
// 这个函数的返回类型包含一个借用来的值，但是签名没有说明它是从参数1还是从参数2借用来的
```

[1] 指 Fn、FnMute 和 FnOnce 这三个 trait。

特殊类型和 trait

special-types-and-traits.md
commit: fcdc0cab546c10921d66054be25c6afc9dd6b3bc
本章译文最后维护日期：2020-11-16

标准库中的某些类型和 trait 在 Rust 编译器中也直接能用。本章就阐述了这些类型和 trait 的特殊特性。

`Box<T>`

`Box<T>` 有一些特殊的特性，Rust 目前还不允许用户定义类型时使用。

- `Box<T>` 的[解引用操作符]会产生一个可从中移出值的内存位置[1]。这（种特殊性）意味着应用在 `Box<T>` 上的 `*` 运算符和 `Box<T>` 的析构函数都是语言内置的。
- 方法可以使用 `Box<Self>` 作为接受者。
- `Box<T>` 可以绕过孤儿规则(orphan rules)，与 `T` 在同一 crate 中实现同一 trait，其他泛型类型无法绕过。

`Rc<T>`

方法可以使用 `Rc<Self>` 作为接受者。

`Arc<T>`

方法可以使用 `Arc<Self>` 作为接受者。

`Pin<P>`

方法可以使用 `Pin<P>` 作为接受者。

`UnsafeCell<T>`

`std::cell::UnsafeCell<T>` 用于内部可变性。它确保编译器不会对此类类型执行不正确的优化。它还能确保具有内部可变类型的静态(`static`)项不会被放在标记为只读的内存中。

`PhantomData<T>`

`std::marker::PhantomData<T>` 是一个零尺寸零的、最小对齐量的、被认为拥有(own) `T` 的类型，这个类型存在目的是应用在确定型变关系、销毁检查和自动trait 中的。

Operator Traits

运算符/操作符trait

`std::ops` 和 `std::cmp` 中的 trait 看用于重载 Rust 的运算符/操作符、索引表达式和调用表达式。

`Deref` and `DerefMut`

除了重载一元 `*` 运算符外， `Deref` 和 `DerefMut` 也用于方法解析(method resolution)和利用 `Deref` 达成自动强转。

`Drop`

`Drop` trait 提供了一个析构函数，每当要销毁此类型的值时就会运行它。

Copy

`Copy` trait 改变实现它的类型的语义。其类型实现了 `Copy` 的值在赋值时将被复制而不是移动。

只能为未实现 `Drop` trait 且字段都是 `Copy` 的类型实现 `Copy`。[2]
对于枚举，这意味着所有变体的所有字段都必须是 `Copy` 的。
对于联合体，这意味着所有的变体都必须是 `Copy` 的。

`Copy` 已由编译器实现给了下述类型：

- 数字类类型
- `char`，`bool`，和 `!`
- 由 `Copy` 类型的元素组成的元组
- 由 `Copy` 类型的元素组成的数组
- 共享引用
- 裸指针
- 函数指针 和 函数项类型

Clone

`Clone` trait 是 `Copy` 的超类trait，所以它也需要编译器生成实现。它被编译器实现给了以下类型：

- 实现了内置的 `Copy` trait 的类型（见上面）
- 由 `Clone` 类型的元素组成的元组
- 由 `Clone` 类型的元素组成的数组

Send

`Send` trait 表明这种类型的值可以安全地从一个线程发送给另一个线程。

Sync

`Sync` trait 表示在多个线程之间共享这种类型的值是安全的。必须为不可变静态(`static`)项中使用的所有类型实现此 trait。

Auto traits

`Send`、`Sync`、`Unpin`、`UnwindSafe` 和 `RefUnwindSafe` trait 都是*自动trait(auto traits)*。自动trait 具有特殊的属性。

对于给定类型，如果没有为其显式实现或否定实现(negative implementation)某自动trait，那么编译器就会根据以下规则去自动此为类型实现该自动trait：

- 如果 `T` 实现了某自动trait，那 `&T`、`&mut T`、`*const T`、`*mut T`、`[T; n]` 和 `[T]` 也会实现此自动trait。
- 函数项类型(function item types)和函数指针会自动实现这些自动trait。
- 如果结构体、枚举、联合体和元组它们的所有字段都实现了这些自动trait，则它们本身也会自动实现这些自动trait。
- 如果闭包捕获的所有变量的类型都实现了这些自动trait，那么闭包会自动实现这些自动trait。一个闭包通过共享引用捕获了一个 `T`，同时通过传值的方式捕获了一个 `U`，那么该闭包会自动实现 `&T` 和 `U` 所共同实现的那些自动trait。

对于泛型类型（上面的这些内置类型也算是建立在 `T` 上的泛型），如果泛型实现在当前已够用，则编译器不会再为其实现其他的自动trait，除非它们不满足必需的 trait约束。例如，标准库在 `T` 是 `Sync` 的地方都为 `&T` 实现了 `Send`；这意味着如果 `T` 是 `Send`，而不是 `Sync`，编译器就不会为 `&T` 自动实现 `Send`。

自动trait 也可以有否定实现，在标准库文档中表示为 `impl !AutoTrait for T`，它覆盖了自动实现。例如，`*mut T` 有一个关于 `Send` 的否定实现，所以 `*mut T` 不是 `Send` 的，即使 `T` 是。目前还没有稳下来的方法来指定其他类型的否定实现；目前否定实现只能在标准库里可以稳定使用。[3]

自动trait 可以作为额外的约束添加到任何 trait对象上，尽管通常我们见到的 trait对象的类型名上一般只显示一个 trait。例如，`Box<dyn Debug + Send + UnwindSafe>` 就是一个有效的类型。

`Sized`

`Sized` trait表明这种类型的尺寸在编译时是已知的；也就是说，它不是一个动态尺寸类型。类型参数默认是 `Sized` 的。`Sized` 总是由编译器自动实现，而不是由实现(implementation items)主动实现的。

[1] 这里是相对普通的借用/引用来说，普通的借用/引用对指向的内存位置不拥有所有权，所以无法从中移出值。

[2] 实现 `Drop` trait 的类型只能是使用移动语义(move)的类型。

[3] 标准库外使用需要打开特性 `#![feature(negative_impls)]`。

名称

keywords.md
commit: a989af055ff4fd7e1212754490fff72c3f7cc1be 本章译文最后维护日期：2020-1-25

*实体(entity)*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(path)][paths]。实体包括类型、程序项、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(declaration)是一种句法结构，它可以引入名称*来引用实体。实体的名称在相关*作用域(scope)*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：

- 程序项:
  - 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
    - `macro_rules` 声明 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
- 高阶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程序项之一来引入名称。

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

use-declarations.md
commit: 10da84befd0e79f5e1490912a02cb74368cd3f4a
本章译文最后维护日期：2021-1-25

*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，名称解析一章有讲到这些。

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
    - `macro_rules` 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[1]
    - 循环标签

如何清晰地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;
    }
}
```

无命名空间的命名实体

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，这取决于所导入的程序项类型。

[1] 目前，在同一个作用域中，标签和生命周期使用相同名称时，`rustc` 会警告出现命名重复，但编译器内部仍然会独立地区别对待它们。这是一个关于该语言可能扩展的未来兼容性警告。具体见PR #24162。

作用域

use-declarations.md
commit: eabdf09207bf3563ae96db9d576de0758c413d5d
本章译文最后维护日期：2021-1-24

注意：这是未来本章内容扩展的占位符。

预导入包

use-declarations.md
commit: 0ce54e64e3c98d99862485c57087d0ab36f40ef0
本章译文最后维护日期：2021-1-24

*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
- `macro_use` 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

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。

注意: 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。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。

```
extern crate alloc;
use alloc::rc::Rc;
```

Cargo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

`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 `macro_use` 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 `macro_use` 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，使用核心预导入包而不是标准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。

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

`macro_use`预导入包

`macro_use` 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。请参阅工具类属性一节，以了解更多细节。

`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。

版本差异: 在 2015版中，`no_implicit_prelude` 属性不会影响 `macro_use` 预导入包，从标准库导出的所有宏仍然包含在 `macro_use` 预导入包中。从 2018版开始，它也会禁止 `macro_use` 预导入包生效。

路径

paths.md
commit: 80241b46c68380735f05fb53bd99632b87ac2872
本章译文最后维护日期：2021-3-13

*路径*是一个或多个由命名空间限定符(`::`)*逻辑*分隔的路径段(path segments)组成的序列（译者注：如果只有一个段的话，`::` 不是必须的）。如果路径仅由一个路径段组成，则它引用局部控制域(control scope)内的程序项或变量。如果路径包含多个路径段，则总是引用程序项。

仅由标识符段组成的简单路径(simple paths)的两个示例：

```
x;
x::y::z;net
```

路径分类

Simple Paths

简单路径

句法

SimplePath :
`::`? *SimplePathSegment* (`::` *SimplePathSegment*)*

SimplePathSegment :
IDENTIFIER | `super` | `self` | `crate` | `$crate`

简单路径可用于可见性标记、属性、宏和 `use` 程序项中。示例：

```
#![allow(unused)]
fn main() {
use std::io::{self, Write};
mod m {
    #[clippy::cyclomatic_complexity = "0"]
    pub (in super) fn f1() {}
}
}
```

表达式中的路径

句法

PathInExpression :\net `::`? *PathExprSegment* (`::` *PathExprSegment*)*

PathExprSegment :
PathIdentSegment (`::` *GenericArgs*)?

PathIdentSegment :
IDENTIFIER | `super` | `self` | `Self` | `crate` | `$crate`

GenericArgs :
`<` `>`
| `<` (*GenericArg* `,`)* *GenericArg* `,`? `>`

GenericArg :\net *Lifetime* | *Type* | *GenericArgsConst* | *GenericArgsBinding*

GenericArgsConst :
BlockExpression
| *LiteralExpression*
| `-` *LiteralExpression*
| *SimplePathSegment*

GenericArgsBinding :
IDENTIFIER `=` *Type*

表达式中的路径允许指定带有泛型参数的路径。它们在各种表达式和模式中都有使用。

token `::` 必须在泛型参数的左尖括号（`<`）的前面，以避免和小于号操作符产生混淆。这就

是俗称的“涡轮鱼(turbofish)”句法。

```
(0..10).collect::<Vec<_>>();
Vec::<u8>::with_capacity(1024);
```

泛型参数的顺序被限制为生存期参数，然后是类型参数，然后是常量参数，再后是相应的约束。

常量实参必须用花括号括起来，除非它们是字面量或单段路径。

限定性路径

句法

QualifiedPathInExpression :
  QualifiedPathType (`::` *PathExprSegment*)+

QualifiedPathType :
  `<` *Type* (`as` *TypePath*)? `>`

QualifiedPathInType :
  QualifiedPathType (`::` *TypePathSegment*)+

完全限定性路径可以用来为 trait实现(trait implementations)消除路径歧义，也可以用来指定规范路径。用在指定具体类型时，它支持使用如下所示的类型句法：

```
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.
```

类型中的路径

Syntax

TypePath :
`::`? *TypePathSegment* (`::` *TypePathSegment*)*

TypePathSegment :
PathIdentSegment `::`? (*GenericArgs* | *TypePathFn*)?

TypePathFn :
`(` *TypePathFnInputs*? `)` (`->` *Type*)?

TypePathFnInputs :
Type (`,` *Type*)* `,`?

类型路径用于类型定义、trait约束(trait bound)、类型参数约束，以及限定性路径。

尽管在泛型参数之前允许使用 token `::`，但它不是必需的，因为在类型路径中不存在像 *PathInExpression* 句法中那样的歧义。

```
impl ops::Index<ops::Range<usize>> for S { /*...*/ }
fn i<'a>() -> impl Iterator<Item = ops::Example<'a>> {
    // ...net
}
type G = std::boxed::Box<dyn std::ops::FnOnce(isize) -> isize>;
```

路径限定符

路径可以被各种能改变其解析方式的前导限定符限定。

`::`

以 `::` 开头的路径被认为是全局路径，其中的路径首段在不同的版本中的解析方式有所不同。但路径中的每个标识符都必须解析为一个程序项。

> **版本差异**: 在2015版中，标识符解析从“create 根模块(crate root)”（2018版中表示为 `crate::`）开始，“create 根模块(crate root)”中包含了一系列不同的程序项，包括外部 crate、默认create（如 `std` 或 `core`），以及 crate下的各种顶层程序项（包括 `use` 导入）。
>
> 从 2018 版开始，以 `::` 开头的路径被解析为外部预导入包中的一个 crate。也就是说，其后必须跟一个 crate的名称。

```
mod a {
    pub fn foo() {}
}
mod b {
    pub fn foo() {
        ::a::foo(); // 调用 `a`'的 foo 函数
        // 在 Rust 2018 中, `::a` 会被解析为 crate `a`.
    }
}
```

`self`

`self` 表示路径是相对于当前模块的路径。`self` 仅可以用作路径的首段，不能有前置 `::`。

```
fn foo() {}
fn bar() {
    self::foo();
}
```

Self

`Self`（首字母大写）用于指代 trait 和实现中的类型。

`Self` 仅可以用作路径的首段，不能有前置 `::`。

```
trait T {
    type Item;
    const C: i32;
    // `Self` 将是实现 `T` 的任何类型。
    fn new() -> Self;
    // `Self::Item` 将是实现中指定的类型的别名。
    fn f(&self) -> Self::Item;
}
struct S;
impl T for S {
    type Item = i32;
    const C: i32 = 9;
    fn new() -> Self {           // `Self` 是类型 `S`.
        S
    }
    fn f(&self) -> Self::Item {  // `Self::Item` 是类型 `i32`.
        Self::C                  // `Self::C` 是常量值 `9`.
    }
}
```

super

`super` 在路径中被解析为父模块。它只能用于路径的前导段，可以置于 `self` 路径段之后。

```
mod a {
    pub fn foo() {}
}
mod b {
    pub fn foo() {
        super::a::foo(); // 调用 a'的 foo 函数
    }
}
```

`super` 可以在第一个 `super` 或 `self` 之后重复多次，以引用祖先模块。

```
mod a {
    fn foo() {}

    mod b {
        mod c {
            fn foo() {
                super::super::foo(); // 调用 a'的 foo 函数
                self::super::super::foo(); // 调用 a'的 foo 函数
            }
        }
    }
}
```

crate

`crate` 解析相对于当前 crate 的路径。crate 仅能用作路径的首段，不能有前置 `::`。

```
fn foo() {}
mod a {
    fn bar() {
        crate::foo();
    }
}
```

$crate

$crate 仅用在宏转码器(macro transcriber)中，且仅能用作路径首段，不能有前置 `::`。`$crate` 将被扩展为从定义宏的 crate 的顶层访问该 crate 中的各程序项的路径，而不用去考虑宏调用发生时所在的现场 crate。

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
```

规范路径

定义在模块或者实现中的程序项都有一个*规范路径*，该路径对应于其在其 crate 中定义的位置。在该路径之外，所有指向这些程序项的路径都是别名（路径）。规范路径被定义为一个*路径前缀*后跟一个代表程序项本身的那段路径段。

尽管实现所定义/实现的程序项有规范路径，但实现作为一个整体，自身没有规范路径。use声明也没有规范路径。块表达式中定义的程序项也没有规范路径。在没有规范路径的模块中定义的程序项也没有规范路径。在实现中定义的关联项(associated items)如果它指向没有规范路径的程序项——例如，实现类型(implementing type)、被实现的 trait、类型参数或类型参数上的约束——那它也没有规范路径。

模块的路径前缀就是该模块的规范路径。对于裸实现(bare implementations)来说，它里面的程序项的路径前缀是：它当前实现的程序项的规范路径，再用尖括号（`<>`）括把此路径括起来。对于 trait实现来说，它内部的程序项的路径前缀是：*当前实现的程序项的规范路径后跟一个 `as`，再后跟 trait 本身的规范路径，然后整个使用尖括号（`<>`）括起来。*

规范路径只在给定的 crate 中有意义。在 crate 之间没有全局命名空间，所以程序项的规范路径只在其 crate 中可标识。

```
// 注释解释了程序项的规范路径

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 {  // 译者注：这是一个裸实现
        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
        }
    }
}
```

名称解析

use-declarations.md
commit: eabdf09207bf3563ae96db9d576de0758c413d5d
本章译文最后维护日期：2021-1-24

注意：这是未来本章内容扩展的占位符。

可见性与隐私权

visibility-and-privacy.md
commit: f41afd4f69a7996ac66b01f75e333bccf43337f7
本章译文最后维护日期：2020-11-10

句法

Visibility :
`pub`
| `pub` `(` `crate` `)`
| `pub` `(` `self` `)`
| `pub` `(` `super` `)`
| `pub` `(` `in` *SimplePath* `)`

这两个术语经常互换使用，它们试图表达的是对“这个地方能使用这个程序项吗？”这个问题的回答。

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程序项。这些程序项就是我们前面提到的那些程序项之一（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 会检查每个程序项的使用，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。（译者注：这句提示可以翻译为：您使用了另一个模块的私有程序项，但不允许这样做。）

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 中的关联程序项默认为公有的；`pub` 枚举中的枚举变体也默认为公有的。当一个程序项被声明为 `pub` 时，它可以被认为是外部世界能以访问的。例如：

```
// 声明一个私有结构体
struct Foo;

// 声明一个带有私有字段的公有结构体
pub struct Bar {
    field: i32,
}

// 声明一个带有两个公有变体的公有枚举
pub enum State {
    PubliclyAccessibleState,
    PubliclyAccessibleState2,
}
```

依据程序项是公有的还是私有的，Rust 分两种情况来访问数据：

1. 如果某个程序项是公有的，那么如果可以从外部的某一模块 `m` 访问到该程序项的所有祖先模块，则一定可以从这个模块 `m` 中访问到该程序项。甚至还可以通过重导出来命名该程序项。具体见后文。

2. 如果某个程序项是私有的，则当前模块及当前模块的后代模块都可以访问它。

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。为了帮助理解，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。

在第二种情况下，我们提到了当前模块及其后代“可以访问”私有项，但是访问一个程序项的确切含义取决于该项是什么。例如，访问一个模块意味着要查看它的内部（以导入更多的程序项）；访问一个函数意味着它被调用了。此外，路径表达式和导入语句也被视为访问一个程序项，但只有当访问目标位于当前可见的作用域内时，它们才算是有效的数据访问。

下面是一段示例程序，它例证了上述三种情况：

```
// 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();
        }
    }
}
```

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。

`pub(in path)`, `pub(crate)`, `pub(super)`, and `pub(self)`

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程序项。声明形式的限制规则如下：

- `pub(in path)` 使一个程序项在提供的 `path` 中可见。`path` 必须是声明其可见性的程序项的祖先模块。
- `pub(crate)` 使一个程序项在当前 crate 中可见。
- `pub(super)` 使一个程序项对父模块可见。这相当于 `pub(in super)` 。
- `pub(self)` 使一个程序项对当前模块可见。这相当于 `pub(in self)` 或者根本不使用 `pub` 。

版本差异: 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。

这里是一些示例：

```
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() }
```

> **注意:** 此句法仅对程序项的可见性添加了另一个限制。它不能保证该程序项在指定作用域的所有部分都可见。要访问一个程序项，当前作用域内它的所有父项还是必须仍然可见。

重导出和可见性

Rust 允许使用指令 `pub use` 公开重导出程序项。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程序项。重导出本质上允许使用公有方式访问重导出的程序项的内部。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(privacy chain)”，而不是像通常那样通过命名空间层次结构来传递“隐私链”。

内存模型

memory-model.md
commit: 2d3085f1bab9d751e8a9b92c3b27c049ad23fdd7
本章译文最后维护日期：2020-11-2

Rust 还没有明确的内存模型。许多学者和行业专业人士正在研究各种提案，但就目前而言，这在该语言中仍是一个未明确定义的地方。

内存分配和生存期

memory-allocation-and-lifetime.md
commit: af1cf6d3ca3b7a8c434c142148742aa912e37c34
本章译文最后维护日期：2020-11-16

程序的*程序项*是那些函数、模块和类型，它们的值在编译时被计算出来，并且唯一地存储在 Rust 进程的内存映像中。程序项既不是动态分配的，也不是动态释放的。

*堆*是描述 box类型（译者注：box是一种堆分配形式，该堆分配返回一个指向该堆分配的内存地址的指针，后文称这个指针为 box指针或 box引用）的通用术语。堆分配的生存期取决于指向它的 box指针的生存期。由于 box指针本身可能被传入或传出栈帧，或者存储在堆中，因此堆分配可能比初始分配它们的栈帧存活的时间长。在堆分配的整个生存期内，该堆分配被保证驻留在堆中的单一位置 — 它永远不会因移动 box指针而重新分配内存地址。

变量

variables.md
commit: 79fcc6e4453919977b8b3bdf5aee71146c89217d
本章译文最后维护日期：2020-11-16

*变量*是栈帧里的一个组件，可以是具名函数参数、匿名的临时变量或具名局部变量。

局部变量（或*本地栈(stack-local)*分配）直接持有一个值，该值在栈内存中分配。该值是栈帧的一部分。

局部变量是不可变的，除非特别声明。例如：`let mut x = ...`。

函数参数是不可变的，除非用 `mut` 声明。关键字 `mut` 只应用于紧跟着它的那个参数。例如：`|mut x, y|` 和 `fn f(mut x: Box<i32>, y: Box<i32>)` 声明了一个可变变量 `x` 和一个不可变变量 `y`。

分配时不会初始化局部变量。此处一反常态的是在帧建立时，以未初始化状态分配整个帧值的局部变量。函数中的后续语句可以初始化局部变量，也可以不初始化局部变量。局部变量只有在通过所有可到达的控制流路径初始化后才能使用

在下面示例中，`init_after_if` 是在 `if` 表达式执行后被初始化的，而 `uninit_after_if` 不是，因为它没有在 `else` 分支里被初始化。

```
fn initialization_example() {
    let init_after_if: ();
    let uninit_after_if: ();

    if random_bool() {
        init_after_if = ();
        uninit_after_if = ();
    } else {
        init_after_if = ();
    }

    init_after_if; // ok
    // uninit_after_if; // 错误：使用可能未初始化的 `uninit_after_if`
}
```

链接

linkage.md
commit: 1426474192ecc9c13165c1d4772b26e8445f5734
本章译文最后维护日期：2021-4-23

注意：本节更多的是从编译器的角度来描述的，而不是语言。

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。本节将聚焦探索这些链接方法，关于本地库的更多信息请参阅 The Book 中的 FFI 相关章节。

在一个编译会话中，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（源码内部指定的）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=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=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=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。

请注意，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。

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：

1. 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。

2. 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。

3. 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）。

4. 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运行时和动态C运行时

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运行时和动态链接型C运行时。例如，目标 `x86_64-pc-windows-msvc` 和 `x86_64-unknown-linux-musl` 通常都带有C运行时，用户可以按自己的偏好去选择静态链接或动态链接到此运行时。编译器中所有的编译目标都有一个链接到 C运行时的默认模式。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运行时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，要启用静态运行时，应该执行：

```
rustc -C target-feature=+crt-static foo.rs
```

如果想动态链接到C运行时，应该执行：

```
rustc -C target-feature=-crt-static foo.rs
```

不支持在到 C运行时的链接类型之间切换的编译目标将忽略这个标志。建议检查生成的二进制文件，以确保在编译成功之后，如预期的那样链接了 C运行时。

crate 本身也可以检测如何链接 C运行时。例如，MSVC平台上的代码需要根据链接运行时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
```

[1] 译者理解此编译选项是将本地 Rust crate 和其上游依赖资源编译输出成一个库，供其他应用程序使用的。所以对当前编译而言，不能依赖还不存在的资源；又因为其他 Rust crate 的编译无法将静态库作为编译时依赖的上游资源，所以编译其他 Rust crate 的编译器也不会来链接此 `staticlib` 输出。

非安全性

unsafety.md
commit: b0e0ad6490d6517c19546b1023948986578fc378
本章译文最后维护日期：2020-11-2

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(`union`)]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

unsafe-functions.md
commit: 4a2bdf896cd2df370a91d14cb8ba04e326cd21db
本章译文最后维护日期：2020-11-16

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

unsafe-blocks.md
commit: 4a2bdf896cd2df370a91d14cb8ba04e326cd21db
本章译文最后维护日期：2020-11-16

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。

未定义的行为

behavior-considered-undefined.md
commit: 5524a17a22c94ad21ab28d545c316909ebda0e31
本章译文最后维护日期：2021-4-11

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(undefined behavior)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)*。

⚠ ***警告：*** 下面的列表并非详尽无遗地罗列了 Rust 中的未定义行为。而且对于在非安全代码中什么是明确不允许的，目前 Rust 还没有正式的语义模型，因此将来可能会有更多的行为被认为是不安全的。下面的列表仅仅是我们当前确定知晓的未定义行为。在编写非安全代码之前，请阅读 Rustonomicon。

- 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(noalias)模型(scoped noalias model)，除非 `&T` 内部包含 `UnsafeCell<U>` 类型。
- 修改不可变的数据。常量(`const`)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(compiler intrinsics)调用未定义行为。[1]
- 执行基于当前平台不支持的平台特性编译的代码（参见 `target_feature`）。
- 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(unwinding)。
- 产生非法值(invalid value)，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)（该虚函数表与该指针或引用所指向的实际动态 trait 相匹配）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**：`rustc`是使用还未稳定下来的属性`rustc_layout_scalar_valid_range_*`来验证这条规则的。

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）。

> **注意**：未定义行为影响整个程序。例如，在 C 中调用一个 C函数已经出现了未定义行为，这意味着包含此调用的整个程序都包含了未定义行为。如果 Rust 再通过 FFI 来调用这段 C程序/代码，那这段 Rust 代码也包含了未定义行为。反之亦然。因此 Rust 中的未定义行为会对任何其他通过 FFI 过来调用的代码造成不利影响。

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(allocation)的一部分（因此，它们都必须是*某次*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。

[1] 因为编译器内部函数(compiler intrinsics)很多都是和平台相关的，移植性差。

不被认为是非安全的行为

behavior-not-considered-unsafe.md
commit: a7473287cc6e2fb972165dc5a7ffd26dad1fc907
本章译文最后维护日期：2021-1-16

虽然程序员可能（应该）发现下列行为是不良的、意外的或错误的，但 Rust 编译器并不认为这些行为是*非安全的(unsafe)*。

死锁(Deadlocks)

内存和其他资源的泄漏(Leaks of memory and other resources)

退出而不调用析构函数(Exiting without calling destructors)

通过指针泄漏暴露随机基地址(Exposing randomized base addresses through pointer leaks)

整数溢出(Integer overflow)

如果程序包含算术溢出(arithmetic overflow)，则说明程序员犯了错误。在下面的讨论中，我们将区分算术溢出和包装算法(wrapping arithmetic)。前者是错误的，而后者是有意为之的。

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(two's complement overflow)约定来提供定义良好的（即使仍然被认为是错误的）溢出包装结果。

整型提供了一些固有方法(inherent methods)，允许程序员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(wrapping addition)。

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。

请参阅 RFC 560 以了解错误条件、基本原理以及有关整数溢出的更多详细信息。

逻辑错误(Logic errors)

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(unspecified)，但不会导致未定义行为(undefined behavior)。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。

常量求值

const_eval.md
commit: 8425f5bad3ac40e807e3f75f13b989944da28b62
本章译文最后维护日期：2021-4-5

常量求值是在编译过程中计算[表达式][expressions]结果的过程。（不是所有表达式都可以在编译时求值，也就是说）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(const)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/计算不会引起任何`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。
    - 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。8425f5bad3ac40e807e3f75f13b989944da28b62
  - 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, while 和 `while let` 表达式。
- if, `if let` 和 [匹配(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(initializer):
  - 常量项
  - 静态项
  - 枚举判别值
- 常量型泛型实参

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` 。

相反，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。
  - 常量上下文允许有限地使用常量型泛型形参。

应用程序二进制接口(ABI)

abi.md.md
commit: e773318a837092d7b5276bbeaf9fda06cca61cee
本章译文最后维护日期：2021-1-16

本节介绍影响 crate 编译输出的 ABI 的各种特性。

有关为导出函数(exporting functions)指定 ABI 的信息，请参阅*外部函数*。参阅[*外部块*]external blocks了解关于指定 ABI 来链接外部库的信息。

`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。

下面的示例显示了编译器在什么条件下在输出对象文件中保留静态(`static`)项。

```rust
// 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
}
```

```
$ 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

可以在任何程序项上使用 *`no_mangle`属性*来禁用标准名称符号名混淆(standard symbol name mangling)。禁用此功能后，此程序项的导出符号(symbol)名将直接是此程序项的原来的名称标识符。

此外，就跟`used`属性一样，此属性修饰的程序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。

```
#[no_mangle]
#[link_section = ".example_section"]
pub static VAR1: u32 = 1;
```

`export_name`属性

*`export_name`属性*指定将在函数或静态项上导出的符号的名称。它使用 *MetaNameValueStr*元项属性句法指定符号名。

```
#[export_name = "exported_symbol_name"]
pub fn name_in_rust() { }
```

Rust 运行时

abi.md.md
commit: f8e76ee9368f498f7f044c719de68c7d95da9972
本章译文最后维护日期：2020-11-3

本节介绍 Rust运行时的某些方面的特性。

`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 panic 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(dependency graph)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 {}
}
```

标准行为

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(subsystem)类别。它使用 *MetaNameValueStr*元项属性句法用 `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。

```
#![windows_subsystem = "windows"]
```

附录

附录：关于宏随集的二义性的形式化规范

macro-ambiguity.md
commit: 184b056086757e89d68f41c8c0e42721cb50a4a9
本章译文最后维护日期：2020-11-17

本文介绍了下述声明宏规则的正式规范。它们最初是在 RFC 550 中指定的（本文的大部分内容都是从其中复制过来的），并在后续的 RFC 中进行了进一步的展开。

定义和约定

- `macro`：宏，源代码中任何可调用的类似 `foo!(...)` 的东西。
- `MBE`：macro-by-example，声明宏，由 `macro_rules` 定义的宏。
- `matcher`：匹配器，`macro_rules`调用中一条规则的左侧部分，或其子部分(subportion)。（译者注：子部分的意思是匹配器可嵌套，可相互包含）
- `macro parser`：宏解释器，Rust 解析器中的一段程序，这段程序使用从所有命中的匹配器中推导出的文法规则来解析宏输入。
- `fragment`：匹配段，给定匹配器将接受（或“匹配”）的 Rust 句法对象。
- `repetition`：重复元，遵循正则重复模式的匹配段。
- `NT`：non-terminal，非终结符，可以出现在匹配器中的各种“元变量”或重复元匹配器，在声明宏(MBE)句法中用前导字符 `$` 标明。
- `simple NT`：简单NT，“元变量”类型的非终结符（下面会进一步讨论）。
- `complex NT`：复杂NT，重复元类型的非终结符，通过重复元操作符（`\*`，`+`，`?`）指定重复次数。
- `token`：匹配器中不可再细分的元素；例如，标识符、操作符、开/闭定界符*和*简单NT(simple NT)。
- `token tree`：token树，token树由 token(叶)、复杂NT 和子token树（token树的有限序列）组成的树形数据结构。
- `delimiter token`：定界符，一种用于划分一个匹配段的结束和下一个匹配段的开始的token。
- `separator token`：分隔符，复杂NT 中的可选定界符，用在重复元里以分隔元素。
- `separated complex NT`：带分隔符的复杂NT，分隔符是重复元的一部分的复杂NT。
- `delimited sequence`：有界序列，在序列的开始和结束处使用了适当的开闭定界符的token树。

- `empty fragment`：空匹配段，一种不可见的 Rust 句法对象，它分割各种 token，例如空白符(whitespace)或者（在某些词法上下文中的）空标记序列。
- `fragment specifier`：匹配段选择器，简单NT 中的后段标识符部分，指定 NT 接受哪种类型的匹配段。
- `language`：与上下文无关的语言。

示例：

```
macro_rules! i_am_an_mbe {
    (start $foo:expr $($i:ident),* end) => ($foo)
}
```

`(start $foo:expr $($i:ident),\* end)` 是一个匹配器(matcher)。整个匹配器是一段有界字符序列（使用开闭定界符 `(` 和 `)` 界定），`$foo` 和 `$i` 是简单NT(simple NT)，`expr` 和 `ident` 是它们各自的匹配段选择器(fragment specifiers)。

`$(i:ident),\*` *也*是一个 NT；它是一个复杂NT，匹配那些被逗号分隔成的标识符类型的重复元。`,` 是这个复杂NT 的分隔符；它出现在匹配段的每对元素（如果有的话）之间。

复杂NT 的另一个例子是 `$(hi $e:expr ;)+`，它匹配 `hi <expr>; hi <expr>; ...` 这种格式的代码，其中 `hi <expr>;` 至少出现一次。注意，这种复杂NT 没有专用的分隔符。

(请注意，Rust 解析器会确保这类有界字符序列始终具有正确的 token树的嵌套结构以及开/闭定界符的正确匹配。)

下面，我们将用变量“M”表示匹配器，变量“t”和“u”表示任意单一 token，变量“tt”和“uu”表示任意 token树。（使用“tt”确实存在潜在的歧义，因为它的额外角色是一个匹配段选择器；但不用太担心，因为从上下文中，可以很清楚地看出哪个解释更符合语义）

用“SEP”代表分隔符，“OP”代表重复元运算符 `\*`, `+`, 和 `?` “OPEN”/“CLOSE”代表包围定界字符序列的 token对（例如 `[` 和 `]` ）。

用希腊字母 "α" "β" "γ" "δ" 代表潜在的空token树序列。（注意没有使用希腊字母 "ε"，"ε" (epsilon)在此表示形式中代表一类特殊的角色，不代表 token树序列。）

- 这种希腊字母约定通常只是在需要展现一段字符序列的技术细节时才被引入；特别是，当我们希望*强调*我们操作的是一个 token树序列时，我们将对该序列使用表义符 "tt ..."，而不是一个希腊字母。

请注意，匹配器仅仅是一个 token树。如前所述，“简单NT”是一个元变量类型的 NT；因此，这是一个非重复元。例如，`$foo:ty` 是一个简单NT，而 `$($foo:ty)+` 是一个复杂NT。

还请注意，在这种形式体系的上下文中，术语“token”通常*包括*简单NT。

最后，读者要记住，根据这种形式体系的定义，简单NT 不会匹配空匹配段，因此也没有 token 会匹配 Rust句法的空匹配段。（因此，能够匹配空匹配段的 NT *唯有*复杂NT。）但这还不是全部事实，因为 `vis` 匹配器可以匹配空匹配段。因此，为了达到这种形式体系自洽统一的目的，我们将把 `$v:vis` 看作是 `$($v:vis)?`，来让匹配器匹配一个空匹配段。

匹配器的不变式

为了有效，匹配器必须满足以下三个不变式。注意其中 FIRST 和 FOLLOW 的定义将在后面进行描述。

1. 对于匹配器 `M` 中的任意两个连续的 token树序列（即 `M = ... tt uu ...`），并且 `uu ...` 非空，必有 FOLLOW(`... tt`) ∪ {ε} ⊇ FIRST(`uu ...`)。
2. 对于匹配器中任何带分隔符的复杂NT， `M = ... $(tt ...) SEP OP ...`，必有 `SEP` ∈ FOLLOW(`tt ...`)
3. 对于匹配器中不带分隔符的复杂NT， `M = ... $(tt ...) OP ...`，如果 OP = `\*` 或 `+`，必有 FOLLOW(`tt ...`) ⊇ FIRST(`tt ...`)。

第一个不变式表示，无论匹配器后出现什么 token（如果有的话），它都必须出现在先决随集(predetermined follow set)中的某个地方。这将确保合法的宏定义将继续对 `... tt` 的结束和 `uu ...` 的开始执行相同的判定(determination)，即使将来语言中添加了新的句法形式。 The first invariant says that whatever actual token that comes after a matcher, if any, must be somewhere in the predetermined follow set. This ensures that a legal macro definition will continue to assign the same determination as to where `... tt` ends and `uu ...` begins, even as new syntactic forms are added to the language.

第二个不变式表示一个带分隔符的复杂NT 必须使用一个分隔符，它是 NT 的内部内容的先决随集的一部分。这将确保合法的宏定义将继续将输入匹配段解析成相同的定界字符序列 `tt ...`，即使在将来语言中添加了新的语法形式。 The second invariant says that a separated complex NT must use a separator token that is part of the predetermined follow set for the internal contents of the NT. This ensures that a legal macro definition will continue to parse an input fragment into the same delimited sequence of `tt ...`'s, even as new syntactic forms are added to the language.

第三个不变式说的是，当我们有一个复杂NT，它可以匹配同一字符序列的两个或多个副本，并且两者之间没有分隔符，那么根据第一个不变式，它们必须可以放在一起。这个不变式还要求它们是非空的，这消除了可能出现的歧义。 The third invariant says that when we have a complex NT that can match two or more copies of the same thing with no separation in

between, it must be permissible for them to be placed next to each other as per the first invariant. This invariant also requires they be nonempty, which eliminates a possible ambiguity.

注意：由于历史疏忽和对行为的严重依赖，第三个不变式目前没有被执行。目前还没有决定下一步该怎么做。不遵循这个不变式的宏可能会在未来的 Rust版本中失效。参见跟踪问题
NOTE：The third invariant is currently unenforced due to historical oversight and significant reliance on the behaviour. It is currently undecided what to do about this going forward. Macros that do not respect the behaviour may become invalid in a future edition of Rust. See the tracking issue.

FIRST and FOLLOW, informally

非正式的 FIRST集合和 FOLLOW集合定义

给定匹配器 M 映射到三个集合：FIRST(M)，LAST(M) 和 FOLLOW(M)。 A given matcher M maps to three sets：FIRST(M), LAST(M) and FOLLOW(M).

这三个集合中的每一个都是由一组 token 组成的。FIRST(M) 和 LAST(M) 也可能包含一个可区分的非token元素 ε ("epsilon")，这表示 M 可以匹配空匹配段。（但是 FOLLOW(M) 始终只是一组 token。） Each of the three sets is made up of tokens. FIRST(M) and LAST(M) may also contain a distinguished non-token element ε ("epsilon"), which indicates that M can match the empty fragment. (But FOLLOW(M) is always just a set of tokens.)

非正式定义(Informally)：

- FIRST(M)：收集匹配段与 M 匹配时可能首先使用的 token。collects the tokens potentially used first when matching a fragment to M.
- LAST(M)：收集匹配段与 M 匹配时可能最后使用的 token。collects the tokens potentially used last when matching a fragment to M.
- FOLLOW(M)：允许紧跟在由 M 匹配的某个匹配段之后的 token集合。the set of tokens allowed to follow immediately after some fragment matched by M.

  换言之：t ∈ FOLLOW(M) 当且仅当存在（可能为空的）token序列 α、β、γ、δ，其中：

  - M 匹配 β,
  - t 与 γ 匹配，并且
  - 连结 α β γ δ 是一段可解析的 Rust程序。 In other words：t ∈ FOLLOW(M) if and

only if there exists (potentially empty) token sequences α, β, γ, δ where:
  - M matches β,
  - t matches γ, and
  - The concatenation α β γ δ is a parseable Rust program.

我们使用简写的 ANYTOKEN 来表示所有 token（包括简单NT）的集合。例如，如果任何 token 在匹配器 M 之后都是合法的，那么 FOLLOW(M) = ANYTOKEN。 We use the shorthand ANYTOKEN to denote the set of all tokens (including simple NTs). For example, if any token is legal after a matcher M, then FOLLOW(M) = ANYTOKEN.

（为了加深对上述非正式定义描述的理解，读者在阅读正式定义之前，可以先在这里读一遍后面 关于 FIRST 和 LAST 的示例。） (To review one's understanding of the above informal descriptions, the reader at this point may want to jump ahead to the examples of FIRST/LAST before reading their formal definitions.)

FIRST, LAST

下面是对 FIRST 和 LAST 的正式归纳定义(formal inductive definitions)。

“A∪B”表示集合并集，“A∩B”表示集合交集，“A\B”表示集合差集（即存在于A中，且不存在于B中的所有元素的集合）。

FIRST

FIRST(M) 是通过对序列 M 及其第一个 token树(如果有的话)的结构进行案例分析来定义的: FIRST(M) is defined by case analysis on the sequence M and the structure of its first token-tree (if any):

- 如果 M 为空序列，则 FIRST(M) = { ε }，if M is the empty sequence, then FIRST(M) = { ε },
- 如果 M 以 token t 开始，则 FIRST(M) = { t }，if M starts with a token t, then FIRST(M) = { t },

  （注意:这涵盖了这样一种情况：M 以一个定界的token树序列开始，`M = OPEN tt ... CLOSE ...`，此时 `t = OPEN`，因此 FIRST(M) = { `OPEN` }。） (Note：this covers the case where M starts with a delimited token-tree sequence, `M = OPEN tt ... CLOSE ...`, in which case `t = OPEN` and thus FIRST(M) = { `OPEN` }.)

  （注意：这主要依赖于没有简单NT与空匹配段匹配这一特性。） (Note：this critically

relies on the property that no simple NT matches the empty fragment.)

- 否则，M 是一个以复杂NT开始的token树序列：`M = $( tt ... ) OP α`，或 `M = $( tt ... ) SEP OP α`，(其中 `α` 是匹配器其余部分的token树序列(可能是空的))。Otherwise, M is a token-tree sequence starting with a complex NT：`M = $( tt ... ) OP α`, or `M = $( tt ... ) SEP OP α`, (where `α` is the (potentially empty) sequence of token trees for the rest of the matcher).
    - Let SEP_SET(M) = { SEP } 如果存在 SEP 且 ε ∈ FIRST(`tt ...`)；否则 SEP_SET(M) = {}。
- Let ALPHA_SET(M) = FIRST(`α`) if OP = `\*` or `?` and ALPHA_SET(M) = {} if OP = `+`.
- FIRST(M) = (FIRST(`tt ...`) \ {ε}) ∪ SEP_SET(M) ∪ ALPHA_SET(M).

复杂NT 的定义值得商榷。SEP_SET(M) 定义了分隔符可能是 M 的第一个有效token的可能性，当定义了分隔符且重复匹配段可能为空时，就会发生这种情况。ALPHA_SET(M)定义了复杂NT可能为空的可能性，这意味着 M 的第一个有效token集合是后继token树序列 `α` 。当使用了操作符 `\*` 或 `?` 时，这种情况下可能没有重复元。理论上，如果 `+` 与一个可能为空的重复匹配段一起使用，也会出现这种情况，但是第三个不变式禁止这样做。 The definition for complex NTs deserves some justification. SEP_SET(M) defines the possibility that the separator could be a valid first token for M, which happens when there is a separator defined and the repeated fragment could be empty. ALPHA_SET(M) defines the possibility that the complex NT could be empty, meaning that M's valid first tokens are those of the following token-tree sequences `α`. This occurs when either `\*` or `?` is used, in which case there could be zero repetitions. In theory, this could also occur if `+` was used with a potentially-empty repeating fragment, but this is forbidden by the third invariant.

From there, clearly FIRST(M) can include any token from SEP_SET(M) or ALPHA_SET(M), and if the complex NT match is nonempty, then any token starting FIRST(`tt ...`) could work too. The last piece to consider is ε. SEP_SET(M) and FIRST(`tt ...`) \ {ε} cannot contain ε, but ALPHA_SET(M) could. Hence, this definition allows M to accept ε if and only if ε ∈ ALPHA_SET(M) does. This is correct because for M to accept ε in the complex NT case, both the complex NT and α must accept it. If OP = `+`, meaning that the complex NT cannot be empty, then by definition ε ∉ ALPHA_SET(M). Otherwise, the complex NT can accept zero repetitions, and then ALPHA_SET(M) = FOLLOW(`α`). So this definition is correct with respect to \varepsilon as well.

LAST

LAST(M), defined by case analysis on M itself (a sequence of token-trees):

- if M is the empty sequence, then LAST(M) = { ε }
- if M is a singleton token t, then LAST(M) = { t }
- if M is the singleton complex NT repeating zero or more times, `M = $( tt ... ) *`, or `M = $( tt ... ) SEP *`
  - Let sep_set = { SEP } if SEP present; otherwise sep_set = {}.
  - if ε ∈ LAST(`tt ...`) then LAST(M) = LAST(`tt ...`) ∪ sep_set
  - otherwise, the sequence `tt ...` must be non-empty; LAST(M) = LAST(`tt ...`) ∪ {ε}.
- if M is the singleton complex NT repeating one or more times, `M = $( tt ... ) +`, or `M = $( tt ... ) SEP +`
  - Let sep_set = { SEP } if SEP present; otherwise sep_set = {}.
  - if ε ∈ LAST(`tt ...`) then LAST(M) = LAST(`tt ...`) ∪ sep_set
  - otherwise, the sequence `tt ...` must be non-empty; LAST(M) = LAST(`tt ...`)
- if M is the singleton complex NT repeating zero or one time, `M = $( tt ...) ?`, then LAST(M) = LAST(`tt ...`) ∪ {ε}.
- if M is a delimited token-tree sequence `OPEN tt ... CLOSE`, then LAST(M) = { `CLOSE` }.
- if M is a non-empty sequence of token-trees `tt uu ...`,
  - If ε ∈ LAST(`uu ...`), then LAST(M) = LAST(`tt`) ∪ (LAST(`uu ...`) \ { ε }).
  - Otherwise, the sequence `uu ...` must be non-empty; then LAST(M) = LAST(`uu ...`).

关于 FIRST 和 LAST 的示例

下面是一些关于 FIRST 和 LAST 的例子。（请特别注意，特殊元素 ε 是如何根据输入匹配段之间的相互作用来引入和消除的。） Below are some examples of FIRST and LAST. (Note in

particular how the special ε element is introduced and eliminated based on the interaction between the pieces of the input.)

我们的第一个例子以树状结构呈现，以详细说明匹配器的分析是如何组成的。（一些较简单的子树已被删除。） Our first example is presented in a tree structure to elaborate on how the analysis of the matcher composes. (Some of the simpler subtrees have been elided.)

```
INPUT:  $(  $d:ident   $e:expr   );*    $( $( h )* );*    $( f ; )+   g
            ~~~~~~~~   ~~~~~~~               ~
                |         |                  |
FIRST:   { $d:ident }  { $e:expr }          { h }


INPUT:  $(  $d:ident   $e:expr   );*    $( $( h )* );*    $( f ; )+
            ~~~~~~~~~~~~~~~~~~             ~~~~~~~          ~~~
                      |                       |              |
FIRST:          { $d:ident }              { h, ε }        { f }

INPUT:  $(  $d:ident   $e:expr   );*    $( $( h )* );*    $( f ; )+   g
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~    ~~~~~~~~~~~~~~    ~~~~~~~~~   ~
                      |                       |              |       |
FIRST:        { $d:ident, ε }            {  h, ε, ;  }      { f }   { g }


INPUT:  $(  $d:ident   $e:expr   );*    $( $( h )* );*    $( f ; )+   g
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                    |
FIRST:                      { $d:ident, h, ;,  f }
```

Thus:

- FIRST(`$($d:ident $e:expr );* $( $(h)* );* $( f ;)+ g`) = { `$d:ident`, `h`, `;`, `f` }

Note however that:

- FIRST(`$($d:ident $e:expr );* $( $(h)* );* $($( f ;)+ g)*`) = { `$d:ident`, `h`, `;`, `f`, ε }

Here are similar examples but now for LAST.

- LAST(`$d:ident $e:expr`) = { `$e:expr` }
- LAST(`$( $d:ident $e:expr );*`) = { `$e:expr`, ε }
- LAST(`$( $d:ident $e:expr );* $(h)*`) = { `$e:expr`, ε, `h` }
- LAST(`$( $d:ident $e:expr );* $(h)* $( f ;)+`) = { `;` }

- LAST(`$( $d:ident $e:expr );* $(h)* $( f ;)+ g`) = { `g` }

FOLLOW(M)

Finally, the definition for FOLLOW(M) is built up as follows. pat, expr, etc. represent simple nonterminals with the given fragment specifier.

- FOLLOW(pat) = { `=>` , `,` , `=` , `|` , `if` , `in` }`.
- FOLLOW(expr) = FOLLOW(stmt) = { `=>` , `,` , `;` }`.
- FOLLOW(ty) = FOLLOW(path) = { `{` , `[` , `,` , `=>` , `:` , `=` , `>` , `>>` , `;` , `|` , `as` , `where` , block nonterminals}.
- FOLLOW(vis) = { `,` l any keyword or identifier except a non-raw `priv` ; any token that can begin a type; ident, ty, and path nonterminals}.
- FOLLOW(t) = ANYTOKEN for any other simple token, including block, ident, tt, item, lifetime, literal and meta simple nonterminals, and all terminals.
- FOLLOW(M), for any other M, is defined as the intersection, as t ranges over (LAST(M) \ {ε}), of FOLLOW(t).

The tokens that can begin a type are, as of this writing, { `(` , `[` , `!` , `\*` , `&` , `&&` , `?` , lifetimes, `>` , `>>` , `::` , any non-keyword identifier, `super` , `self` , `Self` , `extern` , `crate` , `$crate` , `_` , `for` , `impl` , `fn` , `unsafe` , `typeof` , `dyn` }, although this list may not be complete because people won't always remember to update the appendix when new ones are added.

Examples of FOLLOW for complex M:

- FOLLOW(`$( $d:ident $e:expr )\*`) = FOLLOW(`$e:expr`)
- FOLLOW(`$( $d:ident $e:expr )\* $(;)\*`) = FOLLOW(`$e:expr`) ∩ ANYTOKEN = FOLLOW(`$e:expr`)
- FOLLOW(`$( $d:ident $e:expr )\* $(;)\* $( f |)+`) = ANYTOKEN

Examples of valid and invalid matchers

With the above specification in hand, we can present arguments for why particular matchers are legal and others are not.

- `($ty:ty < foo ,)` ： illegal, because FIRST(`< foo ,`) = { `<` } ⊈ FOLLOW(`ty`)
- `($ty:ty , foo <)` ： legal, because FIRST(`, foo <`) = { `,` } is ⊆ FOLLOW(`ty`).
- `($pa:pat $pb:pat $ty:ty ,)` ： illegal, because FIRST(`$pb:pat $ty:ty ,`) = { `$pb:pat` } ⊈ FOLLOW(`pat`), and also FIRST(`$ty:ty ,`) = { `$ty:ty` } ⊈ FOLLOW(`pat`).
- `( $($a:tt $b:tt)* ; )` ： legal, because FIRST(`$b:tt`) = { `$b:tt` } is ⊆ FOLLOW(`tt`) = ANYTOKEN, as is FIRST(`;`) = { `;` }.
- `( $($t:tt),* , $(t:tt),* )` ： legal, (though any attempt to actually use this macro will signal a local ambiguity error during expansion).
- `($ty:ty $(; not sep)* -)` ： illegal, because FIRST(`$(; not sep)* -`) = { `;` , `-` } is not in FOLLOW(`ty`).
- `($($ty:ty)-+)` ： illegal, because separator `-` is not in FOLLOW(`ty`).
- `($($e:expr)*)` ： illegal, because expr NTs are not in FOLLOW(expr NT).

影响来源

influences.md
commit: dc3808468e37ff4c1f663d26c491a3549a42c201
本章译文最后维护日期：2020-11-3

Rust 并不是一种极端原创的语言，它的设计元素来源广泛。下面列出了其中一些（包括已经删除的）：

- SML，OCaml：代数数据类型、模式匹配、类型推断、分号语句分隔
- C++：引用，RAII，智能指针，移动语义，单态化(monomorphization)，内存模型
- ML Kit, Cyclone：基于区域的内存管理(region based memory management)
- Haskell (GHC)：类型属性(typeclasses), 类型簇(type families)
- Newsqueak, Alef, Limbo：通道，并发
- Erlang：消息传递，线程失败，~~链接线程失败~~，~~轻量级并发~~
- Swift：可选绑定(optional bindings)
- Scheme：卫生宏(hygienic macros)
- C#：属性
- Ruby：闭包句法，~~块句法~~
- NIL, Hermes：~~typestate~~
- Unicode Annex #31：标识符和模式句法

术语表

glossary.md
commit: c28dfe483375849678793dbe86c69a1953f3bb00
本章译文最后维护日期：2021-02-21

抽象句法树

“抽象句法树”，或“AST”，是编译器编译程序时，程序结构的中间表示形式。
An ‘abstract syntax tree’, or ‘AST’, is an intermediate representation of the structure of the program when the compiler is compiling it.

对齐量

值的对齐量指定值的首选起始存储地址。对齐量总是2的幂次。值的引用必须是对齐的。更多。
The alignment of a value specifies what addresses values are preferred to start at. Always a power of two. References to a value must be aligned. More.

元数

元数是指函数或运算符接受的参数个数。例如，`f(2, 3)` 和 `g(4, 6)` 的元数为2，而 `h(8, 2, 6)` 的元数为3。`!` 运算符的元数为1。
Arity refers to the number of arguments a function or operator takes. For some examples, `f(2, 3)` and `g(4, 6)` have arity 2, while `h(8, 2, 6)` has arity 3. The `!` operator has arity 1.

数组

数组，有时也称为固定大小数组或内联数组，是描述关于元素集合的值，每个元素都可由程序在运行时计算的索引选择。内存模型上，数组占用连续的内存区域。
An array, sometimes also called a fixed-size array or an inline array, is a value describing a collection of elements, each selected by an index that can be computed at run time by the program. It occupies a contiguous region of memory.

关联程序项/关联项

关联程序项是与另一个程序项关联的程序项。关联程序项在 trait 中声明，在实现中定义。只有函数、常量和类型别名可以作为关联程序项。它与自由程序项形成对比。
An associated item is an item that is associated with another item. Associated items are defined in implementations and declared in traits. Only functions, constants, and type aliases can be associated. Contrast to a free item.

blanket实现

指为无覆盖类型实现的任何实现。`impl<T> Foo for T`、`impl<T> Bar<T> for T`、`impl<T> Bar<Vec<T>> for T`、和 `impl<T> Bar<T> for Vec<T>` 被认为是 blanket实现。但是，`impl<T> Bar<Vec<T>> for Vec<T>` 不被认为是，因为这个 `impl` 中所有的 `T` 的实例都被 `Vec` 覆盖。
Any implementation where a type appears uncovered. `impl<T> Foo for T`, `impl<T> Bar<T> for T`, `impl<T> Bar<Vec<T>> for T`, and `impl<T> Bar<T> for Vec<T>` are considered blanket impls. However, `impl<T> Bar<Vec<T>> for Vec<T>` is not a blanket impl, as all instances of `T` which appear in this `impl` are covered by `Vec`.

约束

约束是对类型或 trait 的限制。例如，如果在函数的形数上设置了约束，则传递给该函数的实参的类型必须遵守该约束。
Bounds are constraints on a type or trait. For example, if a bound is placed on the argument a function takes, types passed to that function must abide by that constraint.

组合算子

组合子是高阶函数，它的参数全是函数或之前定义的组合子。组合子利用这些函数或组合子返回的结果作为入参进行进一步的逻辑计算和输出。组合子可用于以模块化的方式管理控制流。
Combinators are higher-order functions that apply only functions and earlier defined combinators to provide a result from its arguments. They can be used to manage control flow in a modular fashion.

分发

分发是一种机制，用于确定涉及到多态性时实际运行的是哪个版本的代码。分发的两种主要形式是静态分发和动态分发。虽然 Rust 支持静态分发，但它也通过一种称为 trait对象的机制支持动态分发。
Dispatch is the mechanism to determine which specific version of code is actually run when it involves polymorphism. Two major forms of dispatch are static dispatch and dynamic dispatch. While Rust favors static dispatch, it also supports dynamic dispatch through a mechanism called 'trait objects'.

动态尺寸类型

动态尺寸类型(DST)是一种没有静态已知尺寸或对齐量的类型。
A dynamically sized type (DST) is a type without a statically known size or alignment.

实体

*实体(entity)*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(path)。实体包括类型、程序项、泛型参数、变量绑定、循环标签、生存期、字段、属性和lints。
An *entity* is a language construct that can be referred to in some way within the source program, usually via a path. Entities include types, items, generic parameters, variable bindings, loop labels, lifetimes, fields, attributes, and lints.

表达式

表达式是值、常量、变量、运算符/操作符和函数的组合，计算/求值结果为单个值，有或没有副作用都有可能。
比如，`2 + (3 * 4)` 是一个返回值为14的表达式。
An expression is a combination of values, constants, variables, operators and functions that evaluate to a single value, with or without side-effects.
For example, `2 + (3 * 4)` is an expression that returns the value 14.

自由程序项

不是任何实现的成员的程序项，如*自由函数*或*自由常量*。自由程序项是与关联程序项相对的概念。
An item that is not a member of an implementation, such as a *free function* or a *free const*.

Contrast to an associated item.

基础性trait

基础性trait 就是如果为现有的类型实现它，就会为该类型带来突破性改变的 trait。比如 `Fn` 和 `Sized` 就是基础性trait。
A fundamental trait is one where adding an impl of it for an existing type is a breaking change. The `Fn` traits and `Sized` are fundamental.

基本类型构造器

基本类型构造器是这样一种类型，在它之上实现一个 blanket实现是一个突破性的改变。`&`、`&mut`、`Box`、和 `Pin` 是基本类型构造器。
如果任何时候 `T` 都被认为是本地类型，那 `&T`、`&mut T`、`Box<T>`、和 `Pin<T>` 也被认为是本地类型。基本类型构造器不能覆盖其他类型。任何时候使用术语“有覆盖类型”时，都默认把 `&T`、`&mut T`、`Box<T>`、和 `Pin<T>` 排除在外。
A fundamental type constructor is a type where implementing a blanket implementation over it is a breaking change. `&`, `&mut`, `Box`, and `Pin` are fundamental.
Any time a type `T` is considered local, `&T`, `&mut T`, `Box<T>`, and `Pin<T>` are also considered local. Fundamental type constructors cannot cover other types. Any time the term "covered type" is used, the `T` in `&T`, `&mut T`, `Box<T>`, and `Pin<T>` is not considered covered.

Inhabited

如果类型具有构造函数，因此可以实例化，则该类型是 inhabited。inhabited 类型不是“空的”，因为可以有类型对应的值。与之相对的是 Uninhabited。
A type is inhabited if it has constructors and therefore can be instantiated. An inhabited type is not "empty" in the sense that there can be values of the type. Opposite of Uninhabited.

固有实现

单独标称类型上的实现，注意关键字 `impl` 后直接是标称类型，而非 trait-标称类型对(trait-type pair)上的实现。更多。

An implementation that applies to a nominal type, not to a trait-type pair. More.

固有方法

在固有实现中而不是在 trait实现中定义的方法。
A method defined in an inherent implementation, not in a trait implementation.

初始化

如果一个变量已经被分配了一个值，并且此值还没有被移动走，那此变量就被初始化了。对此变量而言，它会假设它之外的所有其他内存位置都未初始化。只有非安全Rust 可以在不初始化的情况下开辟内存新区域。
A variable is initialized if it has been assigned a value and hasn't since been moved from. All other memory locations are assumed to be uninitialized. Only unsafe Rust can create such a memory without initializing it.

本地 trait

本地 trait 是在当前 crate 中定义的 `trait`。它可以在模块局部定义，也可以是依附于其他类型参数而定义。给定 `trait Foo<T, U>`， `Foo` 总是本地的，不管替代 `T` 和 `U` 的类型是什么。
A `trait` which was defined in the current crate. A trait definition is local or not independent of applied type arguments. Given `trait Foo<T, U>`, `Foo` is always local, regardless of the types substituted for `T` and `U`.

Turbofish

表达式中带有泛型参数的路径必须在左尖括号前加上一个 `::`。 这种为表达泛型而结合起来形式（`::<>`）看起来有些像一条鱼。 因此，在口头上就被称为 turbofish句法。
Paths with generic parameters in expressions must prefix the opening brackets with a `::`. Combined with the angular brackets for generics, this looks like a fish `::<>`. As such, this syntax is colloquially referred to as turbofish syntax.

例如：

```
let ok_num = Ok::<_, ()>(5);
let vec = [1, 2, 3].iter().map(|n| n * 2).collect::<Vec<_>>();
```

这里必须使用 `::` 前缀，以便在逗号分隔的列表中进行多次比较时消除泛型路径可能的歧义。参见 the bastion of the turbofish 中因为没有此前缀的而引起歧义的示例。
This `::` prefix is required to disambiguate generic paths with multiple comparisons in a comma-separate list. See the bastion of the turbofish for an example where not having the prefix would be ambiguous.

本地类型

指在当前 crate 中定义的 `struct`、`enum`、或 `union` 。本地类型不会受到类型参数的影响。`struct Foo` 被认为是本地的，但 `Vec<Foo>` 不是。`LocalType<ForeignType>` 是本地的。类型别名不影响本地性。
A `struct`, `enum`, or `union` which was defined in the current crate. This is not affected by applied type arguments. `struct Foo` is considered local, but `Vec<Foo>` is not. `LocalType<ForeignType>` is local. Type aliases do not affect locality.

名称

*名称*是一个指向实体的标识符或生存期或循环标签。*名称绑定(name binding)*是指实体声明时引入了与该实体相关联的标识符或标签。路径、标识符和标签用于引用实体。
A *name* is an identifier or lifetime or loop label that refers to an entity. A *name binding* is when an entity declaration introduces an identifier or label associated with that entity. Paths, identifiers, and labels are used to refer to an entity.

名称解析

*名称解析*是将路径、标识符、标签和实体(entity)声明绑定在一起的的编译过程。
Name resolution is the compile-time process of tying paths, identifiers, and labels to entity declarations.

命名空间

*命名空间*是基于名称所引用的实体类型的声明名称的逻辑分组。命名空间让在一个命名空间中出现的名称不会与另一个命名空间中的相同名称冲突。
A *namespace* is a logical grouping of declared names based on the kind of entity the name refers to. Namespaces allow the occurrence of a name in one namespace to not conflict with the same name in another namespace.
在命名空间中，名称统一组织在一个层次结构中，层次结构的每一层都有自己的命名实体集合。
Within a namespace, names are organized in a hierarchy, where each level of the hierarchy has its own collection of named entities.

标称类型

可用路径直接引用的类型。具体来说就是枚举(`enum`)、结构体(`struct`)、联合体(`union`)和trait对象。
Types that can be referred to by a path directly. Specifically enums, structs, unions, and trait objects.

对象安全 trait

可以用作 trait对象的 trait。只有遵循特定规则的 trait 才是对象安全的。
Traits that can be used as trait objects. Only traits that follow specific rules are object safe.

路径

*路径*是一个或多个路径段组成的序列，用于引用当前作用域或某命名空间层次结构中的实体。
A *path* is a sequence of one or more path segments used to refer to an entity in the current scope or other levels of a namespace hierarchy.

预加载模块集/预导入包

预加载模块集，或者 Rust 预加载模块集，是一个会被导入到每个 crate 中的每个模块的小型程序项集合（其中大部分是 trait）。trait 在预加载模块集中很普遍。
Prelude, or The Rust Prelude, is a small collection of items - mostly traits - that are imported into every module of every crate. The traits in the prelude are pervasive.

作用域

*作用域*是源文本的一个区域，在该区域中可以直接使用其名称来引用在其下命名的命名实体。
A *scope* is the region of source text where a named entity may be referenced with that name.

检验对象\检验对象表达式

检验对象是在匹配(`match`)表达式和类似的模式匹配结构上匹配的表达式。例如，在 `match x { A => 1, B => 2 }` 中，表达式 `x` 是 scrutinee。
A scrutinee is the expression that is matched on in `match` expressions and similar pattern matching constructs. For example, in `match x { A => 1, B => 2 }`, the expression `x` is the scrutinee.

类型尺寸/尺寸

值的尺寸有两个定义。
第一个是必须分配多少内存来存储这个值。
第二个是它是在具有该项类型的数组中连续元素之间的字节偏移量。
它是对齐量的整数倍数，包括零倍。尺寸会根据编译器版本(进行新的优化时)和目标平台(类似于 `usize` 在不同平台上的变化)而变化。
查看更多. The size of a value has two definitions.
The first is that it is how much memory must be allocated to store that value.
The second is that it is the offset in bytes between successive elements in an array with that item type. It is a multiple of the alignment, including zero. The size can change depending on compiler version (as new optimizations are made) and target platform (similar to how `usize` varies per-platform).

切片

切片是一段连续的内存序列上的具有动态尺寸视图功能的类型，写为 `[T]`。
它经常以借用的形式出现，可变借用和共享借用都有可能。共享借用切片类型是 `&[T]`，可变借用切片类型是 `&mut [T]`，其中 `T` 表示元素类型。
A slice is dynamically-sized view into a contiguous sequence, written as `[T]`.
It is often seen in its borrowed forms, either mutable or shared. The shared slice type is `&[T]`, while the mutable slice type is `&mut [T]`, where `T` represents the element type.

语句

语句是编程语言中最小的独立元素，它命令计算机执行一个动作。
A statement is the smallest standalone element of a programming language that commands a computer to perform an action.

字符串字面量

字符串字面量是直接存储在最终二进制文件中的字符串，因此在 `'static` 生存期内是有效的。它的类型是借用形式的生存期 `'static` 的字符串切片，即：`&'static str`。
A string literal is a string stored directly in the final binary, and so will be valid for the `'static` duration.
Its type is `'static` duration borrowed string slice, `&'static str`.

字符串切片(`str`)

字符串切片是 Rust 中最基础的字符串类型，写为 `str`。它经常以借用的形式出现，可变借用和共享借用都有可能。共享借用的字符串切片类型是 `&str`，可变借用的字符串切片类型是 `&mut str`。
字符串切片总是有效的 UTF-8。
A string slice is the most primitive string type in Rust, written as `str`. It is often seen in its borrowed forms, either mutable or shared. The shared string slice type is `&str`, while the mutable string slice type is `&mut str`.
Strings slices are always valid UTF-8.

Trait

trait 是一种程序项，用于描述类型必须提供的功能。它允许类型对其行为做出某些承诺。泛型函数和泛型结构体可以使用 trait 来限制或约束它们所接受的类型。
A trait is a language item that is used for describing the functionalities a type must provide. It allows a type to make certain promises about its behavior.
Generic functions and generic structs can use traits to constrain, or bound, the types they accept.

无覆盖类型

不作为其他类型的参数出现的类型。例如，`T` 就是无覆盖的，但 `Vec<T>` 中的 `T` 就是有覆盖的。这（种说法）只与类型参数相关。
A type which does not appear as an argument to another type. For example, `T` is uncovered, but the `T` in `Vec<T>` is covered. This is only relevant for type arguments.

未定义行为

非指定的编译时或运行时行为。这可能导致，但不限于：进程终止或崩溃；不正确的、不正确的或非预定计算；或特定于平台的结果。查看更多
Compile-time or run-time behavior that is not specified. This may result in, but is not limited to: process termination or corruption; improper, incorrect, or unintended computation; or platform-specific results. More.

Uninhabited

如果类型没有构造函数，因此永远不能实例化，则该类型是 Uninhabited。一个 Uninhabited 类型是“空的”，意思是该类型没有值。Uninhabited 类型的典型例子是 never type `!`，或不带变体的 `enum Never { }`。与之相对的是 Inhabited。
A type is uninhabited if it has no constructors and therefore can never be instantiated. An uninhabited type is "empty" in the sense that there are no values of the type. The canonical example of an uninhabited type is the never type `!`, or an enum with no variants `enum Never { }`. Opposite of Inhabited.