程序设计训练（Rust 语言）

第 5 讲：工程管理与其他

韩文弢

清华大学

2025–2026 学年度夏季学期

本讲内容

5.1 工程管理

5.1.1 工程管理的需求

- 随着工程（project）变大，参与者变多，组织代码的方式会越来越重要。
- 封装可以隐藏实现细节，使用户更好地复用代码。
- 项目（item）的作用域管理是需要支持的核心功能。
  - Rust 中，项目指的是模块、函数、类型、常量等。

5.1.2 模块

- Rust 代码的组织结构基于模块（module），模块组成树状结构。
- 一般来说，每个文件是一个模块，例如模块 `foo` 对应于文件 `foo.rs`。
- 模块的树状结构通过 `mod` 语句连接，例如 `mod foo` 表示模块 `foo` 是当前模块的一个子模块。
- 除了将模块写在一个单独的文件，也可以将模块写在一个代码块中：

```rust
mod foo {
    // 模块 foo 的内容，等效于将这里的代码写在 foo.rs 中
}
```

5.1.3 模块（续）

- 顶层模块一般位于 `src/main.rs` 或 `src/lib.rs`，根模块的子模块 `foo` 位于 `src/foo.rs`，`foo` 模块的子模块 `bar` 则位于 `src/foo/bar.rs`。
- `foo.rs` 也可以放在 `src/foo/mod.rs`，但不推荐这样做（会导致工程中有很多同名文件，不利于查找定位等操作）。

5.1.4 可访问性管理

- Rust 中各个类型、函数、结构体字段以及子模块的可访问性都基于模块。
- 一个模块中定义的各种项目默认只有当前模块以及子模块可以访问。
- 可以通过在项目和结构体的字段前标注 `pub` 使得项目或结构体的字段在模块外也可访问。

5.1.5 项目访问路径

- 直接访问项目时，默认使用相对路径，起始于当前模块：

```rust
mod one {
    mod two { pub fn foo() {} }
    fn bar() {
        two::foo();
    }
}
```

- self / super / crate 分别表示当前模块、上层模块、顶层模块：

```rust
self::two::foo
super::one::two::foo
crate::one::two::foo
```

5.1.6 通过 use 引入项目

- 可以使用 use 将项目引入到当前模块的作用域中，从而可以直接访问。

Rust

```rust
mod one {
    mod two {
        pub fn foo() {}
    }
    use two::foo;  // 将 two::foo 引入当前模块
    fn bar() {
        foo();  // 可以直接使用 foo
    }
}
```

5.1.7 通过 use 引入项目（续）

- 可以在一条 use 语句中通过大括号同时引入多个项目，或者通过 * 引入所有项目（谨慎使用）：

```rust
use one::{two::foo, bar};
use one::*;
```

5.1.8 重新导出

可以使用 pub use 来重新导出其他模块中的项目，等效于该项目就定义在本模块中：

Rust

```rust
mod one {
    mod two {
        pub fn foo() {}
    }
    pub use two::foo;  // 重新导出
}
fn baz() {
    one::foo();
}
```

5.1.9 箱（crate）

- 每棵模块树构成一个**箱**（crate）。
- **包**（package）是由 `Cargo.toml` 文件定义的一套 Rust 代码，一个包可以有零或一个**库箱**（library crate）和任意个**二进制箱**（binary crate）。
- 库箱可以在其他包中作为依赖被调用（即访问顶层模块中暴露的项目）。库箱的顶层模块默认位于 `src/lib.rs`。
- 二进制箱的顶层模块中有 `main` 函数，可以运行。二进制箱的顶层模块默认位于 `src/main.rs`。

5.1.10 代码组织

即使不打算在其他包中引用库箱，也可以

- 将核心逻辑写在库箱中
- 在二进制箱中引用库箱

这样能

- 让代码组织结构更加清晰
- 可以在多个二进制箱中共用代码
- 方便进行集成测试

5.1.11 使用库箱

- 运行命令 `cargo add rand` 或者手动编辑 `Cargo.toml` 来使用 crates.io 上名为 `rand` 的库箱：

```toml
[dependencies]
rand = "0.10.2"
```

- 通过库箱的名称访问其内容，如：`use rand::rng`。
- 在同一个包中，二进制箱里访问库箱也是通过库箱的名称（`Cargo.toml` 中的 package name）来访问。

5.1.12 使用自己写的箱

除了将库箱发布到 crates.io 之后使用，也可以使用 Git 仓库或本地文件系统中的库箱：

```toml
[dependencies]
myfoo = { git = "https://github.com/me/myfoo" }
mybar = { path = "../mybar" }
```

5.1.13 Cargo：默认代码位置

- 多个二进制箱可以放在 `src/bin/*.rs`，作为可执行文件。
  - 可以用 `cargo run --bin foo` 来指定运行的二进制箱。
- 示例代码放在 `examples/*.rs`。
  - 会在 `cargo test` 的时候构建，保证示例代码可以通过编译。
  - 可以用 `cargo run --example foo` 来运行。
- 集成测试（非单元测试）放在 `tests/*.rs`，用来测试正确性。
- 基准测试程序 (benchmarks) 放在 `benches/*.rs`，用来测试性能。

5.1.14 Cargo：特性

- 箱的特性（feature）是可以在构建时做选择性的开关。
- 使用箱作为依赖时，指定启用哪些特性：

```
cargo add rand --features=serde
```

- 在编写箱时支持特性：
  - 在 `Cargo.toml` 中的 `[features]` 里列出特性。
  - 可以将部分依赖设为可选依赖，仅在启用特性时使用可选依赖。
  - 二进制箱可以依赖于特性，只有启用了一些特性时才能构建某个二进制箱。

5.1.15 Cargo：构建脚本

- 如果遇到超过 Cargo 所提供的功能的特殊需求，可以通过构建脚本（build scripts）来实现。构建脚本也是用 Rust 语言编写的。

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

- 当构建脚本存在时，`cargo build` 会先编译并运行构建脚本。

5.1.16 Cargo：更多功能

- `cargo check`：快速验证代码是否能编译。
- `cargo clippy`：代码审查，提供改进建议。
- `cargo fmt`：格式化代码。
- `cargo doc`：生成文档。
- `cargo install`：从 crates.io 上安装二进制文件。
- 把自己写的软件包发布到 crates.io 上。
- 创建工作空间（workspaces），组织多个包协同开发。
- 通过 `cargo-<command>` 来扩展功能。
- Cargo 提供了很多有用的功能，详见 The Cargo Book。

5.2 语法补充

5.2.1 属性

- 属性 (attributes) 是 Rust 代码给编译器传递信息的机制。
  - 例如，`#[test]` 属性用于将函数标注为测试用例。
- `#[foo]` 标注随后的代码段，`#![foo]` 标注所在的代码段。

```rust
#[foo]
fn midterm1() {
    // ...
}
fn midterm2() {
    #![foo]
    // ...
}
```

5.2.2 常见属性

- `#![no_std]` 禁用标准库（用于嵌入式开发等没有标准库的环境）。
- `#[derive(Debug)]` 运行派生宏（例如自动获得特型）。
- `#[inline(always)]` 提示编译器的内联优化行为。
- `#[allow(missing_docs)]` 屏蔽编译器的某些警告。
- `#![crate_type = "lib"]` 提供箱的元数据。
- `#![feature(generic_const_exprs)]` 启用不稳定版本（nightly）的语法。
- `#[cfg(feature = "foo")]` 定义条件编译（启用箱的特性时编译）。
- `#[cfg(target_os = "linux")]` 定义条件编译（在特定操作系统上编译）。

更多属性可见参考手册。

5.2.3 操作符

Rust 语言中操作符的优先级顺序如下：

- 函数调用、下标索引：`()[]`
- 错误传播：`?`
- 单目操作符：`! - * & &mut`
- 类型转换：`as`
- 乘法类：`* / %`
- 加法类：`+ -`
- 位移类：`<< >>`
- 按位与：`&`
- 按位异或：`^`
- 按位或：`|`
- 比较类：`== != < > <= >=`
- 逻辑与：`&&`
- 逻辑或：`||`
- 范围：`.. ..=`
- 赋值：`= += -=` 等等

5.2.4 操作符重载

Rust 使用特型来重载操作符，定义在 `std::ops` 下。

- `Fn`, `FnMut`, `FnOnce`, `Index`, `IndexMut`
- `Neg`, `Not`, `Deref`, `DerefMut`
- `Mul`, `Div`, `Mod`
- `Add`, `Sub`
- `Shl`, `Shr`
- `BitAnd`, `BitXor`, `BitOr`
- `Eq`, `PartialEq`, `Ord`, `PartialOrd`
- `And`, `Or`
- `AddAssign`, `SubAssign`, ...

5.2.5 类型转换

- as 操作符不能重载。
- 使用 From 和 Into 实现自定义类型转换。
  - `trait From<T> { fn from(T) -> Self; }`，调用形式为 `Y::from(x)`。
  - `trait Into<T> { fn into(self) -> T; }`，调用形式为 `x.into()`。
- 如果实现了 From，则 Into 会自动实现。建议优先实现 From。
  - 也就是说，`From<T> for U` 蕴含 `Into<U> for T`。
- 此外，还有 TryFrom 和 TryInto，用于可能失败的转换。

5.2.6 自定义类型转换示例

Rust

```rust
struct A(Vec<i32>);
impl From<Vec<i32>> for A {
    fn from(v: Vec<i32>) -> Self {
        A(v)
    }
}
```

5.2.7 命名规范：标识符

- `CamelCase`：类型、特型
- `snake_case`：箱、模块、函数、方法、变量
- `SCREAMING_SNAKE_CASE`：常量和静态变量
- `T`（单个大写字母）：类型参数
- `'a`（撇 + 短的小写名字）：生命周期参数

5.2.8 命名规范：构造函数和转换函数

- `new`, `new_with_stuff`：构造函数
- `from_foo`：转换构造函数，例如 `from_str`
- `as_foo`：低开销非消耗性转换，例如 `as_ref`
- `to_foo`：高开销非消耗性转换，例如 `to_string`
- `into_foo`：消耗性转换，例如 `into_iter`

5.3 智能指针

5.3.1 &T 和 &mut T

- 基本的、经济型的引用
- 无运行时开销，所有检查在编译时完成。
- 具有固定的生命周期，没有灵活性。
- 如果可以，尽量使用引用。

- Box<T> 用于在堆上分配空间存放数据。
- Box<T> 拥有 T 类型的对象，它的指针是唯一的（类似 C++ 的 std::unique_ptr），不能创建别名（可以借用）。
- 当 Box<T> 超过作用域时，对象自动释放。
- 使用方法和不加 Box<T> 的一般类型相似，主要区别是动态分配。
- 通过 Box::new() 来创建。

```rust
let boxed_five = Box::new(5);
```

5.3.3 Box<T> 的特点

- 优点：
  - 是使用堆的最简单的办法。
  - 是动态分配的零开销抽象。
  - 借用和移动语义同样适用。
  - 自动销毁。
- 缺点：
  - `Box` 严格拥有里面的 `T` 类型的对象，是唯一的所有者。
  - 如果有引用的话，`Box` 本身必须存活到所有引用失效之后。

5.3.4 Box::leak()

`Box::leak()` 可以将 `Box<T>` 转换为 `&'static mut T` 类型，使得数据可以在程序余下的整个生命周期中使用。

```rust
let boxed_five = Box::new(5);
let static_five: &'static _ = Box::leak(boxed_five);
```

可用于全局信息的初始化，例如程序的配置信息。

- `Rc<T>` 是共享所有权的指针类型，相当于 C++ 中的 `std::shared_ptr`。
- 是“Reference Counted”的缩写，记录指针的别名个数。
- 调用 `Rc` 的 `clone()` 方法来获得新的引用。
  - 会增加引用计数。
  - 不会拷贝数据。
- 当引用计数降为 0 时，会释放对象。
- `T` 的值只有在引用计数为 1 时才能修改（通过 `get_mut()` 方法），与 Rust 的借用规则一致。

5.3.6 Rc 使用示例

```rust
let mut shared = Rc::new(6);
// 只有一个 Rc 时允许修改
println!("{:?}", Rc::get_mut(&mut shared)); // ==> Some(6)
// 创建该值的另一个引用
let mut cloned = shared.clone();
// 有多个 Rc 时不允许修改
println!("{:?}", Rc::get_mut(&mut shared)); // ==> None
println!("{:?}", Rc::get_mut(&mut cloned)); // ==> None
```

- 当有环时，引用计数会发生问题，例如：
  - A 有一个 B 的 Rc，B 也有一个 A 的 Rc，两者的引用计数都是 1。
  - 由于构成了环，两个对象都不会被释放，从而引起内存泄露。
- 可以通过引入**弱引用**来避免。
  - 弱引用不会增加**强引用**的计数。
  - 这也表示弱引用不一定总是有效的。
- Rc 可以通过 Rc::downgrade() 降级成 Weak。
  - 需要访问时，使用 weak.upgrade() -> Option<Rc<T>> 再升级回 Rc。
  - 除此之外，Weak 干不了别的事情，通过升级步骤避免使用无效状态。

5.3.8 强引用和弱引用

- 一般情况下，使用的时候需要通过 `Rc` 获得强引用。
- 如果应用场景中需要访问数据而没有所有权，就需要用到 `Weak`。
  - `Weak` 升级时可能会得到 `None`，需要应对这种情况。
- 引用成环的情况也需要用 `Weak` 来避免内存泄露。
  - 由于可变性规则，`Rc` 的环在 Rust 中很难形成。
- 考虑图的情况，保存顶点和边的时候哪里用 `Rc`，哪里用 `Weak`?

5.3.9 std::rc::Rc<T> 的特点

- 优点：
  - 允许共享数据的所有权。
- 缺点：
  - 有一定的运行时开销。
    - 维护两个引用计数（强和弱）。
    - 需要动态更新和检查引用计数。
  - 引用成环会造成内存泄露。

5.3.10 Cell 和 RefCell

- 允许**内部可变性**的机制。
- 通过不可变引用实现修改所含值的操作。
- 有两种常用的类型：`Cell<T>` 和 `RefCell<T>`。
- 此外，还有 `OnceCell<T>`。

```rust
struct Foo {
    x: Cell<i32>,
    y: RefCell<u32>,
}
```

5.3.11 std::cell::Cell<T>

- 为 Copy 类型提供内部可变性的格子类型。
- 用 get() 从 Cell 中取值。
- 用 set() 更新 Cell 中的值。
  - 不能修改 T 的值，只能整个替换。
- 作用比较受限，但是安全、低开销。

```rust
let c = Cell::new(10);
c.set(20);
println!("{}", c.get()); // 20
```

5.3.12 `std::cell::Cell<T>` 的特点

- 优点：
  - 实现内部可变性。
  - 没有运行时开销。
  - 额外空间开销较小。
- 缺点：
  - 较为受限，只能用于 `Copy` 类型。

- 可以为任意类型提供内部可变性的格子类型。
- **使用动态借用检查规则，在运行时进行。**
  - 有可能会引起运行时的恐慌。
- 通过 `borrow()` 或 `borrow_mut()` 来借用内部数据。
  - 如果 `RefCell` 已经被借用，可能会引起恐慌。

5.3.14 RefCell 使用示例

Rust

```rust
use std::cell::RefCell;

let refc = RefCell::new(vec![12]);
let mut inner = refc.borrow_mut();
inner.push(24);
println!("{:?}", *inner); // [12, 24]

let inner2 = refc.borrow();
// 由于 refc 已经被可变借用 (borrow_mut)，上一行会引起恐慌
```

5.3.15 std::cell::RefCell<T> 的常见用法

- RefCell 的常见用法是把它放在 Rc 里，这样可以允许共享的可变性。
  - 如果考虑可以为空，常见的组合是 Option<Rc<RefCell<T>>>，在树或图的场景中经常使用，比 Option<Box<T>> 更加灵活。
  - 辨析与 Rc<RefCell<Option<T>>> 的区别。
- 注意：这样做不是线程安全的，borrow() 等方法不能防止数据竞争（在下一讲中解释）。
- 可以使用 try_borrow() 等方法来检查借用是否成功。

5.3.16 std::cell::RefCell<T> 的特点

- 优点：
  - 为任意类型提供内部可变性
- 缺点：
  - 要保存额外的借用状态。
  - 要在借用时检查状态。
  - 有可能会引起恐慌。
  - 不是线程安全的。

5.3.17 自动解引用

Rust 在大多数情况下会对变量做自动解引用操作。

- 在对一个引用进行方法调用的时候。
- 将引用作为函数参数进行传递的时候。

```rust
fn length(vec_ref: &&Vec<i32>) -> usize {
    // 对引用调用方法时，会自动解引用
    vec_ref.len()
}
fn main() {
    let vector = vec![];
    // 传递引用时，也会自动解引用
    length(&&&&&&&&&&&&&vector);
}
```

5.3.18 Deref 自动转换

- Rust 的自动解引用行为也可以在类型之间工作。

```rust
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

- 因为 String 实现了 Deref<Target=str>，所以 &String 的值在需要的时候会自动解引用成 &str。

5.4 闭包

5.4.1 闭包的引入

很多时候在使用某个功能时需要额外指定一些行为作为参数，例如排序时想使用特定的顺序。

```rust
struct City {
    name: String,
    population: i64,
    country: String,
    // ...
}
cities.sort();
```

按城市名字排序？人口排序？升序？降序？

5.4.2 闭包的引入（续）

可以用函数来指定行为，称为用户自定义函数（user-defined function, UDF）。

```rust
use std::cmp::Ordering;
// struct City
fn cmp_population_desc(a: &City, b: &City) -> Ordering {
    a.population.cmp(&b.population).reverse()
}
cities.sort_by(cmp_population_desc);
```

编写单独函数的不足：逻辑分散、不方便提供额外信息。

5.4.3 闭包的概念

Rust 语言提供了函数式语言经常会用到闭包（closure），也称为匿名函数或者是 lambda 函数。

```rust
// struct City
cities.sort_by(
    |a, b| a.population.cmp(&b.population).reverse()
);
```

5.4.4 闭包的语法

```rust
let foo_v1 = |x: i32| -> i32 { x * x };
let foo_v2 = |x: i32, y: i32| x * y;
let foo_v3 = |x: i32| {
    let y = x * 2;
    let z = 4 + y;
    x + y + z
};
let foo_v4 = |x: i32| if x == 0 { 0 } else { 1 };
```

- 闭包语法：在 || 之间指定参数列表，之后是返回类型（可选）和返回表达式。
- 返回表达式可以是 {} 里的一组表达式。

5.4.5 类型推导

```rust
let square_v4 = |x: u32| { (x * x) as i32 };

let square_v4 = |x| -> i32 { x * x };  // Compile error
let square_v4 = |x|        { x * x };  // Compile error
```

- 与函数不同，闭包**可以不**指定参数或返回值的类型。
  - 但是在编译器无法推导出正确的类型时，仍然需要显式指定。
- 设计决策
  - 对函数来说，可能需要作为跨模块的接口，指定类型可以使得函数的定义更加清楚。
  - 对闭包来说，就在定义的位置使用，易用性更加重要。

5.4.6 闭包与环境：变量捕获

- 闭包可以使用它所在环境中的值，称为变量捕获（variable capture）。

```rust
let magic_num = 5;
let magic_johnson = 32;
let plus_magic = |x: i32| x + magic_num;
```

- 即使没有传参，闭包 `plus_magic` 也可以引用 `magic_num`。
  - `magic_num` 在该闭包的环境中。
  - `magic_johnson` 没有被捕获。
- 捕获同样遵循 Rust 的规则，分为三种：不可变借用、可变借用、转移。

5.4.7 变量捕获：不可变借用示例

Rust

```rust
let s = "Hello".to_string();
let f = |t| s.clone() + t;
println!("{}", f("Alice"));  // ==> HelloAlice
println!("{}", f("Bob"));    // ==> HelloBob
```

5.4.8 变量捕获：可变借用示例

Rust

```rust
let mut s = "Hello".to_string();
let mut f = |t| {
    s.push_str(t);
    s.clone()
};
println!("{}", f("Alice"));  // ==> HelloAlice
println!("{}", f("Bob"));    // ==> HelloAliceBob
```

5.4.9 变量捕获：转移示例

Rust

```rust
let s = "Hello".to_string();
let f = |t| s + t;
println!("{}", f("Alice"));  // ==> HelloAlice
// Error: println!("{}", f("Bob"));
```

5.4.10 闭包的作用域

当闭包本身的生命周期超过环境时，需要用 move 关键字指定转移捕获。

Rust

```rust
let f;
{
    let x = 5;
    f = move |y| x + y;  // 必须加上 move
    println!("{}", f(1));
}
println!("{}", f(2));
```

5.4.11 闭包特型

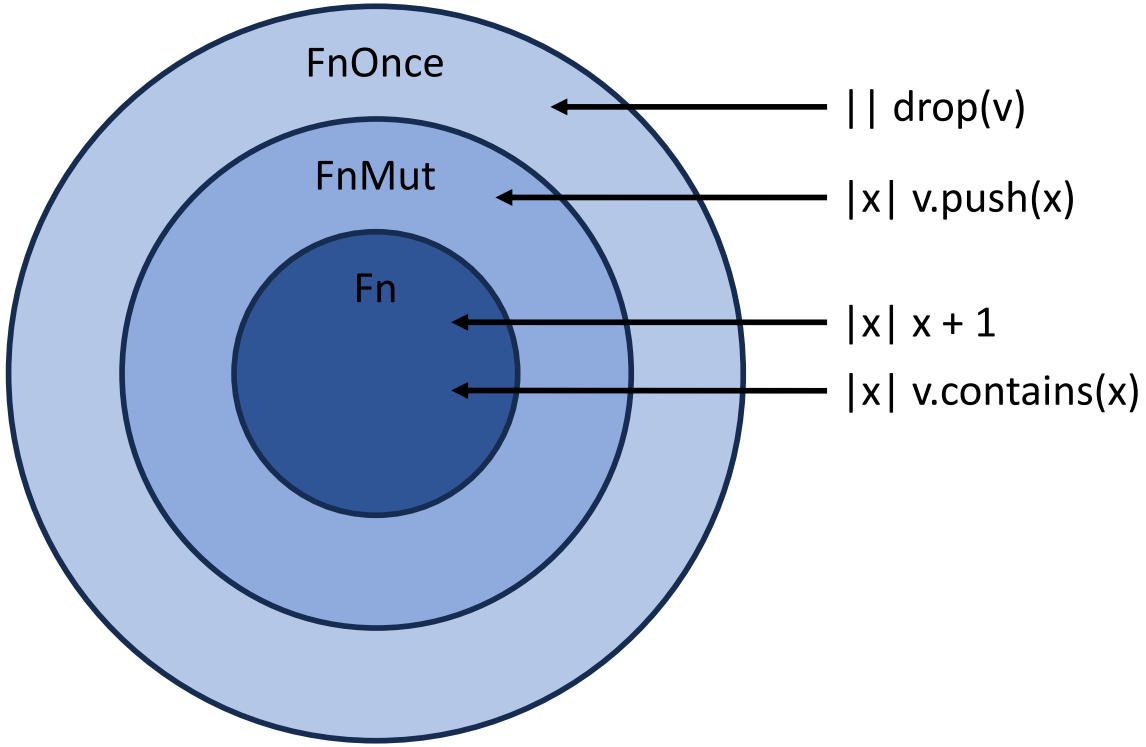
闭包根据所有权情况不同分别实现下列三种特型：Fn、FnMut、FnOnce

- 由编译器根据捕获方式自动确定。
- 实现这些特型的类型可以进行方法调用操作（重载操作符 ()）。

```rust
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}
pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call"
      fn call_mut(&mut self, args: Args) -> Self::Output;
}
pub trait FnOnce<Args> {
    type Output;
    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

5.4.12 闭包特型之间的关系

- `Fn`：不修改闭包的环境变量，可以反复调用。
- `FnMut`：修改闭包的环境变量，可以反复调用。
- `FnOnce`：获取闭包的环境变量的所有权，至少可以调用一次。



5.4.13 闭包特型的解释

- 这三种特型的区别在于处理 `self` 的方式：
  - `Fn` 借用 `self`，也就是使用 `&self`。
  - `FnMut` 以可变方式借用 `self`，也就是使用 `&mut self`。
  - `FnOnce` 获得 `self` 的所有权。
- 因此，`Fn` 是 `FnMut` 的子集，`FnMut` 是 `FnOnce` 的子集。
- 函数也实现了这类特型。
- 实际上 `|| {}` 表示闭包的语法是一种语法糖。Rust 会为闭包的环境生成一个结构体，实现相应的特型，然后使用。

5.4.14 将闭包作为参数

- 可以像函数指针一样传递闭包。
- 例如以下简化版本的 map 函数：

```rust
// self = Vec<A>
fn map<A, B, F>(self, f: F) -> Vec<B>
    where F: FnMut(A) -> B;
```

- map 有一个 f: F 参数，这里 F 是实现 FnMut 特型的类型。
- 可以使用普通函数，因为普通函数实现了 FnMut 特型。

5.4.15 返回闭包①

- 有时候需要将闭包作为函数的返回值。
- 但是闭包是特型对象，它的大小是不确定的。

```rust
fn i_need_some_closure() -> (Fn(i32) -> i32) {
    let local = 2;
    |x| x * local
}
```

```
error[E0782]: expected a type, found a trait
 --> src/main.rs:1:30
  |
1 | fn i_need_some_closure() -> (Fn(i32) -> i32) {
  |                              ^^^^^^^^^^^^^^
help: use `impl Fn(i32) -> i32` to return an opaque type
help: alternatively, you can return an owned trait object
```

5.4.16 返回闭包②

因此需要通过间接方式来返回闭包。

```rust
fn i_need_some_closure_by_reference()
    -> &dyn (Fn(i32) -> i32) {
    let local = 2;
    |x| x * local
}
```

Rust

```
error: missing lifetime specifier
```

- 没有给闭包指定生命周期。原因是这里返回的引用会比闭包本身活得更久，变成了悬垂指针。

5.4.17 返回闭包③

使用 Box 将其变为动态的生命周期。

```rust
fn box_me_up_that_closure()
    -> Box<dyn Fn(i32) -> i32> {
    let local = 2;
    Box::new(|x| x * local)
}
```

```
error[E0373]: closure may outlive the current function, but
it borrows `local`, which is owned by the current function
```

- 要返回的闭包依赖它的环境，当它被返回后，local 已经销毁。

5.4.18 返回闭包④

通过强制转移语法（move 关键字）来修正这个问题。

```rust
fn box_up_your_closure_and_move_out()
    -> Box<dyn Fn(i32) -> i32> {
    let local = 2;
    Box::new(move |x| x * local)
}
```

5.5 小结

5.5.1 本讲小结

- 模块、箱和包
- Cargo 高级用法
- 智能指针
- 闭包

下一讲：并发编程