



Rust 语言备忘清单

2020-05-31

参考书: **Rust 程序设计语言** [BK](#) (中文)、**通过例子学 Rust** [EX](#) (中文)、**标准库文档** [STD](#)、**Rust 死灵书** [NOM](#)、**Rust 参考手册** [REF](#)。

凡例: **已废弃** 、**最低版本** ¹8、**施工中** 、**不好的写法** .

数据结构

数据类型和内存位置由关键字定义。

示例	说明
<code>struct S {}</code>	定义包含命名字段的 结构体 BK EX STD REF 。
<code>struct S { x: T }</code>	定义包含 <code>T</code> 类型命名字段 <code>x</code> 的结构体。
<code>struct S(T);</code>	定义 <code>T</code> 类型数字字段 <code>.0</code> 的“元组”结构体。
<code>struct S;</code>	定义一个 零大小 NOM 单元的结构体。不占空间。
<code>enum E {}</code>	定义 枚举 BK EX REF 。参见数字数据类型、标签联合。
<code>enum E { A, B(), C {} }</code>	定义变体枚举，它可以是单元 <code>A</code> =元组 <code>B()</code> 或者结构体风格的 <code>C{}</code> 。
<code>enum E { A = 1 }</code>	如果所有变体都是单元值，则允许判别式值，可用于 FFI。
<code>union U {}</code>	不安全的 C 风格 联合体 REF ，用于兼容 FFI。
<code>static X: T = T();</code>	有 <code>'static</code> 生命周期的 全局变量 BK EX REF ，内存位置独立。
<code>const X: T = T();</code>	定义 常量 BK EX REF 。使用时会临时复制一份。
<code>let x: T;</code>	在栈 ¹ 上分配 <code>T</code> 大小的字节并命名为 <code>x</code> 。一旦分配，不可修改。
<code>let mut x: T;</code>	类似 <code>let</code> ，但允许修改和可变借用。 ²
<code>x = y;</code>	将 <code>y</code> 移动到 <code>x</code> ，如果 <code>T</code> 不能 <code>Copy</code> ， <code>y</code> 将不再可用，否则会复制一份 <code>y</code> 。

¹ 同步代码中，它们生存在栈上。但对于 `async` 代码，这些变量将会成为异步状态机的一部分，它们最终是在堆上。

² 注意术语**可变**和**不可变**并不准确。尽管你有一个不可变绑定或者共享引用，它也有可能包含一个 `Cell`，它仍支持 *内部可变性*。

下面列出了如何创建和访问数据结构，包括一些**神奇**的类型。

示例	说明
<code>S { x: y }</code>	创建 <code>struct S {}</code> ，或 <code>use</code> 的 <code>enum E::S {}</code> 字段 <code>x</code> 设置为 <code>y</code> 。
<code>S { x }</code>	同上，但字段 <code>x</code> 会设置为局部变量 <code>x</code> 。
<code>S { ..s }</code>	用 <code>s</code> 填充剩余字段，常配合 <code>Default</code> 使用。
<code>S { 0: x }</code>	类似下面的 <code>S(x)</code> 但是用结构体语法初始化字段 <code>.0</code> 。
<code>S(x)</code>	创建 <code>struct S(T)</code> ，或 <code>use</code> 的 <code>enum E::S()</code> 其中字段 <code>.0</code> 设置为 <code>x</code> 。
<code>S</code>	表示 <code>struct S</code> ；或以 <code>S</code> 为值创建 <code>use</code> 来的 <code>enum E::S</code> 。

示例	说明
<code>E::C { x: y }</code>	创建枚举变体 <code>C</code> 。上面的方法依然可用。
<code>()</code>	空元组，既是字面量也是类型，又称 单元 。 <code>STD</code>
<code>(x)</code>	括号表达式。
<code>(x,)</code>	单元素 元组 表达式。 <code>EX STD REF</code>
<code>(S,)</code>	单元素元组类型。
<code>[S]</code>	未指明长度的数组类型，如 切片 。 <code>STD EX REF</code> 不能生存在栈上。 *
<code>[S; n]</code>	元素类型为 <code>S</code> 定长为 <code>n</code> 的 数组类型 <code>EX STD</code> 。
<code>[x; n]</code>	由 <code>n</code> 个 <code>x</code> 的副本构成的数组实例。 <code>REF</code>
<code>[x, y]</code>	由给定元素 <code>x</code> 和 <code>y</code> 构成的数组实例。
<code>x[0]</code>	组合的索引。可重载 <code>Index</code> 和 <code>IndexMut</code> 。
<code>x[..]</code>	组合的切片式索引，全部范围 <code>RangeFull</code> ，参见 切片。
<code>x[a..]</code>	组合的切片式索引，指定起始的范围 <code>RangeFrom</code> 。
<code>x[..b]</code>	组合的切片式索引，指定终止的范围 <code>RangeTo</code> 。
<code>x[a..b]</code>	组合的切片式索引，指定始终的范围 <code>Range</code> 。
<code>a..b</code>	左闭右开 区间 <code>REF</code> ， <code>..b</code> 同理。
<code>a..=b</code>	闭区间， <code>..=b</code> 同理。
<code>s.x</code>	命名 字段访问 <code>REF</code> ，如果 <code>x</code> 不是 <code>S</code> 的一部分的话则会尝试 <code>Deref</code> 。
<code>s.0</code>	数字字段访问，用于元组类型 <code>S(T)</code> 。

* 目前，可以参考 该已知问题 和关联的 `RFC 1909`。

引用和指针

为非所有者内存赋予访问权限。参见 泛型和约束。

示例	说明
<code>&S</code>	共享 引用 <code>BK STD NOM REF</code> (用于存储 任意 <code>&S</code>)。
<code>&[S]</code>	特殊的切片引用，包含地址和长度 (<code>address</code> , <code>length</code>)。
<code>&str</code>	特殊的字符串引用，包含地址和长度 (<code>address</code> , <code>length</code>)。
<code>&mut S</code>	允许修改的独占引用 (参见 <code>&mut [S]</code> , <code>&mut dyn S</code> , ...)
<code>&dyn T</code>	特殊的 trait 对象 <code>BK</code> 引用，包含地址和虚表 (<code>address</code> , <code>vtable</code>)。
<code>*const S</code>	不可变的 裸指针类型 <code>BK STD REF</code> ，内存不安全。
<code>*mut S</code>	可变的裸指针类型，内存不安全。
<code>&s</code>	共享 借用 <code>BK EX STD</code> (例如 该 <code>s</code> 的地址、长度、虚表等，比如 <code>0x1234</code>)。
<code>&mut s</code>	有 可变性 的独占借用。 <code>EX</code>
<code>ref s</code>	引用绑定 。 <code>EX</code> 🗑
<code>*r</code>	对引用 <code>r</code> 解引用 <code>BK STD NOM</code> 以访问其指向的事物。
<code>*r = s;</code>	如果 <code>r</code> 是一个可变引用，则将 <code>s</code> 移动或复制到目标内存。
<code>s = *r;</code>	如果 <code>r</code> 可 <code>Copy</code> ，则将 <code>r</code> 引用的内容复制到 <code>s</code> 。
<code>s = *my_box;</code>	<code>Box</code> 有一个 特例 ，即便它不可 <code>Copy</code> ，也仍会从 <code>Box</code> 里面移动出来。
<code>'a</code>	生命周期参数 ， <code>BK EX NOM REF</code> ，为静态分析声明一块代码的持续时间。

示例	说明
<code>&'a S</code>	仅支持生存时间不短于 <code>'a</code> 的地址 <code>s</code> 。
<code>&'a mut S</code>	同上，但允许改变地址指向的内容。
<code>struct S<'a> {}</code>	表明 <code>S</code> 包含一个生命周期为 <code>'a</code> 的地址。由 <code>S</code> 的创建者决定 <code>'a</code> 。
<code>trait T<'a> {}</code>	表明一个实现了 <code>impl T for S</code> 的 <code>S</code> 可能会包含地址。
<code>fn f<'a>(t: &'a T)</code>	同上，用于函数。调用者决定 <code>'a</code> 。
<code>'static</code>	特殊的生命周期，生存在程序的整个执行过程中。

函数和行为

定义代码单元及其抽象。

示例	说明
<code>trait T {}</code>	定义 trait BK EX REF ，它是一系列可被实现的通用行为。
<code>trait T : R {}</code>	<code>T</code> 是 父 trait REF <code>R</code> 的子 trait。任何要 <code>impl T</code> 的 <code>S</code> 都必须先 <code>impl R</code> 。
<code>impl S {}</code>	类型 <code>S</code> 的函数 实现 REF ，如方法。
<code>impl T for S {}</code>	为类型 <code>S</code> 实现 trait <code>T</code> 。
<code>impl !T for S {}</code>	禁用自动推导的 auto trait NOM REF 。
<code>fn f() {}</code>	定义一个 函数 BK EX REF ，或在 <code>impl</code> 里关联一个函数。
<code>fn f() -> S {}</code>	同上，但会返回一个 <code>S</code> 类型的值。
<code>fn f(&self) {}</code>	定义一个方法。例如，在 <code>impl S {}</code> 里面。
<code>const fn f() {}</code>	编译器常量函数 <code>fn</code> ，例如 <code>const X: u32 = f(Y)</code> 。 ^{'18}
<code>async fn f() {}</code>	异步 ^{'18} 函数转写。令 <code>f</code> 返回 <code>impl Future</code> STD 。
<code>async fn f() -> S {}</code>	同上，但令 <code>f</code> 返回 <code>impl Future<Output=S></code> 。
<code>async { x }</code>	用在函数内部，使 <code>{ x }</code> 变得 <code>impl Future<Output=X></code> 。
<code>fn() -> S</code>	函数指针 BK STD REF ，内存存放的可调用地址。
<code>Fn() -> S</code>	可调用 Trait BK STD （又见 <code>FnMut</code> 和 <code>FnOnce</code> ），可由闭包或函数等实现。
<code> {}</code>	闭包 BK EX REF ，将会借用它所有的捕获。
<code> x {}</code>	有传入参数 <code>x</code> 的闭包。
<code> x x + x</code>	没有块表达式的闭包，仅可由单个表达式组成。
<code>move x x + y</code>	闭包，将会获取它所有捕获的所有权。
<code>return true</code>	闭包，起来像是逻辑或，但这里表示返回一个闭包。
<code>unsafe {}</code>	不安全代码 BK EX NOM REF 。如果你喜欢在周五晚上调试段错误的话~

控制流程

在函数中控制执行。

示例	说明
<code>while x {}</code>	循环 REF ，当表达式 <code>x</code> 为真时运行。
<code>loop {}</code>	无限循环 REF 直到 <code>break</code> 。可以用 <code>break x</code> 来 <code>yield</code> 一个值出来。
<code>for x in iter {}</code>	在 迭代器 上循环的语法糖。 BK STD REF
<code>if x {} else {}</code>	条件分支 REF 。如果表达式为真则.....否则.....

示例	说明
<code>'label: loop {}</code>	循环标签 EX REF ，用于嵌套循环的流程控制。
<code>break</code>	Break 表达式 REF ，用于退出循环。
<code>break x</code>	同上，但将 <code>x</code> 作为循环表达式的值（仅在 <code>loop</code> 中有效）。
<code>break 'label</code>	不单单退出的是当前循环，而是最近一个标记有 <code>'label</code> 的循环。
<code>continue</code>	Continue 表达式 REF ，用于继续该循环的下次迭代。
<code>continue 'label</code>	同上，但继续的是最近标记有 <code>'label</code> 的循环迭代。
<code>x?</code>	如果 <code>x</code> 是 <code>Err</code> 或 <code>None</code> ， 返回并向上传播 。 BK EX STD REF
<code>x.await</code>	仅在 <code>async</code> 中有效。将会 <code>yield</code> 当前流，直到 <code>Future</code> 或 <code>Stream</code> <code>x</code> 就绪。 ¹⁸
<code>return x</code>	从函数中提前返回。然而以表达式结束的方式更惯用。
<code>f()</code>	调用 <code>f</code> （如函数、闭包、函数指针或 <code>Fn</code> 等）。
<code>x.f()</code>	调用成员函数（方法），要求 <code>f</code> 以 <code>self</code> 、 <code>&self</code> 等作为第一个参数。
<code>X::f(x)</code>	同 <code>x.f()</code> 。除非 <code>impl Copy for X {}</code> ，否则 <code>f</code> 仅可调用一次。
<code>X::f(&x)</code>	同 <code>x.f()</code> 。
<code>X::f(&mut x)</code>	同 <code>x.f()</code> 。
<code>S::f(&x)</code>	同 <code>x.f()</code> ，仅当 <code>x</code> 实现了对 <code>S</code> 的 <code>Deref</code> 。这里 <code>x.f()</code> 会去找 <code>S</code> 的方法。
<code>T::f(&x)</code>	同 <code>x.f()</code> ，仅当 <code>x</code> <code>impl T</code> 。这里 <code>x.f()</code> 会去找作用域内 <code>T</code> 的方法。
<code>X::f()</code>	调用关联函数。比如 <code>X::new()</code> 。
<code><X as T>::f()</code>	调用为 <code>X</code> 实现了的 trait 方法 <code>T::f()</code> 。

代码组织

将项目分割成小的单元并最小化相关依赖。

示例	说明
<code>mod m {}</code>	定义 模块 BK EX REF ，其中的定义在 <code>{}</code> 内。
<code>mod m;</code>	定义模块，其中的定义在 <code>m.rs</code> 或 <code>m/mod.rs</code> 内。
<code>a::b</code>	命名空间 路径 EX REF ，表示 <code>a</code> （ <code>mod</code> 或 <code>enum</code> 等）里面的元素 <code>b</code> 。
<code>::b</code>	相对于当前 <code>crate</code> 根下搜索 <code>b</code> 。 
<code>crate::b</code>	相对于当前 <code>crate</code> 根下搜索 <code>b</code> 。 ¹⁸
<code>self::b</code>	相对于当前模块下搜索 <code>b</code> 。
<code>super::b</code>	相对于父级模块下搜索 <code>b</code> 。
<code>use a::b;</code>	Use EX REF 声明，将 <code>b</code> 直接引入到当前作用域，以后就不需要再加 <code>a</code> 前缀了。
<code>use a::{b, c};</code>	同上，但同时将 <code>b</code> 和 <code>c</code> 都引入。
<code>use a::b as x;</code>	将 <code>b</code> 引入作用域但命名为 <code>x</code> 。比如 <code>use std::error::Error as E</code> 。
<code>use a::b as _;</code>	将 <code>b</code> 匿名的引入作用域，用于含有冲突名称的 <code>trait</code> 。
<code>use a::*;</code>	将 <code>a</code> 里面的所有元素都引入作用域。
<code>pub use a::b;</code>	将 <code>a::b</code> 引入作用域，并再次从当前位置导出。
<code>pub T</code>	控制 <code>T</code> 的 可见性 BK 。“如果父级路径公开，我也公开”。
<code>pub(crate) T</code>	可见性仅在当前 <code>crate</code> 内。
<code>pub(self) T</code>	可见性仅在当前模块内。

示例	说明
<code>pub(super) T</code>	可见性仅在父级以下。
<code>pub(in a::b) T</code>	可见性仅在 <code>a::b</code> 内。
<code>extern crate a;</code>	声明依赖一个外部 crate BK EX REF  。换用 <code>use a::b '18</code> 。
<code>extern "C" {}</code>	声明 FFI 的外部依赖和 ABI (如 <code>"C"</code>)。 BK EX NOM REF
<code>extern "C" fn f() {}</code>	定义 FFI 导出成 ABI (如 <code>"C"</code>) 的函数。

类型别名和转换

类型名称的简写，以及转为其他类型的方法。

示例	说明
<code>type T = S;</code>	创建 类型别名 BK REF 。这里表示 <code>S</code> 的另一个名字。
<code>Self</code>	当前类型 REF 的类型别名。如 <code>fn new() -> Self</code> 。
<code>self</code>	<code>fn f(self) {}</code> 的方法主体。同 <code>fn f(self: Self) {}</code> 。
<code>&self</code>	同上，但将借用指向自己的引用。同 <code>f(self: &Self)</code> 。
<code>&mut self</code>	同上，但是可变借用。同 <code>f(self: &mut Self)</code> 。
<code>self: Box<Self></code>	任意自型，为智能指针增加方法 (<code>my_box.f_of_self()</code>)。
<code>S as T</code>	消歧义 BK REF ，将类型 <code>S</code> 作为 trait <code>T</code> 看待。比如 <code><X as T>::f()</code> 。
<code>S as R</code>	在 <code>use</code> 里，将 <code>S</code> 导入为 <code>R</code> 。如 <code>use a::b as x</code> 。
<code>x as u32</code>	裸转换 EX REF ，会发生截断和一些位上的意外。 NOM

宏和属性

实际编译前的代码预展开。

示例	说明
<code>m!()</code>	宏 BK STD REF 咒语。也作 <code>m!{}</code> 或 <code>m![]</code> （取决于宏本身）。
<code>\$x:ty</code>	宏捕获。如 <code>\$x:expr</code> 、 <code>\$x:ty</code> 、 <code>\$x:path</code> 等，见下表。
<code>\$x</code>	宏举例 中的宏代称。 BK EX REF
<code>\$x),*</code>	宏举例中的宏重复数。零或更多次。
<code>\$x),?</code>	同上，零或一次。
<code>\$x),+</code>	同上，一或更多次。
<code>\$x)<+></code>	支持不是 <code>,</code> 的其他分隔符。这里是 <code><<</code> 。
<code>\$crate</code>	特殊变量，指明宏定义在哪个 <code>crate</code> 里。?
<code>#[attr]</code>	外部 属性 EX REF 。注解接下来的内容。
<code>#![attr]</code>	内部属性。注解附近的内容。


在 `macro_rules!` 实现里，会用到下面的宏捕获：

宏捕获	说明
<code>\$x:item</code>	项目。如函数、结构体、模块等。
<code>\$x:block</code>	语句或表达式块 <code>{}</code> 。如 <code>{ let x = 5; }</code>
<code>\$x:stmt</code>	语句。如 <code>let x = 1 + 1;</code> 、 <code>String::new();</code> 或 <code>vec![];</code>
<code>\$x:expr</code>	表达式。如 <code>x</code> 、 <code>1 + 1</code> 、 <code>String::new()</code> 或 <code>vec![]</code>

宏捕获	说明
<code>\$x:pat</code>	匹配。如 <code>Some(t)</code> 、 <code>(17, 'a')</code> 或 <code>_</code>
<code>\$x:ty</code>	类型。如 <code>String</code> 、 <code>usize</code> 或 <code>Vec<u8></code>
<code>\$x:ident</code>	标识符。如 <code>let x = 0;</code> 中的 <code>x</code> 。
<code>\$x:path</code>	路径。如 <code>foo::std::mem::replace</code> 、 <code>transmute::<_, int></code> 等。
<code>\$x:literal</code>	字面量。如 <code>3</code> 、 <code>"foo"</code> 、 <code>b"bar"</code> 等。
<code>\$x:lifetime</code>	生命周期。如 <code>'a</code> 、 <code>'static</code> 等。
<code>\$x:meta</code>	元数据，即 <code>#[...]</code> 和 <code>#![...]</code> 中的属性值。
<code>\$x:vis</code>	可见性修饰符。如 <code>pub</code> 、 <code>pub(crate)</code> 等。
<code>\$x:tt</code>	单棵语法树。详细请参见 这里 。

模式匹配

函数参数、`match` 或 `let` 表达式中的构造。

示例	说明
<code>match m {}</code>	模式匹配 BK EX REF ，下面跟匹配分支。参见 下表。
<code>let S(x) = get();</code>	显然， <code>let</code> 也和下表的模式匹配类似。
<code>let S { x } = s;</code>	仅将 <code>x</code> 绑定到值 <code>s.x</code> 。
<code>let (_, b, _) = abc;</code>	仅将 <code>b</code> 绑定到值 <code>abc.1</code> 。
<code>let (a, ..) = abc;</code>	也可以将“剩余的”都忽略掉。
<code>let Some(x) = get();</code>	不可用  ，因为模式可能会 不匹配 REF 。换用 <code>if let</code> 。
<code>if let Some(x) = get() {}</code>	如果模式匹配则执行该分支（如某个 <code>enum</code> 变体）。语法糖*。
<code>fn f(S { x }: S)</code>	类似于 <code>let</code> ，模式匹配也可用在函数参数上。这里， <code>f(s)</code> 的 <code>x</code> 被绑定到 <code>s.x</code> 。

* 展开后是 `match get() { Some(x) => {}, _ => () }`。

`match` 表达式的模式匹配分支。左列的分支也可用于 `let` 表达式。

匹配分支	说明
<code>E::A => {}</code>	匹配枚举变体 <code>A</code> 。参见 模式匹配 。 BK EX REF
<code>E::B (..) => {}</code>	匹配枚举元组变体 <code>B</code> ，通配所有下标。
<code>E::C { .. } => {}</code>	匹配枚举结构变体 <code>C</code> ，通配所有字段。
<code>S { x: 0, y: 1 } => {}</code>	匹配含特定值的结构体（仅匹配 <code>s</code> 的 <code>s.x</code> 为 <code>0</code> 且 <code>s.y</code> 为 <code>1</code> 的情况）。
<code>S { x: a, y: b } => {}</code>	匹配为 任意(!) 值的该类型结构体，并绑定 <code>s.x</code> 到 <code>a</code> ，绑定 <code>s.y</code> 到 <code>b</code> 。
<code> S { x, y } => {}</code>	同上，但将 <code>s.x</code> 和 <code>s.y</code> 分别简写地绑定为 <code>x</code> 和 <code>y</code> 。
<code>S { .. } => {}</code>	匹配任意值的该类型结构体。
<code>D => {}</code>	匹配枚举变体 <code>E::D</code> 。仅当 <code>D</code> 已由 <code>use</code> 引入。
<code>D => {}</code>	匹配任意事物并绑定到 <code>D</code> 。如果 <code>D</code> 没被 <code>use</code> 进来，怕不是个 <code>E::D</code> 的假朋友。 
<code>_ => {}</code>	通配所有，或者所有剩下的。
<code>(a, 0) => {}</code>	匹配元组，绑定第一个值到 <code>a</code> ，要求第二个是 <code>0</code> 。
<code>[a, 0] => {}</code>	切片模式 REF  。绑定第一个值到 <code>a</code> ，要求第二个是 <code>0</code> 。
<code>[1, ..] => {}</code>	匹配以 <code>1</code> 开始的数组，剩下的不管。 子切片模式 。 [?]

匹配分支	说明
<code>[2, .., 5] => {}</code>	匹配以 1 开始以 5 结束的数组。
<code>[2, x @ .., 5] => {}</code>	同上，但将 x 绑定到中间部分的切片上（见下）。
<code>x @ 1..=5 => {}</code>	绑定匹配到 x ，即 模式绑定 BK EX REF 。这里 x 可以是 1 、 2 直到 5 。
<code>0 1 => {}</code>	替代模式（或模式）。
<code>E::A E::Z</code>	同上，但是枚举变体。
<code>E::C {x} E::D {x}</code>	同上，但将 x 绑定到每个模式都有的 x 上面。
<code>S { x } if x > 10 => {}</code>	模式 匹配条件 BK EX REF 。该匹配会要求这个条件也为真。

泛型和约束

泛型有多种构造方式：`struct S<T>`、`fn f<T>()` 等等。

示例	说明
<code>S<T></code>	泛型 BK EX ，类型参数 T 是占位符。
<code>S<T: R></code>	类型短 trait 约束 BK EX 说明。（ R 必须 是个实际的 trait）。
<code>T: R, P: S</code>	独立 trait 约束 （这里一个对 T ，一个对 P ）。
<code>T: R, S</code>	编译错误🚫。可以用下面的 <code>R + S</code> 代替。
<code>T: R + S</code>	合并 trait 约束 BK EX 。 T 必须同时满足 R 和 S 。
<code>T: R + 'a</code>	同上，但有生命周期。 T 必须满足 R ；如果 T 有生命周期，则必须长于 'a 。
<code>T: ?Sized</code>	在前置定义 trait 约束之外的选项。 <code>Sized?</code> ？
<code>T: 'a</code>	类型 生命周期约束 EX 。 T 应长于 'a 。
<code>'b: 'a</code>	约束生命周期 'b 必须至少和 'a 一样长。
<code>S<T> where T: R</code>	同 <code>S<T: R></code> ，但使得长的约束说明更易读。
<code>S<T = R></code>	关联类型 默认类型参数 BK 。
<code>S<'_></code>	推测 匿名生命周期 。如果显然可见，让编译器“自己搞定”。
<code>S<_></code>	推测 匿名类型 。如 <code>let x: Vec<_> = iter.collect()</code> 。
<code>S::<T></code>	Turbofish STD 消歧义类型调用。如 <code>f::<u32>()</code> 。
<code>trait T<X> {}</code>	X 的 trait 泛型。可以有多个 <code>impl T for S</code> （每个 X 一个）。
<code>trait T { type X; }</code>	定义 关联类型 BK REF X 。仅可有一个 <code>impl T for S</code> 。
<code>type X = R;</code>	设置关联类型。仅在 <code>impl T for S { type X = R; }</code> 内。
<code>impl<T> S<T> {}</code>	实现 <code>S<T></code> 任意类型 T 的功能。
<code>impl S<T> {}</code>	实现确定 <code>S<T></code> 的功能。如 <code>S<u32></code> 。
<code>fn f() -> impl T</code>	Existential 类型 BK 。返回一个对调用者未知的但 <code>impl T</code> 的 S 。
<code>fn f(x: &impl T)</code>	Trait 约束，“ <code>impl trait</code> ” BK 。和 <code>fn f<S:T>(x: &S)</code> 有点类似。
<code>fn f(x: &dyn T)</code>	动态分发 标记 BK REF 。 f 不再单态。
<code>fn f() where Self: R</code>	在 <code>trait T {}</code> 中标记 f 仅可由实现了 <code>impl R</code> 的类型访问。。
<code>for<'a></code>	高阶 trait 约束 。 NOM REF
<code>trait T: for<'a> R<'a> {}</code>	任何 <code>impl T</code> 的 S 在任意生命周期都需满足 R 。

字符串和字符

Rust 提供了若干种创建字符串和字符字面量的办法。

示例	说明
<code>"..."</code>	UTF-8 字符串字面量 REF 。会将 <code>\n</code> 等看作换行 <code>0xA</code> 等。
<code>r"..."</code>	UTF-8 裸字符串字面量 REF 。不会处理 <code>\n</code> 等。
<code>r#"..."#</code> 等	UTF-8 裸字符串字面量。但可以包含 <code>"</code> 。
<code>b"..."</code>	字节串字面量 REF ，由 ASCII <code>[u8]</code> 组成。不是字符串。
<code>br"..."</code> 、 <code>br#"..."#</code> 等	裸字节串字面量，ASCII <code>[u8]</code> 。说明见上。
<code>'🐼'</code>	字符字面量 REF ，固定的 4 字节 Unicode “ 字符 ”。 STD
<code>b'x'</code>	ASCII 字节字面量 。 REF

注释

无需解释。

示例	说明
<code>//</code>	行内注释。用于文档代码流内或_内部组件_。
<code>//!</code>	行内 文档注释 BK EX REF 。用于自动文档生成。
<code>///</code>	外部行内文档注释。在类型上面用。
<code>/*...*/</code>	块级注释。
<code>/*!...*/</code>	内部块级文档注释。
<code>/**...*/</code>	外部块级文档注释。
<code>``rust ... ``</code>	在文档注释中包含 文档测试 （文档代码可以用 <code>cargo test</code> 运行）。
<code>#</code>	隐藏文档测试中某行（ <code>`` # use x::hidden; ``</code> ）。

其他

这些小技巧不属于其他分类但最好了解一下。

示例	说明
<code>!</code>	永远为空的 never 类型 。 BK EX STD REF
<code>_</code>	无名变量绑定。如 <code> x, _ {}</code> 。
<code>_x</code>	变量绑定，明确标记该变量未使用。
<code>1_234_567</code>	为了易读加入的数字分隔符。
<code>1_u8</code>	数字字面量 的类型说明符。 EX REF （又见 <code>i8</code> 、 <code>u16</code> ）。
<code>0xBEEF, 0o777, 0b1001</code>	十六进制（ <code>0x</code> ）、八进制（ <code>0o</code> ）和二进制（ <code>0b</code> ）整型字面量。
<code>r#foo</code>	裸标识符 BK EX 。用于版本兼容。
<code>x;</code>	语句 REF 终止符。见 表达式 EX REF 。

通用操作符


Rust 支持大部分其他语言也有的通用操作符（`+`，`*`，`%`，`=`，`==`...）。因为这在 Rust 里没什么太大差别所以这里不列出来了。Rust 也支持**运算符重载**。[STD](#)

增强设施

语法糖

如果有什么东西让你觉得，“不该能用的啊”，那可能就是这里的原因。

名称	说明
强制类型转换 NOM	“弱”类型匹配签名。如 <code>&mut T</code> 到 <code>&T</code> 。
解引用 NOM	<code>Deref x: T</code> 将会一直解引用 <code>*x</code> 、 <code>**x</code>直到满足目标 <code>S</code> 。
Prelude STD	自动导入基本类型。
重新借用	即便 <code>x: &mut T</code> 不能复制，也可以移动一个新的 <code>&mut *x</code> 代替。
生命周期省略 BK NOM REF	自动注解 <code>f(x: &T)</code> 为 <code>f<'a>(x: &'a T)</code> 。
方法解析 REF	解引用或借用 <code>x</code> 直到 <code>x.f()</code> 有效。

编者记  —— 尽管上面的特性将使简化了开发工作，但它们也会对理解当前发生了什么造成可能的妨碍。如果你对 Rust 还不太了解，想要搞明白到底发生了什么，你应该更详细地阅读相关资料。

标准库

Trait

Trait 定义通用行为。如果 `S` 实现了 `trait T`，意味着 `S` 可以做 `T` 规定的行为。下面是一些常用但有些技巧性的 trait。

线程安全

例	Send [*]	!Send
Sync [*]	Mutex<T>、Arc<T> ^{1,2} 、大多数类型.....	MutexGuard<T> ¹ 、RwLockReadGuard<T> ¹
!Sync	Cell<T> ² 、RefCell<T> ²	Rc<T>、Formatter、&dyn Trait

^{*} **T: Send** 表示实例 `t` 可以移动到另一个线程；**T: Sync** 表示 `&t` 可以移动到另一个线程。

¹ 如果 `T` 为 `Sync`。

² 如果 `T` 为 `Send`。

迭代器

使用迭代器
<h3>基本用法</h3> <p>假设有一系列 <code>c</code> 类型的 <code>c</code>：</p> <ul style="list-style-type: none"><code>c.into_iter()</code> — 将序列 <code>c</code> 转为 <code>Iterator</code> STD <code>i</code>，并消耗掉[*] <code>c</code>。要求为 <code>c</code> 实现 <code>IntoIterator</code> STD。条目类型取决于 <code>c</code>。获取迭代器的“标准”做法。<code>c.iter()</code> — 某些序列提供的更优方法，返回借用的迭代器，不消耗掉 <code>c</code>。<code>c.iter_mut()</code> — 同上，但返回可变借用迭代器来允许改变序列内容。

迭代器

对于迭代条目 `i`：

- `i.next()` — 如果 `c` 有下一个元素则返回 `Some(x)`，否则返回 `None`。

循环

- `for x in c {}` — 语法糖。调用 `c.into_iter()` 并循环 `i` 直到 `None`。

* 如果 `c` 看似并未被消耗掉，是因为类型实现了 `Copy`。例如，调用 `(&c).into_iter()` 将会在 `&c` 调用 `.into_iter()`（将会消耗掉这个引用并将其转为迭代器），但是剩下的 `c` 不受影响。

实现迭代器

基本用法

假设有一系列的 `struct C {}`。

- `struct Iter {}` — 创建用于保存不可变迭代器状态的结构体（比如索引）。
- `impl Iterator for Iter {}` — 实现 `Iterator::next()` 以产生元素。

此外，可以在 `impl C {}` 里提供一个 `fn iter(&self) -> Iter` 方法。

可变迭代器

- `struct IterMut {}` — 创建可变迭代器的结构体，它可以将保存的 `c` 视为 `&mut`。
- `impl Iterator for IterMut {}` — 这里 `Iterator::Item` 就是个 `&mut item` 了。

类似地，也可以实现一个 `fn iter_mut(&mut self) -> IterMut` 方法。

实现循环

- `impl IntoIterator for C {}` — 此时，`for` 循环可以如此使用了 `for x in c {}`。
- `impl IntoIterator for &C {}` — 为使用方便也可实现这个。
- `impl IntoIterator for &mut C {}` — 同理.....

字符串转换

将字符串 `x` 转为目标类型.....

String

x 的类型	转换方法
String	<code>x</code>
CString	<code>x.into_string()? </code>
OsString	<code>x.to_str()?.into()</code>
PathBuf	<code>x.to_str()?.into()</code>

x 的类型	转换方法
<code>Vec<u8></code> ¹	<code>String::from_utf8(x)?</code>
<code>&str</code>	<code>x.into()</code>
<code>&CStr</code>	<code>x.to_str()?.into()</code>
<code>&OSStr</code>	<code>x.to_str()?.into()</code>
<code>&Path</code>	<code>x.to_str()?.into()</code>
<code>&[u8]</code> ¹	<code>String::from_utf8_lossy(x).into()</code>
CString	
x 的类型	转换方法
<code>String</code>	<code>CString::new(x)?</code>
<code>CString</code>	<code>x</code>
<code>OsString</code> ²	<code>CString::new(x.to_str())?</code>
<code>PathBuf</code>	<code>CString::new(x.to_str())?</code>
<code>Vec<u8></code> ¹	<code>CString::new(x)?</code>
<code>&str</code>	<code>CString::new(x)?</code>
<code>&CStr</code>	<code>x.into()</code>
<code>&OSStr</code> ²	<code>CString::new(x.to_os_string().into_string())?</code>
<code>&Path</code>	<code>x.to_str()?.into()</code>
<code>&[u8]</code> ¹	<code>CString::new(Vec::from(x))?</code>
<code>*mut c_char</code> ³	<code>unsafe { CString::from_raw(x) }</code>
OsString	
x 的类型	转换方法
<code>String</code>	<code>x.into()</code>
<code>CString</code>	<code>x.to_str()?.into()</code>
<code>OsString</code>	<code>x</code>
<code>PathBuf</code>	<code>x.into_os_string()</code>
<code>Vec<u8></code> ¹	<code>?</code>
<code>&str</code>	<code>x.into()</code>
<code>&CStr</code>	<code>x.to_str()?.into()</code>
<code>&OSStr</code>	<code>x.into()</code>
<code>&Path</code>	<code>x.as_os_str().into()</code>
<code>&[u8]</code> ¹	<code>?</code>
PathBuf	

x 的类型	转换方法
String	<code>x.into()</code>
CString	<code>x.to_str()?.into()</code>
OsString	<code>x.into()</code>
PathBuf	<code>x</code>
<code>Vec<u8></code> ¹	<code>?</code>
<code>&str</code>	<code>x.into()</code>
<code>&CStr</code>	<code>x.to_str()?.into()</code>
<code>&OSStr</code>	<code>x.into()</code>
<code>&Path</code>	<code>x.into()</code>
<code>&[u8]</code> ¹	<code>?</code>

`Vec<u8>`

x 的类型	转换方法
String	<code>x.into_bytes()</code>
CString	<code>x.into_bytes()</code>
OsString	<code>?</code>
PathBuf	<code>?</code>
<code>Vec<u8></code> ¹	<code>x</code>
<code>&str</code>	<code>x.as_bytes().into()</code>
<code>&CStr</code>	<code>x.to_bytes_with_nul().into()</code>
<code>&OSStr</code>	<code>?</code>
<code>&Path</code>	<code>?</code>
<code>&[u8]</code> ¹	<code>x.into()</code>

`&str`

x 的类型	转换方法
String	<code>x.as_str()</code>
CString	<code>x.to_str()?</code>
OsString	<code>x.to_str()?</code>
PathBuf	<code>x.to_str()?</code>
<code>Vec<u8></code> ¹	<code>std::str::from_utf8(&x)?</code>
<code>&str</code>	<code>x</code>
<code>&CStr</code>	<code>x.to_str()?</code>
<code>&OSStr</code>	<code>x.to_str()?</code>

x 的类型	转换方法
<code>&Path</code>	<code>x.to_str()? </code>
<code>&[u8] ¹</code>	<code>std::str::from_utf8(x)? </code>

`&CStr`

x 的类型	转换方法
<code>String</code>	<code>CString::new(x)?.as_c_str() </code>
<code>CString</code>	<code>x.as_c_str() </code>
<code>OsString ²</code>	<code>x.to_str()? </code>
<code>PathBuf</code>	<code>CStr::from_bytes_with_nul(x.to_str()?.as_bytes())? </code>
<code>Vec<u8> ¹</code>	<code>CStr::from_bytes_with_nul(&x)? </code>
<code>&str</code>	<code>CStr::from_bytes_with_nul(x.as_bytes())? </code>
<code>&CStr</code>	<code>x </code>
<code>&OsStr ²</code>	<code>? </code>
<code>&Path</code>	<code>? </code>
<code>&[u8] ¹</code>	<code>CStr::from_bytes_with_nul(x)? </code>
<code>*const c_char ¹</code>	<code>unsafe { CString::from_ptr(x) } </code>

`&OsStr`

x 的类型	转换方法
<code>String</code>	<code>OsStr::new(&x) </code>
<code>CString</code>	<code>? </code>
<code>OsString</code>	<code>x.as_os_str() </code>
<code>PathBuf</code>	<code>x.as_os_str() </code>
<code>Vec<u8> ¹</code>	<code>? </code>
<code>&str</code>	<code>OsStr::new(x) </code>
<code>&CStr</code>	<code>? </code>
<code>&OsStr</code>	<code>x </code>
<code>&Path</code>	<code>x.as_os_str() </code>
<code>&[u8] ¹</code>	<code>? </code>

`&Path`

x 的类型	转换方法
<code>String</code>	<code>x.as_ref() </code>
<code>CString</code>	<code>x.to_str()?.as_ref() </code>

x 的类型	转换方法
OsString	x.as_ref()
PathBuf	x.as_ref()
Vec<u8> ¹	?
&str	x.as_ref()
&CStr	x.to_str()?.as_ref()
&OSStr	x.as_ref()
&Path	x
&[u8] ¹	?

&[u8]

x 的类型	转换方法
String	x.as_bytes()
CString	x.as_bytes()
OsString	?
PathBuf	?
Vec<u8> ¹	&x
&str	x.as_bytes()
&CStr	x.to_bytes_with_nul()
&OSStr	x.as_bytes() ²
&Path	?
&[u8] ¹	x

*const c_char

目标类型	源类型 x	转换方法
*const c_char	CString	x.as_ptr()

¹ 你应当或必须（当调用了 `unsafe` 时）确保裸数据是有效的字符串表示（比如，`String` 是 UTF-8 编码数据）。

² 仅在某些平台上 `std::os::<your_os>::ffi::OsStrExt` 有辅助方法来访问 `OsStr` 的裸 `&[u8]` 表示。所以有时需要手动再转换一遍：

```
use std::os::unix::ffi::OsStrExt;
let bytes: &[u8] = my_os_str.as_bytes();
CString::new(bytes)?
```

³ `c_char` 必须由前一个 `CString` 转换而来。如果是从 FFI 来的，则用 `&CStr` 代替。

字符串格式化

`println!`、`eprintln!`、`write!`（这些宏和对应的 `-ln` 宏，如 `println!`）都会格式化。格式化参数是 `{}` 或 `{argument}`，或遵循下面的基本语法：

```
{ [argument] ':' [[fill] align] [sign] ['#'] [width [$]] ['.' precision [$]] [type] }
```






元素	含义
argument	数字（0、1……）或参数名。如 <code>println!("{x}", x = 3)</code> 。
fill	当提供了 <code>width</code> 时，用于填充空白的字符（如 0）。
align	当提供了 <code>width</code> 时，表示左（<）、中（^）、右（>）。
sign	为 + 时表示总是显示正负号。
#	变体格式化。如调试信息 <code>?</code> 或十六进制 <code>0x</code> 。
width	用 <code>fill</code> 填充（默认为空格）的最小宽度（≥ 0）。如果以 0 开始则以零填充。
precision	数字位数（≥ 0），或非数字的最大宽度。
\$	将 <code>width</code> 或 <code>precision</code> 解释为参数标识符，以允许动态格式化。
type	调试格式化(?)、十六进制(x)、二进制(b)、八进制(o)、指针(p)、科学计数法(e)……参见更多。

示例	说明
<code>{:?}</code>	打印参数调试信息。
<code>{2:#?}</code>	打印第三个参数，并格式化成更易读的调试信息。
<code>{val:^2\$}</code>	将具名参数 <code>val</code> 居中格式化，宽度由第三个参数指定。
<code>{:<10.3}</code>	左对齐打印，宽度为 10，小数位 3。
<code>{val:#x}</code>	将参数 <code>val</code> 格式化为十六进制，并有前导 <code>0x</code> （ <code>x</code> 的变体格式）。

工具

项目结构

项目结构布局，通用的文件和目录，这是 Rust 工具化的一部分。

文件/目录	代码
 <code>benches/</code>	crate 的性能测试，用 <code>cargo bench</code> 运行，需要 nightly。* 
 <code>examples/</code>	使用 crate 的例程，用 <code>cargo run --example my_example</code> 运行。
 <code>src/</code>	项目实际源代码。
<code>build.rs</code>	预编译脚本。比如，当编译 C / FFI 时需要在 <code>Cargo.toml</code> 中指定的。
<code>main.rs</code>	应用程序默认入口点，即 <code>cargo run</code> 运行的。
<code>lib.rs</code>	库默认入口点。从这里开始找 <code>my_crate::f</code> 。
 <code>tests/</code>	集成测试，用 <code>cargo test</code> 运行。单元测试通常直接写在 <code>src/</code> 里。
<code>.rustfmt.toml</code>	自定义 <code>cargo fmt</code> 格式。
<code>.clippy.toml</code>	特定 <code>clippy lints</code> 配置。

文件/目录	代码
Cargo.toml	主项目配置。定义依赖、选项等.....
Cargo.lock	可复现构建的依赖详情。建议为应用程序加入 git 管理，库则不要。

* stable 可以考虑 Criterion。

Cargo

Cargo 的常用命令和工具。

命令	说明
cargo init	在最新的版本上创建新项目。
cargo build	调试模式构建项目。（--release 开启所有优化）。
cargo check	检查项目是否可以编译（更快）。
cargo test	运行项目测试。
cargo run	运行项目。仅当生成了二进制文件（main.rs）。
cargo doc --open	生成项目代码和依赖的本地文档。
cargo rustc -- -Zunpretty=X expanded	显示预处理过后的 Rust 代码。特别地，当 X 为： 将展开所有宏.....
cargo +{nightly, stable} ...	以给定的工具链运行命令。比如仅“nightly only”的工具。
rustup docs	打开离线 Rust 文档（包括《Rust 程序设计语言》）。在飞机上也可以编程！

命令如 cargo build 表示 cargo build 或 cargo b 都有效。

rustup 的可选组件。用 rustup component add [tool] 安装。

工具	说明
cargo clippy	额外(lints) 检查通用 API 误用和非惯用代码。🔗
cargo fmt	自动代码格式化。(rustup component add rustfmt) 🔗

更多 cargo 插件可以在[这里](#)找到。

交叉编译

- 检查目标是否支持。
- 安装目标依赖：rustup target install X。
- 安装本地工具链（取决于目标可能需要链接）。

应从目标供应商（Google、Apple 等）获取这些资源。也可能不支持本地宿主环境（比如，Windows 不支持 iOS 工具链）。

某些工具链需要额外的构建步骤（比如 Android 的 make-standalone-toolchain.sh）。

- 修改 ~/cargo/.config 如下：

```
[target.aarch64-linux-android]
linker = "[PATH_TO_TOOLCHAIN]/aarch64-linux-android/bin/aarch64-linux-android-clang"
```


或

```
[target.aarch64-linux-android]
linker = "C:/[PATH_TO_TOOLCHAIN]/prebuilt/windows-x86_64/bin/aarch64-linux-android21-clang.cmd"
```

取决于编译器警告，有时需要设置环境变量。某些平台和配置可能对路径或引号**非常**敏感（比如 \ 对 / ）：

```
set CC=C:\[PATH_TO_TOOLCHAIN]\prebuilt\windows-x86_64\bin\aarch64-linux-android21-clang.cmd
```

✓ 用 `cargo build --target=X` 编译。

编码指南

Rust 惯用法

Java 或 C 的使用者需要转换下思维：

习语	代码
用表达式思考	<code>x = if x { a } else { b };</code> <code>x = loop { break 5 };</code> <code>fn f() -> u32 { 0 }</code>
用迭代器思考	<code>(1..10).map(f).collect()</code> <code>names.iter().filter(x x.starts_with("A"))</code>
用 ? 捕获异常	<code>x = try_something()?;</code> <code>get_option()?.run()?;</code>
使用强类型	<code>enum E { Invalid, Valid { ... } }</code> 之于 <code>ERROR_INVALID = -1</code> <code>enum E { Visible, Hidden }</code> 之于 <code>visible: bool</code> <code>struct Charge(f32)</code> 之于 <code>f32</code>
提供生成器	<code>Car::new("Model T").hp(20).run();</code>
分离实现	泛型 <code>S<T></code> 可以对每个 <code>T</code> 都有不同的实现。 Rust 没有面向对象，但通过 <code>impl</code> 可以实现特化。
Unsafe	尽量避免 <code>unsafe {}</code> ，因为总是会有更快更安全的解决方案的。除了 FFI。
实现 Trait	<code>#[derive(Debug, Copy, ...)]</code> 。根据需 要 实现 <code>impl</code> 。
工具化	利用 <code>clippy</code> 可以提升代码质量。 用 <code>rustfmt</code> 格式化可以帮助别人看懂你的代码。 添加单元测试 <code>BK</code> (<code>#[test]</code>)，确保代码正常运行。 添加文档测试 <code>BK</code> (<code>`` my_api::f() ``</code>)，确保文档匹配代码。
文档	以文档注解的 API 可显示在 <code>docs.rs</code> 上。 不要忘记在开始加上 总结句 和 例程 。 如果有这些也加上： Panics 、 Errors 、 Safety 、 Abort 和 未定义行为 。

语法 ¹	说明
<code>Rc::new(); x.await; rc();</code>	非 <code>Send</code> 类型拒绝实现 <code>impl Future</code> 。兼容性差。

¹ 假设 `s` 是非局部可临时进入无效状态的任意变量。 `TL` 是局部保存的线程，`async {}` 包含未保证执行约束的代码。

² 因为当 `Future` 被析构后，`Drop` 可能会在任何情况下执行到。如果必须使 `.await` 保持在这种状态的话，考虑用 `drop guard` 来清理或者修复应用程序状态。

闭包 API

这些子 trait 的关系是 `Fn : FnMut : FnOnce`。即一个实现了 `Fn` 的闭包，也同时实现了 `FnMut` 和 `FnOnce`。同样地，实现了 `FnMut` 的闭包，也必然实现了 `FnOnce`。

从调用者的角度来看这意味着：

签名	函数 <code>g</code> 可以调用什么	函数 <code>g</code> 可以接受什么参数
<code>g<F: FnOnce>(f: F)</code>	... <code>f()</code> 一次	<code>Fn</code> , <code>FnMut</code> , <code>FnOnce</code>
<code>g<F: FnMut>(mut f: F)</code>	... <code>f()</code> 多次	<code>Fn</code> , <code>FnMut</code>
<code>g<F: Fn>(f: F)</code>	... <code>f()</code> 多次	<code>Fn</code>

注意，对调用者来说，如何**确定** `Fn` 闭包，是最为严格的。但是一个**包含** `Fn` 的闭包，对调用者来说，是对任意函数都最兼容的。

站在定义闭包的角度来看：

闭包	实现 [*]	说明
<code> { moved_s; }</code>	<code>FnOnce</code>	调用者必须放弃 <code>moved_s</code> 的所有权。
<code> { &mut s; }</code>	<code>FnOnce</code> , <code>FnMut</code>	允许 <code>g()</code> 改变调用者的局部状态 <code>s</code> 。
<code> { &s; }</code>	<code>FnOnce</code> , <code>FnMut</code> , <code>Fn</code>	可能不会导致状态改变，但可能会共享和重用 <code>s</code> 。

^{*} Rust 偏向于以索引捕获（在调用者视角上最“兼容” `Fn` 的闭包），但也可以用 `move || { }` 语法通过复制或者移动捕获相关环境变量。。

这会带来如下优势和劣势：

要求	优势	劣势
<code>F: FnOnce</code>	容易满足调用者。	仅用一次， <code>g()</code> 只会调用 <code>f()</code> 一次。
<code>F: FnMut</code>	允许 <code>g()</code> 改变调用者状态。	调用者不能在 <code>g()</code> 期间重用捕获。
<code>F: Fn</code>	可同时存在多个。	最难由调用者生成。

理解生命周期

生命周期有时难以理解。这里给出一个简易指南，指导 C 经验者如何阅读和翻译包含生命周期声明的代码。

写法	念法
<code>let s: S = S(0)</code>	一块 <code>s</code> 大小的空间，叫做 <code>s</code> 。包含一个 <code>S(0)</code> 的值。 如果用 <code>let</code> 声明，这块空间将生存在栈上。 ¹ 通常， <code>s</code> 表示 s 的位置 和 s 里面的值 。 作为位置， <code>s = S(1)</code> 表示分配值 <code>S(1)</code> 到位置 <code>s</code> 。 作为值， <code>f(s)</code> 表示用 <code>s</code> 里面的值调用 <code>f</code> 。

写法	念法
<code>&'a S</code>	<p>当明确表示其位置（地址）时使用 <code>&s</code>。</p> <p>当明确表示用于保存这样一个位置的位置时，使用 <code>&s</code>。</p> <p><code>&s</code> 是（至少）可以保存一个地址的位置，叫做引用。</p> <p>存在这里的任意地址，都一定指向有效的 <code>s</code>。</p> <p>存在这里的任意地址必须被证明其生命周期不短于(outlive) <code>'a</code>。</p> <p>换言之，<code>&s</code> 部分限定了这里包含的任意地址的范围。</p> <p><code>&'a</code> 部分限定了地址至少生存的时间。</p> <p>包含的位置与生命周期并不相干，但总比它短。</p> <p><code>'a</code> 仅在编译时期可见，由完全的静态分析得出。</p>
<code>&s</code>	<p>有时 <code>'a</code> 会被省略（或者不能被指定），但它仍然存在。</p> <p>在方法体中，生命周期可以自动确定。</p> <p>在签名中，生命周期可以被“省略”（自动标注）。</p>
<code>&s</code>	<p>会产生位置 <code>s</code> 的实际地址，叫做“借用”。</p> <p>一旦 <code>&s</code> 产生，位置 <code>s</code> 就会进入被借用状态。</p> <p>检查是否处于被借用状态取决于编译期分析。</p> <p>分析取决于可能的地址传播路径。</p> <p>只要任意 <code>&s</code> 存在，那么 <code>s</code> 就不能被直接改变。</p> <p>例如，<code>let a = &s; let b = a;</code> 中的 <code>b</code>。</p> <p>借用 <code>s</code> 会持续到 <code>&s</code> 最后一次使用，并非 <code>&s</code> 析构时。</p>
<code>&mut s</code>	<p>同上，但产生可变借用。</p> <p><code>&mut</code> 允许借用的所有者（地址）改变 <code>s</code> 的内容。</p> <p>这里不是指 <code>s</code> 中的值，而是 <code>s</code> 的位置本身被借用了。</p>

¹ 与上面的数据结构一节比较：对于同步代码显然如此。对于 `async` 来说，“栈帧”可能存储在堆上，并且取决于运行时的实现。

阅读函数或类型签名也有套路：

写法	念法
<code>S<'a> {}</code>	<p>标明 <code>S</code> 会包含*至少一个地址（如引用）。</p> <p><code>'a</code> 由该结构体的使用者自动确定。</p> <p><code>'a</code> 会尽可能选择最小的范围。</p>
<code>f<'a>(x: &'a T) -> &'a S</code>	<p>标明函数会接受一个地址（如引用）。</p> <p>.....也会返回一个地址。</p> <p><code>'a</code> 由调用者自动确定。</p> <p><code>'a</code> 会尽可能选择最小的范围。</p> <p><code>'a</code> 会由调用方选择同时满足输入和输出的。</p> <p>最重要的是，生命周期名称会传播借用状态！</p> <p>所以当 <code>'a</code> 的结果地址被使用了之后，<code>'a</code> 的输入地址会被锁定。</p> <p>这里，当<code>let s = f(&x)</code> 的 <code>s</code> 存在，<code>x</code> 会被标记为“已借用”。</p>
<code><'a, 'b: 'a></code>	<p><code>S<></code> 和 <code>f<></code> 里面声明的生命周期也可以有范围。</p> <p><code><'a, 'b></code> 部分表示类型至少持有两个地址。</p>

写法	念法
----	----

'b: 'a 部分表示**生命周期范围**，意为 'b 必须不短于（**outlive**） 'a.

Ⓔ'b X 中的任意地址的生存时间必须和 Ⓔ'a Y 中的至少一样长。

* 技术上，结构体可能不持有任何数据（比如使用 'a 上的 PhantomData 或者函数指针），但仍然保留 'a 用于交流和约束函数的引用确定生命周期。

Unsafe, Unsound, Undefined

Unsafe 导致 unsound，unsound 导致 undefined，undefined 是一切原力的阴暗面。

<div>Unsafe 代码</div> <div><div>Unsafe 代码</div><ul style="list-style-type: none">标记为 <code>unsafe</code> 的代码有特权。比如，解引用裸指针，或调用其他 <code>unsafe</code> 函数。这是一份特殊的作者必须给编译器的承诺，编译器会相信你。<code>unsafe</code> 代码自身并非有害，但危险的是 FFI 使用方或者异常的数据结构。<pre>// `x` must always point to race-free, valid, aligned, initialized u8 memory. unsafe fn unsafe_f(x: *mut u8) { my_native_lib(x); }</pre></div>	<div>Undefined 行为</div> <div><div>未定义行为 (UB)</div><ul style="list-style-type: none">如前所述，<code>unsafe</code> 代码意味着对编译器的特殊承诺（否则它就不需要是 <code>unsafe</code> 的了）。不遵守承诺会使编译器产生错误的代码，执行错误的代码会导致未定义行为。在触发未定义行为之后，任何事情都可能发生。这种不知不觉的影响可能1)难以捉摸，2)明显远离事发现场，或3)只有在某些条件下才会被发现。一个表面上可以运行的程序（包括任意数量的单元测试），并不能证明含有未定义行为的代码不会因为一些偶然原因而失败。含有未定义行为的代码在客观上是危险的、无效的，根本不应该存在。<pre>if maybe_true() { let r: &u8 = unsafe { &*ptr::null() }; // 一旦运行，整个程序都会处于未定义状态。 } else { // 尽管这一行看似什么都没干，程序可能两条路径 println!("the spanish inquisition"); // 都运行了，然后破坏掉数据，或者发生别的。 }</pre></div>
--	--

Unsound 代码

Unsound 代码

- 任何会由于用户输入而导致 *safe* Rust 产生未定义行为的都是 **unsound**（不健全）的（尽管仅仅可能是理论上的）。
- 比如 `unsafe` 代码可能违反上述承诺而产生未定义行为。
- Unsound 代码对稳定性和安全性造成风险，且违背了大部分 Rust 用户的基本假设。

```
fn unsound_ref<T>(x: &T) -> &u128 {      // Signature looks safe to users. Happens to
be                                         // be
    unsafe { mem::transmute(x) }         // ok if invoked with an &u128, UB for practice
}                                         // everything else.
```

负责任地使用 Unsafe

- 除非非用不可，不要使用 `unsafe`。
- 遵循《死灵书》、《Unsafe 指南》，**永远保证所有**的安全不变性，**绝不**引发未定义行为。
- 最小化 `unsafe` 用例，封装成易于评审的小的、优雅模块。
- 每个 `unsafe` 用例应当同时提供关于其安全性的纯文本理由提要。

API 稳定性

这些更改会破坏客户端代码，请比较 **RFC 1105**。主要更改(●)一定导致破坏，一般更改(○)可能导致破坏：

Crate

- 编写一个 *stable* 的 crate 但却依赖了 *nightly*。
- 修改了 Cargo 的功能（比如添加或移除功能）

模块

- 重命名、移动、移除任何公开项。
- 添加新的公开项，因为 `use your_crate::*` 可能会破坏现有代码。

结构体

- 当所有字段都为公开时添加私有字段。
- 当没有私有字段时添加公开字段。
- 当至少有一个字段时添加或移除私有字段（在更改前或更改后）。
- 将有私有字段（至少有一个字段）的元组结构转换到普通结构，或反之。

枚举

- 添加新的枚举变体。

枚举

- ☒ 为枚举变体添加新字段。

Trait

- ☒ 添加非默认项，将会破坏已有的 `impl T for S {}`。
- ☒ 任何不必要的项签名修改，都会影响到使用者或者实现方。
- ☐ 添加一个默认项，可能会和另一个 trait 产生歧义。
- ☐ 添加默认类型参数。

Trait

- ☒ 实现任何“基本”trait。**不去**实现一个基本 trait 是一种最基本的承诺。
- ☐ 实现任何非基本的 trait，可能会导致歧义。

固有实现

- ☐ 添加内部项，可能会导致客户端倾向于调用这个 trait 的 fn 而导致编译错误。

类型定义签名

- ☒ 强约束（如 `<T>` 到 `<T: Clone>`）。
- ☐ 弱约束。
- ☐ 添加默认类型参数。
- ☐ 泛型归纳。

函数签名

- ☒ 添加或移除参数。
- ☐ 引入新的类型参数。
- ☐ 泛型归纳。

行为更改

- ☒ / ☐ 改变语义可能不会导致编译器错误，但可能会使用户产生错误的逻辑。