# 概述

一些包需要编译第三方非 Rust 代码，例如 C 库。 其他包需要链接到 C 库，这些库可以位于系统上，也可能需要从源代码构建。 其他人仍然需要在构建之前实现代码生成等功能。

Cargo 的目标不是替换针对这些任务进行了良好优化的其他工具，但它确实将它们与自定义构建脚本集成在一起。 在包的根目录中放置一个名为 build.rs 的文件将导致 Cargo 编译该脚本并在构建包之前执行它。例如：

|  |
| --- |
| // Example custom build script.  fn main() {  // Tell Cargo that if the given file changes, to rerun this build script.  println!("cargo:rerun-if-changed=src/hello.c");  // Use the `cc` crate to build a C file and statically link it.  cc::Build::new()  .file("src/hello.c")  .compile("hello");  } |

构建脚本的使用案例有：

* 构建捆绑的 C 库。
* 在主机系统上查找 C 库。
* 从描述生成 Rust 模块。
* 执行 crate 所需的任何特定于平台的配置。

# 生命周期

在构建包之前，Cargo 会将构建脚本编译为可执行文件（如果尚未构建）。 然后它将运行脚本，该脚本可以执行任意数量的任务。该脚本可以通过打印以 "cargo:" 为前缀的特殊格式命令到 stdout 来与 Cargo 通信。

如果构建脚本的任何源文件或依赖项发生变化，构建脚本将被重建。

默认情况下，如果包中的任何文件发生更改，在进行构建时，Cargo 都将重新运行构建脚本。通常最好使用 rerun-if 命令，在下面的更改检测部分中描述，以缩小触发构建脚本再次运行的焦点。

一旦构建脚本成功完成执行，包的其余部分将被编译。 如果出现错误，构建脚本应以非零退出代码退出以停止构建，在这种情况下，构建脚本的输出将显示在终端上。

# 构建脚本输入

当构建脚本运行时，构建脚本有许多输入，所有输入都以环境变量的形式传递。具体见《Cargo环境变量》。

除了环境变量，构建脚本的当前目录是构建脚本包的源目录。

# 构建脚本输出

构建脚本将输出文件保存在 OUT\_DIR 环境变量中指定的目录中。脚本不应修改该目录之外的任何文件。

构建脚本通过打印以 "cargo:" 为前缀的特殊格式命令到 stdout 来与 Cargo 通信。Cargo 会将其解释为影响包编译的指令。 在正常编译期间，脚本的输出对终端是隐藏的。 如果想直接在终端中查看输出，请使用 -vv 标志。

构建脚本打印到 stdout 的所有行都被写入一个文件，如 target/debug/build/<pkg>/output。stderr 输出也保存在同一目录中。

以下是 Cargo 识别的包编译指令：

* cargo:rerun-if-changed=PATH — 指示何时重新运行脚本。
* cargo:rerun-if-env-changed=VAR — 指示何时重新运行脚本。
* cargo:rustc-link-lib=[KIND=]NAME — 添加要链接的库。
* cargo:rustc-link-search=[KIND=]PATH — 添加库搜索路径。
* cargo:rustc-flags=FLAGS — 将标志传递给编译器。
* cargo:rustc-cfg=KEY[="VALUE"] — 启用编译时的 cfg 设置。
* cargo:rustc-env=VAR=VALUE — 设置环境变量。
* cargo:rustc-cdylib-link-arg=FLAG — 将自定义标志传递给 cdylib crate的链接器。
* cargo:warning=MESSAGE — 在终端上显示警告。
* cargo:KEY=VALUE — 元数据，由链接脚本使用。

## cargo:rerun-if-changed=PATH

rerun-if-changed 指令告诉 Cargo 如果给定路径上的文件已更改，则重新运行构建脚本。

目前，Cargo 仅使用文件系统上次修改的“mtime”时间戳来确定文件是否已更改。 它与构建脚本上次运行时的内部缓存时间戳进行比较。

如果路径指向一个目录，它将扫描整个目录以进行任何修改。

如果构建脚本本质上不需要在任何情况下重新运行，那么发出 cargo:rerun-if-changed=build.rs 是一种防止它重新运行的简单方法

## cargo:rerun-if-env-changed=NAME

rerun-if-env-changed 指令告诉 Cargo 如果给定名称的环境变量的值已更改，则重新运行构建脚本。

请注意，这里的环境变量用于全局环境变量，如 CC 等，不必将其用于 Cargo 设置的 TARGET 等环境变量。

## cargo:rustc-link-lib=[KIND=]NAME

rustc-link-lib 指令告诉 Cargo 使用编译器的 -l 标志链接给定的库。 这通常用于使用 FFI 链接本机库。

-l 标志只传递给包的库目标(lib crate)，如果没有库目标，在这种情况下它将被传递给所有目标。

可选的 KIND 可以是 dylib、静态或框架之一。

## cargo:rustc-link-search=[KIND=]PATH

rustc-link-search 指令告诉 Cargo 将 -L 标志传递给编译器以将目录添加到库搜索路径。

可选的 KIND 可以是dependency、crate、native、framework或all。

如果路径在 OUT\_DIR 中，会被添加到动态库搜索路径环境变量中。

## cargo:rustc-flags=FLAGS

rustc-flags 指令告诉 Cargo 将给定的空格分隔标志传递给编译器。 这只允许使用 -l 和 -L 标志，相当于使用 rustc-link-lib 和 rustc-link-search。

## cargo:rustc-cfg=KEY[="VALUE"]

rustc-cfg 指令告诉 Cargo 将给定的值传递给编译器的 --cfg 标志。 这可用于编译时检测以启用条件编译。

注意：这不会影响 Cargo 的依赖项解析。也不能用于启用可选依赖项或启用其他 Cargo 特性。

Cargo 特性使用表单 feature="foo"。 使用此标志传递的 cfg 值不限于该形式，并且可以仅提供单个标识符或任何任意键/值对。 例如，发出 cargo:rustc-cfg=abc 将允许代码使用 #[cfg(abc)] （注意缺少 feature=）。 或者任意键/值对可以与 = 符号一起使用，例如 cargo:rustc-cfg=my\_component="foo"。 键应该是一个 Rust 标识符，值应该是一个字符串。

## cargo:rustc-env=VAR=VALUE

rustc-env 指令告诉 Cargo 在编译包时设置给定的环境变量。代码中可以通过 env! 检索该值！

## cargo:rustc-cdylib-link-arg=FLAG

rustc-cdylib-link-arg 指令告诉 Cargo 将 -C link-arg=FLAG 选项传递给编译器，但仅限于构建 cdylib 库目标时。

## cargo:warning=MESSAGE

warning指令告诉 Cargo 在构建脚本完成运行后显示警告。警告仅针对路径依赖项才会显示。

# Build Dependencies

构建脚本也可以依赖于其他基于 Cargo 的 crate。 依赖项是通过清单的 build-dependencies 部分声明的。例如：

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

构建脚本无权访问dependencies或 dev-dependencies 部分中列出的依赖项。此外，除非在 [dependencies] 表中明确添加，否则构建依赖项对包本身不可用。

# The links Key

package.links 键可以在 Cargo.toml 清单中设置，以声明包与指定的本地库链接。这个清单键的目的是让 Cargo 了解包所具有的一组本机依赖项，​​并提供一个在包构建脚本之间传递元数据的原则系统。

例如：

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

此清单声明包链接到 libfoo 本机库。使用 links 键时，包必须有构建脚本，构建脚本应使用 rustc-link-lib 指令链接库。

Cargo 要求每个links值最多有一个package使用。换句话说，禁止将两个包链接到同一个本地库。这有助于防止crate之间出现重复符号。

# \*-sys Packages

一些链接到系统库的 Cargo 包具有带有 -sys 后缀的命名约定。 任何名为 foo-sys 的包都应该提供两个主要功能：

* 该library crate会链接到本地库 libfoo。 在于从源代码构建之前，这通常会探测当前系统的 libfoo。
* 该library crate 会为 libfoo 中的函数提供声明，但不提供绑定或更高级别的抽象。

\*-sys 包提供了一组用于链接到本机库的通用依赖项。 使用这种与本地库相关的包的约定可以获得许多好处：

* 对 foo-sys 的常见依赖减轻了每个链接值一个包的规则。
* 其他 -sys 包可以利用 DEP\_NAME\_KEY=value 环境变量来更好地与其他包集成。
* 公共依赖允许集中逻辑发现 libfoo 本身（或从源代码构建它）。
* 这些依赖项很容易被覆盖。

# Overriding Build Scripts

如果cargo.toml清单包含 links 键，则 Cargo 支持使用自定义库覆盖指定的构建脚本。

要覆盖构建脚本，请将以下配置放置在任何可接受的 Cargo 配置位置：

|  |
| --- |
| [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" |

使用此配置后，如果包声明它链接到 foo，则不会编译或运行构建脚本，而是直接使用指定的元数据。

注意：warning、rerun-if-changed、和rerun-if-env-changed键不应该被使用，他们将会被忽略。