Chapter 3 第3章 Cargo 项目管理与编译

rustc 一次只能编译一个源文件，适用于编译开发一些简单的程序，而且输入编译命令时需要输入各种各样的编译开关，比较烦琐，不方便。而程序员开发的通常是一个大程序：编译的时候需要知道程序的相关信息，应该如何编译以及编译时需要的依赖包等信息。Rust 提供了功能非常强大的包管理器 Cargo，可以方便地进行项目创建、项目管理（元数据管理、依赖管理、编译配置、条件编译等）、项目运行测试、项目发布等操作。

Cargo 命令的使用方法及参数如下：

```
Rust Programming\sourcecode\chapter2>cargo
Rust's package manager

Usage: cargo [+ toolchain] [OPTIONS] [COMMAND]

Options:
  -V, --version             Print version info and exit
      --list                List installed commands
      --explain <CODE>      Run `rustc --explain CODE`
  -v, --verbose...          Use verbose output (-vv very verbose/build.rs output)
  -q, --quiet               Do not print cargo log messages
      --color <WHEN>        Coloring: auto, always, never
  -C  <DIRECTORY>           Change to DIRECTORY before doing anything (nightly-only)
      --frozen              Require Cargo.lock and cache are up to date
      --locked              Require Cargo.lock is up to date
      --offline             Run without accessing the network
      --config <KEY=VALUE>  Override a configuration value
  -Z  <FLAG>                Unstable (nightly-only) flags to Cargo, see 'cargo -Z help' for details
  -h, --help                Print help

Some common cargo commands are (see all commands with --list):
    build, b    Compile the current package
    check, c    Analyze the current package and report errors, but don't build object files
    clean       Remove the target directory
    doc, d      Build this package's and its dependencies' documentation
    new         Create a new cargo package
    init        Create a new cargo package in an existing directory
    add         Add dependencies to a manifest file
    remove      Remove dependencies from a manifest file
    run, r      Run a binary or example of the local package
    test, t     Run the tests
    bench       Run the benchmarks
    update      Update dependencies listed in Cargo.lock
    search      Search registry for crates
    publish     Package and upload this package to the registry
    install     Install a Rust binary. Default location is $ HOME/.cargo/bin
    uninstall   Uninstall a Rust binary

See 'cargo help <command>' for more information on a specific command.
```

可以用下面的命令(cargo new)来创建两种类型的项目(Package,亦称编译单元(Compile Unit)。为避免歧义,本书以后统一使用 Package 或者其中文名——项目)。

创建一个库/Library Package(也称为 **Library crate 项目**):

```
cargo new --lib encrypt        // 创建一个名为 encrypt 的库/Library 项目(也称为库 crate 项目)
```

创建一个可执行 Package(也称为**二进制 crate 项目**):

```
cargo new projectname --bin    // 创建一个名为 projectname 的可执行项目(也称为二进制 crate 项目)
```

本章介绍一些编程常用的 Cargo 命令和配置。关于 Cargo 命令的更详细的信息,请参考 Cargo 的官方在线书籍①。

3.1　项目结构

一般通用的、也是推荐的 Cargo 项目的目录结构如下:

```
SampleProject          // 项目根目录
├─benches              // 性能测试目录: cargo bench
|     ├─ large-input.rs
|     └─ multi-file-bench/
|           ├─ main.rs
|           └─ bench_module.rs
├─examples             // 该目录下是示例文件: 使用本项目的一些周边的例子: cargo example
|     ├─ simple.rs
|     └─ multi-file-example/
|         ├─ main.rs
|         └─ ex_module.rs
├─src                  // 源代码目录
|   ├─bin              //一个项目如果包含多个二进制 crate,除了 main.rs 以外,可以放在 bin 目录下
|   |   ├─ named-executable.rs
|   |   ├─ another-executable.rs
|   |   └─ multi-file-executable/
|   |         ├─ main.rs
|   |         └─ some_module.rs
|   ├─submodule        // 子模块
|   ├─memeber crate    // 成员 crate 项目
|   ├─lib.rs           // 主库 crate 文件
|   └─main.rs          // 主二进制 crate 文件
├─target               // 编译结果目录
|   ├─debug            // debug 版本编译结果目录
|   └─release          // release 版本编译结果目录
├─tests                // 测试目录: cargo test
|     ├─ some-integration-tests.rs
|     └─ multi-file-test/
|         ├─ main.rs
|         └─ test_module.rs
├─build.rs             // 预构建脚本文件
├─Cargo.lock           //Cargo lock 文件
└─Cargo.toml           //Cargo 配置文件
```

一个 Cargo 编译单元(Package)可以有:

- 一个库 crate(src/lib. rs);

① https://doc. rust-lang. org/cargo

- 可以同时有多个二进制 crate,一个 main.rs 在项目 src 目录下,其他在 src/bin 目录下;
- build.rs 叫作构建脚本(Build Script),每次 cargo build,它首先执行一些任务,如项目依赖的第三方非 Rust 代码、进行一些相关检查等;
- 示例目录(examples);
- 测试目录(tests);
- 基准测试目录(bench);
- Cargo 配置文件 Cargo.toml 和 Cargo.lock;
- 编译的结果目录 target/debug,target/release。

3.2 Cargo 的配置文件

每个编译单元(Package)都有一个 Cargo.toml 和 Cargo.lock 文件,它们是 Cargo 项目代码管理的两个核心文件:

- Cargo.toml 由程序员编写,提供编译单元的元数据、配置以及依赖包等信息;
- Cargo.lock 由 Cargo 命令自动维护,不需要手工编辑,包含编译单元目前的库包依赖信息。

3.2.1 Cargo.toml

Cargo.toml 文件使用 TOML (Tom's Obvious,Minimal Language)[①]格式。表 3.1 列出了 Cargo.toml 中用到的所有关键字以及使用示例。

表 3.1 Cargo.toml 的关键字和示例

表 名	名 称	说 明	示 例
根	cargo-features	不稳定的 nightly 版本专用	*cargo-features = ["edition2021"]*
Package	name	项目名字	
	version	项目版本	
	authors	项目作者	
	edition	Rust 的版本	*edition="2018"* **目前可以有 2015、2018、2021 3 个可能的值**
	rust-version	所支持的 Rust 最低版本	
	description	项目描述	*description = "create books from markdown files"*
	documentation	项目文档的 URL	*documentation = "http://azerupi.github.io/mdBook/index.html"*
	readme	项目的 README 文件	*readme = "README.md"*
	homepage	项目的首页 URL	*homepage = "https://github.com/paritytech/parity-wasm"*
	repository	项目源代码仓库的 URL	*repository = "https://github.com/azerupi/mdBook"*

① https://toml.io/en/

续表

表　名	名　称	说　明	示　例
Package	license	项目的授权证明(开源协议的 License)	`License="MIT"`
	license-file	开源协议的 License 文件的路径	
	keywords	项目关键字	`keywords = ["book","gitbook","rustbook","markdown"]`
	categories	项目的分类	
	workspace	项目的工作空间路径	
	build	项目预编译脚本的路径	`build = "build.rs"`
	links	项目链接的本地库包的名字	

	exclude	在发布版本时排除的文件	`exclude = [`
			`  "book-example/* ",`
			`  "src/theme/stylus",`
			`]`

	include	在发布版本时需要包括的文件	
	publish	可以用来声明不要发布本项目	
	metadata	外部工具的配置	
	default-run	Cargo run 运行的默认的可执行二进制文件	
	autobins	禁止二进制可执行文件自动发现	见官方文档
	autoexamples	禁止例子的自动发现	见官方文档
	autotests	禁止测试案例的自动发现	见官方文档
	autobenches	禁止性能测试的自动发现	见官方文档
	resolver	设置所使用的依赖关系解析器	`resolver = "2"` **在 edition = "2021" 的情况下，默认 resolver = "2"** **见官方文档①**

[target]		目标平台设置	`[target.'cfg(windows)'.dependencies]`
			`winapi = "0.2"`
			
			`[target.'cfg(not(windows))'.dependencies]`
			`daemonize = "0.2"`

	[lib]	库包目标配置	`[lib]`
			`crate-type = ["cdylib"]`
			`bench = false`

① https://doc.rust-lang.org/edition-guide/rust-2021/default-cargo-resolver.html#details

续表

表　名	名　称	说　明	示　例
[target]	[[bin]]	可执行文件设置； 两个方括号([[...]])表示每个 package 可以有多个可执行文件的设置	[[bin]] name = "cool- tool" test = false bench = false [[bin]] name = "frobnicator" **cargo run 命令使用-- bin < bin-name> 开关来运行不同的可执行配置**
	[[example]]	例子程序设置； 两个方括号([[...]])表示每个 package 可以有多个例子程序的设置	[[example]] name = "timeout" path = "examples/timeout.rs" **通过 cargo run -- example NAME 调用**
	[[test]]	目标测试设置； 两个方括号([[...]])表示每个 package 可以有多个测试设置	[[test]] name = "testinit" path = "tests/testinit.rs" [[test]] name = "testtime" path = "tests/testtime.rs"
	[[bench]]	性能测试设置 两个方括号([[...]])表示每个 package 可以有多个性能测试设置	
[dependencies]		项目库包依赖包声明	[dependencies] rand = "0.3.14"
[dev-dependencies]		例子(examples)、测试(tests)和性能测试(benches)的依赖包声明	[dev- dependencies] log = "0.3"
[build-dependencies]		预编译脚本的依赖声明	[build- dependencies] rustc_version = "0.1"
[badges]		项目在 crate. io 发布时使用的徽章	
[features]		条件编译特性	[features] default = ["output","watch","serve"] debug = [] output = [] watch = ["notify","time","crossbeam"]
[patch]		被重载的依赖库包	见官方文档
[replace]		被重载的依赖库包(不推荐使用)	见官方文档
[profile]		编译器设置和优化设置	[profile.release] lto = true 详细见官方文档
[workspace]		工作空间定义	见 3.11 节

具体的讨论示例等请参照官方文档[①]。本章后面主要讨论常用的几个表名：package、patch、profile 和 dependency。

3.2.2　Cargo.lock

Cargo.lock 文件的目的是保持最近成功编译时的项目状态。这么做的原因是保证在不同的机器上编译时，使用的依赖库包的版本都是相同的。这样，在不同的机器上编译同一个项目会得到同样的编译结果。这样可序列化的状态可以在不同的计算机和开发团队之间流畅地共享使用。因此，如果一个依赖包引入了一个错误（如通过打补丁），除非使用了 cargo update，否则在其他计算机和其他开发团队的编译结果中并不会被影响。在实际应用中，**建议在开发库包时把 Cargo.lock 文件也纳入版本管理**，这样可以保证一个可稳定工作的编译版本。即使为了调试目的，Cargo.lock 的版本历史也能帮助程序员很容易地梳理出库与包之间的依赖树。

Cargo.lock 是由 Rust 编译器自动生成和管理的（这也是 2.6 节的例子里为什么没有 Cargo.lock 的原因，因为还没有编译该项目，所以 Cargo.lock 文件还未生成）。一般情况下，推荐使用 cargo update 命令来修改 Cargo.lock 文件；而并不推荐程序员对该文件进行手工修改。cargo update 命令能够根据 Cargo.toml 描述文件，重新检索并更新各种依赖项的信息，并写入 Cargo.lock 文件，例如依赖项版本的更新变化等。

3.2.3　Cargo.lock vs Cargo.toml

如果我们要上传项目到代码仓库（如 GitHub 或者 Gitee），那么，是否需要上传 Cargo.lock 呢？一般来说有如下判断准则：

- 如果项目是作为第三方库类型的服务，就可以把 Cargo.lock 加入 gitignore 中，这意味着不用上传 Cargo.lock；
- 如果项目是一个面向用户的终端产品，例如一个手机 App 的新版本，那么就需要把 Cargo.lock 上传到代码仓库中；这是因为每一个确定的终端版本都依赖于特定的依赖库包及特定的版本，Cargo.lock 能够保证在不同的终端环境里编译出来的终端版本都是相同的。

我们可以通过下面的命令以手动的方式将依赖更新到新版本：

```
$ cargo update              # 更新所有依赖
$ cargo update -p rand      # 只更新"rand"
```

3.3　依赖包

下面罗列了指定依赖包[dependencies]的位置和版本的方式。

- Crates.io 注册表。这是默认的选项，需要以字符串的方式设定 package 的名字和版本：

```
[dependencies]
async-std ="1.7"
```

① https://doc.rust-lang.org/cargo/

- 其他的注册表。注册表的名字必须在 $HOME/.cargo/config 文件里配置。并且,在 Cargo.toml 中,在声明依赖包时,必须声明注册表的名字。例如:假设 $HOME/.cargo/config 文件的内容如下:

```
1.  [registries]
2.  ustc ={ index ="https://mirrors.ustc.edu.cn/crates.io-index/" }     # 中科大镜像源
3.  rustcc={ index ="git://code.aliyun.com/rustcc/crates.io-index" }   # 阿里云镜像源
4.
5.  [source.crates-io]
6.  registry ="https://github.com/rust-lang/crates.io-index"
7.  #replace-with ='ustc'
8.  [source.rustcc]
9.  registry ="git://code.aliyun.com/rustcc/crates.io-index"
10. [source.ustc]
11. registry ="git://mirrors.ustc.edu.cn/crates.io-index"
```

那么,我们就可以用如下方式声明依赖:

```
[dependencies]
cratename ={ version ="2.1", registry ="ustc" }
```

如果更改了注册表,在如上例重新配置为 ustc 后,因为要下载更新 ustc 注册服务的索引文件,所以初次构建可能要比较久的时间。

如果我们注释上面的第 6 行并去除第 7 行的注释,就意味着我们直接使用新注册服务(ustc)来替代默认的 crates.io。这样做的好处是不用再手动地更改每个 crate 的注册表,同时也不需要重新构建,节省了不少时间。

- Git 仓库。Git 仓库可以被用来指定依赖包:

```
[dependencies]
chrono ={ git ="https://github.com/chronotope/chrono" , branch ="master" }
```

Cargo 会自动获取指定分支和指定位置的代码仓库,并寻找 Cargo.toml 文件,下载其依赖包。

- **指定一个本地路径**。Cargo 支持基于路径的依赖包。一个库包可以作为主 package 的子 crate。那么在编译主 package 时,子 crate 作为依赖包也会被编译。但是这种依赖包是不能上传到 crates.io 公共注册表的:

```
[dependencies]
bitcrypto ={ path ="../crypto" }
primitives ={ path ="../primitives" }
```

- **多引用方式混合**。Cargo 支持同时指定一个注册表和 Git 或者路径位置。在本地编译时,使用 Git 或者路径,而注册表的版本会用于 package 发布到 crates.io 之后。实际上,我们可以同时使用多种方式来引入同一个包,例如本地引入和 crates.io:

```
[dependencies]
# 本地使用时,通过 path 引入,
# 发布到 crates.io 时,通过 crates.io 的方式引入:  version ="1.0"
rand ={ path ="my-rand", version ="1.0" }
```

或者

```
# 本地使用时,通过 Git 仓库引入
# 当发布时,通过 crates.io 引入: version ="1.0"
inkwell ={ git ="https://github.com/TheDan64/inkwell", version ="1.0" }
# 如果 version 无法匹配,Cargo 将无法编译
```

- **根据平台引入依赖**：我们还可以根据特定的平台来引入依赖。例如：

```
[target.'cfg(windows)'.dependencies]
hyper ="0.3.0"
[target.'cfg(unix)'.dependencies]
openssl ="1.0.3"
[target.'cfg(target_arch ="riscv")'.dependencies]
platform ={ path ="platform/riscv" }
[target.'cfg(target_arch ="arm")'.dependencies]
platform ={ path ="platform/arm" }
```

还能使用逻辑操作符进行控制。例如：

```
[target.'cfg(not(unix))'.dependencies]
openssl ="1.0.1"     # 如果不是 unix 操作系统时,才对 openssl 进行引入。
```

如果想要知道 cfg 能够作用的目标,可以在终端中运行 rustc --print=cfg 进行查询,也可以通过指定平台查询：rustc --print=cfg --target=x86_64-pc-windows-msvc,该命令将对 64bit 的 Windows 进行查询。示例如下：

```
Rust Programming\sourcecode>rustc --print=cfg
debug_assertions
panic="unwind"
target_arch="x86_64"
target_endian="little"
target_env="msvc"
target_family="windows"
target_feature="fxsr"
target_feature="sse"
target_feature="sse2"
target_has_atomic="16"
target_has_atomic="32"
target_has_atomic="64"
target_has_atomic="8"
target_has_atomic="ptr"
target_os="windows"
target_pointer_width="64"
target_vendor="pc"
windows
```

Cargo 还允许通过下面的方式来引入平台特定的依赖：

```
[target.x86_64-pc-windows-gnu.dependencies]
hyper ="0.3.0"
[target.i686-unknown-linux-gnu.dependencies]
openssl ="1.0.3"
```

常见的几个使用示例如下：

```
[dependencies]
num ="0.1.27"     # 标准的包依赖声明,编译器会自动到 crates.io 上面寻找并下载,相关文档在 doc.rs 上
rand ={ git ="https://github.com/rust-lang-nursery/rand", branch ="0.4" }    #指定特定的位置
```

```
bytemuck ={ version ="1.7.2", features =["derive"]}    #可以指定 crate 的 feature
solana-sdk ={ path ="../sdk", version ="=1.10.0" }     #可以指定本地路径的 crate
codec ={ package ="parity-scale-codec", version ="2.0.0", default-features =false,
features =["derive"]}                                  # 可以使用 derive 属性
```

Rust 还可以单独声明包依赖。下面的例子和上面的 rand 包声明(斜体)是等价的:

```
[dependencies.rand]   // 这相当于将 rand crate 的声明从[dependencies]表移出,单独声明
git ="https://github.com/rust-lang-nursery/rand"
branch ="0.4"         # Git 版本的 0.4 分支
```

声明依赖包的版本信息时,使用如下格式:

```
<major>.<minor>.<patch>
```

可以在版本信息里使用一些特殊字符来向编译器表明包含或者不包含某些特定的版本。

- 波浪线/Tilde (~):如果指定了 major、minor 和 patch 版本,或者只指定了 major 和 minor 版本,那么只允许 patch 部分有变化;如果只指定了 major 版本,那么 minor 和 patch 部分也可以有变化。例如:

```
~1 等价于 >=1.0.0 且 <2.0.0
~1.3 等价于 >=1.3.0 且 <1.4.0
~1.3.2 等价于 >=1.3.2 且 <1.4.0
```

- 补注号/Caret (^):允许在不修改<major>、<minor>、<patch>中最左边非零数字的情况下变化。例如:

```
^0 等价于 >=0.0.0 且 <1.0.0
^0.1 等价于 >=0.1.0 且 <0.2.0
^0.1.3 等价于 >=0.1.3 且 <0.2.0
^0.2.5 等价于 >=0.2.5 且 <0.3.0
^1 等价于 >=1.0.0 且 <2.0.0
^1.4 等价于 >=1.4.0 且 <2.0.0
^1.3.5 等价于 >=1.3.5 且 <2.0.0
```

- 通配符/Wildcard (*):允许任何版本。例如:

```
2.*等价于 >=2.0.0 且<3.0.0
1.8.*等价于 >=1.8.0 且<1.9.0
```

- 手工指定:用 >、>=、<、<=、=来指定版本号。例如:

```
=1.3.2
>=1.5.0
>1.2
<2
>=1.1, <1.4.     # 多个版本的约束需要用逗号隔开
```

- 如果版本指定为 1,则 Cargo 会使用该 crate 版本 2 之前的最新版本。

3.4 开发时依赖包

可以为项目添加只在测试时需要的依赖库。可以在 Cargo.toml 中添加[dev-dependencies]来实现。例如:

```
[dev-dependencies]
base58="0.3"
```

[dev-dependencies]下的依赖只会在运行测试(tests)、示例(examples)和基准性能测试(benches)时才会被引入。并且,假设 A 库包引用了 B 库包,而 B 库包通过[dev-dependencies]的方式引用了 C 库包,那么 A 库包是不会引用 C 库包的。我们还可以指定平台特定的测试依赖包。例如:

```
[target.'cfg(unix)'.dev-dependencies]
tokio="0.1.0"
```

3.5　Cargo 目标对象

Cargo 项目中包含一些对象,它们包含的源代码文件可以编译成相应的包,这些对象称为 Cargo Target。例如之前章节提到的库对象(library)、二进制对象(binary)、示例对象(examples)、测试对象(tests)和基准性能对象(benches)都是 Cargo 的目标对象。

本节介绍如何在 Cargo.toml 清单中配置这些对象,当然,大部分时候都无须手动配置,因为默认的配置通常由项目目录的布局自动推断出来。在开始讲解如何配置对象前,先来看看这些对象究竟是什么。

3.5.1　库对象

库对象用于定义一个库,该库可以被其他的库或者可执行文件链接。该对象包含的默认文件名是 src/lib.rs,且**默认情况下,库对象的名称和项目名是一致的**,除非特别指定。

一个编译单元只能有一个库对象,因此也只能有一个 src/lib.rs 文件,以下是一种自定义配置:

```
# 在 Cargo.toml 中定制化库对象
[lib]                                    # 库对象
crate-type =["cdylib"]                   # 指定库对象的类型
bench =false
```

3.5.2　二进制对象

二进制对象在编译后可以生成可执行文件,默认的文件名是 src/main.rs,二进制对象的名称和项目名也是相同的。Rust 编译器默认会去编译 main.rs。但是,如果源文件名不是 main.rs,则需要在二进制对象中指定,否则 Rust 编译器不知道编译哪个入口文件。

一个项目可能拥有多个二进制文件,因此**一个项目可以拥有多个二进制对象**。当拥有多个二进制对象时,**这些对象的文件默认放在 src/bin/目录下**。

二进制对象可以使用库对象提供的公共 API,也可以通过定义在 Cargo.toml 中的[dependencies]来引入外部的依赖库。可以使用 cargo run --bin <bin-name>的方式来运行指定的二进制对象,以下是二进制对象的配置示例:

```
# Cargo.toml 文件中定制二进制可执行对象.
[[bin]]                  #二进制对象
name ="pestexample"
test =false
```

```
bench =false
path =" src/bin/pestexample.rs"
[[bin]]                    #二进制对象
name =" pestzok"
path =" src/pestzok.rs"
```

使用下面的命令可以运行相应的二进制对象:

```
cargo run --bin pestexample  // 指定要运行的二进制对象
cargo run                    // 等价于 cargo run --bin pestzok
                             // 因为项目名为 pestzok,如果不指定要运行的二进制对象,就会运行 pestzok
```

3.5.3 示例对象

示例对象的文件在根目录下的 examples 目录中。示例程序一般使用项目中库对象的功能进行演示。示例对象编译后的文件会存储在 target/debug/examples 目录下。

示例对象可以使用库对象的公共 API,也可以通过定义在 Cargo. toml 中的[dependencies]或者[dev-dependencies]来引入外部的依赖库。默认情况下,示例对象都是可执行的二进制文件(带有 fn main()函数入口),这主要是因为示例程序是用来测试和演示库对象的,是用来运行的。也可以将示例对象改成库的类型。例如:

```
[[example]]
name =" foo"
crate-type =[" staticlib"]
```

如果想要指定运行某个示例对象,可以使用如下命令:

```
cargo run --example <example-name>
```

如果是库类型的示例对象,则可以使用如下命令进行构建:

```
cargo build --example <example-name>
```

与此类似,还可以使用如下命令将示例对象编译出的可执行文件安装到默认的目录中,将该目录添加到 $PATH 环境变量中,就可以直接全局运行安装的可执行文件。例如:

```
cargo install --example <example-name>
```

cargo test 命令默认会对示例对象进行编译,以防止示例程序因为长久未运行而导致过期以至于无法运行。

3.5.4 测试对象

如果运行 cargo test,从输出可以看到测试内容有以下三部分。

- **单元测试**:单元测试是库项目或者二进制项目中用#[test]标记的函数,这些函数能够访问定义在项目中的私有 API。详情参见 6.1 节。
- **集成测试**:是一个独立的可执行二进制项目,也包含#[test]标记的函数,它与项目的库链接并可以访问项目的公开 API。详情参见 6.2 节。使用示例如下:

```
cargo test --test integration_test  // 此处假设 tests 目录下有 integration_test.rs 测试案例存在
```

- **文档测试**:详情参见 6.3 节。

测试对象的文件位于项目根目录下的 tests 目录中。当运行 cargo test 时，里面的每个文件都会被编译成独立的包，然后被执行。测试对象可以使用库对象提供的公共 API，也可以通过定义在 Cargo.toml 中的[dependencies]和[dev-dependencies] 来引入外部的依赖库。

如果希望在多个集成测试之间共享代码，可以把要共享的部分放在独立的模块，如 tests/common/mod.rs 并定义 mod common 模块。然后，每个测试都可以引入该模块 common。通过这种文件组织和命名方式，Cargo 会忽略 tests/目录的子目录的文件。Rust 不再将 common 模块看作集成测试文件：**tests 目录下的子目录中的文件不会被当作独立的包，也不会有测试输出**。

每个集成测试都会生成一个独立的可执行二进制文件，cargo test 会顺序地执行它们。如果有大量的集成测试，可以把测试分割为多个模块，并生成单一的集成测试。可以运行 cargo test 命令，传入要运行的测试的模块名。例如：

```
cargo test foo ----test-threads 3    // 运行名为 foo 的测试模块.--test-threads 是测试模块的参数
```

Rust 为了保持测试结果的简洁，通常会省略成功测试的标准输出（如 println!的输出）。可以通过传送--nocapture 开关来显示成功测试的输出。例如：

```
cargo test --  --nocapture            // --不可省略
```

或者

```
cargo watch  "test --  --nocapture"
```

3.5.5　基准性能对象

基准性能测试的文件位于 benches 目录下，可以通过 cargo bench 命令来运行。整个结构类似于 tests。每个基准性能测试的函数都使用#[bench]属性标记。官方的基准性能测试的缺点，首先是**不支持 stable 版本的 Rust**，其次是结果有些简单，缺少更详细的统计分布。因此社区基准测试库包（benchmark crate）就应运而生了，如 criterion.rs[①]。

3.5.6　配置一个对象

可以通过 Cargo.toml 中的[lib]、[[bin]]、[[example]]、[[test]]和[[bench]]部分对以上对象进行配置。由于它们的配置内容都是相似的，因此以[lib]为例[②]来说明相应的配置项。例如：

```
[lib]
name ="foo"                # 对象名称：库对象、src/main.rs 二进制对象的名称默认是项目名
path ="src/lib.rs"         # 对象的源文件路径
test =true                 # 能否被测试，默认是 trued
doctest =true              # 文档测试是否开启，默认是 true
bench =true                # 基准测试是否开启
doc =true                  # 文档功能是否开启
plugin =false              # 是否可以用于编译器插件（作废）
proc-macro =false          # 是否是过程宏类型的库
harness =true              # 是否使用 libtest harness
edition ="2018"            # 对象使用的 Rust Edition
crate-type =["dylib"]      # 生成的包类型
required-features =[]      # 构建对象所需的 Cargo Features
```

① https://github.com/bheisler/criterion.rs

② https://doc.rust-lang.org/cargo/reference/cargo-targets.html

1. name

对于库对象和默认的二进制对象(src/main.rs),默认的名称是项目的名称(package.name)。对于其他类型的对象,默认是目录或文件名。除了[lib]外,name 字段对于其他对象都是必需的。

2. edition

对使用的 Rust Edition 版本进行设置。如果没有设置,则默认使用[package]中配置的 package.edition。

3. crate-type

该字段定义了对象生成的包类型,它是一个数组,因此可以为同一个对象指定多个包类型。注意:只有库对象和示例对象可以被指定,而其他的二进制、测试和基准测试对象只能是 bin 包类型,才可执行二进制类型。包类型可用的选项包括 bin、lib、rlib、dylib、cdylib、staticlib 和 proc-macro。更多的可以参考官方的参考手册[①]。

4. required-features

该字段用于指定在构建对象时所需的 features 列表。该字段只对[[bin]]、[bench]]、[[test]]和[[example]]有效,对于[lib]没有任何效果。例如:

```
[features]
# ...
postgres = []
mysql = []
tools = []

[[bin]]
name = "my-tool"
required-features = ["postgres", "tools"]
```

3.6 package 表

package 表包含与本 package 相关的元数据信息。2.6 节中的 helloworld 例子的 Cargo.toml 的内容如下:

```
[package]
name = "helloworld"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018" # 指定 Rust 的 edition 版本
```

其中最重要的是 edition。可能的版本选项有 2015、2018 和 2021。这些版本的具体描述可以访问官方网站[②]。

一个比较复杂(包含更多的项目元数据)的例子如下:

```
[package]
name = "rustbox"
```

① https://doc.rust-lang.org/stable/reference/linkage.html

② https://doc.rust-lang.org/stable/edition-guide/

```
version ="0.9.0"
authors =[
    "Gavin Zheng <gavinzheng@gmail.com>"],
    "Milly Chou <milly.chou@gmail.com>",
    "Chloe Ross <orangesnowfox@gmail.com>",
    "Daniel Akhterov <akhterovd@gmail.com>",
]
description ="A rust implementation of the Hello library"
repository ="https://github.com/gavinzheng/rustbox"
documentation ="https://docs.rs/sp-runtime-interface-proc-macro"
homepage ="https://github.com/gavinzheng/rustbox"
readme ="README.md"
license ="MIT"
keywords =[
    "hello",
    "terminal",
    "gui",
]
categories =["network-programming","asynchronous"]
exclude =[
    "examples/*"
]
publish =false
autobins =false# 对象的自动发现设为 false, 以下同
autoexamples =false
autotests =false
autobenches =false
```

3.7　patch 表

patch 表允许暂时使用一个不同的依赖源，也就是依赖覆盖。依赖源可以在任何位置。这个功能在以下情况下非常有用：

- 程序员在测试一个错误修正版(Bug Fix)；
- 性能提升测试；
- 发布一个微小的版本时，暂时使用一个过渡依赖源。

下面演示了如何定义一个暂时的 patch 依赖源：

```
[patch.crates-io]
# 使用一个本地版本的依赖源
myrand={ path ="/home/milly/myrand" }        // 使用 myrand crate 时，使用本地的 myrand，而不是
                                             //crates-io
# 使用 Git 分支的修改版本，而不是用 crates-io 上的版本
ecdsa ={ git ="https://github.com/paritytech/ecdsa.git", branch ="latest" }

# patch 一个 Git 依赖源
[patch.'https://github.com/milly/rand.git']
project ={ path ="/home/milly/rand" }
```

即使使用了 patch 的依赖源，Cargo 仍然会检查 crate 的版本，以防止程序员使用了 crate 的错误主版本。如果程序员需要在过渡期间使用同一 crate 的多个主版本，即可以为每一个主版本依赖生成不同的标识。示例如下：

```
[patch.crates-io]
rand4 ={ path ="/home/milly/rand4", package ="rand" }
rand5 ={ path ="/home/milly/rand5", package ="rand" }
```

3.8 常用的 cargo 的命令

cargo 常用的命令如下。

项目生成与初始化

```
// 默认生成可执行项目(二进制 crate),如果想生成库包(Library crate),使用 --lib
cargo new crate_name [--bin]
cargo init [--bin]          // 默认初始化二进制可执行项目,如果想初始化库包,使用 --lib
```

项目编译

```
cargo build [--release]
cargo run [--release]
```

cargo build 命令的默认模式是编译一个没有优化的调试版本(Debug Version),输出的二进制执行文件在 target/debug 目录下。cargo run 运行的也是调试版本。-release 开关会进行优化的生产环境的编译,输出的二进制可执行文件在 target/release 目录下。测试版本和生产环境版本的编译区别是非常大的,而且优化编译会比较慢。

项目测试案例运行

```
cargo test
```

项目基准性能测试

```
cargo bench
```

项目依赖包更新(会更新 Cargo.lock 文件)

```
cargo update
```

Cargo 版本信息

```
cargo version
```

项目打包和发布到 crate.io

```
cargo package
cargo publish
```

删除编译过程中生成的中间文件

```
cargo clean
```

有时,我们需要对特定的对象平台生成与 Rust 程序相应的汇编代码,命令如下:

```
RUSTFLAGS="--emit asm" cargo build --target=x86_64-unknown-linux-gnu
```

3.9 扩展 cargo 命令

Rust 2021 Edition 新增了一些与 Cargo Module 相关的命令,可以帮助我们理解和管理模块。

cargo modules

```
cargo install cargo-modules                    // 安装 cargo-modules
cargo modules generate tree --with-types       // 生成项目的模块树
```

cargo Watch

```
cargo install cargo-watch      // 安装 cargo-watch
```

cargo-watch 监视源代码的改变，并在每次改变时触发特定命令，示例如下：

```
cargo watch -x check           // 每次代码修改都触发运行 cargo check,这将减少用户的编译时间
```

cargo-watch 同时支持命令链式调用：

```
cargo watch -x check -x test -x run
```

每次修改代码都会触发运行 cargo check。如果成功，将继续运行 cargo test。如果失败，将继续运行 cargo run。

cargo-audit

可以通过以下命令安装：

```
cargo install cargo-audit
```

一旦安装成功，就可以在 cargo 项目下运行该命令。该命令会检查 Cargo.toml 并且用 RustSec 安全数据库[①]扫描所有的依赖包，并打印所有的安全建议。

cargo-edit

cargo-edit 命令会自动向项目的 Cargo.toml 文件中增加依赖包，包括 dev 依赖包和 build 依赖包，并且可以指定特定的依赖包版本。可以通过下面的命令安装：

```
cargo install cargo-edit
```

它提供了 4 个子命令：cargo add、cargo rm、cargo edit 和 cargo upgrade。

cargo-deb

这个命令由社区开发，用来为 Rust 执行文件建立在 Debian Linux 上的发布版本(.deb)。可以通过下面的命令安装：

```
cargo install cargo-deb
```

cargo-outdated 和 cargo-duplicates

cargo outdated 命令显示在 cargo 项目中已经过期的 crate 包依赖；而 cargo duplicates 命令会在项目的依赖包中找到已升级或者已降级的包。可以通过下面的命令安装：

```
cargo install cargo-outdated
cargo install cargo-duplicates
```

一旦安装成功，就可以在 cargo 项目目录下运行该命令。

① https://rustsec.org/

cargo install

为了和 cargo 无缝连接,所有社区开发的子命令都遵循命名规则 cargo-[cmd]。当使用 cargo install 命令安装二进制 crate 时,cargo 会将安装的二进制文件加入 $PATH 环境变量,这样就可以自由地使用 cargo < cmd >命令了。

可以在 https://github.com/rust-lang/cargo/wiki/Third-party-cargo-subcommands 中找到社区开发的 cargo 扩展子命令。cargo install 也可以用来安装任何用 Rust 语言开发的二进制 crates 或者可执行文件/应用,它们默认安装在/home/< user >/.cargo/bin/目录下。

至于其他的 cargo 命令,如 cargo check、cargo [un]install binary_crate_name、cargo search crate_name、cargo login api_key、cargo crev、cargo-geiger、cargo-osha、cargo-deny 等高级命令的用法,请参考 cargo 官方文档[①]。

3.10 特征

features 是 Rust 用来定制化项目的主要工具。一个特征是一个编译标识,在编译 crate 时会传给其依赖包,从而添加可选的功能。通常,使用特征的方式主要有以下三种:

- 激活可选的依赖;
- 有条件地使用一个 crate 里的某些组件;
- 增强代码的行为。

1. 下游 crate 使用上游 crate 的特征

示例如下:

```
[package]
name ="foo"
...
[features]
derive =["syn"]

[dependencies]
syn ={ version ="1", optional =true}
```

当 cargo 编译 foo crate 时,默认它不会编译 syn crate。只有当一个依赖于 syn crate 的 crate(称为下游/downstream crate)需要使用 derive 特性提供的功能,并且显式地选择包含 syn 时,才会编译 sync。例如:

```
[package]
name ="bar"
...
[dependencies]
foo ={ version ="1", features =["derive"]}   // Cargo 会编译 syn crate
```

2. 选择设置默认特性

Cargo 允许我们为 crate 定义一组默认特征(这些特征可能被频繁使用)。同时还可以选择排除一些默认特征。下例演示了如何默认使用 derive 特征编译 syn,同时排除一些 syn 的

① https://doc.rust-lang.org/cargo/

默认特征，而仅仅包含为 derive 特征所需的那些特征。例如：

```
[package]
name ="foo"
…
[features]
derive =["syn"]
default =["derive"]                                    // 默认是 derive 特征，因为其被频繁使用

[dependencies.syn]
version ="1"
default-features =false                                // 排除默认特征
features =["derive","parsing","printing"]  // syn 只会被编译这三个特征
optional =true
```

3. 一个 feature 可以开启其他 feature

例如 image 图片包含 JPG 和 PNG 格式，因此当 image 被启用后，还得确保启用 JPG 和 PNG。例如：

```
[features]
jpg =[]
png =[]
image =["jpg","png"]                                   // 打开 image 特性就同时打开了 jpg 和 png
```

4. 默认 feature

默认情况下，所有的 feature 都会被自动禁用，可以通过 default 来启用它们。例如：

```
[features]
default =["image","mp3"]
jpg =[]
png =[]
image =["jpg","png"]
mp3 =[]
```

5. 可选依赖

当依赖被标记为“可选(optional)”时，意味着它默认不会被编译。假设我们需要用到一个外部的包来处理 TIF 格式的图片，例如：

```
[dependencies]
tif ={ version ="0.11.1", optional =true} # 默认不会被编译
```

我们可以通过显式定义 feature 的方式来启用这些可选依赖库，例如为了支持 OMNI 图像格式，我们需要引入两个依赖包：一个处理 RGB 格式，另一个处理 HSV 格式。OMNI 通过特性引入了两个可选格式特性：HSV 和 RGB。

```
[dependencies]
hsv ={ version ="0.5.0", optional =true}
rgb ={ version ="0.3.2", optional =true}
[features]
omni =["hsv","rgb"]
```

6. 依赖库自身的 feature

依赖库也可以定义自己的 feature，也有需要启用的 feature 列表。当引入该依赖库时，可

以通过以下方式为其启用相关的 features:

```
[dependencies]
serde ={ version ="1.0.1", features =["derive"]}
```

以上配置为 serde 依赖开启了 derive 特征。

可以通过下面的方式来禁用依赖库的默认特征:

```
[dependencies]
ffmpeg ={ version ="1.0.2", default-features =false, features =["libc"]}
```

上面的例子禁用了 ffmpeg 的默认特征,但是又启用了它的 libc 特征。注意:这种方式不一定能成功禁用 default,原因是可能会有其他依赖也引入了 ffmpeg,并且没有对 default 进行禁用。那样的话,默认特征仍然会被启用。

可以通过下面的方式来间接开启依赖库的 feature:

```
[dependencies]
mp4-decoder ={ version ="0.1.20", default-features =false}

[features]
parallel =["mp4-decoder/rayon"]# 打开 mp4-decoder 的并行特性开关"rayon"
```

上例中定义了一个 parallel 的特性,同时为其启用了 jpeg-decoder 依赖的 rayon 特性。

7. 检视已解析的 features

在复杂的特征依赖图中,如果想要了解不同的特征是如何被多个包多路启用的,可以使用 cargo tree 命令提供的几个选项来检视哪些特征被启用了。例如:

```
cargo tree -e features
```

推荐按照 crates.io 的方式来设置特征(Feature)名称:crate.io 要求名称只能由 ASCII 码字母数字、_、-或者+组成。本节只介绍如何在 Cargo.toml 中使用特征。至于如何在源代码中基于特征编程,需要用到 cfg!宏以及条件编译,请参照 8.1.2 节。

3.11 profile

profile 其实是一种发布配置,默认包含 4 种:dev、release、test 和 bench。正常情况下,我们无须指定,Cargo 会根据我们使用的命令来自动进行选择。

profile 可以通过 Cargo.toml 中的[profile]部分进行设置和改变。例如在 Cargo.toml 中的 profile 设置中,可以直接通过[profile.{profile}]的方式使用它们。又如可以将所有 dev 相关的配置放在[profile.dev]表下,将所有与 release 相关的配置放在[profile.release]表下。例如:

```
[profile.dev]
opt-level =1               # 使用稍高一些的优化级别,最低是 0,最高是 3
debug-assertions =true     # 打开调试断言
```

需要注意的是,每种 profile 都可以单独设置,例如上面的[profile.dev]。

[profile]表可以让程序员指定一些编译 crate 的开关,Cargo 会将这些开关传给 Rust 编

译器。这些开关主要有以下 3 类。

- **性能开关**。主要有 3 个开关：

  ➢ opt-level

这个开关告诉 Rust 编译器如何优化代码。可取的值为 0～3：0 是不要做任何优化，3 是能优化尽量优化。如果取值为 s，则这是优化编译结果文件的大小。嵌入式开发可能会关注到这个值。例如：

```
[profile.dev.package]
mandel ={ opt-level =3}   # 使用第三级的优化
```

opt-level 可能的取值如下。

❖ 0：无优化。
❖ 1：基本优化。
❖ 2：一些优化。
❖ 3：全部优化。
❖ "s"：优化输出的二进制文件的大小。
❖ "z"：优化二进制文件的大小，但也会关闭循环向量化。

  ➢ LTO(Link-Time Optimization)

顾名思义，LTO 是告诉编译器优化不同 crate 之间连接时的代码：每个编译结果单元都带有该单元的相关信息。在多个编译单元连接在一起时，Rust 编译器会利用所有这些信息来对连接后代码进行又一轮的优化。代价是编译时间的加长。

  ➢ codegen-units

codegen-units 是关于编译时效率的，它告诉编译器编译一个 crate 可以分成多少个独立的编译任务，从而可以并行编译，提高编译效率。

- **调试开关**

  [profile]表支持的开关包括 debug、debug-assertions 和 overflow-checks。

  ➢ debug 标志通知 Rust 编译器在编译后的二进制文件里包含调试信息，这样会增大结果文件的大小，但是程序异常的回溯信息中会包含函数名字和其他相关信息，而不是仅仅包含指令地址。

  ➢ debug-assertions 标识使用 debug_assert! 宏并编译 cfg(debug_assertions)标注的调试代码，这会导致代码效率下降，但是可以让程序员在运行时更精确地捕捉信息。

  ➢ overflow-checks 标志可以对整数操作进行溢出检查，这可能会降低效率，但是能帮助程序员更早地捕捉到问题所在。这些标志默认都会在 debug 模式中打开，而在 release 模式中关闭。

- **用户自定义的代码行为**

我们也可以对一个特定的依赖包或者特定的 profile 设定进行重载。下面的例子演示了如何为 serde crate 设置最高优先级的优化，而对其他 crate 进行次一级的优化。下例使用了[profile.<profile-name>.package.<crate-name>]语法。

```
[profile.dev.package.serde]
opt-level =3
[profile.dev.package."* "]
opt-level =2
```

表 3.2 总结了 cargo 命令会使用的 Cargo.toml 中的相关表。

表 3.2 Cargo 命令和 Cargo.toml 的表映射

命　　令	所使用的 Cargo.toml 的表
cargo build	[profile.debug]
cargo build --release	[profile.release]
cargo test	[profile.test]

下面的设定使用了 cargo build --release,编译后生成的二进制文件里包含了调试符号信息:

```
[profile.release]
debug =true # 在发布版本中包含调试符号信息
```

3.11.1 默认 profile

cargo 预定义了以下几个配置(profile)[①]。

- **dev**——开发配置项。cargo build、cargo rustc、cargo check 和 cargo run 默认使用 dev 配置。
- **release**——正式版本配置项,如 cargo install、cargo build --release、cargo run --release。
- **test**——测试配置项,如 cargo test。
- **bench**——性能测试配置项,如 cargo bench。
- **doc**——文档配置项,如 cargo doc。
- **使用自定义 profile**:cargo build --profile release-lto。

1. dev

dev profile 用于开发和调试。cargo build 或 cargo run 默认使用的就是 dev profile。cargo build --debug 也是。注意:dev profile 的结果并没有输出到 target/dev 的同名目录下,而是输出到 target/debug。

默认的 dev profile 设置如下:

```
[profile.dev]
opt-level =0
debug =true
split-debuginfo ='...'  # Platform-specific.
debug-assertions =true
overflow-checks =true
lto =false
panic ='unwind'
incremental =true
codegen-units =256
rpath =false
```

2. release

release 往往用于预发布(staging)/生产环境或性能测试,以下命令使用的就是 release profile:

① https://course.rs/cargo/reference/profiles.html

```
cargo build --release
cargo run --release
cargo install
```

默认的 release profile 设置如下：

```
[profile.release]
opt-level =3
debug =false
split-debuginfo ='...'  # Platform-specific.
debug-assertions =false
overflow-checks =false
lto =false
panic ='unwind'
incremental =false
codegen-units =16
rpath =false
```

3. test

该 profile 用于构建测试，它的设置是继承自 dev。

4. bench

该 profile 用于构建基准测试 benchmark，它的设置默认继承自 release。

3.11.2　自定义 profile

除了默认的四种 profile，还可以定义自己的 profile。自定义的 profile 可以帮助我们建立更灵活的工作发布流和构建模型。

当定义 profile 时，必须指定 inherits 用于说明当配置缺失时该 profile 要从哪个 profile 那里继承配置。例如，我们想在 release profile 的基础上增加 LTO 优化，那么可以在 Cargo.toml 中添加如下内容：

```
[profile.release-lto]
inherits ="release"     // 配置继承自 release
lto =true
```

然后在构建时使用--profile 来指定想要选择的自定义 profile：

```
$ cargo build --profile release-lto
```

3.11.3　重写 profile

可以对特定的包使用 profile 进行重写(override)。例如：

```
# `foo` package 将使用 -Copt-level=3 标志
[profile.dev.package.foo]
opt-level =3
```

这里的 package 名称实际上是一个 Package ID，因此可以通过版本号来选择。例如：

```
[profile.dev.package."foo:2.1.0"]
```

如果要为所有依赖包重写(不包括工作空间的成员)，例如：

```
[profile.dev.package."*"]
opt-level =2
```

为构建脚本、过程宏和它们的依赖重写为:

```
[profile.dev.build-override]
opt-level =3
```

重写的优先级按以下顺序执行(最先被匹配的优先级更高):

(1) [profile. dev. package. name],指定名称进行重写。

(2) [profile. dev. package. "*"],对所有非工作空间成员的 package 进行重写。

(3) [profile. dev. build-override],对构建脚本、过程宏及它们的依赖进行重写。

(4) [profile. dev]。

(5) Cargo 内置的默认值。

3.12 工作空间

在 Rust 中,可以使用 Cargo Workspace 来组织大型的 Rust 项目。Workspace 可能包含一个或者多个项目/包(package),这些项目(称为 Workspace 工作成员)被 Workspace 统一管理。

- Cargo 的通用命令适用于 Workspace 的所有成员,如 cargo check --workspace。
- 所有的 package **共享同一个位于 Workspace 根目录下的 Cargo. lock 文件**。
- 所有的 package **共享同一个输出目录**,默认位于 Workspace 根目下的 target 目录。
- 共享项目的元数据,如 workspace. package。
- Cargo. toml 中的[patch],[replace]和[profile. *] 节仅适用于根目录,不适用于成员项目。成员项目中的 Cargo. toml 中的相应部分将被自动忽略。

在 Cargo. toml 文件中,[workspace]表支持如下的配置,如表 3. 3 所示。

表 3. 3 Cargo. toml 中的 workspace 表

表 名	名 称	说 明	样 例 值
[workspace]	定义一个工作空间		
	resolver	指定使用的依赖解析器	见官方文档①
	members	工作空间的成员项目	
	exclude	工作空间不需要的项目指定	
	default-members	默认的操作项目	
	package	项目中继承的键	
	dependencies	在项目依赖中继承的键	
	metadata	外部工具的设置	
[patch]		被重载的依赖	
[replace]		被重载的依赖(过时作废)	
[profile]		编译器设置和优化配置	

工作空间的示意图如图 3. 1 所示。

① https://doc. rust-lang. org/edition-guide/rust-2021/default-cargo-resolver. htm

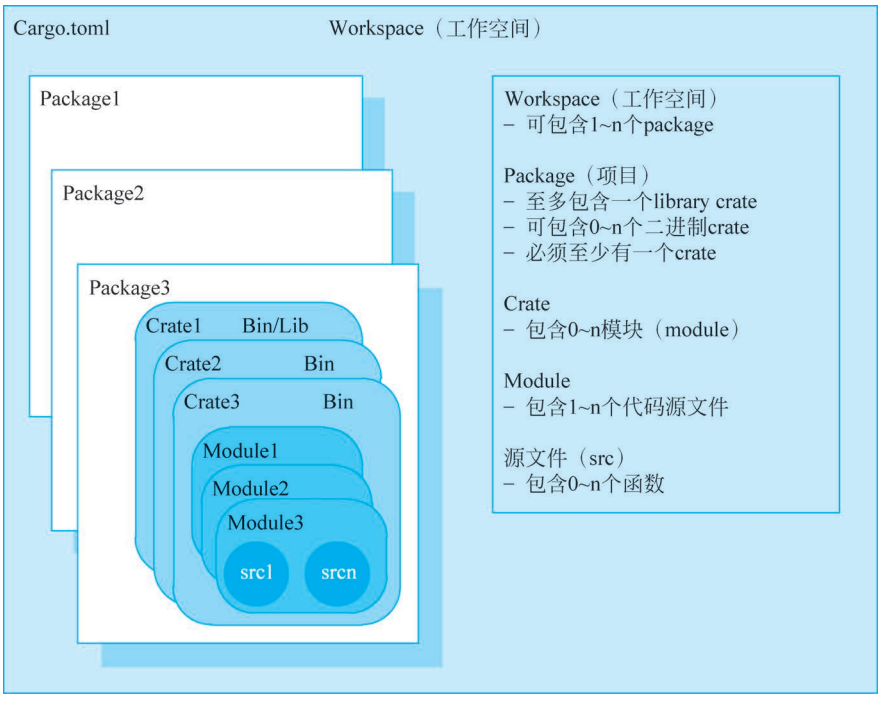

图 3.1　工作空间/Package/Crate/Module/源代码结构层次图

图 3.1 的详细解释如下。

- 一个 Rust 项目最小的独立单元是一个函数({...}代码块可视为函数的一种)。
- 一组函数被组织为一个具有特定文件名的源代码文件(如 main.rs)。
- 文件的上一级别的单元是模块。模块中的代码有唯一的名域空间。模块可以包含用户定义的数据类型(如 structs、traits 和 enums)、常量、类型别名、其他模块导入和函数声明。模块可能嵌套。多个模块可以定义在一个源文件里,或者一个模块的代码也可以散布在多个源代码文件中。
- 多个模块可以被组织为 crate。crates 可以用来在 Rust 项目中共享代码。一个 crate 要么是库,要么是二进制可执行文件。程序员开发并公开发布的 crate 可以被其他开发程序员和开发团队使用。crate 的根是 Rust 编译器开始的源代码。对于二进制 crate,crate 的根为 main.rs;对于库 crates,crate 的根为 lib.rs。
- 一个或者多个 crate 可以被组织为一个 package。一个 package 包含一个 Cargo.toml 配置文件,其中包含如何编译、下载以及链接 package 的信息。一个 package 必须包含至少一个 crate,可以是库,也可以是二进制可执行 crate。一个 package 可能包含任意一个二进制可执行 crates,但是只能包含最多一个库 crate。
- 随着 Rust 项目的增大,可能需要将一个 package 分拆为多个单元,并对它们进行单独管理。一组相关的 package 被组织为一个工作空间,一个工作空间中的一组 package 共享同样的 Cargo.lock 文件(包含工作空间里所有 package 之间共享的特定版本依赖包的详细信息)和输出目录。

工作空间有两种类型[①]:根项目(root package)和虚拟工作空间(virtual workspace)。

① https://doc.rust-lang.org/cargo/reference/workspaces.html

1. 根项目

若一个 package 的 Cargo. toml 在包含[package]的同时又包含[workspace]部分,则该 package 称为工作空间的根项目。换言之,一个工作空间的根(root)是该工作空间的 Cargo. toml 文件所在的目录。

例如:大名鼎鼎的以太坊的客户端 Parity[①] 就在最外层的 package 中定义了[workspace]。例如:

```
[package]
description ="Parity Ethereum client"
name ="parity"
version ="1.9.0"
license ="GPL-3.0"
authors =["Parity Technologies <admin@parity.io>"]
build ="build.rs"

[workspace]
members =[
    "ethstore/cli",
    "ethkey/cli",
    "evmbin",
    "whisper",
    "chainspec",
    "dapps/js-glue"
]
```

那么,最外层的目录就是 Parity 的工作空间的根。具体说明参见 3.16 节。

2. 虚拟工作空间

若一个 Cargo. toml 有[workspace],但是没有[package]部分,则它是虚拟工作空间。对于没有主 package 的场景或当我们希望将所有的 package 组织在单独的目录中时,这种方式非常适合。例如 rust-analyzer[②] 就是这样的项目,它的根目录中的 Cargo. toml 并没有[package],说明该根目录不是一个 package,但是却有[workspace]。例如:

```
[workspace]
members =["xtask/","lib/* ","crates/* "]
exclude =["crates/proc-macro-test/imp"]
resolver ="2"

[workspace.package]
rust-version ="1.70"
edition ="2021"
license ="MIT OR Apache-2.0"
authors =["rust-analyzer team"]
```

结合 rust-analyzer 的目录布局可以看出,该工作空间的所有成员 package 都在单独的目录中。

Cargo. toml 中的[workspace]部分用于定义哪些项目属于工作空间的成员项目。例如:

---

① https://parity.io/

② https://github.com/rust-lang/rust-analyzer

```
[workspace]
members = ["member1", "path/to/member2", "lib/* "]
exclude = ["lib/foo", "path/to/other"]
```

若某个本地依赖包是通过 path 引入的，并且该包位于工作空间的目录中，则该包自动成为工作空间的成员。剩余的成员需要通过 workspace. members 来指定，里面包含各个成员项目所在的目录(成员目录中包含 Cargo. toml)。members 还支持匹配多个路径。例如，上面的例子使用 lib/ * 匹配 lib 目录下的所有包。exclude 可以将指定的目录排除在工作空间之外。例如上面的例子，lib/ * 在包含 lib 目录下的所有包后，又通过 exclude 中的 lib/foo 将 lib 下的 foo 目录排除在外。

3. 选择工作空间

选择工作空间有两种方式：Cargo 自动查找或者手动指定 package. workspace 字段。当位于工作空间的子目录中时，Cargo 会自动在该目录的父目录中寻找带有[workspace]定义的 Cargo. toml，然后再决定使用哪个工作空间。

还可以覆盖 Cargo 的自动查找功能：将成员包中的 package. workspace 字段修改为工作区间根目录的位置，这样就能显式地让一个成员使用指定的工作空间。当成员不在工作空间的子目录下时，这种手动选择工作空间的方法非常适用。因为 Cargo 的自动搜索是沿着父目录往上查找的，而成员并不在工作空间的子目录下，所以顺着成员的父目录往上找是无法找到该工作空间的 Cargo. toml 的，此时就只能手动指定了。

4. 选择项目

在工作空间中，与 package 相关的 Cargo 命令(如 cargo build)可以使用-p、--package 或者--workspace 命令行参数来指定想要操作的 package。若没有指定任何参数，则 Cargo 将使用当前工作目录中的项目。若工作目录是虚拟工作空间类型，则该命令将作用在所有成员项目上(就好像是使用了--workspace 命令行参数)。而 default-members 可以在命令行参数没有被提供时手动指定操作的成员。例如：

```
[workspace]
members = ["path/to/member1", "path/to/member2", "path/to/member3/* "]
default-members = ["path/to/member2", "path/to/member3/foo"]
```

使用 default-members，cargo build 就不会应用到虚拟工作空间的所有成员项目上，而是仅应用到指定的成员项目上。

5. workspace. metadata

workspace. metadata 与 package. metadata 非常类似，会被 Cargo 自动忽略，就算没有使用也不会发出警告。这个部分可以用于在 Cargo. toml 中存储一些工作空间的配置元数据信息。例如：

```
[workspace]
members = ["member1", "member2"]
[workspace.metadata.webrequests]
root = "path/to/webproject"
tool = ["npm", "ruby", "build"]
# ...
```

3.13 Cargo 的使用

3.13.1 Cargo 系统目录

Cargo 的配置目录位于:

Ubuntu

```
$ HOME/.cargo/
```

Windows

```
C:\Users\Administrator\.cargo
```

可以通过修改 CARGO_HOME 环境变量的方式来重新设定该目录的位置。该目录内容如图 3.2 所示。

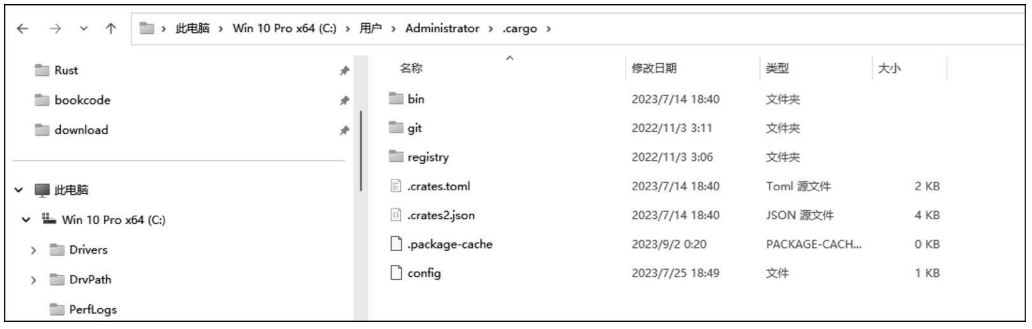

图 3.2 Cargo 安装目录图

具体说明如下。

- config.toml 是 Cargo 的全局配置文件。具体内容格式见 3.14 节。
- config 定义 crate 的注册中心:下载源/镜像源。
- .crates.toml 和.crates2.json 是隐藏文件,请不要手动修改。
- bin 目录包含通过 cargo install 或 rustup 下载的包编译出的可执行文件。我们将该目录加入 $PATH 环境变量中,就可以实现对这些可执行文件的直接访问。
- git 目录中存储了 Git 的资源文件。
  - git/db,当一个包依赖某个 Git 仓库时,Cargo 会将该仓库克隆到 git/db 目录下,如果未来需要,还会对其进行更新。
  - git/checkouts,若指定了 Git 源和 commit,相应的仓库就会从 git/db 中 checkout 到该目录下,因此同一个仓库的不同 checkout 共存成为了可能。
- registry 包含注册中心的元数据和 packages。
  - registry/index 是一个 Git 仓库,包含注册中心中所有可用包的元数据(版本、依赖等)。
  - registry/cache 中保存了已下载的依赖,这些依赖包以 gzip 的压缩档案形式保存,后缀名为 crate。
  - registry/src,若一个已下载的 crate 档案被一个 package 需要,该档案会被解压缩到 registry/src 文件夹下,最终 rustc 可以在其中找到所需的 rs 文件。

3.13.2　Cargo 清除缓存

Cargo 命令的缓存罗列如下。理论上，我们可以手动移除缓存中的任何一部分，当后续有包需要时，Cargo 会尽可能地恢复这些资源。

- 解压缩 registry/cache 下的 crate 档案。
- 从 git 中 checkout 缓存的仓库。

我们可以使用 cargo-cache[①] 包来选择性地清除 cache 中指定的部分。

3.13.3　构建时卡住

在开发过程中，我们会碰到构建时卡住的问题。当 Vscode 重新下载依赖（这个过程可能还会更新 crates.io 使用的索引列表）的过程中，Cargo 会将相关信息写入 $HOME/.cargo/.package_cache 下，并将其锁住。如果在此同时，我们试图在另一个地方对同一个项目进行构建，就会报错 Blocking waiting for file lock on package cache。解决办法是：强行停止正在构建的进程。例如终止 Vscode 使用的 rust-analyer 插件进程，并删除 $HOME/.cargo/.package_cache 文件。

3.13.4　target 目录结构

cargo build 的结果会放入项目根目录下的 target 文件夹。这个位置可以通过以下三种方式更改：

- 设置 CARGO_TARGET_DIR 环境变量；
- build.target-dir 配置项；
- --target-dir 命令行参数。

target 目录结构及其说明如表 3.4 所示。

表 3.4　target 目录结构及其说明

目　　录	说　　明
target/debug/	➢ 包含 dev profile 的构建输出（cargo build 或 cargo build --debug） ➢ dev 和 test profile 的构建结果都存放在 debug 目录下 ➢ 包含编译后的输出，例如二进制可执行文件、库对象（library target）
target/debug/examples/	包含示例对象（example target）
target/debug/deps	依赖和其他输出成果
target/debug/incremental	rustc 增量编译的输出，该缓存可以用于提升后续的编译速度
target/debug/build/	构建脚本（build.rs）的输出
target/release/	➢ releaseprofile 的构建输出，cargo build --release ➢ release 和 bench profile 存放在 release 目录下
target/foo/	➢ 自定义 foo profile 的构建输出，cargo build --profile=foo ➢ 用户定义的 profile 存放在同名的目录下
target/<triple>/debug/	target/thumbv6m-none-eabi/debug/
target/<triple>/release/	target/thumbv6m-none-eabi/release/
target/doc/	包含通过 cargo doc 生成的文档
target/package/	包含 cargo package 或 cargo publish 生成的输出

① https://crates.io/crates/cargo-cache

3.14 config.toml 进行 Cargo 配置

本节讲述如何对 Cargo 相关工具进行配置。我们既可以在全局中设置默认的配置,又可以为每个包设定独立的配置,甚至还能进行版本控制。

1. 层级结构

我们可以对 Cargo 进行全局配置:$HOME/.cargo/config.toml。实际上,我们还可以在一个项目内进行配置。编译时,Cargo 的匹配规则是:Cargo 会顺着当前目录往上查找,直到找到目标配置文件。例如在目录/projects/aaa/bbb/ccc 下调用 Cargo 命令,则查找路径如下:

(1) /projects/aaa/bbb/ccc/.cargo/config.toml

(2) /projects/aaa/bbb/.cargo/config.toml

(3) /projects/aaa/.cargo/config.toml

(4) /projects/.cargo/config.toml

(5) /.cargo/config.toml

(6) $CARGO_HOME/config.toml 默认是

- Windows:%USERPROFILE%\.cargo\config.toml
- UNIX:$HOME/.cargo/config.toml

如果一个键(key)在多个配置中出现,那么这些键只会保留一个:最靠近 Cargo 执行目录的配置文件中的键值将被使用((1)优先级最高,(6)优先级最低)。需要注意的是,如果键的值是数组,那么相应的值将被合并。

对于工作空间而言,Cargo 的搜索策略是从根开始的,内部成员中包含的 Cargo.toml 会自动忽略。假如一个工作空间拥有两个成员项目,那么每个成员都有配置文件:

- /projects/aaa/bbb/ccc/mylib/.cargo/config.toml
- /projects/aaa/bbb/ccc/mybin/.cargo/config.toml

但是 Cargo 并不会读取它们,而是从工作空间的根(/projects/aaa/bbb/ccc/)开始往上查找。

2. 配置文件概览

下面是一个完整的配置文件[①],内容如下:

```
paths = ["/path/to/override"]          # 覆盖 Cargo.toml 中通过 path 引入的本地依赖

[alias]                                # 命令别名
b = "build"                            # cargo b 等同于 cargo build
c = "check"                            # cargo c 等同于 cargo check
t = "test"                             # cargo t 等同于 cargo test
r = "run"                              # cargo r 等同于 cargo run
rr = "run --release"                   # cargo rr 等同于 cargo run --release
space_example = ["run","--release","--","\"command list\""]

[build]
jobs = 1                               # 并行构建任务的数量,默认等于 CPU 的核心数
```

① https://course.rs/cargo/reference/configuration.html

```
rustc ="rustc"                          # Rust 编译器
rustc-wrapper ="…"                      # 使用该 wrapper 来替代 rustc
rustc-workspace-wrapper ="…"            # 为工作空间的成员,使用该 wrapper 来替代 rustc
rustdoc ="rustdoc"                      # 文档生成工具
target ="triple"                        # 为 target triple 构建(cargo install 会忽略该选项)
target-dir ="target"                    # 存放编译输出结果的目录
rustflags =["…","…"]                    # 自定义 flags,会传递给所有的编译器命令调用
rustdocflags =["…","…"]                 # 自定义 flags,传递给 rustdoc
incremental =true                       # 是否开启增量编译
dep-info-basedir ="…"                   # path for the base directory for targets in depfiles
pipelining =true                        # rustc pipelining

[doc]
browser ="chromium"                     # cargo doc --open 使用的浏览器
                                        # 可以通过 BROWSER 环境变量进行重写
[env]
# Set ENV_VAR_NAME=value for any process run by Cargo
ENV_VAR_NAME ="value"
# Set even if already present in environment
ENV_VAR_NAME_2 ={ value ="value", force =true}
# Value is relative to .cargo directory containing `config.toml`, make absolute
ENV_VAR_NAME_3 ={ value ="relative/path", relative =true}

[cargo-new]
vcs ="none"                             # 使用的 VCS  ('git', 'hg', 'pijul', 'fossil', 'none')

[http]
debug =false                            # HTTP debugging
proxy ="host:port"                      # HTTP 代理,libcurl 格式
ssl-version ="tlsv1.3"                  # TLS version to use
ssl-version.max ="tlsv1.3"              # 最高支持的 TLS 版本
ssl-version.min ="tlsv1.1"              # 最小支持的 TLS 版本
timeout =30                             # HTTP 请求的超时时间,单位为秒
low-speed-limit =10                     # 网络超时阈值(bytes/sec)
cainfo ="cert.pem"                      # path to Certificate Authority(CA) bundle
check-revoke =true                      # check for SSL certificate revocation
multiplexing =true                      # HTTP/2 multiplexing
user-agent ="…"                         # the user-agent header

[install]
root ="/some/path"                      # cargo install 安装到的目标目录

[net]
retry =2                                # 网络重试次数
git-fetch-with-cli =true                # 是否使用 git 命令来执行 git 操作
offline =true                           # 不能访问网络
[patch.<registry>]
# Same keys as for [patch] in Cargo.toml

[profile.<name>]                        # profile 配置
opt-level =0
debug =true
split-debuginfo ='...'
debug-assertions =true
overflow-checks =true
lto =false
panic ='unwind'
incremental =true
```

```
codegen-units = 16
rpath = false

[profile.<name>.build-override]
[profile.<name>.package.<name>]

[registries.<name>]              # 设置其他注册服务
index = "..."                    # 注册服务索引列表的 URL
token = "..."                    # 连接注册服务所需的鉴权 token

[registry]
default = "..."                  # 默认的注册服务名称: crates.io
token = "..."

[source.<name>]                  # 注册服务源和替换 source definition and replacement
replace-with = "..."             # 使用给定的 source 来替换当前的 source,例如使用中科大源来替换
                                 # crates.io 源以提升国内的下载速度: [source.crates-io] replace-
                                 # with = 'ustc
'directory = "..."               # path to a directory source
registry = "..."                 # 注册源的 URL ,例如中科大源: [source.ustc] registry = "git://
                                 # mirrors.ustc.edu.cn/crates.io-index"
local-registry = "..."           # path to a local registry source
git = "..."                      # URL of a git repository source
branch = "..."                   # branch name for the git repository
tag = "..."                      # tag name for the git repository
rev = "..."                      # revision for the git repository

[target.<triple>]
linker = "..."                   # linker to use
runner = "..."                   # wrapper to run executables
rustflags = ["...","..."]        # custom flags for `rustc`

[target.<cfg>]
runner = "..."                   # wrapper to run executables
rustflags = ["...","..."]        # custom flags for `rustc`

[target.<triple>.<links>]        # `links` build script override
rustc-link-lib = ["foo"]
rustc-link-search = ["/path/to/foo"]
rustc-flags = ["-L","/some/path"]
rustc-cfg = ['key="value"']
rustc-env = {key = "value"}
rustc-cdylib-link-arg = ["..."]
metadata_key1 = "value"
metadata_key2 = "value"

[term]
verbose = false                  # whether cargo provides verbose output
color = 'auto'                   # whether cargo colorizes output
progress.when = 'auto'           # whether cargo shows progress bar
progress.width = 80              # width of progress bar
```

3. 环境变量

除了 config.toml 配置文件以外,我们还可以使用环境变量对 Cargo 进行配置。配置文件中的键 foo.bar 对应的环境变量形式为 CARGO_FOO_BAR,其中的“.”“-”被转换成“_”,且字母都变成了大写的。例如,键名 target.x86_64-unknown-linux-gnu.runner 转换成环境变量后变成 CARGO_TARGET_X86_64_UNKNOWN_LINUX_GNU_RUNNER。环境变量的优先级比配置文件更高。除了上面的机制,Cargo 还支持一些预定义的环境变量。

下面介绍如何解决多次声明同一 crate 的不同版本时引发的冲突。例如:

```
…
[dependencies]
rand="=0.8"
ark-std={version="=0.3.0", features=["print-trace"]}
serde={version="=1.0", default-features=false, features=["derive"]}
serde_json="=1.0"
log="=0.4"
env_logger="=0.10"
clap={version="=4.1", features=["derive"]}
clap-num="=1.0.2"

#halo2
halo2_proofs={git="https://github.com/privacy-scaling-explorations/halo2.git", tag="v2023_02_02"}

#Axiom's helper API with basic functions
halo2-base={git="https://github.com/axiom-crypto/halo2-lib", tag="v0.3.0-ce"}
#Axiom poseidon chip (adapted from Scroll)
poseidon={git="https://github.com/axiom-crypto/halo2-lib", tag="v0.3.0-ce"}
#Axiom Evm wrapper
#These are just for making proving executables, if you are just building a library you don't need them as dependencies in your project
axiom-eth={git="https://github.com/axiom-crypto/axiom-eth.git", branch="community-edition", default-features=false, features=["halo2-axiom","aggregation","evm","clap"]}
…
```

编译的错误信息如下：

```
    Updating git repository `https://github.com/privacy-scaling-explorations/halo2curves`
error: failed to select a version for `clap`.
    ... required by package `halo2-scaffold v0.2.0 (D:\dev\rust\zero\halo2\halo2-scaffold-main)`
versions that meet the requirements `= 4.1` are: 4.1.14, 4.1.13, 4.1.12, 4.1.11, 4.1.10, 4.1.9, 4.1.8, 4.1.7, 4.1.6, 4.1.5, 4.1.4, 4.1.3, 4.1.2, 4.1.1, 4.1.0

all possible versions conflict with previously selected packages.

  previously selected package `clap v4.0.13`
    ... which satisfies dependency `clap = "= 4.0.13"` of package `axiom-eth v0.2.1 (https://github.com/axiom-crypto/axiom-eth.git?branch=community-edition#014a2d05)`
    ... which satisfies git dependency `axiom-eth` of package `halo2-scaffold v0.2.0 (D:\dev\rust\zero\halo2\halo2-scaffold-main)`

failed to select a version for `clap` which could resolve this conflict
```

解决方案如下：

```
…
[dependencies]
rand="= 0.8"
ark-std={version="= 0.3.0", features=["print-trace"]}
serde={version="= 1.0", default-features=false, features=["derive"]}
serde_json="= 1.0"
log="= 0.4"
env_logger="= 0.10"
clap={version="= 4", features=["derive"]}
clap-num="= 1.0.2"
```

3.15　构建脚本

一些项目希望编译第三方的非 Rust 代码，例如 C 依赖库；一些项目希望链接本地或者基于源码构建的 C 依赖库；还有一些项目需要功能性的工具，例如在构建之间执行一些代码生

成的工作等。对于这些目标,社区已经提供了一些工具,Cargo 并不想替代它们,但是为了给用户带来一些便利,Cargo 提供了自定义构建脚本的方式,以帮助用户更好地解决类似的问题。

3.15.1 build-dependencies

我们可以指定某些依赖仅用于构建脚本。例如:

```
[build-dependencies]
cc ="1.0.3"
```

平台特定的依赖仍然可以使用。例如:

```
[target.'cfg(unix)'.build-dependencies]
cc ="1.0.3"
```

需要注意的是,构建脚本(build.rs)和项目的正常代码是彼此独立的,因此它们的依赖不能互通:构建脚本无法使用[dependencies]或[dev-dependencies]中的依赖,而[build-dependencies]中的依赖也无法被构建脚本之外的代码所使用。

3.15.2 build.rs

若要创建构建脚本,我们只需要在项目的根目录下添加一个 build.rs 文件即可。Cargo 会先编译和执行该构建脚本,然后再去构建整个项目。以下是一个非常简单的脚本示例[①]:

```
fn main() {
    // 以下代码告诉 Cargo,一旦指定的文件 src/hello.c 发生了改变,就重新运行当前的构建脚本
    println!("cargo:rerun-if-changed=src/hello.c");
    // 使用 cc 来构建一个 C 文件,然后进行静态链接
    cc::Build::new()
        .file("src/hello.c")
        .compile("hello");
}
```

构建脚本的一些使用场景如下:

- 构建 C 依赖库;
- 在操作系统中寻找指定的 C 依赖库;
- 根据某个说明描述文件生成一个 Rust 模块;
- 执行一些与平台相关的配置。

下面我们来看看构建脚本具体是如何工作的。下一个章节还提供了一些关于如何编写构建脚本的示例。

3.15.3 构建脚本的生命周期

在项目被构建之前,Cargo 会将构建脚本编译成一个可执行文件,然后运行该文件并执行相应的任务。在运行过程中,脚本可以通过打印以“cargo:”开头的格式化字符串到标准输出来和 Cargo 通信。

Cargo 只有在当脚本源文件或其依赖包发生变化时才会重新编译。默认情况下,任何文

① https://doc.rust-lang.org/cargo/reference/build-scripts.html

件变化都会触发重新编译，如果我们希望对其进行定制，可以使用 rerun-if 命令。

在构建脚本成功执行后，项目就会开始进行编译。如果构建脚本的运行过程中发生错误，脚本应该返回一个非 0 码并立刻退出，在这种情况下，构建脚本的输出会被打印到终端中。我们可以通过环境变量给构建脚本提供一些输入值。

3.15.4　构建脚本的输出

构建脚本如果生成文件，那么这些文件需要放在统一的目录中。该目录可以通过**“OUT_DIR 环境变量”**来指定，**构建脚本不应该修改该目录之外的任何文件**。

之前提到过，构建脚本可以通过打印格式化字符串输出和 Cargo 进行通信：Cargo 会将每行带有“cargo:”前缀的输出解析为一条指令，其他输出内容会被自动忽略。

构建脚本的输出在构建过程中默认是隐藏的，如果想要在终端中看到这些内容，我们可以使用-vv 来调用以下 build.rs。例如：

```
// chapter3\helloworld\build.rs
use std::env;
use std::fs;
use std::path::Path;

fn main() {
    let out_dir = env::var_os("OUT_DIR").unwrap();
    let dest_path = Path::new(&out_dir).join("hello.rs");
    fs::write(
        &dest_path,
        "pub fn message() ->&'static str{
            \"Hello, 北科信链!\"
        }
        "
    ).unwrap();
    println!("cargo:rerun-if-changed=build.rs");
}
```

程序清单 3.1　build.rs 的例子

输入下面的命令，将输出：

```
Rust Programming\sourcecode\chapter3\helloworld>cargo run -vv
       Dirty helloworld v0.1.0 (E:\projects\Rust Programming\sourcecode\chapter3\helloworld): the
dependency build_script_build was rebuilt (13338329108.781659000s, 57s after last build at
13338329051.690492200s)
   Compiling helloworld v0.1.0 (E:\projects\Rust Programming\sourcecode\chapter3\helloworld)
...
    Finished dev [unoptimized + debuginfo] target(s) in 0.26s
     Running `target\debug\helloworld.exe`
Hello, 北科信链!
```

构建脚本打印到标准输出 stdout 的所有内容将保存在文件 target/debug/build/<pkg>/output 中(OUT_DIR 的具体位置取决于配置)，标准错误 stderr 的输出内容也将保存在同一个目录中。程序清单 3.1 中 build.rs 生成的中间文件 hello.rs 的内容如下：

```
// chapter3\helloworld\target\debug\build\helloworld-91696b90d115f9d4\out\hello.rs
pub fn message() ->&'static str{
           "Hello, 北科信链!"
       }
```

表 3.5[①] 列出了 Cargo 能识别的通信指令及简介。如果希望深入了解每个命令,可以通过具体的链接查看官方文档的说明。

表 3.5 Cargo 识别的通信指令列表

命 令	解 释
cargo: rerun-if-changed=PATH	当指定路径的文件发生变化时,Cargo 会重新运行脚本
cargo: rerun-if-env-changed=VAR	当指定的环境变量发生变化时,Cargo 会重新运行脚本
cargo: rustc-link-arg=FLAG	将自定义的 flags 传给 linker,用于后续的基准性能测试 benchmark、可执行文件 binary、cdylib 包、示例和测试
cargo: rustc-link-arg-bin=BIN=FLAG	将自定义的 flags 传给 linker,用于可执行文件 BIN
cargo: rustc-link-arg-bins=FLAG	将自定义的 flags 传给 linker,用于可执行文件
cargo: rustc-link-arg-tests=FLAG	将自定义的 flags 传给 linker,用于测试
cargo: rustc-link-arg-examples=FLAG	将自定义的 flags 传给 linker,用于示例
cargo: rustc-link-arg-benches=FLAG	将自定义的 flags 传给 linker,用于基准性能测试 benchmark
cargo: rustc-cdylib-link-arg=FLAG	将自定义的 flags 传给 linker,用于 cdylib 包
cargo: rustc-link-lib=[KIND=]NAME	告知 Cargo 通过-l 去链接一个指定的库,往往用于通过 FFI 链接一个本地库
cargo: rustc-link-search=[KIND=]PATH	告知 Cargo 通过-L 将一个目录添加到依赖库的搜索路径中
cargo: rustc-flags=FLAGS	将特定的 flags 传给编译器
cargo: rustc-cfg=KEY[="VALUE"]	开启编译时 cfg 设置
cargo: rustc-env=VAR=VALUE	设置一个环境变量
cargo: warning=MESSAGE	在终端打印一条 warning 信息
cargo: KEY=VALUE	links 脚本使用的元数据

3.15.5 构建脚本的依赖

构建脚本也可以引入其他基于 Cargo 的依赖包,只需要在 Cargo.toml 中添加或修改以下内容:

```
[build-dependencies]
cc ="1.0.46"
```

需要这么配置的原因在于构建脚本无法使用通过[dependencies]或[dev-dependencies]引入的依赖包,因为构建脚本的编译运行过程和项目本身的编译过程是分离的,且前者先于后者发生。同样地,我们的项目也无法使用[build-dependencies]中的依赖包。

在引入依赖时,需要仔细考虑它会给编译时间、开源协议和维护性等方面带来什么样的影响。如果我们在[build-dependencies]和[dependencies]引入了同样的包,这种情况下,Cargo 也许会对依赖进行复用,也许不会。如果不能复用,那么编译速度自然会受到不小的影响。

假设 A 项目的构建脚本生成了任意数量的键值对形式的元数据,那么这些元数据会被传递给依赖 A 的依赖包项目的构建脚本。例如,如果包 bar 依赖于 foo,当 foo 生成 key=value 形式的构建脚本元数据时,那么 bar 的构建脚本就可以通过环境变量的形式使用该元数据:DEP_FOO_KEY=value。需要注意的是,该元数据只能传给直接相关的。对于间接相关的,例如依赖的依赖,就无能为力了。

① https://doc.rust-lang.org/cargo/reference/build-scripts.html

3.15.6　覆盖构建脚本

当 Cargo.toml 设置了 links 时，Cargo 就允许我们使用自定义库对现有的构建脚本进行覆盖。例如：

```
[package]
# ...
links = "foo"
```

上面的例子声明了本 package 链接到 libfoo 本地库。如果使用了 links 键，则该 package 一定有构建脚本，而且构建脚本使用 rustc-link-lib 指令链接库。Cargo 要求一个本地库最多只能被一个项目链接，换而言之，我们无法让两个项目链接到同一个本地库。这个规则有助于防止 crate 之间的符号重复。

在 Cargo 的任何可被接受的 config.toml 配置文件中添加以下内容[①]：

```
[target.x86_64-unknown-linux-gnu.foo]
rustc-link-lib = ["foo"]
rustc-link-search = ["/path/to/foo"]
rustc-flags = "-L /some/path"
rustc-cfg = ['key="value"']
rustc-env = {key = "value"}
rustc-cdylib-link-arg = ["..."]
metadata_key1 = "value" metadata_key2 = "value"
```

增加这个配置后，一旦我们的某个项目声明了它要链接到 foo，那么项目的构建脚本将不会被编译和运行，取而代之的是这里的配置将被使用。warning、rerun-if-changed 和 rerun-if-env-changed 这三个键在这里不应使用，就算用了也会被忽略。

3.16　如何组织 Rust 项目

有效地项目组织对于维护干净、可维护和可扩展的 Rust 代码库至关重要。组织良好的项目可以增强协作、简化调试并促进未来的开发。本节将深入研究组织 Rust 项目的最佳实践，涵盖文件和文件夹结构、模块、板条箱、工作区、工具、约定和持续集成等主题。

3.16.1　Rust 中项目组织的重要性

有效地组织 Rust 项目具有许多优点。

- 提高代码可读性：组织良好的项目结构可以使开发人员更容易浏览和理解代码库，从而提高生产力。
- 增强的可维护性：适当的组织可以帮助开发人员更轻松地进行维护和重构，从而减少未来更改所需的时间和精力。
- 有效的协作：一致且结构良好的项目布局有助于团队成员之间的协作，使他们能够无缝地协同工作。
- 简化调试：清晰且有组织的项目结构可以帮助开发人员更轻松地识别和修复错误，从

① https://doc.rust-lang.org/cargo/reference/build-scripts.html

而减少调试时间。

- 可扩展性：组织良好的项目更具可扩展性，可以适应未来的增长和扩展，而不会变得难以管理。
- 简化的文档：清晰且有组织的项目结构使编写和维护文档变得更加容易，从而提高了代码库的可访问性和理解性。
- 增强的可测试性：组织良好的项目可以帮助开发人员更轻松地编写和维护测试，从而确保代码库的可靠性和正确性。

3.16.2 模块、crate 和工作空间

了解 Rust 项目组织的核心概念对于创建结构良好的代码库至关重要。

- 模块(Module)：模块提供了一种将相关代码分组在一起的方法，促进了代码组织和封装。
- crate(也被称为编译单元/Compile Unit，包/Package 或者项目/Project)：crate 是可重用的库或可执行文件，可以在项目之间共享。
- 工作空间：工作空间允许集中管理单个项目中的多个 crate，从而促进共享依赖项和代码。

1. 文件和文件夹结构

组织良好的 Rust 项目始于结构良好的文件和文件夹布局。我们选择的结构将取决于项目的规模和复杂性。然而，许多 Rust 项目都使用以下常见的结构。

- 单 crate 结构：这是最简单的结构，适合由单个 crate 组成的小型项目。在此结构中，项目的所有代码都包含在一个目录中，通常名为 src。
- 多 crate 结构：此结构用于分为多个 crate 的大型项目。每个 crate 都是一个独立的代码单元，可以独立编译和测试。包可以组织到项目根目录的子目录中。
- 工作空间结构：工作空间是一起管理的 crate 的集合。此结构用于由多个相关 crate 组成的项目。工作空间允许我们在 crate 之间共享依赖项和代码，并且它们可以使构建和测试整个项目变得更加容易。

2. 文件和文件夹命名的最佳实践

在命名 Rust 项目中的文件和文件夹时，遵循以下最佳实践非常重要。

- 使用描述性且一致的名称：选择能够清楚描述文件或文件夹内容的名称，避免使用模糊或不明确的名称。
- 避免使用长且不明确的名称：保持文件和文件夹名称简短并切中要点；长名称可能难以阅读和记住。
- 使用一致的命名约定：在整个项目中使用一致的命名约定，这将使其他开发人员更容易理解我们的代码。

例如，可以使用以下命名约定：

- 使用 snake_case(蛇形命名)作为文件和文件夹名称；
- 使用 PascalCase(帕斯卡命名)作为类和结构名称；
- 使用 camelCase(驼峰命名)作为函数和变量名称。

通过遵循这些最佳实践，我们可以创建易于导航和理解的文件与文件夹结构。Rust 项目

被组织成模块和 crate，它们是代码组织和重用的基本构建块。理解这些概念对于有效构建 Rust 项目至关重要。

3.16.3　模块：代码组织的逻辑单元

模块是 crate 内代码组织的逻辑单元，它们允许程序员将相关代码组合在一起，从而更轻松地管理和维护项目。模块可以包含函数、结构、枚举、特征和其他 Rust 语言元素。要创建模块，请使用 mod 关键字，后跟模块名称。例如：

```
mod my_module{
    // 此处是代码
}
```

可以使用 use 关键字访问模块中的内容。例如，要从 mon_module 模块访问 mon_function 函数，可以编写如下：

```
use mon_module::mon_function;
// 我们可以调用`mon_function`
function mon_function();
```

3.16.4　crate：可重用的库或可执行文件

crate 是 Rust 中可重用的库或可执行文件，它可以包含一个或多个模块，是 Rust 生态系统中分发和重用的基本单元。crate 可以是二进制 crate(用于创建可执行文件)或库 crate(用于创建可重用库)。要创建 crate，需要在项目的根目录中定义一个 Cargo.toml 文件。Cargo.toml 文件包含有关 crate 的元数据，例如其名称、版本、依赖项和其他设置。

1. 创建和管理模块

要创建模块，请使用 mod 关键字，后跟模块名称。例如：

```
mod mon_module{
    // 此处是代码
}
```

可以将模块嵌套在其他模块中以创建层次结构。例如：

```
mod outer_module{
    mod inner_module{
        // 此处是代码
    }
}
```

2. 嵌套模块

可以将模块嵌套在其他模块中以创建层次结构，这对于将大型项目组织成更小、更易于管理的单元非常有用。例如，我们可以为应用程序的每个功能创建一个模块，然后将模块嵌套在这些模块中以用于每个功能的不同组件。例如：

```
mod mon_app{
    mod feature_1{
        mod component_1{
```

```
            // component 1 代码
        }
        mod component_2{
            // component 2 代码
        }
    }
    mod feature_2{
        mod component_3{
            // component 3 代码
        }
        mod component_d{
            // component 4 代码
        }
    }
}
```

3. 组织模块以提高清晰度和可维护性

在组织模块时,考虑清晰度和可维护性非常重要。以下是一些提示:

- 为模块使用描述性且一致的名称;
- 将相关代码分组到模块中;
- 保持模块小而集中;
- 使用嵌套模块创建层次结构;
- 用注释记录模块。

3.16.5 创建和管理 crate

1. 创建一个新的 crate

要创建新的 crate,我们需要在项目的根目录中定义一个 Cargo.toml 文件。Cargo.toml 文件包含有关 crate 的元数据,例如其名称、版本、依赖项和其他设置。

下面是一个简单的 Cargo.toml 文件的示例:

```
[package]
name="myoncrate"
version="0.1.0"
authors=["北科信链"]

[dependencies]
rand = "0.8.5"
```

2. 管理依赖关系

Rust crate 可以依赖于其他 crate,这允许程序员重用其他项目中的代码。要管理依赖项,我们可以使用 Cargo.toml 文件。在 Cargo.toml 文件中,可以指定 crate 所依赖的 crate。例如,以下 Cargo.toml 文件指定 mon_crate crate 依赖于 rand 代码:

```
[package]
name="mon_crate"
version="0.1.0"
authors=["Milly Chou"]

[dependencies]
rand="0.9.5"
```

3. 将 crates 发布到 Crates.io

创建 crate 后,我们可以将其发布到 Rust 的官方 crate 存储库 Crates.io。将 crate 发布到

Crates.io 将允许其他 Rust 开发人员在他们的项目中找到并使用我们的 crate。

3.16.6　工作空间

如同 3.12 节所述，Rust 中的工作空间（WorkSpace）是一个包含一个或多个 Rust 项目（也称为 crate）的目录，它作为管理多个 crate 的中心位置，促进共享依赖项和代码。通过在工作空间中组织 crate，开发人员可以轻松地共享代码、管理依赖项以及同时构建和测试多个 crate。

1. 设置工作空间

要创建新工作空间，只需要创建一个新目录并使用 cargo init 命令将其初始化为工作空间。这将创建一个 Cargo.toml 文件，它是工作空间的配置文件。例如：

```
mkdir mon_workspace
cd mon_workspace
cargo init
```

要将新 crate 添加到工作区，请使用 cargo add 命令，这将为 crate 创建一个新目录并将其添加到 Cargo.toml 文件中。例如：

```
cargo add myoncrate
```

2. 管理工作空间内的依赖关系

在工作空间中使用多个 crate 时，有效管理依赖关系非常重要。Rust 的依赖管理系统允许 crate 相互依赖，也允许 crate 依赖于 Crates.io 上发布的外部 crate。

要指定工作空间中的依赖项，请使用 Cargo.toml 文件中的 dependencies 部分。示例如下：

```
[dependencies]
mon_crate={path="mon_crate"}
```

这将告诉 Cargo 名为 mon_crate 的 crate 依赖于另一个同名的 crate，该 crate 位于工作区的 mon_crate 目录中。

3. 使用工作空间的好处

使用工作空间的好处在于：集中依赖管理，重用，易用。具体解释如下：

- 工作空间提供集中的依赖关系管理，使开发人员能够在一个地方轻松管理多个 crate 的依赖关系，简化了更新和解决依赖关系的过程，可以确保工作区中所有 crate 的一致性；
- 工作空间通过允许 crate 共享代码和模块来促进代码的可重用性，可以促进代码组织并减少重复，从而更轻松地跨多个项目维护和更新代码；
- 工作空间允许开发人员使用单个命令运行测试并构建工作空间中的所有 crate，从而简化了测试和构建过程以及开发工作流程，并减少了重复任务所花费的时间。

3.16.7　项目组织工具

除了核心 Rust 语言和标准库之外，还有一些工具可以帮助有效组织 Rust 项目。

- Cargo：Cargo 是官方的 Rust 包管理器，负责处理依赖管理、构建、测试和发布 crate，它通过提供创建、管理和发布 crate 的命令来简化项目组织。

- Rustfmt：Rustfmt 是一种代码格式化程序，可以确保整个项目的代码格式保持一致，有助于维护干净且可读的代码库，使开发人员更轻松地协作和维护项目。
- 文档生成器：Rustdoc 等文档生成器会自动从 Rust 源代码生成 API 文档，有助于为库和应用程序创建全面的文档，使用户更容易理解和使用代码。
- Linters：像 Clippy 这样的 Linters 会分析 Rust 代码来识别潜在的错误、风格问题和性能优化，有助于维护代码质量并遵守最佳实践，防止出现错误并提高代码的可维护性。
- 版本控制系统：像 Git 这样的版本控制系统对于管理代码更改和 Rust 项目协作至关重要，它允许开发人员跟踪更改、解决冲突并维护代码库的多个版本。

3.16.8 惯例和最佳实践

建立编码、测试和文档约定对于维护一致且可读的代码库至关重要，这些约定确保了所有贡献者遵循标准化方法，从而提高代码质量和协作。以下是一些常见的约定和最佳实践。

1. 编码约定

编码约定提供了以一致且可读的方式编写 Rust 代码的指南，这些约定涵盖变量和函数命名、代码结构和布局、错误处理和日志记录等方面。

- 变量和函数命名：为变量和函数使用描述性且一致的名称，使代码不言自明；避免缩写并确保名称反映实体的目的。
  - 使用描述性和简洁的名称：变量和函数名称应清楚地传达其目的和意图；避免使用模糊或通用名称，例如 x、y 或 foo。
  - 遵循 Rust 的命名约定：Rust 对变量、函数和类型有特定的命名约定；例如，变量和函数以小写字母开头，而类型以大写字母开头。
  - 使用一致的命名：在整个项目中保持命名约定的一致性，这使得其他开发人员可以更容易地理解和浏览代码库。
- 代码结构和布局：将代码组织成逻辑块并使用适当的缩进来提高可读性；跨模块和包保持一致的结构，以便其他开发人员更轻松地浏览代码库。
  - 将代码组织成逻辑块：使用函数、模块和 crate 将代码分解成逻辑块，这提高了可读性和可维护性。
  - 使用正确的缩进：缩进有助于直观地对相关代码块进行分组并提高可读性；遵循 Rust 推荐的缩进样式。
  - 保持代码行简短：长代码行可能难以阅读和理解；最大行长度为 80 个字符。
- 错误处理和日志记录：在整个代码中优雅且一致地处理错误；使用 Rust 标准库提供的错误处理机制，并使用适当的级别和消息记录错误。
  - 优雅地处理错误：错误是编程中不可避免的一部分；使用 Rust 的错误处理机制来优雅地处理错误并提供有意义的错误消息。
  - 记录重要事件：使用日志记录来记录应用程序中的重要事件和错误，这有助于调试和监控应用程序。

2. 测试约定

测试约定确保所有代码都经过彻底测试和可维护，这些约定涵盖了编写单元测试、组织和命名测试以及执行集成和端到端测试等方面。

- 编写单元测试：为各个功能和模块编写单元测试，以确保其正确性；使用 test 或 spectest 等测试框架来有效地编写和组织测试。
  - ➢ 为所有代码编写测试：每段代码都应该有相应的单元测试；单元测试验证各个功能和模块的正确性。
  - ➢ 保持测试独立：单元测试应该相互独立，不应该依赖外部因素，这使得它们更容易维护和调试。
  - ➢ 使用富有表现力的测试名称：测试名称应清楚地描述测试的目的，这使得识别和理解测试用例变得更加容易。
- 组织和命名测试：将测试组织成逻辑组，并对其进行描述性命名，以表明每个测试的目的，这使得识别和维护测试变得更加容易。
  - ➢ 将测试组织成逻辑组：将相关测试分组到逻辑模块或类中，这使得导航和维护测试套件变得更加容易。
  - ➢ 使用一致的命名：对测试文件和测试函数使用一致的命名约定，这提高了可读性，并使其更容易识别和理解测试。
- 集成和端到端测试：除了单元测试之外，还可以考虑编写集成和端到端测试来验证整个系统或应用程序的行为。
  - ➢ 执行集成测试：集成测试验证应用程序不同组件之间的交互，确保组件按预期协同工作。
  - ➢ 执行端到端测试：端到端测试模拟应用程序的实际使用情况，从用户的角度验证应用程序的行为是否符合预期。

3. 文档约定

文档约定确保了代码库文档齐全且易于理解，这些约定涵盖了编写清晰简洁的文档、使用 Rustdoc 注释以及生成 API 文档等方面。

- 编写清晰简洁的文档：为代码编写清晰简洁的文档，解释目的、用法和实现细节；使用 Markdown 或 Rustdoc 注释来记录模块、函数和类型。
  - ➢ 使用清晰简洁的语言：文档应使用清晰简洁且易于理解的语言编写；避免使用读者可能不熟悉的行话或技术术语。
  - ➢ 提供上下文和示例：提供上下文和示例以帮助读者理解代码的目的和用法，这使得文档内容更加丰富，更具吸引力。
  - ➢ 保持文档最新：文档应与代码库中的最新更改保持同步，这确保了文档准确地反映项目的当前状态。
- 使用 Rustdoc 注释：使用 Rustdoc 注释来记录函数、结构和枚举等代码元素；这些注释会被 Rustdoc 自动转换为 API 文档。
  - ➢ 使用 Rustdoc 注释：Rustdoc 注释用于为 Rust 项目生成 API 文档，这些注释提供了有关函数、模块和类型的信息。
  - ➢ 遵循 Rustdoc 的注释约定：Rustdoc 有应遵循的特定注释约定，这些约定确保了生成的文档是一致且可读的。
- 生成 API 文档：使用 Rustdoc 生成 API 文档，为 crate 创建 HTML 或 Markdown 文档，该文档对于用户有效理解和使用 crate 至关重要。
  - ➢ 生成 API 文档：API 文档提供了项目公共 API 的详细信息，该文档对于想要使用

项目 API 的开发人员非常有用。

- 发布 API 文档：在线发布 API 文档，以便用户轻松访问，这使得开发者更容易了解和使用项目。

3.16.9 持续集成

持续集成(CI)是一种软件开发实践，是自动执行构建、测试代码更改并将代码更改合并到中央存储库的过程。CI 有助于确保代码更改在合并到主分支之前经过测试和验证，从而降低引入错误的风险，并保持代码质量。

1. CI 概述及其好处

CI 为 Rust 项目提供了几个好处。

- 早期检测问题：CI 管道对每个代码更改运行测试和检查，使开发人员能够在问题导致生产中出现问题之前尽早识别和修复问题。
- 提高代码质量：CI 管道可以强制执行编码标准、运行 linter 并执行静态分析，可以帮助开发人员保持较高的代码质量和最佳实践。
- 减少合并冲突：通过持续集成代码更改，CI 减少了合并冲突的可能性，从而更容易合并更改并保持代码库干净。
- 提高开发人员的信心：CI 使开发人员充满信心，他们的代码更改在合并之前经过了测试和验证，从而减少了引入错误的风险，并提高了代码的整体质量。

2. 设置 CI 管道

要为 Rust 项目设置 CI 管道，可以使用各种 CI 工具，例如 GitHub Actions、Travis CI 或 CircleCI。这些工具提供了一个定义和执行 CI 管道的平台，包括构建、测试和部署代码。

以下是设置 CI 管道涉及的一般步骤。

- 选择 CI 工具：选择适合项目需求和偏好的 CI 工具。GitHub Actions 是 GitHub 上托管的开源项目的热门选择，而 Travis CI 和 CircleCI 对于开源和私有项目都很受欢迎。
- 创建 CI 配置文件：每个 CI 工具都有自己的配置文件格式。例如，GitHub Actions 使用名为 github/workflows/< workflow-name >. yml 的基于 YAML 的配置文件，该文件定义了要在管道中执行的步骤和任务。
- 定义管道步骤：在 CI 配置文件中，定义要在管道中执行的步骤。常见步骤如下。
  - 作为 CI 管道的一部分。
  - 构建项目：此步骤编译 Rust 代码并生成可执行文件或库。
  - 运行测试：此步骤执行单元测试、集成测试和其他类型的测试，以验证代码的正确性。
  - 执行静态分析：此步骤运行 linter 和静态分析工具来识别潜在的代码问题并强制执行编码标准。
  - 生成文档：此步骤使用 Rustdoc 等工具生成项目的 API 文档。
  - 部署项目(可选)：此步骤将项目部署到暂存或生产环境。
  - 配置触发器：指定应触发管道的事件。例如，可以将管道配置为在每次推送到主分支或每次拉取请求时运行。
  - 启用管道：CI 配置完成后，在 CI 工具界面中启用管道。每当指定的触发器发生时，都将启动管道执行。

3. 常用的持续集成工具

Rust 项目常用的 CI 工具如下。

- GitHub Actions：GitHub Actions 是 GitHub 提供的 CI/CD 平台，它易于设置和使用，特别是对于 GitHub 上托管的项目。
- Travis CI：Travis CI 是一种流行的 CI 工具，支持各种编程语言，它为开源项目提供了免费层。
- CircleCI：CircleCI 是另一种流行的 CI 工具，提供广泛的功能和集成，并以其可扩展性和灵活性而闻名。
- Jenkins：开源、自定义插件、报告不像 github 那样精美。

3.16.10　试运行

测试是软件开发的重要组成部分，Rust 为编写和运行测试提供了出色的支持。

1. 在本地运行测试

要在本地运行测试，可以使用 cargo test 命令，该命令编译测试代码并执行测试，测试结果显示在控制台中，指示哪些测试通过，哪些测试失败。这是本地运行测试的示例：

```
cargo test
```

2. CI 管道中的自动化测试

CI 管道通常包括运行测试的步骤，确保了每次代码更改时都执行测试，并且代码库保持稳定。要在 CI 管道中自动执行测试，可以在 CI 配置文件中使用 cargo test 命令。确切的语法取决于所使用的 CI 工具。例如，在 GitHub Actions 工作流程文件中，可以添加以下步骤来运行测试：

```
- name: Run tests
  run: cargo test
```

此步骤将作为 CI 管道的一部分运行测试。

3.16.11　覆盖率报告和代码质量指标

CI 管道还可以生成覆盖率报告和代码质量指标。覆盖率报告显示测试覆盖了代码的哪些部分，有助于识别可能需要更多测试的区域。代码质量指标提供了对代码库整体质量的洞察，包括代码复杂性、可维护性和对编码标准的遵守等指标。这些报告和指标可用于提高代码质量，并确保项目经过良好的测试和维护。

为了说明本节讨论的概念，下面展示一些示例性组织的实际 Rust 项目。我们将分析他们的文件和文件夹结构、模块和包结构，以及他们使用的工具和约定。下面是一些常见的框架。

1. Actix 网络框架

Actix Web 是一种流行的 Web 框架，用于使用 Rust 构建高性能、异步 Web 应用程序。它的组织证明了本节讨论的原则。

- 文件和文件夹结构：Actix Web 遵循多 crate 结构，每个 crate 都有特定用途。actix-web crate 包含核心框架功能，而其他 crate(如 actix-rt 和 actix-service)提供附加功能

和实用程序。

- 模块和 crate 结构：Actix Web 的模块被组织成逻辑单元，例如路由、请求处理和错误处理。每个模块负责框架功能的特定方面，使其易于理解和维护。
- 工具和约定：Actix Web 利用 Cargo 进行依赖管理，并利用 Rustfmt 实现一致的代码格式；它还遵守编码约定，例如命名约定和错误处理实践，以确保一致且可读的代码库。

2. Diesel ORM

Diesel 是 Rust 的对象关系映射(ORM)框架，可以简化数据库交互，它的组织展示了如何有效地构建库 crate。

- 文件和文件夹结构：Diesel 遵循单 crate 结构，其所有代码都驻留在一个 crate 中，这使得管理和分发库变得更容易。
- 模块和 crate 结构：Diesel 的模块根据功能进行组织，包括用于数据库连接、查询构建器和模式定义的模块，这种结构使得查找和使用必要的功能变得更容易。
- 工具和约定：Diesel 使用 Cargo 进行依赖管理，使用 Rustfmt 进行代码格式化，它还遵循编码约定，例如命名约定和错误处理实践，以确保一致且用户友好的 API。

3. Rocket Web 框架

Rocket 是一个 Web 框架，用于在 Rust 中构建快速且易于使用的 Web 应用程序，它的组织展示了如何有效地构建网络框架。

- 文件和文件夹结构：Rocket 遵循多 crate 结构，其中 rocket crate 包含核心框架功能和其他 crate，例如 rocket_contrib 和 rocket_codegen，提供额外的功能和实用程序。
- 模块和 crate 结构：Rocket 的模块被组织成逻辑单元，例如路由、请求处理和错误处理。这种结构使框架的功能更易于理解和维护。
- 工具和约定：Rocket 使用 Cargo 进行依赖管理，使用 Rustfmt 实现一致的代码格式。它还遵守编码约定，例如命名约定和错误处理实践，以确保一致且可读的代码库。

这些示例说明了如何有效地构建和管理组织良好的 Rust 项目。通过遵循本节讨论的原则和最佳实践，可以创建易于理解、维护和扩展的 Rust 项目。

无论我们使用什么工具，都要关注以下核心概念。

- 模块化组织：Rust 的模块化系统允许将代码逻辑分组为模块和 crate，从而提高清晰度和可维护性。
- crate 和工作区：板条箱用作可重用的库或可执行文件，而工作区则有助于集中管理多个板条箱。
- 组织工具：Cargo、Rustfmt 和文档生成器等工具可简化项目组织、格式设置和文档。
- 约定和最佳实践：已建立的编码、测试和文档约定可确保一致性和可读性。
- 持续改进：定期审查和完善项目组织，以适应不断变化的需求和最佳实践。

3.16.12 持续改进的重要性

Rust 是一种快速发展的语言，组织 Rust 项目的最佳实践也是如此。及时了解最新的开发、工具和技术至关重要，以确保我们的项目保持良好的结构和可维护性。通过不断学习和适应，可以确保我们的 Rust 项目保持组织有序、可维护，并且为团队带来愉快的合作体验。

3.17　复杂例子

大多时候，我们会交替使用包（package，有时也称为项目或者编译单元）和 crate，但其实二者是有区别的：crate 是一个 Rust 模块的层级结构，在根目录下至少有一个 lib.rs 或者 main.rs。而 package 是一系列 crates 和元数据的集合，并且其路径和功能都定义在 Cargo.toml 文件中，它可能包括一个库 crate、多个二进制 crate、一些集成测试 crate，甚至包括一些工作空间的成员。下面给出了一个比较复杂的例子——Parity。Parity 是以太坊的客户端[①]，其项目的目录结构如下：

```
D:\DEV\RUST\SYS\PARITY-MASTER
├─.github
├─chainspec
├─dapps
├─devtools
├─docker
├─docs
├─ethash
├─ethcore
├─ethcrypto
├─ethkey
├─ethstore
├─evmbin
├─evmjit
├─hash-fetch
├─hw
├─ipfs
├─js
├─js-old
├─json
├─local-store
├─logger
├─mac
├─machine
├─nsis
├─panic_hook
├─parity
├─price-info
├─rpc
├─rpc_cli
├─rpc_client
├─scripts
├─secret_store
├─snap
├─stratum
├─sync
├─updater
├─util
├─whisper
└─windows
```

（1）首先来看这个 workspace 的 Cargo.toml（有删节）：

```
[package]
description = "Parity Ethereum client"
```

① https://github.com/openethereum/parity-ethereum

```
name ="parity"
version ="1.9.0"
license ="GPL-3.0"
authors =["Parity Technologies <admin@parity.io>"]
build ="build.rs"                                  # 预构建脚本,下面(2)说明

[dependencies]
log ="0.3"
env_logger ="0.4"
...
ctrlc ={ git ="https://github.com/paritytech/rust-ctrlc.git" } # 依赖包可以指向 GitHub,以及具
                                                               # 体分支
jsonrpc-core ={ git ="https://github.com/paritytech/jsonrpc.git", branch ="parity-1.8" }
...
ethcore-util ={ path ="util" }                     # 依赖包可以使用基于本 workspace 的相对路径
ethcore-bytes ={ path ="util/bytes" }
...
rlp ={ path ="util/rlp" }
rpc-cli ={ path ="rpc_cli" }
parity-hash-fetch ={ path ="hash-fetch" }
...
parity-dapps ={ path ="dapps", optional =true } // 可选的特征(feature)
...

[build-dependencies]
rustc_version ="0.2"

[dev-dependencies]
pretty_assertions ="0.1"
ipnetwork ="0.12.6"

[target.'cfg(windows)'.dependencies]           # 使用 cfg 属性来指定条件编译需要引入的包
winapi ="0.2"

[target.'cfg(not(windows))'.dependencies]      # 使用 cfg 属性来指定条件编译需要引入的包
daemonize ="0.2"

[features]
default =["ui-precompiled"]
ui =[
  "ui-enabled",
  "parity-dapps/ui",
]
ui-precompiled =[
  "ui-enabled",
  "parity-dapps/ui-precompiled",
]
ui-enabled =["dapps"]
...

[[bin]]
path ="parity/main.rs"   # 本 workspace 编译的最终结果是要编译 parity 目录下的 main.rs,生成一个
                         # 可执行文件
name ="parity"

[profile.dev]
panic ="abort"           # panic 的情况下直接 abort 退出,交由操作系统来进行清理。可能的选项还有
                         # unwind。参见 5.6 节

[profile.release]
debug =false             # release 版本的条件下不包含调试代码
lto =false               # release 版本没有连接时优化
panic ="abort"

[workspace]              # 下面的目录都是 workspace 的成员项目
members =[
```

```
"ethstore/cli",
"ethkey/cli",
"evmbin",
"whisper",
"chainspec",
"dapps/js-glue"
]
```

(2) 预编译脚本 build.rs(有删节):

```
extern crate rustc_version;

const MIN_RUSTC_VERSION: &'static str ="1.15.1";

fn main() {
  let is =rustc_version::version() .unwrap() ;
  let required =MIN_RUSTC_VERSION.parse() .unwrap() ;   // 获得所需的最低 Rustc 版本号
  assert!( is > =required, format!("              // 断言: 如果低于最低需要版本,则打印信息,错误返回
It looks like you are compiling Parity with an old rustc compiler{} .
Parity requires version{} . Please update your compiler.
If you use rustup, try this:
           rustup update stable
and try building Parity again.", is, required) ) ;
}
```

从上面的内容可以看到,build.rs 主要用来判断 Rust 编译器的版本。每次 cargo build 开始编译,都会首先调用 build.rs。如果版本小于 1.15.1,则编译终止。

(3) Chainspec 是本 workspace 里的一个二进制 crate,也是 workspace 的成员项目(图 3.3)。

图 3.3　Parity 中 chainspec 目录结构

其 Cargo.toml 内容如下:

```
[package]
name ="chainspec"
version ="0.1.0"
authors =["debris <marek.kotewicz@gmail.com> "]

[dependencies]
ethjson ={ path ="../json" }
serde_json ="1.0"
serde_ignored ="0.0.4"
```

(4) devtools 是本 workspace 里的一个库 crate(图 3.4)。

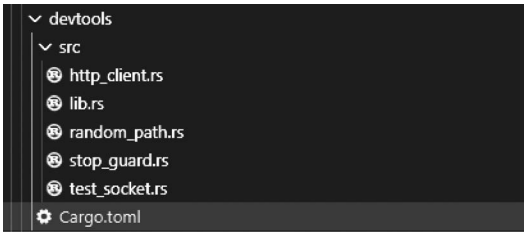

图 3.4　Parity 中 devtools 目录结构

其 Cargo.toml 内容为(有删节):

```
[package]
description = "Ethcore development/test/build tools"
...
[dependencies]
rand = "0.3"

[lib]
path = "src/lib.rs"          # 指定 src 下的 lib.rs 为 crate 的入口文件.文件内容解释参见 13 章
test = true
```

(5) dapps 是本 workspace 里的一个库 crate,dapps 的 js-glue 是本 workspace 的成员(图 3.5)。

图 3.5 Parity 中 dapps 目录结构

(6) ui 是 dapps 的子模块,其 Cargo.toml 的部分内容如下:

```
[features]      # 指定 no-precompiled-js,代表同时指定 parity-ui-dev 和 parity-ui-old-dev
no-precompiled-js = ["parity-ui-dev","parity-ui-old-dev"]                    # 没有预编译的 js
use-precompiled-js = ["parity-ui-precompiled","parity-ui-old-precompiled"]   # 使用预编译的 js
```

其 lib.rs 的部分内容如下:

```
#[cfg(feature = "parity-ui-dev")]          # 编译时如果指定了 parity-ui-dev,就使用其下的内容
mod inner{
  extern crate parity_ui_dev;
  pub use self::parity_ui_dev::*;
}

#[cfg(feature = "parity-ui-precompiled")] # 编译时如果指定了 parity-ui-precompiled,就使用其下
                                          # 的内容
mod inner{
  extern crate parity_ui_precompiled;
  pub use self::parity_ui_precompiled::*;
}

#[cfg(feature = "parity-ui-old-dev")]     # 编译时如果指定了 parity-ui-old-dev,就使用其下的内容
pub mod old{
  extern crate parity_ui_old_dev;
  pub use self::parity_ui_old_dev::*;
}

#编译时如果指定了 parity-ui-old-precompiled,就使用其下的内容
#[cfg(feature = "parity-ui-old-precompiled")]
pub mod old{
  extern crate parity_ui_old_precompiled;
  pub use self::parity_ui_old_precompiled::*;
}
pub use self::inner::*;
```

可以看到，Rust 编译器会根据不同的特征(feature)对不同的代码进行编译。编译时也可以在命令行设置 feature：

```
cargo build --features "no-precompiled-js"
```

如果不需要默认的 feature，请使用--no-default-features。如果想设定一个和项目相关的 feature，可以在 Cargo.toml 中这么写：

```
[dependencies.myproject]                    # myproject 的包依赖
default-features = false                    # 不使用默认 feature
features = ["use-precompiled-js"]           # 定义 feature 标志
```