目　录

致谢

当前文档 《Rust语言学习笔记》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-28。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/rust-notes

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

Rust语言学习笔记

目录

参考资料

- The Rust Programming Language
- The Rust Reference
- The Rustonomicon
- Rust by Example
- This Week in Rust

项目地址

- https://github.com/photino/rust-notes
- http://photino.gitbooks.io/rust-notes

入门指南

入门指南

我们首先介绍如何配置开发环境，编译流程，以及代码规范。

开发环境

在Ubuntu 14.04下安装最新版的Rust:

```
1. $ curl -sSf https://static.rust-lang.org/rustup.sh | sh -s -- --
   channel=nightly
```

为Atom编辑器安装Rust语法高亮支持:

```
1. $ apm install language-rust
```

如果需要一个IDE，推荐使用Tokamak。

编译流程

使用Cargo创建 `hello-world` 项目:

```
1. $ cargo new hello-world --bin
```

加上 `--bin` 选项是因为我们需要生成一个可执行程序，即 `src/` 目录下有一个 `main.rs` 文件。

如果不加则默认生成Rust库，`src/` 下有一个 `lib.rs` 文件。当一个外部包使用 `extern crate` 导入时，
Cargo会自动将连字符 `-` 转化为下划线 `_` 。

在项目目录下，Cargo还生成了一个TOML格式的配置文件 `Cargo.toml` ，
其功能类似于Node.js中的 `package.json` 。

Cargo自动生成的 `src/main.rs` 代码：

```
1. fn main() {
2.     println!("Hello, world!");
3. }
```

其中 `println!` 是一个宏（macro），在Rust中宏通常是以 `!` 结尾。

使用Cargo编译项目非常简单：

```
1. $ cargo build
```

如果需要编译并执行，可以使用 `cargo run` 命令。如果需要发布项目，应该使用 `--release` 选项开启优化：

```
1. $ cargo build --release
```

在Rust中解决依赖性相当容易，只需要在 `Cargo.toml` 中添加 `[dependencies]` 字典：

```
1. [dependencies]
2. semver = "0.1.19"
```

这里的 `semver` 库主要负责按语义化版本规范来匹配版本号：

```
1. // in src/main.rs
```

```
2.
3.  extern crate semver;
4.
5.  use semver::Version;
6.  use semver::Identifier::{AlphaNumeric, Numeric};
7.
8.  fn main() {
9.      assert!(Version::parse("1.2.3-alpha.2") == Ok(Version {
10.         major: 1u64,
11.         minor: 2u64,
12.         patch: 3u64,
13.         pre: vec!(AlphaNumeric("alpha".to_string()), Numeric(2)),
14.         build: vec!(),
15.     }));
16.
17.     println!("Versions compared successfully!");
18. }
```

其中函数 `Ok()` 来自于 `std::Result` ，通过 `std::prelude` 模块被预先导入。

代码规范

可以参考Rust风格指南，
这里简单强调几点：

- 使用4个空格进行缩进。
- 在单行的花括号内侧各使用一个空格。
- 不要特意在行间使用多余的空格来实现对齐。
- 避免使用块注释 `/* ... */` 。
- 文档注释的第一行应该是关于该部分代码的一行简短总结。
- 当结束分隔符出现在一个单独的行尾时，应该在其末尾加上逗号。

基本语法

基本语法

变量绑定

在Rust中，变量绑定 (variable bindings) 是通过 `let` 关键字声明的：

```
1. let x = 5;
2. let mut x = 5;
3. let x: i32 = 5;
4. let (a, b) = (3, 4);
```

其中变量类型如 `i32` 一般都是可以省略的，因为Rust使用了类型推断 (type inference)。
Rust还通过模式匹配 (pattern matching) 对变量进行解构，这允许我们同时对多个变量进行赋值。

有几点是需要特别注意的：

- 变量默认是不可改变的 (immutable)，如果需要改变一个变量的值需要显式加上 `mut` 关键字。

- 变量具有局部作用域，被限制在所属的代码块内，并且允许变量覆盖（variable shadowing）。
- Rust默认开启属性`#[warn(unused_variable)]`，会对未使用的变量（以`_`开头的除外）发出警告。
- Rust允许先声明变量然后再初始化，但是使用未被初始化的变量会产生一个编译时错误。

原生类型

Rust内置的原生类型（primitive types）有以下几类：

- 布尔类型：有两个值`true`和`false`。
- 字符类型：表示单个Unicode字符，存储为4个字节。
- 数值类型：分为有符号整数（`i8`，`i16`，`i32`，`i64`，`isize`）、
  无符号整数（`u8`，`u16`，`u32`，`u64`，`usize`）以及浮点数（`f32`，`f64`）。
- 字符串类型：最底层的是不定长类型`str`，更常用的是字符串切片`&str`和堆分配字符串`String`，
  其中字符串切片是静态分配的，有固定的大小，并且不可变，而堆分配字符串是可变的。
- 数组：具有固定大小，并且元素都是同种类型，可表示为`[T; N]`。
- 切片：引用一个数组的部分数据并且不需要拷贝，可表示为`&[T]`。
- 元组：具有固定大小的有序列表，每个元素都有自己的类型，通过解构或者索引来获得每个元素的值。
- 指针：最底层的是裸指针`*const T`和`*mut T`，但解引用它们是不安全的，必须放到`unsafe`块里。

- 函数：具有函数类型的变量实质上是一个函数指针。
- 元类型：即`()`，其唯一的值也是`()`。

```
// boolean type
let t = true;
let f: bool = false;

// char type
let c = 'c';

// numeric types
let x = 42;
let y: u32 = 123_456;
let z: f64 = 1.23e+2;
let zero = z.min(123.4);
let bin = 0b1111_0000;
let oct = 0o7320_1546;
let hex = 0xf23a_b049;

// string types
let str = "Hello, world!";
let mut string = str.to_string();

// arrays and slices
let a = [0, 1, 2, 3, 4];
let middle = &a[1..4];
let mut ten_zeros: [i64; 10] = [0; 10];

// tuples
let tuple: (i32, &str) = (50, "hello");
let (fifty, _) = tuple;
let hello = tuple.1;

// raw pointers
let x = 5;
let raw = &x as *const i32;
let points_at = unsafe { *raw };

```

```
// functions
fn foo(x: i32) -> i32 { x }
let bar: fn(i32) -> i32 = foo;
```

有几点是需要特别注意的：

- 数值类型可以使用`_`分隔符来增加可读性。
- Rust还支持单字节字符`b'H'`以及单字节字符串`b"Hello"`，仅限制于ASCII字符。
  此外，还可以使用`r#"..."#`标记来表示原始字符串，不需要对特殊字符进行转义。
- 使用`&`符号将`String`类型转换成`&str`类型很廉价，
  但是使用`to_string()`方法将`&str`转换到`String`类型涉及到分配内存，
  除非很有必要否则不要这么做。
- 数组的长度是不可变的，动态的数组称为向量（vector），可以使用宏`vec!`创建。
- 元组可以使用`==`和`!=`运算符来判断是否相同。
- 不多于32个元素的数组和不多于12个元素的元组在值传递时是自动复制的。
- Rust不提供原生类型之间的隐式转换，只能使用`as`关键字显式转换。
- 可以使用`type`关键字定义某个类型的别名，并且应该采用驼峰命名法。

```
// explicit conversion
let decimal = 65.4321_f32;
let integer = decimal as u8;
let character = integer as char;

// type aliases
type NanoSecond = u64;
```

```
type Point = (u8, u8);
```

结构体

结构体（struct）是一种记录类型，所包含的每个域（field）都有一个名称。
每个结构体也都有一个名称，通常以大写字母开头，使用驼峰命名法。
元组结构体（tuple struct）是由元组和结构体混合构成，元组结构体有名称，
但是它的域没有。当元组结构体只有一个域时，称为新类型（newtype）。
没有任何域的结构体，称为类单元结构体（unit-like struct）。
结构体中的值默认是不可变的，需要使用 `mut` 使其可变。

```
// structs
struct Point {
  x: i32,
  y: i32,
}
let mut point = Point { x: 0, y: 0 };

// tuple structs
struct Color(u8, u8, u8);
let android_green = Color(0xa4, 0xc6, 0x39);
let (red, green, blue) = android_green;

// A tuple struct's constructors can be used as functions.
struct Digit(i32);
let v = vec![0, 1, 2];
let d: Vec<Digit> = v.into_iter().map(Digit).collect();

// newtype: a tuple struct with only one element
struct Inches(i32);
let length = Inches(10);
```

```
let Inches(integer_length) = length;

// unit-like structs
struct Null;
let empty = Null;
```

一个包含 `..` 的 `struct` 可以用来从其它结构体拷贝一些值或者在解构时忽略一些域：

```
#[derive(Default)]
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point3d::default();
let point = Point3d { y: 1, ..origin };
let Point3d { x: x0, y: y0, .. } = point;
```

需要注意，Rust在语言级别不支持域可变性 (field mutability)，所以不能这么写：

```
struct Point {
    mut x: i32,
    y: i32,
}
```

这是因为可变性是绑定的一个属性，而不是结构体自身的。可以使用 `Cell<T>` 来模拟：

```
use std::cell::Cell;

struct Point {
    x: i32,
    y: Cell<i32>,
```

```
}

let mut point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);
```

此外，结构体的域默认是私有的，可以使用 `pub` 关键字将其设置成公开。

枚举

Rust有一个集合类型，称为枚举 (enum)，对于一个指定的名称有一组可替代的值，
其中子数据结构可以存储也可以不存储数据，需要使用 `::` 语法来获得每个元素的名称。

```
// enums
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}

let x: Message = Message::Move { x: 3, y: 4 };
```

与结构体一样，枚举中的元素默认不能使用关系运算符进行比较 (如 `==` , `!=` , `>=`)，
也不支持像 `+` 和 `*` 这样的双目运算符，需要自己实现，或者使用 `match` 进行匹配。

枚举默认也是私有的，如果使用 `pub` 使其变为公有，则它的元素也都是默认公有的。
这一点是与结构体不同的：即使结构体是公有的，它的域仍然是默认私

有的。
此外，枚举和结构体也可以是递归的 (recursive)。

函数

要声明一个函数，需要使用关键字 `fn` ，后面跟上函数名，比如

```
1. fn add_one(x: i32) -> i32 {
2.     x + 1
3. }
```

其中函数参数的类型不能省略，可以有多个参数，但是最多只能返回一个值，
提前返回使用 `return` 关键字。Rust编译器会对未使用的函数提出警告，
可以使用属性 `#[allow(dead_code)]` 禁用无效代码检查。

Rust有一个特殊语法适用于分叉函数 (diverging function)，它不返回值：

```
1. fn diverges() -> ! {
2.     panic!("This function never returns!");
3. }
```

其中 `panic!` 是一个宏，使当前执行线程崩溃并打印给定信息。返回类型 `!` 可用作任何类型：

```
1. let x: i32 = diverges();
2. let y: String = diverges();
```

注释

Rust有三种注释：

- 行注释 (line comments): 以`//`开头，仅能注释一行。
- 块注释 (block comments): 以`/*`开头，以`*/`结尾，能注释多行，但是不建议使用。
- 文档注释 (doc comments): 以`///`或者`//!`开头，支持Markdown标记语言，
  其中`///`等价于写属性`#[doc = "..."]`，`//!`等价于`#![doc = "///..."]`，
  配合`rustdoc`可自动生成说明文档。

控制结构

控制结构

If

If是分支（branch）的一种特殊形式，也可以使用 `else` 和 `else if` 。
与C语言不同的是，逻辑条件不需要用小括号括起来，但是条件后面必须跟一个代码块。
Rust中的 `if` 是一个表达式（expression），可以赋给一个变量：

```
1. let x = 5;
2.
3. let y = if x == 5 { 10 } else { 15 };
```

Rust是基于表达式的编程语言，有且仅有两种语句（statement）：

1. 声明语句（declaration statement），比如进行变量绑定的`let`语句。
2. 表达式语句（expression statement），它通过在末尾加上分号`;`来将表达式变成语句，
   丢弃该表达式的值，一律返回元类型`()`。

表示式总是返回一个值，但是语句不返回值或者返回 `()` ，所以以下代

码会报错：

```
let y = (let x = 5);

let z: i32 = if x == 5 { 10; } else { 15; };
```

值得注意的是，在Rust中赋值（如 `x = 5` ）也是一个表达式，返回元类型值 `()` 。

For

Rust中的 `for` 循环与C语言的风格非常不同，抽象结构如下：

```
for var in expression {
    code
}
```

其中 `expression` 是一个迭代器（iterator），具体的例子为 `0..10` (不包含最后一个值)，
或者 `[0, 1, 2].iter()` 。

While

Rust中的 `while` 循环与C语言中的类似。对于无限循环，Rust有一个专用的关键字 `loop` 。
如果需要提前退出循环，可以使用关键字 `break` 或者 `continue` ，
还允许在循环的开头设定标签（同样适用于 `for` 循环）：

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

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

```
    }

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

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

Match

Rust中的 `match` 表达式非常强大，首先看一个例子：

```
let day = 5;

match day {
  0 | 6 => println!("weekend"),
  1 ... 5 => println!("weekday"),
  _ => println!("invalid"),
}
```

其中 `|` 用于匹配多个值， `...` 匹配一个范围（包含最后一个值），并且 `_` 在这里是必须的，
因为 `match` 强制进行穷尽性检查（exhaustiveness checking），必须覆盖所有的可能值。
如果需要得到 `|` 或者 `...` 匹配到的值，可以使用 `@` 绑定变量：

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

使用 `ref` 关键字来得到一个引用：

```
let x = 5;
```

```
let mut y = 5;

match x {
    // the `r` inside the match has the type `&i32`
    ref r => println!("Got a reference to {}", r),
}

match y {
    // the `mr` inside the match has the type `&i32` and is mutable
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

再看一个使用 `match` 表达式来解构元组的例子：

```
let pair = (0, -2);

match pair {
    (0, y) => println!("x is `0` and `y` is `{:?}`", y),
    (x, 0) => println!("`x` is `{:?}` and y is `0`", x),
    _ => println!("It doesn't matter what they are"),
}
```

`match` 的这种解构同样适用于结构体或者枚举。如果有必要，还可以使用 `..` 来忽略域或者数据：

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}

enum OptionalInt {
```

```
13.     Value(i32),
14.     Missing,
15. }
16.
17. let x = OptionalInt::Value(5);
18.
19. match x {
20.     OptionalInt::Value(i) if i > 5 => println!("Got an int bigger
    than five!"),
21.     OptionalInt::Value(..) => println!("Got an int!"),
22.     OptionalInt::Missing => println!("No such luck."),
23. }
```

此外，Rust还引入了 `if let` 和 `while let` 进行模式匹配：

```
1.  let number = Some(7);
2.  let mut optional = Some(0);
3.
4.  // If `let` destructures `number` into `Some(i)`, evaluate the
    block.
5.  if let Some(i) = number {
6.      println!("Matched {:?}!", i);
7.  } else {
8.      println!("Didn't match a number!");
9.  }
10.
11. // While `let` destructures `optional` into `Some(i)`, evaluate the
    block.
12. while let Some(i) = optional {
13.     if i > 9 {
14.         println!("Greater than 9, quit!");
15.         optional = None;
16.     } else {
17.         println!("`i` is `{:?}`. Try again.", i);
18.         optional = Some(i + 1);
19.     }
20. }
```

模块系统

模块系统

Rust有两个与模块（module）系统相关的独特术语：`crate` 和 `module`，
其中包装箱（crate）与其它语言中的 libary 或者 package 作用一样。
每个包装箱都有一个隐藏的根模块，在根模块下可以定义一个子模块树，
其路径采用 `::` 作为分隔符。包装箱由条目（item）构成，多个条目通过模块组织在一起。

定义模块

使用 `mod` 关键字定义我们的模块：

```
// in src/lib.rs

mod chinese {
    mod greetings {

    }

    mod farewells {

    }
```

```
}

mod english {
    mod greetings {

    }

    mod farewells {

    }
}
```

定义了四个子模块 `chinese::{greetings, farewells}` 和 `english::{greetings, farewells}` 。
模块默认是私有的，可以使用 `pub` 关键字将其设置成公开，只有公开的条目才允许在模块外部访问。

实践中更好的组织方式是将一个包装箱分拆到多个文件：

```
// in src/lib.rs

pub mod chinese;

pub mod english;
```

这两句声明告诉Rust查看 `src/chinese.rs` 和 `src/english.rs` ，或者 `src/chinese/mod.rs` 和 `src/english/mod.rs` 。
先添加一些函数：

```
// in src/chinese/greetings.rs

pub fn hello() -> String {
    "你好！".to_string()
}
```

```
// in src/chinese/farewells.rs

pub fn goodbye() -> String {
    "再见！".to_string()
}
```

```
// in src/english/greetings.rs

pub fn hello() -> String {
    "Hello!".to_string()
}
```

```
// in src/english/farewells.rs

pub fn goodbye() -> String {
    "Goodbye!".to_string()
}
```

函数默认也是私有的，为了后面的使用我们需要 `pub` 关键字使其成为公有。

导入 crate

为了使用我们前面创建的名为 `phrases` 的包装箱，需要先声明导入

```
// in src/main.rs

extern crate phrases;

fn main() {
    println!("Hello in Chinese: {}",
phrases::chinese::greetings::hello());
}
```

Rust还有一个 `use` 关键字，允许我们导入包装箱中的条目到当前的作

用域内：

```
// in src/main.rs

extern crate phrases;

use phrases::chinese::greetings;
use phrases::chinese::farewells::goodbye;

fn main() {
    println!("Hello in Chinese: {}", greetings::hello());
    println!("Goodbye in Chinese: {}", goodbye());
}
```

但是，我们不推荐直接导入函数，这样更容易导致命名空间冲突，只导入模块是更好的做法。
如果要导入来自同一模块的多个条目，可以使用大括号简写：

```
use phrases::chinese::{greetings, farewells};
```

如果是导入全部，可以使用通配符 `*` 。重命名可以使用 `as` 关键字：

```
use phrases::chinese::greetings as chinese_greetings;
```

有时我们需要将外部包装箱里面的函数导入到另一个模块内，
这时可以使用 `pub use` 来提供扩展接口而不映射代码层级结构。
比如

```
// in src/english/mod.rs

pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;

```

```
8.  mod farewells;
```

其中 `pub use` 声明将函数带入了当前模块中，

使得我们现在有了 `phrases:hello()` 函数和 `phrases:goodbye()` 函数，

即使它们的定义位于 `phrases:hello()`

和 `phrases:goodbye()` 中，
内部代码的组织结构不能反映我们的扩展接口。

默认情况下， `use` 声明表示从根包装箱开始的绝对路径。
此外，我们可以使用 `use self::` 表示相对于当前模块的位置，
`use super::` 表示当前位置的上一级，以 `::` 为前缀的路径表示根包装箱路径。

```
1.  use foo::baz::foobaz; // foo is at the root of the crate
2.
3.  mod foo {
4.      use foo::bar::foobar; // foo is at crate root
5.      use self::baz::foobaz; // self refers to module 'foo'
6.
7.      pub mod bar {
8.          pub fn foobar() { }
9.      }
10.
11.     pub mod baz {
12.         use super::bar::foobar; // super refers to module 'foo'
13.         pub fn foobaz() { }
14.     }
15. }
```

属性

在Rust中，属性 (attribute) 是应用于包装箱、模块或者条目的元数据 (metadata)，
主要用于：

- 实现条件编译 (conditional compilation)
- 设置包装箱名字、版本以及类型
- 取消可疑代码的警告
- 设置编译器选项
- 链接外部库
- 标记测试函数

属性有两种语法： `#![crate_attribute]` 应用于整个包装箱，
而 `#[crate_attribute]` 应用于紧邻的一个模块或者条目。
属性的参数也有三种不同的形式：

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

下面列举几个经常用到的属性：

- `#[path="foo.rs"]` 用于设置一个模块需要载入的文件路径。
- `#[allow(dead_code)]` 用于取消对死代码的默认lint检查。
- `#[derive(PartialEq, Clone)]` 用于自动推导 `PartialEq` 和 `Clone` 这两个特性的实现。

程序测试

程序测试

测试属性

在测试函数前加上 `#[test]` 属性：

```
#[test]
fn it_works() {
    assert!(false);
}
```

其中 `assert!` 宏接受一个参数，如果参数为 `false` ，它会导致 `panic!` 。
运行 `cargo test` 命令，可见该测试失败。如果要反转测试失败，可以加上 `#[should_panic]` 属性：

```
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

测试模块

在测试模块前加上 `#[cfg(test)]` 属性：

```
1.  pub fn add_two(a: i32) -> i32 {
2.      a + 2
3.  }
4.
5.  #[cfg(test)]
6.  mod test {
7.      use super::add_two;
8.
9.      #[test]
10.     fn it_works() {
11.         assert_eq!(4, add_two(2));
12.     }
13. }
```

测试目录

对于集成测试，可以新建一个 `tests` 目录，这样其中的代码就不需要再引入单元风格的测试模块了。

文档测试

对于包含有测试例子的注释文档中，运行 `cargo test` 时也会运行其中包含的测试。

```
1.  //! The `adder` crate provides functions that add numbers to other
    numbers.
2.  //!
3.  //! # Examples
4.  //!
5.  //! ```
6.  //! assert_eq!(4, adder::add_two(2));
7.  //! ```
```

```
8.
9.  /// This function adds two to its argument.
10. ///
11. /// # Examples
12. ///
13. /// ```
14. /// use adder::add_two;
15. ///
16. /// assert_eq!(4, add_two(2));
17. /// ```
18. pub fn add_two(a: i32) -> i32 {
19.     a + 2
20. }
```

错误处理

Rust明确区分两种形式的错误：失败（failture）和恐慌（panic）。
失败是可以通过某种方式恢复的错误，而恐慌（panic）则不能够恢复。

最简单的表明函数会失败的方法是使用 `Option<T>` 类型：

```
1.  fn from_str<A: FromStr>(s: &str) -> Option<A> {
2.
3.  }
```

其中 `from_str()` 返回一个 `Option<A>` 。如果转换成功，它会返回 `Some(value)` ；
如果失败，直接返回 `None` 。对于需要提供出错信息的情形，可以使用 `Result<T, E>` 类型：

```
1.  enum Result<T, E> {
2.      Ok(T),
3.      Err(E),
```

```
4. }
```

如果不想处理错误，可以使用 `unwrap()` 方法来产生恐慌：

```
1. let mut buffer = String::new();
2. let input = io::stdin().read_line(&mut buffer).unwrap();
```

当 `Result` 是 `Err` 时， `unwrap()` 会 `panic!` ，直接退出程序。另一个更好的做法是：

```
1. let input = io::stdin().read_line(&mut buffer)
2.                        .ok()
3.                        .expect("Failed to read line");
```

其中 `ok()` 将 `Result` 转换为 `Option` ， `expect()` 和 `unwrap()` 功能类似，
可以用来提供更多的错误信息。

此外，还可以使用宏 `try!` 来封装表达式，当 `Result` 是 `Err` 时会从当前函数提早返回 `Err` 。

内存安全

内存安全

Rust推崇安全与速度至上，它没有垃圾回收机制，却成功实现了内存安全 (memory safety)。

所有权

在Rust中，所有权 (ownership) 系统是零成本抽象 (zero-cost abstraction) 的一个主要例子。
对所有权的分析是在编译阶段就完成的，并不带来任何运行时成本 (run-time cost)。
默认情况下，Rust是在栈 (stack) 上分配内存，对栈上空间变量的再赋值都是复制的。
如果要在堆 (heap) 中分配，必须使用盒子来构造：

```
1.  let x = Box::new(5);
```

其中 `Box::new()` 创建了一个 `Box<i32>` 来存储整数 `5` ，此时变量 `x` 具有该盒子的所有权。
当 `x` 退出代码块的作用域时，它所分配的内存资源将随之释放，这是编译器自动完成的。

考虑下面这段代码：

```
1.  fn main() {
2.      let x = Box::new(5);
3.
4.      add_one(x);
5.
6.      println!("{}", x);
7.  }
8.
9.  fn add_one(mut num: Box<i32>) {
10.     *num += 1;
11. }
```

调用 `add_one()` 时，变量 `x` 的所有权也转移 (move) 给了变量 `num` 。
当所有权转移时，可变性可以从不可变变成可变的。函数完成后，`num` 占有的内存将自动释放。当 `println!` 再次使用已经没有所有权的变量 `x` 时，
编译器就会报错。一种可行的解决办法是修改 `add_one()` 函数使其返回 `Box` ，
把所有权再转移回来。更好的做法是引入所有权借用 (borrowing)。

借用

在Rust中，所有权的借用是通过引用 `&` 来实现的:

```
1.  fn main() {
2.      let mut x = 5;
3.
4.      add_one(&mut x);
5.
6.      println!("{}", x);
7.  }
8.
9.  fn add_one(num: &mut i32) {
```

```
10.     *num += 1;
11. }
```

调用 `add_one()` 时，变量 `x` 把它的所有权以可变引用借给了变量 `num` 。函数完成后，
`num` 又把所有权还给了 `x` 。如果是以不可变引用借出，则借用者只能读而不能改。

有几点是需要特别注意的：

- 变量、函数、闭包以及结构体都可以成为借用者。
- 一个资源只能有一个所有者，但是可以有多个借用者。
- 资源一旦以可变借出，所有者就不能再访问资源，也不能再借给其它绑定。
- 资源一旦以不可变借出，所有者就不能再改变资源，也不能再以可变的形式借出，
  但可以以不可变的形式继续借出。

生存期

Rust 通过引入生存期 (lifetime) 的概念来确定一个引用的作用域：

```
1. struct Foo<'a, 'b> {
2.     x: &'a i32,
3.     y: &'b i32,
4. }
5.
6. fn main() {
7.     let a = &5;
8.     let b = &8;
9.     let f = Foo { x: a, y: b };
10.
11.     println!("{}", f.x + f.y);
```

```
12. }
```

因为结构体 `Foo` 有自己的生存期，所以我们需要给它所包含的域指定新的生存期 `'a` 和 `'b` ，
从而确保对 `i32` 的引用比对 `Foo` 的引用具有更长的生存期，避免悬空指针
(dangling pointer) 的问题。

Rust预定义的 `'static` 具有和整个程序运行时相同的生存期，主要用于声明全局变量。
由 `const` 关键字定义的常量也具有 `'static` 生存期，但是它们会被内联到使用它们的地方。

```
1. const N: i32 = 5;
2.
3. static NUM: i32 = 5;
4. static NAME: &'static str = "David";
```

其中类型标注是不可省略的，并且必须使用常量表达式初始化。
对于通过 `static mut` 绑定的变量，则只能在 `unsafe` 代码块里使用。

对于共享所有权，需要使用标准库中的 `Rc<T>` 类型：

```
1. use std::rc::Rc;
2.
3. struct Car {
4.     name: String,
5. }
6.
7. struct Wheel {
8.     size: i32,
9.     owner: Rc<Car>,
10. }
11.
```

```
12. fn main() {
13.     let car = Car { name: "DeLorean".to_string() };
14. 
15.     let car_owner = Rc::new(car);
16. 
17.     for _ in 0..4 {
18.         Wheel { size: 360, owner: car_owner.clone() };
19.     }
20. }
```

如果是在并发中共享所有权，则需要使用线程安全的 `Arc<T>` 类型。

Rust支持生存期省略 (lifetime elision)，它允许在特定情况下不写生存期标记，
此时会遵从三条规则：

- 每个被省略生存期标记的函数参数具有各不相同的生存期。
- 如果只有一个输入生存期 (input lifetime)，那么不管它是否省略，
  这个生存期都会赋给函数返回值中所有被省略的生存期。
- 如果有多个输入生存期，并且其中有一个是 `&self` 或者 `&mut self`，
  那么 `self` 的生存期会赋给所有被省略的输出生存期 (output lifetime)。

编程范式

编程范式

Rust是一个多范式（multi-paradigm）的编译型语言。除了通常的结构化、命令式编程外，
还支持以下范式。

函数式编程

Rust使用闭包（closure）来创建匿名函数：

```
let num = 5;
let plus_num = |x: i32| x + num;
```

其中闭包 `plus_num` 借用了它作用域中的 `let` 绑定 `num`。如果要让闭包获得所有权，
可以使用 `move` 关键字：

```
let mut num = 5;

{
    let mut add_num = move |x: i32| num += x;

    add_num(5);
}
```

```
8.
9.  assert_eq!(5, num);
```

Rust 还支持高阶函数 (high order function)，允许把闭包作为参数来生成新的函数：

```
1.  fn add_one(x: i32) -> i32 { x + 1 }
2.
3.  fn apply<F>(f: F, y: i32) -> i32
4.      where F: Fn(i32) -> i32
5.  {
6.      f(y) * y
7.  }
8.
9.  fn factory(x: i32) -> Box<Fn(i32) -> i32> {
10.     Box::new(move |y| x + y)
11. }
12.
13. fn main() {
14.     let transform: fn(i32) -> i32 = add_one;
15.     let f0 = add_one(2i32) * 2;
16.     let f1 = apply(add_one, 2);
17.     let f2 = apply(transform, 2);
18.     println!("{}, {}, {}", f0, f1, f2);
19.
20.     let closure = |x: i32| x + 1;
21.     let c0 = closure(2i32) * 2;
22.     let c1 = apply(closure, 2);
23.     let c2 = apply(|x| x + 1, 2);
24.     println!("{}, {}, {}", c0, c1, c2);
25.
26.     let box_fn = factory(1i32);
27.     let b0 = box_fn(2i32) * 2;
28.     let b1 = (*box_fn)(2i32) * 2;
29.     let b2 = (&box_fn)(2i32) * 2;
30.     println!("{}, {}, {}", b0, b1, b2);
31.
```

```
    let add_num = &(*box_fn);
    let translate: &Fn(i32) -> i32 = add_num;
    let z0 = add_num(2i32) * 2;
    let z1 = apply(add_num, 2);
    let z2 = apply(translate, 2);
    println!("{}, {}, {}", z0, z1, z2);
}
```

面向对象编程

Rust通过 `impl` 关键字在 `struct` 、 `enum` 或者 `trait` 对象上实现方法调用语法 (method call syntax)。
关联函数 (associated function) 的第一个参数通常为 `self` 参数，有3种变体:

- `self` ，允许实现者移动和修改对象，对应的闭包特性为 `FnOnce` 。
- `&self` ，既不允许实现者移动对象也不允许修改，对应的闭包特性为 `Fn` 。
- `&mut self` ，允许实现者修改对象但不允许移动，对应的闭包特性为 `FnMut` 。

不含 `self` 参数的关联函数称为静态方法 (static method)。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
```

```
        }
    }

    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());

    // use associated function and method chaining
    println!("{}", Circle::new(0.0, 0.0, 2.0).area());
}
```

为了描述类型可以实现的抽象接口 (abstract interface), Rust引入了特性 (trait) 来定义函数类型签名 (function type signature):

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

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
```

```
18.     x: f64,
19.     y: f64,
20.     side: f64,
21. }
22.
23. impl HasArea for Square {
24.     fn area(&self) -> f64 {
25.         self.side * self.side
26.     }
27. }
28.
29. fn print_area<T: HasArea>(shape: T) {
30.     println!("This shape has an area of {}", shape.area());
31. }
```

其中函数 `print_area()` 中的泛型参数 `T` 被添加了一个名为 `HasArea` 的特性约束 (trait constraint)，用以确保任何实现了 `HasArea` 的类型将拥有一个 `.area()` 方法。如果需要多个特性限定 (multiple trait bounds)，可以使用 `+`：

```
1. use std::fmt::Debug;
2.
3. fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
4.     x.clone();
5.     y.clone();
6.     println!("{:?}", y);
7. }
8.
9. fn bar<T, K>(x: T, y: K)
10.     where T: Clone,
11.           K: Clone + Debug
12. {
13.     x.clone();
14.     y.clone();
15.     println!("{:?}", y);
```

```
16. }
```

其中第二个例子使用了更灵活的 `where` 从句，它还允许限定的左侧可以是任意类型，
而不仅仅是类型参数。

定义在特性中的方法称为默认方法 (default method)，可以被该特性的实现覆盖。
此外，特性之间也可以存在继承 (inheritance):

```
1.  trait Foo {
2.      fn foo(&self);
3.
4.      // default method
5.      fn bar(&self) { println!("We called bar."); }
6.  }
7.
8.  // inheritance
9.  trait FooBar: Foo {
10.     fn foobar(&self);
11. }
12.
13. struct Baz;
14.
15. impl Foo for Baz {
16.     fn foo(&self) { println!("foo"); }
17. }
18.
19. impl FooBar for Baz {
20.     fn foobar(&self) { println!("foobar"); }
21. }
```

如果两个不同特性的方法具有相同的名称，可以使用通用函数调用语法 (universal function call syntax):

```
1.  // short-hand form
```

```
2.  Trait::method(args);
3.
4.  // expanded form
5.  <Type as Trait>::method(args);
```

关于实现特性的几条限制：

- 如果一个特性不在当前作用域内，它就不能被实现。
- 不管是特性还是`impl`，都只能在当前的包装箱内起作用。
- 带有特性约束的泛型函数使用单态（monomorphization），所以它是静态派分的（statically dispatched）。

下面列举几个非常有用的标准库特性：

- `Drop`提供了当一个值退出作用域后执行代码的功能，它只有一个`drop(&mut self)`方法。
- `Borrow`用于创建一个数据结构时把拥有和借用的值看作等同。
- `AsRef`用于在泛型中把一个值转换为引用。
- `Deref<Target=T>`用于把`&U`类型的值自动转换为`&T`类型。
- `Iterator`用于在集合（collection）和惰性值生成器（lazy value generator）上实现迭代器。
- `Sized`用于标记运行时长度固定的类型，而不定长的切片和特性必须放在指针后面使其运行时长度已知，
  比如`&[T]`和`Box<Trait>`。

元编程

泛型（generics）在类型理论中称作参数多态（parametric polymorphism），
意为对于给定参数可以有多种形式的函数或类型。先看Rust中的一个泛型例子：

```
enum Option<T> {
    Some(T),
    None,
}

let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

其中 `<T>` 部分表明它是一个泛型数据类型。当然，泛型参数也可以用于函数参数和结构体域：

```
// generic functions
fn make_pair<T, U>(a: T, b: U) -> (T, U) {
    (a, b)
}
let couple = make_pair("man", "female");

// generic structs
struct Point<T> {
    x: T,
    y: T,
}
let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

对于多态函数，存在两种派分（dispatch）机制：静态派分和动态派分。
前者类似于C++的模板，Rust会生成适用于指定类型的特殊函数，然后在被调用的位置进行替换，
好处是允许函数被内联调用，运行比较快，但是会导致代码膨胀（code bloat）；
后者类似于Java或Go的 `interface`，Rust通过引入特性对象（trait object）来实现，
在运行期查找虚表（vtable）来选择执行的方法。特性对象 `&Foo` 具

有和特性 `Foo` 相同的名称，
通过转换（casting）或者强制多态化（coercing）一个指向具体类型的指针来创建。

当然，特性也可以接受泛型参数。但是，往往更好的处理方式是使用关联类型（associated type）：

```
1.  // use generic parameters
2.  trait Graph<N, E> {
3.      fn has_edge(&self, &N, &N) -> bool;
4.      fn edges(&self, &N) -> Vec<E>;
5.  }
6.
7.  fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) ->
    u32 {
8.
9.  }
10.
11. // use associated types
12. trait Graph {
13.     type N;
14.     type E;
15.
16.     fn has_edge(&self, &Self::N, &Self::N) -> bool;
17.     fn edges(&self, &Self::N) -> Vec<Self::E>;
18. }
19.
20. fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> uint
    {
21.
22. }
23.
24. struct Node;
25.
26. struct Edge;
27.
28. struct SimpleGraph;
```

```
29.
30. impl Graph for SimpleGraph {
31.     type N = Node;
32.     type E = Edge;
33.
34.     fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
35.
36.     }
37.
38.     fn edges(&self, n: &Node) -> Vec<Edge> {
39.
40.     }
41. }
42.
43. let graph = SimpleGraph;
44. let object = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

Rust中的宏（macro）允许我们在语法级别上进行抽象。先来看 `vec!` 宏的实现：

```
1.  macro_rules! vec {
2.      ( $( $x:expr ),* ) => {
3.          {
4.              let mut temp_vec = Vec::new();
5.              $(
6.                  temp_vec.push($x);
7.              )*
8.              temp_vec
9.          }
10.     };
11. }
```

其中 `=>` 左边的 `$x:expr` 模式是一个匹配器（matcher），`$x` 是元变量（metavariable），`expr` 是片段指定符（fragment specifier）。匹配器写在 `$(...)` 中，

`*` 会匹配0个或多个表达式，表达式之间的分隔符为逗号。
`=>` 右边的外层大括号只是用来界定整个右侧结构的，也可以使用 `()` 或者 `[]` ，
左边的外层小括号也类似。扩展中的重复与匹配器中的重复会同步进行：
每个匹配的 `$x` 都会在宏扩展中产生一个单独的 `push` 语句。

并发计算

Rust提供了两个特性来处理并发(concurrency)： `Send` 和 `Sync` 。
当一个 `T` 类型实现了 `Send` ，就表明该类型的所有权可以在进程间安全地转移；
而实现了 `Sync` 就表明该类型在多线程并发时能够确保内存安全。

Rust的标准库 `std::thread` 提供了并行执行代码的功能：

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        "Hello from a thread!"
    });

    println!("{}", handle.join().unwrap());
}
```

其中 `thread::spawn()` 方法接受一个闭包，它将在一个新线程中执行。

Rust尝试解决可变状态的共享问题，通过所有权系统来帮助排除数据竞争(data race)：

```
use std::sync::{Arc, Mutex};
```

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let data = Arc::new(Mutex::new(0u32));

    // Creates a shared channel that can be sent along from many
threads
    // where tx is the sending half (tx for transmission),
    // and rx is the receiving half (rx for receiving).
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += i;

            tx.send(*data).unwrap();
        });
    }

    for _ in 0..10 {
        println!("{}", rx.recv().unwrap());
    }
}
```

其中 `Arc<T>` 类型是一个原子引用计数指针 (atomic reference counted pointer)，
实现了 `Sync` ，可以安全地跨线程共享。 `Mutex<T>` 类型提供了互斥锁 (mutex’s lock)，
同一时间只允许一个线程能修改它的值。 `mpsc::channel()` 方法创建了一个通道 (channel)，
来发送任何实现了 `Send` 的数据。 `Arc<T>` 的 `clone()` 方法用来增加

引用计数，
而当离开作用域时计数减少。

高级主题

高级主题

外部函数接口

在Rust中，通过外部函数接口 (foreign function interface) 可以直接调用C语言库：

```
1.  extern crate libc;
2.  use libc::size_t;
3.
4.  #[link(name = "snappy")]
5.  extern {
6.      fn snappy_max_compressed_length(source_length: size_t) ->
    size_t;
7.  }
8.
9.  fn main() {
10.     let x = unsafe { snappy_max_compressed_length(100) };
11.     println!("max compressed length of a 100 byte buffer: {}", x);
12. }
```

其中 `#[link]` 属性用来指示链接器链接 `snappy` 库， `extern` 块是外部库函数标记的列表。
外部函数被假定为不安全的，所以调用它们需要包装在 `unsafe` 块中。

当然，我们也可以把Rust代码编译为动态库来供其它语言调用：

```
1.  // in embed.rs
2.  use std::thread;
```

```
3.
4.  #[no_mangle]
5.  pub extern "C" fn process() {
6.      let handles: Vec<_> = (0..10).map(|_| {
7.          thread::spawn(|| {
8.              let mut _x = 0;
9.              for _ in (0..5_000_001) {
10.                 _x += 1
11.             }
12.         })
13.     }).collect();
14.
15.     for h in handles {
16.         h.join().ok().expect("Could not join a thread!");
17.     }
18. }
```

其中 `#[no_mangle]` 属性用来关闭Rust的命名改编，
默认的应用二进制接口 (application binary interface)
`"C"` 可以省略。
然后使用 `rustc` 进行编译：

```
1.  $ rustc embed.rs -O --crate-type=dylib
```

这将生成一个动态库 `libembed.so` 。在Node.js中调用该函数，需要安装 `ffi` 库：

```
1.  $ npm install ffi
```

```
1.  // in test.js
2.  var ffi = require('ffi');
3.
4.  var lib = ffi.Library('libembed', {
5.    'process': [ 'void', [] ]
6.  });
```

```
7.
8.  lib.process();
9.
10. console.log("done!");
```

其中 `ffi.Library()` 用来加载动态库 `libembed.so` ，
它需要标明外部函数的返回值类型和参数类型（空数组表明没有参数）。

推荐阅读

推荐阅读

- (2015-10-14) Daniel Keep: The Little Book of Rust Macros
- (2015-09-18) Aaron Turon: Specialize to reuse
- (2015-08-01) Aaron Turon, Niko Matsakis: Rust Camp Keynote
- (2015-07-31) Jessy Pelletier: Visualizing Rust’s type-system
- (2015-07-30) Llogiq: Rust’s Built-in Traits, the When, How & Why
- (2015-06-22) Herman J. Radtke III: Effectively Using Iterators In Rust
- (2015-05-17) Alexis Beingessner: Learning Rust With Entirely Too Many Linked Lists
- (2015-05-08) Huon Wilson: Finding Closure in Rust