Rust Error Handling: thiserror, anyhow, and When to Use Each

February 6, 2024 · 6 min

▸ Table of Contents

In this blog post, we'll explore strategies for streamlining error handling in Rust using two popular libraries: thiserror and anyhow. We'll discuss their features, use cases, and provide insights on when to choose each library.

TL;DR

- `thiserror` simplifies the implementation of custom error type, removing boilerplates
- `anyhow` consolidates errors that implement `std::error::Error`
- While `thiserror` provides detailed error information for specific reactions, `anyhow` hides internal details

Return Different Error Types from Function

Let's start by creating a function `decode()` for illustration. The function has 3 steps:

1. Read contents from a file named input
2. Decode each line as a base64 string
3. Print each decoded string

The challenge is determining the return type for decode since `std::fs::read_to_string()`, base64 `decode()`, and `String::from_utf8()`

each return different error types.

```
use base64::{self, engine, Engine};

fn decode() -> /* ? */ {
    let input = std::fs::read_to_string("input")?;
    for line in input.lines() {
        let bytes = engine::general_purpose::STANDARD.decode(line)?;
        println!("{}", String::from_utf8(bytes)?);
    }
    Ok(())
}
```

One approach is to use trait object: `Box<dyn std::error::Error>` . This works because all those types implement `std::error::Error` .

```
fn decode() -> Result<(), Box<dyn std::error::Error>> {
  // ...
}
```

While this is suitable in some cases, it limits the caller's ability to discern the actual error that occurred in `decode()` . Then, using `enum` is a good approach if it is desired to handle each error in different ways.

```
enum AppError {
    ReadError(std::io::Error),
    DecodeError(base64::DecodeError),
    StringError(std::string::FromUtf8Error),
}
```

By implementing `std::error::Error` trait, we can semantically mark `AppError` as an error type.

```
impl std::error::Error for AppError {}
```

However, this code doesn't compile because `AppError` doesn't satisfy the constraints required by `std::error::Error` , implementation of `Display` and `Debug` :

```
error[E0277]: `AppError` doesn't implement `std::fmt::Display`
error[E0277]: `AppError` doesn't implement `Debug`
```

The definition of `std::error::Error` represents the consensus of minimum requirements of an error type in Rust. An error should have two forms of description for users (`Display`) and programmers (`Debug`), and should provide its root cause.

```
pub trait Error: Debug + Display {
    fn source(&self) -> Option<&(dyn Error + 'static)> { ... }
    // ...
}
```

The code will be like this after implementing the required traits:

```
impl std::error::Error for AppError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        use AppError::*;
        match self {
            ReadError(e) => Some(e),
            DecodeError(e) => Some(e),
            StringError(e) => Some(e),
        }
    }
}

impl std::fmt::Display for AppError { // Error message for users.
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result
        use AppError::*;
        let message = match self {
            ReadError(_) => "Failed to read the file.",
            DecodeError(_) => "Failed to decode the input.",
            StringError(_) => "Failed to parse the decoded bytes.",
        };
        write!(f, "{message}")
    }
}

impl std::fmt::Debug for AppError { // Error message for programmers.
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result
        writeln!(f, "{self}")?;
        if let Some(e) = self.source() { // <-- Use source() to retriv
            writeln!(f, "\tCaused by: {e:?}")?;
        }
        Ok(())
    }
}
```

Finally, we can use `AppError` in `decode()`:

```
fn decode() -> Result<(), AppError> {
    let input = std::fs::read_to_string("input").map_err(AppError::Read
    // ...
```

`map_err()` is used to convert `std::io::Error` to `AppError::ReadError`. To use `?` operator for better flow, we can implement `From` trait for `AppError`:

```
impl From<std::io::Error> for AppError {
    fn from(value: std::io::Error) -> Self {
        AppError::ReadError(value)
    }
}

impl From<base64::DecodeError> for AppError {
    fn from(value: base64::DecodeError) -> Self {
        AppError::DecodeError(value)
    }
}

impl From<std::string::FromUtf8Error> for AppError {
    fn from(value: std::string::FromUtf8Error) -> Self {
        AppError::StringError(value)
    }
}

fn decode() -> Result<(), AppError> {
    let input = std::fs::read_to_string("input")?;
    for line in input.lines() {
        let bytes = engine::general_purpose::STANDARD.decode(line)?;
        println!("{}", String::from_utf8(bytes)?);
    }
    Ok(())
}

fn main() {
    if let Err(error) = decode() {
        println!("{error:?}");
    }
}
```

We did several things to use our custom error type fluently:

- implement `std::error::Error`

- implement `Debug` and `Display`
- implement `From`

These can be verbose and tedious, but fortunately, `thiserror` automatically generates most of them.

Remove Boilerplates with `thiserror`

The code above is simplified using `thiserror`：

```
#[derive(thiserror::Error)]
enum AppError {
    #[error("Failed to read the file.")]
    ReadError(#[from] std::io::Error),
    #[error("Failed to decode the input.")]
    DecodeError(#[from] base64::DecodeError),
    #[error("Failed to parse the decoded bytes.")]
    StringError(#[from] std::string::FromUtf8Error),
}

impl std::fmt::Debug for AppError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result
        writeln!(f, "{self}")?;
        if let Some(e) = self.source() {
            writeln!(f, "\tCaused by: {e:?}")?;
        }
        Ok(())
    }
}
```

`#[error]` macro generates `Display`, `#[from]` macro handles `From` implementations and `source()` for `std::error::Error`. The implementation of `Debug` remains to provide detailed error messages, but `#derive[Debug]` can also be used if it's enough:

```

// The manual implementation of Debug
Failed to decode the input.
        Caused by: InvalidPadding

// #[derive(Debug)]
DecodeError(InvalidPadding)
```

Deal with Any Error with `anyhow`

`anyhow` offers an alternative method for simplifying error handling, which is similar to `Box<dyn std::error::Error>>` approach:

```
fn decode() -> Result<(), anyhow::Error> {
    let input = std::fs::read_to_string("input")?;
    for line in input.lines() {
        let bytes = engine::general_purpose::STANDARD.decode(line)?;
        println!("{}", String::from_utf8(bytes)?);
    }
    Ok(())
}
```

It compiles since types implementing `std::error::Error` can be converted to `anyhow::Error` . The error message will be like:

```
Invalid padding
```

For enhanced error messages, `context()` can be used:

```
let bytes = engine::general_purpose::STANDARD
    .decode(line)
    .context("Failed to decode the input")?;
```

Then, the error message will be:

```
Failed to decode the input

Caused by:
    Invalid padding
```

Now our error handling is streamlined thanks to the `anyhow` 's type conversion and `context()` .

Comparison between `thiserror` and `anyhow`

While `thiserror` and `anyhow` might seem similar, they serve different purposes. `thiserror` is suitable when users need to react differently based the actual error type. On the other hand, `anyhow` is effective when internal details can be hidden from the user.

In this sense, it's often said that `thiserror` is for a library, and `anyhow` is for an application. This saying is true to some extent, considering that library developers tend to want to give precise information to users (programmers), and applications don't have to show detailed error information to their users.

Conclusion

In conclusion, we've explored the distinctive features of thiserror and anyhow and discussed scenarios where each library shines. By choosing the right tool for the job, Rust developers can significantly simplify error handling and enhance code maintainability.

- `thiserror` simplifies the implementation of custom error types
- `anyhow` integrates any `std::error::Error`
- `thiserror` is ideal for library development where detailed information is beneficial for users (programmers).
- `anyhow` is Suited for applications where internal details are not crucial, providing simplified information to users.

Rust | Error-Handling

« PREV
PDF Summarizer with Ollama in 20 Lines of Rust

NEXT »
Organize Rust Integration Tests Without Dead Code Warning