Using Crates.io with Buck

Apr 27 2023

In a previous post (using-buck-to-build-rust-projects), I laid out the basics of building a Rust project with buck2 (https://buck2.build/). We compared and contrasted it with Cargo. But what about one of the biggest and best features that Cargo has to offer, the ability to use other Rust packages from crates.io? They don't use buck, so how can we integrate them into our build?

A series

This post is part of a series:

- Using buck to build Rust projects (using-buck-to-build-rust-projects)
- Using Crates.io with Buck (you are here)
- Updating Buck (updating-buck)

This post represents how to do this at the time that this was posted; future posts may update or change something that happens here. Here's a hopefully complete but possibly incomplete list of updates and the posts that talk about it:

- `build-script-build` target is no longer generated, see "Updating Buck"

Depending on semver

Let's use the `semver` package example as our program. I am choosing this for a few reasons:

- I used to be involved in maintaining it, and I'm sentimental
- It's authored by dtolnay, who authors many great Rust crates.
- More specifically than that, his excellent cxx library (https://github.com/dtolnay/cxx/) maintains build rules for Cargo, Buck, and Bazel, and examining how he uses Buck in cxx helped me write both the last post and this one. I wanted to make sure to shout that out.
- It's pure Rust (this is easier than something that depends on C, since as I mentioned I'm still having an issue or two with getting my own C toolchain working so far).
- It has one optional dependecy on serde, but no others. Again, this is just easier, not a fundamental limitation.

Here's the example, from the page on crates.io (https://crates.io/crates/semver):

```
use semver::{BuildMetadata, Prerelease, Version, VersionReq};

fn main() {
    let req = VersionReq::parse(">=1.2.3, <1.8.0").unwrap();

    // Check whether this requirement matches version 1.2.3-a
    let version = Version {
        major: 1,
        minor: 2,
        patch: 3,
        pre: Prerelease::new("alpha.1").unwrap(),
        build: BuildMetadata::EMPTY,
    };
    assert!(!req.matches(&version));

    // Check whether it matches 1.3.0 (yes it does)
    let version = Version::parse("1.3.0").unwrap();
    assert!(req.matches(&version));
}
```

Let’s change `src/bin/main.rs` to contain this code. What do we need to do to add this with Cargo?

```
buck-rust-hello〉 cargo add semver
      Adding semver v1.0.17 to dependencies.
             Features:
             + std
             - serde
buck-rust-hello〉 cargo run
   Compiling semver v1.0.17
   Compiling hello_world v0.1.0 (C:\Users\steve\Documents\Git
    Finished dev [unoptimized + debuginfo] target(s) in 0.71s
     Running `target\debug\main.exe`
buck-rust-hello〉
```

Easy enough! We expect no output, becuase the asserts should pass. (Note that we didn’t ask for the `serde` feature, so we aren’t using it, and therefore don’t depend on anything other than `semver`.)

How Cargo handles this

Before we move on, let’s talk for a moment about what Cargo actually does here.

First, `cargo add` has added this to our `Cargo.toml`:

```
[dependencies]
semver = "1.0.17"
```

This is the latest release of semver at the time of this writing.

When we `cargo build`, Cargo will figure out all of the crates we need. It will then check the cache for the source code for that crate, and if it’s not there, download it. On my system, this lives here:

```
~\.cargo\registry\src\github.com-1ecc6299db9ec823\semver-1.0
```

This means we have a *global* cache of source code.

Cargo will then compile `semver`, as you can see in the output above. It places that output in the `target` directory, more specifically

```
.\target\debug\deps
```

This directory will contain `libsemver-ded1559592aad8f7.rlib`, `libsemver-ded1559592aad8f7.rmeta`, and `semver-ded1559592aad8f7.d`. These have hashes embedded in them so that if we had multiple versions of `semver` in our project, they can be disambiguated. If you’re not familiar with `rustc` output:

- rlib files are simlar to archive files, like Unix `ar`. The details are not standard, and may change at any time. Object code lives in here, as well as other things.
- rmeta files contain crate metadata. This can be used for `cargo check`, for example.
- `.d` files are a dependency file. This format comes from gcc/make, and is sorta standardized. This is in theory useful for use with other build systems, but we won’t be using this today.

Cargo will then build our project, passing in `libsemver-*.rlib` as a dependency.

If you're curious about the exact commands and flags, `cargo build -v` will show that. Make sure to `cargo clean` first, or else you'll get no output, given that we've already built this project.

For example, here's the `rustc` invocation that Cargo makes for building this step:

```
rustc --crate-name main --edition=2021 src\bin\main.rs --erro
```

Since we are not using Cargo, we need to replace all of that stuff, and get Buck to generate that `rustc` line, or at least, some equivalent of it.

Let's talk about the various things we need to do:

The mismatch

Let's start with one thing that is basically the same: `buck-out` and `target` are both directories in our project that cache the output of our build. Yeah the name and details are different, but we're not going to try and somehow unify these, as they're both considered implementaiton details of the respective systems, and trying to get them to share is a lot of work for not much gain.

Buck does not have a central registry of packages that we can download code from.

Buck is interested in reproducable builds, and therefore, a global cache of source code doesn't make as much sense. You want the code stored locally, with your project. The dreaded (or beloved) vendoring.

Buck does not understand the crate index, Cargo configuration for a given package, and other implementation details. As a more general build system, those are pretty much out of scope.

Lucikly, other people have done a lot of work here.

Reindeer

Enter Reindeer (https://github.com/facebookincubator/reindeer/). Reindeer is a project that will help us bridge this gap. Here's how this will work: Reindeer will create and (mostly) manage a `third-party` directory for us. It will generate `BUCK` files that we can then use to depend on these external crates. We can even choose to vendor our sources or not.

You can install reindeer through Cargo:

```
> cargo install --git https://github.com/facebookincubator/re
```

Let's set this up. First off, we need to create some files:

```
> mkdir third-party
> code third-party\Cargo.toml # or vim, or whatever
```

In that `Cargo.toml`, we'll need to put this:

```
[workspace]

[package]
name = "rust-third-party"
version = "0.0.0"
publish = false
edition = "2021"

# Dummy target to keep Cargo happy
[[bin]]
name = "fake"
```

```
path = "/dev/null"

[dependencies]
semver = "1.0.17"
```

We're creating a fake package here, so nothing truly matters except the `[dependencies]` section. Here we depend on `semver` as usual.

We're also going to configure Reindeer to not vendor our source code, because that's how I prefer to do things. Vendoring was the default behavior, with non-vendoring being added very recently, so if you prefer to vendor, that workflow works very well

Anyway put this in `third-party/reindeer.toml`:

```
vendor = false
```

So run this inside of your `third-party` directory:

```
〉 cd third-party
〉 reindeer buckify
[WARN  reindeer::fixups] semver-1.0.17 has a build script, bu
```

Oh no, build scripts! Yeah I said that I picked `semver` because it should be easy, but it does have a build script, which is another Cargo-specific feature. Now, the `semver` crate's build script is used as a feature to support older versions of the compiler; all it does is detect old versions and then spit out some configuration to make sure to not use the newer features of the language. This is why this is a warning, not an error; in this case, we don't actually need the build script since we are using a new compiler. So, we are going to skip this *for now*, but we'll come back and fix it.

At this point, `reindeer` has generated a `BUCK` file for `semver`. Let's see what targets we have now:

```
〉 buck2 targets //...
Jobs completed: 4. Time elapsed: 0.0s.
root//:build
root//src/bin:hello_world
root//src/lib:hello_world
root//third-party:semver
root//third-party:semver-1.0.17
root//third-party:semver-1.0.17-build-script-build
```

We have a few new ones! One for `semver`, one for the specific version of `semver`, and one for the build script of `semver`. The general `semver` is an alias for `semver-1.0.17`.

Do you know how to modify our build so that buck builds successfully?

Here's the answer: change `src\bin\BUCK`:

```
rust_binary(
    name = "hello_world",
    srcs = ["main.rs"],
    crate_root = "main.rs",
    deps = [
        "//third-party:semver",
    ],
)
```

And now we can build and run:

```
〉 buck2 run //src/bin:hello_world
File changed: root//src/bin/BUCK
Build ID: b18ba58d-8a77-439a-9d95-6051f3cf21d4
```

```
Jobs completed: 26. Time elapsed: 1.9s. Cache hits: 0%. Comma
```

Success! Our program has no output, if the assertions failed we would have gotten something, but this is expected given the example code.

Now, whenever we need to add or remove a dependency, we can modify `third-party\Cargo.toml`, re-run the `buckify` command, and we're good.

We *do* have two different `Cargo.toml` s now. That is a bit of a bummer. But at least it is easy to determine if there's a problem: dependency failures are loud, and if you're building with both in CI, you'll notice if stuff goes wrong. There also may be a solution to this I'm just not aware of.

If you'd like to see the whole thing at this point, this commit (https://github.com/steveklabnik/buck-rust-hello/commit/2abd1ada7dbbc7f89cd8678eace1e07b3df2ae2f) should have you covered.

This should get you going with building more advanced projects in Rust using buck2. In the next post, we'll talk about fixups, which you'll need for dependencies that are more complex than `semver`.

By the way, if you're interested in this stuff, I've made a Discord for buck fans (https://discord.gg/ZTEmwypZ6K). Come hang out, chat about stuff, ask questions, it's gonna be a good time.