链滴

Rust 模块系统

作者：lingyundu

原文链接：https://ld246.com/article/1611127187889

来源网站：链滴

Rust 提供了一套模块系统来组织和管理代码，包括：模块（module）、Crate、包（package）和作空间（workspace）。

包和 Crate

Crate 的英文意思是大木箱，它是一个模块树，并且是编译的基本单元，可以将其编译成可执行程序executable）或者库（library）。

因此，crate 基本分为两种：二进制Crate（binary crate）和库Crate（library crate）。

包（package）是包含一个或者多个crate的文件夹（目录）。

包（package）创建规则：

- 一个包中至多 **只能**包含一个库Crate。
- 包中可以包含任意多个 二进制Crate。
- 包中至少包含一个 crate，无论是库的还是二进制的。
- 包中应该包含一个 Cargo.toml 配置文件，用来说明如何去构建这些 crate。

包的简单目录结构：

```
.
├── Cargo.toml
├── src/
│   ├── lib.rs
│   ├── main.rs
│   └── bin/
│       ├── named-executable.rs
│       ├── another-executable.rs
│       └── multi-file-executable/
│           ├── main.rs
│           └── some_module.rs
└── tests/
    ├── some-integration-tests.rs
    └── multi-file-test/
        ├── main.rs
        └── test_module.rs
```

src/ 目录存放源代码，tests/ 目录存放测试代码。其中，main.rs 和 lib.rs 是两个特殊的 Rust 源文件，ib.rs 是 库Crate 的编译入口，main.rs 是 二进制Crate 的编译入口。另外，src/bin 目录下的文件也被编译成独立的可执行文件。

包的创建

使用 cargo new 命令创建包。

示例一：创建一个包含 二进制Crate 的包

```
$ cargo new my-project  --bin  # --bin 可省略
    Created binary (application) `my-project` package
$ ls my-project
```

原文链接：Rust 模块系统

```
Cargo.toml
src
$ ls my-project/src
main.rs   # 二进制 crate 的编译入口，二进制 crate 与包同名
```

编译后会生成一个名为my-project或者my-project.exe的可执行文件。

示例二：创建一个包含 库Crate 的包

```
$ cargo new my-project  --lib
    Created library (application) `my-project` package
$ ls my-project
Cargo.toml
src
$ ls my-project/src
lib.rs   # 库 crate 的编译入口，库 crate 与包同名
```

编译之后会生成一个名为 libmy-project.rlib 的 Rust 库文件。这是一种默认的格式，当然也可以生其他格式的库文件。

在 Linux 系统中使用下面的命令查看可以编译成哪些文件类型（格式）：

```
$ rustc --help|grep crate-type
       --crate-type [bin|lib|rlib|dylib|cdylib|staticlib|proc-macro]
```

示例三：创建一个包含 二进制Crate 和 库Crate 的包

Rust没有提供这样的命令，cargo new中的--bin和--lib选项不能同时使用。

```
$ cargo new my-object --bin --lib
error: can't specify both lib and binary outputs
```

其实很简单，只需在示例一的基础上，在src目录下添加一个lib.rs文件即可。

一般情况下，我们将与程序运行相关的代码放在 main.rs 文件，其他真正的任务逻辑放在 lib.rs 文件。

模块（Moduls）

模块是对 crate 中的代码进行分组的最简单直接的方法。

除了能够提高代码可读性和重用性，还可以控制模块中内容对外的访问权限（公有或私有）。另外，块也是可以嵌套的。

在 Rust 中每个源文件（.rs 后缀）都是一个模块，但不是所有的模块都有自己专属的源文件。

模块定义

使用mod关键字定义模块。

示例一：在 src/lib.rs 中定义三个模块

```
// 其中 hosting 和 serving 是 front_of_house 的子模块。
mod front_of_house {
```

```
    mod hosting {
        fn add_to_waitlist() {}
        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}
        fn server_order() {}
        fn take_payment() {}
    }
}
```

在一个文件中可以定义多个模块，当模块变大变多时，也可以将模块放到单独的文件中。

示例二：在文件中定义单个模块

```
// src/front_of_house.rs 文件中定义了一个单独的模块
pub mod hosting {
    pub fn add_to_waitlist() {}
}
```

模块引用

想要引用其他模块中的代码，需要定位这个模块。与文件系统类似，Rust 提供了两种路径形式：

- 绝对路径（absolute path）从 crate 根开始，以 crate 名或者字面值 crate 开头。
- 相对路径（relative path）从当前模块开始，以 self、super 或当前模块名开头。

示例一：在 src/lib.rs 中定义模块并引用

```
// 在 Rust 中模块、函数等默认都是私有的，只有使用 pub 关键字声明为公共的才能被引用。
// 父模块中的项不能使用子模块中的私有项，但是子模块中的项可以使用他们父模块中的项。
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

pub fn eat_at_restaurant() {
    // 绝对路径
    // `crate` 代表当前 Crate 中的根，类似于文件系统中的根目录
    crate::front_of_house::hosting::add_to_waitlist();

    // 相对路径
    // eat_at_restaurant 函数与 front_of_house 模块在同一个 Crate 的同一层级，所以从 front_of_
ouse 开始
    front_of_house::hosting::add_to_waitlist();
}
```

示例二：使用 super 和 self 消除歧义

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

```
}

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

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

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

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

        // `self` 关键字表示当前的模块 -- `my`。
        // 调用 `self::function()` 和直接调用 `function()` 都得到相同的结果，因为他们表示相同的函数

        self::function();
        function();

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

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

        // 绑定 *crate* 作用域（最外层）内的 `cool::function` 。
        {
            use crate::cool::function as root_function;
            root_function();
        }
    }
}
```

use 关键字

使用绝对路径或者相对路径调用其他模块中的函数虽然简单直接，但是比较麻烦。因此，Rust 提供一种简化方式。与 Java 中导入包类似，在 Rust 中使用的是 use 关键字将模块引入当前作用域。

示例一：使用 use 将模块引入作用域

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}
```

```
// 使用 use 指定模块路径。
use crate::front_of_house::hosting;

// hosting 模块在当前作用域就是有效的，可以直接使用。
// 前提是 hosting 模块中的函数是公有的。
pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

示例二：使用 as 关键字重命名引入作用域的类型

```
use std::fmt::Result;
// 引入其他模块时，可以把引入的结构体、函数等重命名
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
}

fn function2() -> IoResult<()> {
    // --snip--
}
```

示例三：使用外部包

首先要在 Cargo.toml 中配置要依赖的外部包的名称和版本等信息：

```
[dependencies]
rand = "0.5.5"  # Cargo 会自动去下载这个依赖
```

然后就可以在代码中使用 use 关键字引入这个 Crate 了：

```
use rand::Rng;

fn main() {
    let secret_number = rand::thread_rng().gen_range(1, 101);
}
```

标准库（std）也是外部 crate，因为标准库随 Rust 语言一同发行，无需配置 Cargo.toml 来引入 std
不过需要通过 use 将标准库中定义的项引入项目包的作用域中来引用它们，比如我们使用的 HashMap：

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

文件分层

当模块比较多时，我们可以讲模块分到不同的文件中，方便管理。

示例：重构之前的示例

```
.
├── Cargo.toml
├── src/
```

```
|   ├── lib.rs
|   ├── main.rs
|   ├── front_of_house.rs
|   └── front_of_house
|       └── hosting.rs
```

按照上面的目录结构创建一个 Rust 项目：

```
PS D:\Github> cargo new restaurant
    Created binary (application) `restaurant` package
PS D:\Github> cd .\restaurant\src\
PS D:\Github\restaurant\src> New-Item lib.rs

    目录: D:\Github\restaurant\src
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        2021/1/24     17:02              0 lib.rs

PS D:\Github\restaurant\src> New-Item front_of_house.rs

    目录: D:\Github\restaurant\src
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        2021/1/24     17:02              0 front_of_house.rs

PS D:\Github\restaurant\src> mkdir front_of_house

    目录: D:\Github\restaurant\src
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        2021/1/24     17:02                front_of_house

PS D:\Github\restaurant\src> New-Item .\front_of_house\hosting.rs

    目录: D:\Github\restaurant\src\front_of_house
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        2021/1/24     17:03              0 hosting.rs

PS D:\Github\restaurant\src>
```

src/front_of_house/hosting.rs 文件中的内容：

```
pub fn add_to_waitlist() {}
```

src/front_of_house.rs 文件中的内容：

```
// 查找名为 `hosting.rs` 的文件，
// 并将该文件的内容放到一个名为 `hosting` 的模块里面。
pub mod hosting;
```

lib.rs 文件中的内容：

```
// 查找名为 `front_of_house.rs` 的文件，
// 并将该文件的内容放到一个名为 `front_of_house` 的模块里面。
```

```
mod front_of_house;

// 使用 `pub use` 重导出（Re-exports）
// 重导出后，不仅当前模块可以使用 `hosting` 模块，在当前模块之外也可以使用
pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

main.rs 文件的内容：

```
// 引入 restaurant 库Crate 中的代码
// `restaurant` 是 库Crate 的名称，通常与包名相同
use restaurant::*;

fn main() {
    eat_at_restaurant();

    // 此处可以调用 `hosting` 模块中的函数
    // 是因为在 `lib.rs` 中对`hosting` 模块进行了重新导出
    hosting::add_to_waitlist();
}
```

工作空间

随着项目开发的深入，库Crate 持续增大，有必要将其进一步拆分成多个库Crate。对于这种情况，Cargo提供了一个叫 工作空间（workspaces）的功能，它可以帮助我们管理多个相关的协同开发的包。

创建工作空间

1. 新建一个工作空间目录。

示例：

```
$ mkdir add
$ cd add
```

2. 在工作空间目录中创建 Cargo.toml文件，并在文件中配置members （Crate 的名称）。

示例：

```
[workspace]
members = [
    "adder","add-one","add-two",
]
```

3. 根据配置创建包

示例：

```
$ cd add
```

```
$ cargo new adder  # 新建一个包含二进制Crate的包
$ cargo new add-one --lib # 新建一个包含库Crate的包
$ cargo new add-two --lib # 新建另一个包含库Crate的包
```

4. 配置依赖

示例:

假如adder依赖了add-one，需要在adder的Cargo.toml文件中进行相应的配置。

```
[dependencies]
add-one = { path = "../add-one" }
```

配置之后就可以在 adder/src/main.rs 中引用 add-one 库了。

```
use add_one;
fn main() {
    let num = 10;
    println!("Hello, world! {} plus one is {}!", num, add_one::add_one(num));
}
```

其中 add_one::add_one 是 add-one/src/lib.rs 中定义的函数:

```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

5. 构建和运行

在工作空间目录中运行 cargo build 来构建工作空间。该命令会编译所有的 Crate，并将编译好的文统一放到当前目录下/target/debug目录。

如果工作空间中仅有一个 二进制Crate，直接在工作空间目录中使用cargo run 命令来运行。

如果工作空间中有多个 二进制Crate，需要通过 -p 参数指定包名称来运行工作空间中的包。比如:

```
$ cargo run -p adder
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/adder`
Hello, world! 10 plus one is 11!
```

相关资料

Rust Programming Language

Rust by Example

Crates and source files

Modules

Cargo Workspaces