# 概述

Cargo.toml是cargo命令的运行清单，它以 TOML 格式编写。每个清单文件都包含以下部分：

* cargo-features — 指定启用的不稳定特性。
* [package] — 定义包。
* [lib] — Library target设置。
* [[bin]] — Binary target设置。
* [[example]] — Example target设置。
* [[test]] — Test target设置。
* [[bench]] — Benchmark target 设置。
* [dependencies] — 包的库依赖。
* [dev-dependencies] — examples, tests, and benchmarks的依赖。
* [build-dependencies] — build scripts的依赖。
* [target] — 特定于平台的依赖项。
* [badges] — 显示在注册表上的标识。
* [features] — 条件编译。
* [patch] — 覆盖依赖。
* [replace] — 覆盖依赖 (deprecated)，使用patch。
* [profile] —编译器设置和优化。
* [workspace] — 工作区定义。

# 文件布局

Cargo 使用文件放置约定，文件放置具有以下规则：

|  |
| --- |
| .  ├── Cargo.lock  ├── Cargo.toml  ├── src/  │ ├── lib.rs  │ ├── main.rs  │ └── bin/  │ ├── named-executable.rs  │ ├── another-executable.rs  │ └── multi-file-executable/  │ ├── main.rs  │ └── some\_module.rs  ├── benches/  │ ├── large-input.rs  │ └── multi-file-bench/  │ ├── main.rs  │ └── bench\_module.rs  ├── examples/  │ ├── simple.rs  │ └── multi-file-example/  │ ├── main.rs  │ └── ex\_module.rs  └── tests/  ├── some-integration-tests.rs  └── multi-file-test/  ├── main.rs  └── test\_module.rs |

* Cargo.toml 和 Cargo.lock 存储在包的根目录。
* 源代码位于 src 目录中。
* 默认库文件是 src/lib.rs。
* 默认的可执行文件是 src/main.rs。
* 其他可执行文件可以放在 src/bin/ 中。
* 基准测试文件在 benches 目录中。
* 示例文件在examples目录中。
* 集成测试文件在tests目录。

# cargo-features

该配置用于启用一些实验性的特性，并且仅在夜间频道"cargo +nightly build"有用。

例如：

|  |
| --- |
| # This specifies which new Cargo.toml features are enabled.  cargo-features = ["test-dummy-unstable"]  [package]  name = "my-package"  version = "0.1.0"  im-a-teapot = true # This is a new option enabled by test-dummy-unstable. |

具体可用的实验性特性查看[这里](https://doc.rust-lang.org/cargo/reference/unstable.html)。

# [package]

Cargo 所需的唯一字段是package中的name和version。如果发布到注册中心，注册中心可能需要其他字段。

例如：

|  |
| --- |
| [package]  name = "hello\_world" # the name of the package  version = "0.1.0" # the current version, obeying semver |

## name

包名称是用于引用包的标识符。并作为推断的 lib 和 bin 目标的默认名称。

名称只能使用字母数字字符或 - 或 \_，并且不能为空。

cargo new 和cargo init 对包名称施加了一些额外的限制，例如强制它是有效的 Rust 标识符而不是关键字。

crates.io 施加了更多限制，例如强制使用 ASCII 字符，并且不是保留名称，不是特殊的 Windows 名称（例如“nul”），不能太长等。

例如：

|  |
| --- |
| [package]  name = "hello\_world" # the name of the package |

## version

version用于指定当前包的版本。Cargo 融入了语义版本控制的概念，因此确保遵循以下基本规则：

* 使用包含三个数字部分的版本号，例如 1.0.0 而不是 1.0。
* 版本到达 1.0.0 之前，一切都可以发生。但是如果进行了重大更改，例如向结构添加字段或向枚举添加成员，请增加次要版本。
* 版本到达在 1.0.0 之后，只有在增加主要版本时才进行重大更改。
* 版本到达在 1.0.0 之后，不要在补丁级别的版本中添加任何新的公共 API。
* 版本到达在 1.0.0 之后，如果添加任何新的 pub 结构、特征、字段、类型、函数、方法或其他任何内容，请始终增加次要版本。

例如：

|  |
| --- |
| [package]  version = "0.1.0" # the current version, obeying semver |

## authors

authors字段列出了被视为包“作者”的人员或组织。这些名称将列在 crates.io 上的 crate 页面上。每个作者末尾的可以添加一个尖括号，其中可以包含一个可选的电子邮件地址。

例如：

|  |
| --- |
| [package]  authors = ["Alice <a@example.com>", "Bob <b@example.com>"] |

## edition

edition键是一个可选键，它会影响你的包是用哪个 Rust 版本编译的。在 [package] 中设置edition将影响包中的所有targets/crates，包括test suites, benchmarks, binaries, examples等。

大多数manifests的版本字段都由" cargo new"自动填充为最新的稳定版本，默认情况下为"2018"。

如果 Cargo.toml 中不存在edition字段，则假定为 2015 版本以实现向后兼容性。

例如：

|  |
| --- |
| [package]  edition = '2018' |

## description

description字段是关于package的简短介绍。crates.io 将与package一起显示。这应该是纯文本（不是 Markdown）。

例如：

|  |
| --- |
| [package]  description = "A short description of my package" |

## documentation

documentation字段指定托管 crate 文档的网站的 URL。如果cargo.toml清单文件中没有指定 URL，crates.io 将自动将您的 crate 链接到相应的 docs.rs 页面。

例如：

|  |
| --- |
| [package]  documentation = "https://docs.rs/bitflags" |

## readme

readme 字段应该是包根目录中文件的路径（相对于 Cargo.toml），其中包含有关包的一般信息。

当发布包时，此文件将传输到注册表。crates.io 会将其解释为 Markdown 并将其呈现在 crate 的页面上。

如果没有为此字段指定值，并且包根目录中存在名为 README.md、README.txt 或 README 的文件，则将使用该文件的名称。可以通过将此字段设置为 false 来抑制此行为。 如果该字段设置为 true，则将假定值为README.md。

例如：

|  |
| --- |
| [package]  readme = "README.md" |

## homepage

homepage 字段应该是指向包的主页的站点的 URL。

例子：

|  |
| --- |
| [package]  homepage = "https://serde.rs/" |

## repository

repository 字段是包的源存储库的 URL。

例子：

|  |
| --- |
| [package]  repository = "https://github.com/rust-lang/cargo/" |

## license

license字段包含发布包所依据的软件许可证的名称。

crates.io 将许可证字段解释为 SPDX 2.1 许可证表达式。该名称必须是 SPDX 许可证列表 3.11 中的已知许可证。目前不支持括号。

SPDX 许可证表达式支持 AND 和 OR 运算符来组合多个许可证。使用 OR 表示用户可以选择任一许可证。使用 AND 表示用户必须同时遵守两个许可证。 WITH 运算符表示具有特殊例外的许可证。例如：

* MIT OR Apache-2.0
* LGPL-2.1-only AND MIT AND BSD-2-Clause
* GPL-2.0-or-later WITH Bison-exception-2.2

例子：

|  |
| --- |
| [package]  license = "MIT OR Apache-2.0" |

## license-file

license-file 字段包含包含许可证文本的文件的路径（相对于此 Cargo.toml）。

例子：

|  |
| --- |
| [package]  license-file = "LICENSE.txt" |

## keywords

keywords字段是描述此包的字符串数组。在注册表中搜索包时，这会有助于别人找到此包。

crates.io 最多有 5 个关键字。 每个关键字必须是ASCII文本，以字母开头，只能包含字母、数字、\_或-，最多20个字符。

例子：

|  |
| --- |
| [package]  keywords = ["gamedev", "graphics"] |

## categories

categories字段是此包所属类别的字符串数组。

crates.io 最多有 5 个类别。每个类别都应与 https://crates.io/category\_slugs 上可用的字符串之一匹配，并且必须完全匹配。

例子：

|  |
| --- |
| [package]  categories = ["command-line-utilities", "development-tools::cargo-plugins"] |

## workspace

workspace字段可用于配置此包所属的工作空间。

如果未指定，将被推断为文件系统中向上的第一个具有[workspace]表的Cargo.toml。

当成员不在工作区根目录的子目录中，设置此项很有用。

如果当前Cargo.toml清单已定义了[workspace]表，则无法指定此字段。也就是说，一个crate不能既是工作区中的根 crate，又是另一个工作区的成员 crate。

例子：

|  |
| --- |
| [package]  workspace = "path/to/workspace/root" |

## build

build 字段指定包根目录中的一个文件，该文件是用于构建本机代码的构建脚本。

默认值为“build.rs”，从包根目录中名为 build.rs 的文件加载脚本。使用 build = "custom\_build\_name.rs" 指定不同文件的路径。使用 build = false 禁用构建脚本的自动检测。

例子：

|  |
| --- |
| [package]  build = "build.rs" |

## links

links 字段指定要链接到的本机库的名称。和build字段使用构建脚本相关。

|  |
| --- |
| [package]  links = "foo" |

## exclude

使用exclude 字段可以明确的指定一组打包时需要忽略的文件，文件模式应该是 gitignore 风格的模式：

* foo 匹配包中任何位置的名称为 foo 的任何文件或目录。 这相当于模式 \*\*/foo。
* /foo 仅匹配包根目录中名称为 foo 的任何文件或目录。
* foo/ 匹配包中任何位置的名称为 foo 的任何目录。
* \*\*/ 前缀表示在任何目录中匹配。例如，\*\*/foo/bar 匹配任何位置下的目录foo下的文件或者目录bar。
* /\*\* 后缀匹配里面的所有内容。 例如，foo/\*\* 匹配目录 foo 内的所有文件，包括 foo 下子目录中的所有文件。
* /\*\*/ 匹配零个或多个目录。 例如，a/\*\*/b 匹配 a/b、a/x/b、a/x/y/b 等。
* ! 前缀否定模式。例如，src/\*\*.rs and !foo.rs 的模式将匹配 src 目录中所有扩展名为 .rs 的文件，但名为 foo.rs 的文件除外。
* 支持常见的 glob 模式，如 \*、? 和 []：
  + \* 匹配除 / 之外的零个或多个字符。 例如，\*.html 匹配包中任何位置带有 .html 扩展名的任何文件或目录。
  + ? 匹配除 / 之外的任何字符。 例如，foo? 匹配food，但不匹配 foo。
  + [] 允许匹配一系列字符。 例如，[ab] 匹配 a 或 b。 [a-z] 匹配字母 a 到 z。

exclude字段和include字段是互斥的。

例子：

|  |
| --- |
| [package]  exclude = ["build/\*\*/\*.o", "doc/\*\*/\*.html"] |

## include

使用include字段可以明确的指定一组打包时需要包含的文件，文件模式应该是 gitignore 风格的模式：

* foo 匹配包中任何位置的名称为 foo 的任何文件或目录。 这相当于模式 \*\*/foo。
* /foo 仅匹配包根目录中名称为 foo 的任何文件或目录。
* foo/ 匹配包中任何位置的名称为 foo 的任何目录。
* \*\*/ 前缀表示在任何目录中匹配。例如，\*\*/foo/bar 匹配任何位置下的目录foo下的文件或者目录bar。
* /\*\* 后缀匹配里面的所有内容。 例如，foo/\*\* 匹配目录 foo 内的所有文件，包括 foo 下子目录中的所有文件。
* /\*\*/ 匹配零个或多个目录。 例如，a/\*\*/b 匹配 a/b、a/x/b、a/x/y/b 等。
* ! 前缀否定模式。例如，src/\*\*.rs and !foo.rs 的模式将匹配 src 目录中所有扩展名为 .rs 的文件，但名为 foo.rs 的文件除外。
* 支持常见的 glob 模式，如 \*、? 和 []：
  + \* 匹配除 / 之外的零个或多个字符。 例如，\*.html 匹配包中任何位置带有 .html 扩展名的任何文件或目录。
  + ? 匹配除 / 之外的任何字符。 例如，foo? 匹配food，但不匹配 foo。
  + [] 允许匹配一系列字符。 例如，[ab] 匹配 a 或 b。 [a-z] 匹配字母 a 到 z。

exclude字段和include字段是互斥的。

例子：

|  |
| --- |
| [package]  include = ["src/\*\*/\*", "Cargo.toml"] |

## publish

publish字段设置为false可用于防止错误地将包发布到包注册表（如 crates.io），例如将包在公司中保密。

该值也可以是一个字符串数组，这些字符串是允许发布到的注册表名称。如果publish数组仅包含单个注册表，则当未指定--registry 标志时，cargo publish命令将使用它。

例子：

|  |
| --- |
| [package]  publish = false |

## metadata

metadata字段用于存储一些元数据，metadata 表被 Cargo 完全忽略，不会被警告。

例子：

|  |
| --- |
| [package.metadata.android]  package-name = "my-awesome-android-app"  assets = "path/to/static" |

## default-run

default-run 字段可用于指定由cargo run选择的默认二进制文件。例如，当同时存在 src/bin/a.rs 和 src/bin/b.rs 时指定a.rs：

例子：

|  |
| --- |
| [package]  default-run = "a" |

## autobins

autobins字段用于禁用自动bin target发现，以便使用[[bin]]表数组构建手动配置的目标。

例子：

|  |
| --- |
| [package]  autobins = false |

## autoexamples

autoexampless字段用于禁用自动example target发现，以便使用[[example]]表数组构建手动配置的目标。

例子：

|  |
| --- |
| [package]  autoexamples = false |

## autotests

autoetests字段用于禁用自动test target发现，以便使用[[test]]表数组构建手动配置的目标。

例子：

|  |
| --- |
| [package]  autotests = false |

## autobenches

autobenches字段用于禁用自动bench target发现，以便使用[[bench]]表数组构建手动配置的目标。

例子：

|  |
| --- |
| [package]  autobenches = false |

## resolver

resolver字段用于指定解析器版本。

可以使用不同的特征解析器算法：

* 版本 "1" 解析器是 Cargo 1.50 版之前的原始解析器，如果未指定解析器，则为默认解析器。
* 版本“2”解析器引入了功能统一的变化。

解析器是影响整个工作区的全局选项。依赖项中的解析器版本将被忽略，只会使用顶层package中的值。

例子：

|  |
| --- |
| [package]  resolver = "2" |

# [lib]

lib源文件名默认为 src/lib.rs，生成目标名默认为包名(任何破折号都替换为下划线)。lib的设置可以在 Cargo.toml 的 [lib] 表中自定义。

例子：

|  |
| --- |
| [lib]  name = "foo" # The name of the target.  path = "src/lib.rs" # The source file of the target.  test = true # Is tested by default.  doctest = true # Documentation examples are tested by default.  bench = true # Is benchmarked by default.  doc = true # Is documented by default.  plugin = false # Used as a compiler plugin (deprecated).  proc-macro = false # Set to `true` for a proc-macro library.  harness = true # Use libtest harness.  edition = "2015" # The edition of the target.  crate-type = ["lib"] # The crate types to generate. |

## name

name字段指定生成的库名称，这是依赖项将用于引用它的crate名。

## path

path 字段指定了 lib crate 的源文件所在的位置，相对于 Cargo.toml 文件。

如果未指定，则是src/lib.rs。

## test

test 字段指示运行cargo test时，是否需要执行lib crate中的单元测试。默认值为 true。

## doctest

doctest 字段指示运行cargo test时，是否需要执行lib crate中的测试文档示例。默认值为 true。

## bench

bench 字段指示运行cargo bench时，是否需要执行lib crate中的基准测试。默认值为 true。

## doc

doc 字段指示运行cargo doc时，是否需要为lib crate生成文档。默认值为 true。

## proc-macro

proc-macro 字段表示该lib crate是一个程序宏。

## harness

Harness 字段表示 --test 标志将被传递给 rustc，它会自动包含 libtest 库，该库是用于收集和运行标有 #[test] 属性的测试或标有 #[bench] 属性的基准测试的驱动程序。

默认值为 true。如果设置为 false，则需要负责定义 main() 函数来运行tests和benchmarks.

## edition

edition字段定义了将使用的 Rust 版本。如果未指定，则默认为package的 edition 字段。

## crate-type

crate-type字段定义了生成的lib crate类型。它是一个字符串数组，允许您为单个目标指定多种 crate 类型。

可用选项有lib、rlib、dylib、cdylib、staticlib 和 proc-macro：

* lib：生成rust库。具体类型由编译器决定。
* dylib：生成动态 Rust 库。此输出类型将在 Linux 上创建 .so 文件，在 macOS 上创建 .dylib 文件，在 Windows 上创建 .dll 文件。
* rlib：生成静态Rust库。这些 rlib 文件与 staticlib 文件不同，由编译器在将来的链接中进行解释。
* cdylib：产生动态系统库。此输出类型将在 Linux 上创建 .so 文件，在 macOS 上创建 .dylib 文件，在 Windows 上创建 .dll 文件。
* staticlib：产生静态系统库。其中包含所有本地 crate 的代码以及所有上游依赖项。 此输出类型将在 Linux、macOS 和 Windows (MinGW) 上创建 \*.a 文件，在 Windows (MSVC) 上创建 \*.lib 文件。 建议在将 Rust 代码链接到现有的非 Rust 应用程序等情况下使用这种格式，因为它不会动态依赖其他 Rust 代码。
* proc-macro：使用此 crate 类型编译的 crate 只能导出程序宏。 编译器会自动设置 proc\_macro 配置选项。 crate 总是使用与编译器本身构建时相同的目标进行编译。 例如，如果您使用 x86\_64 CPU 从 Linux 执行编译器，目标将是 x86\_64-unknown-linux-gnu，即使 crate 是为不同目标构建的另一个 crate 的依赖项。

# [[bin]]

bin源文件名默认为 src/main.rs，生成目标名默认为包名(任何破折号都替换为下划线)。其他bin的源文件在src/bin/目录中，生成目标名默认为文件名。可以在 Cargo.toml 的 [[bin]] 表数组中自定义每个bin的设置。

例子：

|  |
| --- |
| [[bin]]  name = "cool-tool"  test = false  bench = false  [[bin]]  name = "frobnicator"  required-features = ["frobnicate"] |

## name

name字段指定生成binary的名称。

## path

path 字段指定了bin crate 的源所在的位置，相对于 Cargo.toml 文件。

## test

test 字段指示运行cargo test时，是否需要执行bin crate中的单元测试。默认值为 true。

## bench

bench 字段指示运行cargo bench时，是否需要执行bin crate中的基准测试。默认值为 true。

## doc

doc 字段指示运行cargo doc时，是否需要为bin crate生成文档。默认值为 true。

## harness

Harness 字段表示 --test 标志将被传递给 rustc，它会自动包含 libtest 库，该库是用于收集和运行标有 #[test] 属性的测试或标有 #[bench] 属性的基准测试的驱动程序。

默认值为 true。如果设置为 false，则需要负责定义 main() 函数来运行tests和benchmarks.

## edition

edition字段定义了使用的 Rust 版本。如果未指定，则默认为package的 edition 字段。

## required-features

required-features 字段指定bin需要哪些[features]才能构建。如果未启用任何必需的feature，则将跳过。

例如：

|  |
| --- |
| [features]  # ...  postgres = []  sqlite = []  tools = []  [[bin]]  name = "my-pg-tool"  required-features = ["postgres", "tools"] |

# [[example]]

example源文件默认在examples/目录下，生成目标名默认为文件名。可以在 Cargo.toml 的 [[example]] 表数组中自定义每个example的设置。

默认情况下，example生成的是binary（带有 main() 函数）。 可以通过指定 crate-type 字段，将示例编译为库：

例子：

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

## name

name字段指定生成binary或library的名称。

## path

path 字段指定了 example的源所在的位置，相对于 Cargo.toml 文件。

## test

test 字段指示运行cargo test时，是否需要执行example中的单元测试。默认值为 false。

## bench

bench 字段指示运行cargo bench时，是否需要执行example中的基准测试。默认值为 true。

## doc

doc 字段指示运行cargo doc时，是否需要为example生成文档。默认值为 false。

## harness

Harness 字段表示 --test 标志将被传递给 rustc，它会自动包含 libtest 库，该库是用于收集和运行标有 #[test] 属性的测试或标有 #[bench] 属性的基准测试的驱动程序。

默认值为 true。如果设置为 false，则需要负责定义 main() 函数来运行tests和benchmarks.

## edition

edition字段定义了使用的 Rust 版本。如果未指定，则默认为package的 edition 字段。

## crate-type

crate-type字段定义了生成的example类型。它是一个字符串数组，允许您为单个目标指定多种 crate 类型。

可用选项有bin、lib、rlib、dylib、cdylib、staticlib 和 proc-macro。

## required-features

required-features 字段指定该example需要哪些[features]才能构建。如果未启用任何必需的feature，则将跳过。

例如：

|  |
| --- |
| [features]  # ...  postgres = []  sqlite = []  tools = []  [[example]]  name = "my-pg-tool"  required-features = ["postgres", "tools"] |

# [[test]]

Cargo 项目中有两种测试：单元测试、集成测试。单元测试在lib、bin、example的源文件中，集成测试在tests目录下。

当运行 cargo test 时，Cargo 会将tests目录下的文件中的每一个编译为单独的 crate，名称默认为文件名，然后执行它们。可以在 Cargo.toml 的 [[test]] 表数组中自定义每个test的设置。

## name

name字段指定生成binary的名称。

## path

path 字段指定了test的源所在的位置，相对于 Cargo.toml 文件。

## test

test 字段指示运行cargo test时，是否需要执行该集成测试。默认值为 true。

## bench

bench 字段指示运行cargo bench时，是否需要执行test中的基准测试。默认值为 false。

## doc

doc 字段指示运行cargo doc时，是否需要为test生成文档。默认值为 false。

## harness

Harness 字段表示 --test 标志将被传递给 rustc，它会自动包含 libtest 库，该库是用于收集和运行标有 #[test] 属性的测试或标有 #[bench] 属性的基准测试的驱动程序。

默认值为 true。如果设置为 false，则需要负责定义 main() 函数来运行tests和benchmarks.

## edition

edition字段定义了使用的 Rust 版本。如果未指定，则默认为package的 edition 字段。

## required-features

required-features 字段指定该test需要哪些[features]才能构建。如果未启用任何必需的feature，则将跳过。

例如：

|  |
| --- |
| [features]  # ...  postgres = []  sqlite = []  tools = []  [[test]]  name = "my-pg-tool"  required-features = ["postgres", "tools"] |

# [[bench]]

基准测试提供了两种方法：1、在lib、bin、example的源文件中使用#[bench]属性进行注解，但是仅在夜间频道可用。2、在benches目录下的基准测试。

当运行 cargo bench时，Cargo 会将benches目录下的文件中的每一个编译为单独的crate，名称默认为文件名，然后执行它们。可以在 Cargo.toml 的 [[bench]] 表数组中自定义每个bench的设置。

## name

name字段指定生成binary的名称。

## path

path 字段指定了bench的源所在的位置，相对于 Cargo.toml 文件。

## test

test 字段指示运行cargo test时，是否需要执行bench中的单元测试。默认为false。

## bench

bench 字段指示运行cargo bench时，是否需要执行该基准测试。默认值为 true。

## doc

doc 字段指示运行cargo doc时，是否需要为bench生成文档。默认值为 false。

## harness

Harness 字段表示 --test 标志将被传递给 rustc，它会自动包含 libtest 库，该库是用于收集和运行标有 #[test] 属性的测试或标有 #[bench] 属性的基准测试的驱动程序。

默认值为 true。如果设置为 false，则需要负责定义 main() 函数来运行tests和benchmarks.

## edition

edition字段定义了使用的 Rust 版本。如果未指定，则默认为package的 edition 字段。

## required-features

required-features 字段指定该bench需要哪些[features]才能构建。如果未启用任何必需的feature，则将跳过。

例如：

|  |
| --- |
| [features]  # ...  postgres = []  sqlite = []  tools = []  [[bench]]  name = "my-pg-tool"  required-features = ["postgres", "tools"] |

# [dependencies]

crate 可以依赖 crates.io 或其他注册表、git 存储库或本地文件系统上的子目录中的其他包。

[dependencies]表用于说明当前package依赖哪些其他package。

Cargo 默认配置为在 crates.io 上查找依赖项。在这种情况下，只需要名称和版本字符串。例如：

|  |
| --- |
| [dependencies]  time = "0.1.12" |

## version requirements

cargo中库的版本依赖支持以下几种方式来指定依赖版本号的要求：

* caret requirements：指定兼容版本。如果新版本号不修改主要、次要、补丁分组中最左边的非零数字，则允许更新。例如：

|  |
| --- |
| ^1.2.3 等价于 >=1.2.3, <2.0.0  ^1.2 等价于 >=1.2.0, <2.0.0  ^1 等价于 >=1.0.0, <2.0.0  ^0.2.3 等价于 >=0.2.3, <0.3.0  ^0.2 等价于 >=0.2.0, <0.3.0  ^0.0.3 等价于 >=0.0.3, <0.0.4  ^0.0 等价于 >=0.0.0, <0.1.0  ^0 等价于 >=0.0.0, <1.0.0 |

* tilde requirements：指定最小版本。如果指定了主要、次要和补丁版本或仅指定了主要和次要版本，则只允许进行补丁级别的更改。如果只指定一个主要版本，则允许次要和补丁级别的更改。例如：

|  |
| --- |
| ~1.2.3 等价于 >=1.2.3, <1.3.0  ~1.2 等价于 >=1.2.0, <1.3.0  ~1 等价于 >=1.0.0, <2.0.0 |

* wildcard requirements：允许通配符匹配的任何版本。例如：

|  |
| --- |
| \* 等价于 >=0.0.0  1.\* 等价于 >=1.0.0, <2.0.0  1.2.\* 等价于 >=1.2.0, <1.3.0 |

* comparison requirements：指定要依赖的版本范围或确切版本。例如：

|  |
| --- |
| >= 1.2.0  > 1  < 2  = 1.2.3 |

* multiple requirements：由多个逗号分割的其他requirements组成，他们之间是与的关系。例如：

|  |
| --- |
| >=1.0.0, <2.0.0  >=1.2.0, <1.3.0 |

## dependency location

默认情况下，从crates.io注册表中搜索依赖项。但是同样允许依赖其他位置的crate：

* other registries：可以使用registry指定依赖其他注册表。例如：

|  |
| --- |
| [dependencies]  some-crate = { registry = "my-registry" } |

* git repositories：允许依赖特定的git仓库。例如：

|  |
| --- |
| [dependencies]  rand = { git = "https://github.com/rust-lang-nursery/rand", branch = "next" } |

其中branch可以省略，默认为main分支。

* local path：允许依赖特定的本地lib crate。例如：

|  |
| --- |
| [dependencies]  hello\_utils = { path = "hello\_utils" } |

path指定的是相对路径，其路径相对于当前cargo.toml。

* multiple locations：可以同时指定注册表版本和 git或path。例如：

|  |
| --- |
| [dependencies]  # Uses `my-bitflags` when used locally,  # and uses version 1.0 from crates.io when published.  bitflags = { path = "my-bitflags", version = "1.0" }  # Uses the given git repo when used locally,  # and uses version 1.0 from crates.io when published.  smallvec = { git = "https://github.com/servo/rust-smallvec", version = "1.0" } |

注意：crates.io上不允许发布仅存在other registries或git repositories或local path依赖的包。

## rename dependencies

如果需要在项目中对依赖包进行重命名，可以通过以下方式执行：

|  |
| --- |
| [dependencies]  foo = "0.1"  bar = { git = "https://github.com/example/project", package = "foo" }  baz = { version = "0.1", registry = "custom", package = "foo" } |

## optional dependency

依赖项可以标记为"可选"，这意味着默认情况下不会编译它们，此时隐式地定义了一个与依赖项同名的特性。例如：

|  |
| --- |
| [dependencies]  bar = { version = "0.1", optional = true }  # 实际为  [dependencies]  bar = { version = "0.1", optional = true }  [features]  bar = [] |

注意：实际[features]表中不能存在与依赖项同名的特性。当使用--features bar标志进行编译时，才会编译bar依赖项。

如果[features]中存在与依赖项同名的特性，那么可以通过重命名依赖项来解决冲突。例如：

|  |
| --- |
| [dependencies]  bar = { version = "0.1", package = 'foo', optional = true }  [features]  foo=[] |

## enable dependency feature

如果依赖包提供了特性选择，可以通过以下方式指定dependency中要打开的特性：

|  |
| --- |
| [dependencies]  awesome = { version = "1.3.5", default-features = false, features = ["secure-password", "civet"]} |

其中default-features = false指示关闭默认的特性。

# [dev-dependencies]

[dev-dependencies]表中存放了tests、examples和benchs的依赖项。

[dev-dependencies]表的键值参考[dependencies]表。

例如：

|  |
| --- |
| [dev-dependencies]  tempdir = "0.3" |

# [build-dependencies]

[build-dependencies] 表中存放了build script的依赖项。

[build-dependencies]表的键值参考[dependencies]表。

例如：

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

# [target]

在target表下可以指定特定于平台的依赖项。依赖项可以是dependencies或dev-dependencies或build-dependencies。

使用类似 Rust 的 #[cfg] 语法将用于定义这些部分。例如：

|  |
| --- |
| [target.'cfg(windows)'.dependencies]  winhttp = "0.4.0"  [target.'cfg(unix)'.dependencies]  openssl = "1.0.1"  [target.'cfg(unix)'.dev-dependencies]  mio = "0.0.1"  [target.'cfg(unix)'.build-dependencies]  cc = "1.0.3"  [target.'cfg(target\_arch = "x86")'.dependencies]  native = { path = "native/i686" }  [target.'cfg(target\_arch = "x86\_64")'.dependencies]  native = { path = "native/x86\_64" }  [target.x86\_64-pc-windows-gnu.dependencies]  winhttp = "0.4.0"  [target.i686-unknown-linux-gnu.dependencies]  openssl = "1.0.1"  [target.bar.dependencies]  winhttp = "0.4.0"  [target.my-special-i686-platform.dependencies]  openssl = "1.0.1"  native = { path = "native/i686" } |

# [badges]

[badges]表用于指定可以在发布包时在注册网站上显示的状态标记。

例子：

|  |
| --- |
| [badges]  # The `maintenance` table indicates the status of the maintenance of  # the crate. This may be used by a registry, but is currently not  # used by crates.io. See https://github.com/rust-lang/crates.io/issues/2437  # and https://github.com/rust-lang/crates.io/issues/2438 for more details.  #  # The `status` field is required. Available options are:  # - `actively-developed`: New features are being added and bugs are being fixed.  # - `passively-maintained`: There are no plans for new features, but the maintainer intends to  # respond to issues that get filed.  # - `as-is`: The crate is feature complete, the maintainer does not intend to continue working on  # it or providing support, but it works for the purposes it was designed for.  # - `experimental`: The author wants to share it with the community but is not intending to meet  # anyone's particular use case.  # - `looking-for-maintainer`: The current maintainer would like to transfer the crate to someone  # else.  # - `deprecated`: The maintainer does not recommend using this crate (the description of the crate  # can describe why, there could be a better solution available or there could be problems with  # the crate that the author does not want to fix).  # - `none`: Displays no badge on crates.io, since the maintainer has not chosen to specify  # their intentions, potential crate users will need to investigate on their own.  maintenance = { status = "..." } |

# [features]

Cargo提供了一种机制来提供了两种功能：条件编译以及可选依赖。

## define feature

在[feature] 表中定义了一系列的特性，每个特性可以选择性包含一组定义的特性或者要打开的dependency(隐式特性)，如果一个特性包含了其他特性，那么当该特性启用时，其他特性也相应启用。

注意：注意：crates.io 对特性名称语法施加了额外的限制，即它们只能是 ASCII 字母数字字符或 \_、- 或 +。

例如：

|  |
| --- |
| [features]  bmp = []  png = []  ico = ["bmp", "png"]  webp = [] |

定义了bmp、png、ico、webp四个特性，其中如果启用了ico特性，那么bmp特性和png特性也对应被启用。

## use feature

### conditional compilation

在代码源文件中可以使用类似#[cfg(feature = "std")]注解，只有在指定特性打开的情况下，注解的代码才会被编译。

例如：

|  |
| --- |
| #[cfg(feature = "std")]  pub fn function\_that\_requires\_std() {  // ...  } |

只有std特性打开的情况下，function\_that\_requires\_std函数才会被编译。

### optional dependencies

当依赖项可以标记为"可选"，这意味着默认情况下不会编译它们，此时隐式地定义了一个与依赖项同名的特性。

例如：

|  |
| --- |
| [dependencies]  bar = { version = "0.1", optional = true }  # 实际为  [dependencies]  bar = { version = "0.1", optional = true }  [features]  bar = [] |

只有在bar特性启用的情况下，bar这个dependency才会被编译。

详见[optional dependency](#_optional_dependency)。

## enable feature

默认情况下，除非明确启用，否则包中所有特性都被禁用。这可以通过指定默认特性来设置默认启用的特性，例如：

|  |
| --- |
| [features]  default = ["ico", "webp"]  bmp = []  png = []  ico = ["bmp", "png"]  webp = [] |

上例中默认启用了所有特性。

在进行编译时，以下命令行标志可用于控制启用哪些特性：

* 使用--features标志启用特性。多个特性用逗号或空格分隔。若在工作区中，使用"package-name"/"feature-name"打开指定特性。
* 使用--all-features标志启用所有特性。
* 使用--no-default-features不启用包的默认特性。

在设置包的依赖项时，通过dependencies.<pkg\_name>.features设置依赖项中需要启用的特性，通过dependencies.<pkg\_name>.default-features关闭包的默认特性。例如：

|  |
| --- |
| [dependencies]  awesome = { version = "1.3.5", default-features = false, features = ["secure-password", "civet"]} |

详见[enable dependency feature](#_enable_dependency_feature)。

## feature unification

特性对于定义它们的包来说是独一无二的。在启用一个包上的特性不会同时启用其他包上同名的特性。

当一个依赖项被多个包使用时，Cargo将在构建它时使用该依赖项上启用的所有特性的混合。这有助于确保仅使用依赖项的单个副本。但是在极少数情况下，一个包中的两个特性可能相互不兼容，但是构建时混合特性的功能可能会产生此种冲突，因此在代码中，需要能够在编译时检测这种冲突，例如：

|  |
| --- |
| #[cfg(all(feature = "foo", feature = "bar"))]  compile\_error!("feature \"foo\" and feature \"bar\" cannot be enabled at the same time"); |

# [patch]

[patch]表定义了一系列副本用于覆盖依赖或子依赖。[patch]之后的每个键是需要被修补的git源的URL，或注册表的名称。每个条目都是正常的依赖项规范，与[dependencies]部分中的相同。例如：

|  |
| --- |
| [dependencies.baz]  git = 'https://github.com/example/baz'  [patch.crates-io]  foo = { git = 'https://github.com/example/foo' }  bar = { path = 'my/local/bar' }  [patch.'https://github.com/example/baz']  baz = { git = 'https://github.com/example/patched-baz', branch = 'my-branch' } |

上例中用 foo crate 和 bar crate 修补了 crates-io 源中的foo和bar。使用patched-baz修补 https://github.com/example/baz 源。

注意：[patch] 是可传递的，但只能在顶层定义，如果依赖项中仍然存在patch，必须在当前[patch]中重复依赖项的[patch]部分。例如：

|  |
| --- |
| // lib package  [package]  name = "my-library"  version = "0.1.0"  authors = ["..."]  [dependencies]  uuid = "1.0.1"  [patch.crates-io]  uuid = { git = 'https://github.com/uuid-rs/uuid' }  // binary package  [package]  name = "my-binary"  version = "0.1.0"  authors = ["..."]  [dependencies]  my-library = { git = 'https://example.com/git/my-library' }  uuid = "1.0"  [patch.crates-io]  uuid = { git = 'https://github.com/uuid-rs/uuid' } |

# [profile]

profile提供了一种更改编译器设置的方法，可以影响优化和调试符号等内容。

Cargo 有4个内置profile：dev、release、test 和 bench。会根据正在运行的命令、正在构建的包和目标以及 --release 等命令行标志自动选择profile，选择过程如下：

* cargo build/rustc/check/run命令编译lib或bin目标时，默认使用dev profile。指定--release标志，则使用release profile。
* cargo install命令编译lib或bin目标时，默认使用release profile。指定--debug标志，则使用dev profile。
* test 目标编译默认使用test profile。指定--release标志，则使用bench profile。
* bench 目标编译默认使用bench profile。当使用cargo build --debug编译bench 目标时，使用test profile。
* cargo test/bench命令编译时，test/bench目标使用test/bench profile，但是他们的依赖项仍然使用dev/release profile编译。

在Cargo.toml的[profile]表中可以更改profile设置。在每个命名配置文件中，可以使用键/值对更改单独的设置，如下所示：

|  |
| --- |
| [profile.dev]  opt-level = 1 # Use slightly better optimizations.  overflow-checks = false # Disable integer overflow checks. |

注意：Cargo 仅查看工作区根目录下 Cargo.toml中的profile设置，依赖项中定义的profile设置将被忽略。

## profile settings

### opt-level

opt-level 设置控制 -C opt-level 标志，该标志控制优化级别。有效的选项如下：

* 0：没有优化
* 1：基本优化
* 2：一些优化
* 3：所有优化
* "s"：优化二进制大小
* "z"：优化二进制大小，同时关闭循环矢量化

### debug

debug设置控制 -C debuginfo 标志，该标志控制已编译二进制文件中包含的调试信息量。有效的选项如下：

* 0 或 false：没有调试信息
* 1：仅行表
* 2 或 true：完整的调试信息

### split-debuginfo

split-debuginfo 设置控制 -C split-debuginfo 标志，该标志控制调试信息（如果生成）是放置在可执行文件本身中还是与其相邻。

此选项是一个字符串，可接受的值与编译器接受的值相同。

### debug-assertions

debug-assertions 设置控制 -C debug-assertions 标志，它打开或关闭 cfg(debug\_assertions) 条件编译。调试断言启用标准库中的debug\_assert!宏。有效的选项如下：

* true: enabled
* false: disabled

### overflow-checks

overflow-checks设置控制 -C overflow-checks标志，该标志控制运行时整数溢出的行为。 当启用溢出检查时，溢出时会发生恐慌。有效的选项如下：

* true: enabled
* false: disabled

### lto

lto 设置控制 -C lto 标志，该标志控制 LLVM 的链接时间优化。 LTO 可以使用整个程序分析生成更好的优化代码，但代价是链接时间更长。有效的选项如下：

* false：在本地 crate 上通过代码生成单元执行"thin" LTO。如果代码生成单元为 1 或 opt-level 为 0，则不执行 LTO。
* true或"fat"：执行"fat" LTO，它尝试对依赖图中的所有 crate 执行优化。
* "thin"：执行"thin" LTO。 这类似于“fat”，但运行时间大大减少，同时仍能实现类似于“fat”的性能提升。
* "off"：禁用 LTO。

### panic

panic设置控制 -C panic标志，该标志控制使用哪种恐慌策略。有效的选项如下：

* "unwind"：在panic时展开堆栈。
* "abort"：在panic时终止进程。

当设置为"unwind"时，实际值取决于目标平台的默认值。比如NVPTX平台不支持unwinding，所以一直使用"abort"。

测试、基准测试、构建脚本和 proc 宏会忽略panic设置。 rustc 测试工具目前需要展开行为。

### incremental

incremental设置控制 -C incremental标志，该标志控制是否启用增量编译。 增量编译会导致 rustc 将附加信息保存到磁盘，这些信息将在重新编译 crate 时重复使用，从而缩短重新编译时间。 附加信息存储在目标目录中。有效的选项如下：

* true: enabled
* false: disabled

### codegen-units

codegen-units 设置控制 -C codegen-units 标志，该标志控制将 crate 分成多少个“代码生成单元”。 更多的代码生成单元允许并行处理更多的 crate，可能会减少编译时间，但可能会产生更慢的代码。

此选项采用大于 0 的整数。增量构建的默认值为 256，非增量构建的默认值为 16。

### rpath

rpath 设置控制 -C rpath 标志，该标志控制是否启用 rpath。

## default profiles

### dev

dev profile用于正常的开发和调试。其默认值如下：

|  |
| --- |
| [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 |

### release

release profile用于发布。其默认值如下：

|  |
| --- |
| [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 |

### test

test profile用于构建测试。其默认值如下：

|  |
| --- |
| [profile.test]  opt-level = 0  debug = 2  split-debuginfo = '...' # Platform-specific.  debug-assertions = true  overflow-checks = true  lto = false  panic = 'unwind' # This setting is always ignored.  incremental = true  codegen-units = 256  rpath = false |

### bench

bench profile用于构建基准测试。其默认值如下：

|  |
| --- |
| [profile.bench]  opt-level = 3  debug = false  split-debuginfo = '...' # Platform-specific.  debug-assertions = false  overflow-checks = false  lto = false  panic = 'unwind' # This setting is always ignored.  incremental = false  codegen-units = 16  rpath = false |

## override profile

可以覆盖指定包和构建时crate的profile设置。例如：

|  |
| --- |
| # The `foo` package will use the -Copt-level=3 flag.  [profile.dev.package.foo]  opt-level = 3 |

包名称实际上是一个包 ID 规范，因此可以使用 [profile.dev.package."foo:2.1.0"] 等语法来定位包的各个版本。

要覆盖所有依赖项（但不是任何工作区成员）的设置，请使用"\*"包名称：

|  |
| --- |
| # Set the default for dependencies.  [profile.dev.package."\*"]  opt-level = 2 |

默认情况下，所有profile都不会优化build的依赖项（build script、proc 宏及其依赖项）。build override的默认设置是：

|  |
| --- |
| [profile.dev.build-override]  opt-level = 0  codegen-units = 256  [profile.release.build-override]  opt-level = 0  codegen-units = 256 |

可以通过如下方式进行override：

|  |
| --- |
| # Set the settings for build scripts and proc-macros.  [profile.dev.build-override]  opt-level = 3 |

注意：overrides 不能指定 panic、lto 或 rpath 设置。

# [workspace]

workspace是一个或多个共享公共依赖项解析（具有共享的 Cargo.lock）、输出目录和各种设置（例如配置文件）的包的集合。作为工作区一部分的包称为工作区成员。workspace有两种风格：

* root package：可以通过向 Cargo.toml 添加 [workspace] 表来创建工作区。这可以添加到已经定义了 [package] 的 Cargo.toml 中，在这种情况下，当前包是工作区的根包。工作空间根目录是工作空间的 Cargo.toml 所在的目录。
* virtual manifest：可以使用 [workspace] 部分但没有 [package] 部分创建 Cargo.toml 文件。这称为virtual manifest。当没有"main"包时，这通常很有用，或者您希望将所有包组织在单独的目录中。

workspace的主要特征如下：

* 所有包共享一个位于工作区根目录中的通用 Cargo.lock 文件。
* 所有包共享一个公共输出目录，该目录默认为工作区根目录中名为 target 的目录。
* Cargo.toml 中的 [patch] 和 [profile.\*] 部分仅在根清单中被识别，在成员 crate 的清单中被忽略。

## workspace member

在工作区目录中的所有使用本地路径的依赖项会自动成为该workspace的成员。一个空的 [workspace] 表可以与 [package] 一起使用，以方便地创建一个包及含包其所有路径依赖项的工作空间。

另外在[workspace] 表中使用member键定义哪些包是工作区的成员：

|  |
| --- |
| [workspace]  members = ["member1", "path/to/member2", "crates/\*"] |

成员列表支持使用 \* 和 ? 等典型文件名 glob 模式匹配多个路径的 glob。

另外可以在[workspace] 表中使用exclude 键防止路径包含在工作区中：

|  |
| --- |
| [workspace]  members = ["member1", "path/to/member2", "crates/\*"]  exclude = ["crates/foo", "path/to/other"] |

## workspace selection

当在工作区的子目录中时，Cargo 将自动搜索带有 [workspace] 定义的 Cargo.toml 文件的父目录，以确定要使用的工作区。

package.workspace键可用于将成员crate 指向工作区。如果成员不在工作区根目录的子目录中，手动设置会很有用。详见[package.workspace](#_workspace)。

## package selection

在工作区中，与包相关的cargo命令（例如 cargo build）可以使用 -p/--package 或 --workspace 命令行标志来确定要操作的包。

如果这些标志都没有指定，Cargo 将使用当前工作目录中的包。 如果当前目录是虚拟工作区，它将应用于所有成员（就像在命令行中指定了 --workspace 一样）。

可以指定 workspace.default-members 键来设置要在工作空间根目录中操作的成员，并且不使用包选择标志：

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

## workspace metadata

Workspace.metadata 表被 Cargo 忽略，不会产生警告。此部分可用于希望在 Cargo.toml 中存储工作区配置的工具。 例如：

|  |
| --- |
| [workspace]  members = ["member1", "member2"]  [workspace.metadata.webcontents]  root = "path/to/webproject"  tool = ["npm", "run", "build"]  # ... |