目录

Rust By Example 注解

导读

本书面向的读者，最好有C/C++基础，了解现代C++的各种概念。如果没有C/C++基础也没有关系，Rust远没有C/C++那么复杂。

重点章节：14，15，16 这个是Rust语言的特色部分，也是难点部分，必须要掌握，否则在阅读源码或者使用库的时候，会产生很多疑惑。

我尽力用最简单的语言将原作者要表达的语义表达清晰、准确。但我在注解的过程中，也发现，有些地方，我怎么解释也讲不清楚。这可能本身就是语言的复杂信。英文的表达比较简练，加上必须对Rust有深入的了解，才能有更好的注解。虽然作者对现代C++语言知识比较有自信，但是我也深知，对语言的真正理解来自工程实践，对Rust的了解可能存在盲点，有些注解不够完善，还请读者指正，我会虚心接受改善，为初学者提供更好的入门教材。

目标

提供快速、准确、简单的入门方式，快速建立完整的Rust知识体系。

价值

基础知识，但并不代表他不重要，高级的功能都依赖与我们对基础知识的准确把握和深入理解。

学习一门计算机语言，我认为不是在编写过程中，有了问题去百度、google，最好的方式是在下手coding前，花时间把官网的教程学习一遍，以便减少犯低级错误。

官网的文档有如下的优点：

1. 写这些文档的人都是大牛或者语言本身的创造者，他们对语言的特性的阐述和表达准确性是最高的。
2. 文档经过多次的修改和完善，去除了错误的、过时的信息，避免误导读者。
3. 知识体系完整，上下文知识关联，对于复杂的、需要静一步完善的知识，给出权威的参考链接。
4. 讲解与代码相结合，完整的表达了作者意图，通过错误例子编译，展示出错信息，加强了读者对知识点的理解。

缺点：

1. 文档大多数是英文，学习陡峭，初学者对语言的理解比较费脑。
2. 初学者即使理解字面意思了，缺乏了解Rust领域专业知识。
3. 学习挫败感强，效率低，容易放弃，或者三天打鱼，两天晒网，无法持续学习。

本注释版的价值：

1. 简单、准确，完全参考官网材料1:1编写；
2. 完整，全面覆盖Rust的重要特性和知识；
3. 及时迭代更新，对错误的、过时的内容和新增的内容及时更新发布新版本；
4. 初学者不需要搭建复杂的开发环境，可以直接在官网网页的playground上面运行，复杂的地方参考本书注释，通过代码和注解，快速掌握知识。
5. 降低挫败感，持续学习，提升学习效率和体验。

0.Why Rust?

在学Rust之前，需要搞清楚，为什么要用Rust?以下几点值得考虑：

1. 如果你做前端开发的，想了解后端服务器程序的开发；
2. 如果做Java开发很多年了，想了解系统底层架构，为架构师打基础；
3. 如果是学系统编程，如Linux系统编程和网络编程，Windows API，嵌入式编程
4. 涉及到多线程，大并发领域
5. 嵌入式

以上领域，Rust作为新兴的一门语言，是具有重要的学习价值的。

Rust类似于C++ ,是一门面向系统编程的语言，需要程序员对资源把控有一定的能力，其中语言特性主要参考了C++的新特性，比如Traits类似于C++ 20中的Concepts ，智能指针参考了RAII（Resource Acquisition Is Initialization) .

Rust在嵌入式，系统编程，跨平台方面有出色的表现，解决了C++的一些问题，尤其是在安全方面。微软，华为，google等大公司已经考虑将代码从C++移植到rust。

Rust解决了C++带来的一些问题，是C++的强有力的挑战者，尤其是在安全性方面。

其他重要的特性，读者可以网上搜索一下，这里不再废话。

1.Hello World

1.1 Comments(注释)

Any program requires comments, and Rust supports a few different varieties:

- Regular comments

  which are ignored by the compiler:

  - `// Line comments which go to the end of the line.(行注释)`
  - `/* Block comments which go to the closing delimiter. */(块注释)`
- Doc comments which are parsed into HTML library documentation:

  - `/// Generate library docs for the following item.` (库文档)
  - `//! Generate library docs for the enclosing item.(库文档)`

1.2 Formatted print(格式化输出)

- `format!`: write formatted text to `String`
- `print!`: same as `format!` but the text is printed to the console (io::stdout).(输出到控制台stdout)
- `println!`: same as `print!` but a newline is appended.(和print！一样，但是会换行)
- `eprint!`: same as `print!` but the text is printed to the standard error (io::stderr).(和print!一样，但是会输出到stderr)
- `eprintln!`: same as `eprint!` but a newline is appended.(和eprint!一样，但是会换行)

```
fn main() {
    // In general, the `{}` will be automatically replaced with any
    // arguments. These will be stringified.
    println!("{} days", 31);

    // Positional arguments can be used. Specifying an integer inside `{}`
    // determines which additional argument will be replaced. Arguments start
    // at 0 immediately after the format string
    println!("{0}, this is {1}. {1}, this is {0}", "Alice", "Bob");

    // As can named arguments.
    println!("{subject} {verb} {object}",
             object="the lazy dog",
             subject="the quick brown fox",
             verb="jumps over");

    // Different formatting can invoked by specified format character after a
    // `:`.
    println!("Base 10 repr:               {}",   69420);
    println!("Base 2 (binary) repr:       {:b}", 69420);
    println!("Base 8 (octal) repr:        {:o}", 69420);
    println!("Base 16 (hexadecimal) repr: {:x}", 69420);
    println!("Base 16 (hexadecimal) repr: {:X}", 69420);

    // You can right-align text with a specified width. This will output
    // "     1". 5 white spaces and a "1".
```

```
    println!("{number:>5}", number=1);

    // You can pad numbers with extra zeroes. This will output "000001".
    println!("{number:0>5}", number=1);

    // You can use named arguments in the format specifier by appending a `$`
    println!("{number:0>width$}", number=1, width=5);


    // Rust even checks to make sure the correct number of arguments are
    // used.
    // println!("My name is {0}, {1} {0}", "Bond");
    // FIXME ^ Add the missing argument: "James"

    // Only types that implement fmt::Display can be formatted with `{}`. User-
    // defined types to not implement fmt::Display by default

    #[allow(dead_code)]
    struct Structure(i32);

    // This will not compile because `Structure` does not implement
    // fmt::Display
    // println!("This struct `{}` won't print...", Structure(3));
    // FIXME ^ Comment out this line.

    // For Rust 1.58 and above, you can directly capture the argument from
    // surrounding variable. Just like the above, this will output
    // "     1". 5 white spaces and a "1".
    let number: f64 = 1.0;
    let width: usize = 6;
    println!("{number:>width$}");
}
```

运行结果：

```
31 days
Alice, this is Bob. Bob, this is Alice
the quick brown fox jumps over the lazy dog
Base 10 repr:               69420
Base 2 (binary) repr:       10000111100101100
Base 8 (octal) repr:        207454
Base 16 (hexadecimal) repr: 10f2c
Base 16 (hexadecimal) repr: 10F2C
    1
00001
00001
     1
```

1.2.1 Debug

所有的类型需要实现std::fmt 格式化traits 才能打印，只有部分std库中自动实现了，其他的必须要手动实现

fmt::Debug trait 非常直接，所有的类型可以derive (自动创建)fmt::Debug的实现。但对fmt::Display来说还是要手动实现。

```
#![allow(unused)]
fn main() {
// This structure cannot be printed either with `fmt::Display` or
// with `fmt::Debug`.
struct UnPrintable(i32);

// The `derive` attribute automatically creates the implementation
// required to make this `struct` printable with `fmt::Debug`.
#[derive(Debug)]
struct DebugPrintable(i32);
}
```

所有标准库类型,可以通过{:?}自动打印:

```
// Derive the `fmt::Debug` implementation for `Structure`. `Structure`
// is a structure which contains a single `i32`.
#[derive(Debug)]
struct Structure(i32);

// Put a `Structure` inside of the structure `Deep`. Make it printable
// also.
#[derive(Debug)]
struct Deep(Structure); //嵌套structure

fn main() {
    // Printing with `{:?}` is similar to with `{}`.
    println!("{:?} months in a year.", 12);
    println!("{1:?} {0:?} is the {actor:?} name.",
             "Slater",
             "Christian",
             actor="actor's");

    // `Structure` is printable!
    println!("Now {:?} will print!", Structure(3));

    // The problem with `derive` is there is no control over how
    // the results look. What if I want this to just show a `7`?
    // 'derive'的问题是，我们没有办法直接显示 “7”？
    println!("Now {:?} will print!", Deep(Structure(7)));
}
```

所以fmt::Debug确实让对象具备可以打印性，但是牺牲一些优雅性。

Rust还通过”{:# ?}“提供了优雅打印（pretty printing）。

例如:

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

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

    // Pretty print
    println!("{:#?}", peter);
}
```

运行结果：

```
Person {
    name: "Peter",
    age: 27,
}
```

另外一个方法是，可以手动实现fmt::Display来控制显示。

1.2.2 Display

虽然使用fmt::Debug看起来已经非常简练了，但是未来更好的满足我们格式的需求，需要手动实现fmt::Display，它通过使用{}来打印。

例如：

```
#![allow(unused)]
fn main() {
// Import (via `use`) the `fmt` module to make it available.
use std::fmt;

// Define a structure for which `fmt::Display` will be implemented. This is
// a tuple struct named `Structure` that contains an `i32`.
struct Structure(i32);

// To use the `{}` marker, the trait `fmt::Display` must be implemented
// manually for the type.
impl fmt::Display for Structure {
    // This trait requires `fmt` with this exact signature.
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // Write strictly the first element into the supplied output
        // stream: `f`. Returns `fmt::Result` which indicates whether the
        // operation succeeded or failed. Note that `write!` uses syntax which
        // is very similar to `println!`.
        write!(f, "value = {value}", value=self.0)
    }
}
let s = Structure(10);
println!("{}",s);
}
```

运行结果：

```
value = 10
```

TODO:需要完善

1.2.3 Formatting

2.Primitives 原语

Rust 提供了对各种原语(Primitives)的访问,包括以下:

Scalar Types（标量）

- signed integers: `i8`, `i16`, `i32`, `i64`, `i128` and `isize` (pointer size)
- unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128` and `usize` (pointer size)
- floating point: `f32`, `f64`
- `char` Unicode scalar values like `'a'`, `'α'` and `'∞'` (4 bytes each)
- `bool` either `true` or `false`
- and the unit type `()`, whose only possible value is an empty tuple: `()`

Compound Types（复合类型）

- arrays like `[1, 2, 3]`
- tuples like `(1, true)`

```
fn main() {
    // Variables can be type annotated.
    let logical: bool = true;

    // 在变量后面：指定类型或者在值后面指定类型；
    let a_float: f64 = 1.0;  // Regular annotation
    let an_integer   = 5i32; // Suffix annotation


    // 或者 由编译器推导出 默认类型
    // Or a default will be used.
    let default_float   = 3.0; // `f64`
    let default_integer = 7;   // `i32`

    // 类型也可以从上下文中推导出来
    // A type can also be inferred from context
    let mut inferred_type = 12; // Type i64 is inferred from another line
    inferred_type = 4294967296i64;

    //mutable变量的值可以修改，但是类型不能修改。

    let mut mutable = 12; // Mutable `i32`
    mutable = 21;
```

```
    // Error! The type of a variable can't be changed.
    // mutable = true;

    //变量可以覆写通过shadowning系统，及重复声明变量，会覆盖上一个变量；
    // Variables can be overwritten with shadowing.
    let mutable = true;
}
```

2.1 Literals and Operators

字面量和操作符

- 整数可以分别使用这些前缀:0 x, o或0 b来表示十六进制表示,八进制或二进制符号
- 下划线_*可以插入数字字面值来改善可读性,例如*1_000等于1000,0.000001和0.000 _001是一样的。

```
fn main() {
    // Integer addition
    println!("1 + 2 = {}", 1u32 + 2);

    // Integer subtraction
    println!("1 - 2 = {}", 1i32 - 2);
    // TODO ^ Try changing `1i32` to `1u32` to see why the type is important
    // 1u32 -2 ,将会导致overflow


    // Short-circuiting boolean logic
    //逻辑运算符
    println!("true AND false is {}", true && false);
    println!("true OR false is {}", true || false);
    println!("NOT true is {}", !true);

    // Bitwise operations
    // 位操作符
    //使用后置字面量: 0x, 0o or 0b.可以修饰整形值；

    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);

    // Use underscores to improve readability!
    println!("One million is written as {}", 1_000_000u32);
}
```

2.2 Tuples

元组是**不同类型值**的集合。元组使用括号()构造，每个元组本身是一个具有类型签名(T1，T2，...)的值，其中 T1，T2是其成员的类型。函数可以使用元组返回多个值，因为元组可以保存**任意数量**的值。

```
// Tuples can be used as function arguments and as return values
//Tuples可以用作函数参数和返回值

fn reverse(pair: (i32, bool)) -> (bool, i32) {
    // `let` can be used to bind the members of a tuple to variables
    let (integer, boolean) = pair;

    (boolean, integer)
}

// The following struct is for the activity.
#[derive(Debug)]
struct Matrix(f32, f32, f32, f32);

fn main() {
    // A tuple with a bunch of different types
    //Tuple可以包含不同类型的值；
    let long_tuple = (1u8, 2u16, 3u32, 4u64,
                      -1i8, -2i16, -3i32, -4i64,
                      0.1f32, 0.2f64,
                      'a', true);

    // Values can be extracted from the tuple using tuple indexing
    // 值可以使用下表从Tuple从提取；
    println!("long tuple first value: {}", long_tuple.0);
    println!("long tuple second value: {}", long_tuple.1);

    // Tuples can be tuple members
    // Tuples本身也可以作为Tuple的成员
    let tuple_of_tuples = ((1u8, 2u16, 2u32), (4u64, -1i8), -2i16);

    // Tuples are printable
    // Tuples是可以打印的；
    println!("tuple of tuples: {:?}", tuple_of_tuples);

    //但是超过12元素的Tuples是无法打印的。
    // But long Tuples (more than 12 elements) cannot be printed
    // let too_long_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13);
    // println!("too long tuple: {:?}", too_long_tuple);
    // TODO ^ Uncomment the above 2 lines to see the compiler error

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

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

    // To create one element tuples, the comma is required to tell them apart
    // from a literal surrounded by parentheses
    println!("one element tuple: {:?}", (5u32,));
    println!("just an integer: {:?}", (5u32));
```

```
    //tuples can be destructured to create bindings
    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);

}
```

2.3 Array and Slices

Array 是存储在**连续内存**中的**同一**类型 T 的对象的集合。数组是使用方括号[]创建的，它们的长度(在编译时已知)是它们的类型签名[T; length]的一部分。

Slices类似于数组,但是他们的长度是在编译时不知道。相反,Slices是由两个词对象,第一个词是一个指向数据的指针,第二个单词是切片的长度。这个词usize一样大小,由处理器体系结构如64位x86-64。Slices可以用来借一段数组,指定为& [T]类型。

```
use std::mem;

// This function borrows a slice
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // Fixed-size array (type signature is superfluous)
    //后面的类型声明是多余的，可以省略；
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // All elements can be initialized to the same value
    // 所有的元素可以初始化为同一个值；
    let ys: [i32; 500] = [0; 500];

    // Indexing starts at 0
    //索引从0开始
    println!("first element of the array: {}", xs[0]);
    println!("second element of the array: {}", xs[1]);

    // `len` returns the count of elements in the array
    // len方法返回数组的长度；
    println!("number of elements in array: {}", xs.len());

    // Arrays are stack allocated
    // 数组是栈上分配的；
    println!("array occupies {} bytes", mem::size_of_val(&xs));
```

```
    //数组可以自动作为Slice借用：
    // Arrays can be automatically borrowed as slices
    println!("borrow the whole array as a slice");
    analyze_slice(&xs);

    // Slices can point to a section of an array
    // They are of the form [starting_index..ending_index]
    // starting_index is the first position in the slice
    // ending_index is one more than the last position in the slice
    //Slice可以指向数组的一块区域，它们从[starting_index..ending_index]
    // starting_index是第一个位置的元素，ending_index是最后一个元素下标+1，
    //例如：
    /*
    fn main(){
    let a = [1,2,3,4,8];
    let s = &a;
    let s1 = &s[0..5];//8的下标是4，下标是从0开始。
    println!("{:?}",s1);
    }

    运行结果：
    [1, 2, 3, 4, 8]

    */


    println!("borrow a section of the array as a slice");
    analyze_slice(&ys[1 .. 4]);

    // Example of empty slice `&[]`
    // 空Slice
    let empty_array: [u32; 0] = [];
    assert_eq!(&empty_array, &[]);
    assert_eq!(&empty_array, &[][..]); // same but more verbose

    // Arrays can be safely accessed using `.get`, which returns an
    // `Option`. This can be matched as shown below, or used with
    // `.expect()` if you would like the program to exit with a nice
    // message instead of happily continue.
    //Array（数组）可以通过使用.get方法安全的访问，get方法返回一个Option，它可以通过如下的
match方式获取。
    //如果你希望程序给出一个提示信息后退出，可以使用.expect().
    //例如 ：  println!("{}",a.get(5).expect("Arrar index out of range."));

    for i in 0..xs.len() + 1 { // OOPS, one element too far
        match xs.get(i) {
            Some(xval) => println!("{}: {}", i, xval),
            None => println!("Slow down! {} is too far!", i),
        }
    }

    //访问越界导致编译错误!
    // Out of bound indexing causes compile error
    //println!("{}", xs[5]);
```

```
}
```

3.自定义类型

rust主要是通过以下两个关键词实现自定义数据类型的:

- `struct`: define a structure
- `enum`: define an enumeration

常量也可以通过const和static关键字创建;

3.1 Structures

可以使用 struct 关键字创建三种类型的结构(“ structs”) :

- Tuple structs
- The classic C structs C风格的struct
- Unit structs, which are field-less, are useful for generics.
- Unit struct 没有成员字段，对泛型编程很有用

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

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

// A unit struct
struct Unit;

// A tuple struct
struct Pair(i32, f32);

// A struct with two fields
struct Point {
    x: f32,
    y: f32,
}

// Structs can be reused as fields of another struct
// 结构体可以作为另外一个结构体的字段复用;

struct Rectangle {
    // A rectangle can be specified by where the top left and bottom right
    // corners are in space.
    top_left: Point,
    bottom_right: Point,
}

fn main() {
    // Create struct with field init shorthand
```

```
    let name = String::from("Peter");
    let age = 27;
    let peter = Person { name, age };

    // Print debug struct
    println!("{:?}", peter);

    // Instantiate a `Point`
    let point: Point = Point { x: 10.3, y: 0.4 };

    // Access the fields of the point
    println!("point coordinates: ({}, {})", point.x, point.y);

    // Make a new point by using struct update syntax to use the fields of our
    // other one
    let bottom_right = Point { x: 5.2, ..point };

    // `bottom_right.y` will be the same as `point.y` because we used that field
    // from `point`
    println!("second point: ({}, {})", bottom_right.x, bottom_right.y);

    // Destructure the point using a `let` binding
    // 使用let binding 解构,不会发生所有权转移。
    // 赋值会发生所有权转移 ，如 let p2 = point; 之后point将不可以再用了。


    let Point { x: left_edge, y: top_edge } = point;

    let _rectangle = Rectangle {
        // struct instantiation is an expression too
        top_left: Point { x: left_edge, y: top_edge },
        bottom_right: bottom_right,
    };

    // Instantiate a unit struct
    let _unit = Unit;

    // Instantiate a tuple struct
    let pair = Pair(1, 0.1);

    // Access the fields of a tuple struct
    println!("pair contains {:?} and {:?}", pair.0, pair.1);

    // Destructure a tuple struct
    let Pair(integer, decimal) = pair;

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

3.2 Enums

The `enum` keyword allows the creation of a type which may be one of a few different variants. Any variant which is valid as a `struct` is also valid as an `enum`.

```
// Create an `enum` to classify a web event. Note how both
// names and type information together specify the variant:
// `PageLoad != PageUnload` and `KeyPress(char) != Paste(String)`.
// Each is different and independent.

enum WebEvent {
    // An `enum` may either be `unit-like`,
    PageLoad,
    PageUnload,
    // like tuple structs,
    KeyPress(char),
    Paste(String),
    // or c-like structures.
    Click { x: i64, y: i64 },
}

// A function which takes a `WebEvent` enum as an argument and
// returns nothing.

fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad => println!("page loaded"),
        WebEvent::PageUnload => println!("page unloaded"),
        // Destructure `c` from inside the `enum`.
        WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
        WebEvent::Paste(s) => println!("pasted \"{}\".", s),
        // Destructure `Click` into `x` and `y`.
        WebEvent::Click { x, y } => {
            println!("clicked at x={}, y={}.", x, y);
        },
    }
}

fn main() {
    let pressed = WebEvent::KeyPress('x');
    // `to_owned()` creates an owned `String` from a string slice.
    let pasted  = WebEvent::Paste("my text".to_owned());
    let click   = WebEvent::Click { x: 20, y: 80 };
    let load    = WebEvent::PageLoad;
    let unload  = WebEvent::PageUnload;

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

运行结果：

```
pressed 'x'.
```

```
pasted "my text".
clicked at x=20, y=80.
page loaded
page unloaded
```

Type aliases(类型别名)

如果你使用一个类型别名,您可以通过别名 引用 每个枚举变量 。

这可能是有用的,如果枚举的名字太长或太一般,你想重命名它。

例如:

```
enum VeryVerboseEnumOfThingsToDoWithNumbers {
    Add,
    Subtract,
}

// Creates a type alias
type Operations = VeryVerboseEnumOfThingsToDoWithNumbers;

fn main() {
    // We can refer to each variant via its alias, not its long and inconvenient
    // name.
    let x = Operations::Add;
}
```

```
enum VeryVerboseEnumOfThingsToDoWithNumbers {
    Add,
    Subtract,
}


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

3.2.1 use

使用 use 声明可以自动导入enum 定义的变量 [enum生命的究竟是变量，还是值？？]

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

enum Status {
    Rich,
    Poor,
}

enum Work {
    Civilian,
    Soldier,
}

fn main() {
    // Explicitly `use` each name so they are available without
    // manual scoping.
    use crate::Status::{Poor, Rich};
    // Automatically `use` each name inside `Work`.
    use crate::Work::*;

    // Equivalent to `Status::Poor`.
    let status = Poor;
    // Equivalent to `Work::Civilian`.
    let work = Civilian;

    match status {
        // Note the lack of scoping because of the explicit `use` above.
        Rich => println!("The rich have lots of money!"),
        Poor => println!("The poor have no money..."),
    }

    match work {
        // Note again the lack of scoping.
        Civilian => println!("Civilians work!"),
        Soldier  => println!("Soldiers fight!"),
    }
}
```

3.2.2 C-like

enum 可以像C语言中的enum使用。

例如：

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]
```

```
// enum with implicit discriminator (starts at 0)
enum Number {
    Zero,
    One,
    Two,
}

// enum with explicit discriminator
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // `enums` can be cast as integers.
    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);
}
```

运行结果：

```
zero is 0
one is 1
roses are #ff0000
violets are #0000ff
```

3.2.3 TestCase:linked-list 链表

A common way to implement a linked-list is via `enums`:

最普通的通过enum方法实现一个链表的例子：

```
use crate::List::*;

enum List {
    // Cons: Tuple struct that wraps an element and a pointer to the next node
    Cons(u32, Box<List>),
    // Nil: A node that signifies the end of the linked list
    Nil,
}

// Methods can be attached to an enum
impl List {
    // Create an empty list
    fn new() -> List {
        // `Nil` has type `List`
        Nil
    }
```

```
    // Consume a list, and return the same list with a new element at its front
    fn prepend(self, elem: u32) -> List {
        // `Cons` also has type List
        Cons(elem, Box::new(self))
    }

    // Return the length of the list
    //TODO:???
    fn len(&self) -> u32 {
        // `self` has to be matched, because the behavior of this method
        // depends on the variant of `self`
        // `self` has type `&List`, and `*self` has type `List`, matching on a
        // concrete type `T` is preferred over a match on a reference `&T`
        // after Rust 2018 you can use self here and tail (with no ref) below as
well,
        // rust will infer &s and ref tail.
        // See https://doc.rust-lang.org/edition-guide/rust-2018/ownership-and-
lifetimes/default-match-bindings.html
        match *self {
            // Can't take ownership of the tail, because `self` is borrowed;
            // instead take a reference to the tail
            Cons(_, ref tail) => 1 + tail.len(),
            // Base Case: An empty list has zero length
            Nil => 0
        }
    }

    // Return representation of the list as a (heap allocated) string
    fn stringify(&self) -> String {
        match *self {
            Cons(head, ref tail) => {
                // `format!` is similar to `print!`, but returns a heap
                // allocated string instead of printing to the console
                format!("{}, {}", head, tail.stringify())
            },
            Nil => {
                format!("Nil")
            },
        }
    }
}

fn main() {
    // Create an empty linked list
    let mut list = List::new();

    // Prepend some elements
    list = list.prepend(1);
    list = list.prepend(2);
    list = list.prepend(3);

    // Show the final state of the list
    println!("linked list has length: {}", list.len());
    println!("{}", list.stringify());
```

```
}
```

3.3 constants

- `const`: 不可改变的值 (the common case).
- `static`: A possibly `mut`able variable with `'static` lifetime. 访问static 修饰的变量是unsafe的，需要包含在unsafe语句块内.

```
// Globals are declared outside all other scopes.

const THRESHOLD: i32 = 10;
static mut LANGUAGE: &str = "Rust"; //这里是mut

fn is_big(n: i32) -> bool {
    // Access constant in some function
    n > THRESHOLD
}

fn main() {
    let n = 16;
    // 访问和修改static变量必须要使用unsafe语句块。
    unsafe {
        LANGUAGE = "Cpp";
        println!("This is {}", LANGUAGE);
    }


    println!("The threshold is {}", THRESHOLD);
    println!("{} is {}", n, if is_big(n) { "big" } else { "small" });

    // Error! Cannot modify a `const`.
    // THRESHOLD = 5;
    // FIXME ^ Comment out this line
}
```

4.变量绑定

Variable Bindings（变量绑定）

rust通过静态类型来提供类型安全。当声明变量时，可以绑定类型说明。

然而,在大多数情况下,编译器可以从上下文中推断出变量的类型,大大的降低了类型声明的负担。

值（类似字面量），可以通过let绑定到变量

例如:

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

```
    let unit = ();

    // copy `an_integer` into `copied_integer`
    let copied_integer = an_integer;

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

    // The compiler warns about unused variable bindings; these warnings can
    // be silenced by prefixing the variable name with an underscore
    //没有使用变量，可以在变量名前面加上下划线_;
    let _unused_variable = 3u32;

    let noisy_unused_variable = 2u32;
    // FIXME ^ Prefix with an underscore to suppress the warning
    // Please note that warnings may not be shown in a browser
}
```

4.1 Mutability(可变性)

变量的声明，默认是不可变的，但是可以使用mut 使其可变；

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

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

    // Ok
    mutable_binding += 1;

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

    // Error!
    //_immutable_binding += 1;
    // FIXME ^ Comment out this line
}
```

4.2 Scope and Shadowing

变量的有效性是有范围的,限制在{}语句块中。

```
fn main() {
    // This binding lives in the main function
    let long_lived_binding = 1;

    // 这个语句块的访问要比main函数小
    {
        //这个声明，只在这个语句块中有效
```

```
        //注意：变量超出语句块后，将自动销毁；
        let short_lived_binding = 2;

        println!("inner short: {}", short_lived_binding);
    }
    // 语句块结束

    // 错误！ `short_lived_binding` 不存在这个范围
    // println!("outer short: {}", short_lived_binding);
    // FIXME ^ Comment out this line

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

变量的shadowing（可以理解为太阳的影子）

参考以下例子，运行结果对比一下，很容易发现规律。

- shadowing的范围限制在语句块内，离开语句块后，对原始变量没有影响；
- 如果要影响原始变量，需要在同一个语句块范围重新shadowing.

```
fn main() {
    let shadowed_binding = 1;

    {
        println!("before being shadowed: {}", shadowed_binding);

        // This binding *shadows* the outer one
        let shadowed_binding = "abc";

        println!("shadowed in inner block: {}", shadowed_binding);
    }
    println!("outside inner block: {}", shadowed_binding);

    // This binding *shadows* the previous binding
    let shadowed_binding = 2;
    println!("shadowed in outer block: {}", shadowed_binding);
}

运行结果：

before being shadowed: 1
shadowed in inner block: abc
outside inner block: 1
shadowed in outer block: 2
```

4.3 Declare first（先声明，后bingding)

可以先声明变量,然后再初始化它们。

然而,这种形式是很少使用,因为它可能会导致使用未初始化的变量。

注意：最好像C++一样，声明即初始化。

```
fn main() {
    // Declare a variable binding
    let a_binding;

    {
        let x = 2;

        // Initialize the binding
        a_binding = x * x;
    }

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

    let another_binding;

    // Error! Use of uninitialized binding
    // println!("another binding: {}", another_binding);
    // FIXME ^ Comment out this line

    another_binding = 1;

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

4.4 Freezing

考虑如下情况:

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

    {
        // Shadowing by immutable `_mutable_integer`
        let _mutable_integer = _mutable_integer;

        // Error! `_mutable_integer` is frozen in this scope
        _mutable_integer = 50;
        // FIXME ^ Comment out this line

        // `_mutable_integer` goes out of scope
    }

    // Ok! `_mutable_integer` is not frozen in this scope
    _mutable_integer = 3;
}

   Compiling playground v0.0.1 (/playground)
error[E0384]: cannot assign twice to immutable variable `_mutable_integer`
 --> src/main.rs:9:9
  |
```

```
6 |         let _mutable_integer = _mutable_integer;
  |             ----------------
  |             |
  |             first assignment to `_mutable_integer`
  |             help: consider making this binding mutable: `mut
_mutable_integer`
...
9 |         _mutable_integer = 50;
  |         ^^^^^^^^^^^^^^^^^^^^^ cannot assign twice to immutable variable


error[E0384]: cannot assign twice to immutable variable `_mutable_integer`
```

当变量声明为同一个名字的、不可以修改变量，会发生freeze现象。

frozen 的 data 需要等到离开作用域才能修改；

5.类型

Rust provides several mechanisms to change or define the type of primitive and user defined types. The following sections cover:

- Casting between primitive types
- Specifying the desired type of literals
- Using type inference
- Aliasing types

5.1 Casting(转换)

Rust provides no implicit type conversion (coercion) between primitive types. But, explicit type conversion (casting) can be performed using the `as` keyword.

Rust没有对简单（primitive）类型提供隐式转换功能，但是可以通过as关键字提供显示转换；

Rules for converting between integral types follow C conventions generally, except in cases where C has undefined behavior. The behavior of all casts between integral types is well defined in Rust.

整体类型之间的转换规则遵循C语言约定，除C有UB（undefined behavior.）之外。整体的转换行为在rust都有良好的定义；

注意转换规则：

有符号数和无符号数的转换，需要考虑溢出和符号的变化，

有符号数可能会有正数变为负数。

```
// Suppress all warnings from casts which overflow.
#![allow(overflowing_literals)]
```

```rust
fn main() {
    let decimal = 65.4321_f32;

    // Error! No implicit conversion
    let integer: u8 = decimal;
    // FIXME ^ Comment out this line

    // Explicit conversion
    let integer = decimal as u8;
    let character = integer as char;

    // Error! There are limitations in conversion rules.
    // A float cannot be directly converted to a char.
    let character = decimal as char;
    // 错误信息：only `u8` can be cast as `char`, not `f32`
    // FIXME ^ Comment out this line

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

    // when casting any value to an unsigned type, T,
    // T::MAX + 1 is added or subtracted until the value
    // fits into the new type

    // 1000 already fits in a u16
    println!("1000 as a u16 is: {}", 1000 as u16);

    // 1000 - 256 - 256 - 256 = 232
    // Under the hood, the first 8 least significant bits (LSB) are kept,
    // while the rest towards the most significant bit (MSB) get truncated.
    println!("1000 as a u8 is : {}", 1000 as u8);
    // -1 + 256 = 255
    println!("  -1 as a u8 is : {}", (-1i8) as u8);

    // For positive numbers, this is the same as the modulus
    println!("1000 mod 256 is : {}", 1000 % 256);

    // When casting to a signed type, the (bitwise) result is the same as
    // first casting to the corresponding unsigned type. If the most significant
    // bit of that value is 1, then the value is negative.

    // Unless it already fits, of course.
    println!(" 128 as a i16 is: {}", 128 as i16);

    // 128 as u8 -> -128, whose two's complement in eight bits is:
    println!(" 128 as a i8 is : {}", 128 as i8);

    // repeating the example above
    // 1000 as u8 -> 232
    println!("1000 as a u8 is : {}", 1000 as u8);
    // and the two's complement of 232 is -24
    println!(" 232 as a i8 is : {}", 232 as i8);

    // Since Rust 1.45, the `as` keyword performs a *saturating cast*
    // when casting from float to int. If the floating point value exceeds
    // the upper bound or is less than the lower bound, the returned value
```

```
    // will be equal to the bound crossed.

    // 300.0 is 255
    println!("300.0 is {}", 300.0_f32 as u8);
    // -100.0 as u8 is 0
    println!("-100.0 as u8 is {}", -100.0_f32 as u8);
    // nan as u8 is 0
    println!("nan as u8 is {}", f32::NAN as u8);

    // This behavior incurs a small runtime cost and can be avoided
    // with unsafe methods, however the results might overflow and
    // return **unsound values**. Use these methods wisely:
    unsafe {
        // 300.0 is 44
        println!("300.0 is {}", 300.0_f32.to_int_unchecked::<u8>());
        // -100.0 as u8 is 156
        println!("-100.0 as u8 is {}", (-100.0_f32).to_int_unchecked::<u8>());
        // nan as u8 is 0
        println!("nan as u8 is {}", f32::NAN.to_int_unchecked::<u8>());
    }
}
```

5.2 Literals(字面量)

数字字面量类型，可以在后面添加注释。

例如,42应该有一个类型i32,写作42i32.

如果数字字面量类型，后面没有添加注释，那么它的类型将取决于如何使用它们。

如果没有约束存在,编译器将使用i32作为整数类型,f64作为浮点数类型。

例如

```
fn main() {
    // Suffixed literals, their types are known at initialization
    // 初始化即知道类型
    let x = 1u8;
    let y = 2u32;
    let z = 3f32;

    // Unsuffixed literals, their types depend on how they are used
    // 没有后缀的字面量，他们的类型由他们如何使用决定。
    let i = 1;
    let f = 1.0;

    // `size_of_val` returns the size of a variable in bytes
    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));
```

```
}
```

运行结果：

```
size of `x` in bytes: 1
size of `y` in bytes: 4
size of `z` in bytes: 4
size of `i` in bytes: 4
size of `f` in bytes: 8
```

(整型占用4个byte，浮点占用8个byte)

5.3 Inference(类型推导)

类型推断引擎很聪明。它不但看变量初始化期间值的类型，也看如何使用变量，来推断它的类型。

这是一个先进的类型推断的例子:

```
fn main() {
    // Because of the annotation, the compiler knows that `elem` has type u8.
    let elem = 5u8;

    // Create an empty vector (a growable array).
    let mut vec = Vec::new();
    // At this point the compiler doesn't know the exact type of `vec`, it
    // just knows that it's a vector of something (`Vec<_>`).

    //在这个时间点，编译器并不知道vec的确切类型，它只知道是一个`Vec<_>`的类型。

    // Insert `elem` in the vector.
    vec.push(elem);
    // Aha! Now the compiler knows that `vec` is a vector of `u8`s (`Vec<u8>`)
    // TODO ^ Try commenting out the `vec.push(elem)` line
    // 直到我们在vec中存放一个数值时，vec才知道准确的类型，是`Vec<u8>`

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

No type annotation of variables was needed, the compiler is happy and so is the programmer!

所以编译器后台为程序员做了很多事，可以让我们不需要每一个变量都需要声明类型。

有点类似C++ 11 后的auto 关键字。

5.4 Aliasing(别名)

type 声明 可以用来给现有的类型一个新名字。

类型必须遵守UpperCamelCase规则,否则编译器会提高一个警告。

这个规则的例外是基本类型:usize, f32等等。

```
// `NanoSecond`, `Inch`, and `U64` are new names for `u64`.
type NanoSecond = u64;
type Inch = u64;
type U64 = u64;

fn main() {
    // `NanoSecond` = `Inch` = `U64` = `u64`.
    let nanoseconds: NanoSecond = 5 as U64;
    let inches: Inch = 2 as U64;

    // Note that type aliases *don't* provide any extra type safety, because
    // aliases are *not* new types
    // ???
    println!("{} nanoseconds + {} inches = {} unit?",
             nanoseconds,
             inches,
             nanoseconds + inches);
}
```

别名的主要用途是为了减少boilerplate（重复使用的代码），例如

IoResult 是`Result<T, IoError>`的别名

6.Conversion 转换

简单类型可以互相转换。

Rust提供通过使用traits在自定义类型（struct,enum）间转换。一般的转换使用From 和Into traits。

6.1 From and Into

From和Into traits是内在关联的，如果可以从A转换为B，那么我们认为，也可以从B转换A。

From

From trait允许我们定义如何从其他的类型来创建自己，因此提供了一个非常简单的机制来做转换。在标准库里

有几种实现了该traits来做简单类型的转换。

例如我们可以很容易的从str 转为 String

```
#![allow(unused)]
fn main() {
let my_str = "hello";
let my_string = String::from(my_str);
}
```

我们也能为我们自定义的类型做同样的转换,例如:

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

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

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

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

Into

如果为类型实现了From trait,into()在需要的时候会自动调用From traits。

例如:

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

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

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

fn main() {
    let a = 5;
    // Try removing the type declaration
    let num: Number = a.into();
    println!("My number is {:?}", num);
}
```

6.2 TryFrom and TryInto

和 From 、Into 一样, TryFrom 、TryInto 类型转型通用的trait。

和 From、Into, 不同的是， TryFrom、TryInto traits 被用来会产生转换错误的场景，比如, 还回 Result.

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

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

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

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

fn main() {
    // TryFrom

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

    // TryInto

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

6.3 To and from Strings

Converting to String

任何类型转换为字符串只需要简单实现ToString traits,

我们一般不这样做,应该实现fmt::Display traits ， 它自动提供ToString trait.

例如:

```
use std::fmt;

struct Circle {
    radius: i32
```

```
}

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

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

Parsing a String

一种更加常见的需求是，将一个字符串转换为一个数字，惯用方法是使用解析函数parse()。

例如：

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

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

如要把字符串转换成指定的类型,只需要实现FromStr trait.

例如：

```
use std::{str::FromStr, fmt::Display};
#[derive(Debug)]
struct Money {value:i32}

impl Display for Money {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f,"output is {}",self.value)
    }
}

impl FromStr for Money  {
    type Err = i32;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        if !s.is_empty() {
            Ok(Money{value:100})
        }else {
```

```
            Err(-1)
        }
    }
}
#[allow(unused_variables)]
fn main(){
    let ss = Money{value:33};
    let ss = Money::from_str("10").unwrap();
    println!("{:?}",ss.to_string());
}
```

7.表达式

8.流控制

8.4 for and range

8.5 match

8.5.1 解构

- Dereferencing uses `*`
- Destructuring uses `&`, `ref`, and `ref mut`

```
fn main() {
    // Try changing the values in the array, or make it a slice!
    let array = [1, -2, 6];

    match array {
        // Binds the second and the third elements to the respective variables
        [0, second, third] =>
            println!("array[0] = 0, array[1] = {}, array[2] = {}", second,
third),

        // Single values can be ignored with _
        //_下划线表示忽略该元素;
        [1, _, third] => println!(
            "array[0] = 1, array[2] = {} and array[1] was ignored",
            third
        ),

        // You can also bind some and ignore the rest
        [-1, second, ..] => println!(
            "array[0] = -1, array[1] = {} and all the other ones were ignored",
            second
        ),
        // The code below would not compile
        // [-1, second] => ...

        // Or store them in another array/slice (the type depends on
        // that of the value that is being matched against)
        [3, second, tail @ ..] => println!(
```

```
            "array[0] = 3, array[1] = {} and the other elements were {:?}",
            second, tail
        ),

        // Combining these patterns, we can, for example, bind the first and
        // last values, and store the rest of them in a single array
        //@符合表示剩下的、除了first和last之外的所有元素。
        [first, middle @ .., last] => println!(
            "array[0] = {}, middle = {:?}, array[2] = {}",
            first, middle, last
        ),
    }
}
```

8.5.1.4 Pointer/ref

```
fn main() {
    let value = 5;
    let mut mut_value = 6;

    // Use `ref` keyword to create a reference.
    // 这个地方有点难理解，不用ref关键字也是同样的运行结果：
    match value {
        ref r => println!("Got a reference to a value: {:?}", r),
    }

    // Use `ref mut` similarly.
    match mut_value {
        ref mut m => {
            // Got a reference. Gotta dereference it before we can
            // add anything to it.
            // 获得m的一个可变引用，以便我们可以修改他的值：
            *m += 10;
            println!("We added 10. `mut_value`: {:?}", m);
        },
    }
}
```

8.8.1.5 structs 解构

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

    // Try changing the values in the struct to see what happens
    let foo = Foo { x: (1, 2), y: 3 };

    match foo {
        Foo { x: (2, b), y } => println!("First of x is 1, b = {},  y = {} ", b,
y),
```

```
        // you can destructure structs and rename the variables,
        // the order is not important
        Foo { y: 2, x: i } => println!("y is 2, i = {:?}", i),

        // and you can also ignore some variables:
        Foo { y, .. } => println!("y = {}, we don't care about x", y),
        // this will give an error: pattern does not mention field `x`
        //Foo { y } => println!("y = {}", y),
    }
}
```

运行结果：

```
//y = 3, we don't care about x
```

8.5.2 Guards 守卫

A `match` *guard* can be added to filter the arm.

guard用来做进一步的过滤

```
enum Temperature {
    Celsius(i32),
    Farenheit(i32),
}

fn main() {
    let temperature = Temperature::Celsius(15);
    // ^ TODO try different values for `temperature`

    match temperature {
        Temperature::Celsius(t) if t > 30 => println!("{}C is above 30 Celsius",
t),
        // The `if condition` part ^ is a guard
        // if 条件语句为guard 守卫
        Temperature::Celsius(t) => println!("{}C is below 30 Celsius", t),

        Temperature::Farenheit(t) if t > 86 => println!("{}F is above 86
Farenheit", t),
        Temperature::Farenheit(t) => println!("{}F is below 86 Farenheit", t),
    }
}
```

运行结果：

```
15C is below 30 Celsius
```

```
fn main() {
    let number: u8 = 40;

    match number {
        i if i == 0 => println!("Zero"),
        i if i > 0 => println!("Greater than zero"),
        _ => unreachable!("Should never happen."),
        // 取消这行会报错；
```

```
    }
}
```

运行结果：

```
Greater than zero
```

8.5.3 Binding

rust provides the `@` sigil for binding values to names:

rust提供了@符号，将值绑定到名字

代码：

```
// A function `age` which returns a `u32`.
fn age() -> u32 {
    100
}

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

    match age() {
        0             => println!("I haven't celebrated my first birthday yet"),
        // Could `match` 1 ..= 12 directly but then what age
        // would the child be? Instead, bind to `n` for the
        // sequence of 1 ..= 12. Now the age can be reported.
        n @ 1  ..= 12 => println!("I'm a child of age {:?}", n),
        n @ 13 ..= 19 => println!("I'm a teen of age {:?}", n),
        // Nothing bound. Return the result.
        n             => println!("I'm an old person of age {:?}", n),
    }
}
```

运行结果：

```
Tell me what type of person you are
I'm an old person of age 100
```

8.6 if let

不使用if let

```
// Make `optional` of type `Option<i32>`
let optional = Some(7);

match optional {
    Some(i) => {
        println!("This is a really long string and `{:?}`", i);
        // ^ Needed 2 indentations just so we could destructure
        // `i` from the option.
    },
    _ => {},
    // ^ Required because `match` is exhaustive. Doesn't it seem
```

```
    // like wasted space?
    //比较啰嗦，每个都要这样写，因为match需要穷尽；
};
```

改用if let后的代码：

```
fn main() {
    let number = Some(7);
    if let Some(i) = number {
        //注意：这里是赋值=，不是逻辑==
        println!("Matched {:?}!", i);
    }
}
```

将if let 融入逻辑分支语句：

```
fn main() {
    // All have type `Option<i32>`
    let number = Some(7);
    let letter: Option<i32> = None;
    let emoticon: Option<i32> = None;

    // The `if let` construct reads: "if `let` destructures `number` into
    // `Some(i)`, evaluate the block (`{}`).
    if let Some(i) = number {
        println!("Matched {:?}!", i);
    }

    // If you need to specify a failure, use an else:
    //如果需要指定匹配失败后执行的语句，可以使用else
    if let Some(i) = letter {
        println!("Matched {:?}!", i);
    } else {
        // Destructure failed. Change to the failure case.
        println!("Didn't match a number. Let's go with a letter!");
    }

    // Provide an altered failing condition.
    let i_like_letters = false;


    //if语句不符合，判断else if 也不符合，最后运行到else语句块；
    if let Some(i) = emoticon {
        println!("Matched {:?}!", i);
    // Destructure failed. Evaluate an `else if` condition to see if the
    // alternate failure branch should be taken:
    } else if i_like_letters {
        println!("Didn't match a number. Let's go with a letter!");
    } else {
        // The condition evaluated false. This branch is the default:
        println!("I don't like letters. Let's go with an emoticon :)!");
    }
```

```
}
```

注意:

```
// This enum purposely neither implements nor derives PartialEq.
// 这里不能使用 == ，因为没有 继承 PartialEq，
// That is why comparing Foo::Bar == a fails below.
enum Foo {Bar}

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

    // Variable a matches Foo::Bar
    if Foo::Bar == a {

    // ^-- this causes a compile-time error. Use `if let` instead.
        println!("a is foobar");
    }
}
```

改为如下代码，也可以正常编译:

```
#[derive(PartialEq)]
enum Foo {Bar}

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

    // Variable a matches Foo::Bar
    if Foo::Bar == a {
    // ^-- this causes a compile-time error. Use `if let` instead.
        println!("a is foobar");
    }
}
```

注意:

if let 后面可以加else /else if 语句;

8.7 while let

```
#![allow(unused)]
fn main() {
// Make `optional` of type `Option<i32>`
let mut optional = Some(0);

// Repeatedly try this test.
loop {
```

```
    match optional {
        // If `optional` destructures, evaluate the block.
        Some(i) => {
            if i > 9 {
                println!("Greater than 9, quit!");
                optional = None;
            } else {
                println!("`i` is `{:?}`. Try again.", i);
                optional = Some(i + 1);
            }
            // ^ Requires 3 indentations!
        },
        // Quit the loop when the destructure fails:
        _ => { break; }
        // ^ Why should this be required? There must be a better way!
    }
}
}
```

如果注释掉 _ => { break; }

会报如下错误:

```
match optional {
  |         ^^^^^^^^ pattern `None` not covered
```

改为while let

```
fn main() {
    let mut optional = Some(0);
    while let Some(i) = optional {
        // = 等号为赋值，不要写出==
        if i > 9 {
            println!("Greater than 9, quit!");
            optional = None;
        } else {
            println!("`i` is `{:?}`. Try again.", i);
            optional = Some(i + 1);
        }
    }
}
//注意：
. 不需要额外的None处理；
. 和if let不同，while后面不能加else/else if 语句；
```

9.函数

9.1方法

关联函数和方法

一些函数与特定的类型相关，有两种形式：关联函数和方法，关联函数通常定义在特定的类型上，方法是调用特定类型实例的关联函数。

关联函数，相当于静态方法，或者C++中用statci定义的方法，不依赖于this指针；

代码：

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

// Implementation block, all `Point` associated functions & methods go in here
// 所有Point相关的关联函数和方法都定义在这里；
impl Point {
    // This is an "associated function" because this function is associated with
    // a particular type, that is, Point.
    //这是一个关联函数，因为他与Point关联；
    // Associated functions don't need to be called with an instance.
    // These functions are generally used like constructors.
    // 关联函数调用不需要初始化一个实例，这些函数通常像构造函数一样；
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }

    // Another associated function, taking two arguments:
    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }
    //例如可以这样调用：
    // let p = Point::new(0.0,0.0);
    // 它不需要初始化一个实例，就可以调用；

}

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

impl Rectangle {
    // 这里是方法
    // &self 是self::&Self的语法糖，Self是调用者对象，这里Self=Rectange
    fn area(&self) -> f64 {
        // `self` gives access to the struct fields via the dot operator
        //self通过.点号，访问结构体的字段；
        let Point { x: x1, y: y1 } = self.p1;
        let Point { x: x2, y: y2 } = self.p2;

        // `abs` is a `f64` method that returns the absolute value of the
        // caller
        //这里的abs是调用方法，用来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())
    }

    // This method requires the caller object to be mutable
    // `&mut self` desugars to `self: &mut Self`
    // &mut self 是 self::&mut Self 的语法糖形式；

    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` owns resources: two heap allocated integers
struct Pair(Box<i32>, Box<i32>);

impl Pair {
    // This method "consumes" the resources of the caller object
    // `self` desugars to `self: Self`
    // self 是 self::Self的语法糖；

    fn destroy(self) {
        // Destructure `self`
        // 解构了self，结构体赋值，发生了生命周期转移操作，self move 给了Pair
        // self不存在了，从而实现了销毁操作语义；
        let Pair(first, second) = self;

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

        // `first` 和 `second` 超出作用范围，并且销毁掉了。
    }
}

fn main() {
    let rectangle = Rectangle {
        //关联函数调用采用::(2个冒号)符号
        p1: Point::origin(),
        p2: Point::new(3.0, 4.0),
    };


    // 方法的调用采用. (dot)（点号）
    // 注意第一个&self 为隐式传递；
    // `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),
    };


    //translate调用需要一个Mutable 对象,所有以下调用会失败
    //rectangle.translate(1.0, 0.0);

    // 现在square是一个 Mutable 对象，所有可以条用mutable方法
    square.translate(1.0, 1.0);

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

    pair.destroy();

    //destroy不能多次调用，调用一次已经销毁了，在调用一次会出错；
    // Error! Previous `destroy` call "consumed" `pair`
    //pair.destroy();
    // TODO ^ Try uncommenting this line
}
```

9.2闭包

Closures are functions that can capture the enclosing environment.

For example, a closure that captures the `x` variable:

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

闭包的特性：

- 输入、输出类型可以被推导出来，但是输入的变量名必须指定；
- 输入参数用||代替（）
- 如果是一个表达式，闭包体分隔符{}可以省略
- 可以捕获外面环境变量的值

例如：

```
fn main() {
    // Increment via closures and functions.
    fn function(i: i32) -> i32 { i + 1 }

    // Closures are anonymous, here we are binding them to references
    // Annotation is identical to function annotation but is optional
    // as are the `{}` wrapping the body. These nameless functions
    // are assigned to appropriately named variables.
    let closure_annotated = |i: i32| -> i32 { i + 1 };
    let closure_inferred  = |i     |          i + 1  ;
```

```
    let i = 1;
    // Call the function and closures.
    println!("function: {}", function(i));
    println!("closure_annotated: {}", closure_annotated(i));
    println!("closure_inferred: {}", closure_inferred(i));

    // A closure taking no arguments which returns an `i32`.
    // The return type is inferred.
    let one = || 1;
    println!("closure returning one: {}", one());

}
```

运行结果：

function: 2
closure_annotated: 2
closure_inferred: 2
closure returning one: 1

9.2.1 Capturing （捕获）

闭包可以通过如下方式捕获变量：

- 通过引用：&T
- 通过可变引用：&mut T
- 通过值：T

他们优先通过引用捕获变量，只有在需要时下降（go lower）；

例如：

```
fn main() {
    use std::mem;
    let color = String::from("green");

    // A closure to print `color` which immediately borrows (`&`) `color` and
    // stores the borrow and closure in the `print` variable. It will remain
    // borrowed until `print` is used the last time.
    //
    // `println!` only requires arguments by immutable reference so it doesn't
    // impose anything more restrictive.
    let print = || println!("`color`: {}", color);

    // Call the closure using the borrow.
    print();

    // `color` can be borrowed immutably again, because the closure only holds
    // an immutable reference to `color`.
    let _reborrow = &color;
    print();
```

```
    // A move or reborrow is allowed after the final use of `print`

    let _color_moved = color;

    //注意：color的所有权已经转移，不能再使用color了。

    let mut count = 0;
    // 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`.
    //
    // 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);
    };

    // Call the closure using a mutable borrow.
    inc();

    // The closure still mutably borrows `count` because it is called later.
    // An attempt to reborrow will lead to an error.
    // let _reborrow = &count;
    // ^ TODO: try uncommenting this line.
    inc();

    // The closure no longer needs to borrow `&mut count`. Therefore, it is
    // possible to reborrow without an error
    let _count_reborrowed = &mut count;


    // A 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.
    // A non-copy must move and so `movable` immediately moves into
    // the closure.
    let consume = || {
        println!("`movable`: {:?}", movable);
        mem::drop(movable); //drop掉moveable后，将不能再次调用consume()
    };

    // `consume` consumes the variable so this can only be called once.
    consume();
    // consume();
    // ^ TODO: Try uncommenting this line.
}
```

笔者添加：

补充：

Drop 对于实现 `Copy` 的类型实际上没有任何作用，例如整数。

此类值被复制然后移动到函数中，因此该值在此函数调用后仍然存在。

这个函数并不神奇；它的字面意思是

```
pub fn drop<T>(_x:T) { }
```

因为`_x`被移动到函数中，所以在函数返回之前它会被自动删除。

在||前使用move强制获取捕获变量的所有权（ownship）

例如：

```
fn main() {
    // `Vec` has non-copy semantics.
    let haystack = vec![1, 2, 3];

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

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

    // println!("There're {} elements in vec", haystack.len());
    // ^ Uncommenting above line will result in compile-time error
    // because borrow checker doesn't allow re-using variable after it
    // has been moved.

    // Removing `move` from closure's signature will cause closure
    // to borrow _haystack_ variable immutably, hence _haystack_ is still
    // available and uncommenting above line will not cause an error.
}
```

9.2.2 As input parameters（作为输入参数）

- FnOnce consumes the variables it captures from its enclosing scope, known as the closure's environment. To consume the captured variables, the closure must take ownership of these variables and move them into the closure when it is defined. The Once part of the name represents the fact that the closure can't take ownership of the same variables more than once, so it can be called only once.
- FnMut can change the environment because it mutably borrows values.
- Fn borrows values from the environment immutably.

- `Fn`: the closure uses the captured value by **reference** (`&T`)
- `FnMut`: the closure uses the captured value by **mutable referenc**e (`&mut T`)
- `FnOnce`: the closure uses the captured value by **value** (`T`)

下面的代码，在闭包中，有的需要Fn Trait,有的需要实现FnMut Trait,所以F：的限制，只有FnOnce满足，因为所有的参数都可以满足FnOnce。

```
// A function which takes a closure as an argument and calls it.
// <F> denotes that F is a "Generic type parameter"
fn apply<F>(f: F) where
    // The closure takes no input and returns nothing.
    F: FnOnce() {
    // ^ TODO: Try changing this to `Fn` or `FnMut`.
    // 这里改动就会报错.
    //expected a closure that implements the `Fn` trait,
    //but this closure only implements `FnOnce`

    f();
}

// A function which takes a closure and returns an `i32`.
fn apply_to_3<F>(f: F) -> i32 where
    // The closure takes an `i32` and returns an `i32`.
    F: Fn(i32) -> i32 {

    f(3)
}

fn main() {
    use std::mem;

    let greeting = "hello";
    // A non-copy type.
    // `to_owned` creates owned data from borrowed one
    let mut farewell = "goodbye".to_owned();

    // Capture 2 variables: `greeting` by reference and
    // `farewell` by value.
    let diary = || {
        // `greeting` is by reference: requires `Fn`.
        println!("I said {}.", greeting);

        // Mutation forces `farewell` to be captured by
        // mutable reference. Now requires `FnMut`.
        farewell.push_str("!!!");
        println!("Then I screamed {}.", farewell);
        println!("Now I can sleep. zzzzz");

        // Manually calling drop forces `farewell` to
        // be captured by value. Now requires `FnOnce`.
        mem::drop(farewell);
    };

    // Call the function which applies the closure.
    apply(diary);

    // `double` satisfies `apply_to_3`'s trait bound
```

```
    let double = |x| 2 * x;

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

9.2.3 Type anonymity

```
// `F` must implement `Fn` for a closure which takes no
// inputs and returns nothing - exactly what is required
// for `print`.
fn apply<F>(f: F) where
    F: Fn() {
    f();
}

fn main() {
    let x = 7;

    // Capture `x` into an anonymous type and implement
    // `Fn` for it. Store it in `print`.
    let print = || println!("{}", x);

    apply(print);
}
```

当一个闭包定义时,编译器隐式地创建一个新的匿名结构，其中存储捕获的变量,同时通过这个未知的类型实现这种功能中的一个特征:Fn, FnMut或FnOnce。

这种类型是先分配给变量存储，直到调用才能确定是哪一个。???

参考:

A thorough analysis, `Fn`, `FnMut`, and `FnOnce`

9.2.4 Input functions

Fn, FnMut和FnOnce traits 决定一个闭包捕获的变量范围。

```
// Define a function which takes a generic `F` argument
// bounded by `Fn`, and calls it
fn call_me<F: Fn()>(f: F) {
    f();
}

// Define a wrapper function satisfying the `Fn` bound
fn function() {
    println!("I'm a function!");
}
```

```
fn main() {
    // Define a closure satisfying the `Fn` bound
    let closure = || println!("I'm a closure!");

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

9.2.5 As output parameters

闭包作为输入参数是可行的,所以返回输出闭包也应该是可能的。

然而,闭包匿名类型在定义的时候是未知的,

所以我们必须使用impl trait来return。

闭包返回的有效traits是:

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

除此之外,move关键字必须使用，用来指示，所有的捕获采用By Value.

这是必需的,因为任何通过引用的捕获,在函数退出时候将会立即调用Drop()，使其离开闭包时，保证其引用无效。？？？

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

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

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

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

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

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

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

    fn_plain();
    fn_mut();
    fn_once(); //只能调用一次，为什么？
}
```