

Compile-Time Config Baking in Rust

2026-07-28

#Programming #Rust

Motivation

It's often the case that we load a configuration file at program entry and use it across the codebase. The most direct approach is to put your config loader and parser inside a recently stabilized `std::sync::LazyLock` (R.I.P. `lazy_static`) initializer, which makes your config struct a global static to be imported by other crates in the workspace:

```
rust
1 // crates/some-crate/src/config.rs
2
3 use std::sync::LazyLock;
4
5 #[derive(serde::Deserialize)]
6 pub struct Config { /* ... */ }
7
8 pub static CONFIG: LazyLock<Config> = LazyLock::new(|| {
9     let config_string = std::fs::read_to_string("path/to/
10     config.json").expect("failed to read config");
11     serde_json::from_str(&config_string).expect("failed to parse config")
12 });
```

Several potential problems are hidden in this strategy.

1. Downstream consumers become dependent on parsers like `serde` or `serde_json`, resulting in larger binary sizes. That goes against Rust's philosophy: *you only pay for what you use*.
2. Dereferencing a `LazyLock` is not a *constant operation*: you cannot use the fields of `CONFIG` as array sizes, in `const fn` or other `const` contexts. It also prevents compiler optimizations.

We observe that config reloading is usually handled by external tools like `nodemon` or `bacon` in simple configuration systems. Those tools watch the filesystem and restart the program when config changes. That means the config file could be considered static throughout the program lifetime, which provides an opportunity for compile-time baking that removes all the runtime overheads.

Solution

The idea is, we load config file and parse it as the `Config` value in `build.rs`. After that, we transform the value into token streams, which get dumped into an output file. In `lib.rs`, we statically `include!()` the generated tokens to retrieve the value.

This enables us to make it a compile-time constant, not just a static variable wrapped in `LazyLock`, because the tokens produce a *literal*. Now the compiler can inline this value everywhere and do a lot of optimizations. When config changes, external tools still restart the program, re-running the build script to partially re-compile the code.

In order to minimize re-compilation costs, we split the build script with config loading/parsing into a separate crate (`config`). Since build scripts cannot directly import items from the main source, we extract the type definition of the `Config` struct into another crate (`config-internal`), from which we re-export the type, so it stays internal.

Solution

Wait, if my Config struct contains String or something living on the heap, how could it be a constant? Try `&'a T`, `Cow<'a, [T]>` or `&'a [T]` instead, where `'a` can be bound in `Config<'a>` or just `'static`, according to the compatibility of your parser. Be sure to make the compiler happy!

rust

```
1 /// Bad config - rejected by the compiler 🚫
2 #[derive(serde::Deserialize)]
3 struct BadConfig {
4     pub host: String
5 }
6 const BAD_CONFIG: BadConfig = BadConfig { host:
    String::from("161.161.161.161") };
7
8 /// Good config - compilation passed ✓
9 #[derive(serde::Deserialize)]
10 struct GoodConfig<'a> {
11     #[serde(borrow)]
12     pub host: &'a str
13 }
14 const GOOD_CONFIG: GoodConfig<'static> = GoodConfig { host: "161.161.161.161" };
```

One problem remains: how to bake a Rust value into tokens? Fortunately, the [databake](#) crate works out-of-the-box.

In the config-internal crate, we mark `serde` and `databake` as optional dependencies gated by feature flags.

toml

```
1 # crates/config-internal/Cargo.toml
2
3 [features]
4 serde = ["dep:serde"]
5 databake = ["dep:databake"]
6
7 [dependencies]
8 serde = { workspace = true, optional = true }
9 databake = { workspace = true, optional = true }
```

After making Config const-compatible, conditionally derive `serde::Deserialize` and `databake::Bake` on the existing definition based on feature flags.

rust

```
1 // crates/config-internal/src/lib.rs
2
3 #[cfg_attr(feature = "serde", derive(serde::Deserialize))]
4 #[cfg_attr(feature = "databake", derive(databake::Bake))]
5 #[cfg_attr(feature = "databake", databake(path = config_internal))]
6 pub struct Config { /* ... */ }
```

For the config crate, note that we specify the config-internal dependency twice: one for runtime and one for `build.rs`. The features are only enabled for the build script, so that consumers of this crate get a clean dependency graph with no runtime `serde`, `serde_json` or `databake`. Don't forget to include the dependencies of your normal config parsing pipeline.

toml

```
1 # crates/config/Cargo.toml
2
3 [dependencies]
```

Solution

```
4 config-internal = { path = "../config-internal" }
5
6 [build-dependencies]
7 build-rs.workspace = true
8 config-internal = { path = "../config-internal", features = ["serde",
  "databake"] }
9 serde_json.workspace = true
10 databake.workspace = true
```

The build script loads, parses and transforms the config value into tokens, which get dumped to an output file.

rust

```
1 // crates/config/build.rs
2
3 use std::path::Path;
4
5 use build_rs::output::rerun_if_changed;
6 use databake::Bake;
7 use config_internal::Config;
8
9 fn main() {
10     let config_string = std::fs::read_to_string("path/to/
  config.json").expect("failed to read config");
11     let config: Config = serde_json::from_str(&config_string).expect("failed to
  parse config");
12     let tokens = config.bake(&Default::default()).to_string();
13
14     let output_path = Path::new(std::env::var("OUT_DIR").expect("missing env
  variable OUT_DIR"));
15     let output_file = output_path.join("generated.rs");
16     std::fs::write(&output_file, tokens).expect("failed to write output file");
17
18     rerun_if_changed("path/to/config.json");
19 }
```

The entrypoint lib.rs re-exports the Config type, and includes the generated tokens statically to produce a constant.

rust

```
1 // crates/config/src/lib.rs
2
3 pub use config_internal::Config;
4
5 pub const CONFIG: Config = include!(concat!(env!("OUT_DIR"), "/generated.rs"));
```

Summary

To recap, we moved config loading and parsing from runtime to compile time:

1. In config-internal, the Config type is defined with const-compatible fields, deriving `serde::Deserialize` and `databake::Bake` behind feature flags.
2. In config/build.rs, the config file is read, parsed and baked into tokens via databake, then written to `$OUT_DIR/generated.rs`.
3. In config/src/lib.rs, the generated tokens are statically `include!()`-ed into a `pub const CONFIG: Config`.

Summary

Compared to the LazyLock approach, this removes all runtime parser dependencies (`serde`, `serde_json`) from the downstream binary, turns the config into a true compile-time constant usable in `const` contexts, and unlocks further compiler optimizations such as inlining and `const` propagation. Combined with `rerun-if-changed` and external file watchers, it also keeps a smooth “reload on change” workflow, since editing the config file makes Cargo re-bake the value and re-compile only the affected crates.

The trade-off is that the config is now frozen into the binary: changing it requires a (partial) re-compilation, and the struct may only contain `const`-constructible types such as `&'static str` or `Cow<'static, _>`. If your config genuinely needs hot-reloading without a restart, stick with LazyLock; otherwise, bake it.