

Rust

引用

可变引用常见错误

```
1 //可变引用与可变变量产生数据竞争，导致编译失败。
2 //error[E0499]: cannot borrow `a` as mutable more than once at a time
3 fn main() {
4     let mut a = String::from("abc");
5     let b = &mut a;
6     a.push_str("ghi");
7     b.push_str("def");
8     println!("{}", a);
9 }
10
```

&和ref

解引用通过实现trait Deref

```
1 fn main() {
2     let ref a = 100_i32;
3     //等于
4     let a = &100_i32;
5 }
6
```

字符串

```
1 let mut s2: String = String::from(s);
2 s.as_bytes();
3 s.bytes();
4 s.chars();
5 s.chars().count();
6 s.chars();
7 s.contains("Hello");
8 s.clear();
9 s.parse::<i32>().unwrap();
10 s.trim();
11
```

向量

```
1 let mut vect: Vec<i32> = vec![100, 200, 300, 400, 500];
2 vect.get(0);
3 vect[0];
4 vect[0] = 120;
5 vect.push(600);
```

```

6 vect.insert(2, 250);
7 vect.remove(0);
8 vect.append(&mut vec![700, 800]);
9 //vect一般是引用 (&vect或&mut vect)
10 for i in &vect {
11     println!("{i}");
12 }
13 vect.pop();
14 vect.clear();
15

```

哈希表

```

1 let mut hm: HashMap<String, i32> = HashMap::new();
2 hm.insert(String::from("k"), 100);
3 hm.get("k").cloned().unwrap();
4 hm.entry(String::from("k")).or_insert(100);
5 hm.remove("k");
6 hm.remove_entry("k").unwrap();
7

```

迭代器

```

1 let iter: Copied<Iter<'_, i32>> = vect.iter().cloned();
2 iter.clone().filter(|x| *x == 300);
3 iter.clone().map(|x| x * x);
4 iter.clone().collect::<Vec<i32>>();
5 iter.clone().chain(vec![600, 700, 800].iter().cloned());
6 iter.clone().enumerate();
7 iter.clone().count();
8

```

Result/Option处理

```

1 let err: Result<String, Error> = read_to_string("a.txt");
2 err.unwrap();
3 err.expect("...");
4 err.unwrap_or(String::from("Error!"));
5 err.unwrap_or_default();
6 err.unwrap_or_else(|e| format!("Error:{}", e.to_string()));
7 err?;
8 //Result特有
9 err.is_ok();
10 err.is_err();
11 //Option特有
12 opt.is_some();
13 opt.is_none();
14

```

Enum

```

1 enum A {

```

```

2     X,
3     Y,
4     Z,
5 }
6
7 enum B {
8     X(i32),
9     Y(f32),
10    Z,
11 }
12
13 enum C {
14     X = 0,
15     Y = 1,
16     Z = 2,
17 }
18
19 fn main() {
20     let ax = A::X;
21     let bx = B::X(20);
22     //way1:
23     match ax {
24         A::X => println!("x"),
25         A::Y => println!("y"),
26         _ => println!("neither X nor Y, it may be C"),
27     }
28     match bx {
29         B::X(v) => println!("x:{}", v),
30         B::Y(v) => println!("y:{}", v),
31         _ => println!("neither X nor Y, it may be C"),
32     }
33     if let B::X(v) = bx {
34         println!("x:{}", v)
35     }
36 }
37

```

Derive

```

1  /// Debug: 通过实现 Debug trait, 可以使用 println!("{:?}", my_struct) 来打印结构
   体的调试信息。
2  /// Copy: 可以按位复制（不含引用等），可以在赋值时自动复制，所有实现了 Copy 的类型都必须
   实现 Clone。
3  /// Clone: 通过实现 Clone trait, 可以使用 my_struct.clone() 创建结构体的克隆副本。
4  /// PartialEq 和 Eq: 通过实现 PartialEq trait, 可以进行结构体的部分相等性比较，而 Eq
   trait 则实现了完全相等性比较。
5  ///      PartialEq实现等于和不等于判断，Eq没有方法，但只要实现（必须也实现PartialEq），
   就满足：
6  ///          1.a == a;
7  ///          2.如果a == b, 那么b == a;
8  ///          3.如果a == b且b == c, 那么a == c。
9  /// PartialOrd 和 Ord: 通过实现 PartialOrd trait, 可以对结构体进行部分有序性比较，而
   Ord trait 实现了完全有序性比较。
10 #[derive(Debug,Clone)]
11 struct MyStruct {
12     // 结构体字段

```

```
13 }  
14
```

类型转换

```
1 let decimal = 65.4321_f32;  
2 let integer = decimal as u8;  
3 let character = integer as char;
```

类型别名

```
1 type NanoSecond = u64;  
2 //泛型别名  
3 type AliasedResult<T> = Result<T, ParseIntError>;
```

关联类型

```
1 struct Container(i32, i32);  
2  
3 // `A` 和 `B` 在 trait 里面通过 `type` 关键字来定义。  
4 // （注意：此处的 `type` 不同于为类型取别名时的 `type`）。  
5 trait Contains {  
6     type A;  
7     type B;  
8  
9     // 这种语法能够泛型地表示这些新类型。  
10    fn contains(&self, _: &Self::A, _: &Self::B) -> bool;  
11    fn first(&self) -> i32;  
12    fn last(&self) -> i32;  
13 }  
14  
15 impl Contains for Container {  
16     type A = i32;  
17     type B = i32;  
18  
19     // `&Self::A` 和 `&Self::B` 在这里也是合法的类型。  
20     fn contains(&self, number_1: &i32, number_2: &i32) -> bool {  
21         (&self.0 == number_1) && (&self.1 == number_2)  
22     }  
23  
24     fn first(&self) -> i32 {  
25         self.0  
26     }  
27  
28     fn last(&self) -> i32 {  
29         self.1  
30     }  
31 }  
32  
33 fn difference<C: Contains>(container: &C) -> i32 {  
34     // C类型  
35     // C::A  
36     container.last() - container.first()
```

```

37 }
38
39 // <C: Contains<A = i32>>限制关联类型
40 fn difference2<C: Contains<A = i32>>(container: &C) -> i32 {
41     container.last() - container.first()
42 }
43
44 fn main() {
45     let number_1 = 3;
46     let number_2 = 10;
47
48     let container = Container(number_1, number_2);
49
50     println!(
51         "Does container contain {} and {}: {}",
52         &number_1,
53         &number_2,
54         container.contains(&number_1, &number_2)
55     );
56     println!("First number: {}", container.first());
57     println!("Last number: {}", container.last());
58
59     println!("The difference is: {}", difference(&container));
60 }
61

```

运算符重载

```

1  use std::ops;
2
3  struct Foo;
4  struct Bar;
5
6  #[derive(Debug)]
7  struct FooBar;
8
9  #[derive(Debug)]
10 struct BarFoo;
11
12 // `std::ops::Add` trait 用来指明 `+` 的功能，这里我们实现 `Add<Bar>`，它是用于
13 // 把对象和 `Bar` 类型的右操作数（RHS）加起来的 `trait`。
14 // 下面的代码块实现了 `Foo + Bar = FooBar` 这样的运算。
15 impl ops::Add<Bar> for Foo {
16     type Output = FooBar;
17
18     fn add(self, _rhs: Bar) -> FooBar {
19         println!("> Foo.add(Bar) was called");
20
21         FooBar
22     }
23 }
24
25 // 通过颠倒类型，我们实现了不服从交换律的加法。
26 // 这里我们实现 `Add<Foo>`，它是用于把对象和 `Foo` 类型的右操作数加起来的 `trait`。
27 // 下面的代码块实现了 `Bar + Foo = BarFoo` 这样的运算。
28 impl ops::Add<Foo> for Bar {

```

```

29     type Output = BarFoo;
30
31     fn add(self, _rhs: Foo) -> BarFoo {
32         println!("> Bar.add(Foo) was called");
33
34         BarFoo
35     }
36 }
37
38 fn main() {
39     println!("Foo + Bar = {:?}", Foo + Bar);
40     println!("Bar + Foo = {:?}", Bar + Foo);
41 }
42

```

自动推断类型

```

1 //常用于泛型参数中:
2 fn main() {
3     let x: Vec<_> = (0..10).collect();
4 }
5

```

常量

```

1 ///const: 不可改变的值（通常使用这种）。
2 ///static: 具有 'static 生命周期的，可以是可变的变量（static mut）。
3 ///有个特例就是 "string" 字面量。它可以不经改动就被赋给一个 static 变量，因为它的类型标
  记: &'static str 就包含了所要求的生命周期 'static。其他的引用类型都必须特地声明，使之拥
  有 'static 生命周期。这两种引用类型的差异似乎也无关紧要，因为无论如何，static 变量都得显
  式地声明。
4 static A: &'static str = "Rust";
5 static mut B: &'static str = "Rust";
6 const C: &str = "Rust";

```

From与Into

`From` trait 允许一种类型定义“怎么根据另一种类型生成自己”，因此它提供了一种类型转换的简单机制。

`Into` trait 就是把 `From` trait 倒过来而已。也就是说，如果你为你的类型实现了 `From`，那么同时你也就免费获得了 `Into`。

```

1 use std::convert::From;
2
3 #[derive(Debug)]
4 struct Number {
5     value: i32,
6 }
7
8 impl From<i32> for Number {
9     fn from(item: i32) -> Self {
10         Number { value: item }
11     }

```

```

12 }
13
14 fn main() {
15     // 都是自动类型判断
16     let num = Number::from(30);
17     let num: Number = 30.into();
18     println!("My number is {:?}", num);
19 }
20

```

`TryFrom` 和 `TryInto` trait 用于易出错的转换，也正因如此，其返回值是 `Result` 型。

要把任何类型转换成 `String`，您应该实现 `fmt::Display` trait，它会自动提供 `ToString`，并且还可以用来打印类型，就像 `print!` 一节中讨论的那样。See: [ToString](#), [fmt::Display](#), [FromStr](#)

文件分层

```

1  .
2  ├── Cargo.lock
3  ├── Cargo.toml
4  ├── src/
5  │   ├── lib.rs
6  │   ├── main.rs
7  │   └── bin/
8  │       ├── named-executable.rs
9  │       ├── another-executable.rs
10 │       └── multi-file-executable/
11 │           ├── main.rs
12 │           └── some_module.rs
13 ├── benches/
14 │   ├── large-input.rs
15 │   └── multi-file-bench/
16 │       ├── main.rs
17 │       └── bench_module.rs
18 ├── examples/
19 │   ├── simple.rs
20 │   └── multi-file-example/
21 │       ├── main.rs
22 │       └── ex_module.rs
23 └── tests/
24     ├── some-integration-tests.rs
25     └── multi-file-test/
26         ├── main.rs
27         └── test_module.rs
28

```

<https://rustwiki.org/zh-CN/rust-by-example/mod/split.html>

Drop

<https://rustwiki.org/zh-CN/rust-by-example/trait/drop.html>

```

1 struct Droppable {
2     name: &'static str,

```

```

3 }
4
5 // 这个简单的 `drop` 实现添加了打印到控制台的功能。
6 impl Drop for Droppable {
7     fn drop(&mut self) {
8         println!("> Dropping {}", self.name);
9     }
10 }
11
12 fn main() {
13     let _a = Droppable { name: "a" };
14
15     // 代码块 A
16     {
17         let _b = Droppable { name: "b" };
18
19         // 代码块 B
20         {
21             let _c = Droppable { name: "c" };
22             let _d = Droppable { name: "d" };
23
24             println!("Exiting block B");
25         }
26         println!("Just exited block B");
27
28         println!("Exiting block A");
29     }
30     println!("Just exited block A");
31
32     // 变量可以手动使用 `drop` 函数来销毁。
33     drop(_a);
34     //或std::mem::drop(_a);
35
36     println!("end of the main function");
37
38     // `_a` 不会在这里再次销毁，因为它已经被（手动）销毁。
39 }
40

```

Crate

<https://rustwiki.org/zh-CN/rust-by-example/crates.html>

```

1 # 默认情况下，库会使用 crate 文件的名称，前面加上 “lib” 前缀，但这个默认名称可以使用
  crate_name 属性 覆盖。
2 rustc --crate-type=lib rary.rs
3 rustc main.rs --extern rary=library.rlib --edition=2018

```

```

1 [package]
2 # See: https://doc.rust-lang.org/cargo/reference/manifest.html
3 name = "hello_world"
4 version = "0.1.0"
5 edition = "2018"
6 sql = "0.4.3"# <-- Add dependency here

```


cfg

`#[cfg()]` 是 Rust 中的一个属性，用于根据配置条件来选择性地包含或排除代码。

`#[cfg()]` 属性中使用的逻辑运算符有以下几种：

1. `all(expr1, expr2, ...)`：逻辑与运算符，要求所有条件表达式都为真才返回真。例如：`#[cfg(all(feature = "foo", target_os = "linux"))]` 表示只有在启用 "foo" 功能并且目标操作系统是 Linux 时条件成立。
2. `any(expr1, expr2, ...)`：逻辑或运算符，只要有任一条件表达式为真就返回真。例如：`#[cfg(any(feature = "foo", feature = "bar"))]` 表示只要启用 "foo" 或 "bar" 中的任意一个功能时条件成立。
3. `not(expr)`：逻辑非运算符，对条件表达式取反。例如：`#[cfg(not(debug_assertions))]` 表示只有在非调试断言模式下条件成立。

```
1  #[cfg(target_os = "linux")]
2  fn only_on_linux() {
3      // 仅在 Linux 系统上编译和执行的代码
4  }
5
6  #[cfg(all(unix, not(target_os = "linux")))]
7  fn on_unix_but_not_linux() {
8      // 仅在 Unix 系统但不是 Linux 上编译和执行的代码
9  }
10
11 #[cfg(any(windows, target_os = "macos"))]
12 fn on_windows_or_macos() {
13     // 仅在 windows 或 macOS 上编译和执行的代码
14 }
15
16 #[cfg(not(debug_assertions))]
17 fn when_not_debug_assertions() {
18     // 仅在非调试断言模式下编译和执行的代码
19 }
20
21 #[cfg(feature = "my_feature")]
22 fn with_my_feature_enabled() {
23     // 仅在启用 "my_feature" 功能时编译和执行的代码
24 }
25
```

```
1  if cfg!(target_os = "linux") {
2      println!("Yes. It's definitely linux!");
3  } else {
4      println!("Yes. It's definitely *not* linux!");
5  }
6
```

宏

```

1 macro_rules! $name {
2     $rule0 ;
3     $rule1 ;
4     // ...
5     $ruleN ;
6 }

```

\$rule:

`($pattern:$identifier) => {$expansion}`

\$identifier:

- `item`: 条目, 比如函数、结构体、模组等。
- `block`: 区块(即由花括号包起的一些语句加上/或是一项表达式)。
- `stmt`: 语句
- `pat`: 模式
- `expr`: 表达式
- `ty`: 类型
- `ident`: 标识符
- `path`: 路径 (例如 `foo, ::std::mem::replace, transmute::<_, int>, ...`)
- `meta`: 元条目, 即被包含在 `#[...]` 及 `#![...]` 属性内的东西。
- `tt`: 标记树

重复

- `$` 是字面标记。
- `(...)` 代表了将要被重复匹配的模式, 由小括号包围。
- `sep` 是一个可选的分隔标记。常用例子包括, 和 `;`。
- `rep` 是重复控制标记。当前有两种选择, 分别是 `*` (代表接受0或多次重复)以及 `+` (代表1或多次重复)。目前没有办法指定“0或1”或者任何其它更加具体的重复计数或区间。

```

1 macro_rules! vec_strs {
2     (
3         // 重复开始:
4         $(
5             // 每次重复必须有一个表达式...
6             $element:expr
7         )
8         // ...重复之间由“,”分隔...
9         ,
10        // ...总共重复0或多次.
11        *
12    ) => {
13        // 为了能包含多条语句,
14        // 我们将扩展部分包裹在花括号中...
15        {
16            let mut v = Vec::new();
17            // 重复开始:
18            $(
19                // 每次重复将包含如下元素, 其中
20                // “$element”将被替换成其相应的展开...
21                v.push(format!("{}", $element));
22            )*

```

```

23         v
24     }
25 };
26 }
27

```

智能指针

- `Box<T>`：它提供了最简单的堆资源分配方式，单所有权。Box类型拥有其中的值，并且可用于保存结构体中的值，或者从函数返回它们。
- `Rc<T>`：它用于引用计数。每当获取新引用时，计数器会执行递增操作，并在用户释放引用时对计数器执行递减操作。当计数器的值为零时，该值将被移除。
- `Arc<T>`：它用于原子引用计数。这与之前的类型类似，但具有原子性以保证多线程的安全性。
- `Cell<T>`：它为我们提供实现了Copy特征的类型的内部可变性。换句话说，我们有可能获得多个可变引用。
- `RefCell<T>`：它为我们提供了类型的内部可变性，并且不需要实现Copy特征。它用于运行时的锁定以确保安全性。

RefCell内部可变性

```

1  let a = RefCell::new(100_i32);
2  //编译失败（RefCell不能解引用）：
3  // println!("{}", *a);
4  println!("{}", *a.borrow());
5  //编译失败（a不可变）：
6  // *a = 120;
7  *a.borrow_mut() = 120;
8  //Panic（同时两个可变引用）：
9  let b = a.borrow_mut();
10 let c = a.borrow_mut();

```

多线程用 `Arc<T>` + `Mutex<T>`。

RefCell+Rc: 多个所有者 并且 可以修改值

```

1  #[derive(Debug)]
2  enum List {
3      Cons(Rc<RefCell<i32>>, Rc<List>),
4      Nil,
5  }
6
7  use crate::List::{Cons, Nil};
8  use std::cell::RefCell;
9  use std::rc::Rc;
10
11 fn main() {
12     let value = Rc::new(RefCell::new(5));
13     let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
14     let b = Cons(Rc::clone(&value), Rc::clone(&a));
15     *value.borrow_mut() += 10;
16     println!("{:?}\n{:?}", a, b);
17 }
18

```

强/弱引用

```
1 let strong = Rc::new(100_i32);
2 //由强引用创建弱引用。
3 let weak = Rc::downgrade(&strong);
4 //由弱引用创建强引用。
5 //如果弱引用已无效，就返回None。
6 let strong = weak::upgrade(&weak).unwrap();
7
```

编译

编译模式

```
1 cargo build
2 cargo build --release
3
```

编译目标类型

[链接\(linkage\) - Rust 参考手册 中文版\(rustwiki.org\)](#)

- `--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=dllib` 或 `#[crate_type = "dllib"]` - 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=cdllib` 或 `#[crate_type = "cdllib"]` - 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。

Cargo.toml

字段含义

See [The Manifest Format - The Cargo Book \(rust-lang.org\)](https://rust-lang.org/cargo/reference/the-manifest-format.html).

[[xxx]]双中括号表示可以有多个

- [cargo-features](#) — 不稳定的、仅限夜间的功能。
- [\[package\]](#) — 定义包。
 - [name](#) — 包的名称。
 - [version](#) — 包的版本。
 - [authors](#) — 包的作者。
 - [edition](#) — Rust 版本。
 - [rust-version](#) — 支持的最低 Rust 版本。
 - [description](#) — 包的描述。
 - [documentation](#) — 软件包文档的 URL。
 - [readme](#) — 软件包的 README 文件的路径。
 - [homepage](#) — 软件包主页的 URL。
 - [repository](#) — 包源存储库的 URL。
 - [license](#) — 软件包许可证。
 - [license-file](#) — 许可证文本的路径。
 - [keywords](#) — 包的关键词。
 - [categories](#) — 包的类别。
 - [workspace](#) — 包的工作区路径。
 - [build](#) — 包构建脚本的路径。
 - [links](#) — 包链接的本机库的名称。
 - [exclude](#) — 发布时要排除的文件。
 - [include](#) — 发布时要包含的文件。
 - [publish](#) — 可用于阻止发布包。
 - [metadata](#) — 外部工具的额外设置。
 - [default-run](#) — 由 [cargo run](#) 运行的默认二进制文件。
 - [autobins](#) — 禁用二进制自动发现。
 - [autoexamples](#) — 禁用示例自动发现。
 - [autotests](#) — 禁用测试自动发现。
 - [autobenches](#) — 禁用工作台自动发现。
 - [resolver](#) — 设置要使用的依赖关系解析程序。
- 目标表: (有关设置, 请参阅[配置](#))
 - [\[\[lib\]\]](#) — 库目标设置。
 - [\[\[bin\]\]](#) (<https://doc.rust-lang.org/cargo/reference/cargo-targets.html#binaries>) — 二进制目标设置。
 - [\[\[example\]\]](#) (<https://doc.rust-lang.org/cargo/reference/cargo-targets.html#examples>) — 目标设置示例。

- `[[test]]` (<https://doc.rust-lang.org/cargo/reference/cargo-targets.html#tests>) — 测试目标设置。
- `[[bench]]` (<https://doc.rust-lang.org/cargo/reference/cargo-targets.html#benchmark>) — 基准目标设置。
- 依赖项表：
 - `[dependencies]` — 包库依赖项。
 - `[dev-dependencies]` — 示例、测试和基准测试的依赖项。
 - `[build-dependencies]` — 构建脚本的依赖关系。
 - `[target]` — 特定于平台的依赖项。
- `[badges]` — 要在注册表中显示的锁屏提醒。
- `[features]` — 条件编译功能。
- `[lints]` — 为此软件包配置 linter。
- `[patch]` — 覆盖依赖项。
- `[replace]` — 覆盖依赖项（已弃用）。
- `[profile]` — 编译器设置和优化。
- `[workspace]` — 工作区定义。

默认

```

1  [package]
2  name = "tool_name"
3  version = "0.1.0"
4  edition = "2021"
5
6  # See more keys and their definitions at https://doc.rust-
   lang.org/cargo/reference/manifest.html
7
8  [dependencies]
9
```

常用

```

1  [package]
2  name = "hello_opensource"
3  version = "0.1.0"
4  authors = ["user<user@mail.com>"]
5  edition = "2018"
6
7  # See more keys and their definitions at https://doc.rust-
   lang.org/cargo/reference/manifest.html
8
9  [dependencies]
10 rand = "0.3.14"
11
```

更新依赖

```
1 cargo update [-p rand]
```

描述

```

1 [package]
2 #-snip-
3 description = "A fun game where you guess what number the computer has
  chosen."

```

license

```

1 [package]
2 #-snip-
3 license = "MIT"

```

bin

```

1 [[bin]]
2 #-snip-
3 name = "tool_bin_name"# 可执行文件名源，会查找src/bin/tool_bin_name.rs或
  src/bin/tool_bin_name/main.rs
4 test = false
5 bench = false

```

lib

```

1 [lib]
2 name = "foo"           # 目标名
3 path = "src/lib.rs"    # 目标的源文件
4 crate-type = ["lib"]   # 要生成的crate类型，可选的有bin, lib, rlib, dylib,
  cdylib, staticlib, and proc-macro
5 test = true            # 默认为已测试
6 doc = true             # 是否包含在默认情况下，由Cargo Doc生成文档
7 edition = "2015"       # 定义目标将使用的 Rust 版本，通常不应设置此字段
8 bench = true           # Is benchmarked by default.
9 plugin = false         # Used as a compiler plugin (deprecated).
10 proc-macro = false     # Set to `true` for a proc-macro library.
11 harness = true        # Use libtest harness.
12 required-features = [] # Features required to build this target (N/A for
  lib).
13

```

优化等级

```

1 #opt-level 设置控制 Rust 会对代码进行何种程度的优化。这个配置的值从 0 到 3。越高的优化级
  别需要更多的时间编译，所以如果你在进行开发并经常编译，可能会希望在牺牲一些代码性能的情况下减
  少优化以便编译得快一些。因此 dev 的 opt-level 默认为 0。当你准备发布时，花费更多时间在编
  译上则更好。只需要在发布模式编译一次，而编译出来的程序则会运行很多次，所以发布模式用更长的编
  译时间换取运行更快的代码。这正是为什么 release 配置的 opt-level 默认为 3。
2 [profile.dev]
3 opt-level = 0
4
5 [profile.release]
6 opt-level = 3
7

```

去除debug信息

```
1 [profile.release]
2 # 可选"none", "debuginfo"和"symbols",
3 # 或者是"true"和"false",
4 # 默认strip = "none"
5 strip = "debuginfo"
6
```

工作空间

See [Cargo 工作空间 - Rust 程序设计语言 简体中文版 \(kaisery.github.io\)](https://kaisery.github.io/cargo-workspace-zh-cn/)

汇编ASM

See [内联汇编 - Rust语言圣经\(Rust Course\)](https://kaisery.github.io/rust-course-zh-cn/)

```
1 use std::arch::asm;
2
3 fn main() {
4     let mut x: u64 = 3;
5     unsafe {
6         //out表示输出，in表示输入，inout表示既输入又输出（可以保证使用同一个寄存器来完成任务）。
7         //reg表示让编译器自动选择合适的寄存器。
8         //可以像格式化字符串一样用{0}等复用。
9         asm!("add {0}, 5", inout(reg) x);
10    }
11    println!("{}", x);
12 }
13
```