TWEAG by Modus Create

BUILDING A RUST WORKSPACE WITH BAZEL

27 July 2023 — by Ilya Polyakovskiy

bazel cargo rust

The vast majority of the Rust projects are using Cargo as a build tool. Cargo is great when you are developing and packaging a single Rust library or application, but when it comes to a fast-growing and complex workspace, one could be attracted to the idea of using a more flexible and scalable build system. Here is a nice article elaborating on why Cargo should not be considered as a such a build system. But there are a handful of reasons to consider Bazel:

- Bazel's focus on hermeticity and aggressive caching allows us to improve median build and test times, especially for a single Pull Request against a relatively large codebase.
- Remote caching and execution can significantly reduce the amount of Rust compilation done locally on developers' machines.
- The polyglot nature of Bazel allows expressing connections between Rust code and targets written in other languages in a much more simple and straightforward manner. Be it building Python packages from Rust code with PyO3, connecting JavaScript code with WASM compiled from Rust, or managing Rust crates incorporating FFI calls from a C-library; with Bazel you have a solid solution.

That's all great, but how am I going to make my Cargo workspace use Bazel? To show this, I'm going to take an open source Rust project and guide you through the steps to migrate it to Bazel.

RIPGREP

I chose ripgrep, since it is well-known in the Rust community. The project is organized as a Cargo workspace consisting of several crates:

```toml
[[bin]]
bench = false
path = "crates/core/main.rs"
name = "rg"

[[test]]
name = "integration"
path = "tests/tests.rs"

[workspace]
members = [
  "crates/globset",
  "crates/grep",
  "crates/cli",
  "crates/matcher",
  "crates/pcre2",
  "crates/printer",
  "crates/regex",
  "crates/searcher",
  "crates/ignore",
]
```

Let's see what it will take to build and test this workspace with Bazel.

SETTING UP A WORKSPACE

Luckily, there is a Bazel extension for building Rust projects: `rules_rust`. It supports handling Rust toolchains, building Rust libraries, binaries and `proc_macro` crates, running `build.rs` scripts, automatically converting Cargo dependencies to Bazel and a lot more.

First, we'll create a Bazel workspace. If you are not familiar with Bazel and Bazel workspaces, we have an article on the Tweag blog covering this topic.

If you would like to follow along, you can use the ripgrep fork here, which is based off of commit 4fcb1b2202 of the upstream project. The builds were tested with Bazel version 6.1.2.

So, let's start with creating a `WORKSPACE` file and importing `rules_rust`:

```
load("@bazel_tools//tools/build_defs/repo:http.bzl", "http_archive")

http_archive(
    name = "rules_rust",
    sha256 = "48e715be2368d79bc174efdb12f34acfc89abd7ebfcbffbc02568f
    urls = ["https://github.com/bazelbuild/rules_rust/releases/downl
)

load("@rules_rust//rust:repositories.bzl", "rules_rust_dependencies"

rules_rust_dependencies()

rust_register_toolchains(
    edition = "2018",
)
```

Here we've specified the edition. Editions are used by the Rust team to perform changes which are backwards incompatible, you can leave it unspecified and just use the latest one. `ripgrep` is using "2018", and we are doing the same. There is also a standard mechanism of distributing and updating the Rust toolchain through channels. `rust_register_toolchains` allows us to set the Rust toolchain version for all three channels (stable, beta and nightly) we want to use for our workspace. Regarding the `rustc` version, if we look at the `ripgrep` CI configuration, we notice it uses the nightly toolchain pinned by the `dtolnay/rust-toolchain` Github action. `rust_register_toolchains` allows us to omit the `versions` attribute. In this case, we will end up using the stable and nightly versions pinned by the version of `rules_rust`. One could reason this behavior is closer to what is done by `ripgrep`'s CI configuration, but I would rather suggest being more explicit for the sake of reproducibility and clarity:

```
rust_register_toolchains(
    edition = "2018",
    versions = [
        "1.70.0",
        "nightly/2023-06-01",
    ],
)
```

Here we've defined explicitly which versions of stable and nightly Rust we want to use in our build. By default, Bazel invokes `rustc` from the `stable` channel. For switching to `nightly`, we need to invoke the build with `--@rules_rust//rust/toolchain/channel=nightly` flag. To make `nightly` default, we can create a `.bazelrc` file inside the repository root and add this line:

```
build --@rules_rust//rust/toolchain/channel=nightly
```

EXTERNAL DEPENDENCIES

Cargo makes it easy to specify the dependencies and build your Rust project on top of them. In Bazel we also need to explicitly declare all the external dependencies and it would be extremely painful to manually write `BUILD` files for every Rust crate our project depends on. Luckily, there are some tools to generate Bazel targets from `Cargo.lock` files, for external dependencies. The rules_rust documentation mentions two: crate_universe and cargo-raze. Since crate_universe is a successor to cargo-raze, included into the rules_rust package, I'm going to focus on this tool.

To configure it we need to add the following to our WORKSPACE file:

```
load("@rules_rust//crate_universe:repositories.bzl", "crate_universe_

crate_universe_dependencies()

load("@rules_rust//crate_universe:defs.bzl", "crates_repository")

crates_repository(
    name = "crate_index",
    cargo_lockfile = "//:Cargo.lock",
    lockfile = "//:cargo-bazel-lock.json",
    manifests = [
        "//:Cargo.toml",
        "//:crates/globset/Cargo.toml",
        "//:crates/grep/Cargo.toml",
        "//:crates/cli/Cargo.toml",
        "//:crates/matcher/Cargo.toml",
        "//:crates/pcre2/Cargo.toml",
        "//:crates/printer/Cargo.toml",
        "//:crates/regex/Cargo.toml",
        "//:crates/searcher/Cargo.toml",
        "//:crates/ignore/Cargo.toml",
    ],
)

load("@crate_index//:defs.bzl", "crate_repositories")

crate_repositories()
```

crates_repository creates a repository_rule containing targets for all external dependencies explicitly mentioned in the Cargo.toml files as dependencies. We need to specify several attributes:

- `cargo_lockfile` : the actual `Cargo.lock` of the Cargo workspace.
- `lockfile` : this is the file used by `crate_universe` to store metadata gathered from Cargo files. Initially it should be created empty, then it will be automatically updated and maintained by `crate_universe` .
- `manifests` : the list of `Cargo.toml` files in the workspace.

Let's create an empty lock file for `crate_universe` and the empty `BUILD.bazel` file for a so far empty Bazel package:

```
$ touch cargo-bazel-lock.json BUILD.bazel
```

Now we can run `bazel sync` to pin cargo dependencies as Bazel targets[1]:

```
$ CARGO_BAZEL_REPIN=1 bazel sync --only=crate_index
```

You should run this command whenever you update dependencies in Cargo files. We can also use a Bazel query to examine targets generated by `crate_universe` :

```
$ bazel query @crate_index//...
@crate_index//:aho-corasick
@crate_index//:base64
@crate_index//:bstr
@crate_index//:bytecount
@crate_index//:clap
@crate_index//:crossbeam-channel
@crate_index//:encoding_rs
@crate_index//:encoding_rs_io
@crate_index//:fnv
@crate_index//:glob
@crate_index//:jemallocator
@crate_index//:lazy_static
@crate_index//:log
```

```
@crate_index//:memchr
@crate_index//:memmap
@crate_index//:pcre2
@crate_index//:regex
@crate_index//:regex-automata
@crate_index//:regex-syntax
@crate_index//:same-file
@crate_index//:serde
@crate_index//:serde_derive
@crate_index//:serde_json
@crate_index//:srcs
@crate_index//:termcolor
@crate_index//:thread_local
@crate_index//:walkdir
@crate_index//:winapi-util
```

Here we can see that the `@crate_index` repository consists of targets for dependencies explicitly mentioned in the Cargo files.

WRITING BUILD FILES AND CARGO-BAZEL PARITY

Now it's time to build some crates in our workspace. Let's look at the `matcher` crate for example, we will handle the rest of the crates the same way.

```
[package]
name = "grep-matcher"
version = "0.1.6"  #:version
authors = ["Andrew Gallant <jamslam@gmail.com>"]
description = """
A trait for regular expressions, with a focus on line oriented searc
"""
documentation = "https://docs.rs/grep-matcher"
homepage = "https://github.com/BurntSushi/ripgrep/tree/master/crates
repository = "https://github.com/BurntSushi/ripgrep/tree/master/crat
```

```
readme = "README.md"
keywords = ["regex", "pattern", "trait"]
license = "Unlicense OR MIT"
autotests = false
edition = "2018"

[dependencies]
memchr = "2.1"

[dev-dependencies]
regex = "1.1"

[[test]]
name = "integration"
path = "tests/tests.rs"
```

To build this crate with Bazel we create `crate/matcher/BUILD.bazel` :

```
load("@rules_rust//rust:defs.bzl", "rust_library")

rust_library(
    name = "grep-matcher",
    srcs = glob([
        "src/**/*.rs",
    ]),
    deps = [
        "@crate_index//:memchr,
    ],
    proc_macro_deps = [],
    visibility = ["//visibility:public"],
)
```

Here we simply define the Rust library according to the documentation. Bazel requires us to specify all dependencies explicitly, and since we are generating a `@crate_index` repository based on Cargo files, to add new dependencies, we'll have to change Cargo files, run `bazel sync` and update `BUILD` files accordingly. This will create two sources of the same information that need to be synchronized manually, which is inconvenient and error-prone. Luckily, there are some handy functions in `crate_universe` to address this. We can rewrite the same `BUILD` file like this:

```
load("@crate_index//:defs.bzl", "aliases", "all_crate_deps")
load("@rules_rust//rust:defs.bzl", "rust_library")

rust_library(
    name = "grep-matcher",
    srcs = glob([
        "src/**/*.rs",
    ]),
    aliases = aliases(),
    deps = all_crate_deps(),
    proc_macro_deps = all_crate_deps(
        proc_macro = True,
    ),
    visibility = ["//visibility:public"],
)
```

We don't need to specify dependencies explicitly any more. The `all_crate_deps` function returns the list of dependencies for the crate defined in the same directory as a `BUILD` file this function was called from, based on the gathered metadata saved in the `cargo-bazel-lock.json` file. To see the `BUILD` file with these functions expanded one could run:

```
$ bazel query //crates/matcher:grep-matcher  --output=build
rust_library(
  name = "grep-matcher",
  visibility = ["//visibility:public"],
  aliases = {},
  deps = ["@crate_index__memchr-2.5.0//:memchr"],
  proc_macro_deps = [],
  srcs = ["//crates/matcher:src/interpolate.rs", "//crates/matcher:s
)
```

We need `aliases = aliases()` here in case the crate is using dependency renaming. There is an example of it in the `searcher` crate:

```
memmap = { package = "memmap2", version = "0.5.3" }
```

Otherwise we would have to write explicitly:

```
aliases = {
    "@crate_index//:memmap2": "memmap",
}
```

This allows us to have Cargo files as a single source of external dependencies, so when we need to add a new dependency, for example, we could just use `cargo add` and repin Bazel dependencies with `CARGO_BAZEL_REPIN=1 bazel sync --only=crate_index`. An important limitation is that `crate_universe` ignores `path` dependencies. This means we need to manually specify internal dependencies inside the workspace. We'll see how this works later.

Next, it's time to migrate the tests. Crate `grep-matcher` has two types of tests: unit and integration. Unit tests are defined in the source files of each library, while integration tests have their own source files. Let's migrate the unit tests first:

```
load("@rules_rust//rust:defs.bzl", "rust_test")
rust_test(
    name = "tests",
    crate = ":grep-matcher",
    aliases = aliases(
        normal_dev = True,
        proc_macro_dev = True,
    ),
    deps = all_crate_deps(
        normal_dev = True,
    ),
    proc_macro_deps = all_crate_deps(
        proc_macro_dev = True,
    ),
)
```

We are using the `crate` attribute instead of `srcs` here because those tests don't have their own sources.

And here is a declaration for integration tests:

```
rust_test(
    name = "integration",
    srcs = glob([
        "tests/**/*.rs",
    ]),
    crate_root = "tests/tests.rs",
    deps = all_crate_deps(
        normal_dev = True
    ) + [
        ":grep-matcher",
    ],
    proc_macro_deps = all_crate_deps(
        proc_macro_dev = True
    ),
)
```

Here we've added the dependency on the crate and used `crate_root` since the name of the target and the name of the main file are different.

BUILD AND TEST RIPGREP BINARY

Let's look at the root `Cargo.toml` file:

```
build = "build.rs"
...
[[bin]]
bench = false
path = "crates/core/main.rs"
name = "rg"
...
[[test]]
name = "integration"
path = "tests/tests.rs"
...
[dependencies]
...
grep = { version = "0.2.12", path = "crates/grep" }
ignore = { version = "0.4.19", path = "crates/ignore" }
```

First, we need to deal with Cargo invoking `build.rs` before compiling the binary. And again, there is a rule created specifically for this in `rules_rust`:

```
load("@rules_rust//cargo:defs.bzl", "cargo_build_script")

cargo_build_script(
    name = "build",
    srcs = [
        "build.rs",
        "crates/core/app.rs",
    ],
    deps = all_crate_deps(
        normal = True,
        build = True,
    ) + [
        "//crates/grep",
        "//crates/ignore",
    ],
    crate_root = "build.rs",
)
```

The `build.rs` file imports code from `crates/core/app.rs` to generate shell completions for example, which in turn depends on some crates from the workspace. Now we can build and test the `rg` binary:

```
rust_binary(
    name = "rg",
    srcs = glob([
        "crates/core/**/*.rs",
    ]),
    aliases = aliases(),
    deps = all_crate_deps() + [
        "//crates/grep",
        "//crates/ignore",
        ":build",
    ],
    proc_macro_deps = all_crate_deps(
        proc_macro = True,
    ),
    visibility = ["//visibility:public"],
)
```

Let’s see how it works:

```
$ bazel build //:rg
INFO: Analyzed target //:rg (156 packages loaded, 2780 targets confi
INFO: Found 1 target...
Target //:rg up-to-date:
  bazel-bin/rg
INFO: Elapsed time: 111.921s, Critical Path: 101.86s
INFO: 238 processes: 115 internal, 123 linux-sandbox.
INFO: Build completed successfully, 238 total actions
$ ./bazel-bin/rg
error: The following required arguments were not provided:
    <PATTERN>

USAGE:

    rg [OPTIONS] PATTERN [PATH ...]
    rg [OPTIONS] -e PATTERN ... [PATH ...]
    rg [OPTIONS] -f PATTERNFILE ... [PATH ...]
    rg [OPTIONS] --files [PATH ...]
    rg [OPTIONS] --type-list
    command | rg [OPTIONS] PATTERN
    rg [OPTIONS] --help
    rg [OPTIONS] --version

For more information try --help
```

Voilà! You can also find artifacts created by `build.rs` in the `./bazel-bin/build.out_dir` directory.

Now we can add top-level tests in the same way we did for `grep-matcher` and other crates, with the only difference being: the `rg` integration tests are using some data files in `tests/data` directory, so we need to specify this explicitly in the `BUILD` file:

```
rust_test(
    name = "tests",
    crate = ":rg",
    deps = all_crate_deps(
        normal_dev = True,
    ),
    proc_macro_deps = all_crate_deps(
        proc_macro_dev = True,
    ),
)

rust_test(
    name = "integration",
    srcs = glob([
        "tests/**/*.rs",
    ]),
    deps = all_crate_deps(
        normal = True,
        normal_dev = True,
    ),
    data = glob([
        "tests/data/**",
    ]),
    proc_macro_deps = all_crate_deps(
        proc_macro_dev = True
    ),
    crate_root = "tests/tests.rs",
)
```

Now we can execute all our tests:

```
$ bazel test //...
```

IMPROVING HERMETICITY

One of Bazel's main concerns is the hermeticity of builds. It means Bazel aims to always return the same output for the same input source code and product configuration by isolating the build from changes to the host system. One of the major sources of non-hermetic builds in Rust, in turn, is Cargo build scripts executed when compiling dependencies. For example, crates with

Rust bindings to some well-known C-libraries usually have a build script that looks up a dynamic library to link globally in your system. Usual practice for such libraries is to use `pkg-config` crate for this lookup, so if you have `pkg-config` somewhere in the dependency chain, chances are high that your build is not hermetic. `crate_universe` generates `cargo_build_script` targets for dependencies automatically, so we'll have the same problem in our Bazel build. Let's look at what we have so far:

```
$ bazel query "deps(//:rg)" | grep pkg_config
@crate_index__pkg-config-0.3.27//:pkg_config
```

Okay, let's try to figure out which library brings in this dependency:

```
$ bazel query "allpaths(//:rg, @crate_index__pkg-config-0.3.27//:pkg_
//:build
//:build_
//:rg
//crates/grep:grep
//crates/pcre2:grep-pcre2
@crate_index__pcre2-0.2.3//:pcre2
@crate_index__pcre2-sys-0.2.5//:build_script_build
@crate_index__pcre2-sys-0.2.5//:pcre2-sys_build_script
@crate_index__pcre2-sys-0.2.5//:pcre2-sys_build_script_
@crate_index__pcre2-sys-0.2.5//:pcre2_sys
@crate_index__pkg-config-0.3.27//:pkg_config
```

Now it's clear there is only one library using it: `pcre2-sys`. If we look at its build script we'll see that it looks for `libpcre2` unless the environment variable `PCRE2_SYS_STATIC` is set. In this case, it builds the static library `libpcre2.a` from sources and links with it. This means that in order to make our build hermetic, we need to pass this environment variable to the build script automatically generated by `crate_universe` for `pcre2-sys`. Fortunately, there is a tool for this in `crate_universe`. Let's go back to the `WORKSPACE` file and change it a bit:

```
load("@rules_rust//crate_universe:defs.bzl", "crates_repository", "c

crates_repository(
    name = "crate_index",
    cargo_lockfile = "//:Cargo.lock",
    lockfile = "//:cargo-bazel-lock.json",
    manifests = [
        "//:Cargo.toml",
        "//:crates/globset/Cargo.toml",
        "//:crates/grep/Cargo.toml",
        "//:crates/cli/Cargo.toml",
        "//:crates/matcher/Cargo.toml",
        "//:crates/pcre2/Cargo.toml",
        "//:crates/printer/Cargo.toml",
        "//:crates/regex/Cargo.toml",
        "//:crates/searcher/Cargo.toml",
        "//:crates/ignore/Cargo.toml",
    ],
    annotations = {
        "pcre2-sys": [crate.annotation(
            build_script_env = {
                "PCRE2_SYS_STATIC": "1",
            }
        )],
    },
)
```

IMPORTANT NOTE ABOUT INCREMENTALITY

Since Rust 1.24 `rustc` has incremental compilation feature. Unfortunately, it is not supported in `rules_rust` yet, which makes a crate the smallest possible compilation unit for Rust project in Bazel. For some crates, it could significantly increase the compilation time for an arbitrary code change. Nevertheless there is some ongoing work in `rules_rust` to support incremental features of `rustc` : here, here and here

CONCLUSION

Now we have a fully hermetic Bazel build for `ripgrep` [2]. You can find the complete implementation here. It keeps Cargo files as a source of truth regarding external dependencies and project structure, which helps with managing those dependencies or IDE setup, since IDEs can use Cargo files to configure themselves. There is still work to be done to automate `path` dependencies and there are some projects out there aiming for that. Maybe we'll look at it closer next time. Stay tuned!

---

1. You can find more details about this command here↩
2. Well, technically not completely hermetic; Bazel still picks up CC toolchain from the system. There are some resources regarding hermetic CC toolchains in Bazel here, here and here.↩

BEHIND THE SCENES

IP

ILYA POLYAKOVSKIY

TECH GROUP

Research, create, improve and maintain programming languages and their tooling to enhance developer productivity and to deliver reliable, maintainable, correct and performant software with minimum effort.

PROGRAMMING LANGUAGES AND COMPILERS

Learn more!

TECH GROUP

SCALABLE BUILDS

Correct, efficient, and reliable builds are critical for developers to work and collaborate effectively.

Learn more!

If you enjoyed this article, you might be interested in joining the Tweag team.

← How to Prevent GHC from Inferring Types with Undesirable Constraints

Supercharging your Rust static executables with mimalloc →

COMPANY

About
Open Source
Careers
Contact Us

WHAT WE DO

Strategy
Product Development
Platform Modernization
Digital Operations
Work

INSIGHTS

Modus Blog
Ospo Blog

Research
Innovation podcast

CONNECT WITH US