Table of Contents

通过例子学 **Rust**

> 已于 2016-08-07 翻译完全部内容，欢迎纠正——最后更新时间 2017-10-03
> Chinese translation of the Rust by Example

通过例子学 Rust（Rust by Example 中文版）包含在线代码编辑器。

使用

如果想阅读《通过例子学 Rust》，可以直接访问 https://rustwiki.org/rust-by-example/ 在线上阅读。（英文阅读地址：https://rustbyexample.com/）

若想在本地阅读，请先安装 Rust，然后进行下面操作：

```
$ git clone https://github.com/rust-lang-cn/rust-by-example-cn
$ cd rust-by-example-cn
$ cargo install mdbook
$ mdbook build
$ mdbook serve
```

如何贡献

请查看 CONTRIBUTING.md 文件了解详细内容。

其他语言版本

- English
- Japanese

授权协议

《通过例子学 Rust》（中文版与英文版 Rust by Example 均） 使用 Apache 2.0 license 和 MIT license 两种协议进行授权。

详情参见 LICENSE-APACHE 和 LICENSE-MIT。

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
// 可以单击前面的按钮 "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，你可以自由地改进代码！
// 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() {
    // 调用已编译成的可执行文件时，在这里面的语句将会运行

    // 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` 可以将源程序生成可执行文件：

```
$ rustc hello.rs
```

`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，其注释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，其注释内容将被解析成 HTML 帮助文档:
  - `/// 对接下来的项生成帮助文档。`
  - `//! 对封闭项生成帮助文档。`

```
fn main() {
    // 这是行注释的例子
    // 注意这里有两个斜线在本行的开头
    // 在这里面的所有内容编译器都不会读取

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

    // 想要运行上述语句？现在请将上述语句的两条斜线删掉，并重新运行。

    /*
     * 这是另外一种格式的注释——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在 main() 函数中的行。/*/*/* 赶紧试试！*/*/*/
     */

     /*
      注意，上面的例子中纵向都有 `*`，这完全是基于格式考虑，实际上这并不是
      必须的。
      */

     // 观察块注释是如何对简单的表达式进行控制，而行注释不能这样操作。
     // 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，其中包括：

- `format!`：将格式化文本写到 `字符串` (String)。(译注: `字符串` 是返回值不是参数。)
- `print!`：与 `format!` 类似，但将文本输出到控制台。
- `println!`：与 `print!` 类似，但输出结果追加一个换行符。

所有的解析文本都以相同的方式进行。另外一点是格式化的正确性在编译时检查。

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

    // 不加后缀的话，31自动成为 I32 类型。
    // 你可以添加后缀来改变 31 的原来类型。

    // 下面有多种可选形式。
    // 可以使用的位置参数。
    println!("{0}, this is {1}. {1}, this is {0}", "Alice", "Bob");

    // 可以使用赋值语句。
    println!("{subject} {verb} {object}",
             object="the lazy dog",
             subject="the quick brown fox",
             verb="jumps over");

    // 特殊的格式实现可以在后面加上 `:` 符号。
    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);

    // 你可以对数字左边位数上补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` I32 `类型结构体(structure)。命名为 `Structure`。
    #[allow(dead_code)]
    struct Structure(i32);
```

```
    // 但是像结构体这样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 `traits`（traits翻译成中文有“特征，特性”等意思）来控制文字显示。这里面有两个重要的基本格式类型如下：

- `fmt::Debug`：使用 `{:?}` 作标记。格式化文本以便调试。
- `fmt::Display`：使用 `{}` 作标记。以优雅和友好的方式来格式文本。

在本书中我们使用 `fmt::Display`，因为标准库提供了这些类型的实现。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），使得运行不会报错。
- 添加一个 `println!` 宏来打印：`Pi is roughly 3.142`（Pi约等于3.142），通过控制有效数字得到显示的结果。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 `traits`

调试

所有要用到 `std::fmt` 格式化的 `traits` 类型都需要转化成可打印的实现。 `std` 库这些类型能够自动实现。但所有其他类型都必须手动来实现。

`fmt::Debug` `trait` 使上面工作变得相当简单。所有类型都能推导（自动创建） `fmt::Debug` 的实现。但是 `fmt::Display` 需要手动来实现。

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

// `derive`属性会自动创建实现，借助`fmt::Debug`使得这个`struct`能够打印。
#[derive(Debug)]
struct DebugPrintable(i32);
```

所有 `std` 库类型加上 `{:?}` 后也能够自动打印：

```
// 从 `fmt::Debug` 获得实现给 `Structure`。
// `Structure` 是一个包含`i32`基本类型的结构体。
#[derive(Debug)]
struct Structure(i32);

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

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

    // `Structure` 是能够打印的类型。
    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`？
    println!("Now {:?} will print!", Deep(Structure(7)));
}
```

所以 `fmt::Debug` 确实使这些内容可以打印，但是牺牲了美感。手动执行 `fmt::Display` 将能够弥补这些问题。

参见：

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

显示

`fmt::Debug` 看起来并不简洁，然而它对自定义输出外观通常是有好处的。而 `fmt::Display` 是通过手动的方式来实现，采用了 `{}` 来打印标记。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，使用 `fmt::Display` 来实现。这只是简单地给元组结构体`Structure` 包含
// 一个 `i32` 元素。
struct Structure(i32);

// 为了使用 `{}` 标记，必须手动实现 `fmt::Display` trait 来支持相应类型。
impl fmt::Display for Structure {
    // 这个 trait 要求 `fmt` 带有正确的标记
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // 严格将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。注意这里的 `write!` 用法和 `println!` 很相似。
        write!(f, "{}", self.0)
    }
}
```

`fmt::display` 的使用形式可能比 `fmt::Debug` 简洁，但它对于标准库的处理有一个问题。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>` 都实现了单一样式，那么它应该 是那种样式？随意一种或者包含两种？

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

答案是否定的，因为没有合适的样式适用于所有类型，标准库也没规定一种情况。对于 `Vec<T>` 或其 他任意泛型容器(container)，`fmt::Display` 都没有实现形式。在这种含有泛型的情况下要用到 `fmt::Debug` 。

而对于非泛型的容器类型的输出，`fmt::Display` 都能够实现。

```
use std::fmt; // 导入 `fmt`

// 带有两个数字的结构体。`Debug` 将被派生，可以看到输出结果和 `Display` 的差异。
#[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` 方式来表示各个数据。
```

```
        write!(f, "({}, {})", self.0, self.1)
    }
}

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

// 类似地对 Point2D 进行实现
impl fmt::Display for Point2D {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // 自定义方式实现，仅让 `x` 和 `y` 标识出来。
        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` 都实现了，而 `fmt::Binary` 都没有，因此 `fmt::Binary` 不能使用。 `std::fmt` 有很多这样的 `traits` ，使用这些 trait 都要有各自的实现。这些内容将 在后面的 `std::fmt` 章节中详细介绍。

动手试一试

对上面程序的运行结果检验完毕后，在上述示例程序中，仿照 `Point2` 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

对一个结构体来说，须对各个元素逐个实现 `fmt::Display` 可能会很麻烦。问题在于每个 `write!` 都要生成一个 `fmt::Result` 。彻底地实现需要处理所有的结果。出于这方面考虑，Rust 提供了 `try!` 宏。

在 `write!` 上使用 `try!` 类似这样：

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

在有 `try!` 的基础上，对一个 `Vec` 实现 `fmt::Display` 是相当直接的：

```
use std::fmt; // 导入 `fmt` 模块。

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

impl fmt::Display for List {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // 对 `self` 解引用，并通过解构创建一个指向 `vec` 的引用。
        let List(ref vec) = *self;

        try!(write!(f, "["));

        // 对 `vec` 进行迭代，`v` 为每次迭代的值，`count` 为计数。
        for (count, v) in vec.iter().enumerate() {
            // 在调用 `write!` 前对每个元素（第一个元素除外）加上逗号。
            // 使用 `try!` ，在出错的情况返回。
            if count != 0 { try!(write!(f, ", ")); }
            try!(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 里面元素的索引也能够打印出来。新的结果如下：

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

参见：

`for` , `ref` , `Result` , `struct` , `try!` , `vec!`

格式化

我们可以看到格式化就是通过格式字符串得到特定格式：

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

根据使用的参数类型，同样的变量（`foo`）能够格式化成不同的形式：`X`， `o` 和未指定形式。

这个格式化的功能是通过 trait 实现，并且是一种 trait 来实现各种参数类型。最常见的格式化 trait 就是 `Display`，它可以处理多种情形，但没有指明参数类型，比如 `{}`。

```
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到一个缓冲区
        // 中（第一个参数f）
        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() {
        // 一旦添加了针对 fmt::Display 的实现，则要用 {} 对输出内容进行转换
        println!("{:?}", *color)
    }
}
```

在 `fmt::fmt` 文档中可以查看全部系列的格式 traits和它们的参数类型。

动手试一试

在上面的 `Color` 结构体加上一个 `fmt::Display` 的实现，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各种颜色，
- 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，包括：

- 有符号整型（signed integers）： `i8`， `i16`， `i32`， `i64` 和 `isize` （指针 size）
- 无符号整型（unsigned integers）： `u8`， `u16`， `u64` 和 `usize` （指针 size）
- 浮点类型（floating point）： `f32`， `f64`
- `char` （字符）：单独的 Unicode 字符，如 `'a'`， `'α'` 和 `'∞'` （大小都是4个字节）
- `bool` （布尔型）：只能是 `true` 或 `false`
- 单元类型(unit type，空元组)： 只有 `()` 这个唯一值
- 数组：如 `[1, 2, 3]`
- 元组： 如 `(1, true)`

变量都能够显式地给出类型声明。数字可以通过加后缀或默认方式来额外地声明。整型默认为 `i32` 类型，浮点型默认为 `f64` 类型（译注：此说法不明确，Rust语言参考指出： 未声明类型数值的具体类型由实际使用情况推断，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，仅当使用环境无法推断时，整型数值时才断定为 `i32`，浮点数值才 断定为 `f64` ）。

```
fn main() {
    // 变量可以声明类型。
    let logical: bool = true;

    let a_float: f64 = 1.0;  // 常规声明
    let an_integer   = 5i32; // 后缀声明

    // 否则自动推断类型。
    let default_float   = 3.0; // `f64`
    let default_integer = 7;   // `i32`

    let mut mutable = 12; // 可变类型 `i32`。

    // 报错！变量的类型不可改变。
    mutable = true;
}
```

参见：

`std` 库

数据和运算符

整型 `1`，浮点型 `1.2`，字符 `'a'`，字符串 `"abc"`，布尔型 `true` 和 单元类型 `()` 可以 用数字、文字或符号的字面意义表示出来。

数字可以加上前缀 `0x`、`0o`、`0b` 分别表示十六进制数、八进制数、二进制数。

为了改善数字的可读性，可以在数字类型之间加上下划线(_)，比如： `1_000` 等同于 `1000`，`0.000_001` 等同于 `0.000001`。

我们需要告诉计算机使用到的数据类型。如前面学过的，我们使用 `u32` 后缀来表明该数据是一个 32 位 存储的无符号整数， `i32` 后缀表明数据是一个 32 位存储的带符号整数。

Rust 提供了一系列的运算符号，它们的优先级和类C语言的类似。（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

    // 短路的布尔类型的逻辑运算
    println!("true AND false is {}", true && false);
    println!("true OR false is {}", true || false);
    println!("NOT true is {}", !true);

    // 位运算符
    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的组合。元组使用括号 `()` 来构成，每个元组的值都是 `(T1, T2, ...)` 类型标记的形式，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 拥有任意数量的值。

```
// 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` 来绑定元组的各个变量
    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);

    // 通过元组的索引来访问具体的值
    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);

    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，这是为了和括号包含的普通数据作区分。
    println!("one element tuple: {:?}", (5u32,));
    println!("just an integer: {:?}", (5u32));

    // 解构元组，将值赋给创建的绑定变量
    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. 重新试一下：在上面的例子中给 Matrix `结构体` 加上 `fmt::Display` trait，让你能够将 打印调试格式 `{:?}` 切换成一般显示格式 `{}`，得到如下的输出：

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

   可以回顾之前学过的例子 打印显示。
2. 以 `reverse` 函数作为样本，添加一个 `transpose` 函数，使它可以接受一个 Matrix 的参数，并 返回一个两个元素元素交换后 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 **slice** 类型

数组是一组包含相同数据类型 `T` 的组合，并存储在连续的内存区中。数组使用中括号 `[]` 来创建， 另外它们的大小在编译期间就已确定，数组的类型标记为 `[T; size]` （译注：T 为元素的类型，size 表示数组的大小）。

slice（中文有“切片”之意） 类型和数组类似，但 slice 类型的大小在编译期间是不确定的。相反，slice 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。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() {
    // 固定大小的数组（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];

    // 索引从 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());

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

    // 数组可以自动地借用成为 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]);

    // 越界的索引会引发 panic（中文意思是：惊恐、恐慌等意）
    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
- `enum`： 定义一个枚举类型

而常量的创建可以通过 `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，总的来说是根据元组来命名。
- C 语言风格的结构体 c_struct。
- 单元结构体，不带字段，在泛型中很有用。

```
// 单元结构体
struct Nil;

// 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 {
    p1: Point,
    p2: Point,
}

fn main() {
    // 实例化结构体 `Point`
    let point: Point = Point { x: 0.3, y: 0.4 };

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

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

    let _rectangle = Rectangle {
        // 结构体的实例化也是一个表达式
        p1: Point { x: my_y, y: my_x },
        p2: point,
    };

    // 实例化一个单元结构体
    let _nil = Nil;
```

```
    // 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）的信息，包括左下角的点，以及长和宽的浮点数值。

参见：

`attributes` 和 解构

枚举

`enum` 关键字允许创建一个代表数个可能变量的数据的类型（原文：The `enum` keyword allows the creation of a type which may be one of a few different variants.若您对此句有 更好的翻译或理解，希望指出来，谢谢。）。在 `struct` 中任何合法的变量在 `enum` 同样是合法的。

```
// 隐藏未使用代码警告的属性。
#![allow(dead_code)]

// 创建一个 `enum` （枚举）来划分人的类别。注意命名和类型的信息是如何一起
// 明确规定变量的：
// `Engineer != Scientist` 和 `Height(i32) != Weight(i32)`。每者都不相同且
// 相互独立。
enum Person {
    // 一个 `enum` 可能是个 `unit-like`（类单元结构体），
    Engineer,
    Scientist,
    // 或像一个元组结构体，
    Height(i32),
    Weight(i32),
    // 或像一个普通的结构体。
    Info { name: String, height: i32 }
}

// 此函数将一个 `Person` enum 作为参数，无返回值。
fn inspect(p: Person) {
    // `enum` 的使用必须覆盖所有情形（无可辩驳的），所以使用 `match`
    // 以分支方式覆盖所有类型。
    match p {
        Person::Engineer    => println!("Is engineer!"),
        Person::Scientist       => println!("Is scientist!"),
        // 从 `enum` 内部解构 `i`
        Person::Height(i) => println!("Has a height of {}.", i),
        Person::Weight(i) => println!("Has a weight of {}.", i),
        // 将 `Info` 解构成 `name` 和 `height`。
        Person::Info { name, height } => {
            println!("{} is {} tall!", name, height);
        },
    }
}

fn main() {
    let person   = Person::Height(18);
    let amira    = Person::Weight(10);
    // `to_owned()` 从一个字符串 slice 创建一个具有所有权的 `String`。
    let dave     = Person::Info { name: "Dave".to_owned(), height: 72 };
```

```
    let rebecca  = Person::Scientist;
    let rohan    = Person::Engineer;

    inspect(person);
    inspect(amira);
    inspect(dave);
    inspect(rebecca);
    inspect(rohan);
}
```

参见：

`attributes`，`match`，`fn`，和 `String`

使用 **use**

使用 `use` 声明，这样就不必手动加上作用域了：

```
// 隐藏未使用代码警告的属性。
#![allow(dead_code)]

enum Status {
    Rich,
    Poor,
}

enum Work {
    Civilian,
    Soldier,
}

fn main() {
    // 明确地 `use` 各个名称使他们直接可用而不需要手动加上作用域。
    use Status::{Poor, Rich};
    // 自动地 `use` `Work` 内部的各个名称。
    use Work::*;

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

    match status {
        // 注意这里少了作用域，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 {
        // 再次注意到这里没有作用域。
        Civilian => println!("Civilians work!"),
        Soldier  => println!("Soldiers fight!"),
    }
}
```

参见：

`match` 和 `use`

C 风格用法

`enum` 也可以像 C 语言枚举那样使用。

```
// 隐藏未使用代码警告的属性。
#![allow(dead_code)]

// 拥有隐式辨别值（implicit discriminator）的 enum（从0开始计数）
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` 可以转成整形。
    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）：

```
use List::*;

enum List {
    // Cons：  元组结构体，包含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,
}

// 方法可以在 enum 定义
impl List {
    // 创建一个空列表
    fn new() -> List {
        // `Nil` 为 `List` 类型
        Nil
    }

    // 处理一个列表，得到一个头部带上一个新元素的同样类型的列表并返回此值
    fn prepend(self, elem: u32) -> List {
        // `Cons` 同样为 List 类型
        Cons(elem, Box::new(self))
    }

    // 返回列表的长度
    fn len(&self) -> u32 {
        // `self` 必须匹配，因为这个方法的行为取决于 `self` 的变化类型
        // `self` 为 `&List` 类型，`*self` 为 `List` 类型，一个具体的 `T` 类型的匹配
        // 要参考引用 `&T` 的匹配
        match *self {
            // 不能得到 tail 的所有权，因为 `self` 是借用的；
            // 而是得到一个 tail 引用
            Cons(_, ref tail) => 1 + tail.len(),
            // 基本情形：空列表的长度为 0
            Nil => 0
        }
    }

    // 将列表以字符串（堆分配的）的形式返回
    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。这两种常量都要显式地标注：

- `const`： 不可改变的值（常用类型）。
- `static`： 在 `'static` 生命周期内可能发生改变的变量。

有个特例就是 `"string"` 原始类型。可以给它直接赋一个不可改变的 `static` 变量，是因为它的类型标记：`&'static str` 包含了生命周期 `'static`。其他的引用类型都必须特别注明从而拥有 `'static` 生命周期。这似乎是无关紧要的，因为所需的显式标记会隐藏差异（This may seem minor though because the required explicit annotation hides the distinction.）。

```
// 在所有的作用域外声明全局变量。
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变量时标注类型。不过在多数情况下，编译器能够 从字面内容推导出变量的类型，大大减少了标注类型的负担。

使用 `let` 绑定操作可以将值（像具体数据）绑定到变量中。

```
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);

    // 编译器会对未使用变量绑定产生警告；可在变量名加上下划线的前缀来消除这些警告内容。
    let _unused_variable = 3u32;

    let noisy_unused_variable = 2u32;
    // 改正 ^ 在变量名加上下划线前缀消除警告
}
```

可变变量

变量绑定默认是不可变的，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
}
```

编译器将会抛出一堆关于变量可变性的错误提示信息。

作用域和隐藏

变量绑定有一个作用域，并且限定在一个代码块（block）中存活（live）。代码块是一个被 `{}` 包围的 语句集合。另外也允许变量隐藏。

```
fn main() {
    // 此绑定存在于 main 函数中
    let long_lived_binding = 1;

    // 这是一个代码块，比 main 函数拥有一个更小的作用域
    {
        // 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*隐藏*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*隐藏*了前面的绑定
    let long_lived_binding = 'a';

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

变量先声明

Rust 语言可以先声明变量绑定，后面才将它们初始化。但是这种情况用得很少，因为这样很可能导致使用未 初始的变量。

```
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）。

类型转换

Rust 在基本类型之间没有提供隐式类型转换（强制类型转换）（implicit type conversion，coercion ）。不过使用 `as` 关键字进行显式类型转换（explict type conversion，casting）。

一般来说，Rust 的整型类型的转换规则遵循 C 语言的惯例，除了那些在 C 语言是未定义行为的情况。在 Rust 中，所有的整型类型转换的行为都得到了很好的定义。

```
// 消除会溢出的类型转换的所有警告。
#![allow(overflowing_literals)]

fn main() {
    let decimal = 65.4321_f32;

    // 报错！不能隐式转换类型
    let integer: u8 = decimal;
    // 改正 ^ 注释掉此行

    // 显式转换类型
    let integer = decimal as u8;
    let character = integer as char;

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

    // 当将任意整数值转成无符号类型（unsigned 类型） T 时，
    // 将会加上或减去 std::T::MAX + 1，直到值符合新的类型

    // 1000 原本就符合 u16 类型
    println!("1000 as a u16 is: {}", 1000 as u16);

    // 1000 - 256 - 256 - 256 = 232
    // 在计算机底层会截取数字的低8位（the least significant bit，LSB），
    // 而高位（the most significant bit，MSB）数字会被抛掉。
    // （译注：此操作是按二进数存储的数字位进行）
    println!("1000 as a u8 is : {}", 1000 as u8);
    // -1 + 256 = 255
    println!("  -1 as a u8 is : {}", (-1i8) as u8);

    // 对正数来说，上面的类型转换操作和取模效果一样
    println!("1000 mod 256 is : {}", 1000 % 256);

    // 当将整数值转成有符号类型（signed 类型）时，同样要先将（二进制）数值
    // 转成相应的无符号类型（译注：如 i32 和 u32 对应，i16 和 u16对应），
    // 然后再求此值的补码（two's complement）。如果数值的最高位是 1，则数值
    // 是负数。
```

```
    // 除非值本来就已经符合所要转的类型。
    println!(" 128 as a i16 is: {}", 128 as i16);
    // 128 as u8 -> 128，再求数字128的8位二进制补码得到：
    println!(" 128 as a i8 is : {}", 128 as i8);

    // 重复上面的例子
    // 1000 as u8 -> 232
    println!("1000 as a i8 is : {}", 1000 as i8);
    // 232 的补码是 -24
    println!(" 232 as a i8 is : {}", 232 as i8);
}
```

字面量

数字字面量可以加上类型标记作为后缀来标注类型。举个例子，要指定字面量 `42` 为 `i32` 类型，可以写成 `42i32`。

未加上后缀的数字字面量的类型视使用它们的情况而定。如果没有限定，编译器会将整型定为 `i32` 类型，将浮点数定为 `f64` 类型。

```
fn main() {
    // 有后缀的字面量，它们的类型在初始化时就确定
    let x = 1u8;
    let y = 2u32;
    let z = 3f32;

    // 无后缀的字面量，它们的类型视使用情况而定
    let i = 1;
    let f = 1.0;

    // `size_of_val` 返回变量的大小，以字节（byte）为单位
    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));
}
```

前面代码中用了一些尚未解释过的概念，这里列出一些简短的说明：

- `fun(&foo)` 是通过引用传参给一个函数，而不是通过值来传参（`fun(foo)`）。更多内容参见 借用（borrowing）。
- `std::mem::size_of_val` 是一个函数，不过是通过完整的路径调用的。代码可以划分到称为模块（module）的逻辑单元中。在这个例子中，`size_of_val` 函数是定义在 `mem` 模块的，`mem` 模块是定义在 `std` 包（crate）中。更多内容参考模块 和 crate。

类型推导

类型推导引擎是相当智能的。它不仅仅在初始化期间分析右值的类型，还会通过分析变量在后面是怎么使用的来推导该变量的类型。这里给出一个类型推导的高级例子：

```
fn main() {
    // 借助类型标注，编译器知道 `elem` 具有 u8 类型。
    let elem = 5u8;

    // 创建一个空 vector（可增长数组）。
    let mut vec = Vec::new();
    // 此时编译器并未知道 `vec` 的确切类型，它只知道 `vec` 是一个含有某种类型
    // 的 vector（`Vec<_>`）。

    // 将 `elem` 插入 vector。
    vec.push(elem);
    // Aha! 现在编译器就知道了 `vec` 是一个含有 `u8` 类型的 vector(`Vec<u8>`)
    // 试一试 ^ 尝试将 `vec.push(elem)` 那行注释掉

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

无需变量的类型标注，编译器和程序员都很开心（the compiler is happy and so is the programmer）！

别名

`type` 语句可以给一个已存在类型起一个新的名字。类型必须要有 `CamelCase` （驼峰方式）的名称，否则 编译器会产生一个警告。对规则为例外的是基本类型： `usize` ， `f32` 等等。

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

// 使用一个属性来忽略警告。
#[allow(non_camel_case_types)]
type u64_t = u64;
// 试一试 ^ 试着删掉属性。

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

    // 注意类型的别名*没有*提供任何额外的类型安全，因为别名*不是*新的类型
    println!("{} nanoseconds + {} inches = {} unit?",
             nanoseconds,
             inches,
             nanoseconds + inches);
}
```

别名的主要作用是减少按键，举个例子， `IoResult<T>` 类型是 `Result<T, IoError>` 类型的别名。

参见：

属性

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：一种是绑定变量，另一种是表达式带上分号：

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

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

代码块也是表达式，所以它们在赋值操作中可以充当右值（r-values）。代码块中的最后一条 表达式将赋给左值（l-value）。需要注意的是，如果代码块最后一条表达式结尾处有分号，那 么返回值将变成 `()`。（译注：代码块中的最后一条语句是代码块中实际执行的最后一条语句，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。让我们讲述 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。与很多语言不同的是，Rust 语言中的布尔判断条件不用小括号包住， 每个判断条件后连着一个代码块。 `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");

            // 这条表达式返回一个 `i32` 类型。
            10 * n
        } else {
            println!(", and is a big number, reduce by two");

            // 这条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` 关键字来实现一个无限循环。

可以使用 `break` 语可以在任何时刻退出一个循环，另外可用 `continue` 跳过迭代的剩余部分并重新开始 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 `中断（break）` 或 `继续（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，并由 `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` 循环写一个不怎么出名的 FizzBuzz 程序。

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

    // 当 `n` 小于 101 时进入循环操作
    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 循环和区间

`for in` 结构可以通过一个计数器来迭代。创建计算器的一个最简便的方法就是使用区间标记 `a..b` 。这 会生成从 `a` （包含此值） 到 `b` （不含此值）增幅为 1 的一系列值。

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

```
fn main() {
    // `n` 将从 1, 2, ..., 100 这些值依次获取进行每次循环
    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);
        }
    }
}
```

参见：

Iterator

match 匹配

Rust 通过 `match` 关键字来提供模式匹配，用法和 C 语言的的 `switch` 类似。

```
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"),
        // 匹配一个闭区间范围
        13...19 => println!("A teen"),
        // 处理其他情况
        _ => println!("Ain't special"),
    }

    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 ^ 试着将其中一条分支注释掉
    };

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

解构

`match` 代码块可以以多种方式解构内容。

元组

元组可以在 `match` 中解构，如下所示：

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

    println!("Tell me about {:?}", pair);
    // match 可以解构一个元组
    match pair {
        // 绑定到第二个元素
        (0, y) => println!("First is `0` and `y` is `{:?}`", y),
        (x, 0) => println!("`x` is `{:?}` and last is `0`", x),
        _      => 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` 方式如下：

```
// 需要 `allow` 来消除警告，因为只使用了一个变量。
#[allow(dead_code)]
enum Color {
    // 这三者仅由它们的名字来表示。
    Red,
    Blue,
    Green,
    // 这些元组含有类似的 `u32` 元素，分别对应不同的名字：颜色模型（color models）。
    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(...)]`, color models 和 `enum`

指针和引用

对指针来说，解构（destructuring）和解引用（dereferencing）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`

```
fn main() {
    // 获得一个 `i32` 类型的引用。`&` 表示获取一个引用。
    let reference = &4;

    match reference {
        // 如果 `reference` 是对 `&val` 进行模式匹配，则会产生如下比较行为：
        // `&i32`
        // `&val`
        // ^ 我们看到，如果匹配的 `&` 都去掉了，那么就是 `i32` 赋给 `val`。
        &val => println!("Got a value via destructuring: {:?}", val),
    }

    // 为了避免 `&` 的使用，需要在匹配前解引用。
    match *reference {
        val => println!("Got a value via dereferencing: {:?}", val),
    }

    // 如果没有一个引用头部（以 & 开头）会是怎样？ `reference` 是一个 `&`，
    // 因为右边已经是一个引用。
    // 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，使得可以对具体值
    // 创建引用。这将得到一个引用。
    let ref _is_a_reference = 3;

    // 相应地，定义两个非引用的值，通过 `ref` 和 `mut` 可以取得引用。
    let value = 5;
    let mut mut_value = 6;

    // 使用 `ref` 关键字来创建引用。
    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 => {
```

```
            // 获得一个引用。在增加内容之前，要先得到解引用（Gotta
            // dereference it before we can add anything to it）。
            *m += 10;
            println!("We added 10. `mut_value`: {:?}", m);
        },
    }
}
```

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，成员顺序是不重要的

    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

守卫

可以加上 `match` 守卫（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 condition`(if 条件)部分是一个守卫
        (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：

Tuples

绑定

间接地访问一个变量不可能在分支中使用这个没有重新绑定的变量。 `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 type of person you are");

    match age() {
        0             => println!("I'm not born yet I guess"),
        // 不能直接 `匹配（match）` 1 ... 12，但是孩子是几岁呢？
        // 相反，将 1 ... 12 序列绑定到 `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),
        // 没有绑定。返回结果。
        n             => println!("I'm an old person of age {:?}", n),
    }
}
```

参见：

函数

if let

在一些例子中，`match` 使用起来并不优雅。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个缩进，就这样可以从 option 类型中对 `i`
        // 进行解构
    },
    _ => {},
    // ^ 必需内容，因为 `match` 需要覆盖全部情况。难道不觉得冗余吗？
};
```

`if let` 对这样的用法要简洁得多，并且允许指明特定的各种不同的失败可选项 内容（options）：

```
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` 结构解读：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。换到失败情形（Change to the failure case）。
        println!("Didn't match a number. Let's go with a letter!");
    };

    // 提供一个改变的失败条件（Provide an altered failing condition）。
    let i_like_letters = false;

    if let Some(i) = emoticon {
        println!("Matched {:?}!", i);
```

```
    // 解构失败。执行 `else if` 条件来判断轮到的失败分支是否需要执行
    } else if i_like_letters {
        println!("Didn't match a number. Let's go with a letter!");
    } else {
        // 条件执行错误。这是默认的分支：
        println!("I don't like letters. Let's go with an emoticon :)!");
    };
}
```

参见：

`枚举`， `Option`，和 RFC

while let

和 `if let` 类似， `while let` 会产生更加难看的 `match` 的一连串内容。 考虑下面的有关增量 `i` 的一连串内容：

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

// Repeatedly try this test.
// 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);
            }
            // ^ 需要三个缩进！
        },
        // 当解构失败时退出循环：
        _ => { break; }
        // ^ 为什么要这样的语句呢？肯定有更优雅的处理方式！
    }
}
```

使用 `while let` 可以使这一连串内容变得更加优雅：

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

    // 分析：当 `let` 将 `optional` 解构成 `Some(i)` 时，就
    // 执行语句块（`{}`）。否则中断退出（`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` 有额外可选的 `else`/`else if` 分句，
    // 而 `while let` 没有。
}
```

参见：

枚举， Option，和 RFC

函数

函数使用 `fn` 关键字来声明。函数的参数需要标注类型，就和变量一样，另外如果 函数返回一个值，返回类型必须在箭头 `->` 之后特别指出来。

函数最后的表达式将作为返回值。或者在函数内使用 `return` 语句来提前返回值， 甚至在循环或 `if` 内部使用。

让我们使用函数来重写 FizzBuzz 函数吧！

```
// 和 C/C++ 不一样，Rust 的函数定义位置是没有限制的
fn main() {
    // 我们在这里使用函数，并在后面的其他位置定义它
    fizzbuzz_to(100);
}

// 函数返回一个布尔（boolean）值
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    // 极端情况，提前返回（Corner case, early return）
    if rhs == 0 {
        return false;
    }

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

// 函数不返回值，而实际上是返回一个单元类型 `()`
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，可以从标记中删除返回类型
fn fizzbuzz_to(n: u32) {
    for n in 1..n + 1 {
        fizzbuzz(n);
    }
}
```

方法

方法是从属于对象的函数(Mathods are functions attached to objects)。这些方法通过 关键字 `self` 来访问对象中的数据和其他方法。方法在 `impl` 代码块下定义。

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

// 实现的代码块，所有的 `Point` 方法都在这里给出
impl Point {
    // 这是一个静态方法（static method）
    // 静态方法不需要通过实例来调用
    // 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，带有两个参数：
    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }
}

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

impl Rectangle {
    // 这是实例方法（instance method）
    // `&self` 是 `self: &Self` 的语法糖（sugar），其中 `Self` 是所调用对象
    // 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对象是可变的
    // `&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` 含有的资源：两个堆分配的整型
struct Pair(Box<i32>, Box<i32>);

impl Pair {
    // 这个方法“消费”调用者对象的资源
    // `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 {
        // 静态方法使用双重冒号来调用
        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，比如：
    // `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 ^ 将此行注释去掉

    // 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“消费了” `pair`
    //pair.destroy();
    // 试一试 ^ 将此行注释去掉
}
```

闭包

闭包（closure）在 Rust 中也称为 lambda，是一类捕获封闭环境的函数。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）使用相当方便。调用一个闭包和调用一个函数完全相同。然而，输入和返回类型两者都可以自动推导，且输入变量名必须指明。

其他的特点包括：

- 使用 `||` 替代 `()` 将输入变量括起来。
- 区块定界符（ `{}` ）对于单条表达式是可选的，其他情况必须加上。
- 有能力捕获到外部环境变量。

```
fn main() {
    // 通过闭包和函数实现增量。
    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 `{}` 来围住代码都是可选的。
    // 这些匿名函数（nameless function）赋值给合适命名的变量。
    let closure_annotated = |i: i32| -> i32 { i + 1 };
    let closure_inferred  = |i     |          i + 1  ;

    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

闭包本身是相当灵活的，可以实现所需功能来让闭包运行而不用类型标注（原文：Closures are inherently flexible and will do what the functionality requires to make the closure work without annotation）。这允许变量捕获灵活地适应使用 情况，有时是移动（moving）有时是借用（borrowing）（原文：This allows capturing to flexibly adapt to the use case, sometimes moving and sometimes borrowing.）。闭包可以捕获变量：

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

它们更倾向于通过引用来捕获变量并且只在需要时才用后面用法（原文：They preferentially capture variables by reference and only go lower when required.）。

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

    let color = "green";

    // 闭包打印 `color`，它会马上借用（`&`）`color` 并将该借用和闭包存储
    // 到 `print` 变量中。它会一保持借用状态直到 `print` 离开作用域。
    // `println!` 只需要`通过引用`，所以它没有采用更多任何限制性的内容。
    // （原文：`println!` only requires `by reference` so it doesn't
    // impose anything more restrictive.）
    let print = || println!("`color`: {}", color);

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

    let mut count = 0;

    // 闭包使 `count` 值增加，可以使用 `&mut count` 或者 `count`，
    // 但 `&mut count` 限制更少，所以采用它。立刻借用 `count`。
    // （原文： A closure to increment `count` could take either
    // `&mut count` or `count` but `&mut count` is less restrictive so
    // it takes that. Immediately borrows `count`.）
    //
    // `inc` 前面需要加上 `mut`，因为 `&mut` 会必存储的内部。
    // 因此，调用该闭包转变成需要一个 `mut` 的闭包。
    // （原文：A `mut` is required on `inc` because a `&mut` is stored
    // inside. Thus, calling the closure mutates the closure which requires
    // a `mut`.）
    let mut inc = || {
        count += 1;
```

```
        println!("`count`: {}", count);
    };

    // 调用闭包。
    inc();
    inc();

    //let reborrow = &mut count;
    // ^ 试一试：  将此行注释去掉。

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

    // `mem::drop` requires `T` so this must take by value. A copy type
    // would copy into the closure leaving the original untouched.
    //
    // `mem::drop` 要求 `T`，所以这必须通过值来实现（原文：`mem::drop`
    // requires `T` so this must take by value.）。可复制类型将会复制
    // 值到闭包而不会用到原始值。不可复制类型必须移动（move），从而
    // `可移动`（movable） 立即移动到闭包中（原文：A non-copy must
    // move and so `movable` immediately moves into the closure）。
    let consume = || {
        println!("`movable`: {:?}", movable);
        mem::drop(movable);
    };

    // `consume` 消费（consume）了该变量，所以这只能调用一次。
    consume();
    //consume();
    // ^ 试一试：将此行注释去掉。
}
```

参见：

`Box` 和 `std::mem::drop`

作为输入参量

虽然 Rust 在捕获临时变量的方式大多选择不带标注，但在编写函数时，这种不确定性是不允许的。当以闭包作为输入参数时，闭包的完整类型必须使用以下的其中一种 `trait` 来标注。它们的受限程度依次递减，依次是（原文：In order of decreasing restriction, they are）：

- `Fn`：闭包需要通过引用（ `&T` ）捕获
- `FnMut`：闭包需要通过可变引用（ `&mut T` ）捕获
- `FnOnce`：闭包需要通过值（ `T` ）捕获

在值传值（variable-by-variable）的基础上，编译器将以限制最少的方式来捕获变量。

例如考虑一个标注为 `FnOnce` 的参量。这意味着闭包可能通过 `&T`， `&mut T` 或 `T` 来捕获，但是编译器将根据所捕获变量在闭包的使用情况做出最终选择。

这是因为若移动语义（move）可能的话，则任意借用类型也应该是可行的。注意反过来就不再成立：如果参量是 `Fn`，那么通过 `&mut T` 或 `T` 捕获的情况就不允许了。

在下面的例子中，试着换换 `Fn`、`FnMut` 和 `FnOnce` 的使用，看看会发生什么：

```
// 将闭包作为参数并调用它的函数。
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();
}

// 使用闭包并返回一个 `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` 从借用的数据创建属于自己的数据。
    let mut farewell = "goodbye".to_owned();

    // 捕获 2 个变量：通过引用方式的 `greeting` 和
    // 通过值方式的 `farewell`。
```

```
    let diary = || {
        // `greeting` 使用引用方式：需要 `Fn`。
        println!("I said {}.", greeting);

        // 改变迫使 `farewell` 变成了通过可变引用来捕获。
        // （原文：Mutation forces `farewell` to be
        // captured by mutable reference.）
        // 现在需要 `FnMut`。
        farewell.push_str("!!!");
        println!("Then I screamed {}.", farewell);
        println!("Now I can sleep. zzzzz");

        // 手动调用 drop 将 `farewell` 强制转成通过值来捕获。
        // （原文：Manually calling drop forces `farewell` to
        // be captured by value. Now requires `FnOnce`.）
        // 现在需要 `FnOnce`。
        mem::drop(farewell);
    };

    // 调用处理闭包的函数（原文：Call the function
    // which applies the closure）。
    apply(diary);

    // `double` 满足 `apply_to_3` 的 trait 限定。
    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

闭包从封闭的作用中域捕获变量简单明了。这样会有某些后果吗？当然会。观察一下使用闭包作为函数参量的方式是要求为泛型的，它们定义的方式决定了这是必要的（原文：Observe how using a closure as a function parameter requires generics, which is necessary because of how they are defined）：

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

当定义一个闭包时，编译器将隐式地创建一个新的匿名结构体来存储内部的捕获变量， 同时针对此未知类型通过其中的一种 `trait`： `Fn`， `FnMut`，或 `FnOnce` 来实现功能 （原文：implementing the functionality via one of the `traits`：`Fn`, `FnMut`, or `FnOnce` for this unknown type）。这个类型被赋给所存储的变量直到调用（原文： This type is assigned to the variable which is stored until calling）。

由于这个新类型是未知的类型，所以在函数中的任何用法都要求是泛型。然而， 未限定的类型参量 `<T>` 仍然是不明确的并且是不允许的。因此通过其中一种 `trait`： `Fn`， `FnMut`，或 `RnOnce`（已经实现）就足以指明它的类型。

```
// `F` 必须针对一个没有输入参数和返回值的闭包实现 `Fn`
// —— 确切地讲是 `print` 要求的类型。
fn apply<F>(f: F) where
    F: Fn() {
    f();
}

fn main() {
    let x = 7;

    // 捕获的 `x` 成为一个匿名类型并为它实现 `Fn`。
    // 将它存储到 `print` 中。
    let print = || println!("{}", x);

    apply(print);
}
```

参见：

透彻分析，`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参数传递。

```
// 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` 限定的装包函数（wrapper function）。
fn function() {
    println!("I'm a function!");
}

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

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

额外说明，`Fn`，`FnMut`，和 `FnOnce` 这些 `trait` 明确了闭包如何从封闭的作用域中捕获变量。

参见：

`Fn`，`FnMut`，和 `FnOnce`

作为输出参量

闭包作为输入参量是可能的，所以返回闭包作为输出参量（output parameter）也应该是可能的。然而返回的闭包类型会有问题，因为目前的 Rust 只支持返回具体（非泛型）的类型。按照定义匿名的闭包类型是未知的，所以想要返回一个闭包只有使它变成具体的类型。通过 box 操作可以实现这点。

关于返回值的有效的 trait 和前面的略有不同：

- `Fn`：和前面的一样（normal）
- `FnMut`：和前面的一样
- `FnOnce`：这里运行会有些独特的地方（There are some unusual things at play here），所以目前需要 `FnBox` 类型，这属于不稳定的内容。此处预计将来会发生改变。

除此之外，还必须使用 `move` 关键字，它表明了通过值来产生全部的捕获（which signals that all captures occur by value）。这是必需的，因为在函数退出的同时任何通过引用捕获的值将被丢弃（dropped），在闭包中留下无效的引用。

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

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

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

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

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

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

参见：

Boxing, `Fn`, `FnMut`, 和 泛型.

`std` 中的例子

本小节列出几个标准库中使用闭包例子。

Iterator::any

`Iterator::any` 是一个函数，在处理一个迭代器（iterator）时，当其中任一元素符合条件（predicate）时将返回 `true`，否则返回 `false`。它的原型如下：

```
pub trait Iterator {
    // 迭代相关的类型（原文：The type being iterated over）。
    type Item;

    // `any` 接受 `&mut self` 作为调用者可能被借用和修改，但不会消费掉。
    // （原文： `any` takes `&mut self` meaning the caller may be
    // borrowed and modified, but not consumed.）
    fn any<F>(&mut self, f: F) -> bool where
        // `FnMut` 表示任意捕获变量很可能都被修改，而非消费。
        // `Self::Item` 表明了通过值来接受闭包类型参数。
        // （原文：`FnMut` meaning any captured variable may at
        // most be modified, not consumed. `Self::Item` states it
        // takes arguments to the closure by value.）
        F: FnMut(Self::Item) -> bool {}
}
```

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

    // 对 vec 的 `iter()` 产出 `&i32`（原文：`iter()` for vecs yields
    // `&i32`）。解构成 `i32` 类型。
    println!("2 in vec1: {}", vec1.iter()     .any(|&x| x == 2));
    // 对 vec 的 `into_iter()` 产出 `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（array）的 `iter()` 产出 `&i32`。
    println!("2 in array1: {}", array1.iter()     .any(|&x| x == 2));
    // 对数组的 `into_iter()` 通常产出 `&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，在处理一个迭代器（iterator）时，将返回第一个满足条件的元素作为一个 `Option` 类型。它的原型如下：

```
pub trait Iterator {
    // 迭代相关的类型。
    type Item;

    // `find` 接受 `&mut self` 作为调用者可能被借用和修改，但不会消费掉。
    // （原文：`find` takes `&mut self` meaning the caller may be borrowed
    // and modified, but not consumed.）
    fn find<P>(&mut self, predicate: P) -> Option<Self::Item> where
        // `FnMut` 表示任意捕获变量很可能都被修改，而非消费。
        // `&Self::Item` 表明了通过引用接受闭包类型的参数。
        // （原文：`FnMut` meaning any captured variable may at most be
        // modified, not consumed. `&Self::Item` states it takes
        // arguments to the closure by reference.）
        P: FnMut(&Self::Item) -> bool {}
}
```

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

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

    // 产出内容的引用是 `&&i32` 类型。解构成 `i32` 类型。
    println!("Find 2 in vec1: {:?}", iter     .find(|&&x| x == 2));
    // 产出内容的引用是 `&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 `iter()`  产出 `&i32`。
    println!("Find 2 in array1: {:?}", array1.iter()     .find(|&&x| x == 2));
    // 对数组的 `into_iter()` 通常产出 `&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）。执行一个或多个函数来产生一个用处更大的函数。HOF 和惰性迭代器（lazy iterator）给 Rust 带来了函数式的风格（英文原文：HOFs and lazy iterators give Rust its functional flavor.）。

```
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;

    // 命令式方式（imperative approach）
    // 声明累加器变量
    let mut acc = 0;
    // 重复：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（upper limit）则退出循环
            break;
        } else if is_odd(n_squared) {
            // 如果是奇数就累加值
            acc += n_squared;
        }
    }
    println!("imperative style: {}", acc);

    // 函数式方式（functional approach）
    let sum_of_squared_odd_numbers: u32 =
        (0..).map(|n| n * n)             // 所有自然数的平方
             .take_while(|&n| n < upper) // 小于上限
             .filter(|&n| is_odd(n))     // 为奇数
             .fold(0, |sum, i| sum + i); // 最后其后
    println!("functional style: {}", sum_of_squared_odd_numbers);
}
```

Option 和 迭代器 实现了它们自己的高阶函数（英语原文：Option and Iterator implement their fair share of HOFs.）。

模块

Rust 提供了一套强大的模块系统，可以将代码按层次分成多个逻辑单元（模块），并在这些模块之间管理可见性（公开 public/私有 private）。

模块是一系列项的集合：函数，结构体，trait，`impl` 块，甚至其它模块。

可见性

项（item）默认情况下拥有私有的可见性（private visibility），不过可以加上 `pub` （public 的前 3 个字母）修饰语（modifier）来改变默认行为。一个模块之外的作用域只能访问该模块里面的公有项（public item）。

```
// 一个名为 `my` 的模块
mod my {
    // 在模块中的项默认带有私有可见性。
    fn private_function() {
        println!("called `my::private_function()`");
    }

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

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

    // 项也可以嵌套。
    pub mod nested {
        pub fn function() {
            println!("called `my::nested::function()`");
        }

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

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

fn function() {
```

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

fn main() {
    // 模块允许在拥有相同名字的项之间消除歧义。
    function();
    my::function();

    // 公有项，包括内部嵌套的公有项，可以在父级的模块中访问到。
    my::indirect_access();
    my::nested::function();

    // 一个模块中的私有项不能被直接访问，即使私有项嵌套在公有的模块中：

    // 报错！`private_function` 是私有的。
    //my::private_function();
    // 试一试 ^ 将此行注释去掉

    // 报错！ `private_function` 是私有的。
    //my::nested::private_function();
    // 试一试 ^ 将此行注释去掉

    // 报错！ `private_nested` 是私有的模块。
    //my::private_nested::function();
    // 试一试 ^ 将此行注释去掉

}
```

结构体的可见性

结构体对字段的可见性有额外的规定（Structs have an extra level of visibility with their fields）。其可见性默认为私有，也可以加上 `pub` 修饰语来改变默认属性。只有当从定义在外部的模块访问一个结构体时，这可见性才显得重要，并具有隐藏信息的目的（封装，encapsulatoin）（原文：This visibility only matters when a struct is accessed from outside the module where it is defined, and has the goal of hiding information (encapsulation)）。

```
mod my {
    // 一个公有的结构体，带有一个公有的泛型类型 `T` 的字段
    pub struct WhiteBox<T> {
        pub contents: T,
    }

    // 一个公开的结构体，带有一个私有的泛型类型 `T` 的字段
    #[allow(dead_code)]
    pub struct BlackBox<T> {
        contents: T,
    }

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

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

    // 并且它们的字段可以正常访问到。
    println!("The white box contains: {}", white_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！`BlackBox` 含有私有字段。
    //let black_box = my::BlackBox { contents: "classified information" };
    // 试一试 ^ 将此行注释去掉


    // 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 _black_box = my::BlackBox::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 black box contains: {}", _black_box.contents);
    // 试一试 ^ 将此行注释去掉

}
```

参见：

泛型 and 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()` 将覆盖掉的外部同名函数。
        use deeply::nested::function;
        function();

        // `use` 绑定拥有局部作用域。在这个例子中，`function()`
        // 的覆盖只限定在这个代码块中。
        println!("Leaving block");
    }

    function();
}
```

`super` 和 `self`

在路径上使用 `super` （父级）和 `self` （自身）关键字，可以在访问项时消除歧义和防止不必要的路径的硬编码。

```
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()` 和直接访问 `function()` 两者都得到相同的结果，
        // 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` 关键字表示父级作用域（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 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。让我们将可见性小节例子 的代码拆开分到多个文件中：

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

在 `split.rs` 文件：

```
// 此声明将会查找名为 `my.rs` 或 `my/mod.rs` 的文件，并将该文件的内容插入到
// 此作用域名为 `my` 的模块里面。
mod my;

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

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

    function();

    my::indirect_access();

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

在 `my/mod.rs` 文件：

```
// 类似地，`mod inaccessible` 和 `mod nested` 将找到 `nested.rs` 和
// `inaccessible.rs` 文件，并在它们各自的模块中插入它们的内容。
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();
}
```

在 `my/nested.rs` 文件：

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

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

在 `my/inaccessible.rs` 文件：

```
#[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 中的编译单元。不管什么时候调用 `rustc some_file.rs`， `some_file.rs` 都被当作 **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运行编译器之前与 crate 文件合并。换句话说， 模块不会单独进行编译，只有 crate 文件进行了编译（英文：modules do *not* get compiled individually, only crates get compiled）。

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 产生库文件。这种行为可以通过 `rustc` 的选项 `--crate-type` 覆盖。

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
```

库的前缀为 “lib”，默认情况下它们跟随着 crate 文件命名（原文：by default they get named after their crate file），但此默认名称可以使用 `crate_name` 属性 覆盖。

extern crate

链接一个 crate 到这个新库，必须使用 `extern crate` 声明。这不仅会链接库，还会导入与库名相同的模块里面的所有项。适用于模块的可见性规则也适用于库。

```
// 链接到 `library`（库），导入 `rary` 模块里面的项
extern crate rary;

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，假设在这里它在同一目录下：
# （原文：Where library.rlib is the path to to the compiled library,
# assumed that it's in the same directory here:）
$ rustc executable.rs --extern rary=library.rlib && ./executable
called rary's `public_function()`
called rary's `indirect_access()`, that
> called rary's `private_function()`
```

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：

- 代码的条件编译
- 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（unit test）
- 标记作为基准某个部分的函数

当属性用于一个完整的 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)]`

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。可以加上属性来抑制这个 lint。

```
fn used_function() {}

// `#[allow(dead_code)]` 属性可以抑制 `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。在这些例子中，我们是出于知识点讲解的需要才特意加上了一些死代码。

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_time` 属性可以设定 crate 的名称。

```
// 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：

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

两种形式使用参数的语法都相同。

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

// 而这个函数仅当操作系统**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` 在使用 `rustc` 时会隐式地提供，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();
}
```

不使用自定义的 `cfg` 标记：

```
$ 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）可以泛化类型和功能，以扩大适用范围。减少代码的重复是相当重要的，这可以通过多种方式实现，不过需要相当繁琐的语法。也就是说，用到泛型需要特别谨慎地指出哪种类型对于泛型类型来说是有效的。使用泛型最简单且最常见的方式就是用到类型参量（type parameter）。（本段原文： *Generics* is the topic of generalizing types and functionalities to broader cases. This is extremely useful for reducing code duplication in many ways, but can call for rather involving syntax. Namely, being generic requires taking great care to specify over which types a generic type is actually considered valid. The simplest and most common use of generics is for type parameters.）

类型参量指定为泛型要使用尖括号和 CamelCase（驼峰式命名）： `<Aaa, Bbb, ...>` 。“泛型类型参量”一般用 `<T>` 来表示。在 Rust 中，“泛型”也表示可以接受一个或多个泛型类型参量 `<T>` 的任何内容。任何指定为泛型类型参量的类型都是泛型，其他的都是具体类型（非泛型）。

例如定义一个名为 `foo` 的 泛型函数，可接受一个任意类型的参数 `T` ：

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

因为 `T` 被指定为一个使用 `<T>` 的泛型类型参量，所以在这里用到的 `(T)` 会变成泛型 。即使 `T` 在前面被定义为 `struct` 也是如此。

下面例子展示了一些操作中的语法：

```
// 具体的类型 `A`。
struct A;

// 在定义类型 `Single` 时，在 `A` 的首次使用之前没有出现 `<A>`。
// 因此，`Single` 是一个具体的类型，`A` 在上面已经定义。
// （原文：In defining the type `Single`, the first use of `A` is not preceded
// by `<A>`. Therefore, `Single` is a concrete type, and `A` is defined as above.）
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。
// 因为类型参量 `T` 是泛型，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` 是具体类型并显式地接受 `A`。
    let _s = Single(A);

    // 创建一个 `SingleGen<char>` 类型的变量 `_char`，并给一个 `SingleGen('a')` 值。
    // 这里的 `SingleGen` 拥有显式指定的类型参量。
    let _char: SingleGen<char> = SingleGen('a');
```

```
    // `SingleGen` 也可以拥有隐式指定的类型参量：
    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

同样的规则也可以适用于函数：在使用前给出 `<T>` 后，类型 `T` 就变成了泛型。

使用泛型函数有时需要显式地指明类型参量。这种可能的情况包括，调用返回类型是泛型的函数，或者编译器没有足够的信息来推导类型参量。

函数调用使用显式指定的类型参量，如下所示： `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，传递给函数的变量在离开作用域时立即释放。
// （原文：The following functions all take ownership of the variable passed
// into them and immediately go out of scope, freeing the variable.）

// 定义一个函数 `reg_fn`，接受一个 `S` 类型的参数 `_s`。
// 因为没有 `<T>`，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<T>` 类型的参数 `_s`。
// 这里显式地给出了类型参量 `A`，但因为 `A` 没有被指明为针对 `gen_spec_t` 的
// 泛型类型参量，所以这不是一个泛型。
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`。
// 这里显式地给出了类型参量 `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。
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>` 之前给定了 `<T>`，所以这个函数是关于 `T` 的泛型。
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));   // 隐式地指定类型参量 `A`。
    gen_spec_i32(SGen(6)); // 隐式地指定类型参量 `i32`。

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

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

参见：

函数 和 `struct S`

实现

和函数类似，实现（implementation）也需要关注保持泛型。（原文：Similar to functions, implementations require care to remain generic.）

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

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

// `<T>` 必须在类型之前给出来以保持泛型。
// （原文：`<T>` Must precede the type to remain generic）
impl <T> GenericVal<T> {}
```

```
struct Val {
    val: f64
}

struct GenVal<T>{
    gen_val: T
}

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

// GenVal 针对泛型类型 `T` 的实现
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：

函数返回引用, `impl` , 和 `struct`

特性 trait

当然 `trait` 也可以是泛型。我们在这里定义了一个实现 `Drop` 的 `trait`，作为泛型方法来 `drop`（丢弃）它本身和输入参数。

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

// 用到 `T` 的trait 泛型。
trait DoubleDrop<T> {
    // 定义一个关于调用者的方法，接受一个额外的单一参量 `T`，
    // 且没有任何操作。
    fn double_drop(self, _: T);
}

// 针对泛型参量 `T` 和调用者 `U` 实现了 `DoubleDrop<T>` 。
impl<T, U> DoubleDrop<T> for U {
    // 此方法获得了两个传入参数的所有权，并释放这两个参数。
    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`

限定

在运用泛型时，类型参量常常必须使用 trait 作为限定（bound）来明确规定一个类型实现了哪些功能。例如下面的例子用到了 `Display` trait 来打印，所以它要求 `T` 由 `Display` 限定，也就是说 `T` 必须实现 `Display` 。

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

限定限制了泛型为符合限定的类型。即：

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

// 报错！`Vec<T>` 未实现 `Display`。
// 此特例化将失败。
let s = S(vec![1]);
```

限定的另一个作用是泛型实例允许访问在指定在限定中的 trait 的方法。例如：

```
// 这个 trait 实现了打印标记：`{:?}`。
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`。不管什么类型，都可以正常工作。
fn print_debug<T: Debug>(t: &T) {
    println!("{:?}", t);
}

// `T` 必须实现 `HasArea`。任意符合限定的函数都可以访问
```

```
// `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：将上述语句的注释去掉。
    // | 报错：未实现 `Debug` 或 `HasArea`。
}
```

额外补充内容，某些情况下为了提高代码的表现力， `where` 从句也可以在限定上使用。

参见：

`std::fmt` , `struct` , 和 `trait`

测试实例：空限定

限定的工作机制有一个效果是，即使一个 `trait` 不包含任何功能，你仍然可以使用它作为一个限定。在标准库中的 `Eq` 和 `Ord` 就是这样的例子。

```
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。实际情况中 trait 内部
// 是否为空都无所谓。
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;

    // 由于限定，`red()` 不能调用 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：将此行注释去掉。
}
```

参见：

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

多重限定

使用多重限定（multiple bounds）可以用 `+` 连接。和平常一样，不同的类型使用 `,` 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 从句

限定也可以使用 `where` 从句来表达，这样可以让限定写在 `{` 紧邻的前面，而不需写在类型第一次提到的位置上。另外 `where` 从句可以用于任意类型的限定，而不局限于类型参量。

`where` 在一些情况下有很用：

- 当分开指定泛型类型和限定时更清晰情况：

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

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

- 当使用 `where` 从句比正常语法更富表现力的情况。要是没有 `where` 从句的话，例子中的 `impl` 就不能直接表达出来：

```
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`
// 或使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` 作为限定，因为那是要打印的内容。
    // 不这样做的话，很可能就用到错误的限定。
    fn print_in_option(self) {
        println!("{:?}", Some(self));
    }
}

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

    vec.print_in_option();
}
```

参见：

RFC, `struct`, 和 `trait`

关联项

“关联项”（associted items）是指一系列有关各种变量类型的 `item`（项） 的规则。它是 `trait` 泛型的扩展（extension），允许 `trait` 在内部定义新的项。

关联类型（*associated type*）就是这种项的其中一个。当 `trait` 在其容器类型（container type）上是泛型时，关联类型提供了更简单的使用模式。（原文：One such item is called an *associated type*, providing simpler usage patterns when the `trait` is generic over its container type.）

参见：

RFC

存在问题

对容器类型为泛型的 `trait` 有类型规范需要—— `trait` 的成员必须指出全部关于它的泛型类型。

在下面例子中， `Contains` `trait` 允许使用泛型类型 `A` 或 `B` 。然后这个 trait 针对 `Container` 类型实现，指定 `i32` 为 `A` 和 `B` ，因而它可以用到 `fn difference()` 。（本段原文：In the example below, the `Contains` `trait` allows the use of the generic types `A` and `B` . The trait is then implemented for the `Container` type, specifying `i32` for `A` and `B` so that it can be used with `fn difference()` .）

因为 `Contains` 是泛型，所以我们被迫显式地指出了针对 `fn difference()` 的所有泛型类型。实际上，我们只想要一种方式来表示由输入的 `C` 确定的 `A` 和 `B` 。正如你就要看到的下一节内容，关联类型正好提供了这方面能力。

```
struct Container(i32, i32);

// 这个 trait 检查 2 个项是否存到 Container（容器）中。
// 还会获得第一个值或最后一个值。
trait Contains<A, B> {
    fn contains(&self, &A, &B) -> bool; // 显式指出需要 `A` 和 `B`
    fn first(&self) -> i32; // 未显式指出需要 `A` 或 `B`
    fn last(&self) -> i32;  // 未显式指出需要 `A` 或 `B`
}

impl Contains<i32, i32> for Container {
    // 如果存储的数字相等则为真。
    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 }
}

// `C` 包含 `A` 和 `B` 。鉴于此，必须重复表达 `A` 和 `B` 真麻烦。
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

使用“关联类型”可以增强代码的可读性，其方式是移动内部类型到一个 trait 作为*output*（输出）类型。这个 `trait` 的定义的语法如下：

```
// `A` 和 `B` 在 trait 里面通过`type` 关键字来定义。
// （注意：此处的 `type` 不同于用作别名时的 `type`）。
trait Contains {
    type A;
    type B;

    // 通常提供新语法来表示这些新的类型。
    // （原文：Updated syntax to refer to these new types generically.）
    fn contains(&self, &Self::A, &Self::B) -> bool;
}
```

注意到上面函数用到了 `Contains` `trait`，再也不需要表达 `A` 或 `B`：

```
// 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：

```
struct Container(i32, i32);

// 这个 trait 检查 2 个项是否存到 Container（容器）中。
// 还会获得第一个值或最后一个值。
trait Contains {
    // 在这里定义可以被方法利用的泛型类型。
    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` 在这里也是有效的。
    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));
}
```

虚位类型参量

虚位类型参量（phantom type parameter）是一种在运行时（runtime）不出现，而在（且只在）编译期进行静态方式检查的参量。

数据类型可以使用额外的泛型类型参量来充当标记或在编译期执行类型检查。这些额外的参量没有存储值，且没有运行时行为（runtime behavior）。

在下面例子中，我们把 std::marker::PhantomData 和虚位类型参量概念结合起来创建包含不同数据类型的元组。

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

// 虚位元组结构体，这是一个带有 `A` 和隐藏参量（hidden parameter） `B` 的泛型。
#[derive(PartialEq)] // 允许这种类型进行相等测试（equality test）。
struct PhantomTuple<A, B>(A,PhantomData<B>);

// 模型元组结构体，这是一个带有 `A` 和隐藏参量 `B` 的泛型。
#[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` 是隐藏参量。
    // 被指定为 `<char, f32>` 的虚位元组（PhantomTuple）类型。
    let _tuple1: PhantomTuple<char, f32> = PhantomTuple('Q', PhantomData);
    // 被指定为 `<char, f64>` 的虚位元组。
    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（compile-time）报错！类型不匹配，所以这些值不能够比较：
    //println!("_tuple1 == _tuple2 yields: {}",
    //          _tuple1 == _tuple2);
```

```
    // 编译期报错！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：单位阐明

单位转换（unit conversion）中的一个有效方法可以通过实现 `Add` trait 来检验，其中 `Add` 带有虚位类型参量（原文：A useful method of unit conversions can be examined by implementing `Add` with a phantom type parameter）。用作检验 `Add` `trait` 的代码如下：

```
// 这个结构得到加强：`Self + RHS = Output`，其中 RHS 要
// 是没有给出特定实现的话会默认成为 Self。
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}

// `Output` 必须是 `T<U>` 类型，所以 `T<U> + T<U> = T<U>`。
impl<U> Add for T<U> {
    type Output = T<U>;
    ...
}
```

完整实现：

```
use std::ops::Add;
use std::marker::PhantomData;

/// 创建空枚举来定义单位类型。
#[derive(Debug, Clone, Copy)]
enum Inch {}
#[derive(Debug, Clone, Copy)]
enum Mm {}

/// `Length` 是一个带有虚位类型参量的 `Unit`（单位），
/// 而且不是关于长类型（即 `f64`）的泛型。
///
/// `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() 返回一个全新的包含总和的 `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() {
    // 指出 `one_foot` 拥有虚位类型参量 `Inch`。
    let one_foot:  Length<Inch> = Length(12.0, PhantomData);
    // `one_meter` 拥有虚位类型参量 `Mm`。
    let one_meter: Length<Mm>   = Length(1000.0, PhantomData);

    // `+` 调用了 `add()` 方法，该方法对 `Length<Unit>` 进行了实现。
    //
    // 由于 `Length` 了实现了 `Copy`，于是 `add()` 不会消费 `one_foot`
    // 和 `one_meter`，但会复制它们到 `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);

    // 无意义的操作将会失败，因为它们会导致：
    // 编译期报错：类型不匹配（Compile-time Error: type mismatch.）。
    //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（borrowing）和生命周期（lifetime）中起着重要作用。也就是说，当借用有效，当资源可以释放，还有当变量被创建或销毁时，作用域都在指导编译器（原文：That is, they indicate to the compiler when borrows are valid, when resources can be freed, and when variables are created or destroyed.）。

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>` 占有堆中的内存。Rust 强制实行 RAII（Resource Acquisition Is Initiallization，资源获取即初始化），所以任何一个对象在离开作用域时，它的析构器（destructor）都被调用以及它的资源都被释放。

这种行为避免了资源泄露（*resource leak*）的错误，所以你再也不用手动释放内存或者担心内存泄露（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` 在这里销毁，而且内存得到释放
}

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` 在这里销毁，而且内存得到释放
    }

    // 创建很多 box，纯属娱乐。
    // 完全不需要手动释放内存！
    for _ in 0u32..1_000 {
        create_box();
    }

    // `_box2` 在这里销毁，而且内存得到释放
}
```

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)
```

完全没有泄露！

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 references）。

在进行赋值（ `let x = y` ）或通过值来传递函数参数的时候，资源的所有权（*ownership*)会发生转移（transfer）。按照 Rust 的说法，这种方式被称为移动（*move*）。

在移动资源之后，原来的所有者不能再使用，这可避免悬垂指针的产生。

```
// 此函数取倒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**（*Copy*）到 `y`——不存在资源移动
    let y = x;

    // 两个值都可以独立地使用
    println!("x is {}, and y is {}", x, y);

    // `a` 是一个指向堆分配的整型的指针
    let a = Box::new(5i32);

    println!("a contains: {}", a);

    // **移动**（*Move*) `a` 到 `b`
    let b = a;
    // 把 `a` 的指针地址（非数据）复制到 `b`。现在两者都是指向
    // 同一个堆分配的数据，但是现在是 `b` 占有它。

    // 报错！`a` 再也不能访问数据，因为它不再拥有堆上的内存。
    //println!("a contains: {}", a);
    // 试一试 ^ 将此行注释去掉

    // 此函数从 `b` 中取得栈分配的内存的所有权
    destroy_box(b);

    // 此时堆上的内存已经释放掉，而这个操作会导致解引用已释放的内存，
    // 但这种情况会被编译器会禁止。
```

```
    // 报错！和前面出错的原因一样。
    //println!("b contains: {}", b);
    // 试一试 ^ 将此行注释去掉
}
```

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);
}
```

借用

多数情况下，我们更希望访问数据本身而不需要取得它的所有权。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。

编译器静态地保证了（通过借用检查器）引用总是（*always*）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。

```
// 此函数拥有 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 的内容可以
    // 再次借用。
    borrow_i32(&boxed_i32);
    borrow_i32(&stacked_i32);

    {
        // 给出一个指向 box 里面所包含数据的引用
        let _ref_to_i32: &i32 = &boxed_i32;

        // 报错!
        // 当 `boxed_i32` 里面的值被借用时，不能销毁 `boxed_int`。
        eat_box_i32(boxed_i32);
        // 改正 ^ 注释掉此行

        // `_ref_to_i32` 离开作用域且不再被借用。
    }

    // box 现在可以放弃 `eat_i32` 的所有权且可以销毁
    eat_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*），并赋予了借用者读/写访问能力。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` 是一个指向分配在只读内存区的字符串的引用
    author: &'static str,
    title: &'static str,
    year: u32,
}

// 此函数接受一个指向图书 Book 的引用
fn borrow_book(book: &Book) {
    println!("I immutably borrowed {} - {} edition", book.title, book.year);
}

// 此函数接受一个指向可变的图书 Book 的引用，同时把年份 `year` 改为 2004 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` 的不可变的图书 Book
    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);

    // 借用一个可变对象作为可变类型
    new_edition(&mut mutabook);
```

```
    // 报错！不能借用一个不可变对象来充当可变类型
    new_edition(&mut immutabook);
    // 改正 ^ 注释掉此行
}
```

参见：

`static`

冻结

当数据被不可变地借用时，它还会冻结（*freeze*）。已冻结（*frozen*）数据无法通过原始对象来修改，直到指向这些数据的所有引用离开作用域为止。

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

    {
        // 借用 `_mutable_integer`
        let _large_integer = &_mutable_integer;

        // 报错！`_mutable_integer` 在本作用域被冻结
        _mutable_integer = 50;
        // 改正 ^ 注释掉此行

        // `_large_integer` 离开作用域
    }

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

别名使用

数据可以进行多次不可变借用，但是在不可变借用的期间，原始数据不可进行可变借用。也就是说，在同一段时间内只允许单独一个可变借用。原始数据在可变引用离开作用域之后可再次被借用。

```
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;

        // 通过引用和原始所有者来访问数据
        println!("Point has coordinates: ({}, {}, {})",
                 borrowed_point.x, another_borrow.y, point.z);

        // 报错！不能借用 `point` 作为可变内容，因为目前已被借用成为
        // 不可变内容。
        //let mutable_borrow = &mut point;
        // 动手试一试 ^ 将此行注释去掉。

        // 不可变引用离开作用域
    }

    {
        let mutable_borrow = &mut point;

        // 通过可变引用来改变数据
        mutable_borrow.x = 5;
        mutable_borrow.y = 2;
        mutable_borrow.z = 1;

        // 报错！不能借用 `point` 作为不可变内容，因为目前它已被借用成为
        // 可变内容。
        //let y = &point.y;
        // 动手试一试 ^ 将此行注释去掉。

        // 报错！不能打印，因为 `println!` 接受了一个不可变引用。
        //println!("Point Z coordinate is {}", point.z);
        // 动手试一试 ^ 将此行注释去掉。

        // 好！可变引用可以作为不可变的传给 `println!`。
        println!("Point has coordinates: ({}, {}, {})",
```

```
                  mutable_borrow.x, mutable_borrow.y, mutable_borrow.z);

        // 可变引用离开作用域
    }

    // `point` 的不可变引用再次可用。
    let borrowed_point = &point;
    println!("Point now has coordinates: ({}, {}, {})",
             borrowed_point.x, borrowed_point.y, borrowed_point.z);
}
```

`ref` 模式

在通过 `let` 绑定来进行模式匹配或解构时， `ref` 关键字可用来接受结构体/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：

```
#[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` 可以结合 `mut` 来接受可变引用。
        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*）是一个结构成分，编译器（也称为借用检查器）使用它来确保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所在地方决定。因此，只要在出借者（lender）被销毁前结束，借用都是有效的。而借用的作用域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。

```
// 下面使用连线来标注各个变量的生命周期的创建和销毁。
// `i` 的生命周期最长，因为它的作用域完全覆盖了 `borrow1` 和
// `borrow2` 两者。`borrow1` 和 `borrow2` 的周期没有关联，
// 因为它们各不相交。
fn main() {
    let i = 3; // `i` 的生命周期开始。──────────────────────┐
    //                                                    │
    { //                                                  │
        let borrow1 = &i; // `borrow1` 的生命周期开始。 ──┐│
        //                                                ││
        println!("borrow1: {}", borrow1); //              ││
    } // `borrow1` 结束。──────────────────────────────────┘│
    //                                                    │
    //                                                    │
    { //                                                  │
        let borrow2 = &i; // `borrow2` 生命周期开始。──────┐│
        //                                                ││
        println!("borrow2: {}", borrow2); //              ││
    } // `borrow2` 结束。──────────────────────────────────┘│
    //                                                    │
}   // 生命周期结束。 ──────────────────────────────────────┘
```

注意到这里没有用到名称或类型来标记生命周期，这限制了生命周期的表现能力，在后面我们将会看到生命周期更强大的功能。

显示标注

借用检查器使用显式的生命周期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，Rust 需要显式标注来确定引用的生命周期应该是什么样的。对于显式地标注引用的生命周期的语法如下：

```
foo<'a>
// `foo` 带有一个生命周期参量 `'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。类型的显式标注有 `&'a T` 这样的形式，其中 `'a` 已引入。

In cases with multiple lifetimes, the syntax is similar: 对于多个生命周期的情况，语法是类似的：

```
foo<'a, 'b>
// `foo` 带有生命周期参量 `'a` 和 `'b`
```

在这种情形中， `foo` 的生命周期不能超出 `'a` 或 `'b` 的周期。

看下面的例子，了解显式生命周期标注的运用：

```
// 生命周期 `'a` 和 `'b`。这两个生命周期都必须至少要和 `print_refs`
// 函数的一样长。
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}

// 不带参量的函数，不过有一个生命周期参量 `'a`。
fn failed_borrow<'a>() {
    let _x = 12;

    // 报错：`_x` 存活时间长度不够（`_x` does not live long enough）
    //let y: &'a i32 = &_x;
    // 尝试使用生命周期 `'a` 作为函数内部的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，给下面代码借用。
    let (four, nine) = (4, 9);

    // 两个变量的借用（`&`）都传进函数。
    print_refs(&four, &nine);
    // 任何借用得来的输入量都必须比借入者“活”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` 未包含引用来迫使 `'a` 长于函数的生命周期，
    // 但 `'a` 寿命更长。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

忽视省略（elision）情况，带上生命周期的函数签名（function signature）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一个和输入量相同的生命周期或是静态类型（ `static` ）。

另外要注意，若会导致返回的引用指向无效数据，则返回不带输入量的引用是被禁止的。下面例子展示了一些带有生命周期的函数的有效形式：

```
// 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 i32 { &7 }
// 上面代码是无效的：`'a` 存活的时间必须比函数的长。
// 这里的 `&7` 将会创建一个 `i32` 类型，跟在引用后面。
// 然后数据在离开作用域时删掉，留下一个指向无效数据的引用，
// 此引用将被返回。

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();
}
```

参见：

methods

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。
// 指向 `i32` 的引用必须比 `Borrowed` 寿命更长。
// （原望：A type `Borrowed` which houses a reference to an
// `i32`. The reference to `i32` must outlive `Borrowed`.）
#[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，不是 `i32` 类型就是一个指向某个量的引用。
//（原文： An enum which is either an `i32` or a reference to one.）
#[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：

structs

限定

就如泛型类型能够被限定一样，生命周期（它们本身就是泛型）也可以使用限定。`:` 字符的意义在这里稍微有些不同，不过 `+` 是相同的。注意下面是怎么说明的：

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; // 用于限定的 trait。

#[derive(Debug)]
struct Ref<'a, T: 'a>(&'a T);
// `Ref` 包含一个指向指向泛型类型 `T` 的引用，其中 `T` 拥有
// 一个未知的生命周期 `'a`。`T` 是被限定的，从而在 `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函数存活时间更长。
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, 泛型中的限定, 以及 泛型中的多重限定

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。这种形式出现在编译器推导强制转换的时候，也出现在声明生命周期不同的时候（原文：This comes in the form of inferred coercion by the Rust compiler, and also in the form of declaring a lifetime difference）：

```
// 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>` 理解为生命周期 `'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));
    };
}
```

静态

`'static` 生命周期在可能存在的生命周期中是最长的，并在运行程序的周期中持续存在。 `static` 生命周期也可能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，这两种方式都是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，其中`NUM` 的 `'static`
// 生命周期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
        // 数据会保留在二进制文件里面。
    }

    {
        // 产生一个整型给 `coerce_static` 使用：
        let lifetime_num = 9;

        // 将 `NUM` 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过普遍了，所以借用检查器将会隐式地添加它们来以减少字母输入和增强可读性。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` 本质上拥有相同的识别标志，是因为
// `elided_input` 的生命周期被编译器省略掉了：
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` 也拥有相同的识别标志，
// 是因为生命周期被隐式地添加进 `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

特性 **trait**

`trait` 是对未知类型定义的方法集： `Self` 。它们可以访问同一个 trait 中定义的方法。

对任何数据类型实现 trait 都是可行的。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` ，允许使用来自带有 `Sheep` 的 `Animal` 的方法（原文：allowing the use of methods from `Animal` with a `Sheep` ）。

```
struct Sheep { naked: bool, name: &'static str }

trait Animal {
    // 静态方法标记：`Self` 表示实现者类型（implementor type）。
    fn new(name: &'static str) -> Self;

    // 实例方法（instance method）标记；这些方法将返回一个字符串。
    fn name(&self) -> &'static str;
    fn noise(&self) -> &'static str;

    // trait 可以提供默认方法定义（method definition）。
    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（implementor）可以使用实现者的 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` 是该实现者类型：`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) {
        // 例如完们可以增加一些安静的沉思（quiet contemplation）。
        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，编译器能够提供一些对于 trait 的基本实现。如果需要一个更复杂的业务，这些 trait 仍然可以手动实现。（原文：The compiler is capable of providing basic implementations for some traits via the `#[derive]` attribute. These traits can still be manually implemented if a more complex behavior is required.）

下面列举了 “derivable”（可派生的）trait：

- 比较 trait: `Eq`，`PartialEq`，`Ord`，`PartialOrd`
- `Clone`，采用复制（copy）方式从 `&T` 创建 `T`。
- `Copy`，给出“复制语义”（'copy semantics'）来替代“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。
- `Zero`，创建数字数据类型的一个零值实例（zero instance）。
- `Debug`，使用 `{:?}` 格式化程序（formatter）格式化一个值。

```
// `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 ^ 将此行注释去掉

    // 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 ^ 将此行注释去掉

    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);
}
```

`derive`

运算符重载

在 Rust 中，大部分运算符都可以通过 trait 来重载。也就是说，这些运算符可以根据它们输入的参数来完成不同的任务。为什么这样做是可行的呢，是因为运算符是对方法调用的语法糖。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 `Add` trait 的实现者（implementor）使用。

点击这里查看列举的重载运算符 trait，比如 `Add`。（原文：A list of the traits, such as `Add`, that overload operators are available here.）

```
use std::ops;

struct Foo;
struct Bar;

#[derive(Debug)]
struct FooBar;

#[derive(Debug)]
struct BarFoo;

// `std::ops::Add` trait 在这里用来指明 `+` 的功能，我们给出 `Add<Bar>`——关于
// 加法的 trait，带有一个 `Bar` 类型的右操作数（RHS）。下面代码块实现了这样的
// 运算： Foo + Bar = FooBar。
impl ops::Add<Bar> for Foo {
    type Output = FooBar;

    fn add(self, _rhs: Bar) -> FooBar {
        println!("> Foo.add(Bar) was called");

        FooBar
    }
}

// 通过反转类型，我们以实现非交换的加法作为结束。
// 这里我们给出 `Add<Foo>`——关于加法的 trait，带有一个 `Foo` 类型的右操作数。
// 这个代码块实现了这样的操作：Bar + Foo = BarFoo。
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`，当一个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 来释放资源的类型的例子。`Drop` trait 也可以针对任意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。（原文：The following example adds a print to console to the `drop` function to announce when it is called.）

```
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。
}
```

Iterators

`Iterator` trait 用来实现关于集合（collection）类型（比如数组）的迭代器。

这个 trait 只需定义一个指向 `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` 结构通常使用 `.into_iterator()` 方法将一些集合类型转换为迭代器。

下面例子展示了如何访问使用 `Iterator` trait 的方法，关于这方面的更多内容可点击这里查看。

```
struct Fibonacci {
    curr: u32,
    next: u32,
}

// 实现关于 `Fibonacci` （斐波那契）的 `Iterator`。
// `Iterator` trait 只需定义一个指向 `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（wrapped）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（generator）
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` 通过 `Iterator` 进行工作，直到 `Iterator` 为 `None`。
    // 每个 `Some` 值都被解包（unwrap）且限定为一个变量（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)` 方法通过跳过前 `n` 项缩短了 `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);
    }
}
```

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得到一份资源的复制。

`Clone` trait 正好帮助我们完成这任务。更普遍地，我们可以使用由 `Clone` trait 定义的方法。

```
// 不含资源的单元结构体
#[derive(Debug, Clone, Copy)]
struct Nil;

// 包含实现 `Clone` trait 的资源的元组结构体
#[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` 复制到 `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 ^ 将此行注释去掉。

    // 将 `moved_pair` 克隆到 `cloned_pair`（包含资源）
    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);
}
```

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）。正如你已经看过了前面章节，宏看起来和函数很像，除了名称末尾连着一个感叹号 `!`，但宏并不产生一个函数调用，而是展开成源码并结合程序的其余代码一起进行编译。

宏是通过 `macro_rules!` 宏来创建的。

```
// 这是一个简单简单的宏，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!()
}
```

指示符

宏里面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，将它转换成一个字符串，
    // 并伴随着表达式的结果。
    // `expr` 指示符用于表达式。
    ($expression:expr) => (
        // `stringify!` 把表达式转换成一个字符串，正如 stringify
        // （意思为“字符串化”） 所表达的意思那样。
        println!("{:?} = {:?}",
                 stringify!($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`
- `pat` (模式 *pattern*)
- `path`
- `stmt` (语句 *statement*)
- `tt` (令牌树 *token tree*)
- `ty` (类型 *type*)

重载

宏可以重载，从而接受参数的不同组合。`macro_rules!` 在这方面可以类似于匹配（match）代码块那样工作：

```
// `test!` 将以不同的方式来比较 `$left` 和 `$right`，
// 根据所调用的情况确定。
macro_rules! test {
    // 参数不需要使用逗号隔开。
    // 可以使用任意模板（原文：Any template can be used!）！
    ($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，使用 `$(...),+` 包含的内容将匹配一个或多个表达式，使用逗号隔开。还注意到分号对于最后一种情形是可选的。

```
// `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),+) => (
        // 对尾部的 `$y` 调用 `find_min!`
        std::cmp::min($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单元的公共部分，宏允许编写 DRY 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，实现并测试了关于 `Vec<T>` 的 `+=`、`*=` 和 `-=` 等运算符。

```
use std::ops::{Add, Mul, Sub};

macro_rules! assert_equal_len {
    // `tt` （token tree，令牌树）指示符用于运算符和令牌。
    // （原文：The `tt` (token tree) designator is used for
    // operators and tokens.）
    ($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
```

错误处理

错误处理（error handling）是处理可能发生失败情况的过程。例如读取一个文件失败，然后继续使用这个失效的输入显然是有问题的。错误处理允许我们以一种显式的方式来发现并处理这类错误，避免了其余代码发生潜在的问题。

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，开始展开任务（译注：感觉此句翻译不好，望指正，原文为 starts unwinding the task），且通常退出程序。这里我们显式地在错误条件上调用 `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` & `unwrap`

在上个例子中，我们显示出我们能够任意引入程序失败（program failure）。当公主收到蛇这件不合适的礼物时，我们就告诉程序产生 `panic` 。但是，如果公主期待一件礼物却没收到呢？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（ `""` ），就像处理蛇那样的方式。既然我们使用了 Rust，那我们就让编译器指出没有礼物的情况。

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，用于变量可能不存在的情景（原文：An `enum` called `Option<T>` in the `std` library is used when absence is a possibility. ）。它表现为以下两个 “options”（选项）中的其中一个：

- `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。隐式处理会返回内部元素或 `panic` 。

请注意，手动使用 expect 方法自定义 `panic` 是可能的，而 `unwrap` 相比显式处理则留下不太有意义的输出。在下面例子中，显式处理得到更具可控性的结果，同时若需要的话，可将选项保留为 `panic` 。（本段原文：Note that it's possible to manually customize `panic` with expect, but `unwrap` otherwise leaves us with a less meaningful output than explicit handling. In the following example, explicit handling yields a more controlled result while retaining the option to `panic` if desired. ）

```
// 平民（commoner）已经见过所有东西，并能妥善处理好各种情况。
// 所有礼物都通过手动使用 `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."),
    }
}

// 我们受保护的公主见到蛇将会 `panic`（恐慌）。
fn give_princess(gift: Option<&str>) {
    // 使用 `unwrap`，当接收到 `None` 时返回一个 `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);
}
```

组合算子：`map`

`match` 是处理 `Option` 的一个有效方法。但是你最终会发现很多用例都相当繁琐，特别是操作只有一个有效输入的情况。在这些情况下，可以使用 组合算子（combinator）以模块化方式来管理控制流。

`Option` 有一个内置方法 `map()`，这个组合算子可用于简单映射 `Some -> Some` 和 `None -> None` 的情况。多个不同的 `map()` 调用可以更灵活地链式连接在一起。

在下面例子中，`process()` 轻松取代了前面的所有函数，且更加紧凑。

```
#![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);

// 削水果皮。如果没有水果，就返回 `None`。
// 否则返回削好皮的水果。
fn peel(food: Option<Food>) -> Option<Peeled> {
    match food {
        Some(food) => Some(Peeled(food)),
        None       => None,
    }
}

// 和上面一样，我们要在切水果之前确认水果是否已经削皮。
fn chop(peeled: Option<Peeled>) -> Option<Chopped> {
    match peeled {
        Some(Peeled(food)) => Some(Chopped(food)),
        None               => None,
    }
}

// 和前面的检查类似，但是使用 `map()` 来替代 `match`。
fn cook(chopped: Option<Chopped>) -> Option<Cooked> {
    chopped.map(|Chopped(food)| Cooked(food))
}

// 另外一种实现，我们可以链式调用 `map()` 来简化上述的流程。
fn process(food: Option<Food>) -> Option<Cooked> {
    food.map(|f| Peeled(f))
        .map(|Peeled(f)| Chopped(f))
        .map(|Chopped(f)| Cooked(f))
```

```
}

// 在尝试吃水果之前确认水果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)));
    // 现在让我们试试更简便的方式 `process()`。
    // （原文：Let's try the simpler looking `process()` now.）
    // （翻译疑问：looking 是什么意思呢？望指教。）
    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，在返回类型是 `Option<T>` 的函数中使用 `map()` 会导致出现嵌套形式 `Option<Option<T>>` 。多层链式调用也会变得混乱。所以有必要引入 `and_them()` ，就像某些熟知语言中的 flatmap。

`and_then()` 使用包裹的值（wrapped value）调用其函数输入并返回结果。 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>` 。使用 `map()` 替代 `and_then()` 将会得到 `Option<Option<Food>>` ，对 `eat()` 来说是一个无效类型。

```
#![allow(dead_code)]

#[derive(Debug)] enum Food { CordonBleu, Steak, Sushi }
#[derive(Debug)] enum Day { Monday, Tuesday, Wednesday }

// 我们没有原材料（ingredient）来制作寿司。
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，除了欠缺高超的烹饪手艺。
fn have_recipe(food: Food) -> Option<Food> {
    match food {
        Food::CordonBleu => None,
        _                => Some(food),
    }
}

// 做一份好菜，我们需要原材料和食谱这两者。
// 我们可以借助一系列 `match` 来表达相应的逻辑：
// （原文：We can represent the logic with a chain of `match`es:）
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),
        },
    }
}
```

```
// 这可以使用 `and_then()` 方便重写出更紧凑的代码：
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>` ：发现错误，使用元素 `E` 表示（An error was found with element `E` ）

按照约定，预期结果是 “Ok”，而意外结果是 “Err”。

和 `Option` 类似， `Result` 也有很多相关联的方法。例如 `unwrap（）` ，能够产生元素 `T` 或 `panic` 。 对于事件的处理， `Result` 和 `Option` 两者间有很多组合算子重叠。

使用 Rust 过程中，你可能会遇到返回 `Result` 类型的方法，例如 `parse()` 方法。 它在某些情况下可能不能将一个字符串解析为另一种类型，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 double_number(number_str: &str) -> i32 {
    // 让我们尝试使用 `unwrap()` 把数字取出来。它会咬我们吗？
    2 * number_str.parse::<i32>().unwrap()
}

fn main() {
    let twenty = double_number("10");
    println!("double is {}", twenty);

    let tt = double_number("t");
    println!("double is {}", tt);
}
```

在失败的情况下， `parse()` 留给我们一个错误，让 `unwrap()` 产生 `panic` （原文： `parse()` leaves us with an error for `unwrap()` to `panic` on）。另外， `panic` 会退出我们的程序，并提供一个不愉快的错误消息。

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。

关于 `Result` 的 `map`

前面关于 panic 例子，提供给我们的是一个无用的错误消息。为了避免这样，我们需要更具体地指定返回类型。在那个例子中，该常规元素为 `i32` 类型。

为了确定 `Err` 的类型，我们可以借助 `parse()`，它使用 `FromStr` trait 来针对 `i32` 实现。结果是，`Err` 类型被指定为 `ParseIntError`。

在下面例子中要注意，使用简单的 `match` 语句会导致更加繁琐的代码。事实证明，用到 `Option` 的 `map` 方法也对 `Result` 进行了实现。

幸运的是，`Option` 的 `map` 方法是对 `Result` 进行了实现的许多组合算子之一。`enum.Result` 包含一个完整的列表。

```
use std::num::ParseIntError;

// 返回类型重写之后，我们使用模式匹配，而不使用 `unwrap()`。
fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
    match number_str.parse::<i32>() {
        Ok(n)  => Ok(2 * n),
        Err(e) => Err(e),
    }
}

// 就像 `Option`，我们可以使用组合算子，如 `map()`。
// 此函数在其他方面和上述的示例一样，并表示：
// 若值有效则修改 n，否则传递错误。
fn double_number_map(number_str: &str) -> Result<i32, ParseIntError> {
    number_str.parse::<i32>().map(|n| 2 * n)
}

fn print(result: Result<i32, ParseIntError>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    // 这里仍然给出一个合理的答案。
    let twenty = double_number("10");
    print(twenty);

    // 下面提供了更加有用的错误消息。
    let tt = double_number_map("t");
    print(tt);
```

```
}
```

给 `Result` 起别名

当我们要重复多次使用特定的 `Result` 类型怎么办呢？回忆一下，Rust 允许我们创建别名。对问题中提到的特定 `Result`，我们可以很方便地给它定义一个别名。

在单个模块的级别上创建别名特别有帮助。在特定模块中发现的错误常常会有相同的 `Err` 类型，所以一个单一的别名就能简便地定义所有的关联 `Result`。这点太重要了，甚至标准库也提供了一个：`io::Result`！

下面给出一个快速示例来展示语法：

```
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我们特指的 `Result` 类型。
fn double_number(number_str: &str) -> AliasedResult<i32> {
    number_str.parse::<i32>().map(|n| 2 * n)
}

// 这里的别名又让我们节省了一些空间（save some space）。
fn print(result: AliasedResult<i32>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(double_number("10"));
    print(double_number("t"));
}
```

参见：

`Result` 和 `io::Result`

各种错误类型

前面出现的例子确实很方便；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 empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    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
}
```

使用组合算子的知识，我们能够重写上述代码来显式地处理错误。为了做到两种错误类型都能够出现，我们需要将他们转换为一种通用类型，比如 `String` 类型。

就这样，我们将 `Option` 和 `Result` 都转换成 `Result`，从而将他们的错误类型映射成相同的类型：

```
// 使用 `String` 作为错误类型
type Result<T> = std::result::Result<T, String>;

fn double_first(vec: Vec<&str>) -> Result<i32> {
    vec.first()
       // 若值存在则将 `Option` 转换成 `Result`。
       // 否则提供一个包含该字符串（`String`） 的 `Err`。
       .ok_or("Please use a vector with at least one element.".to_owned())
       // 回想一下，`parse` 返回一个 `Result<T, ParseIntError>`。
       .and_then(|s| s.parse::<i32>()
                      // 映射任意错误 `parse` 产生得到 `String`。
```

```
                        // （原文：Map any errors `parse` yields to `String`.）
                        .map_err(|e| e.to_string())
                        // `Result<T, String>` 成为新的返回类型，
                        // 我们可以给里面的数字扩大两倍。
                        .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 empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(empty));
    print(double_first(strings));
}
```

在下一节，我们将学到一个替代方法来显式处理这型错误。

参见：

`Option::ok_or`，`Result::map_err`

提前返回

在前面的例子中，我们使用组合算子显式地处理错误。 另一种处理这种情形分解的方法是使用 `match` 语句和提前返回（*early returns*）的组合形式。

也就是说，我们可以简单地停止执行函数并返回错误（若发生的话）。 而且这种形式的代码更容易阅读和编写。考虑如下版本，这是将之前的例子使用提前返回方式重写的：

```
// 使用 `String` 作为错误类型
type Result<T> = std::result::Result<T, String>;

fn double_first(vec: Vec<&str>) -> Result<i32> {
    // 若存在值时，则将 `Option` 转换成 `Result`。
    // 否则提供一个包含此 `String` 的 `Err`。
    let first = match vec.first() {
        Some(first) => first,
        None => return Err("Please use a vector with at least one element.".to_owned())
    };

    // 若 `parse` 操作正常的话，则将内部的数字扩大 2 倍。
    // 否则映射任意错误，来自 `parse` 产生的 `String`。
    // （原文：Double the number inside if `parse` works fine.
    // Otherwise, map any errors that `parse` yields to `String`.）
    match first.parse::<i32>() {
        Ok(i) => Ok(2 * i),
        Err(e) => Err(e.to_string()),
    }
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(empty));
    print(double_first(strings));
}
```

现在我们已经学会了使用组合算子和提前返回来显式地处理错误。虽然我们通常希望避免 `panic`，但显式处理我们所有的错误将会麻烦。

在下一节，我们将介绍 `try!`，用在这样的场景，我们只需要简单地 `unwrap` 而避免可能的 `panic`。

介绍 `try!`

有时我们只是想要 `unwrap` 的简单，而又不会产生 `panic` 。截至目前， `unwrap` 迫使我们嵌套了一层又一层，而我们想要的只不过是将相应的变量取出来。正因为这样，我们引入了 `try!` 。

在发现错误（ `Err` ）时，有两个有效的操作：

1. `panic!` ，但我们已经尽可能回避这种情况
2. `return` ，因为 `Err` 意味着它不能被处理

`try!` 几乎完全[1]等同于一个这样的 `unwrap` ——对待错误（ `Err` ）采用返回的方式而不是`panic。我们来看看如何简化之前使用组合算子的示例：

```
// 使用 `String` 作为错误类型
type Result<T> = std::result::Result<T, String>;

fn double_first(vec: Vec<&str>) -> Result<i32> {
    let first = try!(vec.first()
        .ok_or("Please use a vector with at least one element.".to_owned()));

    let value = try!(first.parse::<i32>()
        .map_err(|e| e.to_string()));

    Ok(2 * value)
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(empty));
    print(double_first(strings));
}
```

注意到目前为止，我们一直使用 `String` 作为错误类型。但它们作为错误类型是有一定限制的。在下一节中，我们将学习如何通过自定义类型来创建更具结构化和更多信息量的错误。

> [1]. 参考 re-enter try!，查看更多详细内容。 ↩

参见：

`Result` 和 `io::Result`

定义一个错误类型

前面我们一直使用字符串（ `String` ）作为错误消息。实际上，字符串作为错误类型是存在一些局限的。下面是友好的错误类型标准。字符串（ `String` ）很好地实现了前两点，但无法做到后两点： Rust 允许自定义错误类型。一般而言，一个“良好”的错误类型：

- 使用相同类型来表达不同的错误
- 给用户提供友好的错误信息
- 方便和其他类型比较
  - Good: `Err(EmptyVec)`
  - Bad: `Err("Please use a vector with at least one element".to_owned())`
- 能够保存错误的信息（原文：Can hold information about the error.）：
  - Good: `Err(BadChar(c, position))`
  - Bad: `Err("+ cannot be used here".to_owned())`

可以看到字符串（ `String` ）（前面我一们一值在用）可以地满足前两点标准，但后两条无法满足。这使得 `String` 错误既难以创建，也难以达到要求。仅仅为了优雅地显示，实在不应该使用 `String` 格式化方式污染大量的逻辑代码（原文：It should not be necessary to pollute logic heavy code with `String` formatting simply to display nicely.）。

```
use std::num::ParseIntError;
use std::fmt;

type Result<T> = std::result::Result<T, DoubleError>;

#[derive(Debug)]
// 定义我们的错误类型。不管对我们的错误处理情况有多重要，这些都可能自定义。
// 现在我们能够按照底层工具的错误实现，写下我们的错误，或者两者之间的内容。
// （原文：Define our error types. These may be customized however is useful for our er
ror
// handling cases. Now we will be able to defer to the underlying tools error
// implementation, write our own errors, or something in between.）
enum DoubleError {
    // 我们不需要任何额外的信息来描述这个错误。
    EmptyVec,
    // 我们将推迟对于这些错误的解析错误的实现。（原文：We will defer to the parse
    // error implementation for their error.）提供额外信息将要增加更多针对类型的数据。
    Parse(ParseIntError),
}

// 类型的展示方式的和类型的产生方式是完全独立的。我们无需担心显示样式会搞乱我们
// 工具集所需的复杂逻辑。它们是独立的，就是说它们处理起来是相互独立的。
//
// 我们没有存储关于错误的额外信息。若确实想要，比如，要指出哪个字符串无法解析，
// 那么我们不得不修改我们类型来携带相应的信息。
```

```
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 wrapper，所以按照底层类型来给出我们的 `fmt` 实现。
            // （原上：This is a wrapper so defer to the underlying types' own implemen
tation
            // of `fmt`.）
            DoubleError::Parse(ref e) => e.fmt(f),
        }
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    vec.first()
       // 将错误改成我们新的类型。
       .ok_or(DoubleError::EmptyVec)
       .and_then(|s| s.parse::<i32>()
             // 在这里也更新成新的错误类型。
            .map_err(DoubleError::Parse)
            .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!["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：

`Result` 和 `io::Result`

`try!` 的其他用法

注意在前面的例子中，我们对调用 `parse` 的最直接反应就是将错误从库错误映射到我们的新的自定义错误类型（原文：Notice in the previous example that our immediate reaction to calling `parse` is to `map` the error from a library error into our new custom error type）：

```
.and_then(|s| s.parse::<i32>()
    .map_err(DoubleError::Parse)
```

这是一个很简单且常见的操作，要是它能够省略的话将会相当方便。可惜的是，因为 `and_then` 不够灵活，所以它不能。但是，我们可改用 `try!`。

`try!` 在前面已经解释过，它可以充当 `unwrap` 或 `return Err(err)`，这说法只是很大程度上是对的。实际上它意味着 `unwrap` 或者 `return Err(From::from(err))`。由于 `From::from` 是一个不同类型间相互转换的工具，所以如果你使用 `try!`，当中的错误若能够转换成返回类型，这将会自动转换。

在这里，我们使用 `try!` 重写前面的例子。结果可看到，`From::from` 已对我们的错误类型提供实现时，`map_err` 将会消失：

```
use std::num::ParseIntError;
use std::fmt;

type Result<T> = std::result::Result<T, DoubleError>;

#[derive(Debug)]
enum DoubleError {
    EmptyVec,
    Parse(ParseIntError),
}

// 实现从 `ParseIntError` 到 `DoubleError` 的转换。如果一个 `ParseIntError`
// 需要转换成 `DoubleError`，这将会被 `try!` 自动调用。
impl From<ParseIntError> for DoubleError {
    fn from(err: ParseIntError) -> DoubleError {
        DoubleError::Parse(err)
    }
}

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"),
            DoubleError::Parse(ref e) => e.fmt(f),
```

```
        }
    }
}

// 和前面的结构一样，但没有将全部的 `Results` 和 `Options` 链接在一起，
// 我们使用 `try!` 立即得到内部的值。
// （原文：// The same structure as before but rather than chain all `Results`
// and `Options` along, we `try!` to get the inner value out immediately.）
fn double_first(vec: Vec<&str>) -> Result<i32> {
    // 仍然转为 `Result`，通过规定怎样转为 `None`。
    // （原上：// Still convert to `Result` by stating how to convert `None`.）
    let first = try!(vec.first().ok_or(DoubleError::EmptyVec));
    let parsed = try!(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!["93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

现在变得整洁多了。如果和原始的 `panic` 进行比较，可以看到它好像就是使用 `try!` 替换 `unwrap`，除了返回类型是 `Result` 类型这点不同，所以它们必须在顶层被解构出来。

然而，不要指望这种错误处理经常取代 `unwrap` 的用法。这种错误处理会使代码行数扩大三倍，即便是很小的一段代码也不能称之为简单。

实际中要是将 1000 行库代码从 `unwrap` 移动到更适当的错误处理，可能会导致代码量增加 100 行，但这做法是可取的，当然这重构的过程并非易事。

很多库可能只是抛弃了实现 `Display`，然后在所需的基础上增加 `From`（原文：Many libraries might get away with only implementing `Display` and then adding `From` on an as needed basis.）。然而，越是正式的库最后越是需要面临实现更高级的错误处理的需求。

参见：

`From::from` 和 `try!`

使用 `Box` 处理错误

通过对错误类型实现 `Display` 和 `From`，我们能够利用上绝大部分标准库错误处理工具。然而，我们遗漏了一个功能：轻松 `Box` 我们错误类型的能力。

标准库会自动通过 `Form` 将任意实现了 `Error` trait 的类型转换成 trait 对象 `Box<Error>` 的类型（原文：The `std` library automatically converts any type that implements the `Error` trait into the trait object `Box<Error>`, via `From`.）。对于一个库用户，下面可以很容易做到：

```
fn foo(...) -> Result<T, Box<Error>> { ... }
```

用户可以使用一系列外部库，其中每个都提供各自错误类型。为了定义一个有效的 `Result<T, E>` 类型，用户有几个选择：

- 定义一个新的限定在外部库错误类型的包装（wrapper）错误类型（原文：define a new wrapper error type around the libraries error types）
- 将错误类型转换成 `String` 或者其他合适的选择
- 通过类型擦除（type erasure）将错误类型装包（ `Box` ）成 `Box<Error>`

将内容“装包”（"Boxing"）是一个常见的选择。缺点是潜在的错误类型只能在运行时知道，且不能静态确定（statically determined）。正如刚才提到的，要做到这点所有要做的事情就是实现 `Error` trait：

```
trait Error: Debug + Display {
    fn description(&self) -> &str;
    fn cause(&self) -> Option<&Error>;
}
```

有了这个实现后，我们再来回顾前面学过的最近例子。注意到它所带的错误类型 `Box<Error>` 也变成有效的了，就像前面用到的 `DoubleError` 那样（原文：With this implementation, let's look at our most recent example. Note that it is just as valid with the error type of `Box<Error>` as it was before with `DoubleError`）：

```
use std::error;
use std::fmt;
use std::num::ParseIntError;

// 将别名更改为 `Box<error::Error>`。
type Result<T> = std::result::Result<T, Box<error::Error>>;

#[derive(Debug)]
enum DoubleError {
    EmptyVec,
    Parse(ParseIntError),
```

```
}

impl From<ParseIntError> for DoubleError {
    fn from(err: ParseIntError) -> DoubleError {
        DoubleError::Parse(err)
    }
}

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"),
            DoubleError::Parse(ref e) => e.fmt(f),
        }
    }
}

impl error::Error for DoubleError {
    fn description(&self) -> &str {
        match *self {
            // 错误的简短说明。不需要和 `Display` 一样。
            DoubleError::EmptyVec => "empty vectors not allowed",
            // 这已经实现了 `Error`，所以遵循它自己的实现。
            DoubleError::Parse(ref e) => e.description(),
        }
    }

    fn cause(&self) -> Option<&error::Error> {
        match *self {
            // 没有潜在的差错，所以返回 `None`。
            DoubleError::EmptyVec => None,
            // 差错为底层实现的错误类型。被隐式地转换成 trait 对象 `&error::Error`。
            // 这会正常工作，因为底层的类型已经实现了 `Error` trait。
            DoubleError::Parse(ref e) => Some(e),
        }
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    let first = try!(vec.first().ok_or(DoubleError::EmptyVec));
    let parsed = try!(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!["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：

Dynamic dispatch 和 `Error` trait

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`（可增长的字符串），如: `"hello world"`
- 可增长的 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

Box, 以及栈和堆

在 Rust 中，所有值默认都由栈分配。值也可以通过创建 `Box<T>` 来装箱（boxed，分配在堆上）。装箱类型是一个智能指针，指向堆分配的 `T` 类型的值。当一个装箱类型离开作用域时，它的析构器会被调用，内部的对象会被销毁，分配在堆上内存会被释放。

装箱的值可以使用 `*` 运算符进行解引用；这会移除掉一个间接层（this removes one layer of indirection. ）。

```
use std::mem;

#[derive(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() {
    // （所有的类型标注都是可要可不要的）
    // 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（boxed）
    let boxed_point: Box<Point> = Box::new(origin());

    // 双重间接装箱（Double indirection）
    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 的大小 = 指针 大小（box size = pointer size）
    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` 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 是可变大小的数组。和 slice（切片）类似，它们的大小在编译期不可预知，但他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，它的长度，还有它的容量。此容量表明了分配多少内存给这 vector。vector 只要小于该容量，就可以随意增长。当临界值就要达到时，vector 会重新分配一个更大的容量。

```
fn main() {
    // 迭代器可以收集到 vector
    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());

    // 在中括号上加索引（索引从 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());

    // 超出索引范围将抛出一个 panic
    println!("Fourth element: {}", xs[3]);
}
```

更多 `Vec` 方法可以在 std::vec 模块中找到。

字符串 **String**

Rust 中有两种字符串类型： `String` 和 `&str` 。

`String` 被存储为一个字节形式（ `Vec<u3>` ）的vector ，但确保一定是一个有效的 UTF-8 序列。 `String` 是堆分配的，可增大且无上限。

`&str` 是一个指向有效 UTF-8 序列的切片（ `&[u8]` ），并可在使用中看成是 `String` ，就如同 `&[T]` 是 `Vec<T>` 的一个视图。（原文： `&str` is a slice (`&[u8]`) that always points to a valid UTF-8 sequence, and can be used to view into a `String` , just like `&[T]` is a view into `Vec<T>` .）（您是否有更好的翻译？请改进此句翻译，感谢！）

```
fn main() {
    // （所有的类型标注都是都是多余）
    // 一个指向在只读内存中堆分配字符串的引用
    let pangram: &'static str = "the quick brown fox jumps over the lazy dog";
    println!("Pangram: {}", pangram);

    // 逆序迭代单词，不用分配新的字符串
    // （原文：Iterate over words in reverse, no new string is allocated）
    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(", ");
    }

    // 此切割的字符串是原字符串的一个切片，所以没有执行新分配操作
    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。

选项 `Option`

有时候想要捕捉到程序某部分的失败信息，而不调用 `panic!`；这可使用 `Option` 枚举来完成。

`Option<T>` 枚举有两个变量：

- `None`，表明失败或缺少值
- `Some(value)`，元组结构体，使用 `T` 类型装包了一个值 `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` 变量
        None
    } else {
        // 结果 Result 被装包成 `Some` 变量
        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` 变量将展开解包后的值。
    // （原文：Unwrapping a `Some` variant will extract the value wrapped.）
    println!("{:?} unwraps to {:?}", optional_float, optional_float.unwrap());

    // 解包 `None` 变量将会引发 `panic!`。
    println!("{:?} unwraps to {:?}", none, none.unwrap());
```

```
}
```

结果 `Result`

我们前面已经看到 `Option` 枚举可以用于函数可能失败的返回值，其中 `None` 可以返回以表明失败。但是有时要强调为什么一个操作会失败。为达成这点，我们提供了 `Result` 枚举。

`Result<T, E>` 枚举拥有两个变量：

- `Ok(value)` 表示操作成功，并装包操作返回的 `value`（ `value` 拥有 `T` 类型）。
- `Err(why)`，表示操作失败，并装包 `why`，它（能按照所希望的方式）解释了失败的原因（ `why` 拥有 `E` 类型）。

```
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，反而让我们返回失败的理由，并装包成 `Err`
            Err(MathError::DivisionByZero)
        } else {
            // 此操作是有效的，返回装包成 `Ok` 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 {
    // 这是一个三层的匹配金字塔!
    // （原文：This is a three level match pyramid!）
    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));
}
```

?

使用匹配链接结果会得到极其繁琐的内容；幸运的是，`?` 运算符可以使事情再次变得干净漂亮。`?` 运算符用在返回值为 `Result` 的表式式后面，等同于这样一个匹配表式，其中 `Err(err)` 分支展开成提前（返回）`return Err(err)`，同时 `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 `DivisionByZero` 将被返回
        let ratio = div(x, y)?;

        // 如果 `ln` “失败”了，那么 `NegativeLogarithm` 将被返回
```

```
        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` 。

panic!

`panic!` 宏可用于产生一个 panic （恐慌），并开始展开它的栈。在展开栈的同时，运行时将会释放该线程所拥有的所有资源，是通过调用对象的析构函数完成。

因为我们正在处理的程序只有一个线程， `panic!` 将会引发程序上报 panic 消息并退出。

```
// 再次实现整型的除法（/）
fn division(dividend: i32, divisor: i32) -> i32 {
    if divisor == 0 {
        // 除以一个 0 时会引发一个 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` 在此处将被销毁
}
```

由分析知道， 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 通过整型索引来存储值，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 在空间多余时能够缩小自身（原文：HashMaps can also shrink themselves when they have excess space. ）。创建 HashMap，可以使用适当的初始化容器（starting capacity） `HashMap::with_capacity(unit)` ，或者使用 `HashMap::new()` 来获得一个带有默认初始容器的 HashMap（推荐方式）。

```
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，该迭代器获得
    // 任意顺序的 (&'a key, &'a value) 对。
    // （原文：`HashMap::iter()` returns an iterator that yields
    // (&'a key, &'a value) pairs in arbitrary order.）
    for (contact, &number) in contacts.iter() {
        println!("Calling {}: {}", contact, call(number));
    }
}
```

了解更多关于映射（map）和散列映射（hash map）（通常也称作散列表，哈希表）的实现原理，可以查看 Wikipedia 的词条散列表。

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`，以及所有这类型的变量
- `String` 和 `&str` （友情提示：可以创建一个由 `String` 构成键的 `HashMap`，并以一个 `&str` 来调用 `.get()` ）（原文： `String` and `&str` (protip: you can have a `HashMap` keyed by `String` and call `.get()` with an `&str`)）

需要注意的是 `f32` 和 `f64` 没有实现 `Hash`，很大程度上是由于浮点精度误差（floating-point precision error）会使浮点类型作为散列映射键发生严重的错误。

对于所有的集合类（collection），如果它们包含的类型都分别实现 `Eq` 和 `Hash`，那么这些集合类也都会实现 `Eq` 和 `Hash` 。例如，若 `T` 实现了 `Hash`，则 `Vec<T>` 也会实现 `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内容，你可以实现自己定制的 `Eq` 和/或 `Hash` 。本指南不包含实现 `Hash` 的细节内容。

为了玩玩怎么使用 `HashMap` 中的 `struct`，让我们试着做一个非常简易的登录系统：

```
use std::collections::HashMap;

// Eq 要求你对此类型派生了 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**

考虑 `HashSet` 作为一个 `HashMap`，在此处我们只关心键（`HashSet<T>` 实际上只是一个包围 `HashMap<T, ()>` 的装包（wrapper））。（原文：Consider a `HashSet` as a `HashMap` where we just care about the keys (`HashSet<T>` is, in actuality, just a wrapper around `HashMap<T, ()>`).）

“关键点是什么呢？”你可能会这样问。“我可以将键只存储到一个 `Vec` 中。”

`HashSet` 的独特之处在于，它保证了不会拥有重复的元素。这是任何集合组合遵循的规定。`HashSet` 只是一个实现。（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。

对于从来不多次保存同一事物，以及判断是否已经得到某个事物的情况，这是相当棒的。（原文：This is great for when you never want more than one of something, or when you want to know if you've already got something.）

不过集合（set）可以做更多的事。

集合拥有 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）：获取落在第一个集合而不在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元素的集合，但不同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。

```
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);
```

```
    // 若一个组合的元素类型实现了 `Debug`，那么该组合也就实现了 `Debug`。
    // 这通常将元素打印成这样的格式 `[dlem1, 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>>());
}
```

（例子修改自文档。）

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转移闭包（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 {
        // 等待线程到结束。返回一个结果。
        let _ = child.join();
    }
}
```

这些线程由操作系统调度（schedule）。

通道

Rust 针对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
    // 消息的类型（类型标注是可有可无的）
    let (tx, rx): (Sender<i32>, Receiver<i32>) = mpsc::channel();

    for id in 0..NTHREADS {
        // sender 发送端可被复制
        let thread_tx = tx.clone();

        // 每个线程都将通过通道来发送它的 id
        thread::spawn(move || {
            // 此线程取得 `thread_tx` 所有权
            // 每个线程都在通道中排队列出消息
            // （原文：The thread takes ownership over `thread_tx`
            // Each thread queues a message in the channel）
            thread_tx.send(id).unwrap();

            // 发送是一个非阻塞操作，线程将在发送完消息后继续进行
            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());
    }

    // 显示已发送消息的次序
    println!("{:?}", ids);
}
```

路径 **Path**

`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。预处理会导入适合特定平台的 `Path` 变量（原文：The prelude exports the appropriate platform-specific `Path` variant.）。

`Path` 可从多种类型创建，几乎所有实现了 `BytesContainer` trait 的类型都可以，比如 string，并提供了几种方法从路径指向的文件/目录中获取信息。（原文：A `Path` can be created from almost any type that implements the `BytesContainer` trait, like a string, and provides several methods to get information from the file/directory the path points to.）

注意 `Path` 在内部并没有表示为一个 UTF-8 字符串，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（free），且可能失败（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` 使用操作系统的特定分隔符来合并路径，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 slice
    match new_path.to_str() {
        None => panic!("new path is not a valid UTF-8 sequence"),
        Some(s) => println!("new path is {}", s),
    }
}
```

要记得查阅其他的 `Path` 方法（`posix::Path` 或 `windows::Path`），还有 `FileStat` 结构体。

文件输入输出 I/O

`File` 结构体表示一个被打开的文件（它装包了一个文件描述符），并赋予了针对底层文件的读和/或写能力。（原文：The `File` struct represents a file that has been opened (it wraps a file descriptor), and gives read and/or write access to the underlying file.）

由于在进行文件 I/O（输入/输出）操作时很多情形都可能出现错误，因此所有的 `File` 方法都返回 `io::Result<T>` 类型，这是 `Result<T, io::Error>` 的别名。

这使得所有 I/O 操作的失败都变成显式内容。借助这点，程序员可以看到所有的失败路径，并鼓励主动去处理这些情形。

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` 拥有一个资源，文件描述符（file descriptor），以及在文件丢弃时管理好关闭文件的操作。（原文：A `File` owns a resource, the file descriptor and takes care of closing the file when it is `drop`ed.）

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;
use std::path::Path;

fn main() {
    // 给所需的文件创建一个路径
    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.description()),
        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.description()),
        Ok(_) => print!("{} contains:\n{}", display, s),
    }

    // `file` 离开作用域，并且 `hello.txt` 文件将被关闭。
}
```

下面是预期成功的输出：

```
$ 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::error::Error;
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.description()),
        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.description())
        },
        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。

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);
    }
}
```

（再试试前面的例子，给 `rustc` 命令传入一个错误的标志）

管道

`Process` 结构体代表了一个正在运行的子进程，并公开了 `stdin` （标准输入）， `stdout` （标准输出） 和 `stderr` （标准错误） 句柄，通过管道和底层的进程交互。（原文：The `Process` struct represents a running child process, and exposes the `stdin` , `stdout` and `stderr` handles for interaction with the underlying process via pipes.）

```
use std::error::Error;
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() {
    // 触发 `wc` 命令（原文：Spawn the `wc` command）
    let process = match Command::new("wc")
                                .stdin(Stdio::piped())
                                .stdout(Stdio::piped())
                                .spawn() {
        Err(why) => panic!("couldn't spawn wc: {}", why.description()),
        Ok(process) => process,
    };

    // 将字符串写入 `wc` 的 `stdin`。
    //
    // `stdin` 拥有 `Option<ChildStdin>` 类型，不过既然我们已经知道这个实例
    // 只能拥有一个，那么我们可以直接解包（`unwrap`）它。
    // （原文：`stdin` has type `Option<ChildStdin>`, but since we know this instance
    // must have one, we can directly `unwrap` it.）
    match process.stdin.unwrap().write_all(PANGRAM.as_bytes()) {
        Err(why) => panic!("couldn't write to wc stdin: {}",
                           why.description()),
        Ok(_) => println!("sent pangram to wc"),
    }

    // 因为 `stdin` 在上面调用后就不再存活，所以它被销毁了，且管道被关闭。
    //
    // 这点非常重要，否则 `wc` 不会开始处理我们刚刚发送的输入。

    // `stdout` 域也拥有 `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.description()),
        Ok(_) => print!("wc responded with:\n{}", s),
```

```
    }
}
```

等待 Wait

如果你想等待 `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。

```
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 = try!(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 = try!(File::create(path));

    f.write_all(s.as_bytes())
}

// `% touch path`（忽略已存在文件）的简单实现
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());
});
```

```
}
```

下面是预期成功的输出：

```
$ 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
`walk a`
> a/c
> a/c/b.txt
> a/c/e.txt
> a/c/d
> a/b.txt
`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

命令行参数可使用 `std::env::args` 进行接收，这将返回一个迭代器，该迭代器会对各个参数产生一个字符串。

```
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]);

    // 其余的参数充当一般的命令行参量。
    // 调用程序方式如下：
    //   $ ./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"].
```

参数分析

匹配可以用来解析简单的参数：

```
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."),
            }
        },
        // 一条命令和一个传入参数
        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 提供了外部语言函数接口（Foreign Function Interface，FFI）到 C 语言库。外部语言函数必须声明在一个 `extern` 代码块，且该代码块要带有一个包含外部语言库名称的 `#[link]` 属性。

```
use std::fmt;

// 此外部代码块链接到 libm 库
#[link(name = "m")]
extern {
    // 这是外部语言函数
    // 这计算了一个单精度复数的平方根
    fn csqrtf(z: Complex) -> Complex;
}

fn main() {
    // z = -1 + 0i
    let z = Complex { re: -1., im: 0. };

    // 调用一个外部语言函数是一种不安全的操作
    let z_sqrt = unsafe {
        csqrtf(z)
    };

    println!("the square root of {:?} is {:?}", z, z_sqrt);
}

// 最小化实现单精度复数
#[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)
        }
    }
}
```

由于调用外部语言函数通常被认为是不安全的，因此围绕它们编写安全的装包代码是相当普遍的。

```
use std::fmt;

#[link(name = "m")]
extern {
    fn ccosf(z: Complex) -> Complex;
}

// 安全装包（原文：safe wrapper）
fn cos(z: Complex) -> Complex {
    unsafe { ccosf(z) }
}

fn main() {
    // z = 0 + 1i
    let z = Complex { re: 0., im: 1. };

    println!("cos({:?}) = {:?}", z, cos(z));
}

// 最小化实现单精度复数
#[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)
        }
    }
}
```

补充

有些主题并非没有教你怎么编写程序，但为你提供工具和基础设施支持，这会让编程工作变得更美好。这些主题包括：

- 文档：通过附带的 `rustdoc` 生成库文档给用户。
- 测试：对库创建测试套件，确保库准确地实现了你想要的功能。
- 基准测试（benchmarking）：生成基准以保证高效运行。

文档

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，这些注释就会编译成文档。它们使用 `///` 标记，并支持 `Markdown` 。

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

/// 这里给出一个人类
pub struct Person {
    /// 一个人必须有名字，不管 Juliet 多讨厌他/她。
    name: String,
}

impl Person {
    /// 返回给定名字的人
    ///
    /// # 参数
    ///
    /// * `name` - 字符串 slice，代表人物的名称
    ///
    /// # 示例：
    ///
    ///
```

```
/// // 可以在注释的特定标记内编写 Rust。
/// // 如果可以通过 --- 测试传递给 Rustdoc，它将会帮你进行测试！
/// 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，以便它可以将代码链接成各个文档测试程序：

```bash
$ rustc doc.rs --crate-type lib
$ rustdoc --test --extern doc="libdoc.rs"
```

（当你在库 crate 上运行 `cargo test` 时， `Cargo` 将自动生成并运行正确的 `rustc` 和 `rustdoc` 命令。）
```

测试

函数可以通过这些属性（attribute） 进行测试：

- `#[test]` 将一个函数标记为一个单元测试。该函数不能接受参数且返回空。
- `#[should_panic]` 将一个函数标记为 panic 测试。

```
// 当且仅当测试套件没有运行时，才条件编译 `main` 函数。
#[cfg(not(test))]
fn main() {
    println!("If you see this, the tests were not compiled nor ran!");
}

// 当且仅当测试套件运行时，才条件编译 `test` 模块。
#[cfg(test)]
mod test {
    // 需要一个辅助函数 `distance_test`。
    fn distance(a: (f32, f32), b: (f32, f32)) -> f32 {
        (
            (b.0 - a.0).powi(2) +
            (b.1 - a.1).powi(2)
        ).sqrt()
    }

    #[test]
    fn distance_test() {
        assert!(distance((0f32, 0f32), (1f32, 1f32)) == (2f32).sqrt());
    }

    #[test]
    #[should_panic]
    fn failing_test() {
        assert!(1i32 == 2i32);
    }
}
```

通过 `cargo test` 或 `rustc --test` 运行测试。

```
$ rustc --test unit_test.rs
$ ./unit_test

running 2 tests
test test::distance_test ... ok
test test::failing_test ... ok

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

若 `--test` 没有包含进来，则会出现这样的情况：

```
$ rustc unit_test.rs
$ ./unit_test
If you see this, the tests were not compiled nor ran!
```

参见：

属性, 条件编译, 和 `mod` .

不安全操作

为了介绍本章内容，我们借用官方文档的一句话, “在基本代码中尽可能减少不安全的代码”（"one should try to minimize the amount of unsafe code in a code base."）。记住这句话，接着我们进入学习！在 Rust 中，不安全代码块是用于避开编译器的保护策略；具体地说，不安全代码块主要有 4 方面内容：

- 解引用裸指针
- 通过 FFI 调用函数（这个内容在本书其他章节介绍过了）
- 使用 `std::mem::transmute` 来强制转型（change type）
- 内联汇编(inline assembly)

原始指针

原始指针（裸指针） `*` 和引用 `&T` 有类似的功能，但引用总是安全的，因为它们保证指向一个有效的数据，这得益于借用检查器（borrow checker）。解引用一个裸指针只能通过不安全代码块中来完成。

```
fn main() {
    let raw_p: *const u32 = &10;

    unsafe {
        assert!(*raw_p == 10);
    }
}
```

Transmute（转变）

从一种类型变到另一种类型的允许简单转换，但是两种类型必须拥有相同的大小和排列：

```
fn main() {
    let u: &[u8] = &[49, 50, 51];

    unsafe {
        assert!(u == std::mem::transmute::<&str, &[u8]>("123"));
    }
}
```