blog | sven kanoldt

sharing thoughts on learning rust, open source projects and other tech stuff

Designing Error Types in Rust Libraries

May 31, 2025 #rust #error #lib #practical rust bites

When designing error types in Rust, especially for libraries with a public API, there are several important considerations to keep in mind. This post will explore some of the important implications and common pitfalls when designing error types in Rust libraries. Further, we will look at `thiserror` and how the Rust standard library's `std::io::Error` can serve as a reference for designing your own error types.

UPDATE 2025-06-02: Added a new section about using own wrapper types to avoid inner error type leakage, thanks to a hint from vlovich123 on HN.

Library vs. Binary crate

When designing error types in Rust, it's important to consider whether your code is part of a library or a binary. The design principles can differ significantly between the two.

Usually for a binary crate, you simply use crates like `anyhow` or `eyre` to handle errors, because your main goal is often to provide a good user error message and you don't need to expose your error types to other crates. So there is no need to design a custom error type unless you want to `match` on it in your code or log it differently than you would present it to the user.

For a library crate, however, you want to design your error types carefully, as they will be part of the public API and will be used by other crates. This means you should consider how your error types will be used, what information they will provide, and how they will interact with other error types in the ecosystem.

Usually there are two main approaches to designing error types in Rust libraries:

1. **Using `thiserror` crate**: This is a popular choice for library crates, as it provides a convenient way to define error types with minimal boilerplate. It allows you to derive the

`std::error::Error` trait and provides a way to convert other error types into your custom error type using the `#[from]` attribute.

2. **Using a self-defined error type** similar to `std::io::Error` and `ErrorKind` : This approach involves defining a custom error type with an enum that represents different error variants.

Each variant would be very plain and only used on one very specific occasion and not carry any additional information. The error struct would then implement the `std::error::Error` trait and associate the `ErrorKind` variant with the source error (see the std lib).

Using `thiserror` with `#[from]`

The `thiserror` crate is a popular choice for defining error types in Rust libraries. It allows you to derive the `std::error::Error` trait and provides a convenient way to convert other error types into your custom error type using the `#[from]` attribute. This can significantly reduce boilerplate code and make error handling more ergonomic and implementation time efficient.

Common mistake: inner error type leakage

Imagine you write a library that uses a crate like `sqlx` or `reqwest` and you have an error variant that looks like this:

```
#[derive(Debug, thiserror::Error)]
pub enum MyError {
    #[error("Database error: {0}")]
    DatabaseError(#[from] sqlx::Error),
    #[error("Network error: {0}")]
    NetworkError(#[from] reqwest::Error),
}
```

This might seem like a reasonable approach at first glance, as it allows you to convert the inner error types into your custom error type using the `#[from]` attribute. However, this design has a significant drawback: it leaks the inner error types (`sqlx::Error` and `reqwest::Error`) to your library consumers.

So unless you are re-exporting these error types (and the types they use and expose), a library consumer will have to depend on `sqlx` and `reqwest` crates to use these error variants, even if they never use the database or network functionality directly.

This is bad design because it imposes unnecessary dependencies on your library users and increases the complexity.

Furthermore, this can lead to version mismatch issues. For example, if you depend on `sqlx` version 0.5.0, but your library user depends on `sqlx` version 0.6.0, then they will run into compilation errors and they're forced to downgrade their `sqlx` version to match your used version, which is really problematic.

Solution 1: Boxing the inner error type as a trait object

A better approach is to avoid exposing the inner error types directly in your library's error type. Instead, you can use a boxed trait object to encapsulate the inner error types. This way, your library users won't have to depend on `sqlx` or `reqwest`, and you can still provide meaningful error messages.

```
#[derive(Debug, thiserror::Error)]
pub enum MyError {
    #[error("Database error: {0}")]
    DatabaseError(#[from] Box<dyn std::error::Error + Send + Sync>),
    #[error("Network error: {0}")]
    NetworkError(#[from] Box<dyn std::error::Error + Send + Sync>),
}
```

This creates an opaque error type that can hold any error that implements the `std::error::Error` trait, while still allowing you to provide a meaningful error message and providing access to the inner error message via the `Display` implementation and even the underlying error type through dynamic downcasting if needed.

As a side bonus, this approach gives you the flexibility to exchange the inner error type later without breaking the public API of your library. You can change the inner error type to any other type that implements `std::error::Error`, as long as it is boxed.

Solution 2: Using own wrapper types

UPDATE 2025-06-02: On HN vlovich123 pointed out there is another solution to the problem, that I'm going to describe here. Thanks for the hint!

So instead of Boxing the concrete underlying error type, you would define an own wrapper type that holds the inner error as a private field. This way you can still provide a meaningful error

message and access the inner error type, but you don't expose the concrete error type to your library users.

You would name the wrapper type after the error you want to wrap, e.g. `sqlx::Error` would be wrapped by `SqlError` and `reqwest::Error` would be wrapped by `ReqwestError`. Like this:

```
/// Note: `sqlx::Error` is not re-exported, so the library user does not have to depend on `
sqlx`.
pub struct SqlError(sqlx::Error);

/// Note: `reqwest::Error` is also not re-exported and remains private to the library.
pub struct ReqwestError(reqwest::Error);

#[derive(Debug, thiserror::Error)]
pub enum MyError {
    #[error("Database error: {0}")]
    DatabaseError(SqlError),
    #[error("Network error: {0}")]
    NetworkError(ReqwestError),
}
```

Now you can implement `Display` and `Debug` for the wrapper types to show the specifics you want to show.

Furhter you can implement `From<sqlx::Error>` and `From<reqwest::Error>` for the wrapper types to convert them into your custom error type when using the `?` operator at the call site.

This approach has the advantage of keeping the inner error types private to your library, while still allowing you to provide meaningful error messages and access to the inner error type if needed. It also avoids the need for boxing, which can be more efficient in some cases.

But it does require more manual work to implement the wrapper types and the `From` trait for each inner error type.

Using `std::io::Error` as a reference

The `std::io::Error` type is a good reference for designing your own error types in Rust libraries. It uses an enum called `ErrorKind` to represent different kinds of errors, without carrying any additional data, and it provides a way to associate a source error with each variant.

```
pub struct Error {
```

```
  repr: Repr,
}

enum Repr {
  Os(i32),
  Simple(ErrorKind),
  Custom(Box<Custom>),
}

struct Custom {
  kind: ErrorKind,
  error: Box<dyn error::Error + Send + Sync>,
}

#[derive(Clone, Copy)]
#[non_exhaustive]
pub enum ErrorKind {
  NotFound,
  PermissionDenied,
  Interrupted,
  ...
  Other,
}

impl Error {
  pub fn kind(&self) -> ErrorKind {
    match &self.repr {
      Repr::Os(code) => sys::decode_error_kind(*code),
      Repr::Custom(c) => c.kind,
      Repr::Simple(kind) => *kind,
    }
  }
}
```

Things to note about this design:

- **ErrorKind enum**: The `ErrorKind` enum is used to represent different kinds of errors without carrying any additional data. This is what consumers will use and see.
- **Custom error type**: The `Custom` struct is used to encapsulate a custom error type that can carry additional information. This allows you to provide more context about the error while still keeping the public API clean.
- **Boxed trait object**: The `error` field in the `Custom` struct is a boxed trait object that can hold any error type that implements the `std::error::Error` trait. This allows you to provide a meaningful error message while keeping the inner error type opaque to library consumers.
- **Non-exhaustive enum**: The `ErrorKind` enum is marked as `#[non_exhaustive]`, which allows you to add new error kinds in the future without breaking existing code. This is a good

practice for library design, as it allows for future extensibility.

- **Repr enum**: The `Repr` enum is used to represent the different representations of the error, such as an OS error code, a simple error kind, or a custom error type. This allows you to handle different error representations in a clean and organized way. It's not exposed to the library consumers, but used internally to handle the error representation.
- The whole approach is open for future extensibility, as you can add new error kinds or custom error types without breaking existing code.

For further reading on the design of `std::io::Error`, you can check out this blog post by matklad, which provides a detailed analysis of the design choices made in the standard library.

Conclusion: When to use which approach?

The choice between using `thiserror` with `#[from]` or defining a custom error type depends on your specific use case:

- If you are building a library that will be used by other crates and you want to provide a clear and consistent error handling API, consider using `thiserror` with `Box<dyn Error + Send + Sync>`. This allows you to encapsulate the inner error types without exposing them directly.

- If you don't mind a bit of manual work and efficiency thoughts would suggest you staying away from boxing trait objects, you can define your own wrapping types for the inner error types that holds the inner error as a private field. This way you can still provide a meaningful error message and access the inner error type, but you don't expose the concrete error type to your library users.

- If you are building a library that will be used **"internally"** (e.g., inside of a workspace project or a single application), and you don't need to worry about exposing inner error types, you can use `thiserror` with `#[from]` to simplify error handling and reduce boilerplate code.

- You can also mix both approaches, if you are careful about which error types are part of the public API and which are not. For example, you can use `thiserror` with `#[from]` for internal error types, while using `Box<dyn Error + Send + Sync>` or wrapper Types for public error types that should not expose inner error types.

- If it's important to you to not pull in dependencies and keep the public API clean and free of breaking changes, you can use a custom error type with an enum like `std::io::Error` and `ErrorKind`, as described earlier. But be warned that it might require more manual implementation work.