通过例子学 Rust

中文翻译注（Chinese translation of the Rust By Example）：

- 👉 查看更多 **Rust 官方文档中英文双语教程**，包括双语版《Rust 程序设计语言》（出版书名为《Rust 权威指南》），本站还提供了 Rust 标准库中文版。
- 《通过例子学 Rust》(Rust By Example 中文版)翻译自 Rust By Example，中文版最后更新时间：2022-1-26。查看此书的 Github 翻译项目和源码。
- 本站支持文档中英文切换，点击页面右上角语言图标可切换到相同章节的英文页面，**英文版每天都会自动同步一次官方的最新版本**。
- 若发现本页表达错误或帮助我们改进翻译，可点击右上角的编辑按钮打开本页对应源码文件进行编辑和修改，Rust 中文资源的开源组织发展离不开大家，感谢您的支持和帮助！

Rust 是一门注重安全（safety）、速度（speed）和并发（concurrency）的现代系统编程语言。Rust 通过内存安全来实现以上目标，但不使用垃圾回收机制（garbage collection, GC）。

《通过例子学 Rust》（Rust By Example, RBE）内容由一系列可运行的实例组成，通过这些例子阐明了各种 Rust 的概念和基本库。想获取这些例子外的更多内容，不要忘了安装 Rust 到本地并查阅官方标准库文档。另外为了满足您的好奇心，您还可以查阅本网站的源代码。

现在让我们开始学习吧！

- Hello World - 从经典的 “Hello World” 程序开始学习。
- 原生类型 - 学习有符号整型，无符号整型和其他原生类型。
- 自定义类型 - 结构体 `struct` 和 枚举 `enum`。
- 变量绑定 - 变量绑定，作用域，变量遮蔽。
- 类型系统 - 学习改变和定义类型。
- 类型转换
- 表达式
- 流程控制 - `if`/`else`， `for`，以及其他流程控制有关内容。
- 函数 - 学习方法、闭包和高阶函数。
- 模块 - 使用模块来组织代码。
- Crate - crate 是 Rust 中的编译单元。学习创建一个库。
- Cargo - 学习官方的 Rust 包管理工具的一些基本功能。

- 属性 - 属性是应用于某些模块、crate 或项的元数据（metadata）。
- 泛型 - 学习编写能够适用于多种类型参数的函数或数据类型。
- 作用域规则 - 作用域在所有权（ownership）、借用（borrowing）和生命周期（lifetime）中起着重要作用。
- 特性 trait - trait 是对未知类型（`Self`）定义的方法集。
- 宏
- 错误处理 - 学习 Rust 语言处理失败的方式。
- 标准库类型 - 学习 `std` 标准库提供的一些自定义类型。
- 标准库更多介绍 - 更多关于文件处理、线程的自定义类型。
- 测试 - Rust 语言的各种测试手段。
- 不安全操作
- 兼容性
- 补充 - 文档和基准测试

Hello World

这是传统的 Hello World 程序的源码。

```
// 这是注释内容，将会被编译器忽略掉
// 可以单击那边的按钮 "Run" 来测试这段代码 ->
// 若想用键盘操作，可以使用快捷键 "Ctrl + Enter" 来运行

// 这段代码支持编辑，你可以自由地修改代码！
// 通过单击 "Reset" 按钮可以使代码恢复到初始状态 ->

// 这是主函数
fn main() {
    // 调用编译生成的可执行文件时，这里的语句将被运行。

    // 将文本打印到控制台
    println!("Hello World!");
}
```

`println!` 是一个宏（macros），可以将文本输出到控制台（console）。

使用 Rust 的编译器 `rustc` 可以从源程序生成可执行文件：

```
$ rustc hello.rs
```

使用 `rustc` 编译后将得到可执行文件 `hello`。

```
$ ./hello
Hello World!
```

动手试一试

单击上面的 "Run" 按钮并观察输出结果。然后增加一行代码，再一次使用宏 `println!`，得到下面结果：

```
Hello World!
I'm a Rustacean!
```

注释

注释对任何程序都不可缺少，同样 Rust 支持几种不同的注释方式。

- **普通注释**，其内容将被编译器忽略掉：
  - `// 单行注释，注释内容直到行尾。`
  - `/* 块注释，注释内容一直到结束分隔符。 */`
- **文档注释**，其内容将被解析成 HTML 帮助文档：
  - `/// 为接下来的项生成帮助文档。`
  - `//! 为注释所属于的项（译注：如 crate、模块或函数）生成帮助文档。`

```
fn main() {
    // 这是行注释的例子
    // 注意有两个斜线在本行的开头
    // 在这里面的所有内容都不会被编译器读取

    // println!("Hello, world!");

    // 请运行一下，你看到结果了吗？现在请将上述语句的两条斜线删掉，并重新运行。

    /*
     * 这是另外一种注释——块注释。一般而言，行注释是推荐的注释格式，
     * 不过块注释在临时注释大块代码特别有用。/* 块注释可以 /* 嵌套，*/ */
     * 所以只需很少按键就可注释掉这些 main() 函数中的行。/*/*/* 自己试试！ */*/*/
     */

    /*
    注意，上面的例子中纵向都有 `*`，这只是一种风格，实际上这并不是必须的。
    */

    // 观察块注释是如何简单地对表达式进行修改的，行注释则不能这样。
    // 删除注释分隔符将会改变结果。
    let x = 5 + /* 90 + */ 5;
    println!("Is `x` 10 or 100? x = {}", x);
}
```

参见:

文档注释

格式化输出

打印操作由 `std::fmt` 里面所定义的一系列宏来处理，包括：

- `format!`：将格式化文本写到字符串。
- `print!`：与 `format!` 类似，但将文本输出到控制台（io::stdout）。
- `println!`：与 `print!` 类似，但输出结果追加一个换行符。
- `eprint!`：与 `print!` 类似，但将文本输出到标准错误（io::stderr）。
- `eprintln!`：与 `eprint!` 类似，但输出结果追加一个换行符。

这些宏都以相同的做法解析文本。有个额外优点是格式化的正确性会在编译时检查。

```rust
fn main() {
    // 通常情况下，`{}` 会被任意变量内容所替换。
    // 变量内容会转化成字符串。
    println!("{} days", 31);

    // 不加后缀的话，31 就自动成为 i32 类型。
    // 你可以添加后缀来改变 31 的类型（例如使用 31i64 声明 31 为 i64 类型）。

    // 用变量替换字符串有多种写法。
    // 比如可以使用位置参数。
    println!("{0}, this is {1}. {1}, this is {0}", "Alice", "Bob");

    // 可以使用命名参数。
    println!("{subject} {verb} {object}",
             object="the lazy dog",
             subject="the quick brown fox",
             verb="jumps over");

    // 可以在 `:` 后面指定特殊的格式。
    println!("{} of {:b} people know binary, the other half don't", 1, 2);

    // 你可以按指定宽度来右对齐文本。
    // 下面语句输出 "     1"，5 个空格后面连着 1。
    println!("{number:>width$}", number=1, width=6);

    // 你可以在数字左边补 0。下面语句输出 "000001"。
    println!("{number:>0width$}", number=1, width=6);

    // println! 会检查使用到的参数数量是否正确。
    println!("My name is {0}, {1} {0}", "Bond");
    // 改正 ^ 补上漏掉的参数："James"

    // 创建一个包含单个 `i32` 的结构体（structure）。命名为 `Structure`。
    #[allow(dead_code)]
    struct Structure(i32);

    // 但是像结构体这样的自定义类型需要更复杂的方式来处理。
    // 下面语句无法运行。
    println!("This struct `{}` won't print...", Structure(3));
    // 改正 ^ 注释掉此行。
}
```

`std::fmt` 包含多种 `trait` （特质）来控制文字显示，其中重要的两种 trait 的基本形式如下：

- `fmt::Debug`：使用 `{:?}` 标记。格式化文本以供调试使用。
- `fmt::Display`：使用 `{}` 标记。以更优雅和友好的风格来格式化文本。

上例使用了 `fmt::Display`，因为标准库提供了那些类型的实现。若要打印自定义类型的文本，需要更多的步骤。

动手试一试

- 改正上面代码中的两个错误（见 “改正”），使它可以没有错误地运行。
- 再用一个 `println!` 宏，通过控制显示的小数位数来打印：`Pi is roughly 3.142`（Pi 约等于 3.142）。为了达到练习目的，使用 `let pi = 3.141592` 作为 Pi 的近似值（提示：设置小数位的显示格式可以参考文档 `std::fmt`）。

参见：

`std::fmt`，`macros`，`struct` 和 `trait`

调试（Debug）

所有的类型，若想用 `std::fmt` 的格式化打印，都要求实现至少一个可打印的 `traits`。仅有一些类型提供了自动实现，比如 `std` 库中的类型。所有其他类型都**必须**手动实现。

`fmt::Debug` 这个 `trait` 使这项工作变得相当简单。所有类型都能推导（`derive`，即自动创建）`fmt::Debug` 的实现。但是 `fmt::Display` 需要手动实现。

```
// 这个结构体不能使用 `fmt::Display` 或 `fmt::Debug` 来进行打印。
struct UnPrintable(i32);

// `derive` 属性会自动创建所需的实现，使这个 `struct` 能使用 `fmt::Debug` 打印。
#[derive(Debug)]
struct DebugPrintable(i32);
```

所有 `std` 库类型都天生可以使用 `{:?}` 来打印：

```
// 推导 `Structure` 的 `fmt::Debug` 实现。
// `Structure` 是一个包含单个 `i32` 的结构体。
#[derive(Debug)]
struct Structure(i32);

// 将 `Structure` 放到结构体 `Deep` 中。然后使 `Deep` 也能够打印。
#[derive(Debug)]
struct Deep(Structure);

fn main() {
    // 使用 `{:?}` 打印和使用 `{}` 类似。
    println!("{:?} months in a year.", 12);
    println!("{1:?} {0:?} is the {actor:?} name.",
             "Slater",
             "Christian",
             actor="actor's");

    // `Structure` 也可以打印！
    println!("Now {:?} will print!", Structure(3));

    // 使用 `derive` 的一个问题是不能控制输出的形式。
    // 假如我只想展示一个 `7` 怎么办？
    println!("Now {:?} will print!", Deep(Structure(7)));
}
```

所以 `fmt::Debug` 确实使这些内容可以打印，但是牺牲了一些美感。Rust 也通过 `{:#?}` 提供了“美化打印”的功能：

```
#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
    age: u8
}

fn main() {
    let name = "Peter";
    let age = 27;
    let peter = Person { name, age };

    // 美化打印
    println!("{:#?}", peter);
}
```

你可以通过手动实现 `fmt::Display` 来控制显示效果。

参见：

attributes, `derive`, `std::fmt` 和 `struct`

显示（Display）

`fmt::Debug` 通常看起来不太简洁，因此自定义输出的外观经常是更可取的。这需要通过手动实现 `fmt::Display` 来做到。`fmt::Display` 采用 `{}` 标记。实现方式看起来像这样：

```
// （使用 `use`）导入 `fmt` 模块使 `fmt::Display` 可用
use std::fmt;

// 定义一个结构体，咱们会为它实现 `fmt::Display`。以下是个简单的元组结构体
// `Structure`，包含一个 `i32` 元素。
struct Structure(i32);

// 为了使用 `{}` 标记，必须手动为类型实现 `fmt::Display` trait。
impl fmt::Display for Structure {
    // 这个 trait 要求 `fmt` 使用与下面的函数完全一致的函数签名
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // 仅将 self 的第一个元素写入到给定的输出流 `f`。返回 `fmt:Result`，此
        // 结果表明操作成功或失败。注意 `write!` 的用法和 `println!` 很相似。
        write!(f, "{}", self.0)
    }
}
```

`fmt::Display` 的效果可能比 `fmt::Debug` 简洁，但对于 `std` 库来说，这就有一个问题。模棱两可的类型该如何显示呢？举个例子，假设标准库对所有的 `Vec<T>` 都实现了同一种输出样式，那么它应该是哪种样式？下面两种中的一种吗？

- `Vec<path>`：`/:/etc:/home/username:/bin`（使用 `:` 分割）
- `Vec<number>`：`1,2,3`（使用 `,` 分割）

我们没有这样做，因为没有一种合适的样式适用于所有类型，标准库也并不擅自规定一种样式。对于 `Vec<T>` 或其他任意泛型容器（generic container），`fmt::Display` 都没有实现。因此在这些泛型的情况下要用 `fmt::Debug`。

这并不是一个问题，因为对于任何**非**泛型的**容器**类型，`fmt::Display` 都能够实现。

```rust
use std::fmt; // 导入 `fmt`

// 带有两个数字的结构体。推导出 `Debug`，以便与 `Display` 的输出进行比较。
#[derive(Debug)]
struct MinMax(i64, i64);

// 实现 `MinMax` 的 `Display`。
impl fmt::Display for MinMax {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // 使用 `self.number` 来表示各个数据。
        write!(f, "({}, {})", self.0, self.1)
    }
}

// 为了比较，定义一个含有具名字段的结构体。
#[derive(Debug)]
struct Point2D {
    x: f64,
    y: f64,
}

// 类似地对 `Point2D` 实现 `Display`
impl fmt::Display for Point2D {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // 自定义格式，使得仅显示 `x` 和 `y` 的值。
        write!(f, "x: {}, y: {}", self.x, self.y)
    }
}

fn main() {
    let minmax = MinMax(0, 14);

    println!("Compare structures:");
    println!("Display: {}", minmax);
    println!("Debug: {:?}", minmax);

    let big_range =   MinMax(-300, 300);
    let small_range = MinMax(-3, 3);

    println!("The big range is {big} and the small is {small}",
             small = small_range,
             big = big_range);

    let point = Point2D { x: 3.3, y: 7.2 };

    println!("Compare points:");
    println!("Display: {}", point);
    println!("Debug: {:?}", point);

    // 报错。`Debug` 和 `Display` 都被实现了，但 `{:b}` 需要 `fmt::Binary`
    // 得到实现。这语句不能运行。
    // println!("What does Point2D look like in binary: {:b}?", point);
}
```

`fmt::Display` 被实现了，而 `fmt::Binary` 没有，因此 `fmt::Binary` 不能使用。 `std::fmt` 有很多这样的 `trait`，它们都要求有各自的实现。这些内容将在后面的 `std::fmt` 章节中详细介绍。

动手试一试

检验上面例子的输出，然后在示例程序中，仿照 `Point2D` 结构体增加一个复数结构体。 使用一样的方式打印，输出结果要求是这个样子：

```
Display: 3.3 + 7.2i
Debug: Complex { real: 3.3, imag: 7.2 }
```

参见：

`derive`, `std::fmt`, macros, `struct`, `trait`, 和 use

测试实例：List

对一个结构体实现 `fmt::Display`，其中的元素需要一个接一个地处理到，这可能会很麻烦。问题在于每个 `write!` 都要生成一个 `fmt::Result`。正确的实现需要处理**所有**的 Result。Rust 专门为解决这个问题提供了 `?` 操作符。

在 `write!` 上使用 `?` 会像是这样：

```
// 对 `write!` 进行尝试（try），观察是否出错。若发生错误，返回相应的错误。
// 否则（没有出错）继续执行后面的语句。
write!(f, "{}", value)?;
```

另外，你也可以使用 `try!` 宏，它和 `?` 是一样的。这种写法比较罗嗦，故不再推荐， 但在老一些的 Rust 代码中仍会看到。使用 `try!` 看起来像这样：

```
try!(write!(f, "{}", value));
```

有了 `?`，对一个 `Vec` 实现 `fmt::Display` 就很简单了：

```
use std::fmt; // 导入 `fmt` 模块。

// 定义一个包含单个 `Vec` 的结构体 `List`。
struct List(Vec<i32>);

impl fmt::Display for List {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // 使用元组的下标获取值，并创建一个 `vec` 的引用。
        let vec = &self.0;

        write!(f, "[")?;

        // 使用 `v` 对 `vec` 进行迭代，并用 `count` 记录迭代次数。
        for (count, v) in vec.iter().enumerate() {
            // 对每个元素（第一个元素除外）加上逗号。
            // 使用 `?` 或 `try!` 来返回错误。
            if count != 0 { write!(f, ", ")?; }
            write!(f, "{}", v)?;
        }

        // 加上配对中括号，并返回一个 fmt::Result 值。
        write!(f, "]")
    }
}

fn main() {
    let v = List(vec![1, 2, 3]);
    println!("{}", v);
}
```

动手试一试：

更改程序使 vector 里面每个元素的下标也能够打印出来。新的结果如下：

```
[0: 1, 1: 2, 2: 3]
```

参见：

`for`，`ref`，`Result`，`struct`，`?`，和 `vec!`

格式化

我们已经看到，格式化的方式是通过**格式字符串**来指定的：

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

根据使用的**参数类型**是 `X`、`o` 还是**未指定**，同样的变量（`foo`）能够格式化成不同的形式。

这个格式化的功能是通过 trait 实现的，每种参数类型都对应一种 trait。最常见的格式化 trait 就是 `Display`，它可以处理参数类型为未指定的情况，比如 `{}`。

```rust
use std::fmt::{self, Formatter, Display};

struct City {
    name: &'static str,
    // 纬度
    lat: f32,
    // 经度
    lon: f32,
}

impl Display for City {
    // `f` 是一个缓冲区（buffer），此方法必须将格式化后的字符串写入其中
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        let lat_c = if self.lat >= 0.0 { 'N' } else { 'S' };
        let lon_c = if self.lon >= 0.0 { 'E' } else { 'W' };

        // `write!` 和 `format!` 类似，但它会将格式化后的字符串写入
        // 一个缓冲区（即第一个参数f）中。
        write!(f, "{}: {:.3}°{} {:.3}°{}",
               self.name, self.lat.abs(), lat_c, self.lon.abs(), lon_c)
    }
}

#[derive(Debug)]
struct Color {
    red: u8,
    green: u8,
    blue: u8,
}

fn main() {
    for city in [
        City { name: "Dublin", lat: 53.347778, lon: -6.259722 },
        City { name: "Oslo", lat: 59.95, lon: 10.75 },
        City { name: "Vancouver", lat: 49.25, lon: -123.1 },
    ].iter() {
        println!("{}", *city);
    }
    for color in [
        Color { red: 128, green: 255, blue: 90 },
        Color { red: 0, green: 3, blue: 254 },
        Color { red: 0, green: 0, blue: 0 },
    ].iter() {
        // 在添加了针对 fmt::Display 的实现后，请改用 {} 检验效果。
        println!("{:?}", *color)
    }
}
```

在 `fmt::fmt` 文档中可以查看格式化 traits 一览表和它们的参数类型。

动手试一试

为上面的 `Color` 结构体实现 `fmt::Display`，应得到如下的输出结果：

```
RGB (128, 255, 90) 0x80FF5A
RGB (0, 3, 254) 0x0003FE
RGB (0, 0, 0) 0x000000
```

如果感到疑惑，可看下面两条提示：

- 你可能需要多次列出每个颜色，
- 你可以使用 `:02` 补零使位数为 2 位。

参见：

`std::fmt`

原生类型

Rust 提供了多种原生类型（`primitives`），包括：

标量类型（scalar type）

- 有符号整数（signed integers）：`i8`、`i16`、`i32`、`i64`、`i128` 和 `isize`（指针宽度）
- 无符号整数（unsigned integers）：`u8`、`u16`、`u32`、`u64`、`u128` 和 `usize`（指针宽度）
- 浮点数（floating point）：`f32`、`f64`
- `char`（字符）：单个 Unicode 字符，如 `'a'`，`'α'` 和 `'∞'`（每个都是 4 字节）
- `bool`（布尔型）：只能是 `true` 或 `false`
- 单元类型（unit type）：`()`。其唯一可能的值就是 `()` 这个空元组

尽管单元类型的值是个元组，它却并不被认为是复合类型，因为并不包含多个值。

复合类型（compound type）

- 数组（array）：如 `[1, 2, 3]`
- 元组（tuple）：如 `(1, true)`

变量都能够显式地给出**类型说明**（type annotation）。数字还可以通过**后缀**（suffix）或**默认方式**来声明类型。整型默认为 `i32` 类型，浮点型默认为 `f64` 类型。注意 Rust 还可以根据上下文来推断（infer）类型（译注：比如一个未声明类型整数和 `i64` 的整数相加，则该整数会自动推断为 `i64` 类型。仅当根据环境无法推断时，才按默认方式取整型数值为 `i32`，浮点数值为 `f64`）。

```rust
fn main() {
    // 变量可以给出类型说明。
    let logical: bool = true;

    let a_float: f64 = 1.0;  // 常规说明
    let an_integer   = 5i32; // 后缀说明

    // 否则会按默认方式决定类型。
    let default_float   = 3.0; // `f64`
    let default_integer = 7;   // `i32`

    // 类型也可根据上下文自动推断。
    let mut inferred_type = 12; // 根据下一行的赋值推断为 i64 类型
    inferred_type = 4294967296i64;

    // 可变的（mutable）变量，其值可以改变。
    let mut mutable = 12; // Mutable `i32`
    mutable = 21;

    // 报错！变量的类型并不能改变。
    mutable = true;

    // 但可以用遮蔽（shadow）来覆盖前面的变量。
    let mutable = true;
}
```

参见：

`std` 库、`mut`、类型推断 和 变量遮蔽

字面量和运算符

整数 `1`、浮点数 `1.2`、字符 `'a'`、字符串 `"abc"`、布尔值 `true` 和单元类型 `()` 可以用数字、文字或符号之类的“字面量”（literal）来表示。

另外，通过加前缀 `0x`、`0o`、`0b`，数字可以用十六进制、八进制或二进制记法表示。

为了改善可读性，可以在数值字面量中插入下划线，比如：`1_000` 等同于 `1000`，`0.000_001` 等同于 `0.000001`。

我们需要把字面量的类型告诉编译器。如前面学过的，我们使用 `u32` 后缀来表明字面量是一个 32 位无符号整数，`i32` 后缀表明字面量是一个 32 位有符号整数。

Rust 提供了一系列的运算符（operator），它们的优先级和类 C 语言类似。（译注：类 C 语言包括 C/C++、Java、PHP 等语言）

```
fn main() {
    // 整数相加
    println!("1 + 2 = {}", 1u32 + 2);

    // 整数相减
    println!("1 - 2 = {}", 1i32 - 2);
    // 试一试 ^ 尝试将 `1i32` 改为 `1u32`，体会为什么类型声明这么重要

    // 短路求值的布尔逻辑
    println!("true AND false is {}", true && false);
    println!("true OR false is {}", true || false);
    println!("NOT true is {}", !true);

    // 位运算
    println!("0011 AND 0101 is {:04b}", 0b0011u32 & 0b0101);
    println!("0011 OR 0101 is {:04b}", 0b0011u32 | 0b0101);
    println!("0011 XOR 0101 is {:04b}", 0b0011u32 ^ 0b0101);
    println!("1 << 5 is {}", 1u32 << 5);
    println!("0x80 >> 2 is 0x{:x}", 0x80u32 >> 2);

    // 使用下划线改善数字的可读性！
    println!("One million is written as {}", 1_000_000u32);
}
```

元组

元组是一个可以包含各种类型值的组合。元组使用括号 `()` 来构造（construct），而每个元组自身又是一个类型标记为 `(T1, T2, ...)` 的值，其中 `T1`、`T2` 是每个元素的类型。函数可以使用元组来返回多个值，因为元组可以拥有任意多个值。

```rust
// 元组可以充当函数的参数和返回值
fn reverse(pair: (i32, bool)) -> (bool, i32) {
    // 可以使用 `let` 把一个元组的成员绑定到一些变量
    let (integer, boolean) = pair;

    (boolean, integer)
}

// 在“动手试一试”的练习中要用到下面这个结构体。
#[derive(Debug)]
struct Matrix(f32, f32, f32, f32);

fn main() {
    // 包含各种不同类型的元组
    let long_tuple = (1u8, 2u16, 3u32, 4u64,
                      -1i8, -2i16, -3i32, -4i64,
                      0.1f32, 0.2f64,
                      'a', true);

    // 通过元组的下标来访问具体的值
    println!("long tuple first value: {}", long_tuple.0);
    println!("long tuple second value: {}", long_tuple.1);

    // 元组也可以充当元组的元素
    let tuple_of_tuples = ((1u8, 2u16, 2u32), (4u64, -1i8), -2i16);

    // 元组可以打印
    println!("tuple of tuples: {:?}", tuple_of_tuples);

    // 但很长的元组无法打印
    // let too_long_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13);
    // println!("too long tuple: {:?}", too_long_tuple);
    // 试一试 ^ 取消上面两行的注释，阅读编译器给出的错误信息。

    let pair = (1, true);
    println!("pair is {:?}", pair);

    println!("the reversed pair is {:?}", reverse(pair));

    // 创建单元素元组需要一个额外的逗号，这是为了和被括号包含的字面量作区分。
    println!("one element tuple: {:?}", (5u32,));
    println!("just an integer: {:?}", (5u32));

    // 元组可以被解构（deconstruct），从而将值绑定给变量
    let tuple = (1, "hello", 4.5, true);

    let (a, b, c, d) = tuple;
    println!("{:?}, {:?}, {:?}, {:?}", a, b, c, d);

    let matrix = Matrix(1.1, 1.2, 2.1, 2.2);
    println!("{:?}", matrix)

}
```

动手试一试

1. **复习**：在上面的例子中给 Matrix `结构体` 加上 `fmt::Display` trait，这样当你从 Debug 格式化 `{:?}` 切换到 Display 格式化 `{}` 时，会得到如下的输出：

   ```
   ( 1.1 1.2 )
   ( 2.1 2.2 )
   ```

   可以回顾之前学过的显示（display）的例子。

2. 以 `reverse` 函数作为样板，写一个 `transpose` 函数，它可以接受一个 Matrix 作为参数，并返回一个右上 - 左下对角线上的两元素交换后的 Matrix。举个例子：

   ```
   println!("Matrix:\n{}", matrix);
   println!("Transpose:\n{}", transpose(matrix));
   ```

   输出结果：

   ```
   Matrix:
   ( 1.1 1.2 )
   ( 2.1 2.2 )
   Transpose:
   ( 1.1 2.1 )
   ( 1.2 2.2 )
   ```

数组和切片

数组（array）是一组拥有相同类型 `T` 的对象的集合，在内存中是连续存储的。数组使用中括号 `[]` 来创建，且它们的大小在编译时会被确定。数组的类型标记为 `[T; length]`（译注：`T` 为元素类型，`length` 表示数组大小）。

切片（slice）类型和数组类似，但其大小在编译时是不确定的。相反，切片是一个双字对象（two-word object），第一个字是一个指向数据的指针，第二个字是切片的长度。这个 “字” 的宽度和 usize 相同，由处理器架构决定，比如在 x86-64 平台上就是 64 位。slice 可以用来借用数组的一部分。slice 的类型标记为 `&[T]`。

```
use std::mem;

// 此函数借用一个 slice
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // 定长数组（类型标记是多余的）
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // 所有元素可以初始化成相同的值
    let ys: [i32; 500] = [0; 500];

    // 下标从 0 开始
    println!("first element of the array: {}", xs[0]);
    println!("second element of the array: {}", xs[1]);

    // `len` 返回数组的大小
    println!("array size: {}", xs.len());

    // 数组是在栈中分配的
    println!("array occupies {} bytes", mem::size_of_val(&xs));

    // 数组可以自动被借用成为 slice
    println!("borrow the whole array as a slice");
    analyze_slice(&xs);

    // slice 可以指向数组的一部分
    println!("borrow a section of the array as a slice");
    analyze_slice(&ys[1 .. 4]);

    // 越界的下标会引发致命错误（panic）
    println!("{}", xs[5]);
}
```

自定义类型

Rust 自定义数据类型主要是通过下面这两个关键字来创建：

- `struct`： 定义一个结构体（structure）
- `enum`： 定义一个枚举类型（enumeration）

而常量（constant）可以通过 `const` 和 `static` 关键字来创建。

结构体

结构体（structure，缩写成 struct）有 3 种类型，使用 `struct` 关键字来创建：

- 元组结构体（tuple struct），事实上就是具名元组而已。
- 经典的 C 语言风格结构体（C struct）。
- 单元结构体（unit struct），不带字段，在泛型中很有用。

```rust
#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

// 单元结构体
struct Unit;

// 元组结构体
struct Pair(i32, f32);

// 带有两个字段的结构体
struct Point {
    x: f32,
    y: f32,
}

// 结构体可以作为另一个结构体的字段
#[allow(dead_code)]
struct Rectangle {
    // 可以在空间中给定左上角和右下角在空间中的位置来指定矩形。
    top_left: Point,
    bottom_right: Point,
}

fn main() {
    // 使用简单的写法初始化字段，并创建结构体
    let name = String::from("Peter");
    let age = 27;
    let peter = Person { name, age };

    // 以 Debug 方式打印结构体
    println!("{:?}", peter);

    // 实例化结构体 `Point`
    let point: Point = Point { x: 10.3, y: 0.4 };

    // 访问 point 的字段
    println!("point coordinates: ({}, {})", point.x, point.y);

    // 使用结构体更新语法创建新的 point，
    // 这样可以用到之前的 point 的字段
    let bottom_right = Point { x: 5.2, ..point };

    // `bottom_right.y` 与 `point.y` 一样，因为这个字段就是从 `point` 中来的
    println!("second point: ({}, {})", bottom_right.x, bottom_right.y);

    // 使用 `let` 绑定来解构 point
    let Point { x: left_edge, y: top_edge } = point;

    let _rectangle = Rectangle {
        // 结构体的实例化也是一个表达式
        top_left: Point { x: left_edge, y: top_edge },
        bottom_right: bottom_right,
    };

    // 实例化一个单元结构体
```

```
    let _unit = Unit;

    // 实例化一个元组结构体
    let pair = Pair(1, 0.1);

    // 访问元组结构体的字段
    println!("pair contains {:?} and {:?}", pair.0, pair.1);

    // 解构一个元组结构体
    let Pair(integer, decimal) = pair;

    println!("pair contains {:?} and {:?}", integer, decimal);
}
```

动手试一试:

1. 增加一个计算 `Rectangle` （长方形）面积的函数 `rect_area` （尝试使用嵌套的解构方式）。
2. 增加一个函数 `square`，接受的参数是一个 `Point` 和一个 `f32`，并返回一个 `Rectangle` （长方形），其左上角位于该点上，长和宽都对应于 `f32`。

参见:

属性 和 解构

枚举

`enum` 关键字允许创建一个从数个不同取值中选其一的枚举类型（enumeration）。任何一个在 `struct` 中合法的取值在 `enum` 中也合法。

```
// 该属性用于隐藏对未使用代码的警告。
#![allow(dead_code)]

// 创建一个 `enum`（枚举）来对 web 事件分类。注意变量名和类型共同指定了 `enum`
// 取值的种类：`PageLoad` 不等于 `PageUnload`，`KeyPress(char)` 不等于
// `Paste(String)`。各个取值不同，互相独立。
enum WebEvent {
    // 一个 `enum` 可以是单元结构体（称为 `unit-like` 或 `unit`），
    PageLoad,
    PageUnload,
    // 或者一个元组结构体，
    KeyPress(char),
    Paste(String),
    // 或者一个普通的结构体。
    Click { x: i64, y: i64 }
}

// 此函数将一个 `WebEvent` enum 作为参数，无返回值。
fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad => println!("page loaded"),
        WebEvent::PageUnload => println!("page unloaded"),
        // 从 `enum` 里解构出 `c`。
        WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
        WebEvent::Paste(s) => println!("pasted \"{}\".", s),
        // 把 `Click` 解构给 `x` and `y`。
        WebEvent::Click { x, y } => {
            println!("clicked at x={}, y={}.", x, y);
        },
    }
}

fn main() {
    let pressed = WebEvent::KeyPress('x');
    // `to_owned()` 从一个字符串切片中创建一个具有所有权的 `String`。
    let pasted  = WebEvent::Paste("my text".to_owned());
    let click   = WebEvent::Click { x: 20, y: 80 };
    let load    = WebEvent::PageLoad;
    let unload  = WebEvent::PageUnload;

    inspect(pressed);
    inspect(pasted);
    inspect(click);
    inspect(load);
    inspect(unload);
}
```

类型别名

若使用类型别名，则可以通过其别名引用每个枚举变量。当枚举的名称太长或者太一般化，且你想要对其重命名，那么这对你会有所帮助。

```
enum VeryVerboseEnumOfThingsToDoWithNumbers {
    Add,
    Subtract,
}

// 创建一个类型别名
type Operations = VeryVerboseEnumOfThingsToDoWithNumbers;

fn main() {
    // 我们可以通过别名引用每个枚举变量，避免使用又长又不方便的枚举名字
    let x = Operations::Add;
}
```

最常见的情况就是在 `impl` 块中使用 `Self` 别名。

```
enum VeryVerboseEnumOfThingsToDoWithNumbers {
    Add,
    Subtract,
}

impl VeryVerboseEnumOfThingsToDoWithNumbers {
    fn run(&self, x: i32, y: i32) -> i32 {
        match self {
            Self::Add => x + y,
            Self::Subtract => x - y,
        }
    }
}
```

该功能已在 Rust 中稳定下来， 可以阅读 stabilization report 来了解更多有关枚举和类型别名的知识。

参见：

`match`，`fn`，和 `String`，“类型别名枚举变量” 的 RFC

使用 use

使用 `use` 声明的话，就可以不写出名称的完整路径了：

```
// 该属性用于隐藏对未使用代码的警告。
#![allow(dead_code)]

enum Status {
    Rich,
    Poor,
}

enum Work {
    Civilian,
    Soldier,
}

fn main() {
    // 显式地 `use` 各个名称使他们直接可用，而不需要指定它们来自 `Status`。
    use Status::{Poor, Rich};
    // 自动地 `use` `Work` 内部的各个名称。
    use Work::*;

    // `Poor` 等价于 `Status::Poor`。
    let status = Poor;
    // `Civilian` 等价于 `Work::Civilian`。
    let work = Civilian;

    match status {
        // 注意这里没有用完整路径，因为上面显式地使用了 `use`。
        Rich => println!("The rich have lots of money!"),
        Poor => println!("The poor have no money..."),
    }

    match work {
        // 再次注意到没有用完整路径。
        Civilian => println!("Civilians work!"),
        Soldier  => println!("Soldiers fight!"),
    }
}
```

参见：

`match` 和 `use`

C 风格用法

`enum` 也可以像 C 语言风格的枚举类型那样使用。

```
// 该属性用于隐藏对未使用代码的警告。
#![allow(dead_code)]

// 拥有隐式辨别值（implicit discriminator，从 0 开始）的 enum
enum Number {
    Zero,
    One,
    Two,
}

// 拥有显式辨别值（explicit discriminator）的 enum
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // `enum` 可以转成整型。
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

参考：

类型转换

测试实例：链表

`enum` 的一个常见用法就是创建链表（linked-list）：

```rust
use List::*;

enum List {
    // Cons：元组结构体，包含链表的一个元素和一个指向下一节点的指针
    Cons(u32, Box<List>),
    // Nil：末结点，表明链表结束
    Nil,
}

// 可以为 enum 定义方法
impl List {
    // 创建一个空的 List 实例
    fn new() -> List {
        // `Nil` 为 `List` 类型（译注：因 `Nil` 的完整名称是 `List::Nil`）
        Nil
    }

    // 处理一个 List，在其头部插入新元素，并返回该 List
    fn prepend(self, elem: u32) -> List {
        // `Cons` 同样为 List 类型
        Cons(elem, Box::new(self))
    }

    // 返回 List 的长度
    fn len(&self) -> u32 {
        // 必须对 `self` 进行匹配（match），因为这个方法的行为取决于 `self` 的
        // 取值种类。
        // `self` 为 `&List` 类型，`*self` 为 `List` 类型，匹配一个具体的 `T`
        // 类型要好过匹配引用 `&T`。
        match *self {
            // 不能得到 tail 的所有权，因为 `self` 是借用的；
            // 因此使用一个对 tail 的引用
            Cons(_, ref tail) => 1 + tail.len(),
            // （递归的）基准情形（base case）：一个长度为 0 的空列表
            Nil => 0
        }
    }

    // 返回列表的字符串表示（该字符串是堆分配的）
    fn stringify(&self) -> String {
        match *self {
            Cons(head, ref tail) => {
                // `format!` 和 `print!` 类似，但返回的是一个堆分配的字符串，
                // 而不是打印结果到控制台上
                format!("{}, {}", head, tail.stringify())
            },
            Nil => {
                format!("Nil")
            },
        }
    }
}

fn main() {
    // 创建一个空链表
    let mut list = List::new();

    // 追加一些元素
```

```
    list = list.prepend(1);
    list = list.prepend(2);
    list = list.prepend(3);

    // 显示链表的最后状态
    println!("linked list has length: {}", list.len());
    println!("{}", list.stringify());
}
```

参见：

`Box` 和 方法

常量

Rust 有两种常量，可以在任意作用域声明，包括全局作用域。它们都需要显式的类型声明：

- `const`：不可改变的值（通常使用这种）。
- `static`：具有 `'static` 生命周期的，可以是可变的变量（译注：须使用 `static mut` 关键字）。

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。

```
// 全局变量是在所有其他作用域之外声明的。
static LANGUAGE: &'static str = "Rust";
const  THRESHOLD: i32 = 10;

fn is_big(n: i32) -> bool {
    // 在一般函数中访问常量
    n > THRESHOLD
}

fn main() {
    let n = 16;

    // 在 main 函数（主函数）中访问常量
    println!("This is {}", LANGUAGE);
    println!("The threshold is {}", THRESHOLD);
    println!("{} is {}", n, if is_big(n) { "big" } else { "small" });

    // 报错！不能修改一个 `const` 常量。
    THRESHOLD = 5;
    // 改正 ^ 注释掉此行
}
```

参见：

`const` / `static` RFC, `'static` 生命周期

变量绑定

Rust 通过静态类型确保类型安全。变量绑定可以在声明时说明类型，不过在多数情况下，编译器能够从上下文推导出变量的类型，从而大大减少了类型说明的工作。

使用 `let` 绑定操作可以将值（比如字面量）绑定（bind）到变量。

```
fn main() {
    let an_integer = 1u32;
    let a_boolean = true;
    let unit = ();

    // 将 `an_integer` 复制到 `copied_integer`
    let copied_integer = an_integer;

    println!("An integer: {:?}", copied_integer);
    println!("A boolean: {:?}", a_boolean);
    println!("Meet the unit value: {:?}", unit);

    // 编译器会对未使用的变量绑定产生警告；可以给变量名加上下划线前缀来消除警告。
    let _unused_variable = 3u32;

    let noisy_unused_variable = 2u32;
    // 改正 ^ 在变量名前加上下划线以消除警告
}
```

可变变量

变量绑定默认是不可变的（immutable），但加上 `mut` 修饰语后变量就可以改变。

```
fn main() {
    let _immutable_binding = 1;
    let mut mutable_binding = 1;

    println!("Before mutation: {}", mutable_binding);

    // 正确代码
    mutable_binding += 1;

    println!("After mutation: {}", mutable_binding);

    // 错误!
    _immutable_binding += 1;
    // 改正 ^ 将此行注释掉
}
```

编译器会给出关于变量可变性的详细诊断信息。

作用域和遮蔽

变量绑定有一个作用域（scope），它被限定只在一个**代码块**（block）中生存（live）。代码块是一个被 `{}` 包围的语句集合。另外也允许变量遮蔽（variable shadowing）。

```
fn main() {
    // 此绑定生存于 main 函数中
    let long_lived_binding = 1;

    // 这是一个代码块，比 main 函数拥有更小的作用域
    {
        // 此绑定只存在于本代码块
        let short_lived_binding = 2;

        println!("inner short: {}", short_lived_binding);

        // 此绑定*遮蔽*了外面的绑定
        let long_lived_binding = 5_f32;

        println!("inner long: {}", long_lived_binding);
    }
    // 代码块结束

    // 报错！`short_lived_binding` 在此作用域上不存在
    println!("outer short: {}", short_lived_binding);
    // 改正 ^ 注释掉这行

    println!("outer long: {}", long_lived_binding);

    // 此绑定同样*遮蔽*了前面的绑定
    let long_lived_binding = 'a';

    println!("outer long: {}", long_lived_binding);
}
```

变量先声明

可以先声明（declare）变量绑定，后面才将它们初始化（initialize）。但是这种做法很少用，因为这样可能导致使用未初始化的变量。

```
fn main() {
    // 声明一个变量绑定
    let a_binding;

    {
        let x = 2;

        // 初始化一个绑定
        a_binding = x * x;
    }

    println!("a binding: {}", a_binding);

    let another_binding;

    // 报错！使用了未初始化的绑定
    println!("another binding: {}", another_binding);
    // 改正 ^ 注释掉此行

    another_binding = 1;

    println!("another binding: {}", another_binding);
}
```

编译器禁止使用未经初始化的变量，因为这会产生未定义行为（undefined behavior）。

冻结

当数据被相同的名称不变地绑定时，它还会**冻结**（freeze）。在不可变绑定超出作用域之前，无法修改已冻结的数据：

```
fn main() {
    let mut _mutable_integer = 7i32;

    {
        // 被不可变的 `_mutable_integer` 遮蔽
        let _mutable_integer = _mutable_integer;

        // 报错！`_mutable_integer` 在本作用域被冻结
        _mutable_integer = 50;
        // 改正 ^ 注释掉上面这行

        // `_mutable_integer` 离开作用域
    }

    // 正常运行！ `_mutable_integer` 在这个作用域没有冻结
    _mutable_integer = 3;
}
```

类型系统

Rust 提供了多种机制，用于改变或定义原生类型和用户定义类型。接下来会讲到：

- 原生类型的类型转换（cast）。
- 指定字面量的类型。
- 使用类型推断（type inference）。
- 给类型取别名（alias）。

类型转换

Rust 不提供原生类型之间的隐式类型转换（coercion），但可以使用 `as` 关键字进行显式类型转换（casting）。

整型之间的转换大体遵循 C 语言的惯例，除了 C 会产生未定义行为的情形。在 Rust 中所有整型转换都是定义良好的。

```rust
// 不显示类型转换产生的溢出警告。
#![allow(overflowing_literals)]

fn main() {
    let decimal = 65.4321_f32;

    // 错误！不提供隐式转换
    let integer: u8 = decimal;
    // 改正 ^ 注释掉这一行

    // 可以显式转换
    let integer = decimal as u8;
    let character = integer as char;

    println!("Casting: {} -> {} -> {}", decimal, integer, character);

    // 当把任何类型转换为无符号类型 T 时，会不断加上或减去 (std::T::MAX + 1)
    // 直到值位于新类型 T 的范围内。

    // 1000 已经在 u16 的范围内
    println!("1000 as a u16 is: {}", 1000 as u16);

    // 1000 - 256 - 256 - 256 = 232
    // 事实上的处理方式是：从最低有效位（LSB，least significant bits）开始保留
    // 8 位，然后剩余位置，直到最高有效位（MSB，most significant bit）都被抛弃。
    // 译注：MSB 就是二进制的最高位，LSB 就是二进制的最低位，按日常书写习惯就是
    // 最左边一位和最右边一位。
    println!("1000 as a u8 is : {}", 1000 as u8);
    // -1 + 256 = 255
    println!("  -1 as a u8 is : {}", (-1i8) as u8);

    // 对正数，这就和取模一样。
    println!("1000 mod 256 is : {}", 1000 % 256);

    // 当转换到有符号类型时，（位操作的）结果就和“先转换到对应的无符号类型，
    // 如果 MSB 是 1，则该值为负”是一样的。

    // 当然如果数值已经在目标类型的范围内，就直接把它放进去。
    println!(" 128 as a i16 is: {}", 128 as i16);
    // 128 转成 u8 还是 128，但转到 i8 相当于给 128 取八位的二进制补码，其值是：
    println!(" 128 as a i8 is : {}", 128 as i8);

    // 重复之前的例子
    // 1000 as u8 -> 232
    println!("1000 as a u8 is : {}", 1000 as u8);
    // 232 的二进制补码是 -24
    println!(" 232 as a i8 is : {}", 232 as i8);
}
```

字面量

对数值字面量，只要把类型作为后缀加上去，就完成了类型说明。比如指定字面量 `42` 的类型是 `i32`，只需要写 `42i32`。

无后缀的数值字面量，其类型取决于怎样使用它们。如果没有限制，编译器会对整数使用 `i32`，对浮点数使用 `f64`。

```
fn main() {
    // 带后缀的字面量，其类型在初始化时已经知道了。
    let x = 1u8;
    let y = 2u32;
    let z = 3f32;

    // 无后缀的字面量，其类型取决于如何使用它们。
    let i = 1;
    let f = 1.0;

    // `size_of_val` 返回一个变量所占的字节数
    println!("size of `x` in bytes: {}", std::mem::size_of_val(&x));
    println!("size of `y` in bytes: {}", std::mem::size_of_val(&y));
    println!("size of `z` in bytes: {}", std::mem::size_of_val(&z));
    println!("size of `i` in bytes: {}", std::mem::size_of_val(&i));
    println!("size of `f` in bytes: {}", std::mem::size_of_val(&f));
}
```

上面的代码使用了一些还没有讨论过的概念。心急的读者可以看看下面的简短解释：

- `fun(&foo)` 用**传引用**（pass by reference）的方式把变量传给函数，而非传值（pass by value，写法是 `fun(foo)`）。更多细节请看借用。
- `std::mem::size_of_val` 是一个函数，这里使用其**完整路径**（full path）调用。代码可以分成一些叫做**模块**（module）的逻辑单元。在本例中，`size_of_val` 函数是在 `mem` 模块中定义的，而 `mem` 模块又是在 `std` **crate** 中定义的。更多细节请看模块和crate.

类型推断

Rust 的类型推断引擎是很聪明的，它不只是在初始化时看看右值（r-value）的类型而已，它还会考察变量之后会怎样使用，借此推断类型。这是一个类型推导的进阶例子：

```
fn main() {
    // 因为有类型说明，编译器知道 `elem` 的类型是 u8。
    let elem = 5u8;

    // 创建一个空向量（vector，即不定长的，可以增长的数组）。
    let mut vec = Vec::new();
    // 现在编译器还不知道 `vec` 的具体类型，只知道它是某种东西构成的向量（`Vec<_>

    // 在向量中插入 `elem`。
    vec.push(elem);
    // 啊哈！现在编译器知道 `vec` 是 u8 的向量了（`Vec<u8>`）。
    // 试一试 ^ 注释掉 `vec.push(elem)` 这一行。

    println!("{:?}", vec);
}
```

没有必要写类型说明，编译器和程序员皆大欢喜！

别名

可以用 `type` 语句给已有的类型取个新的名字。类型的名字必须遵循驼峰命名法（像是 `CamelCase` 这样），否则编译器将给出警告。原生类型是例外，比如： `usize` 、 `f32` ，等等。

```
// `NanoSecond` 是 `u64` 的新名字。
type NanoSecond = u64;
type Inch = u64;

// 通过这个属性屏蔽警告。
#[allow(non_camel_case_types)]
type u64_t = u64;
// 试一试 ^ 移除上面那个属性

fn main() {
    // `NanoSecond` = `Inch` = `u64_t` = `u64`.
    let nanoseconds: NanoSecond = 5 as u64_t;
    let inches: Inch = 2 as u64_t;

    // 注意类型别名*并不能*提供额外的类型安全，因为别名*并不是*新的类型。
    println!("{} nanoseconds + {} inches = {} unit?",
             nanoseconds,
             inches,
             nanoseconds + inches);
}
```

别名的主要用途是避免写出冗长的模板化代码（boilerplate code）。如 `IoResult<T>` 是 `Result<T, IoError>` 类型的别名。

参见:

属性

类型转换

Rust 使用 trait 解决类型之间的转换问题。最一般的转换会用到 `From` 和 `Into` 两个 trait。不过，即便常见的情况也可能会用到特别的 trait，尤其是从 `String` 转换到别的类型，以及把别的类型转换到 `String` 时。

From 和 Into

From 和 Into 两个 trait 是内部相关联的，实际上这是它们实现的一部分。如果我们能够从类型 B 得到类型 A，那么很容易相信我们也能够把类型 B 转换为类型 A。

From

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。在标准库中有无数 From 的实现，规定原生类型及其他常见类型的转换功能。

比如，可以很容易地把 str 转换成 String：

```
let my_str = "hello";
let my_string = String::from(my_str);
```

也可以为我们自己的类型定义转换机制：

```
use std::convert::From;

#[derive(Debug)]
struct Number {
    value: i32,
}

impl From<i32> for Number {
    fn from(item: i32) -> Self {
        Number { value: item }
    }
}

fn main() {
    let num = Number::from(30);
    println!("My number is {:?}", num);
}
```

Into

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。

使用 Into trait 通常要求指明要转换到的类型，因为编译器大多数时候不能推断它。不过考虑到我们免费获得了 Into，这点代价不值一提。

```rust
use std::convert::From;

#[derive(Debug)]
struct Number {
    value: i32,
}

impl From<i32> for Number {
    fn from(item: i32) -> Self {
        Number { value: item }
    }
}

fn main() {
    let int = 5;
    // 试试删除类型说明
    let num: Number = int.into();
    println!("My number is {:?}", num);
}
```

`TryFrom` and `TryInto`

类似于 `From` 和 `Into`，`TryFrom` 和 `TryInto` 是类型转换的通用 trait。不同于 `From` / `Into` 的是，`TryFrom` 和 `TryInto` trait 用于易出错的转换，也正因如此，其返回值是 `Result` 型。

```
use std::convert::TryFrom;
use std::convert::TryInto;

#[derive(Debug, PartialEq)]
struct EvenNumber(i32);

impl TryFrom<i32> for EvenNumber {
    type Error = ();

    fn try_from(value: i32) -> Result<Self, Self::Error> {
        if value % 2 == 0 {
            Ok(EvenNumber(value))
        } else {
            Err(())
        }
    }
}

fn main() {
    // TryFrom

    assert_eq!(EvenNumber::try_from(8), Ok(EvenNumber(8)));
    assert_eq!(EvenNumber::try_from(5), Err(()));

    // TryInto

    let result: Result<EvenNumber, ()> = 8i32.try_into();
    assert_eq!(result, Ok(EvenNumber(8)));
    let result: Result<EvenNumber, ()> = 5i32.try_into();
    assert_eq!(result, Err(()));
}
```

ToString 和 FromStr

ToString

要把任何类型转换成 `String`，只需要实现那个类型的 `ToString` trait。然而不要直接这么做，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。

```
use std::fmt;

struct Circle {
    radius: i32
}

impl fmt::Display for Circle {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Circle of radius {}", self.radius)
    }
}

fn main() {
    let circle = Circle { radius: 6 };
    println!("{}", circle.to_string());
}
```

译注：一个实现 `ToString` 的例子

```
use std::string::ToString;

struct Circle {
    radius: i32
}

impl ToString for Circle {
    fn to_string(&self) -> String {
        format!("Circle of radius {:?}", self.radius)
    }
}

fn main() {
    let circle = Circle { radius: 6 };
    println!("{}", circle.to_string());
}
```

解析字符串

我们经常需要把字符串转成数字。完成这项工作的标准手段是用 `parse` 函数。我们得提供要转换到的类型，这可以通过不使用类型推断，或者用 “涡轮鱼” 语法（turbo fish， `<>` ）实现。

只要对目标类型实现了 `FromStr` trait，就可以用 `parse` 把字符串转换成目标类型。 标准库中已经给无数种类型实现了 `FromStr` 。如果要转换到用户定义类型，只要手动实现 `FromStr` 就行。

```
fn main() {
    let parsed: i32 = "5".parse().unwrap();
    let turbo_parsed = "10".parse::<i32>().unwrap();

    let sum = parsed + turbo_parsed;
    println!{"Sum: {:?}", sum};
}
```

表达式

Rust 程序（大部分）由一系列语句构成：

```
fn main() {
    // 语句
    // 语句
    // 语句
}
```

Rust 有多种语句。最普遍的语句类型有两种：一种是声明绑定变量，另一种是表达式带上英文分号(;)：

```
fn main() {
    // 变量绑定
    let x = 5;

    // 表达式;
    x;
    x + 1;
    15;
}
```

代码块也是表达式，所以它们可以用作赋值中的值。代码块中的最后一个表达式将赋给适当的表达式，例如局部变量。但是，如果代码块的最后一个表达式结尾处有分号，则返回值为 `()`（译注：代码块中的最后一个语句是代码块中**实际执行**的最后一个语句，而不一定是代码块中最后一行的语句）。

```
fn main() {
    let x = 5u32;

    let y = {
        let x_squared = x * x;
        let x_cube = x_squared * x;

        // 将此表达式赋给 `y`
        x_cube + x_squared + x
    };

    let z = {
        // 分号结束了这个表达式，于是将 `()` 赋给 `z`
        2 * x;
    };

    println!("x is {:?}", x);
    println!("y is {:?}", y);
    println!("z is {:?}", z);
}
```

流程控制

任何编程语言都包含的一个必要部分就是改变控制流程：`if`/`else`，`for` 等。让我们谈谈 Rust 语言中的这部分内容。

if/else

`if-else` 分支判断和其他语言类似。不同的是，Rust 语言中的布尔判断条件不必使用小括号包裹，且每个条件后面都跟着一个代码块。`if-else` 条件选择是一个表达式，并且所有分支都必须返回相同的类型。

```
fn main() {
    let n = 5;

    if n < 0 {
        print!("{} is negative", n);
    } else if n > 0 {
        print!("{} is positive", n);
    } else {
        print!("{} is zero", n);
    }

    let big_n =
        if n < 10 && n > -10 {
            println!(", and is a small number, increase ten-fold");

            // 这个表达式返回一个 `i32` 类型。
            10 * n
        } else {
            println!(", and is a big number, half the number");

            // 这个表达式也必须返回一个 `i32` 类型。
            n / 2
            // 试一试 ^ 试着加上一个分号来结束这条表达式。
        };
    //   ^ 不要忘记在这里加上一个分号！所有的 `let` 绑定都需要它。

    println!("{} -> {}", n, big_n);
}
```

loop 循环

Rust 提供了 `loop` 关键字来表示一个无限循环。

可以使用 `break` 语句在任何时候退出一个循环，还可以使用 `continue` 跳过循环体的剩余部分并开始下一轮循环。

```
fn main() {
    let mut count = 0u32;

    println!("Let's count until infinity!");

    // 无限循环
    loop {
        count += 1;

        if count == 3 {
            println!("three");

            // 跳过这次迭代的剩下内容
            continue;
        }

        println!("{}", count);

        if count == 5 {
            println!("OK, that's enough");

            // 退出循环
            break;
        }
    }
}
```

嵌套循环和标签

在处理嵌套循环的时候可以 `break` 或 `continue` 外层循环。在这类情形中，循环必须用一些 `'label` （标签）来注明，并且标签必须传递给 `break`/`continue` 语句。

```
#![allow(unreachable_code)]

fn main() {
    'outer: loop {
        println!("Entered the outer loop");

        'inner: loop {
            println!("Entered the inner loop");

            // 这只是中断内部的循环
            //break;

            // 这会中断外层循环
            break 'outer;
        }

        println!("This point will never be reached");
    }

    println!("Exited the outer loop");
}
```

从 loop 循环中返回

`loop` 有个用途是尝试一个操作直到成功为止。若操作返回一个值，则可能需要将其传递给代码的其余部分：将该值放在 `break` 之后，它就会被 `loop` 表达式返回。

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    assert_eq!(result, 20);
}
```

while 循环

`while` 关键字可以用作当型循环（当条件满足时循环）。

让我们用 `while` 循环写一下臭名昭著的 FizzBuzz（译者补充：LeetCode 上的 FizzBuzz 问题描述） 程序。

```
fn main() {
    // 计数器变量
    let mut n = 1;

    // 当 `n` 小于 101 时循环
    while n < 101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }

        // 计数器值加 1
        n += 1;
    }
}
```

for 循环

for 与区间

`for in` 结构可以遍历一个 `Iterator` （迭代器）。创建迭代器的一个最简单的方法是使用区间标记 `a..b` 。这会生成从 `a` （包含此值） 到 `b` （不含此值）的，步长为 1 的一系列值。

让我们使用 `for` 代替 `while` 来写 FizzBuzz 程序。

```
fn main() {
    // `n` 将在每次迭代中分别取 1, 2, ..., 100
    for n in 1..101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }
    }
}
```

或者，可以使用 `a..=b` 表示两端都包含在内的范围。上面的代码可以写成：

```
fn main() {
    // `n` 将在每次迭代中分别取 1, 2, ..., 100
    for n in 1..=100 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }
    }
}
```

for 与迭代器

`for in` 结构能以几种方式与 `Iterator` 互动。在 迭代器 trait 一节将会谈到，如果没有特别指定，`for` 循环会对给出的集合应用 `into_iter` 函数，把它转换成一个迭代器。这并不是把集合变成迭代器的唯一方法，其他的方法有 `iter` 和 `iter_mut` 函数。

这三个函数会以不同的方式返回集合中的数据。

- `iter` - 在每次迭代中借用集合中的一个元素。这样集合本身不会被改变，循环之后仍可以使用。

```
fn main() {
    let names = vec!["Bob", "Frank", "Ferris"];

    for name in names.iter() {
        match name {
            &"Ferris" => println!("There is a rustacean among us!"),
            _ => println!("Hello {}", name),
        }
    }
}
```

译注：Ferris 是 Rust 的非官方吉祥物。

- `into_iter` - 会消耗集合。在每次迭代中，集合中的数据本身会被提供。一旦集合被消耗了，之后就无法再使用了，因为它已经在循环中被 “移除”（move）了。

```
fn main() {
    let names = vec!["Bob", "Frank", "Ferris"];

    for name in names.into_iter() {
        match name {
            "Ferris" => println!("There is a rustacean among us!"),
            _ => println!("Hello {}", name),
        }
    }
}
```

- `iter_mut` - 可变地（mutably）借用集合中的每个元素，从而允许集合被就地修改。

```
fn main() {
    let mut names = vec!["Bob", "Frank", "Ferris"];

    for name in names.iter_mut() {
        *name = match name {
            &mut "Ferris" => "There is a rustacean among us!",
            _ => "Hello",
        }
    }
    println!("names: {:?}", names);
}
```

在上面这些代码中，注意 `match` 的分支中所写的类型不同，这是不同迭代方式的关键区别。因为类型不同，能够执行的操作当然也不同。

参见：

Iterator

match 匹配

Rust 通过 `match` 关键字来提供模式匹配，和 C 语言的 `switch` 用法类似。第一个匹配分支会被比对，并且所有可能的值都必须被覆盖。

```
fn main() {
    let number = 13;
    // 试一试 ^ 将不同的值赋给 `number`

    println!("Tell me about {}", number);
    match number {
        // 匹配单个值
        1 => println!("One!"),
        // 匹配多个值
        2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
        // 试一试 ^ 将 13 添加到质数列表中
        // 匹配一个闭区间范围
        13..=19 => println!("A teen"),
        // 处理其他情况
        _ => println!("Ain't special"),
        // 试一试 ^ 注释掉这个总括性的分支
    }

    let boolean = true;
    // match 也是一个表达式
    let binary = match boolean {
        // match 分支必须覆盖所有可能的值
        false => 0,
        true => 1,
        // 试一试 ^ 将其中一条分支注释掉
    };

    println!("{} -> {}", boolean, binary);
}
```

解构

`match` 代码块能以多种方式解构物件。

- 解构元组
- 解构枚举
- 解构指针
- 解构结构体

元组

元组可以在 `match` 中解构，如下所示：

```
fn main() {
    let triple = (0, -2, 3);
    // 试一试 ^ 将不同的值赋给 `triple`

    println!("Tell me about {:?}", triple);
    // match 可以解构一个元组
    match triple {
        // 解构出第二个和第三个元素
        (0, y, z) => println!("First is `0`, `y` is {:?}, and `z` is {:?}", y, z),
        (1, ..)  => println!("First is `1` and the rest doesn't matter"),
        // `..` 可用来忽略元组的其余部分
        _      => println!("It doesn't matter what they are"),
        // `_` 表示不将值绑定到变量
    }
}
```

参见：

元组

枚举

和前面相似，解构 `enum` 的方式如下：

```
// 需要 `allow` 来消除警告，因为只使用了枚举类型的一种取值。
#[allow(dead_code)]
enum Color {
    // 这三个取值仅由它们的名字（而非类型）来指定。
    Red,
    Blue,
    Green,
    // 这些则把 `u32` 元组赋予不同的名字，以色彩模型命名。
    RGB(u32, u32, u32),
    HSV(u32, u32, u32),
    HSL(u32, u32, u32),
    CMY(u32, u32, u32),
    CMYK(u32, u32, u32, u32),
}

fn main() {
    let color = Color::RGB(122, 17, 40);
    // 试一试 ^ 将不同的值赋给 `color`

    println!("What color is it?");
    // 可以使用 `match` 来解构 `enum`。
    match color {
        Color::Red   => println!("The color is Red!"),
        Color::Blue  => println!("The color is Blue!"),
        Color::Green => println!("The color is Green!"),
        Color::RGB(r, g, b) =>
            println!("Red: {}, green: {}, and blue: {}!", r, g, b),
        Color::HSV(h, s, v) =>
            println!("Hue: {}, saturation: {}, value: {}!", h, s, v),
        Color::HSL(h, s, l) =>
            println!("Hue: {}, saturation: {}, lightness: {}!", h, s, l),
        Color::CMY(c, m, y) =>
            println!("Cyan: {}, magenta: {}, yellow: {}!", c, m, y),
        Color::CMYK(c, m, y, k) =>
            println!("Cyan: {}, magenta: {}, yellow: {}, key (black): {}!",
                c, m, y, k),
        // 不需要其它分支，因为所有的情形都已覆盖
    }
}
```

参见：

`#[allow(...)]`，色彩模型 和 `enum`

指针和引用

对指针来说，解构（destructure）和解引用（dereference）要区分开，因为这两者的概念是不同的，和 `C` 那样的语言用法不一样。

- 解引用使用 `*`
- 解构使用 `&`、`ref`、和 `ref mut`

```rust
fn main() {
    // 获得一个 `i32` 类型的引用。`&` 表示取引用。
    let reference = &4;

    match reference {
        // 如果用 `&val` 这个模式去匹配 `reference`，就相当于做这样的比较：
        // `&i32`（译注：即 `reference` 的类型）
        // `&val`（译注：即用于匹配的模式）
        // ^ 我们看到，如果去掉匹配的 `&`，`i32` 应当赋给 `val`。
        // 译注：因此可用 `val` 表示被 `reference` 引用的值 4。
        &val => println!("Got a value via destructuring: {:?}", val),
    }

    // 如果不想用 `&`，需要在匹配前解引用。
    match *reference {
        val => println!("Got a value via dereferencing: {:?}", val),
    }

    // 如果一开始就不用引用，会怎样？ `reference` 是一个 `&` 类型，因为赋值语句
    // 的右边已经是一个引用。但下面这个不是引用，因为右边不是。
    let _not_a_reference = 3;

    // Rust 对这种情况提供了 `ref`。它更改了赋值行为，从而可以对具体值创建引用。
    // 下面这行将得到一个引用。
    let ref _is_a_reference = 3;

    // 相应地，定义两个非引用的变量，通过 `ref` 和 `ref mut` 仍可取得其引用。
    let value = 5;
    let mut mut_value = 6;

    // 使用 `ref` 关键字来创建引用。
    // 译注：下面的 r 是 `&i32` 类型，它像 `i32` 一样可以直接打印，因此用法上
    // 似乎看不出什么区别。但读者可以把 `println!` 中的 `r` 改成 `*r`，仍然能
    // 正常运行。前面例子中的 `println!` 里就不能是 `*val`，因为不能对整数解
    // 引用。
    match value {
        ref r => println!("Got a reference to a value: {:?}", r),
    }

    // 类似地使用 `ref mut`。
    match mut_value {
        ref mut m => {
            // 已经获得了 `mut_value` 的引用，先要解引用，才能改变它的值。
            *m += 10;
            println!("We added 10. `mut_value`: {:?}", m);
        },
    }
}
```

参见：

ref 模式

结构体

类似地，解构 `struct` 如下所示：

```
fn main() {
    struct Foo { x: (u32, u32), y: u32 }

    // 解构结构体的成员
    let foo = Foo { x: (1, 2), y: 3 };
    let Foo { x: (a, b), y } = foo;

    println!("a = {}, b = {},  y = {} ", a, b, y);

    // 可以解构结构体并重命名变量，成员顺序并不重要

    let Foo { y: i, x: j } = foo;
    println!("i = {:?}, j = {:?}", i, j);

    // 也可以忽略某些变量
    let Foo { y, .. } = foo;
    println!("y = {}", y);

    // 这将得到一个错误：模式中没有提及 `x` 字段
    // let Foo { y } = foo;
}
```

参见：

结构体, ref 模式

卫语句

可以加上 `match` **卫语句**（guard） 来过滤分支。

```
fn main() {
    let pair = (2, -2);
    // 试一试 ^ 将不同的值赋给 `pair`

    println!("Tell me about {:?}", pair);
    match pair {
        (x, y) if x == y => println!("These are twins"),
        // ^ `if` 条件部分是一个卫语句
        (x, y) if x + y == 0 => println!("Antimatter, kaboom!"),
        (x, _) if x % 2 == 1 => println!("The first one is odd"),
        _ => println!("No correlation..."),
    }
}
```

参见：

元组

绑定

在 `match` 中，若间接地访问一个变量，则不经过重新绑定就无法在分支中再使用它。`match` 提供了 `@` 符号来绑定变量到名称：

```
// `age` 函数，返回一个 `u32` 值。
fn age() -> u32 {
    15
}

fn main() {
    println!("Tell me what type of person you are");

    match age() {
        0             => println!("I haven't celebrated my first birthday yet"),
        // 可以直接匹配（`match`） 1 ..= 12，但那样的话孩子会是几岁？
        // 相反，在 1 ..= 12 分支中绑定匹配值到 `n` 。现在年龄就可以读取了。
        n @ 1  ..= 12 => println!("I'm a child of age {:?}", n),
        n @ 13 ..= 19 => println!("I'm a teen of age {:?}", n),
        // 不符合上面的范围。返回结果。
        n             => println!("I'm an old person of age {:?}", n),
    }
}
```

你也可以使用绑定来“解构” `enum` 变体，例如 `Option`：

```
fn some_number() -> Option<u32> {
    Some(42)
}

fn main() {
    match some_number() {
        // 得到 `Some` 可变类型，如果它的值（绑定到 `n` 上）等于 42，则匹配。
        Some(n @ 42) => println!("The Answer: {}!", n),
        // 匹配任意其他数字。
        Some(n)      => println!("Not interesting... {}", n),
        // 匹配任意其他值（`None` 可变类型）。
        _            => (),
    }
}
```

参见：

函数， 枚举 和 Option

if let

在一些场合下，用 `match` 匹配枚举类型并不优雅。比如：

```
// 将 `optional` 定为 `Option<i32>` 类型
let optional = Some(7);

match optional {
    Some(i) => {
        println!("This is a really long string and `{:?}`", i);
        // ^ 行首需要 2 层缩进。这里从 optional 中解构出 `i`。
        // 译注：正确的缩进是好的，但并不是 “不缩进就不能运行” 这个意思。
    },
    _ => {},
    // ^ 必须有，因为 `match` 需要覆盖全部情况。不觉得这行很多余吗?
};
```

`if let` 在这样的场合要简洁得多，并且允许指明数种失败情形下的选项：

```rust
fn main() {
    // 全部都是 `Option<i32>` 类型
    let number = Some(7);
    let letter: Option<i32> = None;
    let emoticon: Option<i32> = None;

    // `if let` 结构读作：若 `let` 将 `number` 解构成 `Some(i)`，则执行
    // 语句块（`{}`）
    if let Some(i) = number {
        println!("Matched {:?}!", i);
    }

    // 如果要指明失败情形，就使用 else：
    if let Some(i) = letter {
        println!("Matched {:?}!", i);
    } else {
        // 解构失败。切换到失败情形。
        println!("Didn't match a number. Let's go with a letter!");
    };

    // 提供另一种失败情况下的条件。
    let i_like_letters = false;

    if let Some(i) = emoticon {
        println!("Matched {:?}!", i);
    // 解构失败。使用 `else if` 来判断是否满足上面提供的条件。
    } else if i_like_letters {
        println!("Didn't match a number. Let's go with a letter!");
    } else {
        // 条件的值为 false。于是以下是默认的分支：
        println!("I don't like letters. Let's go with an emoticon :)!");
    };
}
```

同样，可以用 `if let` 匹配任何枚举值：

```
// 以这个 enum 类型为例
enum Foo {
    Bar,
    Baz,
    Qux(u32)
}

fn main() {
    // 创建变量
    let a = Foo::Bar;
    let b = Foo::Baz;
    let c = Foo::Qux(100);

    // 变量 a 匹配到了 Foo::Bar
    if let Foo::Bar = a {
        println!("a is foobar");
    }

    // 变量 b 没有匹配到 Foo::Bar，因此什么也不会打印。
    if let Foo::Bar = b {
        println!("b is foobar");
    }

    // 变量 c 匹配到了 Foo::Qux，它带有一个值，就和上面例子中的 Some() 类似。
    if let Foo::Qux(value) = c {
        println!("c is {}", value);
    }
}
```

另一个好处是：`if let` 允许匹配枚举非参数化的变量，即枚举未注明 `#[derive(PartialEq)]`，我们也没有为其实现 `PartialEq`。在这种情况下，通常 `if Foo::Bar==a` 会出错，因为此类枚举的实例不具有可比性。但是，`if let` 是可行的。

你想挑战一下吗？使用 `if let` 修复以下示例：

```
// 该枚举故意未注明 `#[derive(PartialEq)]`，
// 并且也没为其实现 `PartialEq`。这就是为什么下面比较 `Foo::Bar==a` 会失败的原因。
enum Foo {Bar}

fn main() {
    let a = Foo::Bar;

    // 变量匹配 Foo::Bar
    if Foo::Bar == a {
    // ^-- 这就是编译时发现的错误。使用 `if let` 来替换它。
        println!("a is foobar");
    }
}
```

参见：

`枚举`， `Option`，和相关的 RFC

while let

和 `if let` 类似，`while let` 也可以把别扭的 `match` 改写得好看一些。考虑下面这段使 `i` 不断增加的代码：

```
// 将 `optional` 设为 `Option<i32>` 类型
let mut optional = Some(0);

// 重复运行这个测试。
loop {
    match optional {
        // 如果 `optional` 解构成功，就执行下面语句块。
        Some(i) => {
            if i > 9 {
                println!("Greater than 9, quit!");
                optional = None;
            } else {
                println!("`i` is `{:?}`. Try again.", i);
                optional = Some(i + 1);
            }
            // ^ 需要三层缩进！
        },
        // 当解构失败时退出循环：
        _ => { break; }
        // ^ 为什么必须写这样的语句呢？肯定有更优雅的处理方式！
    }
}
```

使用 `while let` 可以使这段代码变得更加优雅：

```
fn main() {
    // 将 `optional` 设为 `Option<i32>` 类型
    let mut optional = Some(0);

    // 这读作：当 `let` 将 `optional` 解构成 `Some(i)` 时，就
    // 执行语句块（`{}`）。否则就 `break`。
    while let Some(i) = optional {
        if i > 9 {
            println!("Greater than 9, quit!");
            optional = None;
        } else {
            println!("`i` is `{:?}`. Try again.", i);
            optional = Some(i + 1);
        }
        // ^ 使用的缩进更少，并且不用显式地处理失败情况。
    }
    // ^ `if let` 有可选的 `else`/`else if` 分句，
    // 而 `while let` 没有。
}
```

参见：

`枚举`， `Option`，和相关的 RFC

函数

函数（function）使用 `fn` 关键字来声明。函数的参数需要标注类型，就和变量一样，如果函数返回一个值，返回类型必须在箭头 `->` 之后指定。

函数最后的表达式将作为返回值。也可以在函数内使用 `return` 语句来提前返一个值，甚至可以在循环或 `if` 内部使用。

让我们使用函数来重写 FizzBuzz 程序吧！

```
// 和 C/C++ 不一样，Rust 的函数定义位置是没有限制的
fn main() {
    // 我们可以在这里使用函数，后面再定义它
    fizzbuzz_to(100);
}

// 一个返回布尔值的函数
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    // 边界情况，提前返回
    if rhs == 0 {
        return false;
    }

    // 这是一个表达式，可以不用 `return` 关键字
    lhs % rhs == 0
}

// 一个“不”返回值的函数。实际上会返回一个单元类型 `()`。
fn fizzbuzz(n: u32) -> () {
    if is_divisible_by(n, 15) {
        println!("fizzbuzz");
    } else if is_divisible_by(n, 3) {
        println!("fizz");
    } else if is_divisible_by(n, 5) {
        println!("buzz");
    } else {
        println!("{}", n);
    }
}

// 当函数返回 `()` 时，函数签名可以省略返回类型
fn fizzbuzz_to(n: u32) {
    for n in 1..=n {
        fizzbuzz(n);
    }
}
```

方法

方法（method）是依附于对象的函数。这些方法通过关键字 `self` 来访问对象中的数据和其他。方法在 `impl` 代码块中定义。

```rust
struct Point {
    x: f64,
    y: f64,
}

// 实现的代码块，`Point` 的所有方法都在这里给出
impl Point {
    // 这是一个静态方法（static method）
    // 静态方法不需要被实例调用
    // 这类方法一般用作构造器（constructor）
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }

    // 另外一个静态方法，需要两个参数：
    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }
}

struct Rectangle {
    p1: Point,
    p2: Point,
}

impl Rectangle {
    // 这是一个实例方法（instance method）
    // `&self` 是 `self: &Self` 的语法糖（sugar），其中 `Self` 是方法调用者的
    // 类型。在这个例子中 `Self` = `Rectangle`
    fn area(&self) -> f64 {
        // `self` 通过点运算符来访问结构体字段
        let Point { x: x1, y: y1 } = self.p1;
        let Point { x: x2, y: y2 } = self.p2;

        // `abs` 是一个 `f64` 类型的方法，返回调用者的绝对值
        ((x1 - x2) * (y1 - y2)).abs()
    }

    fn perimeter(&self) -> f64 {
        let Point { x: x1, y: y1 } = self.p1;
        let Point { x: x2, y: y2 } = self.p2;

        2.0 * ((x1 - x2).abs() + (y1 - y2).abs())
    }

    // 这个方法要求调用者是可变的
    // `&mut self` 为 `self: &mut Self` 的语法糖
    fn translate(&mut self, x: f64, y: f64) {
        self.p1.x += x;
        self.p2.x += x;

        self.p1.y += y;
        self.p2.y += y;
    }
}

// `Pair` 拥有资源：两个堆分配的整型
struct Pair(Box<i32>, Box<i32>);
```

```
impl Pair {
    // 这个方法会“消耗”调用者的资源
    // `self` 为 `self: Self` 的语法糖
    fn destroy(self) {
        // 解构 `self`
        let Pair(first, second) = self;

        println!("Destroying Pair({}, {})", first, second);

        // `first` 和 `second` 离开作用域后释放
    }
}

fn main() {
    let rectangle = Rectangle {
        // 静态方法使用双冒号调用
        p1: Point::origin(),
        p2: Point::new(3.0, 4.0),
    };

    // 实例方法通过点运算符来调用
    // 注意第一个参数 `&self` 是隐式传递的，亦即：
    // `rectangle.perimeter()` === `Rectangle::perimeter(&rectangle)`
    println!("Rectangle perimeter: {}", rectangle.perimeter());
    println!("Rectangle area: {}", rectangle.area());

    let mut square = Rectangle {
        p1: Point::origin(),
        p2: Point::new(1.0, 1.0),
    };

    // 报错！ `rectangle` 是不可变的，但这方法需要一个可变对象
    //rectangle.translate(1.0, 0.0);
    // 试一试 ^ 去掉此行的注释

    // 正常运行！可变对象可以调用可变方法
    square.translate(1.0, 1.0);

    let pair = Pair(Box::new(1), Box::new(2));

    pair.destroy();

    // 报错！前面的 `destroy` 调用“消耗了” `pair`
    //pair.destroy();
    // 试一试 ^ 将此行注释去掉
}
```

闭包

Rust 中的闭包（closure），也叫做 lambda 表达式或者 lambda，是一类能够捕获周围作用域中变量的函数。例如，一个可以捕获 x 变量的闭包如下：

```
|val| val + x
```

它们的语法和能力使它们在临时（on the fly）使用时相当方便。调用一个闭包和调用一个函数完全相同，不过调用闭包时，输入和返回类型两者都**可以**自动推导，而输入变量名**必须**指明。

其他的特点包括：

- 声明时使用 `||` 替代 `()` 将输入参数括起来。
- 函数体定界符（`{}`）对于单个表达式是可选的，其他情况必须加上。
- 有能力捕获外部环境的变量。

```
fn main() {
    // 通过闭包和函数分别实现自增。
    // 译注：下面这行是使用函数的实现
    fn  function            (i: i32) -> i32 { i + 1 }

    // 闭包是匿名的，这里我们将它们绑定到引用。
    // 类型标注和函数的一样，不过类型标注和使用 `{}` 来围住函数体都是可选的。
    // 这些匿名函数（nameless function）被赋值给合适地命名的变量。
    let closure_annotated = |i: i32| -> i32 { i + 1 };
    let closure_inferred  = |i     |          i + 1  ;

    // 译注：将闭包绑定到引用的说法可能不准。
    // 据[语言参考](https://doc.rust-lang.org/beta/reference/types.html#closure-ty
    // 闭包表达式产生的类型就是“闭包类型”，不属于引用类型，而且确实无法对上面两个
    // `closure_xxx` 变量解引用。

    let i = 1;
    // 调用函数和闭包。
    println!("function: {}", function(i));
    println!("closure_annotated: {}", closure_annotated(i));
    println!("closure_inferred: {}", closure_inferred(i));

    // 没有参数的闭包，返回一个 `i32` 类型。
    // 返回类型是自动推导的。
    let one = || 1;
    println!("closure returning one: {}", one());
}
```

捕获

闭包本质上很灵活，能做功能要求的事情，使闭包在没有类型标注的情况下运行。这使得捕获（capture）能够灵活地适应用例，既可移动（move），又可借用（borrow）。闭包可以通过以下方式捕获变量：

- 通过引用：`&T`
- 通过可变引用：`&mut T`
- 通过值：`T`

闭包优先通过引用来捕获变量，并且仅在需要时使用其他方式。

```rust
fn main() {
    use std::mem;

    let color = String::from("green");

    // 这个闭包打印 `color`。它会立即借用（通过引用，`&`）`color` 并将该借用和
    // 闭包本身存储到 `print` 变量中。`color` 会一直保持被借用状态直到
    // `print` 离开作用域。
    //
    // `println!` 只需传引用就能使用，而这个闭包捕获的也是变量的引用，因此无需
    // 进一步处理就可以使用 `println!`。
    let print = || println!("`color`: {}", color);

    // 使用借用来调用闭包 `color`。
    print();

    // `color` 可再次被不可变借用，因为闭包只持有一个指向 `color` 的不可变引用。
    let _reborrow = &color;
    print();

    // 在最后使用 `print` 之后，移动或重新借用都是允许的。
    let _color_moved = color;

    let mut count = 0;
    // 这个闭包使 `count` 值增加。要做到这点，它需要得到 `&mut count` 或者
    // `count` 本身，但 `&mut count` 的要求没那么严格，所以我们采取这种方式。
    // 该闭包立即借用 `count`。
    //
    // `inc` 前面需要加上 `mut`，因为闭包里存储着一个 `&mut` 变量。调用闭包时，
    // 该变量的变化就意味着闭包内部发生了变化。因此闭包需要是可变的。
    let mut inc = || {
        count += 1;
        println!("`count`: {}", count);
    };

    // 使用可变借用调用闭包
    inc();

    // 因为之后调用闭包，所以仍然可变借用 `count`
    // 试图重新借用将导致错误
    // let _reborrow = &count;
    // ^ 试一试：将此行注释去掉。
    inc();

    // 闭包不再借用 `&mut count`，因此可以正确地重新借用
    let _count_reborrowed = &mut count;

    // 不可复制类型（non-copy type）。
    let movable = Box::new(3);

    // `mem::drop` 要求 `T` 类型本身，所以闭包将会捕获变量的值。这种情况下，
    // 可复制类型将会复制给闭包，从而原始值不受影响。不可复制类型必须移动
    // （move）到闭包中，因而 `movable` 变量在这里立即移动到了闭包中。
    let consume = || {
        println!("`movable`: {:?}", movable);
        mem::drop(movable);
    };
```

```
    // `consume` 消耗了该变量，所以该闭包只能调用一次。
    consume();
    //consume();
    // ^ 试一试：将此行注释去掉。
}
```

在竖线 `|` 之前使用 `move` 会强制闭包取得被捕获变量的所有权：

```
fn main() {
    // `Vec` 在语义上是不可复制的。
    let haystack = vec![1, 2, 3];

    let contains = move |needle| haystack.contains(needle);

    println!("{}", contains(&1));
    println!("{}", contains(&4));

    //println!("There're {} elements in vec", haystack.len());
    // ^ 取消上面一行的注释将导致编译时错误，因为借用检查不允许在变量被移动走
    // 之后继续使用它。

    // 在闭包的签名中删除 `move` 会导致闭包以不可变方式借用 `haystack`，因此之后
    // `haystack` 仍然可用，取消上面的注释也不会导致错误。
}
```

参见：

`Box` 和 `std::mem::drop`

作为输入参数

虽然 Rust 无需类型说明就能在大多数时候完成变量捕获，但在编写函数时，这种模糊写法是不允许的。当以闭包作为输入参数时，必须指出闭包的完整类型，它是通过使用以下 `trait` 中的一种来指定的。其受限制程度按以下顺序递减：

- `Fn`：表示捕获方式为通过引用（`&T`）的闭包
- `FnMut`：表示捕获方式为通过可变引用（`&mut T`）的闭包
- `FnOnce`：表示捕获方式为通过值（`T`）的闭包

> 译注：顺序之所以是这样，是因为 `&T` 只是获取了不可变的引用，`&mut T` 则可以改变变量，`T` 则是拿到了变量的所有权而非借用。

对闭包所要捕获的每个变量，编译器都将以限制最少的方式来捕获。

> 译注：这句可能说得不对，事实上是在满足使用需求的前提下尽量以限制最多的方式捕获。

例如用一个类型说明为 `FnOnce` 的闭包作为参数。这说明闭包可能采取 `&T`，`&mut T` 或 `T` 中的一种捕获方式，但编译器最终是根据所捕获变量在闭包里的使用情况决定捕获方式。

这是因为如果能以移动的方式捕获变量，则闭包也有能力使用其他方式借用变量。注意反过来就不再成立：如果参数的类型说明是 `Fn`，那么不允许该闭包通过 `&mut T` 或 `T` 捕获变量。

在下面的例子中，试着分别用一用 `Fn`、`FnMut` 和 `FnOnce`，看看会发生什么：

```rust
// 该函数将闭包作为参数并调用它。
fn apply<F>(f: F) where
    // 闭包没有输入值和返回值。
    F: FnOnce() {
    // ^ 试一试：将 `FnOnce` 换成 `Fn` 或 `FnMut`。

    f();
}

// 输入闭包，返回一个 `i32` 整型的函数。
fn apply_to_3<F>(f: F) -> i32 where
    // 闭包处理一个 `i32` 整型并返回一个 `i32` 整型。
    F: Fn(i32) -> i32 {

    f(3)
}

fn main() {
    use std::mem;

    let greeting = "hello";
    // 不可复制的类型。
    // `to_owned` 从借用的数据创建有所有权的数据。
    let mut farewell = "goodbye".to_owned();

    // 捕获 2 个变量：通过引用捕获 `greeting`，通过值捕获 `farewell`。
    let diary = || {
        // `greeting` 通过引用捕获，故需要闭包是 `Fn`。
        println!("I said {}.", greeting);

        // 下文改变了 `farewell` ，因而要求闭包通过可变引用来捕获它。
        // 现在需要 `FnMut`。
        farewell.push_str("!!!");
        println!("Then I screamed {}.", farewell);
        println!("Now I can sleep. zzzzz");

        // 手动调用 drop 又要求闭包通过值获取 `farewell`。
        // 现在需要 `FnOnce`。
        mem::drop(farewell);
    };

    // 以闭包作为参数，调用函数 `apply`。
    apply(diary);

    // 闭包 `double` 满足 `apply_to_3` 的 trait 约束。
    let double = |x| 2 * x;

    println!("3 doubled: {}", apply_to_3(double));
}
```

参见：

`std::mem::drop`，`Fn`，`FnMut`，和 `FnOnce`

类型匿名

闭包从周围的作用域中捕获变量是简单明了的。这样会有某些后果吗？确实有。观察一下使用闭包作为函数参数，这要求闭包是泛型的，闭包定义的方式决定了这是必要的。

```
// `F` 必须是泛型的。
fn apply<F>(f: F) where
    F: FnOnce() {
    f();
}
```

当闭包被定义，编译器会隐式地创建一个匿名类型的结构体，用以储存闭包捕获的变量，同时为这个未知类型的结构体实现函数功能，通过 `Fn`、`FnMut` 或 `FnOnce` 三种 `trait` 中的一种。

若使用闭包作为函数参数，由于这个结构体的类型未知，任何的用法都要求是泛型的。然而，使用未限定类型的参数 `<T>` 过于不明确，并且是不允许的。事实上，指明为该结构体实现的是 `Fn`、`FnMut`、或 `FnOnce` 中的哪种 `trait`，对于约束该结构体的类型而言就已经足够了。

```
// `F` 必须为一个没有输入参数和返回值的闭包实现 `Fn`，这和对 `print` 的
// 要求恰好一样。
fn apply<F>(f: F) where
    F: Fn() {
    f();
}

fn main() {
    let x = 7;

    // 捕获 `x` 到匿名类型中，并为它实现 `Fn`。
    // 将闭包存储到 `print` 中。
    let print = || println!("{}", x);

    apply(print);
}
```

参见：

详尽分析，`Fn`，`FnMut`，和 `FnOnce`

输入函数

既然闭包可以作为参数，你很可能想知道函数是否也可以呢。确实可以！如果你声明一个接受闭包作为参数的函数，那么任何满足该闭包的 trait 约束的函数都可以作为其参数。

```
// 定义一个函数，可以接受一个由 `Fn` 限定的泛型 `F` 参数并调用它。
fn call_me<F: Fn()>(f: F) {
    f()
}

// 定义一个满足 `Fn` 约束的封装函数（wrapper function）。
fn function() {
    println!("I'm a function!");
}

fn main() {
    // 定义一个满足 `Fn` 约束的闭包。
    let closure = || println!("I'm a closure!");

    call_me(closure);
    call_me(function);
}
```

多说一句，`Fn`、`FnMut` 和 `FnOnce` 这些 `trait` 明确了闭包如何从周围的作用域中捕获变量。

参见：

`Fn`，`FnMut`，和 `FnOnce`

作为输出参数

闭包作为输入参数是可能的，所以返回闭包作为输出参数（output parameter）也应该是可能的。然而返回闭包类型会有问题，因为目前 Rust 只支持返回具体（非泛型）的类型。按照定义，匿名的闭包的类型是未知的，所以只有使用`impl Trait`才能返回一个闭包。

返回闭包的有效特征是：

- `Fn`
- `FnMut`
- `FnOnce`

除此之外，还必须使用`move`关键字，它表明所有的捕获都是通过值进行的。这是必须的，因为在函数退出时，任何通过引用的捕获都被丢弃，在闭包中留下无效的引用。

```
fn create_fn() -> impl Fn() {
    let text = "Fn".to_owned();

    move || println!("This is a: {}", text)
}

fn create_fnmut() -> impl FnMut() {
    let text = "FnMut".to_owned();

    move || println!("This is a: {}", text)
}

fn create_fnonce() -> impl FnOnce() {
    let text = "FnOnce".to_owned();

    move || println!("This is a: {}", text)
}

fn main() {
    let fn_plain = create_fn();
    let mut fn_mut = create_fnmut();
    let fn_once = create_fnonce();

    fn_plain();
    fn_mut();
    fn_once();
}
```

参见：

`Fn`, `FnMut`, 泛型 和 impl Trait.

`std` 中的例子

本小节列出几个标准库中使用闭包的例子。

Iterator::any

`Iterator::any` 是一个函数，若传给它一个迭代器（iterator），当其中任一元素满足谓词（predicate）时它将返回 `true`，否则返回 `false`（译注：谓词是闭包规定的，`true` / `false` 是闭包作用在元素上的返回值）。它的签名如下：

```
pub trait Iterator {
    // 被迭代的类型。
    type Item;

    // `any` 接受 `&mut self` 参数（译注：回想一下，这是 `self: &mut Self` 的简写）
    // 表明函数的调用者可以被借用和修改，但不会被消耗。
    fn any<F>(&mut self, f: F) -> bool where
        // `FnMut` 表示被捕获的变量最多只能被修改，而不能被消耗。
        // `Self::Item` 表明变量是通过值传递给闭包（译注：是迭代器对应的元素的类型）
        F: FnMut(Self::Item) -> bool {}
}
```

```
fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // 对 vec 的 `iter()` 举出 `&i32`。（通过用 `&x` 匹配）把它解构成 `i32`。
    // 译注：注意 `any` 方法会自动地把 `vec.iter()` 举出的迭代器的元素一个个地
    // 传给闭包。因此闭包接收到的参数是 `&i32` 类型的。
    println!("2 in vec1: {}", vec1.iter()     .any(|&x| x == 2));
    // 对 vec 的 `into_iter()` 举出 `i32` 类型。无需解构。
    println!("2 in vec2: {}", vec2.into_iter().any(| x| x == 2));

    let array1 = [1, 2, 3];
    let array2 = [4, 5, 6];

    // 对数组的 `iter()` 举出 `&i32`。
    println!("2 in array1: {}", array1.iter()     .any(|&x| x == 2));
    // 对数组的 `into_iter()` 举出 `i32`。
    println!("2 in array2: {}", array2.into_iter().any(|x| x == 2));
}
```

参见：

`std::iter::Iterator::any`

Iterator::find

`Iterator::find` 是一个函数，在传给它一个迭代器时，将用 `Option` 类型返回第一个满足谓词的元素。它的签名如下：

```
pub trait Iterator {
    // 被迭代的类型。
    type Item;

    // `find` 接受 `&mut self` 参数，表明函数的调用者可以被借用和修改，
    // 但不会被消耗。
    fn find<P>(&mut self, predicate: P) -> Option<Self::Item> where
        // `FnMut` 表示被捕获的变量最多只能被修改，而不能被消耗。
        // `&Self::Item` 指明了被捕获变量的类型（译注：是对迭代器元素的引用类型）
        P: FnMut(&Self::Item) -> bool {}
}
```

```
fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // 对 vec1 的 `iter()` 举出 `&i32` 类型。
    let mut iter = vec1.iter();
    // 对 vec2 的 `into_iter()` 举出 `i32` 类型。
    let mut into_iter = vec2.into_iter();

    // 对迭代器举出的元素的引用是 `&&i32` 类型。解构成 `i32` 类型。
    // 译注：注意 `find` 方法会把迭代器元素的引用传给闭包。迭代器元素自身
    // 是 `&i32` 类型，所以传给闭包的是 `&&i32` 类型。
    println!("Find 2 in vec1: {:?}", iter     .find(|&&x| x == 2));
    // 对迭代器举出的元素的引用是 `&i32` 类型。解构成 `i32` 类型。
    println!("Find 2 in vec2: {:?}", into_iter.find(| &x| x == 2));

    let array1 = [1, 2, 3];
    let array2 = [4, 5, 6];

    // 对数组的 `iter()` 举出 `&i32`。
    println!("Find 2 in array1: {:?}", array1.iter()     .find(|&&x| x == 2));
    // 对数组的 `into_iter()` 通常举出 `&i32``。
    println!("Find 2 in array2: {:?}", array2.into_iter().find(|&x| x == 2));
}
```

参见：

`std::iter::Iterator::find`

高阶函数

Rust 提供了高阶函数（Higher Order Function, HOF），指那些输入一个或多个函数，并且/或者产生一个更有用的函数的函数。HOF 和惰性迭代器（lazy iterator）给 Rust 带来了函数式（functional）编程的风格。

```
fn is_odd(n: u32) -> bool {
    n % 2 == 1
}

fn main() {
    println!("Find the sum of all the squared odd numbers under 1000");
    let upper = 1000;

    // 命令式（imperative）的写法
    // 声明累加器变量
    let mut acc = 0;
    // 迭代：0, 1, 2, ... 到无穷大
    for n in 0.. {
        // 数字的平方
        let n_squared = n * n;

        if n_squared >= upper {
            // 若大于上限则退出循环
            break;
        } else if is_odd(n_squared) {
            // 如果是奇数就计数
            acc += n_squared;
        }
    }
    println!("imperative style: {}", acc);

    // 函数式的写法
    let sum_of_squared_odd_numbers: u32 =
        (0..).map(|n| n * n)             // 所有自然数取平方
             .take_while(|&n| n < upper) // 取小于上限的
             .filter(|&n| is_odd(n))     // 取奇数
             .fold(0, |sum, i| sum + i); // 最后加起来
    println!("functional style: {}", sum_of_squared_odd_numbers);
}
```

Option 和 迭代器 都实现了不少高阶函数。

发散函数

发散函数（diverging function）绝不会返回。 它们使用 `!` 标记，这是一个空类型。

```
fn foo() -> ! {
    panic!("This call never returns.");
}
```

和所有其他类型相反，这个类型无法实例化，因为此类型可能具有的所有可能值的集合为空。 注意，它与 `()` 类型不同，后者只有一个可能的值。

如下面例子，虽然返回值中没有信息，但此函数会照常返回。

```
fn some_fn() {
    ()
}

fn main() {
    let a: () = some_fn();
    println!("This function returns and you can see this line.")
}
```

下面这个函数相反，这个函数永远不会将控制内容返回给调用者。

```
#![feature(never_type)]

fn main() {
    let x: ! = panic!("This call never returns.");
    println!("You will never see this line!");
}
```

虽然这看起来像是一个抽象的概念，但实际上这非常有用且方便。这种类型的主要优点是它可以被转换为任何其他类型，从而可以在需要精确类型的地方使用，例如在 `match` 匹配分支。 这允许我们编写如下代码：

```
fn main() {
    fn sum_odd_numbers(up_to: u32) -> u32 {
        let mut acc = 0;
        for i in 0..up_to {
            // 注意这个 match 表达式的返回值必须为 u32，
            // 因为 “addition” 变量是这个类型。
            let addition: u32 = match i%2 == 1 {
                // “i” 变量的类型为 u32，这毫无问题。
                true => i,
                // 另一方面，“continue” 表达式不返回 u32，但它仍然没有问题，
                // 因为它永远不会返回，因此不会违反匹配表达式的类型要求。
                false => continue,
            };
            acc += addition;
        }
        acc
    }
    println!("Sum of odd numbers up to 9 (excluding): {}", sum_odd_numbers(9));
}
```

这也是永远循环（如 `loop {}`）的函数（如网络服务器）或终止进程的函数（如 `exit()`）的返回类型。

模块

Rust 提供了一套强大的模块（module）系统，可以将代码按层次分成多个逻辑单元（模块），并管理这些模块之间的可见性（公有（public）或私有（private））。

模块是项（item）的集合，项可以是：函数，结构体，trait，`impl` 块，甚至其它模块。

可见性

默认情况下，模块中的项拥有私有的可见性（private visibility），不过可以加上 `pub` 修饰语来重载这一行为。模块中只有公有的（public）项可以从模块外的作用域访问。

```rust
// 一个名为 `my_mod` 的模块
mod my_mod {
    // 模块中的项默认具有私有的可见性
    fn private_function() {
        println!("called `my_mod::private_function()`");
    }

    // 使用 `pub` 修饰语来改变默认可见性。
    pub fn function() {
        println!("called `my_mod::function()`");
    }

    // 在同一模块中，项可以访问其它项，即使它是私有的。
    pub fn indirect_access() {
        print!("called `my_mod::indirect_access()`, that\n> ");
        private_function();
    }

    // 模块也可以嵌套
    pub mod nested {
        pub fn function() {
            println!("called `my_mod::nested::function()`");
        }

        #[allow(dead_code)]
        fn private_function() {
            println!("called `my_mod::nested::private_function()`");
        }

        // 使用 `pub(in path)` 语法定义的函数只在给定的路径中可见。
        // `path` 必须是父模块（parent module）或祖先模块（ancestor module）
        pub(in crate::my_mod) fn public_function_in_my_mod() {
            print!("called `my_mod::nested::public_function_in_my_mod()`, that\n >
            public_function_in_nested()
        }

        // 使用 `pub(self)` 语法定义的函数则只在当前模块中可见。
        pub(self) fn public_function_in_nested() {
            println!("called `my_mod::nested::public_function_in_nested");
        }

        // 使用 `pub(super)` 语法定义的函数只在父模块中可见。
        pub(super) fn public_function_in_super_mod() {
            println!("called my_mod::nested::public_function_in_super_mod");
        }
    }

    pub fn call_public_function_in_my_mod() {
        print!("called `my_mod::call_public_funcion_in_my_mod()`, that\n> ");
        nested::public_function_in_my_mod();
        print!("> ");
        nested::public_function_in_super_mod();
    }

    // `pub(crate)` 使得函数只在当前 crate 中可见
    pub(crate) fn public_function_in_crate() {
        println!("called `my_mod::public_function_in_crate()");
    }
```

```
    // 嵌套模块的可见性遵循相同的规则
    mod private_nested {
        #[allow(dead_code)]
        pub fn function() {
            println!("called `my_mod::private_nested::function()`");
        }
    }
}

fn function() {
    println!("called `function()`");
}

fn main() {
    // 模块机制消除了相同名字的项之间的歧义。
    function();
    my_mod::function();

    // 公有项，包括嵌套模块内的，都可以在父模块外部访问。
    my_mod::indirect_access();
    my_mod::nested::function();
    my_mod::call_public_function_in_my_mod();

    // pub(crate) 项可以在同一个 crate 中的任何地方访问
    my_mod::public_function_in_crate();

    // pub(in path) 项只能在指定的模块中访问
    // 报错！函数 `public_function_in_my_mod` 是私有的
    //my_mod::nested::public_function_in_my_mod();
    // 试一试 ^ 取消该行的注释

    // 模块的私有项不能直接访问，即便它是嵌套在公有模块内部的

    // 报错！`private_function` 是私有的
    //my_mod::private_function();
    // 试一试 ^ 取消此行注释

    // 报错！`private_function` 是私有的
    //my_mod::nested::private_function();
    // 试一试 ^ 取消此行的注释

    // Error! `private_nested` is a private module
    //my_mod::private_nested::function();
    // 试一试 ^ 取消此行的注释
}
```

结构体的可见性

结构体的字段也是一个可见性的层次。字段默认拥有私有的可见性，也可以加上 `pub` 修饰语来重载该行为。只有从结构体被定义的模块之外访问其字段时，这个可见性才会起作用，其意义是隐藏信息（即封装，encapsulation）。

```
mod my {
    // 一个公有的结构体，带有一个公有的字段（类型为泛型 `T`）
    pub struct OpenBox<T> {
        pub contents: T,
    }

    // 一个公有的结构体，带有一个私有的字段（类型为泛型 `T`）
    #[allow(dead_code)]
    pub struct ClosedBox<T> {
        contents: T,
    }

    impl<T> ClosedBox<T> {
        // 一个公有的构造器方法
        pub fn new(contents: T) -> ClosedBox<T> {
            ClosedBox {
                contents: contents,
            }
        }
    }
}

fn main() {
    // 带有公有字段的公有结构体，可以像平常一样构造
    let open_box = my::OpenBox { contents: "public information" };

    // 并且它们的字段可以正常访问到。
    println!("The open box contains: {}", open_box.contents);

    // 带有私有字段的公有结构体不能使用字段名来构造。
    // 报错！`ClosedBox` 含有私有字段。
    //let closed_box = my::ClosedBox { contents: "classified information" };
    // 试一试 ^ 取消此行注释


    // 不过带有私有字段的结构体可以使用公有的构造器来创建。
    let _closed_box = my::ClosedBox::new("classified information");

    // 并且一个结构体中的私有字段不能访问到。
    // 报错！`content` 字段是私有的。
    //println!("The closed box contains: {}", _closed_box.contents);
    // 试一试 ^ 取消此行注释

}
```

参见：

泛型 和 方法

use 声明

`use` 声明可以将一个完整的路径绑定到一个新的名字，从而更容易访问。

```
// 将 `deeply::nested::function` 路径绑定到 `other_function`。
use deeply::nested::function as other_function;

fn function() {
    println!("called `function()`");
}

mod deeply {
    pub mod nested {
        pub fn function() {
            println!("called `deeply::nested::function()`")
        }
    }
}

fn main() {
    // 更容易访问 `deeply::nested::funcion`
    other_function();

    println!("Entering block");
    {
        // 这和 `use deeply::nested::function as function` 等价。
        // 此 `function()` 将遮蔽外部的同名函数。
        use deeply::nested::function;
        function();

        // `use` 绑定拥有局部作用域。在这个例子中，`function()`
        // 的遮蔽只存在这个代码块中。
        println!("Leaving block");
    }

    function();
}
```

`super` 和 `self`

可以在路径中使用 `super` （父级）和 `self` （自身）关键字，从而在访问项时消除歧义，以及防止不必要的路径硬编码。

```rust
fn function() {
    println!("called `function()`");
}

mod cool {
    pub fn function() {
        println!("called `cool::function()`");
    }
}

mod my {
    fn function() {
        println!("called `my::function()`");
    }

    mod cool {
        pub fn function() {
            println!("called `my::cool::function()`");
        }
    }

    pub fn indirect_call() {
        // 让我们从这个作用域中访问所有名为 `function` 的函数!
        print!("called `my::indirect_call()`, that\n> ");

        // `self` 关键字表示当前的模块作用域——在这个例子是 `my`。
        // 调用 `self::function()` 和直接调用 `function()` 都得到相同的结果,
        // 因为他们表示相同的函数。
        self::function();
        function();

        // 我们也可以使用 `self` 来访问 `my` 内部的另一个模块:
        self::cool::function();

        // `super` 关键字表示父作用域（在 `my` 模块外面）。
        super::function();

        // 这将在 *crate* 作用域内绑定 `cool::function` 。
        // 在这个例子中，crate 作用域是最外面的作用域。
        {
            use crate::cool::function as root_function;
            root_function();
        }
    }
}

fn main() {
    my::indirect_call();
}
```

文件分层

模块可以分配到文件/目录的层次结构中。让我们将《可见性》一节中的例子的代码拆分到多个文件中：

```
$ tree .
.
|-- my
|   |-- inaccessible.rs
|   |-- mod.rs
|   `-- nested.rs
`-- split.rs
```

`split.rs` 的内容：

```
// 此声明将会查找名为 `my.rs` 或 `my/mod.rs` 的文件，并将该文件的内容放到
// 此作用域中一个名为 `my` 的模块里面。
mod my;

fn function() {
    println!("called `function()`");
}

fn main() {
    my::function();

    function();

    my::indirect_access();

    my::nested::function();
}
```

`my/mod.rs` 的内容：

```
// 类似地，`mod inaccessible` 和 `mod nested` 将找到 `nested.rs` 和
// `inaccessible.rs` 文件，并在它们放到各自的模块中。
mod inaccessible;
pub mod nested;

pub fn function() {
    println!("called `my::function()`");
}

fn private_function() {
    println!("called `my::private_function()`");
}

pub fn indirect_access() {
    print!("called `my::indirect_access()`, that\n> ");

    private_function();
}
```

`my/nested.rs` 的内容：

```
pub fn function() {
    println!("called `my::nested::function()`");
}

#[allow(dead_code)]
fn private_function() {
    println!("called `my::nested::private_function()`");
}
```

`my/inaccessible.rs` 的内容：

```
#[allow(dead_code)]
pub fn public_function() {
    println!("called `my::inaccessible::public_function()`");
}
```

我们看到代码仍然正常运行，就和前面的一样：

```
$ rustc split.rs && ./split
called `my::function()`
called `function()`
called `my::indirect_access()`, that
> called `my::private_function()`
called `my::nested::function()`
```

crate

crate（中文有 “包，包装箱” 之意）是 Rust 的编译单元。当调用 `rustc some_file.rs` 时，`some_file.rs` 被当作 **crate 文件**。如果 `some_file.rs` 里面含有 `mod` 声明，那么模块文件的内容将在编译之前被插入 crate 文件的相应声明处。换句话说，模块**不会**单独被编译，只有 crate 才会被编译。

crate 可以编译成二进制可执行文件（binary）或库文件（library）。默认情况下，`rustc` 将从 crate 产生二进制可执行文件。这种行为可以通过 `rustc` 的选项 `--crate-type` 重载。

库

让我们创建一个库，然后看看如何把它链接到另一个 crate。

```
pub fn public_function() {
    println!("called rary's `public_function()`");
}

fn private_function() {
    println!("called rary's `private_function()`");
}

pub fn indirect_access() {
    print!("called rary's `indirect_access()`, that\n> ");

    private_function();
}
```

```
$ rustc --crate-type=lib rary.rs
$ ls lib*
library.rlib
```

默认情况下，库会使用 crate 文件的名字，前面加上 “lib” 前缀，但这个默认名称可以使用 `crate_name` 属性 覆盖。

使用库

要将一个 crate 链接到上节新建的库，可以使用 `rustc` 的 `--extern` 选项。然后将所有的物件导入到与库名相同的模块下。此模块的操作通常与任何其他模块相同。

```
// extern crate rary; // 在 Rust 2015 版或更早版本需要这个导入语句

fn main() {
    rary::public_function();

    // 报错！ `private_function` 是私有的
    //rary::private_function();

    rary::indirect_access();
}
```

```
# library.rlib 是已编译好的库的路径，这里假设它在同一目录下：
$ rustc executable.rs --extern rary=library.rlib --edition=2018 && ./executable
called rary's `public_function()`
called rary's `indirect_access()`, that
> called rary's `private_function()`
```

cargo

`cargo` 是官方的 Rust 包管理工具。 它有很多非常有用的功能来提高代码质量和开发人员的开发效率！ 这些功能包括：

- 依赖管理和与 crates.io（官方 Rust 包注册服务）集成
- 方便的单元测试
- 方便的基准测试

本章将介绍一些快速入门的基础知识，不过你可以在 cargo 官方手册中找到详细内容。

依赖

大多数程序都会依赖于某些库。如果你曾经手动管理过库依赖，那么你就知道这会带来的极大的痛苦。幸运的是，Rust 的生态链标配 `cargo` 工具！ `cargo` 可以管理项目的依赖关系。

下面创建一个新的 Rust 项目：

```
# 二进制可执行文件
cargo new foo

# 或者库
cargo new --lib foo
```

对于本章的其余部分，我们选定创建的都是二进制可执行文件而不是库，但所有的概念都是相同的。

完成上述命令后，将看到如下内容：

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

`main.rs` 是新项目的入口源文件——这里没什么新东西。 `Cargo.toml` 是本项目（ `foo` ）的 `cargo` 的配置文件。 浏览 `Cargo.toml` 文件，将看到类似以下的的内容：

```
[package]
name = "foo"
version = "0.1.0"
authors = ["mark"]

[dependencies]
```

`package` 下面的 `name` 字段表明项目的名称。 如果您发布 crate（后面将做更多介绍），那么 `crates.io` 将使用此字段标明的名称。 这也是编译时输出的二进制可执行文件的名称。

`version` 字段是使用语义版本控制（Semantic Versioning）的 crate 版本号。

`authors` 字段表明发布 crate 时的作者列表。

`dependencies` 这部分可以让你为项目添加依赖。

举个例子，假设我们希望程序有一个很棒的命令行界面（command-line interface，CLI）。 你可以在 crates.io（官方的 Rust 包注册服务）上找到很多很棒的 Rust 包。其中一个受欢迎的包是 clap（译注：一个命令行参数的解析器）。在撰写本文时，[clap] 最新发布的版本为 `2.27.1` 。要在程序中添加依赖，我们可以很简单地在 `Cargo.toml` 文件中的 `dependencies` 项后面将以下内

容添加进来：`clap = "2.27.1"`。当然，在 `main.rs` 文件中写上 `extern crate clap`，就和平常一样。 就是这样！你就可以在程序中开始使用 `clap` 了。

`cargo` 还支持其他类型的依赖。 下面是一个简单的示例：

```
[package]
name = "foo"
version = "0.1.0"
authors = ["mark"]

[dependencies]
clap = "2.27.1" # 来自 crates.io
rand = { git = "https://github.com/rust-lang-nursery/rand" } # 来自网上的仓库
bar = { path = "../bar" } # 来自本地文件系统的路径
```

`cargo` 不仅仅是一个包依赖管理器。`Cargo.toml` 的所有可用配置选项都列在 格式规范中。

要构建我们的项目，我们可以在项目目录中的任何位置（包括子目录！）执行 `cargo build`。我们也可以执行 `cargo run` 来构建和运行。请注意，这些命令将处理所有依赖，在需要时下载 crate，并构建所有内容，包括 crate。（请注意，它只重新构建尚未构建的内容，这和 `make` 类似）。

瞧！这里的所有都和 `cargo` 有关！

约定规范

在上一小节中，我们看到了以下目录层次结构：

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

假设我们要在同一个项目中有两个二进制可执行文件。 那要怎样做呢?

很显然，`cargo` 支持这一点。正如我们之前看到的，默认二进制名称是 `main`，但可以通过将文件放在 `bin/` 目录中来添加其他二进制可执行文件：

```
foo
├── Cargo.toml
└── src
    ├── main.rs
    └── bin
        └── my_other_bin.rs
```

为了使得 `cargo` 编译或运行这个二进制可执行文件而不是默认或其他二进制可执行文件，我们只需给 `cargo` 增加一个参数 `--bin my_other_bin`，其中 `my_other_bin` 是我们想要使用的二进制可执行文件的名称。

除了可添加其他二进制可执行文件外，`cargo` 还支持更多功能，如基准测试，测试和示例。

在下一节中，我们将更仔细地学习测试的内容。

测试

我们知道测试是任何软件不可缺少的一部分！Rust 对单元和集成测试提供一流的支持（参见《Rust 程序设计语言》中的关于测试的章节）。

通过上面链接的关于测试章节，我们看到了如何编写单元测试和集成测试。在代码目录组织上，我们可以将单元测试放在需要测试的模块中，并将集成测试放在源码中 `tests/` 目录中：

```
foo
├── Cargo.toml
├── src
│   └── main.rs
└── tests
    ├── my_test.rs
    └── my_other_test.rs
```

`tests` 目录下的每个文件都是一个单独的集成测试。

`cargo` 很自然地提供了一种便捷的方法来运行所有测试！

```
cargo test
```

你将会看到像这样的输出：

```
$ cargo test
   Compiling blah v0.1.0 (file:///nobackup/blah)
    Finished dev [unoptimized + debuginfo] target(s) in 0.89 secs
     Running target/debug/deps/blah-d3b32b97275ec472

running 3 tests
test test_bar ... ok
test test_baz ... ok
test test_foo_bar ... ok
test test_foo ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

你还可以运行如下测试，其中名称匹配一个模式：

```
cargo test test_foo
```

```
$ cargo test test_foo
   Compiling blah v0.1.0 (file:///nobackup/blah)
    Finished dev [unoptimized + debuginfo] target(s) in 0.35 secs
     Running target/debug/deps/blah-d3b32b97275ec472

running 2 tests
test test_foo ... ok
test test_foo_bar ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out
```

需要注意的一点是：`cargo` 可能同时进行多项测试，因此请确保它们不会相互竞争。例如，如果它们都输出到文件，则应该将它们写入不同的文件。

构建脚本

有时使用 `cargo` 正常构建还是不够的。也许你的 crate 在 cargo 成功编译之前需要一些先决条件，比如代码生成或者需要编译的一些本地代码。为了解决这个问题，我们构建了 cargo 可以运行的脚本。

要向包中添加构建脚本，可以在 `Cargo.toml` 中指定它，如下所示：

```
[package]
...
build = "build.rs"
```

跟默认情况不同，这里 cargo 将在项目目录中优先查找 `build.rs` 文件。（本句采用意译，英文原文为：Otherwise Cargo will look for a `build.rs` file in the project directory by default.）

怎么使用构建脚本

构建脚本只是另一个 Rust 文件，此文件将在编译包中的任何其他内容之前，优先进行编译和调用。 因此，此文件可实现满足 crate 的先决条件。

cargo 通过此处指定的可以使用的环境变量为脚本提供输入。（英文原文：Cargo provides the script with inputs via environment variables specified here that can be used.）

此脚本通过 stdout （标准输出）提供输出。打印的所有行都写入到 `target/debug/build/<pkg>/output`。另外，以 `cargo:` 为前缀的行将由 cargo 直接解析，因此可用于定义包编译的参数。

有关进一步的说明和示例，请阅读 cargo 规定说明文档。

属性

属性是应用于某些模块、crate 或项的元数据（metadata）。这元数据可以用来：

- 条件编译代码
- 设置 crate 名称、版本和类型（二进制文件或库）
- 禁用 lint （警告）
- 启用编译器的特性（宏、全局导入（glob import）等）
- 链接到一个非 Rust 语言的库
- 标记函数作为单元测试
- 标记函数作为基准测试的某个部分

当属性作用于整个 crate 时，它们的语法为 `#![crate_attribute]`，当它们用于模块或项时，语法为 `#[item_attribute]` （注意少了感叹号 `!`）。

属性可以接受参数，有不同的语法形式：

- `#[attribute = "value"]`
- `#[attribute(key = "value")]`
- `#[attribute(value)]`

属性可以多个值，它们可以分开到多行中：

```
#[attribute(value, value2)]

#[attribute(value, value2, value3,
            value4, value5)]
```

死代码 `dead_code`

编译器提供了 `dead_code`（死代码，无效代码）*lint*，这会对未使用的函数产生警告。可以用一个**属性**来禁用这个 lint。

```
fn used_function() {}

// `#[allow(dead_code)]` 属性可以禁用 `dead_code` lint
#[allow(dead_code)]
fn unused_function() {}

fn noisy_unused_function() {}
// 改正 ^ 增加一个属性来消除警告

fn main() {
    used_function();
}
```

注意在实际程序中，需要将死代码清除掉。由于本书的例子是交互性的，因而其中需要允许一些死代码的出现。

crate

`crate_type` 属性可以告知编译器 crate 是一个二进制的可执行文件还是一个库（甚至是哪种类型的库），`crate_name` 属性可以设定 crate 的名称。

不过，一定要注意在使用 cargo 时，这两种类型时都**没有**作用。由于大多数 Rust 工程都使用 cargo，这意味着 `crate_type` 和 `crate_name` 的作用事实上很有限。

```
// 这个 crate 是一个库文件
#![crate_type = "lib"]
// 库的名称为 “rary”
#![crate_name = "rary"]

pub fn public_function() {
    println!("called rary's `public_function()`");
}

fn private_function() {
    println!("called rary's `private_function()`");
}

pub fn indirect_access() {
    print!("called rary's `indirect_access()`, that\n> ");

    private_function();
}
```

当用到 `crate_type` 属性时，就不再需要给 `rustc` 命令加上 `--crate-type` 标记。

```
$ rustc lib.rs
$ ls lib*
library.rlib
```

cfg

条件编译可能通过两种不同的操作符实现：

- `cfg` 属性：在属性位置中使用 `#[cfg(...)]`
- `cfg!` 宏：在布尔表达式中使用 `cfg!(...)`

两种形式使用的参数语法都相同。

```
// 这个函数仅当目标系统是 Linux 的时候才会编译
#[cfg(target_os = "linux")]
fn are_you_on_linux() {
    println!("You are running linux!")
}

// 而这个函数仅当目标系统 **不是** Linux 时才会编译
#[cfg(not(target_os = "linux"))]
fn are_you_on_linux() {
    println!("You are *not* running linux!")
}

fn main() {
    are_you_on_linux();

    println!("Are you sure?");
    if cfg!(target_os = "linux") {
        println!("Yes. It's definitely linux!");
    } else {
        println!("Yes. It's definitely *not* linux!");
    }
}
```

参见：

引用, `cfg!`, 和 宏.

自定义条件

有部分条件如 `target_os` 是由 `rustc` 隐式地提供的，但是自定义条件必须使用 `--cfg` 标记来传给 `rustc`。

```
#[cfg(some_condition)]
fn conditional_function() {
    println!("condition met!")
}

fn main() {
    conditional_function();
}
```

试试不使用自定义的 `cfg` 标记会发生什么：

```
$ rustc custom.rs && ./custom
No such file or directory (os error 2)
```

使用自定义的 `cfg` 标记：

```
$ rustc --cfg some_condition custom.rs && ./custom
condition met!
```

泛型

泛型（generic）是关于泛化类型和函数功能，以扩大其适用范围的话题。泛型极大地减少了代码的重复，但它自身的语法很要求细心。也就是说，采用泛型意味着仔细地指定泛型类型具体化时，什么样的具体类型是合法的。泛型最简单和常用的用法是用于类型参数。

> 译注：定义泛型类型或泛型函数之类的东西时，我们会用 `<A>` 或者 `<T>` 这类标记 作为类型的代号，就像函数的形参一样。在使用时，为把 `<A>`、`<T>` 具体化，我们 会把类型说明像实参一样使用，像是 `<i32>` 这样。这两种把（泛型的或具体的）类型 当作参数的用法就是**类型参数**。

泛型的类型参数是使用尖括号和大驼峰命名的名称：`<Aaa, Bbb, ...>` 来指定的。泛型类型参数一般用 `<T>` 来表示。在 Rust 中，“泛型的” 除了表示类型，还表示可以接受一个或多个泛型类型参数 `<T>` 的任何内容。任何用泛型类型参数表示的类型都是泛型，其他的类型都是具体（非泛型）类型。

例如定义一个名为 `foo` 的 **泛型函数**，它可接受类型为 `T` 的任何参数 `arg`：

```
fn foo<T>(arg: T) { ... }
```

因为我们使用了泛型类型参数 `<T>`，所以这里的 `(arg: T)` 中的 `T` 就是泛型类型。即使 `T` 在之前被定义为 `struct`，这里的 `T` 仍然代表泛型。

下面例子展示了泛型语法的使用：

```rust
// 一个具体类型 `A`。
struct A;

// 在定义类型 `Single` 时，第一次使用类型 `A` 之前没有写 `<A>`。
// 因此，`Single` 是个具体类型，`A` 取上面的定义。
struct Single(A);
//            ^ 这里是 `Single` 对类型 `A` 的第一次使用。

// 此处 `<T>` 在第一次使用 `T` 前出现，所以 `SingleGen` 是一个泛型类型。
// 因为 `T` 是泛型的，所以它可以是任何类型，包括在上面定义的具体类型 `A`。
struct SingleGen<T>(T);

fn main() {
    // `Single` 是具体类型，并且显式地使用类型 `A`。
    let _s = Single(A);

    // 创建一个 `SingleGen<char>` 类型的变量 `_char`，并令其值为 `SingleGen('a')`
    // 这里的 `SingleGen` 的类型参数是显式指定的。
    let _char: SingleGen<char> = SingleGen('a');

    // `SingleGen` 的类型参数也可以隐式地指定。
    let _t    = SingleGen(A); // 使用在上面定义的 `A`。
    let _i32  = SingleGen(6); // 使用 `i32` 类型。
    let _char = SingleGen('a'); // 使用 `char`。
}
```

参见：

`struct`

函数

同样的规则也适用于函数：在使用类型 `T` 前给出 `<T>`，那么 `T` 就变成了泛型。

调用泛型函数有时需要显式地指明类型参量。这可能是因为调用了返回类型是泛型的函数，或者编译器没有足够的信息来推断类型参数。

调用函数时，使用显式指定的类型参数会像是这样：`fun::<A, B, ...>()`。

```
struct A;          // 具体类型 `A`。
struct S(A);       // 具体类型 `S`。
struct SGen<T>(T); // 泛型类型 `SGen`。

// 下面全部函数都得到了变量的所有权，并立即使之离开作用域，将变量释放。

// 定义一个函数 `reg_fn`，接受一个 `S` 类型的参数 `_s`。
// 因为没有 `<T>` 这样的泛型类型参数，所以这不是泛型函数。
fn reg_fn(_s: S) {}

// 定义一个函数 `gen_spec_t`，接受一个 `SGen<A>` 类型的参数 `_s`。
// `SGen<>` 显式地接受了类型参数 `A`，且在 `gen_spec_t` 中，`A` 没有被用作
// 泛型类型参数，所以函数不是泛型的。
fn gen_spec_t(_s: SGen<A>) {}

// 定义一个函数 `gen_spec_i32`，接受一个 `SGen<i32>` 类型的参数 `_s`。
// `SGen<>` 显式地接受了类型参量 `i32`，而 `i32` 是一个具体类型。
// 由于 `i32` 不是一个泛型类型，所以这个函数也不是泛型的。
fn gen_spec_i32(_s: SGen<i32>) {}

// 定义一个函数 `generic`，接受一个 `SGen<T>` 类型的参数 `_s`。
// 因为 `SGen<T>` 之前有 `<T>`，所以这个函数是关于 `T` 的泛型函数。
fn generic<T>(_s: SGen<T>) {}

fn main() {
    // 使用非泛型函数
    reg_fn(S(A));          // 具体类型。
    gen_spec_t(SGen(A));   // 隐式地指定类型参数 `A`。
    gen_spec_i32(SGen(6)); // 隐式地指定类型参数 `i32`。

    // 为 `generic()` 显式地指定类型参数 `char`。
    generic::<char>(SGen('a'));

    // 为 `generic()` 隐式地指定类型参数 `char`。
    generic(SGen('c'));
}
```

参见：

函数 和 `structs`

实现

和函数类似，`impl` 块要想实现泛型，也需要很仔细。

```
struct S; // 具体类型 `S`
struct GenericVal<T>(T,); // 泛型类型 `GenericVal`

// GenericVal 的 `impl`，此处我们显式地指定了类型参数：
impl GenericVal<f32> {} // 指定 `f32` 类型
impl GenericVal<S> {} // 指定为上面定义的 `S`

// `<T>` 必须在类型之前写出来，以使类型 `T` 代表泛型。
impl <T> GenericVal<T> {}
```

```
struct Val {
    val: f64
}

struct GenVal<T>{
    gen_val: T
}

// Val 的 `impl`
impl Val {
    fn value(&self) -> &f64 { &self.val }
}

// GenVal 的 `impl`，指定 `T` 是泛型类型
impl <T> GenVal<T> {
    fn value(&self) -> &T { &self.gen_val }
}

fn main() {
    let x = Val { val: 3.0 };
    let y = GenVal { gen_val: 3i32 };

    println!("{}, {}", x.value(), y.value());
}
```

参见：

返回引用的函数，`impl`，和 `struct`

trait

当然 `trait` 也可以是泛型的。我们在这里定义了一个 `trait`，它把 `Drop` `trait` 作为泛型方法实现了，可以 `drop`（丢弃）调用者本身和一个输入参数。

```
// 不可复制的类型。
struct Empty;
struct Null;

// `T` 的泛型 trait。
trait DoubleDrop<T> {
    // 定义一个调用者的方法，接受一个额外的参数 `T`，但不对它做任何事。
    fn double_drop(self, _: T);
}

// 对泛型的调用者类型 `U` 和任何泛型类型 `T` 实现 `DoubleDrop<T>` 。
impl<T, U> DoubleDrop<T> for U {
    // 此方法获得两个传入参数的所有权，并释放它们。
    fn double_drop(self, _: T) {}
}

fn main() {
    let empty = Empty;
    let null  = Null;

    // 释放 `empty` 和 `null`。
    empty.double_drop(null);

    //empty;
    //null;
    // ^ 试一试：去掉这两行的注释。
}
```

参见：

`Drop`，`struct`，和 `trait`

约束

在使用泛型时，类型参数常常必须使用 trait 作为**约束**（bound）来明确规定类型应实现哪些功能。例如下面的例子用到了 `Display` trait 来打印，所以它用 `Display` 来约束 `T`，也就是说 `T` **必须**实现 `Display`。

```
// 定义一个函数 `printer`，接受一个类型为泛型 `T` 的参数，
// 其中 `T` 必须实现 `Display` trait。
fn printer<T: Display>(t: T) {
    println!("{}", t);
}
```

约束把泛型类型限制为符合约束的类型。请看：

```
struct S<T: Display>(T);

// 报错！`Vec<T>` 未实现 `Display`。此次泛型具体化失败。
let s = S(vec![1]);
```

约束的另一个作用是泛型的实例可以访问作为约束的 trait 的方法。例如：

```rust
// 这个 trait 用来实现打印标记：`{:?}`。
use std::fmt::Debug;

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Rectangle {
    fn area(&self) -> f64 { self.length * self.height }
}

#[derive(Debug)]
struct Rectangle { length: f64, height: f64 }
#[allow(dead_code)]
struct Triangle  { length: f64, height: f64 }

// 泛型 `T` 必须实现 `Debug` 。只要满足这点，无论什么类型
// 都可以让下面函数正常工作。
fn print_debug<T: Debug>(t: &T) {
    println!("{:?}", t);
}

// `T` 必须实现 `HasArea`。任意符合该约束的泛型的实例
// 都可访问 `HasArea` 的 `area` 函数
fn area<T: HasArea>(t: &T) -> f64 { t.area() }

fn main() {
    let rectangle = Rectangle { length: 3.0, height: 4.0 };
    let _triangle = Triangle  { length: 3.0, height: 4.0 };

    print_debug(&rectangle);
    println!("Area: {}", area(&rectangle));

    //print_debug(&_triangle);
    //println!("Area: {}", area(&_triangle));
    // ^ 试一试：取消上述语句的注释。
    // | 报错：未实现 `Debug` 或 `HasArea`。
}
```

多说一句，某些情况下也可使用 `where` 分句来形成约束，这拥有更好的表现力。

参见：

`std::fmt`，`struct`，和 `trait`

测试实例：空约束

约束的工作机制会产生这样的效果：即使一个 `trait` 不包含任何功能，你仍然可以用它作为约束。标准库中的 `Eq` 和 `Ord` 就是这样的 `trait`。

```
struct Cardinal;
struct BlueJay;
struct Turkey;

trait Red {}
trait Blue {}

impl Red for Cardinal {}
impl Blue for BlueJay {}

// 这些函数只对实现了相应的 trait 的类型有效。
// 事实上这些 trait 内部是空的，但这没有关系。
fn red<T: Red>(_: &T)   -> &'static str { "red" }
fn blue<T: Blue>(_: &T) -> &'static str { "blue" }

fn main() {
    let cardinal = Cardinal;
    let blue_jay = BlueJay;
    let _turkey   = Turkey;

    // 由于约束，`red()` 不能作用于 blue_jay （蓝松鸟），反过来也一样。
    println!("A cardinal is {}", red(&cardinal));
    println!("A blue jay is {}", blue(&blue_jay));
    //println!("A turkey is {}", red(&_turkey));
    // ^ 试一试：去掉此行注释。
}
```

参见：

`std::cmp::Eq`，`std::cmp::Ord`，和 `trait`

多重约束

多重约束（multiple bounds）可以用 `+` 连接。和平常一样，类型之间使用 `,` 隔开。

```
use std::fmt::{Debug, Display};

fn compare_prints<T: Debug + Display>(t: &T) {
    println!("Debug: `{:?}`", t);
    println!("Display: `{}`", t);
}

fn compare_types<T: Debug, U: Debug>(t: &T, u: &U) {
    println!("t: `{:?}`", t);
    println!("u: `{:?}`", u);
}

fn main() {
    let string = "words";
    let array = [1, 2, 3];
    let vec = vec![1, 2, 3];

    compare_prints(&string);
    //compare_prints(&array);
    // 试一试 ^ 将此行注释去掉。

    compare_types(&array, &vec);
}
```

参见：

`std::fmt` 和 `trait`

where 分句

约束也可以使用 `where` 分句来表达，它放在 `{` 的前面，而不需写在类型第一次出现之前。另外 `where` 从句可以用于任意类型的限定，而不局限于类型参数本身。

`where` 在下面一些情况下很有用：

- 当分别指定泛型的类型和约束会更清晰时：

```
impl <A: TraitB + TraitC, D: TraitE + TraitF> MyTrait<A, D> for YourType {}

// 使用 `where` 从句来表达约束
impl <A, D> MyTrait<A, D> for YourType where
    A: TraitB + TraitC,
    D: TraitE + TraitF {}
```

- 当使用 `where` 从句比正常语法更有表现力时。本例中的 `impl` 如果不用 `where` 从句，就无法直接表达。

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

trait PrintInOption {
    fn print_in_option(self);
}

// 这里需要一个 `where` 从句，否则就要表达成 `T: Debug`（这样意思就变了），
// 或者改用另一种间接的方法。
impl<T> PrintInOption for T where
    Option<T>: Debug {
    // 我们要将 `Option<T>: Debug` 作为约束，因为那是要打印的内容。
    // 否则我们会给出错误的约束。
    fn print_in_option(self) {
        println!("{:?}", Some(self));
    }
}

fn main() {
    let vec = vec![1, 2, 3];

    vec.print_in_option();
}
```

参见：

相关的 RFC、`struct` 和 `trait`

new type 惯用法

`newtype` 惯用法（译注：即为不同种类的数据分别定义新的类型）能保证在编译时，提供给程序的都是正确的类型。

比如说，实现一个 “年龄认证” 函数，它要求输入**必须**是 `Years` 类型。

```
struct Years(i64);

struct Days(i64);

impl Years {
    pub fn to_days(&self) -> Days {
        Days(self.0 * 365)
    }
}


impl Days {
    /// 舍去不满一年的部分
    pub fn to_years(&self) -> Years {
        Years(self.0 / 365)
    }
}

fn old_enough(age: &Years) -> bool {
    age.0 >= 18
}

fn main() {
    let age = Years(5);
    let age_days = age.to_days();
    println!("Old enough {}", old_enough(&age));
    println!("Old enough {}", old_enough(&age_days.to_years()));
    // println!("Old enough {}", old_enough(&age_days));
}
```

取消最后一行的注释，就可以发现提供给 `old_enough` 的必须是 `Years` 类型。

See also:

`structs`

关联项

“关联项”（associated item）指与多种类型的项有关的一组规则。它是 `trait` 泛型的扩展，允许在 `trait` 内部定义新的项。

一个这样的项就叫做一个关联类型。当 `trait` 对于实现了它的容器类型是泛型的，关联项就提供了简单的使用方法。

> 译注：“关联项”这个说法实际上只在 RFC 里出现了，官方的《The Rust Programming Language》第一版和第二版都只有“关联类型”的说法。如果觉得这里的说法很别扭的话 不要理会就是了。TRPL 对关联类型的定义是：“一种将类型占位符与 trait 联系起来的 做法，这样 trait 中的方法签名中就可以使用这些占位符类型。trait 的实现会指定在 该实现中那些占位符对应什么具体类型。”等看完这一节再回头看这个定义就很明白了。

参见：

RFC

存在问题

`trait` 如果对实现了它的容器类型是泛型的，则须遵守类型规范要求——`trait` 的使用者**必须**指出 `trait` 的全部泛型类型。

在下面例子中，`Contains` `trait` 允许使用泛型类型 `A` 和 `B`。然后我们为 `Container` 类型实现了这个 trait，将 `A` 和 `B` 指定为 `i32`，这样就可以对它们使用 `difference()` 函数。

因为 `Contains` 是泛型的，我们必须在 `fn difference()` 中显式地指出**所有的**泛型类型。但实际上，我们想要表达，`A` 和 `B` 究竟是什么类型是由输入 `C` 决定的。在下一节会看到，关联类型恰好提供了这样的功能。

```rust
struct Container(i32, i32);

// 这个 trait 检查给定的 2 个项是否储存于容器中
// 并且能够获得容器的第一个或最后一个值。
trait Contains<A, B> {
    fn contains(&self, _: &A, _: &B) -> bool; // 显式地要求 `A` 和 `B`
    fn first(&self) -> i32; // 未显式地要求 `A` 或 `B`
    fn last(&self) -> i32;  // 未显式地要求 `A` 或 `B`
}

impl Contains<i32, i32> for Container {
    // 如果存储的数字和给定的相等则为真。
    fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
        (&self.0 == number_1) && (&self.1 == number_2)
    }

    // 得到第一个数字。
    fn first(&self) -> i32 { self.0 }

    // 得到最后一个数字。
    fn last(&self) -> i32 { self.1 }
}

// 容器 `C` 就包含了 `A` 和 `B` 类型。鉴于此，必须指出 `A` 和 `B` 显得很麻烦。
fn difference<A, B, C>(container: &C) -> i32 where
    C: Contains<A, B> {
    container.last() - container.first()
}

fn main() {
    let number_1 = 3;
    let number_2 = 10;

    let container = Container(number_1, number_2);

    println!("Does container contain {} and {}: {}",
        &number_1, &number_2,
        container.contains(&number_1, &number_2));
    println!("First number: {}", container.first());
    println!("Last number: {}", container.last());

    println!("The difference is: {}", difference(&container));
}
```

参见：

`struct`，和 `trait`

关联类型

通过把容器内部的类型放到 `trait` 中作为**输出类型**，使用 “关联类型” 增加了代码的可读性。这样的 `trait` 的定义语法如下：

```
// `A` 和 `B` 在 trait 里面通过 `type` 关键字来定义。
// （注意：此处的 `type` 不同于为类型取别名时的 `type`）。
trait Contains {
    type A;
    type B;

    // 这种语法能够泛型地表示这些新类型。
    fn contains(&self, _: &Self::A, _: &Self::B) -> bool;
}
```

注意使用了 `Contains` `trait` 的函数就不需要写出 `A` 或 `B` 了：

```
// 不使用关联类型
fn difference<A, B, C>(container: &C) -> i32 where
    C: Contains<A, B> { ... }

// 使用关联类型
fn difference<C: Contains>(container: &C) -> i32 { ... }
```

让我们使用关联类型来重写上一小节的例子：

```rust
struct Container(i32, i32);

// 这个 trait 检查给定的 2 个项是否储存于容器中
// 并且能够获得容器的第一个或最后一个值。
trait Contains {
    // 在这里定义可以被方法使用的泛型类型。
    type A;
    type B;

    fn contains(&self, _: &Self::A, _: &Self::B) -> bool;
    fn first(&self) -> i32;
    fn last(&self) -> i32;
}

impl Contains for Container {
    // 指出 `A` 和 `B` 是什么类型。如果 `input`（输入）类型
    // 为 `Container(i32, i32)`，那么 `output`（输出）类型
    // 会被确定为 `i32` 和 `i32`。
    type A = i32;
    type B = i32;

    // `&Self::A` 和 `&Self::B` 在这里也是合法的类型。
    fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
        (&self.0 == number_1) && (&self.1 == number_2)
    }

    // 得到第一个数字。
    fn first(&self) -> i32 { self.0 }

    // 得到最后一个数字。
    fn last(&self) -> i32 { self.1 }
}

fn difference<C: Contains>(container: &C) -> i32 {
    container.last() - container.first()
}

fn main() {
    let number_1 = 3;
    let number_2 = 10;

    let container = Container(number_1, number_2);

    println!("Does container contain {} and {}: {}",
        &number_1, &number_2,
        container.contains(&number_1, &number_2));
    println!("First number: {}", container.first());
    println!("Last number: {}", container.last());

    println!("The difference is: {}", difference(&container));
}
```

虚类型参数

虚类型（phantom type）参数是一种在运行时不出现，而在（且仅在）编译时进行静态检查的类型参数。

可以用额外的泛型类型参数指定数据类型，该类型可以充当标记，也可以供编译时类型检查使用。这些额外的参数没有存储值，也没有运行时行为。

在下面例子中，我们使用 std::marker::PhantomData 作为虚类型参数的类型，创建包含不同数据类型的元组。

```
use std::marker::PhantomData;

// 这个虚元组结构体对 `A` 是泛型的，并且带有隐藏参数 `B`。
#[derive(PartialEq)] // 允许这种类型进行相等测试（equality test）。
struct PhantomTuple<A, B>(A,PhantomData<B>);

// 这个虚类型结构体对 `A` 是泛型的，并且带有隐藏参数 `B`。
#[derive(PartialEq)] // 允许这种类型进行相等测试。
struct PhantomStruct<A, B> { first: A, phantom: PhantomData<B> }

// 注意：对于泛型 `A` 会分配存储空间，但 `B` 不会。
//       因此，`B` 不能参与运算。

fn main() {
    // 这里的 `f32` 和 `f64` 是隐藏参数。
    // 被指定为 `<char, f32>` 的 `PhantomTuple` 类型。
    let _tuple1: PhantomTuple<char, f32> = PhantomTuple('Q', PhantomData);
    // 被指定为 `<char, f64>` `PhantomTuple` 类型。
    let _tuple2: PhantomTuple<char, f64> = PhantomTuple('Q', PhantomData);

    // 被指定为 `<char, f32>` 的类型。
    let _struct1: PhantomStruct<char, f32> = PhantomStruct {
        first: 'Q',
        phantom: PhantomData,
    };
    // 被指定为 `<char, f64>` 的类型。
    let _struct2: PhantomStruct<char, f64> = PhantomStruct {
        first: 'Q',
        phantom: PhantomData,
    };

    // 编译期错误！类型不匹配，所以这些值不能够比较：
    //println!("_tuple1 == _tuple2 yields: {}",
    //          _tuple1 == _tuple2);

    // 编译期错误！类型不匹配，所以这些值不能够比较：
    //println!("_struct1 == _struct2 yields: {}",
    //          _struct1 == _struct2);
}
```

参见：

Derive, 结构体, 和 元组结构体

测试实例：单位说明

通过实现一个带虚类型参数的 `Add` trait 可以实现单位检查。这种 `Add` trait 的代码如下：

```
// 这个 `trait` 会要求 `Self + RHS = Output`。`<RHS = Self>` 表示 RHS 的默认值
// 为 Self 类型，也就是如果没有在实现中另行指定，RHS 就取 Self 类型。
pub trait Add<RHS = Self> {
    type Output;

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

// `Output` 必须是 `T<U>` 类型，所以是 `T<U> + T<U> = T<U>`。
impl<U> Add for T<U> {
    type Output = T<U>;
    ...
}
```

完整实现：

```rust
use std::ops::Add;
use std::marker::PhantomData;

/// 创建空枚举类型来表示单位。
#[derive(Debug, Clone, Copy)]
enum Inch {}
#[derive(Debug, Clone, Copy)]
enum Mm {}

/// `Length` 是一个带有虚类型参数 `Unit` 的类型，
/// 而且对于表示长度的类型（即 `f64`）而言，`Length` 不是泛型的。
///
/// `f64` 已经实现了 `Clone` 和 `Copy` trait.
#[derive(Debug, Clone, Copy)]
struct Length<Unit>(f64, PhantomData<Unit>);

/// `Add` trait 定义了 `+` 运算符的行为。
impl<Unit> Add for Length<Unit> {
    type Output = Length<Unit>;

    // add() 返回一个含有和的新的 `Length` 结构体。
    fn add(self, rhs: Length<Unit>) -> Length<Unit> {
        // `+` 调用了针对 `f64` 类型的 `Add` 实现。
        Length(self.0 + rhs.0, PhantomData)
    }
}

fn main() {
    // 指定 `one_foot` 拥有虚类型参数 `Inch`。
    let one_foot:  Length<Inch> = Length(12.0, PhantomData);
    // `one_meter` 拥有虚类型参数 `Mm`。
    let one_meter: Length<Mm>   = Length(1000.0, PhantomData);

    // `+` 调用了我们对 `Length<Unit>` 实现的 `add()` 方法。
    //
    // 由于 `Length` 了实现了 `Copy`，`add()` 不会消耗 `one_foot`
    // 和 `one_meter`，而是复制它们作为 `self` 和 `rhs`。
    let two_feet = one_foot + one_foot;
    let two_meters = one_meter + one_meter;

    // 加法正常执行。
    println!("one foot + one_foot = {:?} in", two_feet.0);
    println!("one meter + one_meter = {:?} mm", two_meters.0);

    // 无意义的运算当然会失败：
    // 编译期错误：类型不匹配。
    //let one_feter = one_foot + one_meter;
}
```

参见：

Borrowing (`&`), Bounds (`X: Y`), enum, impl & self, Overloading, ref, Traits (`X for Y`), 和 TupleStructs.

作用域规则

作用域在所有权（ownership）、借用（borrow）和生命周期（lifetime）中起着重要作用。也就是说，作用域告诉编译器什么时候借用是合法的、什么时候资源可以释放、以及变量何时被创建或销毁。

RAII

Rust 的变量不只是在栈中保存数据：它们也**占有**资源，比如 `Box<T>` 占有堆（heap）中的内存。Rust 强制实行 RAII（Resource Acquisition Is Initialization，资源获取即初始化），所以任何对象在离开作用域时，它的析构函数（destructor）就被调用，然后它占有的资源就被释放。

这种行为避免了**资源泄漏**（resource leak），所以你再也不用手动释放内存或者担心内存泄漏（memory leak）！下面是个快速入门示例：

```
// raii.rs
fn create_box() {
    // 在堆上分配一个整型数据
    let _box1 = Box::new(3i32);

    // `_box1` 在这里被销毁，内存得到释放
}

fn main() {
    // 在堆上分配一个整型数据
    let _box2 = Box::new(5i32);

    // 嵌套作用域：
    {
        // 在堆上分配一个整型数据
        let _box3 = Box::new(4i32);

        // `_box3` 在这里被销毁，内存得到释放
    }

    // 创建一大堆 box（只是因为好玩）。
    // 完全不需要手动释放内存！
    for _ in 0u32..1_000 {
        create_box();
    }

    // `_box2` 在这里被销毁，内存得到释放
}
```

当然我们可以使用 `valgrind` 对内存错误进行仔细检查：

```
$ rustc raii.rs && valgrind ./raii
==26873== Memcheck, a memory error detector
==26873== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26873== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==26873== Command: ./raii
==26873==
==26873==
==26873== HEAP SUMMARY:
==26873==     in use at exit: 0 bytes in 0 blocks
==26873==   total heap usage: 1,013 allocs, 1,013 frees, 8,696 bytes allocated
==26873==
==26873== All heap blocks were freed -- no leaks are possible
==26873==
==26873== For counts of detected and suppressed errors, rerun with: -v
==26873== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

完全没有泄漏!

析构函数

Rust 中的析构函数概念是通过 `Drop` trait 提供的。当资源离开作用域，就调用析构函数。你无需为每种类型都实现 `Drop` trait，只要为那些需要自己的析构函数逻辑的类型实现就可以了。

运行下列例子，看看 `Drop` trait 是怎样工作的。当 `main` 函数中的变量离开作用域，自定义的析构函数就会被调用：

```
struct ToDrop;

impl Drop for ToDrop {
    fn drop(&mut self) {
        println!("ToDrop is being dropped");
    }
}

fn main() {
    let x = ToDrop;
    println!("Made a ToDrop!");
}
```

参见：

Box

所有权和移动

因为变量要负责释放它们拥有的资源，所以**资源只能拥有一个所有者**。这也防止了资源的重复释放。注意并非所有变量都拥有资源（例如引用）。

在进行赋值（`let x = y`）或通过值来传递函数参数（`foo(x)`）的时候，资源的**所有权**（ownership）会发生转移。按照 Rust 的说法，这被称为资源的**移动**（move）。

在移动资源之后，原来的所有者不能再被使用，这可避免悬挂指针（dangling pointer）的产生。

```
// 此函数取得堆分配的内存的所有权
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);

    // `c` 被销毁且内存得到释放
}

fn main() {
    // 栈分配的整型
    let x = 5u32;

    // 将 `x` *复制*到 `y`——不存在资源移动
    let y = x;

    // 两个值各自都可以使用
    println!("x is {}, and y is {}", x, y);

    // `a` 是一个指向堆分配的整数的指针
    let a = Box::new(5i32);

    println!("a contains: {}", a);

    // *移动* `a` 到 `b`
    let b = a;
    // 把 `a` 的指针地址（而非数据）复制到 `b`。现在两者都指向
    // 同一个堆分配的数据，但是现在是 `b` 拥有它。

    // 报错！`a` 不能访问数据，因为它不再拥有那部分堆上的内存。
    //println!("a contains: {}", a);
    // 试一试 ^ 去掉此行注释

    // 此函数从 `b` 中取得堆分配的内存的所有权
    destroy_box(b);

    // 此时堆内存已经被释放，这个操作会导致解引用已释放的内存，而这是编译器禁止的
    // 报错！和前面出错的原因一样。
    //println!("b contains: {}", b);
    // 试一试 ^ 去掉此行注释
}
```

可变性

当所有权转移时，数据的可变性可能发生改变。

```
fn main() {
    let immutable_box = Box::new(5u32);

    println!("immutable_box contains {}", immutable_box);

    // 可变性错误
    //*immutable_box = 4;

    // *移动* box，改变所有权（和可变性）
    let mut mutable_box = immutable_box;

    println!("mutable_box contains {}", mutable_box);

    // 修改 box 的内容
    *mutable_box = 4;

    println!("mutable_box now contains {}", mutable_box);
}
```

部分移动

在单个变量的解构内，可以同时使用 `by-move` 和 `by-reference` 模式绑定。这样做将导致变量的**部分移动**（partial move），这意味着变量的某些部分将被移动，而其他部分将保留。在这种情况下，后面不能整体使用父级变量，但是仍然可以使用只引用（而不移动）的部分。

```
fn main() {
    #[derive(Debug)]
    struct Person {
        name: String,
        age: u8,
    }

    let person = Person {
        name: String::from("Alice"),
        age: 20,
    };

    // `name` 从 person 中移走，但 `age` 只是引用
    let Person { name, ref age } = person;

    println!("The person's age is {}", age);

    println!("The person's name is {}", name);

    // 报错！部分移动值的借用：`person` 部分借用产生
    //println!("The person struct is {:?}", person);

    // `person` 不能使用，但 `person.age` 因为没有被移动而可以继续使用
    println!("The person's age from person struct is {}", person.age);
}
```

参见：

解构

借用

多数情况下，我们更希望能访问数据，同时不取得其所有权。为实现这点，Rust 使用了**借用**（borrowing）机制。对象可以通过引用（`&T`）来传递，从而取代通过值（`T`）来传递。

编译器（通过借用检查）静态地保证了引用**总是**指向有效的对象。也就是说，当存在引用指向一个对象时，该对象不能被销毁。

```rust
// 此函数取得一个 box 的所有权并销毁它
fn eat_box_i32(boxed_i32: Box<i32>) {
    println!("Destroying box that contains {}", boxed_i32);
}

// 此函数借用了一个 i32 类型
fn borrow_i32(borrowed_i32: &i32) {
    println!("This int is: {}", borrowed_i32);
}

fn main() {
    // 创建一个装箱的 i32 类型，以及一个存在栈中的 i32 类型。
    let boxed_i32 = Box::new(5_i32);
    let stacked_i32 = 6_i32;

    // 借用了 box 的内容，但没有取得所有权，所以 box 的内容之后可以再次借用。
    // 译注：请注意函数自身就是一个作用域，因此下面两个函数运行完成以后，
    // 在函数中临时创建的引用也就不复存在了。
    borrow_i32(&boxed_i32);
    borrow_i32(&stacked_i32);

    {
        // 取得一个对 box 中数据的引用
        let _ref_to_i32: &i32 = &boxed_i32;

        // 报错！
        // 当 `boxed_i32` 里面的值之后在作用域中被借用时，不能将其销毁。
        eat_box_i32(boxed_i32);
        // 改正 ^ 注释掉此行

        // 在 `_ref_to_i32` 里面的值被销毁后，尝试借用 `_ref_to_i32`
        // （译注：如果此处不借用，则在上一行的代码中，eat_box_i32(boxed_i32)可以将
        borrow_i32(_ref_to_i32);
        // `_ref_to_i32` 离开作用域且不再被借用。
    }

    // `boxed_i32` 现在可以将所有权交给 `eat_i32` 并被销毁。
    // （译注：能够销毁是因为已经不存在对 `boxed_i32` 的引用）
    eat_box_i32(boxed_i32);
}
```

可变性

可变数据可以使用 `&mut T` 进行可变借用。这叫做**可变引用**（mutable reference），它使借用者可以读/写数据。相反，`&T` 通过不可变引用（immutable reference）来借用数据，借用者可以读数据而不能更改数据：

```
#[allow(dead_code)]
#[derive(Clone, Copy)]
struct Book {
    // `&'static str` 是一个对分配在只读内存区的字符串的引用
    author: &'static str,
    title: &'static str,
    year: u32,
}

// 此函数接受一个对 Book 类型的引用
fn borrow_book(book: &Book) {
    println!("I immutably borrowed {} - {} edition", book.title, book.year);
}

// 此函数接受一个对可变的 Book 类型的引用，它把年份 `year` 改为 2014 年
fn new_edition(book: &mut Book) {
    book.year = 2014;
    println!("I mutably borrowed {} - {} edition", book.title, book.year);
}

fn main() {
    // 创建一个名为 `immutabook` 的不可变的 Book 实例
    let immutabook = Book {
        // 字符串字面量拥有 `&'static str` 类型
        author: "Douglas Hofstadter",
        title: "Gödel, Escher, Bach",
        year: 1979,
    };

    // 创建一个 `immutabook` 的可变拷贝，命名为 `mutabook`
    let mut mutabook = immutabook;

    // 不可变地借用一个不可变对象
    borrow_book(&immutabook);

    // 不可变地借用一个可变对象
    borrow_book(&mutabook);

    // 可变地借用一个可变对象
    new_edition(&mut mutabook);

    // 报错！不能可变地借用一个不可变对象
    new_edition(&mut immutabook);
    // 改正 ^ 注释掉此行
}
```

参见：

`static`

别名使用

数据可以多次不可变借用，但是在不可变借用的同时，原始数据不能使用可变借用。或者说，同一时间内只允许**一次**可变借用。仅当最后一次使用可变引用**之后**，原始数据才可以再次借用。

```
struct Point { x: i32, y: i32, z: i32 }

fn main() {
    let mut point = Point { x: 0, y: 0, z: 0 };

    let borrowed_point = &point;
    let another_borrow = &point;

    // 数据可以通过引用或原始类型来访问
    println!("Point has coordinates: ({}, {}, {})",
                 borrowed_point.x, another_borrow.y, point.z);

    // 报错！`point` 不能以可变方式借用，因为当前还有不可变借用。
    // let mutable_borrow = &mut point;
    // TODO ^ 试一试去掉此行注释

    // 被借用的值在这里被重新使用
    println!("Point has coordinates: ({}, {}, {})",
                 borrowed_point.x, another_borrow.y, point.z);

    // 不可变的引用不再用于其余的代码，因此可以使用可变的引用重新借用。
    let mutable_borrow = &mut point;

    // 通过可变引用来修改数据
    mutable_borrow.x = 5;
    mutable_borrow.y = 2;
    mutable_borrow.z = 1;

    // 报错！不能再以不可变方式来借用 `point`，因为它当前已经被可变借用。
    // let y = &point.y;
    // TODO ^ 试一试去掉此行注释

    // 报错！无法打印，因为 `println!` 用到了一个不可变引用。
    // println!("Point Z coordinate is {}", point.z);
    // TODO ^ 试一试去掉此行注释

    // 正常运行！可变引用能够以不可变类型传入 `println!`
    println!("Point has coordinates: ({}, {}, {})",
                 mutable_borrow.x, mutable_borrow.y, mutable_borrow.z);

    // 可变引用不再用于其余的代码，因此可以重新借用
    let new_borrowed_point = &point;
    println!("Point now has coordinates: ({}, {}, {})",
                 new_borrowed_point.x, new_borrowed_point.y, new_borrowed_point.z);
}
```

`ref` 模式

在通过 `let` 绑定来进行模式匹配或解构时，`ref` 关键字可用来创建结构体/元组的字段的引用。下面的例子展示了几个实例，可看到 `ref` 的作用：

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

fn main() {
    let c = 'Q';

    // 赋值语句中左边的 `ref` 关键字等价于右边的 `&` 符号。
    let ref ref_c1 = c;
    let ref_c2 = &c;

    println!("ref_c1 equals ref_c2: {}", *ref_c1 == *ref_c2);

    let point = Point { x: 0, y: 0 };

    // 在解构一个结构体时 `ref` 同样有效。
    let _copy_of_x = {
        // `ref_to_x` 是一个指向 `point` 的 `x` 字段的引用。
        let Point { x: ref ref_to_x, y: _ } = point;

        // 返回一个 `point` 的 `x` 字段的拷贝。
        *ref_to_x
    };

    // `point` 的可变拷贝
    let mut mutable_point = point;

    {
        // `ref` 可以与 `mut` 结合以创建可变引用。
        let Point { x: _, y: ref mut mut_ref_to_y } = mutable_point;

        // 通过可变引用来改变 `mutable_point` 的字段 `y`。
        *mut_ref_to_y = 1;
    }

    println!("point is ({}, {})", point.x, point.y);
    println!("mutable_point is ({}, {})", mutable_point.x, mutable_point.y);

    // 包含一个指针的可变元组
    let mut mutable_tuple = (Box::new(5u32), 3u32);

    {
        // 解构 `mutable_tuple` 来改变 `last` 的值。
        let (_, ref mut last) = mutable_tuple;
        *last = 2u32;
    }

    println!("tuple is {:?}", mutable_tuple);
}
```

生命周期

生命周期（lifetime）是这样一种概念，编译器（中的借用检查器）用它来保证所有的借用都是有效的。确切地说，一个变量的生命周期在它创建的时候开始，在它销毁的时候结束。虽然生命周期和作用域经常被一起提到，但它们并不相同。

例如考虑这种情况，我们通过 `&` 来借用一个变量。该借用拥有一个生命周期，此生命周期由它声明的位置决定。于是，只要该借用在出借者（lender）被销毁前结束，借用就是有效的。然而，借用的作用域则是由使用引用的位置决定的。

在下面的例子和本章节剩下的内容里，我们将看到生命周期和作用域的联系与区别。

> 译注：如果代码中的生命周期示意图乱掉了，请把它复制到任何编辑器中，用等宽字体查看。为避免中文的显示问题，下面一些注释没有翻译。

```
// 下面使用连线来标注各个变量的创建和销毁，从而显示出生命周期。
// `i` 的生命周期最长，因为它的作用域完全覆盖了 `borrow1` 和
// `borrow2` 的。`borrow1` 和 `borrow2` 的周期没有关联，
// 因为它们各不相交。
fn main() {
    let i = 3; // Lifetime for `i` starts. ────────────────┐
    //                                                     │
    { //                                                   │
        let borrow1 = &i; // `borrow1` lifetime starts. ──┐│
        //                                                ││
        println!("borrow1: {}", borrow1); //              ││
    } // `borrow1 ends. ──────────────────────────────────┘│
    //                                                     │
    //                                                     │
    { //                                                   │
        let borrow2 = &i; // `borrow2` lifetime starts. ──┐│
        //                                                ││
        println!("borrow2: {}", borrow2); //              ││
    } // `borrow2` ends. ──────────────────────────────────┘│
    //                                                     │
}   // Lifetime ends. ──────────────────────────────────────┘
```

注意到这里没有用到名称或类型来标注生命周期，这限制了生命周期的用法，在后面我们将会看到生命周期更强大的功能。

显式标注

借用检查器使用显式的生命周期标记来明确引用的有效时间应该持续多久。在生命周期没有省略[1]的情况下，Rust 需要显式标注来确定引用的生命周期应该是什么样的。可以用撇号显式地标出生命周期，语法如下：

```
foo<'a>
// `foo` 带有一个生命周期参数 `'a`
```

和闭包类似，使用生命周期需要泛型。另外这个生命周期的语法也表明了 `foo` 的生命周期不能超出 `'a` 的周期。若要给类型显式地标注生命周期，其语法会像是 `&'a T` 这样，其中 `'a` 的作用刚刚已经介绍了。

```
foo<'a, 'b>
// `foo` 带有生命周期参数 `'a` 和 `'b`
```

在上面这种情形中， `foo` 的生命周期不能超出 `'a` 和 `'b` 中任一个的周期。

看下面的例子，了解显式生命周期标注的运用：

```
// `print_refs` 接受两个 `i32` 的引用，它们有不同的生命周期 `'a` 和 `'b`。
// 这两个生命周期都必须至少要和 `print_refs` 函数一样长。
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}

// 不带参数的函数，不过有一个生命周期参数 `'a`。
fn failed_borrow<'a>() {
    let _x = 12;

    // 报错：`_x` 的生命周期不够长
    //let y: &'a i32 = &_x;
    // 在函数内部使用生命周期 `'a` 作为显式类型标注将导致失败，因为 `&_x` 的
    // 生命周期比 `y` 的短。短生命周期不能强制转换成长生命周期。
}

fn main() {
    // 创建变量，稍后用于借用。
    let (four, nine) = (4, 9);

    // 两个变量的借用（`&`）都传进函数。
    print_refs(&four, &nine);
    // 任何被借用的输入量都必须比借用者生存得更长。
    // 也就是说，`four` 和 `nine` 的生命周期都必须比 `print_refs` 的长。

    failed_borrow();
    // `failed_borrow` 未包含引用，因此不要求 `'a` 长于函数的生命周期，
    // 但 `'a` 寿命确实更长。因为该生命周期从未被约束，所以默认为 `'static`。
}
```

[1] 省略 隐式地标注了生命周期，所以情况不同。

参见：

泛型 和 闭包

函数

排除省略（elision）的情况，带上生命周期的函数签名有一些限制：

- 任何引用**都必须**拥有标注好的生命周期。
- 任何被返回的引用**都必须**有和某个输入量相同的生命周期或是静态类型（`static`）。

另外要注意，如果没有输入的函数返回引用，有时会导致返回的引用指向无效数据，这种情况下禁止它返回这样的引用。下面例子展示了一些合法的带有生命周期的函数：

```
// 一个拥有生命周期 `'a` 的输入引用，其中 `'a` 的存活时间
// 至少与函数的一样长。
fn print_one<'a>(x: &'a i32) {
    println!("`print_one`: x is {}", x);
}

// 可变引用同样也可能拥有生命周期。
fn add_one<'a>(x: &'a mut i32) {
    *x += 1;
}

// 拥有不同生命周期的多个元素。对下面这种情形，两者即使拥有
// 相同的生命周期 `'a` 也没问题，但对一些更复杂的情形，可能
// 就需要不同的生命周期了。
fn print_multi<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("`print_multi`: x is {}, y is {}", x, y);
}

// 返回传递进来的引用也是可行的。
// 但必须返回正确的生命周期。
fn pass_x<'a, 'b>(x: &'a i32, _: &'b i32) -> &'a i32 { x }

//fn invalid_output<'a>() -> &'a String { &String::from("foo") }
// 上面代码是无效的：`'a` 存活的时间必须比函数的长。
// 这里的 `&String::from("foo")` 将会创建一个 `String` 类型，然后对它取引用。
// 数据在离开作用域时删掉，返回一个指向无效数据的引用。

fn main() {
    let x = 7;
    let y = 9;

    print_one(&x);
    print_multi(&x, &y);

    let z = pass_x(&x, &y);
    print_one(z);

    let mut t = 3;
    add_one(&mut t);
    print_one(&t);
}
```

参见：

函数

方法

方法的标注和函数类似：

```
struct Owner(i32);

impl Owner {
    // 标注生命周期，就像独立的函数一样。
    fn add_one<'a>(&'a mut self) { self.0 += 1; }
    fn print<'a>(&'a self) {
        println!("`print`: {}", self.0);
    }
}

fn main() {
    let mut owner  = Owner(18);

    owner.add_one();
    owner.print();
}
```

> 译注：方法一般是不需要标明生命周期的，因为 `self` 的生命周期会赋给所有的输出生命周期参数，详见 TRPL。

参见：

方法

结构体

在结构体中标注生命周期也和函数的类似：

```
// 一个 `Borrowed` 类型，含有一个指向 `i32` 类型的引用。
// 该引用必须比 `Borrowed` 寿命更长。
#[derive(Debug)]
struct Borrowed<'a>(&'a i32);

// 和前面类似，这里的两个引用都必须比这个结构体长寿。
#[derive(Debug)]
struct NamedBorrowed<'a> {
    x: &'a i32,
    y: &'a i32,
}

// 一个枚举类型，其取值不是 `i32` 类型就是一个指向 `i32` 的引用。
#[derive(Debug)]
enum Either<'a> {
    Num(i32),
    Ref(&'a i32),
}

fn main() {
    let x = 18;
    let y = 15;

    let single = Borrowed(&x);
    let double = NamedBorrowed { x: &x, y: &y };
    let reference = Either::Ref(&x);
    let number    = Either::Num(y);

    println!("x is borrowed in {:?}", single);
    println!("x and y are borrowed in {:?}", double);
    println!("x is borrowed in {:?}", reference);
    println!("y is *not* borrowed in {:?}", number);
}
```

参见：

结构体

trait

trait 方法中生命期的标注基本上与函数类似。注意，`impl` 也可能有生命周期的标注。

```
// 带有生命周期标注的结构体。
#[derive(Debug)]
 struct Borrowed<'a> {
     x: &'a i32,
 }

// 给 impl 标注生命周期。
impl<'a> Default for Borrowed<'a> {
    fn default() -> Self {
        Self {
            x: &10,
        }
    }
}

fn main() {
    let b: Borrowed = Default::default();
    println!("b is {:?}", b);
}
```

参见：

`trait`

约束

就如泛型类型能够被约束一样，生命周期（它们本身就是泛型）也可以使用约束。`:` 字符的意义在这里稍微有些不同，不过 `+` 是相同的。注意下面的说明：

1. `T: 'a`：在 `T` 中的**所有**引用都必须比生命周期 `'a` 活得更长。
2. `T: Trait + 'a`：`T` 类型必须实现 `Trait` trait，并且在 `T` 中的**所有**引用都必须比 `'a` 活得更长。

下面例子展示了上述语法的实际应用：

```
use std::fmt::Debug; // 用于约束的 trait。

#[derive(Debug)]
struct Ref<'a, T: 'a>(&'a T);
// `Ref` 包含一个指向泛型类型 `T` 的引用，其中 `T` 拥有一个未知的生命周期
// `'a`。`T` 拥有生命周期限制，`T` 中的任何*引用*都必须比 `'a` 活得更长。另外
// `Ref` 的生命周期也不能超出 `'a`。

// 一个泛型函数，使用 `Debug` trait 来打印内容。
fn print<T>(t: T) where
    T: Debug {
    println!("`print`: t is {:?}", t);
}

// 这里接受一个指向 `T` 的引用，其中 `T` 实现了 `Debug` trait，并且在 `T` 中的
// 所有*引用*都必须比 `'a'` 存活时间更长。另外，`'a` 也要比函数活得更长。
fn print_ref<'a, T>(t: &'a T) where
    T: Debug + 'a {
    println!("`print_ref`: t is {:?}", t);
}

fn main() {
    let x = 7;
    let ref_x = Ref(&x);

    print_ref(&ref_x);
    print(ref_x);
}
```

参见：

泛型, 泛型中的约束, 以及 泛型中的多重约束

强制转换

一个较长的生命周期可以强制转成一个较短的生命周期，使它在一个通常情况下不能工作的作用域内也能正常工作。强制转换可由编译器隐式地推导并执行，也可以通过声明不同的生命周期的形式实现。

```
// 在这里，Rust 推导了一个尽可能短的生命周期。
// 然后这两个引用都被强制转成这个生命周期。
fn multiply<'a>(first: &'a i32, second: &'a i32) -> i32 {
    first * second
}

// `<'a: 'b, 'b>` 读作生命周期 `'a` 至少和 `'b` 一样长。
// 在这里我们我们接受了一个 `&'a i32` 类型并返回一个 `&'b i32` 类型，这是
// 强制转换得到的结果。
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}

fn main() {
    let first = 2; // 较长的生命周期

    {
        let second = 3; // 较短的生命周期

        println!("The product is {}", multiply(&first, &second));
        println!("{} is the first", choose_first(&first, &second));
    };
}
```

static

`'static` 生命周期是可能的生命周期中最长的，它会在整个程序运行的时期中存在。`'static` 生命周期也可被强制转换成一个更短的生命周期。有两种方式使变量拥有 `'static` 生命周期，它们都把数据保存在可执行文件的只读内存区：

- 使用 `static` 声明来产生常量（constant）。
- 产生一个拥有 `&'static str` 类型的 `string` 字面量。

看下面的例子，了解列举到的各个方法：

```
// 产生一个拥有 `'static` 生命周期的常量。
static NUM: i32 = 18;

// 返回一个指向 `NUM` 的引用，该引用不取 `NUM` 的 `'static` 生命周期，
// 而是被强制转换成和输入参数的一样。
fn coerce_static<'a>(_: &'a i32) -> &'a i32 {
    &NUM
}

fn main() {
    {
        // 产生一个 `string` 字面量并打印它：
        let static_string = "I'm in read-only memory";
        println!("static_string: {}", static_string);

        // 当 `static_string` 离开作用域时，该引用不能再使用，不过
        // 数据仍然存在于二进制文件里面。
    }

    {
        // 产生一个整型给 `coerce_static` 使用：
        let lifetime_num = 9;

        // 将对 `NUM` 的引用强制转换成 `lifetime_num` 的生命周期：
        let coerced_static = coerce_static(&lifetime_num);

        println!("coerced_static: {}", coerced_static);
    }

    println!("NUM: {} stays accessible!", NUM);
}
```

参见：

`'static` 常量

省略

有些生命周期的模式太常用了，所以借用检查器将会隐式地添加它们以减少程序输入量和增强可读性。这种隐式添加生命周期的过程称为省略（elision）。在 Rust 使用省略仅仅是因为这些模式太普遍了。

下面代码展示了一些省略的例子。对于省略的详细描述，可以参考官方文档的生命周期省略。

```
// `elided_input` 和 `annotated_input` 事实上拥有相同的签名，
// `elided_input` 的生命周期会被编译器自动添加：
fn elided_input(x: &i32) {
    println!("`elided_input`: {}", x)
}

fn annotated_input<'a>(x: &'a i32) {
    println!("`annotated_input`: {}", x)
}

// 类似地，`elided_pass` 和 `annotated_pass` 也拥有相同的签名，
// 生命周期会被隐式地添加进 `elided_pass`：
fn elided_pass(x: &i32) -> &i32 { x }

fn annotated_pass<'a>(x: &'a i32) -> &'a i32 { x }

fn main() {
    let x = 3;

    elided_input(&x);
    annotated_input(&x);

    println!("`elided_pass`: {}", elided_pass(&x));
    println!("`annotated_pass`: {}", annotated_pass(&x));
}
```

参见：

省略

特质 trait

`trait` 是对未知类型 `Self` 定义的方法集。该类型也可以访问同一个 trait 中定义的其他方法。

对任何数据类型都可以实现 trait。在下面例子中，我们定义了包含一系列方法的 `Animal` 。然后针对 `Sheep` 数据类型实现 `Animal` `trait`，因而 `Sheep` 的实例可以使用 `Animal` 中的所有方法。

```rust
struct Sheep { naked: bool, name: &'static str }

trait Animal {
    // 静态方法签名；`Self` 表示实现者类型（implementor type）。
    fn new(name: &'static str) -> Self;

    // 实例方法签名；这些方法将返回一个字符串。
    fn name(&self) -> &'static str;
    fn noise(&self) -> &'static str;

    // trait 可以提供默认的方法定义。
    fn talk(&self) {
        println!("{} says {}", self.name(), self.noise());
    }
}

impl Sheep {
    fn is_naked(&self) -> bool {
        self.naked
    }

    fn shear(&mut self) {
        if self.is_naked() {
            // 实现者可以使用它的 trait 方法。
            println!("{} is already naked...", self.name());
        } else {
            println!("{} gets a haircut!", self.name);

            self.naked = true;
        }
    }
}

// 对 `Sheep` 实现 `Animal` trait。
impl Animal for Sheep {
    // `Self` 是实现者类型：`Sheep`。
    fn new(name: &'static str) -> Sheep {
        Sheep { name: name, naked: false }
    }

    fn name(&self) -> &'static str {
        self.name
    }

    fn noise(&self) -> &'static str {
        if self.is_naked() {
            "baaaaah?"
        } else {
            "baaaaah!"
        }
    }

    // 默认 trait 方法可以重载。
    fn talk(&self) {
        // 例如我们可以增加一些安静的沉思。
        println!("{} pauses briefly... {}", self.name, self.noise());
    }
}
```

```
fn main() {
    // 这种情况需要类型标注。
    let mut dolly: Sheep = Animal::new("Dolly");
    // 试一试 ^ 移除类型标注。

    dolly.talk();
    dolly.shear();
    dolly.talk();
}
```

派生

通过 `#[derive]` 属性，编译器能够提供某些 trait 的基本实现。如果需要更复杂的行为，这些 trait 也可以手动实现。

下面是可以自动派生的 trait：

- 比较 trait: `Eq`，`PartialEq`，`Ord`，`PartialOrd`
- `Clone`，用来从 `&T` 创建副本 `T`。
- `Copy`，使类型具有“复制语义”（copy semantics）而非“移动语义”（move semantics）。
- `Hash`，从 `&T` 计算哈希值（hash）。
- `Default`，创建数据类型的一个空实例。
- `Debug`，使用 `{:?}` formatter 来格式化一个值。

```rust
// `Centimeters`, 可以比较的元组结构体
#[derive(PartialEq, PartialOrd)]
struct Centimeters(f64);

// `Inches`, 可以打印的元组结构体
#[derive(Debug)]
struct Inches(i32);

impl Inches {
    fn to_centimeters(&self) -> Centimeters {
        let &Inches(inches) = self;

        Centimeters(inches as f64 * 2.54)
    }
}

// `Seconds`, 不带附加属性的元组结构体
struct Seconds(i32);

fn main() {
    let _one_second = Seconds(1);

    // 报错：`Seconds` 不能打印；它没有实现 `Debug` trait
    //println!("One second looks like: {:?}", _one_second);
    // 试一试 ^ 取消此行注释

    // 报错：`Seconds`不能比较；它没有实现 `PartialEq` trait
    //let _this_is_true = (_one_second == _one_second);
    // 试一试 ^ 取消此行注释

    let foot = Inches(12);

    println!("One foot equals {:?}", foot);

    let meter = Centimeters(100.0);

    let cmp =
        if foot.to_centimeters() < meter {
            "smaller"
        } else {
            "bigger"
        };

    println!("One foot is {} than one meter.", cmp);
}
```

参见

`derive`

使用 dyn 返回 trait

Rust 编译器需要知道每个函数的返回类型需要多少空间。这意味着所有函数都必须返回一个具体类型。与其他语言不同，如果你有个像 `Animal` 那样的的 trait，则不能编写返回 `Animal` 的函数，因为其不同的实现将需要不同的内存量。

但是，有一个简单的解决方法。相比于直接返回一个 trait 对象，我们的函数返回一个包含一些 `Animal` 的 `Box`。`box` 只是对堆中某些内存的引用。因为引用的大小是静态已知的，并且编译器可以保证引用指向已分配的堆 `Animal`，所以我们可以从函数中返回 trait！

每当在堆上分配内存时，Rust 都会尝试尽可能明确。因此，如果你的函数以这种方式返回指向堆的 trait 指针，则需要使用 `dyn` 关键字编写返回类型，例如 `Box<dyn Animal>`。

```
struct Sheep {}
struct Cow {}

trait Animal {
    // 实例方法签名
    fn noise(&self) -> &'static str;
}

// 实现 `Sheep` 的 `Animal` trait。
impl Animal for Sheep {
    fn noise(&self) -> &'static str {
        "baaaaah!"
    }
}

// 实现 `Cow` 的 `Animal` trait。
impl Animal for Cow {
    fn noise(&self) -> &'static str {
        "mooooo!"
    }
}

// 返回一些实现 Animal 的结构体，但是在编译时我们不知道哪个结构体。
fn random_animal(random_number: f64) -> Box<dyn Animal> {
    if random_number < 0.5 {
        Box::new(Sheep {})
    } else {
        Box::new(Cow {})
    }
}

fn main() {
    let random_number = 0.234;
    let animal = random_animal(random_number);
    println!("You've randomly chosen an animal, and it says {}", animal.noise());
}
```

运算符重载

在 Rust 中，很多运算符可以通过 trait 来重载。也就是说，这些运算符可以根据它们的输入参数来完成不同的任务。这之所以可行，是因为运算符就是方法调用的语法糖。例如，`a + b` 中的 `+` 运算符会调用 `add` 方法（也就是 `a.add(b)`）。这个 `add` 方法是 `Add` trait 的一部分。因此，`+` 运算符可以被任何 `Add` trait 的实现者使用。

会重载运算符的 `trait`（比如 `Add` 这种）可以在这里查看。

```
use std::ops;

struct Foo;
struct Bar;

#[derive(Debug)]
struct FooBar;

#[derive(Debug)]
struct BarFoo;

// `std::ops::Add` trait 用来指明 `+` 的功能，这里我们实现 `Add<Bar>`，它是用于
// 把对象和 `Bar` 类型的右操作数（RHS）加起来的 `trait`。
// 下面的代码块实现了 `Foo + Bar = FooBar` 这样的运算。
impl ops::Add<Bar> for Foo {
    type Output = FooBar;

    fn add(self, _rhs: Bar) -> FooBar {
        println!("> Foo.add(Bar) was called");

        FooBar
    }
}

// 通过颠倒类型，我们实现了不服从交换律的加法。
// 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。
// 下面的代码块实现了 `Bar + Foo = BarFoo` 这样的运算。
impl ops::Add<Foo> for Bar {
    type Output = BarFoo;

    fn add(self, _rhs: Foo) -> BarFoo {
        println!("> Bar.add(Foo) was called");

        BarFoo
    }
}

fn main() {
    println!("Foo + Bar = {:?}", Foo + Bar);
    println!("Bar + Foo = {:?}", Bar + Foo);
}
```

参见：

Add, 语法索引

Drop

`Drop` trait 只有一个方法：`drop`，当对象离开作用域时会自动调用该方法。`Drop` trait 的主要作用是释放实现者的实例拥有的资源。

`Box`，`Vec`，`String`，`File`，以及 `Process` 是一些实现了 `Drop` trait 来释放资源的类型。`Drop` trait 也可以为任何自定义数据类型手动实现。

下面示例给 `drop` 函数增加了打印到控制台的功能，用于宣布它在什么时候被调用。

```
struct Droppable {
    name: &'static str,
}

// 这个简单的 `drop` 实现添加了打印到控制台的功能。
impl Drop for Droppable {
    fn drop(&mut self) {
        println!("> Dropping {}", self.name);
    }
}

fn main() {
    let _a = Droppable { name: "a" };

    // 代码块 A
    {
        let _b = Droppable { name: "b" };

        // 代码块 B
        {
            let _c = Droppable { name: "c" };
            let _d = Droppable { name: "d" };

            println!("Exiting block B");
        }
        println!("Just exited block B");

        println!("Exiting block A");
    }
    println!("Just exited block A");

    // 变量可以手动使用 `drop` 函数来销毁。
    drop(_a);
    // 试一试 ^ 将此行注释掉。

    println!("end of the main function");

    // `_a` *不会*在这里再次销毁，因为它已经被（手动）销毁。
}
```

Iterator

`Iterator` trait 用来对集合（collection）类型（比如数组）实现迭代器。

这个 trait 只需定义一个返回 `next`（下一个）元素的方法，这可手动在 `impl` 代码块中定义，或者自动定义（比如在数组或区间中）。

为方便起见，`for` 结构会使用 `.into_iter()` 方法将一些集合类型转换为迭代器。

下面例子展示了如何使用 `Iterator` trait 的方法，更多可用的方法可以看这里。

```rust
struct Fibonacci {
    curr: u32,
    next: u32,
}

// 为 `Fibonacci`（斐波那契）实现 `Iterator`。
// `Iterator` trait 只需定义一个能返回 `next`（下一个）元素的方法。
impl Iterator for Fibonacci {
    type Item = u32;

    // 我们在这里使用 `.curr` 和 `.next` 来定义数列（sequence）。
    // 返回类型为 `Option<T>`：
    //     * 当 `Iterator` 结束时，返回 `None`。
    //     * 其他情况，返回被 `Some` 包裹（wrap）的下一个值。
    fn next(&mut self) -> Option<u32> {
        let new_next = self.curr + self.next;

        self.curr = self.next;
        self.next = new_next;

        // 既然斐波那契数列不存在终点，那么 `Iterator` 将不可能
        // 返回 `None`，而总是返回 `Some`。
        Some(self.curr)
    }
}

// 返回一个斐波那契数列生成器
fn fibonacci() -> Fibonacci {
    Fibonacci { curr: 1, next: 1 }
}

fn main() {
    // `0..3` 是一个 `Iterator`，会产生：0、1 和 2。
    let mut sequence = 0..3;

    println!("Four consecutive `next` calls on 0..3");
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());

    // `for` 遍历 `Iterator` 直到返回 `None`，
    // 并且每个 `Some` 值都被解包（unwrap），然后绑定给一个变量（这里是 `i`）。
    println!("Iterate through 0..3 using `for`");
    for i in 0..3 {
        println!("> {}", i);
    }

    // `take(n)` 方法提取 `Iterator` 的前 `n` 项。
    println!("The first four terms of the Fibonacci sequence are: ");
    for i in fibonacci().take(4) {
        println!("> {}", i);
    }

    // `skip(n)` 方法移除前 `n` 项，从而缩短了 `Iterator` 。
    println!("The next four terms of the Fibonacci sequence are: ");
    for i in fibonacci().skip(4).take(4) {
        println!("> {}", i);
```

```
    }

    let array = [1u32, 3, 3, 7];

    // `iter` 方法对数组/slice 产生一个 `Iterator`。
    println!("Iterate the following array {:?}", &array);
    for i in array.iter() {
        println!("> {}", i);
    }
}
```

impl Trait

如果函数返回实现了 `MyTrait` 的类型，可以将其返回类型编写为 `-> impl MyTrait`。这可以大大简化你的类型签名！

```
use std::iter;
use std::vec::IntoIter;

// 该函数组合了两个 `Vec <i32>` 并在其上返回一个迭代器。
// 看看它的返回类型多么复杂！
fn combine_vecs_explicit_return_type(
    v: Vec<i32>,
    u: Vec<i32>,
) -> iter::Cycle<iter::Chain<IntoIter<i32>, IntoIter<i32>>> {
    v.into_iter().chain(u.into_iter()).cycle()
}

// 这是完全相同的函数，但其返回类型使用 `impl Trait`。
// 看看它多么简单！
fn combine_vecs(
    v: Vec<i32>,
    u: Vec<i32>,
) -> impl Iterator<Item=i32> {
    v.into_iter().chain(u.into_iter()).cycle()
}

fn main() {
    let v1 = vec![1, 2, 3];
    let v2 = vec![4, 5];
    let mut v3 = combine_vecs(v1, v2);
    assert_eq!(Some(1), v3.next());
    assert_eq!(Some(2), v3.next());
    assert_eq!(Some(3), v3.next());
    assert_eq!(Some(4), v3.next());
    assert_eq!(Some(5), v3.next());
    println!("all done");
}
```

更重要的是，某些 Rust 类型无法写出。例如，每个闭包都有自己未命名的具体类型。在使用 `impl Trait` 语法之前，必须在堆上进行分配才能返回闭包。但是现在你可以像下面这样静态地完成所有操作：

```
// 返回一个将输入和 `y` 相加的函数
fn make_adder_function(y: i32) -> impl Fn(i32) -> i32 {
    let closure = move |x: i32| { x + y };
    closure
}

fn main() {
    let plus_one = make_adder_function(1);
    assert_eq!(plus_one(2), 3);
}
```

您还可以使用 `impl Trait` 返回使用 `map` 或 `filter` 闭包的迭代器！这使得使用 `map` 和 `filter` 更容易。因为闭包类型没有名称，所以如果函数返回带闭包的迭代器，则无法写出显式的返回类型。但是有了 `impl Trait`，你就可以轻松地做到这一点：

```
fn double_positives<'a>(numbers: &'a Vec<i32>) -> impl Iterator<Item = i32> + 'a {
    numbers
        .iter()
        .filter(|x| x > &&0)
        .map(|x| x * 2)
}
```

Clone

当处理资源时，默认的行为是在赋值或函数调用的同时将它们转移。但是我们有时候也需要把资源复制一份。

`Clone` trait 正好帮助我们完成这任务。通常，我们可以使用由 `Clone` trait 定义的 `.clone()` 方法。

```
// 不含资源的单元结构体
#[derive(Debug, Clone, Copy)]
struct Nil;

// 一个包含资源的结构体，它实现了 `Clone` trait
#[derive(Clone, Debug)]
struct Pair(Box<i32>, Box<i32>);

fn main() {
    // 实例化 `Nil`
    let nil = Nil;
    // 复制 `Nil`，没有资源用于移动（move）
    let copied_nil = nil;

    // 两个 `Nil` 都可以独立使用
    println!("original: {:?}", nil);
    println!("copy: {:?}", copied_nil);

    // 实例化 `Pair`
    let pair = Pair(Box::new(1), Box::new(2));
    println!("original: {:?}", pair);

    // 将 `pair` 绑定到 `moved_pair`，移动（move）了资源
    let moved_pair = pair;
    println!("copy: {:?}", moved_pair);

    // 报错！`pair` 已失去了它的资源。
    //println!("original: {:?}", pair);
    // 试一试 ^ 取消此行注释。

    // 将 `moved_pair`（包括其资源）克隆到 `cloned_pair`。
    let cloned_pair = moved_pair.clone();
    // 使用 std::mem::drop 来销毁原始的 pair。
    drop(moved_pair);

    // 报错！`moved_pair` 已被销毁。
    //println!("copy: {:?}", moved_pair);
    // 试一试 ^ 将此行注释掉。

    // 由 .clone() 得来的结果仍然可用！
    println!("clone: {:?}", cloned_pair);
}
```

父 trait

Rust 没有“继承”，但是您可以将一个 trait 定义为另一个 trait 的超集（即父 trait）。例如：

```
trait Person {
    fn name(&self) -> String;
}

// Person 是 Student 的父 trait。
// 实现 Student 需要你也 impl 了 Person。
trait Student: Person {
    fn university(&self) -> String;
}

trait Programmer {
    fn fav_language(&self) -> String;
}

// CompSciStudent (computer science student, 计算机科学的学生) 是 Programmer 和 S
// 实现 CompSciStudent 需要你同时 impl 了两个父 trait。
trait CompSciStudent: Programmer + Student {
    fn git_username(&self) -> String;
}

fn comp_sci_student_greeting(student: &dyn CompSciStudent) -> String {
    format!(
        "My name is {} and I attend {}. My favorite language is {}. My Git usernar
        student.name(),
        student.university(),
        student.fav_language(),
        student.git_username()
    )
}

fn main() {}
```

参见：

《Rust 程序设计语言》的“父级 trait”章节

消除重叠 trait

一个类型可以实现许多不同的 trait。如果两个 trait 都需要相同的名称怎么办？例如，许多 trait 可能拥有名为 `get()` 的方法。他们甚至可能有不同的返回类型！

有个好消息：由于每个 trait 实现都有自己的 `impl` 块，因此很清楚您要实现哪个 trait 的 `get` 方法。

何时需要**调用**这些方法呢？为了消除它们之间的歧义，我们必须使用完全限定语法（Fully Qualified Syntax）。

```rust
trait UsernameWidget {
    // 从这个 widget 中获取选定的用户名
    fn get(&self) -> String;
}

trait AgeWidget {
    // 从这个 widget 中获取选定的年龄
    fn get(&self) -> u8;
}

// 同时具有 UsernameWidget 和 AgeWidget 的表单
struct Form {
    username: String,
    age: u8,
}

impl UsernameWidget for Form {
    fn get(&self) -> String {
        self.username.clone()
    }
}

impl AgeWidget for Form {
    fn get(&self) -> u8 {
        self.age
    }
}

fn main() {
    let form = Form{
        username: "rustacean".to_owned(),
        age: 28,
    };

    // 如果取消注释此行，则会收到一条错误消息，提示 “multiple `get` found”（找到了
    // 因为毕竟有多个名为 `get` 的方法。
    // println!("{}", form.get());

    let username = <Form as UsernameWidget>::get(&form);
    assert_eq!("rustacean".to_owned(), username);
    let age = <Form as AgeWidget>::get(&form);
    assert_eq!(28, age);
}
```

参见：

《Rust 程序设计语言》中关于“完全限定语法”的章节

使用 `macro_rules!` 来创建宏

Rust 提供了一个强大的宏系统，可进行元编程（metaprogramming）。你已经在前面的章节中看到，宏看起来和函数很像，只不过名称末尾有一个感叹号 `!` 。宏并不产生函数调用，而是展开成源码，并和程序的其余部分一起被编译。Rust 又有一点和 C 以及其他语言都不同，那就是 Rust 的宏会展开为抽象语法树（AST，abstract syntax tree），而不是像字符串预处理那样直接替换成代码，这样就不会产生无法预料的优先权错误。

宏是通过 `macro_rules!` 宏来创建的。

```
// 这是一个简单的宏，名为 `say_hello`。
macro_rules! say_hello {
    // `()` 表示此宏不接受任何参数。
    () => (
        // 此宏将会展开成这个代码块里面的内容。
        println!("Hello!");
    )
}

fn main() {
    // 这个调用将会展开成 `println("Hello");`!
    say_hello!()
}
```

为什么宏是有用的?

1. 不写重复代码（DRY，Don't repeat yourself.）。很多时候你需要在一些地方针对不同 的类型实现类似的功能，这时常常可以使用宏来避免重复代码（稍后详述）。
2. 领域专用语言（DSL，domain-specific language）。宏允许你为特定的目的创造特定的 语法（稍后详述）。
3. 可变接口（variadic interface）。有时你需要能够接受不定数目参数的接口，比如 `println!`，根据格式化字符串的不同，它需要接受任意多的参数（稍后详述）。

语法

在下面的小节中，我们将展示如何在 Rust 中定义宏。基本的概念有三个：

- 模式与指示符
- 重载
- 重复

指示符

宏的参数使用一个美元符号 `$` 作为前缀，并使用一个**指示符**（designator）来注明类型：

```
macro_rules! create_function {
    // 此宏接受一个 `ident` 指示符表示的参数，并创建一个名为 `$func_name` 的函数。
    // `ident` 指示符用于变量名或函数名
    ($func_name:ident) => (
        fn $func_name() {
            // `stringify!` 宏把 `ident` 转换成字符串。
            println!("You called {:?}()",
                     stringify!($func_name))
        }
    )
}

// 借助上述宏来创建名为 `foo` 和 `bar` 的函数。
create_function!(foo);
create_function!(bar);

macro_rules! print_result {
    // 此宏接受一个 `expr` 类型的表达式，并将它作为字符串，连同其结果一起
    // 打印出来。
    // `expr` 指示符表示表达式。
    ($expression:expr) => (
        // `stringify!` 把表达式*原样*转换成一个字符串。
        println!("{:?} = {:?}",
                 stringify!($expression),
                 $expression)
    )
}

fn main() {
    foo();
    bar();

    print_result!(1u32 + 1);

    // 回想一下，代码块也是表达式！
    print_result!({
        let x = 1u32;

        x * x + 2 * x - 1
    });
}
```

这里列出全部指示符：

- `block`
- `expr` 用于表达式
- `ident` 用于变量名或函数名

- `item`
- `literal` 用于字面常量
- `pat` (**模式** *pattern*)
- `path`
- `stmt` (**语句** *statement*)
- `tt` (**标记树** *token tree*)
- `ty` (**类型** *type*)
- `vis` (*可见性描述符*)

完整列表详见 Rust Reference。

重载

宏可以重载，从而接受不同的参数组合。在这方面，`macro_rules!` 的作用类似于匹配（match）代码块：

```
// 根据你调用它的方式，`test!` 将以不同的方式来比较 `$left` 和 `$right`。
macro_rules! test {
    // 参数不需要使用逗号隔开。
    // 参数可以任意组合！
    ($left:expr; and $right:expr) => (
        println!("{:?} and {:?} is {:?}",
                 stringify!($left),
                 stringify!($right),
                 $left && $right)
    );
    // ^ 每个分支都必须以分号结束。
    ($left:expr; or $right:expr) => (
        println!("{:?} or {:?} is {:?}",
                 stringify!($left),
                 stringify!($right),
                 $left || $right)
    );
}

fn main() {
    test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);
    test!(true; or false);
}
```

重复

宏在参数列表中可以使用 `+` 来表示一个参数可能出现一次或多次，使用 `*` 来表示该参数可能出现零次或多次。

在下面例子中，把模式这样： `$(...),+` 包围起来，就可以匹配一个或多个用逗号隔开的表达式。另外注意到，宏定义的最后一个分支可以不用分号作为结束。

```
// `min!` 将求出任意数量的参数的最小值。
macro_rules! find_min {
    // 基本情形：
    ($x:expr) => ($x);
    // `$x` 后面跟着至少一个 `$y,`
    ($x:expr, $($y:expr),+) => (
        // 对 `$x` 后面的 `$y` 们调用 `find_min!`
        std::cmp::min($x, find_min!($($y),+))
    )
}

fn main() {
    println!("{}", find_min!(1u32));
    println!("{}", find_min!(1u32 + 2 , 2u32));
    println!("{}", find_min!(5u32, 2u32 * 3, 4u32));
}
```

DRY (不写重复代码)

通过提取函数或测试集的公共部分，宏可以让你写出 DRY 的代码（DRY 是 Don't Repeat Yourself 的缩写，意思为 “不要写重复代码”）。这里给出一个例子，对 `Vec<T>` 实现并测试了关于 `+=`、`*=` 和 `-=` 等运算符。

```rust
use std::ops::{Add, Mul, Sub};

macro_rules! assert_equal_len {
    // `tt`（token tree，标记树）指示符表示运算符和标记。
    ($a:ident, $b: ident, $func:ident, $op:tt) => (
        assert!($a.len() == $b.len(),
                "{:?}: dimension mismatch: {:?} {:?} {:?}",
                stringify!($func),
                ($a.len(),),
                stringify!($op),
                ($b.len(),));
    )
}

macro_rules! op {
    ($func:ident, $bound:ident, $op:tt, $method:ident) => (
        fn $func<T: $bound<T, Output=T> + Copy>(xs: &mut Vec<T>, ys: &Vec<T>) {
            assert_equal_len!(xs, ys, $func, $op);

            for (x, y) in xs.iter_mut().zip(ys.iter()) {
                *x = $bound::$method(*x, *y);
                // *x = x.$method(*y);
            }
        }
    )
}

// 实现 `add_assign`、`mul_assign` 和 `sub_assign` 等函数。
op!(add_assign, Add, +=, add);
op!(mul_assign, Mul, *=, mul);
op!(sub_assign, Sub, -=, sub);

mod test {
    use std::iter;
    macro_rules! test {
        ($func: ident, $x:expr, $y:expr, $z:expr) => {
            #[test]
            fn $func() {
                for size in 0usize..10 {
                    let mut x: Vec<_> = iter::repeat($x).take(size).collect();
                    let y: Vec<_> = iter::repeat($y).take(size).collect();
                    let z: Vec<_> = iter::repeat($z).take(size).collect();

                    super::$func(&mut x, &y);

                    assert_eq!(x, z);
                }
            }
        }
    }

    // 测试 `add_assign`、`mul_assign` 和 `sub_assign`
    test!(add_assign, 1u32, 2u32, 3u32);
    test!(mul_assign, 2u32, 3u32, 6u32);
    test!(sub_assign, 3u32, 2u32, 1u32);
}
```

```
$ rustc --test dry.rs && ./dry
running 3 tests
test test::mul_assign ... ok
test test::add_assign ... ok
test test::sub_assign ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

DSL（领域专用语言）

DSL 是 Rust 的宏中集成的微型 “语言”。这种语言是完全合法的，因为宏系统会把它转换成普通的 Rust 语法树，它只不过看起来像是另一种语言而已。这就允许你为一些特定功能创造一套简洁直观的语法（当然是有限制的）。

比如说我想要定义一套小的计算器 API，可以传给它表达式，它会把结果打印到控制台上。

```
macro_rules! calculate {
    (eval $e:expr) => {{
        {
            let val: usize = $e; // 强制类型为整型
            println!("{} = {}", stringify!{$e}, val);
        }
    }};
}

fn main() {
    calculate! {
        eval 1 + 2 // 看到了吧，`eval` 可并不是 Rust 的关键字！
    }

    calculate! {
        eval (1 + 2) * (3 / 4)
    }
}
```

输出:

```
1 + 2 = 3
(1 + 2) * (3 / 4) = 0
```

这个例子非常简单，但是已经有很多利用宏开发的复杂接口了，比如 `lazy_static` 和 `clap`。

可变参数接口

可变参数接口可以接受任意数目的参数。比如说 `println` 就可以，其参数的数目是由格式化字符串指定的。

我们可以把之前的 `calculate!` 宏改写成可变参数接口：

```
macro_rules! calculate {
    // 单个 `eval` 的模式
    (eval $e:expr) => {{
        {
            let val: usize = $e; // Force types to be integers
            println!("{} = {}", stringify!{$e}, val);
        }
    }};

    // 递归地拆解多重的 `eval`
    (eval $e:expr, $(eval $es:expr),+) => {{
        calculate! { eval $e }
        calculate! { $(eval $es),+ }
    }};
}

fn main() {
    calculate! { // 妈妈快看，可变参数的 `calculate!`！
        eval 1 + 2,
        eval 3 + 4,
        eval (2 * 3) + 1
    }
}
```

输出：

```
1 + 2 = 3
3 + 4 = 7
(2 * 3) + 1 = 7
```

错误处理

错误处理（error handling）是处理可能发生的失败情况的过程。例如读取一个文件时失败了，如果继续使用这个**无效的**输入，那显然是有问题的。注意到并且显式地处理这种错误可以避免程序的其他部分产生潜在的问题。

在 Rust 中有多种处理错误的方式，在接下来的小节中会一一介绍。它们多少有些区别，使用场景也不尽相同。总的来说：

- 显式的 `panic` 主要用于测试，以及处理不可恢复的错误。在原型开发中这很有用，比如 用来测试还没有实现的函数，不过这时使用 `unimplemented` 更能表达意图。另外在 测试中，`panic` 是一种显式地失败（fail）的好方法。
- `Option` 类型是为了值是可选的、或者缺少值并不是错误的情况准备的。比如说寻找 父目录时， `/` 和 `C:` 这样的目录就没有父目录，这应当并不是一个错误。当处理 `Option` 时，`unwrap` 可用于原型开发，也可以用于能够确定 `Option` 中一定有值 的情形。然而 `expect` 更有用，因为它允许你指定一条错误信息，以免万一还是出现 了错误。
- 当错误有可能发生，且应当由调用者处理时，使用 `Result`。你也可以 `unwrap` 然后 使用 `expect`，但是除了在测试或者原型开发中，请不要这样做。

有关错误处理的更多内容，可参考官方文档的错误处理的章节。

panic

我们将要看到的最简单的错误处理机制就是 `panic`。它会打印一个错误消息，开始回退（unwind）任务，且通常会退出程序。这里我们显式地在错误条件下调用 `panic`：

```
fn give_princess(gift: &str) {
    // 公主讨厌蛇，所以如果公主表示厌恶的话我们要停止！
    if gift == "snake" { panic!("AAAaaaaa!!!!"); }

    println!("I love {}s!!!!!", gift);
}

fn main() {
    give_princess("teddy bear");
    give_princess("snake");
}
```

`Option` 和 `unwrap`

上个例子展示了如何主动地引入程序失败（program failure）。当公主收到蛇这件不合适的礼物时，我们就让程序 `panic`。但是，如果公主期待收到礼物，却没收到呢？这同样是一件糟糕的事情，所以我们要想办法来解决这个问题！

我们**可以**检查空字符串（`""`），就像处理蛇那样。但既然我们在用 Rust，不如让编译器辨别没有礼物的情况。

在标准库（`std`）中有个叫做 `Option<T>`（option 中文意思是 “选项”）的枚举类型，用于有 “不存在” 的可能性的情况。它表现为以下两个 “option”（选项）中的一个：

- `Some(T)`：找到一个属于 `T` 类型的元素
- `None`：找不到相应元素

这些选项可以通过 `match` 显式地处理，或使用 `unwrap` 隐式地处理。隐式处理要么返回 `Some` 内部的元素，要么就 `panic`。

请注意，手动使用 expect 方法自定义 `panic` 信息是可能的，但相比显式处理，`unwrap` 的输出仍显得不太有意义。在下面例子中，显式处理将举出更受控制的结果，同时如果需要的话，仍然可以使程序 `panic`。

```rust
// 平民（commoner）们见多识广，收到什么礼物都能应对。
// 所有礼物都显式地使用 `match` 来处理。
fn give_commoner(gift: Option<&str>) {
    // 指出每种情况下的做法。
    match gift {
        Some("snake") => println!("Yuck! I'm throwing that snake in a fire."),
        Some(inner)   => println!("{}? How nice.", inner),
        None          => println!("No gift? Oh well."),
    }
}

// 养在深闺人未识的公主见到蛇就会 `panic`（恐慌）。
// 这里所有的礼物都使用 `unwrap` 隐式地处理。
fn give_princess(gift: Option<&str>) {
    // `unwrap` 在接收到 `None` 时将返回 `panic`。
    let inside = gift.unwrap();
    if inside == "snake" { panic!("AAAaaaaa!!!!"); }

    println!("I love {}s!!!!!", inside);
}

fn main() {
    let food  = Some("chicken");
    let snake = Some("snake");
    let void  = None;

    give_commoner(food);
    give_commoner(snake);
    give_commoner(void);

    let bird = Some("robin");
    let nothing = None;

    give_princess(bird);
    give_princess(nothing);
}
```

使用 ? 解开 Option

你可以使用 `match` 语句来解开 `Option`，但使用 `?` 运算符通常会更容易。如果 `x` 是 `Option`，那么若 `x` 是 `Some`，对 `x?` 表达式求值将返回底层值，否则无论函数是否正在执行都将终止且返回 `None`。

```
fn next_birthday(current_age: Option<u8>) -> Option<String> {
    // 如果 `current_age` 是 `None`，这将返回 `None`。
    // 如果 `current_age` 是 `Some`，内部的 `u8` 将赋值给 `next_age`。
    let next_age: u8 = current_age?;
    Some(format!("Next year I will be {}", next_age))
}
```

你可以将多个 `?` 链接在一起，以使代码更具可读性。

```
struct Person {
    job: Option<Job>,
}

#[derive(Clone, Copy)]
struct Job {
    phone_number: Option<PhoneNumber>,
}

#[derive(Clone, Copy)]
struct PhoneNumber {
    area_code: Option<u8>,
    number: u32,
}

impl Person {

    // 获取此人的工作电话号码的区号（如果存在的话）。
    fn work_phone_area_code(&self) -> Option<u8> {
        // 没有`?`运算符的话，这将需要很多的嵌套的 `match` 语句。
        // 这将需要更多代码—尝试自己编写一下，看看哪个更容易。
        self.job?.phone_number?.area_code
    }
}

fn main() {
    let p = Person {
        job: Some(Job {
            phone_number: Some(PhoneNumber {
                area_code: Some(61),
                number: 439222222,
            }),
        }),
    };

    assert_eq!(p.work_phone_area_code(), Some(61));
}
```

组合算子：`map`

`match` 是处理 `Option` 的一个可用的方法，但你会发现大量使用它会很繁琐，特别是当操作只对一种输入是有效的时。这时，可以使用组合算子（combinator），以模块化的风格来管理控制流。

`Option` 有一个内置方法 `map()`，这个组合算子可用于 `Some -> Some` 和 `None -> None` 这样的简单映射。多个不同的 `map()` 调用可以串起来，这样更加灵活。

在下面例子中，`process()` 轻松取代了前面的所有函数，且更加紧凑。

```rust
#![allow(dead_code)]

#[derive(Debug)] enum Food { Apple, Carrot, Potato }

#[derive(Debug)] struct Peeled(Food);
#[derive(Debug)] struct Chopped(Food);
#[derive(Debug)] struct Cooked(Food);

// 削皮。如果没有食物，就返回 `None`。否则返回削好皮的食物。
fn peel(food: Option<Food>) -> Option<Peeled> {
    match food {
        Some(food) => Some(Peeled(food)),
        None       => None,
    }
}

// 切食物。如果没有食物，就返回 `None`。否则返回切好的食物。
fn chop(peeled: Option<Peeled>) -> Option<Chopped> {
    match peeled {
        Some(Peeled(food)) => Some(Chopped(food)),
        None               => None,
    }
}

// 烹饪食物。这里，我们使用 `map()` 来替代 `match` 以处理各种情况。
fn cook(chopped: Option<Chopped>) -> Option<Cooked> {
    chopped.map(|Chopped(food)| Cooked(food))
}

// 这个函数会完成削皮切块烹饪一条龙。我们把 `map()` 串起来，以简化代码。
fn process(food: Option<Food>) -> Option<Cooked> {
    food.map(|f| Peeled(f))
        .map(|Peeled(f)| Chopped(f))
        .map(|Chopped(f)| Cooked(f))
}

// 在尝试吃食物之前确认食物是否存在是非常重要的！
fn eat(food: Option<Cooked>) {
    match food {
        Some(food) => println!("Mmm. I love {:?}", food),
        None       => println!("Oh no! It wasn't edible."),
    }
}

fn main() {
    let apple = Some(Food::Apple);
    let carrot = Some(Food::Carrot);
    let potato = None;

    let cooked_apple = cook(chop(peel(apple)));
    let cooked_carrot = cook(chop(peel(carrot)));

    // 现在让我们试试看起来更简单的 `process()`。
    let cooked_potato = process(potato);

    eat(cooked_apple);
    eat(cooked_carrot);
    eat(cooked_potato);
```

```
}
```

参见：

闭包, `Option`, 和 `Option::map()`

组合算子：and_then

`map()` 以链式调用的方式来简化 `match` 语句。然而，如果以返回类型是 `Option<T>` 的函数作为 `map()` 的参数，会导致出现嵌套形式 `Option<Option<T>>`。这样多层串联调用就会变得混乱。所以有必要引入 `and_then()`，在某些语言中它叫做 flatmap。

`and_then()` 使用被 `Option` 包裹的值来调用其输入函数并返回结果。 如果 `Option` 是 `None`，那么它返回 `None`。

在下面例子中，`cookable_v2()` 会产生一个 `Option<Food>`。如果在这里使用 `map()` 而不是 `and_then()` 将会得到 `Option<Option<Food>>`，这对 `eat()` 来说是一个无效类型。

```rust
#![allow(dead_code)]

#[derive(Debug)] enum Food { CordonBleu, Steak, Sushi }
#[derive(Debug)] enum Day { Monday, Tuesday, Wednesday }

// 我们没有制作寿司所需的原材料（ingredient）（有其他的原材料）。
fn have_ingredients(food: Food) -> Option<Food> {
    match food {
        Food::Sushi => None,
        _           => Some(food),
    }
}

// 我们拥有全部食物的食谱，除了法国蓝带猪排（Cordon Bleu）的。
fn have_recipe(food: Food) -> Option<Food> {
    match food {
        Food::CordonBleu => None,
        _                => Some(food),
    }
}


// 要做一份好菜，我们需要原材料和食谱。
// 我们可以借助一系列 `match` 来表达这个逻辑：
fn cookable_v1(food: Food) -> Option<Food> {
    match have_ingredients(food) {
        None       => None,
        Some(food) => match have_recipe(food) {
            None       => None,
            Some(food) => Some(food),
        },
    }
}

// 也可以使用 `and_then()` 把上面的逻辑改写得更紧凑：
fn cookable_v2(food: Food) -> Option<Food> {
    have_ingredients(food).and_then(have_recipe)
}

fn eat(food: Food, day: Day) {
    match cookable_v2(food) {
        Some(food) => println!("Yay! On {:?} we get to eat {:?}.", day, food),
        None       => println!("Oh no. We don't get to eat on {:?}?", day),
    }
}

fn main() {
    let (cordon_bleu, steak, sushi) = (Food::CordonBleu, Food::Steak, Food::Sushi);

    eat(cordon_bleu, Day::Monday);
    eat(steak, Day::Tuesday);
    eat(sushi, Day::Wednesday);
}
```

参见：

闭包，`Option::map()`，和 `Option::and_then()`

结果 `Result`

`Result` 是 `Option` 类型的更丰富的版本，描述的是可能的**错误**而不是可能的**不存在**。

也就是说， `Result<T, E>` 可以有两个结果的其中一个：

- `Ok<T>`：找到 `T` 元素
- `Err<E>`：找到 `E` 元素， `E` 即表示错误的类型。

按照约定，预期结果是 “Ok”，而意外结果是 “Err”。

`Result` 有很多类似 `Option` 的方法。例如 `unwrap()`，它要么举出元素 `T`，要么就 `panic`。对于事件的处理， `Result` 和 `Option` 有很多相同的组合算子。

在使用 Rust 时，你可能会遇到返回 `Result` 类型的方法，例如 `parse()` 方法。它并不是总能把字符串解析成指定的类型，所以 `parse()` 返回一个 `Result` 表示可能的失败。

我们来看看当 `parse()` 字符串成功和失败时会发生什么：

```
fn multiply(first_number_str: &str, second_number_str: &str) -> i32 {
    // 我们试着用 `unwrap()` 把数字放出来。它会咬我们一口吗？
    let first_number = first_number_str.parse::<i32>().unwrap();
    let second_number = second_number_str.parse::<i32>().unwrap();
    first_number * second_number
}

fn main() {
    let twenty = multiply("10", "2");
    println!("double is {}", twenty);

    let tt = multiply("t", "2");
    println!("double is {}", tt);
}
```

在失败的情况下， `parse()` 产生一个错误，留给 `unwrap()` 来解包并产生 `panic`。另外，`panic` 会退出我们的程序，并提供一个让人很不爽的错误消息。

为了改善错误消息的质量，我们应该更具体地了解返回类型并考虑显式地处理错误。

`Result` 的 `map`

上一节的 `multiply` 函数的 panic 设计不是健壮的（robust）。一般地，我们希望把错误返回给调用者，这样它可以决定回应错误的正确方式。

首先，我们需要了解需要处理的错误类型是什么。为了确定 `Err` 的类型，我们可以用 `parse()` 来试验。Rust 已经为 `i32` 类型使用 `FromStr` trait 实现了 `parse()`。结果表明，这里的 `Err` 类型被指定为 `ParseIntError`。

译注：原文没有具体讲如何确定 `Err` 的类型。由于目前用于获取类型的函数仍然是不 稳定的，我们可以用间接的方法。使用下面的代码：

```
fn main () {
    let i: () = "t".parse::<i32>();
}
```

由于不可能把 `Result` 类型赋给单元类型变量 `i`，编译器会提示我们：

```
note: expected type `()`
         found type `std::result::Result<i32, std::num::ParseIntError>`
```

这样就知道了 `parse<i32>` 函数的返回类型详情。

在下面的例子中，使用简单的 `match` 语句导致了更加繁琐的代码。

```rust
use std::num::ParseIntError;

// 修改了上一节中的返回类型，现在使用模式匹配而不是 `unwrap()`。
fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseI
    match first_number_str.parse::<i32>() {
        Ok(first_number)  => {
            match second_number_str.parse::<i32>() {
                Ok(second_number)  => {
                    Ok(first_number * second_number)
                },
                Err(e) => Err(e),
            }
        },
        Err(e) => Err(e),
    }
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    // 这种情形下仍然会给出正确的答案。
    let twenty = multiply("10", "2");
    print(twenty);

    // 这种情况下就会提供一条更有用的错误信息。
    let tt = multiply("t", "2");
    print(tt);
}
```

幸运的是，`Option` 的 `map`、`and_then`、以及很多其他组合算子也为 `Result` 实现了。官方文档的 `Result` 一节包含完整的方法列表。

```rust
use std::num::ParseIntError;

// 就像 `Option` 那样，我们可以使用 `map()` 之类的组合算子。
// 除去写法外，这个函数与上面那个完全一致，它的作用是：
// 如果值是合法的，计算其乘积，否则返回错误。
fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseI
    first_number_str.parse::<i32>().and_then(|first_number| {
        second_number_str.parse::<i32>().map(|second_number| first_number * second
    })
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    // 这种情况下仍然会给出正确的答案。
    let twenty = multiply("10", "2");
    print(twenty);

    // 这种情况下就会提供一条更有用的错误信息。
    let tt = multiply("t", "2");
    print(tt);
}
```

给 `Result` 取别名

当我们要重用某个 `Result` 类型时，该怎么办呢？回忆一下，Rust 允许我们创建别名。若某个 `Result` 有可能被重用，我们可以方便地给它取一个别名。

在模块的层面上创建别名特别有帮助。同一模块中的错误常常会有相同的 `Err` 类型，所以单个别名就能简便地定义**所有**相关的 `Result`。这太有用了，以至于标准库也提供了一个别名：`io::Result`！

下面给出一个简短的示例来展示语法：

```
use std::num::ParseIntError;

// 为带有错误类型 `ParseIntError` 的 `Result` 定义一个泛型别名。
type AliasedResult<T> = Result<T, ParseIntError>;

// 使用上面定义过的别名来表示上一节中的 `Result<i32,ParseIntError>` 类型。
fn multiply(first_number_str: &str, second_number_str: &str) -> AliasedResult<i32>
    first_number_str.parse::<i32>().and_then(|first_number| {
        second_number_str.parse::<i32>().map(|second_number| first_number * second
    })
}

// 在这里使用别名又让我们节省了一些代码量。
fn print(result: AliasedResult<i32>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(multiply("10", "2"));
    print(multiply("t", "2"));
}
```

参见：

`io::Result`

提前返回

在上一个例子中，我们显式地使用组合算子处理了错误。另一种处理错误的方式是使用 `match` 语句和**提前返回**（early return）的结合。

这也就是说，如果发生错误，我们可以停止函数的执行然后返回错误。对有些人来说，这样的代码更好写，更易读。这次我们使用提前返回改写之前的例子：

```
use std::num::ParseIntError;

fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseI
    let first_number = match first_number_str.parse::<i32>() {
        Ok(first_number)  => first_number,
        Err(e) => return Err(e),
    };

    let second_number = match second_number_str.parse::<i32>() {
        Ok(second_number)  => second_number,
        Err(e) => return Err(e),
    };

    Ok(first_number * second_number)
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(multiply("10", "2"));
    print(multiply("t", "2"));
}
```

到此为止，我们已经学会了如何使用组合算子和提前返回显式地处理错误。我们一般是想要避免 panic 的，但显式地处理所有错误确实显得过于繁琐。

在下一部分，我们将看到，当只是需要 `unwrap` 并且不产生 `panic` 时，可以使用 `?` 来达到同样的效果。

引入 `?`

有时我们只是想 `unwrap` 且避免产生 `panic`。到现在为止，对 `unwrap` 的错误处理都在强迫我们一层层地嵌套，然而我们只是想把里面的变量拿出来。`?` 正是为这种情况准备的。

当找到一个 `Err` 时，可以采取两种行动：

1. `panic!`，不过我们已经决定要尽可能避免 panic 了。
2. 返回它，因为 `Err` 就意味着它已经不能被处理了。

`?` **几乎**[1] 就等于一个会返回 `Err` 而不是 `panic` 的 `unwrap`。我们来看看怎样简化之前使用组合算子的例子：

```
use std::num::ParseIntError;

fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseI
    let first_number = first_number_str.parse::<i32>()?;
    let second_number = second_number_str.parse::<i32>()?;

    Ok(first_number * second_number)
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(multiply("10", "2"));
    print(multiply("t", "2"));
}
```

`try!` 宏

在 `?` 出现以前，相同的功能是使用 `try!` 宏完成的。现在我们推荐使用 `?` 运算符，但是在老代码中仍然会看到 `try!`。如果使用 `try!` 的话，上一个例子中的 `multiply` 函数看起来会像是这样：

```rust
use std::num::ParseIntError;

fn multiply(first_number_str: &str, second_number_str: &str) -> Result<i32, ParseI
    let first_number = try!(first_number_str.parse::<i32>());
    let second_number = try!(second_number_str.parse::<i32>());

    Ok(first_number * second_number)
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(multiply("10", "2"));
    print(multiply("t", "2"));
}
```

[1] 更多细节请看 `?` 的更多用法。

处理多种错误类型

前面出现的例子都是很方便的情况；都是 `Result` 和其他 `Result` 交互，还有 `Option` 和其他 `Option` 交互。

有时 `Option` 需要和 `Result` 进行交互，或是 `Result<T, Error1>` 需要和 `Result<T, Error2>` 进行交互。在这类情况下，我们想要以一种方式来管理不同的错误类型，使得它们可组合且易于交互。

在下面代码中，`unwrap` 的两个实例生成了不同的错误类型。`Vec::first` 返回一个 `Option`，而 `parse::<i32>` 返回一个 `Result<i32, ParseIntError>`：

```
fn double_first(vec: Vec<&str>) -> i32 {
    let first = vec.first().unwrap(); // 生成错误 1
    2 * first.parse::<i32>().unwrap() // 生成错误 2
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    println!("The first doubled is {}", double_first(numbers));

    println!("The first doubled is {}", double_first(empty));
    // 错误1：输入 vector 为空

    println!("The first doubled is {}", double_first(strings));
    // 错误2：此元素不能解析成数字
}
```

在下面几节中，我们会看到处理这类问题的几种策略。

从 `Option` 中取出 `Result`

处理混合错误类型的最基本的手段就是让它们互相包含。

```
use std::num::ParseIntError;

fn double_first(vec: Vec<&str>) -> Option<Result<i32, ParseIntError>> {
    vec.first().map(|first| {
        first.parse::<i32>().map(|n| 2 * n)
    })
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    println!("The first doubled is {:?}", double_first(numbers));

    println!("The first doubled is {:?}", double_first(empty));
    // Error 1: the input vector is empty

    println!("The first doubled is {:?}", double_first(strings));
    // Error 2: the element doesn't parse to a number
}
```

有时候我们不想再处理错误（比如使用 `?` 的时候），但如果 `Option` 是 `None` 则继续处理错误。一些组合算子可以让我们轻松地交换 `Result` 和 `Option`。

```
use std::num::ParseIntError;

fn double_first(vec: Vec<&str>) -> Result<Option<i32>, ParseIntError> {
    let opt = vec.first().map(|first| {
        first.parse::<i32>().map(|n| 2 * n)
    });

    opt.map_or(Ok(None), |r| r.map(Some))
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    println!("The first doubled is {:?}", double_first(numbers));
    println!("The first doubled is {:?}", double_first(empty));
    println!("The first doubled is {:?}", double_first(strings));
}
```

定义一个错误类型

有时候把所有不同的错误都视为一种错误类型会简化代码。我们将用一个自定义错误类型来演示这一点。

Rust 允许我们定义自己的错误类型。一般来说，一个 “好的” 错误类型应当：

- 用同一个类型代表了多种错误
- 向用户提供了清楚的错误信息
- 能够容易地与其他类型比较
  - 好的例子：`Err(EmptyVec)`
  - 坏的例子：`Err("Please use a vector with at least one element".to_owned())`
- 能够容纳错误的具体信息
  - 好的例子：`Err(BadChar(c, position))`
  - 坏的例子：`Err("+ cannot be used here".to_owned())`
- 能够与其他错误很好地整合

```rust
use std::error;
use std::fmt;

type Result<T> = std::result::Result<T, DoubleError>;

#[derive(Debug, Clone)]
// 定义我们的错误类型，这种类型可以根据错误处理的实际情况定制。
// 我们可以完全自定义错误类型，也可以在类型中完全采用底层的错误实现，
// 也可以介于二者之间。
struct DoubleError;

// 错误的生成与它如何显示是完全没关系的。没有必要担心复杂的逻辑会导致混乱的显示。
//
// 注意我们没有储存关于错误的任何额外信息，也就是说，如果不修改我们的错误类型定义，
// 就无法指明是哪个字符串解析失败了。
impl fmt::Display for DoubleError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "invalid first item to double")
    }
}

// 为 `DoubleError` 实现 `Error` trait，这样其他错误可以包裹这个错误类型。
impl error::Error for DoubleError {
    fn source(&self) -> Option<&(dyn error::Error + 'static)> {
        // 泛型错误，没有记录其内部原因。
        None
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    vec.first()
       // 把错误换成我们的新类型。
       .ok_or(DoubleError)
       .and_then(|s| {
           s.parse::<i32>()
               // 这里也换成新类型。
               .map_err(|_| DoubleError)
               .map(|i| 2 * i)
       })
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

把错误“装箱”

如果又想写简单的代码，又想保存原始错误信息，一个方法是把它们装箱（`Box`）。这样做的坏处就是，被包装的错误类型只能在运行时了解，而不能被静态地判别。

对任何实现了 `Error` trait 的类型，标准库的 `Box` 通过 `From` 为它们提供了到 `Box<Error>` 的转换。

```rust
use std::error;
use std::fmt;

// 为 `Box<error::Error>` 取别名。
type Result<T> = std::result::Result<T, Box<dyn error::Error>>;

#[derive(Debug, Clone)]
struct EmptyVec;

impl fmt::Display for EmptyVec {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "invalid first item to double")
    }
}

impl error::Error for EmptyVec {
    fn description(&self) -> &str {
        "invalid first item to double"
    }

    fn cause(&self) -> Option<&dyn error::Error> {
        // 泛型错误。没有记录其内部原因。
        None
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    vec.first()
       .ok_or_else(|| EmptyVec.into())  // 装箱
       .and_then(|s| {
           s.parse::<i32>()
               .map_err(|e| e.into())  // 装箱
               .map(|i| 2 * i)
       })
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

参见：

动态分发 and `Error` trait

? 的其他用法

注意在上一个例子中，我们调用 `parse` 后总是立即将错误从标准库的错误 `map` （映射）到装箱错误。

```
.and_then(|s| s.parse::<i32>()
    .map_err(|e| e.into()))
```

因为这个操作很简单常见，如果有省略写法就好了。遗憾的是 `and_then` 不够灵活，所以实现不了这样的写法。不过，我们可以使用 `?` 来代替它。

`?` 之前被解释为要么 `unwrap`，要么 `return Err(err)`，这只是在大多数情况下是正确的。`?` 实际上是指 `unwrap` 或 `return Err(From::from(err))`。由于 `From::from` 是不同类型之间的转换工具，也就是说，如果在错误可转换成返回类型地方使用 `?`，它将自动转换成返回类型。

我们在这里使用 `?` 重写之前的例子。重写后，只要为我们的错误类型实现 `From::from`，就可以不再使用 `map_err`。

```rust
use std::error;
use std::fmt;

// 为 `Box<error::Error>` 取别名。
type Result<T> = std::result::Result<T, Box<dyn error::Error>>;

#[derive(Debug)]
struct EmptyVec;

impl fmt::Display for EmptyVec {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "invalid first item to double")
    }
}

impl error::Error for EmptyVec {}

// 这里的结构和之前一样，但是这次没有把所有的 `Result` 和 `Option` 串起来，
// 而是使用 `?` 立即得到内部值。
fn double_first(vec: Vec<&str>) -> Result<i32> {
    let first = vec.first().ok_or(EmptyVec)?;
    let parsed = first.parse::<i32>()?;
    Ok(2 * parsed)
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

这段代码已经相当清晰了。与原来的 `panic` 相比，除了返回类型是 `Result` 之外，它就像是把所有的 `unwrap` 调用都换成 `?` 一样。因此必须在顶层解构它们。

参见：

`From::from` 和 `?`

包裹错误

把错误装箱这种做法也可以改成把它包裹到你自己的错误类型中。

```rust
use std::error;
use std::num::ParseIntError;
use std::fmt;

type Result<T> = std::result::Result<T, DoubleError>;

#[derive(Debug)]
enum DoubleError {
    EmptyVec,
    // 在这个错误类型中，我们采用 `parse` 的错误类型中 `Err` 部分的实现。
    // 若想提供更多信息，则该类型中还需要加入更多数据。
    Parse(ParseIntError),
}

impl fmt::Display for DoubleError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            DoubleError::EmptyVec =>
                write!(f, "please use a vector with at least one element"),
            // 这是一个封装（wrapper），它采用内部各类型对 `fmt` 的实现。
            DoubleError::Parse(ref e) => e.fmt(f),
        }
    }
}

impl error::Error for DoubleError {
    fn source(&self) -> Option<&(dyn error::Error + 'static)> {
        match *self {
            DoubleError::EmptyVec => None,
            // 原因采取内部对错误类型的实现。它隐式地转换成了 trait 对象 `&error:
            // 这可以工作，因为内部的类型已经实现了 `Error` trait。
            DoubleError::Parse(ref e) => Some(e),
        }
    }
}

// 实现从 `ParseIntError` 到 `DoubleError` 的转换。
// 在使用 `?` 时，或者一个 `ParseIntError` 需要转换成 `DoubleError` 时，它会被自动
impl From<ParseIntError> for DoubleError {
    fn from(err: ParseIntError) -> DoubleError {
        DoubleError::Parse(err)
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    let first = vec.first().ok_or(DoubleError::EmptyVec)?;
    let parsed = first.parse::<i32>()?;

    Ok(2 * parsed)
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}
```

```
fn main() {
    let numbers = vec!["42", "93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

这种做法会在错误处理中增加一些模板化的代码，而且也不是所有的应用都需要这样做。一些库可以帮你处理模板化代码的问题。

See also:

`From::from` and `枚举类型`

遍历 `Result`

`Iter::map` 操作可能失败，比如：

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let numbers: Vec<_> = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .collect();
    println!("Results: {:?}", numbers);
}
```

我们来看一些处理这种问题的策略：

使用 `filter_map()` 忽略失败的项

`filter_map` 会调用一个函数，过滤掉为 `None` 的所有结果。

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let numbers: Vec<_> = strings
        .into_iter()
        .filter_map(|s| s.parse::<i32>().ok())
        .collect();
    println!("Results: {:?}", numbers);
}
```

使用 `collect()` 使整个操作失败

`Result` 实现了 `FromIter`，因此结果的向量（`Vec<Result<T, E>>`）可以被转换成结果包裹着向量（`Result<Vec<T>, E>`）。一旦找到一个 `Result::Err`，遍历就被终止。

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let numbers: Result<Vec<_>, _> = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .collect();
    println!("Results: {:?}", numbers);
}
```

同样的技巧可以对 `Option` 使用。

使用 `Partition()` 收集所有合法的值与错误

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let (numbers, errors): (Vec<_>, Vec<_>) = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .partition(Result::is_ok);
    println!("Numbers: {:?}", numbers);
    println!("Errors: {:?}", errors);
}
```

当你看着这些结果时，你会发现所有东西还在 `Result` 中保存着。要取出它们，需要一些模板化的代码。

```
fn main() {
    let strings = vec!["tofu", "93", "18"];
    let (numbers, errors): (Vec<_>, Vec<_>) = strings
        .into_iter()
        .map(|s| s.parse::<i32>())
        .partition(Result::is_ok);
    let numbers: Vec<_> = numbers.into_iter().map(Result::unwrap).collect();
    let errors: Vec<_> = errors.into_iter().map(Result::unwrap_err).collect();
    println!("Numbers: {:?}", numbers);
    println!("Errors: {:?}", errors);
}
```

标准库类型

标准库提供了很多自定义类型，在**原生类型**基础上进行了大量扩充。这是部分自定义类型：

- 可增长的 `String`（字符串），如: `"hello world"`
- 可增长的向量（vector）：`[1, 2, 3]`
- 选项类型（optional types）：`Option<i32>`
- 错误处理类型（error handling types）：`Result<i32, i32>`
- 堆分配的指针（heap allocated pointers）：`Box<i32>`

参见：

原生类型 和 标准库

箱子、栈和堆

在 Rust 中，所有值默认都是栈分配的。通过创建 `Box<T>`，可以把值**装箱**（boxed）来使它在堆上分配。箱子（box，即 `Box<T>` 类型的实例）是一个智能指针，指向堆分配的 `T` 类型的值。当箱子离开作用域时，它的析构函数会被调用，内部的对象会被销毁，堆上分配的内存也会被释放。

被装箱的值可以使用 `*` 运算符进行解引用；这会移除掉一层装箱。

```rust
use std::mem;

#[allow(dead_code)]
#[derive(Debug, Clone, Copy)]
struct Point {
    x: f64,
    y: f64,
}

#[allow(dead_code)]
struct Rectangle {
    p1: Point,
    p2: Point,
}

fn origin() -> Point {
    Point { x: 0.0, y: 0.0 }
}

fn boxed_origin() -> Box<Point> {
    // 在堆上分配这个点（point），并返回一个指向它的指针
    Box::new(Point { x: 0.0, y: 0.0 })
}

fn main() {
    //  (所有的类型标注都不是必需的)
    // 栈分配的变量
    let point: Point = origin();
    let rectangle: Rectangle = Rectangle {
        p1: origin(),
        p2: Point { x: 3.0, y: 4.0 }
    };

    // 堆分配的 rectangle（矩形）
    let boxed_rectangle: Box<Rectangle> = Box::new(Rectangle {
        p1: origin(),
        p2: origin()
    });

    // 函数的输出可以装箱
    let boxed_point: Box<Point> = Box::new(origin());

    // 两层装箱
    let box_in_a_box: Box<Box<Point>> = Box::new(boxed_origin());

    println!("Point occupies {} bytes in the stack",
             mem::size_of_val(&point));
    println!("Rectangle occupies {} bytes in the stack",
             mem::size_of_val(&rectangle));

    // box 的宽度就是指针宽度
    println!("Boxed point occupies {} bytes in the stack",
             mem::size_of_val(&boxed_point));
    println!("Boxed rectangle occupies {} bytes in the stack",
             mem::size_of_val(&boxed_rectangle));
    println!("Boxed box occupies {} bytes in the stack",
             mem::size_of_val(&box_in_a_box));
```

```
    // 将包含在 `boxed_point` 中的数据复制到 `unboxed_point`
    let unboxed_point: Point = *boxed_point;
    println!("Unboxed point occupies {} bytes in the stack",
             mem::size_of_val(&unboxed_point));
}
```

动态数组 vector

vector 是大小可变的数组。和 slice（切片）类似，它们的大小在编译时是未知的，但它们可以随时扩大或缩小。一个 vector 使用 3 个词来表示：一个指向数据的指针，vector 的长度，还有它的容量。此容量指明了要为这个 vector 保留多少内存。vector 的长度只要小于该容量，就可以随意增长；当需要超过这个阈值时，会给 vector 重新分配一段更大的容量。

```rust
fn main() {
    // 迭代器可以被收集到 vector 中
    let collected_iterator: Vec<i32> = (0..10).collect();
    println!("Collected (0..10) into: {:?}", collected_iterator);

    // `vec!` 宏可用来初始化一个 vector
    let mut xs = vec![1i32, 2, 3];
    println!("Initial vector: {:?}", xs);

    // 在 vector 的尾部插入一个新的元素
    println!("Push 4 into the vector");
    xs.push(4);
    println!("Vector: {:?}", xs);

    // 报错！不可变 vector 不可增长
    collected_iterator.push(0);
    // 改正 ^ 将此行注释掉

    // `len` 方法获得一个 vector 的当前大小
    println!("Vector size: {}", xs.len());

    // 下标使用中括号表示（从 0 开始）
    println!("Second element: {}", xs[1]);

    // `pop` 移除 vector 的最后一个元素并将它返回
    println!("Pop last element: {:?}", xs.pop());

    // 超出下标范围将抛出一个 panic
    println!("Fourth element: {}", xs[3]);
    // 改正 ^ 注释掉此行

    // 迭代一个 `Vector` 很容易
    println!("Contents of xs:");
    for x in xs.iter() {
        println!("> {}", x);
    }

    // 可以在迭代 `Vector` 的同时，使用独立变量（`i`）来记录迭代次数
    for (i, x) in xs.iter().enumerate() {
        println!("In position {} we have value {}", i, x);
    }

    // 多亏了 `iter_mut`，可变的 `Vector` 在迭代的同时，其中每个值都能被修改
    for x in xs.iter_mut() {
        *x *= 3;
    }
    println!("Updated vector: {:?}", xs);
}
```

更多 `Vec` 方法可以在 std::vec 模块中找到。

字符串

Rust 中有两种字符串类型：`String` 和 `&str`。

`String` 被存储为由字节组成的 vector（`Vec<u8>`），但保证了它一定是一个有效的 UTF-8 序列。`String` 是堆分配的，可增长的，且不是零结尾的（null terminated）。

`&str` 是一个总是指向有效 UTF-8 序列的切片（`&[u8]`），并可用来查看 `String` 的内容，就如同 `&[T]` 是 `Vec<T>` 的全部或部分引用。

```
fn main() {
    // （所有的类型标注都不是必需的）
    // 一个对只读内存中分配的字符串的引用
    let pangram: &'static str = "the quick brown fox jumps over the lazy dog";
    println!("Pangram: {}", pangram);

    // 逆序迭代单词，这里并没有分配新的字符串
    println!("Words in reverse");
    for word in pangram.split_whitespace().rev() {
        println!("> {}", word);
    }

    // 复制字符到一个 vector，排序并移除重复值
    let mut chars: Vec<char> = pangram.chars().collect();
    chars.sort();
    chars.dedup();

    // 创建一个空的且可增长的 `String`
    let mut string = String::new();
    for c in chars {
        // 在字符串的尾部插入一个字符
        string.push(c);
        // 在字符串尾部插入一个字符串
        string.push_str(", ");
    }

    // 这个缩短的字符串是原字符串的一个切片，所以没有执行新的分配操作
    let chars_to_trim: &[char] = &[' ', ','];
    let trimmed_str: &str = string.trim_matches(chars_to_trim);
    println!("Used characters: {}", trimmed_str);

    // 堆分配一个字符串
    let alice = String::from("I like dogs");
    // 分配新内存并存储修改过的字符串
    let bob: String = alice.replace("dog", "cat");

    println!("Alice says: {}", alice);
    println!("Bob says: {}", bob);
}
```

更多 `str` / `String` 方法可以在 std::str 和 std::string 模块中找到。

字面量与转义字符

书写含有特殊字符的字符串字面量有很多种方法。它们都会产生类似的 `&str`，所以最好选择最方便的写法。类似地，字节串（byte string）字面量也有多种写法，它们都会产生 `&[u8; N]` 类型。

通常特殊字符是使用反斜杠字符 `\` 来转义的，这样你就可以在字符串中写入各种各样的字符，甚至是不可打印的字符以及你不知道如何输入的字符。如果你需要反斜杠字符，再用另一个反斜杠来转义它就可以，像这样：`\\`。

字面量中出现的字符串或字符定界符必须转义：`"\""`、`'\''`。

```
fn main() {
    // 通过转义，可以用十六进制值来表示字节。
    let byte_escape = "I'm writing \x52\x75\x73\x74!";
    println!("What are you doing\x3F (\\x3F means ?) {}", byte_escape);

    // 也可以使用 Unicode 码位表示。
    let unicode_codepoint = "\u{211D}";
    let character_name = "\"DOUBLE-STRUCK CAPITAL R\"";

    println!("Unicode character {} (U+211D) is called {}",
                unicode_codepoint, character_name );


    let long_string = "String literals
                        can span multiple lines.
                        The linebreak and indentation here ->\
                        <- can be escaped too!";
    println!("{}", long_string);
}
```

有时会有太多需要转义的字符，或者是直接原样写出会更便利。这时可以使用原始字符串（raw string）。

```
fn main() {
    let raw_str = r"Escapes don't work here: \x3F \u{211D}";
    println!("{}", raw_str);

    // 如果你要在原始字符串中写引号，请在两边加一对 #
    let quotes = r#"And then I said: "There is no escape!""#;
    println!("{}", quotes);

    // 如果字符串中需要写 "#，那就在定界符中使用更多的 #。
    // 可使用的 # 的数目没有限制。
    let longer_delimiter = r###"A string with "# in it. And even "##!"###;
    println!("{}", longer_delimiter);
}
```

想要非 UTF-8 字符串（记住，`&str` 和 `String` 都必须是合法的 UTF-8 序列），或者需要一个字节数组，其中大部分是文本？请使用字节串（byte string）！

```
use std::str;

fn main() {
    // 注意这并不是一个 &str
    let bytestring: &[u8; 20] = b"this is a bytestring";

    // 字节串没有实现 Display，所以它们的打印功能有些受限
    println!("A bytestring: {:?}", bytestring);

    // 字节串可以使用单字节的转义字符...
    let escaped = b"\x52\x75\x73\x74 as bytes";
    // ...但不能使用 Unicode 转义字符
    // let escaped = b"\u{211D} is not allowed";
    println!("Some escaped bytes: {:?}", escaped);


    // 原始字节串和原始字符串的写法一样
    let raw_bytestring = br"\u{211D} is not escaped here";
    println!("{:?}", raw_bytestring);

    // 把字节串转换为 &str 可能失败
    if let Ok(my_str) = str::from_utf8(raw_bytestring) {
        println!("And the same as text: '{}'", my_str);
    }

    let quotes = br#"You can also use "fancier" formatting, \
                    like with normal raw strings"#;

    // 字节串可以不使用 UTF-8 编码
    let shift_jis = b"\x82\xe6\x82\xa8\x82\xb1\x82"; // SHIFT-JIS 编码的 "ようこそ

    // 但这样的话它们就无法转换成 &str 了
    match str::from_utf8(shift_jis) {
        Ok(my_str) => println!("Conversion successful: '{}'", my_str),
        Err(e) => println!("Conversion failed: {:?}", e),
    };
}
```

若需要在编码间转换，请使用 encoding crate。

Rust 参考中的 Tokens 一章详细地列出了书写字符串字面量和转义字符的方法。

选项 Option

有时候想要捕捉到程序某部分的失败信息，而不是调用 `panic!`；这可使用 `Option` 枚举类型来实现。

`Option<T>` 有两个变量：

- `None`，表明失败或缺少值
- `Some(value)`，元组结构体，封装了一个 `T` 类型的值 `value`

```
// 不会 `panic!` 的整数除法。
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        // 失败表示成 `None` 取值
        None
    } else {
        // 结果 Result 被包装到 `Some` 取值中
        Some(dividend / divisor)
    }
}

// 此函数处理可能失败的除法
fn try_division(dividend: i32, divisor: i32) {
    // `Option` 值可以进行模式匹配，就和其他枚举类型一样
    match checked_division(dividend, divisor) {
        None => println!("{} / {} failed!", dividend, divisor),
        Some(quotient) => {
            println!("{} / {} = {}", dividend, divisor, quotient)
        },
    }
}

fn main() {
    try_division(4, 2);
    try_division(1, 0);

    // 绑定 `None` 到一个变量需要类型标注
    let none: Option<i32> = None;
    let _equivalent_none = None::<i32>;

    let optional_float = Some(0f32);

    // 解包 `Some` 将取出被包装的值。
    println!("{:?} unwraps to {:?}", optional_float, optional_float.unwrap());

    // 解包 `None` 将会引发 `panic!`。
    println!("{:?} unwraps to {:?}", none, none.unwrap());
}
```

结果 `Result`

我们已经看到 `Option` 枚举类型可以用作可能失败的函数的返回值，其中返回 `None` 可以表明失败。但是有时要强调**为什么**一个操作会失败。为做到这点，我们提供了 `Result` 枚举类型。

`Result<T, E>` 类型拥有两个取值：

- `Ok(value)` 表示操作成功，并包装操作返回的 `value`（`value` 拥有 `T` 类型）。
- `Err(why)`，表示操作失败，并包装 `why`，它（但愿）能够解释失败的原因（`why` 拥有 `E` 类型）。

```rust
mod checked {
    // 我们想要捕获的数学“错误”
    #[derive(Debug)]
    pub enum MathError {
        DivisionByZero,
        NegativeLogarithm,
        NegativeSquareRoot,
    }

    pub type MathResult = Result<f64, MathError>;

    pub fn div(x: f64, y: f64) -> MathResult {
        if y == 0.0 {
            // 此操作将会失败，那么（与其让程序崩溃）不如把失败的原因包装在
            // `Err` 中并返回
            Err(MathError::DivisionByZero)
        } else {
            // 此操作是有效的，返回包装在 `Ok` 中的结果
            Ok(x / y)
        }
    }

    pub fn sqrt(x: f64) -> MathResult {
        if x < 0.0 {
            Err(MathError::NegativeSquareRoot)
        } else {
            Ok(x.sqrt())
        }
    }

    pub fn ln(x: f64) -> MathResult {
        if x < 0.0 {
            Err(MathError::NegativeLogarithm)
        } else {
            Ok(x.ln())
        }
    }
}

// `op(x, y)` === `sqrt(ln(x / y))`
fn op(x: f64, y: f64) -> f64 {
    // 这是一个三层的 match 金字塔!
    match checked::div(x, y) {
        Err(why) => panic!("{:?}", why),
        Ok(ratio) => match checked::ln(ratio) {
            Err(why) => panic!("{:?}", why),
            Ok(ln) => match checked::sqrt(ln) {
                Err(why) => panic!("{:?}", why),
                Ok(sqrt) => sqrt,
            },
        },
    }
}

fn main() {
    // 这会失败吗?
```

```
    println!("{}", op(1.0, 10.0));
}
```

? 运算符

把 result 用 match 连接起来会显得很难看；幸运的是，`?` 运算符可以把这种逻辑变得干净漂亮。`?` 运算符用在返回值为 `Result` 的表达式后面，它等同于这样一个匹配表达式：其中 `Err(err)` 分支展开成提前返回的 `return Err(err)`，而 `Ok(ok)` 分支展开成 `ok` 表达式。

```
mod checked {
    #[derive(Debug)]
    enum MathError {
        DivisionByZero,
        NegativeLogarithm,
        NegativeSquareRoot,
    }

    type MathResult = Result<f64, MathError>;

    fn div(x: f64, y: f64) -> MathResult {
        if y == 0.0 {
            Err(MathError::DivisionByZero)
        } else {
            Ok(x / y)
        }
    }

    fn sqrt(x: f64) -> MathResult {
        if x < 0.0 {
            Err(MathError::NegativeSquareRoot)
        } else {
            Ok(x.sqrt())
        }
    }

    fn ln(x: f64) -> MathResult {
        if x < 0.0 {
            Err(MathError::NegativeLogarithm)
        } else {
            Ok(x.ln())
        }
    }

    // 中间函数
    fn op_(x: f64, y: f64) -> MathResult {
        // 如果 `div` “失败”了，那么返回 `DivisionByZero`
        let ratio = div(x, y)?;

        // 如果 `ln` “失败”了，那么返回 `NegativeLogarithm`
        let ln = ln(ratio)?;

        sqrt(ln)
    }

    pub fn op(x: f64, y: f64) {
        match op_(x, y) {
            Err(why) => panic!("{}",match why {
                MathError::NegativeLogarithm
                    => "logarithm of negative number",
                MathError::DivisionByZero
                    => "division by zero",
                MathError::NegativeSquareRoot
                    => "square root of negative number",
            }),
            Ok(value) => println!("{}", value),
        }
    }
```

```
}

fn main() {
    checked::op(1.0, 10.0);
}
```

记得查阅文档，里面有很多匹配/组合 `Result` 的方法。

panic!

`panic!` 宏可用于产生一个 panic （恐慌），并开始回退（unwind）它的栈。在回退栈的同时，运行时将会释放该线程所**拥有**的所有资源，这是通过调用线程中所有对象的析构函数完成的。

因为我们正在处理的程序只有一个线程，`panic!` 将会引发程序报告 panic 消息并退出。

```
// 整型除法（/）的重新实现
fn division(dividend: i32, divisor: i32) -> i32 {
    if divisor == 0 {
        // 除以 0 会引发 panic
        panic!("division by zero");
    } else {
        dividend / divisor
    }
}

// `main` 任务
fn main() {
    // 堆分配的整数
    let _x = Box::new(0i32);

    // 此操作将会引发一个任务失败
    division(3, 0);

    println!("This point won't be reached!");

    // `_x` 应当会在此处被销毁
}
```

可以看到，`panic!` 不会泄露内存：

```
$ rustc panic.rs && valgrind ./panic
==4401== Memcheck, a memory error detector
==4401== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4401== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==4401== Command: ./panic
==4401==
thread '<main>' panicked at 'division by zero', panic.rs:5
==4401==
==4401== HEAP SUMMARY:
==4401==     in use at exit: 0 bytes in 0 blocks
==4401==   total heap usage: 18 allocs, 18 frees, 1,648 bytes allocated
==4401==
==4401== All heap blocks were freed -- no leaks are possible
==4401==
==4401== For counts of detected and suppressed errors, rerun with: -v
==4401== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

散列表 HashMap

vector 通过整型下标来存储值，而 `HashMap`（散列表）通过键（key）来存储值。`HashMap` 的键可以是布尔型、整型、字符串，或任意实现了 `Eq` 和 `Hash` trait 的其他类型。在下一节将进一步介绍。

和 vector 类似，`HashMap` 也是可增长的，但 HashMap 在占据了多余空间时还可以缩小自己。可以使用 `HashMap::with_capacity(unit)` 创建具有一定初始容量的 HashMap，也可以使用 `HashMap::new()` 来获得一个带有默认初始容量的 HashMap（这是推荐方式）。

```rust
use std::collections::HashMap;

fn call(number: &str) -> &str {
    match number {
        "798-1364" => "We're sorry, the call cannot be completed as dialed.
            Please hang up and try again.",
        "645-7689" => "Hello, this is Mr. Awesome's Pizza. My name is Fred.
            What can I get for you today?",
        _ => "Hi! Who is this again?"
    }
}

fn main() {
    let mut contacts = HashMap::new();

    contacts.insert("Daniel", "798-1364");
    contacts.insert("Ashley", "645-7689");
    contacts.insert("Katie", "435-8291");
    contacts.insert("Robert", "956-1745");

    // 接受一个引用并返回 Option<&V>
    match contacts.get(&"Daniel") {
        Some(&number) => println!("Calling Daniel: {}", call(number)),
        _ => println!("Don't have Daniel's number."),
    }

    // 如果被插入的值为新内容，那么 `HashMap::insert()` 返回 `None`，
    // 否则返回 `Some(value)`
    contacts.insert("Daniel", "164-6743");

    match contacts.get(&"Ashley") {
        Some(&number) => println!("Calling Ashley: {}", call(number)),
        _ => println!("Don't have Ashley's number."),
    }

    contacts.remove(&("Ashley"));

    // `HashMap::iter()` 返回一个迭代器，该迭代器以任意顺序举出
    // (&'a key, &'a value) 对。
    for (contact, &number) in contacts.iter() {
        println!("Calling {}: {}", contact, call(number));
    }
}
```

想要了解更多关于散列（hash）与散列表（hash map）（有时也称作 hash table）的工作原理，可以查看 Wikipedia 的散列表词条。

更改或自定义关键字类型

任何实现了 `Eq` 和 `Hash` trait 的类型都可以充当 `HashMap` 的键。这包括：

- `bool` （当然这个用处不大，因为只有两个可能的键）
- `int`， `unit`，以及其他整数类型
- `String` 和 `&str` （友情提示：如果使用 `String` 作为键来创建 `HashMap`，则可以 将 `&str` 作为散列表的 `.get()` 方法的参数，以获取值）

注意到 `f32` 和 `f64` **没有**实现 `Hash`，这很大程度上是由于若使用浮点数作为散列表的键，浮点精度误差会很容易导致错误。

对于所有的集合类（collection class），如果它们包含的类型都分别实现了 `Eq` 和 `Hash`，那么这些集合类也就实现了 `Eq` 和 `Hash`。例如，若 `T` 实现了 `Hash`，则 `Vec<T>` 也实现了 `Hash`。

对自定义类型可以轻松地实现 `Eq` 和 `Hash`，只需加上一行代码：`#[derive(PartialEq, Eq, Hash)]`。

编译器将会完成余下的工作。如果你想控制更多的细节，你可以手动实现 `Eq` 和/或 `Hash`。本指南不包含实现 `Hash` 的细节内容。

为了试验 `HashMap` 中的 `struct`，让我们试着做一个非常简易的用户登录系统：

```rust
use std::collections::HashMap;

// Eq 要求你对此类型推导 PartiaEq。
#[derive(PartialEq, Eq, Hash)]
struct Account<'a>{
    username: &'a str,
    password: &'a str,
}

struct AccountInfo<'a>{
    name: &'a str,
    email: &'a str,
}

type Accounts<'a> = HashMap<Account<'a>, AccountInfo<'a>>;

fn try_logon<'a>(accounts: &Accounts<'a>,
        username: &'a str, password: &'a str){
    println!("Username: {}", username);
    println!("Password: {}", password);
    println!("Attempting logon...");

    let logon = Account {
        username: username,
        password: password,
    };

    match accounts.get(&logon) {
        Some(account_info) => {
            println!("Successful logon!");
            println!("Name: {}", account_info.name);
            println!("Email: {}", account_info.email);
        },
        _ => println!("Login failed!"),
    }
}

fn main(){
    let mut accounts: Accounts = HashMap::new();

    let account = Account {
        username: "j.everyman",
        password: "password123",
    };

    let account_info = AccountInfo {
        name: "John Everyman",
        email: "j.everyman@email.com",
    };

    accounts.insert(account, account_info);

    try_logon(&accounts, "j.everyman", "psasword123");

    try_logon(&accounts, "j.everyman", "password123");
}
```

散列集 HashSet

请把 `HashSet` 当成这样一个 `HashMap`：我们只关心其中的键而非值（`HashSet<T>` 实际上只是对 `HashMap<T, ()>` 的封装）。

你可能会问："这有什么意义呢？我完全可以将键存储到一个 `Vec` 中呀。"

`HashSet` 的独特之处在于，它保证了不会出现重复的元素。这是任何 set 集合类型（set collection）遵循的规定。`HashSet` 只是它的一个实现。（参见：`BTreeSet`）

如果插入的值已经存在于 `HashSet` 中（也就是，新值等于已存在的值，并且拥有相同的散列值），那么新值将会替换旧的值。

如果你不想要一样东西出现多于一次，或者你要判断一样东西是不是已经存在，这种做法就很有用了。

不过集合（set）可以做更多的事。

集合（set）拥有 4 种基本操作（下面的调用全部都返回一个迭代器）：

- `union`（并集）：获得两个集合中的所有元素（不含重复值）。
- `difference`（差集）：获取属于第一个集合而不属于第二集合的所有元素。
- `intersection`（交集）：获取同时属于两个集合的所有元素。
- `symmetric_difference`（对称差）：获取所有只属于其中一个集合，而不同时属于 两个集合的所有元素。

在下面的例子中尝试使用这些操作。

```rust
use std::collections::HashSet;

fn main() {
    let mut a: HashSet<i32> = vec!(1i32, 2, 3).into_iter().collect();
    let mut b: HashSet<i32> = vec!(2i32, 3, 4).into_iter().collect();

    assert!(a.insert(4));
    assert!(a.contains(&4));

    // 如果值已经存在，那么 `HashSet::insert()` 返回 false。
    assert!(b.insert(4), "Value 4 is already in set B!");
    // 改正 ^ 将此行注释掉。

    b.insert(5);

    // 若一个集合（collection）的元素类型实现了 `Debug`，那么该集合也就实现了 `Del
    // 这通常将元素打印成这样的格式 `[elem1, elem2, ...]
    println!("A: {:?}", a);
    println!("B: {:?}", b);

    // 乱序打印 [1, 2, 3, 4, 5]。
    println!("Union: {:?}", a.union(&b).collect::<Vec<&i32>>());

    // 这将会打印出 [1]
    println!("Difference: {:?}", a.difference(&b).collect::<Vec<&i32>>());

    // 乱序打印 [2, 3, 4]。
    println!("Intersection: {:?}", a.intersection(&b).collect::<Vec<&i32>>());

    // 打印 [1, 5]
    println!("Symmetric Difference: {:?}",
             a.symmetric_difference(&b).collect::<Vec<&i32>>());
}
```

（例子改编自文档。）

引用计数 Rc

当需要多个所有权时，可以使用 `Rc`（引用计数，Reference Counting 缩写）。`Rc` 跟踪引用的数量，这相当于包裹在 `Rc` 值的所有者的数量.

每当克隆一个 `Rc` 时，`Rc` 的引用计数就会增加 1，而每当克隆得到的 `Rc` 退出作用域时，引用计数就会减少 1。当 `Rc` 的引用计数变为 0 时，这意味着已经没有所有者，`Rc` 和值两者都将被删除。

克隆 `Rc` 从不执行深拷贝。克隆只创建另一个指向包裹值的指针，并增加计数。

```
use std::rc::Rc;

fn main() {
    let rc_examples = "Rc examples".to_string();
    {
        println!("--- rc_a is created ---");

        let rc_a: Rc<String> = Rc::new(rc_examples);
        println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

        {
            println!("--- rc_a is cloned to rc_b ---");

            let rc_b: Rc<String> = Rc::clone(&rc_a);
            println!("Reference Count of rc_b: {}", Rc::strong_count(&rc_b));
            println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

            // 如果两者内部的值相等的话，则两个 `Rc` 相等。
            println!("rc_a and rc_b are equal: {}", rc_a.eq(&rc_b));

            // 我们可以直接使用值的方法
            println!("Length of the value inside rc_a: {}", rc_a.len());
            println!("Value of rc_b: {}", rc_b);

            println!("--- rc_b is dropped out of scope ---");
        }

        println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

        println!("--- rc_a is dropped out of scope ---");
    }

    // 报错！`rc_examples` 已经移入 `rc_a`。
    // 而且当 `rc_a` 被删时，`rc_examples` 也被一起删除。
    // println!("rc_examples: {}", rc_examples);
    // 试一试 ^ 注释掉此行代码
}
```

参见：

std::rc 和 std::sync::arc。

共享引用计数 Arc

当线程之间所有权需要共享时，可以使用`Arc`（共享引用计数，Atomic Reference Counted 缩写）可以使用。这个结构通过`Clone`实现可以为内存堆中的值的位置创建一个引用指针，同时增加引用计数器。由于它在线程之间共享所有权，因此当指向某个值的最后一个引用指针退出作用域时，该变量将被删除。

```
use std::sync::Arc;
use std::thread;

fn main() {
    // 这个变量声明用来指定其值的地方。
    let apple = Arc::new("the same apple");

    for _ in 0..10 {
        // 这里没有数值说明，因为它是一个指向内存堆中引用的指针。
        let apple = Arc::clone(&apple);

        thread::spawn(move || {
            // 由于使用了Arc，线程可以使用分配在 `Arc` 变量指针位置的值来生成。
            println!("{:?}", apple);
        });
    }
}
```

标准库更多介绍

标准库也提供了很多其他类型来支持某些功能，例如：

- 线程（Threads）
- 信道（Channels）
- 文件输入输出（File I/O）

这些内容在原生类型之外进行了有效扩充。

参见：

原生类型 和 标准库类型

线程

Rust 通过 `spawn` 函数提供了创建本地操作系统（native OS）线程的机制，该函数的参数是一个通过值捕获变量的闭包（moving closure）。

```
use std::thread;

static NTHREADS: i32 = 10;

// 这是主（`main`）线程
fn main() {
    // 提供一个 vector 来存放所创建的子线程（children）。
    let mut children = vec![];

    for i in 0..NTHREADS {
        // 启动（spin up）另一个线程
        children.push(thread::spawn(move || {
            println!("this is thread number {}", i)
        }));
    }

    for child in children {
        // 等待线程结束。返回一个结果。
        let _ = child.join();
    }
}
```

这些线程由操作系统调度（schedule）。

测试实例：map-reduce

Rust 使数据的并行化处理非常简单，在 Rust 中你无需面对并行处理的很多传统难题。

标准库提供了开箱即用的线程类型，把它和 Rust 的所有权概念与别名规则结合起来，可以自动地避免数据竞争（data race）。

当某状态对某线程是可见的，别名规则（即一个可变引用 XOR 一些只读引用。译注：XOR 是异或的意思，即「二者仅居其一」）就自动地避免了别的线程对它的操作。（当需要同步处理时，请使用 `Mutex` 或 `Channel` 这样的同步类型。）

在本例中，我们将会计算一堆数字中每一位的和。我们将把它们分成几块，放入不同的线程。每个线程会把自己那一块数字的每一位加起来，之后我们再把每个线程提供的结果再加起来。

注意到，虽然我们在线程之间传递了引用，但 Rust 理解我们是在传递只读的引用，因此不会发生数据竞争等不安全的事情。另外，因为我们把数据块 `move` 到了线程中，Rust 会保证数据存活至线程退出，因此不会产生悬挂指针。

```rust
use std::thread;

// 这是 `main` 线程
fn main() {

    // 这是我们要处理的数据。
    // 我们会通过线程实现 map-reduce 算法，从而计算每一位的和
    // 每个用空白符隔开的块都会分配给单独的线程来处理
    //
    // 试一试：插入空格，看看输出会怎样变化！
    let data = "86967897737416471853297327050364959
11861322575564723963297542624962850
70856234701860851907960690014725639
38397966707106094172783238747669219
52380795257888236525459303330302837
58495327135744041048897885734297812
69920216438980873548808413720956532
16278424637452589860345374828574668";

    // 创建一个向量，用于储存将要创建的子线程
    let mut children = vec![];

    /*************************************************************************
     * "Map" 阶段
     *
     * 把数据分段，并进行初始化处理
     ************************************************************************/

    // 把数据分段，每段将会单独计算
    // 每段都是完整数据的一个引用（&str）
    let chunked_data = data.split_whitespace();

    // 对分段的数据进行迭代。
    // .enumerate() 会把当前的迭代计数与被迭代的元素以元组 (index, element)
    // 的形式返回。接着立即使用“解构赋值”将该元组解构成两个变量，
    // `i` 和 `data_segment`。
    for (i, data_segment) in chunked_data.enumerate() {
        println!("data segment {} is \"{}\"", i, data_segment);

        // 用单独的线程处理每一段数据
        //
        // spawn() 返回新线程的句柄（handle），我们必须拥有句柄，
        // 才能获取线程的返回值。
        //
        // 'move || -> u32' 语法表示该闭包：
        // * 没有参数（'||'）
        // * 会获取所捕获变量的所有权（'move'）
        // * 返回无符号 32 位整数（'-> u32'）
        //
        // Rust 可以根据闭包的内容推断出 '-> u32'，所以我们可以不写它。
        //
        // 试一试：删除 'move'，看看会发生什么
        children.push(thread::spawn(move || -> u32 {
            // 计算该段的每一位的和：
            let result = data_segment
                        // 对该段中的字符进行迭代..
                        .chars()
                        // ..把字符转成数字..
```

```
                .map(|c| c.to_digit(10).expect("should be a digit"))
                // ..对返回的数字类型的迭代器求和
                .sum();

            // println! 会锁住标准输出，这样各线程打印的内容不会交错在一起
            println!("processed segment {}, result={}", i, result);

            // 不需要“return”，因为 Rust 是一种“表达式语言”，每个代码块中
            // 最后求值的表达式就是代码块的值。
            result

        }));
    }


    /*************************************************************************
     * "Reduce" 阶段
     *
     * 收集中间结果，得出最终结果
     *************************************************************************/

    // 把每个线程产生的中间结果收入一个新的向量中
    let mut intermediate_sums = vec![];
    for child in children {
        // 收集每个子线程的返回值
        let intermediate_sum = child.join().unwrap();
        intermediate_sums.push(intermediate_sum);
    }

    // 把所有中间结果加起来，得到最终结果
    //
    // 我们用“涡轮鱼”写法 ::<> 来为 sum() 提供类型提示。
    //
    // 试一试：不使用涡轮鱼写法，而是显式地指定 intermediate_sums 的类型
    let final_result = intermediate_sums.iter().sum::<u32>();

    println!("Final sum result: {}", final_result);
}
```

作业

根据用户输入的数据来决定线程的数量是不明智的。如果用户输入的数据中有一大堆空格怎么办？我们**真的**想要创建 2000 个线程吗？

请修改程序，使得数据总是被分成有限数目的段，这个数目是由程序开头的静态常量决定的。

参见:

- 线程

- 向量和迭代器
- 闭包、移动语义和 `move` 闭包
- 解构赋值
- 使用涡轮鱼写法帮助类型推断
- unwrap vs. expect
- 枚举类型

通道

Rust 为线程之间的通信提供了异步的通道（`channel`）。通道允许两个端点之间信息的单向流动：`Sender`（发送端）和 `Receiver`（接收端）。

```
use std::sync::mpsc::{Sender, Receiver};
use std::sync::mpsc;
use std::thread;

static NTHREADS: i32 = 3;

fn main() {
    // 通道有两个端点：`Sender<T>` 和 `Receiver<T>`，其中 `T` 是要发送
    // 的消息的类型（类型标注是可选的）
    let (tx, rx): (Sender<i32>, Receiver<i32>) = mpsc::channel();

    for id in 0..NTHREADS {
        // sender 端可被复制
        let thread_tx = tx.clone();

        // 每个线程都将通过通道来发送它的 id
        thread::spawn(move || {
            // 被创建的线程取得 `thread_tx` 的所有权
            // 每个线程都把消息放在通道的消息队列中
            thread_tx.send(id).unwrap();

            // 发送是一个非阻塞（non-blocking）操作，线程将在发送完消息后
            // 会立即继续进行
            println!("thread {} finished", id);
        });
    }

    // 所有消息都在此处被收集
    let mut ids = Vec::with_capacity(NTHREADS as usize);
    for _ in 0..NTHREADS {
        // `recv` 方法从通道中拿到一个消息
        // 若无可用消息的话，`recv` 将阻止当前线程
        ids.push(rx.recv());
    }

    // 显示消息被发送的次序
    println!("{:?}", ids);
}
```

路径

`Path` 结构体代表了底层文件系统的文件路径。`Path` 分为两种：`posix::Path`，针对类 UNIX 系统；以及 `windows::Path`，针对 Windows。prelude 会选择并输出符合平台类型的 `Path` 种类。

> 译注：prelude 是 Rust 自动地在每个程序中导入的一些通用的东西，这样我们就不必每写一个程序就手动导入一番。

`Path` 可从 `OsStr` 类型创建，并且它提供数种方法，用于获取路径指向的文件/目录的信息。

注意 `Path` 在内部并不是用 UTF-8 字符串表示的，而是存储为若干字节（`Vec<u8>`）的 vector。因此，将 `Path` 转化成 `&str` 并非零开销的（free），且可能失败（因此它返回一个 `Option`）。

```
use std::path::Path;

fn main() {
    // 从 `&'static str` 创建一个 `Path`
    let path = Path::new(".");

    // `display` 方法返回一个可显示（showable）的结构体
    let display = path.display();

    // `join` 使用操作系统特定的分隔符来合并路径到一个字节容器，并返回新的路径
    let new_path = path.join("a").join("b");

    // 将路径转换成一个字符串切片
    match new_path.to_str() {
        None => panic!("new path is not a valid UTF-8 sequence"),
        Some(s) => println!("new path is {}", s),
    }
}
```

记得看看其他的 `Path` 方法（`posix::Path` 或 `windows::Path` 的），还有 `Metadata` 结构体类型。

参见

OsStr 和 Metadata。

文件输入输出（I/O）

`File` 结构体表示一个被打开的文件（它包裹了一个文件描述符），并赋予了对所表示的文件的读写能力。

由于在进行文件 I/O（输入/输出）操作时可能出现各种错误，因此 `File` 的所有方法都返回 `io::Result<T>` 类型，它是 `Result<T, io::Error>` 的别名。

这使得所有 I/O 操作的失败都变成**显式的**。借助这点，程序员可以看到所有的失败路径，并被鼓励主动地处理这些情形。

打开文件 open

`open` 静态方法能够以只读模式（read-only mode）打开一个文件。

`File` 拥有资源，即文件描述符（file descriptor），它会在自身被 `drop` 时关闭文件。

```
use std::fs::File;
use std::io::prelude::*;
use std::path::Path;

fn main() {
    // 创建指向所需的文件的路径
    let path = Path::new("hello.txt");
    let display = path.display();

    // 以只读方式打开路径，返回 `io::Result<File>`
    let mut file = match File::open(&path) {
        // `io::Error` 的 `description` 方法返回一个描述错误的字符串。
        Err(why) => panic!("couldn't open {}: {:?}", display, why),
        Ok(file) => file,
    };

    // 读取文件内容到一个字符串，返回 `io::Result<usize>`
    let mut s = String::new();
    match file.read_to_string(&mut s) {
        Err(why) => panic!("couldn't read {}: {:?}", display, why),
        Ok(_) => print!("{} contains:\n{}", display, s),
    }

    // `file` 离开作用域，并且 `hello.txt` 文件将被关闭。
}
```

下面是所希望的成功的输出：

```
$ echo "Hello World!" > hello.txt
$ rustc open.rs && ./open
hello.txt contains:
Hello World!
```

（我们鼓励您在不同的失败条件下测试前面的例子：hello.txt 不存在，或 hello.txt 不可读，等等。）

创建文件 `create`

`create` 静态方法以只写模式（write-only mode）打开一个文件。若文件已经存在，则旧内容将被销毁。否则，将创建一个新文件。

```
static LOREM_IPSUM: &'static str =
"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
";

use std::io::prelude::*;
use std::fs::File;
use std::path::Path;

fn main() {
    let path = Path::new("out/lorem_ipsum.txt");
    let display = path.display();

    // 以只写模式打开文件，返回 `io::Result<File>`
    let mut file = match File::create(&path) {
        Err(why) => panic!("couldn't create {}: {:?}", display, why),
        Ok(file) => file,
    };

    // 将 `LOREM_IPSUM` 字符串写进 `file`，返回 `io::Result<()>`
    match file.write_all(LOREM_IPSUM.as_bytes()) {
        Err(why) => {
            panic!("couldn't write to {}: {:?}", display, why)
        },
        Ok(_) => println!("successfully wrote to {}", display),
    }
}
```

下面是预期成功的输出：

```
$ mkdir out
$ rustc create.rs && ./create
successfully wrote to out/lorem_ipsum.txt
$ cat out/lorem_ipsum.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
```

（和前面例子一样，我们鼓励你在失败条件下测试这个例子。）

还有一个更通用的 `open_mode` 方法，这能够以其他方式来来打开文件，如：read+write（读 + 写），追加（append），等等。

读取行

方法 `lines()` 在文件的行上返回一个迭代器。

`File::open` 需要一个泛型 `AsRef<Path>`。这正是 `read_lines()` 期望的输入。

```
use std::fs::File;
use std::io::{self, BufRead};
use std::path::Path;

fn main() {
    // 在生成输出之前，文件主机必须存在于当前路径中
    if let Ok(lines) = read_lines("./hosts") {
        // 使用迭代器，返回一个（可选）字符串
        for line in lines {
            if let Ok(ip) = line {
                println!("{}", ip);
            }
        }
    }
}

// 输出包裹在 Result 中以允许匹配错误，
// 将迭代器返回给文件行的读取器（Reader）。
fn read_lines<P>(filename: P) -> io::Result<io::Lines<io::BufReader<File>>>
where P: AsRef<Path>, {
    let file = File::open(filename)?;
    Ok(io::BufReader::new(file).lines())
}
```

运行此程序将一行行将内容打印出来。

```
$ echo -e "127.0.0.1\n192.168.0.1\n" > hosts
$ rustc read_lines.rs && ./read_lines
127.0.0.1
192.168.0.1
```

这个过程比在内存中创建 `String` 更有效，特别是处理更大的文件。

子进程

`process::Output` 结构体表示已结束的子进程（child process）的输出，而 `process::Command` 结构体是一个进程创建者（process builder）。

```
use std::process::Command;

fn main() {
    let output = Command::new("rustc")
        .arg("--version")
        .output().unwrap_or_else(|e| {
            panic!("failed to execute process: {}", e)
    });

    if output.status.success() {
        let s = String::from_utf8_lossy(&output.stdout);

        print!("rustc succeeded and stdout was:\n{}", s);
    } else {
        let s = String::from_utf8_lossy(&output.stderr);

        print!("rustc failed and stderr was:\n{}", s);
    }
}
```

（再试试上面的例子，给 `rustc` 命令传入一个错误的 flag）

管道

`std::Child` 结构体代表了一个正在运行的子进程，它暴露了 `stdin`（标准输入），`stdout`（标准输出）和 `stderr`（标准错误）句柄，从而可以通过管道与所代表的进程交互。

```
use std::io::prelude::*;
use std::process::{Command, Stdio};

static PANGRAM: &'static str =
"the quick brown fox jumped over the lazy dog\n";

fn main() {
    // 启动 `wc` 命令
    let process = match Command::new("wc")
                                .stdin(Stdio::piped())
                                .stdout(Stdio::piped())
                                .spawn() {
        Err(why) => panic!("couldn't spawn wc: {:?}", why),
        Ok(process) => process,
    };

    // 将字符串写入 `wc` 的 `stdin`。
    //
    // `stdin` 拥有 `Option<ChildStdin>` 类型，不过我们已经知道这个实例不为空值，
    // 因而可以直接 `unwrap` 它。
    match process.stdin.unwrap().write_all(PANGRAM.as_bytes()) {
        Err(why) => panic!("couldn't write to wc stdin: {:?}", why),
        Ok(_) => println!("sent pangram to wc"),
    }

    // 因为 `stdin` 在上面调用后就不再存活，所以它被 `drop` 了，管道也被关闭。
    //
    // 这点非常重要，因为否则 `wc` 就不会开始处理我们刚刚发送的输入。

    // `stdout` 字段也拥有 `Option<ChildStdout>` 类型，所以必需解包。
    let mut s = String::new();
    match process.stdout.unwrap().read_to_string(&mut s) {
        Err(why) => panic!("couldn't read wc stdout: {:?}", why),
        Ok(_) => print!("wc responded with:\n{}", s),
    }
}
```

等待

如果你想等待一个 `process::Child` 完成，就必须调用 `Child::wait`，这会返回一个 `process::ExitStatus`。

```
use std::process::Command;

fn main() {
    let mut child = Command::new("sleep").arg("5").spawn().unwrap();
    let _result = child.wait().unwrap();

    println!("reached end of main");
}
```

```
$ rustc wait.rs && ./wait
reached end of main
# `wait` keeps running for 5 seconds
# `sleep 5` command ends, and then our `wait` program finishes
```

文件系统操作

`std::io::fs` 模块包含几个处理文件系统的函数。

```rust
use std::fs;
use std::fs::{File, OpenOptions};
use std::io;
use std::io::prelude::*;
use std::os::unix;
use std::path::Path;

// `% cat path` 的简单实现
fn cat(path: &Path) -> io::Result<String> {
    let mut f = File::open(path)?;
    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}

// `% echo s > path` 的简单实现
fn echo(s: &str, path: &Path) -> io::Result<()> {
    let mut f = File::create(path)?;

    f.write_all(s.as_bytes())
}

// `% touch path` 的简单实现（忽略已存在的文件）
fn touch(path: &Path) -> io::Result<()> {
    match OpenOptions::new().create(true).write(true).open(path) {
        Ok(_) => Ok(()),
        Err(e) => Err(e),
    }
}

fn main() {
    println!("`mkdir a`");
    // 创建一个目录，返回 `io::Result<()>`
    match fs::create_dir("a") {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(_) => {},
    }

    println!("`echo hello > a/b.txt`");
    // 前面的匹配可以用 `unwrap_or_else` 方法简化
    echo("hello", &Path::new("a/b.txt")).unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`mkdir -p a/c/d`");
    // 递归地创建一个目录，返回 `io::Result<()>`
    fs::create_dir_all("a/c/d").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`touch a/c/e.txt`");
    touch(&Path::new("a/c/e.txt")).unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });
```

```
    println!("`ln -s ../b.txt a/c/b.txt`");
    // 创建一个符号链接，返回 `io::Resutl<()>`
    if cfg!(target_family = "unix") {
        unix::fs::symlink("../b.txt", "a/c/b.txt").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
        });
    }

    println!("`cat a/c/b.txt`");
    match cat(&Path::new("a/c/b.txt")) {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(s) => println!("> {}", s),
    }

    println!("`ls a`");
    // 读取目录的内容，返回 `io::Result<Vec<Path>>`
    match fs::read_dir("a") {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(paths) => for path in paths {
            println!("> {:?}", path.unwrap().path());
        },
    }

    println!("`rm a/c/e.txt`");
    // 删除一个文件，返回 `io::Result<()>`
    fs::remove_file("a/c/e.txt").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`rmdir a/c/d`");
    // 移除一个空目录，返回 `io::Result<()>`
    fs::remove_dir("a/c/d").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });
}
```

下面是所期望的成功的输出：

```
$ rustc fs.rs && ./fs
`mkdir a`
`echo hello > a/b.txt`
`mkdir -p a/c/d`
`touch a/c/e.txt`
`ln -s ../b.txt a/c/b.txt`
`cat a/c/b.txt`
> hello
`ls a`
> "a/b.txt"
> "a/c"
`rm a/c/e.txt`
`rmdir a/c/d`
```

且 `a` 目录的最终状态为：

```
$ tree a
a
|-- b.txt
`-- c
    `-- b.txt -> ../b.txt

1 directory, 2 files
```

另一种定义 `cat` 函数的方式是使用 `?` 标记：

```
fn cat(path: &Path) -> io::Result<String> {
    let mut f = File::open(path)?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

参见：

`cfg!`

程序参数

标准库

命令行参数可使用 `std::env::args` 进行接收，这将返回一个迭代器，该迭代器会对每个参数举出一个字符串。

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    // 第一个参数是调用本程序的路径
    println!("My path is {}.", args[0]);

    // 其余的参数是被传递给程序的命令行参数。
    // 请这样调用程序：
    //   $ ./args arg1 arg2
    println!("I got {:?} arguments: {:?}.", args.len() - 1, &args[1..]);
}
```

```
$ ./args 1 2 3
My path is ./args.
I got 3 arguments: ["1", "2", "3"].
```

crate

另外，也有很多 crate 提供了编写命令行应用的额外功能。Rust Cookbook 展示了使用最流行的命令行参数 crate，即 `clap` 的最佳实践。

参数解析

可以用模式匹配来解析简单的参数：

```rust
use std::env;

fn increase(number: i32) {
    println!("{}", number + 1);
}

fn decrease(number: i32) {
    println!("{}", number - 1);
}

fn help() {
    println!("usage:
match_args <string>
    Check whether given string is the answer.
match_args {{increase|decrease}} <integer>
    Increase or decrease given integer by one.");
}

fn main() {
    let args: Vec<String> = env::args().collect();

    match args.len() {
        // 没有传入参数
        1 => {
            println!("My name is 'match_args'. Try passing some arguments!");
        },
        // 一个传入参数
        2 => {
            match args[1].parse() {
                Ok(42) => println!("This is the answer!"),
                _ => println!("This is not the answer."),
            }
        },
        // 传入一条命令和一个参数
        3 => {
            let cmd = &args[1];
            let num = &args[2];
            // 解析数字
            let number: i32 = match num.parse() {
                Ok(n) => {
                    n
                },
                Err(_) => {
                    println!("error: second argument not an integer");
                    help();
                    return;
                },
            };
            // 解析命令
            match &cmd[..] {
                "increase" => increase(number),
                "decrease" => decrease(number),
                _ => {
                    println!("error: invalid command");
                    help();
                },
            }
        },
```

```
        // 所有其他情况
        _ => {
            // 显示帮助信息
            help();
        }
    }
}
```

```
$ ./match_args Rust
This is not the answer.
$ ./match_args 42
This is the answer!
$ ./match_args do something
error: second argument not an integer
usage:
match_args <string>
    Check whether given string is the answer.
match_args {increase|decrease} <integer>
    Increase or decrease given integer by one.
$ ./match_args do 42
error: invalid command
usage:
match_args <string>
    Check whether given string is the answer.
match_args {increase|decrease} <integer>
    Increase or decrease given integer by one.
$ ./match_args increase 42
43
```

外部语言函数接口

Rust 提供了到 C 语言库的外部语言函数接口（Foreign Function Interface，FFI）。外部语言函数必须在一个 `extern` 代码块中声明，且该代码块要带有一个包含库名称的 `#[link]` 属性。

```
use std::fmt;

// 这个 extern 代码块链接到 libm 库
#[link(name = "m")]
extern {
    // 这个外部函数用于计算单精度复数的平方根
    fn csqrtf(z: Complex) -> Complex;

    // 这个用来计算单精度复数的复变余弦
    fn ccosf(z: Complex) -> Complex;
}

// 由于调用其他语言的函数被认为是不安全的，我们通常会给它们写一层安全的封装
fn cos(z: Complex) -> Complex {
    unsafe { ccosf(z) }
}

fn main() {
    // z = -1 + 0i
    let z = Complex { re: -1., im: 0. };

    // 调用外部语言函数是不安全操作
    let z_sqrt = unsafe { csqrtf(z) };

    println!("the square root of {:?} is {:?}", z, z_sqrt);

    // 调用不安全操作的安全的 API 封装
    println!("cos({:?}) = {:?}", z, cos(z));
}

// 单精度复数的最简实现
#[repr(C)]
#[derive(Clone, Copy)]
struct Complex {
    re: f32,
    im: f32,
}

impl fmt::Debug for Complex {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        if self.im < 0. {
            write!(f, "{}-{}i", self.re, -self.im)
        } else {
            write!(f, "{}+{}i", self.re, self.im)
        }
    }
}
```

测试

Rust 是一门非常重视正确性的语言，这门语言本身就提供了对编写软件测试的支持。

测试有三种风格：

- 单元测试。
- 文档测试。
- 集成测试。

Rust 也支持在测试中指定额外的依赖：

- 开发依赖

参见

- TRPL 中关于测试的章节
- API 指导原则中关于文档测试的部分

单元测试

测试（test）是这样一种 Rust 函数：它保证其他部分的代码按照所希望的行为正常运行。测试函数的函数体通常会进行一些配置，运行我们想要测试的代码，然后断言（assert）结果是不是我们所期望的。

大多数单元测试都会被放到一个叫 `tests` 的、带有 `#[cfg(test)]` 属性的模块中，测试函数要加上 `#[test]` 属性。

当测试函数中有什么东西 panic 了，测试就失败。有一些这方面的辅助宏：

- `assert!(expression)` - 如果表达式的值是 `false` 则 panic。
- `assert_eq!(left, right)` 和 `assert_ne!(left, right)` - 检验左右两边是否 相等/不等。

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

// 这个加法函数写得很差，本例中我们会使它失败。
#[allow(dead_code)]
fn bad_add(a: i32, b: i32) -> i32 {
    a - b
}

#[cfg(test)]
mod tests {
    // 注意这个惯用法：在 tests 模块中，从外部作用域导入所有名字。
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }

    #[test]
    fn test_bad_add() {
        // 这个断言会导致测试失败。注意私有的函数也可以被测试！
        assert_eq!(bad_add(1, 2), 3);
    }
}
```

可以使用 `cargo test` 来运行测试。

```
$ cargo test

running 2 tests
test tests::test_bad_add ... FAILED
test tests::test_add ... ok

failures:

---- tests::test_bad_add stdout ----
        thread 'tests::test_bad_add' panicked at 'assertion failed: `(left ==
right)`
  left: `-1`,
 right: `3`', src/lib.rs:21:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    tests::test_bad_add

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

测试 panic

一些函数应当在特定条件下 panic。为测试这种行为，请使用 `#[should_panic]` 属性。这个属性接受可选参数 `expected =` 以指定 panic 时的消息。如果你的函数能以多种方式 panic，这个属性就保证了你在测试的确实是所指定的 panic。

```rust
pub fn divide_non_zero_result(a: u32, b: u32) -> u32 {
    if b == 0 {
        panic!("Divide-by-zero error");
    } else if a < b {
        panic!("Divide result is zero");
    }
    a / b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_divide() {
        assert_eq!(divide_non_zero_result(10, 2), 5);
    }

    #[test]
    #[should_panic]
    fn test_any_panic() {
        divide_non_zero_result(1, 0);
    }

    #[test]
    #[should_panic(expected = "Divide result is zero")]
    fn test_specific_panic() {
        divide_non_zero_result(1, 10);
    }
}
```

运行这些测试会输出：

```
$ cargo test

running 3 tests
test tests::test_any_panic ... ok
test tests::test_divide ... ok
test tests::test_specific_panic ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests tmp-test-should-panic

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

运行特定的测试

要运行特定的测试，只要把测试名称传给 `cargo test` 命令就可以了。

```
$ cargo test test_any_panic
running 1 test
test tests::test_any_panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out

   Doc-tests tmp-test-should-panic

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

要运行多个测试，可以仅指定测试名称中的一部分，用它来匹配所有要运行的测试。

```
$ cargo test panic
running 2 tests
test tests::test_any_panic ... ok
test tests::test_specific_panic ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out

   Doc-tests tmp-test-should-panic

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

忽略测试

可以把属性 `#[ignore]` 赋予测试以排除某些测试，或者使用 `cargo test -- --ignored` 命令来运行它们。

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_add() {
        assert_eq!(add(2, 2), 4);
    }

    #[test]
    fn test_add_hundred() {
        assert_eq!(add(100, 2), 102);
        assert_eq!(add(2, 100), 102);
    }

    #[test]
    #[ignore]
    fn ignored_test() {
        assert_eq!(add(0, 0), 0);
    }
}
```

```
$ cargo test
running 1 test
test tests::ignored_test ... ignored

test result: ok. 0 passed; 0 failed; 1 ignored; 0 measured; 0 filtered out

   Doc-tests tmp-ignore

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

$ cargo test -- --ignored
running 1 test
test tests::ignored_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests tmp-ignore

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

文档测试

为 Rust 工程编写文档的主要方式是在源代码中写注释。文档注释使用 markdown 语法书写，支持代码块。Rust 很注重正确性，这些注释中的代码块也会被编译并且用作测试。

```
/// 第一行是对函数的简短描述。
///
/// 接下来数行是详细文档。代码块用三个反引号开启，Rust 会隐式地在其中添加
/// `fn main()` 和 `extern crate <cratename>`。比如测试 `doccomments` crate：
///
/// ```
/// let result = doccomments::add(2, 3);
/// assert_eq!(result, 5);
/// ```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

/// 文档注释通常可能带有 "Examples"、"Panics" 和 "Failures" 这些部分。
///
/// 下面的函数将两数相除。
///
/// # Examples
///
/// ```
/// let result = doccomments::div(10, 2);
/// assert_eq!(result, 5);
/// ```
///
/// # Panics
///
/// 如果第二个参数是 0，函数将会 panic。
///
/// ```rust,should_panic
/// // panics on division by zero
/// doccomments::div(10, 0);
/// ```
pub fn div(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("Divide-by-zero error");
    }

    a / b
}
```

这些测试仍然可以通过 `cargo test` 执行：

```
$ cargo test
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests doccomments

running 3 tests
test src/lib.rs - add (line 7) ... ok
test src/lib.rs - div (line 21) ... ok
test src/lib.rs - div (line 31) ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

文档测试的目的

文档测试的主要目的是作为使用函数功能的例子，这是最重要的指导原则之一。文档测试应当可以作为完整的代码段被直接使用（很多用户会复制文档中的代码来用，所以例子要写得完善）。但使用 `?` 符号会导致编译失败，因为 `main` 函数会返回 `unit` 类型。幸运的是，我们可以在文档中隐藏几行源代码：你可以写 `fn try_main() -> Result<(), ErrorType>` 这样的函数，把它隐藏起来，然后在隐藏的 `main` 中展开它。听起来很复杂？请看例子：

```
/// 在文档测试中使用隐藏的 `try_main`。
///
/// ```
/// # // 被隐藏的行以 `#` 开始，但它们仍然会被编译！
/// # fn try_main() -> Result<(), String> { // 隐藏行包围了文档中显示的函数体
/// let res = try::try_div(10, 2)?;
/// # Ok(()) // 从 try_main 返回
/// # }
/// # fn main() { // 开始主函数，其中将展开 `try_main` 函数
/// #    try_main().unwrap(); // 调用并展开 try_main，这样出错时测试会 panic
/// # }
pub fn try_div(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Divide-by-zero"))
    } else {
        Ok(a / b)
    }
}
```

参见

- 关于文档风格的 RFC505
- API 指导原则中关于文档的原则

集成测试

单元测试一次仅能单独测试一个模块，这种测试是小规模的，并且能测试私有代码；集成测试是 crate 外部的测试，并且仅使用 crate 的公共接口，就像其他使用该 crate 的程序那样。集成测试的目的是检验你的库的各部分是否能够正确地协同工作。

cargo 在与 `src` 同级别的 `tests` 目录寻找集成测试。

文件 `src/lib.rs`：

```
// 在一个叫做 'adder' 的 crate 中定义此函数。
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

包含测试的文件：`tests/integration_test.rs`：

```

#[test]
fn test_add() {
    assert_eq!(adder::add(3, 2), 5);
}
```

使用 `cargo test` 命令：

```
$ cargo test
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

     Running target/debug/deps/integration_test-bcd60824f5fbfe19

running 1 test
test test_add ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

`tests` 目录中的每一个 Rust 源文件都被编译成一个单独的 crate。在集成测试中要想共享代码，一种方式是创建具有公用函数的模块，然后在测试中导入并使用它。

文件 `tests/common.rs`：

```
pub fn setup() {
    // 一些配置代码，比如创建文件/目录，开启服务器等等。
}
```

包含测试的文件：`tests/integration_test.rs`

```
// 导入共用模块。
mod common;

#[test]
fn test_add() {
    // 使用共用模块。
    common::setup();
    assert_eq!(adder::add(3, 2), 5);
}
```

带有共用代码的模块遵循和普通的模块一样的规则，所以完全可以把公共模块写在 `tests/common/mod.rs` 文件中。

开发依赖

有时仅在测试中才需要一些依赖（比如基准测试相关的）。这种依赖要写在 `Cargo.toml` 的 `[dev-dependencies]` 部分。这些依赖不会传播给其他依赖于这个包的包。

比如说使用 `pretty_assertions`，这是扩展了标准的 `assert!` 宏的一个 crate。

文件 `Cargo.toml`：

```
# 这里省略了标准的 crate 数据
[dev-dependencies]
pretty_assertions = "1"
```

文件 `src/lib.rs`：

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]
mod tests {
    use super::*;
    use pretty_assertions::assert_eq; // 仅用于测试，不能在非测试代码中使用

    #[test]
    fn test_add() {
        assert_eq!(add(2, 3), 5);
    }
}
```

参见

Cargo 文档中关于指定依赖的部分。

不安全操作

在本章一开始，我们借用官方文档的一句话，“在整个代码库（code base，指构建一个软件系统所使用的全部代码）中，要尽可能减少不安全代码的量”。记住这句话，接着我们进入学习！在 Rust 中，不安全代码块用于避开编译器的保护策略；具体地说，不安全代码块主要用于四件事情：

- 解引用裸指针
- 通过 FFI 调用函数（这已经在之前的章节介绍过了）
- 调用不安全的函数
- 内联汇编（inline assembly）

原始指针

原始指针（raw pointer，裸指针）`*` 和引用 `&T` 有类似的功能，但引用总是安全的，因为借用检查器保证了它指向一个有效的数据。解引用一个裸指针只能通过不安全代码块执行。

```
fn main() {
    let raw_p: *const u32 = &10;

    unsafe {
        assert!(*raw_p == 10);
    }
}
```

调用不安全函数

一些函数可以声明为不安全的（`unsafe`），这意味着在使用它时保证正确性不再是编译器的责任，而是程序员的。一个例子就是 `std::slice::from_raw_parts`，向它传入指向第一个元素的指针和长度参数，它会创建一个切片。

```rust
use std::slice;

fn main() {
    let some_vector = vec![1, 2, 3, 4];

    let pointer = some_vector.as_ptr();
    let length = some_vector.len();

    unsafe {
        let my_slice: &[u32] = slice::from_raw_parts(pointer, length);

        assert_eq!(some_vector.as_slice(), my_slice);
    }
}
```

`slice::from_raw_parts` 假设传入的指针指向有效的内存，且被指向的内存具有正确的数据类型，我们**必须**满足这一假设，否则程序的行为是未定义的（undefined），于是我们就不能预测会发生些什么了。

兼容性

Rust 语言正在快速发展，因此尽管努力确保尽可能向前兼容，但仍可能出现某些兼容性问题。

- 原始标识符

原始标志符

与许多编程语言一样，Rust 拥有“关键字”的概念。 这些标识符对语言有特定意义，所以不能在变量名、函数名和其他位置使用它们。 原始标识符允许你使用通常不允许的关键字。 当 Rust 引入新关键字时，使用旧版 Rust 的库拥有与新版本中引入的关键字同名的变量或函数，这一点就特别有用。

举个例子，使用 2015 版 Rust 编译的 crate `foo`，它导出一个名为 `try` 的函数。 此关键字（*try*）在 2018 版本的新功能中保留下来，因此如果没有原始标识符，我们将无法命名该功能。

```
extern crate foo;

fn main() {
    foo::try();
}
```

将得到如下错误：

```
error: expected identifier, found keyword `try`
 --> src/main.rs:4:4
  |
4 | foo::try();
  |      ^^^ expected identifier, found keyword
```

使用原始标志符重写上述代码：

```
extern crate foo;

fn main() {
    foo::r#try();
}
```

补充

在软件工程中，有些主题和写程序并没有直接的关联，但它们为你提供了工具和基础设施支持，使得软件对每个人都变得更易用。这些主题包括：

- 文档：通过附带的 `rustdoc` 生成库文档给用户。
- 测试：为库创建测试套件，确保库准确地实现了你想要的功能。
- 基准测试（benchmark）：对功能进行基准测试，保证其运行速度足够快。

文档

用 `cargo doc` 构建文档到 `target/doc`。

用 `cargo test` 运行所有测试（包括文档测试），用 `cargo test --doc` 仅运行文档测试。

这些命令会恰当地按需调用 `rustdoc` （以及 `rustc`）。

文档注释

文档注释对于需要文档的大型项目来说非常重要。当运行 `rustdoc`，文档注释就会编译成文档。它们使用 `///` 标记，并支持 Markdown。

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

/// 这里给出一个“人”的表示
pub struct Person {
    /// 一个人必须有名字（不管 Juliet 多讨厌她自己的名字）。
    name: String,
}

impl Person {
    /// 返回具有指定名字的一个人
    ///
    /// # 参数
    ///
    /// * `name` - 字符串切片，代表人的名字
    ///
    /// # 示例
    ///
    /// ```
    /// // 在文档注释中，你可以书写代码块
    /// // 如果向 `rustdoc` 传递 --test 参数，它还会帮你测试注释文档中的代码！
    /// use doc::Person;
    /// let person = Person::new("name");
    /// ```
    pub fn new(name: &str) -> Person {
        Person {
            name: name.to_string(),
        }
    }

    /// 给一个友好的问候！
    /// 对被叫到的 `Person` 说 "Hello, [name]" 。
    pub fn hello(& self) {
        println!("Hello, {}!", self.name);
    }
}

fn main() {
    let john = Person::new("John");

    john.hello();
}
```

要运行测试，首先将代码构建为库，然后告诉 `rustdoc` 在哪里找到库，这样它就可以使每个文档中的程序链接到库：

```
$ rustc doc.rs --crate-type lib
$ rustdoc --test --extern doc="libdoc.rlib" doc.rs
```

文档属性

下面是一些使用 `rustdoc` 时最常使用的 `#[doc]` 属性的例子。

inline

用于内联文档，而不是链接到单独的页面。

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

/// bar 的文档
mod bar {
    /// Bar 的文档
    pub struct Bar;
}
```

no_inline

用于防止链接到单独的页面或其他位置。

```
// 来自 libcore/prelude 的例子
#[doc(no_inline)]
pub use crate::mem::drop;
```

hidden

使用此属性来告诉 `rustdoc` 不要包含此项到文档中：

```
// 来自 futures-rs 库的例子
#[doc(hidden)]
pub use self::async_await::*;
```

对文档来说，`rustdoc` 被社区广泛采用。标准库文档也是用它生成的。

参见:

- The Rust Book: Making Useful Documentation Comments
- The rustdoc Book
- The Reference: Doc comments
- RFC 1574: API Documentation Conventions
- RFC 1946: Relative links to other items from doc comments (intra-rustdoc links)
- Is there any documentation style guide for comments? (reddit)

Playpen

Rust Playpen 是一个在线运行 Rust 代码的网络接口。现在该项目通常称为 Rust Playground。

在 mdbook 使用

在 `mdbook` 中，你可以让示例代码运行和编辑。

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

这使读者既可以运行你的代码示例，也可以对其进行修改和调整。此处的关键是将单词添加 `editable` 到代码块中，并用逗号分隔。

````
```rust,editable
//...将你的代码写在这里
```
````

此外，如果想要 `mdbook` 在构建和测试时跳过该代码，则可以添加 `ignore` 。

````
```rust,editable,ignore
//...将你的代码写在这里
```
````

在文档中使用

可能你已经在某些 Rust 官方文档中注意到了一个名为 “Run” 的按钮，该按钮在 Rust Playground 的新选项卡中打开了代码示例。如果使用名为的 `html_playground_url` 的 #[doc] 属性，则启用此功能。

参见：

- The Rust Playground
- 下一代 playpen
- 官方文档