# Rust语言实战

通过有挑战性的示例、练习题、实践项目来提升 Rust 水平，建立从入门学习到上手实战的直通桥梁


*Rust语言实战* 的目标是通过大量的实战练习帮助大家更好的学习和上手使用 Rust 语言。书中的练习题非常易于使用：你所需的就是在线完成练习，并让它通过编译。

## 在线阅读

- https://zh.practice.rs

## 本地运行

我们使用 mdbook 构建在线练习题，你也可以下载到本地运行：

```
$ cargo install mdbook
$ cd rust-by-practice && mdbook serve zh-CN/
```

在本地win 10或者linux服务器上运行时，应当使用 -n 参数指定mdbook服务所监听的IP地址（-p 参数指定服务监听的端口，不指定则为默认的3000），以win 10本地运行为例：

```
$ mdbook serve -p 8888 -n 127.0.0.1 zh-CN/
```

## 特性

部分示例和习题借鉴了 Rust By Example, 书中的示例真的非常棒！

尽管它们非常优秀，我们这本书也有自己的秘密武器 :)

- 每个章节分为三个可选部分：示例、练习和实践项目
- 除了示例外，我们还有大量的高质量练习题，你可以在线阅读、修改和编译它们
- 覆盖了 Rust 语言的几乎所有方面：基础语言特性、高级语言特性、async/await 异步编程、多线程、并发原语、性能优化、工具链使用、标准库、数据结构和算法等

- 每一道练习题都提供了解答
- 整体难度相对更高，更加贴近于实战难度: 简单 🌟 ，中等 🌟🌟 ，困难 🌟🌟🌟 ，地狱 🌟🌟🌟🌟

**总之，我们想做的就是解决入门学习后，不知道该如何运用的问题，毕竟对于 Rust 来说，从学习到实战，中间还隔着数个 Go语言 的难度**

# 关于我们

*Rust语言实战* 由 Rust 编程学院倾情打造。

同时我们还提供了一本目前最好也是最用心的开源 Rust 书籍 - Rust语言圣经， 适合从入门到精通所有阶段的学习，欢迎大家阅读使用。

对我们来说，来自读者大大的肯定比什么都重要，因此一个 Github star 要比一杯咖啡更让我们开心，而且现在它在跳楼打折，无需 998 ， 仅需 0 元钱 :)

# 值得学习的小型项目

在国内外的各大 Rust 论坛上，以下问题非常常见:

- 作为 Rust 新手，有哪些项目值得推荐学习?
- 求推荐代码优雅的小型项目
- 有哪些值得推荐的简单、易读的项目

这些问题的答案往往只有一个，那就是实践：做一些优秀的练习题，然后阅读一些小而美的 Rust 项目。

这个恰恰跟本书的目标吻合，因此，我们决定收集一些优秀的资源，并在 *Rust语言实战* 中呈现给大家。

## 1. Ripgrep

以上的问题通常都会伴随着 `ripgrep` 的推荐, 虽然我不认为它是一个小型项目，但是依然非常推荐大家学习，当然，首先你得做好深挖的准备和耐心。

## 2. 教程：构建一个文本编辑器

该教程 `https://www.philippflenker.com/hecto/` 将带领我们从零开始构建一个文本编辑器.

## 3. Ncspot

Ncspot 是一个终端访问的 Spotify 客户端，小巧、简单、良好的代码组织以及异步编程，值得学习.

## 4. 命令行 Rust

这个项目 是书本 `Command-Line Rust(O'Reily)` 的配套项目，可以帮助大家理解该如何更好的编写命令行程序，例如 `head`, `cat`, `ls` 。

## 5. 在 PNG 中隐藏你的秘密

这本书 将带领大家编写一个命令行程序，功能是在 PNG 文件中隐藏一些秘密信息，首要目标是让我们熟悉 Rust 代码。

## 6. 使用 Rust 写一个小型 OS

这个博客系列 会带领大家使用 Rust 语言创建一个小型的操作系统。其中每一篇文章都是一个小的教程并包含完整的代码。

你也可以在以下地址找到完整的源代码。

## 7. CodeCrafters.io: 写一个你自己的 Git， Docker，SQLite 或 Redis

在 CodeCrafters 上，你可以从头开始重新创建您最喜欢的开发人员工具。这是一种掌握 Rust 的实践、最低限度指导的方法，同时欣赏我们每天使用的流行技术的内部结构和文档。

## 8. mini-redis

mini-redis 是一个不完整的 Redis 客户端、服务器实现，由 tokio 官方出品，代码质量非常高，而且有详细的注释，非常适合学习 Rust 和异步编程。

## 9. 使用 Rust 写一个解析器

这本开源书 是一个使用 Rust 语言实现编译型语言的教程。

**To be continued...**

# 变量绑定与解构

## 绑定和可变性

1. 🌟 变量只有在初始化后才能被使用

2. 🌟🌟 可以使用 `mut` 将变量标记为可变

## 变量作用域

3. 🌟 作用域是一个变量在程序中能够保持合法的范围

4. 🌟🌟

## 变量遮蔽( Shadowing )

5. 🌟🌟 若后面的变量声明的名称和之前的变量相同，则我们说：第一个变量被第二个同名变量遮蔽了( shadowing )

6. 🌟🌟 修改一行代码以通过编译

## 未使用的变量

7. 使用以下方法来修复编译器输出的 warning :

- 🌟 一种方法
- 🌟🌟 两种方法

---

注意: 你可以使用两种方法解决，但是它们没有一种是移除 `let x = 1` 所在的代码行

---

## 变量解构

8. 🌟🌟 我们可以将 `let` 跟一个模式一起使用来解构一个元组，最终将它解构为多个独立的变量

提示: 可以使用变量遮蔽或可变性

## 解构式赋值

该功能于 Rust 1.59 版本引入：你可以在赋值语句的左式中使用元组、切片或结构体进行匹配赋值。

9. 🌟🌟

Note: 解构式赋值只能在 Rust 1.59 或者更高版本中使用

答案 在 solutions 下面

# 基本类型

学习资料:

- English: Rust Book 3.2 and 3.3
- 简体中文: Rust语言圣经 - 基本类型

# 数值类型

## 整数

1. 🌟

Tips: 如果我们没有显式的给予变量一个类型，那编译器会自动帮我们推导一个类型

2. 🌟

3. 🌟🌟🌟

Tips: 如果我们没有显式的给予变量一个类型，那编译器会自动帮我们推导一个类型

4. 🌟🌟

5. 🌟🌟

6. 🌟🌟

## 浮点数

7. 🌟

8. 🌟🌟 使用两种方法来让下面代码工作

## 序列Range

9. 🌟🌟 两个目标: 1. 修改 `assert!` 让它工作 2. 让 `println!` 输出: 97 - 122

10. 🌟🌟

## 计算

11. 🌟

你可以在这里找到答案(在 solutions 路径下)

# 字符、布尔、单元类型

## 字符

1. 🌟

2. 🌟

## 布尔

3. 🌟

4. 🌟

## 单元类型

5. 🌟🌟

6. 🌟🌟 单元类型占用的内存大小是多少?

---

你可以在这里找到答案(在 solutions 路径下)

---

# 语句与表达式

## 示例

## 练习

1. 🌟🌟

2. 🌟

3. 🌟

> 你可以在这里找到答案(在 solutions 路径下)

# 函数

1. 🌟🌟🌟

2. 🌟🌟

3. 🌟🌟🌟

4. 🌟🌟 发散函数( Diverging function )不会返回任何值，因此它们可以用于替代需要返回任何值的地方

5. 🌟🌟

---

你可以在这里找到答案(在 solutions 路径下)

---

# 所有权与借用

学习资料 :

- English: Rust Book 4.1-4.4
- 简体中文: Rust语言圣经 - 所有权与借用

# 所有权

1. 🌟🌟

2. 🌟🌟

3. 🌟🌟

4. 🌟🌟

5. 🌟🌟

### 可变性

当所有权转移时，可变性也可以随之改变。

6. 🌟

7. 🌟🌟🌟

## 部分 move

当解构一个变量时，可以同时使用 `move` 和引用模式绑定的方式。当这么做时，部分 `move` 就会发生：变量中一部分的所有权被转移给其它变量，而另一部分我们获取了它的引用。

在这种情况下，原变量将无法再被使用，但是它没有转移所有权的那一部分依然可以使用，也就是之前被引用的那部分。

### 示例

### 练习

8. 🌟

9. 🌟🌟

你可以在这里找到答案(在 solutions 路径下)

# 引用和借用

## 引用

1. 🌟

2. 🌟

3. 🌟

4. 🌟

5. 🌟 🌟

### ref

`ref` 与 `&` 类似，可以用来获取一个值的引用，但是它们的用法有所不同。

6. 🌟 🌟 🌟

## 借用规则

7. 🌟

### 可变性

8. 🌟 错误: 从不可变对象借用可变

9. 🌟 🌟 Ok: 从可变对象借用不可变

## NLL

10. 🌟🌟

11. 🌟🌟

你可以在这里找到答案(在 solutions 路径下)

# 复合类型

学习资料:

- English: Rust Book 4.3, 5.1, 6.1, 8.2
- 简体中文: Rust语言圣经 - 复合类型

# 字符串

字符串字面量的类型是 `&str`， 例如 `let s: &str = "hello, world"` 中的 `"hello, world"` 的类型就是 `&str`。

## `str` 和 `&str`

1. 🌟 正常情况下我们无法使用 `str` 类型，但是可以使用 `&str` 来替代

2. 🌟🌟 如果要使用 `str` 类型，只能配合 `Box`。 `&` 可以用来将 `Box<str>` 转换为 `&str` 类型

## String

`String` 是定义在标准库中的类型，分配在堆上，可以动态的增长。它的底层存储是动态字节数组的方式( `Vec<u8>` )，但是与字节数组不同， `String` 是 `UTF-8` 编码。

3. 🌟

4. 🌟🌟🌟

5. 🌟🌟 我们可以用 `replace` 方法来替换指定的子字符串

在标准库的 String 模块中，有更多的实用方法，感兴趣的同学可以看看。

6. 🌟🌟 你只能将 `String` 跟 `&str` 类型进行拼接，并且 `String` 的所有权在此过程中会被 move

## `&str` 和 `String`

与 `str` 的很少使用相比， `&str` 和 `String` 类型却非常常用，因此也非常重要。

7. 🌟🌟 我们可以使用两种方法将 `&str` 转换成 `String` 类型

8. 🌟🌟 我们可以使用 `String::from` 或 `to_string` 将 `&str` 转换成 `String` 类型

## 字符串转义

9. 🌟

10. 🌟🌟🌟 有时候需要转义的字符很多，我们会希望使用更方便的方式来书写字符串: raw string.

## 字节字符串

想要一个非 UTF-8 形式的字符串吗(我们之前的 `str`, `&str`, `String` 都是 UTF-8 字符串)？可以试试字节字符串或者说字节数组:

**示例**:

如果大家想要了解更多关于字符串字面量、转义字符的话，可以看看 Rust Reference 的 'Tokens' 章节.

## 字符串索引string index

11. 🌟🌟 你无法通过索引的方式去访问字符串中的某个字符，但是可以使用切片的方式 `&s1[start..end]` ，但是 `start` 和 `end` 必须准确落在字符的边界处.

## 操作 UTF-8 字符串

12. 🌟

### utf8_slice

我们可以使用三方库 utf8_slice 来访问 UTF-8 字符串的某个子串，但是与之前不同的是，该库索引的是字符，而不是字节.

### Example

```rust
use utf8_slice;
fn main() {
    let s = "The 🚀 goes to the 🌑!";

    let rocket = utf8_slice::slice(s, 4, 5);
    // 结果是 "🚀"
}
```

---

你可以在这里找到答案(在 solutions 路径下)

---

# 数组

数组的类型是 `[T; Length]`，就如你所看到的，数组的长度是类型签名的一部分，因此数组的长度必须在编译期就已知，例如你不能使用以下方式来声明一个数组:

```
fn create_arr(n: i32) {
    let arr = [1; n];
}
```

以上函数将报错，因为编译器无法在编译期知道 `n` 的具体大小。

1. 🌟

2. 🌟🌟

3. 🌟 数组中的所有元素可以一起初始化为同一个值

4. 🌟 数组中的所有元素必须是同一类型

5. 🌟 数组的下标索引从 0 开始.

6. 🌟 越界索引会导致代码的 `panic`.

---

你可以在这里找到答案(在 solutions 路径下)

---

# 切片( Slice )

切片跟数组相似，但是切片的长度无法在编译期得知，因此你无法直接使用切片类型。

1. 🌟🌟 这里，`[i32]` 和 `str` 都是切片类型，但是直接使用它们会造成编译错误，如下代码所示。为了解决，你需要使用切片的引用： `&[i32]`， `&str` 。

一个切片引用占用了2个字大小的内存空间( 从现在开始，为了简洁性考虑，如无特殊原因，**我们统一使用切片来特指切片引用** )。 该切片的第一个字是指向数据的指针，第二个字是切片的长度。字的大小取决于处理器架构，例如在 `x86-64` 上，字的大小是 64 位也就是 8 个字节，那么一个切片引用就是 16 个字节大小。

切片( 引用 )可以用来借用数组的某个连续的部分，对应的签名是 `&[T]`，大家可以与数组的签名对比下 `[T; Length]` 。

2. 🌟🌟🌟

3. 🌟🌟

## 字符串切片

4. 🌟

5. 🌟

6. 🌟🌟 `&String` 可以被隐式地转换成 `&str` 类型.

---

你可以在这里找到答案(在 solutions 路径下)

---

# 元组( Tuple )

1. 🌟 元组中的元素可以是不同的类型。元组的类型签名是 `(T1, T2, ...)`, 这里 `T1`, `T2` 是相对应的元组成员的类型.

2. 🌟 可以使用索引来获取元组的成员

3. 🌟 过长的元组无法被打印输出

4. 🌟 使用模式匹配来解构元组

5. 🌟🌟 解构式赋值

6. 🌟🌟 元组可以用于函数的参数和返回值

---

你可以在这里找到答案(在 solutions 路径下)

---

# 结构体

## 三种类型的结构体

1. 🌟 对于结构体，我们必须为其中的每一个字段都指定具体的值

2. 🌟 单元结构体没有任何字段。

3. 🌟🌟🌟 元组结构体看起来跟元组很像，但是它拥有一个结构体的名称，该名称可以赋予它一定的意义。由于它并不关心内部数据到底是什么名称，因此此时元组结构体就非常适合。

## 结构体上的一些操作

4. 🌟 你可以在实例化一个结构体时将它整体标记为可变的，但是 Rust 不允许我们将结构体的某个字段专门指定为可变的.

5. 🌟 使用结构体字段初始化缩略语法可以减少一些重复代码

6. 🌟 你可以使用结构体更新语法基于一个结构体实例来构造另一个

## 打印结构体

7. 🌟🌟 我们可以使用 `#[derive(Debug)]` 让结构体变成可打印的.

## 结构体的所有权

当解构一个变量时，可以同时使用 `move` 和引用模式绑定的方式。当这么做时，部分 `move` 就会发生：变量中一部分的所有权被转移给其它变量，而另一部分我们获取了它的引用。

在这种情况下，原变量将无法再被使用，但是它没有转移所有权的那一部分依然可以使用，也就是之前被引用的那部分。

### 示例

```
```

### 练习

8. 🌟🌟

```
```

你可以在这里找到答案(在 solutions 路径下)

# 枚举 Enum

1. 🌟🌟 在创建枚举时，你可以使用显式的整数设定枚举成员的值。

2. 🌟 枚举成员可以持有各种类型的值

3. 🌟🌟 枚举成员中的值可以使用模式匹配来获取

4. 🌟🌟 使用枚举对类型进行同一化

5. 🌟🌟 Rust 中没有 `null`，我们通过 `Option<T>` 枚举来处理值为空的情况

6. 🌟🌟🌟🌟 使用枚举来实现链表.

---

你可以在这里找到答案(在 solutions 路径下)

---

# 流程控制

## if/else

1. 🌟

2. 🌟🌟 if/else 可以用作表达式来进行赋值

## for

3. 🌟 `for in` 可以用于迭代一个迭代器，例如序列 `a..b`.

4. 🌟🌟

5. 🌟

## while

6. 🌟🌟 当条件为 true 时， `while` 将一直循环

## continue and break

7. 🌟 使用 `break` 可以跳出循环

8. 🌟🌟 `continue` 会结束当次循环并立即开始下一次循环

## loop

9. 🌟🌟 loop 一般都需要配合 `break` 或 `continue` 一起使用。

10. 🌟🌟 loop 是一个表达式，因此我们可以配合 `break` 来返回一个值

11. 🌟🌟🌟 当有多层循环时，你可以使用 `continue` 或 `break` 来控制外层的循环。要实现这一点，外部的循环必须拥有一个标签 `'label`，然后在 `break` 或 `continue` 时指定该标签

---

你可以在这里找到答案(在 solutions 路径下)

---

# Pattern Match

# match, matches! 和 if let

## match

1. 🌟🌟

2. 🌟🌟 `match` 是一个表达式，因此可以用在赋值语句中

3. 🌟🌟 使用 match 匹配出枚举成员持有的值

## matches!

`matches!` 看起来像 `match`，但是它可以做一些特别的事情

4. 🌟🌟

5. 🌟🌟

## if let

在有些时候, 使用 `match` 匹配枚举有些太重了，此时 `if let` 就非常适合.

6. 🌟

7. 🌟🌟

8. 🌟🌟

## 变量遮蔽( Shadowing )

9. 🌟🌟

你可以在这里找到答案(在 solutions 路径下)

# 模式

1. 🌟🌟 使用 `|` 可以匹配多个值, 而使用 `..=` 可以匹配一个闭区间的数值序列

2. 🌟🌟🌟 `@` 操作符可以让我们将一个与模式相匹配的值绑定到新的变量上

3. 🌟🌟🌟

4. 🌟🌟 匹配守卫（match guard）是一个位于 match 分支模式之后的额外 if 条件，它能为分支模式提供更进一步的匹配条件。

5. 🌟🌟🌟 使用 `..` 忽略一部分值

6. 🌟🌟 使用模式 `&mut V` 去匹配一个可变引用时，你需要格外小心，因为匹配出来的 `V` 是一个值，而不是可变引用

---

你可以在这里找到答案(在 solutions 路径下)

---

# 方法和关联函数

## 示例

## Exercises

### Method

1. 🌟🌟 方法跟函数类似：都是使用 `fn` 声明，有参数和返回值。但是与函数不同的是，方法定义在结构体的上下文中(枚举、特征对象也可以定义方法)，而且方法的第一个参数一定是 `self` 或其变体 `&self` 、`&mut self`， `self` 代表了当前调用的结构体实例。

2. 🌟🌟 `self` 会拿走当前结构体实例(调用对象)的所有权，而 `&self` 却只会借用一个不可变引用， `&mut self` 会借用一个可变引用

3. 🌟🌟 `&self` 实际上是 `self: &Self` 的缩写或者说语法糖

### Associated function

4. 🌟🌟 定义在 `impl` 语句块中的函数被称为关联函数，因为它们跟当前类型关联在一起。关联函数与方法最大的区别就是它第一个参数不是 `self` ，原因是它们不需要使用当前的实例，因此关联函数往往可以用于构造函数：初始化一个实例对象。

### 多个 `impl` 语句块

5. 🌟 每一个结构体允许拥有多个 `impl` 语句块

## Enums

6. 🌟🌟🌟 我们还可以为枚举类型定义方法

# Practice

@todo

---

你可以在这里找到答案(在 solutions 路径下)

---

# Generics and Traits

# 泛型

## 函数

1. 🌟🌟🌟

2. 🌟🌟

## 结构体和 `impl`

3. 🌟

4. 🌟🌟

5. 🌟🌟

## 方法

6. 🌟🌟🌟

7. 🌟🌟

你可以在这里找到答案(在 solutions 路径下)

# Const 泛型

在之前的泛型中，可以抽象为一句话：针对类型实现的泛型，所有的泛型都是为了抽象不同的类型，那有没有针对值的泛型？答案就是 `Const 泛型`。

## 示例

1. 下面的例子同时使用泛型和 const 泛型来实现一个结构体，该结构体的字段中的数组长度是可变的

2. 目前，const 泛型参数只能使用以下形式的实参:

- 一个单独的 const 泛型参数
- 一个字面量 (i.e. 整数, 布尔值或字符).
- 一个具体的 const 表达式( 表达式中不能包含任何 泛型参数)

3. const 泛型还能帮我们避免一些运行时检查，提升性能

```
pub struct MinSlice<T, const N: usize> {
    pub head: [T; N],
    pub tail: [T],
}

fn main() {
    let slice: &[u8] = b"Hello, world";
    let reference: Option<&u8> = slice.get(6);
    // 我们知道 `.get` 返回的是 `Some(b' ')`
    // 但编译器不知道
    assert!(reference.is_some());

    let slice: &[u8] = b"Hello, world";

    // 当编译构建 MinSlice 时会进行长度检查，也就是在编译期我们就知道它的长度是 12
    // 在运行期，一旦 `unwrap` 成功，在 `MinSlice` 的作用域内，就再无需任何检查
    let minslice = MinSlice::<u8, 12>::from_slice(slice).unwrap();
    let value: u8 = minslice.head[6];
    assert_eq!(value, b' ')
}
```

# 练习

1. 🌟🌟 `<T, const N: usize>` 是结构体类型的一部分，和数组类型一样，这意味着长度不同会导致类型不同： `Array<i32, 3>` 和 `Array<i32, 4>` 是不同的类型

2. 🌟🌟

3. 🌟🌟🌟 有时我们希望能限制一个变量占用内存的大小，例如在嵌入式环境中，此时 const 泛型参数的第三种形式 `const 表达式` 就非常适合.

---

你可以在这里找到答案(在 solutions 路径下)

---

# Traits

特征 Trait 可以告诉编译器一个特定的类型所具有的、且能跟其它类型共享的特性。我们可以使用特征通过抽象的方式来定义这种共享行为，还可以使用特征约束来限定一个泛型类型必须要具有某个特定的行为。

---

Note: 特征跟其它语言的接口较为类似，但是仍然有一些区别

---

## 示例

## Exercises

1. 🌟🌟

### Derive 派生

我们可以使用 `#[derive]` 属性来派生一些特征，对于这些特征编译器会自动进行默认实现，对于日常代码开发而言，这是非常方便的，例如大家经常用到的 `Debug` 特征，就是直接通过派生来获取默认实现，而无需我们手动去完成这个工作。

想要查看更多信息，可以访问这里。

2. 🌟🌟

### 运算符

在 Rust 中，许多运算符都可以被重载，事实上，运算符仅仅是特征方法调用的语法糖。例如 `a + b` 中的 `+` 是 `std::ops::Add` 特征的 `add` 方法调用，因此我们可以为自定义类型实现该特征来支持该类型的加法运算。

3. 🌟🌟

4. 🌟🌟🌟

## 使用特征作为函数参数

除了使用具体类型来作为函数参数，我们还能通过 `impl Trait` 的方式来指定实现了该特征的参数：该参数能接受的类型必须要实现指定的特征。

5. 🌟🌟🌟

## 使用特征作为函数返回值

我们还可以在函数的返回值中使用 `impl Trait` 语法。然后只有在返回值是同一个类型时，才能这么使用，如果返回值是不同的类型，你可能更需要特征对象。

6. 🌟🌟

## 特征约束

`impl Trait` 语法非常直观简洁，但它实际上是特征约束的语法糖。

当使用泛型参数时，我们往往需要为该参数指定特定的行为，这种指定方式就是通过特征约束来实现的。

7. 🌟🌟

8. 🌟🌟

9. 🌟🌟🌟

你可以在这里找到答案(在 solutions 路径下)

# 特征对象

在特征练习中 我们已经知道当函数返回多个类型时，`impl Trait` 是无法使用的。

对于数组而言，其中一个限制就是无法存储不同类型的元素，但是通过之前的学习，大家应该知道枚举可以在部分场景解决这种问题，但是这种方法局限性较大。此时就需要我们的主角登场了。

## 使用 dyn 返回特征

Rust 编译器需要知道一个函数的返回类型占用多少内存空间。由于特征的不同实现类型可能会占用不同的内存，因此通过 `impl Trait` 返回多个类型是不被允许的，但是我们可以返回一个 `dyn` 特征对象来解决问题。

1. 🌟🌟🌟

## 在数组中使用特征对象

2. 🌟🌟

## &dyn and Box<dyn>

3. 🌟🌟

## 静态分发和动态分发Static and Dynamic dispatch

关于这块内容的解析介绍，请参见 Rust语言圣经。

4. 🌟🌟

# 对象安全

一个特征能变成特征对象，首先该特征必须是对象安全的，即该特征的所有方法都必须拥有以下特点：

- 返回类型不能是 `Self`.
- 不能使用泛型参数

5. 🌟🌟🌟🌟

---

你可以在这里找到答案(在 solutions 路径下)

---

# 进一步深入特征

## 关联类型

关联类型主要用于提升代码的可读性，例如以下代码：

```
pub trait CacheableItem: Clone + Default + fmt::Debug + Decodable + Encodable {
  type Address: AsRef<[u8]> + Clone + fmt::Debug + Eq + Hash;
  fn is_null(&self) -> bool;
}
```

相比 `AsRef<[u8]> + Clone + fmt::Debug + Eq + Hash`， `Address` 的使用可以极大的减少其它类型在实现该特征时所需的模版代码.

1. 🌟🌟🌟

## 定义默认的泛型类型参数

当我们使用泛型类型参数时，可以为该泛型参数指定一个具体的默认类型，这样当实现该特征时，如果该默认类型可以使用，那用户再无需手动指定具体的类型。

2. 🌟🌟

## 完全限定语法

在 Rust 中，两个不同特征的方法完全可以同名，且你可以为同一个类型同时实现这两个特征。这种情况下，就出现了一个问题：该如何调用这两个特征上定义的同名方法。为了解决这个问题，我们需要使用完全限定语法( Fully Qualified Syntax )。

### 示例

**练习题**

3. 🌟🌟

## Supertraits

有些时候我们希望在特征上实现类似继承的特性，例如让一个特征 `A` 使用另一个特征 `B` 的功能。这种情况下，一个类型要实现特征 `A` 首先要实现特征 `B`， 特征 `B` 就被称为 `supertrait`

4. 🌟🌟🌟

## 孤儿原则

关于孤儿原则的详细介绍请参见特征定义与实现的位置孤儿规则 和 在外部类型上实现外部特征。

5. 🌟🌟

---

你可以在这里找到答案(在 solutions 路径下)

---

# 集合类型

学习资源:

- 简体中文: Rust语言圣经 - 集合类型

# String

`std::string::String` 是 UTF-8 编码、可增长的动态字符串. 它也是我们日常开发中最常用的字符串类型，同时对于它所拥有的内容拥有所有权。

## 基本操作

1. 🌟🌟

## String and &str

虽然 `String` 的底层是 `Vec<u8>` 也就是字节数组的形式存储的，但是它是基于 UTF-8 编码的字符序列。`String` 分配在堆上、可增长且不是以 `null` 结尾。

而 `&str` 是切片引用类型( `&[u8]` )，指向一个合法的 UTF-8 字符序列，总之， `&str` 和 `String` 的关系类似于 `&[T]` 和 `Vec<T>` 。

如果大家想了解更多，可以看看易混淆概念解析 - &str 和 String。

2. 🌟🌟

3. 🌟🌟

## UTF-8 & 索引

由于 String 都是 UTF-8 编码的，这会带来几个影响:

- 如果你需要的是非 UTF-8 字符串，可以考虑 OsString
- 无法通过索引的方式访问一个 String

具体请看字符串索引。

4. 🌟🌟🌟 我们无法通过索引的方式访问字符串中的某个字符，但是可以通过切片的方式来获取字符串的某一部分 `&s1[start..end]`

### utf8_slice

我们可以使用 utf8_slice 来按照字符的自然索引方式对 UTF-8 字符串进行切片访问，与之前的切片方式相比，它索引的是字符，而之前的方式索引的是字节.

#### 示例

```
use utf8_slice;
fn main() {
    let s = "The 🚀 goes to the 🌑!";

    let rocket = utf8_slice::slice(s, 4, 5);
    // Will equal "🚀"
}
```

5. 🌟🌟🌟

---

提示: 也许你需要使用 `from_utf8` 方法

---

## 内部表示

事实上 `String` 是一个智能指针，它作为一个结构体存储在栈上，然后指向存储在堆上的字符串底层数据。

存储在栈上的智能指针结构体由三部分组成：一个指针只指向堆上的字节数组，已使用的长度以及已分配的容量 capacity (已使用的长度小于等于已分配的容量，当容量不够时，会重新分配内存空间)。

6. 🌟🌟 如果 String 的当前容量足够，那么添加字符将不会导致新的内存分配

7. 🌟🌟🌟

## 常用方法(TODO)

关于 String 的常用方法练习，可以查看这里.

---

你可以在这里找到答案(在 solutions 路径下)

---

# Vector

相比 `[T; N]` 形式的数组， `Vector` 最大的特点就是可以动态调整长度。

## 基本操作

1. 🌟🌟🌟

2. 🌟🌟 `Vec` 可以使用 `extend` 方法进行扩展

## 将 X 类型转换(From/Into 特征)成 Vec

只要为 `Vec` 实现了 `From<T>` 特征，那么 `T` 就可以被转换成 `Vec` 。

3. 🌟🌟🌟

## 索引

4. 🌟🌟🌟

## 切片

与 `String` 的切片类似， `Vec` 也可以使用切片。如果说 `Vec` 是可变的，那它的切片就是不可变或者说只读的，我们可以通过 `&` 来获取切片。

在 Rust 中，将切片作为参数进行传递是更常见的使用方式，例如当一个函数只需要可读性时，那传递 `Vec` 或 `String` 的切片 `&[T]` / `&str` 会更加适合。

5. 🌟🌟

## 容量

容量 `capacity` 是已经分配好的内存空间，用于存储未来添加到 `Vec` 中的元素。而长度 `len` 则是当前 `Vec` 中已经存储的元素数量。如果要添加新元素时，长度将要超过已有的容量，那容量会自动进行增长：Rust 会重新分配一块更大的内存空间，然后将之前的 `Vec` 拷贝过去，因此，这里就会发生新的内存分配( 目前 Rust 的容量调整策略是加倍，例如 2 -> 4 -> 8 ..)。

若这段代码会频繁发生，那频繁的内存分配会大幅影响我们系统的性能，最好的办法就是提前分配好足够的容量，尽量减少内存分配。

6. 🌟🌟

## 在 Vec 中存储不同类型的元素

`Vec` 中的元素必须是相同的类型，例如以下代码会发生错误:

```
fn main() {
    let v = vec![1, 2.0, 3];
}
```

但是我们可以使用枚举或特征对象来存储不同的类型.

7. 🌟🌟

8. 🌟🌟

> 你可以在这里找到答案(在 solutions 路径下)

# HashMap

`HashMap` 默认使用 `SipHash 1-3` 哈希算法，该算法对于抵抗 `HashDos` 攻击非常有效。在性能方面，如果你的 key 是中型大小的，那该算法非常不错，但是如果是小型的 key( 例如整数 )亦或是大型的 key ( 例如字符串 )，那你需要采用社区提供的其它算法来提高性能。

哈希表的算法是基于 Google 的 SwissTable，你可以在这里找到 C++ 的实现，同时在 CppCon talk 上也有关于算法如何工作的演讲。

## 基本操作

1. 🌟🌟

2. 🌟🌟

3. 🌟🌟

## HashMap key 的限制

任何实现了 `Eq` 和 `Hash` 特征的类型都可以用于 `HashMap` 的 key，包括:

- `bool` (虽然很少用到，因为它只能表达两种 key)
- `int`, `uint` 以及它们的变体，例如 `u8` 、 `i32` 等
- `String` 和 `&str` (提示: `HashMap` 的 key 是 `String` 类型时，你其实可以使用 `&str` 配合 `get` 方法进行查询

需要注意的是， `f32` 和 `f64` 并没有实现 `Hash`，原因是 浮点数精度 的问题会导致它们无法进行相等比较。

如果一个集合类型的所有字段都实现了 `Eq` 和 `Hash`,那该集合类型会自动实现 `Eq` 和 `Hash` 。例如 `Vect<T>` 要实现 `Hash`，那么首先需要 `T` 实现 `Hash` 。

4. 🌟🌟

## 容量

关于容量，我们在之前的 Vector 中有详细的介绍，而 `HashMap` 也可以调整容量: 你可以通过 `HashMap::with_capacity(uint)` 使用指定的容量来初始化，或者使用 `HashMap::new()` ，后者会提供一个默认的初始化容量。

### 示例

## 所有权

对于实现了 `Copy` 特征的类型，例如 `i32`，那类型的值会被拷贝到 `HashMap` 中。而对于有所有权的类型，例如 `String`，它们的值的所有权将被转移到 `HashMap` 中。

5. 🌟🌟

## 三方库 Hash 库

在开头，我们提到过如果现有的 `SipHash 1-3` 的性能无法满足你的需求，那么可以使用社区提供的替代算法。

例如其中一个社区库的使用方式如下：

```
use std::hash::BuildHasherDefault;
use std::collections::HashMap;
// 引入第三方的哈希函数
use twox_hash::XxHash64;


let mut hash: HashMap<_, _, BuildHasherDefault<XxHash64>> = Default::default();
hash.insert(42, "the answer");
assert_eq!(hash.get(&42), Some(&"the answer"));
```

---

你可以在这里找到答案(在 solutions 路径下)

---

# Type conversions

There are several ways we can use to perform type conversions, such as `as`, `From/Intro`, `TryFrom/TryInto`, `transmute` etc.

# 使用 as 进行类型转换

Rust 并没有为基本类型提供隐式的类型转换( coercion )，但是我们可以通过 `as` 来进行显式地转换。

1. 🌟

2. 🌟🌟 默认情况下, 数值溢出会导致编译错误，但是我们可以通过添加一行全局注解的方式来避免编译错误(溢出还是会发生)

3. 🌟🌟 当将任何数值转换成无符号整型 `T` 时，如果当前的数值不在新类型的范围内，我们可以对当前数值进行加值或减值操作( 增加或减少 `T::MAX + 1` )，直到最新的值在新类型的范围内，假设我们要将 `300` 转成 `u8` 类型，由于 `u8` 最大值是 255，因此 `300` 不在新类型的范围内并且大于新类型的最大值，因此我们需要减去 `T::MAX + 1`，也就是 `300 - 256 = 44`。

4. 🌟🌟🌟 裸指针可以和代表内存地址的整数互相转换

5. 🌟🌟🌟

---

你可以在这里找到答案(在 solutions 路径下)

---

# From/Into

`From` 特征允许让一个类型定义如何基于另一个类型来创建自己，因此它提供了一个很方便的类型转换的方式。

`From` 和 `Into` 是配对的，我们只要实现了前者，那后者就会自动被实现：只要实现了 `impl From<T> for U`， 就可以使用以下两个方法: `let u: U = U::from(T)` 和 `let u:U = T.into()`，前者由 `From` 特征提供，而后者由自动实现的 `Into` 特征提供。

需要注意的是，当使用 `into` 方法时，你需要进行显式地类型标注，因为编译器很可能无法帮我们推导出所需的类型。

来看一个例子，我们可以简单的将 `&str` 转换成 `String`

```
fn main() {
    let my_str = "hello";

    // 以下三个转换都依赖于一个事实：String 实现了 From<&str> 特征
    let string1 = String::from(my_str);
    let string2 = my_str.to_string();
    // 这里需要显式地类型标注
    let string3: String = my_str.into();
}
```

这种转换可以发生是因为标准库已经帮我们实现了 `From` 特征： `impl From<&'_ str> for String`。你还可以在这里)找到其它实现 `From` 特征的常用类型。

1. 🌟🌟🌟

## 为自定义类型实现 From 特征

2. 🌟🌟

3. 🌟🌟🌟 当执行错误处理时，为我们自定义的错误类型实现 `From` 特征是非常有用。这样就可以通过 `?` 自动将某个错误类型转换成我们自定义的错误类型

## TryFrom/TryInto

类似于 `From` 和 `Into`，`TryFrom` 和 `TryInto` 也是用于类型转换的泛型特征。

但是又与 `From/Into` 不同，`TryFrom` 和 `TryInto` 可以对转换后的失败进行处理，然后返回一个 `Result`。

4. 🌟🌟

5. 🌟🌟🌟

你可以在这里找到答案(在 solutions 路径下)

# 其它转换

## 将任何类型转换成 String

只要为一个类型实现了 `ToString`，就可以将任何类型转换成 `String`。事实上，这种方式并不是最好的，大家还记得 `fmt::Display` 特征吗？它可以控制一个类型如何打印，在实现它的时候还会自动实现 `ToString`。

1. 🌟🌟

## 解析 String

2. 🌟🌟🌟 使用 `parse` 方法可以将一个 `String` 转换成 `i32` 数字，这是因为在标准库中为 `i32` 类型实现了 `FromStr`：`impl FromStr for i32`

3. 🌟🌟 还可以为自定义类型实现 `FromStr` 特征

## Deref 特征

Deref 特征在智能指针 - Deref章节中有更加详细的介绍。

## transmute

`std::mem::transmute` 是一个 unsafe 函数，可以把一个类型按位解释为另一个类型，其中这两个类型必须有同样的位数( bits )。

`transmute` 相当于将一个类型按位移动到另一个类型，它会将源值的所有位拷贝到目标值中，然后遗忘源值。该函数跟 C 语言中的 `memcpy` 函数类似。

正因为此，`transmute` **非常非常不安全!** 调用者必须要自己保证代码的安全性，当然这也是 unsafe 的目的。

### 示例

1. `transmute` 可以将一个指针转换成一个函数指针，该转换并不具备可移植性，原因是在不同机器上，函数指针和数据指针可能有不同的位数( size )。

2. `transmute` 还可以扩展或缩短一个不变量的生命周期，将 Unsafe Rust 的不安全性体现的淋漓尽致!

3. 事实上我们还可以使用一些安全的方法来替代 `transmute`.

---

你可以在这里找到答案(在 solutions 路径下)

---

# Result and panic

Learning resources:

- English: Rust Book 9.1, 9.2
- 简体中文: Rust语言圣经 - 返回值和错误处理

# panic!

Rust 中最简单的错误处理方式就是使用 `panic` 。它会打印出一条错误信息并打印出栈调用情况，最终结束当前线程:

- 若 panic 发生在 `main` 线程，那程序会随之退出
- 如果是在生成的( spawn )子线程中发生 panic, 那么当前的线程会结束，但是程序依然会继续运行

1. 🌟🌟

# 常见的 panic

2. 🌟🌟

### 详细的栈调用信息

默认情况下，栈调用只会展示最基本的信息:

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is
99', src/main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

但是有时候，我们还希望获取更详细的信息:

3. 🌟

```
## 填空以打印全部的调用栈
## 提示: 你可以在之前的默认 panic 信息中找到相关线索
$ __ cargo run
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `[97, 98, 99]`,
 right: `[96, 97, 98]`', src/main.rs:3:5
stack backtrace:
   0: rust_begin_unwind
             at
/rustc/9d1b2106e23b1abd32fce1f17267604a5102f57a/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
             at
/rustc/9d1b2106e23b1abd32fce1f17267604a5102f57a/library/core/src/panicking.rs:116
   2: core::panicking::assert_failed_inner
   3: core::panicking::assert_failed
             at
/rustc/9d1b2106e23b1abd32fce1f17267604a5102f57a/library/core/src/panicking.rs:154
   4: study_cargo::main
             at ./src/main.rs:3:5
   5: core::ops::function::FnOnce::call_once
             at
/rustc/9d1b2106e23b1abd32fce1f17267604a5102f57a/library/core/src/ops/function.rs:1
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

## unwinding 和 abort

当出现 `panic!` 时，程序提供了两种方式来处理终止流程：**栈展开**和**直接终止**。

其中，默认的方式就是 `栈展开`，这意味着 Rust 会回溯栈上数据和函数调用，因此也意味着更多的善后工作，好处是可以给出充分的报错信息和栈调用信息，便于事后的问题复盘。`直接终止`，顾名思义，不清理数据就直接退出程序，善后工作交与操作系统来负责。

对于绝大多数用户，使用默认选择是最好的，但是当你关心最终编译出的二进制可执行文件大小时，那么可以尝试去使用直接终止的方式，例如下面的配置修改 `Cargo.toml` 文件，实现在 `release` 模式下遇到 `panic` 直接终止：

```
[profile.release]
panic = 'abort'
```

# result and ?

`Result<T>` 是一个枚举类型用于描述返回的结果或错误，它包含两个成员(变体 variants)：

- `Ok(T)`：返回一个结果值 T
- `Err(e)`：返回一个错误， `e` 是具体的错误值

简而言之，如果期待一个正确的结果，就返回 `Ok`，反之则是 `Err`。

1. 🌟🌟

### ?

`?` 跟 `unwrap` 非常像，但是 `?` 会返回一个错误，而不是直接 panic.

2. 🌟🌟

3. 🌟🌟

### map & and_then

map and and_then 是两个常用的组合器( combinator )，可以用于 `Result<T, E>` (也可用于 `Option<T>` ).

4. 🌟🌟

5. 🌟🌟🌟

### 类型别名

如果我们要在代码中到处使用 `std::result::Result<T, ParseIntError>` ，那毫无疑问，代码将变得特别冗长和啰嗦，对于这种情况，可以使用类型别名来解决。

例如在标准库中，就在大量使用这种方式来简化代码: `io::Result`.

6. 🌟

## 在 `fn main` 中使用 `Result`

一个典型的 `main` 函数长这样:

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

事实上 `main` 函数还可以返回一个 `Result` 类型：如果 `main` 函数内部发生了错误，那该错误会被返回并且打印出一条错误的 debug 信息。

# 包和模块

学习资料:

- 简体中文: Rust语言圣经 - 包和模块

# Package and Crate

`package` 是你通过 `Cargo` 创建的工程或项目，因此在 `package` 的根目录下会有一个 `Cargo.toml` 文件。

1. 🌟 创建一个 `package`，拥有以下目录结构:

```
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

```
# in Cargo.toml
[package]
name = "hello-package"
version = "0.1.0"
edition = "2021"
```

---

注意! 我们会在包与模块中使用上面的项目作为演示，因此不要删除

---

2. 🌟 创建一个 package，拥有以下目录结构:

```
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

```
# in Cargo.toml
[package]
name = "hello-package1"
version = "0.1.0"
edition = "2021"
```

---

该项目可以安全的移除

---

3. 🌟

# 包Crate

一个包可以是二进制也可以一个依赖库。每一个包都有一个包根，例如二进制包的包根是 `src/main.rs`，库包的包根是 `src/lib.rs`。包根是编译器开始处理源代码文件的地方，同时也是包模块树的根部。

在 package `hello-package` 中，有一个二进制包，该包与 `package` 同名：`hello-package`，其中 `src/main.rs` 是该二进制包的包根.

与 `hello-package` 类似，`hello-package1` 同样包含一个包，但是与之前的二进制包不同，该 package 包含的是库包，其中 `src/lib.rs` 是其包根.

4. 🌟

5. 🌟🌟 为 `hello-package` 添加一个库包，并且完成以下目录结构的填空:

在上一个步骤后，我们的 `hello-package` 中已经存在两个包：一个二进制包和一个库包，两个包的名称都与 package 相同： `hello-package`。

6. 🌟🌟🌟 一个 package 最多只能包含一个库包，但是却可以包含多个二进制包：通过将二进制文件放入到 `src/bin` 目录下实现: **该目录下的每个文件都是一个独立的二进制包，包名与文件名相同，不再与 package 的名称相同。**.

可以看到，上面的 package 结构非常标准，你可以在很多 Rust 项目中看到该结构的身影。

---

你可以在这里找到答案(在 solutions 路径下)

---

# Module

在 Rust 语言圣经中，我们已经深入讲解过模块module，这里就不再赘述，直接开始我们的练习。

之前我们创建了一个 package `hello-package`，它的目录结构在经过多次修改后，变成了以下模样:

```
.
├── Cargo.toml
├── src
│   ├── lib.rs
│   └── main.rs
```

下面，我们来为其中的库包创建一些模块，然后在二进制包中使用这些模块。

1. 🌟🌟 根据以下的模块树描述实现模块 `front_of_house`：

```
库包的根(src/lib.rs)
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         ├── take_payment
         └── complain
```

2. 🌟🌟 让我们在库包的根中定义一个函数 `eat_at_restaurant`，然后在该函数中调用之前创建的函数 `eat_at_restaurant`

```
// in lib.rs

// 填空并修复错误

// 提示：你需要通过 `pub` 将一些项标记为公有的，这样模块 `front_of_house` 中的项才能被模块外的项访问
mod front_of_house {
    /* ...snip... */
}

pub fn eat_at_restaurant() {
    // 使用绝对路径调用
    __.add_to_waitlist();

    // 使用相对路径调用
     __.add_to_waitlist();
}
```

3. 🌟🌟 我们还可以使用 `super` 来导入父模块中的项

## 将模块分离并放入独立的文件中

```rust
// in lib.rs
pub mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}

        pub fn seat_at_table() -> String {
            String::from("sit down please")
        }
    }

    pub mod serving {
        pub fn take_order() {}

        pub fn serve_order() {}

        pub fn take_payment() {}

        // 我猜你不希望顾客听到你在抱怨他们，因此让这个函数私有化吧
        fn complain() {}
    }
}

pub fn eat_at_restaurant() -> String {
    front_of_house::hosting::add_to_waitlist();

    back_of_house::cook_order();

    String::from("yummy yummy!")
}

pub mod back_of_house {
    pub fn fix_incorrect_order() {
        cook_order();
        crate::front_of_house::serving::serve_order();
    }

    pub fn cook_order() {}
}
```

4. 🌟🌟🌟🌟 请将上面的模块和代码分离到以下目录文件中e :

```
.
├── Cargo.toml
├── src
│   ├── back_of_house.rs
│   ├── front_of_house
│   │   ├── hosting.rs
│   │   ├── mod.rs
│   │   └── serving.rs
│   ├── lib.rs
│   └── main.rs
```

## 从二进制包中访问库包的代码

**请确保你已经完成了第四题，然后再继续进行.**

当到底此处时，你的项目结构应该如下所示:

```
.
├── Cargo.toml
├── src
│   ├── back_of_house.rs
│   ├── front_of_house
│   │   ├── hosting.rs
│   │   ├── mod.rs
│   │   └── serving.rs
│   ├── lib.rs
│   └── main.rs
```

5. 🌟🌟🌟 现在我们可以从二进制包中发起函数调用了.

---

你可以在这里找到答案(在 solutions 路径下)

---

# use and pub

1. 🌟 使用 `use` 可以将两个同名类型引入到当前作用域中，但是别忘了 `as` 关键字.

2. 🌟🌟 如果我们在使用来自同一个包或模块中的多个不同项，那么可以通过简单的方式将它们一次性引入进来

## 使用 `pub use` 进行再导出

3. 🌟🌟🌟 在之前创建的`hello-package` 的库包中, 添加一些代码让下面的代码能够正常工作

## pub(in Crate)

有时我们希望某一个项只对特定的包可见，那么就可以使用 `pub(in Crate)` 语法.

### 示例

## 完整代码

至此，包与模块章节已经结束，关于 `hello-package` 的完整代码可以在这里 找到.

> 你可以在这里找到答案(在 solutions 路径下)

# 注释和文档

本章的学习资料在这里，大家可以先行学习后再来做题。

## 注释

1. 🌟🌟

## 文档注释

文档注释会被解析为 HTML 文件，并支持 `Markdown` 语法。

在开始之前，我们需要创建一个新的项目用于后面的练习: `cargo new --lib doc-comments`.

### 行文档注释 `///`

为 `add_one` 函数添加文档

```
// in lib.rs

/// Add one to the given value and return the value
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

### Cargo doc

我们可以使用 `cargo doc --open` 来生成 HTML 文件，并自动在浏览器中打开网页。

### 块文档注释 `/** ... */`

为函数 `add_two` 添加文档:

```
/** Add two to the given value and return a new value


let arg = 5;
let answer = my_crate::add_two(arg);

assert_eq!(7, answer);

*/
pub fn add_two(x: i32) -> i32 {
    x + 2
}
```

## 为包和模块创建文档注释

我们还可以创建包和模块的注释，用于描述它们的功能。

首先，来为我们的库包添加一些文档注释:

---

注意: 必须要将包、模块注释放置在包根或模块文件的最顶部

---

```
//! # 文档注释
//!
//! 该库用于文档注释的教学

// in lib.rs
pub mod compute;
```

同样的，我们还可以使用块注释来达成目的:

```
/*! # 文档注释

 该库用于文档注释的教学 */
```

下一步，创建一个新的模块文件 `src/compute.rs`，然后在其中添加以下注释:

```
//! 本模块用于处理一些复杂计算

// in compute.rs
```

然后运行 `cargo doc --open` 查看下结果。

## 文档测试

细心的同学可能会发现之前的 `add_one` 和 `add_tow` 的文档注释中，包含了两个示例代码块.

以上示例不仅仅是作为文档用于演示你的函数该如何使用，它的另一个作用就是用于文档测试 `cargo test`。

2. 🌟🌟 但是在这两个函数的示例中，存在错误，请修复它们并使用 `cargo test` 获取以下输出结果:

```
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests doc-comments

running 2 tests
test src/lib.rs - add_one (line 11) ... ok
test src/lib.rs - add_two (line 26) ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.55s
```

3. 🌟🌟 有时我们会期望示例的结果是一个 panic。将以下代码添加到 `src/compute.rs`，并且让 `cargo test` 成功运行.

---

你只能修改注释，不要修改 `fn div`

---

```rust
// in src/compute.rs

/// # Panics
///
/// The function panics if the second argument is zero.
///
/// ```rust,should_panic
/// // panics on division by zero
/// doc_comments::compute::div(10, 0);
/// ```
pub fn div(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("Divide-by-zero error");
    }

    a / b
}
```

4. 🌟🌟 有时我们会想要隐藏文档，但是保留文档测试

将以下代码添加到 `src/compute.rs`，

```
// in src/compute.rs

/// ```
/// # fn try_main() -> Result<(), String> {
/// let res = doc_comments::compute::try_div(10, 0)?;
/// # Ok(()) // returning from try_main
/// # }
/// # fn main() {
/// #    try_main().unwrap();
/// #
/// # }
/// ```
pub fn try_div(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Divide-by-zero"))
    } else {
        Ok(a / b)
    }
}
```

然后修改以上代码已实现两个目标:

- 文档注释不能出现在 `cargo doc --open` 生成的网页中
- 运行测试，并成功看到以下结果:

```
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.00s

   Doc-tests doc-comments

running 4 tests
test src/compute.rs - compute::div (line 7) ... ok
test src/lib.rs - add_two (line 27) ... ok
test src/lib.rs - add_one (line 11) ... ok
test src/compute.rs - compute::try_div (line 20) ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 0.51s
```

## 代码跳转

Rust 为我们提供一个非常强大的特性：可以在文档注释中实现代码跳转。

将以下代码添加到 `src/lib.rs`:

```
// in lib.rs

/// Add one to the given value and return a [`Option`] type
pub fn add_three(x: i32) -> Option<i32> {
    Some(x + 3)
}
```

除了跳转到标准库中，我们还能跳转到项目中的其它模块。

```
// in lib.rs

mod a {
    /// Add four to the given value and return a [`Option`] type
    /// [`crate::MySpecialFormatter`]
    pub fn add_four(x: i32) -> Option<i32> {
        Some(x + 4)
    }
}

struct MySpecialFormatter;
```

## 文档属性

下面是很常用的 `#[doc]` 属性，该属性可以被 `rustdoc` 所使用。

### `inline`

可以用于内联文档, 而不是链接到一个单独的页面。

```
#[doc(inline)]
pub use bar::Bar;

/// bar docs
mod bar {
    /// the docs for Bar
    pub struct Bar;
}
```

### `no_inline`

用于防止链接到单独的页面或其它地方。

```
// Example from libcore/prelude
#[doc(no_inline)]
pub use crate::mem::drop;
```

`hidden`

通过这个属性让 `rustdoc` 不要将下面的项包含在文档中:

对文档来说，`rustdoc` 被社区广泛采用，大家所看到的标准库文档也是基于此生成的。

## 完整的代码

`doc-comments` 的完整代码可以在这里找到.

# 格式化输出

## 位置参数

1.🌟🌟

## 具名参数

2.🌟🌟

## 字符串对齐

3.🌟🌟 默认情况下，通过空格来填充字符串

4.🌟🌟🌟 左对齐, 右对齐, 使用指定的字符填充

5.🌟🌟 我们还能使用 0 来填充数字

## 精度

6.🌟🌟 浮点数精度

7.🌟🌟🌟 字符串长度

# 二进制, 八进制, 十六进制

- format!("{}", foo) -> "3735928559"
- format!("0x{:X}", foo) -> "0xDEADBEEF"
- format!("0o{:o}", foo) -> "0o33653337357"

8.🌟🌟

# 捕获环境中的值

9.🌟🌟🌟

# Others

### Example

# 生命周期

学习资料:

- 简体中文: Rust语言圣经 - 生命周期

# 生命周期基础

编译器通过生命周期来确保所有的借用都是合法的，典型的，一个变量在创建时生命周期随之开始，销毁时生命周期也随之结束。

# 生命周期的范围

1. 🌟

2. 🌟🌟

**示例**

```
{
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

# 生命周期标注

Rust 的借用检查器使用显式的生命周期标注来确定一个引用的合法范围。但是对于用户来说，我们在大多数场景下，都无需手动去标注生命周期，原因是编译器会在某些情况下自动应用生命周期消除规则。

在了解编译器使用哪些规则帮我们消除生命周期之前，首先还是需要知道该如何手动标记生命周期。

**函数**

**大家先忽略生命周期消除规则**，让我们看看，函数签名中的生命周期有哪些限制:

- 需要为每个引用标注上合适的生命周期
- 返回值中的引用，它的生命周期要么跟某个引用参数相同，要么是 `'static`

### 示例

3. 🌟

4. 🌟🌟🌟

5. 🌟🌟

### Structs

6. 🌟

7. 🌟🌟

8. 🌟🌟

## 方法

方法的生命周期标注跟函数类似。

### 示例

9. 🌟🌟

## 生命周期消除( Elision )

有一些生命周期的标注方式很常见，因此编译器提供了一些规则，可以让我们在一些场景下无需去标注生命周期，既节省了敲击键盘的繁琐，又能提升可读性。

这种规则被称为生命周期消除规则( Elision )，该规则之所以存在，仅仅是因为这些场景太通用了，为了方便用户而已。事实上对于借用检查器而言，该有的生命周期一个都不能少，只不过对于用户而言，可以省去一些。

10. 🌟🌟

# &'static and T: 'static

`'static` 是一个 Rust 保留的生命周期名称，在之前我们可能已经见过好几次了:

```
// 引用的生命周期是 'static :
let s: &'static str = "hello world";

// 'static 也可以用于特征约束中:
fn generic<T>(x: T) where T: 'static {}
```

虽然它们都是 `'static`，但是也稍有不同。

## &'static

作为一个引用生命周期，`&'static` 说明该引用指向的数据可以跟程序活得一样久，但是该引用的生命周期依然有可能被强转为一个更短的生命周期。

1. 🌟🌟 有好几种方法可以将一个变量标记为 `'static` 生命周期, 其中两种都是和保存在二进制文件中相关( 例如字符串字面量就是保存在二进制文件中，它的生命周期是 `'static` )。

2. 🌟🌟🌟🌟 使用 `Box::leak` 也可以产生 `'static` 生命周期

3. 🌟 `&'static` 只能说明引用指向的数据是能一直存活的，但是引用本身依然受限于它的作用域

4. `&'static` 可以被强转成一个较短的生命周期

**Example**

## T: 'static

关于 `'static` 的特征约束详细解释，请参见 Rust 语言圣经，这里就不再赘述。

5. 🌟🌟

6. 

# 深入生命周期

## 特征约束

就像泛型类型可以有约束一样，生命周期也可以有约束，如下所示：

- `T: 'a`，所有引用在 `T` 必须超过生命周期 `'a`
- `T: Trait + 'a`：`T` 必须实现特征 `Trait` 并且所有引用在 `T` 必须超过生命周期 `'a`

**示例**

1. 🌟

2. 🌟🌟

3. 🌟🌟

## HRTB（更高等级特征约束）(Higher-ranked trait bounds)

类型约束可能在生命周期中排名更高。这些约束指定了一个约束对于所有生命周期都为真。例如，诸如此类的约束 `for<'a> &'a T: PartialEq<i32>` 需要如下实现：

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

然后可以用于将一个 `&'a T` 与任何生命周期进行比较 `i32` 。

这里只能使用更高级别的约束，因为引用的生命周期比函数上任何可能的生命周期参数都短。

4. 🌟🌟🌟

```
/* 添加 HRTB 使下面代码正常运行！ */
fn call_on_ref_zero<'a, F>(f: F) where F: Fn(&'a i32) {
    let zero = 0;
    f(&zero);
}

fn main() {
    println!("Success!")
}
```

# NLL（非词汇生命周期）(Non-Lexical Lifetime)

在解释 NLL 之前，我们先看一段代码：

```
fn main() {
   let mut s = String::from("hello");

    let r1 = &s;
    let r2 = &s;
    println!("{} and {}", r1, r2);

    let r3 = &mut s;
    println!("{}", r3);
}
```

根据我们目前的知识，这段代码会因为违反 Rust 中的借用规则而导致错误。

但是，如果您执行 `cargo run` ，那么一切都没问题，那么这里发生了什么?

编译器在作用域结束之前判断不再使用引用的能力称为 **非词法生命周期**（简称 **NLL** ）。

有了这种能力，编译器就知道最后一次使用引用是什么时候，并根据这些知识优化借用规则。

```
let mut u = 0i32;
let mut v = 1i32;
let mut w = 2i32;

// lifetime of `a` = α ∪ β ∪ γ
let mut a = &mut u;     // --+ α. lifetime of `&mut u`  --+ lexical "lifetime"
of `&mut u`,`&mut u`, `&mut w` and `a`
use(a);                 //   |                           |
*a = 3; // <-----------------+                           |
...                     //                              |
a = &mut v;             // --+ β. lifetime of `&mut v`   |
use(a);                 //   |                           |
*a = 4; // <-----------------+                           |
...                     //                              |
a = &mut w;             // --+ γ. lifetime of `&mut w`   |
use(a);                 //   |                           |
*a = 5; // <-----------------+ <-------------------------+
```

# 再借用

学习了 NLL 之后，我们现在可以很容易地理解再借用了。

### 示例

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

impl Point {
    fn move_to(&mut self, x: i32, y: i32) {
        self.x = x;
        self.y = y;
    }
}

fn main() {
    let mut p = Point { x: 0, y: 0 };
    let r = &mut p;
    // 这里是再借用
    let rr: &Point = &*r;

    println!("{:?}", rr); // 这里结束再借用

    // 再借用结束，现在我们可以继续使用 `r`
    r.move_to(10, 10);
    println!("{:?}", r);
}
```

5. 🌟🌟

# 未约束的生命周期

在 Nomicon - Unbounded Lifetimes 中查看更多信息。

# 更多省略规则

```
impl<'a> Reader for BufReader<'a> {
    // 'a 在以下方法中不使用
}

// 可以写为：
impl Reader for BufReader<'_> {

}
```

```
// Rust 2015
struct Ref<'a, T: 'a> {
    field: &'a T
}

// Rust 2018
struct Ref<'a, T> {
    field: &'a T
}
```

# 艰难的练习

6. 🌟🌟🌟🌟

```
/* 使下面代码正常运行 */
struct Interface<'a> {
    manager: &'a mut Manager<'a>
}

impl<'a> Interface<'a> {
    pub fn noop(self) {
        println!("interface consumed");
    }
}

struct Manager<'a> {
    text: &'a str
}

struct List<'a> {
    manager: Manager<'a>,
}

impl<'a> List<'a> {
    pub fn get_interface(&'a mut self) -> Interface {
        Interface {
            manager: &mut self.manager
        }
    }
}

fn main() {
    let mut list = List {
        manager: Manager {
            text: "hello"
        }
    };

    list.get_interface().noop();

    println!("Interface should be dropped here and the borrow released");

    use_list(&list);
}

fn use_list(list: &List) {
    println!("{}", list.manager.text);
}
```

# Functional programing

# Closure

下面代码是Rust圣经课程中闭包章节的课内练习题答案：

```
struct Cacher<T,E>
where
    T: Fn(E) -> E,
    E: Copy
{
    query: T,
    value: Option<E>,
}

impl<T,E> Cacher<T,E>
where
    T: Fn(E) -> E,
    E: Copy
{
    fn new(query: T) -> Cacher<T,E> {
        Cacher {
            query,
            value: None,
        }
    }

    fn value(&mut self, arg: E) -> E {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.query)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
fn main() {

}

#[test]
fn call_with_different_values() {
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    let v2 = c.value(2);

    assert_eq!(v2, 1);
}
```

# Iterator

# newtype and Sized

# Smart pointers

# Box

# Deref

# Drop

# Rc and Arc

# Cell and RefCell

# Weak and Circle reference

# Self referential

# Threads

# Basic using

# Message passing

# Sync

# Atomic

# Send and Sync

# Global variables

# Errors

# Unsafe doing

# 内联汇编

Rust provides support for inline assembly via the `asm!` macro. It can be used to embed handwritten assembly in the assembly output generated by the compiler. Generally this should not be necessary, but might be where the required performance or timing cannot be otherwise achieved. Accessing low level hardware primitives, e.g. in kernel code, may also demand this functionality.

---

> **Note**: the examples here are given in x86/x86-64 assembly, but other architectures are also supported.

---

Inline assembly is currently supported on the following architectures:

- x86 and x86-64
- ARM
- AArch64
- RISC-V

## Basic usage

Let us start with the simplest possible example:

```
use std::arch::asm;

unsafe {
    asm!("nop");
}
```

This will insert a NOP (no operation) instruction into the assembly generated by the compiler. Note that all `asm!` invocations have to be inside an `unsafe` block, as they could insert arbitrary instructions and break various invariants. The instructions to be inserted are listed in the first argument of the `asm!` macro as a string literal.

## Inputs and outputs

Now inserting an instruction that does nothing is rather boring. Let us do something that actually acts on data:

```
use std::arch::asm;

let x: u64;
unsafe {
    asm!("mov {}, 5", out(reg) x);
}
assert_eq!(x, 5);
```

This will write the value `5` into the `u64` variable `x`. You can see that the string literal we use to specify instructions is actually a template string. It is governed by the same rules as Rust format strings. The arguments that are inserted into the template however look a bit different than you may be familiar with. First we need to specify if the variable is an input or an output of the inline assembly. In this case it is an output. We declared this by writing `out`. We also need to specify in what kind of register the assembly expects the variable. In this case we put it in an arbitrary general purpose register by specifying `reg`. The compiler will choose an appropriate register to insert into the template and will read the variable from there after the inline assembly finishes executing.

Let us see another example that also uses an input:

```
use std::arch::asm;

let i: u64 = 3;
let o: u64;
unsafe {
    asm!(
        "mov {0}, {1}",
        "add {0}, 5",
        out(reg) o,
        in(reg) i,
    );
}
assert_eq!(o, 8);
```

This will add `5` to the input in variable `i` and write the result to variable `o`. The particular way this assembly does this is first copying the value from `i` to the output, and then adding `5` to it.

The example shows a few things:

First, we can see that `asm!` allows multiple template string arguments; each one is treated as a separate line of assembly code, as if they were all joined together with newlines between them. This makes it easy to format assembly code.

Second, we can see that inputs are declared by writing `in` instead of `out`.

Third, we can see that we can specify an argument number, or name as in any format string. For inline assembly templates this is particularly useful as arguments are often used more than once. For more complex inline assembly using this facility is generally recommended,

as it improves readability, and allows reordering instructions without changing the argument order.

We can further refine the above example to avoid the `mov` instruction:

```
use std::arch::asm;

let mut x: u64 = 3;
unsafe {
    asm!("add {0}, 5", inout(reg) x);
}
assert_eq!(x, 8);
```

We can see that `inout` is used to specify an argument that is both input and output. This is different from specifying an input and output separately in that it is guaranteed to assign both to the same register.

It is also possible to specify different variables for the input and output parts of an `inout` operand:

```
use std::arch::asm;

let x: u64 = 3;
let y: u64;
unsafe {
    asm!("add {0}, 5", inout(reg) x => y);
}
assert_eq!(y, 8);
```

## Late output operands

The Rust compiler is conservative with its allocation of operands. It is assumed that an `out` can be written at any time, and can therefore not share its location with any other argument. However, to guarantee optimal performance it is important to use as few registers as possible, so they won't have to be saved and reloaded around the inline assembly block. To achieve this Rust provides a `lateout` specifier. This can be used on any output that is written only after all inputs have been consumed. There is also a `inlateout` variant of this specifier.

Here is an example where `inlateout` *cannot* be used:

```
use std::arch::asm;

let mut a: u64 = 4;
let b: u64 = 4;
let c: u64 = 4;
unsafe {
    asm!(
        "add {0}, {1}",
        "add {0}, {2}",
        inout(reg) a,
        in(reg) b,
        in(reg) c,
    );
}
assert_eq!(a, 12);
```

Here the compiler is free to allocate the same register for inputs `b` and `c` since it knows they have the same value. However it must allocate a separate register for `a` since it uses `inout` and not `inlateout`. If `inlateout` was used, then `a` and `c` could be allocated to the same register, in which case the first instruction to overwrite the value of `c` and cause the assembly code to produce the wrong result.

However the following example can use `inlateout` since the output is only modified after all input registers have been read:

```
use std::arch::asm;

let mut a: u64 = 4;
let b: u64 = 4;
unsafe {
    asm!("add {0}, {1}", inlateout(reg) a, in(reg) b);
}
assert_eq!(a, 8);
```

As you can see, this assembly fragment will still work correctly if `a` and `b` are assigned to the same register.

## Explicit register operands

Some instructions require that the operands be in a specific register. Therefore, Rust inline assembly provides some more specific constraint specifiers. While `reg` is generally available on any architecture, explicit registers are highly architecture specific. E.g. for x86 the general purpose registers `eax`, `ebx`, `ecx`, `edx`, `ebp`, `esi`, and `edi` among others can be addressed by their name.

```
use std::arch::asm;

let cmd = 0xd1;
unsafe {
    asm!("out 0x64, eax", in("eax") cmd);
}
```

In this example we call the `out` instruction to output the content of the `cmd` variable to port `0x64`. Since the `out` instruction only accepts `eax` (and its sub registers) as operand we had to use the `eax` constraint specifier.

---

> **Note**: unlike other operand types, explicit register operands cannot be used in the template string: you can't use `{}` and should write the register name directly instead. Also, they must appear at the end of the operand list after all other operand types.

---

Consider this example which uses the x86 `mul` instruction:

```
use std::arch::asm;

fn mul(a: u64, b: u64) -> u128 {
    let lo: u64;
    let hi: u64;

    unsafe {
        asm!(
            // The x86 mul instruction takes rax as an implicit input and writes
            // the 128-bit result of the multiplication to rax:rdx.
            "mul {}",
            in(reg) a,
            inlateout("rax") b => lo,
            lateout("rdx") hi
        );
    }

    ((hi as u128) << 64) + lo as u128
}
```

This uses the `mul` instruction to multiply two 64-bit inputs with a 128-bit result. The only explicit operand is a register, that we fill from the variable `a`. The second operand is implicit, and must be the `rax` register, which we fill from the variable `b`. The lower 64 bits of the result are stored in `rax` from which we fill the variable `lo`. The higher 64 bits are stored in `rdx` from which we fill the variable `hi`.

# Clobbered registers

In many cases inline assembly will modify state that is not needed as an output. Usually this is either because we have to use a scratch register in the assembly or because instructions modify state that we don't need to further examine. This state is generally referred to as being "clobbered". We need to tell the compiler about this since it may need to save and restore this state around the inline assembly block.

```
use core::arch::asm;

fn main() {
    // three entries of four bytes each
    let mut name_buf = [0_u8; 12];
    // String is stored as ascii in ebx, edx, ecx in order
    // Because ebx is reserved, we get a scratch register and move from
    // ebx into it in the asm.  The asm needs to preserve the value of
    // that register though, so it is pushed and popped around the main asm
    // (in 64 bit mode for 64 bit processors, 32 bit processors would use ebx)

    unsafe {
        asm!(
            "push rbx",
            "cpuid",
            "mov [{0}], ebx",
            "mov [{0} + 4], edx",
            "mov [{0} + 8], ecx",
            "pop rbx",
            // We use a pointer to an array for storing the values to simplify
            // the Rust code at the cost of a couple more asm instructions
            // This is more explicit with how the asm works however, as opposed
            // to explicit register outputs such as `out("ecx") val`
            // The *pointer itself* is only an input even though it's written
behind
            in(reg) name_buf.as_mut_ptr(),
            // select cpuid 0, also specify eax as clobbered
            inout("eax") 0 => _,
            // cpuid clobbers these registers too
            out("ecx") _,
            out("edx") _,
        );
    }

    let name = core::str::from_utf8(&name_buf).unwrap();
    println!("CPU Manufacturer ID: {}", name);
}
```

In the example above we use the `cpuid` instruction to read the CPU manufacturer ID. This instruction writes to `eax` with the maximum supported `cpuid` argument and `ebx`, `esx`, and `ecx` with the CPU manufacturer ID as ASCII bytes in that order.

Even though `eax` is never read we still need to tell the compiler that the register has been modified so that the compiler can save any values that were in these registers before the

asm. This is done by declaring it as an output but with `_` instead of a variable name, which indicates that the output value is to be discarded.

This code also works around the limitation that `ebx` is a reserved register by LLVM. That means that LLVM assumes that it has full control over the register and it must be restored to its original state before exiting the asm block, so it cannot be used as an output. To work around this we save the register via `push`, read from `ebx` inside the asm block into a temporary register allocated with `out(reg)` and then restoring `ebx` to its original state via `pop`. The `push` and `pop` use the full 64-bit `rbx` version of the register to ensure that the entire register is saved. On 32 bit targets the code would instead use `ebx` in the `push` / `pop`.

This can also be used with a general register class (e.g. `reg`) to obtain a scratch register for use inside the asm code:

```
use std::arch::asm;

// Multiply x by 6 using shifts and adds
let mut x: u64 = 4;
unsafe {
    asm!(
        "mov {tmp}, {x}",
        "shl {tmp}, 1",
        "shl {x}, 2",
        "add {x}, {tmp}",
        x = inout(reg) x,
        tmp = out(reg) _,
    );
}
assert_eq!(x, 4 * 6);
```

## Symbol operands and ABI clobbers

By default, `asm!` assumes that any register not specified as an output will have its contents preserved by the assembly code. The `clobber_abi` argument to `asm!` tells the compiler to automatically insert the necessary clobber operands according to the given calling convention ABI: any register which is not fully preserved in that ABI will be treated as clobbered. Multiple `clobber_abi` arguments may be provided and all clobbers from all specified ABIs will be inserted.

```
use std::arch::asm;

extern "C" fn foo(arg: i32) -> i32 {
    println!("arg = {}", arg);
    arg * 2
}

fn call_foo(arg: i32) -> i32 {
    unsafe {
        let result;
        asm!(
            "call *{}",
            // Function pointer to call
            in(reg) foo,
            // 1st argument in rdi
            in("rdi") arg,
            // Return value in rax
            out("rax") result,
            // Mark all registers which are not preserved by the "C" calling
            // convention as clobbered.
            clobber_abi("C"),
        );
        result
    }
}
```

## Register template modifiers

In some cases, fine control is needed over the way a register name is formatted when inserted into the template string. This is needed when an architecture's assembly language has several names for the same register, each typically being a "view" over a subset of the register (e.g. the low 32 bits of a 64-bit register).

By default the compiler will always choose the name that refers to the full register size (e.g. `rax` on x86-64, `eax` on x86, etc).

This default can be overridden by using modifiers on the template string operands, just like you would with format strings:

```
use std::arch::asm;

let mut x: u16 = 0xab;

unsafe {
    asm!("mov {0:h}, {0:l}", inout(reg_abcd) x);
}

assert_eq!(x, 0xabab);
```

In this example, we use the `reg_abcd` register class to restrict the register allocator to the 4 legacy x86 registers ( `ax` , `bx` , `cx` , `dx` ) of which the first two bytes can be addressed independently.

Let us assume that the register allocator has chosen to allocate `x` in the `ax` register. The `h` modifier will emit the register name for the high byte of that register and the `l` modifier will emit the register name for the low byte. The asm code will therefore be expanded as `mov ah, al` which copies the low byte of the value into the high byte.

If you use a smaller data type (e.g. `u16` ) with an operand and forget the use template modifiers, the compiler will emit a warning and suggest the correct modifier to use.

## Memory address operands

Sometimes assembly instructions require operands passed via memory addresses/memory locations. You have to manually use the memory address syntax specified by the target architecture. For example, on x86/x86_64 using Intel assembly syntax, you should wrap inputs/outputs in `[]` to indicate they are memory operands:

```
use std::arch::asm;

fn load_fpu_control_word(control: u16) {
    unsafe {
        asm!("fldcw [{}]", in(reg) &control, options(nostack));
    }
}
```

## Labels

Any reuse of a named label, local or otherwise, can result in an assembler or linker error or may cause other strange behavior. Reuse of a named label can happen in a variety of ways including:

- explicitly: using a label more than once in one `asm!` block, or multiple times across blocks.
- implicitly via inlining: the compiler is allowed to instantiate multiple copies of an `asm!` block, for example when the function containing it is inlined in multiple places.
- implicitly via LTO: LTO can cause code from *other crates* to be placed in the same codegen unit, and so could bring in arbitrary labels.

As a consequence, you should only use GNU assembler **numeric** local labels inside inline assembly code. Defining symbols in assembly code may lead to assembler and/or linker

errors due to duplicate symbol definitions.

Moreover, on x86 when using the default Intel syntax, due to an LLVM bug, you shouldn't use labels exclusively made of `0` and `1` digits, e.g. `0`, `11` or `101010`, as they may end up being interpreted as binary values. Using `options(att_syntax)` will avoid any ambiguity, but that affects the syntax of the *entire* `asm!` block. (See Options, below, for more on `options`.)

```
use std::arch::asm;

let mut a = 0;
unsafe {
    asm!(
        "mov {0}, 10",
        "2:",
        "sub {0}, 1",
        "cmp {0}, 3",
        "jle 2f",
        "jmp 2b",
        "2:",
        "add {0}, 2",
        out(reg) a
    );
}
assert_eq!(a, 5);
```

This will decrement the `{0}` register value from 10 to 3, then add 2 and store it in `a`.

This example shows a few things:

- First, that the same number can be used as a label multiple times in the same inline block.
- Second, that when a numeric label is used as a reference (as an instruction operand, for example), the suffixes “b” (“backward”) or ”f” (“forward”) should be added to the numeric label. It will then refer to the nearest label defined by this number in this direction.

# Options

By default, an inline assembly block is treated the same way as an external FFI function call with a custom calling convention: it may read/write memory, have observable side effects, etc. However, in many cases it is desirable to give the compiler more information about what the assembly code is actually doing so that it can optimize better.

Let's take our previous example of an `add` instruction:

```
use std::arch::asm;

let mut a: u64 = 4;
let b: u64 = 4;
unsafe {
    asm!(
        "add {0}, {1}",
        inlateout(reg) a, in(reg) b,
        options(pure, nomem, nostack),
    );
}
assert_eq!(a, 8);
```

Options can be provided as an optional final argument to the `asm!` macro. We specified three options here:

- `pure` means that the asm code has no observable side effects and that its output depends only on its inputs. This allows the compiler optimizer to call the inline asm fewer times or even eliminate it entirely.
- `nomem` means that the asm code does not read or write to memory. By default the compiler will assume that inline assembly can read or write any memory address that is accessible to it (e.g. through a pointer passed as an operand, or a global).
- `nostack` means that the asm code does not push any data onto the stack. This allows the compiler to use optimizations such as the stack red zone on x86-64 to avoid stack pointer adjustments.

These allow the compiler to better optimize code using `asm!`, for example by eliminating pure `asm!` blocks whose outputs are not needed.

See the reference for the full list of available options and their effects.

# macro

# Tests

# Write Tests

# Benchmark

https://doc.rust-lang.org/unstable-book/library-features/test.html

# Unit and Integration

# Assertions

# Async/Await

# async and await!

# Future

# Pin and Unpin

# Stream